Lab Report: 01
Title: Take image and display it using MATLAB/Python

Course title: Digital Image Processing Laboratory
Course code: CSE-406
4th Year 1st Semester Examination 2023

Date of Submission: 01/09/2024

Submitted to-
Dr. Md. Golam Moazzam
Professor
Department of Computer Science and Engineering
Jahangirnagar University
&
Dr. Morium Akter
Professor
Department of Computer Science and Engineering
Jahangirnagar University
Savar, Dhaka-1342

Class Roll	Exam Roll	Name
353	202165	Shanjida Alam

Department of Computer Science and Engineering
Jahangirnagar University
Savar, Dhaka, Bangladesh

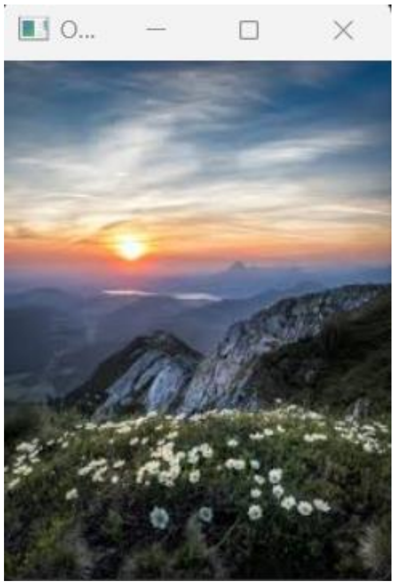
Objectives:

1. **Understanding basic image handling:** Learn how to load and manipulate image data in both MATLAB and Python.
2. **Basic image processing skills:** Acquire foundational skills that can be applied more complex image processing tasks like filtering, edge detection etc.

Code 01: Python

```
import cv2
img = cv2.imread("Image/nature.jpeg")
cv2.imshow("Output: ", img)
cv2.waitKey(0)
```

Output:



Code 02: MATLAB

```
A = imread('nature.jpeg');
image(A)
```

Output:

Lab Report: 02
Title: Take an image and convert it digital negative & Image enhancement in contrast stretching

Course title: Digital Image Processing Laboratory
Course code: CSE-406
4th Year 1st Semester Examination 2023

Date of Submission: 08/09/2024

Submitted to-
Dr. Md. Golam Moazzam
Professor
Department of Computer Science and Engineering
Jahangirnagar University
&
Dr. Morium Akter
Professor
Department of Computer Science and Engineering
Jahangirnagar University
Savar, Dhaka-1342

Class Roll	Exam Roll	Name
353	202165	Shanjida Alam

Department of Computer Science and Engineering
Jahangirnagar University
Savar, Dhaka, Bangladesh

Experiment Name: Take an image and convert it digital negative

Objectives:

1. Invert the colors of the image
2. Adjust the contrast and brightness
3. Add grain or texture

Code-01: Python

```
from PIL import Image, ImageOps
image_path = 'Image/nature.jpeg'
image = Image.open(image_path)
negative_image = ImageOps.invert(image)
negative_image.save('negative_image.jpg')
negative_image.show()
```

Output:

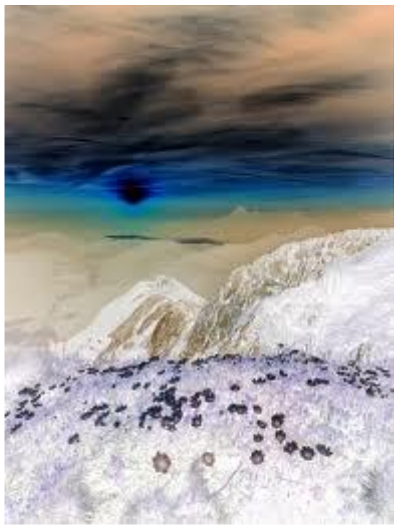


Figure 1.1: Showing the digital negative image using python code

Code-02: MATLAB

```
skI = imread("nature.jpeg");
subplot(1, 2, 1),
imshow(skI);
title("Original image");
L = 2 ^ 8;
neg = (L - 1) - skI;
subplot(1, 2, 2),
imshow(neg);
title("Negative Image")
```

Output:

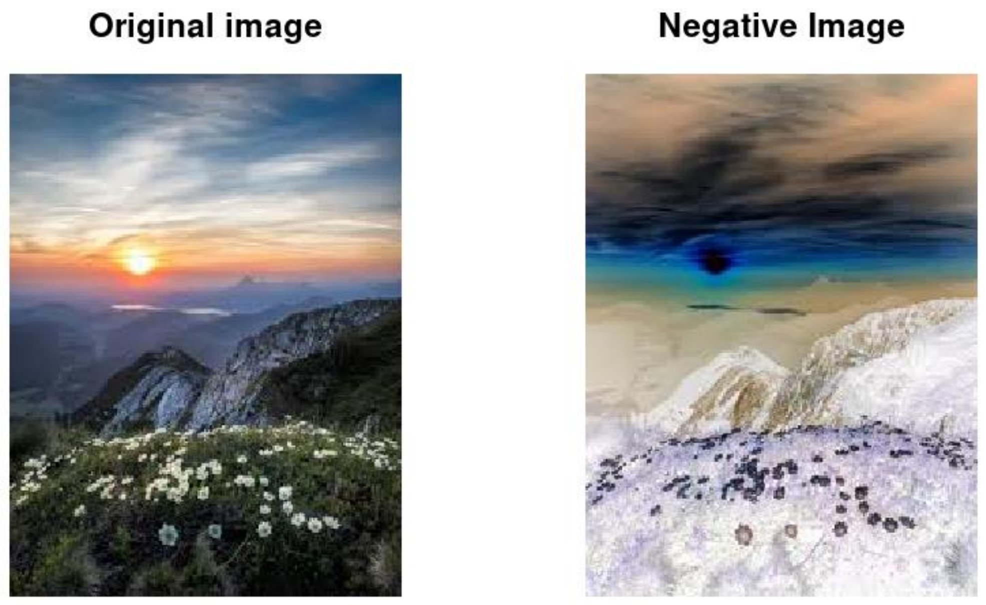


Figure 1.2: Showing the digital negative image using MATLAB

Experiment Name: Image enhancement in contrast stretching.

Objectives:

1. Make the dark areas of the image darker.
2. Make the bright areas of the image brighter

Code-01: Python

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
def contrast_stretching(image, m, E):
    normalized_image = np.array(image) / 255.0
    transformed_image = 1 / (1 + np.exp(-E * (normalized_image - m)))
    output_image = (transformed_image * 255).astype(np.uint8)
    return output_image
image_path = 'Image/nature.jpeg'
image = Image.open(image_path)
m = 0.5
E = 10
output_image = contrast_stretching(image, m, E)
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
axes[0].imshow(image)
axes[0].set_title('Original Image')
axes[0].axis('off')
axes[1].imshow(output_image)
axes[1].set_title('Contrast Stretched Image')
axes[1].axis('off')
plt.tight_layout()
plt.show()
```

Output:

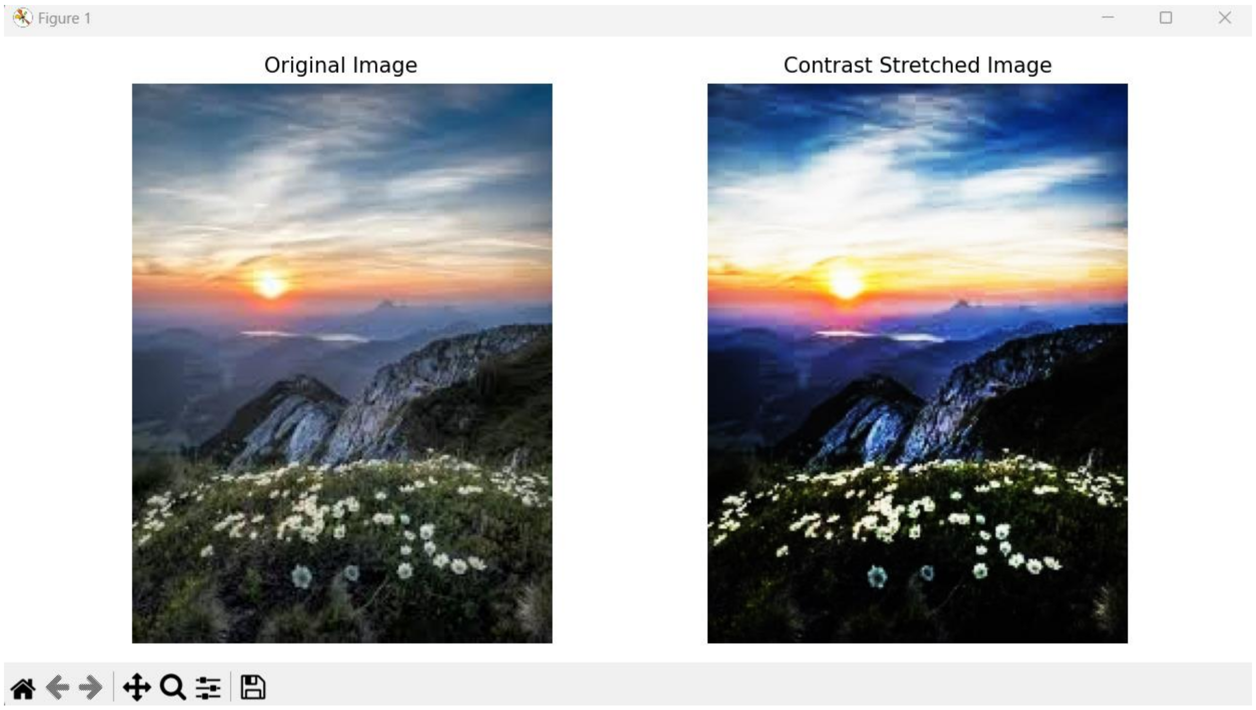


Figure 2.1: Showing the image enhancement in contrast stretching using python code

Code-02: MATLAB

```
image_path = 'nature.jpeg';
image = imread(image_path);
normalized_image = im2double(image);
m = 0.5;
E = 10;
transformed_image = 1 ./ (1 + exp(-E * (normalized_image - m)));
output_image = uint8(transformed_image * 255);
figure;
subplot(1, 2, 1);
imshow(image);
title('Original Image');
subplot(1, 2, 2);
imshow(output_image);
title('Contrast Stretched Image');
```

Output:

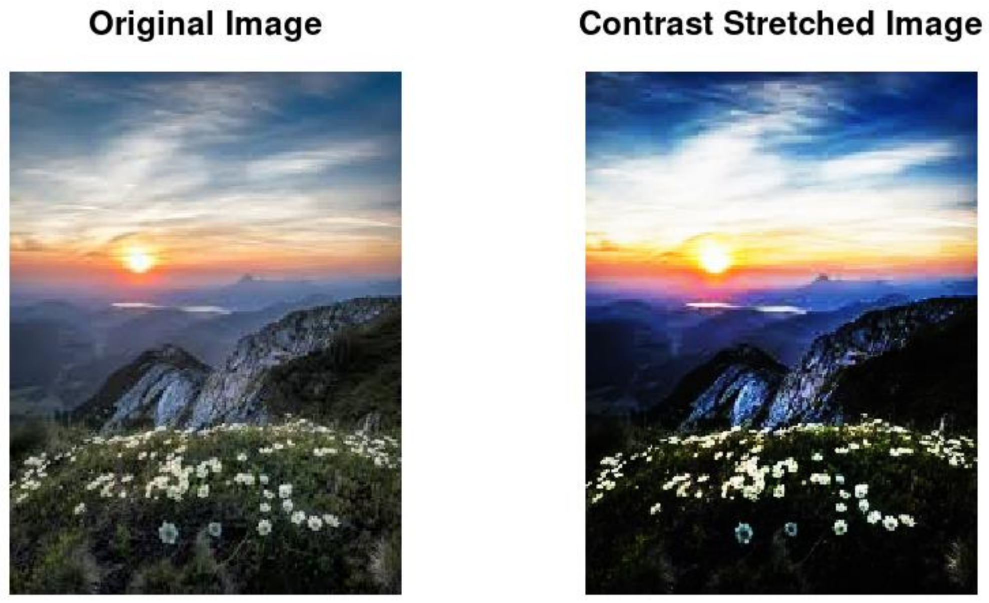


Figure 2.2: Showing the image enhancement in contrast stretching in MATLAB

Lab Report: 03
Title: Histogram Equalization, Cumulative Distribution Function (CDF) & Histogram Matching in Digital Image Processing

Course title: Digital Image Processing Laboratory
Course code: CSE-406
4th Year 1st Semester Examination 2023

Date of Submission: 15/09/2024

Submitted to-
Dr. Md. Golam Moazzam
Professor
Department of Computer Science and Engineering
Jahangirnagar University
&
Dr. Morium Akter
Professor
Department of Computer Science and Engineering
Jahangirnagar University
Savar, Dhaka-1342

Class Roll	Exam Roll	Name
353	202165	Shanjida Alam

Department of Computer Science and Engineering
Jahangirnagar University
Savar, Dhaka, Bangladesh

Experiment Name: Histogram Equalization

Objectives:

1. Improve Image Contrast
2. Enhance Image Details
3. Reduce Noise and Artifacts

Code-01: Python

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
input_image = cv2.imread('nature.jpeg', cv2.IMREAD_GRAYSCALE)
plt.figure(figsize=(10, 8))
plt.subplot(2, 2, 1)
plt.imshow(input_image, cmap='gray')
plt.title('Input Image')
plt.axis('off')
plt.subplot(2, 2, 2)
plt.hist(input_image.ravel(), bins=256, range=[0, 256])
plt.title('Histogram of Input Image')
equalized_image = cv2.equalizeHist(input_image)
plt.subplot(2, 2, 3)
plt.imshow(equalized_image, cmap='gray')
plt.title('Histogram-Equalized Image')
plt.axis('off')
plt.subplot(2, 2, 4)
plt.hist(equalized_image.ravel(), bins=256, range=[0, 256])
plt.title('Histogram of Equalized Image')
plt.tight_layout()
plt.show()
import cv2
import numpy as np
from matplotlib import pyplot as plt
input_image = cv2.imread('your_image.jpg', cv2.IMREAD_GRAYSCALE)
plt.figure(figsize=(10, 8))
plt.subplot(2, 2, 1)
plt.imshow(input_image, cmap='gray')
plt.title('Input Image')
plt.axis('off')
plt.subplot(2, 2, 2)
plt.hist(input_image.ravel(), bins=256, range=[0, 256])
plt.title('Histogram of Input Image')
equalized_image = cv2.equalizeHist(input_image)
plt.subplot(2, 2, 3)
plt.imshow(equalized_image, cmap='gray')
plt.title('Histogram-Equalized Image')
plt.axis('off')
plt.subplot(2, 2, 4)
plt.hist(equalized_image.ravel(), bins=256, range=[0, 256])
plt.title('Histogram of Equalized Image')
plt.tight_layout()
plt.show()
```

Output:

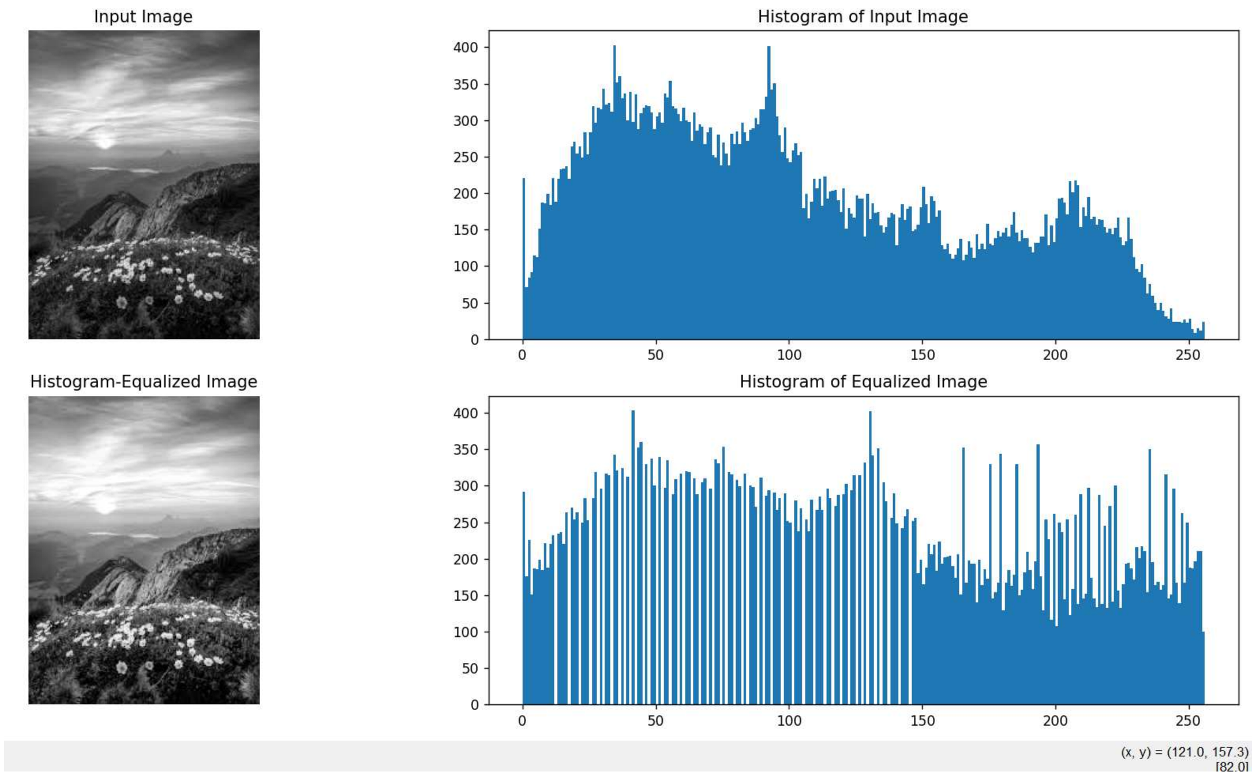


Figure 1.1: Showing the histogram equalization in python

Explanation:

1. Importing libraries
2. Reading the input image
3. Displaying the Input Image and Its Histogram
4. Performing Histogram Equalization
5. Displaying the Equalized Image and Its Histogram
6. Displaying the Final Plot

Code-02: MATLAB

```
 input_image = imread('flower.jpeg');
if size(input_image, 3) == 3
    input_image = rgb2gray(input_image);
end
figure;
subplot(2, 2, 1);
imshow(input_image);
title('Input Image');
subplot(2, 2, 2);
imhist(input_image);
title('Histogram of Input Image');
equalized_image = histeq(input_image);
subplot(2, 2, 3);
imshow(equalized_image);
title('Histogram-Equalized Image');
subplot(2, 2, 4);
imhist(equalized_image);
title('Histogram of Equalized Image');
```

Output:

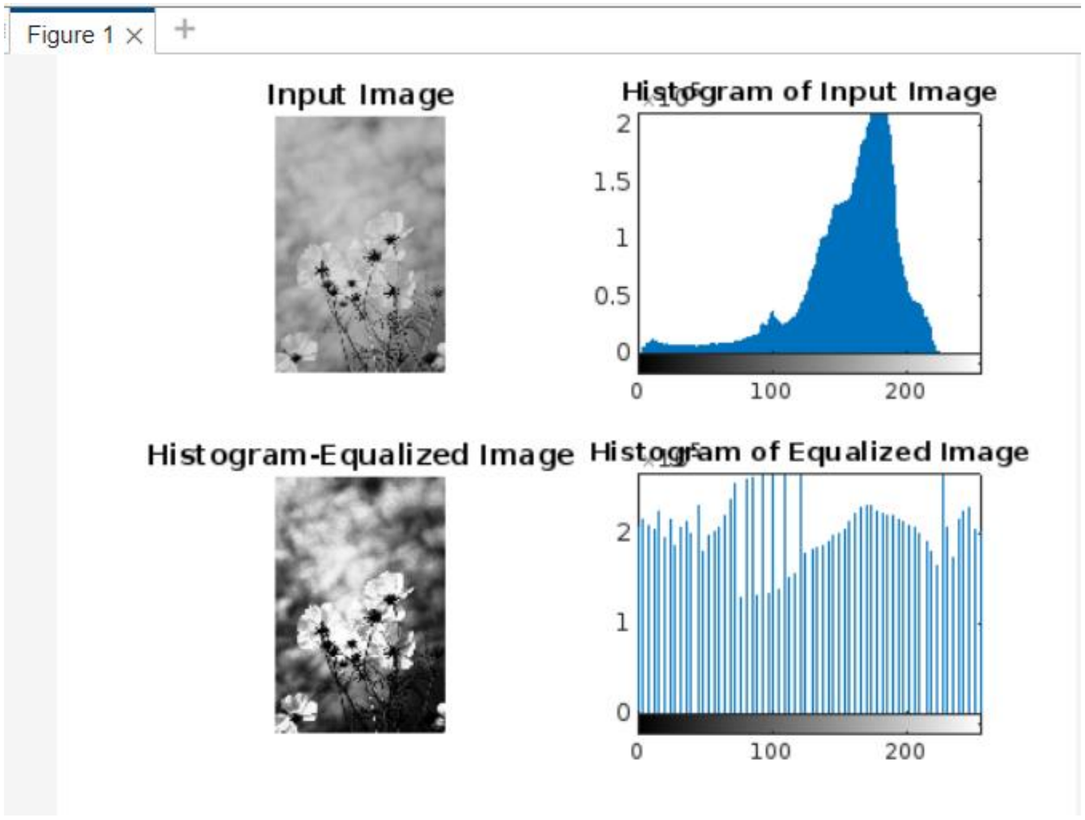


Figure 1.2: Showing the histogram equalization in MATLAB

Explanation:

1. Reading the Input Image
2. Converting the Image to Grayscale (If Necessary)
3. Displaying the Input Image and Its Histogram
4. The code uses histeq() to perform histogram equalization on the input image.
5. The code displays the equalized image and its histogram in the third and fourth subplots, respectively.

Experiment Name: Cumulative Distributed Function (CDF)

Objectives:

1. Image Normalization
2. Histogram Equalization
3. Image Segmentation

Code-01: Python

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
input_image = cv2.imread('nature.jpeg')
if len(input_image.shape) == 3:
   input_image = cv2.cvtColor(input_image, cv2.COLOR_BGR2GRAY)
def compute_cdf(image):
   hist, bins = np.histogram(image.flatten(), 256, [0, 256])
   cdf = hist.cumsum()
   cdf_normalized = cdf / cdf.max()
   return cdf_normalized
plt.figure(figsize=(12, 6))
plt.subplot(2, 3, 1)
plt.imshow(input_image, cmap='gray')
plt.title('Input Image')
plt.axis('off')
plt.subplot(2, 3, 2)
plt.hist(input_image.ravel(), bins=256, range=[0, 256], color='black')
plt.title('Histogram of Input Image')
cdf_input = compute_cdf(input_image)
plt.subplot(2, 3, 3)
plt.plot(cdf_input, color='black', linewidth=2)
plt.title('CDF of Input Image')
plt.xlabel('Pixel Intensity Values')
plt.ylabel('CDF')
equalized_image = cv2.equalizeHist(input_image)
plt.subplot(2, 3, 4)
plt.imshow(equalized_image, cmap='gray')
plt.title('Histogram-Equalized Image')
plt.axis('off')
plt.subplot(2, 3, 5)
plt.hist(equalized_image.ravel(), bins=256, range=[0, 256], color='black')
plt.title('Histogram of Equalized Image')
cdf_equalized = compute_cdf(equalized_image)
plt.subplot(2, 3, 6)
plt.plot(cdf_equalized, color='black', linewidth=2)
plt.title('CDF of Equalized Image')
plt.xlabel('Pixel Intensity Values')
plt.ylabel('CDF')
plt.tight_layout()
plt.show()
```

Output:

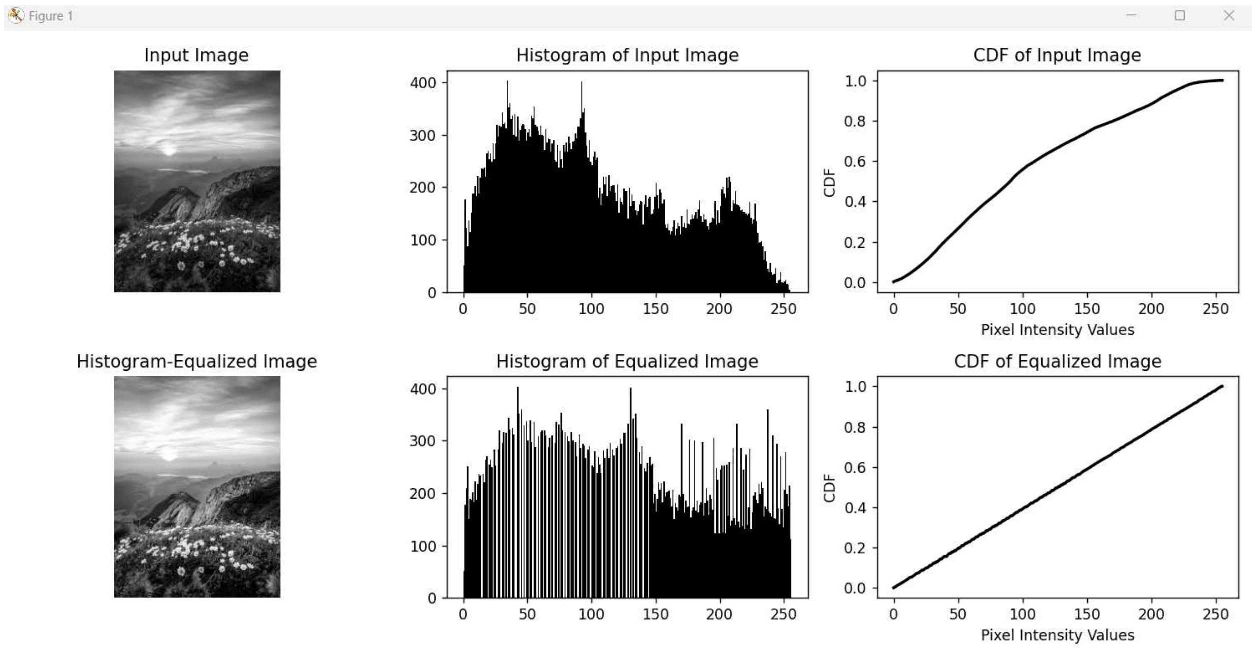


Figure 2.1: Showing the CDF using python code

Explanation:

1. Importing Libraries such as cv2, numpy, matplotlib.pyplot
2. Loading the Input Image
3. The code applies histogram equalization to the input image using OpenCV's equalizeHist function.
4. The code creates another three subplots
5. The code uses tight_layout to ensure the subplots fit nicely in the figure, and finally, it displays the figure using show.

Code-02: MATLAB

```
input_image = imread('flower.jpeg');
if size(input_image, 3) == 3
   input_image = rgb2gray(input_image);
end
figure;
subplot(2, 3, 1);
imshow(input_image);
title('Input Image');
subplot(2, 3, 2);
imhist(input_image);
title('Histogram of Input Image');
[counts, binLocations] = imhist(input_image);
cdf_input_image = cumsum(counts) / numel(input_image);
subplot(2, 3, 3);
plot(binLocations, cdf_input_image, 'LineWidth', 2);
title('CDF of Input Image');
xlabel('Pixel Intensity Values');
ylabel('CDF');
equalized_image = histeq(input_image);
subplot(2, 3, 4);
imshow(equalized_image);
title('Histogram-Equalized Image');
subplot(2, 3, 5);
imhist(equalized_image);
title('Histogram of Equalized Image');
[counts_eq, binLocations_eq] = imhist(equalized_image);
cdf_equalized_image = cumsum(counts_eq) / numel(equalized_image);
subplot(2, 3, 6);
plot(binLocations_eq, cdf_equalized_image, 'LineWidth', 2);
title('CDF of Equalized Image');
xlabel('Pixel Intensity Values');
ylabel('CDF');
```

Output:

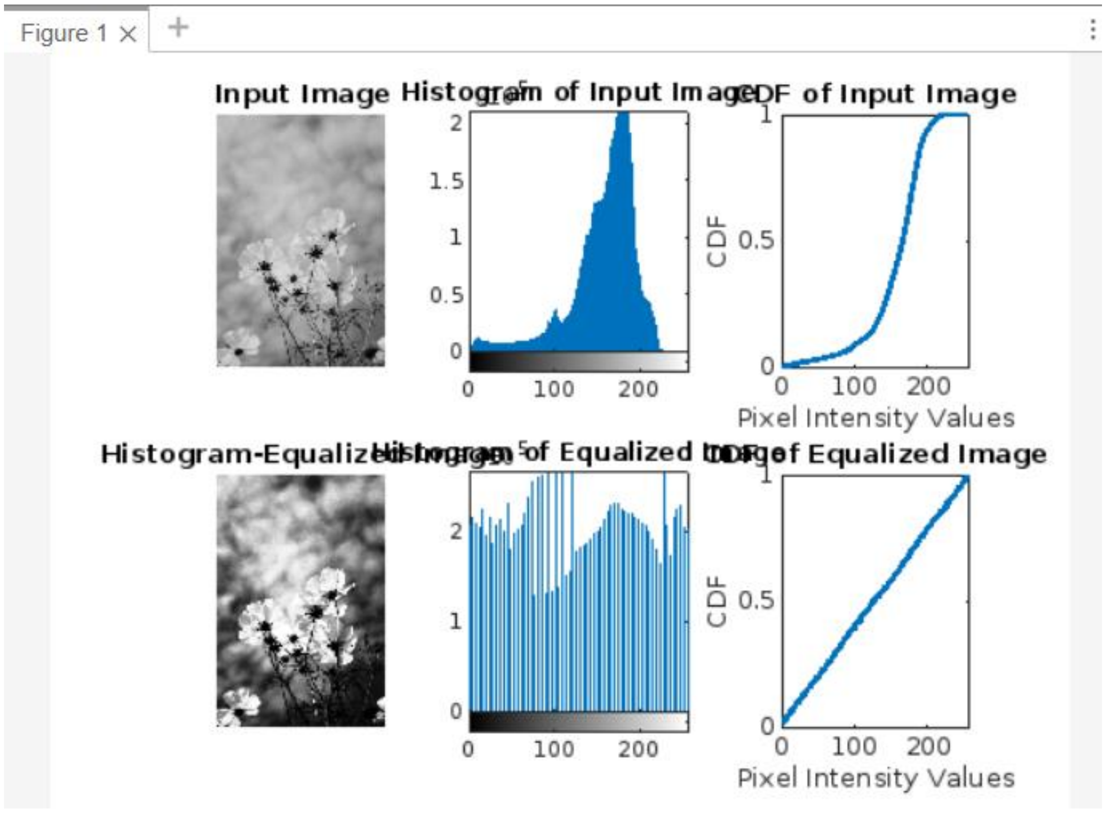


Figure 2.2: Showing the CDF using MATLAB

Explanation:

1. Reading the Input Image
2. Converting the Image to Grayscale
3. The input image is displayed using imshow.
4. The histogram of the input image is plotted using imhist.
5. Calculating and Plotting the CDF of the Input Image

Experiment Name: Histogram Matching in Digital Image Processing

Objectives:

1. Improve Image Contrast
2. By stretching the histogram, histogram specification can reveal hidden details in the image that were previously not visible.
3. By adjusting the histogram, histogram specification can improve the segmentation of objects in an image.
4. Histogram specification can also be used to enhance the color of an image by adjusting the histogram of each color channel (e.g., RGB).

Code-01: Python

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
from skimage.exposure import match_histograms
input_image = cv2.imread('nature.jpeg', cv2.IMREAD_GRAYSCALE)
reference_image = cv2.imread('flower.jpeg', cv2.IMREAD_GRAYSCALE)
matched_image = match_histograms(input_image, reference_image)
plt.figure(figsize=(12, 6))
plt.subplot(2, 3, 1)
plt.imshow(input_image, cmap='gray')
plt.title('Input Image')
plt.axis('off')
plt.subplot(2, 3, 2)
plt.imshow(reference_image, cmap='gray')
plt.title('Reference Image')
plt.axis('off')
plt.subplot(2, 3, 3)
plt.imshow(matched_image, cmap='gray')
plt.title('Histogram-Matched Image')
plt.axis('off')
plt.subplot(2, 3, 4)
plt.hist(input_image.ravel(), bins=256, range=[0, 256], color='black')
plt.title('Histogram of Input Image')
plt.subplot(2, 3, 5)
plt.hist(reference_image.ravel(), bins=256, range=[0, 256], color='black')
plt.title('Histogram of Reference Image')
plt.subplot(2, 3, 6)
plt.hist(matched_image.ravel(), bins=256, range=[0, 256], color='black')
plt.title('Histogram of Matched Image')
plt.tight_layout()
plt.show()
```

Output:

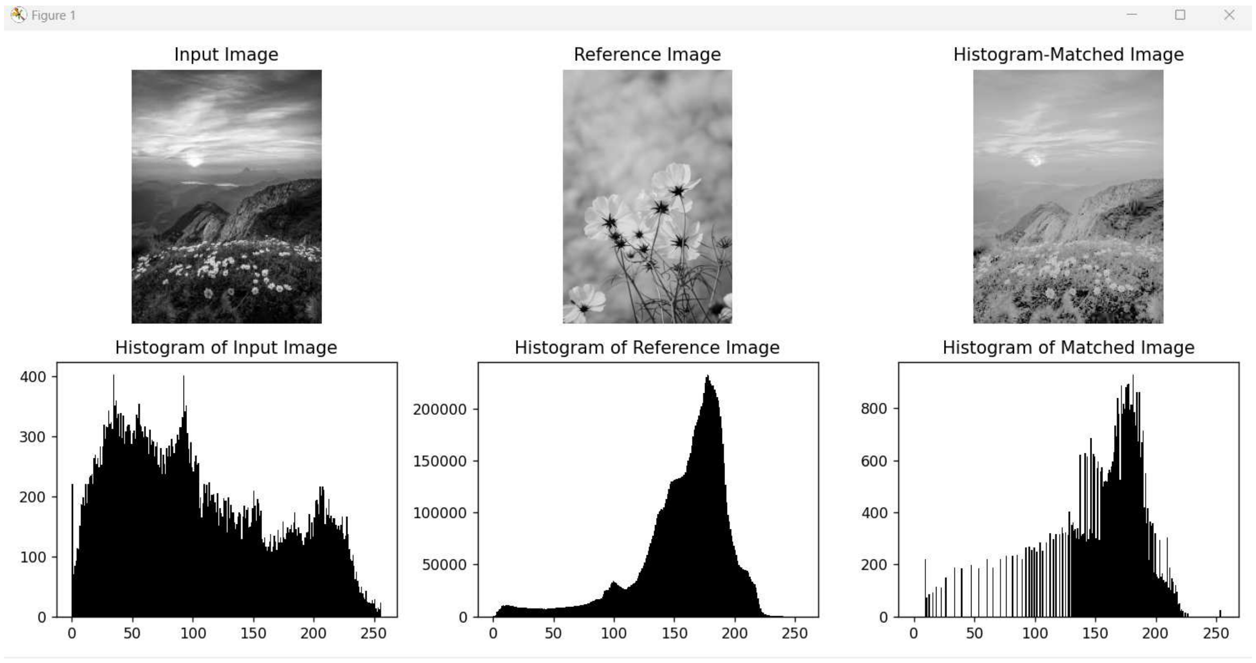


Figure 3.1: Showing the histogram specification using python code

Explanation:

1. Importing Libraries
2. input_image (the image to be transformed)
3. reference_image (the image whose histogram will be matched)
4. The code uses the match_histograms function from scikit-image to match the histogram of the input_image to the histogram of the reference_image. This function returns the histogram-matched image, which is stored in the matched_image variable.
5. The code plots the histograms of the three images using hist.

Code-02: MATLAB

```
input_image = imread('nature.jpeg');
reference_image = imread('flower.jpeg');
if size(input_image, 3) == 3
    input_image = rgb2gray(input_image);
end
if size(reference_image, 3) == 3
    reference_image = rgb2gray(reference_image);
end
matched_image = imhistmatch(input_image, reference_image);
figure;
subplot(2, 3, 1);
imshow(input_image);
title('Input Image');
subplot(2, 3, 2);
imshow(reference_image);
title('Reference Image');
subplot(2, 3, 3);
imshow(matched_image);
title('Histogram-Matched(I)');
subplot(2, 3, 4);
imhist(input_image);
title('Input Image');
subplot(2, 3, 5);
imhist(reference_image);
title('Reference Image');
subplot(2, 3, 6);
imhist(matched_image);
title('Matched Image');
```

Output:

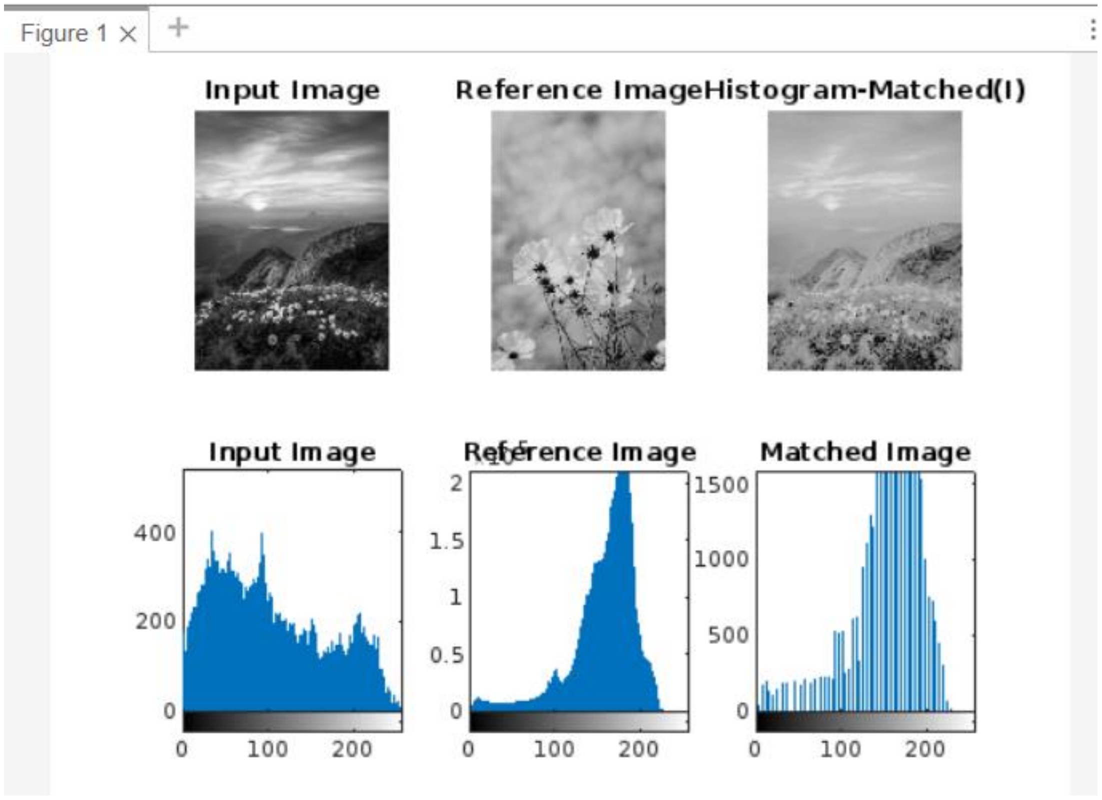


Figure 3.2: Showing the histogram specification in MATLAB

Explanation:

1. Loading Images
2. Converting to Grayscale
3. Performing Histogram Matching
4. Displaying Images and Histograms

Lab Report: 04

Title: Image Resizing & Filtering

Course title: Digital Image Processing Laboratory

Course code: CSE-406

4th Year 1st Semester Examination 2023

Date of Submission: 22/09/2024

Submitted to-

Dr. Md. Golam Moazzam
Professor
Department of Computer Science and Engineering
Jahangirnagar University
&
Dr. Morium Akter
Professor
Department of Computer Science and Engineering
Jahangirnagar University
Savar, Dhaka-1342

Class Roll	Exam Roll	Name
353	202165	Shanjida Alam

Department of Computer Science and Engineering
Jahangirnagar University
Savar, Dhaka, Bangladesh

Experiment No: 01

Experiment Name: Gaussian Filter

Objectives:

1. The primary goal of the Gaussian filter is to reduce image noise and detail.
2. Gaussian filters are used for blurring images.
3. Gaussian filters remove high-frequency components from the image

Code-01: Python

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
image = cv2.imread('nature.jpeg')
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
gaussian_filtered_image = cv2.GaussianBlur(image_rgb, (7, 7), 0)
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image_rgb)
plt.title('Original Image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(gaussian_filtered_image)
plt.title('Gaussian Filtered Image')
plt.axis('off')
plt.show()
```

Output:

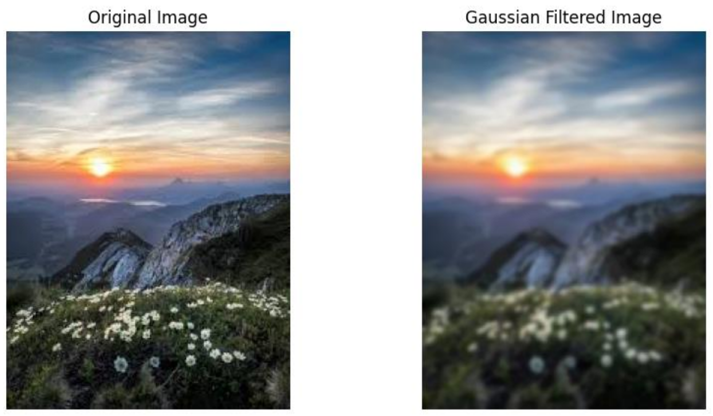


Figure 1.1: Showing the Gaussian Filter in python

Explanation:

1. Importing libraries.
2. Loading and Converting the Image.
3. **cv2.GaussianBlur(image_rgb, (7, 7), 0)** applies a Gaussian blur with a kernel size of 7x7. The 0 refers to automatic computation of the standard deviation.
4. Plotting the Results.

Code-02: MATLAB

```
image = imread('nature.jpeg');
grayImage = rgb2gray(image);
sigma = 2;
filteredImage = imgaussfilt(grayImage, sigma);
figure;
subplot(1, 2, 1);
imshow(grayImage);
title('Original Image');
subplot(1, 2, 2);
imshow(filteredImage);
title('Gaussian Filtered Image');
```

Output:

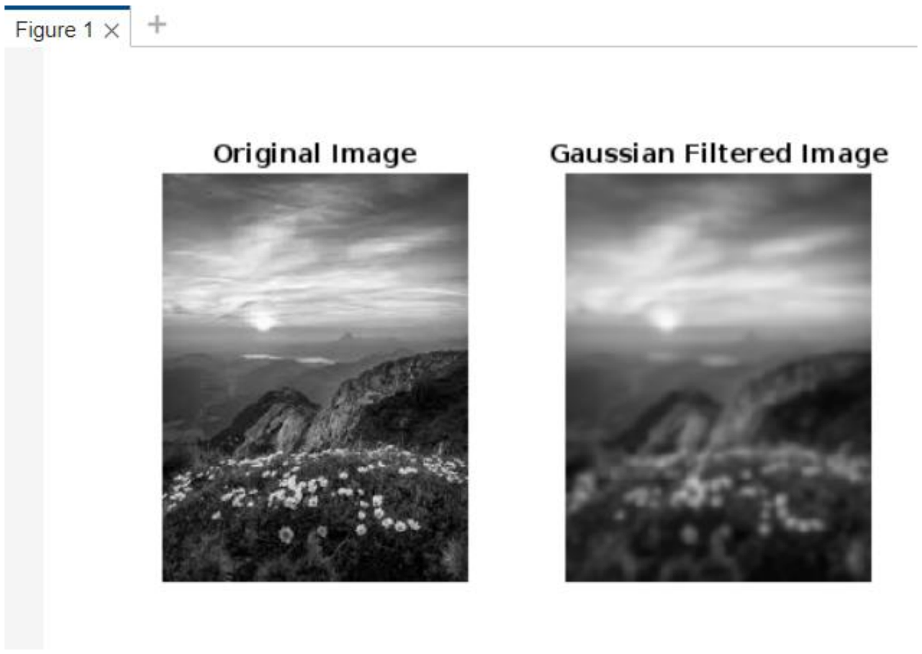


Figure 1.2: Showing the Gaussian Filter in MATLAB

Explanation:

1. Reading the Input Image
2. Converting the Image to Grayscale (If Necessary)
3. **filteredImage = imgaussfilt(grayImage, sigma);** applies a Gaussian filter with **sigma = 2** to the grayscale image, producing a smoothed (blurred) image.The code uses histeq() to perform histogram equalization on the input image.
4. Displaying the Images

Experiment No: 02

Experiment Name: Mean Filter

Objectives:

1. Noise Re.duction
2. Blurring.
3. Edge Softening.
4. Preprocessing for Other Algorithms.

Code-01: Python

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
image = cv2.imread('nature.jpeg')
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
mean_filtered_image = cv2.blur(image_rgb, (7, 7))
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image_rgb)
plt.title('Original Image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(mean_filtered_image)
plt.title('Mean Filtered Image')
plt.axis('off')
plt.show()
```

Output:

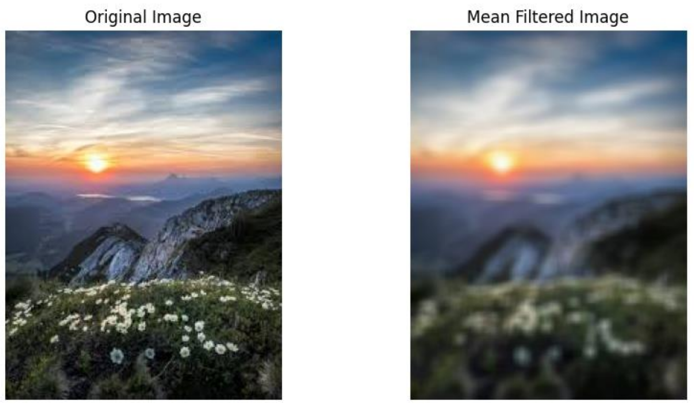


Figure 2.1: Showing the Mean Filter using python code

Explanation:

1. Importing Libraries such as cv2, numpy, matplotlib.pyplot
2. Loading and Converting the Image
3. **cv2.blur(image_rgb, (7, 7));** Applies a mean filter (also known as a box filter) with a 7x7 kernel size to the image. This averages the pixel values in a 7x7 neighborhood, resulting in a blurred image.
4. Displaying the Results

Code-02: MATLAB

```
 image = imread('nature.jpeg');
grayImage = rgb2gray(image);
meanFilterKernel = ones(5, 5) / 9;
filteredImage = imfilter(grayImage, meanFilterKernel);
figure;
subplot(1, 2, 1);
imshow(grayImage);
title('Original Image');
subplot(1, 2, 2);
imshow(filteredImage);
title('Mean Filtered Image');
```

Output:

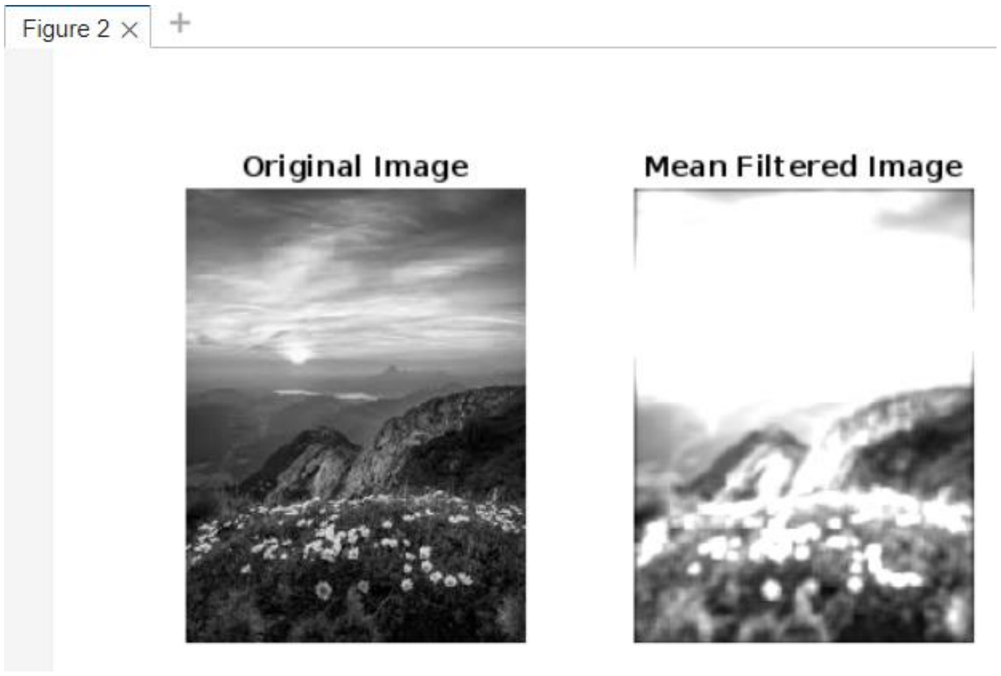


Figure 2.2: Showing the Mean Filter using MATLAB

Explanation:

1. Reading the Input Image
2. Converting the Image to Grayscale
3. **ones(5, 5) / 9** creates a 5x5 matrix (mean filter kernel) filled with 1s and divides it by 9. This kernel averages the values in a neighborhood, simulating a mean filter effect.
4. **imfilter(grayImage, meanFilterKernel)** applies the mean filter kernel to the grayscale image using convolution, resulting in a smoothed (blurred) image.
5. Displaying the results.

Experiment No: 03

Experiment Name: Median Filter

Objectives:

1. Noise Reduction.
2. Edge Preservation.
3. Non-linear Filtering.
4. Smoothing without Blurring.

Code-01: Python

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
image = cv2.imread('nature.jpeg')
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
median_filtered_image = cv2.medianBlur(image_rgb, 7)
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image_rgb)
plt.title('Original Image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(median_filtered_image)
plt.title('Median Filtered Image')
plt.axis('off')
plt.show()
```

Output:

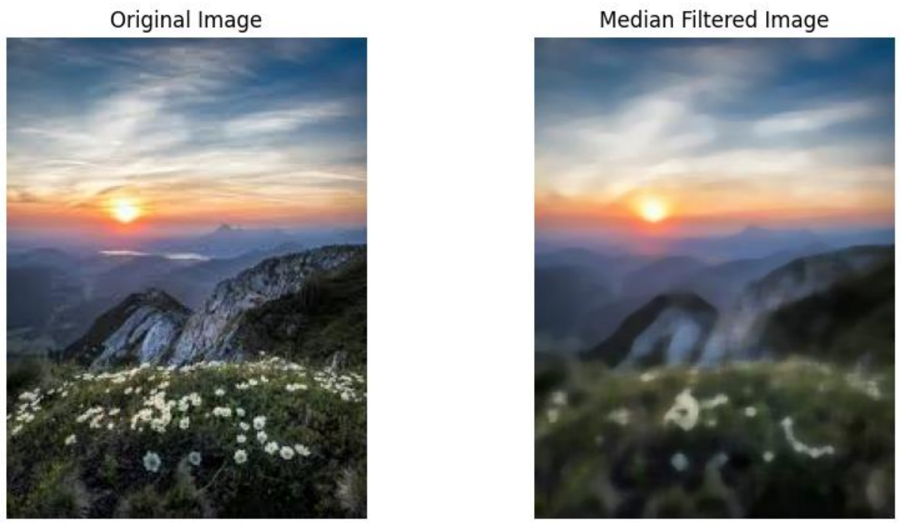


Figure 3.1: Showing the Median Filter using python code

Explanation:

1. Importing Libraries
2. Loading and Converting the Image
3. **cv2.medianBlur(image_rgb, 7)** Applies a median filter with a kernel size of 7x7 to the image. The median filter works by replacing each pixel value with the median value of the neighboring pixels, effectively reducing noise while preserving edges.
4. Displaying the Results.

Code-02: MATLAB

```
 image = imread('nature.jpeg');
if size(image, 3) == 3
   grayImage = rgb2gray(image);
else
   grayImage = image;
end
filterSize = 3;
filteredImage = medfilt2(grayImage, [filterSize filterSize]);
figure;
subplot(1, 2, 1);
imshow(grayImage);
title('Original Image');
```

```
subplot(1, 2, 2);
imshow(filteredImage);
title('Median Filtered Image');
```

Output:

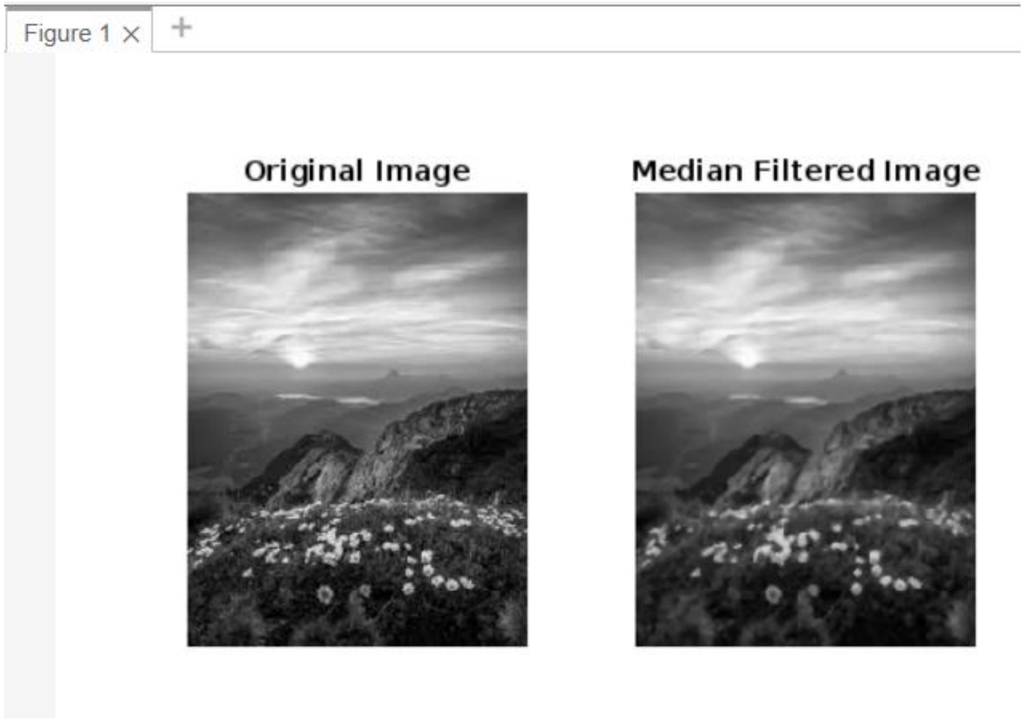


Figure 3.2: Showing the median filter in MATLAB

Explanation:

1. image = imread('nature.jpeg'); reads the image file nature.jpeg into the image variable.
2. if size(image, 3) == 3: Checks if the image is an RGB image (i.e., it has 3 color channels). If true, it converts the image to grayscale.
   rgb2gray(image) converts the color image to grayscale.
   else grayImage = image; ensures that if the image is already grayscale, it is assigned directly to grayImage.
3. **medfilt2(grayImage, [filterSize filterSize])** applies the 3x3 median filter to the grayscale image, reducing noise while preserving edges.
4. Displaying the Images.

Experiment No: 04

Experiment Name: Laplacian Filter

Objectives:

1. Zero-Crossing Detection.
2. Edge Detection.
3. Isotropic Filtering.

Code-01: Python

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
image = cv2.imread('nature.jpeg')
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
gray_image = cv2.cvtColor(image_rgb, cv2.COLOR_RGB2GRAY)
laplacian_filtered_image = cv2.Laplacian(gray_image, cv2.CV_64F)
laplacian_filtered_image = np.uint8(np.absolute(laplacian_filtered_image))
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(gray_image, cmap='gray')
plt.title('Original Grayscale Image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(laplacian_filtered_image, cmap='gray')
plt.title('Laplacian Filtered Image')
plt.axis('off')
plt.show()
```

Output:

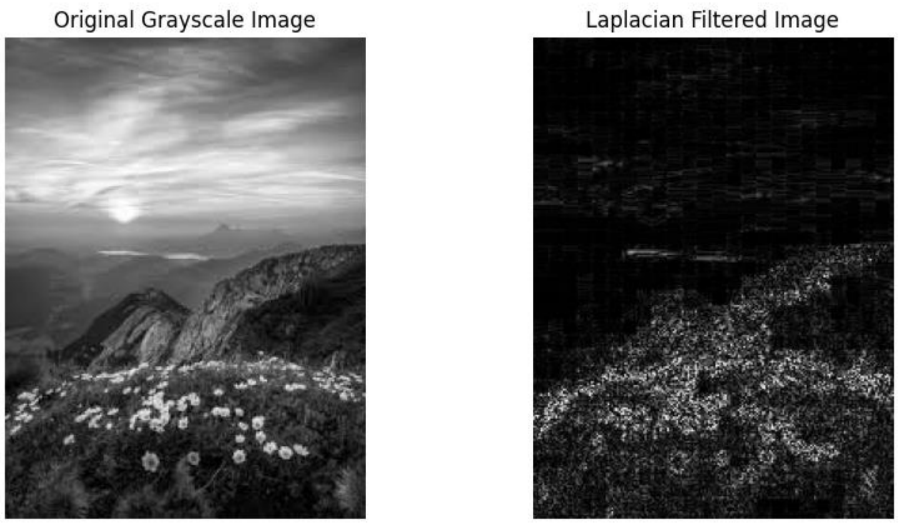


Figure 4.1: Showing the Laplacian Filter using python code

Explanation:

1. Importing Libraries
2. Loading and Converting the Image
3. **cv2.Laplacian(gray_image, cv2.CV_64F):** Applies the Laplacian filter to the grayscale image using a 64-bit floating-point data type. The Laplacian filter highlights regions of rapid intensity change (edges).
4. Displaying the Results.

Code-02: MATLAB

```
image = imread('nature.jpeg');
grayImage = rgb2gray(image);
laplacianKernel = fspecial('laplacian', 0.2);
filteredImage = imfilter(grayImage, laplacianKernel);
figure;
subplot(1, 2, 1);
imshow(grayImage);
title('Original Image');
subplot(1, 2, 2);
imshow(filteredImage, []);
title('Laplacian Filtered Image');
```

Output:

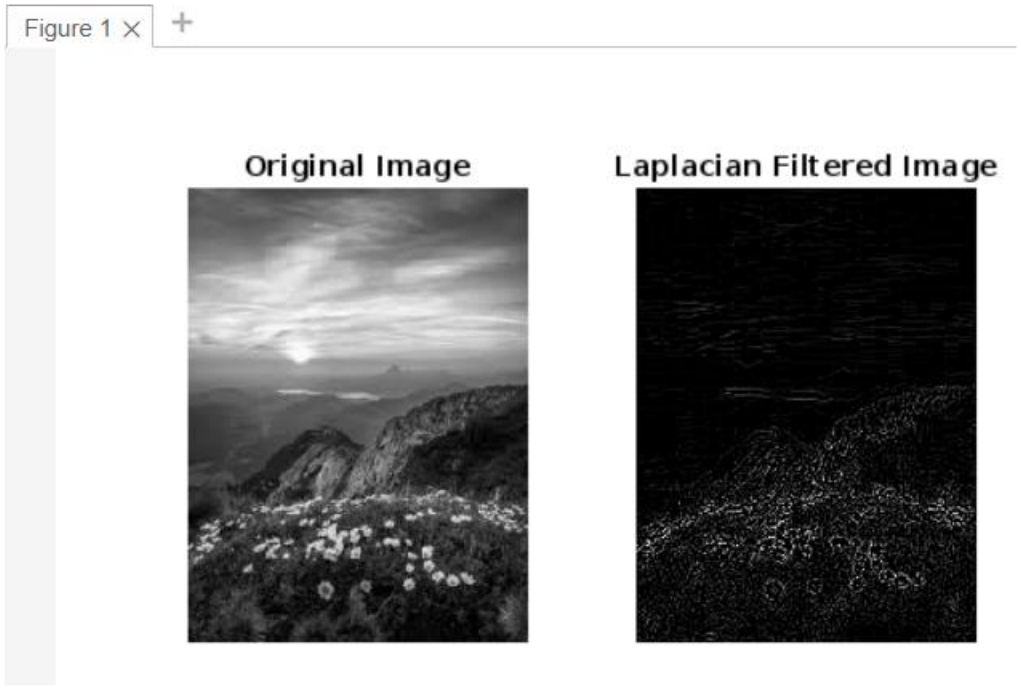


Figure 4.2: Showing the Laplacian Filter in MATLAB

Explanation:

1. **imread('nature.jpeg'):** Reads the image file nature.jpeg and stores it in the image variable.
2. **rgb2gray(image):** Converts the original RGB image to a grayscale image, reducing it to a single intensity channel, which simplifies the Laplacian filtering process.
3. **fspecial('laplacian', 0.2):** Creates a Laplacian filter kernel with a parameter 0.2, which defines the level of edge enhancement (the higher the parameter, the stronger the edge detection).
4. **imfilter(grayImage, laplacianKernel):** Applies the Laplacian filter to the grayscale image using convolution, resulting in an image that highlights the edges where rapid intensity changes occur.
5. Displaying the results.

Experiment No: 05

Experiment Name: Sharpening Filter

Objectives:

1. Enhancing Image Details.
2. Edge Enhancement.
3. Restoring Blurred Images.

Code-01: Python

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
image = cv2.imread('nature.jpeg')
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
sharpening_kernel = np.array([[-1, -1, -1],
                              [-1,  9, -1],
                              [-1, -1, -1]])
sharpened_image = cv2.filter2D(image_rgb, -1, sharpening_kernel)
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image_rgb)
plt.title('Original Image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(sharpened_image)
plt.title('Sharpened Image')
plt.axis('off')
plt.show()
```

Output:

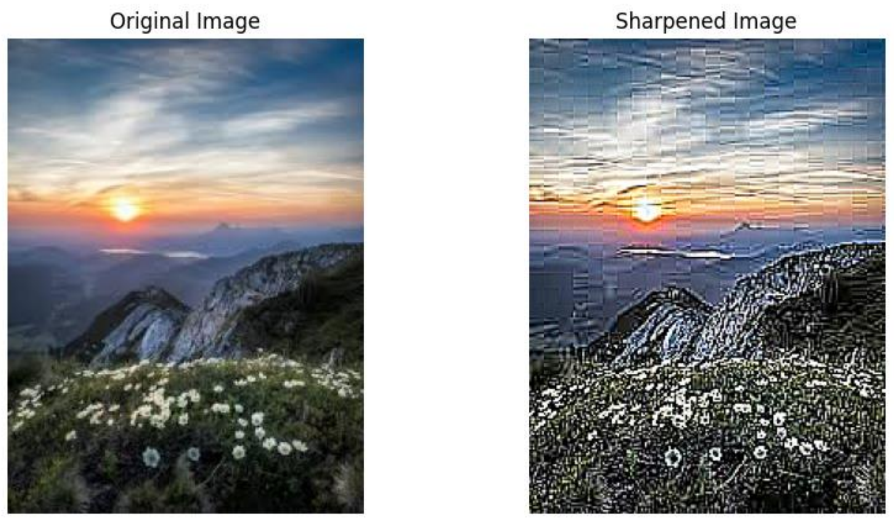


Figure 5.1: Showing the Sharpening Filter using python code

Explanation:

1. Importing Libraries
2. Loading and Converting the Image
3. **sharpening_kernel = np.array([...]):** Defines a 3x3 sharpening kernel. This kernel emphasizes the center pixel while reducing the influence of the surrounding pixels. The center weight is 9, and the surrounding weights are -1, which helps sharpen the image by enhancing edges and details.
4. **cv2.filter2D(image_rgb, -1, sharpening_kernel):** Applies the sharpening filter to the RGB image. The filter2D function convolves the sharpening kernel with the image, creating a sharpened effect by increasing the contrast at edges.
5. Displaying the Results.

Code-02: MATLAB

```
image = imread('nature.jpeg');
grayImage = rgb2gray(image);
sharpeningKernel = fspecial('unsharp');
sharpenedImage = imfilter(grayImage, sharpeningKernel);
figure;
subplot(1, 2, 1);
imshow(grayImage);
title('Original Image');
```

```
subplot(1, 2, 2);
imshow(sharpenedImage);
title('Sharpened Image (Using fspecial)');
```

Output:

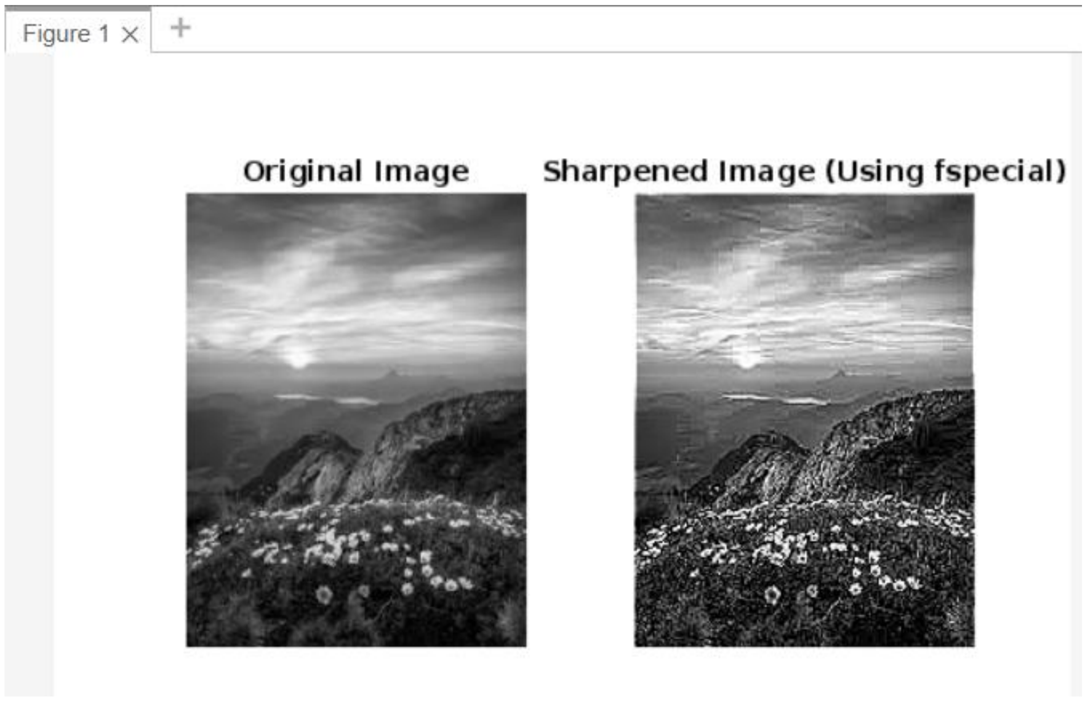


Figure 5.2: Showing the Sharpening Filter in MATLAB

Explanation:

1. **imread('nature.jpeg'):** Reads the image file nature.jpeg into the variable image.
2. **rgb2gray(image):** Converts the RGB color image into a grayscale image, simplifying it to a single intensity channel.
3. **fspecial('unsharp'):** Creates an unsharp mask filter. The unsharp mask works by subtracting a blurred version of the image from the original, which emphasizes edges and details, effectively sharpening the image.
4. **imfilter(grayImage, sharpeningKernel):** Applies the unsharp mask filter to the grayscale image using convolution, resulting in a sharpened image where edges and fine details are more pronounced.
5. Displaying the results.

Experiment No: 06

Experiment Name: Pixel Skipping

Objectives:

1. Reducing Computational Load.
2. Pixel skipping helps to reduce the size of an image by selecting fewer pixels from the original image, effectively reducing resolution.
3. Pixel skipping can be used to send a lower-resolution version of an image or video, reducing data size while still retaining important information.

Code-01: Python

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
image = cv2.imread('nature.jpeg')
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
skip_factor = 3
skipped_image = image_rgb[::skip_factor, ::skip_factor]
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.imshow(image_rgb)
plt.title('Original Image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(skipped_image)
plt.title('Pixel Skipped Image')
plt.axis('off')
plt.show()
```

Output:

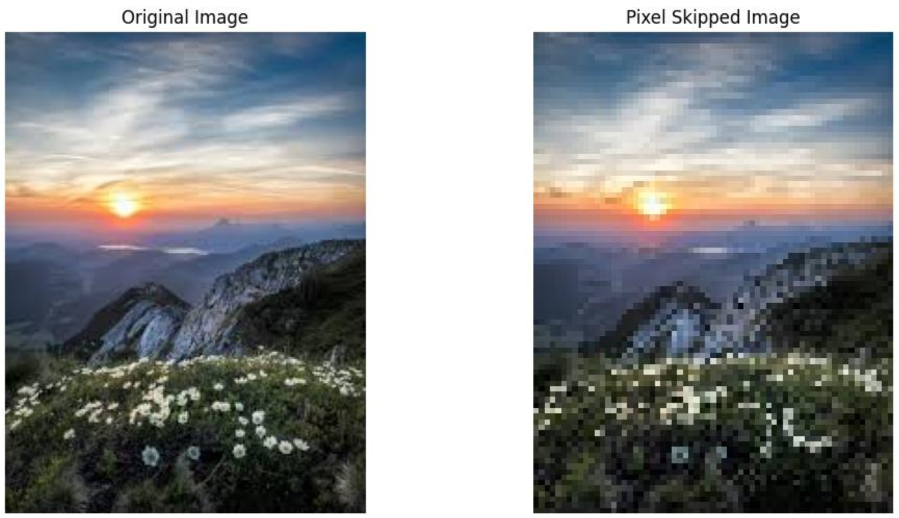


Figure 6.1: Showing the Pixel Skipping using python code

Explanation:

1. Importing Libraries
2. Loading and Converting the Image
3. **skip_factor = 3:** Defines the factor by which pixels will be skipped (every 3rd pixel).
4. **image_rgb[::skip_factor, ::skip_factor]:** Slices the image using a skip factor of 3 for both rows and columns, effectively downsampling the image. This means that only every third pixel is kept, reducing the image resolution.
5. Displaying the Results.

Code-02: MATLAB

```
image = imread('nature.jpeg');
if size(image, 3) == 3
   grayImage = rgb2gray(image);
else
   grayImage = image;
end
skipFactor = 4;
skippedImage = grayImage(1:skipFactor:end, 1:skipFactor:end);
figure;
subplot(1, 2, 1);
```

```
imshow(grayImage);
title('Original Image');
subplot(1, 2, 2);
imshow(skippedImage);
title('Pixel Skipped Image');
```

Output:

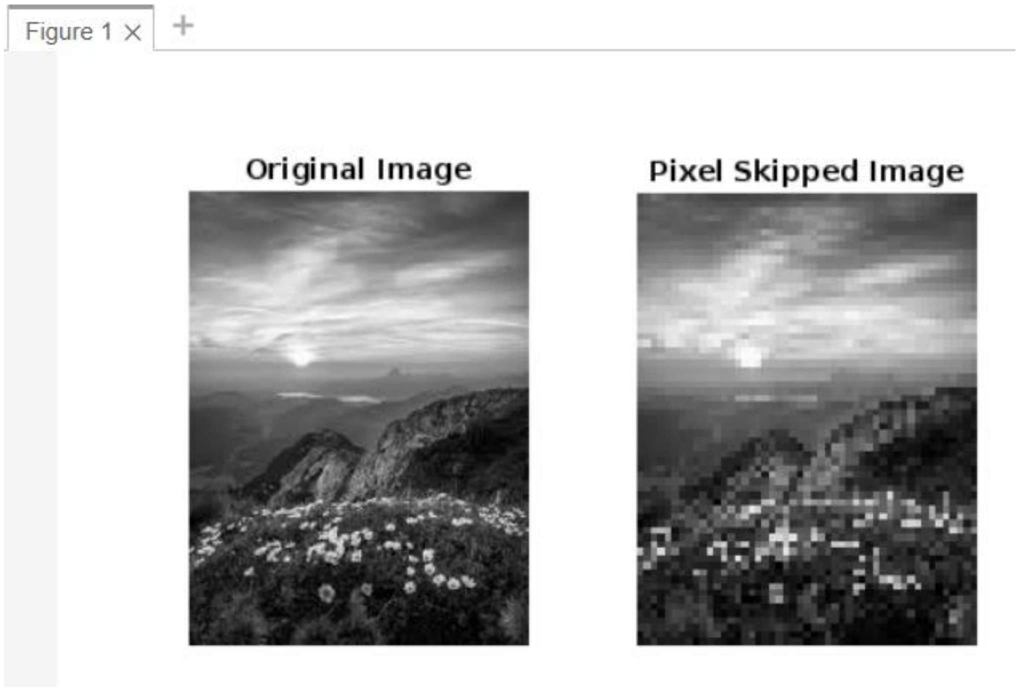


Figure 6.2: Showing the Pixel Skipping in MATLAB

Explanation:

1. **imread('nature.jpeg'):** Reads the image file nature.jpeg into the variable image.
2. **if size(image, 3) == 3:** Checks if the image has 3 channels (i.e., it's an RGB image).
   rgb2gray(image): Converts the RGB image to grayscale.
   else grayImage = image: If the image is already grayscale, it skips the conversion and assigns it directly to grayImage.
3. **skipFactor = 4:** Defines the factor by which pixels will be skipped (every 4th pixel).
   grayImage(1:skipFactor:end, 1:skipFactor:end): Selects every 4th pixel along both rows and columns, effectively down sampling the grayscale image.
4. Displaying the results.

Experiment No: 07

Experiment Name: Image Resize using Replication Method

Objectives:

1. The replication method is simple and computationally efficient.
2. This method is effective at maintaining sharp, hard edges in an image, as it does not blend or smooth pixel values.
3. The method's low complexity means it is easy to implement and uses minimal computational resources, making it suitable for resource-constrained systems or devices.

Code-01: Python

```
import cv2
from matplotlib import pyplot as plt
image = cv2.imread('nature.jpeg')
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
width, height = 400, 300
resized_image = cv2.resize(image_rgb, (width, height), interpolation=cv2.INTER_NEAREST)
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image_rgb)
plt.title('Original Image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(resized_image)
plt.title('Resized Image (Replication Method)')
plt.axis('off')
plt.show()
```

Output:

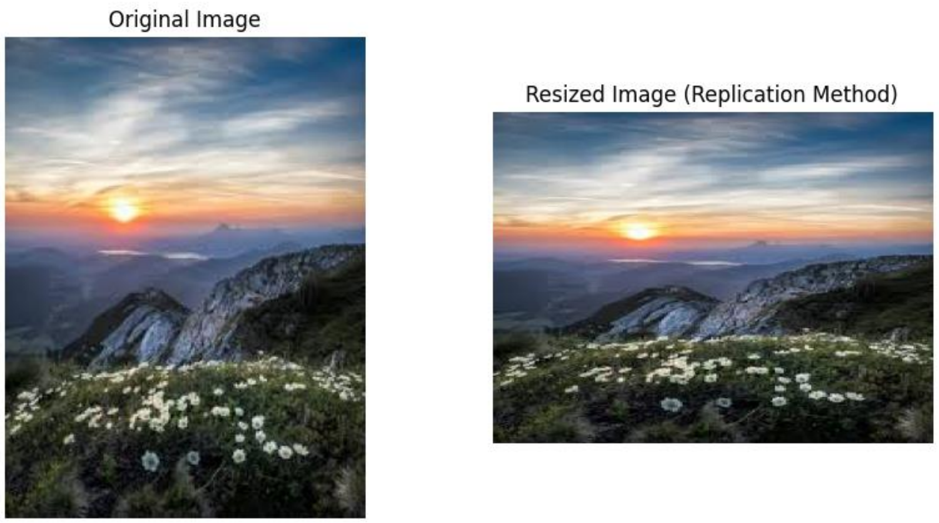


Figure 7.1: Showing the Image Resizing in Replication method using python code

Explanation:

1. Importing Libraries
2. Loading and Converting the Image
3. **width, height = 400, 300:** Defines the new width and height for the resized image. **cv2.resize(image_rgb, (width, height), interpolation=cv2.INTER_NEAREST):** Resizes the image to the specified width and height using nearest-neighbor interpolation (replication method), where the pixel values are copied from the nearest neighbor without blending.
4. Displaying the Results.

Code-02: MATLAB

```
image = imread('nature.jpeg');
if size(image, 3) == 3
   grayImage = rgb2gray(image);
else
   grayImage = image;
end
newSize = [300 400];
resizedImage = imresize(grayImage, newSize, 'nearest');
figure;
```

```
subplot(1, 2, 1);
imshow(grayImage);
title('Original Image');
subplot(1, 2, 2);
imshow(resizedImage);
title('Resized Image (Replication Method)');
```

Output:

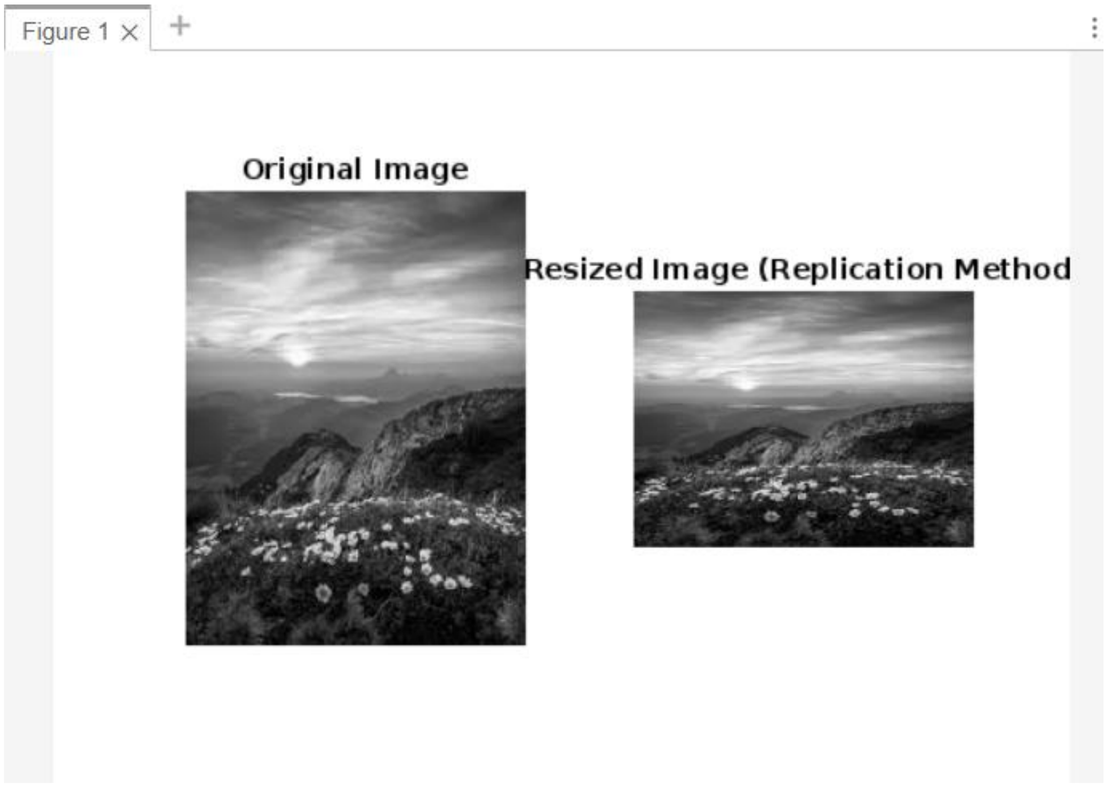


Figure 7.2: Showing the Image Resizing in Replication method using MATLAB

Explanation:

1. **imread('nature.jpeg'):** Reads the image file nature.jpeg into the variable image.
2. **if size(image, 3) == 3:** Checks if the image has 3 channels (i.e., it's an RGB image).
   rgb2gray(image): Converts the RGB image to grayscale.
   else grayImage = image: If the image is already grayscale, it skips the conversion and assigns it directly to grayImage.
3. **newSize = [300 400]:** Specifies the new size for the resized image (300 pixels in height and 400 pixels in width).

4. **imresize(grayImage, newSize, 'nearest'):** Resizes the grayscale image using nearest-neighbor interpolation (replication method), where pixel values are copied from the nearest neighbor without interpolation or smoothing.
5. Displaying the results.

Experiment No: 08

Experiment Name: Image Resize using Interpolation Method

Objectives:

1. The goal of interpolation is to preserve as much of the original image's quality and detail as possible during the resizing process.
2. Interpolation methods help maintain the correct proportions and prevent distortion when resizing an image.
3. During image downsizing, interpolation helps determine the most appropriate pixel values to retain, reducing the loss of important image details while discarding redundant data.

Code-01: Python

```
import cv2
from matplotlib import pyplot as plt
image = cv2.imread('nature.jpeg')
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
new_width, new_height = 400, 300
resized_image = cv2.resize(image_rgb, (new_width, new_height),
interpolation=cv2.INTER_LINEAR)
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.imshow(image_rgb)
plt.title('Original Image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(resized_image)
plt.title('Resized Image (Linear Interpolation)')
plt.axis('off')
plt.show()
```

Output:

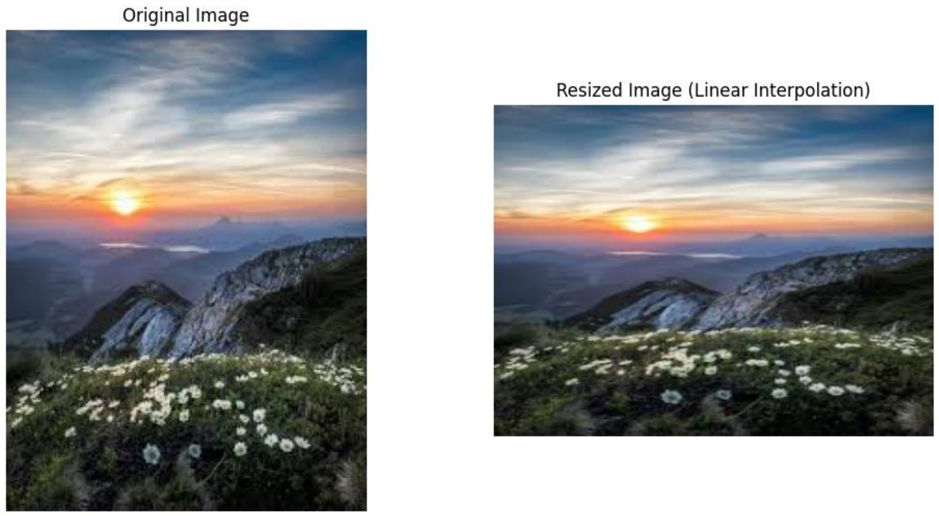


Figure 8.1: Showing the Image Resizing in Interpolation method using python code

Explanation:

1. Importing Libraries
2. Loading and Converting the Image
3. **new_width, new_height = 400, 300:** Sets the desired dimensions for the resized image (400 pixels wide and 300 pixels tall).
   cv2.resize(image_rgb, (new_width, new_height), interpolation=cv2.INTER_LINEAR): Resizes the image to the specified dimensions using linear interpolation. This method estimates the value of new pixels by taking a weighted average of the four nearest neighboring pixels, producing smoother results than the nearest-neighbor method.
4. Displaying the Results.

Code-02: MATLAB

```
image = imread('nature.jpeg');
grayImage = rgb2gray(image);
newSize = [300 400];
resizedImage = imresize(grayImage, newSize, 'bilinear');
figure;
subplot(1, 2, 1);
imshow(grayImage);
```

```
title('Original Image');
subplot(1, 2, 2);
imshow(resizedImage);
title('Resized Image (Linear Interpolation)');
```

Output:

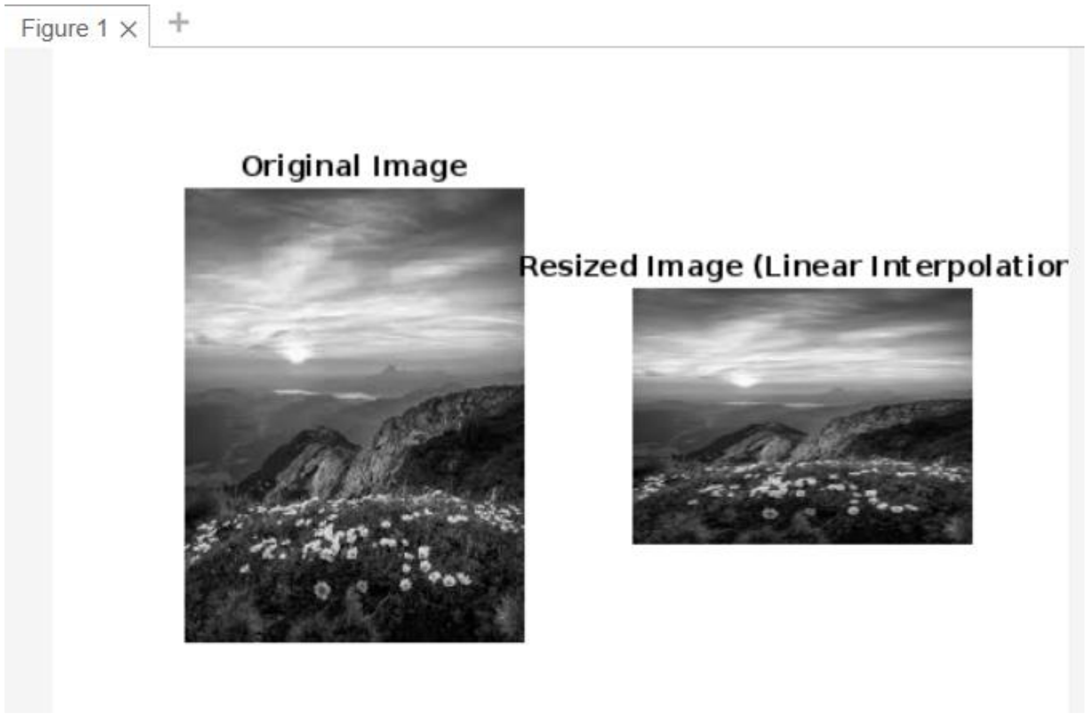


Figure 8.2: Showing the Image Resizing in Interpolation method using MATLAB

Explanation:

1. **imread('nature.jpeg'):** Reads the image file nature.jpeg into the variable image.
2. **if size(image, 3) == 3:** Checks if the image has 3 channels (i.e., it's an RGB image).
3. **rgb2gray(image):** Converts the RGB image to grayscale.
4. **else grayImage = image:** If the image is already grayscale, it skips the conversion and assigns it directly to grayImage.
5. **newSize = [300 400**]: Specifies the new size for the resized image, with a height of 300 pixels and a width of 400 pixels.
   imresize(grayImage, newSize, 'bilinear'): Resizes the grayscale image to the specified dimensions using bilinear interpolation. Bilinear interpolation calculates the new pixel value by taking a weighted average of the four nearest neighboring pixels, resulting in a smoother and more visually appealing image.
6. Displaying the results.

Lab Report: 05

Title: Edge Detection

Course title: Digital Image Processing Laboratory
Course code: CSE-406
4th Year 1st Semester Examination 2023

Date of Submission: 29/09/2024

Submitted to-

Dr. Md. Golam Moazzam
Professor
Department of Computer Science and Engineering
Jahangirnagar University
&
Dr. Morium Akter
Professor
Department of Computer Science and Engineering
Jahangirnagar University
Savar, Dhaka-1342

Class Roll	Exam Roll	Name
353	202165	Shanjida Alam

Department of Computer Science and Engineering
Jahangirnagar University
Savar, Dhaka, Bangladesh

Experiment No: 01

Experiment Name: Edge Detection of image using Sobel Operator

Objectives:

1. Edge Detection
2. Gradient Approximation.
3. Noise Resistance

Code-01: Python

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
image = cv2.imread('nature.jpeg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=3)
sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=3)
sobel_edge = np.sqrt(sobelx**2 + sobely**2)
plt.figure(figsize=(10,5))
plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(sobel_edge, cmap='gray')
plt.title('Edge Detection using Sobel Operator')
plt.axis('off')
plt.show()
```

Output:

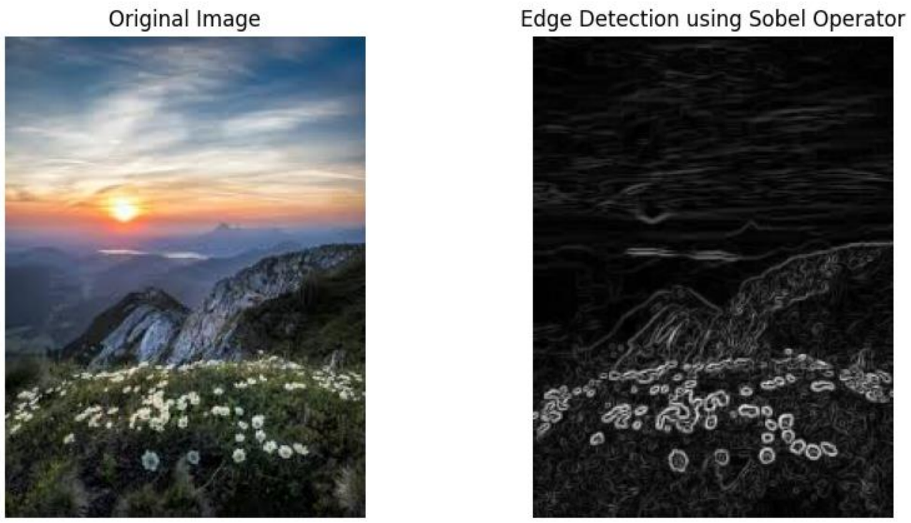


Figure 1.1: Showing the image detection using Sobel operator in python

Explanation:

1. Read the Input Image: The image is read using cv2.imread() from OpenCV.
2. Convert to Grayscale: The image is converted to grayscale using cv2.cvtColor() since edge detection is commonly applied to grayscale images.
3. Sobel Operator:
4. cv2.Sobel() is used to compute the gradient in the x and y directions. The ksize=3 parameter defines the kernel size.
5. The sobelx variable holds the horizontal edge gradients, and sobely holds the vertical edge gradients.
6. Calculate the Gradient Magnitude: The edge magnitude is calculated using the formula Gx2+Gy2\sqrt{G_x^2 + G_y^2}Gx2+Gy2, where sobelx and sobely are the gradients in the x and y directions.
7. Display the Images: matplotlib.pyplot is used to display the original and edge-detected images side by side.

Code-02: MATLAB

```
I = imread('nature.jpeg');
I_gray = rgb2gray(I);
Gx = [-1 0 1; -2 0 2; -1 0 1];
Gy = [-1 -2 -1; 0 0 0; 1 2 1];
Ix = imfilter(double(I_gray), Gx);
Iy = imfilter(double(I_gray), Gy);
SobelEdge = sqrt(Ix.^2 + Iy.^2);
figure;
subplot(1, 2, 1), imshow(I), title('Original Image');
subplot(1, 2, 2), imshow(uint8(SobelEdge)), title('Edge Detection using Sobel Operator');
```

Output:

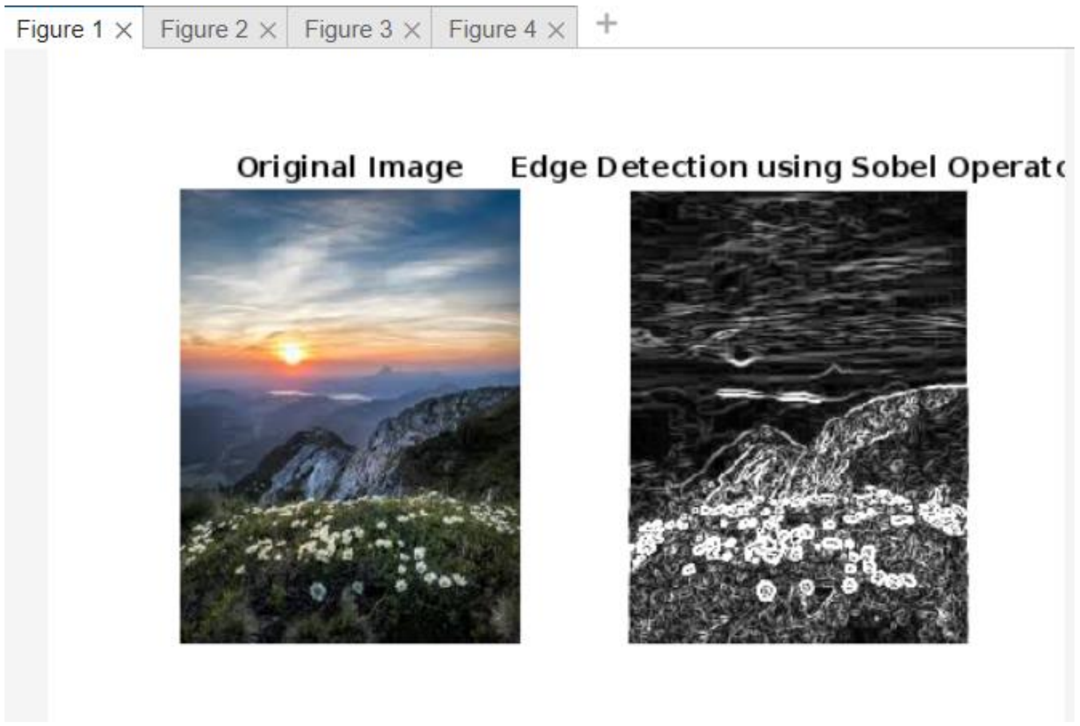


Figure 1.2: Edge detection using Sobel operator in MATLAB

Explanation:

1. **Read the Input Image:** The image is loaded using the imread() function..
2. **Convert to Grayscale:** The image is converted to grayscale using rgb2gray() to simplify the edge detection process, as edge detection usually works better in grayscale images.
3. **Sobel Operators:** Two Sobel kernels, GxG_xGx and GyG_yGy, are used to compute the gradient in the x and y directions.
4. **Gradient Magnitude**: The magnitude of the gradient is calculated as SobelEdge = sqrt(Ix.^2 + Iy.^2), giving the edge intensity.
5. **Display the Results:** The original image and the result of the Sobel edge detection are displayed side by side using subplot().

Experiment: 02

Experiment Name: Edge Detection of image using Prewitt Operator

Objectives:

1. Edge Detection
2. Gradient Approximation
3. Simplicity and Efficiency

Code-01: Python

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
image = cv2.imread('nature.jpeg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
prewittx = np.array([[ -1,  0,  1],
                     [ -1,  0,  1],
                     [ -1,  0,  1]])

prewitty = np.array([[  1,  1,  1],
                     [  0,  0,  0],
                     [ -1, -1, -1]])
edge_x = cv2.filter2D(gray, -1, prewittx)
edge_y = cv2.filter2D(gray, -1, prewitty)
prewitt_edge = np.sqrt(edge_x**2 + edge_y**2)
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(prewitt_edge, cmap='gray')
plt.title('Edge Detection using Prewitt Operator')
plt.axis('off')
plt.show()
```

Output:

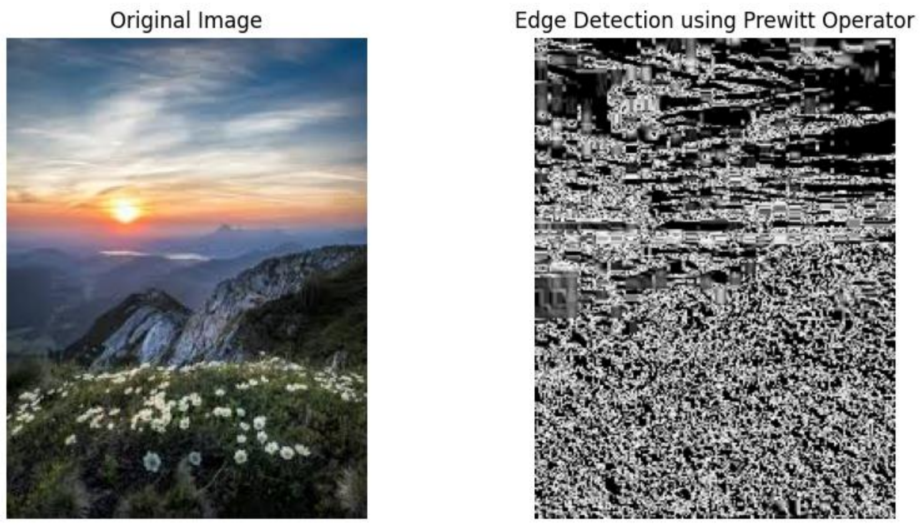


Figure 2.1: Edge detection using Prewitt operator in python code

Explanation:

1. **Read the Input Image:** The image is loaded using cv2.imread().
2. **Convert to Grayscale:** The image is converted to grayscale using cv2.cvtColor() since edge detection is more effective in grayscale.
3. **Prewitt Operator:**
   The Prewitt X and Y kernels are defined manually in prewittx and prewitty respectively. cv2.filter2D() is used to apply the Prewitt filter, which performs convolution with the Prewitt kernels on the grayscale image.
4. **Gradient Magnitude**: The edge magnitude is computed using Gx2+Gy2\sqrt{G_x^2 + G_y^2}Gx2+Gy2, where edge_x and edge_y represent the gradients in the x and y directions.
5. **Display the Images:** The original image and the Prewitt edge detection result are displayed side by side using matplotlib.pyplot.

Code-02: MATLAB

```
I = imread('nature.jpeg');
I_gray = rgb2gray(I);
Gx = [-1 0 1; -1 0 1; -1 0 1];
Gy = [-1 -1 -1; 0 0 0; 1 1 1];
Ix = imfilter(double(I_gray), Gx);
Iy = imfilter(double(I_gray), Gy);
PrewittEdge = sqrt(Ix.^2 + Iy.^2);
figure;
subplot(1, 2, 1), imshow(I), title('Original Image');
subplot(1, 2, 2), imshow(uint8(PrewittEdge)), title('Edge Detection using Prewitt Operator');
```

Output:

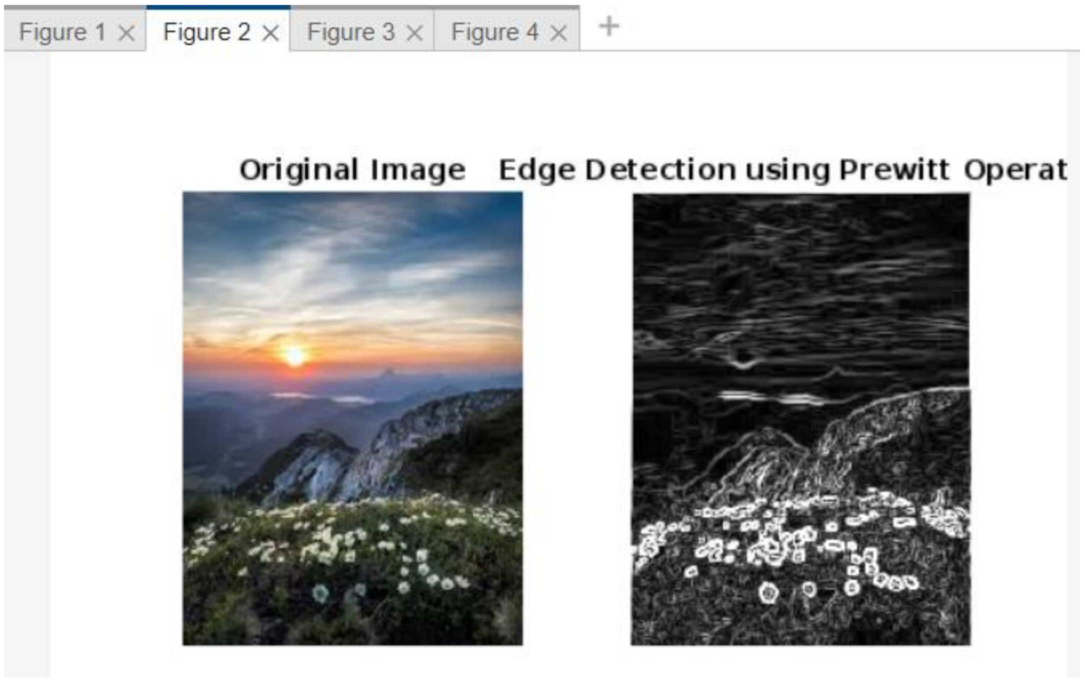


Figure 2.2: Edge detection using Prewitt operator in MATLAB

Explanation:

1. **Read the Input Image**: The image is loaded using the imread() function.
2. **Convert to Grayscale**: The image is converted to grayscale using rgb2gray() to simplify the edge detection process.

3. **Prewitt Operator**: Two Prewitt kernels, GxG_xGx and GyG_yGy, are used to compute the gradient in the x and y directions. The histogram of the input image is plotted using imhist.
4. **Gradient Magnitude**: The magnitude of the gradient is calculated as Ix2+Iy2\sqrt{I_x^2 + I_y^2}Ix2+Iy2, providing the edge intensity.
5. **Display the Results**: The original image and the result of the Prewitt edge detection are displayed side by side using subplot().

Experiment No: 03

Experiment Name: Edge Detection of image using Isotropic Operator

Objectives:

1. Edge Detection
2. Gradient Approximation
3. Simplicity and Efficiency

Code-01: Python

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
image = cv2.imread('nature.jpeg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
isotropic_x = np.array([[-1, 0, 1],
                        [-np.sqrt(2), 0, np.sqrt(2)],
                        [-1, 0, 1]])

isotropic_y = np.array([[-1, -np.sqrt(2), -1],
                        [ 0, 0, 0],
                        [ 1, np.sqrt(2), 1]])
edge_x = cv2.filter2D(gray, -1, isotropic_x)
edge_y = cv2.filter2D(gray, -1, isotropic_y)
isotropic_edge = np.sqrt(edge_x**2 + edge_y**2)
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(isotropic_edge, cmap='gray')
plt.title('Edge Detection using Isotropic Operator')
plt.axis('off')
plt.show()
```

Output:

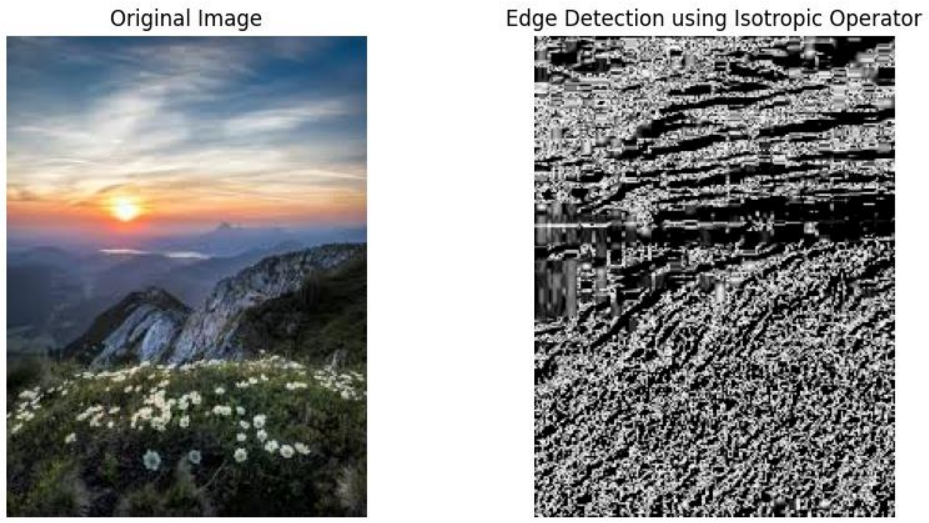


Figure 3.1: Edge detection using Isotropic operator in python code

Explanation:

1. **Read the Input Image**: The image is loaded using cv2.imread().
2. **Convert to Grayscale**: The image is converted to grayscale using cv2.cvtColor() since edge detection is commonly performed on grayscale images.
3. **Isotropic Operator**:
   The Isotropic operator kernels (isotropic_x and isotropic_y) are manually defined to approximate gradients in all directions (circular symmetry).
   cv2.filter2D() is used to apply these filters to the grayscale image.
4. **Gradient Magnitude**: The magnitude of the edge is calculated as Gx2+Gy2\sqrt{G_x^2 + G_y^2}Gx2+Gy2, where edge_x and edge_y represent the gradient in the x and y directions, respectively.
5. **Display the Results**: The original image and the result of the Isotropic edge detection are displayed side by side using matplotlib.pyplot.

Code-02: MATLAB

```
I = imread('nature.jpeg');
I_gray = rgb2gray(I);
Gx = [-1 0 1; -sqrt(2) 0 sqrt(2); -1 0 1];
Gy = [-1 -sqrt(2) -1; 0 0 0; 1 sqrt(2) 1];
Ix = imfilter(double(I_gray), Gx);
Iy = imfilter(double(I_gray), Gy);
IsotropicEdge = sqrt(Ix.^2 + Iy.^2);
figure;
subplot(1, 2, 1), imshow(I), title('Original Image');
subplot(1, 2, 2), imshow(uint8(IsotropicEdge)), title('Edge Detection using Isotropic Operator');
```

Output:

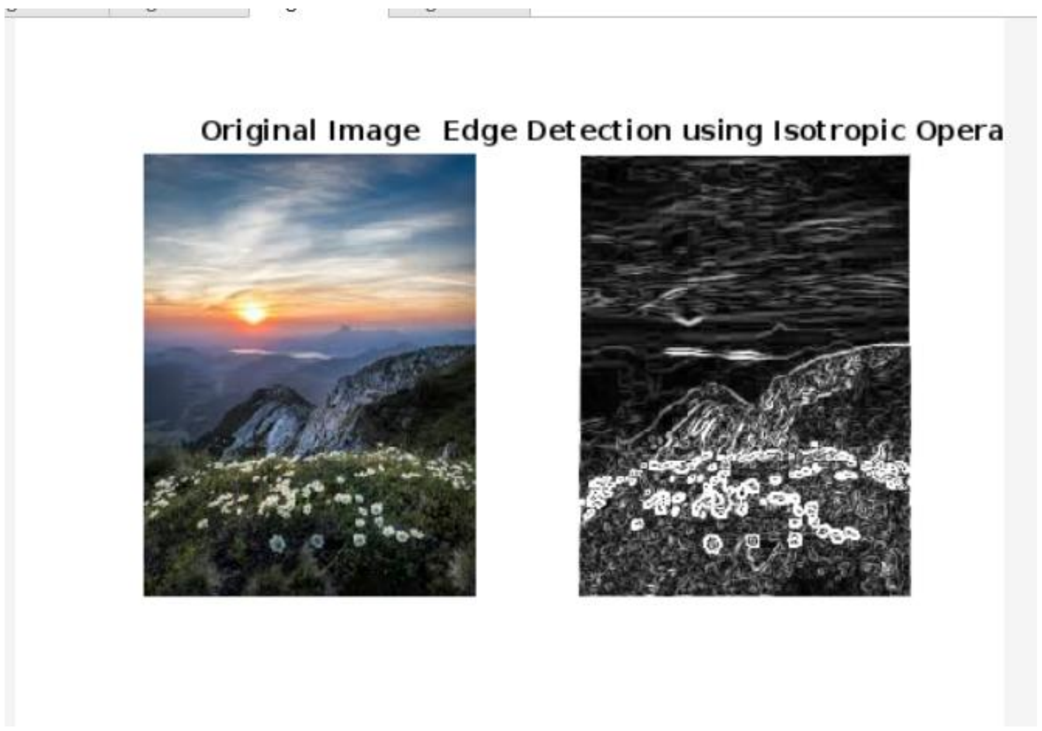


Figure 3.2: Edge detection using Isotropic operator in MATLAB

Explanation:

1. **Read the Input Image**: The image is loaded using the imread() function.
2. **Convert to Grayscale**: The image is converted to grayscale using rgb2gray() since edge detection works well on grayscale images.
3. **Isotropic Operator**: Two kernels, GxG_xGx and GyG_yGy, approximate edge detection in all directions by considering circular symmetry (approximating isotropic gradients).

4. **Gradient Magnitude**: The magnitude of the gradient is computed as Ix2+Iy2\sqrt{I_x^2 + I_y^2}Ix2+Iy2, giving the edge intensity in the image.
5. **Display the Results**: The original image and the result of the Isotropic edge detection are displayed side by side using subplot().

Experiment No: 04

Experiment Name: Edge Detection of image using Robert Operator

Objectives:

1. Edge Detection
2. Gradient Approximation
3. Simplicity and Efficiency

Code-01: Python

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
image = cv2.imread('nature.jpeg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
roberts_x = np.array([[1, 0],
                      [0, -1]])
roberts_y = np.array([[0, 1],
                      [-1, 0]])
edge_x = cv2.filter2D(gray, -1, roberts_x)
edge_y = cv2.filter2D(gray, -1, roberts_y)
roberts_edge = np.sqrt(edge_x**2 + edge_y**2)
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(roberts_edge, cmap='gray')
plt.title('Edge Detection using Roberts Operator')
plt.axis('off')
plt.show()
```

Output:

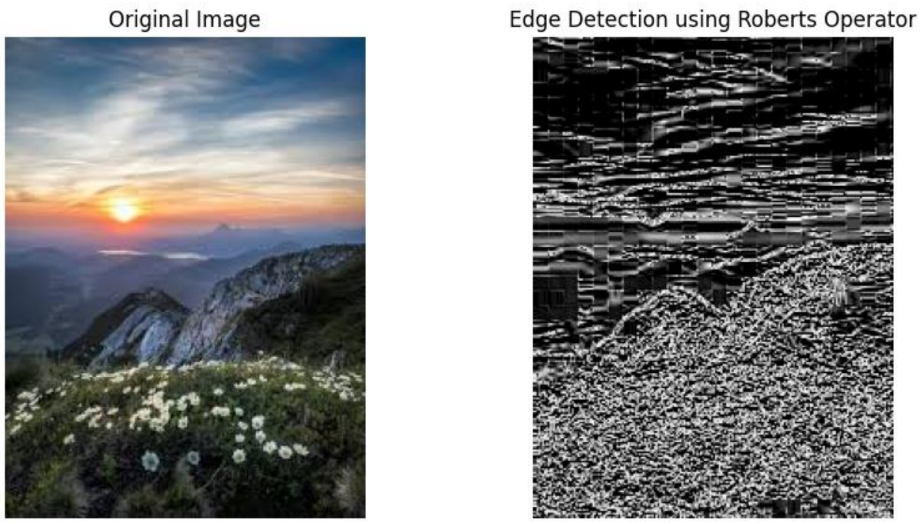


Figure 4.1: Edge detection using Robert operator in python code

Explanation:

1. **Read the Input Image**: The image is loaded using cv2.imread().
2. **Convert to Grayscale**: The image is converted to grayscale using cv2.cvtColor(), as edge detection is commonly applied on grayscale images.
3. **Roberts Operator**:
   The Roberts operator uses 2x2 kernels for diagonal edge detection. These kernels are manually defined (roberts_x and roberts_y).
   cv2.filter2D() is used to apply the Roberts filter on the grayscale image.
4. **Gradient Magnitude**: The gradient magnitude is calculated as Gx2+Gy2\sqrt{G_x^2 + G_y^2}Gx2+Gy2, where edge_x and edge_y represent the gradient in the x and y directions, respectively.
5. **Display the Results**: The original image and the edge-detected result using the Roberts operator are displayed side by side using matplotlib.pyplot.

Code-02: MATLAB

```
I = imread('nature.jpeg');
I_gray = rgb2gray(I);
Gx = [1 0; 0 -1];
Gy = [0 1; -1 0];
Ix = imfilter(double(I_gray), Gx);
Iy = imfilter(double(I_gray), Gy);
RobertsEdge = sqrt(Ix.^2 + Iy.^2);
figure;
subplot(1, 2, 1), imshow(I), title('Original Image');
subplot(1, 2, 2), imshow(uint8(RobertsEdge)), title('Edge Detection using Roberts Operator');
```

Output:

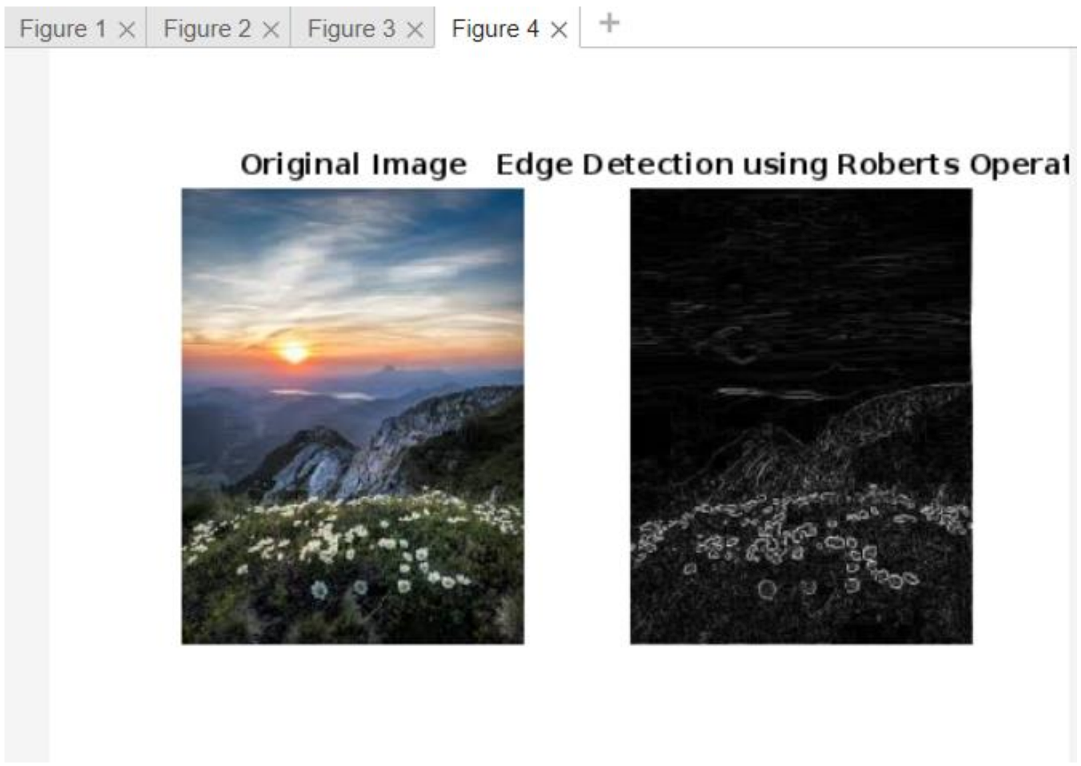


Figure 4.2: Edge detection using Robert operator in MATLAB

Explanation:

1. **Read the Input Image**: The image is loaded using the imread() function.
2. **Convert to Grayscale**: The image is converted to grayscale using rgb2gray(), as edge detection is typically performed on grayscale images.
3. **Roberts Operator**: Two Roberts kernels, GxG_xGx and GyG_yGy, are used to compute the gradient in diagonal directions. This operator is particularly sensitive to diagonal edges.
4. **Gradient Magnitude**: The magnitude of the gradient is calculated as Ix2+Iy2\sqrt{I_x^2 + I_y^2}Ix2+Iy2, which gives the edge intensity.
5. **Display the Results**: The original image and the result of the Roberts edge detection are displayed side by side using subplot().

Lab Report: 06

Title: Convert RGB to HSI color model and Grayscale

Course title: Digital Image Processing Laboratory
Course code: CSE-406
4th Year 1st Semester Examination 2023

Date of Submission: 06/10/2024

Submitted to-

Dr. Md. Golam Moazzam
Professor
Department of Computer Science and Engineering
Jahangirnagar University
&
Dr. Morium Akter
Professor
Department of Computer Science and Engineering
Jahangirnagar University
Savar, Dhaka-1342

Class Roll	Exam Roll	Name
353	202165	Shanjida Alam

Department of Computer Science and Engineering
Jahangirnagar University
Savar, Dhaka, Bangladesh

Experiment Number: 01

Experiment Name: Convert RGB color model to HIS color model

Objectives:

1. Understand the RGB Color Model and HIS Model.
2. Apply Normalization and Scaling Techniques.
3. Optimize for Computational Efficiency

Code-01: Python

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
def rgb2hsi_components(rgb_image):
    rgb_image = np.array(rgb_image, dtype=np.float64) / 255.0
    R = rgb_image[:, :, 0]
    G = rgb_image[:, :, 1]
    B = rgb_image[:, :, 2]
    I = (R + G + B) / 3
    min_RGB = np.minimum(np.minimum(R, G), B)
    S = 1 - (3 / (R + G + B + 1e-8)) * min_RGB
    num = 0.5 * ((R - G) + (R - B))
    den = np.sqrt((R - G) ** 2 + (R - B) * (G - B)) + 1e-8
    theta = np.arccos(num / den)
    H = np.copy(theta)
    H[B > G] = 2 * np.pi - H[B > G]
    H = H / (2 * np.pi)
    return H, S, I
rgb_image = Image.open('nature.jpeg')
H, S, I = rgb2hsi_components(rgb_image)
fig, ax = plt.subplots(2, 2, figsize=(10, 8))
ax[0, 0].imshow(rgb_image)
ax[0, 0].set_title('RGB Image')
ax[0, 0].axis('off')
ax[0, 1].imshow(H, cmap='hsv')
ax[0, 1].set_title('Hue')
ax[0, 1].axis('off')
ax[1, 0].imshow(S, cmap='gray')
ax[1, 0].set_title('Saturation')
ax[1, 0].axis('off')
ax[1, 1].imshow(I, cmap='gray')
```

```
ax[1, 1].set_title('Intensity')
ax[1, 1].axis('off')
plt.tight_layout()
plt.show()
```

Output:

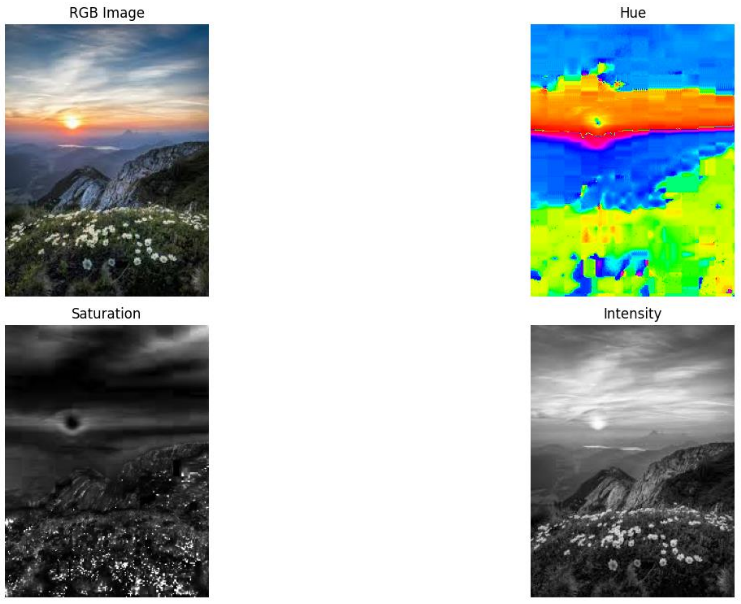


Figure 1.1: Converting RGB color model to HIS color model in python

Explanation:

1. We use the **PIL** library to load the RGB image. The image is normalized to the range [0, 1] by dividing by 255.
2. The **intensity (I)** is the average of the R, G, and B channels.
3. **Saturation (S)** is calculated based on the minimum of the R, G, and B values relative to the sum of the R, G, and B values.
4. **Hue (H)** is calculated using an arccosine formula that considers the relative differences between the R, G, and B channels.
5. We display the original RGB image, along with the Hue, Saturation, and Intensity components using **matplotlib's imshow()** function.

Code-02: MATLAB

```
function [H, S, I] = rgb2hsi_components(rgb_image)
    [rows, cols, ~] = size(rgb_image);
    H = zeros(rows, cols);
    S = zeros(rows, cols);
    I = zeros(rows, cols);
    rgb_image = im2double(rgb_image);
    R = rgb_image(:, :, 1);
    G = rgb_image(:, :, 2);
    B = rgb_image(:, :, 3);
    I = (R + G + B) / 3;
    min_RGB = min(min(R, G), B);
    S = 1 - (3 ./ (R + G + B + eps)) .* min_RGB;
    theta = acos((0.5 * ((R - G) + (R - B))) ./ sqrt((R - G).^2 + (R - B).*(G - B) + eps));
    H(B > G) = 2 * pi - theta(B > G);
    H(G >= B) = theta(G >= B);
    H = H / (2 * pi);
end
rgb_image = imread('nature.jpeg');
[H, S, I] = rgb2hsi_components(rgb_image);

figure;
subplot(2, 2, 1), imshow(rgb_image), title('RGB Image');
subplot(2, 2, 2), imshow(H), title('Hue');
subplot(2, 2, 3), imshow(S), title('Saturation');
subplot(2, 2, 4), imshow(I), title('Intensity');
```

Output:

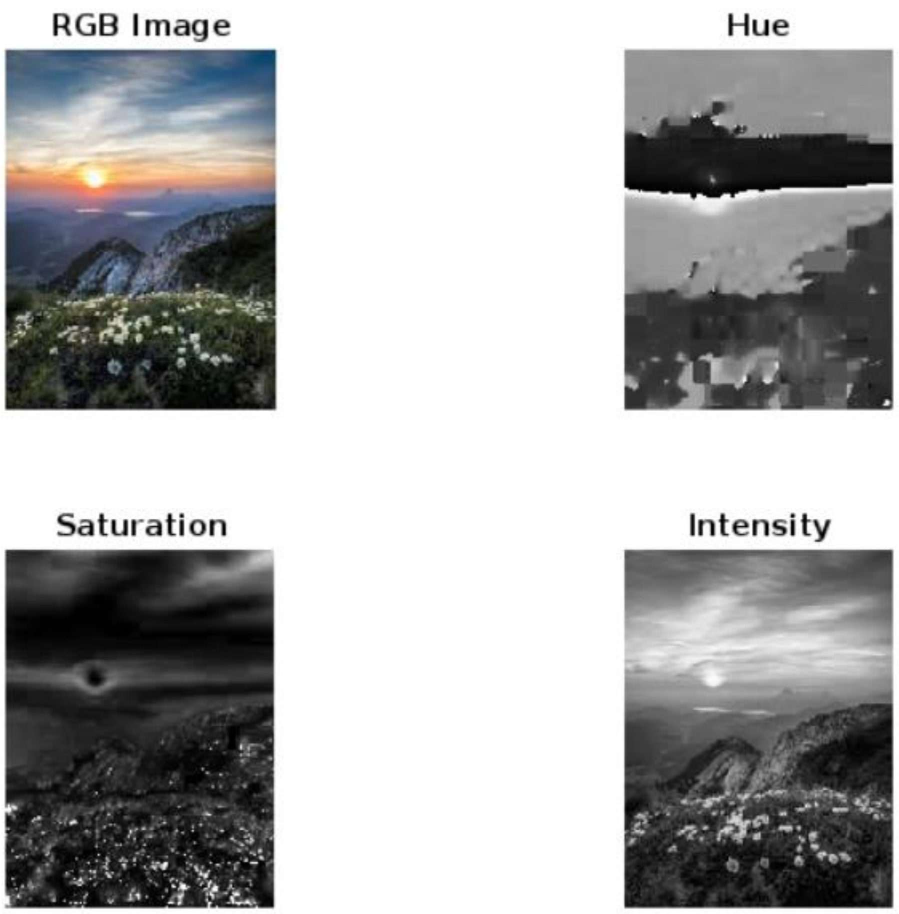


Figure 1.2: Converting RGB color model to HIS color model in MATLAB

Explanation:

1. The hue is calculated using the arccosine formula, adjusted based on the comparison of B and G values.
2. Saturation is calculated based on the minimum RGB value divided by the total sum of RGB components. It is scaled between 0 (no color saturation) and 1 (full color saturation).
3. Intensity is simply the average of the R, G, and B channels.

Experiment Number: 02

Experiment Name: Convert RGB to Grayscale

Objectives:

1. Understand the RGB Color Model.
2. Understand the concept of grayscale, where images contain only shades of gray (intensity values) ranging from black (0) to white (255), without any color information.
3. Optimize for Performance and Efficiency.

Code-01: Python

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
def rgb_components_to_grayscale(image):
   image = image.astype(np.float32) / 255.0
   R = image[:, :, 0]
   G = image[:, :, 1]
   B = image[:, :, 2]
   grayscale_image = 0.2989 * R + 0.5870 * G + 0.1140 * B
   plt.figure(figsize=(10, 8))
   plt.subplot(3, 2, 3)
   plt.imshow(R, cmap='gray')
   plt.title('Red Component')
   plt.axis('off')
   plt.subplot(3, 2, 4)
   plt.imshow(G, cmap='gray')
   plt.title('Green Component')
   plt.axis('off')
   plt.subplot(3, 2, 5)
   plt.imshow(B, cmap='gray')
   plt.title('Blue Component')
   plt.axis('off')
   plt.subplot(3, 2, 6)
   plt.imshow(grayscale_image, cmap='gray')
   plt.title('Grayscale Image')
   plt.axis('off')
   plt.tight_layout()
   plt.show()
image = cv2.imread('nature.jpeg')
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
```

rgb_components_to_grayscale(image)

Output:

Figure 2.1: Convert RGB to Grayscale python code

Explanation:

1. Importing Libraries such as cv2, numpy, matplotlib.pyplot
2. Defining the Function **rgb_components_to_grayscale(image)**
3. Plotting the Components and Grayscale Image

Code-02: MATLAB

```
function rgb_components_to_grayscale(image)

    image = double(image) / 255;
    R = image(:, :, 1);
    G = image(:, :, 2);
    B = image(:, :, 3);
    grayscale_image = 0.2989 * R + 0.5870 * G + 0.1140 * B;
    figure;

    subplot(3, 2, 2), imshow(R), title('Red Component');
    subplot(3, 2, 3), imshow(G), title('Green Component');
    subplot(3, 2, 4), imshow(B), title('Blue Component');
    subplot(3, 2, 5), imshow(grayscale_image), title('Grayscale Image');
    subplot(3, 2, 1), imshow(image), title('Original Image');
end
image = imread('nature.jpeg');
rgb_components_to_grayscale(image);
```

Output:

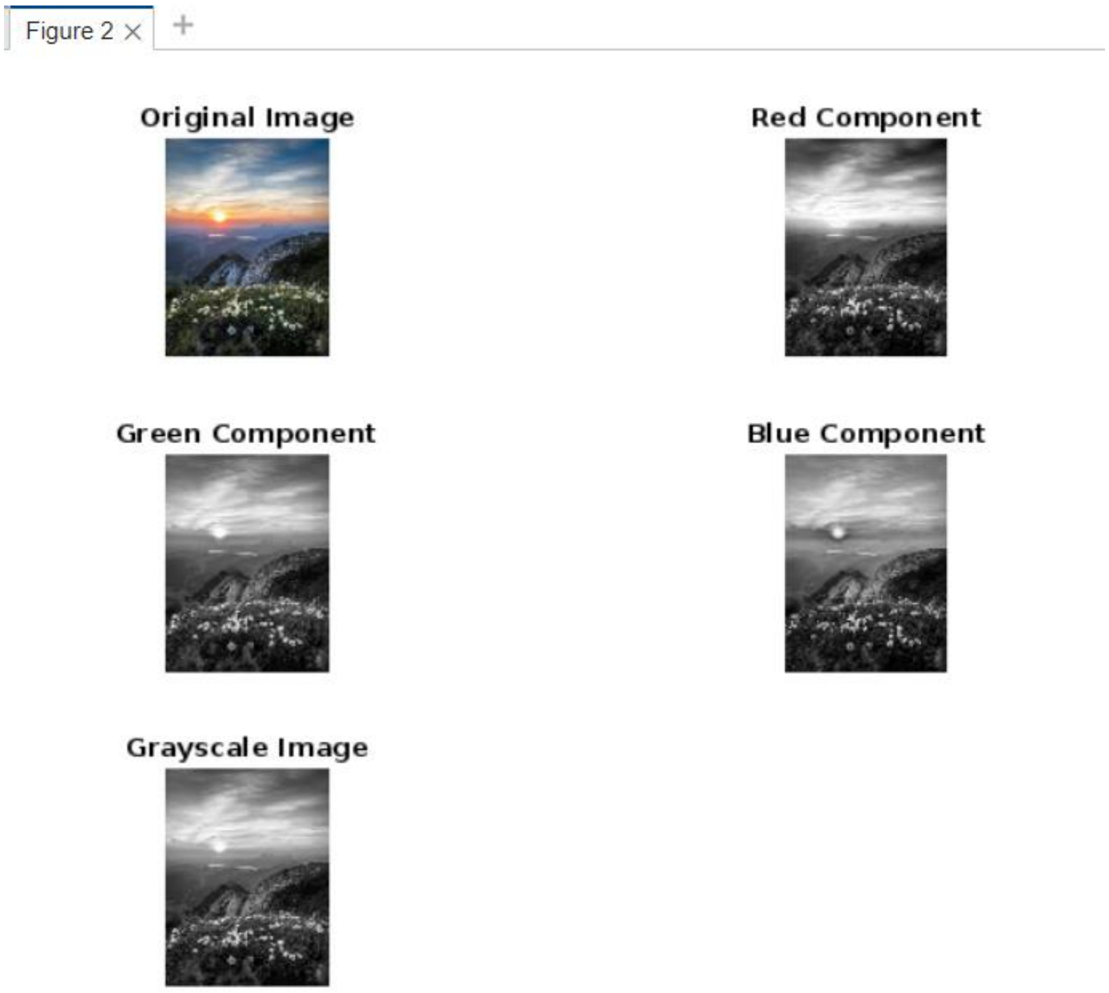


Figure 2.2: Converting the RGB model to Grayscale image in MATLAB

Explanation:

1. This defines a function called **rgb_components_to_grayscale** that takes an image (RGB image) as input.
2. The image is converted from uint8 (range 0-255) to double (range 0-1) by dividing by 255. This is done because many image processing operations work best with floating-point values between 0 and 1.
3. The image is a 3D matrix, where the third dimension contains the color channels.
4. This formula is used to convert the RGB image to grayscale. The weights are based on the human eye's sensitivity to different colors.

Lab Report: 07

Title: Morphological Operations and Fourier Transform of an Image

Course title: Digital Image Processing Laboratory

Course code: CSE-406

4th Year 1st Semester Examination 2023

Date of Submission: 20/10/2024

Submitted to-

Dr. Md. Golam Moazzam

Professor

Department of Computer Science and Engineering

Jahangirnagar University

&

Dr. Morium Akter

Professor

Department of Computer Science and Engineering

Jahangirnagar University

Savar, Dhaka-1342

Class Roll	Exam Roll	Name
353	202165	Shanjida Alam

Department of Computer Science and Engineering
Jahangirnagar University
Savar, Dhaka, Bangladesh

Experiment No: 01

Experiment Name: Dilation of an Image Using Morphological Operations

Objectives:

1. The objective of this experiment is to apply the dilation operation, which enlarges the boundaries of objects in an image.
2. Dilation is useful for emphasizing specific features, such as connecting broken parts of an object.

Code-01: Python

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
input_image = cv2.imread('nature.jpeg', cv2.IMREAD_GRAYSCALE)
kernel = np.ones((5, 5), np.uint8)
dilated_image = cv2.dilate(input_image, kernel, iterations=1)
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(input_image, cmap='gray')
plt.title('Original Grayscale Image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(dilated_image, cmap='gray')
plt.title('Dilated Grayscale Image')
plt.axis('off')
plt.show()
```

Output:

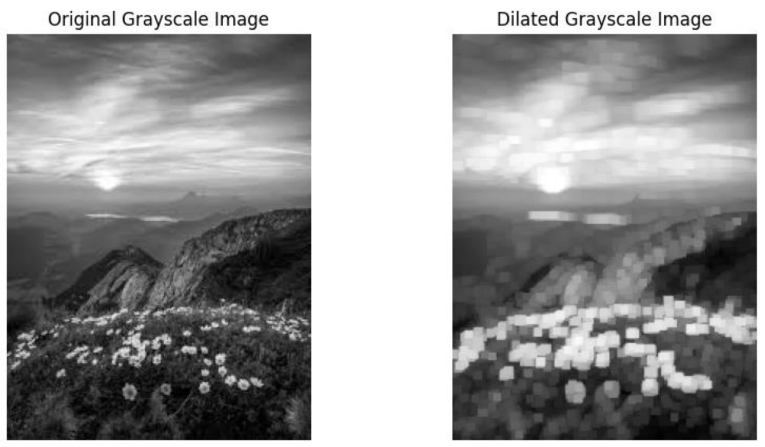


Figure 1.1: Dilation of an image using morphological operations in Python

Explanation:

1. This code reads an image in grayscale and applies a dilation operation using a 5x5 kernel.
2. The dilation process expands the white regions in the image, which emphasizes the boundaries of the objects.
3. The images before and after dilation are displayed side by side.

Code-02: MATLAB

```
inputImage = imread('nature.jpeg');
if size(inputImage, 3) == 3
   inputImage = rgb2gray(inputImage);
end
se = strel('square', 3);
dilatedImage = imdilate(inputImage, se);
figure;
subplot(1, 2, 1);
imshow(inputImage);
title('Original Image');
subplot(1, 2, 2);
imshow(dilatedImage);
title('Dilated Image');
```

Output:

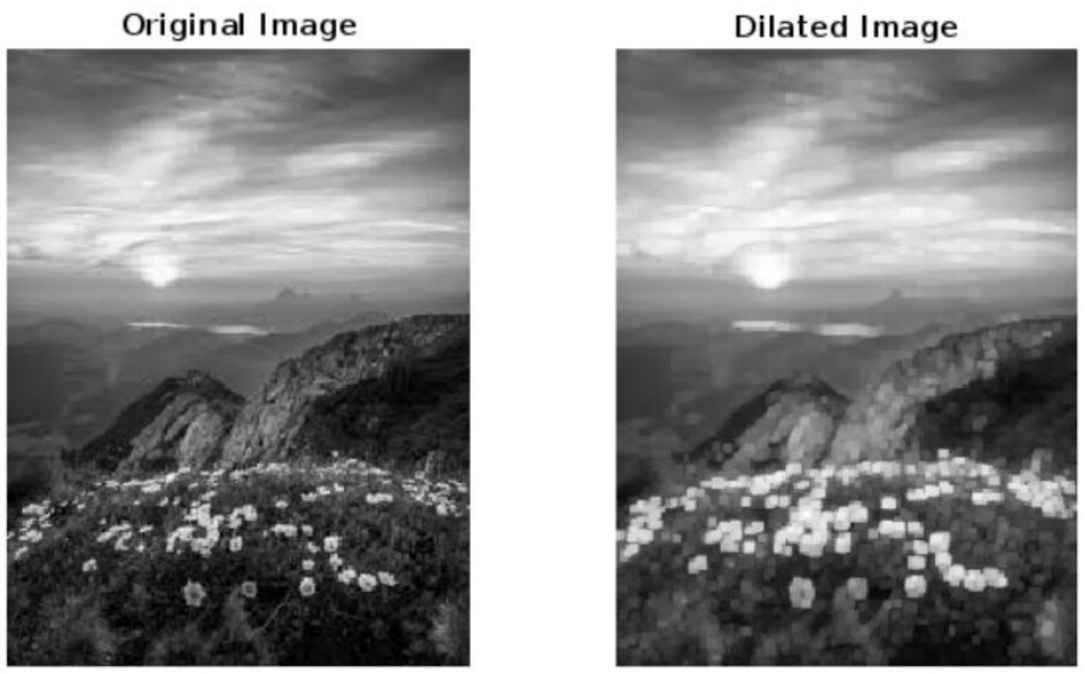


Figure 1.2: Dilation of an image using morphological operations in MATLAB

Explanation:

1. The code reads an input image and creates a 3x3 square structuring element (strel).
2. The imdilate() function is used to apply dilation, which expands object boundaries and fills gaps in the image.
3. The original and dilated images are displayed side by side using subplot for comparison.

Experiment No: 02

Experiment Name: Erosion Operation of an Image Using Morphological Transformations
Objectives:

1. The objective of this experiment is to apply the erosion operation to an image.
2. Erosion reduces the size of objects by eroding away the boundaries, which helps in eliminating small noise and separating objects that are close to each other.

Code-01: Python

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
input_image = cv2.imread('nature.jpeg', cv2.IMREAD_GRAYSCALE)
kernel = np.ones((3, 3), np.uint8)
eroded_image = cv2.erode(input_image, kernel, iterations=1)
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(input_image, cmap='gray')
plt.title('Original Image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(eroded_image, cmap='gray')
plt.title('Eroded Image')
plt.axis('off')
plt.show()
```

Output:

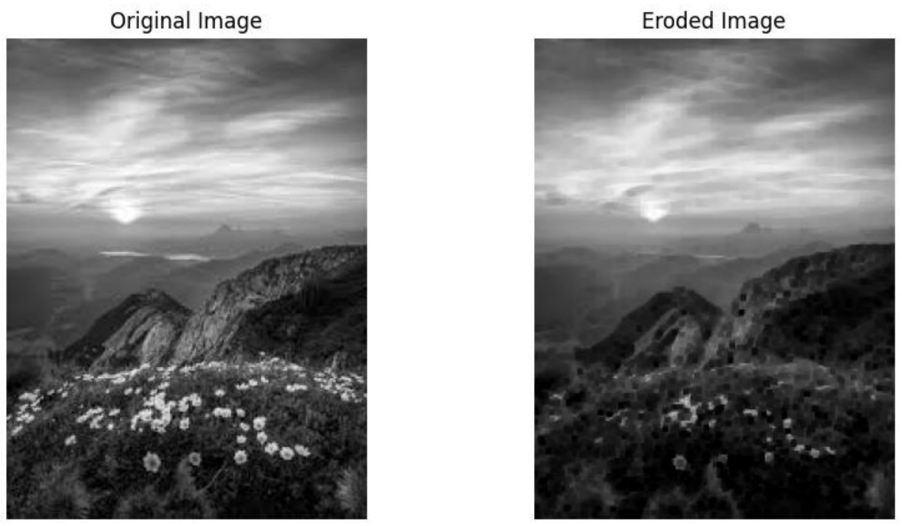


Figure 2.1: Erosion operation of an image in python

Explanation:

1. This code begins by loading an image in grayscale, followed by the creation of a structuring element (5x5 matrix) used for the erosion operation.
2. The cv2.erode() function is applied to shrink the object boundaries in the image. Erosion is effective in removing small artifacts and noise, and also separates objects that are connected by thin structures.
3. The original and eroded images are displayed side by side for comparison.

Code-02: MATLAB

```
inputImage = imread('nature.jpeg');
if size(inputImage, 3) == 3
    inputImage = rgb2gray(inputImage);
end
se = strel('square', 3);
erodedImage = imerode(inputImage, se);
figure;
subplot(1, 2, 1);
imshow(inputImage);
title('Original Image');
subplot(1, 2, 2);
imshow(erodedImage);
title('Eroded Image');
```

Output:

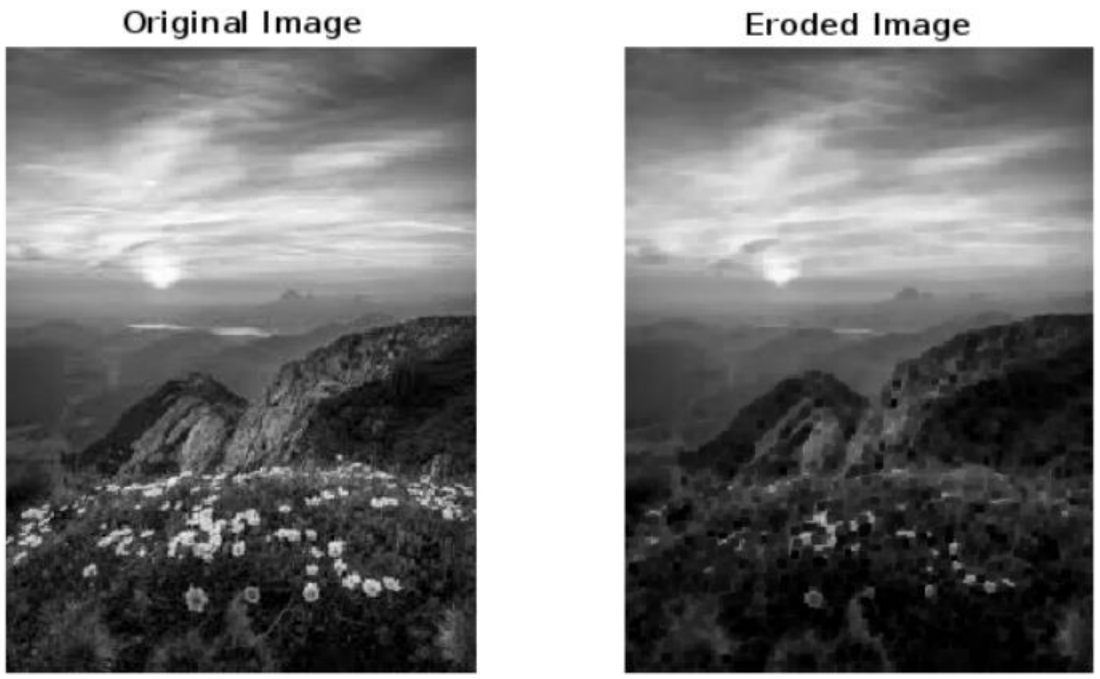


Figure 2.2: Erosion operation of an image in MATLAB

Explanation:

1. The image is read and a 3x3 square structuring element is created.
2. The imerode() function applies erosion to shrink the object boundaries in the image.
3. This operation is commonly used to eliminate small noise or detach objects that are connected.
4. The original and eroded images are displayed for visual comparison.

Experiment No: 03

Experiment Name: Opening and Closing Operations in Image Processing

Objectives:

1. This experiment aims to apply both opening and closing operations on an image.
2. Opening is used to remove small objects or noise from the image, while closing is used to fill small holes or gaps inside objects.

Code-01: Python

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
input_image = cv2.imread('nature.jpeg', cv2.IMREAD_GRAYSCALE)
kernel = np.ones((5, 5), np.uint8)
opened_image = cv2.morphologyEx(input_image, cv2.MORPH_OPEN, kernel)
closed_image = cv2.morphologyEx(input_image, cv2.MORPH_CLOSE, kernel)
plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.imshow(input_image, cmap='gray')
plt.title('Original Image')
plt.axis('off')
plt.subplot(1, 3, 2)
plt.imshow(opened_image, cmap='gray')
plt.title('Opened Image')
plt.axis('off')
plt.subplot(1, 3, 3)
plt.imshow(closed_image, cmap='gray')
plt.title('Closed Image')
plt.axis('off')
plt.show()
```

Output:

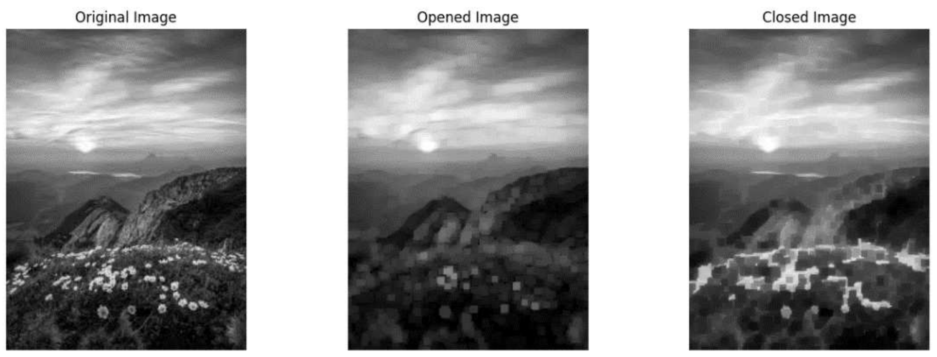


Figure 3.1: Opening and closing operations of an image using Python

Explanation:

1. This code uses cv2.morphologyEx() to perform both opening and closing operations on the image.
2. Opening involves erosion followed by dilation to remove small noise, while closing involves dilation followed by erosion to fill small gaps inside objects.
3. The input image is processed through both operations, and the results are displayed side by side with the original image for comparison.
4. The same structuring element is used for both opening and closing.

Code-02: MATLAB

```
inputImage = imread('nature.jpeg');
if size(inputImage, 3) == 3
    inputImage = rgb2gray(inputImage);
end
se = strel('square', 3);
openedImage = imopen(inputImage, se);
closedImage = imclose(inputImage, se);
figure;
subplot(1, 3, 1);
imshow(inputImage);
title('Original Image');
subplot(1, 3, 2);
imshow(openedImage);
title('Opened Image');
subplot(1, 3, 3);
imshow(closedImage);
title('Closed Image');
```

Output:

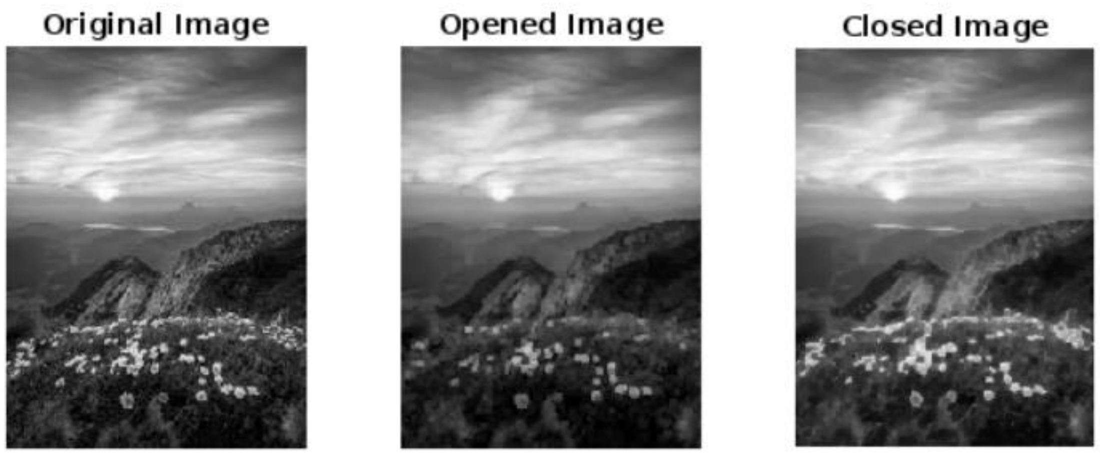


Figure 3.2: Opening and closing operations of an image using MATLAB

Explanation:

1. The input image is processed using both opening and closing operations.
2. Opening (via imopen()) helps remove small objects and noise, while closing (via imclose()) fills small holes and gaps in objects.
3. The structuring element for both operations is a 3x3 square.
4. The results are shown alongside the original image for a clear comparison of their effects.

Experiment No: 04

Experiment Name: Fast Fourier Transform (FFT) for Frequency Domain Analysis of an Image

Objectives:

1. The goal of this experiment is to apply the Fast Fourier Transform (FFT) on an image to convert it from the spatial domain to the frequency domain.
2. This technique helps to analyze the frequency components of the image, allowing for frequency-based processing.

Code-01: Python

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
input_image = cv2.imread('nature.jpeg', cv2.IMREAD_GRAYSCALE)
fft_image = np.fft.fft2(input_image)
fft_shifted = np.fft.fftshift(fft_image)
magnitude_spectrum = 20 * np.log(np.abs(fft_shifted) + 1)
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.imshow(input_image, cmap='gray')
plt.title('Original Image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(magnitude_spectrum, cmap='gray')
plt.title('Magnitude Spectrum (FFT)')
plt.axis('off')
plt.show()
```

Output:

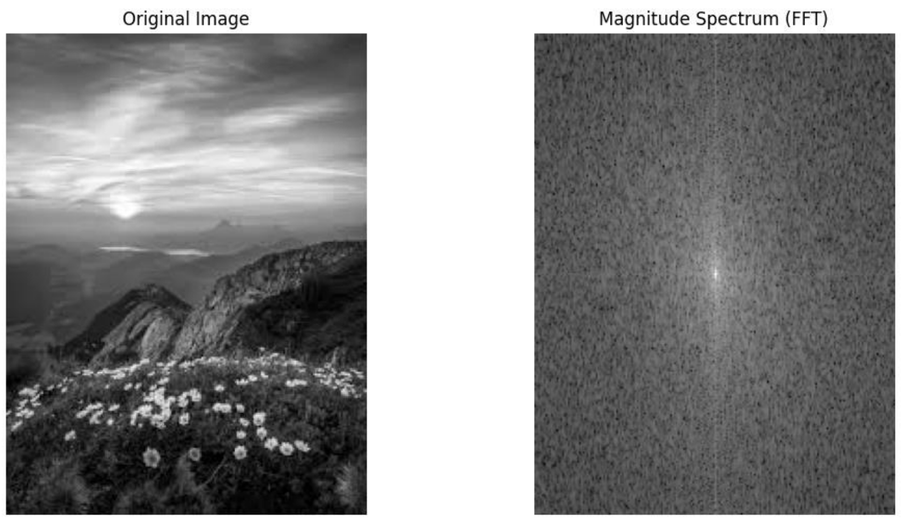


Figure 4.1: FFT of an image using python code

Explanation:

1. The code begins by reading an image in grayscale.
2. The np.fft.fft2() function computes the 2D Fast Fourier Transform (FFT) of the image, converting it to the frequency domain.
3. The np.fft.fftshift() function centers the low-frequency components. The magnitude spectrum of the FFT is then computed and displayed using a logarithmic scale to enhance visibility.
4. This technique allows for visualizing the frequency components of the image, highlighting areas with low and high-frequency content.

Code-02: MATLAB

```
inputImage = imread('nature.jpeg');
if size(inputImage, 3) == 3
    inputImage = rgb2gray(inputImage);
end
fftImage = fft2(double(inputImage));
fftImageShifted = fftshift(fftImage);
magnitudeFFT = abs(fftImageShifted);
magnitudeFFTLog = log(1 + magnitudeFFT);
figure;
subplot(1, 2, 1);
imshow(inputImage);
```

```
title('Original Image');
subplot(1, 2, 2);
imshow(magnitudeFFTLog, []);
title('Magnitude Spectrum (FFT)');
```

Output:

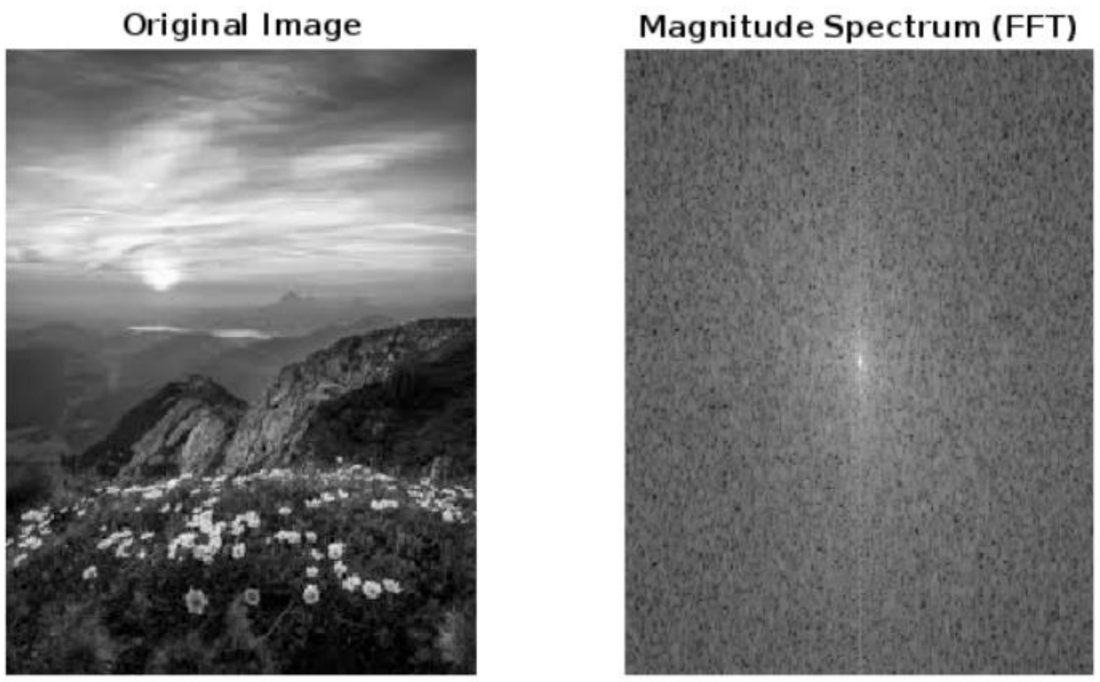


Figure 5.2: FFT of an image using MATLAB

Explanation:

1. The image is converted to double precision before applying the 2D Fast Fourier Transform using fft2().
2. The zero-frequency component is shifted to the center using fftshift().
3. The magnitude spectrum is computed using the logarithmic scale for better visualization.
4. The original image and its frequency representation are displayed side by side for comparison.

Experiment No: 05

Experiment Name: Homomorphic Transformation of an Image

Objectives:

1. The objective is to apply homomorphic filtering to enhance the contrast of an image by separating the illumination and reflectance components.
2. This method helps in improving the dynamic range of the image by enhancing details and reducing shadows.

Code-01: Python

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
def homomorphic_filter(input_image, low=0.5, high=2.0, D0=30):
    input_image = np.float32(input_image) / 255.0
    log_image = np.log1p(input_image)
    fft_image = np.fft.fft2(log_image)
    fft_shifted = np.fft.fftshift(fft_image)
    rows, cols = input_image.shape
    crow, ccol = rows // 2 , cols // 2
    H = np.zeros((rows, cols), np.float32)
    for u in range(rows):
        for v in range(cols):
            D = np.sqrt((u - crow) ** 2 + (v - ccol) ** 2)
            H[u, v] = (high - low) * (1 - np.exp((-D ** 2) / (2 * (D0 ** 2)))) + low
    filtered_fft = fft_shifted * H
    ifft_shifted = np.fft.ifftshift(filtered_fft)
    inverse_fft = np.fft.ifft2(ifft_shifted)
    filtered_image = np.real(inverse_fft)
    homomorphic_image = np.expm1(filtered_image)
    homomorphic_image = cv2.normalize(homomorphic_image, None, 0, 1,
cv2.NORM_MINMAX)
    return homomorphic_image
input_image = cv2.imread('nature.jpeg', cv2.IMREAD_GRAYSCALE)
filtered_image = homomorphic_filter(input_image)
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.imshow(input_image, cmap='gray')
plt.title('Original Image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(filtered_image, cmap='gray')
plt.title('Homomorphic Filtered Image')
plt.axis('off')
plt.show()
```

Output:

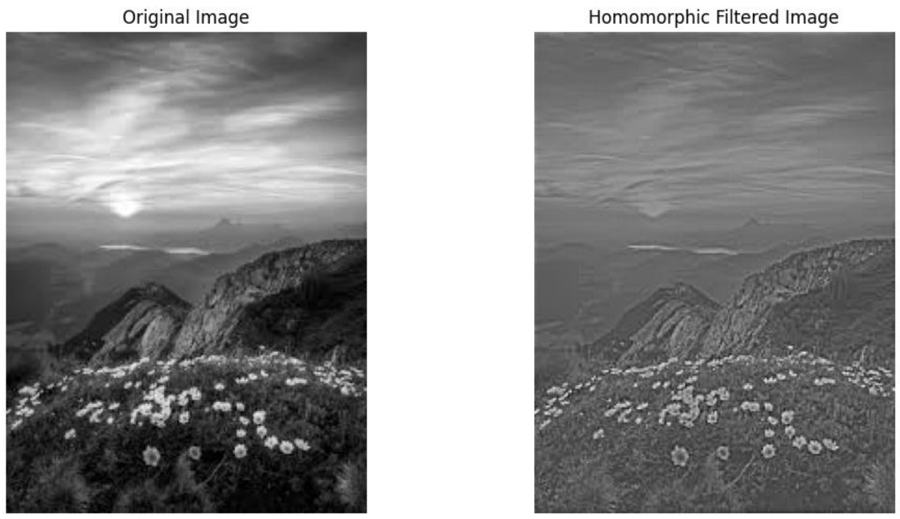


Figure 5.1: Homomorphic transform of an image using python code

Explanation:

1. The code first applies a logarithmic transformation to the image, which allows separating illumination and reflectance components.
2. Then, a high-pass filter is created and applied in the frequency domain using FFT.
3. The inverse FFT is used to transform the filtered image back into the spatial domain, and the exponential function is applied to reverse the log transformation.
4. The result is a contrast-enhanced image where shadowed regions are brightened, and overall dynamic range is improved.

Code-02: MATLAB

```
inputImage = imread('nature.jpeg');
if size(inputImage, 3) == 3
    inputImage = rgb2gray(inputImage);
end
inputImage = im2double(inputImage);
logImage = log(1 + inputImage);
fftImage = fft2(logImage);
fftImageShifted = fftshift(fftImage);
[M, N] = size(inputImage);
D0 = 30;
n = 2;
[X, Y] = meshgrid(1:N, 1:M);
centerX = ceil(N/2);
centerY = ceil(M/2);
D = sqrt((X - centerX).^2 + (Y - centerY).^2);
H = 1 - exp(-(D.^2 / (2 * D0^2)));
filteredFFT = fftImageShifted .* H;
filteredFFTShiftedBack = ifftshift(filteredFFT);
inverseFFT = ifft2(filteredFFTShiftedBack);
filteredImage = real(inverseFFT);
finalImage = exp(filteredImage) - 1;
figure;
subplot(1, 2, 1);
imshow(inputImage, []);
title('Original Image');
subplot(1, 2, 2);
imshow(finalImage, []);
title('Homomorphic Filtered Image');
```

Output:

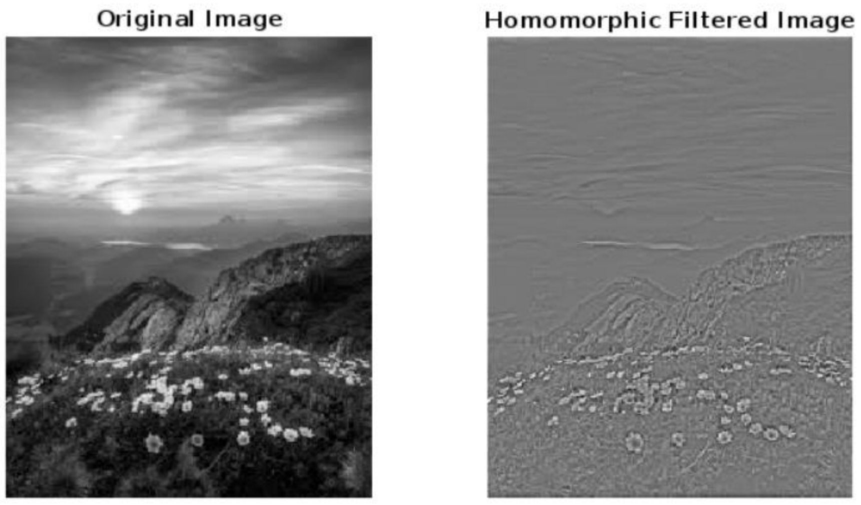


Figure 5.2: Homomorphic transform of an image using MATLAB

Explanation:

1. The image is first converted to double precision and a logarithmic transformation is applied to separate illumination and reflectance.
2. After performing FFT, a high-pass Gaussian filter is applied to suppress low-frequency components (illumination).
3. The filtered image is then converted back to the spatial domain using inverse FFT. The final result is an enhanced image with improved contrast, which is compared to the original image.

Lab Report: 08

Title: Noise Simulation and Removal Techniques

Course title: Digital Image Processing Laboratory

Course code: CSE-406

4th Year 1st Semester Examination 2023

Date of Submission: 27/10/2024

Submitted to-

Dr. Md. Golam Moazzam

Professor

Department of Computer Science and Engineering

Jahangirnagar University

&

Dr. Morium Akter

Professor

Department of Computer Science and Engineering

Jahangirnagar University

Savar, Dhaka-1342

Class Roll	Exam Roll	Name
353	202165	Shanjida Alam

Department of Computer Science and Engineering
Jahangirnagar University
Savar, Dhaka, Bangladesh

Experiment No: 01

Experiment Name: How will a image look after adding Gaussian, Rayleigh and Erlang noise

Objectives:

1. The primary objective of this code is to add different types of noise (Gaussian, Rayleigh, and Erlang) to an image and visualize their effects.
2. It can also be used to understand how various noise types affect image quality, a key concept in image processing for preparing data, filtering, and studying the robustness of image analysis algorithms.

Code-01: Python

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
image = cv2.imread('nature.jpeg', cv2.IMREAD_GRAYSCALE)
def add_gaussian_noise(image, mean=0, std=25):
   gauss = np.random.normal(mean, std, image.shape).astype('float32')
   noisy_image = image + gauss
   noisy_image = np.clip(noisy_image, 0, 255).astype('uint8')
   return noisy_image
def add_rayleigh_noise(image, scale=25):
   rayleigh = np.random.rayleigh(scale, image.shape).astype('float32')
   noisy_image = image + rayleigh
   noisy_image = np.clip(noisy_image, 0, 255).astype('uint8')
   return noisy_image
def add_erlang_noise(image, shape=2, scale=15):
   erlang = np.random.gamma(shape, scale, image.shape).astype('float32')
   noisy_image = image + erlang
   noisy_image = np.clip(noisy_image, 0, 255).astype('uint8')
   return noisy_image
gaussian_noisy_image = add_gaussian_noise(image)
rayleigh_noisy_image = add_rayleigh_noise(image)
erlang_noisy_image = add_erlang_noise(image)
plt.figure(figsize=(12, 8))
plt.subplot(2, 2, 1)
plt.title("Original Image")
plt.imshow(image, cmap='gray')
plt.axis('off')
plt.subplot(2, 2, 2)
plt.title("Gaussian Noise")
plt.imshow(gaussian_noisy_image, cmap='gray')
plt.axis('off')
plt.subplot(2, 2, 3)
plt.title("Rayleigh Noise")
plt.imshow(rayleigh_noisy_image, cmap='gray')
plt.axis('off')
plt.subplot(2, 2, 4)
plt.title("Erlang Noise")
plt.imshow(erlang_noisy_image, cmap='gray')
plt.axis('off')
plt.tight_layout()
plt.show()
```

Output:

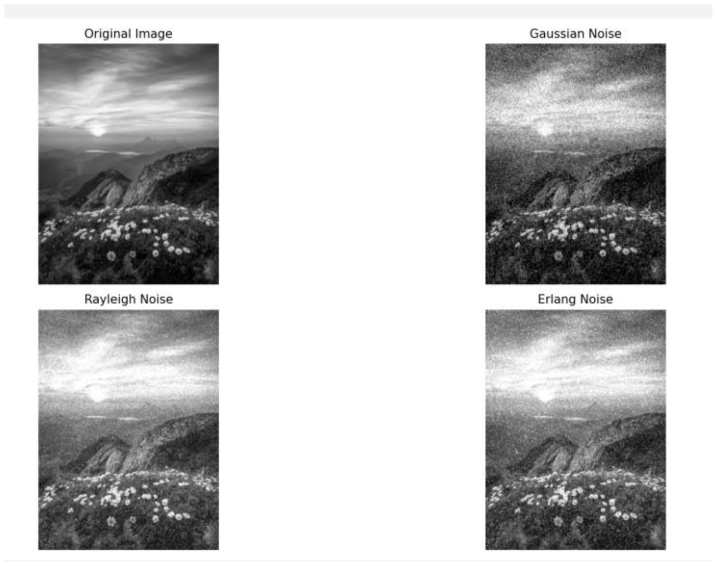


Figure 1.1: Adding Gaussian, Rayleigh and Erlang noise in Python

Explanation:

1. Import libraries.
2. Adds gaussian noise with a normal distribution (mean=0 and std=25).
3. The noise is generated using `np.random.normal` and then added to the original image.
4. Generated using the Rayleigh distribution (scale=25). This noise is added to the image similarly to the Gaussian noise, with clipping to maintain valid pixel values.
5. Adds noise using `np.random.gamma`(shape, scale).
6. Parameter's shape=2 and scale=15 define the characteristics of the Erlang distribution.

Code-02: MATLAB

```
image = imread('nature.jpeg');
image = rgb2gray(image);
gaussian_noisy_image = imnoise(image, 'gaussian', 0, 0.01);
rayleigh_noise = raylrnd(25, size(image));
rayleigh_noisy_image = double(image) + rayleigh_noise;
rayleigh_noisy_image = uint8(min(rayleigh_noisy_image, 255));
shape = 2;
scale = 15;
erlang_noise = gamrnd(shape, scale, size(image));
erlang_noisy_image = double(image) + erlang_noise;
erlang_noisy_image = uint8(min(erlang_noisy_image, 255));
figure;
subplot(2, 2, 1);
imshow(image);
title('Original Image');
subplot(2, 2, 2);
imshow(gaussian_noisy_image);
title('Gaussian Noise');
subplot(2, 2, 3);
imshow(rayleigh_noisy_image);
title('Rayleigh Noise');
subplot(2, 2, 4);
imshow(erlang_noisy_image);
title('Erlang Noise');
```

Output:

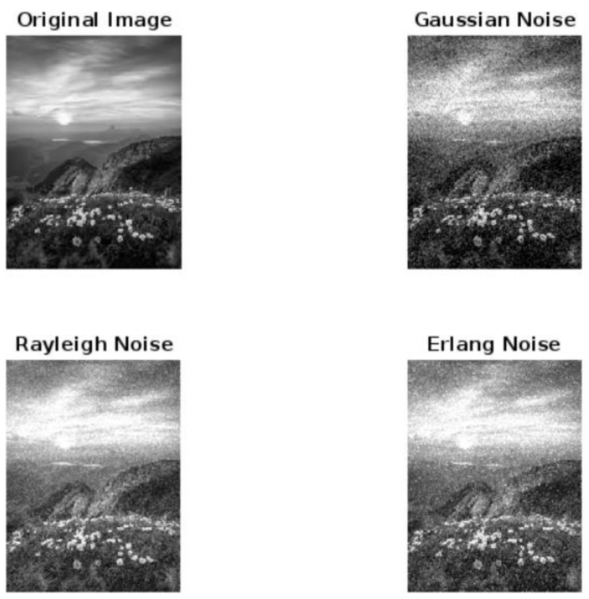


Figure 1.2: Adding Gaussian, Rayleigh and Erlang noise in Python

Explanation:

1. The original image (nature.jpeg) is loaded using `imread`. Since the analysis focuses on a grayscale version, the image is converted from RGB to grayscale using `rgb2gray`.
2. `imnoise` function is used to add Gaussian noise to the grayscale image. Mean (0) and variance (0.01) are specified to control the noise characteristics.
3. Rayleigh noise is generated using `raylrnd`, with a mode parameter of 25. The `size(image)` argument ensures the noise array has the same dimensions as the image.
4. Erlang noise (a type of Gamma noise) is added with `parameters shape = 2` and `scale = 15` using `gamrnd`, which simulates the noise distribution.

Experiment No: 02

Experiment Name: Noise Simulation (Gaussian Noise Adding and Removing)

Objectives:

1. To study and analyze the effects of different noise types—Gaussian, Rayleigh, and Erlang—on a grayscale image.
2. To understand the methods for generating and adding noise distributions to images.
3. To visualize and compare the impact of these noise distributions on image quality and appearance.

Code-01: Python

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
def add_gaussian_noise(image, mean=0, var=0.01):
   row, col = image.shape
   sigma = var ** 0.5
   gaussian = np.random.normal(mean, sigma, (row, col))
   noisy_image = image + gaussian * 255
   return np.clip(noisy_image, 0, 255).astype(np.uint8)
def arithmetic_mean_filter(image, kernel_size=3):
   return cv2.blur(image, (kernel_size, kernel_size)),
def geometric_mean_filter(image, kernel_size=3):
   image_float = image.astype(np.float32)
   img_log = np.log1p(image_float)
   kernel = np.ones((kernel_size, kernel_size), np.float32) / (kernel_size * kernel_size)
   log_filtered = cv2.filter2D(img_log, -1, kernel)
```

```
    geometric_mean_img = np.expm1(log_filtered)
    return np.clip(geometric_mean_img, 0, 255).astype(np.uint8)
image = cv2.imread('nature.jpeg', 0)
noisy_image = add_gaussian_noise(image)
arithmetic_filtered_image = arithmetic_mean_filter(noisy_image)
geometric_filtered_image = geometric_mean_filter(noisy_image)
plt.figure(figsize=(10, 5))
plt.subplot(1, 4, 1), plt.imshow(image, cmap='gray'), plt.title('Original Image')
plt.subplot(1, 4, 2), plt.imshow(noisy_image, cmap='gray'), plt.title('Noisy (Gaussian)')
plt.subplot(1, 4, 3), plt.imshow(arithmetic_filtered_image, cmap='gray'), plt.title('Arithmetic
Mean Filter')
plt.subplot(1, 4, 4), plt.imshow(geometric_filtered_image, cmap='gray'), plt.title('Geometric Mean
Filter')
plt.show()
```

Output:

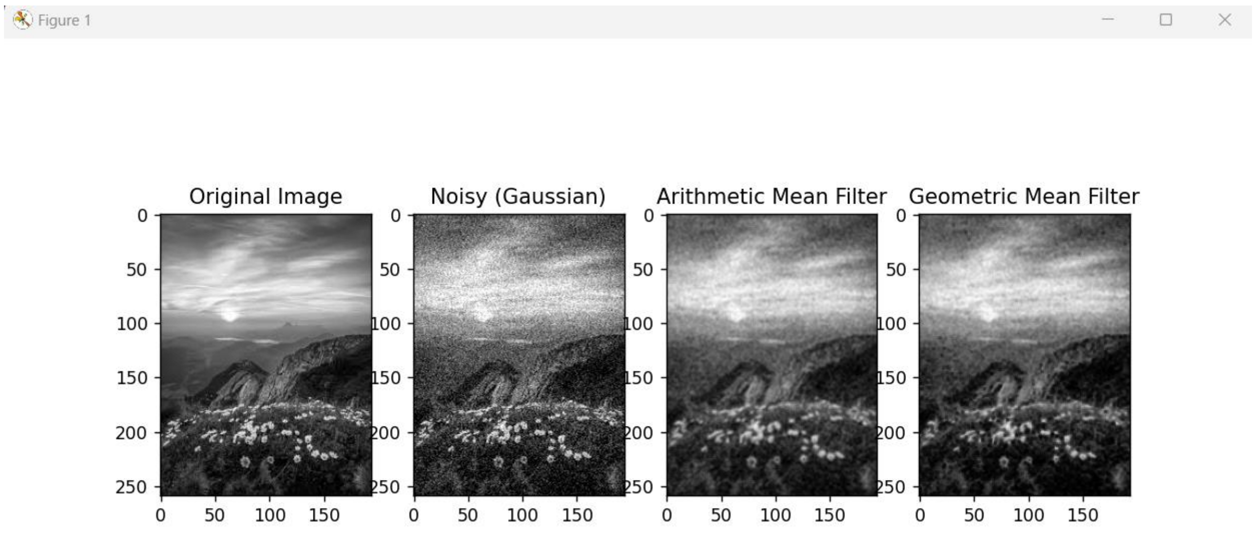


Figure 2.1: Noise Simulation (Gaussian Noise Adding and Removing) in Python code

Explanation:

1. Image Loading and Grayscale Conversion.
2. Gaussian noise is added using the `imnoise` function. Parameters are set to a mean of 0 and variance of `0.01`, adding subtle random intensity variations that simulate electronic noise.

3. Rayleigh noise is generated with a mode parameter of 25, creating an image-sized noise matrix. Adding this noise to the image requires converting it to double to prevent overflow. The pixel values are clipped and cast back to uint8 to retain the image format.
4. Erlang noise is generated using the gamma distribution with parameters `shape = 2` and `scale = 15`. The addition and clipping process mirrors that of Rayleigh noise.

Code-02: MATLAB

```
img = imread('nature.jpeg');
img = rgb2gray(img);
noisy_img = imnoise(img, 'gaussian', 0, 0.01);
arithmetic_mean_img = imfilter(noisy_img, fspecial('average', [3 3]));
[rows, cols] = size(noisy_img);
geo_img = double(noisy_img);
for i = 2:rows-1
    for j = 2:cols-1
        patch = geo_img(i-1:i+1, j-1:j+1);
        geo_mean = exp(mean(mean(log(double(patch) + 1))));
        geo_img(i,j) = geo_mean - 1;
    end
end
geo_img = uint8(geo_img);
figure;
subplot(2, 2, 1), imshow(img), title('Original Image');
subplot(2, 2, 2), imshow(noisy_img), title('Noisy (Gaussian) Image');
subplot(2, 2, 3), imshow(arithmetic_mean_img), title('Arithmetic Mean Filtered
Image');
subplot(2, 2, 4), imshow(geo_img), title('Geometric Mean Filtered Image');
```

Output:

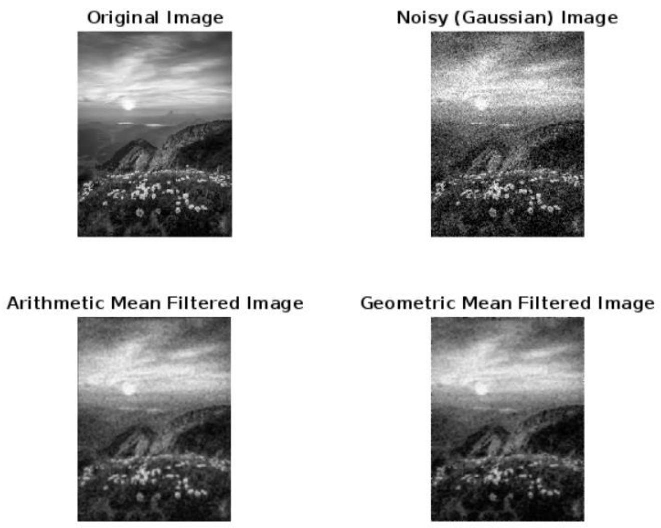


Figure 2.2: Noise Simulation (Gaussian Noise Adding and Removing) in MATLAB

Explanation:

1. Image Loading and Conversion to Grayscale.
2. Gaussian noise is added to the grayscale image using `imnoise`. The mean (0) and variance (0.01) parameters control the amount and spread of the noise. The result is a noisy version of the `image (noisy_img),` simulating random variations in pixel intensities.
3. This line applies an arithmetic mean filter (also known as an averaging filter) to reduce noise`. fspecial('average', [3 3])` creates a 3x3 averaging filter, and imfilter applies this filter to `noisy_img`, resulting in a smoothed image (`arithmetic_mean_img`). This method reduces noise by averaging pixel values in a 3x3 neighborhood, which helps reduce sharp noise variations.
4. Here, the code initializes `geo_img` as a copy of noisy_img in double format (to accommodate non-integer calculations) and determines the size of the image.

Experiment No: 03

Experiment Name: Noise Simulation (Periodic Noise Adding and Removing)

Objectives:

1. The objective of this experiment is to understand the effects of Periodic noise on images and to explore different filtering techniques, namely the arithmetic mean filter and the geometric mean filter, for noise reduction.
2. The experiment aims to evaluate the effectiveness of each filtering method and compare the results visually.

Code-01: Python

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
image = cv2.imread('nature.jpeg', 0)
dft = np.fft.fft2(image)
dft_shift = np.fft.fftshift(dft)
magnitude_spectrum = 20 * np.log(np.abs(dft_shift))
def notch_filter(dft_shift, center, radius=10):
    rows, cols = dft_shift.shape
    crow, ccol = int(rows / 2), int(cols / 2)
    mask = np.ones((rows, cols), np.uint8)
    cv2.circle(mask, center, radius, 0, thickness=-1)
    fshift = dft_shift * mask
    return fshift
filtered_dft = notch_filter(dft_shift, (130, 130), radius=15)
filtered_dft = notch_filter(filtered_dft, (170, 170), radius=15)
f_ishift = np.fft.ifftshift(filtered_dft)
img_back = np.fft.ifft2(f_ishift)
img_back = np.abs(img_back)
plt.figure(figsize=(12, 6))
plt.subplot(1, 3, 1), plt.imshow(image, cmap='gray'), plt.title('Original Image')
plt.subplot(1, 3, 2), plt.imshow(magnitude_spectrum, cmap='gray'), plt.title('Magnitude
Spectrum')
plt.subplot(1, 3, 3), plt.imshow(img_back, cmap='gray'), plt.title('Filtered Image')
plt.show()
```

Output:

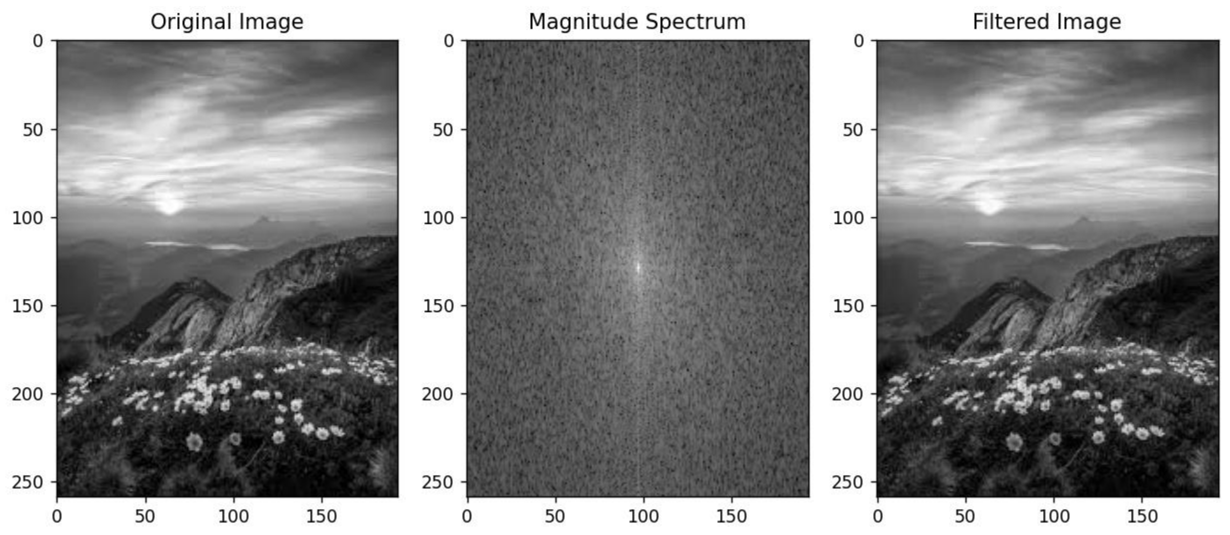


Figure 3.1: Noise Simulation (Periodic Noise Adding and Removing) using Python

Explanation:

1. Image Loading and Conversion to Grayscale.
2. Gaussian noise is added to the grayscale image with a mean of 0 and a variance of 0.01. This simulates random variations in pixel intensity, creating a noisy version of the image.

3. An arithmetic mean filter is applied using a 3x3 averaging filter kernel. This filter reduces noise by averaging the values in a 3x3 neighborhood around each pixel, producing a smoother, less noisy image (`arithmetic_mean_img`).
4. Initializes `geo_img` as a copy of `noisy_img` with a double type to accommodate precise calculations.
5. Converts `geo_img` back to `uint8` format to be compatible with image display functions.

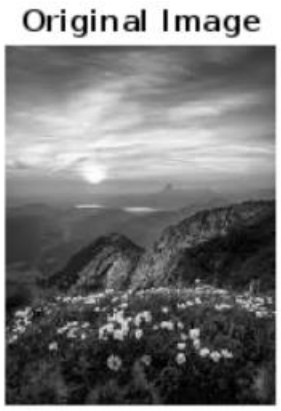
Code-02: MATLAB

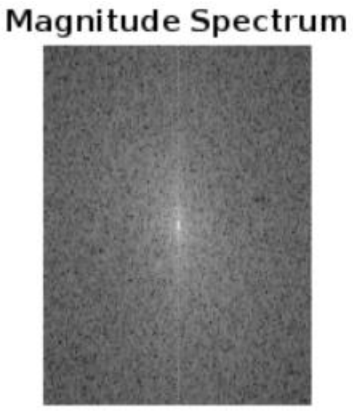
```
img = imread('nature.jpeg');
img = rgb2gray(img);
F = fft2(double(img));
F_shifted = fftshift(F);
magnitude_spectrum = log(1 + abs(F_shifted));
figure;
subplot(2, 2, 1), imshow(img, []), title('Original Image');
subplot(2, 2, 2), imshow(magnitude_spectrum, []), title('Magnitude Spectrum');
function F_filtered = notch_filter(F_shifted, center, radius)
    [rows, cols] = size(F_shifted);
    mask = ones(rows, cols);
    [X, Y] = meshgrid(1:cols, 1:rows);

    mask(((X - center(2)).^2 + (Y - center(1)).^2) < radius^2) = 0;

    F_filtered = F_shifted .* mask;
end
F_filtered = notch_filter(F_shifted, [130, 130], 15);
F_filtered = notch_filter(F_filtered, [170, 170], 15);
F_ishifted = ifftshift(F_filtered);
img_filtered = ifft2(F_ishifted);
img_filtered = abs(img_filtered);
subplot(2, 2, 3), imshow(img_filtered, []), title('Filtered Image');
```

Output:





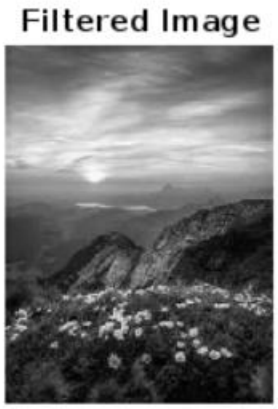


Figure 3.2 Noise Simulation (Periodic Noise Adding and Removing) using MATLAB

Explanation:

1. Image Loading and Grayscale Conversion.
2. The fft2 function is used to compute the 2D Fourier Transform of the image (`img`), converting it from the spatial domain to the frequency domain. This results in F, a complex-valued matrix representing the image in the frequency domain.
3. The `fftshift` function is applied to `F` to shift the zero-frequency component to the center of the spectrum. This makes it easier to visualize and manipulate specific frequencies in the frequency domain.
4. The magnitude of the frequency components is calculated by taking the absolute value of `F_shifted`. Applying `log(1 + abs(F_shifted))` enhances the visibility of the

spectrum by reducing the dynamic range, which helps in visualizing lower-intensity frequencies.

Experiment No: 04

Experiment Name: Noise Simulation (Salt and Pepper Noise Adding and Removing)
Objectives:

1. Add artificial salt-and-pepper noise to a grayscale image.
2. Use a median filter to remove the salt-and-pepper noise.
3. Visualize the original, noisy, and denoised images side-by-side for comparison.

Code-01: Python

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
def add_salt_and_pepper_noise(image, salt_prob=0.05, pepper_prob=0.05):
    noisy_image = np.copy(image)
    num_salt = np.ceil(salt_prob * image.size)
    coords = [np.random.randint(0, i - 1, int(num_salt)) for i in image.shape]
    noisy_image[coords[0], coords[1]] = 255
    num_pepper = np.ceil(pepper_prob * image.size)
    coords = [np.random.randint(0, i - 1, int(num_pepper)) for i in image.shape]
    noisy_image[coords[0], coords[1]] = 0
    return noisy_image
def remove_salt_and_pepper_noise(image, kernel_size=3):
    denoised_image = cv2.medianBlur(image, kernel_size)
    return denoised_image
image = cv2.imread('nature.jpeg', 0)
noisy_image = add_salt_and_pepper_noise(image)
denoised_image = remove_salt_and_pepper_noise(noisy_image)
plt.figure(figsize=(10, 5))
plt.subplot(1, 3, 1), plt.imshow(image, cmap='gray'), plt.title('Original')
plt.subplot(1, 3, 2), plt.imshow(noisy_image, cmap='gray'), plt.title('Noisy (Salt & Pepper)')
plt.subplot(1, 3, 3), plt.imshow(denoised_image, cmap='gray'), plt.title('Denoised (Median Filter)')
plt.show()
```

Output:

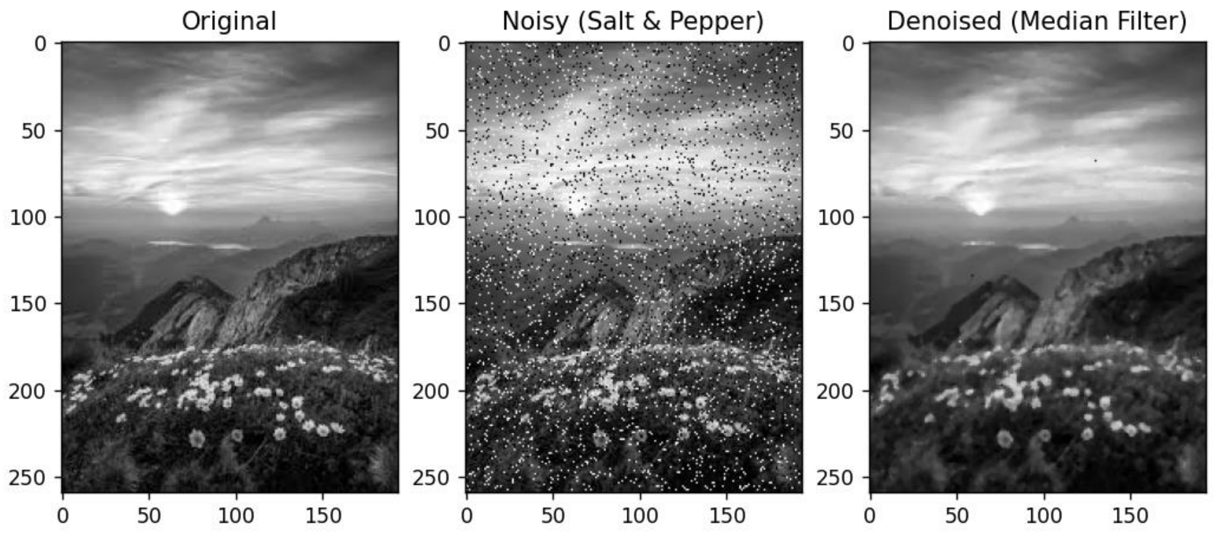


Figure 4.1: Noise Simulation (Salt and Pepper Noise Adding and Removing) using python code

Explanation:

1. The necessary libraries are imported, including cv2 for image processing, numpy for numerical operations, and matplotlib.pyplot for visualization.
2. Randomly selects pixel coordinates to `add salt (white pixels) and pepper (black pixels)`, using `np.random.randint`.
3. Takes the noisy image and a `kernel_size` parameter.
4. This technique allows for visualizing the frequency components of the image, highlighting areas with low and high-frequency content.

Code-02: MATLAB

```
img = imread('nature.jpeg');
img = rgb2gray(img);
noisy_img = imnoise(img, 'salt & pepper', 0.05);
denoised_img = medfilt2(noisy_img, [3 3]);
figure;
subplot(2, 2, 1), imshow(img), title('Original Image');
subplot(2, 2, 2), imshow(noisy_img), title('Noisy (Salt & Pepper) Image');
subplot(2, 2, 4), imshow(denoised_img), title('Denoised (Median Filter) Image');
subplot(1, 2, 2);
imshow(magnitudeFFTLog, []);
title('Magnitude Spectrum (FFT)');
```

Output:

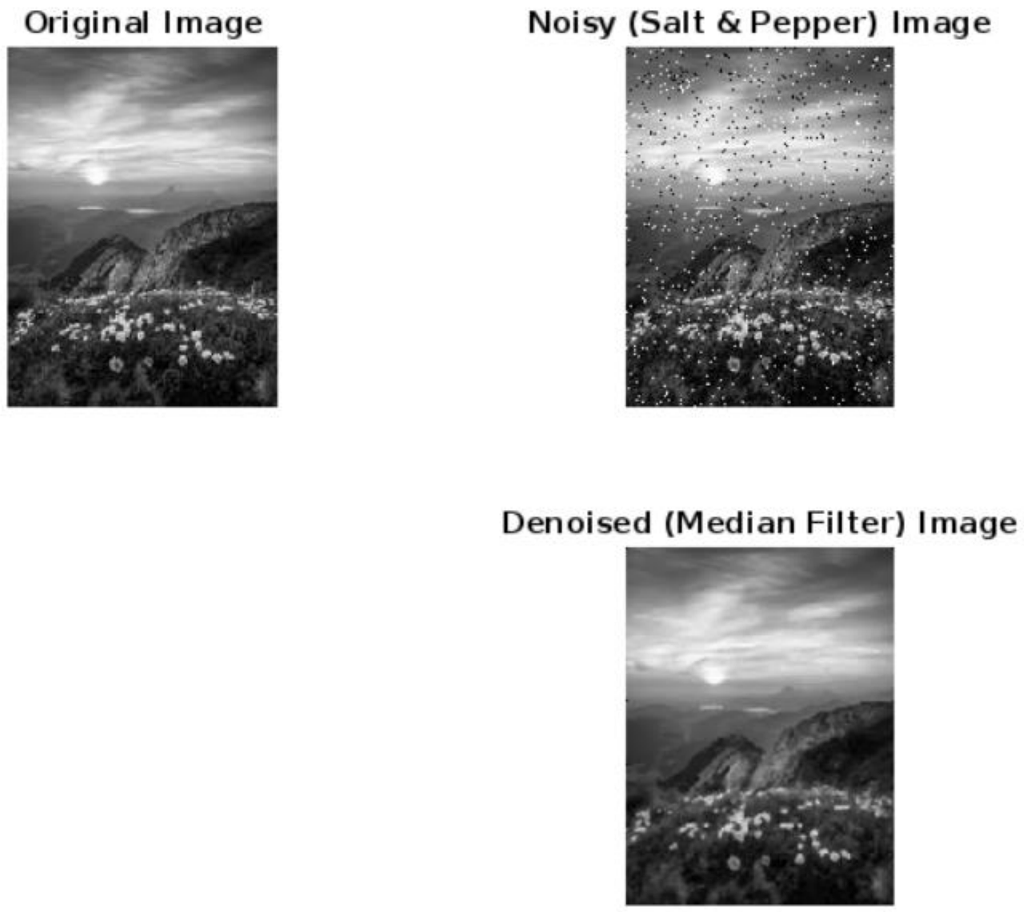


Figure 5.2: Noise Simulation (Salt and Pepper Noise Adding and Removing) using MATLAB

Explanation:

1. Image Reading and Grayscale Conversion.
2. `noisy_img = imnoise(img, 'salt & pepper', 0.05);` adds salt-and-pepper noise to the grayscale image with a noise density of 0.05. This means 5% of the pixels are randomly set to either black (pepper) or white (salt), simulating noise that resembles dust or scratches on a photo.
3. denoised_img = medfilt2(noisy_img, [3 3]); applies a median filter of size 3x3 to the noisy image. The median filter is effective in reducing salt-and-pepper noise, as it replaces each pixel's intensity with the median of the intensities in its local 3x3 neighborhood, removing isolated black and white spots.

Lab Report: 09

Title: Line and Circle Detection using Hough Transform Algorithm

Course title: Digital Image Processing Laboratory
Course code: CSE-406
4th Year 1st Semester Examination 2023

Date of Submission: 03/11/2024

Submitted to-

Dr. Md. Golam Moazzam
Professor
Department of Computer Science and Engineering
Jahangirnagar University
&
Dr. Morium Akter
Professor
Department of Computer Science and Engineering
Jahangirnagar University
Savar, Dhaka-1342

Class Roll	Exam Roll	Name
353	202165	Shanjida Alam

Experiment No: 01

Experiment Name: Straight Line Detection using Hough Transform Algorithm

Objectives:

1. The objective of this lab is to demonstrate line detection in an image using the Hough Transform in Python with OpenCV.
2. Line detection is essential in computer vision applications for identifying boundaries, lane detection in self-driving cars, and various other applications where linear structures need to be identified.
3. This lab includes preprocessing the image, applying edge detection, detecting lines with the Hough Transform, and visualizing the results.

Code-01: Python

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
image = cv2.imread('road.jpg')
inputImage = cv2.imread('road.jpg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
edges = cv2.Canny(gray, 50, 150, apertureSize=3)
lines = cv2.HoughLines(edges, 1, np.pi / 180, 200)
if lines is not None:
    for line in lines:
        rho, theta = line[0]
        a = np.cos(theta)
        b = np.sin(theta)
        x0 = a * rho
        y0 = b * rho
        x1 = int(x0 + 1000 * (-b))
        y1 = int(y0 + 1000 * (a))
        x2 = int(x0 - 1000 * (-b))
        y2 = int(y0 - 1000 * (a))

        cv2.line(image, (x1, y1), (x2, y2), (0, 0, 255), 2)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title("Detected Lines")
plt.axis('off')
plt.show()
```

Output:

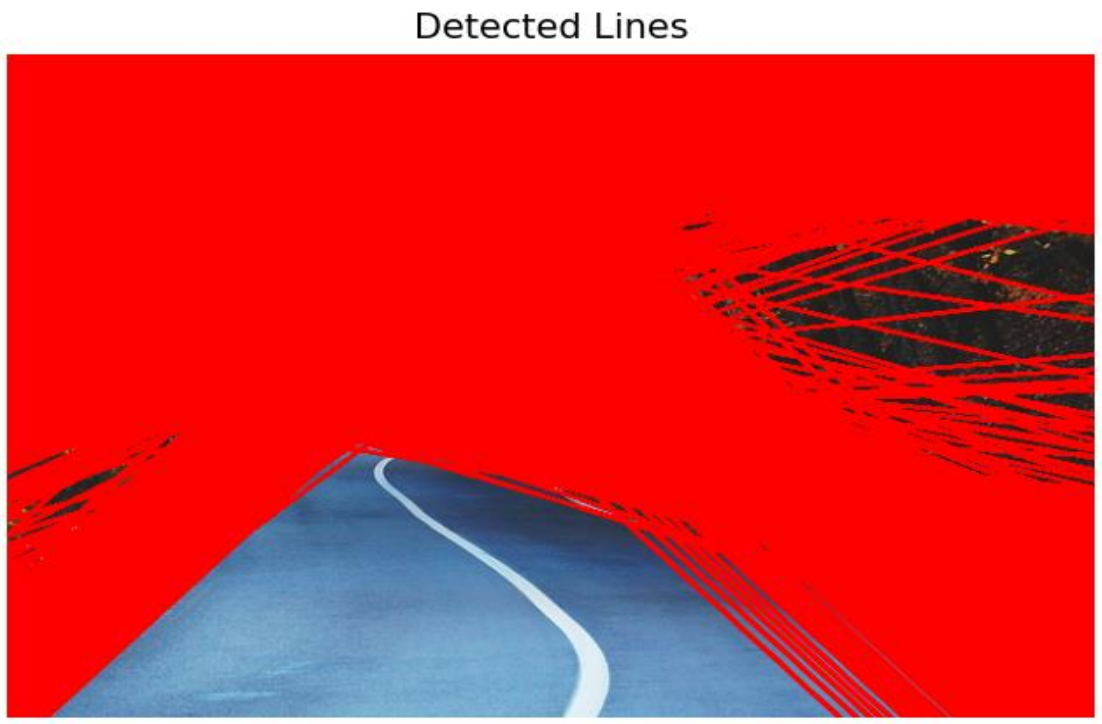


Figure 1.1: Line detection by Hough transform in Python

Explanation:

1. Import libraries.
2. `cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)` converts the image from color (BGR) to grayscale, which is a typical preprocessing step for edge detection and reduces computational load.
3. `cv2.Canny(gray, 50, 150, apertureSize=3)` applies Canny edge detection to the grayscale image. Canny edge detection detects edges in the image based on intensity gradients and returns a binary image with detected edges highlighted.
4. `cv2.HoughLines(edges, 1, np.pi / 180, 200)` is used to detect lines in the edge-detected image.
5. `cv2.line(image, (x1, y1), (x2, y2), (0, 0, 255), 2)` draws the detected line in red ((0, 0, 255)) with a thickness of 2.
6. `plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))` is used to display the image with detected lines in the red color. cv2.cvtColor converts the image from BGR to RGB for correct color display with Matplotlib.

Code-02: MATLAB

```matlab
image = imread('flower.jpeg');
gray = rgb2gray(image);
edges = edge(gray, 'Canny');
[H, T, R] = hough(edges);
P = houghpeaks(H, 5, 'threshold', ceil(0.3 * max(H(:))));
lines = houghlines(edges, T, R, P, 'FillGap', 20, 'MinLength', 40);
figure, imshow(image), hold on
for k = 1:length(lines)
    xy = [lines(k).point1; lines(k).point2];
    plot(xy(:, 1), xy(:, 2), 'LineWidth', 2, 'Color', 'red');
    plot(xy(1, 1), xy(1, 2), 'x', 'LineWidth', 2, 'Color', 'yellow');
    plot(xy(2, 1), xy(2, 2), 'x', 'LineWidth', 2, 'Color', 'green');
end
title('Detected Lines using Hough Transform');
hold off;
```

Output:

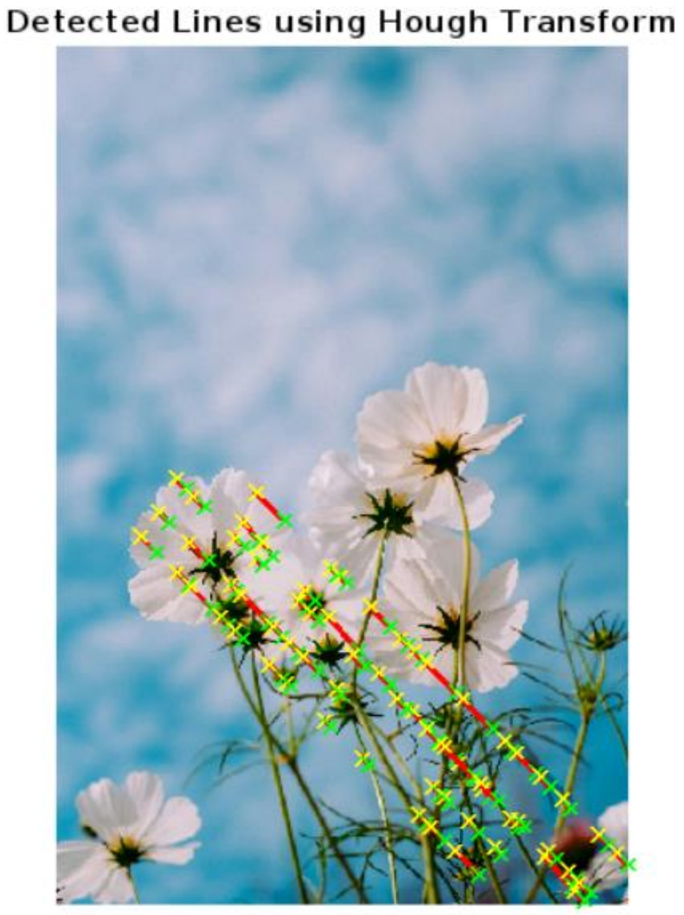


Figure 1.2: Line detection by Hough transform in MTLAB

Explanation:

1. The `edge()` function with the `'Canny'` method detects edges in the grayscale image. Canny edge detection is a popular method for finding edges in an image by identifying areas with strong intensity gradients.
2. The `hough()` function applies the Hough Transform to the edge-detected image, producing a Hough matrix `H`, and vectors `T (theta)` and `R (rho).`
3. `houghpeaks()` identifies the most significant peaks in the Hough matrix `H`, which represent potential lines.
4. `FillGap` specifies the maximum gap between line segments to connect them into a single line, while `MinLength` sets the minimum length for a line to be considered.

Experiment No: 02

Experiment Name: Circle Detection using Hough Transform

Objectives:

1. The purpose of this lab is to demonstrate circle detection in an image using the Hough Transform with OpenCV in Python.
2. Detecting circular shapes is essential in various applications, including object tracking, detecting traffic signs, and analyzing medical images.
3. In this lab, we preprocess an image, apply a circular Hough Transform, and visualize the detected circles on the image.

Code-01: Python

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
image = cv2.imread('road.jpg')
inputImage = cv2.imread('road.jpg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
gray = cv2.GaussianBlur(gray, (9, 9), 2)
circles = cv2.HoughCircles(
    gray,
    cv2.HOUGH_GRADIENT,
    dp=1.2,
    minDist=30,
    param1=50,
    param2=30,
    minRadius=10,
    maxRadius=80
)
if circles is not None:
    circles = np.uint16(np.around(circles))
    for i in circles[0, :]:
        center = (i[0], i[1])
        radius = i[2]
        cv2.circle(image, center, radius, (0, 255, 0), 3)
        cv2.circle(image, center, 3, (0, 0, 255), 3)
plt.figure(figsize=(10, 10))
plt.subplot(2, 2, 1)
plt.imshow(cv2.cvtColor(inputImage, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.axis('off')
plt.subplot(2, 2, 2)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title("Detected Circles")
plt.axis('off')
plt.show()
```

Output:

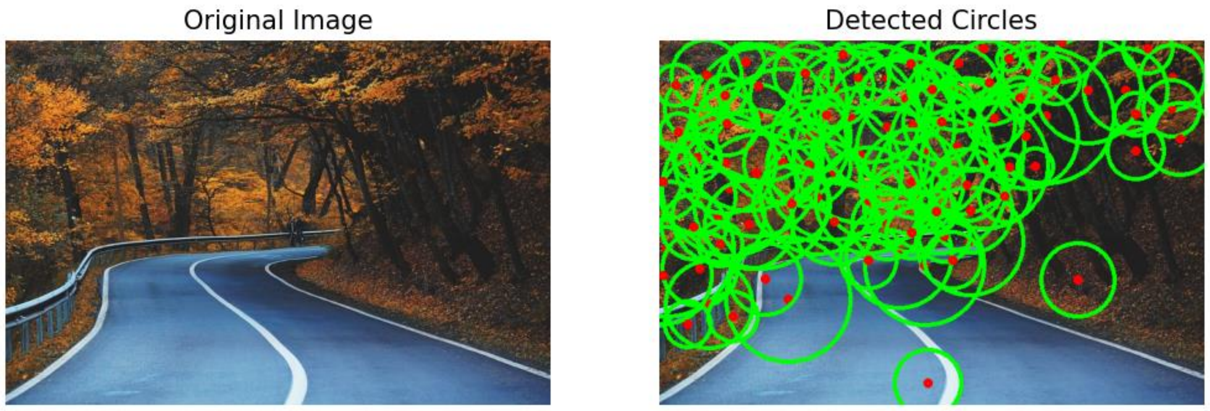


Figure 2.1: Circle Detection by Hough Transform in python

Explanation:

1. The code imports cv2 for image processing, `numpy` for numerical operations, and `matplotlib.pyplot` for displaying images.
2. The image is converted to grayscale using `cv2.cvtColor()`, a standard preprocessing step to reduce the computational load by eliminating color information.

3. A Gaussian blur with a `kernel size of (9, 9)` and standard deviation 2 is applied to smooth the image. This step helps reduce noise, making circle detection more accurate by blurring out minor image details.
4. The `cv2.HoughCircles()` function applies the Hough Circle Transform to detect circular shapes.
5. If any circles are detected, their properties are rounded and converted to integers using `np.uint16(np.around(circles))`.

Code-02: MATLAB

```
image = imread('flower.jpeg');
gray = rgb2gray(image);
[centers, radii, metric] = imfindcircles(gray, [20 80], 'Sensitivity', 0.9,
'EdgeThreshold', 0.1);
imshow(image);
hold on;
viscircles(centers, radii, 'EdgeColor', 'r');
title('Detected Circles using Hough Transform');
hold off;
```

Output:

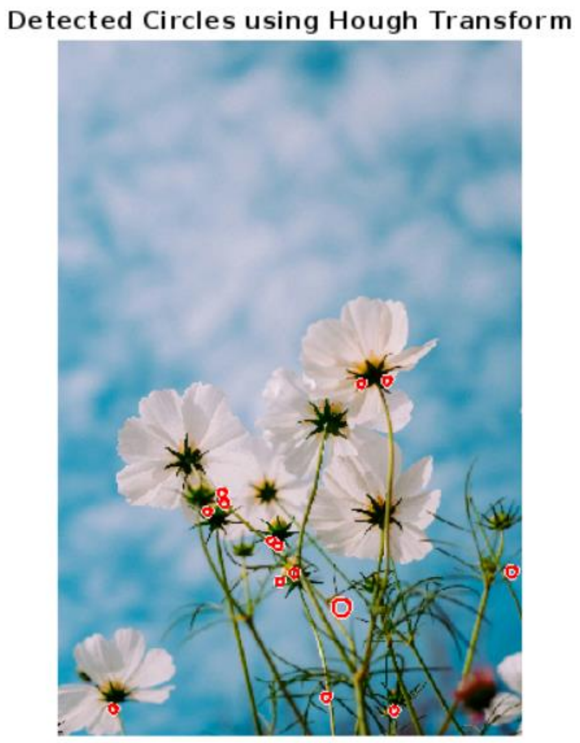


Figure 2.2: Circle Detection by Hough Transform in MATLAB

Explanation:

1. The `imfindcircles` function uses the Hough Circle Transform to detect circles in the grayscale image.
2. `[20 80] that` specifies the minimum and maximum radius range of circles to detect (between 20 and 80 pixels).
3. `'Sensitivity', 0.9` that sets the sensitivity level for detection. Higher sensitivity (close to 1) allows the detection of fainter circles but may increase false positives.
4. `'EdgeThreshold', 0.1` that determines the edge threshold used to detect circles. Lower values allow detecting weaker edges.
5. `imshow(image)` displays the original image, and `hold on` ensures that additional plots (e.g., circles) can be overlaid on this image.
6. `viscircles` overlays the detected circles on the image.