2019147531 컴퓨터과학과 박윤정

Task 1

• Explain your implementation with screenshots of the code

• For task 1-2, state the optimal solution for each image and the analysis of such result.

Task 1-1 Noise Addition

<각noise를 더해준 다음 rms 결과값>

```
dbw2140@ubuntu:~/Desktop/ComputerVision/CV-Proj
RMS for Gaussian noise: 29.856391799645028
RMS for Uniform noise: 16.84343304753214
RMS for Impulse noise: 51.23514685947474
dbw2140@ubuntu:~/Desktop/ComputerVision/CV-Proj
```

Add_gaussian_noise

```
def add_gaussian_noise(image):
    # Use mean of 0, and standard deviation of image itself to generate gaussian noise
    mean = 0
    std = np.std(image) #standard deviation of image
    #print("image is {}".format(image))
    gaussian_noise = np.random.normal(mean, std, image.shape)
    image = image + gaussian_noise #add gaussian_noise to original image
    return image
```

Np.std를 통해 np.ndarray 형태인 image의 standard deviation을 구한다. 그리고 np.random.normal을 통해 image.shape만큼의 (mean, std)range의 gaussian_noise를 구하고 원래 이미지에 더한 다음 return한다.

Bird.jpg에 gaussian_noise를 더해준 후의 이미지는 다음과 같다.

Add_uniform_noise

```
def add_uniform_noise(image):
    # Generate noise of uniform distribution in range [0, standard deviation of image)
    #print("image is {}".format(type(image)))
    std = np.std(image)
    uniform_noise = np.random.uniform(0, std, image.shape)
    image = image + uniform_noise
    return image
```

마찬가지로 np.std를 통해 image의 standard deviation을 고르고 np.random.uniform을 통해 image.shape만큼 (0,std)range의 uniform distribution에서 uniform noise를 구하고 원래 이미지에 더한 다음 return한다.

Bird.jpg에 uniform_noise를 더해준 후의 이미지는 다음과 같다.

Add_impulse_noise

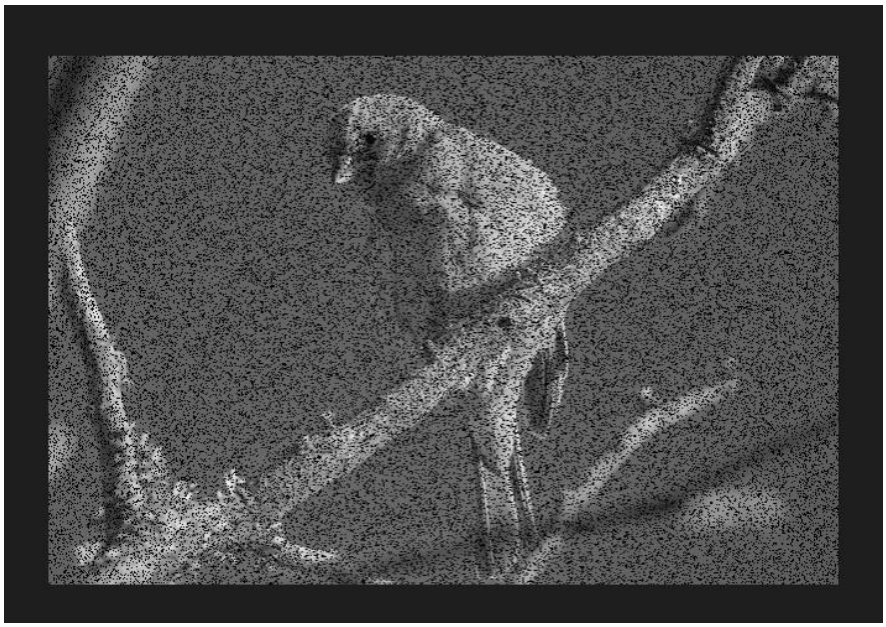
```python
def apply_impulse_noise(image): #color image
    # # Implement pepper noise so that 20% of the image is noisy

    row = image.shape[0]
    col = image.shape[1]
    #print("row is {} col is {}".format(row, col))
    num_pixels = row * col
    #print("image shape is {}".format(image.shape)) #427*640
    #print("num_pixels is {}".format(num_pixels))
    num_pepper = int(0.2*num_pixels)


    pixel = np.random.choice(num_pixels, num_pepper, replace =False)
    #print(type(pixel))
    #print(pixel)
    #print("pixel is {}".foramt(pixel))
    for i in range(num_pepper):
        x = pixel[i]//col
        y = int(pixel[i]%col)
        image[x][y] = 0 #black
    #print(len(tlist)) #54656
```

'implement pepper noise so that 20% of the image is noisy'이기 때문에 row와 column의 개수를 곱해 전체 pixels의 개수를 구하고 그중 20%인 pepper noise를 추가할 픽셀의 개수를 구하였다.

그런 다음, pepper noise를 추가할 pixel의 index(=pixel)를 random.choice로 num_pixels(전체 픽셀 갯수)에서 num_pepper(pepper noise를 추가할 픽셀의 개수)만큼 중복없이(replace=False)구하였다.

이는 행렬을 array로 나타냈을 때 pixel의 index(=pixel)를 구한 것이므로 행렬에서 x, y값의 index를 각각 구하려면 x는 pixel을 column으로 나눈 몫, y는 pixel을 column으로 나눈 나머지가 된다. 따라서, num_pepper만큼, image의 pixel을 검정색으로 만들어준다.

Bird.jpg에 impulse_noise를 더해준 후의 이미지는 다음과 같다.



Task1-2 Noise Estimation & Removal

-각 필터 코드 설명

-이미지 denoise할 때 optimal한 거 설명 및 결과 분석 + rms 결과값 + 저장된 결과이미지

<3가지 필터 구현 코드 캡쳐 및 설명>

Apply_median_filter

```
    You should return result image.
    """
    #print(img.shape)
    p = int((kernel_size-1)/2) #number of padding
    w = int((kernel_size + 1) / 2 )#use to move index in output image
    k = int(kernel_size*kernel_size / 2) #index for median

    #get median
    new_shape0 = img.shape[0]+2*p
    new_shape1 = img.shape[1]+2*p
    #image_output = np.zeros((new_shape0,new_shape1,img.shape[2])) #padding #img.copy() #np.zeros(img.shape)
    #image_output = np.full((new_shape0,new_shape1,img.shape[2]), 0.5)
    image_output = np.ones((new_shape0,new_shape1,img.shape[2]))
    image_output[p:new_shape0-p, p:new_shape1-p] = img.copy()
    img = image_output.copy()
    for x in range(0, new_shape0-kernel_size+1):
        for y in range(0, new_shape1-kernel_size+1):
            for c in range(0, img.shape[2]):
                #print("x, y is {} {}".format(x, y))
                tmp = (img[x:x+kernel_size, y:y+kernel_size,c]).copy()
                #print(tmp)
                t = tmp.flatten() #flatten -> array
                # print(t)
                t.sort()
                median = t[k]
                #print("median is {}".format(median))
                image_output[int(x+w-1),int(y+w-1),c] = median
                #print(image_output)
    #print("happy")
    img = image_output[p:new_shape0-p, p:new_shape1-p]
    #print(img.shape)
    return img
```

Median filter를 구현하였다. 마찬가지로, Input image와 output image의 크기가 같아야하므로, zero padding을 먼저 해준다. Padding의 크기(=p)는 kernel_size에 의해 결정이 된다. W는 output image에서 filter와의 convolution후 저장되는 픽셀의 index를 바꾸는데 사용되는 값이다. K는 median index이다. Padding을 하게 되면 새로운 이미지의 row와 column의 개수가 각각 new_shape0, new_shape1로 변한다. For loop 3개를 통해, filter와 convolution할 이미지의 영역을 정한다. Tmp는 img중 필터와 convolution을 하는 영역이다. 행렬 Tmp를 flatten()을 통해 array로 만들고 sort로 정렬해준다. median값은 k(=kernel_size*kernel_size)번째 값이다. median 값을 image_output의 해당하는 center pixel의 값으로 정한다. 또한, 최종 return값인 img에서 padding 부분을 제외한 나머지 부분([p:new_shape0-p, p:new_shape1-p])은 image_output과 같다.

Apply_bilateral_filter

```
    'sigma_s' is a int value, which is a sigma value for G_s(gaussian function for space)
    'sigma_r' is a int value, which is a sigma value for G_r(gaussian function for range)
    You should return result image.
    """
    #print("bilateral filter")
    p = int((kernel_size-1)/2) #number of padding
    w = int((kernel_size + 1) / 2 )#use to move index in output image
    center = w - 1 #center

    #get
    new_shape0 = img.shape[0]+2*p
    new_shape1 = img.shape[1]+2*p
    image_output = np.zeros((new_shape0,new_shape1,img.shape[2])) #padding #img.copy() #np.zeros(img.shape)
    image_output[p:new_shape0-p, p:new_shape1-p] = img.copy()
    img = image_output.copy()
    #image_output = img.copy() #np.zeros(image.shape)
    #print("image shape is {} {}".format(img.shape[0], img.shape[1]))
    for x in range(0, new_shape0-kernel_size+1):
        for y in range(0, new_shape1-kernel_size+1):
            for c in range(0, img.shape[2]):
                #print("x, y is {} {}".format(x, y))
                sum = 0
                weights = 0
                tmp = img[x:x+kernel_size, y:y+kernel_size,c]
                #print(tmp)
                for m in range(0, kernel_size):
                    for n in range(0, kernel_size):
                        distance2 = pow(m-center,2) + pow(n-center,2)
                        diff = int(tmp[m,n])-int(tmp[center, center])   ###
                        #diff = int(tmp[m,n])/255-int(tmp[center, center])/255 ###
                        diff_range = pow(diff,2)
                        gaussian_s = np.exp((-0.5)*distance2/pow(sigma_s,2)) #(1/(2*np.pi*pow(sigma_s,2))) multiply do not need.
                        gaussian_r = np.exp((-0.5)*diff_range/pow(sigma_r,2)) #
                        weights += gaussian_s*gaussian_r
                        sum += gaussian_s*gaussian_r*tmp[m,n] #g_s*g_r*intensity of pixel
                image_output[int(x+w-1),int(y+w-1),c] = sum/weights
                #print(image_output)
        #print("x is {}".format(x))
    #img = image_output
    img = image_output[p:new_shape0-p, p:new_shape1-p]
    return img
```

Input image와 output image의 크기가 같아야하므로, zero padding을 먼저 해준다. Padding의 크기(=p)는 kernel_size에 의해 결정이 된다. W는 output image에서 filter와의 convolution후 저장되는 픽셀의 index를 바꾸는데 사용되는 값이고, center는 filter의 center pixel index를 의미한다. Padding을 하게 되면 새로운 이미지의 row와 column의 개수가 각각 new_shape0, new_shape1로 변한다. For loop 3개를 통해, filter와 convolution할 이미지의 영역을 정한다. Tmp는 img중 필터와 convolution을 하는 영역이다. Bilateral은 space와 range를 모두 고려하므로, distance2는 center pixel과 neighborhood pixel의 거리제곱값으로 guassian_s에 사용되고 diff_range는 center pixel값과 neighborhood pixel값의 차의 제곱값으로 guassian_r에 사용된다. 이때, 원래 공식은 (1/2*np.pi*pow(sigma_s,2))가 곱해지지만, normalize과정(sum/weights)에서 나누기되어 사라지기에 연산 속도를 생각하여 곱하지 않았다. Bilateral filter와의 convolution으로 구한값(sum/weights)를 image_output의 해당하는 center pixel의 값으로 정한다. 또한, 최종 return값인 img에서 padding 부분을 제외한 나머지 부분([p:new_shape0-p, p:new_shape1-p])은 image_output과 같다.

Apply_my_filter (average filter 구현했음)

```python
def apply_my_filter(img, kernel_size): #average filter
    """
    You should implement additional filter using convolution.
    You can use any filters for this function, except median, bilateral filter.
    You can add more arguments for this function if you need.

    You should return result image.
    """
    #print("my filter")

    ## AVerage
    p = int((kernel_size-1)/2) #number of padding
    w = int((kernel_size + 1) / 2 ) #use to move index in output image

    #get mean(average)
    new_shape0 = img.shape[0]+2*p
    new_shape1 = img.shape[1]+2*p
    image_output = np.zeros((new_shape0,new_shape1,img.shape[2])) #padding #img.copy() #np.zeros(img.shape)
    image_output[p:new_shape0-p, p:new_shape1-p] = img.copy()
    img = image_output.copy()
    for x in range(0, new_shape0-kernel_size+1):
        for y in range(0, new_shape1-kernel_size+1):
            for c in range(0, img.shape[2]):
                #print("x, y is {} {}".format(x, y))
                tmp = img[x:x+kernel_size, y:y+kernel_size,c]
                #print(tmp)
                # t = tmp.flatten() #flatten -> array
                # average = np.mean(t)
                sum = 0
                for m in range(0, kernel_size):
                    for n in range(0, kernel_size):
                        sum += tmp[m,n]
                image_output[int(x+w-1),int(y+w-1),c] = sum / (kernel_size*kernel_size)
                #print(image_output)
    img = image_output[p:new_shape0-p, p:new_shape1-p]
    #print("happy")
    return img
```

Average filter를 구현하였다. 마찬가지로, Input image와 output image의 크기가 같아야하므로, zero padding을 먼저 해준다. Padding의 크기(=p)는 kernel_size에 의해 결정이 된다. W는 output image에서 filter와의 convolution후 저장되는 픽셀의 index를 바꾸는데 사용되는 값이다. Padding을 하게 되면 새로운 이미지의 row와 column의 개수가 각각 new_shape0, new_shape1로 변한다. For loop 3개를 통해, filter와 convolution할 이미지의 영역을 정한다. Tmp는 img중 필터와 convolution을 하는 영역이다. Tmp에 있는 pixel 값들을 모두 더한다음(sum), 개수만큼 나눈 값(=평균값. average)을 image_output의 해당하는 center pixel의 값으로 정한다. 또한, 최종 return값인 img에서 padding부분을 제외한 나머지 부분([p:new_shape0-p, p:new_shape1-p])은 image_output과 같다.

Task1_2 function

```python
def task1_2(src_path, clean_path, dst_path):
    """
    This is main function for task 1.
    It takes 3 arguments,
    'src_path' is path for source image.
    'clean_path' is path for clean image.
    'dst_path' is path for output image, where your result image should be saved.

    You should load image in 'src_path', and then perform task 1-2,
    and then save your result image to 'dst_path'.
    """
    #print("task1_2")
    noisy_img = cv2.imread(src_path)
    clean_img = cv2.imread(clean_path)
    result_img = None
    # # # do noise removal
    sigma_s = 20 #need to change
    sigma_r = 40 #need to change (65 for snow) (40 for fox)

    window_size = 100
    b_rms = 2000 #best rms
    tmp_filter = 0
    filter = {0:'median', 1:'bilateral', 2: 'my(average)'}

    for i in range(0, 9):
        j = i%3
        kernel_size = 2*j+3 #3,5,7
        if(0<=i<=2): #MEDIAN FILTER
            tmp = apply_median_filter(noisy_img, kernel_size)
            rms = calculate_rms(clean_img, tmp)
            k = 0
            #print("rms now is {}".format(rms))
        elif(3<=i<=5): #BILATERAL FILTER
            tmp = apply_bilateral_filter(noisy_img, kernel_size, sigma_s, sigma_r)
            rms = calculate_rms(clean_img, tmp)
            k = 1
            #print("rms now is {}".format(rms))
```

```python
        else: #MY(average) FILTER
            tmp = apply_my_filter(noisy_img, kernel_size)
            rms = calculate_rms(clean_img, tmp)
            k = 2
            #print("rms now is {}".format(rms))
        if (rms<b_rms):
            result_img = tmp.copy()
            b_rms = rms
            window_size = kernel_size
            tmp_filter = k
    print("optimal filter is {}".format(filter[tmp_filter]))
    print("window_size is {}".format(window_size))
    print("best rms is {}".format(b_rms))
    #print("b_rms is {}".format(b_rms))
    #print("Best RMS:", calculate_rms(clean_img, result_img))
    cv2.imwrite(dst_path, result_img)
    pass
```

주어진 source image에 대해 3가지 filter로 denoise한 후 rms를 최소화하는 filter와 window_size를 구하고 dst_path에 denoised image filtered with optimal solution을 저장하는 task1_2 function의 코드이다. Src_path, clean_path에서 noisy_image와 clean_image를 읽어온다. Window_size와 b_rms, tmp_filter는 최소의 rms를 주는 window_size와 filter의 종류를 구하기 위한 변수이다. 3개의 필터, window_size(=kernel_size) 3,5,7에 대해 denoise 할 것이므로 9번 for loop가 실행되고 i는 각 필터 3번(window size 3,5,7로 총 3번)을 돌리기 위해 필요한 변수이다. 한번 filter할 때 마다 rms를 구해 b_rms와 크기 비교를 통해 더 작은 값이 b_rms에 저장되고 result_img에 해당하는 enoised_image가 저장되고 window_size에 해당하는 window_size가 저장되고 tmp_filter에 해당하는 filter종류가 저장된다. 따라서, for loop로 각 필터 3번씩 행한 후, filter dictionary를 통해 필터의 종류를 string값으로 받고 print, 최적의 window_size를 print, minimum rms가 print되고 result_img가 dst_path에 저장된다.

<Denoise test images>

1. **Cat**

```
  diff = np.abs(img1.astype(dtype=np.int) - img2.a
optimal filter is median
window size is 3
best rms is 8.323375808277767
```

Optimal solution for cat image window_size= 3인 median filter이다. Minimum Rms = 8.32337580827767

Window_size가 3, 5, 7로 각각 median, bilateral, my(average) filter에 대해 실행했었고, median filter가 가장 적은 rms를 기록했다. Salt&pepper noise에는 중앙값으로 바꾸는 median filter가 좋기에 이런 결과가 나온 것이라고 생각한다.

저장된 결과 이미지는 다음과 같다.

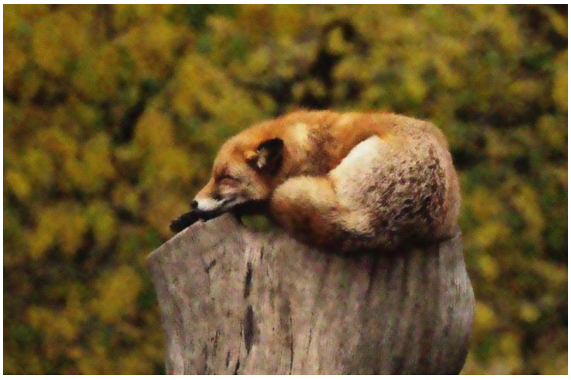
2. **Fox**

```
    diff = np.abs(img1.astype(dtype=np.int) -
optimal filter is bilateral
window_size is 7
best rms is 10.361508924259054
dbw2140@ubuntu:~/Desktop/ComputerVision/CV-
```

Optimal solution for fox image는 sigma_s = 20, sigma_r = 40, window_size= 7인 bilateral filter이다. Minimum Rms = 10.361508924259054

Window_size가 3, 5, 7로 각각 median, bilateral, my(average) filter에 대해 실행했었고, bilateral filter가 가장 적은 rms를 기록했다. Bilateral filter가 space뿐만 아니라 range까지도 고려하기에 edge를 유지하면서 blur를 해서 가장 좋은 결과가 나온 것이라고 생각한다. sigma값은 10단위로 실행했었다.

저장된 결과 이미지는 다음과 같다.



3. **Snowman**

```
    diff = np.abs(img1.astype(dtype=np.int) - img2.astype(dtype=np
optimal filter is bilateral
window_size is 7
best rms is 9.611358718541007
dbw2140@ubuntu:~/Desktop/ComputerVision/CV-Project1_v2/task1$
```

Optimal solution for snowman image는 sigma_s = 20, sigma_r = 65, window_size= 7인 bilateral filter이다. Minimum Rms = 9.611358718541007

Window_size가 3, 5, 7로 각각 median, bilateral, my(average) filter에 대해 실행했었고, bilateral filter가 가장 적은 rms를 기록했다. Bilateral filter가 space뿐만 아니라 range까지도 고려하기에 edge를 유지하면서 blur를 해서 가장 좋은 결과가 나온 것이라고 생각한다. sigma값은 sigma_r을 fox와 같이 40으로 하니 rms가 잘 안나와서 r을 바꾸다 65에서 baseline보다 작아 65로 하게 되었다.

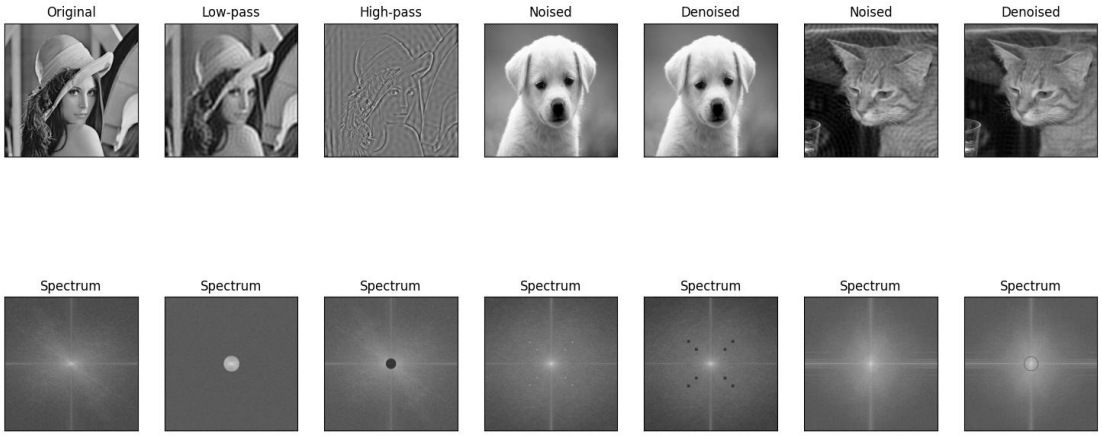
저장된 결과 이미지는 다음과 같다.

Task 2

• Explain your implementation with screenshots of the code

• Make sure to include the results after going through high-pass filter, low-pass filter, denoise1, and denoise2 as well as fm spectrum of those images

<전체 결과>



Fftshift(img)

```python
def fftshift(img):
    '''
    This function should shift the spectrum image to the center.
    You should not use any kind of built in shift function. Please implement your own.
    '''
    # print('img is {}'.format(img))
    #t = img #
    row, col = img.shape
    r = round(row / 2)
    c = round(col / 2)

    # print("row is {}".format(row))
    # print("col is {}".format(col))
    # print("r is {}".format(r))
    # print("c is {}".format(c))

    m1 = img[0:r, 0:c]
    m2 = img[0:r, c:]
    m3 = img[r:, 0:c]
    m4 = img[r: , c:]

    m2_m1 = np.concatenate((m2,m1), axis = 1)
    m4_m3 = np.concatenate((m4,m3), axis = 1)
    img = np.concatenate((m4_m3,m2_m1), axis = 0)
    # print("m1 is {}".format(m1))
    # print("m2 is {}".format(m2))
```

Input이 행렬인 경우, fftshift는 행렬의 첫번째 사분면을 세번째 사분면으로 바꾸고 두번째 사분면을 네번째 사분면으로 바꾼다. 따라서, row, column의 개수를 구하고 사분면으로 나눈다. M1,m2,m3,m4는 각각 사분면이다. 시계방향으로 m3,m4,m1,m2가 되도록 이렇게 분할된 네 개의 행렬을 다시 np.concatenate을 이용해 합친다.

Ifftshift(img)

```
def ifftshift(img):
    '''
    This function should do the reverse of what fftshift function does.
    You should not use any kind of built in shift function. Please implement your own.
    '''
    #t = img #
    row, col = img.shape
    r = round(row / 2)
    c = round(col / 2)

    # print("row is {}".format(row))
    # print("col is {}".format(col))
    # print("r is {}".format(r))
    # print("c is {}".format(c))

    m1 = img[0:r, 0:c]
    m2 = img[0:r, c:]
    m3 = img[r:, 0:c]
    m4 = img[r: , c:]

    # print("m1 is {}".format(m1))
    # print("m2 is {}".format(m2))
    # print("m3 is {}".format(m3))
    # print("m4 is {}".format(m4))

    m2_m1 = np.concatenate((m2,m1), axis = 1)
    m4_m3 = np.concatenate((m4,m3), axis = 1)
    img = np.concatenate((m4_m3,m2_m1), axis = 0)

    #check is it correct
    #print((img == np.fft.ifftshift(t))) #

    return img
```

fftshift함수의 반대 역할이다. 따라서 input 행렬을 fftshift와 마찬가지로, 1,4분면을 서로 바꾸고, 3,4분면을 서로 바꾼다.

Fm_spectrum(img)

```
def fm_spectrum(img):
    '''
    This function should get the frequency magnitude spectrum of the input image.
    Make sure that the spectrum image is shifted to the center using the implemented fftshift function.
    You may have to multiply the resultant spectrum by a certain magnitude in order to display it correctly.
    '''
    #print("fm_spectrum") #
    f_image = np.fft.fft2(img)
    #k = f_image #
    f_shift = fftshift(f_image)
    #f_shift = f_image
    m_spectrum = 20*np.log(np.abs(f_shift))
    img = m_spectrum
    #print("m_spectrum is {}".format(m_spectrum)) #
    #check fftshift, ifftshift implementation right
    #f_image = ifftshift(img) #
    #print(f_image == k) #
    #print("fm_spectrum") #

    return img
```

frequency magnitude spectrum of the input image을 구하는 함수이다. Np.fft.fft2를 이용하여 frequency domain으로 바꾼다. 이때, center로 shift해주기 위해 위에서 구현한 fftshift함수를 사용한다. magnitude이므로 np.abs를 사용해 절대값을 구하고 잘 보이기 위해 숫자를 곱해준다. 곱해준 숫자는 인터넷에서 참고하였다.

Low_pass_filter(img, r=30)

```python
def low_pass_filter(img, r=30):
    '''
    This function should return an image that goes through low-pass filter.
    '''
    #create low_pass_filter(ndarray) in frequency domain
    row, col = img.shape
    low_filter_fs = np.zeros((row, col),dtype = 'complex') #dtype=complex <- multiply with image in frequency domain
    #print(low_filter_fs)
    #print(low_filter_fs.size)
    id_center_row = row // 2 #index of center row
    id_center_col = col // 2 #index of center col
    #print(id_center_row, id_center_col)

    for i in range(row):
        for j in range(col): #(i,j) - (id_center_row, id_center_col)
            distance =(i - id_center_row) ** 2 + (j-id_center_col)**2
            d = np.sqrt(distance)
            if (d<=r):
                low_filter_fs[i, j] = 255 #white

    f_image = np.fft.fft2(img)
    f_shift = fftshift(f_image)
    f_low = low_filter_fs * f_shift #element-wise multiplication
    f_low_ishift = ifftshift(f_low)
    img = np.fft.ifft2(f_low_ishift) #inverse Fourier Transform
    img = np.abs(img)
    #0-255 values
    img -= img.min()
    img = img*255 / img.max()
    img = img.astype(np.uint8)

    return img
```

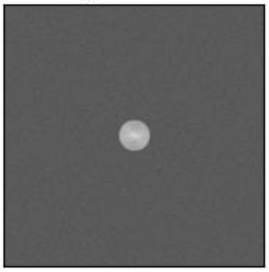
Low-frequency를 통과시키는 함수이다. 먼저 image크기에 맞게 0으로 된 ndarray를 만든다. 나중에 input image의 frequency domain값과 곱해줄 것이므로 complex가 data type이다.

image에서 center값을 구한다. 여기서 threshold는 center에서의 거리와 관계가 있다. 인자로 주어진 r보다 작은 거리에 있는(low-frequency)만 통과한다. 따라서, center와의 거리가 r보다 작은 pixel만 255로 정한다. Spatial domain에서의 filter와 convolution은 frequency domain에서 filter와 element-wise multiplication이므로 input image를 fourier transform한 다음 low_filter_fs와 element-wise multiplication하고 inverse fourier transform을 한다. 이때, F/T다음에 fftshift, I/F/T하기 전에는 ifftshift를 해야 한다. 그리고 이미지의 실수부분만 return한다.

Low-pass filter후 이미지가 blurr처리되었음을 알 수 있다. 또, spectrum을 보았을 때, center에서 가까운 원(r=30)부분만 밝은 것을 알 수 있다.

Low-pass

Spectrum



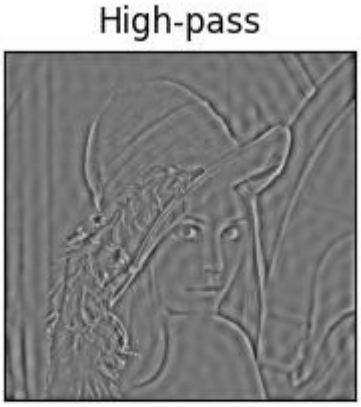
High_pass_filter(img, r=20)

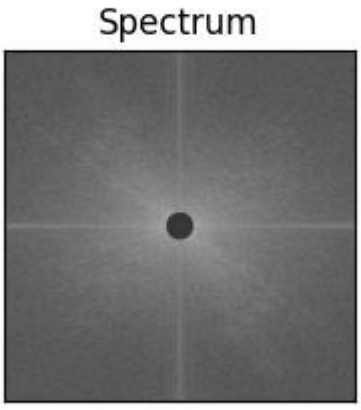
```
def high_pass_filter(img, r=20):
    '''
    This function should return an image that goes through high-pass filter.
    '''
    #create low_pass_filter(ndarray) in frequency domain
    row, col = img.shape
    high_filter_fs = np.ones((row, col),dtype = 'complex') #dtype=complex <- mult
    id_center_row = row // 2 #index of center row
    id_center_col = col // 2 #index of center col
    #print(id_center_row, id_center_col)

    for i in range(row):
        for j in range(col): #(i,j) - (id_center_row, id_center_col)
            distance =(i - id_center_row) ** 2 + (j-id_center_col)**2
            d = np.sqrt(distance)
            if (d<=r):
                high_filter_fs[i, j] = 0 #black

    f_image = np.fft.fft2(img)
    f_shift = fftshift(f_image)
    f_low = high_filter_fs * f_shift #element-wise multiplication
    f_low_ishift = ifftshift(f_low)
    img = np.fft.ifft2(f_low_ishift) #inverse Fourier Transform
    img = np.abs(img)
    #0-255 values
    img -= img.min()
    img = img*255 / img.max()
    img = img.astype(np.uint8)

    return img
```

High-frequency를 통과시키는 함수이다. 먼저 image크기에 맞게 1로 된 ndarray를 만든다. 나중에 input image의 frequency domain값과 곱해줄 것이므로 complex가 data type이다.

image에서 center값을 구한다. 여기서 threshold는 center에서의 거리와 관계가 있다. 인자로 주어진 r보다 먼 거리에 있는(high-frequency)만 통과한다. 따라서, center와의 거리가 r보다 작은 pixel을 0으로(black) 정한다. Spatial domain에서의 filter와 convolution은 frequency domain에서 filter와 element-wise multiplication이므로 input image를 fourier transform한 다음 low_filter_fs와 element-wise multiplication하고 inverse fourier transform을 한다. 이때, F/T다음에 fftshift, I/F/T하기 전에는 ifftshift를 해야 한다. 그리고 이미지의 실수부분만 return한다.

High-pass filter 후, high-frequency만 남아, edge부분, 즉 변화가 큰 부분만 남은 것을 알 수 있다. 또, spectrum에서 중심과 가까운 원 부분(r=20)이 어두운 것이 보인다.



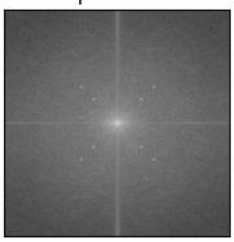


Denoise1(img)

```python
def denoise1(img):
    '''
    Use adequate technique(s) to denoise the image.
    Hint: Use fourier transform
    '''
    # #create denoise1 filter
    row, col = img.shape
    denoise1_filter = np.ones((row, col),dtype = 'complex') #dtype=complex <- mul
    #print(denoise1_filter.shape)
    id_center_row = row // 2 #index of center row
    id_center_col = col // 2 #index of center col

    for i in range(row):
        for j in range(col): #tried a lot...haha
            if(50<=np.abs(i-id_center_row)<=60):
                if(50<=np.abs(j-id_center_col)<=60):
                    denoise1_filter[i][j] = 0 #black
            elif(80<=np.abs(i-id_center_row)<=90):
                if(80<=np.abs(j-id_center_col)<=90):
                    denoise1_filter[i][j] = 0 #black


    f_image = np.fft.fft2(img)
    f_shift = fftshift(f_image)
    f_low = denoise1_filter * f_shift #element-wise multiplication
    f_low_ishift = ifftshift(f_low)
    img = np.fft.ifft2(f_low_ishift) #inverse Fourier Transform
    img = np.abs(img)
    #0-255 values
    img -= img.min()
    img = img*255 / img.max()
    img = img.astype(np.uint8)
```

Noised

Spectrum

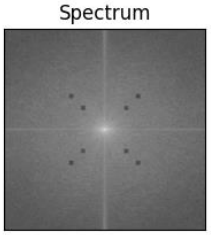


Noised를 보면 가장자리에 체크 모양의 noise가 있고 spectrum을 확인해보면 8개의 사각점 같은 밝은 부분이 있다. 이것이 noise를 일으킴을 알 수 있어 저 부분을 어둡게 하는 denoise1_filter를 코드로 작성하였다. 처음에 중심에서 거리가 얼마인지 감이 안 와 여러10단위의 숫자를 반복했다.

그리고 대각선이기에, row, column 범위를 같게 하였다. 대칭이어서 크게 2가지 경우를 생각했다. 중심과 비교적 가까운 점의 경우, row, column값이 center의 row, column값과 차이가 50, 60사이인 경우로 그 경우에 해당하는 pixel의 denoise1_filter을 0(black)으로, 비교적 먼 점의 경우, 80,90사이로 구현했다. 그런 다음 spatial domain의 이미지를 F/T를 통해 frequency domain으로 바꾸고 filter와 element-wise multiplication한 후, 다시 inverse fourier transform하였다.

Denoised

Spectrum



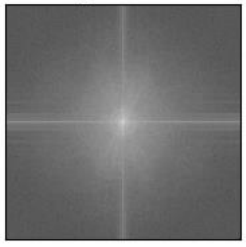
Denoise1후, spectrum에서 밝은 8개의 부분이 어두워졌고 spatial domain image의 가장자리에서 check noise가 사라졌음을 확인하였다.

Denoise2(img)

```
    return img

def denoise2(img):
    '''
    Use adequate technique(s) to denoise the image.
    Hint: Use fourier transform
    '''
    #create denoise2 filter
    row, col = img.shape
    denoise2_filter = np.ones((row, col),dtype = 'complex') #dtype=complex <- multiply with image in frequency domain
    #print(denoise1_filter.shape)
    id_center_row = row // 2 #index of center row
    id_center_col = col // 2 #index of center col

    for i in range(row):
        for j in range(col): #(i,j): diagonal
            distance =(i - id_center_row) ** 2 + (j-id_center_col)**2
            d = np.sqrt(distance)
            if (26<=d<=28): #26, 28.   27:x
                denoise2_filter[i, j] = 0 #black

    f_image = np.fft.fft2(img)
    f_shift = fftshift(f_image)
    f_low = denoise2_filter * f_shift #element-wise multiplication
    f_low_ishift = ifftshift(f_low)
    img = np.fft.ifft2(f_low_ishift) #inverse Fourier Transform
    img = np.abs(img)
    #0-255 values
    img -= img.min()
    img = img*255 / img.max()
    img = img.astype(np.uint8)

    return img
```

Noised

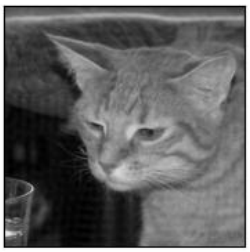
Spectrum



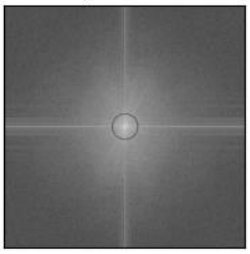
Noise 이미지를 보면 wave noise(periodic noise)가 있다. Spectrum을 보면 밝은 원이 있음을 알

수 있고 아까 low-pass에서 r=30, high-pass에서 r=20이었던 것을 생각하여 spectrum에서 원의 위치를 비교해보면 20-30사이에 존재함을 알 수 있었다. 이것저것 숫자를 넣어보다가 r=27에서 문제가 생김을 알 수 있어서 center에서 거리가 26-28일 경우, denoise2_filter의 값을 0(black)으로 정하였다. (band-reject filter) Denoise1과 마찬가지로 noise image를 F/T하고 denoise2_filter와 element-wise multiplication후 다시 I/F/T하였다.

Denoised



Spectrum



Denoise2후 이미지에서 wave noise가 사라진 것을 확인하였고 spectrum으로 보면 코드로 구현한 center에서 거리가 26-28인 곳에 어두운 원이 있음을 알 수 있다.