الاسم : محمد محمد عفيفي فتوح

السكشن : 3

[Code]

import math

import os

# setting output folder

output\_path = str(os.path.dirname(os.path.realpath("\_\_file\_\_"))) + "/output"

if not os.path.exists(output\_path):

    os.makedirs(output\_path)

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* The Canny operator as a multi-step algorithm \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* #

import numpy as np

import cv2

def canny(img):

    # Step 1: Gaussian smoothing & normalization

    gaussian\_output = gaussian\_smoothing(img)

    # Step 2: Gradient Operator

    gradient\_magnitude, gradient\_angle = gradient\_operation(gaussian\_output)

    # Step 3:Non maxima suppression

    non\_maxima\_suppressed = non\_maxima\_suppression(gradient\_magnitude, gradient\_angle)

    # Step 4:Double thresholding

    double\_thresholding(non\_maxima\_suppressed, gradient\_angle)

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* #

# Step 1: Gaussian smoothing & normalization

def gaussian\_smoothing(input\_image):

    # Loading image in form of a matrix

    image = cv2.imread(input\_image, 0)

    # gaussian mask initiated

    gaussian\_mask = np.array(

        [

            [ 1, 1, 2, 2, 2, 1, 1 ],

            [ 1, 2, 2, 4, 2, 2, 1 ],

            [ 2, 2, 4, 8, 4, 2, 2 ],

            [ 2, 4, 8, 16, 8, 4, 2 ],

            [ 2, 2, 4, 8, 4, 2, 2 ],

            [ 1, 2, 2, 4, 2, 2, 1 ],

            [ 1, 1, 2, 2, 2, 1, 1 ]

        ])

    image\_height = image.shape[ 0 ]

    image\_width = image.shape[ 1 ]

    # Initializing 2-D array to store convolution values

    convoluted\_gaussian\_arr = np.empty((image\_height, image\_width))

    for row in range(image\_height):

        for col in range(image\_width):

            # setting the undefined pixel value to 0

            if 0 <= row < 3 or image\_height - 3 <= row <= image\_height - 1 or 0 <= col < 3 or image\_width - 3 <= col <= image\_width - 1:

                convoluted\_gaussian\_arr[ row ][ col ] = 0

            else:

                x = 0

                # calculating convoluted values for each pixel position and storing it in 'convoluted\_gaussian\_arr'

                for k in range(7):

                    for l in range(7):

                        x = x + (image[ row - 3 + k ][ col - 3 + l ] \* gaussian\_mask[ k ][ l ])

                # normalizing the value

                convoluted\_gaussian\_arr[ row ][ col ] = x / 140

    # Writing output image from Gaussian Smoothing

    cv2.imwrite(os.path.join(output\_path, "gaussian\_smoothing.bmp"), convoluted\_gaussian\_arr)

    return convoluted\_gaussian\_arr

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* #

# step 2: gradient calculation function which takes in the gaussian convoluted matrix as argument

def gradient\_operation(smoothed\_image):

    # Sobel's operator for Gx

    sobel\_gx = np.array([ [ -1, 0, 1 ], [ -2, 0, 2 ], [ -1, 0, 1 ] ])

    # Sobel's operator for Gy

    sobel\_gy = np.array([ [ 1, 2, 1 ], [ 0, 0, 0 ], [ -1, -2, -1 ] ])

    image\_height = smoothed\_image.shape[ 0 ]

    image\_width = smoothed\_image.shape[ 1 ]

    # initializing 2-d array to store the horizontal and vertical gradient

    horizontal\_gradient = np.zeros((image\_height, image\_width))

    vertical\_gradient = np.zeros((image\_height, image\_width))

    for row in range(image\_height):

        for col in range(image\_width):

            # sets the undefined pixel value to 0

            if 0 <= row < 4 or image\_height - 4 <= row <= image\_height - 1 or 0 <= col < 4 or image\_width - 4 <= col <= image\_width - 1:

                horizontal\_gradient[ row ][ col ] = 0

            else:

                x = 0

                for i in range(3):

                    for j in range(3):

                        # calculating horizontal gradient value

                        x = x + ((smoothed\_image[ row - 1 + i ][ col - 1 + j ]) \* sobel\_gx[ i ][ j ])

                # normalizing the pixel values

                horizontal\_gradient[ row ][ col ] = x / 3

    for row in range(image\_height):

        for col in range(image\_width):

            # setting the undefined pixel value to 0

            if 0 <= row < 4 or image\_height - 4 <= row <= image\_height - 1 or 0 <= col < 4 or image\_width - 4 <= col <= image\_width - 1:

                vertical\_gradient[ row ][ col ] = 0

            else:

                x = 0

                # calculating vertical gradient value for each pixel

                for i in range(3):

                    for j in range(3):

                        x = x + (smoothed\_image[ row - 1 + i ][ col - 1 + j ] \* sobel\_gy[ i ][ j ])

                # normalizing the pixel values

                vertical\_gradient[ row ][ col ] = x / 3

    # initializing 2-d array to store gradient angles

    gradient\_angle = np.empty((image\_height, image\_width))

    for row in range(image\_height):

        for col in range(image\_width):

            # checking if the denominator is 0

            if horizontal\_gradient[ row ][ col ] == 0:

                gradient\_angle[ row ][ col ] = 90

            else:

                gradient\_angle[ row ][ col ] = math.degrees(

                    math.atan((vertical\_gradient[ row ][ col ]) / (horizontal\_gradient[ row ][ col ])))

    # calculating absolute values

    for row in range(image\_height):

        for col in range(image\_width):

            horizontal\_gradient[ row ][ col ] = abs(horizontal\_gradient[ row ][ col ])

            vertical\_gradient[ row ][ col ] = abs(vertical\_gradient[ row ][ col ])

    # writing the horizontal and vertical gradient

    cv2.imwrite(os.path.join(output\_path, "horizontal\_gradient.bmp"), horizontal\_gradient)

    cv2.imwrite(os.path.join(output\_path, "vertical\_gradient.bmp"), vertical\_gradient)

    # initializing 2-d array to store gradient magnitude with normalized values (divided by root(2))

    gradient\_mag = np.empty((image\_height, image\_width))

    # inserts values in the gradient magnitude matrix

    for row in range(image\_height):

        for col in range(image\_width):

            x = math.pow(horizontal\_gradient[ row ][ col ], 2)

            y = math.pow(vertical\_gradient[ row ][ col ], 2)

            gradient\_mag[ row ][ col ] = (math.sqrt(x + y)) / math.sqrt(2)

    # writing gradient magnitude

    cv2.imwrite(os.path.join(output\_path, "gradient\_image.bmp"), gradient\_mag)

    return gradient\_mag, gradient\_angle

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* #

# 3. Apply non-maxima suppression to gradient magnitude

def non\_maxima\_suppression(gradient\_mag, gradient\_angle):

    image\_height = gradient\_mag.shape[ 0 ]

    image\_width = gradient\_mag.shape[ 1 ]

    # a zero matrix initiated to hold the values after non-maxima suppression

    suppressed\_arr = np.zeros((image\_height, image\_width), dtype='int')

    # for loops iterating each pixel

    for i in range(image\_height):

        for j in range(image\_width):

            # sets the undefined pixel value to 0

            if 0 <= i < 5 or image\_height - 5 <= i <= image\_height - 1 or 0 <= j < 5 or image\_width - 5 <= j <= image\_width - 1:

                suppressed\_arr[ i ][ j ] = 0

            else:

                x = gradient\_angle[ i ][ j ]

                y = gradient\_mag[ i ][ j ]

                # checking if gradient angle is in 0th sector

                if (-22.5 < x <= 22.5) or ((180 >= x > 157.5) and (-180 <= x <= -157.5)):

                    # checks if the current pixel is greater than the sector neighbors

                    if y >= gradient\_mag[ i ][ j - 1 ] and y >= gradient\_mag[ i ][ j + 1 ]:

                        suppressed\_arr[ i ][ j ] = int(round(y))

                # checking if gradient angle is in 1st sector

                elif (22.5 < x <= 67.5) or (-112.5 >= x > -157.5):

                    # checks if the current pixel is greater than the sector neighbors

                    if y >= gradient\_mag[ i - 1 ][ j + 1 ] and y >= gradient\_mag[ i + 1 ][ j - 1 ]:

                        suppressed\_arr[ i ][ j ] = int(round(y))

                # checking if gradient angle is in 2nd sector

                elif (67.5 < x <= 112.5) or (-67.5 >= x > -112.5):

                    # checks if the current pixel is greater than the sector neighbors

                    if y >= gradient\_mag[ i - 1 ][ j ] and y >= gradient\_mag[ i + 1 ][ j ]:

                        suppressed\_arr[ i ][ j ] = int(round(y))

                # checking if gradient angle is in 3rd sector

                elif (112.5 < x <= 157.5) or (-22.5 >= x > -67.5):

                    # checks if the current pixel is greater than the sector neighbors

                    if y >= gradient\_mag[ i - 1 ][ j - 1 ] and y >= gradient\_mag[ i + 1 ][ j + 1 ]:

                        suppressed\_arr[ i ][ j ] = int(round(y))

    # Writing suppressed matrix

    cv2.imwrite(os.path.join(output\_path, "non-maxima\_suppressed.bmp"), suppressed\_arr)

    return suppressed\_arr

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* #

# Step 4. Thresholding

def double\_thresholding(suppressed\_image, gradient\_angle):

    image\_height = suppressed\_image.shape[ 0 ]

    image\_width = suppressed\_image.shape[ 1 ]

    maximum\_value = np.amax(suppressed\_image)

    high\_threshold\_ratio = 0.20

    low\_threshold\_ratio = 0.10

    high\_threshold = maximum\_value \* high\_threshold\_ratio

    low\_threshold = maximum\_value \* low\_threshold\_ratio

    output\_image = np.empty((image\_height, image\_width), dtype='int')

    strong\_edge\_pixel = suppressed\_image > high\_threshold

    output\_image[ strong\_edge\_pixel ] = 255

    no\_edge\_pixel = suppressed\_image < low\_threshold

    output\_image[ no\_edge\_pixel ] = 0

    weak\_edge\_pixel = (suppressed\_image >= low\_threshold) & (suppressed\_image <= high\_threshold)

    index\_weak\_edge\_pixel = np.argwhere(weak\_edge\_pixel)

    for i in index\_weak\_edge\_pixel:

        x = i[ 0 ]

        y = i[ 1 ]

        pixel\_gradient = gradient\_angle[ x ][ y ]

        if 0 < x < image\_width and 0 < y < image\_height:

            # checking if any of the 8 neighbors is a strong edge pixel and classifying the pixel as an edge pixel

            if (((suppressed\_image[ x - 1 ][ y ] in strong\_edge\_pixel) and abs(

                    gradient\_angle[ x - 1 ][ y ] - pixel\_gradient) > 45) or

                    ((suppressed\_image[ x + 1 ][ y ] in strong\_edge\_pixel) and abs(

                        gradient\_angle[ x + 1 ][ y ] - pixel\_gradient) > 45) or

                    ((suppressed\_image[ x ][ y + 1 ] in strong\_edge\_pixel) and abs(

                        gradient\_angle[ x ][ y + 1 ] - pixel\_gradient) > 45) or

                    ((suppressed\_image[ x ][ y - 1 ] in strong\_edge\_pixel) and abs(

                        gradient\_angle[ x ][ y - 1 ] - pixel\_gradient) > 45) or

                    ((suppressed\_image[ x - 1 ][ y + 1 ] in strong\_edge\_pixel) and abs(

                        gradient\_angle[ x - 1 ][ y + 1 ] - pixel\_gradient) > 45) or

                    ((suppressed\_image[ x - 1 ][ y - 1 ] in strong\_edge\_pixel) and abs(

                        gradient\_angle[ x - 1 ][ y - 1 ] - pixel\_gradient) > 45) or

                    ((suppressed\_image[ x + 1 ][ y - 1 ] in strong\_edge\_pixel) and abs(

                        gradient\_angle[ x + 1 ][ y - 1 ] - pixel\_gradient) > 45) or

                    ((suppressed\_image[ x + 1 ][ y - 1 ] in strong\_edge\_pixel) and abs(

                        gradient\_angle[ x + 1 ][ y - 1 ] - pixel\_gradient) > 45)):

                output\_image[ x ][ y ] = 255

    # writing the output after double thresholding

    cv2.imwrite(os.path.join(output\_path, "double\_thresholding.bmp"), output\_image)

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* #

if \_\_name\_\_ == "\_\_main\_\_":

    # input name of image with extensions on which you want to apply Canny Edge Detection

    image = input("Please Enter image name: ")

    canny(image)

# reading image

img = cv2.imread(image)

# The Canny operator

cany\_operator = cv2.Canny(img,100,200)

cv2.imwrite(os.path.join(output\_path, "Cany\_Operator.bmp"), cany\_operator)

# converting to gray scale

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

# Smoothing the image with a Gaussian filter

gaussian = cv2.GaussianBlur(gray, (3, 3), 0)

# Sobel Edge Detector

sobelx = cv2.Sobel(gaussian, cv2.CV\_8U, 1, 0, ksize=5)

sobely = cv2.Sobel(gaussian, cv2.CV\_8U, 0, 1, ksize=5)

sobel = sobelx + sobely

cv2.imwrite(os.path.join(output\_path, "Sobel\_Edge\_Detector.bmp"), sobel)

# Prewitt Gradient Operator

kernelx = np.array([[1, 1, 1], [0, 0, 0], [-1, -1, -1]])

kernely = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])

prewittx = cv2.filter2D(gaussian, -1, kernelx)

prewitty = cv2.filter2D(gaussian, -1, kernely)

cv2.imwrite(os.path.join(output\_path, "Prewitt\_Gradient\_Operator.bmp"), prewittx + prewitty)

# Laplacian of Gaussian

laplacian = cv2.Laplacian(img, cv2.CV\_64F)

cv2.imwrite(os.path.join(output\_path, "Laplacian\_of\_Gaussian.bmp"), laplacian)

[Input image]



[Output]

