# 图像处理与可视化 Homework 05

学号: 21307140051
姓名: 雍崔扬

## Problem 1

生成以下两种不同类型和不同强度的噪声，并使用生成的噪声污染图像，对比展示噪声污染前后的图像:

- ① 生成白噪声
- ② 生成其他噪声 (如 Gauss、Rayleigh 和椒盐噪声)

### Part (1)

生成白噪声.

**Solution:**
当噪声的 Fourier 频谱 $|\tilde{\eta}(\mu,\nu)|$ 是常数函数时，我们称 $\eta(x,y)$ 为**白噪声** (white noise)

我们设置 $|\tilde{\eta}(\mu,\nu)| \equiv 1$
并通过 $[0,2\pi]$ 的均匀分布生成 $\tilde{\eta}(\mu,\nu)$ 的相位谱 $\phi(\mu,\nu) := \arctan\left(\frac{\text{Im}(\tilde{\eta}(\mu,\nu))}{\text{Re}(\tilde{\eta}(\mu,\nu))}\right)$
然后得到白噪声 $\eta(x,y) = \mathscr{F}^{-1}\{\tilde{\eta}(\mu,\nu)\}$
舍入误差可能导致出现寄生复数项，故我们通常还需取实部来形成 $\eta(x,y)$

---

生成并应用白噪声的函数 `generate_white_noise()`:

```python
def generate_white_noise(image, coefficient=0.1):
    """
    Generate white noise with the same size as the input image.

    :param image: The input image (numpy array) to match the size.
    :param coefficient: A coefficient to control the amount of noise to be added
to the image.
    :return: The spatial domain representation of the generated noise.
    """
    # Get the image size (height and width)
    height, width = image.shape

    # Generate random phase values uniformly distributed between [0, 2pi]
    random_phase = np.random.uniform(0, 2 * np.pi, (height, width))

    # The magnitude of the Fourier transform is set to 1 (constant), and the
phase is random
    magnitude = np.ones((height, width))
    complex_spectrum = magnitude * np.exp(1j * random_phase)

    # Perform inverse Fourier transform to get the noise in the spatial domain
    noise = np.fft.ifft2(complex_spectrum).real

    # 1. Display the original image
    plt.figure(figsize=(12, 8))
    plt.subplot(3, 3, 1)
    plt.imshow(image, cmap='gray')
    plt.title('Original Image')
    plt.axis('off')  # Hide axes for better visualization

    # 2. Display the Fourier transform magnitude spectrum of the original image
    f_transform = np.fft.fftshift(np.fft.fft2(image))  # Apply FFT and shift
zero frequency to center
    magnitude_spectrum = np.abs(f_transform)  # Compute the magnitude spectrum
    plt.subplot(3, 3, 2)
    plt.imshow(np.log(magnitude_spectrum + 1), cmap='gray')  # Apply log scale
for better visibility
    plt.title('Fourier Spectrum of Image')
    plt.axis('off')  # Hide axes

    # 3. Plot the grayscale histogram of the original image
    hist_image, bin_edges = compute_histogram(image)  # Compute histogram
    plt.subplot(3, 3, 3)
    plt.plot(bin_edges[:-1], hist_image)  # Plot histogram
    plt.title('Original Image Histogram')

    # 4. Display the phase spectrum of the noise
    phase_spectrum = np.angle(complex_spectrum)  # Compute the phase of the
noise
    plt.subplot(3, 3, 4)
    plt.imshow(phase_spectrum, cmap='gray')  # Display the phase spectrum
    plt.title('Phase Spectrum of Noise')
    plt.axis('off')

    # 5. Show the frequency spectrum of the noise
    plt.subplot(3, 3, 5)
    plt.imshow(np.log(1 + np.abs(complex_spectrum)), cmap='gray')  # Display the
phase spectrum
    plt.title('Power Spectrum of Noise')
    plt.axis('off')

    # 6. Plot the grayscale histogram of the noise after normalization
    noise_normalized = 255 * (noise - np.min(noise)) / (np.max(noise) -
np.min(noise))  # Normalize to [0, 255]
    hist_noise, bin_edges = compute_histogram(noise_normalized.astype(np.uint8))
 # Compute histogram of normalized noise
    plt.subplot(3, 3, 6)
    plt.plot(bin_edges[:-1], hist_noise)  # Plot histogram
    plt.title('Noise Histogram')

    # 7. Display the noise in the spatial domain (normalized to [0, 255] range)
    plt.subplot(3, 3, 7)
    plt.imshow(noise_normalized.astype(np.uint8), cmap='gray')  # Display noise
in spatial domain
    plt.title('Noise in Spatial Domain')
    plt.axis('off')

    # 8. Add the noise to the original image and display the result
    noisy_image = np.clip(image + noise_normalized * coefficient, 0,
255).astype(np.uint8)  # Clip values to [0, 255] range
    plt.subplot(3, 3, 8)
    plt.imshow(noisy_image, cmap='gray')  # Display noisy image
    plt.title(f'Noisy Image (Coeff={coefficient})')
    plt.axis('off')
    # 9. Plot the grayscale histogram of the noisy image
    hist_noisy_image, bin_edges = compute_histogram(noisy_image)  # Compute
histogram of noisy image
    plt.subplot(3, 3, 9)
    plt.plot(bin_edges[:-1], hist_noisy_image)  # Plot histogram
    plt.title('Histogram of Noisy Image')
    plt.tight_layout()  # Adjust subplots for better spacing

    # Save the figure containing all the images
    save_path = f"white_noise.png"
    plt.savefig(save_path)
    plt.show()

    # Save the final filtered image
    final_image_path = f"image_with_white_noise.tif"
    Image.fromarray(noisy_image).save(final_image_path, format="TIFF")

    return noise
```

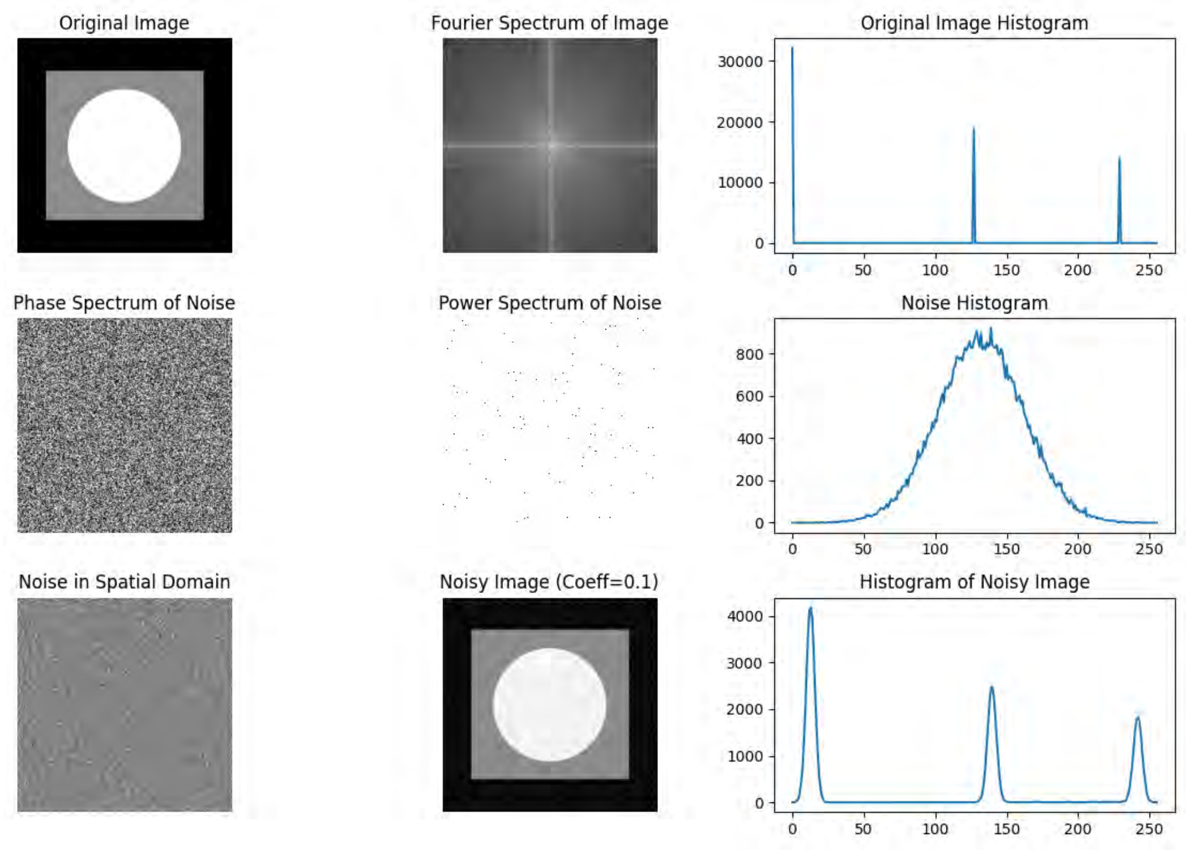
计算灰度直方图的函数 `compute_histogram`:

```
def compute_histogram(image, num_bins=256):
    """
    Compute the grayscale histogram of an image.

    :param image: Grayscale image as a numpy array.
    :param num_bins: Number of bins for the histogram (default is 256).
    :return: The histogram and bin edges.
    """
    # Flatten the image and compute the histogram with the specified number of
bins
    histogram, bin_edges = np.histogram(image.ravel(), bins=num_bins, range=[0,
num_bins])
    return histogram, bin_edges
```

函数调用:

```
if __name__ == "__main__":
    # Set seed
    np.random.seed(51)

    # Load the image
    image_path = 'DIP Fig 05.03 (original_pattern).tif'  # Path to the image
file
    image_name = 'DIP Fig 05.03 (original_pattern)'  # Image name (for
reference)

    # Open the image, convert it to grayscale and then convert to a numpy array
    img = Image.open(image_path).convert('L')  # Convert the image to grayscale
('L' mode)
    image_array = np.array(img)  # Convert the grayscale image to a numpy array

    # Generate white noise with the same size as the image
    noise = generate_white_noise(image_array, coefficient=0.1)
```

**运行结果:**



## Part (2)

生成其他噪声 (如 Gauss、Rayleigh 和椒盐噪声)

**Solution:**
我们用 $z$ 表示灰度.

- **① Gauss 噪声:**

$$p(z) := \frac{1}{\sqrt{2\pi}\sigma}\exp\{-\frac{1}{2\sigma^2}(z-\mu)^2\}$$

  灰度值 $z$ 落在区间 $\bar{z}\pm\sigma$ 内的概率约为 $0.68$，灰度值 $z$ 落在区间 $\bar{z}\pm 2\sigma$ 内的概率约为 $0.95$

- **② Rayleigh 噪声:**

$$p(z) := \begin{cases} \frac{2}{b}(z-a)\exp\{-\frac{1}{b}(z-a)^2\} & \text{if } z \geq a \\ 0 & \text{otherwise} \end{cases}$$

$$\mu = a + \frac{1}{2}\sqrt{\pi b}$$

$$\sigma^2 = \frac{1}{4}b(4-\pi)$$

  Rayleigh 分布对倾斜形状直方图的建模非常有用.

- **③ Gamma 噪声:**

$$p(z) := \begin{cases} \frac{\lambda^\alpha z^{\alpha-1}}{\Gamma(\alpha)}\exp\left(-\lambda z\right) & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\mu = \frac{\alpha}{\lambda}$$

$$\sigma^2 = \frac{\alpha}{\lambda^2}$$

取 $\alpha = 1$，便得到指数噪声:

$$p(z) := \begin{cases} \lambda \exp(-\lambda z) & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\mu = \frac{1}{\lambda}$$

$$\sigma^2 = \frac{1}{\lambda^2}$$

- **④ 均匀噪声:**

$$p(z) := \begin{cases} \frac{1}{b-a} & \text{if } a \leq z \leq b \\ 0 & \text{otherwise} \end{cases}$$

$$\mu = \frac{1}{2}(a+b)$$

$$\sigma^2 = \frac{1}{12}(b-a)^2$$

- **⑤ 椒盐噪声:**

$$p(z) := \begin{cases} p_{\text{salt}} & \text{if } z = 255 \\ p_{\text{pepper}} & \text{if } z = 0 \\ 1 - p_{\text{salt}} - p_{\text{pepper}} & \text{otherwise} \end{cases}$$

我们称 $p_{\text{salt}} + p_{\text{pepper}}$ 为噪声密度.
若仅有盐粒 (灰度为 255 的白点)，则称为**盐粒噪声** (单纯盐粒噪声并不常见)
若仅有胡椒 (灰度为 0 的黑点)，则称为**胡椒噪声** (与盐粒噪声相对)

---

生成空间域噪声的函数 `generate_noise()`:

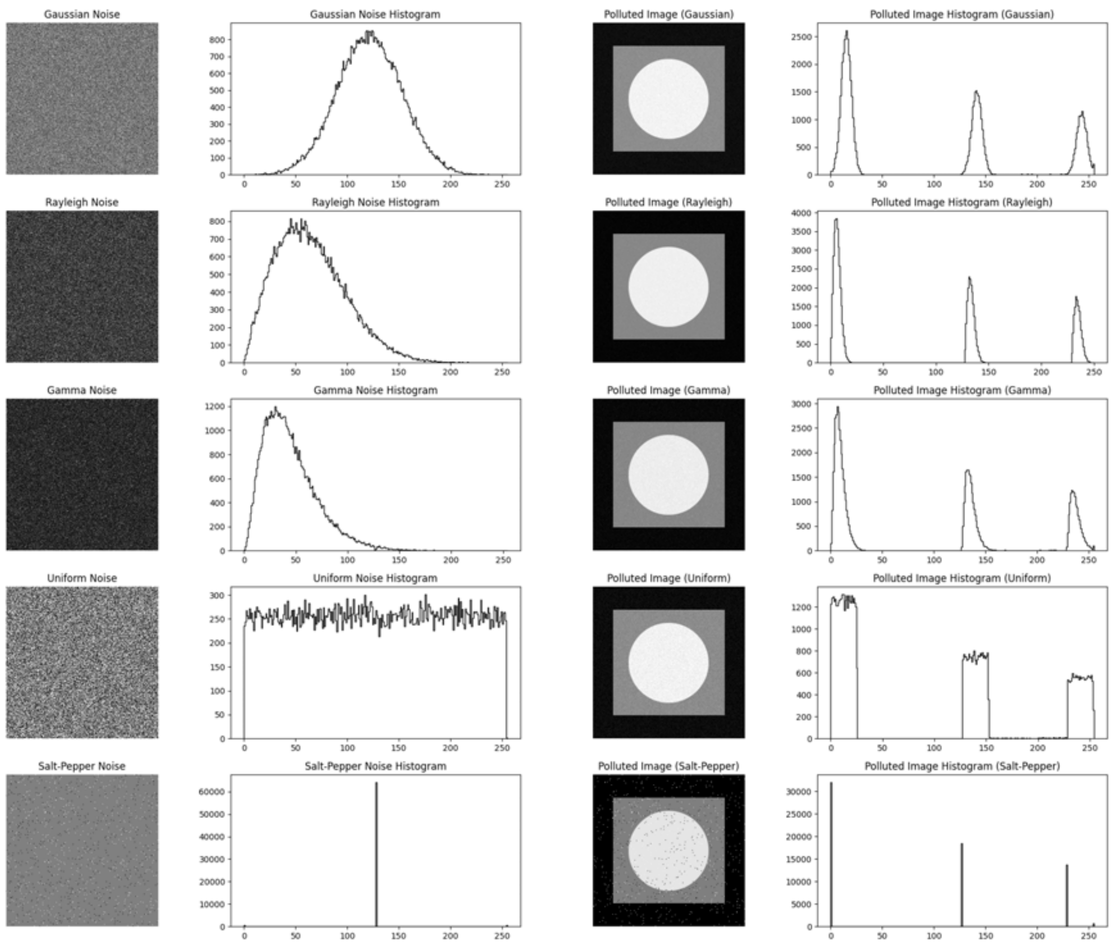
```python
def generate_noise(image, noise_type="gaussian", coefficient=0.1, **params):
    """
    Generate different types of noise and add it to the input image.

    :param image: The input image (numpy array) to match the size.
    :param noise_type: Type of noise to generate ("gaussian", "rayleigh",
"gamma", "uniform", "salt_pepper").
    :param coefficient: A coefficient to control the amount of noise to be added
to the image.
    :param params: Additional parameters specific to the noise type.
    :return: The spatial domain representation of the generated noise.
    """
    height, width = image.shape

    if noise_type == "gaussian":
        # Gaussian noise: mean = mu, std deviation = sigma
        mu = params.get("mu", 0)
        sigma = params.get("sigma", 25)
        noise = np.random.normal(mu, sigma, (height, width))

    elif noise_type == "rayleigh":
        # Rayleigh noise: scale = b, offset = a
        a = params.get("a", 0)
        b = params.get("b", 25)
        noise = np.random.rayleigh(b, (height, width)) + a
    elif noise_type == "gamma":
        # Gamma noise: shape = alpha, rate = lambda
        alpha = params.get("alpha", 2)
        lambd = params.get("lambda", 1)
        noise = np.random.gamma(alpha, 1/lambd, (height, width))

    elif noise_type == "uniform":
        # Uniform noise: min = a, max = b
        a = params.get("a", 0)
        b = params.get("b", 255)
        noise = np.random.uniform(a, b, (height, width))

    elif noise_type == "salt_pepper":
        # Salt-and-pepper noise: p_salt, p_pepper (density)
        p_salt = params.get("p_salt", 0.01)
        p_pepper = params.get("p_pepper", 0.01)

        # Generate a random mask for salt and pepper noise
        mask = np.random.random((height, width))
        image[mask < p_salt] = 255          # salt (white pixels)
        image[mask > (1 - p_pepper)] = 0    # pepper (black pixels)

        noise = 128 * np.ones_like(image).astype(np.uint8)
        noise[mask < p_salt] = 255
        noise[mask > (1 - p_pepper)] = 0

        return noise, image

    else:
        raise ValueError("Unsupported noise type. Choose from 'gaussian',
'rayleigh', 'gamma', 'uniform', 'salt_pepper'.")

    # Use noise to he noisy image
    noise_normalized = (255 * (noise - np.min(noise)) / (np.max(noise) -
np.min(noise))).astype(np.uint8)
    image = np.clip(image + noise * coefficient, 0, 255).astype(np.uint8)

    return noise_normalized, image
```

函数调用:

```
if __name__ == "__main__":
    # Set seed
    np.random.seed(51)

    # Load the image
    image_path = 'DIP Fig 05.03 (original_pattern).tif'  # Path to the image
file
    image_name = 'DIP Fig 05.03 (original_pattern)'  # Image name (for
reference)

    # Open the image, convert it to grayscale and then convert to a numpy array
    image = Image.open(image_path).convert('L')  # Convert the image to
grayscale ('L' mode)
    image = np.array(image)  # Convert the grayscale image to a numpy array

    # 1. Gaussian Noise
```

```python
    noise, noisy_image = generate_noise(image, noise_type="gaussian",
coefficient=5, mu=3, sigma=1)
    plt.figure(figsize=(20, 16))
    plt.subplot(5, 4, 1)
    plt.imshow(noise, cmap='gray')
    plt.title('Gaussian Noise')
    plt.axis('off')

    plt.subplot(5, 4, 2)
    plt.hist(noise.ravel(), bins=256, color='black', histtype='step')
    plt.title('Gaussian Noise Histogram')

    plt.subplot(5, 4, 3)
    plt.imshow(noisy_image, cmap='gray')
    plt.title('Polluted Image (Gaussian)')
    plt.axis('off')

    plt.subplot(5, 4, 4)
    plt.hist(noisy_image.ravel(), bins=256, color='black', histtype='step')
    plt.title('Polluted Image Histogram (Gaussian)')

    # 2. Rayleigh Noise
    noise, noisy_image = generate_noise(image, noise_type="rayleigh",
coefficient=0.2, a=0, b=25)
    plt.subplot(5, 4, 5)
    plt.imshow(noise, cmap='gray')
    plt.title('Rayleigh Noise')
    plt.axis('off')

    plt.subplot(5, 4, 6)
    plt.hist(noise.ravel(), bins=256, color='black', histtype='step')
    plt.title('Rayleigh Noise Histogram')

    plt.subplot(5, 4, 7)
    plt.imshow(noisy_image, cmap='gray')
    plt.title('Polluted Image (Rayleigh)')
    plt.axis('off')

    plt.subplot(5, 4, 8)
    plt.hist(noisy_image.ravel(), bins=256, color='black', histtype='step')
    plt.title('Polluted Image Histogram (Rayleigh)')

    # 3. Gamma Noise
    noise, noisy_image = generate_noise(image, noise_type="gamma",
coefficient=3, alpha=3, lambd=1)
    plt.subplot(5, 4, 9)
    plt.imshow(noise, cmap='gray')
    plt.title('Gamma Noise')
    plt.axis('off')

    plt.subplot(5, 4, 10)
    plt.hist(noise.ravel(), bins=256, color='black', histtype='step')
    plt.title('Gamma Noise Histogram')

    plt.subplot(5, 4, 11)
    plt.imshow(noisy_image, cmap='gray')
    plt.title('Polluted Image (Gamma)')
    plt.axis('off')
    plt.subplot(5, 4, 12)
    plt.hist(noisy_image.ravel(), bins=256, color='black', histtype='step')
    plt.title('Polluted Image Histogram (Gamma)')

    # 4. Uniform Noise
    noise, noisy_image = generate_noise(image, noise_type="uniform",
coefficient=0.1, a=0, b=255)
    plt.subplot(5, 4, 13)
    plt.imshow(noise, cmap='gray')
    plt.title('Uniform Noise')
    plt.axis('off')

    plt.subplot(5, 4, 14)
    plt.hist(noise.ravel(), bins=256, color='black', histtype='step')
    plt.title('Uniform Noise Histogram')

    plt.subplot(5, 4, 15)
    plt.imshow(noisy_image, cmap='gray')
    plt.title('Polluted Image (Uniform)')
    plt.axis('off')

    plt.subplot(5, 4, 16)
    plt.hist(noisy_image.ravel(), bins=256, color='black', histtype='step')
    plt.title('Polluted Image Histogram (Uniform)')

    # 5. Salt-Pepper Noise
    noise, noisy_image = generate_noise(image, noise_type="salt_pepper",
p_salt=0.01, p_pepper=0.01)
    plt.subplot(5, 4, 17)
    plt.imshow(noise, cmap='gray')
    plt.title('Salt-Pepper Noise')
    plt.axis('off')

    plt.subplot(5, 4, 18)
    plt.hist(noise.ravel(), bins=256, color='black', histtype='step')
    plt.title('Salt-Pepper Noise Histogram')

    plt.subplot(5, 4, 19)
    plt.imshow(noisy_image, cmap='gray')
    plt.title('Polluted Image (Salt-Pepper)')
    plt.axis('off')

    plt.subplot(5, 4, 20)
    plt.hist(noisy_image.ravel(), bins=256, color='black', histtype='step')
    plt.title('Polluted Image Histogram (Salt-Pepper)')
    plt.tight_layout()

    # Save the figure containing all the images
    save_path = f"spatial_noise.png"
    plt.savefig(save_path)
    plt.show()
```

**运行结果:**



## Problem 2

编程实现最佳陷波滤波器.

**Solution:**
最佳陷波滤波的流程总结如下:

- **① 首先提取干扰模式的主频率分量.**
  提取方法照例是在每个尖峰位置放一个陷波带通滤波器传递函数 $\tilde{h}_{\text{NP}}(\mu,\nu)$
  若将该滤波器构建为只通过与干扰模式相关联的分量，则加性噪声项的二维离散 Fourier 变换为:

$$\tilde{\eta}(\mu,\nu)=\tilde{h}_{\text{NP}}(\mu,\nu)\tilde{g}(\mu,\nu)$$

  其中 $\tilde{g}(\mu,\nu)$ 是退化图像 $g(x,y)$ 的二维离散 Fourier 变换.
  而陷波带通滤波器传递函数 $\tilde{h}_{\text{NP}}(\mu,\nu)$ 通常是通过观察频谱 $\tilde{g}(\mu,\nu)$ 来交互地构建的.

- **② 计算加性噪声项的空间表示.**

$$\eta(x,y)=\text{Re}\{\mathscr{F}^{-1}(\tilde{h}_{\text{NP}}(\mu,\nu)\tilde{g}(\mu,\nu))\}$$

- **③ 通过减去 $\eta(x,y)$ 的某个加权项来得到 $f(x,y)$ 的估计.**

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

  其中加权函数 $w(x,y)$ 待定.
  我们通常选取 $w(x,y)$ 使 $\hat{f}(x,y)$ 在每点 $(x,y)$ 的规定邻域上方差最小.
  考虑一个中心为 $(x,y)$，尺寸为 $m\times n$ (均为奇数) 的邻域 $N_{xy}$，假设 $w(x,y)$ 在 $N_{xy}$ 中为常数.
  我们定义:

$$\bar{g} := \frac{1}{mn} \sum_{(x',y')\in N_{xy}} g(x',y')$$

$$\bar{\eta} := \frac{1}{mn} \sum_{(x',y')\in N_{xy}} \eta(x',y')$$

$$\sigma_{xy}^2 := \frac{1}{mn} \sum_{(x',y')\in N_{xy}} \left\{[g(x',y') - w(x,y)\eta(x',y')] - (\bar{g} - w(x,y)\bar{\eta})\right\}^2$$

以 $w(x,y)$ 为优化变量最小化 $\sigma_{xy}^2$ 即得:

$$w(x,y) := \frac{\overline{g\eta} - \bar{g}\bar{\eta}}{\overline{\eta^2} - \bar{\eta}^2}$$

$$\text{where} \begin{cases} \bar{g} := \frac{1}{mn}\sum_{(x',y')\in N_{xy}} g(x',y') \\ \bar{\eta} := \frac{1}{mn}\sum_{(x',y')\in N_{xy}} \eta(x',y') \\ \overline{g\eta} := \frac{1}{mn}\sum_{(x',y')\in N_{xy}} g(x',y')\eta(x',y') \\ \overline{\eta^2} := \frac{1}{mn}\sum_{(x',y')\in N_{xy}} \eta^2(x',y') \end{cases}$$

---

根据噪声图像 $\eta(x,y)$ 和退化图像 $g(x,y)$ 计算权重 $w(x,y)$ 的函数 `compute_weight_function()`:

```
def compute_weight_function(g, eta, window_size=(5, 5)):
    """
    Compute the weight function w(x, y) that minimizes the variance of the
filtered image.

    The weight function is calculated by analyzing the correlation between the
noisy image (g) and the noise pattern (eta)
    within a local neighborhood of each pixel.

    :param g: The degraded image g(x, y), which contains both the original image
and noise.
    :param eta: The additive noise pattern eta(x, y), which has been obtained
after filtering.
    :param window_size: The height and width of the local neighborhood (should be
two odd integers).
    :return: The weight function w(x, y), which will be used to filter the image
and minimize the noise.
    """
    # Create an empty array for the weight function w
    w = np.ones_like(g).astype(np.float64)
    M, N = g.shape
    radius_x = window_size[0] // 2
    radius_y = window_size[1] // 2

    # Iterate over each pixel in the image
    for x in range(radius_x, M-radius_x):
        for y in range(radius_y, N-radius_y):

            g_window = g[x-radius_x:x+radius_x+1, y-radius_y:y+radius_y+1]
            eta_window = eta[x-radius_x:x+radius_x+1, y-radius_y:y+radius_y+1]

            # Calculate the means
            bar_g = np.mean(g_window)
            bar_eta = np.mean(eta_window)
            bar_eta2 = np.mean(eta_window * eta_window)
            bar_g_eta = np.mean(g_window * eta_window)
            w[x, y] = (bar_g_eta - bar_g * bar_eta) / (bar_eta2 - bar_eta *
bar_eta + 1e-3)

    return w
```

实现最佳陷波滤波的函数 `frequency_domain_notch`:

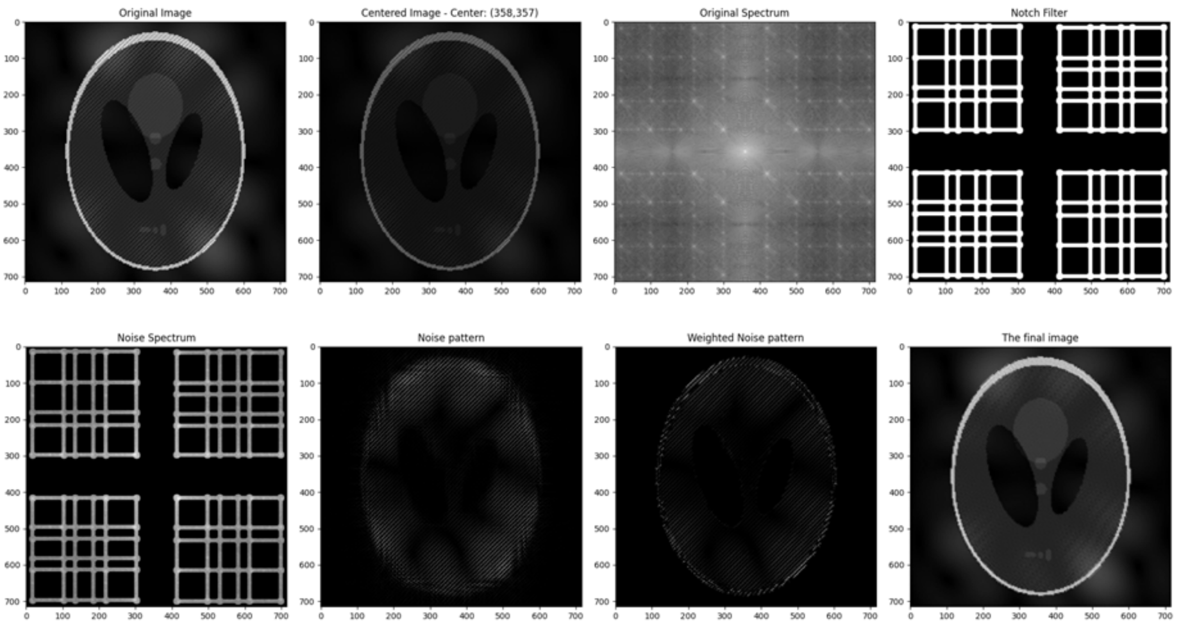
```
def frequency_domain_notch(image, centers, cutoffs, image_name, filter_types=
[0], window_size=(5,5)):
    # Assert that filter_type is a list of integers, each greater than -2
    assert all(isinstance(filter_type, int) and filter_type >= -2 for
filter_type in filter_types), \
        "All elements of filter_type must be integers greater than or equal to
-2"

    # Plotting setup
    fig, axs = plt.subplots(2, 4, figsize=(20, 16))
    axs[0, 0].imshow(image, cmap="gray")
    axs[0, 0].set_title("Original Image")

    # Step 1: Multiply padded image by (-1)^(x+y) to center the Fourier
transform
    M, N = image.shape
    centered_image = image * np.fromfunction(lambda x, y: (-1)**(x + y), (M, N))

    # Display centered image
    axs[0, 1].imshow(np.clip(centered_image, 0, 255).astype(np.uint8),
cmap="gray")
    axs[0, 1].set_title(f"Centered Image - Center: ({N // 2},{M // 2})")

    # Step 2: Compute the 2D Fourier Transform of the centered image
    G_uv = np.fft.fft2(centered_image)

    # Display original spectrum
    axs[0, 2].imshow(np.log(1 + np.abs(G_uv)), cmap="gray")
    axs[0, 2].set_title("Original Spectrum")

    # Step 3: Create the notch filter with multiple notch centers
    u, v = np.meshgrid(np.arange(0, M), np.arange(0, N), indexing='ij')

    # Initialize the combined notch filter with ones (multiplicative identity for
filters)
    H_uv = np.ones((M, N))

    # Loop over each "filter_type" to construct the filter
    for i, filter_type in enumerate(filter_types):
        for center, cutoff in zip(centers[i], cutoffs[i]):
            if filter_type == -2:  # Line filter (horizontal or vertical)
                if center[3] == 0:  # Horizontal line
                    y_val = center[0]  # Fixed y-value for the horizontal line
                    x_start, x_end = np.clip(center[1], 0, N),
np.clip(center[2], 0, N-1)
                    mask_x_range = np.arange(x_start, x_end + 1)
                    mask_y_range = np.arange(np.clip(y_val - cutoff, 0, M),
np.clip(y_val + cutoff, 0, M-1) + 1)
```

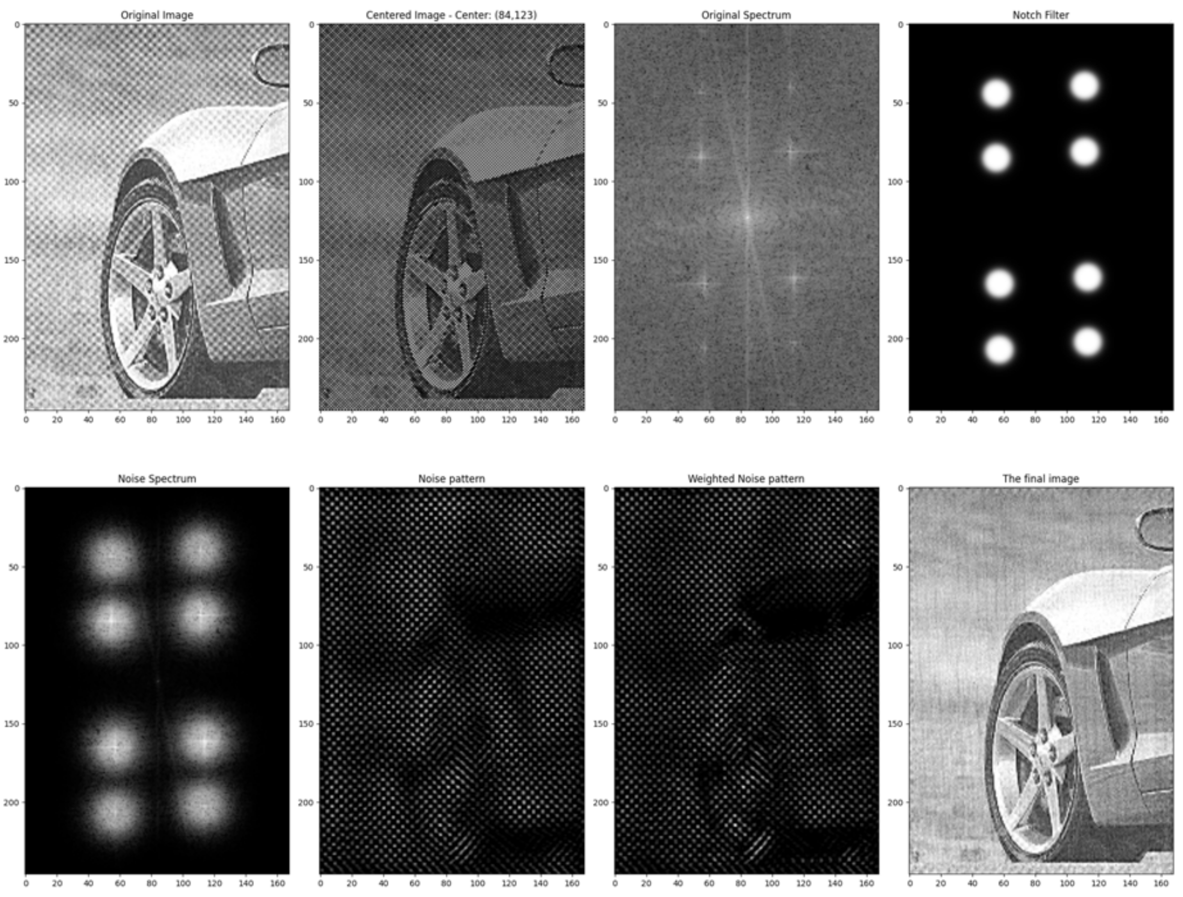
```
                    mask = list(itertools.product(mask_y_range, mask_x_range))
                    mask += [(M - i - 1, N - j - 1) for i, j in mask]

                    # Convert to numpy arrays for vectorized assignment
                    mask_y, mask_x = zip(*mask)
                    mask_y, mask_x = np.array(mask_y), np.array(mask_x)
                    H_uv[mask_y, mask_x] = 0

                else:  # Vertical line
                    x_val = center[0]  # Fixed x-value for the vertical line
                    y_start, y_end = np.clip(center[1], 0, M), 
np.clip(center[2], 0, M-1)
                    mask_y_range = np.arange(y_start, y_end + 1)
                    mask_x_range = np.arange(np.clip(x_val - cutoff, 0, N), 
np.clip(x_val + cutoff, 0, N-1) + 1)
                    mask = list(itertools.product(mask_y_range, mask_x_range))
                    mask += [(M - i - 1, N - j - 1) for i, j in mask]

                    # Convert to numpy arrays for vectorized assignment
                    mask_y, mask_x = zip(*mask)
                    mask_y, mask_x = np.array(mask_y), np.array(mask_x)
                    H_uv[mask_y, mask_x] = 0

            else:
                # Calculate the distances from notch centers
                D1 = np.sqrt((u - center[1])**2 + (v - center[0])**2)
                D2 = np.sqrt((u - (M - center[1]))**2 + (v - (N - 
center[0]))**2)

                if filter_type == -1:  # Ideal filter
                    H_uv[D1 <= cutoff] = 0
                    H_uv[D2 <= cutoff] = 0

                elif filter_type == 0:  # Gaussian filter
                    H_k = (1 - np.exp(-D1**2 / (2 * cutoff**2))) * (1 - np.exp(-
D2**2 / (2 * cutoff**2)))
                    H_uv *= H_k

                else:  # Butterworth filter (ft now acts as the parameter "n")
                    n = filter_type
                    H_k = (1 - 1 / (1 + (D1 / cutoff)**(2 * n))) * (1 - 1 / (1 + 
(D2 / cutoff)**(2 * n)))
                    H_uv *= H_k

    # Norch Band Resist filter to Norch Band Pass filter
    H_uv = np.ones_like(H_uv) - H_uv

    # Display combined notch filter
    axs[0, 3].imshow(H_uv, cmap="gray")
    axs[0, 3].set_title("Notch Filter")

    # Step 4: Apply the filter in the frequency domain to extract noise spectrum
    noise_uv = H_uv * G_uv

    # Display filtered spectrum
    axs[1, 0].imshow(np.log(1 + np.abs(noise_uv)), cmap="gray")
    axs[1, 0].set_title("Noise Spectrum")
    # Step 5: Compute the inverse Fourier Transform and remove centering shift
    noise = np.fft.ifft2(noise_uv)
    noise = np.real(noise) * np.fromfunction(lambda x, y: (-1)**(x + y), (M, N))

    # Display noise pattern
    axs[1, 1].imshow(np.clip(noise, 0, 255).astype(np.uint8), cmap="gray")
    axs[1, 1].set_title("Noise pattern")

    # Step 6: Compute the weight function
    w = compute_weight_function(image, noise, window_size=window_size)
    weighted_noise = w * noise

    # Display weighted noise pattern
    axs[1, 2].imshow(np.clip(weighted_noise, 0, 255).astype(np.uint8),
cmap="gray")
    axs[1, 2].set_title("Weighted Noise pattern")

    # Step 7: The final image
    f = np.clip(image - weighted_noise, 0, 255).astype(np.uint8)

    # Display the final image
    axs[1, 3].imshow(f, cmap="gray")
    axs[1, 3].set_title("The final image")

    # Save the figure containing all the images
    save_path = f"frequency_domain_notch_{image_name}.png"
    plt.tight_layout()
    plt.savefig(save_path)
    plt.show()

    # Save the final filtered image
    final_image_path = f"final_filtered_image_notch_{image_name}.tif"
    Image.fromarray(f).save(final_image_path, format="TIFF")

    return f
```

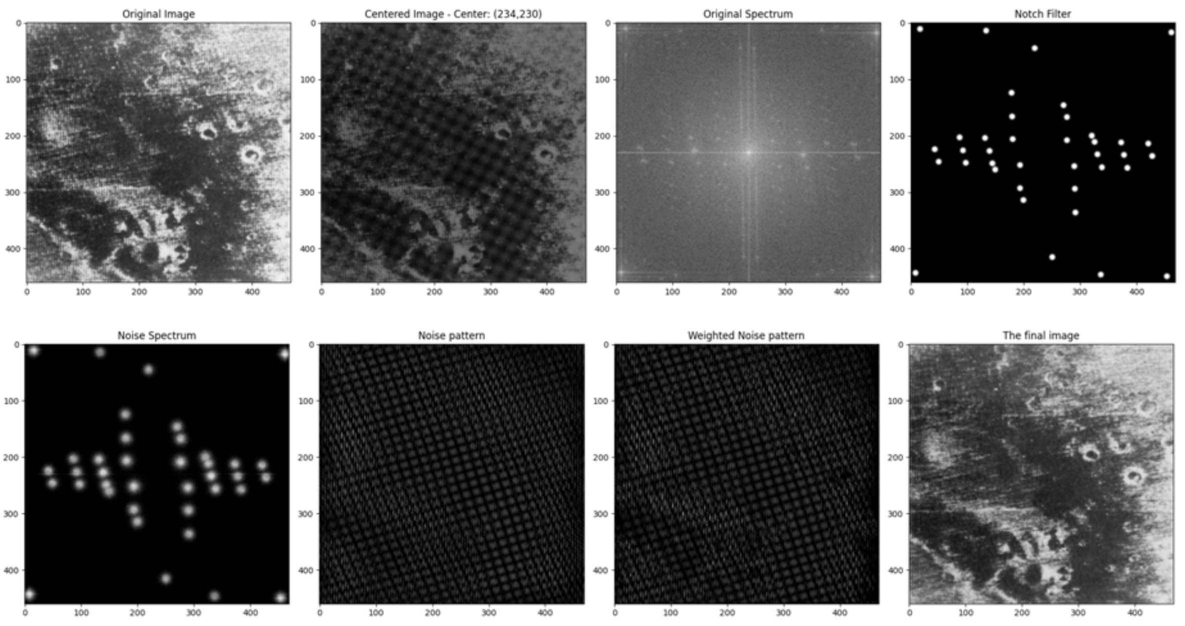
函数调用:

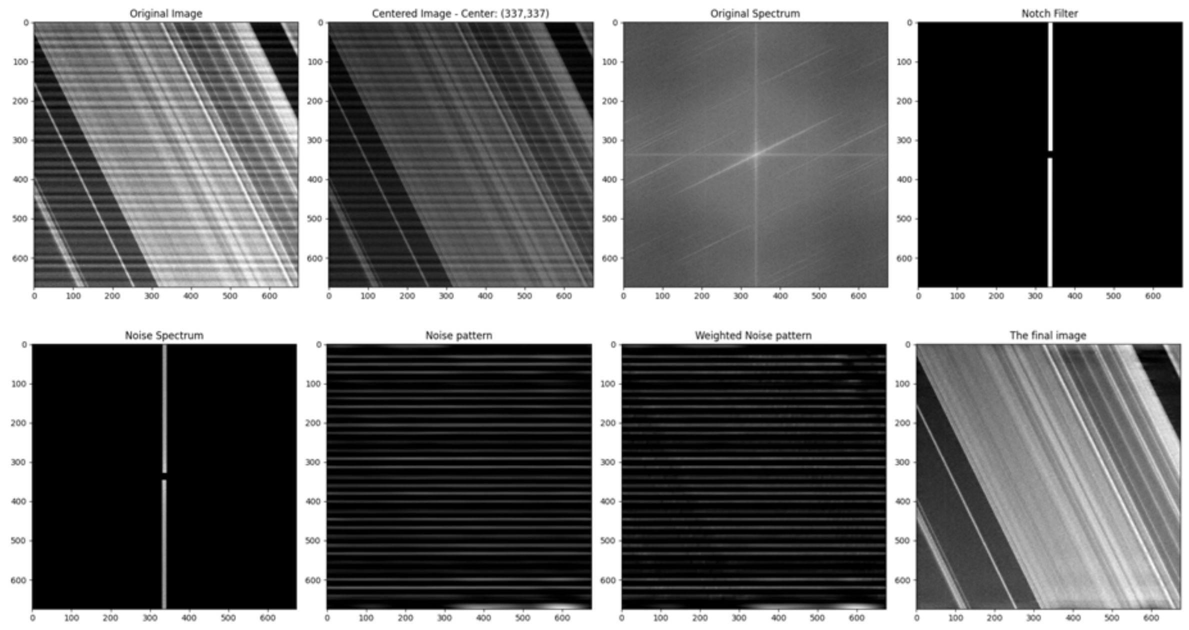
```
# Define filter type mappings
filter_type_dict = {
    "line filter": -2,
    "ideal filter": -1,
    "Gaussian filter": 0,
    "Butterworth filter (n=1)": 1,
    "Butterworth filter (n=2)": 2,
    "Butterworth filter (n=3)": 3,
    "Butterworth filter (n=4)": 4,
    "Butterworth filter (n=5)": 5,
    # Add more Butterworth filter orders if needed
}

if __name__ == "__main__":
    # Initialize the option to select which image and filter settings to load
    option = 4

    if option == 1:
        # Load 'shepp_logan.png' and set parameters specific to this image
        image_path = 'shepp_logan.png'
        image_name = 'shepp_logan'
        min_distance = 75  # Minimum distance between detected centers

        # Define x-coordinates for line filters
        centers_x = [413, 497, 531, 581, 612, 698]

        # Define two sets of y-coordinates for line filters
        centers_y1 = [17, 101, 132, 186, 218, 298]
        centers_y2 = [416, 498, 533, 615, 700]

        # Concatenate y-coordinates for general use in 2D product combinations
        centers_y = centers_y1 + centers_y2

        # Define horizontal line filters in each y-set with x bounds
        centers_1 = list(itertools.product(centers_y1, [[centers_x[0],
centers_x[-1], 0]]))
        centers_1.extend(itertools.product(centers_y2, [[centers_x[0],
centers_x[-1], 0]]))

        # Define vertical line filters in each x-set with y bounds
        centers_1.extend(itertools.product(centers_x, [[centers_y1[0],
centers_y1[-1], 1]]))
        centers_1.extend(itertools.product(centers_x, [[centers_y2[0],
centers_y2[-1], 1]]))

        # Flatten tuple list for centers to format (y, x_start, x_end,
orientation)
        centers_1 = [(x, *y) for x, y in centers_1]
        cutoffs_1 = [5] * len(centers_1)  # Set a 20-pixel cutoff for all
centers_1 entries

        # Retrieve the filter type code for line filter from dictionary
        filter_type_1 = filter_type_dict.get("line filter", -2)

        # Define circular filter centers across the full x and y sets
        centers_2 = list(itertools.product(centers_x, centers_y))
        cutoffs_2 = [10] * len(centers_2)  # Set 30-pixel cutoff for all
centers_2 entries

        # Retrieve the filter type code for ideal filter from dictionary
        filter_type_2 = filter_type_dict.get("ideal filter", -1)

        # Group centers, cutoffs, and filter types for use in frequency domain
function
        centers = [centers_1, centers_2]
        cutoffs = [cutoffs_1, cutoffs_2]
        filter_types = [filter_type_1, filter_type_2]

        # Set window size
        window_size = (9, 9)

    elif option == 2:
        # Load 'DIP Fig 04.64(a)(car_75DPI_Moire).tif' and set filter settings
        image_path = 'DIP Fig 04.64(a)(car_75DPI_Moire).tif'
        image_name = 'DIP Fig 04.64(a)(car_75DPI_Moire)'
        min_distance = 25  # Set minimum distance between detected centers
        # Define specific coordinates and cutoff values for Butterworth filters
        centers = [[[111, 81],
                    [113, 161],
                    [111, 39],
                    [113, 202]]]
        cutoffs = [[9, 9, 9, 9]]  # Cutoffs correspond to each center in centers

        # Use dictionary to get the filter type code for Butterworth (defaulting
to 0)
        filter_types = [filter_type_dict.get("Butterworth filter (n=4)", 4)]

        # Set window size
        window_size = (9, 9)

    elif option == 3:
        # Load 'DIP Fig 04.65(a)(cassini).tif' and set line filter settings
        image_path = 'DIP Fig 05.20(a)(NASA_Mariner6_Mars).tif'
        image_name = 'DIP Fig 05.20(a)(NASA_Mariner6_Mars)'
        min_distance = 25  # Set minimum distance for center detection

        # Define two vertical line filters with specific x and y ranges
        centers = [[[453, 449],
                    [336, 446],
                    [250, 415],
                    [289, 254],
                    [338, 256],
                    [330, 233],
                    [377, 234],
                    [383, 257],
                    [427, 236],
                    [420, 214],
                    [372, 212],
                    [325, 211],
                    [276, 208],
                    [461, 17 ],
                    [291, 336],
                    [290, 294],
                    [276, 167],
                    [270, 146],
                    [320, 200]]]
        cutoffs = [[5] * len(centers[0])]
        filter_types = [filter_type_dict.get("Butterworth filter (n=4)", 4)]

        # Set window size
        window_size = (9, 9)

    else:
        # Load 'DIP Fig 04.65(a)(cassini).tif' and set line filter settings
        image_path = 'DIP Fig 04.65(a)(cassini).tif'
        image_name = 'DIP Fig 04.65(a)(cassini)'
        min_distance = 25  # Set minimum distance for center detection

        # Define two vertical line filters with specific x and y ranges
        centers = [[[337, 0, 327, 1]]]
        cutoffs = [[5] * len(centers)]  # Set 10-pixel cutoff for each center

        # Retrieve the filter type code for line filter from dictionary
        filter_types = [filter_type_dict.get("line filter", -2)]
        # Set window size
        window_size = (31, 31)

    # Load and convert the image to grayscale format
    img = Image.open(image_path).convert('L')
    image_array = np.array(img)  # Convert image to a numpy array for processing

    # Run the center detection function in frequency domain with specified
distance
    frequency_domain_center_detection(image_array, image_name, min_distance)

    # Apply the frequency domain notch filtering with specified centers and
cutoffs
    filter_image = frequency_domain_notch(image_array, centers, cutoffs,
image_name, filter_types, window_size)
```

**运行结果:**




Original Image
Centered Image - Center: (84,123)
Original Spectrum
Notch Filter
Noise Spectrum
Noise pattern
Weighted Noise pattern
The final image


Original Image
Centered Image - Center: (234,230)
Original Spectrum
Notch Filter
Noise Spectrum
Noise pattern
Weighted Noise pattern
The final image


Original Image
Centered Image - Center: (337,337)
Original Spectrum
Notch Filter
Noise Spectrum
Noise pattern
Weighted Noise pattern
The final image