Lab Report: 04
Title: Image Resizing & Filtering
Course title: Digital Image Processing Laboratory
Course code: CSE-406
4^{th} Year 1^{st} 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:

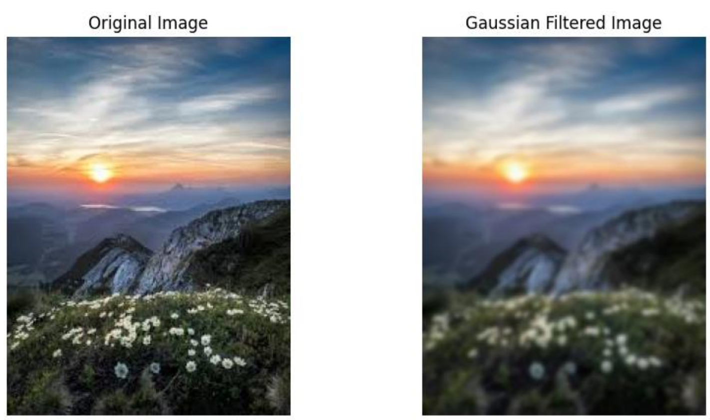

Figure 1.1: Showing the Gaussian Filter in python

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:

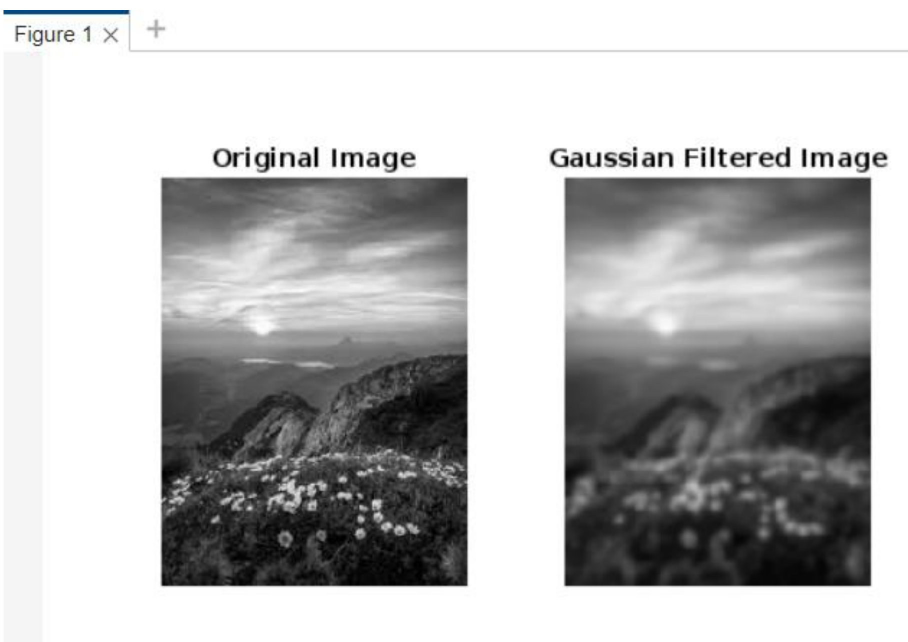

Figure 1.2: Showing the Gaussian Filter in MATLAB

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:

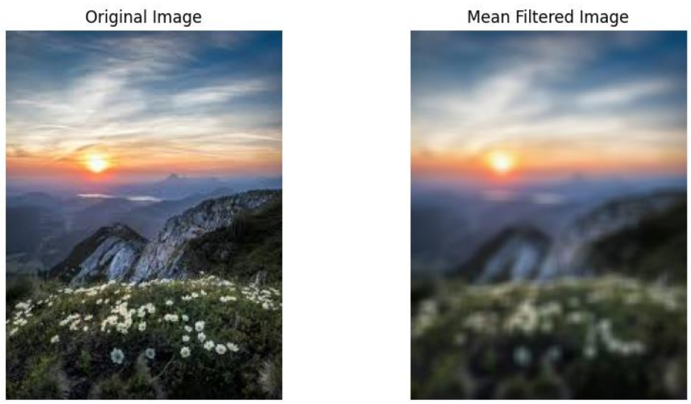

Figure 2.1: Showing the Mean Filter using python code

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:

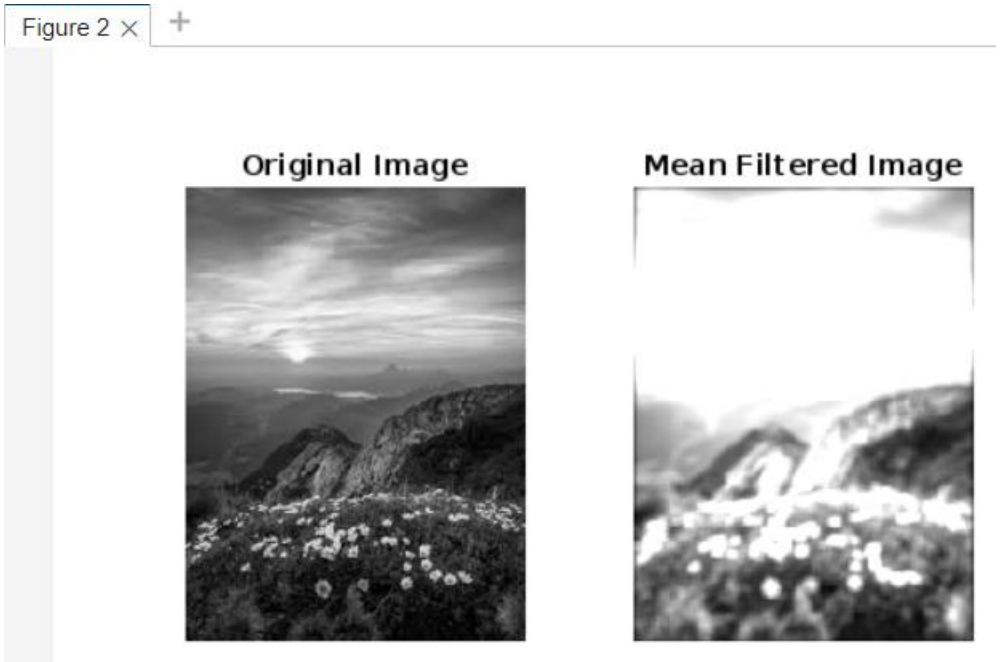

Figure 2.2: Showing the Mean Filter using MATLAB

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:

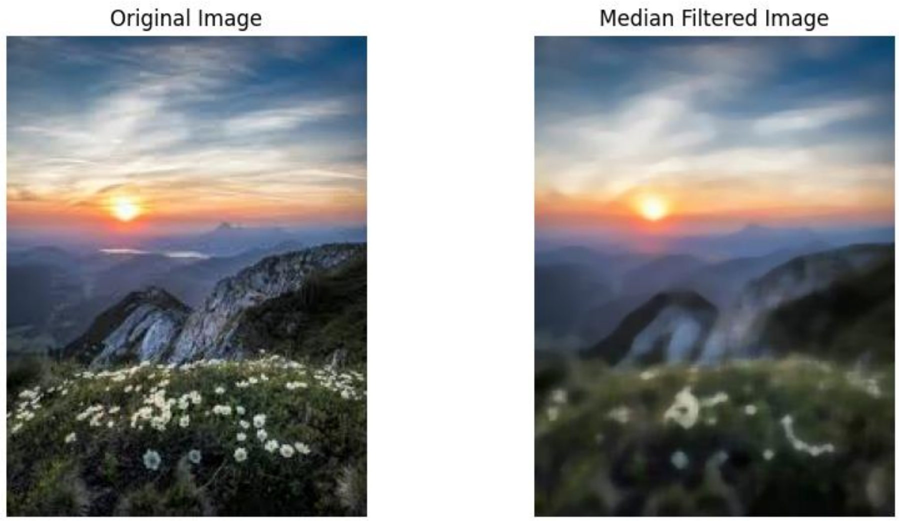

Figure 3.1: Showing the Median Filter using python code

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:

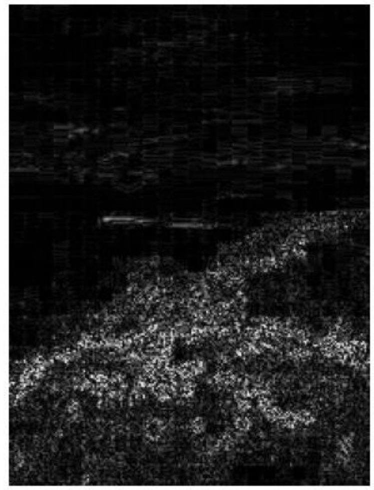

Figure 4.1: Showing the Laplacian Filter using python code

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:

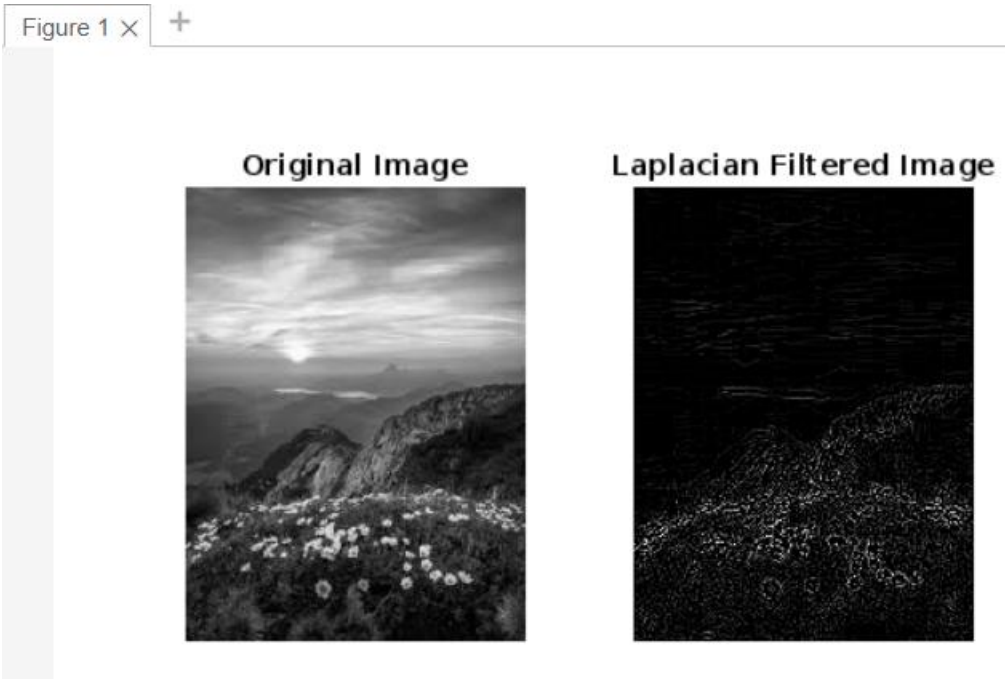

Figure 4.2: Showing the Laplacian Filter in MATLAB

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:

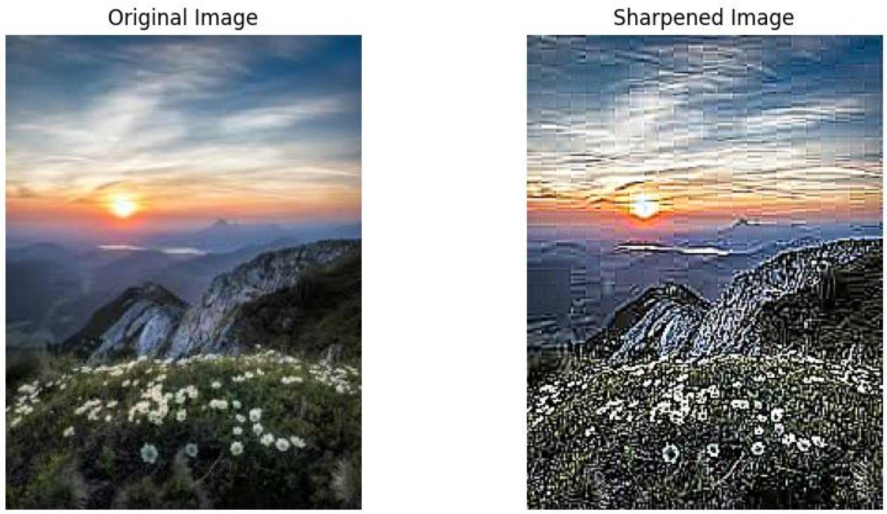

Figure 5.1: Showing the Sharpening Filter using python code

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:

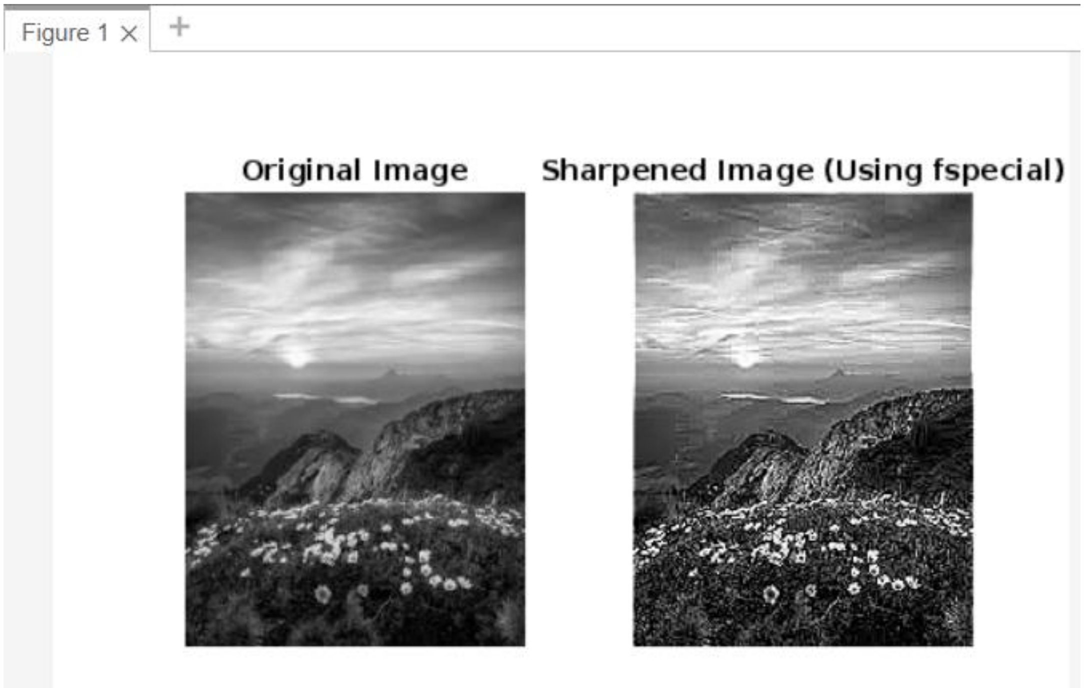

Figure 5.2: Showing the Sharpening Filter in MATLAB

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:

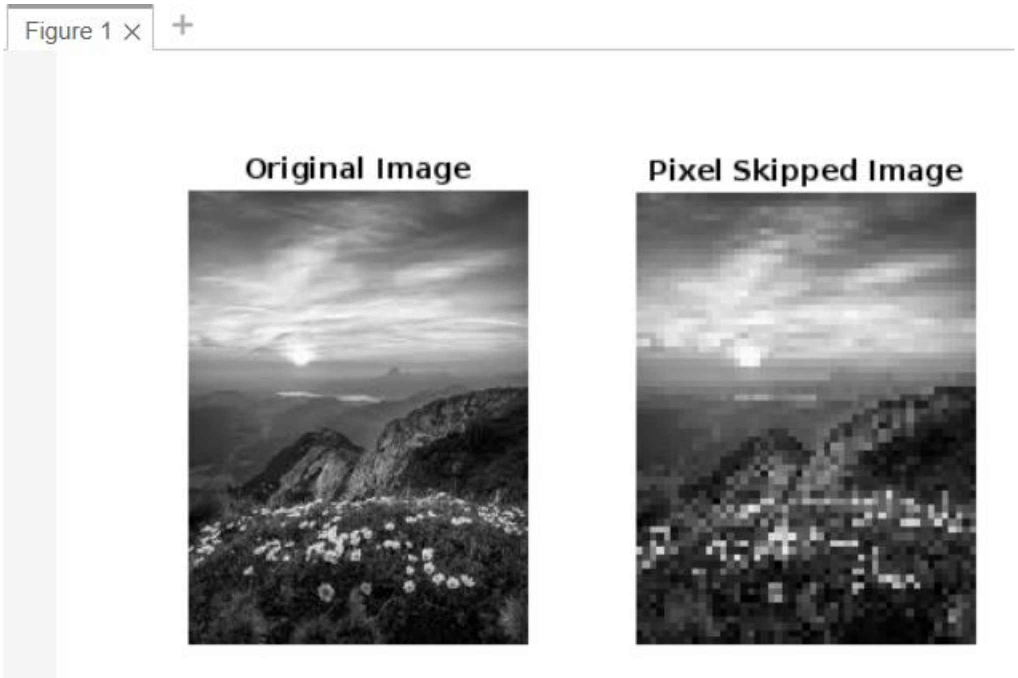

Figure 6.2: Showing the Pixel Skipping in MATLAB

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:

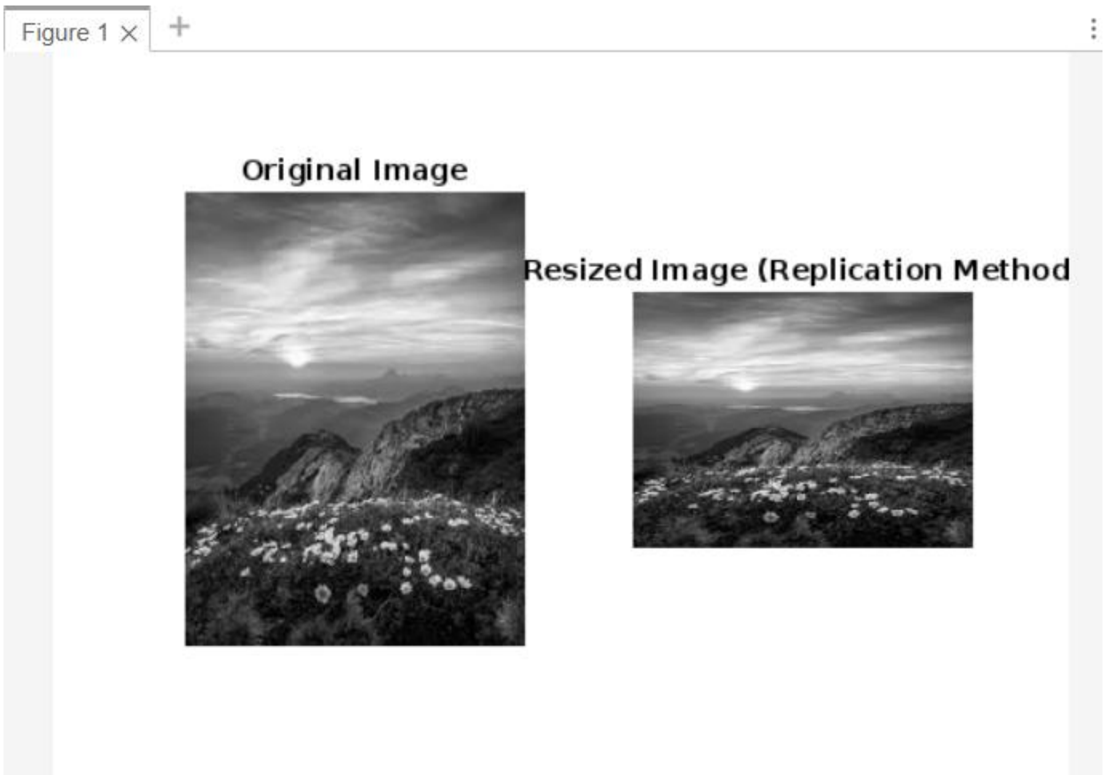

Figure 7.2: Showing the Image Resizing in Replication method using MATLAB

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:

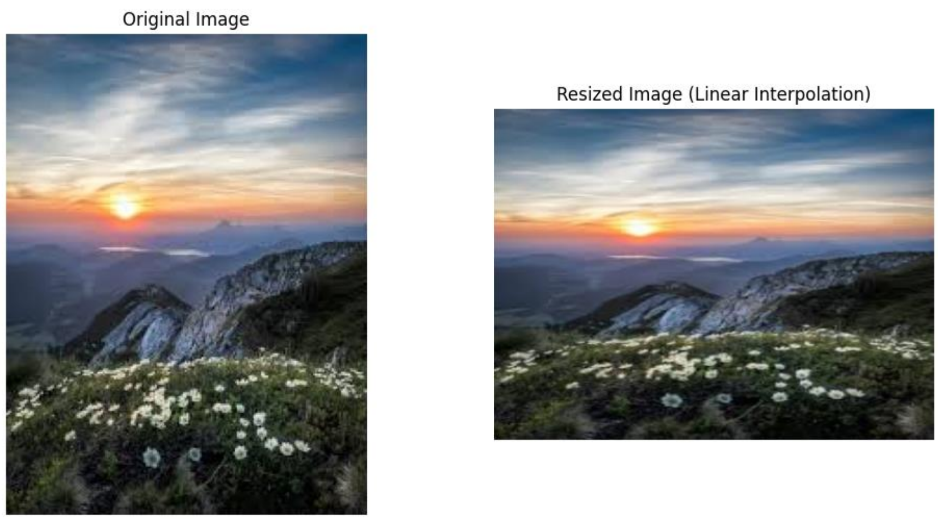

Figure 8.1: Showing the Image Resizing in Interpolation method using python code

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:

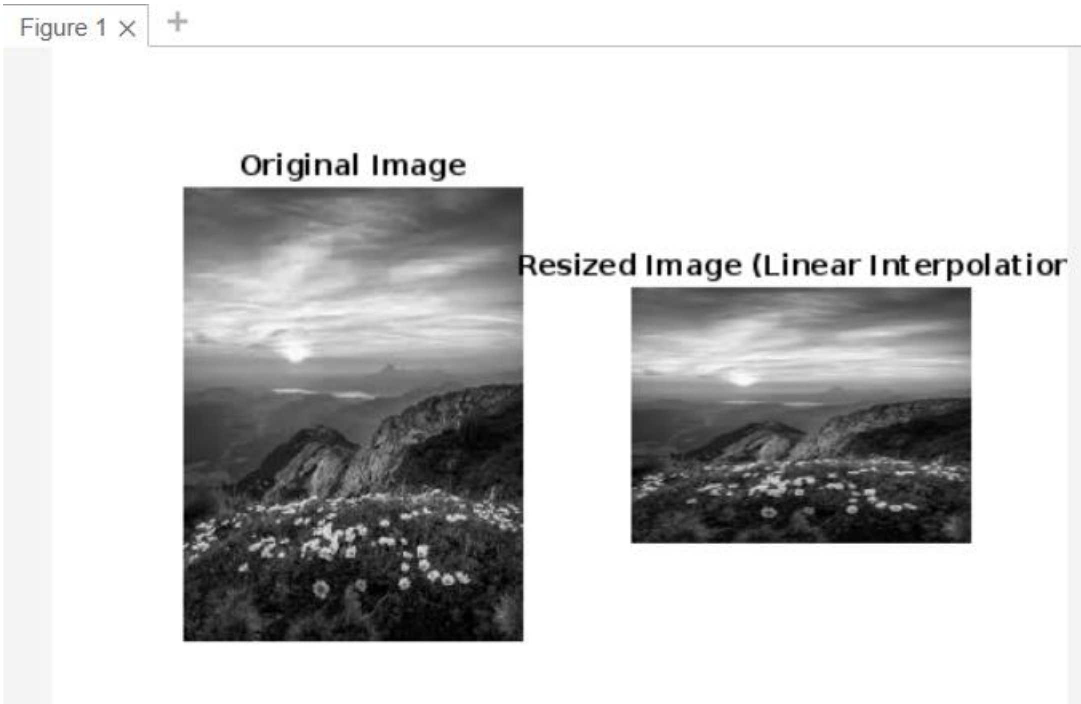

Figure 8.2: Showing the Image Resizing in Interpolation method using MATLAB

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.