Image Processing Homework 5

吴嘉骜 21307130203

2023 年 11 月 12 日

1

Implement the generation of noise.
For images of the brain, heart (or other images), generate the following two different types of noise with varying intensities:
(1) Generate white noise;
(2) Generate another type of noise (such as Gaussian, Rayleigh, or salt-and-pepper noise).
Contaminate the images with the generated noise and visually compare the images before and after the noise pollution.
Solution:
The code from `noisegen.py` is shown as follows.

```
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
from numpy.fft import fft2, ifft2, fftshift, ifftshift
...
# Auxiliary functions are descibed below and omitted here.
def white_noise_gen(shape, amplitude):
    '''
    Generate white noise.

    Parameters:
        - shape: the size of the noise, a tuple of (height, width).
        - spectrum: the spectrum of the noise, a constant.
    Returns:
        - noise: the generated noise, a numpy array.
    '''
    f = np.full(shape, amplitude) # generate constant spectrum
    u0, v0 = (shape[0]//2, shape[1]//2) # get the center of the spectrum

    phase = np.exp(2j * np.pi * np.random.random(shape)) # generate random phase
    f = f * phase # generate half of the noise spectrum
```

```python
    for i in range(u0+1, shape[0]):
        for j in range(shape[1]):
            if 0 <= 2*u0-i <= 255 and 0 <= 2*v0-j <= 255:
                f[i][j] = np.conj(f[2*u0-i][2*v0-j])
    f[u0, v0] = amplitude # set the center of the spectrum to real
    noise = np.real(ifft2(ifftshift(f))) # generate white noise in the spatial domain

    return noise

def gaussian_noise_gen(shape, mean, std):
    '''
    Generate gaussian noise.

    Parameters:
        - shape: the size of the noise, a tuple of (height, width).
        - mean: the mean of the gaussian distribution.
        - std: the standard deviation of the gaussian distribution.
    Returns:
        - noise: the generated noise, a numpy array.
    '''
    noise = np.random.normal(mean, std, shape) # generate gaussian random number
    return noise

def rayleigh_noise_gen(shape, a, b):
    '''
    Generate raleigh noise.

    Parameters:
        - shape: the size of the noise, a tuple of (height, width).
        - a, b: parameters of the rayleigh distribution.
    Returns:
        - noise: the generated noise, a numpy array.
    '''
    u = np.random.uniform(0, 1, shape) # generate uniform random number
    noise = a + np.sqrt(-b * np.log(1 - u)) # inverse transform sampling
    return noise

def img_interfere(img, noise=None, s_p=0 , ps=0, pp=0):
    '''
    Interfere the image with the noise.

    Parameters:
        - img: the image to be interfered, a numpy array.
        - noise: the noise to interfere the image, a numpy array. For salt and pepper noise, the input noise is None.
        - s_p: a flag to indicate whether the noise is salt and pepper noise. 0 for no, 1 for yes.
        - ps: the probability of salt noise.
        - pp: the probability of pepper noise.
```

```
    Returns:
        - img_interfered: the interfered image, a numpy array.
    '''
    if s_p ==0:
        img_interfered = img + noise
        return img_interfered
    else:
        # salt and pepper noise
        img_interfered = img.copy()
        for i in range(img.shape[0]):
            for j in range(img.shape[1]):
                if np.random.random() < ps:
                    img_interfered[i][j] = 255
                elif np.random.random() < pp:
                    img_interfered[i][j] = 0
        return img_interfered
```

We first interpret the code structure briefly, and later we will explain the key functions in detail.

read_img: Reads an image from the given path and converts it to a grayscale numpy array.

show_image: Displays the image using matplotlib with specified figure size and colormap.

img_modify: Normalizes and processes the image for display based on the specified modification type, including logarithmic transformation, clipping, and scaling.

show_spectrum: Computes and displays the frequency spectrum of the image.

show_spectrum2: Displays the frequency spectrum from a given discrete Fourier transform (DFT).

white_noise_gen: Generates white noise with the specified shape and amplitude.

gaussian_noise_gen: Generates Gaussian noise with the specified shape, mean, and standard deviation.

rayleigh_noise_gen: Generates Rayleigh noise with the specified shape, location parameter, and scale parameter.

img_interfere: Interferes the image with the specified noise, including salt-and-pepper noise.

Details:

For white noise, we generate a constant spectrum, and then apply a random phase spectrum to it. To ensure the noise in the spatial domain is real, we set the frequency domain conjugate symmetric.

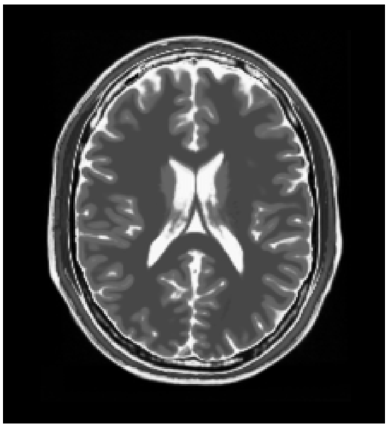
For Gaussian noise, we directly generate normal distribution random numbers by `np.random.normal`.

For Rayleigh noise, we generate random numbers by the inverse transform method. Note tbar the cumulative distribution function of Rayleigh distribution is $F(z;a,b) = 1 - e^{-\frac{(z-a)^2}{2b^2}}$, where a is the location parameter and b is the scale parameter. Let u be a uniform random variable in $[0,1]$, then $F^{-1}(u;a,b) = a + b\sqrt{-2\ln(1-u)}$. Thus, we can generate Rayleigh distribution random numbers by applying F^{-1} to uniform random numbers.

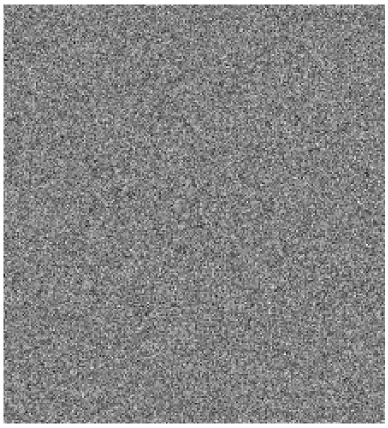
Above three noises can be added to the image simply. For salt-and-pepper noise, however, we determine the polluting probability by comparing u with a preset threshold p, and directly interfere the image with it.

Results:

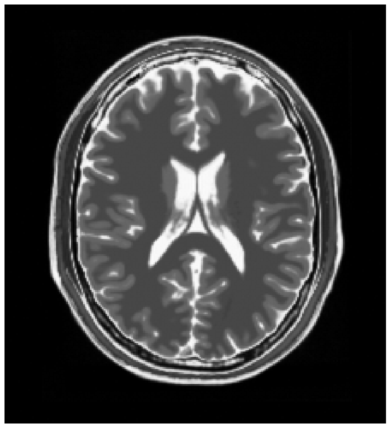
The results are presented in Figure 1 and Figure 2.



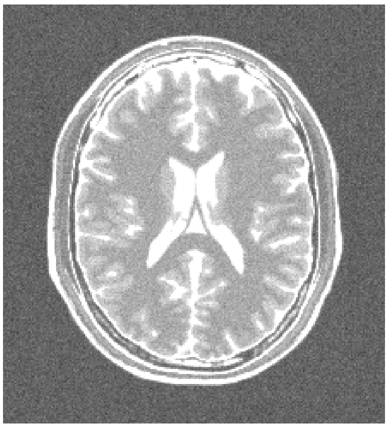
(a) Original image



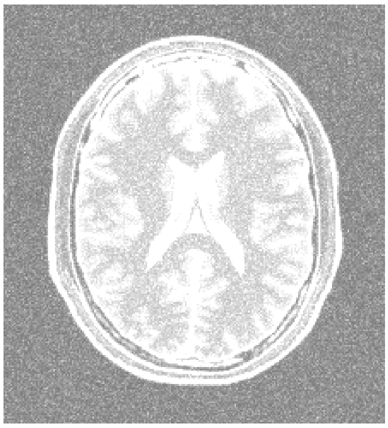
(b) White noise spectrum $A = 100$



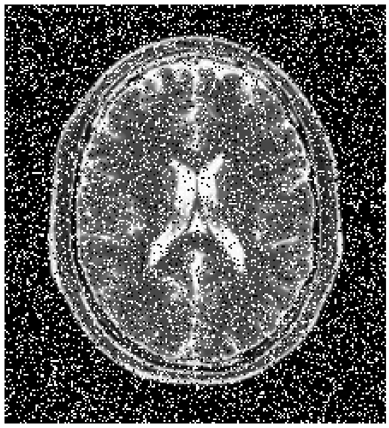
(c) White noise pollution



(d) Gaussian noise pollution, $\mu = 100$, $\sigma = 10$



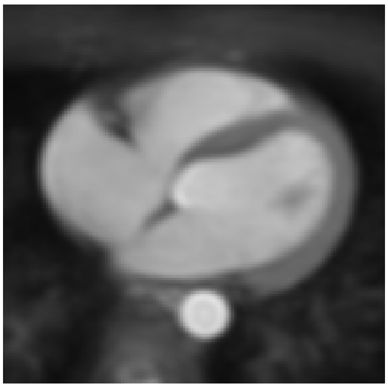
(e) Rayleigh noise pollution, $a = 120$, $b = 1000$



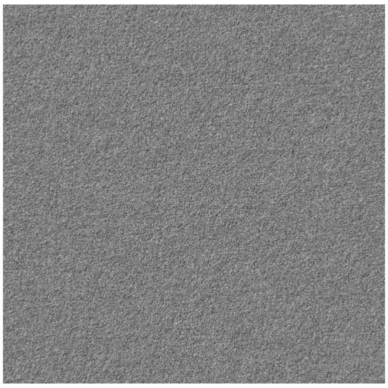
(f) Salt-and-pepper noise pollution, $p_s = 0.15$, $p_p = 0.08$

Figure 1: Brain image after noise pollution

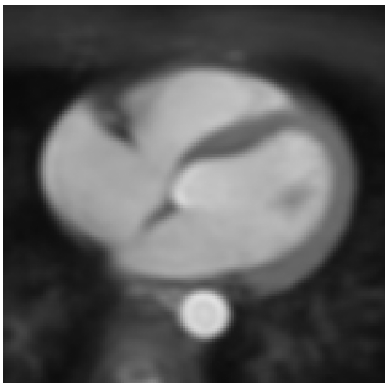
Analysis:
White noise is hard to observe visually in the polluted image, but it can be seen from the spectrum tbar the noise is evenly distributed in the frequency domain.
Gaussian noise is characterized by the mean and standard deviation, which determine the brightness and the intensity range, respectively. Both polluted images are blurred and brighter than the original image.
Rayleigh noise is characterized by parameters a and b, which determine the shift in the brightness and the probability of the noise, respectively. The effect is similar to Gaussian noise.
For salt-and-pepper noise, It is observable tbar the larger the probability of salt noise, the brighter the noise points, and the larger the probability of pepper noise, the darker the noise points.



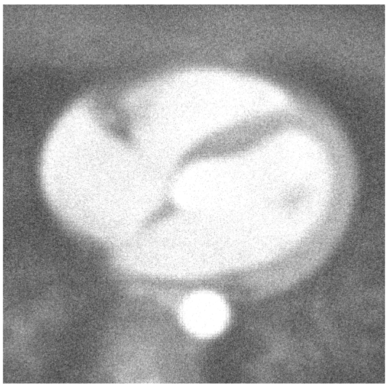
(a) Original image



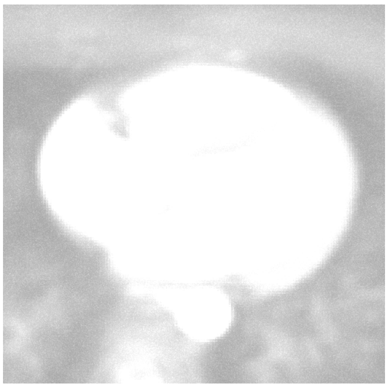
(b) White noise spectrum $A = 200$



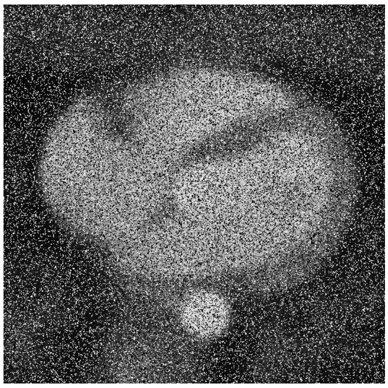
(c) White noise pollution



(d) Gaussian noise pollution, $\mu = 80$, $\sigma = 20$



(e) Rayleigh noise pollution, $a = 150$, $b = 200$



(f) Salt-and-pepper noise pollution, $p_s = 0.13$, $p_p = 0.27$

Figure 2: Heart image after noise pollution

2

Implement an optimal notch filter, and test its effect with images.

Solution:

The code from `opt_nf.py` is shown as follows.

```python
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
...
# Auxiliary functions are the same as noisegen.py, and omitted here.
def ghpf_shift(img, d0, u0, v0):
    '''
    Gaussian high pass filter (GHPF) with center shifted to (u0, v0).

    Parameters:
        - img: the input image, a 2D numpy array
        - d0: the cutoff frequency
        - u0, v0: the center coordinates of the highpass filter

    Returns:
        - filter_transfun: the filter transfer function of GHPF, with size m*n
    '''
    m, n = img.shape
    filter_transfun = np.zeros((m, n))
    for u in range(m):
        for v in range(n):
            d2 = (u-u0)**2 + (v-v0)**2
            filter_transfun[u, v] = 1 - np.exp(-d2/(2*d0**2))
    return filter_transfun

def notch_reject(img, coord, d0):
    '''
    Notch reject filter.

    Parameters:
        - img: the input image, a 2D numpy array
        - coord: the center coordinates of each highpass filter, k*2 array, k is the number of filters
        - d0: the cutoff frequency of the highpass filter

    Returns:
        - filter_transfun: the filter transfer function of notch reject filter, with size m*n
    '''
    m, n = img.shape
    k = coord.shape[0]
    nr = np.ones((m,n))
```

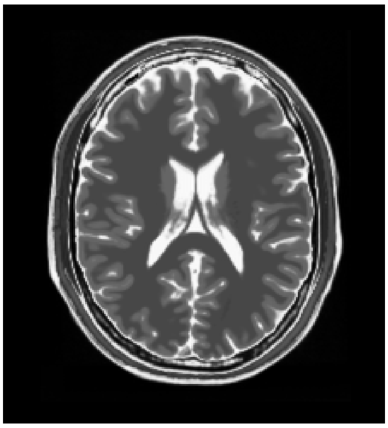
```
    for i in range(k):
        u, v = coord[i]
        nr *= ghpf_shift(img, d0, u, v) * ghpf_shift(img, d0, m-u, n-v)
    return nr

def notch_pass(img, coord, d0):
    '''
    Notch pass filter.
    '''
    return 1-notch_reject(img, coord, d0)

def optimum_notch(img, notch, m1, n1):
    '''
    Optimum notch filter.

    Parameters:
        - img: the input image, a 2D numpy array
        - notch: the notch filter transfer function, the same size as img
        - m1, n1: the size of the neighborhood, two odd integers

    returns:
        - img_filtered: the filtered image, a 2D numpy array
    '''
    def meanvalue(img, x, y, m1, n1):
        '''
        Calculate the mean value of the neighborhood of each pixel.
        - x,y: the center coordinates of the neighborhood
        '''
        m, n = img.shape
        img_mean = 0
        i1 = max(0, x-m1//2)
        i2 = min(m, x+m1//2+1)
        j1 = max(0, y-n1//2)
        j2 = min(n, y+n1//2+1)
        img_mean = np.mean(img[i1:i2, j1:j2])
        return img_mean

    m, n = img.shape
    w = np.zeros((m, n))

    # DFT of the image
    fg = np.fft.fft2(img)
    fg = np.fft.fftshift(fg)

    # interference pattern in sptaial domain
    eta = fg * notch
    eta = np.fft.ifftshift(eta)
    eta = np.fft.ifft2(eta)
```

```python
    eta = np.real(eta)

    # weighting function
    for x in range(m):
        for y in range(n):
            eta_bar = meanvalue(eta, x, y, m1, n1)
            eta2_bar = meanvalue(eta**2, x, y, m1, n1)
            geta_bar = meanvalue(img*eta, x, y, m1, n1)
            g_bareta_bar = meanvalue(img, x, y, m1, n1)*eta_bar
            w[x, y] = (geta_bar - g_bareta_bar) / (eta2_bar - eta_bar**2)

    # estimated f
    img_filtered = img - w * eta

    return img_filtered

# test
img_path = './noisy_image.png'
img = read_img(img_path)
m, n = img.shape
show_spectrum(img)

# create the filter transfer function
ys = np.arange(100, 37, -9)
xs = np.arange(105, -14, -17)
coor = np.column_stack((xs, ys)) # coordinates of bursts
k = coor.shape[0]
nr = np.ones((m,n))
for i in range(k):
    u, v = coor[i]
    nr *= ghpf_shift(img, 4.5, u, v) * ghpf_shift(img, 4.5, m-u, n-v)
nr *= ghpf_shift(img, 2, 137.8, 85) * ghpf_shift(img, 2, m-137.8, n-85)
notch = 1 - nr

img_filtered = optimum_notch(img, notch, 7, 5)
img_out = img_modify(img_filtered, modified=2)
show_spectrum(img_out)
show_image(img_out)
```

The key function is `optimum_notch`, which implements the optimum notch filter. It receives the image, a notch filter transfer function, and the size of the neighborhood as input, and returns the filtered image. Inside, we define a function `meanvalue` to calculate the mean value of the neighborhood of each pixel, carefully considering the boundary conditions. Then we calculate the DFT of the image g, and multiply it with the notch filter transfer function. After inverse DFT, we get the interference pattern η in the spatial domain. Then we calculate the weighting function w by the formula $\frac{\overline{g\eta}-\bar{g}\bar{\eta}}{\overline{\eta^2}-\bar{\eta}^2}$ for each (x, y), and the estimated image f is obtained by $f = g - w\eta$.

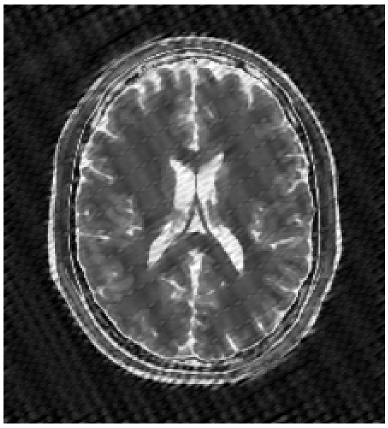
Test and Results:

For testing, we manually pollute the brain image with peroidic noise, and then apply the optimum notch filter to it. We generate a noise pattern by superimposing multiple sine and cosine waves with varying frequencies and amplitudes. This pattern is then rotated to create a diagonal effect and added to the clean brain image, as shown in Figure 3(b). To construct a notch filter transfer function, we estimate the locations of the bursts by visual inspection and mouse hovering on the spectrum. The parameters such as d_0 in GHPF and the window size are determined after several trials. Then we use shifted Gaussian high pass filters product as input to formulate our optimum notch filter. The processed spectrum and estimated image are also presented below.

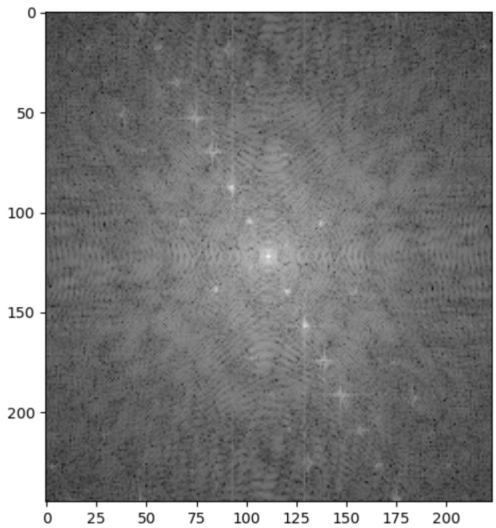


(a) Original image

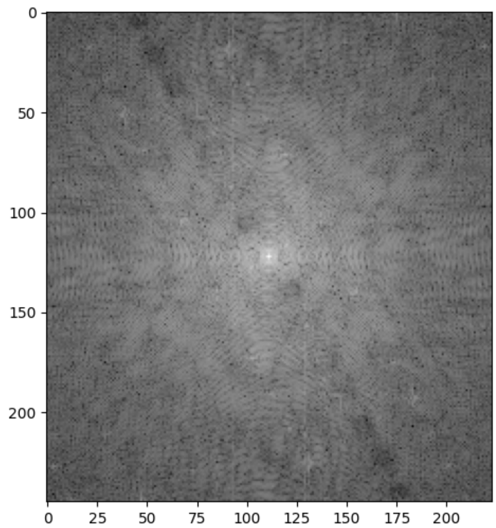
(b) Image after noise pollution



(c) Processed image



(d) Spectrum of polluted image



(e) Filtered spectrum

Figure 3: Optimum notch filter test

Analysis:

From the result, we observe that the spikes in the spectrum have been filtered out, and the evident, disturbing stripe noises are removed. However, examining seriously, we still notice some artifacts in the background and the image boundary. The image is also slightly blurred, with some details within the brain not clear as before. We may perform smoothing and sharpening in further work to get a more satisfactory result.