Lab Report-04

Course title: Digital Image Processing Laboratory

Course code: CSE-406

4th Year 1st Semester Examination 2023

Date of Submission: 22 September 2024

Submitted to-

Dr. Morium Akter
Professor

Dr. Md. Golam Moazzam
Professor

Department of Computer Science and Engineering
Jahangirnagar University
Savar, Dhaka-1342

Sl	Class Roll	Exam Roll	Name
01	408	202220	Sudipta Singha

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

Lab Report Title

Image Smoothing Using a Gaussian Filter in Python

Introduction

A Gaussian filter is used to smooth images by reducing noise and detail, making it ideal for preprocessing steps in computer vision applications like edge detection. It works by applying a Gaussian function to each pixel and its neighbors, resulting in a weighted average that blurs the image. This filter is useful in fields such as medical imaging, object recognition, and remote sensing for reducing noise and enhancing features.

Python code

```
# gaussian_filter.py

from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage import gaussian_filter

def apply_gaussian_filter(input_path, output_path, sigma):
    # Open the image and convert it to grayscale
    img = Image.open(input_path).convert("L")
    img_array = np.array(img)

    # Apply Gaussian filter
    blurred_array = gaussian_filter(img_array, sigma=sigma)
    blurred_img =
       ↪ Image.fromarray(blurred_array.astype(np.uint8))

    # Save the blurred image
    blurred_img.save(output_path)

    # Display the original and blurred images side by side
    plt.figure(figsize=(10, 5))

    # Display original image
    plt.subplot(1, 2, 1)
    plt.imshow(img, cmap="gray")
    plt.title("Original Image")
    plt.axis("off")

    # Display Gaussian-blurred image
    plt.subplot(1, 2, 2)
    plt.imshow(blurred_img, cmap="gray")
```

```
    plt.title(f"Gaussian Filtered Image (sigma={sigma})")
    plt.axis("off")

    plt.tight_layout()
    plt.show()

# Example usage
if __name__ == "__main__":
    input_path = "input.jpg"           # Replace with the
      ↪ path to your input image
    output_path = "gaussian_blurred.jpg"  # Desired output
      ↪ image path
    sigma = 2                          # Standard deviation
      ↪ for Gaussian kernel
    apply_gaussian_filter(input_path, output_path, sigma)
    print(f"Gaussian-filtered image saved as {output_path}")
```

Input

Figure 1:

Output


```
Gaussian-filtered image saved as gaussian_blurred.jpg
```

Figure 2:

Lab Report Title

Edge Detection Using Laplacian Kernel in Python

Introduction

The Laplacian operator is a second-order derivative filter used for edge detection in images. It highlights regions of rapid intensity change, helping to detect edges and boundaries. The Laplacian kernel is often used in image processing for feature extraction, specifically to detect edges in an image. It works by calculating the second derivative of the image, emphasizing areas with significant changes in intensity.

Python code

```
# laplacian_filter.py

from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage import convolve

def apply_laplacian_filter(input_path, output_path):
    # Open the image and convert it to grayscale
    img = Image.open(input_path).convert("L")
    img_array = np.array(img)

    # Define the Laplacian kernel
    laplacian_kernel = np.array([[0, 1, 0],
                                 [1, -4, 1],
                                 [0, 1, 0]])

    # Apply the Laplacian filter using convolution
    laplacian_filtered = convolve(img_array,
       ↪ laplacian_kernel)

    # Clip the result to ensure pixel values are valid
    laplacian_filtered = np.clip(laplacian_filtered, 0,
       ↪ 255).astype(np.uint8)
    laplacian_img = Image.fromarray(laplacian_filtered)

    # Save the filtered image
    laplacian_img.save(output_path)

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

```
    # Display original image
    plt.subplot(1, 2, 1)
    plt.imshow(img, cmap="gray")
    plt.title("Original Image")
    plt.axis("off")

    # Display Laplacian-filtered image
    plt.subplot(1, 2, 2)
    plt.imshow(laplacian_img, cmap="gray")
    plt.title("Laplacian Filtered Image")
    plt.axis("off")

    plt.tight_layout()
    plt.show()

# Example usage
if __name__ == "__main__":
    input_path = "input.jpg"           # Replace with the
      ↪ path to your input image
    output_path = "laplacian_filtered.jpg"  # Desired output
      ↪ image path
    apply_laplacian_filter(input_path, output_path)
    print(f"Laplacian-filtered image saved as {output_path}")
```

Input

Figure 3:

Output

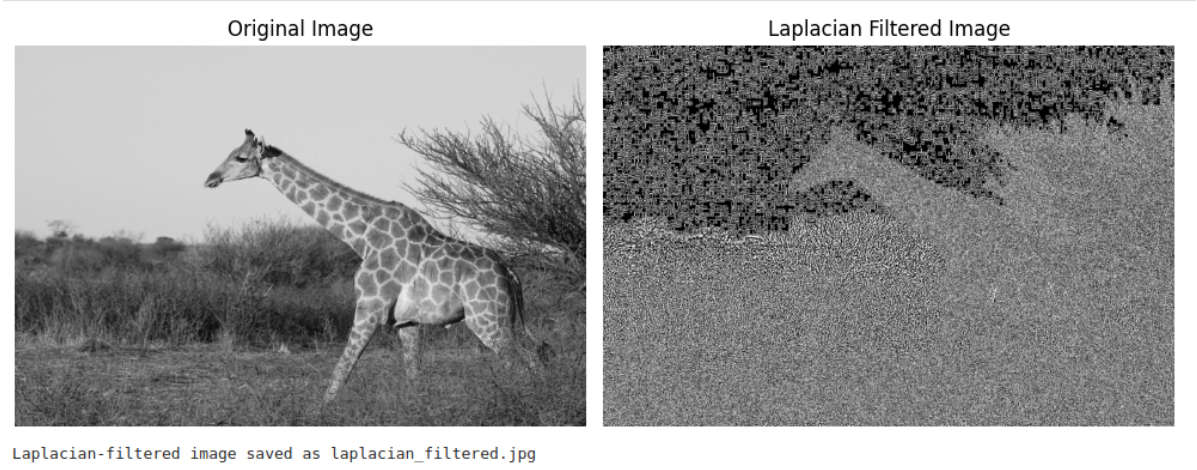

Figure 4:

Lab Report Title

Image Resizing Using Linear Interpolation in Python

Introduction

Linear interpolation is a mathematical method used for estimating unknown values that lie between known values. In image processing, linear interpolation is often used for image resizing, where pixel values in the new image are computed as a weighted average of surrounding pixel values in the original image. This method works well for simple resizing tasks, as it preserves the general structure and appearance of the original image while changing its resolution.

Python code

```
# linear_interpolation.py

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

def linear_interpolation(input_path, output_path, new_width,
    ↪ new_height):
    # Open the image and convert it to grayscale
    img = Image.open(input_path).convert("L")
    img_array = np.array(img)

    # Get the original image dimensions
    original_height, original_width = img_array.shape

    # Create an empty array for the resized image
    resized_array = np.zeros((new_height, new_width),
        ↪ dtype=np.uint8)

    # Compute scale factors
    x_scale = original_width / new_width
    y_scale = original_height / new_height

    # Perform linear interpolation
    for i in range(new_height):
        for j in range(new_width):
            # Find corresponding coordinates in the original
                ↪ image
            orig_x = int(j * x_scale)
            orig_y = int(i * y_scale)
```

```
            # Linear interpolation for x-direction
              ↪ (horizontal)
            if orig_x + 1 < original_width:
                weight_x = (j * x_scale) - orig_x
                pixel_value_x = (1 - weight_x) *
                  ↪ img_array[orig_y, orig_x] + weight_x *
                  ↪ img_array[orig_y, orig_x + 1]
            else:
                pixel_value_x = img_array[orig_y, orig_x]

            # Linear interpolation for y-direction (vertical)
            if orig_y + 1 < original_height:
                weight_y = (i * y_scale) - orig_y
                pixel_value_y = (1 - weight_y) *
                  ↪ pixel_value_x + weight_y *
                  ↪ img_array[orig_y + 1, orig_x]
            else:
                pixel_value_y = pixel_value_x

            # Assign the interpolated pixel value to the
              ↪ resized image
            resized_array[i, j] = pixel_value_y

    # Convert the resized array back to an image
    resized_img = Image.fromarray(resized_array)

    # Save the resized image
    resized_img.save(output_path)

    # Display the original and resized images
    plt.figure(figsize=(10, 5))

    # Display original image
    plt.subplot(1, 2, 1)
    plt.imshow(img, cmap="gray")
    plt.title("Original Image")
    plt.axis("off")

    # Display resized image
    plt.subplot(1, 2, 2)
    plt.imshow(resized_img, cmap="gray")
    plt.title(f"Resized Image ({new_width}x{new_height})")
    plt.axis("off")

    plt.tight_layout()
    plt.show()

# Example usage
```

```
if __name__ == "__main__":
    input_path = "input.jpg"            # Replace with the
        ↪ path to your input image
    output_path = "resized_image.jpg"   # Desired output
        ↪ image path
    new_width = 300                      # New width for the
        ↪ resized image
    new_height = 300                     # New height for the
        ↪ resized image
    linear_interpolation(input_path, output_path, new_width,
        ↪ new_height)
    print(f"Resized image saved as {output_path}")
```

Input

Figure 5:

Output


```
Resized image saved as resized_image.jpg
```

Figure 6:

Lab Report Title

Image Smoothing Using Mean Filter in Python

Introduction

A mean filter, also known as a box filter or averaging filter, is a simple and commonly used image processing technique for noise reduction. It works by replacing each pixel's value with the average value of its neighboring pixels within a defined window. The mean filter is particularly effective for removing random noise (salt-and-pepper noise), but it may blur the image if the window size is too large.

Python code

```
# mean_filter.py

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

def apply_mean_filter(input_path, output_path, kernel_size):
    # Open the image and convert it to grayscale
    img = Image.open(input_path).convert("L")
    img_array = np.array(img)

    # Get the dimensions of the image
    height, width = img_array.shape

    # Create an empty array for the filtered image
    filtered_array = np.zeros_like(img_array)

    # Define the size of the kernel (window)
    pad_size = kernel_size // 2

    # Pad the image with zeros around the border (to handle
    ↪ edge cases)
    padded_img = np.pad(img_array, pad_width=pad_size,
    ↪ mode='constant', constant_values=0)

    # Apply the mean filter
    for i in range(height):
        for j in range(width):
            # Extract the neighborhood for each pixel
            ↪ (kernel_size x kernel_size window)
            neighborhood = padded_img[i:i+kernel_size,
            ↪ j:j+kernel_size]
```

```
            # Compute the mean of the neighborhood
            mean_value = np.mean(neighborhood)
            # Assign the mean value to the corresponding
               ↪ pixel in the filtered image
            filtered_array[i, j] = mean_value

    # Convert the filtered array back to an image
    filtered_img =
       ↪ Image.fromarray(filtered_array.astype(np.uint8))

    # Save the filtered image
    filtered_img.save(output_path)

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

    # Display original image
    plt.subplot(1, 2, 1)
    plt.imshow(img, cmap="gray")
    plt.title("Original Image")
    plt.axis("off")

    # Display filtered image
    plt.subplot(1, 2, 2)
    plt.imshow(filtered_img, cmap="gray")
    plt.title(f"Mean Filtered Image (Kernel size:
       ↪ {kernel_size})")
    plt.axis("off")

    plt.tight_layout()
    plt.show()

# Example usage
if __name__ == "__main__":
    input_path = "input.jpg"           # Replace with the
       ↪ path to your input image
    output_path = "mean_filtered.jpg"  # Desired output
       ↪ image path
    kernel_size = 3                    # Kernel size (3x3
       ↪ window)
    apply_mean_filter(input_path, output_path, kernel_size)
    print(f"Mean-filtered image saved as {output_path}")
```

Input

Figure 7:

Output

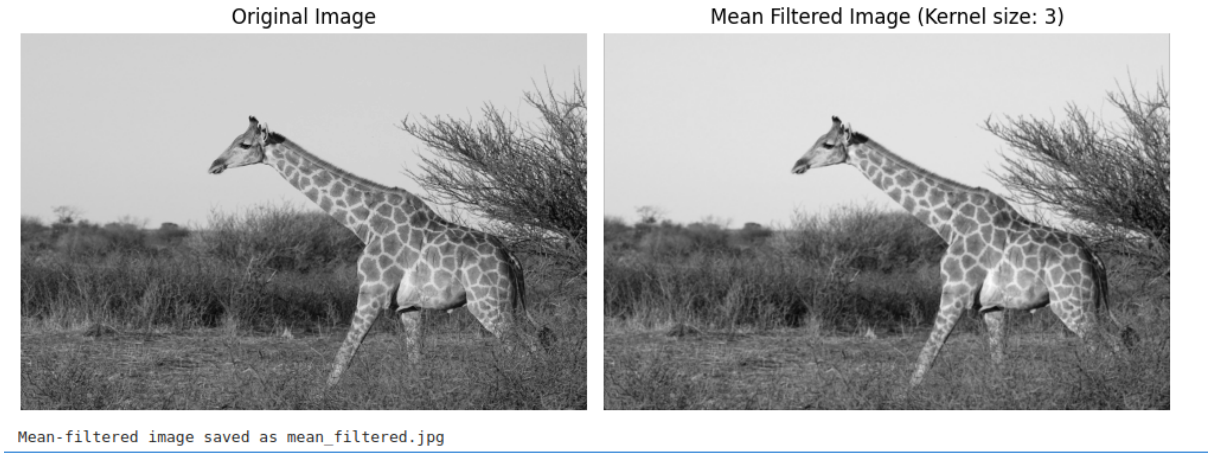

Figure 8:

Lab Report Title

Image Smoothing Using Median Filter in Python

Introduction

The Median Filter is a non-linear image processing technique used to remove noise, particularly salt-and-pepper noise. It works by replacing each pixel in the image with the median value of the pixels in its neighborhood. The median filter is more effective than the mean filter in preserving edges while removing noise. It is commonly used in applications where noise removal is essential without blurring edges significantly.

Python code

```
# median_filter.py

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

def apply_median_filter(input_path, output_path,
    ↪ kernel_size):
    # Open the image and convert it to grayscale
    img = Image.open(input_path).convert("L")
    img_array = np.array(img)

    # Get the dimensions of the image
    height, width = img_array.shape

    # Create an empty array for the filtered image
    filtered_array = np.zeros_like(img_array)

    # Define the size of the kernel (window)
    pad_size = kernel_size // 2

    # Pad the image with zeros around the border (to handle
       ↪ edge cases)
    padded_img = np.pad(img_array, pad_width=pad_size,
       ↪ mode='constant', constant_values=0)

    # Apply the median filter
    for i in range(height):
        for j in range(width):
            # Extract the neighborhood for each pixel
               ↪ (kernel_size x kernel_size window)
```

```
            neighborhood = padded_img[i:i+kernel_size,
               ↪ j:j+kernel_size]
            # Compute the median of the neighborhood
            median_value = np.median(neighborhood)
            # Assign the median value to the corresponding
               ↪ pixel in the filtered image
            filtered_array[i, j] = median_value

    # Convert the filtered array back to an image
    filtered_img =
       ↪ Image.fromarray(filtered_array.astype(np.uint8))

    # Save the filtered image
    filtered_img.save(output_path)

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

    # Display original image
    plt.subplot(1, 2, 1)
    plt.imshow(img, cmap="gray")
    plt.title("Original Image")
    plt.axis("off")

    # Display filtered image
    plt.subplot(1, 2, 2)
    plt.imshow(filtered_img, cmap="gray")
    plt.title(f"Median Filtered Image (Kernel size:
       ↪ {kernel_size})")
    plt.axis("off")

    plt.tight_layout()
    plt.show()

# Example usage
if __name__ == "__main__":
    input_path = "input.jpg"           # Replace with the
       ↪ path to your input image
    output_path = "median_filtered.jpg"  # Desired output
       ↪ image path
    kernel_size = 3                    # Kernel size (3x3
       ↪ window)
    apply_median_filter(input_path, output_path, kernel_size)
    print(f"Median-filtered image saved as {output_path}")
```

Input

Figure 9:

Output

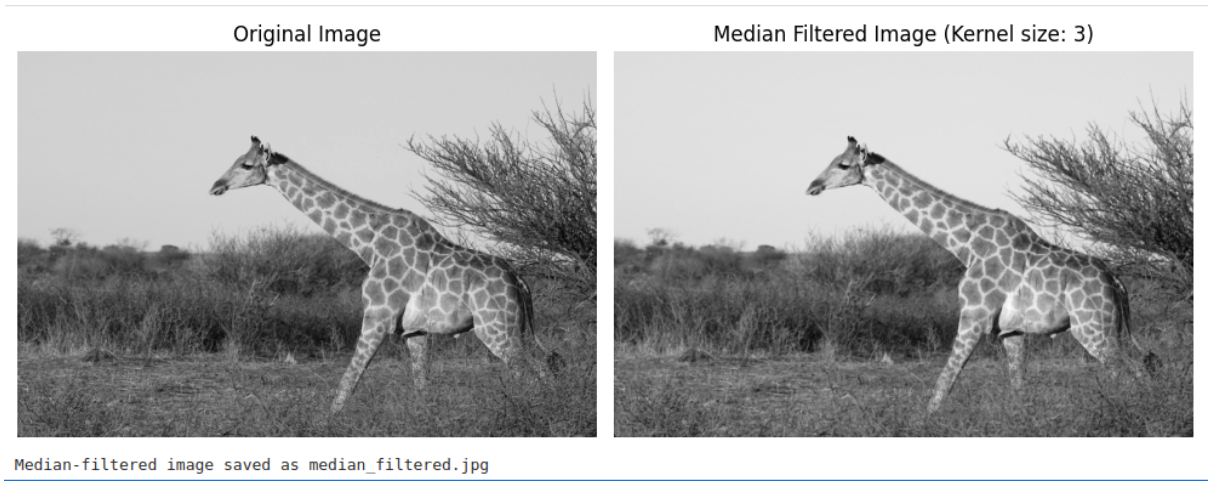

Figure 10:

Lab Report Title

Replication Method in image processing

Introduction

The Replication Method is a technique used for image padding where the border pixels are replicated to fill the padding region. This is useful when applying filters to images, as it avoids introducing artificial values in the padded areas by simply replicating the outermost pixels.

Python code

```
# replication_method.py

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

def apply_replication_padding(input_path, output_path,
    ↪ kernel_size):
    img = Image.open(input_path).convert("L")
    img_array = np.array(img)

    # Define padding size
    pad_size = kernel_size // 2

    # Apply replication padding
    padded_img = np.pad(img_array, pad_width=pad_size,
        ↪ mode='edge')

    # Create a filtered image with the padded image (for
        ↪ example, simply taking a part of the image)
    filtered_array = padded_img[pad_size:-pad_size,
        ↪ pad_size:-pad_size]

    filtered_img =
        ↪ Image.fromarray(filtered_array.astype(np.uint8))
    filtered_img.save(output_path)

    # Display original and padded image
    plt.figure(figsize=(10, 5))
    plt.subplot(1, 2, 1)
    plt.imshow(img, cmap="gray")
    plt.title("Original Image")
    plt.axis("off")
```

```
    plt.subplot(1, 2, 2)
    plt.imshow(filtered_img, cmap="gray")
    plt.title("Image with Replication Padding")
    plt.axis("off")
    plt.tight_layout()
    plt.show()

# Example usage
if __name__ == "__main__":
    input_path = "input.jpg"
    output_path = "replication_padded.jpg"
    kernel_size = 3
    apply_replication_padding(input_path, output_path,
        ↪ kernel_size)
    print(f"Image with replication padding saved as
        ↪ {output_path}")
```

Input

Figure 11:

Output

Figure 12: