深 圳 大 学 实 验 报 告

课程名称：数字图像处理

实验项目名称：**Exp3 Image Processing in Frequency Domain and Image Restoration**

学院：电子与信息工程学院

专业：电子信息工程

指导教师：李斌

报告人：贾苏健　学号：**2022280485**　班级：**06**

实验时间：**2024** 年 **4** 月 **23** 日、**4** 月 **30** 日

实验报告提交时间：**2024** 年 **5** 月 **7** 日

教务部制

实验目的（Aim of Experiment）：

(1) Understand the basic principles of Discrete Fourier Transform, and learn how to perform FFT and IFFT with Python.

(2) Be familiar with the image processing methods in the frequency domain, using Python to perform frequency domain filtering.

(3) Master the basic principles of image restoration, and learn some image restoration algorithms in Python.

实验内容与要求（Experiment Steps and Requirements）：

(1) **FFT and IFFT.**

(a) Load the image rhino.jpg, convert it to grayscale.

(b) Perform FFT. Shift the DC component to the center, and show the phase angles and the magnitudes.

(c) Perform IFFT and show the reconstructed image (Tips: remember to shift the DC component back).

(d) Display the images in the same figure with sub-figures. Add the corresponding title to the sub-figures.

(2) **Ideal Lowpass Filtering.**

(a) Load the image rhino.jpg. Convert it to grayscale.

(b) Perform FFT.

(c) Design an ideal lowpass filter.

(d) Perform frequency domain filtering with the ideal lowpass filter.

(e) Display the original image, the filtered image, the original FFT magnitude, and filtered FFT magnitude in the same figure with sub-figures. Add the corresponding title to the sub-figure. Observe whether there is any ringing artifact in the filtered image.

(3) **Gaussian Lowpass Filter.**

(a) Load the image lena.jpg. Convert it to grayscale.

(b) Perform FFT.

(c) Perform Gaussian lowpass filtering.

(d) Display the original image, the filtered image, the original FFT magnitude, and filtered FFT magnitude in the same figure with sub-figures. Add the corresponding title to the sub-figure. Observe whether there is any ringing artifact in the filtered image.

(4) **Butterworth Lowpass Filter.**

(a) Load the RGB image lena.jpg.

(b) Perform FFT. (Note that when using color images, pay attention to the parameter axes of functions such as fft, ifft, fftshift and ifftshift).

(c) Design three Butterworth lowpass filters with different cutoff frequencies D_0 and orders n (cut-off frequency D_0 and order n are free to choose).

(d) Perform frequency domain filtering with the designed Butterworth lowpass filters. (e) Obtain filtered images with IFFT. (f) Display the original image and the filtered images in the same figure with sub-figures. Observe their differences. Add the corresponding title to the sub-figures.

(5) **Butterworth Highpass Filter.**

(a) Load the RGB image lena.jpg.

(b) Perform FFT. (Note that when using color images, pay attention to the parameter axes of functions such as fft, ifft, fftshift and ifftshift).

(c) Design three Butterworth highpass filters with different cutoff frequencies D_0 and orders n (cut-off frequency D_0 and order n are free to choose).

(d) Perform frequency domain filtering with the designed Butterworth highpass filters.

(e) Obtain filtered images with IFFT.

(f) Display the original image and the filtered images in the same figure with sub-figures. Observe their differences. Add the corresponding title to the sub-figures.

(6) **Adaptive Median Filter.**

(a) Load the grayscale image noisy_salt_pepper.png.

(b) Use ADAPTIVE median filter for denoising (write code based on the implementation principle of adaptive median filter).

(c) Display the original image and the filtered images in the same figure with sub-figures. Add the corresponding title to the sub-figures.

(7) **Motion Blur, Inverse filtering and Wiener filtering.**

(a) Load the RGB image lena.jpg.

(b) Apply motion blur to it.

(c) Recovering images by using inverse filtering and Wiener filtering, respectively. (Note that when using color images, pay attention to the axes parameters of functions such as fft, ifft, fftshift and ifftshift).

(d) Add noise to the blurred image, and then use Inverse filtering and Wiener filtering to recovere the image, respectively.

(e) Display them in the same figure with sub-figures. Add the corresponding title to the sub-figures.

(8) **Estimating Noise Parameters. (Bonus Task)**

(a) Load the grayscale images noisy_1.png and noisy_2.png respectively.

(b) Select a smooth region in each image and compute the histogram to estimate the noise distribution. Please specify the noise type.

(c) Display the image and the histograms in sub-figures. Add the corresponding title.

(d) Use moment estimation to estimate the noise parameters.

实验代码及数据结果（Experiment Codes and Results）：

(1) **FFT and IFFT.**

(a) Load the image rhino.jpg, convert it to grayscale.

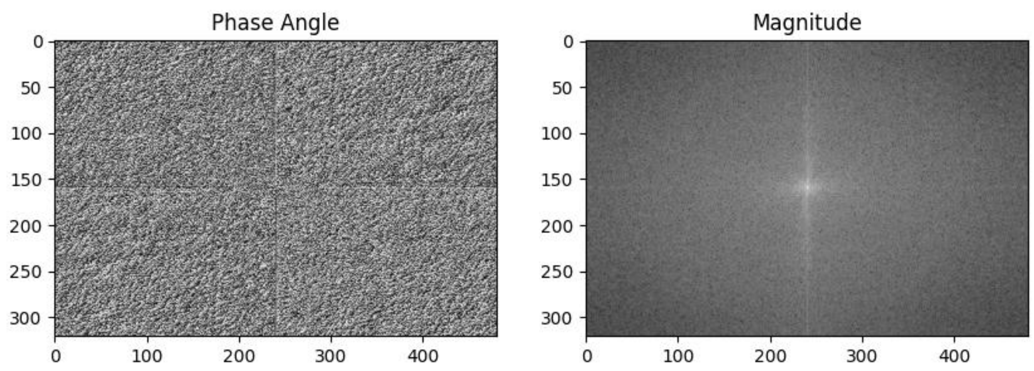
(b) Perform FFT. Shift the DC component to the center, and show the phase angles and the magnitudes.

(c) Perform IFFT and show the reconstructed image (Tips: remember to shift the DC component back).

(d) Display the images in the same figure with sub-figures. Add the corresponding title to the sub-figures.

```python
# 加载图像并转换为灰度
image = np.array(Image.open('images/rhino.jpg').convert('L'))

# 执行FFT
freq = fp.fft2(image)

# 将直流分量移到中心
freq_shifted = fp.fftshift(freq)
```

```python
# 显示相位角和幅度
plt.figure(figsize=(10, 5))

# 相位角
plt.subplot(1, 2, 1)
plt.imshow(np.angle(freq_shifted), cmap='gray')
plt.title('Phase Angle')

# 幅度
plt.subplot(1, 2, 2)
plt.imshow(np.log(np.abs(freq_shifted) + 1), cmap='gray')
plt.title('Magnitude')

plt.show()
```

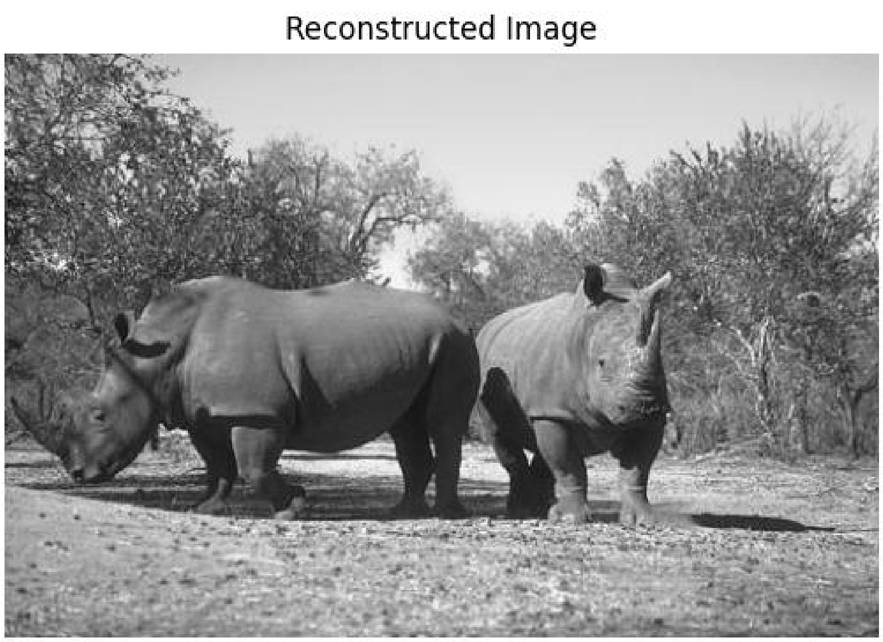



```python
# 执行IFFT
freq_inverse = fp.ifftshift(freq_shifted)
image_reconstructed = fp.ifft2(freq_inverse).real
# 将重建的图像转换为uint8
image_reconstructed = np.round(image_reconstructed)
```

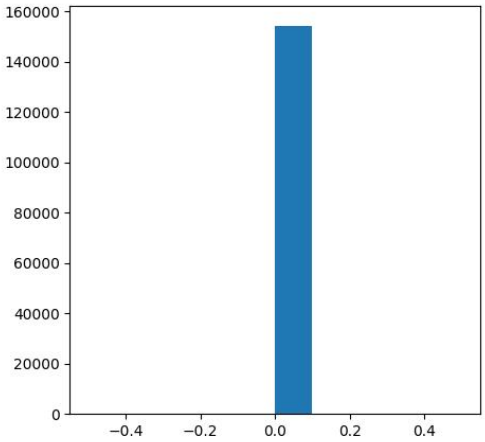
```python
# 检查重建的图像和原始图像是否相等
print(np.equal(image, image_reconstructed).all())
```

✓ 0.0s

```
True
```



```python
dimage = image - image_reconstructed
plt.figure(figsize=(5, 5))
plt.hist(dimage.ravel())
plt.show()
```



(2) **Ideal Lowpass Filtering.**

(a) Load the image rhino.jpg. Convert it to grayscale.

(b) Perform FFT.

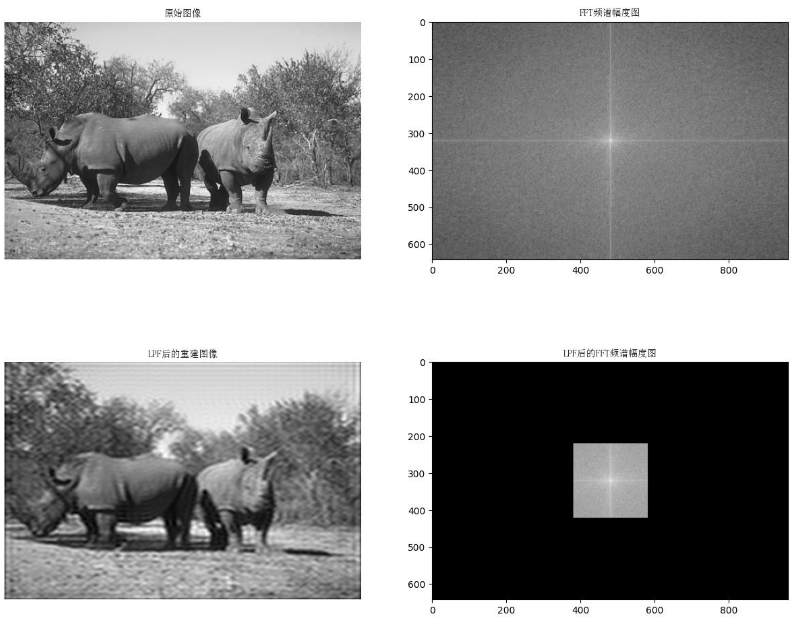
(c) Design an ideal lowpass filter.

(d) Perform frequency domain filtering with the ideal lowpass filter.

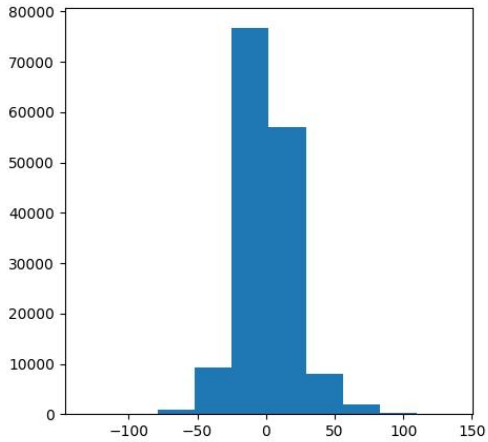
(e) Display the original image, the filtered image, the original FFT magnitude, and filtered FFT magnitude in the same figure with sub-figures. Add the corresponding title to the sub-figure. Observe whether there is any ringing artifact in the filtered image.

```python
# 导入一张图像
H, W = image.shape
freq = fp.fft2(image, s=(H*2, W*2))
(w, h) = freq.shape
half_w, half_h = int(w/2), int(h/2)

freq1 = np.copy(freq)
freq2 = fp.fftshift(freq1)

freq2_low = np.copy(freq2)
freq2_low[half_w-100: half_w+101, half_h-100: half_h+101] = 0
# block the lowfrequencies

freq3 = freq2 - freq2_low
# select only the first 100x100 (low) frequencies, block the high frequencies

image1 = fp.ifft2(fp.ifftshift(freq3)).real[0:H, 0:W]
image1 = np.clip(image1, 0, 255)
image1 = np.rint(image1)
```

```python
plt.figure(figsize=(15, 12))
subplot = plt.subplot(2, 2, 1)
plt.imshow(image, cmap='gray')
subplot.set_title('原始图像', fontproperties="SimSun")
subplot.axis('off')

subplot = plt.subplot(2, 2, 2)
plt.imshow(20*np.log10(0.01 + np.abs(freq2)), cmap='gray')
subplot.set_title('FFT频谱幅度图', fontproperties="SimSun")

subplot = plt.subplot(2, 2, 3)
plt.imshow(image1, cmap='gray')
subplot.set_title('LPF后的重建图像', fontproperties="SimSun")
subplot.axis('off')

subplot = plt.subplot(2, 2,4)
plt.imshow(20*np.log10(0.01 + np.abs(freq3)), cmap='gray')
subplot.set_title('LPF后的FFT频谱幅度图', fontproperties="SimSun")

plt.show()
```



```python
dimage = image - image1
plt.figure(figsize=(5, 5))
plt.hist(dimage.ravel())
plt.show()
```




```python
print(np.equal(image, image_reconstructed).all())
```

✓ 0.0s

```
True
```

(3) **Gaussian Lowpass Filter.**

(a) Load the image lena.jpg. Convert it to grayscale.

(b) Perform FFT.

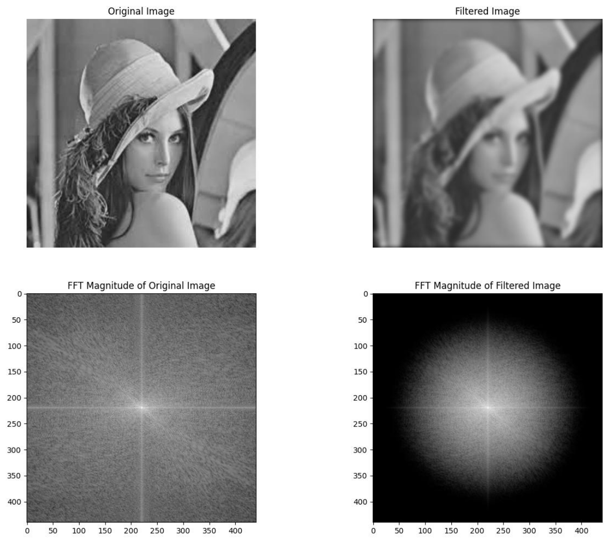
(c) Perform Gaussian lowpass filtering.

(d) Display the original image, the filtered image, the original FFT magnitude, and filtered FFT magnitude in the same figure with sub-figures. Add the corresponding title to the sub-figure. Observe whether there is any ringing artifact in the filtered image.

```python
# Generate a 2D Gaussian kernel array
def gaussian_kernel(kernlen=21, std=3):
    gkern1d = signal.gaussian(kernlen, std=std).reshape(kernlen, 1)
    gkern2d = np.outer(gkern1d, gkern1d)
    return gkern2d
```

```python
# Load image
image_path = 'images/lena.jpg'
image = np.array(Image.open(image_path).convert('L'))

# Perform FFT on the image
H, W = image.shape
freq = fp.fft2(image, s=(H * 2, W * 2))

# Apply Gaussian filter in frequency domain
freq_filtered = ndimage.fourier_gaussian(freq, sigma=2)
# Transform back to spatial domain
filtered_image = fp.ifft2(freq_filtered).real[0:H, 0:W]
# Clip values to [0, 255] and round to integers
filtered_image = np.clip(filtered_image, 0, 255)
filtered_image = np.rint(filtered_image).astype(np.uint8)
```

```python
# Display images and FFT magnitudes
plt.figure(figsize=(15, 12))

# Original image
subplot = plt.subplot(2, 2, 1)
plt.imshow(image, vmin=0, vmax=255, cmap='gray')
subplot.set_title('Original Image')
subplot.axis('off')

# FFT of original image
subplot = plt.subplot(2, 2, 3)
plt.imshow(20 * np.log10(0.01 + np.abs(fp.fftshift(freq))), cmap='gray')
subplot.set_title('FFT Magnitude of Original Image')

# Filtered image
subplot = plt.subplot(2, 2, 2)
plt.imshow(filtered_image, vmin=0, vmax=255, cmap='gray')
subplot.set_title('Filtered Image')
subplot.axis('off')

# FFT of filtered image
subplot = plt.subplot(2, 2, 4)
plt.imshow(20 * np.log10(0.01 + np.abs(fp.fftshift(freq_filtered))), cmap='gray')
subplot.set_title('FFT Magnitude of Filtered Image')

plt.show()
```



(4) **Butterworth Lowpass Filter.**

(a) Load the RGB image lena.jpg.

(b) Perform FFT. (Note that when using color images, pay attention to the parameter axes of functions such as fft, ifft, fftshift and ifftshift).

(c) Design three Butterworth lowpass filters with different cutoff frequencies D_0 and orders n (cut-off frequency D_0 and order n are free to choose).

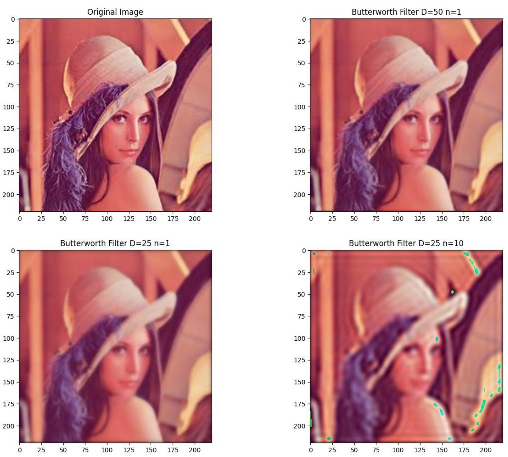
(d) Perform frequency domain filtering with the designed Butterworth lowpass filters.

(e) Obtain filtered images with IFFT. (f) Display the original image and the filtered images in the same figure with sub-figures. Observe their differences. Add the corresponding title to the sub-figures.

```
def calculate_distance(pa, pb):
    distance = sqrt((pa[0] - pb[0]) ** 2 + (pa[1] - pb[1]) ** 2)
    return distance

def create_butterworth_lowpass_filter(size, d, n):
    butterworth_filter = np.zeros(size, dtype=np.float32)
    center_point = tuple(map(lambda x: int((x - 1) / 2), size[0:2]))
    for i in range(size[0]):
        for j in range(size[1]):
            distance = calculate_distance(center_point, (i, j))
            if len(size) == 2:
                butterworth_filter[i, j] = 1 / (1 + (distance / d) ** (2 * n))
            elif len(size) == 3:
                butterworth_filter[i, j, :] = 1 / (1 + (distance / d) ** (2 * n))
    return butterworth_filter
```

```
# Load an RGB image
image_path = 'images/lena.jpg'
image = np.array(Image.open(image_path).convert('RGB'))
H, W, C = image.shape
freq_shift = fp.fftshift(fp.fft2(image, axes=(0, 1), s=(H * 2, W * 2)), axes=(0, 1))

filter_d50_n1 = create_butterworth_lowpass_filter(freq_shift.shape, 100, 1)
filter_d25_n1 = create_butterworth_lowpass_filter(freq_shift.shape, 50, 1)
filter_d25_n10 = create_butterworth_lowpass_filter(freq_shift.shape, 50, 10)

image_d50_n1 = fp.ifft2(fp.ifftshift(freq_shift * filter_d50_n1, axes=(0, 1)), axes=(0, 1)).real[0:H, 0:W, :]
image_d25_n1 = fp.ifft2(fp.ifftshift(freq_shift * filter_d25_n1, axes=(0, 1)), axes=(0, 1)).real[0:H, 0:W, :]
image_d25_n10 = fp.ifft2(fp.ifftshift(freq_shift * filter_d25_n10, axes=(0, 1)), axes=(0, 1)).real[0:H, 0:W, :]
```

```
# Create subplots
plt.figure(figsize=(15, 12))
subplot = plt.subplot(2, 2, 1)
plt.imshow(image, cmap='gray')
subplot.set_title('Original Image')

subplot = plt.subplot(2, 2, 2)
plt.imshow(image_d50_n1.astype(np.uint8), cmap='gray')
subplot.set_title('Butterworth Filter D=50 n=1')

subplot = plt.subplot(2, 2, 3)
plt.imshow(image_d25_n1.astype(np.uint8), cmap='gray')
subplot.set_title('Butterworth Filter D=25 n=1')

subplot = plt.subplot(2, 2, 4)
plt.imshow(image_d25_n10.astype(np.uint8), cmap='gray')
subplot.set_title('Butterworth Filter D=25 n=10')

plt.show()
```



(5) **Butterworth Highpass Filter.**

(a) Load the RGB image lena.jpg.

(b) Perform FFT. (Note that when using color images, pay attention to the parameter axes of functions such as fft, ifft, fftshift and ifftshift).

(c) Design three Butterworth highpass filters with different cutoff frequencies D_0 and orders n (cut-off frequency D_0 and order n are free to choose).

(d) Perform frequency domain filtering with the designed Butterworth highpass filters.

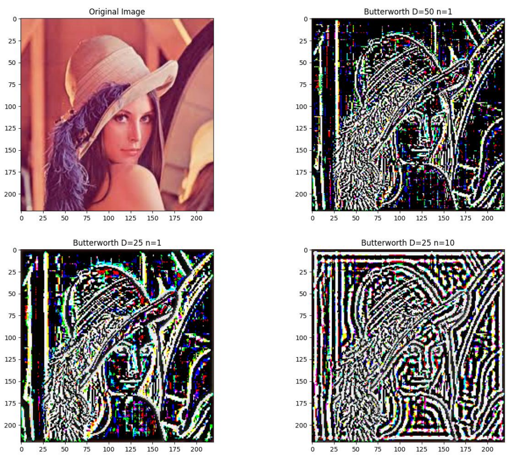
(e) Obtain filtered images with IFFT.

(f) Display the original image and the filtered images in the same figure with sub-figures. Observe their differences. Add the corresponding title to the sub-figures.

```python
def calculate_distance(point_a, point_b):
    distance = sqrt((point_a[0] - point_b[0])**2 + (point_a[1] - point_b[1])**2)
    return distance

def create_butterworth_highpass_filter(size, d_cutoff, n_order):
    transform_matrix = np.zeros(size, dtype=np.float32)
    center_point = tuple(map(lambda x: int((x - 1) / 2), size[0:2]))
    for i in range(size[0]):
        for j in range(size[1]):
            distance = calculate_distance(center_point, (i, j))
            if len(size) == 2:
                transform_matrix[i, j] = 1 / (1 + (distance / d_cutoff)**(2 * n_order))
            elif len(size) == 3:
                transform_matrix[i, j, :] = 1 / (1 + (distance / d_cutoff)**(2 * n_order))
    return 1.0 - transform_matrix
```

```python
# Load an RGB image
image = np.array(Image.open('images/lena.jpg').convert('RGB'))
image_height, image_width, channels = image.shape
freq_shift = fp.fftshift(fp.fft2(image, axes=(0, 1), s=(image_height * 2, image_width * 2)), axes=(0, 1))

filter_d50_n1 = create_butterworth_highpass_filter(freq_shift.shape, 100, 1)
filter_d25_n1 = create_butterworth_highpass_filter(freq_shift.shape, 50, 1)
filter_d25_n10 = create_butterworth_highpass_filter(freq_shift.shape, 50, 10)

image_d50_n1 = fp.ifft2(fp.ifftshift(freq_shift * filter_d50_n1, axes=(0, 1)), axes=(0, 1)).real[0:image_height, 0:image_width, :]
image_d25_n1 = fp.ifft2(fp.ifftshift(freq_shift * filter_d25_n1, axes=(0, 1)), axes=(0, 1)).real[0:image_height, 0:image_width, :]
image_d25_n10 = fp.ifft2(fp.ifftshift(freq_shift * filter_d25_n10, axes=(0, 1)), axes=(0, 1)).real[0:image_height, 0:image_width, :]
```

```python
# Create subplots
plt.figure(figsize=(15, 12))

subplot = plt.subplot(2, 2, 1)
plt.imshow(image, vmin=0, vmax=255)
subplot.set_title('Original Image')

subplot = plt.subplot(2, 2, 2)
plt.imshow(image_d50_n1.astype(np.uint8), vmin=0, vmax=255)
subplot.set_title('Butterworth D=50 n=1')

subplot = plt.subplot(2, 2, 3)
plt.imshow(image_d25_n1.astype(np.uint8), vmin=0, vmax=255)
subplot.set_title('Butterworth D=25 n=1')

subplot = plt.subplot(2, 2, 4)
plt.imshow(image_d25_n10.astype(np.uint8), vmin=0, vmax=255)
subplot.set_title('Butterworth D=25 n=10')

plt.show()
```



(6) **Adaptive Median Filter.**

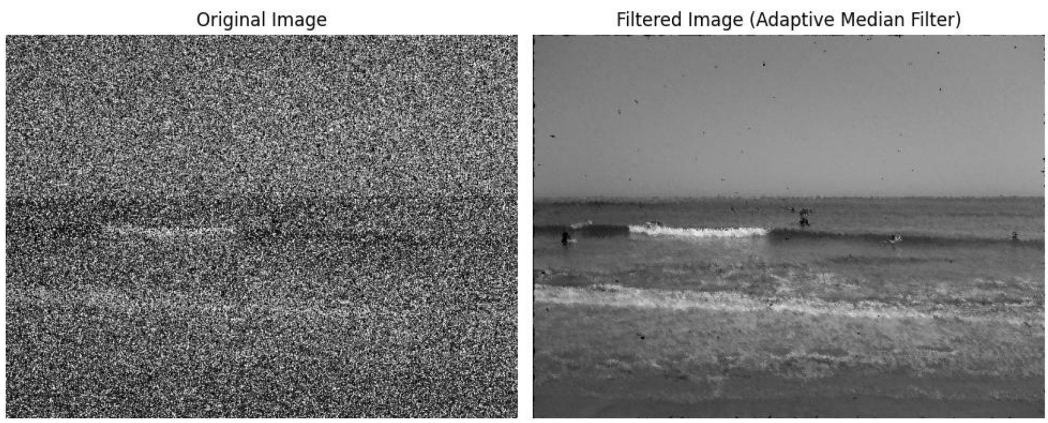
(a) Load the grayscale image noisy_salt_pepper.png.

(b) Use ADAPTIVE median filter for denoising (write code based on the implementation principle of adaptive median filter).

(c) Display the original image and the filtered images in the same figure with sub-figures. Add the corresponding title to the sub-figures.

```python
# Define the adaptive median filter function
def adaptive_median_filter(image, window_size_max):
    height, width = image.shape
    filtered_image = np.zeros_like(image)

    # Pad the image with zeros
    padded_image = np.pad(image, ((window_size_max//2, window_size_max//2), (window_size_max//2, window_size_max//2)), mode='constant')

    for i in range(height):
        for j in range(width):
            window_size = 3  # Initial window size
            while window_size <= window_size_max:
                window = padded_image[i:i+window_size, j:j+window_size]
                window_flattened = window.flatten()
                window_median = np.median(window_flattened)
                window_min = np.min(window_flattened)
                window_max = np.max(window_flattened)

                if window_min < window_median < window_max:
                    if window_min < image[i, j] < window_max:
                        filtered_image[i, j] = image[i, j]
                    else:
                        filtered_image[i, j] = window_median
                    break
                else:
                    window_size += 2  # Expand the window size
                    # Ensure window size does not exceed image boundaries
                    window_size = np.clip(window_size, 3, min(height, width))

            # If the window size exceeds the maximum window size, use the median of the entire window
            if window_size > window_size_max:
                filtered_image[i, j] = window_median

    return filtered_image
```

```python
# Display original and filtered images
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')

plt.subplot(1, 2, 2)
plt.imshow(filtered_image, cmap='gray')
plt.title('Filtered Image (Adaptive Median Filter)')

plt.tight_layout()
plt.show()
```



(7) **Motion Blur, Inverse filtering and Wiener filtering.**

(a) Load the RGB image lena.jpg.

(b) Apply motion blur to it.

(c) Recovering images by using inverse filtering and Wiener filtering, respectively. (Note that when using color images, pay attention to the axes parameters of functions such as fft, ifft, fftshift and ifftshift).

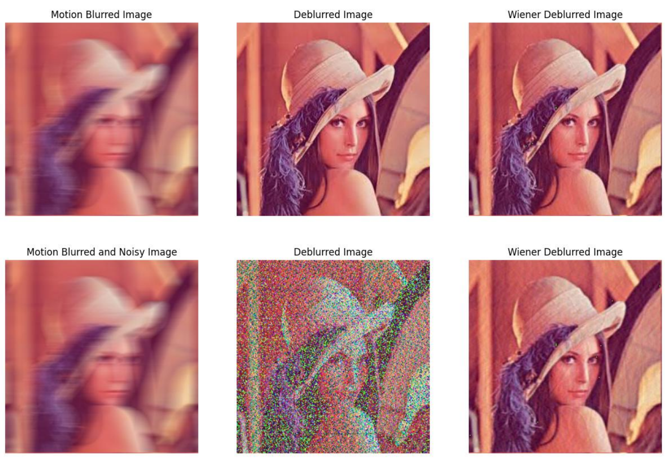
(d) Add noise to the blurred image, and then use Inverse filtering and Wiener filtering to recovere the image, respectively.

(e) Display them in the same figure with sub-figures. Add the corresponding title to the sub-figures.

```
def pad_with_zeros(vector, pad_width, iaxis, kwargs):
    vector[:pad_width[0]] = 0
    vector[-pad_width[1]:] = 0
    return vector

def motion_blur_kernel(kernel_size=15, angle=0):
    kernel = np.zeros((kernel_size, kernel_size))
    kernel[int((kernel_size-1)/2), :] = np.ones(kernel_size)
    M = cv2.getRotationMatrix2D((int(kernel_size/2), int(kernel_size / 2)), angle, 1)
    kernel = cv2.warpAffine(kernel, M, (kernel_size, kernel_size), flags=cv2.INTER_LINEAR)
    kernel = kernel / kernel.sum()
    return kernel
```

```
# Load an image
image = cv2.imread('images/lena.jpg')
size, angle, epsilon = 21, 0, 10**-6
kernel = motion_blur_kernel(kernel_size=size, angle=angle)
psf = np.copy(kernel)

# Frequency domain
kernel = np.pad(kernel, (((image.shape[0]-size)//2,(image.shape[0]-size)//2+1), ((image.shape[1]-size)//2,(image.shape[1]-size)//2+1)), pad_with_zeros)
kernel2 = np.zeros(image.shape)
kernel2[:, :, 0] = kernel
kernel2[:, :, 1] = kernel
kernel2[:, :, 2] = kernel

freq_shift = fp.fft2(image, axes=(0,1))
freq_kernel = fp.fft2(fp.ifftshift(kernel2, axes=(0,1)), axes=(0,1))
im_blur = fp.ifft2(freq_shift*freq_kernel, axes=(0,1)).real

im_blur_noise = im_blur + 0.01 * im_blur.std() * np.random.standard_normal(im_blur.shape)

freq = fp.fft2(im_blur, axes=(0,1))
freq_kernel2 =  1 / (epsilon + freq_kernel)
im_restored1 = fp.ifft2(freq*freq_kernel2, axes=(0,1)).real

im_restored2 = np.zeros(im_restored1.shape)
im_restored2[:,:,0] = restoration.unsupervised_wiener(im_blur[:,:,0]/255.0, psf)[0]
im_restored2[:,:,1] = restoration.unsupervised_wiener(im_blur[:,:,1]/255.0, psf)[0]
im_restored2[:,:,2] = restoration.unsupervised_wiener(im_blur[:,:,2]/255.0, psf)[0]

freq = fp.fft2(im_blur_noise, axes=(0,1))
im_restored3 = fp.ifft2(freq*freq_kernel2, axes=(0,1)).real
```

```
im_restored4 = np.zeros(im_restored1.shape)
im_restored4[:,:,0] = restoration.unsupervised_wiener(im_blur_noise[:,:,0]/255.0, psf)[0]
im_restored4[:,:,1] = restoration.unsupervised_wiener(im_blur_noise[:,:,1]/255.0, psf)[0]
im_restored4[:,:,2] = restoration.unsupervised_wiener(im_blur_noise[:,:,2]/255.0, psf)[0]

im_blur = im_blur.astype(np.uint8)
im_restored1 = im_restored1.astype(np.uint8)
im_restored2 = (im_restored2*255).astype(np.uint8)
im_blur_noise = im_blur_noise.astype(np.uint8)
im_restored3 = im_restored3.astype(np.uint8)
im_restored4 = (im_restored4*255).astype(np.uint8)
```



(8) **Estimating Noise Parameters. (Bonus Task)**

(a) Load the grayscale images noisy_1.png and noisy_2.png respectively.

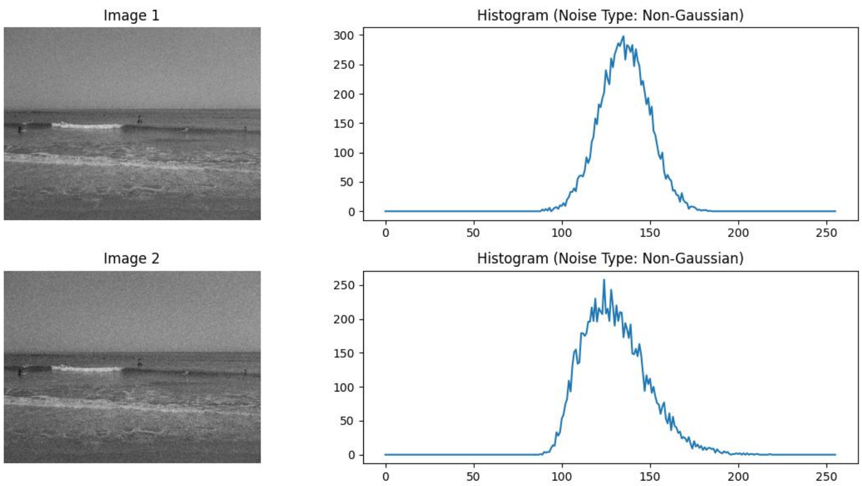
(b) Select a smooth region in each image and compute the histogram to estimate the noise distribution. Please specify the noise type.

(c) Display the image and the histograms in sub-figures. Add the corresponding title.

(d) Use moment estimation to estimate the noise parameters.

```python
# (b) Compute histograms to estimate noise distribution
smooth_region1 = image1[100:200, 100:200]
smooth_region2 = image2[100:200, 100:200]

hist1 = cv2.calcHist([smooth_region1], [0], None, [256], [0, 256])
hist2 = cv2.calcHist([smooth_region2], [0], None, [256], [0, 256])
```
✓ 0.0s

```python
# Determine noise type by examining the histograms
# If the histogram has a Gaussian-like shape, it indicates Gaussian noise. Otherwise, it may indicate other types of noise.
def estimate_noise_type(hist):
    if np.argmax(hist) > 200:
        return "Gaussian"
    else:
        return "Non-Gaussian"

noise_type1 = estimate_noise_type(hist1)
noise_type2 = estimate_noise_type(hist2)
```




```python
# (d) Moment estimation to estimate noise parameters
# Assuming Gaussian noise, estimate mean and standard deviation using image moments
def estimate_gaussian_noise_parameters(image):
    mean = np.mean(image)
    variance = np.var(image)
    std_dev = np.sqrt(variance)
    return mean, std_dev
```
✓ 0.0s

```python
mean1, std_dev1 = estimate_gaussian_noise_parameters(smooth_region1)
mean2, std_dev2 = estimate_gaussian_noise_parameters(smooth_region2)
```
✓ 0.0s

```python
print("Estimated Noise Parameters for Image 1 (Assuming Gaussian Noise):")
print(f"Mean: {mean1}, Standard Deviation: {std_dev1}")

print("Estimated Noise Parameters for Image 2 (Assuming Gaussian Noise):")
print(f"Mean: {mean2}, Standard Deviation: {std_dev2}")
```
✓ 0.0s

```
Estimated Noise Parameters for Image 1 (Assuming Gaussian Noise):
Mean: 135.5295, Standard Deviation: 13.880076719888834
Estimated Noise Parameters for Image 2 (Assuming Gaussian Noise):
Mean: 130.2961, Standard Deviation: 18.137266188430935
```

实验分析与结论（Analysis and Conclusion）：

In these experiments, we learned basic image processing techniques, particularly frequency domain filtering and image restoration algorithms. By performing Discrete Fourier Transform (DFT) and Inverse DFT (IDFT), we can analyze and process images in the frequency domain.

(1) FFT and IFFT: We learned how to use FFT and IFFT to analyze and reconstruct the frequency information of images.

(2) Ideal Low Pass Filtering: By designing an ideal low pass filter, we can remove high frequency components to achieve image smoothing. However, this method may result in ringing artifacts.

(3) Gaussian Low Pass Filter: Compared to the ideal low pass filter, Gaussian low pass filtering provides a smoother frequency response and reduces the occurrence of ringing artifacts.

(4) Butterworth Low Pass Filter: Compared to Gaussian filtering, Butterworth low pass filters offer more flexible frequency selectivity, allowing adjustment of cutoff frequency and order based on specific requirements.

(5) Butterworth High Pass Filter: Similar to low pass filters, Butterworth high pass filters enhance high frequency details of images by adjusting cutoff frequency and controlling frequency selectivity.

(6) Adaptive Median Filter: An effective denoising method, the adaptive median filter dynamically adjusts filter size based on local image characteristics.

(7) Motion Blur, Inverse Filtering, and Wiener Filtering: These experiments demonstrate common methods for handling blurred and noisy images. Inverse filtering and Wiener filtering are commonly used image restoration techniques that can mitigate the effects of blur and noise to some extent.

(8) Noise Parameter Estimation: By analyzing the histogram of images, we can estimate parameters of noise distribution, which aids in subsequent image processing and enhancement.

Experimental Conclusion

Through these experiments, we gained a deeper understanding of frequency domain filtering and image restoration techniques in image processing. Specific conclusions include:

- Frequency domain filtering techniques can effectively process images, including noise removal, image smoothing, and detail enhancement.
- Different types of filters have different characteristics and applications when processing images, and selection should be based on specific requirements.
- Image restoration techniques can partially compensate for image quality degradation due to blur and noise, but excessive processing may introduce additional artifacts or distortion.
- Noise parameter estimation is an important preprocessing step that helps us choose appropriate denoising methods and optimize image processing effects.
- In practical applications, we need to select appropriate methods and parameters based on specific image processing tasks and requirements to achieve optimal processing results.

指导教师批阅意见：

成绩评定：

实验态度 10 分	实验步骤及代码 40 分	实验数据与结果 40 分	实验分析与结论 10 分

指导教师签字：李斌

2024 年 5 月 10 日

备注：