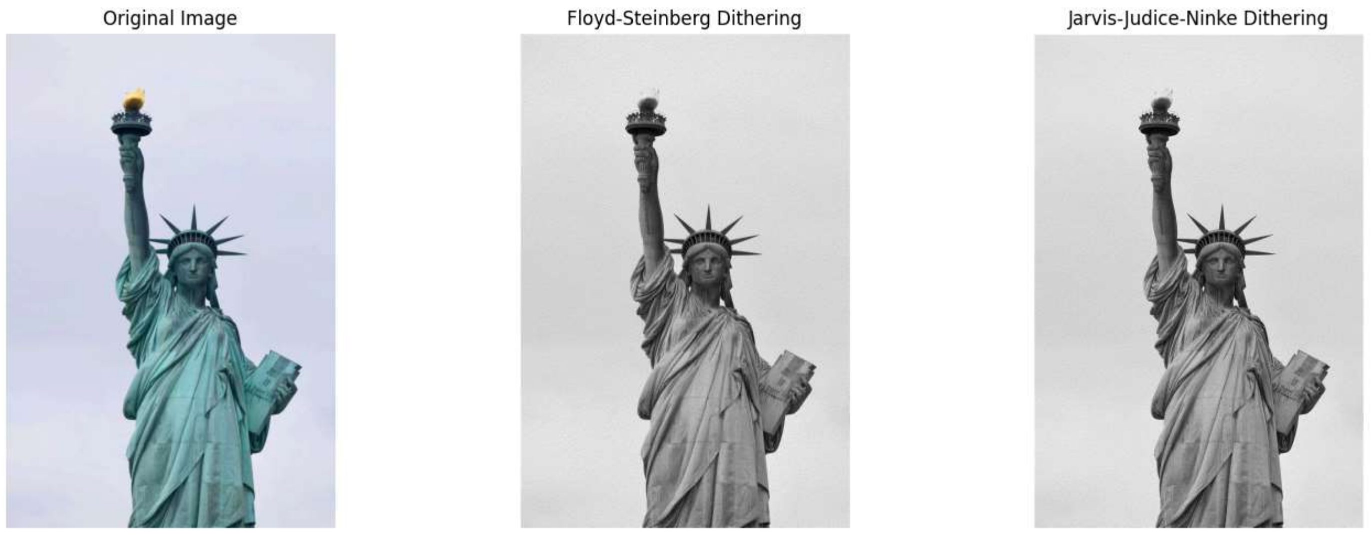
2. *Implement both the Floyd-Steinberg and Jarvis-Judice-Ninke dithering algorithms onthe image in either Python or MATLAB, then compare the results obtained fromeach method*

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

def load_image(file_path):
    return Image.open(file_path)

def quantize(value):
    return 255 if value > 127 else 0

def floyd_steinberg_dithering(image):
    img = image.convert('L')
    pixels = np.array(img, dtype=float)
    height, width = pixels.shape

    for y in range(height):
        for x in range(width):
            old_pixel = pixels[y, x]
            new_pixel = quantize(old_pixel)
            pixels[y, x] = new_pixel
            error = old_pixel - new_pixel

            if x + 1 < width:
                pixels[y, x + 1] += error * 7/16
            if y + 1 < height:
                if x > 0:
                    pixels[y + 1, x - 1] += error * 3/16
                pixels[y + 1, x] += error * 5/16
                if x + 1 < width:
                    pixels[y + 1, x + 1] += error * 1/16

    return Image.fromarray(pixels.astype(np.uint8))

def jarvis_judice_ninke_dithering(image):
    img = image.convert('L')
    pixels = np.array(img, dtype=float)
    height, width = pixels.shape

    for y in range(height):
        for x in range(width):
            old_pixel = pixels[y, x]
            new_pixel = quantize(old_pixel)
            pixels[y, x] = new_pixel
            error = old_pixel - new_pixel

            if x + 1 < width:
                pixels[y, x + 1] += error * 7/48
            if x + 2 < width:
                pixels[y, x + 2] += error * 5/48

            if y + 1 < height:
                if x - 2 >= 0:
                    pixels[y + 1, x - 2] += error * 3/48
                if x - 1 >= 0:
                    pixels[y + 1, x - 1] += error * 5/48
                pixels[y + 1, x] += error * 7/48
                pixels[y + 1, x] += error * 7/48
                if x + 1 < width:
                    pixels[y + 1, x + 1] += error * 5/48
                if x + 2 < width:
                    pixels[y + 1, x + 2] += error * 3/48

            if y + 2 < height:
                if x - 2 >= 0:
                    pixels[y + 2, x - 2] += error * 1/48
                if x - 1 >= 0:
                    pixels[y + 2, x - 1] += error * 3/48
                pixels[y + 2, x] += error * 5/48
                if x + 1 < width:
                    pixels[y + 2, x + 1] += error * 3/48
                if x + 2 < width:
                    pixels[y + 2, x + 2] += error * 1/48

    return Image.fromarray(pixels.astype(np.uint8))

# Load Image
image_path = "/content/liberty.jpeg"
original_image = load_image(image_path)

# Apply dithering algorithms
floyd_steinberg_result = floyd_steinberg_dithering(original_image)
jarvis_judice_ninke_result = jarvis_judice_ninke_dithering(original_image)

# Display results
plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)
plt.imshow(original_image, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(floyd_steinberg_result, cmap='gray')
plt.title('Floyd-Steinberg Dithering')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(jarvis_judice_ninke_result, cmap='gray')
plt.title('Jarvis-Judice-Ninke Dithering')
plt.axis('off')

plt.tight_layout()
plt.show()

print("Dithering complete. Results displayed above.")
```

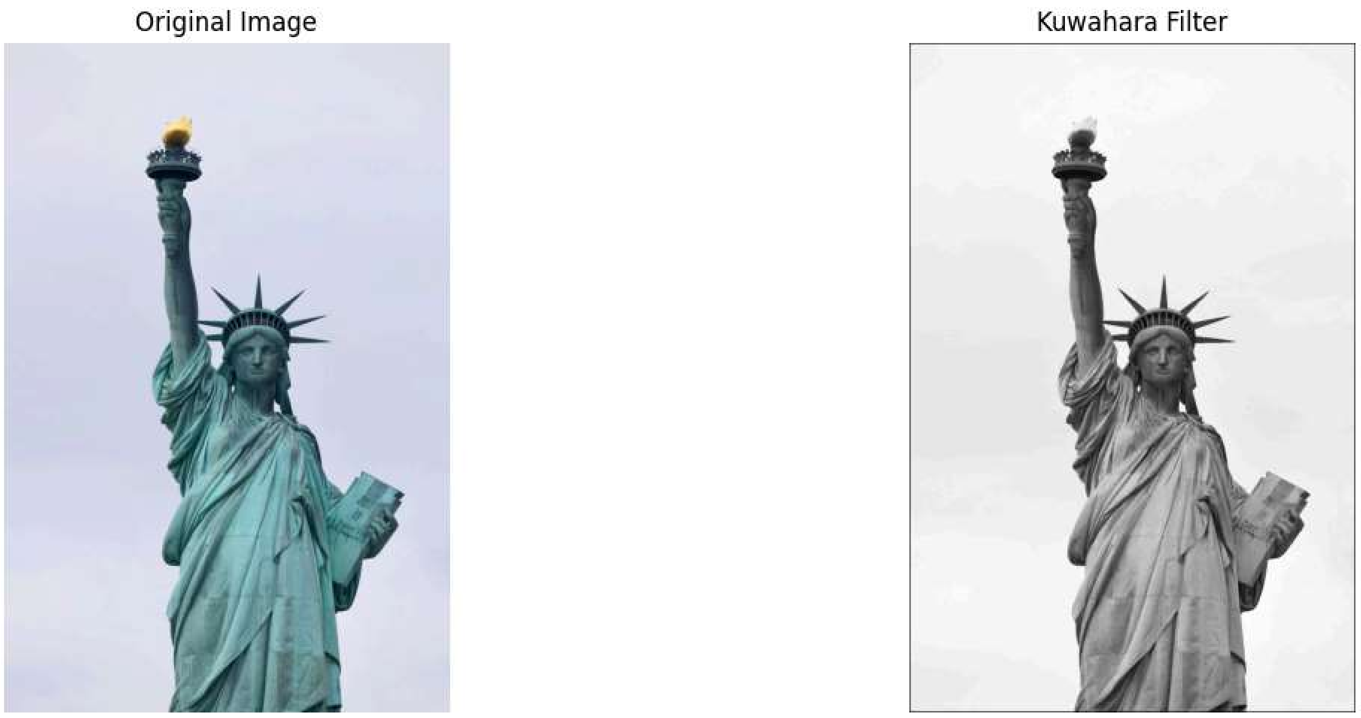


```
Dithering complete. Results displayed above.
```

3. *Explain what a Kuwahara filter is, and apply it to the image using either Python or MATLAB to demonstrate its effects.*

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

def kuwahara_filter(image, window_size):
    if len(image.shape) == 3:
        image = image.mean(axis=2)

    height, width = image.shape
    offset = window_size // 2
    filtered = np.zeros_like(image)

    for y in range(offset, height - offset):
        for x in range(offset, width - offset):
            windows = [
                image[y-offset:y+1, x-offset:x+1],
                image[y-offset:y+1, x:x+offset+1],
                image[y:y+offset+1, x-offset:x+1],
                image[y:y+offset+1, x:x+offset+1]
            ]

            means = [np.mean(window) for window in windows]
            variances = [np.var(window) for window in windows]
            filtered[y, x] = means[np.argmin(variances)]

    return filtered

# Load the image and convert it to a NumPy array
image_path = "/content/liberty.jpeg"
original_image = np.array(Image.open(image_path))

# Apply Kuwahara filter
window_size = 5
kuwahara_image = kuwahara_filter(original_image, window_size)

# Display results
```

```
    plt.figure(figsize=(15, 5))

    plt.subplot(1, 2, 1)
    plt.imshow(original_image)
    plt.title('Original Image')
    plt.axis('off')

    plt.subplot(1, 2, 2)
    plt.imshow(kuwahara_image, cmap='gray')
    plt.title('Kuwahara Filter')
    plt.axis('off')

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



The Kuwahara filter is an edge-preserving smoothing filter that's particularly useful in image processing and computer vision. It works by dividing the area around each pixel into four overlapping windows, calculating the mean and variance of each window, and then replacing the central pixel with the mean of the window that has the smallest variance. This approach helps to reduce noise while preserving edges, which is why it's often used in medical imaging and artistic photo effects. The key characteristics of the Kuwahara filter are:

- It smooths out textures in relatively uniform areas of the image.
- It preserves and even enhances edges between different regions.
- It can create a painterly or watercolor-like effect when applied to photographs.

The filter's behavior makes it particularly good at reducing noise in images while maintaining important structural features. However, it can also create a somewhat cartoonish or painted look, which is why it's sometimes used for artistic effects in photography.

4. *Take any image and apply the Fourier Transform to this image and the followingfilters:(Python or MATLAB)*

- (b) Butterworth filters
- (c) Gaussian filtersList item

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

def load_image(path):
    return np.array(Image.open(path).convert('L'))

def fourier_transform(image):
    f = np.fft.fft2(image)
    return np.fft.fftshift(f)

def butterworth_filter(shape, d0, n):
    rows, cols = shape
    x = np.linspace(-0.5, 0.5, cols) * cols
    y = np.linspace(-0.5, 0.5, rows) * rows
    radius = np.sqrt((x**2)[np.newaxis] + (y**2)[:, np.newaxis])
    return 1 / (1 + (radius / d0)**(2*n))

def gaussian_filter(shape, sigma):
    rows, cols = shape
    x = np.linspace(-0.5, 0.5, cols) * cols
    y = np.linspace(-0.5, 0.5, rows) * rows
    radius = np.sqrt((x**2)[np.newaxis] + (y**2)[:, np.newaxis])
    return np.exp(-(radius**2) / (2 * sigma**2))

def apply_filter(f_transform, filter_func):
    return np.fft.ifft2(np.fft.ifftshift(f_transform * filter_func))

def plot_results(original, filtered_butterworth, filtered_gaussian):
    fig, axs = plt.subplots(1, 3, figsize=(15, 5))
    axs[0].imshow(original, cmap='gray')
    axs[0].set_title('Original Image')
    axs[0].axis('off')

    axs[1].imshow(np.abs(filtered_butterworth), cmap='gray')
    axs[1].set_title('Butterworth Filter')
    axs[1].axis('off')

    axs[2].imshow(np.abs(filtered_gaussian), cmap='gray')
    axs[2].set_title('Gaussian Filter')
    axs[2].axis('off')

    plt.tight_layout()
    plt.show()

# Main execution
image_path = "/content/liberty.jpeg"
original_image = load_image(image_path)

# Apply Fourier Transform
f_transform = fourier_transform(original_image)

# Apply Butterworth filter
butterworth = butterworth_filter(original_image.shape, d0=30, n=2)
filtered_butterworth = apply_filter(f_transform, butterworth)
```

```
# Apply Gaussian filter
gaussian = gaussian_filter(original_image.shape, sigma=30)
filtered_gaussian = apply_filter(f_transform, gaussian)

# Plot results
plot_results(original_image, filtered_butterworth, filtered_gaussian)
```