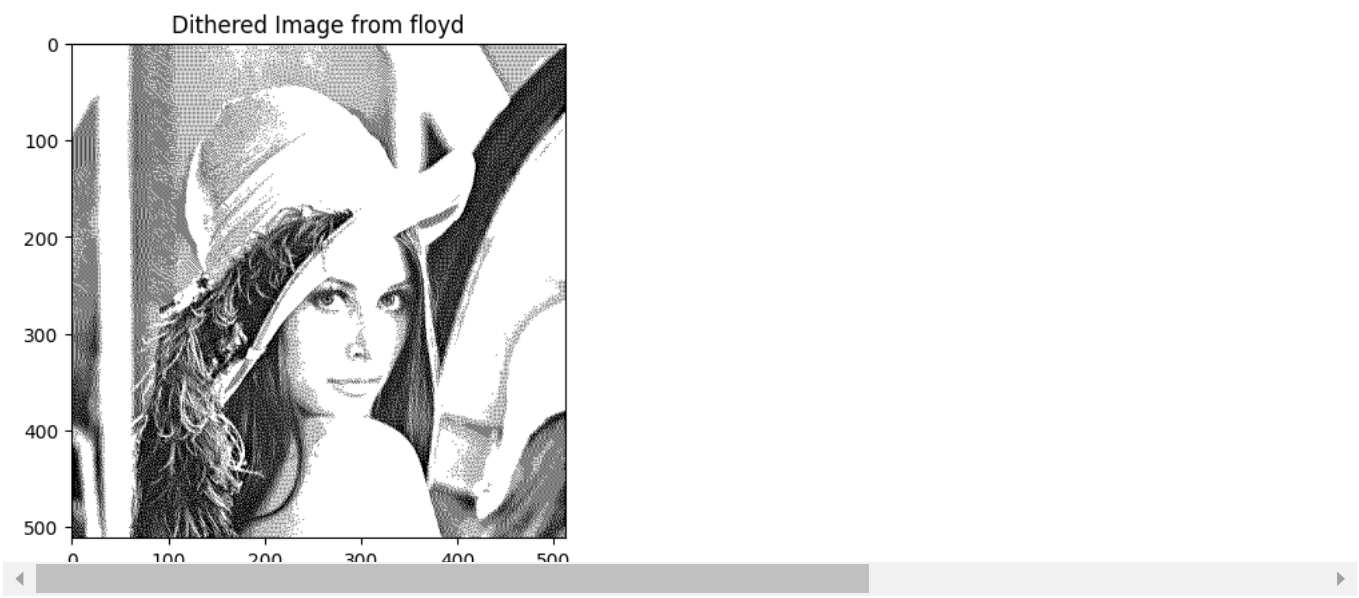
2. Implement both the Floyd-Steinberg and Jarvis-Judice-Ninke dithering algorithms on the image in either Python or MATLAB, then compare the results obtained from each method.

```
import cv2
import numpy as np

class ditherModule(object):
  def dither(self, img, method='floyd-steinberg', resize=False):
    # Resize the image if specified
    if resize:
      img = cv2.resize(img, (int(0.5 * (np.shape(img)[1])), int(0.5 * (np.shape(img)[0]))))

    if method == 'floyd-steinberg':
      # Floyd-Steinberg dithering
      img = cv2.copyMakeBorder(img, 1, 1, 1, 1, cv2.BORDER_REPLICATE)
      rows, cols = np.shape(img)
      out = cv2.normalize(img.astype('float'), None, 0.0, 1.0, cv2.NORM_MINMAX)
      for i in range(1, rows - 1):
        for j in range(1, cols - 1):
          # Threshold step
          if out[i][j] > 0.5:
            err = out[i][j] - 1
            out[i][j] = 1
          else:
            err = out[i][j]
            out[i][j] = 0

            # Error diffusion step for Floyd-Steinberg
            out[i][j + 1] += (7 / 16) * err
            out[i + 1][j - 1] += (3 / 16) * err
            out[i + 1][j] += (5 / 16) * err
            out[i + 1][j + 1] += (1 / 16) * err

      return out[1:rows - 1, 1:cols - 1]

    elif method == 'jarvis-judice-ninke':
      # Jarvis-Judice-Ninke dithering
      img = cv2.copyMakeBorder(img, 2, 2, 2, 2, cv2.BORDER_REPLICATE)
      rows, cols = np.shape(img)
      out = cv2.normalize(img.astype('float'), None, 0.0, 1.0, cv2.NORM_MINMAX)
      for i in range(2, rows - 2):
        for j in range(2, cols - 2):
          # Threshold step
          if out[i][j] > 0.5:
            err = out[i][j] - 1
            out[i][j] = 1
          else:
            err = out[i][j]
            out[i][j] = 0

            # Error diffusion step for Jarvis-Judice-Ninke
            out[i][j + 1] += (7 / 48) * err
            out[i][j + 2] += (5 / 48) * err
            out[i + 1][j - 2] += (3 / 48) * err
            out[i + 1][j - 1] += (5 / 48) * err
            out[i + 1][j] += (7 / 48) * err
            out[i + 1][j + 1] += (5 / 48) * err
            out[i + 1][j + 2] += (3 / 48) * err
            out[i + 2][j - 2] += (1 / 48) * err
            out[i + 2][j - 1] += (3 / 48) * err
            out[i + 2][j] += (5 / 48) * err
            out[i + 2][j + 1] += (3 / 48) * err
            out[i + 2][j + 2] += (1 / 48) * err

      return out[2:rows - 2, 2:cols - 2]

    else:
      raise TypeError('Invalid method. Available methods: "floyd-steinberg", "jarvis-judice-ninke"')


def dither(img, method='floyd-steinberg', resize=False):
  dither_object = ditherModule()
  return dither_object.dither(img, method, resize)


img = cv2.imread('/content/lena.png', cv2.IMREAD_GRAYSCALE)
dithered_img_floyd = dither(img, method='floyd-steinberg')  # Or use 'jarvis-judice-ninke'
```

```
dithered_img_floyd
```

```
array([[1., 1., 1., ..., 1., 1., 0.],
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 0.],
       ...,
       [0., 0., 0., ..., 1., 0., 1.],
       [0., 0., 0., ..., 1., 1., 0.],
       [0., 0., 0., ..., 1., 0., 1.]])
```

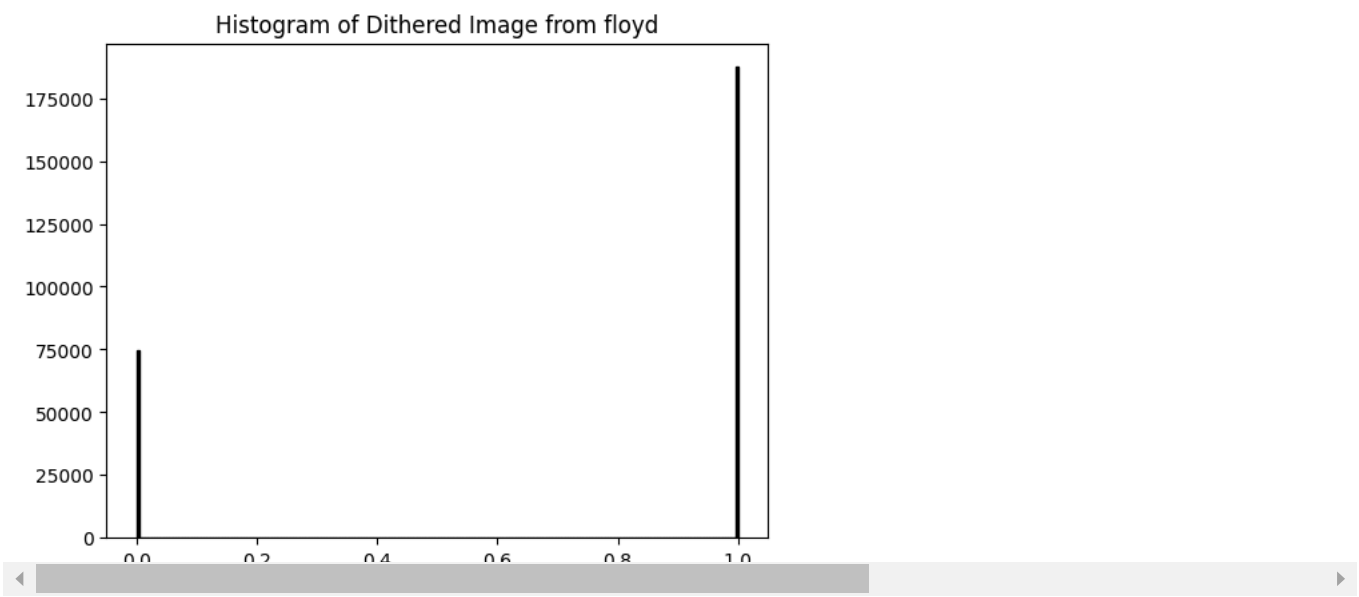
```
import matplotlib.pyplot as plt

# Display the dithered image
plt.imshow(dithered_img_floyd, cmap='gray')
plt.title('Dithered Image from floyd')
plt.show()
```




```
import cv2

# Save the dithered image
cv2.imwrite('dithered_img_floyd.jpg', dithered_img_floyd * 255)  # Multiply by 255 to scale values from 0 to 255
```

```
True
```

```
plt.hist(dithered_img_floyd.ravel(), bins=256, range=[0, 1], fc='k', ec='k')
plt.title('Histogram of Dithered Image from floyd')
plt.show()
```



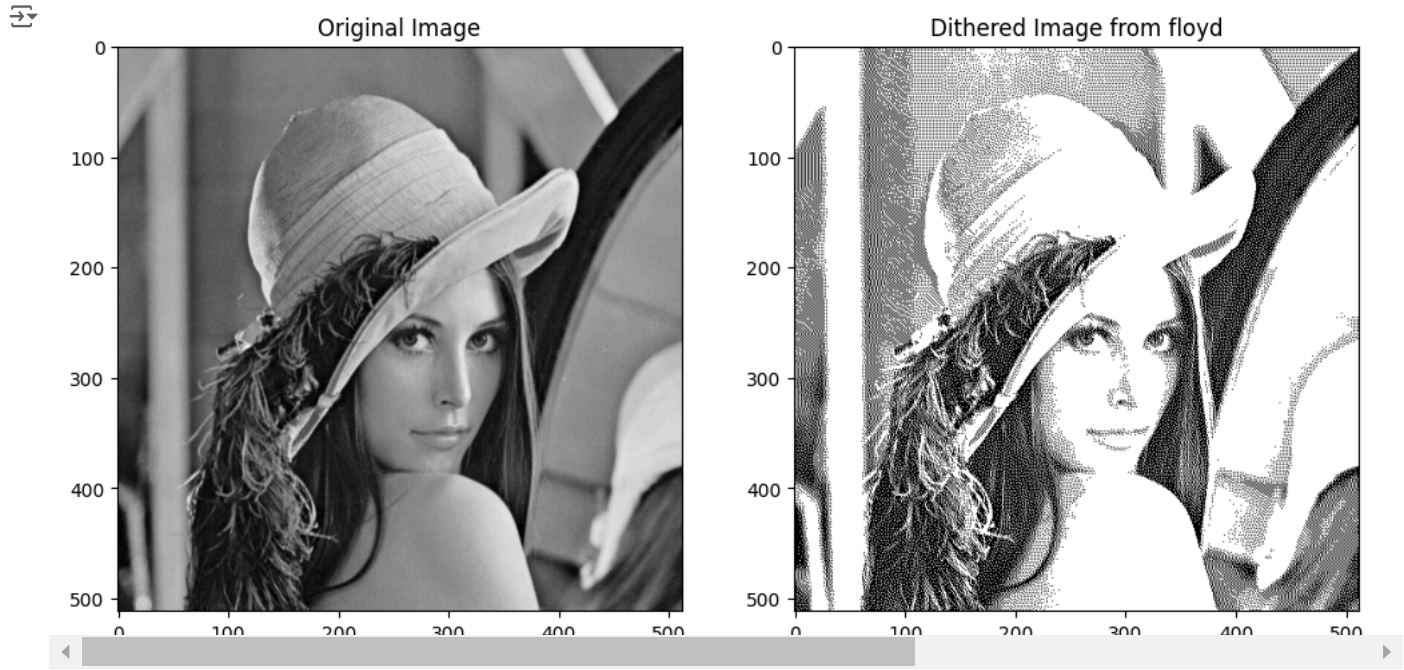

```
print("Original Image Shape:", img.shape)
print("Dithered Image floyd Shape:", dithered_img_floyd.shape)
```

```
Original Image Shape: (512, 512)
Dithered Image floyd Shape: (512, 512)
```

```
print("Unique pixel values in the dithered image from floyd:", np.unique(dithered_img_floyd))
```

```
Unique pixel values in the dithered image from floyd: [0. 1.]
```

```
fig, axs = plt.subplots(1, 2, figsize=(12, 6))

# Display original image
axs[0].imshow(img, cmap='gray')
axs[0].set_title('Original Image')

# Display dithered image
axs[1].imshow(dithered_img_floyd, cmap='gray')
axs[1].set_title('Dithered Image from floyd')

plt.show()
```



COMPARISON:

Floyd-Steinberg Dithering Output:

Appearance: A Floyd-Steinberg dithered image is a black-and-white image where grayscale values are represented by patterns of black and white pixels. The method spreads the difference (error) from rounding pixel values to nearby pixels, creating a smooth approximation of the original image.

Error Diffusion: The algorithm spreads the error to nearby pixels in a specific pattern, affecting 4 surrounding pixels, creating a halftone effect.

Pixel Patterns: You will notice pixel patterns, especially at the edges of light and dark areas. Medium intensity regions may look grainy or dotted. Lighter areas have fewer black dots, while darker areas show more black dots.

Quality: Floyd-Steinberg dithering creates high-quality black-and-white images with smooth transitions between dark and light areas, making it a popular method for image dithering.

Example Analysis: Smooth Transitions: The Floyd-Steinberg dithering creates smoother transitions between light and dark areas, even in a binary image, by spreading the error from each pixel to its neighbors. This makes the changes between light and dark feel less harsh.

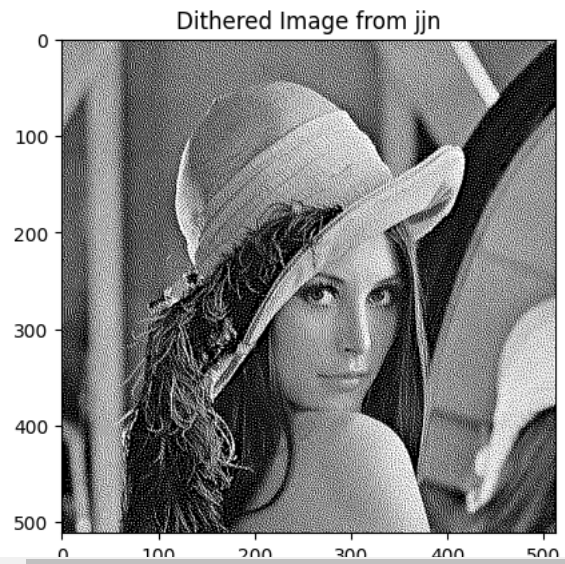
Detail Preservation: Fine details in the image, like edges and textures, will still be visible but in a simpler, less detailed form with black-and-white pixels.

```
Start coding or generate with AI.
```

```
img = cv2.imread('/content/lena.png', cv2.IMREAD_GRAYSCALE)
dithered_img_jjn = dither(img, method='jarvis-judice-ninke')  # Or use 'jarvis-judice-ninke'
```

```
import matplotlib.pyplot as plt

# Display the dithered image
```

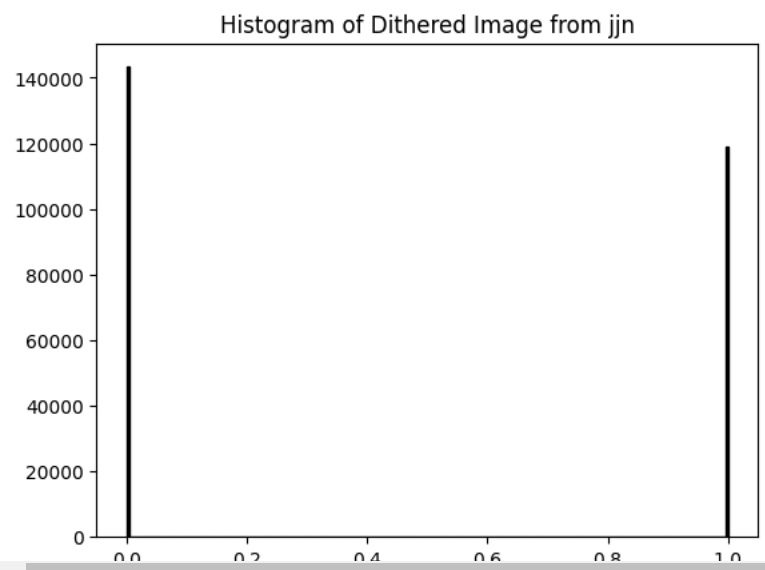
```
plt.imshow(dithered_img_jjn, cmap='gray')
plt.title('Dithered Image from jjn')
plt.show()
```




```
import cv2

# Save the dithered image
cv2.imwrite('dithered_img_jjn.jpg', dithered_img_jjn * 255)  # Multiply by 255 to scale values from 0 to 255
```

```
True
```

```
plt.hist(dithered_img_jjn.ravel(), bins=256, range=[0, 1], fc='k', ec='k')
plt.title('Histogram of Dithered Image from jjn')
plt.show()
```




```
print("Original Image Shape:", img.shape)
print("Dithered Image jjn Shape:", dithered_img_jjn.shape)
```

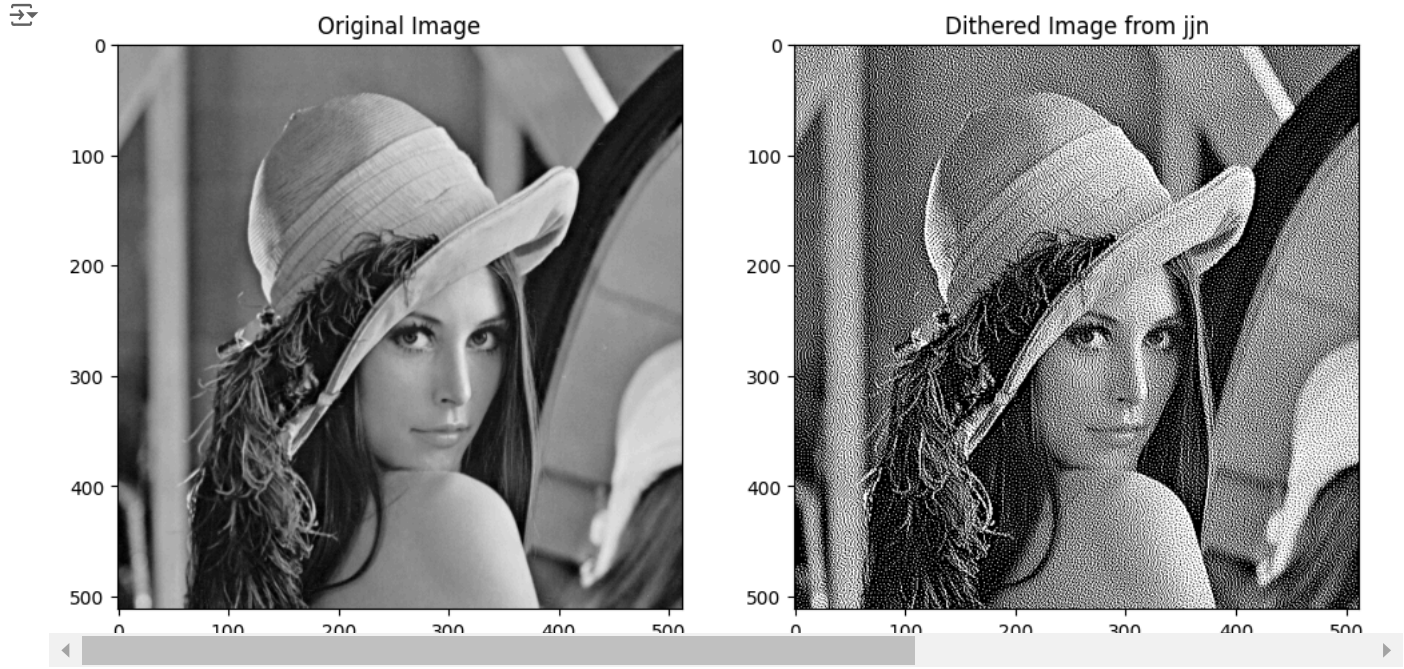
```
Original Image Shape: (512, 512)
Dithered Image jjn Shape: (512, 512)
```

```
print("Unique pixel values in the dithered image from jjn:", np.unique(dithered_img_jjn))
```

```
Unique pixel values in the dithered image from jjn: [0. 1.]
```

```
fig, axs = plt.subplots(1, 2, figsize=(12, 6))

# Display original image
axs[0].imshow(img, cmap='gray')
axs[0].set_title('Original Image')
```

```
# Display dithered image
axs[1].imshow(dithered_img_jjn, cmap='gray')
axs[1].set_title('Dithered Image from jjn')

plt.show()
```



COMPARISON:

Jarvis-Judice-Ninke (JJN) Dithering Output:

Appearance: The JJN dithering method creates a binary (black and white) image but gives smoother results than Floyd-Steinberg. It spreads the error across 12 nearby pixels instead of just 4, making transitions between light and dark areas look more gradual.

Error Diffusion: JJN dithering spreads the error over a larger area, making the image look smoother and less noisy compared to Floyd-Steinberg dithering, though it may appear slightly more blurred.

Pixel Patterns: In JJN dithered images, the black and white pixels are spread more evenly compared to Floyd-Steinberg, creating a finer, softer pattern. The black dots are less clustered, giving the image a smoother appearance.

Quality: JJN dithering creates a smoother look in areas with gradual changes in intensity, but it can be less sharp compared to Floyd-Steinberg due to its wider pixel spreading.

Example Analysis: Smoother Halftones: In a JJN dithered image, mid-range tones look smoother and more balanced. The larger diffusion pattern helps create softer transitions between light and dark, making it great for images with gradual shading.

Softening of Details: Fine details may look softer and less sharp due to the spread of errors, making the image smoother but slightly less detailed, especially in textured areas.

Start coding or generate with AI.