JGI JAIN DEEMED-TO-BE UNIVERSITY | SCHOOL OF COMPUTER SCIENCE AND IT

SCHOOL OF COMPUTER SCIENCE AND IT

DEPARTMENT OF CS & IT

BCA PROGRAMME

SEMESTER: V

SUBJECT NAME : Image Processing

SUBJECT CODE : 24BCA6IOE02

ACTIVITY 2

MINI PROJECT

Implementation of Spatial, Morphological and Thresholding Operations in Image Processing

Date of Submission: 12th February 2026

Submitted by:

23BCAR0339 – Somnath Gorai

Faculty In-Charge:

Dr. Zion Ramdinthara

Professor

JGI JAIN DEEMED-TO-BE UNIVERSITY | SCHOOL OF COMPUTER SCIENCE AND IT

CERTIFICATE

This is to certify that Ms. /Mr. **Somnath Gorai** has satisfactorily completed Activity 2 prescribed by JAIN (Deemed to be University) for the 6th Semester BCA degree course in the year 2026.

Signature of the Faculty In Charge

Assessment Sheet with Rubrics for Grading & Evaluation

Students have to complete the online courses in the given timeline and submit the report as per format given.

Sr. No.	USN No.	Student Name	Report	Presentation	Viva-Voce	Total
			5 Marks	5 Marks	5 Marks	15 Marks
1	23BCAR0339	Somnath Gorai				

ABSTRACT

Image processing plays a crucial role in enhancing, analyzing, and interpreting digital images for various real-world applications such as medical imaging, remote sensing, surveillance, and computer vision. The primary objective of this activity is to study and implement fundamental image processing techniques using spatial filtering, morphological operations, feature detection, and segmentation methods. These techniques form the foundation for understanding how digital images can be manipulated and analyzed mathematically and computationally.

In this activity, Gaussian Blur, Image Sharpening, and Unsharp Masking are applied to enhance image quality and control image smoothness. Gaussian Blur is used to reduce noise and suppress high-frequency components by convolving the image with a Gaussian kernel. Image Sharpening enhances edges and fine details by emphasizing high-frequency components, while Unsharp Masking improves clarity by combining the original image with its blurred version. These techniques demonstrate how spatial domain filters influence image appearance and detail.

Morphological image processing operations such as Dilation, Erosion, Opening, and Closing are implemented to analyze and manipulate binary images based on their shapes. Dilation and erosion modify object boundaries, while opening and closing are used for noise removal and gap filling. These operations are particularly effective in processing images containing text and structured objects.

Further, Line Detection and Point Detection techniques are applied using suitable convolution masks to identify structural features within an image. Line detection highlights linear patterns and edges, whereas point detection identifies isolated pixels and sharp intensity variations. These operations are essential for feature extraction and pattern recognition tasks.

Finally, a Global Thresholding technique is implemented to segment an image into foreground and background regions based on a predefined threshold value. This method demonstrates a simple yet effective approach to image segmentation.

All algorithms are implemented using Python with standard development tools such as Spyder, Jupyter Notebook, Google Colab, PyCharm, and Visual Studio Code. The results obtained validate the theoretical concepts and highlight the practical importance of basic image processing techniques.

Input Image:

Above images are the input images that was being used in all the operation performed bellow.

Table of Contents

SL.NO	Title	Page
1	ABSTRACT	3
2	CHAPTER 1: GAUSSIAN BLUR, SHARPENING AND UNSHARP MASKING	6
3	CHAPTER 2: MORPHOLOGICAL OPERATIONS	15
4	CHAPTER 3: LINE DETECTION AND POINT DETECTION	27
5	CHAPTER 4: GLOBAL THRESHOLDING	33
6	CONCLUSION	36

CHAPTER 1: GAUSSIAN BLUR, SHARPENING AND UNSHARP MASKING

1.1 Gaussian Blur

Gaussian Blur is a linear smoothing filter used to reduce noise and minor details in an image. It works by averaging pixel values with their neighboring pixels using weights derived from the Gaussian distribution. Pixels closer to the center of the kernel have higher influence than distant pixels. This operation suppresses high-frequency components such as noise and sharp edges, resulting in a smoother image.

Kernal:

Gaussian Blur smooths an image using a weighted averaging kernel derived from the Gaussian distribution. The Gaussian kernel used is:

$$G = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Algorithm:

1. Read the input image and convert it to grayscale.
2. Add zero padding around the image borders.
3. Define the 3×3 Gaussian kernel.
4. Slide the kernel over the image pixel by pixel.
5. Multiply kernel values with corresponding image pixels.
6. Sum the results and divide by 16.
7. Assign the computed value to the output pixel.

Implementation:

Gaussian Blur is implemented manually using nested loops to perform convolution between the input image and the Gaussian kernel. Padding is added explicitly to handle border pixels.

Code:

```python
import cv2
import numpy as np

# Read image
img = cv2.imread("image.png", cv2.IMREAD_GRAYSCALE)

rows, cols = img.shape

# Gaussian kernel (textbook)
kernel = np.array([[1, 2, 1],
                   [2, 4, 2],
                   [1, 2, 1]])

# Zero padding
padded = np.zeros((rows + 2, cols + 2), dtype=np.uint8)
padded[1:rows+1, 1:cols+1] = img

# Output image
blurred = np.zeros_like(img)

# Manual convolution
for i in range(rows):
    for j in range(cols):
        region = padded[i:i+3, j:j+3]
        value = np.sum(region * kernel) / 16
        blurred[i, j] = int(value)

# Save output
cv2.imwrite("gaussian_blur.png", blurred)
```

Code Explanation:

- **Lines 1–2:** Required libraries are imported. cv2 is used for image I/O, and numpy is used for array operations.
- **Line 5:** The input image is read in **grayscale**, converting it into a 2D pixel intensity array.
- **Line 7:** The number of rows and columns of the image are obtained to control looping and padding.
- **Lines 10–12:** A **3×3 Gaussian kernel** is defined exactly as given in the textbook. This kernel provides weighted averaging, where the center pixel has the highest weight.
- **Lines 15–16: Zero padding** is manually added around the image to allow convolution at the borders. The original image is placed in the center of the padded array.
- **Line 19:** An empty output image is created to store the blurred pixel values.
- **Lines 22–23:** Nested loops slide the kernel **pixel by pixel** across the image.

- **Line 24:** A **3×3 region** from the padded image is extracted for convolution.
- **Line 25:** Element-wise multiplication between the image region and the kernel is performed, summed, and normalized by dividing by 16 to preserve brightness.
- **Line 26:** The computed value is assigned to the corresponding pixel in the output image.
- **Line 29:** The final Gaussian blurred image is saved.

Output Image:

Observation:

After applying Gaussian Blur, noise and minor intensity variations are reduced. Edges become slightly smoother while the overall brightness and structure of the image are preserved due to kernel normalization.

1.2 Image Sharpening

Image sharpening is a spatial domain technique used to enhance edges and fine details in an image. It works by emphasizing high-frequency components such as edges and boundaries, making the image appear clearer and more defined. Sharpening is commonly applied after smoothing operations to restore lost details.

Kernal:

Image sharpening can be achieved using a sharpening mask that highlights intensity differences between a pixel and its neighbors.

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Algorithm:

1. Read the input image and convert it to grayscale.
2. Add zero padding around the image.
3. Define the sharpening kernel.
4. Slide the kernel across the image manually.
5. Multiply kernel values with corresponding image pixels.
6. Sum the results to obtain the sharpened pixel value.
7. Clip values to valid range and store in output image.

Implementation:

Sharpening is implemented manually by convolving the input image with a sharpening kernel. Padding and convolution are performed explicitly using nested loops without using built-in filtering functions.

Code:

```
1   import cv2
2   import numpy as np
3
4   # Read image
5   img = cv2.imread("image.png", cv2.IMREAD_GRAYSCALE)
6
7   rows, cols = img.shape
8
9   # Sharpening kernel
10  kernel = np.array([[0, -1, 0],
11                     [-1, 5, -1],
12                     [0, -1, 0]])
13
14  # Zero padding
15  padded = np.zeros((rows + 2, cols + 2), dtype=np.int16)
16  padded[1:rows+1, 1:cols+1] = img
17
18  # Output image
19  sharpened = np.zeros_like(img)
20
21  # Manual convolution
22  for i in range(rows):
23      for j in range(cols):
24          region = padded[i:i+3, j:j+3]
25          value = np.sum(region * kernel)
26          value = min(max(value, 0), 255)
27          sharpened[i, j] = value
28
29  # Save output
30  cv2.imwrite("sharpened_image.png", sharpened)
```

Code Explanation:

- **Lines 1–2:** Required libraries are imported.
- **Line 5:** The input image is read in grayscale format.
- **Line 7:** Image dimensions are stored for looping and padding.
- **Lines 10–12:** The sharpening kernel is defined. The center value strengthens the pixel while surrounding negative values highlight edges.
- **Lines 15–16:** Zero padding is manually applied to handle border pixels.
- **Line 19:** An empty output image is created to store results.
- **Lines 22–23:** Nested loops slide the kernel across the image.
- **Line 24:** A 3×3 region is extracted for convolution.
- **Line 25:** Kernel and image region are multiplied and summed.
- **Line 26:** Pixel values are clipped to the range 0–255.
- **Line 27:** The computed value is assigned to the output image.
- **Line 30:** The sharpened image is saved.

Output Image:

Observation:

After applying image sharpening, edges and fine details become more prominent. The image appears clearer, but excessive sharpening may introduce noise or edge artifacts if applied repeatedly.

1.3 Unsharp Masking

Unsharp masking is an image enhancement technique used to increase sharpness by emphasizing edges. It works by subtracting a blurred version of the image from the original image to obtain edge details (mask), and then adding this mask back to the original image. This enhances fine details while preserving the overall structure of the image.

Formula:

$$\text{Mask} = I - I_{blurred}$$

$$\text{Sharpened Image} = I + k \times \text{Mask}$$

Where:

- I= original image
- $I_{blurred}$= Gaussian blurred image
- k= sharpening factor (usually 1)

Algorithm:

1. Read the input image and convert it to grayscale.
2. Apply Gaussian Blur manually to obtain a smoothed image.
3. Subtract the blurred image from the original to get the mask.
4. Add the mask back to the original image.
5. Clip values to valid pixel range and store output.

Implementation

Unsharp masking is implemented manually by first performing Gaussian blurring using convolution, followed by pixel-wise subtraction and addition. No built-in sharpening or filtering functions are used.

Code:

```
1   import cv2
2   import numpy as np
3
4   # Read image
5   img = cv2.imread("image.png", cv2.IMREAD_GRAYSCALE)
6
7   rows, cols = img.shape
8
9   # Gaussian kernel
10  kernel = np.array([[1, 2, 1],
11                     [2, 4, 2],
12                     [1, 2, 1]])
13
14  # Zero padding
15  padded = np.zeros((rows + 2, cols + 2), dtype=np.int16)
16  padded[1:rows+1, 1:cols+1] = img
17
18  # Gaussian blurred image
19  blurred = np.zeros_like(img)
20
21  for i in range(rows):
22      for j in range(cols):
23          region = padded[i:i+3, j:j+3]
24          blurred[i, j] = np.sum(region * kernel) / 16
25
26  # Unsharp masking
27  mask = img - blurred
28  sharpened = img + mask
29
30  # Clip values
31  sharpened = np.clip(sharpened, 0, 255)
32
33  # Save output
34  cv2.imwrite("unsharp_masking.png", sharpened)
```

Code Explanation:

- **Lines 1–2:** Required libraries are imported.
- **Line 5:** Input image is read in grayscale format.
- **Line 7:** Image dimensions are extracted.
- **Lines 10–12:** Gaussian kernel is defined as per textbook.
- **Lines 15–16:** Zero padding is manually applied.
- **Line 19:** Output array for blurred image is created.
- **Lines 21–24:** Gaussian blur is performed using manual convolution.
- **Line 27:** The **mask** is obtained by subtracting the blurred image from the original image.
- **Line 28:** The mask is added back to the original image to enhance edges.
- **Line 31:** Pixel values are clipped to the valid range.
- **Line 34:** The final sharpened image is saved.

Output Image:

Observation:

After applying unsharp masking, edges and fine details are enhanced more naturally compared to direct sharpening. The image appears clearer while avoiding excessive noise amplification.

CHAPTER 2: MORPHOLOGICAL OPERATIONS

2.1 Dilation

Dilation is a morphological operation used to **expand foreground (object) regions** in a binary image. It increases the size of objects, fills small gaps, and connects nearby components. Dilation works by sliding a structuring element over the image and applying logical rules based on pixel matches.

Kernel:

$$\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Dilation Rules:

- **FIT:** Output pixel = 1, when **all** image pixels match the kernel
- **HIT:** Output pixel = 1, when **some** pixels match the kernel
- **MISS:** Output pixel = 0, when **no** pixels match the kernel

Algorithm:

1. Read the input image and convert it to binary.
2. Add zero padding to the image borders.
3. Define the vertical structuring element.
4. Slide the kernel vertically over the image.
5. Check FIT, HIT, or MISS condition.
6. Assign output pixel based on dilation rules.

Implementation:

Dilation is implemented manually using nested loops. For each pixel position, the vertical kernel is matched against the image pixels, and the output pixel is decided using HIT–FIT–MISS logic without using built-in morphological functions.

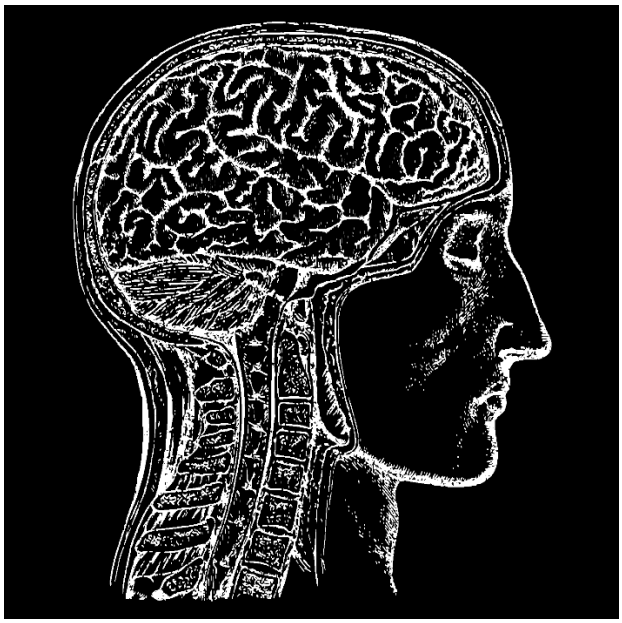
Code:

```
1   import cv2
2   import numpy as np
3
4   # Read image
5   img = cv2.imread("image.png", cv2.IMREAD_GRAYSCALE)
6
7   # Convert to binary image
8   _, binary = cv2.threshold(img, 127, 1, cv2.THRESH_BINARY)
9
10  rows, cols = binary.shape
11
12  # Zero padding (top and bottom)
13  padded = np.zeros((rows + 2, cols), dtype=np.uint8)
14  padded[1:rows+1, :] = binary
15
16  # Output image
17  dilated = np.zeros_like(binary)
18
19  # Manual dilation
20  for i in range(rows):
21      for j in range(cols):
22          region = padded[i:i+3, j]
23
24          if np.sum(region) >= 1:    # HIT or FIT
25              dilated[i, j] = 1
26          else:                      # MISS
27              dilated[i, j] = 0
28
29  # Save output
30  cv2.imwrite("dilated_image.png", dilated * 255)
```

Code Explanation:

- **Lines 1–2:** Required libraries are imported.
- **Line 5:** Input image is read in grayscale format.
- **Line 8:** The image is converted into a **binary image** (0 and 1).
- **Line 10:** Image dimensions are extracted.
- **Lines 13–14:** Zero padding is added vertically to support the vertical kernel.
- **Line 17:** Output image array is initialized.
- **Lines 20–21:** Nested loops slide the kernel across the image.
- **Line 22:** A vertical 3×1 region is extracted.
- **Line 24:** HIT or FIT condition is checked (at least one match).
- **Lines 25–26:** Output pixel is set according to dilation rules.
- **Line 30:** Output image is saved after converting back to display range.

Output:



Observation

After applying dilation, foreground objects become thicker and small gaps within objects are filled. Vertical structures are enhanced due to the use of a vertical structuring element.

2.2 Erosion

Erosion is a morphological operation used to **shrink foreground objects** in a binary image. It removes small protrusions, separates connected objects, and eliminates thin noise. Erosion works by sliding a structuring element over the image and retaining a pixel only when the kernel completely fits inside the foreground region.

Kernel:

$$\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Erosion Rules:

- **FIT:** Output pixel = 1, when **all** image pixels match the kernel
- **HIT:** Output pixel = 0, when **some** pixels match the kernel
- **MISS:** Output pixel = 0, when **no** pixels match the kernel

Algorithm:

1. Read the input image and convert it to binary.
2. Add zero padding to the image borders.
3. Define the vertical structuring element.
4. Slide the kernel vertically over the image.
5. Check FIT, HIT, or MISS condition.
6. Assign output pixel based on erosion rules.

Implementation:

Erosion is implemented manually using nested loops. At each pixel location, the vertical kernel is compared with the image pixels and the output is determined strictly using the FIT condition.

Code:

```python
import cv2
import numpy as np

# Read image
img = cv2.imread("image.png", cv2.IMREAD_GRAYSCALE)

# Convert to binary image
_, binary = cv2.threshold(img, 127, 1, cv2.THRESH_BINARY)

rows, cols = binary.shape

# Zero padding (top and bottom)
padded = np.zeros((rows + 2, cols), dtype=np.uint8)
padded[1:rows+1, :] = binary

# Output image
eroded = np.zeros_like(binary)

# Manual erosion
for i in range(rows):
    for j in range(cols):
        region = padded[i:i+3, j]

        if np.sum(region) == 3:   # FIT
            eroded[i, j] = 1
        else:                     # HIT or MISS
            eroded[i, j] = 0

# Save output
cv2.imwrite("eroded_image.png", eroded * 255)
```

Code Explanation:

- **Lines 1–2:** Required libraries are imported.
- **Line 5:** Input image is read in grayscale format.
- **Line 8:** Image is converted to binary form (0 and 1).
- **Line 10:** Image dimensions are extracted.
- **Lines 13–14:** Vertical zero padding is added for kernel operation.
- **Line 17:** Output image array is initialized.
- **Lines 20–21:** Nested loops slide the kernel over the image.
- **Line 22:** A vertical 3×1 neighborhood is extracted.
- **Line 24:** FIT condition is checked (all pixels must be 1).
- **Lines 25–26:** Output pixel is assigned based on erosion rules.
- **Line 30:** Final eroded image is saved.

Output:

Observation:

After erosion, foreground objects become thinner and small noise elements are removed. Vertical structures shrink due to the vertical structuring element, helping in separating closely connected components.

2.3 Opening

Opening is a morphological operation used to remove small foreground objects and thin noise from a binary image while preserving the shape and size of larger objects. It is particularly useful for eliminating narrow connections and smoothing object boundaries.

Formula:

$$A \circ B = (A \ominus B) \oplus B$$

Where:

- A = input binary image
- B = structuring element
- $\ominus$ = erosion
- $\oplus$ = dilation

Algorithm:

1. Read the input image and convert it to binary.
2. Apply **erosion** on the image using structuring element B.
3. Apply **dilation** on the eroded image using the same structuring element.
4. Store the final result as the opened image.

Implementation

Opening is implemented manually by first eroding the input image and then dilating the eroded result. Both operations follow the HIT–FIT–MISS rules as defined in the textbook, and no built-in morphological functions are used.

Code:

```python
import cv2
import numpy as np

# Read image
img = cv2.imread("image.png", cv2.IMREAD_GRAYSCALE)

# Convert to binary image
_, binary = cv2.threshold(img, 127, 1, cv2.THRESH_BINARY)

rows, cols = binary.shape

# Padding for vertical kernel
padded = np.zeros((rows + 2, cols), dtype=np.uint8)
padded[1:rows+1, :] = binary

# ----- Erosion -----
eroded = np.zeros_like(binary)

for i in range(rows):
    for j in range(cols):
        region = padded[i:i+3, j]
        if np.sum(region) == 3:      # FIT
            eroded[i, j] = 1
        else:
            eroded[i, j] = 0

# Padding eroded image
padded_eroded = np.zeros((rows + 2, cols), dtype=np.uint8)
padded_eroded[1:rows+1, :] = eroded

# ----- Dilation -----
opened = np.zeros_like(binary)

for i in range(rows):
    for j in range(cols):
        region = padded_eroded[i:i+3, j]
        if np.sum(region) >= 1:      # HIT or FIT
            opened[i, j] = 1
        else:
            opened[i, j] = 0

# Save output
cv2.imwrite("opened_image.png", opened * 255)
```

Code Explanation:

- **Lines 1–2:** Required libraries cv2 and numpy are imported for image processing and array operations.
- **Line 5:** The input image is read in grayscale format.
- **Line 8:** The grayscale image is converted into a **binary image** with pixel values 0 and 1 using thresholding.
- **Line 10:** The number of rows and columns of the binary image are extracted.
- **Lines 13–14:** Zero padding is applied vertically (top and bottom) to allow the vertical structuring element to slide over border pixels.

Erosion Stage

- **Line 17:** An empty array is initialized to store the eroded image.
- **Lines 19–26:** Manual erosion is performed using nested loops.
- **Line 21:** A vertical **3×1 region** is extracted from the padded image.
- **Line 22:** The **FIT condition** is checked. If all pixels under the kernel are 1, erosion output is set to 1.
- **Lines 24–25:** If FIT condition is not satisfied, the output pixel is set to 0.

Dilation Stage

- **Lines 28–29:** The eroded image is padded vertically to prepare for dilation.
- **Line 32:** An empty array is initialized to store the opened image.
- **Lines 34–41:** Manual dilation is performed using nested loops.
- **Line 36:** A vertical **3×1 region** is extracted from the padded eroded image.
- **Line 37:** The **HIT or FIT condition** is checked. If at least one pixel matches, output is set to 1.
- **Lines 39–40:** If no pixel matches, the output is set to 0.

- **Line 43:** The final opened image is saved after converting pixel values back to display range.

Output:

Observation

After applying opening, small isolated pixels and thin noise structures are removed while larger objects retain their original shape. This confirms the noise-removal property of opening as defined in morphological theory.

2.4 Closing

Closing is a morphological operation used to **fill small holes, gaps, and breaks** in foreground objects of a binary image. It smoothens object boundaries and connects nearby components. Closing is a **compound operation** that consists of **dilation followed by erosion** using the same structuring element.

Formula:

$$A \cdot B = (A \oplus B) \ominus B$$

Where:

- A= input binary image
- B= structuring element
- $\oplus$= dilation
- $\ominus$= erosion

Algorithm:

1. Read the input image and convert it to binary.
2. Apply **dilation** on the image using structuring element B.
3. Apply **erosion** on the dilated image using the same structuring element.
4. Store the final result as the closed image.

Implementation:

Closing is implemented manually by first dilating the input image and then eroding the dilated result. Both operations strictly follow the HIT–FIT–MISS rules as defined in the textbook, without using any built-in morphological functions.

Code:

```python
import cv2
import numpy as np

# Read image
img = cv2.imread("image.png", cv2.IMREAD_GRAYSCALE)

# Convert to binary image
_, binary = cv2.threshold(img, 127, 1, cv2.THRESH_BINARY)

rows, cols = binary.shape

# Padding for vertical kernel
padded = np.zeros((rows + 2, cols), dtype=np.uint8)
padded[1:rows+1, :] = binary

# ----- Dilation -----
dilated = np.zeros_like(binary)

for i in range(rows):
    for j in range(cols):
        region = padded[i:i+3, j]
        if np.sum(region) >= 1:      # HIT or FIT
            dilated[i, j] = 1
        else:
            dilated[i, j] = 0

# Padding dilated image
padded_dilated = np.zeros((rows + 2, cols), dtype=np.uint8)
padded_dilated[1:rows+1, :] = dilated

# ----- Erosion -----
closed = np.zeros_like(binary)

for i in range(rows):
    for j in range(cols):
        region = padded_dilated[i:i+3, j]
        if np.sum(region) == 3:      # FIT
            closed[i, j] = 1
        else:
            closed[i, j] = 0

# Save output
cv2.imwrite("closed_image.png", closed * 255)
```

Code Explanation:

General

- **Lines 1–2:** Required libraries cv2 and numpy are imported.
- **Line 5:** Input image is read in grayscale format.
- **Line 8:** Image is converted into a binary image with values 0 and 1.
- **Line 10:** Image dimensions (rows and columns) are extracted.
- **Lines 13–14:** Vertical zero padding is applied to allow kernel movement at borders.

Dilation Stage

- **Line 17:** Output array for the dilated image is initialized.
- **Lines 19–25:** Manual dilation is performed using nested loops.
- **Line 21:** A vertical **3×1 neighborhood** is extracted.
- **Line 22:** HIT or FIT condition is checked.
- **Lines 23–25:** Output pixel is set according to dilation rules.

Erosion Stage

- **Lines 28–29:** Padding is applied to the dilated image.
- **Line 32:** Output array for the closed image is initialized.
- **Lines 34–40:** Manual erosion is performed using nested loops.
- **Line 36:** A vertical **3×1 neighborhood** is extracted.
- **Line 37:** FIT condition is checked (all pixels must match).
- **Lines 38–40:** Output pixel is set according to erosion rules.

- **Line 43:** Final closed image is saved after converting values to display range.

Output:

Observation

After applying closing, small holes and gaps inside foreground objects are filled. Disconnected regions become connected, and object boundaries appear smoother, confirming the gap-filling property of closing.

CHAPTER 3: LINE DETECTION AND POINT DETECTION

3.1 Line Detection

Line detection is an image processing technique used to identify straight-line structures such as edges, boundaries, and linear features in an image. It works by highlighting pixels where there is a significant change in intensity along a specific direction (horizontal, vertical, or diagonal).

Formula:

Horizontal Line Mask

$$\begin{bmatrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{bmatrix}$$

Vertical Line Mask

$$\begin{bmatrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{bmatrix}$$

Algorithm:

1. Read the input image and convert it to grayscale.
2. Add zero padding to the image.
3. Select a line detection mask (horizontal or vertical).
4. Slide the mask over the image manually.
5. Multiply and sum the neighborhood pixels with the mask.
6. Store the resulting pixel values in the output image.

Implementation:

Line detection is implemented manually by convolving the input image with a line detection mask using nested loops. Padding and convolution are performed explicitly without using any built-in edge detection functions.

Code:

```
1   import cv2
2   import numpy as np
3
4   # Read image
5   img = cv2.imread("image.png", cv2.IMREAD_GRAYSCALE)
6
7   rows, cols = img.shape
8
9   # Vertical line detection mask
10  kernel = np.array([[-1,  2, -1],
11                     [-1,  2, -1],
12                     [-1,  2, -1]])
13
14  # Zero padding
15  padded = np.zeros((rows + 2, cols + 2), dtype=np.int16)
16  padded[1:rows+1, 1:cols+1] = img
17
18  # Output image
19  line_image = np.zeros_like(img)
20
21  # Manual convolution
22  for i in range(rows):
23      for j in range(cols):
24          region = padded[i:i+3, j:j+3]
25          value = np.sum(region * kernel)
26          value = min(max(value, 0), 255)
27          line_image[i, j] = value
28
29  # Save output
30  cv2.imwrite("line_detected.png", line_image)
```

Code Explanation:

- **Lines 1–2:** Required libraries are imported.
- **Line 5:** Input image is read in grayscale format.
- **Line 7:** Image dimensions are extracted.
- **Lines 10–12:** Vertical line detection mask is defined.
- **Lines 15–16:** Zero padding is applied to handle border pixels.
- **Line 19:** Output image array is initialized.
- **Lines 22–23:** Nested loops slide the mask across the image.
- **Line 24:** A 3×3 neighborhood is extracted.
- **Line 25:** Mask values are multiplied with image pixels and summed.

- **Line 26:** Pixel values are clipped to valid range.
- **Line 27:** Output pixel is assigned.
- **Line 30:** Line-detected image is saved.

Output:

Observation

After applying line detection, vertical linear features in the image are clearly highlighted, while non-linear regions are suppressed. This confirms that the line detection mask responds strongly to intensity variations along the vertical direction.

3.2 Point Detection

Point detection is an image processing technique used to identify **isolated points** or pixels that differ significantly in intensity from their surrounding neighbors. It highlights locations where sudden intensity changes occur in all directions.

Formula:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Algorithm:

1. Read the input image and convert it to grayscale.
2. Add zero padding around the image.
3. Slide the point detection mask across the image.
4. Multiply and sum the neighborhood pixel values with the mask.
5. Store the resulting pixel values in the output image.

Implementation:

Point detection is implemented manually by convolving the input image with a point detection mask using nested loops. No built-in edge or point detection functions are used.

Code:

```
1   import cv2
2   import numpy as np
3
4   # Read image
5   img = cv2.imread("image.png", cv2.IMREAD_GRAYSCALE)
6
7   rows, cols = img.shape
8
9   # Point detection mask
10  kernel = np.array([[-1, -1, -1],
11                     [-1,  8, -1],
12                     [-1, -1, -1]])
13
14  # Zero padding
15  padded = np.zeros((rows + 2, cols + 2), dtype=np.int16)
16  padded[1:rows+1, 1:cols+1] = img
17
18  # Output image
19  point_image = np.zeros_like(img)
20
21  # Manual convolution
22  for i in range(rows):
23      for j in range(cols):
24          region = padded[i:i+3, j:j+3]
25          value = np.sum(region * kernel)
26          value = min(max(value, 0), 255)
27          point_image[i, j] = value
28
29  # Save output
30  cv2.imwrite("point_detected.png", point_image)
```

Code Explanation:

- **Lines 1–2:** Required libraries are imported.
- **Line 5:** Input image is read in grayscale format.
- **Line 7:** Image dimensions are extracted.
- **Lines 10–12:** Point detection mask is defined.
- **Lines 15–16:** Zero padding is applied to support border processing.
- **Line 19:** Output image array is initialized.
- **Lines 22–23:** Nested loops slide the mask across the image.
- **Line 24:** A 3×3 neighborhood is extracted.
- **Line 25:** Kernel and pixel values are multiplied and summed.
- **Line 26:** Pixel values are clipped to valid range.

- **Line 27:** Output pixel is assigned.
- **Line 30:** Point-detected image is saved.

Output:

Observation

After applying point detection, isolated pixels and sharp intensity changes are clearly highlighted, while uniform regions remain suppressed. This confirms the effectiveness of the point detection mask.

CHAPTER 4: GLOBAL THRESHOLDING

Global thresholding is an image segmentation technique used to separate foreground objects from the background based on a **single fixed threshold value**. Each pixel in the image is compared with the threshold, and the pixel is classified as foreground or background accordingly. This method is simple and effective when there is a clear intensity difference between objects and background.

Formula:

$$g(x,y) = \begin{cases} 1, & \text{if } f(x,y) \geq T \\ 0, & \text{if } f(x,y) < T \end{cases}$$

Where:

- $f(x,y)$ = input image pixel
- $g(x,y)$ = output binary image
- T = global threshold value

Algorithm:

1. Read the input image and convert it to grayscale.
2. Select a fixed global threshold value T.
3. Compare each pixel value with T.
4. Assign output pixel as 1 if pixel $\geq T$, else assign 0.
5. Store the result as a binary image.

Implementation:

Global thresholding is implemented manually by comparing each pixel intensity with a predefined threshold value using nested loops. No built-in thresholding functions are used.

Code:

```python
import cv2
import numpy as np

# Read image
img = cv2.imread("image.png", cv2.IMREAD_GRAYSCALE)

rows, cols = img.shape

# Threshold value
T = 170

# Output binary image
binary = np.zeros_like(img)

# Manual thresholding
for i in range(rows):
    for j in range(cols):
        if img[i, j] >= T:
            binary[i, j] = 255
        else:
            binary[i, j] = 0

# Save output
cv2.imwrite("global_threshold.png", binary)
```

Code Explanation:

- **Lines 1–2:** Required libraries cv2 and numpy are imported.
- **Line 5:** The input image is read in grayscale format.
- **Line 7:** The number of rows and columns of the image are extracted.
- **Line 10:** A fixed global threshold value **T** is defined.
- **Line 13:** An empty array is initialized to store the binary output image.

Thresholding Process

- **Lines 16–21:** Manual global thresholding is performed using nested loops.
- **Line 18:** If the pixel intensity is greater than or equal to the threshold value, it is classified as foreground.
- **Lines 19–21:** Pixels below the threshold are classified as background.
- **Line 24:** The final globally thresholded image is saved.

Output:

Observation

After applying global thresholding, the image is successfully segmented into foreground and background regions. This method works well when the intensity difference between objects and background is distinct, but may fail under uneven lighting conditions.

CONCLUSION

This activity successfully demonstrated the implementation of fundamental image processing techniques using manual, textbook-based approaches in Python. Spatial filtering operations such as Gaussian Blur, Image Sharpening, and Unsharp Masking were implemented using explicit convolution with predefined kernels, helping in understanding noise reduction and edge enhancement mechanisms. Morphological operations including Dilation, Erosion, Opening, and Closing were performed using a vertical structuring element and HIT–FIT–MISS rules, clearly illustrating their effects on binary images such as object expansion, shrinkage, noise removal, and gap filling.

Further, Line Detection and Point Detection techniques were implemented using standard masks to identify linear features and isolated points, highlighting intensity variations in specific patterns. Finally, Global Thresholding was applied using a fixed threshold value to segment the image into foreground and background regions.

By avoiding built-in image processing functions and implementing all operations manually, this activity provided a deeper insight into the underlying algorithms and mathematical concepts of image processing. The results obtained validate the theoretical principles studied in the course and emphasize the importance of spatial and morphological operations in real-world image analysis applications.