write program to display image matrix

# Example image matrix

image\_matrix = [

[0, 0, 0, 0, 0],

[0, 255, 255, 255, 0],

[0, 255, 255, 255, 0],

[0, 255, 255, 255, 0],

[0, 0, 0, 0, 0]

]

# Displaying the image matrix

for row in image\_matrix:

row\_str = ""

for pixel in row:

if pixel == 0:

row\_str += " . " # Print a dot for 0 (black)

else:

row\_str += " # " # Print a hash for 255 (white)

print(row\_str)

write program to display Image Histogram

# Example image matrix

image\_matrix = [

[0, 0, 0, 0, 0],

[0, 255, 255, 255, 0],

[0, 255, 255, 255, 0],

[0, 255, 255, 255, 0],

[0, 0, 0, 0, 0]

]

# Calculate the histogram

def calculate\_histogram(image):

histogram = [0] \* 256

for row in image:

for pixel in row:

histogram[pixel] += 1

return histogram

# Display the histogram

def display\_histogram(histogram):

for i, count in enumerate(histogram):

print(f"Intensity {i}: {'\*' \* count}")

# Example usage

image\_histogram = calculate\_histogram(image\_matrix)

display\_histogram(image\_histogram)

write program to find Histogram equalization and display that image

# Example image matrix

image\_matrix = [

[10, 20, 30, 40, 50],

[60, 70, 80, 90, 100],

[110, 120, 130, 140, 150],

[160, 170, 180, 190, 200],

[210, 220, 230, 240, 250]

]

# Calculate histogram

def calculate\_histogram(image):

histogram = [0] \* 256

for row in image:

for pixel in row:

histogram[pixel] += 1

return histogram

# Perform histogram equalization

def histogram\_equalization(image, image\_histogram):

total\_pixels = sum(image\_histogram)

cum\_histogram = [sum(image\_histogram[:i+1]) for i in range(len(image\_histogram))]

equalized\_image = [[0 for \_ in range(len(row))] for row in image]

for i in range(len(image)):

for j in range(len(image[i])):

equalized\_image[i][j] = 255 \* cum\_histogram[image[i][j]] / total\_pixels

return equalized\_image

# Display the image

def display\_image(image):

for row in image:

row\_str = ""

for pixel in row:

row\_str += f"{int(pixel):<4}"

print(row\_str)

# Example usage

image\_histogram = calculate\_histogram(image\_matrix)

equalized\_image = histogram\_equalization(image\_matrix, image\_histogram)

display\_image(equalized\_image)

write program to smooth image using gaussian filter and display image

def convolve2D(image, kernel):

m, n = kernel.shape

y, x = image.shape

y = y - m + 1

x = x - m + 1

new\_image = np.zeros((y, x))

for i in range(y):

for j in range(x):

new\_image[i][j] = np.sum(image[i:i+m, j:j+m]\*kernel)

return new\_image

def gaussian\_kernel(size, sigma=1):

size = int(size) // 2

x, y = np.mgrid[-size:size+1, -size:size+1]

normal = 1 / (2.0 \* np.pi \* sigma\*\*2)

kernel = np.exp(-((x\*\*2 + y\*\*2) / (2.0\*sigma\*\*2))) \* normal

return kernel

# Implementing Gaussian Filter

def gaussian\_filter(image, kernel\_size, sigma):

kernel = gaussian\_kernel(kernel\_size, sigma)

return convolve2D(image, kernel)

# Example Usage

# Assume 'image' is your grayscale image data

smoothed\_image = gaussian\_filter(image, 3, 1) # 3x3 kernel size with sigma = 1

# Display the smoothed image (you need to implement the display logic as per your environment)

write program to find 1st order derivative and display image and first order derivative

# Example image matrix

image\_matrix = [

[10, 20, 30, 40, 50],

[60, 70, 80, 90, 100],

[110, 120, 130, 140, 150],

[160, 170, 180, 190, 200],

[210, 220, 230, 240, 250]

]

# Calculate the first-order derivative in the x-direction

def first\_order\_derivative\_x(image):

derivative\_x = [[0 for \_ in range(len(image[0]))] for \_ in range(len(image))]

for i in range(len(image)):

for j in range(1, len(image[0])):

derivative\_x[i][j] = image[i][j] - image[i][j-1]

return derivative\_x

# Calculate the first-order derivative in the y-direction

def first\_order\_derivative\_y(image):

derivative\_y = [[0 for \_ in range(len(image[0]))] for \_ in range(len(image))]

for j in range(len(image[0])):

for i in range(1, len(image)):

derivative\_y[i][j] = image[i][j] - image[i-1][j]

return derivative\_y

# Display the derivative matrices

def display\_derivative(derivative\_matrix):

for row in derivative\_matrix:

row\_str = ""

for pixel in row:

row\_str += f"{int(pixel):<5}"

print(row\_str)

# Example usage

derivative\_x\_matrix = first\_order\_derivative\_x(image\_matrix)

print("First Order Derivative in X-Direction:")

display\_derivative(derivative\_x\_matrix)

derivative\_y\_matrix = first\_order\_derivative\_y(image\_matrix)

print("\nFirst Order Derivative in Y-Direction:")

display\_derivative(derivative\_y\_matrix)

write program to find second order derivative and display image and second order derivative

# Example image matrix

image\_matrix = [

[10, 20, 30, 40, 50],

[60, 70, 80, 90, 100],

[110, 120, 130, 140, 150],

[160, 170, 180, 190, 200],

[210, 220, 230, 240, 250]

]

# Calculate the second-order derivative in the x-direction

def second\_order\_derivative\_x(image):

derivative\_x = [[0 for \_ in range(len(image[0]))] for \_ in range(len(image))]

for i in range(len(image)):

for j in range(1, len(image[0])-1):

derivative\_x[i][j] = image[i][j+1] - 2 \* image[i][j] + image[i][j-1]

return derivative\_x

# Calculate the second-order derivative in the y-direction

def second\_order\_derivative\_y(image):

derivative\_y = [[0 for \_ in range(len(image[0]))] for \_ in range(len(image))]

for j in range(len(image[0])):

for i in range(1, len(image)-1):

derivative\_y[i][j] = image[i+1][j] - 2 \* image[i][j] + image[i-1][j]

return derivative\_y

# Display the derivative matrices

def display\_derivative(derivative\_matrix):

for row in derivative\_matrix:

row\_str = ""

for pixel in row:

row\_str += f"{int(pixel):<5}"

print(row\_str)

# Example usage

derivative\_x\_matrix = second\_order\_derivative\_x(image\_matrix)

print("Second Order Derivative in X-Direction:")

display\_derivative(derivative\_x\_matrix)

derivative\_y\_matrix = second\_order\_derivative\_y(image\_matrix)

print("\nSecond Order Derivative in Y-Direction:")

display\_derivative(derivative\_y\_matrix)

write program to determine the edges using connected set V. Draw line to draw an object border

# Example image matrix

image\_matrix = [

[0, 0, 0, 0, 0],

[0, 1, 1, 1, 0],

[0, 1, 1, 1, 0],

[0, 1, 1, 1, 0],

[0, 0, 0, 0, 0]

]

# Function to determine the edges using the connected set V

def connected\_set\_v(image):

edges = set()

rows = len(image)

cols = len(image[0])

# Check each pixel for edge condition

for i in range(rows):

for j in range(cols):

if image[i][j] == 1:

if i == 0 or i == rows - 1 or j == 0 or j == cols - 1:

edges.add((i, j))

else:

if image[i-1][j] == 0 or image[i+1][j] == 0 or image[i][j-1] == 0 or image[i][j+1] == 0:

edges.add((i, j))

return edges

# Function to display the object border using lines

def display\_object\_border(edges):

max\_row = max(edges, key=lambda x: x[0])[0]

max\_col = max(edges, key=lambda x: x[1])[1]

for row in range(max\_row + 1):

line = ""

for col in range(max\_col + 1):

if (row, col) in edges:

line += "# "

else:

line += ". "

print(line)

# Example usage

edges = connected\_set\_v(image\_matrix)

display\_object\_border(edges)

write program to work on 1st order and second order derivative

# Example image matrix

image\_matrix = [

[10, 20, 30, 40, 50],

[60, 70, 80, 90, 100],

[110, 120, 130, 140, 150],

[160, 170, 180, 190, 200],

[210, 220, 230, 240, 250]

]

# Calculate the first-order derivative in the x-direction

def first\_order\_derivative\_x(image):

derivative\_x = [[0 for \_ in range(len(image[0]))] for \_ in range(len(image))]

for i in range(len(image)):

for j in range(1, len(image[0])):

derivative\_x[i][j] = image[i][j] - image[i][j-1]

return derivative\_x

# Calculate the second-order derivative in the x-direction

def second\_order\_derivative\_x(image):

derivative\_x = [[0 for \_ in range(len(image[0]))] for \_ in range(len(image))]

for i in range(len(image)):

for j in range(1, len(image[0])-1):

derivative\_x[i][j] = image[i][j+1] - 2 \* image[i][j] + image[i][j-1]

return derivative\_x

# Display the derivative matrices

def display\_derivative(derivative\_matrix):

for row in derivative\_matrix:

row\_str = ""

for pixel in row:

row\_str += f"{int(pixel):<5}"

print(row\_str)

# Example usage

print("First Order Derivative in X-Direction:")

derivative\_x\_matrix = first\_order\_derivative\_x(image\_matrix)

display\_derivative(derivative\_x\_matrix)

print("\nSecond Order Derivative in X-Direction:")

second\_derivative\_x\_matrix = second\_order\_derivative\_x(image\_matrix)

display\_derivative(second\_derivative\_x\_matrix)

write program to determine Image gradient using Sobel operators

# Example image matrix

image\_matrix = [

[10, 20, 30, 40, 50],

[60, 70, 80, 90, 100],

[110, 120, 130, 140, 150],

[160, 170, 180, 190, 200],

[210, 220, 230, 240, 250]

]

# Sobel operators for gradient calculation

sobel\_x = [[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]]

sobel\_y = [[-1, -2, -1], [0, 0, 0], [1, 2, 1]]

# Perform convolution operation

def convolve2D(image, kernel):

m, n = len(kernel), len(kernel[0])

y, x = len(image), len(image[0])

y = y - m + 1

x = x - n + 1

new\_image = [[0 for \_ in range(x)] for \_ in range(y)]

for i in range(y):

for j in range(x):

new\_image[i][j] = sum(sum(image[i+k][j+l] \* kernel[k][l] for l in range(n)) for k in range(m)

)

return new\_image

# Calculate image gradients using Sobel operators

gradient\_x = convolve2D(image\_matrix, sobel\_x)

gradient\_y = convolve2D(image\_matrix, sobel\_y)

# Display the gradients

print("Gradient in X direction:")

for row in gradient\_x:

print(row)

print("\nGradient in Y direction:")

for row in gradient\_y:

print(row)

image enhancement using Fourier transformations

# Example grayscale image matrix

image\_matrix = [

[10, 20, 30, 40, 50],

[60, 70, 80, 90, 100],

[110, 120, 130, 140, 150],

[160, 170, 180, 190, 200],

[210, 220, 230, 240, 250]

]

# Discrete Fourier Transform (DFT) calculation

def dft2D(image):

M = len(image)

N = len(image[0])

dft\_result = [[0 for \_ in range(N)] for \_ in range(M)]

for u in range(M):

for v in range(N):

sum\_val = 0

for x in range(M):

for y in range(N):

pixel\_val = image[x][y]

term = pixel\_val \* (complex((2 \* 3.1415 \* (u \* x / M + v \* y / N))).exp())

sum\_val += term

dft\_result[u][v] = sum\_val

return dft\_result

# Inverse Discrete Fourier Transform (IDFT) calculation

def idft2D(dft\_result):

M = len(dft\_result)

N = len(dft\_result[0])

idft\_image = [[0 for \_ in range(N)] for \_ in range(M)]

for x in range(M):

for y in range(N):

sum\_val = 0

for u in range(M):

for v in range(N):

dft\_val = dft\_result[u][v]

term = dft\_val \* (complex((-2 \* 3.1415 \* (u \* x / M + v \* y / N))).exp())

sum\_val += term

idft\_image[x][y] = int(sum\_val.real / (M \* N))

return idft\_image

# Example usage

dft\_result = dft2D(image\_matrix)

enhanced\_image\_matrix = idft2D(dft\_result)

# Display enhanced image

for row in enhanced\_image\_matrix:

print(row)