Ryan Sunwoo Lee /2022847310

CV-Project1 Report

Task 1.1

· *add_gaussian_noise(image)*

```
7  def add_gaussian_noise(image):
8      # Use mean of 0, and standard deviation of image itself to generate gaussian noise
9
10     gauss_noise = np.random.normal(0,image.std(),image.shape)
11
12     return image + gauss_noise
```

지시대로 gauss_noise를 생성하기 위해, 'numpy.random.normal'을 사용했다. 첫번째 매개변수는 평균값인 0을 넣고, 표준 편차와 사이즈는 이미지의 표준 편차와 사이즈를 그대로 사용했다. 노이즈 값을 gauss_noise에 저장한 뒤, 이미지에 더해 return했다.

가우시안 노이즈를 적용한 이미지

· *add_uniform_noise(image)*

```
14 def add_uniform_noise(image):
15     # Generate noise of uniform distribution in range [0, standard deviation of image)
16
17     uniform_noise = np.random.uniform(0,image.std(),image.shape)
18
19     return image + uniform_noise
```

Uniform noise를 생성하기 위해 'numpy.random.uniform' 함수를 사용하였다. 노이즈의 범위는 0부터 이미지의 표준 편차 값을 사용했고, 사이즈 또한 이미지와 동일한 사이즈로 적용하였다. 노이즈를 이미즈에 더한 후 그 값을 return했다.

균일 노이즈를 적용한 이미지

· *add_impulse_noise(image)*

```
21 def apply_impulse_noise(image):
22     # Implement pepper noise so that 20% of the image is noisy
23 
24     pepper_image = image
25     row = image.shape[0]
26     col = image.shape[1]
27 
28     pepper_number = int(row*col*0.2)
29 
30     while pepper_number >=0:
31         pepper_row = np.random.randint(0,row-1)
32         pepper_col = np.random.randint(0,col-1)
33 
34         if pepper_image[pepper_row][pepper_col] != 0 :
35             pepper_image[pepper_row][pepper_col] = 0
36             pepper_number = pepper_number -1
37 
38     return pepper_image
```

먼저 행과 열 값을 row, col 변수에 저장한 후 후추 노이즈의 양을 이미지의 20%가 되도록 계산했다. 그 후 while 루프를 사용해 후추 노이즈의 수만큼 이미지의 픽셀을 랜덤하게 0으로 바꿔주었다. 만약 픽셀값이 이미 0이라면, 픽셀이 이미 바뀐 값일 수도 있고, 0으로 바꿔주더라도 노이즈의 의미가 없기 때문에 다시 다른 픽셀을 찾도록 구현했다.

임펄스 노이즈를 적용한 이미지

노이즈가 성공적으로 추가된 것을 확인할 수 있다. 하지만 이 방법으로는 만약 완전히 검은 색 이미지를 입력하면 코드가 끝나지 않는다는 단점을 가지고 있다. 그래서 다른 방법을 사용해 코드를 하나 더 구현하였다.

```
21 def apply_impulse_noise(image):
22     # Implement pepper noise so that 20% of the image is noisy
23
24     pepper_image = image.copy()
25     row = image.shape[0]
26     col = image.shape[1]
27
28     for i in range(row):
29         for j in range(col):
30             random_number = np.random.rand()
31
32             if(random_number<=0.2):
33                 pepper_image[i,j] = 0
34
35     return pepper_image
```

이 코드는 각각의 픽셀에 20%만큼의 확률로 이미지의 픽셀 값을 0으로 바꾼다. 이 방식을 사용하면 정확히 20% 개수의 후추 픽셀이 존재하지는 않게 되지만, 무한히 루프할 수 있는 가능성을 없앤다.

```
RMS for Gaussian noise: 29.856391799645028
RMS for Uniform noise: 16.84343304753214
RMS for Impulse noise: 50.881642751454585
```

노이즈 추가 함수의 RMS 값

임펄스 노이즈를 추가하는 것에는 두 번째 방법을 사용했다. 후추 노이즈의 RMS가 기준

선보다 낮긴 하지만 이미지의 20%를 후추 노이즈로 제대로 변환했음으로 문제가 없을 것으로 생각된다. 나머지 노이즈는 RMS 수치가 정상 범위에 있는 것을 확인할 수 있다.

Task 1.2

· apply_median_filter(img, kernel_size)

코드의 79번째 줄에서 83번째 줄은 필터링을 하기 위해 필요한 이미지 행렬에 관한 변수들을 저장했다. Half_kernel은 커널이 행렬 인덱스 값 밖을 지정하지 않도록 하기 위해 만든 변수다. Sort 함수는 커널 안에 있는 값들의 중간 값을 찾아주는 함수다. 코드 92줄부터 95줄은 이미지의 가장자리를 제외한 부분의 픽셀 값을 중간 값으로 구현하기 위한 코드이다. 이미지의 가장자리를 따로 처리하기 위해, 97줄부터 113줄까지의 코드를 따로 구현하였다. For문 안은 커널 안의 값을 sort 함수로 보내 중간 값을 찾아서 이미지의 픽셀 값을 고치는 작업을 반복한다.

```
77  def apply_median_filter(img, kernel_size):
78  |
79      row = img.shape[0]
80      col = img.shape[1]
81      channel = img.shape[2]
82
83      half_kernel = int(kernel_size / 2)
84
85      #median_image = np.zeros([row + 2 * half_kernel,col + 2 * half_kernel,channel])
86
87      ##Method 1
88
89      median_image = img.copy()
90
91
92      for i in range(half_kernel,row-half_kernel):
93          for j in range(half_kernel,col-half_kernel):
94              for k in range(channel):
95                      median_image[i][j][k] = sort(img[i-half_kernel:i+half_kernel*2,j-half_kernel:j+half_kernel*2,k],kernel_size)
96
97      for i in range(half_kernel,col - half_kernel):
98          for k in range(channel):
99              x = [img[0][i-1][k],img[0][i][k],img[0][i+1][k],img[1][i-1][k],img[1][i][k]]
00              x.sort()
01              median_image[0][i][k] = x[2]
02              x = [img[row-1][i-1][k],img[row-1][i][k],img[row-1][i+1][k],img[row-2][i-1][k],img[row-2][i][k]]
03              x.sort()
04              median_image[row-1][i][k] = x[2]
05
06      for i in range(half_kernel,row-half_kernel):
07          for k in range(channel):
08              x = [img[i-1][0][k],img[i][0][k],img[i+1][0][k],img[i-1][1][k],img[i][1][k]]
09              x.sort()
10              median_image[i][0][k] = x[2]
11              x = [img[i-1][col-1][k],img[i][col-1][k],img[i+1][col-1][k],img[i-1][col-2][k],img[i][col-2][k]]
12              x.sort()
13              median_image[i][col-1][k] = x[2]
14
15      return median_image
```

중간 값 필터의 코드

```python
def sort(array,kernel_size):

    mid = int((kernel_size**2)/2)

    row = array.shape[0]
    col = array.shape[1]

    temp = []

    for i in range(row):
        for j in range(col):
            temp.append((array[i][j]))

    temp.sort()


    return temp[mid]
```

Sort 함수의 코드

· *apply_bilateral_filter(img, kernel_size, sigma_s, sigma_r)*

```
143 def apply_bilateral_filter(img, kernel_size, sigma_s, sigma_r):
144 
145     row = img.shape[0]
146     col = img.shape[1]
147     channel = img.shape[2]
148 
149     half_kernel = int(kernel_size/2)
150 
151     bi_image = np.zeros([row + 2 * half_kernel,col + 2 * half_kernel,channel])
152     bi_image[half_kernel:half_kernel+row,half_kernel:half_kernel+col] = img.copy()
153     compare_img = bi_image.copy()
154 
155 
156     for i in range(row):
157         for j in range(col):
158             for k in range(channel):
159                 wp = 0.0
160                 kernel_sum = 0.0
161 
162                 kernel = compare_img[i:i+kernel_size,j:j+kernel_size,k]
163                 kernel2 = compare_img[i:i+kernel_size,j:j+kernel_size]
164 
165                 for x in range(kernel_size):
166                     for y in range(kernel_size):
167                         if (i<half_kernel or j<half_kernel) and np.array_equal(kernel2[x,y],[0,0,0]):
168                             continue
169                         if (i>row-half_kernel or j>col-half_kernel) and np.array_equal(kernel2[x,y],[0,0,0]):
170                             continue
171                         space = gaussian_2d(half_kernel-x,half_kernel-y,sigma_s)
172                         color = gaussian(kernel[half_kernel,half_kernel] - kernel[x,y],sigma_r)
173                         wp += space * color
174 
175                         kernel_sum+= space*color*kernel[x,y]
176 
177                 new_color = kernel_sum // wp
178                 bi_image[i+half_kernel,j+half_kernel,k] = new_color
179 
180     bi_image = bi_image[half_kernel:half_kernel+row,half_kernel:half_kernel+col]
181 
182     return bi_image
183 
184 def gaussian(x,sigma):
185     return 1/(np.sqrt(2*np.pi*sigma)) * np.exp(-1 * (x**2) / (2* sigma**2))
186 def gaussian_2d(x,y,sigma):
187     return 1/(2*np.pi*(sigma**2)) * np.exp(-1 * (x**2+y**2) / (2 * sigma**2))
```

양방향 필터의 코드

145줄부터 153줄의 코드는 중간 값 필터에서와 같이 함수에 필요한 값을 미리 저장해두기 위한 부분이다. 중간 값 필터와 다르게 양방향 필터를 좀 더 쉽게 구현하기 위해 이미지의 가장자리 픽셀에서 이미지 바깥으로 커널이 튀어나가는 것을 방지(Index Error)하기 위한 zero-padding을 했다. gaussian과 gaussian_2d는 1차원과 2차원 함수에 대한 가우시안 공식의 값을 return한다.

$$BF[I]_{\mathbf{p}} = \frac{1}{W_{\mathbf{p}}} \sum_{\mathbf{q} \in \mathcal{S}} G_{\sigma_s}(\|\mathbf{p} - \mathbf{q}\|)\, G_{\sigma_r}(|I_{\mathbf{p}} - I_{\mathbf{q}}|)\, I_{\mathbf{q}},$$

rmalization factor $W_{\mathbf{p}}$ ensures pixel weights sum to 1.0:

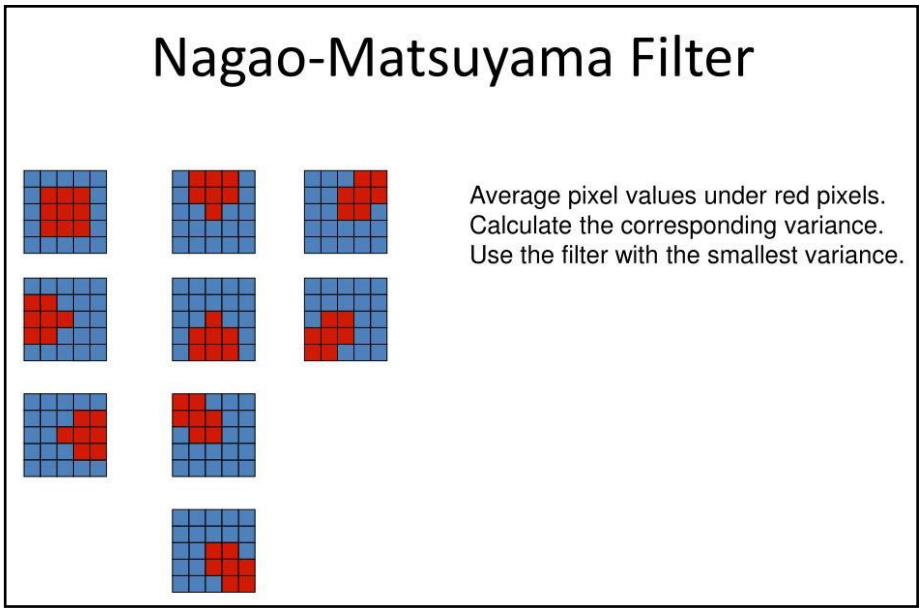
$$W_{\mathbf{p}} = \sum_{\mathbf{q} \in \mathcal{S}} G_{\sigma_s}(\|\mathbf{p} - \mathbf{q}\|)\, G_{\sigma_r}(|I_{\mathbf{p}} - I_{\mathbf{q}}|).$$

양방향 필터 공식

165줄에서 167줄은 위의 양방향 필터 식을 표현하는 부분이다. 커널 안의 Wp값과 커널 값을 모두 찾아 더해 커널 값을 Wp 값으로 나누어 준다. '//' 연산자는 int(kernel_sum/wp)와 같은 역할을 한다. Zero-padding이 필터의 값에 영향을 주는 것을 방지하기 위해 167에서 170줄의 코드를 구현했다. 마지막으로 zero-padding을 제거한 이미지를 return했다. 양방향 필터는 edge를 보존한 채로 가우시안 노이즈를 제거할 수 있는 장점이 있다.

· apply_my_filter(img, kernel_size)

Nagao-Matsuyama 필터를 사용하였다.



위의 이미지처럼, Nagao-Matsuyama 필터는 커널을 9부분으로 나누고 그 중 가장 분산이 적은 곳의 평균값으로 픽셀의 값을 바꾼다. 먼저, 커널의 사이즈를 5로 고정한 뒤, 216줄부터

224줄까지의 코드를 통해 9부분으로 나누고 min_var에 분산이 가장 적은 값을 저장해서 그 부분의 평균값으로 픽셀 값을 바꾸도록 구현했다. 픽셀의 값은 정수여야 하기 때문에 전환하는 값 또한 int로 따로 캐스팅해주었다.

```
def apply_my_filter(img,kernel_size=5):

    row = img.shape[0]
    col = img.shape[1]
    channel = img.shape[2]

    half_kernel = 2

    result = np.zeros([row + 2 * half_kernel,col + 2 * half_kernel,channel])
    result[half_kernel:half_kernel+row,half_kernel:half_kernel+col] = img.copy()
    compare_img = result.copy()

    for i in range(row):
        for j in range(col):
            for k in range(channel):
                kernel = compare_img[i:i+kernel_size,j:j+kernel_size,k]

                temp1 = [kernel[0,1],kernel[0,2],kernel[0,3],kernel[1,1],kernel[1,2],kernel[1,3],kernel[2,2]]
                temp2 = [kernel[1,3],kernel[1,4],kernel[2,2],kernel[2,3],kernel[2,4],kernel[3,3],kernel[3,4]]
                temp3 = [kernel[2,2],kernel[3,1],kernel[3,2],kernel[3,3],kernel[4,1],kernel[4,2],kernel[4,3]]
                temp4 = [kernel[1,0],kernel[1,1],kernel[2,0],kernel[2,1],kernel[2,2],kernel[3,0],kernel[3,1]]
                temp5 = [kernel[0,0],kernel[0,1],kernel[1,0],kernel[1,1],kernel[1,2],kernel[2,1],kernel[2,2]]
                temp6 = [kernel[0,3],kernel[0,4],kernel[1,2],kernel[1,3],kernel[1,4],kernel[2,2],kernel[2,3]]
                temp7 = [kernel[2,2],kernel[2,3],kernel[3,2],kernel[3,3],kernel[3,4],kernel[4,3],kernel[4,4]]
                temp8 = [kernel[2,1],kernel[2,2],kernel[3,0],kernel[3,1],kernel[3,2],kernel[4,0],kernel[4,1]]
                temp9 = [kernel[1,1],kernel[1,2],kernel[1,3],kernel[2,1],kernel[2,2],kernel[2,3],kernel[3,1],kernel[3,2],kernel[3,3]]

                min_var =  min(np.var(temp1), np.var(temp2), np.var(temp3), np.var(temp4), np.var(temp5), np.var(temp6), np.var(temp7),
np.var(temp8),np.var(temp9))
                if min_var == np.var(temp1):
                    result[i+half_kernel,j+half_kernel,k] = int(np.mean(temp1))
                elif min_var == np.var(temp2):
                    result[i+half_kernel,j+half_kernel,k] = int(np.mean(temp2))
                elif min_var == np.var(temp3):
                    result[i+half_kernel,j+half_kernel,k] = int(np.mean(temp3))
                elif min_var == np.var(temp4):
                    result[i+half_kernel,j+half_kernel,k] = int(np.mean(temp4))
                elif min_var == np.var(temp5):
                    result[i+half_kernel,j+half_kernel,k] = int(np.mean(temp5))
                elif min_var == np.var(temp6):
                    result[i+half_kernel,j+half_kernel,k] = int(np.mean(temp6))
                elif min_var == np.var(temp7):
                    result[i+half_kernel,j+half_kernel,k] = int(np.mean(temp7))
                elif min_var == np.var(temp8):
                    result[i+half_kernel,j+half_kernel,k] = int(np.mean(temp8))
                elif min_var == np.var(temp9):
                    result[i+half_kernel,j+half_kernel,k] = int(np.mean(temp9))


    result = result[half_kernel:half_kernel+row,half_kernel:half_kernel+col]

    return result
```

Nagao-Matsuyama 필터의 코드

· task1_2(src_path, clean_path, dst_path)

Task1_2에서, 모든 필터를 테스트해보고 RMS값을 저장해 비교하는 방식을 사용하였다. Result1은 메디안 필터를 사용했는데, 커널의 값이 커질수록 중간 값을 사용하는 방식의 정확도가 떨어지는 것을 확인하였기 때문에 작은 커널 사이즈인 3을 사용하였다. Result2, result3, result4는 모두 양방향 필터를 저장하는데 각각 range sigma 값을 다르게 주었다. 만약 커널의 사이즈가 7보다 커지면 계산 시간이 기하급수적으로 늘고 결과값에 큰 차이가 생기지도 않았으며, space sigma 또한 커널을 7로 고정했을 때 9에서 가장 낮은 RMS를 기록했기 때문에 그 값으로 고정했다. Result5는 직접 구현한 Nagao-Matsuyama 필터를 사용했다. 이 5개의 결과 중 가장 낮은 RMS 값을 가지는 이미지가 최종 result로 저장된다. Task1_2는 안의 필터 함수들이 큰 시간 복잡

도를 가지고 있기 때문에 실행 시간 또한 매우 길어지게 된다.

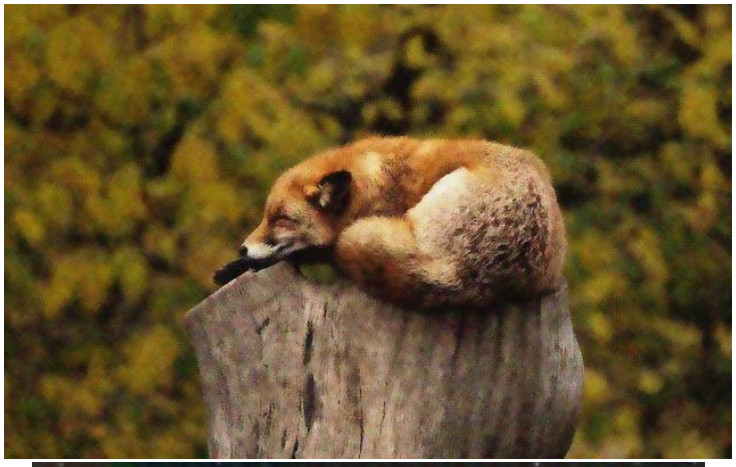
```
kernel size:  7  space_size:  9  color_size:  40  rms:  10.291617111357841  time:  0:04:47.698976
kernel size:  7  space_size:  9  color_size:  50  rms:  10.37508534101447  time:  0:04:40.514416
kernel size:  7  space_size:  9  color_size:  60  rms:  10.694059695009031  time:  0:04:38.288018
This is the image for snowman!!!!!!!!!!!!
kernel size:  7  space_size:  9  color_size:  40  rms:  10.642864346105675  time:  0:05:01.975983
kernel size:  7  space_size:  9  color_size:  50  rms:  9.832541297155947  time:  0:05:05.354882
kernel size:  7  space_size:  9  color_size:  60  rms:  9.549670476844751  time:  0:05:05.106290
```

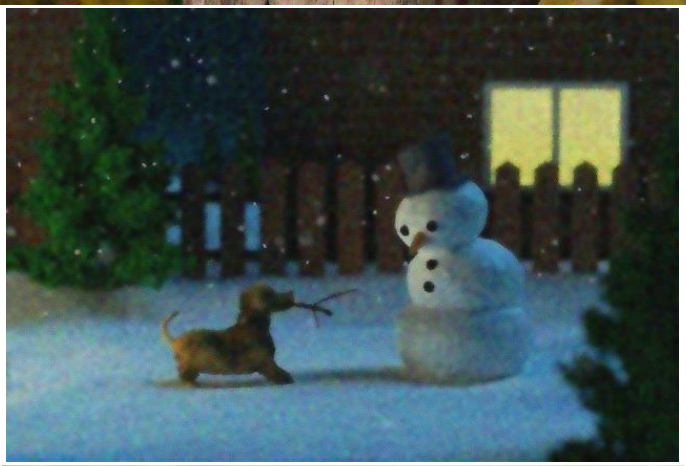
```
kernel size:  7  space_size:  9  color_size:  70  rms:  11.042458245996828  time:  0:03:54.231143
kernel size:  7  space_size:  9  color_size:  80  rms:  11.356903015071934  time:  0:03:53.526292
This is the image for snowman!!!!!!!!!!!!
kernel size:  7  space_size:  9  color_size:  70  rms:  9.505637217195218  time:  0:04:10.277543
kernel size:  7  space_size:  9  color_size:  80  rms:  9.55945516111092  time:  0:04:08.405776
```

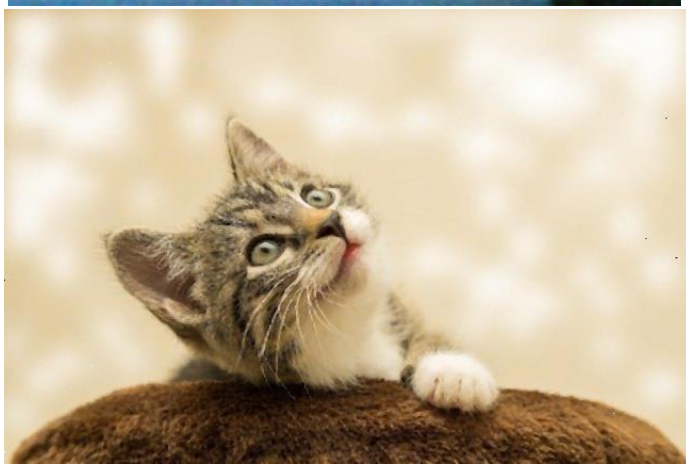
위의 결과값들은 최적의 sigma 값을 찾기 위한 결과값의 일부분이다. Fox_noisy.jpg 이미지의 경우 range sigma가 40일 때 가장 낮은 값을 보여주었고, snowman_noisy.jpg의 경우는 70일 때 가장 낮은 값을 보여주었다. 이 결과들을 바탕으로 좀 더 여러 이미지에 대응하기 위해 result2, result3, result4는 각각 40, 70, 100의 range sigma를 가지게 되었다.

```python
def task1_2(src_path, clean_path, dst_path):


    noisy_img = cv2.imread(src_path)
    clean_img = cv2.imread(clean_path)
    result_img = None

    # do noise removal

    result1 = apply_median_filter(noisy_img,3)
    rms_1 = calculate_rms(result1,clean_img)

    result2 = apply_bilateral_filter(noisy_img,7,9,40)
    rms_2 = calculate_rms(result2,clean_img)
    result3 = apply_bilateral_filter(noisy_img,7,9,70)
    rms_3 = calculate_rms(result3,clean_img)
    result4 = apply_bilateral_filter(noisy_img,7,9,100)
    rms_4 = calculate_rms(result4,clean_img)

    result5 = apply_my_filter(noisy_img)
    rms_5 = calculate_rms(result5,clean_img)


    if min(rms_1,rms_2, rms_3, rms_4, rms_5) == rms_1:
        print(rms_1)
        result_img = result1.copy()
    elif min(rms_1,rms_2, rms_3, rms_4, rms_5) == rms_2:
        print(rms_2)
        result_img = result2.copy()
    elif min(rms_1,rms_2, rms_3, rms_4, rms_5) == rms_3:
        print(rms_3)
        result_img = result3.copy()
    elif min(rms_1,rms_2, rms_3, rms_4, rms_5) == rms_4:
        print(rms_4)
        result_img = result4.copy()
    elif min(rms_1,rms_2, rms_3, rms_4, rms_5) == rms_5:
        print(rms_5)
        result_img = result5.copy()

    cv2.imwrite(dst_path, result_img)
    pass
```

Task1_2의 코드







```
Filtered with: Bilateral filter  RMS:  10.291617113578 41
Filtered with: Bilateral filter  RMS:  9.505637217195218
Filtered with: Median filter  RMS:  7.8275655285551675
```

위의 이미지는 구현한 필터링 함수의 결과 이미지와 RMS 값이다. 모든 이미지가 성공적으로 RMS 기준선을 통과한 것을 확인할 수 있다. 중간 값 필터는 임펄스 노이즈를 제거하는데 강하므로 cat_noisy.jpg와 같은 salt&pepper 노이즈를 가진 항목에 적용되었다. 반면, fox_noisy.jpg나 snowman_noisy.jpg처럼 가우시안 노이즈와 같은 정규 분포를 가지는 백색 노이즈는 양방향 필터나 Nagao-Matsuyama로 줄일 수 있는데, 결과로는 양쪽 다 양방향 필터가 적용된 것을 확인할 수 있다.

Task2

· *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.
    '''

    row = img.shape[0]
    col = img.shape[1]

    half_row = int(np.ceil(row/2))
    half_col = int(np.ceil(col/2))

    shift_img =img.copy()

    if(row%2 == 0):
        shift_img[:half_row] = img[half_row:].copy()
        shift_img[half_row:] = img[:half_row].copy()
    else:
        shift_img[:half_row-1] = img[half_row:].copy()
        shift_img[half_row-1:] = img[:half_row].copy()

    temp = shift_img.copy()

    for i in range(row):
        if(col%2 == 0):
            shift_img[i][:half_col] = temp[i][half_col:].copy()
            shift_img[i][half_col:] = temp[i][:half_col].copy()

        else:
            shift_img[i][:half_col-1] = temp[i][half_col:].copy()
            shift_img[i][half_col-1:] = temp[i][:half_col].copy()

    return shift_img
```

fftshift의 코드

fftshift는 행렬의 1사분면과 3사분면, 그리고 2사분면과 4사분면을 교체하는 함수다. 이 함수는 주파수가 0인 부분을 중앙으로 위치시키기 위해 사용된다. 먼저 21줄부터 26줄까지의 코드를 통해 배열의 행의 중간 값을 기준으로 반으로 나눠 둘의 위치를 뒤바꾸었다. 그 후, 30줄부터 37줄의 코드로 바뀐 행렬을 열의 중간 값을 기준으로 반으로 나눠 둘의 값을 바꾸어 주었다. 행과 열이 홀수일수도, 짝수일수도 있기 때문에 각각의 경우에 대응하는 코드를 나누어서 작성했다.

```
a = np.array([[1,2,3],[4,5,6],[7,8,9]])
b = fftshift(a)
c = np.fft.fftshift(a)
print("ORIGINAL ARRAY")
print(a)
print("My fftshift")
print(b)
print("np.fft.fftshift")
print(c)
exit()
```

```
ORIGINAL ARRAY
[[1 2 3]
 [4 5 6]
 [7 8 9]]
My fftshift
[[9 7 8]
 [3 1 2]
 [6 4 5]]
np.fft.fftshift
[[9 7 8]
 [3 1 2]
 [6 4 5]]
```

구현한 fftshift가 np.fft.fftshift와 동일한지 체크해보는 코드를 작성해 테스트해보았다. 성공적으로 배열이 뒤바뀐 것을 확인할 수 있었다.

· *ifftshift(img)*

```
41  def ifftshift(img):
42      '''
43      This function should do the reverse of what fftshift function does.
44      You should not use any kind of built in shift function. Please implement your own.
45      '''
46
47      row = img.shape[0]
48      col = img.shape[1]
49
50      half_row = int(np.ceil(row/2))
51      half_col = int(np.ceil(col/2))
52
53      shift_img =img.copy()
54
55
56      if(row%2 == 0):
57          shift_img[:half_row] = img[half_row:].copy()
58          shift_img[half_row:] = img[:half_row].copy()
59      else:
60          shift_img[:half_row] = img[half_row-1:].copy()
61          shift_img[half_row:] = img[:half_row-1].copy()
62
63      temp = shift_img.copy()
64
65      for i in range(row):
66          if(col%2 == 0):
67              shift_img[i][:half_col] = temp[i][half_col:].copy()
68              shift_img[i][half_col:] = temp[i][:half_col].copy()
69
70          else:
71              shift_img[i][:half_col] = temp[i][half_col-1:].copy()
72              shift_img[i][half_col:] = temp[i][:half_col-1].copy()
73
74
75      return shift_img
```

ifftshift의 코드

ifftshift는 fftshift로 변환한 배열을 다시 원래 배열로 되돌리는 함수다. fftshift에서 적용한 전략을 그대로 사용하여 ifftshift 또한 구현하였다. 하지만 행렬의 길이가 홀수일 때 나누어야 하는 기준선이 변하기 때문에 그 부분에 대한 코드만 고쳐주었다. Else 밑의 코드를 보면 바뀐 부분을 확인할 수 있다.

```
    a = np.array([[9,7,8],[3,1,2],[6,4,5]])
    b = ifftshift(a)
    c = np.fft.ifftshift(a)
    print("ORIGINAL ARRAY")
    print(a)
    print("My ifftshift")
    print(b)
    print("np.fft.ifftshift")
    print(c)
    exit()
```

```
ORIGINAL ARRAY
[[9 7 8]
 [3 1 2]
 [6 4 5]]
My ifftshift
[[1 2 3]
 [4 5 6]
 [7 8 9]]
np.fft.ifftshift
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

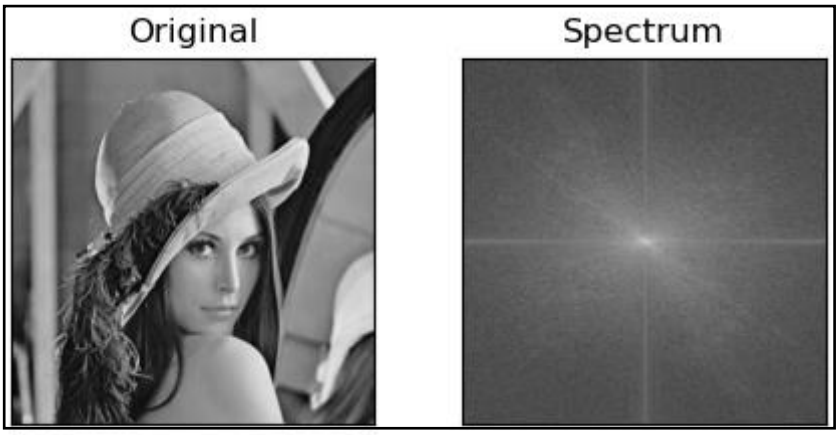
fftshift에서 바뀐 배열에 ifftshift와 np.fft.ifftshift를 적용한 후 비교하여 제대로 구현되었는지 확인할 수 있다.

· *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.
    '''

    fft_image = np.fft.fft2(img)
    shift_fft = fftshift(fft_image)

    spectrum = np.log1p(np.abs(shift_fft))


    return spectrum
```

fm_specturm의 코드

fm_specturm은 이미지를 frequency magnitude spectrum으로 변환해준다. 푸리에 이미지를 얻기 위해, np.fft.fft2 함수를 통해 이미지를 변환시켜주고 fftshift로 위치를 재조정해주었다. 변환한 이미지의 magnitude값을 얻기 위해 np.abs를 적용한 후, 더 명확하게 보기 위해 크기를 재조정하기 위해서 np.log1p를 적용해주었다. 아래의 사진을 통해 성공적으로 원본 이미지의 spectrum을 확인할 수 있다.



• *low_pass_filter(img, r)*

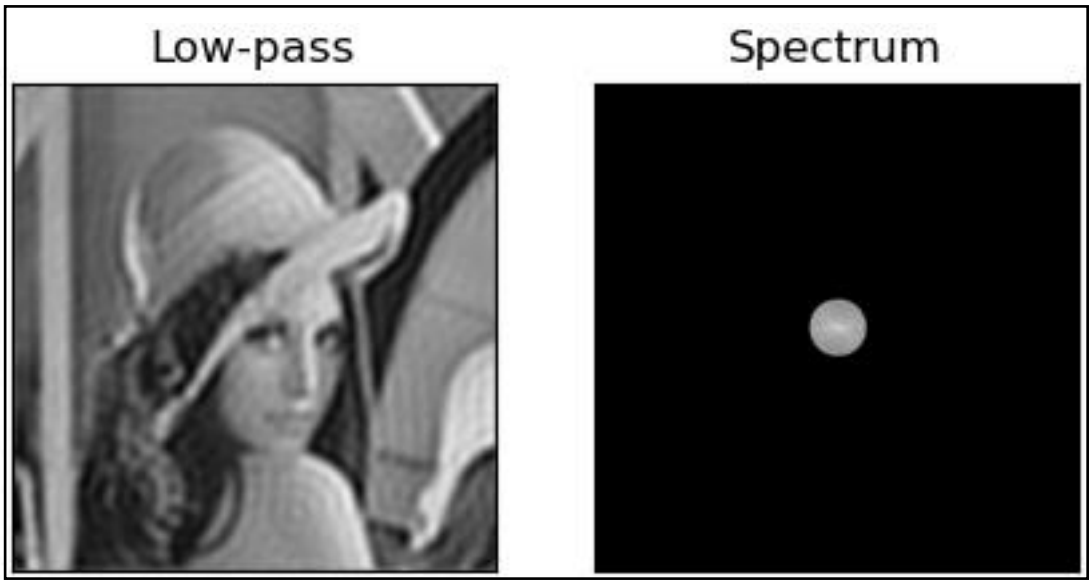
```python
def low_pass_filter(img, r=30):
    '''
    This function should return an image that goes through low-pass filter.
    '''

    row = img.shape[0]
    col = img.shape[1]
    middle_row = int(np.ceil(row/2))
    middle_col = int(np.ceil(col/2))


    fft_image = np.fft.fft2(img)
    shift_image = fftshift(fft_image)

    low_pass = np.zeros((row,col))

    for i in range(row):
        for j in range(col):
            if distance(np.abs(i-middle_row),np.abs(j-middle_col)) <= r:
                low_pass[i,j]=1

    filter_image = shift_image * low_pass


    ishift_image = ifftshift(filter_image)
    ifft_image = np.fft.ifft2(ishift_image)

    final_image = np.real(ifft_image)


    return final_image
```

Low-pass filter의 코드

```
247 def distance(x,y):
248     return np.sqrt(x**2 + y**2)
```

Distance 함수의 코드

로우 패스 필터는 선택한 값 이하의 frequency만 통과시키는 필터를 뜻한다. 98줄에서 105줄에서 이미지 행렬의 길이와 높이, 이미지의 중심 위치의 값을 구하고 이미지를 푸리에 변환 후 fftshift를 적용한다. 이미지의 크기와 같은 사이즈인 0으로 초기화된 low_pass배열을 생성한다. 직접 구현한 distance함수를 사용해 중심 위치에서 현재 위치 사이의 거리를 구하고, 거리가 r 이하일 때만 low_pass[i,j]의 값을 1로 변환해준다. 처리가 완료된 low_pass를 미리 변환해둔 이미지에 곱한다. 이를 통해, r범위 안의 값 외에는 모두 0으로 변해, r 범위 이하의 값만 남길 수 있다. 필터를 거친 이미지를 ifftshift를 적용하고 다시 역 푸리에 변환해준다. 허수 부분의 값은 이미지에 필요하지 않기 때문에 np.real를 통해 배열의 허수 부분을 삭제한 후 return한다.



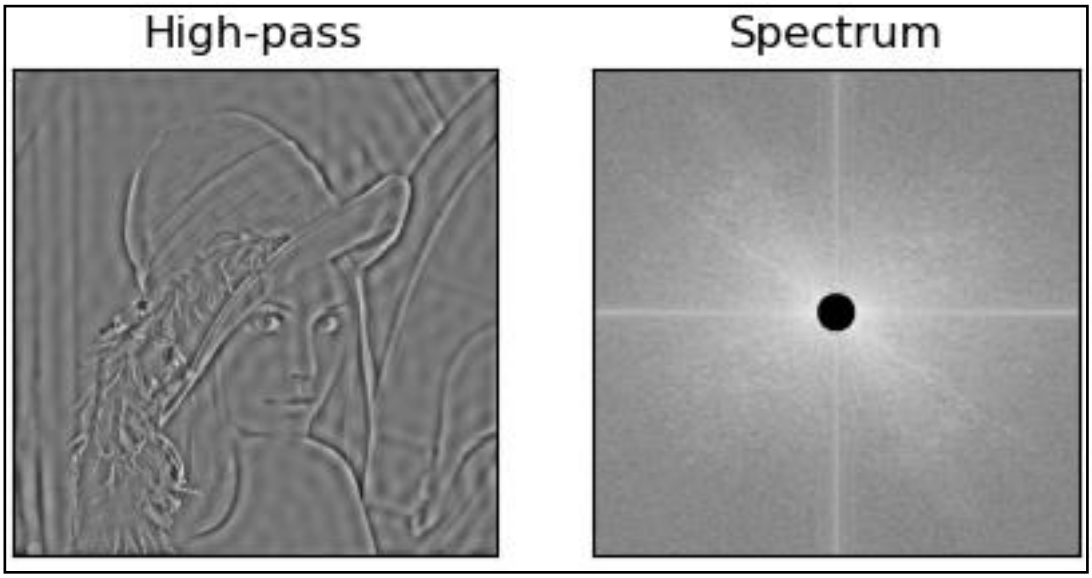
위 이미지는 로우 패스 필터를 통한 이미지와 스펙트럼이다. 사진이 원본 이미지에 비해 흐려졌고, 스펙트럼을 봤을 때 중간의 원을 제외한 나머지 부분이 모두 사라진 것을 확인할 수 있다.

· *high_pass_filter(img, r)*

```python
def high_pass_filter(img, r=20):
    '''
    This function should return an image that goes through high-pass filter.
    '''

    row = img.shape[0]
    col = img.shape[1]
    middle_row = int(np.ceil(row/2))
    middle_col = int(np.ceil(col/2))


    fft_image = np.fft.fft2(img)
    shift_image = fftshift(fft_image)

    high_pass = np.ones((row,col))
    for i in range(row):
        for j in range(col):
            if distance(np.abs(i-middle_row),np.abs(j-middle_col)) <= r:
                high_pass[i,j] = 0

    filter_image = shift_image * high_pass


    ishift_image = ifftshift(filter_image)
    ifft_image = np.fft.ifft2(ishift_image)
    final_image = np.real(ifft_image)

    return final_image
```

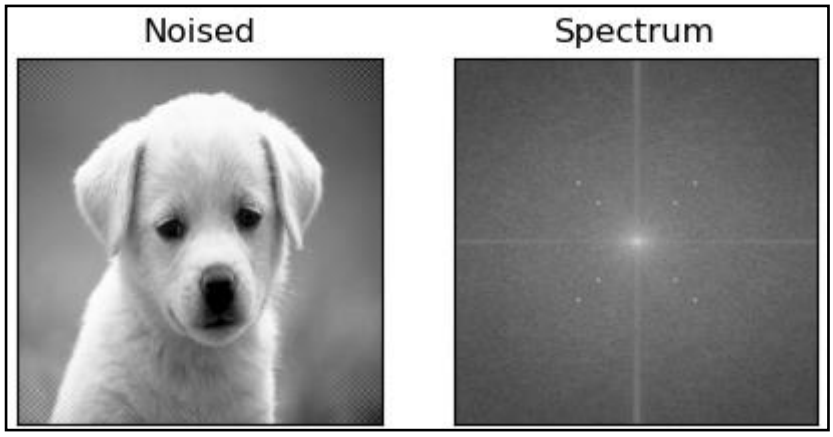
High-pass filter의 코드

하이 패스 필터는 지정한 값보다 높은 frequency만 통과시키는 필터이다. 따라서 로우 패스 필터와 하이패스 필터는 완전히 반대되는 결과를 가진다. 코드는 기본적으로 로우 패스 필터와 같지만, high_pass는 중심에서 r보다 같거나 큰 값만 1로 가지는 배열이다. 이를 위해 high_pass를 이미지와 같은 크기의 1로 초기화 된 배열로 선언하고, r 이하의 거리를 가진 위치의 값만 0으로 바꾸었다. 전과 같이 푸리에 변환한 이미지에 하이 패스 필터를 적용시키고 역 푸리에 변환해 return한다.



이미지에 하이 패스 필터가 적용된 모습을 확인할 수 있다. Matplotlib을 사용했기 때문에, 사진의 모양이 지시 사항과는 조금 다르다. 하지만 스펙트럼에서 r 이하의 값이 성공적으로 사라진 것을 확인할 수 있다.

· denoise1(img)

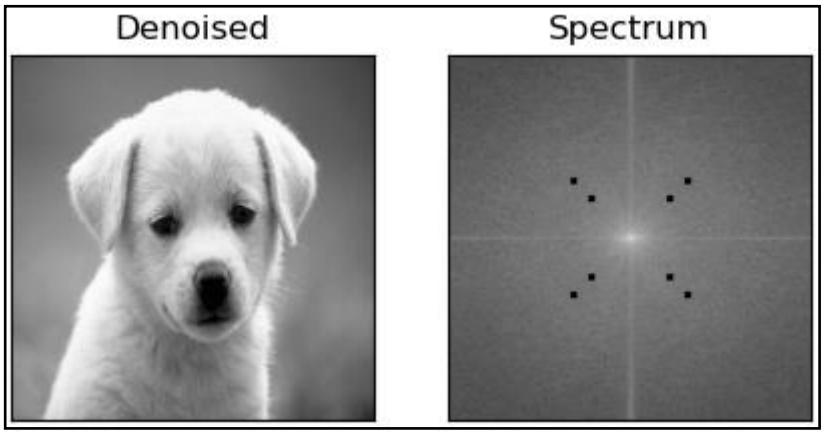


가장 먼저 노이즈가 있는 이미지와 스펙트럼을 확인해보았다. 이미지의 가장자리에는 체크 무늬의 노이즈가 있고, 스펙트럼에 8개의 점이 있는 것을 볼 수 있다. 이 점을 제거하는 것을 목표로 두고 코드를 작성했다.

```python
def denoise1(img):
    '''
    Use adequate technique(s) to denoise the image.
    Hint: Use fourier transform
    '''
    row = img.shape[0]
    col = img.shape[1]
    middle_row = int(np.ceil(row/2))
    middle_col = int(np.ceil(col/2))

    fft_image = np.fft.fft2(img)
    shift_image = fftshift(fft_image)

    denoise = np.ones((row,col))

    for i in range(middle_row-85,middle_row-75):
        for j in range(middle_col-85,middle_col-75):
            denoise[i][j]=0
    for i in range(middle_row+75,middle_row+85):
        for j in range(middle_col+75,middle_col+85):
            denoise[i][j]=0

    for i in range(middle_row-60,middle_row-50):
        for j in range(middle_col-60,middle_col-50):
            denoise[i][j]=0
    for i in range(middle_row+50,middle_row+60):
        for j in range(middle_col+50,middle_col+60):
            denoise[i][j]=0

    for i in range(middle_row-60,middle_row-50):
        for j in range(middle_col+50,middle_col+60):
            denoise[i][j]=0
    for i in range(middle_row+50,middle_row+60):
        for j in range(middle_col-60,middle_col-50):
            denoise[i][j]=0

    for i in range(middle_row-85,middle_row-75):
        for j in range(middle_col+75,middle_col+85):
            denoise[i][j]=0
    for i in range(middle_row+75,middle_row+85):
        for j in range(middle_col-85,middle_col-75):
            denoise[i][j]=0



    filter_image = shift_image * denoise

    ishift_image = ifftshift(filter_image)
    ifft_image = np.fft.ifft2(ishift_image)
    final_image = np.real(ifft_image)


    return final_image
```

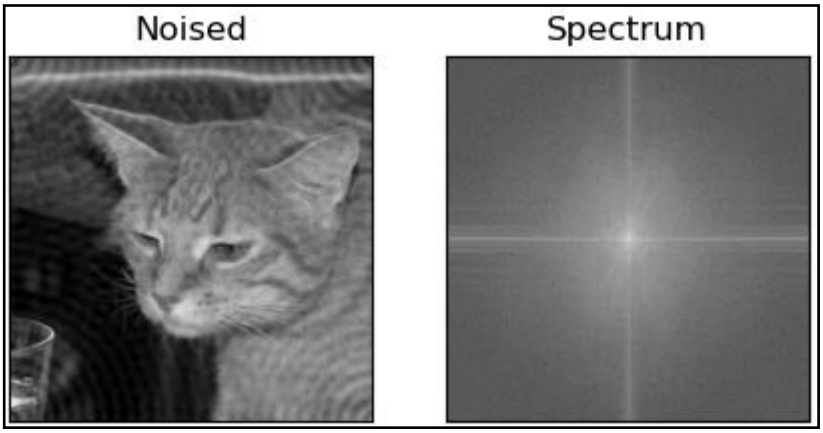
Denoise1의 코드

이와 같이 이미지1의 노이즈를 제거하는 코드를 구현하였다. 먼저, 이미지를 푸리에 변환을 한 후, 이미지의 크기와 같은 1로 초기화 된 배열 denoise를 생성했다. 172줄부터 198줄은 원본 스펙트럼에 있는 점들의 위치를 단순 대입으로 파악해, 그 위치에 해당하는 denoise 배열의 값을 0으로 바꾸는 코드이다. Denoise를 푸리에 변환된 이미지에 곱해 점이 있는 위치의 값을 없애고, 역 푸리에 변환한 후 return한다.



Denoise1 함수로 노이즈가 제거된 이미지의 모습이다. 강아지 사진의 가장자리에 있던 체크무늬 노이즈가 사라지고, 스펙트럼에 존재하는 점들의 값 또한 까맣게 사라진 것을 확인할 수 있다.

· denoise2(img)

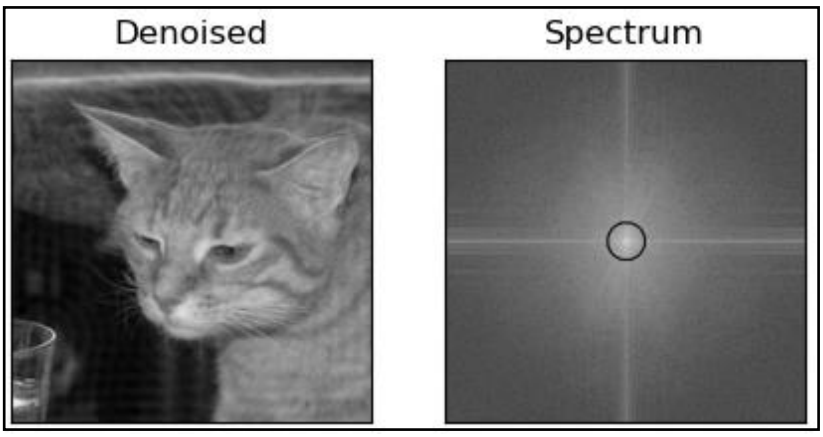


고양이의 이미지에는 파형 모양의 노이즈가 존재하고, 스펙트럼에는 중간에 링 모양의 노이즈가 존재하는 것을 확인할 수 있다. 노이즈를 제거하기 위해 band-reject filter를 구현하기로 결정했다.

```python
def denoise2(img):
    '''
    Use adequate technique(s) to denoise the image.
    Hint: Use fourier transform
    '''

    row = img.shape[0]
    col = img.shape[1]
    middle_row = int(np.ceil(row/2))
    middle_col = int(np.ceil(col/2))


    fft_image = np.fft.fft2(img)
    shift_image = fftshift(fft_image)

    denoise = np.ones((row,col))


    for i in range(row):
        for j in range(col):
            if distance(np.abs(i-middle_row),np.abs(j-middle_col)) >= 25 and distance(np.abs(i-middle_row),np.abs(j-middle_col)) <=28:
                denoise[i][j]= 0


    filter_image = shift_image * denoise


    ishift_image = ifftshift(filter_image)
    ifft_image = np.fft.ifft2(ishift_image)
    final_image = np.real(ifft_image)

    return final_image
```

Denoise2의 코드

226줄까지는 위에 있는 다른 함수들과 같은 준비단계이다. 229줄부터 235줄까지의 코드를 통해서 푸리에 변환된 이미지의 중심에서 25~28 거리에 있는 값만 삭제했다. 그 후, 역 푸리에 변환을 통해 최종 이미지를 return했다.



앞서 구현한 코드를 통해, 원본 이미지의 파형 노이즈가 제거된 것을 확인할 수 있다. 스펙트럼에 있던 원 모양의 노이즈 또한 성공적으로 제거되었다.

Ryan Sunwoo Lee /2022847310

· *dft2(img) and idft2(img)*

$$F(x,y) = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} f(m,n)e^{-j2\pi(x\frac{m}{M}+y\frac{n}{N})}$$

$$f(m,n) = \frac{1}{MN}\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} F(x,y)e^{j2\pi(x\frac{m}{M}+y\frac{n}{N})}$$

첫번째 수식은 2차원 배열의 DFT, 두번째 수식은 2차원 배열의 inverse DFT를 나타낸다.

```
251 def dft2(img):
252     '''
253     Extra Credit.
254     Implement 2D Discrete Fourier Transform.
255     Naive implementation runs in O(N^4).
256     '''
257 
258     row = img.shape[0]
259     col = img.shape[1]
260     result = np.zeros((row,col)).astype(complex)
261 
262     for u in range(row):
263         for v in range(col):
264             for x in range(row):
265                 for y in range(col):
266                     result[u,v] += img[x,y] * np.exp(-2j*np.pi*(u*x/row+v*y/col))
267 
268     return result
269 
270 def idft2(img):
271     '''
272     Extra Credit.
273     Implement 2D Inverse Discrete Fourier Transform.
274     Naive implementation runs in O(N^4).
275     '''
276 
277     row = img.shape[0]
278     col = img.shape[1]
279     result = np.zeros((row,col)).astype(complex)
280 
281     for u in range(row):
282         for v in range(col):
283 
284             temp = complex(0)
285 
286             for x in range(row):
287                 for y in range(col):
288                     temp += img[x,y] * np.exp(2j*np.pi*(u*x/row + v*y/col))
289 
290             result[u,v] = temp/(row*col)
291 
292     return result
```

Ryan Sunwoo Lee /2022847310

다음과 같은 코드를 통해 푸리에 변환을 구현하였다. 이미지의 크기와 같은 0으로 초기화 된 result 배열을 complex타입으로 선언했다. 타입을 complex로 설정한 이유는 허수부까지 저장해야 할 필요가 있기 때문이다. 그 후, 4개의 for문을 통해 제일 안 쪽의 루프에 DFT의 식을 구현하였다. IDFT도 같은 방식으로 구현하였지만, 값을 넣기 전에 이미지의 사이즈로 값을 나누어야 하기 때문에 그 부분에 대한 계산만 더해주었다. DFT는 4개의 for문이 들어가기 때문에 row*col*row*col 만큼의 계산이 필요하다. 따라서 O(M^2 * N^2)의 시간 복잡도를 가져 계산에 많은 시간이 걸리게 된다.