Assignment 5

Junyi Liao 20307110289

November 11, 2022

0 Image Degradation and Restoration

In image restoration, we model the image degradation as an operator $\mathcal{H}$ and a noise term η,

$$g(x,y) = \mathcal{H}\{f(x,y)\} + \eta(x,y),$$

where f is the input image and g is the degraded image.
A linear and position-invariant operator $\mathcal{H}$ can be specified in spatial domain as the convolution with a degradation function h:

$$g(x,y) = \{h * f\}(x,y) + \eta(x,y),$$

and in frequency domain the convolution theorem implies

$$G(u,v) = H(u,v) \cdot F(u,v) + N(u,v).$$

In the subsequent discussion, we suppose the degradation function h is an impulse, then the degradation reduces to

$$g(x,y) = f(x,y) + \eta(x,y);$$

Then we can construct an estimator $\hat{\eta}$ for noise term, and restore the original image as

$$\hat{f}(x,y) = g(x,y) - \hat{\eta}(x,y).$$

1 Noise Generation

1.1 Noise Models

White Noise has a constant spectrum over the entire frequency domain.
Suppose η is the spatial representation of a white noise term, and $N = \mathfrak{I}\{\eta\}$ is the frequency representation, it implies

$$|N(u,v)| = C, \quad u \in \{0,1,\cdots,M-1\},\ v \in \{0,1,\cdots,N-1\};$$

Periodic Noise distributes periodically in the spatial domain, say it can be expanded as Fourier series:

$$\eta(x,y) = \sum_{m=-\infty}^{+\infty} \sum_{n=-\infty}^{+\infty} c_{m,n} \exp\left\{ j2\pi \left(\frac{mx}{T_x} + \frac{ny}{T_y} \right) \right\},$$

where T_x and T_y is the least positive period on corresponding component.

Gaussian Noise is a random noise model; the noise term on each pixel $\eta(x, y)$ is a random variable, with the density

$$f_\eta(z \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left\{-\frac{(z-\mu)^2}{2\sigma^2}\right\};$$

Pepper-and-Salt Noise is another random noise model; the distribution of the noise term is

$$\mathbb{P}(g(x, y) = z) = \begin{cases} p_s, & z = 2^L - 1, \\ p_p, & z = 0, \\ 1 - p_s - p_p, & z = f(x, y), \end{cases}$$

say the noise dyes an pixel with probability p_s to white (salt) and p_p to black (pepper) or leaves it intact with probability $1 - p_s - p_p$.

1.2 Experiment

For convenience and fidelity, we simply cut off the pixel values outside $[0, 255]$ after adding noise.

1.2.1 White Noise

We first generate a matrix B with each entry independently subject to Uniform$(0, 1)$, and compute its DFT. Then we extract the phase of $DFT(B)$:

$$\varphi(u, v) = \frac{\mathcal{B}(u, v)}{|\mathcal{B}(u, v)|},$$

and multiply the phase with the amplitude $C > 0$ of noise to get the noise in frequency domain:

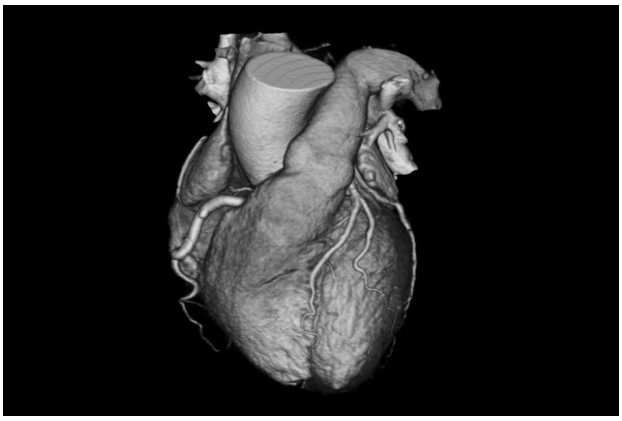
$$N(u, v) = C \cdot \varphi(u, v),$$

and $\eta = IDFT(N)$, which is the white noise term.

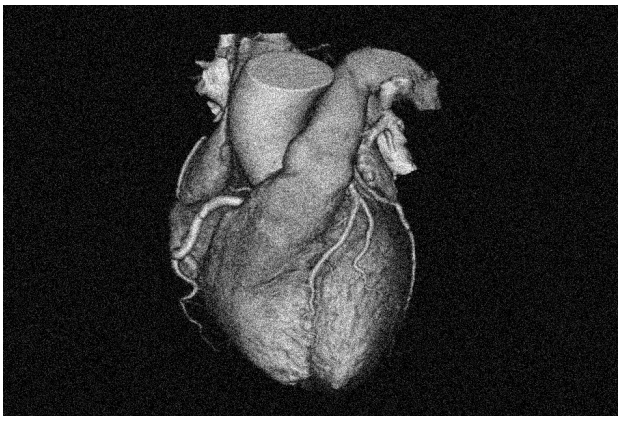
Code The code is shown below. We generate uniform random variables by `numpy.random.rand`.

```
def white_noise(h, w, amp=2.5e+04):
    # Generate random white noise term.
    # Randomly generate a base on Uniform(0,1).
    base = np.random.rand(h, w)
    base_ft = np.fft.fft2(base)
    # Uniformize the spectrum to get a phase matrix.
    phase = base_ft / np.abs(base_ft)
    noise = amp * np.fft.ifft2(phase)
    return np.real(noise).astype(int)

def pollute_wn(img, amp=2.5e+04, nbins=256):
    # Pollute an image with white noise.
    h, w = img.shape
    img_wn = img + white_noise(h, w, amp=amp)
```

```
    # Cut-off.
    img_wn[img_wn >= nbins] = nbins - 1
    img_wn[img_wn < 0] = 0
    return img_wn.astype(int)
```



(1) heart.jpg (Original)



(2) heart.jpg with white noise added

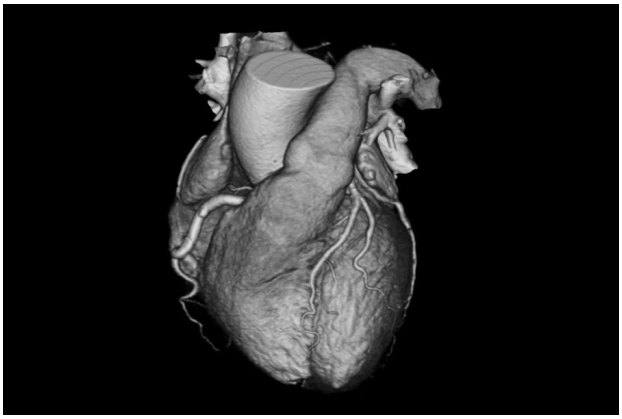
1.2.2 Periodic Noise

We randomly amplify pairs of symmetric points in the frequency domain, which corresponds to signals of certain frequencies, then use IDFT to compute the noise term in spatial domain.

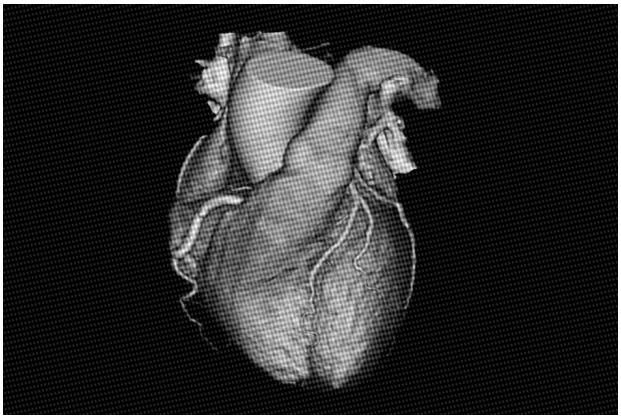
Code The code is shown below. We randomly amplify two pairs of symmetric points in the spatial domain.

```
def period_noise(h, w, amp=2.5e+04, nbins=256):
    # Generate random periodical noise term.
    noise_ft = np.zeros((h, w))
    t = 2 + np.abs(np.random.randn(4))
    u1, v1 = int(h / t[0]), int(w / t[1])
    u2, v2 = int(h / t[2]), int(w / t[3])
    # Amplify two pairs of symmetric points in frequency domain.
    noise_ft[u1, v1] = amp
    noise_ft[h - u1, w - v1] = amp
    noise_ft[u2, w - v2] = amp
    noise_ft[h - u2, v2] = amp
    # Centralize.
    sign = -np.ones((h, w))
    for i in range(h):
        sign[i, i % 2::2] = 1
    noise = np.fft.ifft2(noise_ft) * sign * (nbins - 1)
    return np.real(noise).astype(int)


```

```
def pollute_prd(img, amp=2.5e+04, nbins=256):
    # Pollute an image with periodic noise.
    h, w = img.shape
    img_prd = img + period_noise(h, w, amp=amp)
    # Cut-off.
    img_prd[img_prd >= nbins] = nbins - 1
    img_prd[img_prd < 0] = 0
    return img_prd.astype(int)
```



(3) heart.jpg (Original)



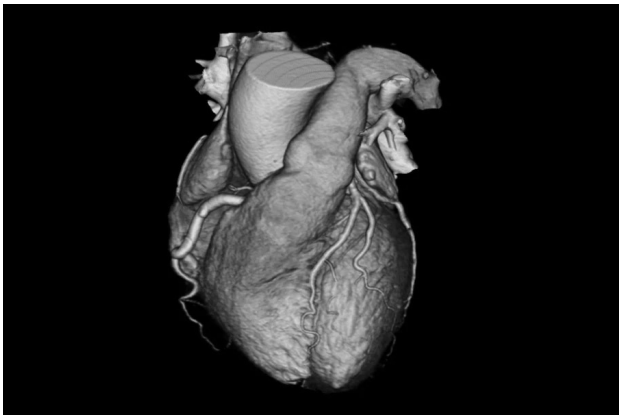
(4) heart.jpg with periodic noise added

1.2.3 Gaussian Noise

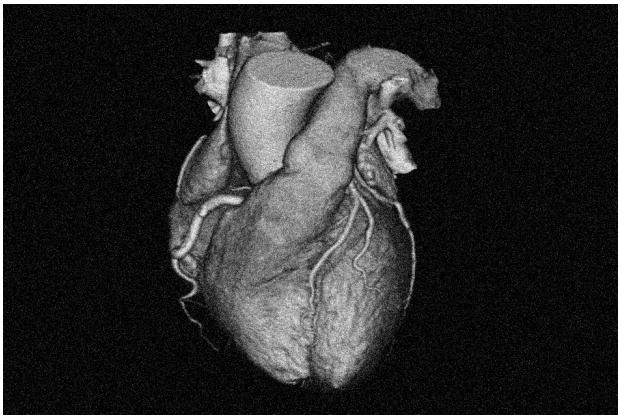
We generate random variables from $N(\mu, \sigma^2)$.

Code The code is shown below. We generate Gaussian random variables by `numpy.random.randn`.

```
def pollute_gn(img, mu=0, amp=25, nbins=256):
    # Pollute an image with Gaussian noise.
    h, w = img.shape
    img_gn = img + (np.random.randn(h, w) + mu) * amp
    img_gn[img_gn >= nbins] = nbins - 1
    img_gn[img_gn < 0] = 0
    return img_gn.astype(int)
```



(5) heart.jpg (Original)



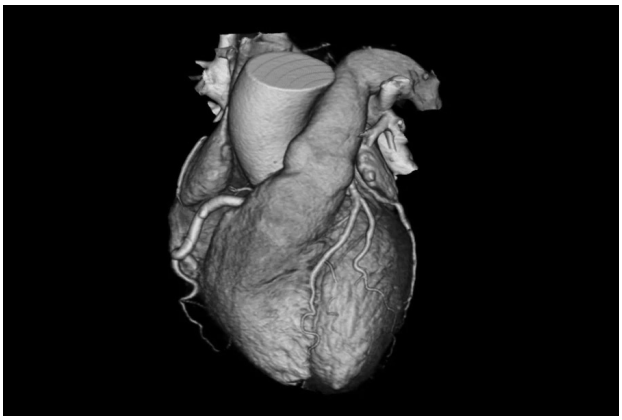
(6) Added Gaussian noise ($\mu = 0$, $\sigma^2 = 625$)

1.2.4 Pepper-and-Salt Noise

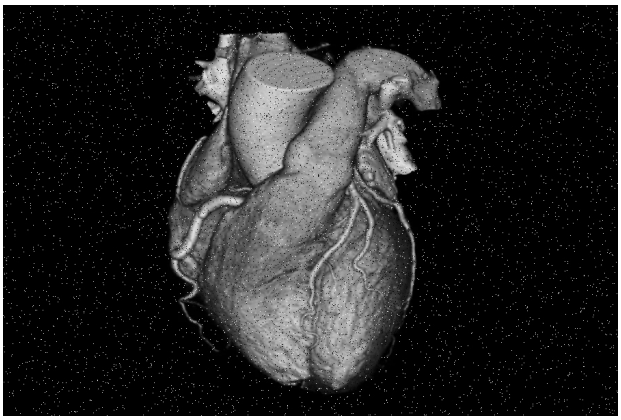
For each pixel, we generate a uniform random variable $U \in \text{Uniform}(0, 1)$; if $U < p_s$, we set the pixel to white; if $U > 1 - p_p$, we set it to black; otherwise leave it intact.

Code The code is shown below. We generate uniform random variables by `numpy.random.rand`.

```
def pollute_pn(img, pepper=0.03, salt=0.01, nbins=256):
    # Pollute an image with pepper-salt noise.
    h, w = img.shape
    img_pn = np.zeros_like(img)
    peppersalt = np.random.rand(h, w)
    img_pn[:, :] = img
    img_pn[peppersalt <= pepper] = 0
    img_pn[peppersalt > 1 - salt] = nbins - 1
    return img_pn.astype(int)
```

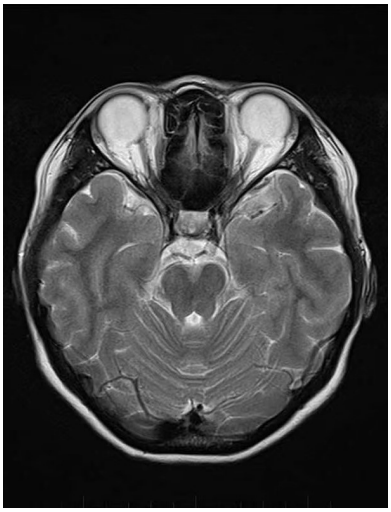


(7) heart.jpg (Original)

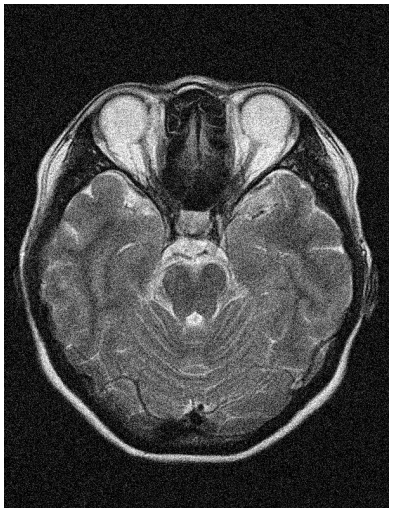


(8) Added pepper-and-salt noise ($p_s = 0.01$, $p_p = 0.03$)

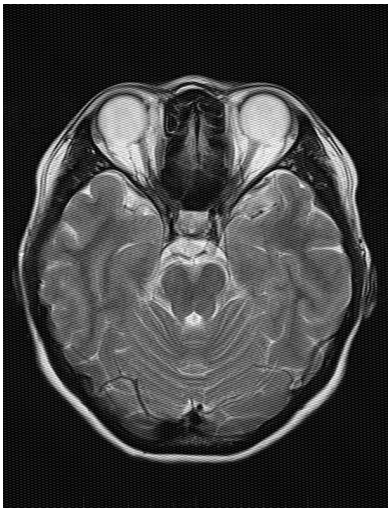
1.3 Additional Result



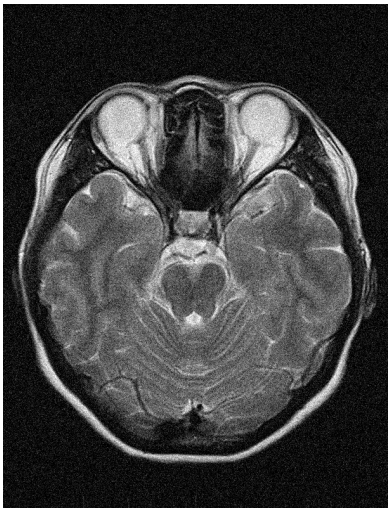
(9) brain.jpg (Original)



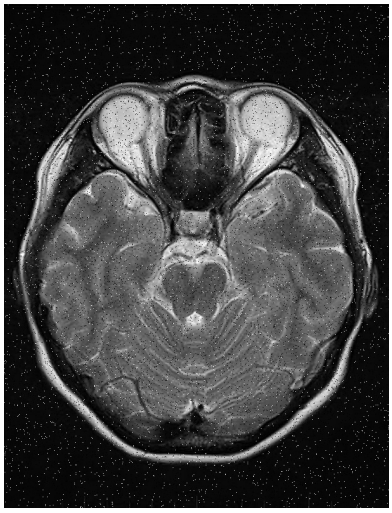
(10) Added white noise ($C = 2.5 \times 10^4$)



(11) Added periodic noise ($C = 2.5 \times 10^4$)

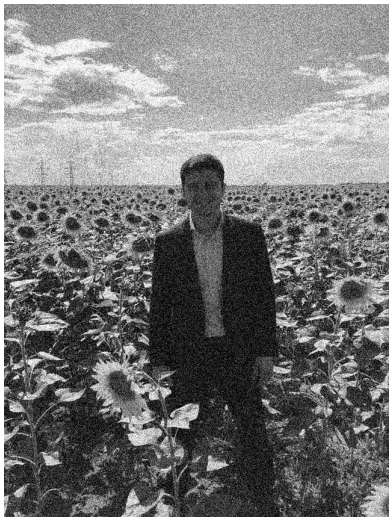


(12) Added Gaussian noise ($\mu = 0$, $\sigma^2 = 625$)



(13) Added pepper-and-salt noise ($p_s = 0.01$, $p_p = 0.03$)

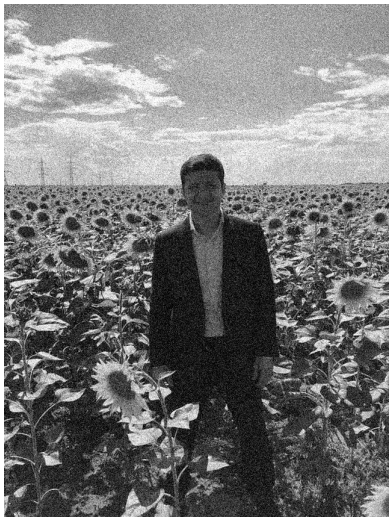
(14) zelenskyy.jpg (Original)



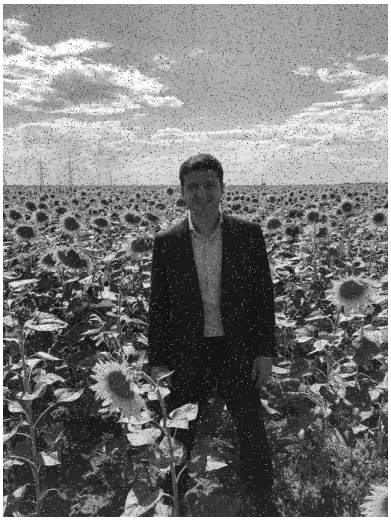
(15) Added white noise ($C = 2.5 \times 10^4$)

(16) Added white noise ($C = 1 \times 10^5$)

(17) Added periodic noise ($C = 2.5 \times 10^4$)



(18) Added Gaussian noise



(19) Added pepper-and-salt noise

Gaussian noise: $\mu = 0$, $\sigma^2 = 625$;
Pepper-and-salt noise: $p_s = 0.01$, $p_p = 0.03$.

2 Optimal Notch Filter

2.1 Theory and Algorithm

Notch filter can be used for noise reduction, especially the periodic noise. A general form of notch reject filter is

$$H_{NR}(u,v) = \sum_{k=1}^{Q} H_k(u,v)\, H_{-k}(u,v),$$

where H_k and H_{-k} are highpass filters centered at (u_k, v_k) and $(-u_k, -v_k)$, $k = 1, \cdots, Q$.
Similarly, the corresponding notch pass filter is $H_{NP} = 1 - H_{NR}$. We can extract noise of certain frequencies from an image by the notch pass filter, which is constrcuted by observing the spectrum of the image. Suppose we construct an notch pass filter on $G(u,v) = \mathfrak{J}\{g(x,y)\}$, then the noise is extracted by

$$\eta(x,y) = \mathfrak{J}^{-1}\{H_{NP}(u,v)\, G(u,v)\}.$$

However, the extracted noise is only an approximation for the true noise pattern. To reduce the effect of estimation error, we retore the original image by subtracting $g(x,y)$ a weighted portion $w(x,y)$ of $\eta(x,y)$:

$$\hat{f}(x,y) = g(x,y) - w(x,y)\eta(x,y),$$

where $w(x,y)$ is optimized conditioning on $g(x,y)$ and $\eta(x,y)$ according to certain criteria.
Here we assume that restored image $\hat{f}(x,y)$ has least variance on neighborhood S_{xy} of each pixel (x,y):

$$\sigma^2_{S_{xy}} = \frac{1}{|S_{xy}|} \sum_{(r,c)\in S_{xy}} \left[\hat{f}(r,c) - \bar{\hat{f}}\right]^2$$

$$= \frac{1}{|S_{xy}|} \sum_{(r,c)\in S_{xy}} [g(r,c) - w(r,c)\eta(r,c) - (\overline{g} - \overline{w\eta})\,]^2$$

$$= \frac{1}{|S_{xy}|} \sum_{(r,c)\in S_{xy}} [g(r,c) - w(r,c)\eta(r,c)]^2 - (\overline{g} - \overline{w\eta})^2$$

Additionally, suppose w is approximately a constant in a small neighborhood region S_{xy}, which is w_{xy}, then w is the solution of the optimization problem

$$\min_{w_{xy}} \frac{1}{|S_{xy}|} \sum_{(r,c)\in S_{xy}} [g(r,c) - w_{xy}\eta(r,c)]^2 - (\overline{g} - w_{xy}\overline{\eta})^2;$$

The convex function is minimized when

$$\frac{\partial \sigma^2_{S_{xy}}}{\partial w_{xy}} = -\frac{2}{|S_{xy}|} \sum_{(r,c)\in S_{xy}} \eta(r,c)[g(r,c) - w_{xy}\eta(r,c)] + 2\bar{\eta}(\bar{g} - w_{xy}\bar{\eta})$$

$$= -2\overline{g\eta} + 2w_{xy}\overline{\eta^2} + 2\bar{g}\bar{\eta} - 2w_{xy}\bar{\eta}^2 = 0;$$

the optimal solution

$$w^*_{xy} = \frac{\overline{g\eta} - \bar{g}\bar{\eta}}{\overline{\eta^2} - \bar{\eta}^2},$$

by which $w(x,y)$ is computed on neighborhood S_{xy}.

Algorithm The pseudocode of optimal notch filter is shown below.

OptimalNotch(g, H)

Input: image g with interference pattern, notch filter H;

Output: restored image $\hat{f}$.

Get g_p by padding g with zeros

$G(u,v) = DFT\{g_p(x,y)(-1)^{x+y}\}$

$\eta_p(x,y) = IDFT\{G(u,v)H(u,v)\}(-1)^{x+y}$

Extract $\eta(x,y)$ from $\eta_p(x,y)$ according to padding position

$w(x,y) = \left(\overline{g\eta} - \bar{g}\bar{\eta}\right) / \left(\overline{\eta^2} - \bar{\eta}^2\right)$

$\hat{f}(x,y) = g(x,y) - w(x,y)\eta(x,y)$

return $\hat{f}$

2.2 Experiment

Code The notch pass filter is a user-defined parameter, and we create it according to our observation of the Fourier spectrum. In implement we use Gaussian highpass filter as the component of notch filter. The code for creating filter is shown below.

```
def point_notch(h, w, u0, v0, sd=5):
    # Remove a small circle at (u0, v0) from the spectrum.
    # It's equivalent to Gaussian highpass filter centered at (u0, v0).
    u_mat = np.array([np.arange(h) - u0] * w)
    v_mat = np.array([np.arange(w) - v0] * h)
    dist_sq = (u_mat ** 2).T + v_mat ** 2
    return 1 - np.exp(-dist_sq / (2 * sd * sd))

def notch_pass(h, w, u0, v0, sd=5):
    # (u0, v0): coordinates of centroids.
    # u0: the list of vertical coordinates.
    # v0: the list of horizontal coordinates.
    # Yield a notch pass filter.
    nr = np.ones((h, w))
    if len(u0) - len(v0):
        raise ValueError
    for i in range(len(u0)):
        # Operate on pairs.
        nr *= point_notch(h, w, u0[i], v0[i], sd)
        nr *= point_notch(h, w, h - 1 - u0[i], w - 1 - v0[i], sd)
    return 1 - nr
```

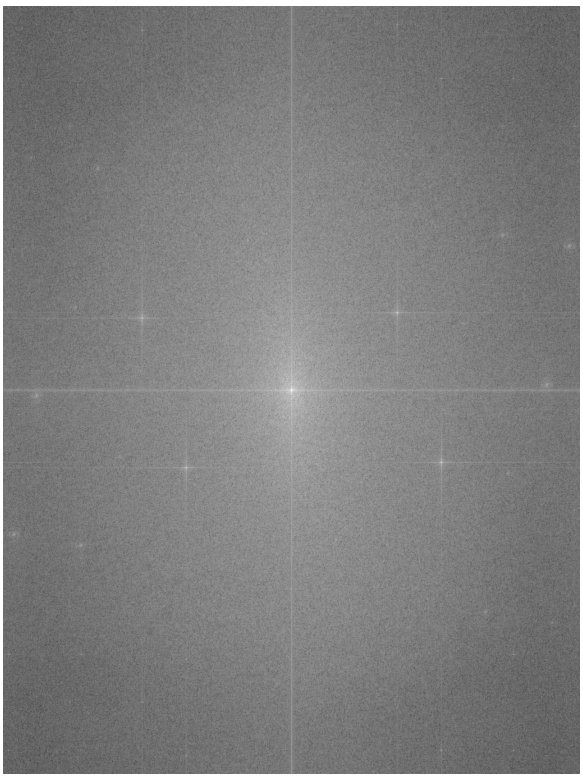
Given the notch filter, the code for extracting noise pattern and restore input image is shown below.

```
def noise_mod(img, notch):
    # Simulate the noise pattern in space domain.
    h, w = img.shape

    # Observe the Fourier spectrum of the image.
    img_pad = np.zeros((2 * h, 2 * w))
    sign = -np.ones((2 * h, 2 * w))
    for i in range(2 * h):
        sign[i, i % 2::2] = 1
    img_pad[:h, :w] = img
    img_ft = np.fft.fft2(img_pad * sign)

    spectrum = np.log(1 + np.abs(img_ft))
    spectrum = spectrum / np.max(spectrum) * 255

    plt.figure()
    plt.imshow(spectrum.astype(int), cmap='gray')
    plt.show()
    cv2.imwrite('images/spectrum.jpg', spectrum)

    # Generate the noise pattern.
    noise_ft = img_ft * notch
    noise_pad = np.fft.ifft2(noise_ft) * sign
    return np.real(noise_pad[:h, :w])

def moving_avg(mat, size=1):
    # Compute the average of a neighborhood while moving on a matrix.
    avg = np.zeros_like(mat)
    for i in range(mat.shape[0]):
        for j in range(mat.shape[1]):
            a, b = max(i - size, 0), min(i + size + 1, mat.shape[0])
            l, r = max(j - size, 0), min(j + size + 1, mat.shape[1])
            avg[i, j] = np.mean(mat[a:b, l:r])
    return avg


def opt_filter(img, noise, size=1, nbins=256):
    # Compute the weight matrix.
    avg_noise = moving_avg(noise, size)
    sxy = moving_avg(img * noise, size) - moving_avg(img, size) * avg_noise
```

```
    sxx = moving_avg(noise * noise, size) - avg_noise ** 2
    sxx[sxx == 0] = 1e-5  # Avoid dividing by 0.

    # Denoise.
    img_denoise = img - noise * sxy / sxx
    # Cut-off.
    img_denoise[img_denoise >= nbins] = nbins - 1
    img_denoise[img_denoise < 0] = 0
    return img_denoise.astype(int)
```

Result We tesedt our code on image `zelenskyy.jpg` with interference pattern. At first we visualized the spectrum of the image (has been centralized and log-scaled), and we observed there are four light spots near the center of the spectrum, which may correspond to the periodical noise pattern. The result is shown below.

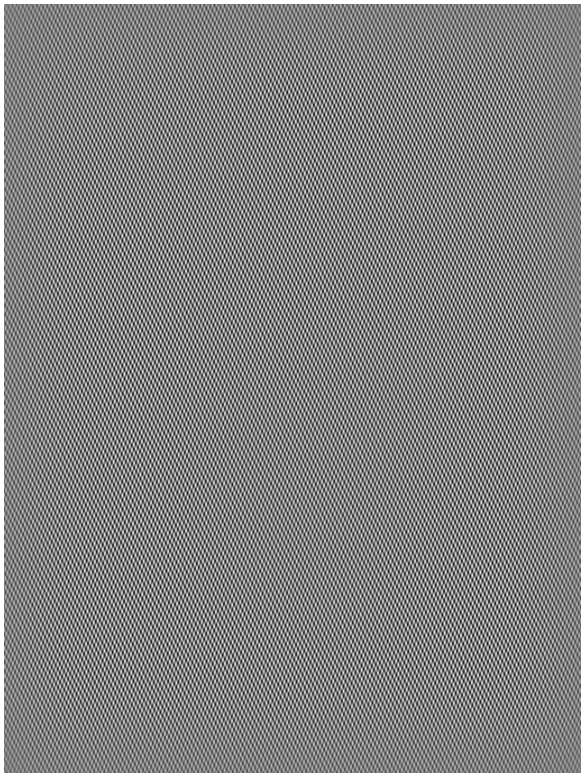
(20) zelenskyy.jpg with noise pattern



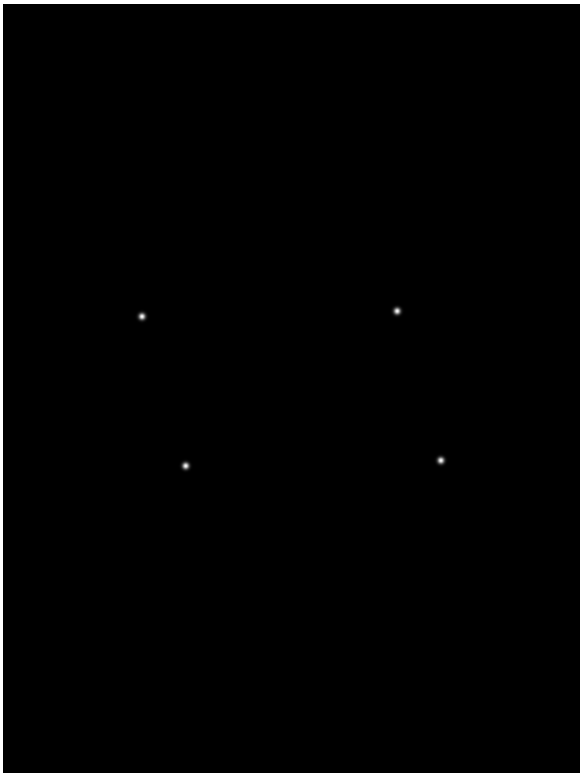
(21) Fourier spectrum (log-scaled)

Then we created a notch pass filter with the notches centered at the positions of four light spots in the spectrum, and extract the noise pattern from the input image with the filter. We empirically set the standard deviation of Gaussian highpass filter as 5.

With the extracted noise pattern, we restored the image by the optimal notch filtering method. To emphasize the effect of weighted proportion $w(x, y)$, we did an ablation study by directly subtracting the noise pattern from the input image. The result is shown below.



(22) Extracted noise pattern



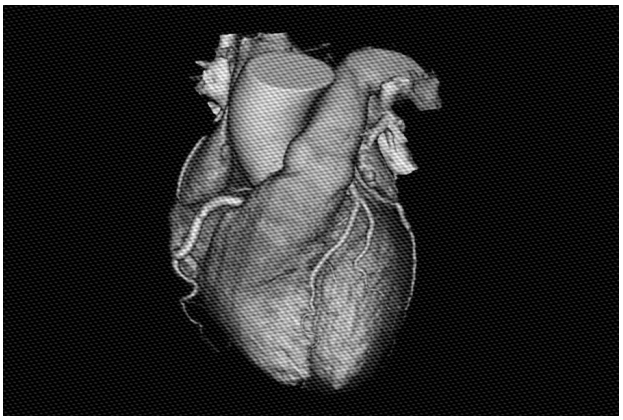
(23) Notch pass filter

(24) Directly subtract the noise

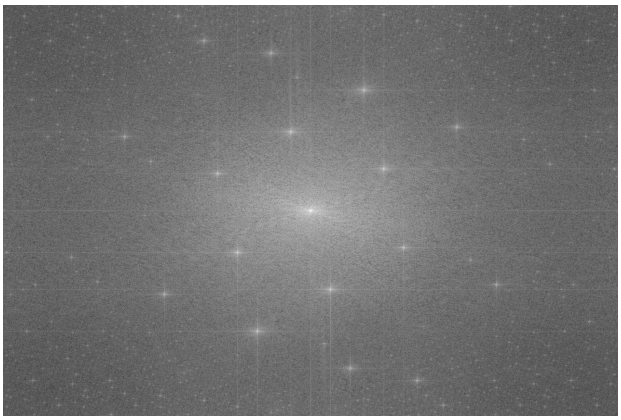
(25) Apply optimal notch filter

The result shows that the optimal notch filter achieves a better effect in periodic noise reduction. As we can observe in figure (24), there are traces of noise remaining in the filtered image, especially near the border. However in figure (25), we can observe little noise. That indicates the effectiveness of weight matrix w, which is computed according to the least square criteria.

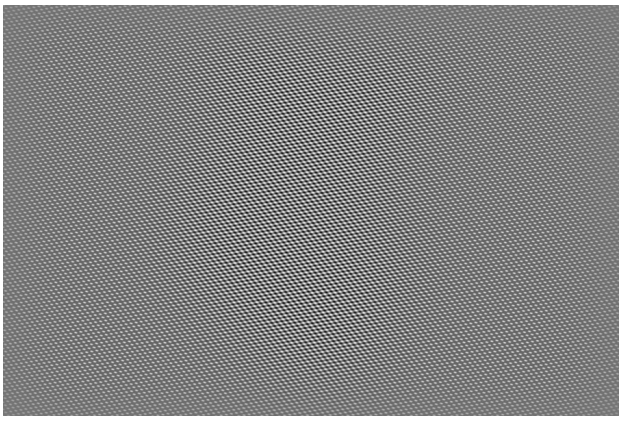
Following is some **additional result**.



(26) heart.jpg with noise pattern

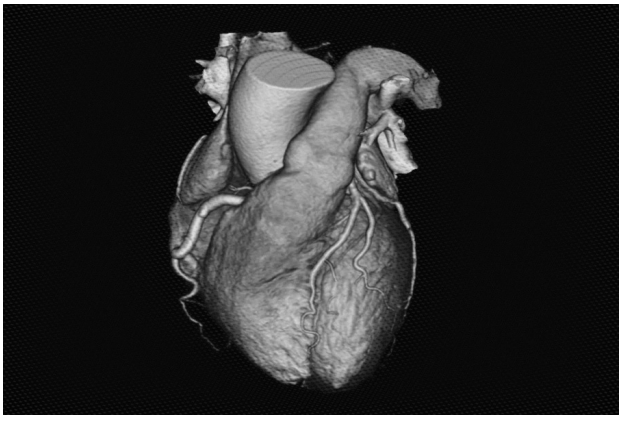


(27) Fourier spectrum (log-scaled)

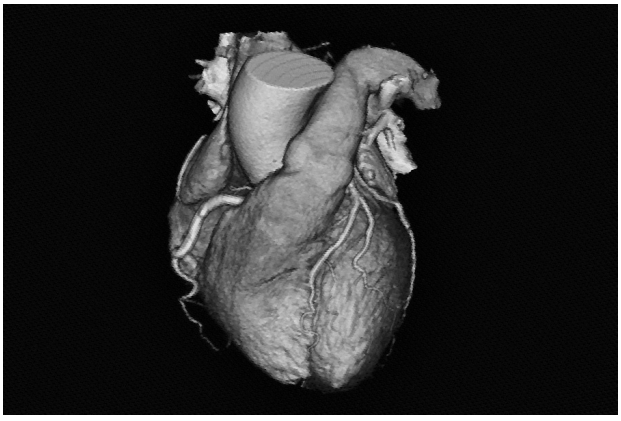


(28) Extracted noise pattern

(29) Notch pass filter



(30) Directly subtract the noise



(31) Using optimal notch filter

3 Appendix

3.1 Libraries

We use the `numpy` library to support the large-scale computation of matrices which store the grey level information of images.
We use `opencv` library (`cv2`) to read the input images and save the output images.
We use `matplotlib` to show images before and after processing.
We also use `argparse` to read in the arguments for processing, which can be specified in the command line.

3.2 IDE

All our experiments are done on PyCharm Community Edition 2022.1.2.