Assignment1 Report

2014147546 송재우

-task1-
다음은 task1의 최소 rms 값 및 해당 결과를 만들어낸 kernel size와 sigma_s, sigma_r 값을 정리한 것이다.

Test	RMS	Filter	Kernel Size	Sigma_s	Sigma_r
test1	8.255267821124486	average	5	-	-
test2	6.750776971827662	bilateral	7	90	90
test3	3.5363150246060364	bilateral	5	75	75
test4	2.364106497006894	bilateral	3	15	15
test5	1.8964557089793916	bilateral	3	15	15

apply_average_filter 함수(Illustration 1)는 kerner_size를 받아 그 제곱으로 1을 나누어 mean 값을 우선 구해준 후 각 픽셀을 돌며 해당 픽셀 주변으로 사이즈에 맞는 dist만큼 다시 돌아 모든 값들에 mean을 곱해서 더해주었다. 이 때 경계선에 있을 경우 이미지 바깥으로 넘어가는 경우엔 모든 값을 0으로 가정하여 더해주었다.

```
def apply_average_filter(img, kernel_size):

    mean = 1 / (kernel_size * kernel_size)

    for row in range(img.shape[0]):
        for col in range(img.shape[1]):
            value = 0
            dist = kernel_size//2
            for partial_row in range(row-dist, row+dist+1):
                for partial_col in range(col-dist, col+dist+1):
                    if partial_row < 0 or partial_col < 0
                                or partial_row >= img.shape[0] or partial_col >=
img.shape[1]:
                        pass
                    else:
                        value += img[partial_row, partial_col] * mean

            img[row, col] = value

    return img
```

다음으로 apply_median_filter 함수이다. (Illustration 2)median 값을 구하기 위해 각 픽셀의 3채널(R, G, B)의 값을 담을 수 있는 배열을 만들고 apply_average_filter 함수에서와 마찬가지로 각 픽셀을 돌고 dist만큼 해당 픽셀 주변을 다시 돌며 값을 각 리스트에 저장하였다. 그리고 최종적으로 numpy.median 함수를 이용하여 중간값을 구한 뒤 새로 업데이트 시켜주었다.

```
def apply_median_filter(img, kernel_size):

    for row in range(img.shape[0]):
        for col in range(img.shape[1]):
            dist = kernel_size//2
            color_b = np.array([], dtype=np.uint8)
            color_g = np.array([], dtype=np.uint8)
            color_r = np.array([], dtype=np.uint8)
            for partial_row in range(row-dist, row+dist+1):
                for partial_col in range(col-dist, col+dist+1):
                    if partial_row < 0 or partial_col < 0
                                or partial_row >= img.shape[0] or partial_col >=
img.shape[1]:
                        color_b = np.append(color_b, 0)
                        color_g = np.append(color_g, 0)
                        color_r = np.append(color_r, 0)
                    else:
                        color_b = np.append(color_b, img[partial_row, partial_col]
[0])
                        color_g = np.append(color_g, img[partial_row, partial_col]
[1])
                        color_r = np.append(color_r, img[partial_row, partial_col]
[2])

            img[row, col] = np.array([np.median(color_b), np.median(color_g),
np.median(color_r)],
                dtype=np.uint8)

    return img
```

마지막으로 apply_bilateral_filter(Illustration 3)는 추가로 sigma_s와 sigma_r을 받았고 픽셀을 탐색하는 방식은 위 두 필터와 동일하다. 각 픽셀의 3 채널에 대하여 따로 처리를 하였고 이미지 경계를 넘어갈 경우엔 역시 0으로 처리하였다. 지시사항대로 space_diff는 유클리드 거리로, range_diff는 L1 거리로 산정하였는데 두 픽셀의 각 3채널의 차이값을 더하여 구하였다. 그리고 각각 g_s, g_r을 가우스 함수 공식을 이용하여 계산하고 두 값을 곱한 값을 sum_weight에다가 저장해주었다. 그리고 픽셀 값에 마찬가지로 두 값을 곱하여 리스트에 저장해놓았다. 최종적으로 normalize를 위해 리스트에 저장된 값들을 sum_weight로 나눈 후 다 더해서 픽셀 값에 업데이트해주었다.

```
def apply_bilateral_filter(img, kernel_size, sigma_s, sigma_r):
    for row in range(img.shape[0]):
        for col in range(img.shape[1]):
            color_b = []
            color_g = []
            color_r = []
            sum_weight = 0
            dist = kernel_size//2
            for partial_row in range(row-dist, row+dist+1):
                for partial_col in range(col-dist, col+dist+1):
                    if partial_row < 0 or partial_col < 0
```

```
                                    or partial_row >= img.shape[0] or partial_col >=
img.shape[1]:
                            color_b.append(0)
                            color_g.append(0)
                            color_r.append(0)
                        else:
                            space_diff = math.sqrt((row-partial_row)*(row-partial_row)
                                    + (col-partial_col)*(col-partial_col))
                            range_diff = abs(int(img[row, col][0])-
int(img[partial_row, partial_col][0])) \
                                                + abs(int(img[row, col][1])-
int(img[partial_row, partial_col][1])) \
                                                + abs(int(img[row, col][2])-
int(img[partial_row, partial_col][2]))
                            g_s = math.exp(-1*(1/2)*(space_diff/sigma_s)*
(space_diff/sigma_s))
                            g_r = math.exp(-1*(1/2)*(range_diff/sigma_r)*
(range_diff/sigma_r))
                            sum_weight += g_s * g_r
                            color_b.append(img[partial_row, partial_col][0] * g_s *
g_r)
                            color_g.append(img[partial_row, partial_col][1] * g_s *
g_r)
                            color_r.append(img[partial_row, partial_col][2] * g_s *
g_r)

                b_value = 0
                g_value = 0
                r_value = 0
                for i in range(0, kernel_size*kernel_size):
                    b_value += (color_b[i] / sum_weight)
                    g_value += (color_g[i] / sum_weight)
                    r_value += (color_r[i] / sum_weight)

                img[row, col][0] = b_value
                img[row, col][1] = g_value
                img[row, col][2] = r_value

        return img
```

그리고 task1 함수(Illustration4, 5, 6)는 src_img_path는 노이즈가 포함된 원래 이미지를, clean_img_path는 노이즈가 제거된 깨끗한 이미지를 받도록 하고 dist_img_path는 처리된 이미지를 저장할 이미지이름을 지정하도록 했다. average, median, bilateral 필터를 각 사이즈 별로 적용하는데 bilateral은 두 시그마 값을 90부터 15까지 점진적으로 줄여나가면서 이미지 처리를 해보았다. 각각의 필터에서 낸 rms와 이미지 배열을 별도의 리스트에 저장해놓고 최종적으로 가장 작은 rms를 갖는 필터와 사이즈, 그리고 그 해당 이미지만 꺼내어 사진으로 저장하였다.

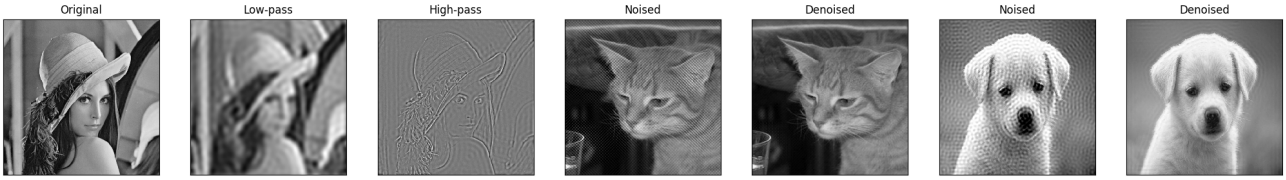
```
def task1(src_img_path, clean_img_path, dst_img_path):
    optimal_solution = ""
    optimal_size = 0
    total_rms = []
```

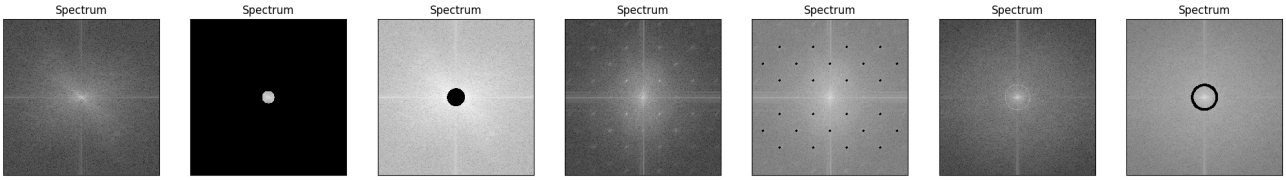
```
    src_img = cv2.imread(src_img_path)
    clean_img = cv2.imread(clean_img_path)

    print(src_img_path + "\n")

    original_rms = utils.calculate_rms_cropped(src_img, clean_img)
    print("original rms: " + str(original_rms) + "\n")

    average_rms = []
    average_imgs = []
    for size in range(3, 10, 2):
        src_img_average = cv2.imread(src_img_path)
        dist_img = apply_average_filter(src_img_average, size)
        rms = utils.calculate_rms_cropped(clean_img, dist_img)
        print("average filter" + " kernel size: " + str(size) + " rms: " +
str(rms) + "\n")
        average_rms.append(rms)
        average_imgs.append(dist_img)

    total_rms.append(min(average_rms))
    average_index = average_rms.index(min(average_rms))
    average_size = average_index * 2 + 3

    median_rms = []
    median_imgs = []
    for size in range(3, 10, 2):
        src_img_median = cv2.imread(src_img_path)
        dist_img = apply_median_filter(src_img_median, size)
        rms = utils.calculate_rms_cropped(clean_img, dist_img)
        print("median filter" + " kernel size: " + str(size) + " rms: " + str(rms)
+ "\n")
        median_rms.append(rms)
        median_imgs.append(dist_img)

    total_rms.append(min(median_rms))
    median_index = median_rms.index(min(median_rms))
    median_size = median_index * 2 + 3

    sigmas = [
        [90, 90],
        [75, 75],
        [60, 60],
        [45, 45],
        [30, 30],
        [15, 15]
    ]

    bilateral_rms = []
    bilateral_sigmas = []
    bilateral_imgs = []
    for size in range(3, 10, 2):
        for sigma_pair in sigmas:
            src_img_bilateral = cv2.imread(src_img_path)
```

```
            dist_img = apply_bilateral_filter(src_img_bilateral, size,
sigma_pair[0], sigma_pair[1])
            rms = utils.calculate_rms_cropped(clean_img, dist_img)
            print("bilateral filter" + " kernel size: " + str(size)
                  + " sigma_s: " + str(sigma_pair[0])
                  + " sigma_r: " + str(sigma_pair[1])
                  + " rms: " + str(rms) + "\n")
            bilateral_rms.append(rms)
            bilateral_sigmas.append(sigma_pair)
            bilateral_imgs.append(dist_img)

    total_rms.append(min(bilateral_rms))
    bilateral_index = bilateral_rms.index(min(bilateral_rms))
    optimal_sigma_s = bilateral_sigmas[bilateral_rms.index(min(bilateral_rms))][0]
    optimal_sigma_r = bilateral_sigmas[bilateral_rms.index(min(bilateral_rms))][1]
    bilateral_size = (bilateral_index // len(sigmas)) * 2 + 3

    optimal_rms = min(total_rms)
    if total_rms.index(optimal_rms) == 0:
        optimal_solution = "average"
        optimal_size = average_size
        cv2.imwrite(dst_img_path, average_imgs[average_index])
    elif total_rms.index(optimal_rms) == 1:
        optimal_solution = "median"
        optimal_size = median_size
        cv2.imwrite(dst_img_path, median_imgs[median_index])
    elif total_rms.index(optimal_rms) == 2:
        optimal_solution = "bilateral"
        optimal_size = bilateral_size
        cv2.imwrite(dst_img_path, bilateral_imgs[bilateral_index])

    print("Optimal Solution: " + optimal_solution + "\n"
          + "Optimal Size: " + str(optimal_size) + "\n"
          + "Optimal rms: " + str(optimal_rms) + "\n")

    if optimal_solution == "bilateral":
        print("Optimal sigma_s: " + str(optimal_sigma_s) + " "
              + "Optimal sigma_r: " + str(optimal_sigma_r) + "\n")
```

위와 같은 구조로 각 필터 함수와 task1 함수를 짰고 나온 이미지를 저장하면서 동시에 깨끗한 이미지와 utils.calculate_rms_cropped 함수를 이용하여 rms 값을 계산, 가장 rms 값이 작게 나오는 필터, 사이즈, 그리고 그 필터가 bilateral 일 경우 그 시그마까지 출력하도록 만들었다.

-task2-
task2 수행 결과는 다음과 같다.





위와 같이 모든 이미지가 스펙트럼으로 표현이 되었고 각 필터가 정상적으로 입혀졌음을 알 수 있다.
fm_spectrum 함수에선 ndarray 형식으로 표현된 이미지를 numpy.fft.fft2를 이용하여 complex형으로 바꿔준 후 이미지화를 위해 다시 abs를 이용, real value로 바꿔주었다. 밝은 부분이 가장 자리에 치우쳐 있어 visualizing에 적합하지 않으므로 shift를 해주었는데 여기선 numpy.roll함수를 사용하여 각 배열의 절반 전반부와 후반부를 바꿔주었다. 최종적으로 산출된 이미지 값은 값의 차이가 너무 커서 이미지가 제대로 표현되지 않으므로 자연로그를 사용하여 값을 조정해주었고 밝기 조절을 위해 임의의 값 30을 곱해주었다. 그리고 log안에 0이 들어가는 것을 방지하기 위해 0.001이라는 임의의 값을 더해주었다. 작은 값이므로 영향을 크게 주지 않는다.

```
def fm_spectrum(img):
    fft_img = abs(np.fft.fft2(img))
    size = len(fft_img)

    for i in range(size):
        fft_img[i] = np.roll(fft_img[i], size//2)

    fft_img = np.roll(fft_img, size // 2, axis=0)

    return 30 * np.log(fft_img+0.001)
```

low_pass_filter에선 이미지와 동일한 크기의 필터를 만들고 모든 값을 0으로 초기화시킨 후 중심 좌표를 정한 후 중심과의 거리가 threshold보다 작은 점들의 경우만 1로 만들어주었다. 이와 같이 만들어진 필터를 푸리에 변환한 이미지와 곱해주었고 다시 원래 이미지로 돌려주는 작업을 수행 후에 return 해주었다.

```
def low_pass_filter(img, th=20):
    fft_img = np.fft.fft2(img)
    size = len(fft_img)

    for i in range(size):
        fft_img[i] = np.roll(fft_img[i], size//2)

    fft_img = np.roll(fft_img, size // 2, axis=0)

    filter = np.zeros((img.shape[0], img.shape[1]), dtype=np.float32)
```

```
    center = [(img.shape[0]-1)/2, (img.shape[1]-1)/2]

    for row in range(img.shape[0]):
        for col in range(img.shape[1]):
            dist = np.sqrt((row-center[0])*(row-center[0]) + (col-center[1])*(col-
center[1]))
            if dist < th:
                filter[row][col] = 1.0

    fft_img = filter * fft_img

    for i in range(size):
        fft_img[i] = np.roll(fft_img[i], size // 2)

    fft_img = np.roll(fft_img, size // 2, axis=0)

    ifft_img = np.fft.ifft2(fft_img)
    original_img = np.real(ifft_img)

    return original_img
```

high_pass_filter도 마찬가지의 방법으로 진행하였다. 다른 점은 중심과의 거리가 threshold보다 클 경우 1로 만들고 나머진 0으로 유지시켰다는 점이다.

```
def high_pass_filter(img, th=30):
    fft_img = np.fft.fft2(img)
    size = len(fft_img)

    for i in range(size):
        fft_img[i] = np.roll(fft_img[i], size // 2)

    fft_img = np.roll(fft_img, size // 2, axis=0)

    filter = np.zeros((img.shape[0], img.shape[1]), dtype=np.float32)
    center = [(img.shape[0] - 1) / 2, (img.shape[1] - 1) / 2]

    for row in range(img.shape[0]):
        for col in range(img.shape[1]):
            dist = np.sqrt((row - center[0]) * (row - center[0]) + (col -
center[1]) * (col - center[1]))
            if dist > th:
                filter[row][col] = 1.0

    fft_img = filter * fft_img

    for i in range(size):
        fft_img[i] = np.roll(fft_img[i], size // 2)

    fft_img = np.roll(fft_img, size // 2, axis=0)

    ifft_img = np.fft.ifft2(fft_img)
```

```
    original_img = np.real(ifft_img)

    return original_img
```

첫 번째 denoise의 경우 우선 스펙트럼 변환된 이미지를 통해 체크 노이즈가 어떠한 방식으로 표현되는지를 알 수 있었다. 이미지 스펙트럼 위에 점으로 표시된 노이즈를 제거하기 위해 pyplot 레이아웃 상에서 직접 각 노이즈에 커서를 올려 좌표를 확인할 수 있었고 해당 노이즈를 중심으로 거리가 5인 지점의 값들을 0으로 만든 필터를 적용시켰다. 필터의 나머지 부분은 전부 1로 초기화되었다.

```
def denoise1(img):
    fft_img = np.fft.fft2(img)
    size = len(fft_img)

    for i in range(size):
        fft_img[i] = np.roll(fft_img[i], size // 2)

    fft_img = np.roll(fft_img, size // 2, axis=0)

    filter = np.ones((img.shape[0], img.shape[1]), dtype=np.float32)

    noises = [
        [145,35],[365,35],
        [90,90],[200,90],[310,90],[420,90],
        [145,145],[365,145],
        [90,200],[200,200],[310,200],[420,200],
        [145,255],[365,255],
        [90,310],[200,310],[310,310],[420,310],
        [145,365],[365,365],
        [90,420],[200,420],[310,420],[420,420],
        [145,475],[365,475]
    ]

    for row in range(img.shape[0]):
        for col in range(img.shape[1]):
            for point in noises:
                dist = np.sqrt((row - point[0]) * (row - point[0]) + (col -
point[1]) * (col - point[1]))
                if 0 <= dist <= 5:
                    filter[row][col] = 0

    fft_img = filter * fft_img

    for i in range(size):
        fft_img[i] = np.roll(fft_img[i], size // 2)

    fft_img = np.roll(fft_img, size // 2, axis=0)

    ifft_img = np.fft.ifft2(fft_img)
    original_img = np.real(ifft_img)

    cv2.imwrite("denoised1.png", original_img)
```

```
    return original_img
```

denoise2의 경우 스펙트럼 중앙에 원형으로 표시된 노이즈를 제거하기 위해 band width filter를 이용하였다. outer_radius와 inner_radius 값을 정하여 그 사이의 고리 모양을 이루는 점들의 값을 0으로 두고 나머지 부분들의 값을 1로 만든 필터를 곱하였다.

```
def denoise2(img):
    fft_img = np.fft.fft2(img)
    size = len(fft_img)

    for i in range(size):
        fft_img[i] = np.roll(fft_img[i], size // 2)

    fft_img = np.roll(fft_img, size // 2, axis=0)

    filter = np.ones((img.shape[0], img.shape[1]), dtype=np.float32)
    center = [(img.shape[0] - 1) / 2, (img.shape[1] - 1) / 2]

    outer_radius = 45
    inner_radius = 35

    for row in range(img.shape[0]):
        for col in range(img.shape[1]):
            dist = np.sqrt((row - center[0]) * (row - center[0]) + (col -
center[1]) * (col - center[1]))
            if dist < outer_radius and dist > inner_radius:
                filter[row][col] = 0.0

    fft_img = filter * fft_img

    for i in range(size):
        fft_img[i] = np.roll(fft_img[i], size // 2)

    fft_img = np.roll(fft_img, size // 2, axis=0)

    ifft_img = np.fft.ifft2(fft_img)
    original_img = np.real(ifft_img)

    cv2.imwrite("denoised2.png", original_img)

    return original_img
```

(references) https://dsp.stackexchange.com/questions/45581/what-is-the-effect-of-the-natural-logarithm-in-the-frequency-domain