图像处理与可视化 Homework 04

学号: 21307140051
姓名: 雍崔扬

Problem 1

Part (1)

实现频域低通平滑操作，并把算法应用与图片上，显示原图的频谱图、频域操作结果的频谱图以及操作结果.
(离散 Fourier 变换和逆变换可以调用库函数)

Solution:
频率域中的滤波过程如下:

- ① 将大小为 $M \times N$ 的输入图像 $f(x,y)$ 零填充为 $P \times Q$:

$$P = 2M$$
$$Q = 2N$$
$$f_{\text{padded}}(x,y) := \begin{cases} f(x,y) & \text{if } 0 \le x \le M-1 \text{ and } 0 \le y \le N-1 \\ 0 & \text{if } M \le x \le 2M-1 \text{ and } N \le y \le 2N-1 \end{cases}$$

- ② 对 $f_{\text{padded}}(x,y)(-1)^{x+y}$ 计算二维离散 Fourier 变换 $\tilde{f}(\mu,\nu)$ (其中心位于 $(\frac{P}{2},\frac{Q}{2}) = (M,N)$)
- ③ 构建一个中心在 $(\frac{P}{2},\frac{Q}{2}) = (M,N)$ 处、大小为 $P \times Q$ 的实对称滤波器传递函数 $\tilde{h}(\mu,\nu)$
  这里我们使用 Gauss 低通滤波器 (Gaussian lowpass filter, GLPF):

$$\tilde{h}(\mu,\nu) = \exp\{-\frac{1}{2d_0^2}d^2(\mu,\nu)\}$$

  其中 $d(\mu,\nu) := \sqrt{(\mu-\frac{P}{2})^2 + (\nu-\frac{Q}{2})^2}$ 是频率域中的点 (μ,ν) 到 $P \times Q$ 频率矩形中心的距离.
  而标准差 d_0 是关于中心分离度的测度，相当于截止频率.
- ④ 计算 $\tilde{g}(\mu,\nu) = \tilde{h}(\mu,\nu)\tilde{f}(\mu,\nu)$
- ⑤ 计算 $g_{\text{padded}}(x,y) = \text{Re}\{\mathscr{F}^{-1}(\tilde{g}(\mu,\nu))\}(-1)^{x+y}$ (尺寸为 $P \times Q$)
- ⑥ 从 $g_{\text{padded}}(x,y)$ 的左上象限提取一个大小为 $M \times N$ 的区域，得到滤波结果 $g(x,y)$

上述算法的 Python 实现如下:

```
def frequency_domain_smooth(image, cutoff, image_name):

    # Plotting setup
    fig, axs = plt.subplots(2, 4, figsize=(20, 10))
    axs[0, 0].imshow(image, cmap="gray")
    axs[0, 0].set_title("Original Image")

    # Step 1: Zero-padding the image to size P x Q where P = 2M, Q = 2N
    M, N = image.shape
    P, Q = 2 * M, 2 * N
    padded_image = np.zeros((P, Q))
```

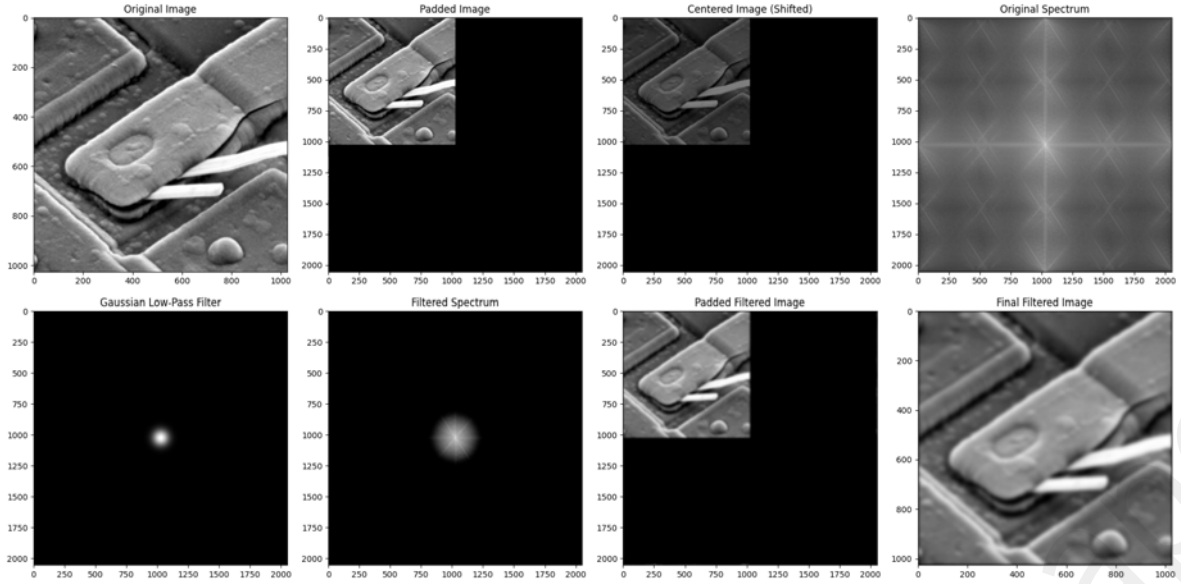
```
    padded_image[:M, :N] = image

    # Display padded image
    axs[0, 1].imshow(padded_image, cmap="gray")
    axs[0, 1].set_title("Padded Image")

    # Step 2: Multiply padded image by (-1)^(x+y) to center the Fourier
transform
    centered_image = padded_image * np.fromfunction(lambda x, y: (-1)**(x + y),
(P, Q))

    # Display centered image
    axs[0, 2].imshow(np.clip(centered_image, 0, 255).astype(np.uint8),
cmap="gray")
    axs[0, 2].set_title("Centered Image (Shifted)")

    # Step 3: Compute the 2D Fourier Transform of the centered image
    F_uv = np.fft.fft2(centered_image)

    # Display original spectrum
    axs[0, 3].imshow(np.log(1 + np.abs(F_uv)), cmap="gray")
    axs[0, 3].set_title("Original Spectrum")

    # Step 4: Create the Gaussian low-pass filter with correct shape
    u, v = np.meshgrid(np.arange(-P//2, P//2), np.arange(-Q//2, Q//2),
indexing='ij')
    D = np.sqrt(u**2 + v**2)
    H_uv = np.exp(-D**2 / (2 * cutoff**2))  # Gaussian low-pass filter

    # Display Gaussian filter
    axs[1, 0].imshow(H_uv, cmap="gray")
    axs[1, 0].set_title("Gaussian Low-Pass Filter")

    # Apply the filter in the frequency domain
    G_uv = H_uv * F_uv

    # Display the filter spectrum
    axs[1, 1].imshow(np.log(1 + np.abs(G_uv)), cmap="gray")
    axs[1, 1].set_title("Filtered Spectrum")

    # Step 5: Compute the inverse Fourier Transform and remove centering shift
    g_padded = np.fft.ifft2(G_uv)
    g_padded = np.real(g_padded) * np.fromfunction(lambda x, y: (-1)**(x + y),
(P, Q))
    g_padded = np.clip(g_padded, 0, 255).astype(np.uint8)

    # Display g_padded
    axs[1, 2].imshow(g_padded, cmap="gray")
    axs[1, 2].set_title("Padded Filtered Image")

    # Step 6: Extract the top-left M x N portion to obtain the final result
    g = g_padded[:M, :N]

    # Display final filtered image
    axs[1, 3].imshow(g, cmap="gray")
    axs[1, 3].set_title("Final Filtered Image")

    # Save the figure containing all the images
```

```
    save_path = f"frequency_domain_smooth_cutoff_{cutoff}_{image_name}.png"
    plt.tight_layout()
    plt.savefig(save_path)

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

    plt.show()

    return g
```

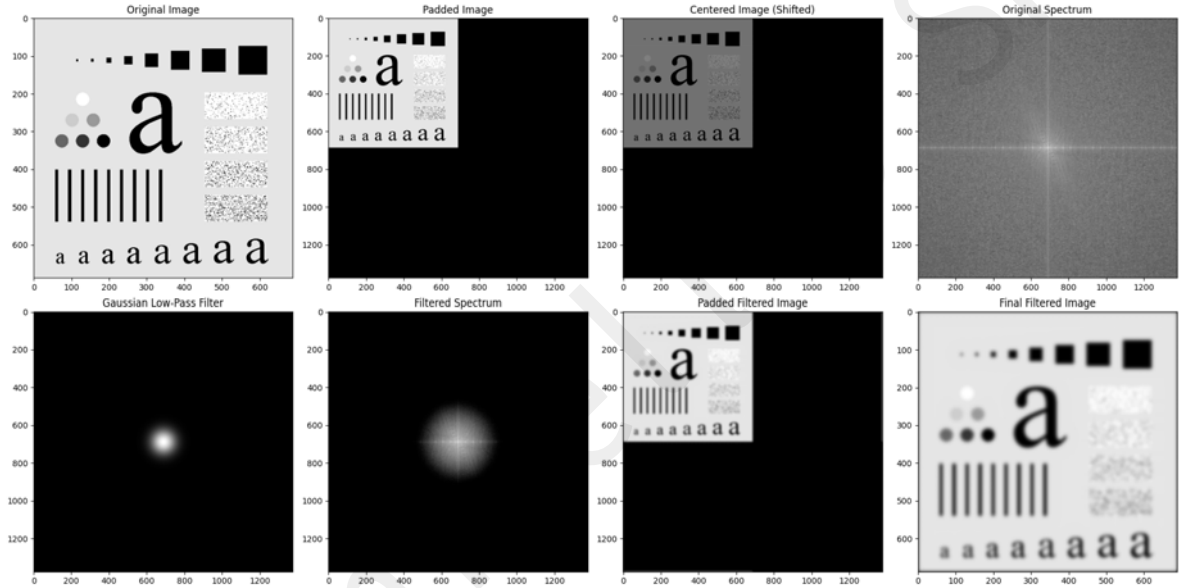
函数调用:

```
if __name__ == "__main__":
    # Load the image
    option = 1
    if option == 1:
        image_path = 'DIP Fig 04.29(a)(blown_ic).tif'
        cutoff = 45
        image_name = 'DIP Fig 04.29(a)(blown_ic)'
    elif option == 2:
        image_path = 'DIP Fig 04.41(a)(characters_test_pattern).tif'
        cutoff = 45
        image_name = 'DIP Fig 04.41(a)(characters_test_pattern)'
    elif option == 3:
        image_path = 'DIP Fig 04.58(a)(blurry_moon).tif'
        cutoff = 60
        image_name = 'DIP Fig 04.58(a)(blurry_moon)'
    else:
        image_path = 'DIP Fig 04.27(a)(woman).tif'
        cutoff = 60
        image_name = 'DIP Fig 04.27(a)(woman)'

    img = Image.open(image_path).convert('L')  # Convert it in grayscale
    image_array = np.array(img)

    # Apply the frequency domain lowpass filter with a chosen cutoff frequency
    filtered_image = frequency_domain_smooth(image_array, cutoff=cutoff,
image_name=image_name)
```

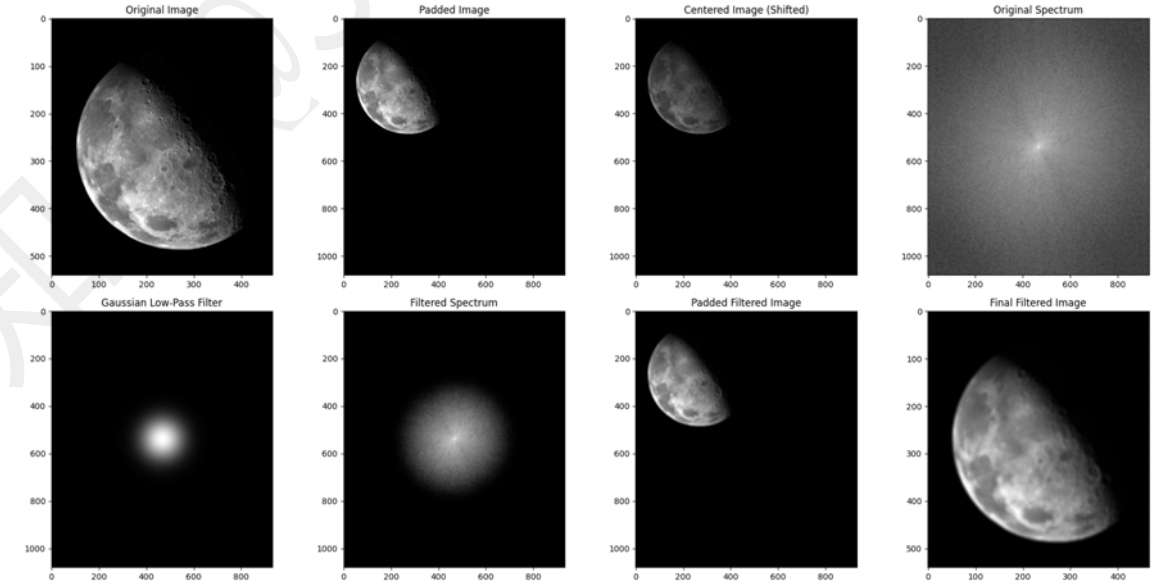
运行结果 1:



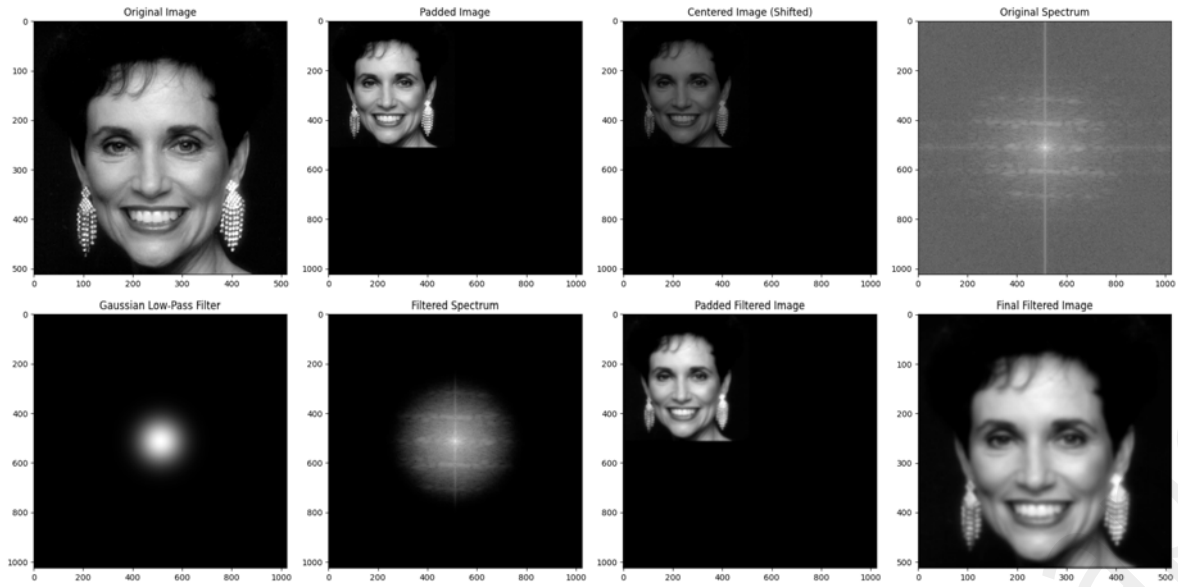
运行结果 2:



运行结果 3:



运行结果 4:



Part (2)

实现频域高通锐化操作，并把算法应用与图片上，显示原图的频谱图、频域操作结果的频谱图，以及操作结果.
(离散 Fourier 变换和逆变换可以调用库函数)

Solution:
中心化的 Laplace 滤波器传递函数:

$$\tilde{h}(\mu,\nu) := -4\pi^2 d^2(\mu,\nu)$$

其中 $d(\mu,\nu) := \sqrt{(\mu-\frac{P}{2})^2+(\nu-\frac{Q}{2})^2}$ 是频率域中的点 (μ,ν) 到 $P\times Q$ 频率矩形中心的距离.
使用这个传递函数，我们可以得到空间域中图像 $f(x,y)$ 的 Laplace 滤波结果:

$$\nabla^2 f(x,y) = \mathscr{F}^{-1}\{\tilde{h}(\mu,\nu)\tilde{f}(\mu,\nu)\}$$

值得注意的是，为避免引入 DFT 标定因子，我们需要做以下归一化处理:

- ① 将 $f(x,y)$ 归一化到 $[0,1]$ 后再通过对 $f(x,y)(-1)^{x+y}$ 做二维离散 Fourier 变换得到 $\tilde{f}(\mu,\nu)$
- ② 计算 $\nabla^2 f(x,y) = \mathscr{F}^{-1}\{\tilde{h}(\mu,\nu)\tilde{f}(\mu,\nu)\}$ 后除以其绝对值的最大值，进而将其限定到区间 $[-1,1]$

锐化图像 $g(x,y) := f(x,y) + c\nabla^2 f(x,y)$ (最后还需放缩至 $f(x,y)$ 的原灰度范围)
其中 $c=-1$，因为 $\tilde{h}(\mu,\nu)$ 是负的，对应的 Laplace 空间核具有负中心系数.

上述算法的 Python 代码为:

```
def frequency_domain_sharpen(image, image_name):

    # Plotting setup
    fig, axs = plt.subplots(2, 4, figsize=(20, 10))
    axs[0, 0].imshow(image, cmap="gray")
    axs[0, 0].set_title("Original Image")

    # Step 1: Zero-padding the image to size P x Q where P = 2M, Q = 2N
    M, N = image.shape
    P, Q = 2 * M, 2 * N
```

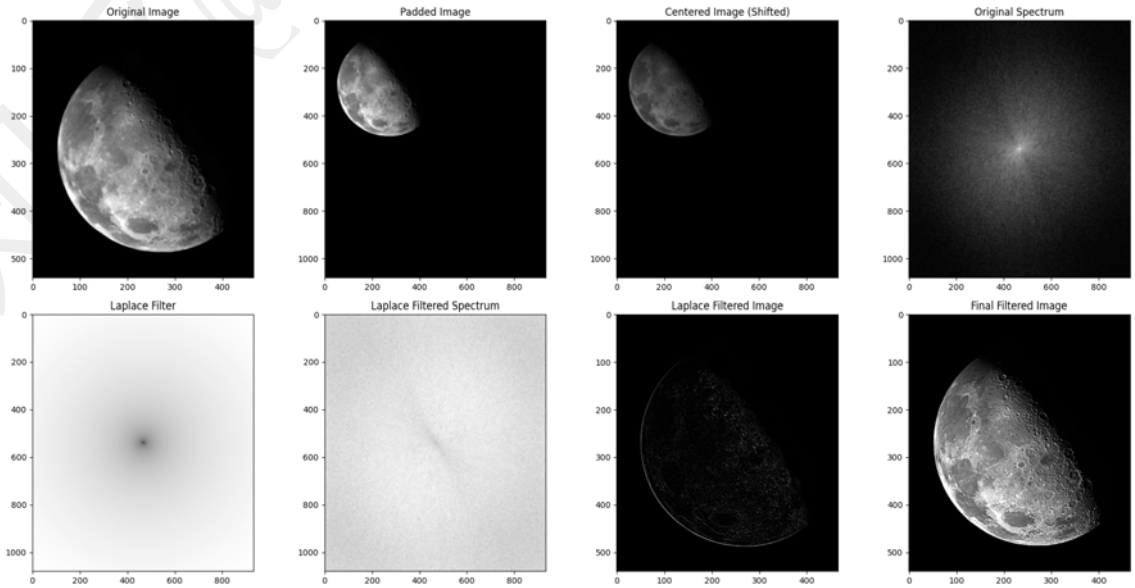
```
    image_normalized = (image.astype(np.float64) - np.min(image)) /
(np.max(image) - np.min(image))
    padded_image = np.zeros((P, Q)).astype(np.float64)
    padded_image[:M, :N] = image_normalized

    # Display padded image
    axs[0, 1].imshow(np.clip(255 * padded_image, 0, 255).astype(np.uint8),
cmap="gray")
    axs[0, 1].set_title("Padded Image")

    # Step 2: Multiply padded image by (-1)^(x+y) to center the Fourier
transform
    centered_image = padded_image * np.fromfunction(lambda x, y: (-1)**(x + y),
(P, Q))

    # Display centered image
    axs[0, 2].imshow(np.clip(255 * centered_image, 0, 255).astype(np.uint8),
cmap="gray")
    axs[0, 2].set_title("Centered Image (Shifted)")

    # Step 3: Compute the 2D Fourier Transform of the centered image
    F_uv = np.fft.fft2(centered_image)

    # Display original spectrum
    axs[0, 3].imshow(np.log(1 + np.abs(F_uv)), cmap="gray")
    axs[0, 3].set_title("Original Spectrum")

    # Step 4: Create the Laplace filter
    u, v = np.meshgrid(np.arange(-P//2, P//2), np.arange(-Q//2, Q//2),
indexing='ij')
    D = np.sqrt(u**2 + v**2)
    H_uv = -4 * np.pi**2 * D ** 2 # Laplace filter

    # Display Laplace filter
    axs[1, 0].imshow(np.log(1 + np.abs(H_uv)), cmap="gray")
    axs[1, 0].set_title("Laplace Filter")

    # Apply the filter in the frequency domain
    Laplace_uv = H_uv * F_uv

    # Display Laplace filter
    axs[1, 1].imshow(np.log(1 + np.abs(Laplace_uv)), cmap="gray")
    axs[1, 1].set_title("Laplace Filtered Spectrum")

    Laplace_filtered = np.fft.ifft2(Laplace_uv)
    Laplace_filtered = np.real(Laplace_filtered) * np.fromfunction(lambda x, y:
(-1)**(x + y), (P, Q))
    Laplace_filtered = Laplace_filtered / np.max(np.abs(Laplace_filtered))
    Laplace_filtered = Laplace_filtered[:M, :N]

    # Display Laplace filter
    axs[1, 2].imshow(np.clip(255 * Laplace_filtered, 0, 255).astype(np.uint8),
cmap="gray")
    axs[1, 2].set_title("Laplace Filtered Image")

    # Step 6: Extract the top-left M x N portion to obtain the final result
    g = image - Laplace_filtered * (np.max(image) - np.min(image))
    g = np.clip(g, 0, 255).astype(np.uint8)
```

```
    # Display final filtered image
    axs[1, 3].imshow(g, cmap="gray")
    axs[1, 3].set_title("Final Filtered Image")

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

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

    plt.show()

    return g
```

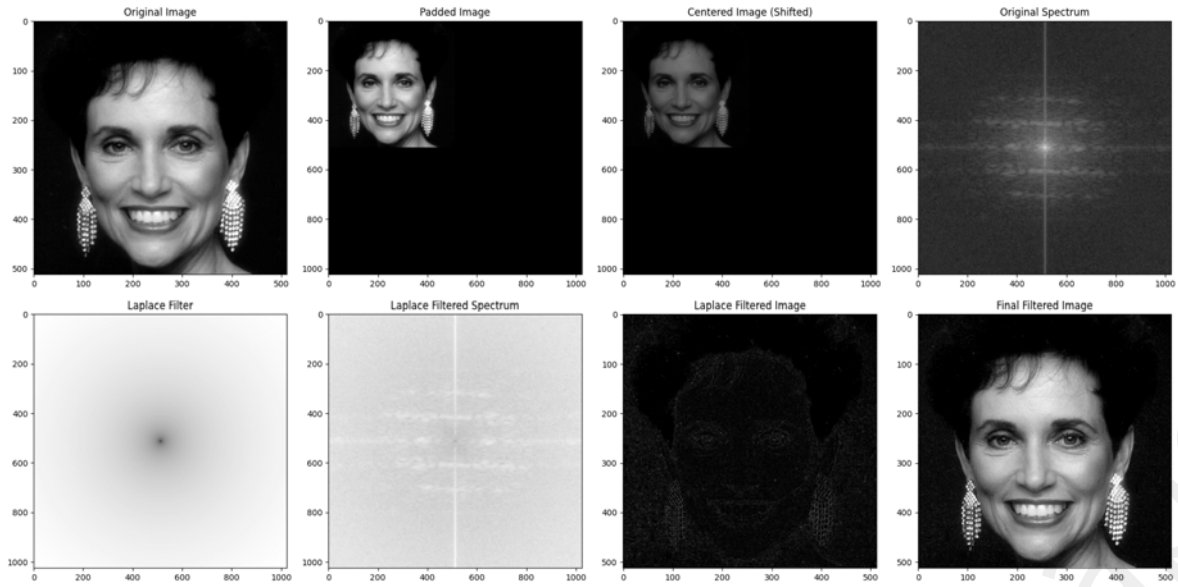
函数调用:

```
if __name__ == "__main__":
    # Load the image
    option = 2
    if option == 1:
        image_path = 'DIP Fig 04.58(a)(blurry_moon).tif'
        image_name = 'DIP Fig 04.58(a)(blurry_moon)'
    else:
        image_path = 'DIP Fig 04.27(a)(woman).tif'
        image_name = 'DIP Fig 04.27(a)(woman)'

    img = Image.open(image_path).convert('L')  # Convert it in grayscale
    image_array = np.array(img)

    # Apply the frequency domain lowpass filter with a chosen cutoff frequency
    filtered_image = frequency_domain_sharpen(image_array,
image_name=image_name)
```

运行结果 1:



运行结果 2:



Problem 2

编程实现基于频域的选择滤波器方法，去除大脑 CT 体膜图像 (Shepp-Logan) 中的条纹.
或自己设计一个有周期噪声的图片，并用频域选择滤波器去除噪声.
(离散 Fourier 变换和逆变换可以调用库函数)

Solution:

(1) 算法基础

陷波滤波器 (notch filters) 是最有用的选择性滤波器，它可以选择性地修改 DFT 的局部区域.
陷波滤波器阻止 (或通过) 事先定义的频率矩形邻域中的频率.
由于零相移滤波器必须中心 $(\frac{P}{2}, \frac{Q}{2})$ 对称，
故中心为 $(\frac{P}{2}+\mu_0, \frac{Q}{2}+\nu_0)$ 的陷波滤波器传递函数在 $(\frac{P}{2}-\mu_0, \frac{Q}{2}-\nu_0)$ 位置必须有一个对应的陷波.

陷波带阻滤波器 (notch reject filter) 传递函数可用中心被平移到陷波滤波中心的高通滤波器函数的乘积来产生.
其一般形式为:

$$\tilde{h}_{\mathrm{NR}}(\mu,\nu) := \prod_{k=1}^{K} \tilde{h}_k(\mu,\nu)\tilde{h}_{-k}(\mu,\nu)$$

其中 $\tilde{h}_k(\mu,\nu)$ 和 $\tilde{h}_{-k}(\mu,\nu)$ 是同形的高通滤波器传递函数，
中心分别为 $(\frac{P}{2}+\mu_k, \frac{Q}{2}+\nu_k)$ 和 $(\frac{P}{2}-\mu_k, \frac{Q}{2}-\nu_k)$

例如包含 K 个陷波对的 Butterworth 陷波带阻滤波器:

$$\tilde{h}_{\mathrm{NR}}(\mu,\nu) := \prod_{k=1}^{K} \left\{ \frac{1}{1+(d(\mu-\mu_k,\nu-\nu_k)/d_k)^{2n}} \right\} \left\{ \frac{1}{1+(d(\mu+\mu_k,\nu+\nu_k)/d_k)^{2n}} \right\}$$

其中 d_k 为第 k 个陷波对的截止频率，$d(\mu-\mu_k,\nu-\nu_k)$ 和 $d(\mu+\mu_k,\nu+\nu_k)$ 为距离函数:

$$d(\mu-\mu_k,\nu-\nu_k) = \sqrt{(\mu-\frac{P}{2}-\mu_k)^2+(\nu-\frac{Q}{2}-\nu_k)^2}$$

$$d(\mu+\mu_k,\nu+\nu_k) = \sqrt{(\mu-\frac{P}{2}+\mu_k)^2+(\nu-\frac{Q}{2}+\nu_k)^2}$$

陷波带通滤波器 (notch pass filter) 可由 $\tilde{h}_{\mathrm{NP}}(\mu,\nu) := 1 - \tilde{h}_{\mathrm{NR}}(\mu,\nu)$ 简单得到.

(2) 算法实现

从图像频谱提取亮点中心的函数 `frequency_domain_center_detection`:

```
def frequency_domain_center_detection(image, image_name, min_distance=50):
    # Plotting setup: Create a 2x3 grid of subplots
    fig, axs = plt.subplots(2, 3, figsize=(20, 10))

    # Display the original image in the first subplot
    M, N = image.shape
    axs[0, 0].imshow(image, cmap="gray")
    axs[0, 0].set_title(f"Original Image - size: {M}x{N}")

    # Step 1: Zero-padding the image to twice its size, and display it in the
second subplot
    P, Q = 2 * M, 2 * N  # Create new dimensions for the padded image
    padded_image = np.zeros((P, Q)).astype(np.float64)
    padded_image[:M, :N] = image.astype(np.float64)  # Place original image in
the top-left corner of the padded image
    axs[0, 1].imshow(padded_image, cmap="gray")
    axs[0, 1].set_title(f"Padded Image - size: {P}x{Q}")

    # Step 2: Apply Fourier transform centering by multiplying with (-1)^(x+y)
    # This shifts the zero-frequency component to the center
    centered_image = padded_image * np.fromfunction(lambda x, y: (-1)**(x + y),
(P, Q))
    axs[0, 2].imshow(np.clip(centered_image, 0, 255).astype(np.uint8),
cmap="gray")
    axs[0, 2].set_title(f"Centered Image - Center: ({N},{M})")

    # Step 3: Perform 2D Fourier Transform on the centered image to obtain the
frequency spectrum
    F_uv = np.fft.fft2(centered_image)
    display_spectrum = np.log(1 + np.abs(F_uv))  # Apply log scaling to
visualize the spectrum better
    axs[1, 0].imshow(display_spectrum, cmap="gray")
    axs[1, 0].set_title("Original Spectrum")

    # Step 4: Calculate the mean and standard deviation of the spectrum to
identify bright spots
    mean_val = np.mean(display_spectrum)
    std_val = np.std(display_spectrum)
    threshold = mean_val + 2 * std_val  # Set threshold for identifying bright
spots (based on mean and std)

    # Get coordinates of bright spots where the spectrum exceeds the threshold
    bright_spots = np.argwhere(display_spectrum > threshold)
    highlight_spectrum = np.ones_like(display_spectrum) * 255  # Start with a
white background
    for i, (x, y) in enumerate(bright_spots):
        distance_from_center = np.sqrt((x - M)**2 + (y - N)**2)
        if distance_from_center > min_distance:
            highlight_spectrum[x, y] = 0  # Mark bright spots in black
```

```
    axs[1, 1].imshow(highlight_spectrum, cmap="gray")
    axs[1, 1].set_title("Highlighted Spectrum")

    # Step 5: Apply DBSCAN clustering algorithm to group the bright spots
    db = DBSCAN(eps=5, min_samples=5).fit(bright_spots)  # DBSCAN clustering
    labels = db.labels_  # Get cluster labels for each bright spot

    # Step 6: Calculate and display the center of each cluster
    unique_labels = set(labels)
    cluster_centers = []

    # Loop through each unique label to calculate the center of each cluster
    for label in unique_labels:
        if label == -1:  # Skip noise points (DBSCAN labels noise as -1)
            continue

        # Get the coordinates of all points in this cluster
        cluster_points = bright_spots[labels == label]

        # Calculate the center of the cluster by taking the mean of the
coordinates
        center = cluster_points.mean(axis=0)
        cluster_centers.append(center)

    # Convert cluster centers to integer values for proper indexing
    cluster_centers = np.round(np.array(cluster_centers)).astype(np.int32)

    # Print the coordinates of each cluster center
    for i, center in enumerate(cluster_centers):
        print(f"Cluster {i} center: (x, y) = ({center[0]}, {center[1]})")

    # Create a new image to highlight the cluster centers and their neighborhood
points
    highlight_spectrum = np.ones_like(display_spectrum) * 255  # Background is
white

    # Set the neighborhood radius around each cluster center to highlight
surrounding points
    radius = 3  # The radius of the neighborhood to highlight around the cluster
center

    # Loop through each cluster center and mark the center and its neighborhood
as black
    for i, center in enumerate(cluster_centers):
        x, y = center[0], center[1]

        # Highlight the neighborhood around the cluster center within the given
radius
        for dx in range(-radius, radius + 1):
            for dy in range(-radius, radius + 1):
                nx, ny = np.clip(x + dx, 0, Q-1), np.clip(y + dy, 0, P-1)
                if 0 <= nx < display_spectrum.shape[0] and 0 <= ny <
display_spectrum.shape[1]:
                    highlight_spectrum[nx, ny] = 0  # Set the center and
neighborhood points to black

        # Annotate the cluster center with its coordinates in red text
```

```
        axs[1, 2].text(y, x, f"({y},{x})", color="red", fontsize=10,
ha="center", va="center")

    # Display the highlighted spectrum with cluster centers and their
neighborhoods
    axs[1, 2].imshow(highlight_spectrum, cmap="gray", vmin=0, vmax=255)
    axs[1, 2].set_title("Cluster Centers")

    # Save the figure containing all the images to a file
    save_path = f"frequency_domain_center_detection_{image_name}.png"
    plt.tight_layout()  # Adjust subplots for better layout
    plt.savefig(save_path)  # Save the figure
    plt.show()  # Display the plot
```

构造陷波滤波器去除周期性噪声的函数 `frequency_domain_notch`：

```
def frequency_domain_notch(image, centers, cutoffs, image_name, filter_types=
[0]):
    # 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, 5, figsize=(20, 10))
    axs[0, 0].imshow(image, cmap="gray")
    axs[0, 0].set_title("Original Image")

    # Step 1: Zero-padding the image to size P x Q where P = 2M, Q = 2N
    M, N = image.shape
    P, Q = 2 * M, 2 * N
    padded_image = np.zeros((P, Q))
    padded_image[:M, :N] = image

    # Display padded image
    axs[0, 1].imshow(padded_image, cmap="gray")
    axs[0, 1].set_title(f"Padded Image - size: {P}x{Q}")

    # Step 2: Multiply padded image by (-1)^(x+y) to center the Fourier
transform
    centered_image = padded_image * np.fromfunction(lambda x, y: (-1)**(x + y),
(P, Q))

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

    # Step 3: Compute the 2D Fourier Transform of the centered image
    F_uv = np.fft.fft2(centered_image)

    # Display original spectrum
    axs[0, 3].imshow(np.log(1 + np.abs(F_uv)), cmap="gray")
    axs[0, 3].set_title("Original Spectrum")

    # Step 4: Create the notch filter with multiple notch centers
```

```
    u, v = np.meshgrid(np.arange(0, P), np.arange(0, Q), indexing='ij')

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

    # 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 = center[1], center[2]  # x-range for the
horizontal line
                    mask_x_range = np.arange(x_start, x_end + 1)
                    mask_y_range = np.arange(np.clip(y_val - cutoff, 0, P),
np.clip(y_val + cutoff, 0, P) + 1)
                    mask = list(itertools.product(mask_y_range, mask_x_range))
                    mask += [(P - i - 1, Q - 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 = center[1], center[2]  # y-range for the
vertical line
                    mask_y_range = np.arange(y_start, y_end + 1)
                    mask_x_range = np.arange(np.clip(x_val - cutoff, 0, Q),
np.clip(x_val + cutoff, 0, Q) + 1)
                    mask = list(itertools.product(mask_y_range, mask_x_range))
                    mask += [(P - i - 1, Q - 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 - (P - center[1]))**2 + (v - (Q -
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

    # Display combined notch filter
    axs[0, 4].imshow(H_uv, cmap="gray")
    axs[0, 4].set_title("Notch Filter")

    # Apply the filter in the frequency domain
    G_uv = H_uv * F_uv

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

    # Step 5: Compute the inverse Fourier Transform and remove centering shift
    g_padded = np.fft.ifft2(G_uv)
    g_padded = np.real(g_padded) * np.fromfunction(lambda x, y: (-1)**(x + y),
(P, Q))
    g_padded = np.clip(g_padded, 0, 255).astype(np.uint8)

    # Display g_padded
    axs[1, 1].imshow(g_padded, cmap="gray")
    axs[1, 1].set_title("Padded Filtered Image")

    # Step 6: Extract the top-left M x N portion to obtain the final result
    g = g_padded[:M, :N]

    # Display final filtered image
    axs[1, 2].imshow(g, cmap="gray")
    axs[1, 2].set_title("Final Filtered Image")

    # Step 7: Compute the noise pattern
    noise_uv = F_uv - G_uv
    noise = image - g

    # Display the notch spectrum
    axs[1, 3].imshow(np.log(1 + np.abs(noise_uv)), cmap="gray")
    axs[1, 3].set_title("Noise Spectrum")

    # Display the noise pattern
    axs[1, 4].imshow(noise, cmap="gray")
    axs[1, 4].set_title("Noise pattern")

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

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

    plt.show()

    return g
```

(3) 运行结果

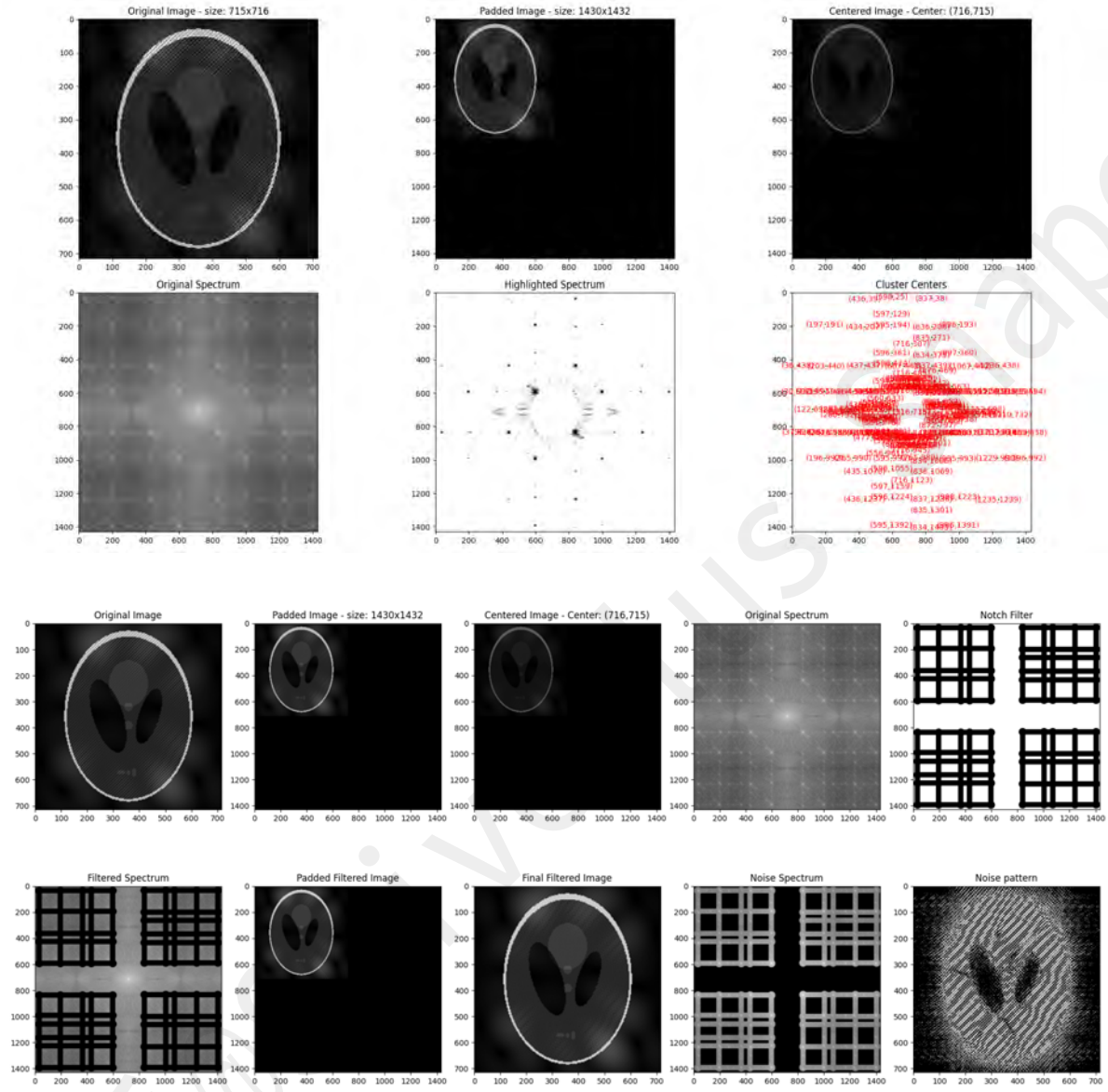
函数调用:

```
# 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 = 2

    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 = 150  # Minimum distance between detected centers

        # Define x-coordinates for line filters
        centers_x = [837, 1001, 1068, 1238, 1400]

        # Define two sets of y-coordinates for line filters
        centers_y1 = [35, 202, 265, 369, 436, 591]
        centers_y2 = [835, 1002, 1064, 1234, 1393]

        # Concatenate y-coordinates for general use in 2D product combinations
        centers_y = [35, 202, 265, 369, 436, 591, 835, 1002, 1064, 1234, 1393]

        # 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 = [20] * len(centers_1)  # Set a 20-pixel cutoff for all
centers_1 entries
```

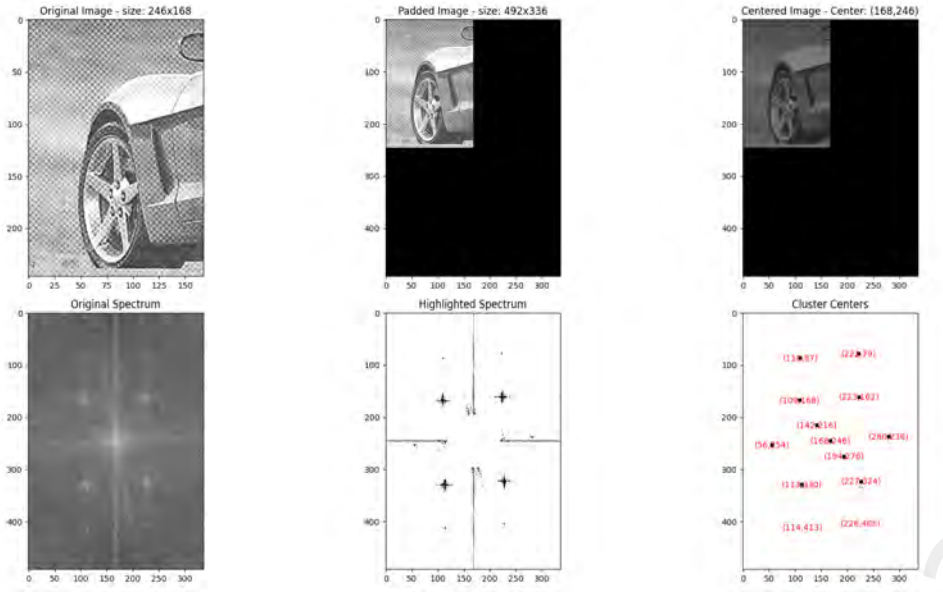
```
        # Retrieve the filter type code for line filter from dictionary
        filter_type_1 = filter_type_dict.get("line filter", -2)

        # Define circular filter centers across the full x and y sets
        centers_2 = list(itertools.product(centers_x, centers_y))
        cutoffs_2 = [30] * 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]

    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 = 50  # Set minimum distance between detected centers

        # Define specific coordinates and cutoff values for Butterworth filters
        centers = [[[109, 168],
                    [113, 330],
                    [114, 413],
                    [110, 87 ],
                    [56 , 254]]]
        cutoffs = [[10, 10, 5, 5, 5]]  # 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)]

    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 = 50  # Set minimum distance for center detection

        # Define two vertical line filters with specific x and y ranges
        centers = [[[674, 0, 659, 1], [674, 689, 1348, 1]]]
        cutoffs = [[10] * 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)]

    # 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)
```

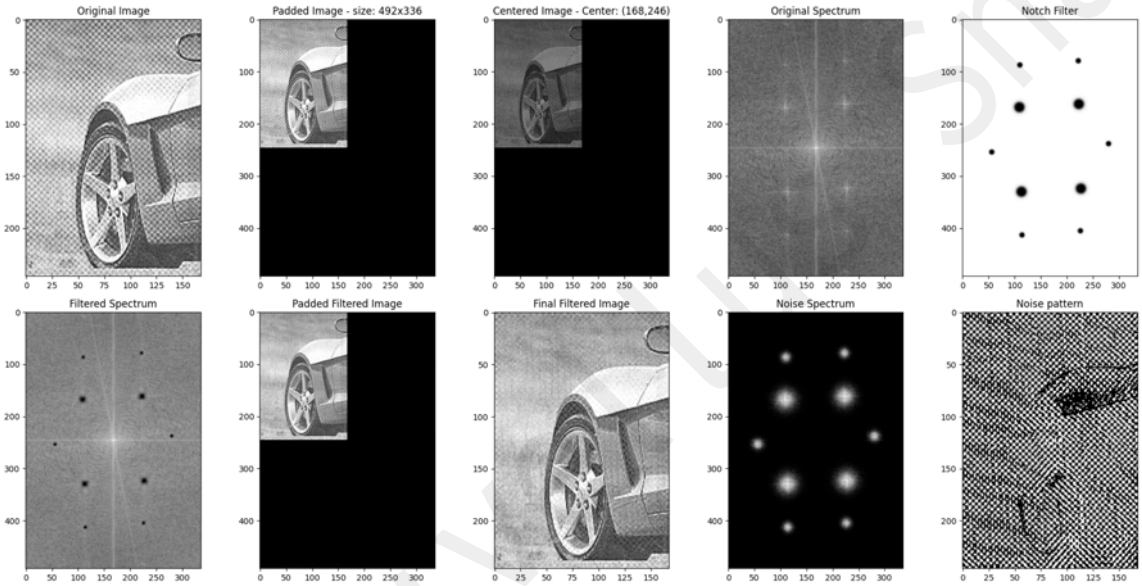
```
    # Apply the frequency domain notch filtering with specified centers and
cutoffs
    filter_image = frequency_domain_notch(image_array, centers, cutoffs,
image_name, filter_types)
```

运行结果 1:

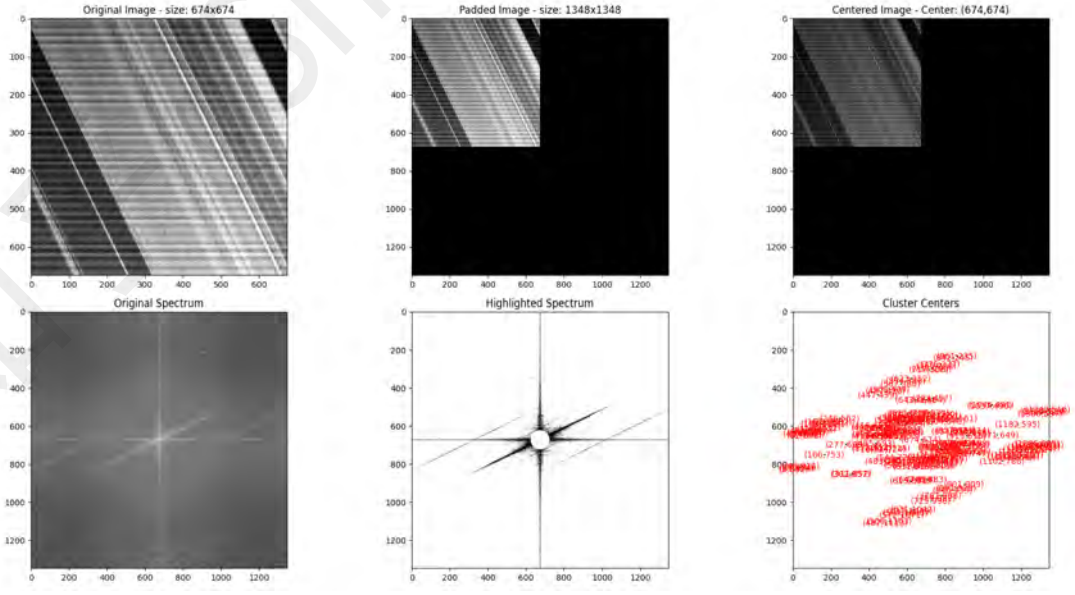


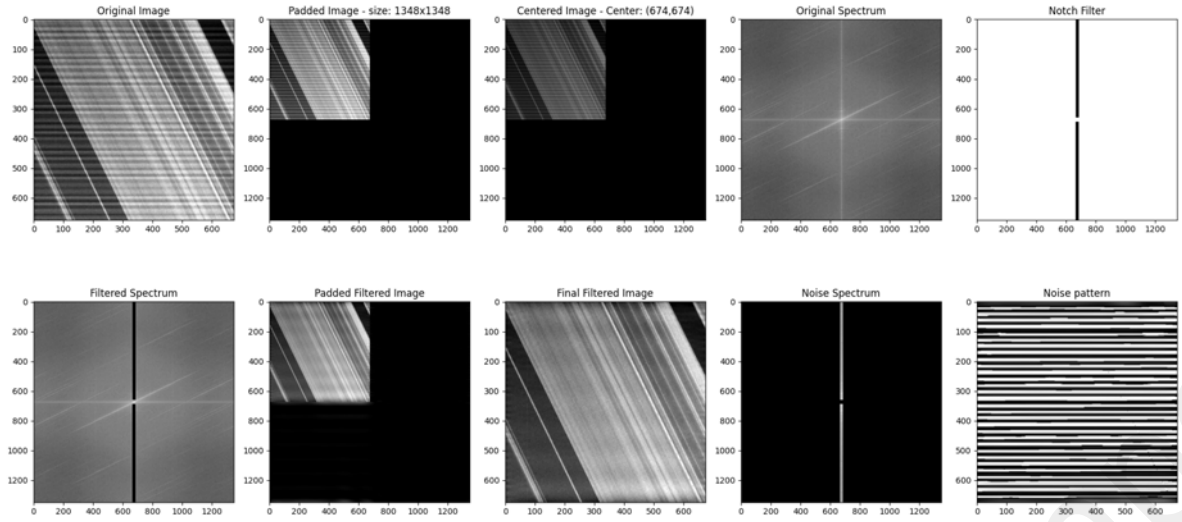
运行结果 2:


Original Image - size: 246x168
Padded Image - size: 492x336
Centered Image - Center: (168,246)
Original Spectrum
Highlighted Spectrum
Cluster Centers


Original Image
Padded Image - size: 492x336
Centered Image - Center: (168,246)
Original Spectrum
Notch Filter
Filtered Spectrum
Padded Filtered Image
Final Filtered Image
Noise Spectrum
Noise pattern

运行结果 3:


Original Image - size: 674x674
Padded Image - size: 1348x1348
Centered Image - Center: (674,674)
Original Spectrum
Highlighted Spectrum
Cluster Centers



The End