Homework 8

R13525009 羅筠笙

a. Gaussian noise with amplitude = 10

Gaussian noise with amplitude = 10, SNR = 13.597449

box_3x3, SNR = 17.745861

box_5x5, SNR = 14.869506

median_3x3, SNR = 17.655002	median_5x5, SNR = 16.015192
	
open-then-closing, SNR = 13.240792	closing-then-opening, SNR = 13.621662

b. Gaussian noise with amplitude = 30

Gaussian noise with amplitude = 30, SNR = 4.173514

	
box_3x3, SNR = 12.633835	box_5x5, SNR = 13.331065

median_3x3, SNR = 11.091201

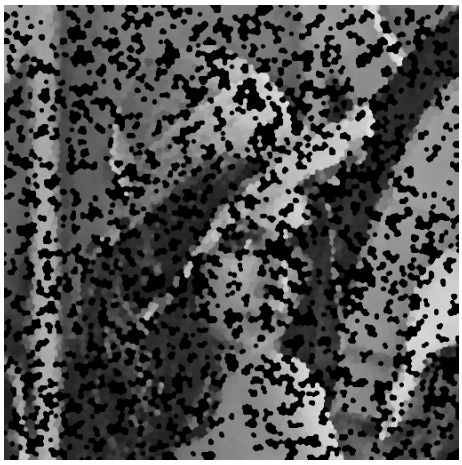
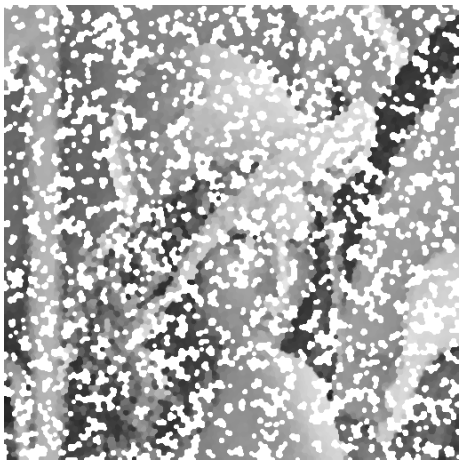
median_5x5, SNR = 12.932721

open-then-closing, SNR = 11.175065

closing-then-opening, SNR = 11.190557

c. Salt-and-pepper noise with probability = 0.1

salt-and-pepper with probability = 0.1, SNR = -2.108260

 box_3x3, SNR = 6.302114	 box_5x5, SNR = 8.470839
 median_3x3, SNR = 14.836461	 median_5x5, SNR = 15.786323
 open-then-closing, SNR = -2.150020	 closing-then-opening, SNR = -2.724415

d. Salt-and-pepper noise with probability = 0.05

salt-and-pepper with probability = 0.05, SNR = 0.882124

box_3x3, SNR = 9.414893

box_5x5, SNR = 11.098395

median_3x3, SNR = 19.160758

median_5x5, SNR = 16.277215

	
opening-then-closing, SNR = 5.435218	closing-then-opening, SNR = 5.602341

```python
# Calculate the SNR
def SNR(origin, noisy):
    m, n = origin.shape
    mu = np.mean(origin)
    mu_n = np.mean(noisy - origin)
    vs = np.mean((origin - mu) ** 2)
    vn = np.mean((noisy - origin - mu_n) ** 2)
    return 20 * np.log10(np.sqrt(vs) / np.sqrt(vn))

# Generate the gaussian noise
def gaussian_noise(imgarr, amp):
    img_gauss_arr = np.zeros(imgarr.shape)
    for i in range(imgarr.shape[0]):
        for j in range(imgarr.shape[1]):
            img_gauss_arr[i, j] = max(0, min(255, imgarr[i, j] + amp * random.gauss(0, 1)))

    snr = SNR(imgarr, img_gauss_arr)
    print(f"SNR for Gaussian noisy with amp = {amp} is {snr:.6f}")
    img_gauss = im.fromarray(img_gauss_arr.astype(np.uint8))
    # img_gauss.save(f"./gauss_{amp}.bmp")
    # img_gauss.show()
    return img_gauss_arr

# Add salt and pepper noise to the image
def salt_and_pepper(imgarr, threshold):
    img_st_arr = np.zeros(imgarr.shape)
```

```
    for i in range(imgarr.shape[0]):
        for j in range(imgarr.shape[1]):
            tmp = random.random()
            if tmp <= threshold:
                img_st_arr[i, j] = 0
            elif tmp > 1 - threshold:
                img_st_arr[i, j] = 255
            else:
                img_st_arr[i, j] = imgarr[i, j]

    # Calculate the SNR
    snr = SNR(imgarr, img_st_arr)
    print(f"SNR for salt & pepper with threshold = {threshold} is {snr:.6f}")

    img_st = im.fromarray(img_st_arr.astype(np.uint8))
    # img_st.save(f"./st_{threshold}.bmp")
    # img_st.show()
    return img_st_arr

# Expand the image array using replicate padding
def expand_with_replicate(arr, pad_size):
    m, n = arr.shape
    result = np.zeros((m + 2 * pad_size, n + 2 * pad_size), dtype=arr.dtype)
    # Fill the borders
    result[pad_size:-pad_size, pad_size:-pad_size] = arr
    result[:pad_size, pad_size:-pad_size] = arr[0, :]  # Top edge
    result[-pad_size:, pad_size:-pad_size] = arr[-1, :]  # Bottom edge
    result[pad_size:-pad_size, :pad_size] = arr[:, 0][:, None]  # Left edge
    result[pad_size:-pad_size, -pad_size:] = arr[:, -1][:, None]  # Right edge
    # Fill the corners
    result[:pad_size, :pad_size] = arr[0, 0]     # Top-left corner
    result[:pad_size, -pad_size:] = arr[0, -1]  # Top-right corner
    result[-pad_size:, :pad_size] = arr[-1, 0]  # Bottom-left corner
    result[-pad_size:, -pad_size:] = arr[-1, -1]     # Bottom-right corner
    return result

# Define box filter
```

```
64. def box_filter(img, k, noise_name):
65.     pad_size = (k - 1) // 2
66.     img_padded = expand_with_replicate(img, pad_size)
67.     m, n = img.shape
68.     result = np.zeros((m, n), dtype=float)
69.
70.     for i in range(m):
71.         for j in range(n):
72.             result[i, j] = img_padded[i:i + k, j:j + k].mean()
73.
74.     result_img = im.fromarray(result.astype(np.uint8))
75.     # result_img.show()
76.     result_img.save(f"./box_{k}_{noise_name}.bmp")
77.
78.     snr = SNR(img_arr, result)
79.     print(f"SNR for box filter {k}x{k}: {snr:.6f}")
80.
81. # Define median filter
82. def median(img, k, noise_name):
83.     pad_size = (k - 1) // 2
84.     img_padded = expand_with_replicate(img, pad_size)
85.     result = np.zeros([img_size0, img_size1])
86.
87.     for i in range(img_size0):
88.         for j in range(img_size1):
89.             result[i, j] = np.median(img_padded[i:i+k, j:j+k])
90.
91.     result_img = im.fromarray(result.astype(np.uint8))
92.     # result_img.show()
93.     result_img.save(f"./median_{k}_{noise_name}.bmp")
94.
95.     snr = SNR(img_arr, result)
96.     print(f"SNR for mdeian filter {k}x{k}: {snr:.6f}")
97.
98. # Define the kernel the hw asked
99. oct_kernel = np.array([[0, 1, 1, 1, 0],
100.                        [1, 1, 1, 1, 1],
101.                        [1, 1, 1, 1, 1],
102.                        [1, 1, 1, 1, 1],
103.                        [0, 1, 1, 1, 0]])
104.
```

```
105. def dilation(img, kernel):
106.     kernel_size = len(kernel)
107.     dilation_img = np.zeros_like(img)
108. 
109.     for i in range(img_size0):
110.         for j in range(img_size1):
111.             max_value = 0
112.             for ki in range(kernel_size):
113.                 for kj in range(kernel_size):
114.                     ni, nj = i + ki - kernel_size // 2, j + kj - kernel_s
    ize // 2
115.                     if 0 <= ni < img_size0 and 0 <= nj < img_size1:
116.                         if kernel[ki][kj] == 1:
117.                             pixel_value = img[ni][nj]
118.                             if pixel_value > max_value:
119.                                 max_value = pixel_value
120.             dilation_img[i, j] = max_value
121. 
122.     return dilation_img.astype(np.uint8)
123. 
124. def erosion(img, kernel):
125.     kernel_size = len(kernel)
126.     erosion_img = np.zeros_like(img)
127. 
128.     for i in range(img_size0):
129.         for j in range(img_size1):
130.             min_value = 255
131.             for ki in range(kernel_size):
132.                 for kj in range(kernel_size):
133.                     ni, nj = i + ki - kernel_size // 2, j + kj - kernel_s
    ize // 2
134.                     if 0 <= ni < img_size0 and 0 <= nj < img_size1:
135.                         if kernel[ki][kj] == 1:
136.                             pixel_value = img[ni][nj]
137.                             if pixel_value < min_value:
138.                                 min_value = pixel_value
139.             erosion_img[i, j] = min_value
140. 
141.     return erosion_img.astype(np.uint8)
142. 
```

```
143. def opening(img, kernel):
144.     return (dilation(erosion(img, kernel), kernel))
145. 
146. def closing(img, kernel):
147.     return (erosion(dilation(img, kernel), kernel))
148. 
149. def opening_then_closing(img, noise_name="", kernel=oct_kernel):
150.     otc_arr = closing(opening(img, kernel), kernel)
151.     snr = SNR(img_arr/255, otc_arr/255)
152.     print(f"SNR for opening-then-closing filter : {snr:.6f}")
153. 
154.     otc = im.fromarray(otc_arr)
155.     otc.save(f"otc_{noise_name}.bmp")
156. 
157.     return otc_arr
158. 
159. def closing_then_opening(img, noise_name="", kernel=oct_kernel):
160.     cto_arr = opening(closing(img, kernel), kernel)
161.     snr = SNR(img_arr/255, cto_arr/255)
162.     print(f"SNR for closing-then-opening filter : {snr:.6f}")
163. 
164.     cto = im.fromarray(cto_arr)
165.     cto.save(f"cto_{noise_name}.bmp")
166. 
167.     return cto_arr
```