**Canny Edge Detector**

**Eashan Kaushik – ek3575, N19320245**

**Srijan Malhotra – sm9439, N18390405**

**Instruction to Run Code:**

cv2 library was used for ready so make sure you have cv2 installed.

Or install using **pip install opencv-python** or **pip3 install opencv-python**

Method 1:

From your terminal run the file test.py. This can be done using – **python test.py**

Method 2:

Open the test.ipynb file from jupyter notebook, and run all cells

The output will be generated in the **output folder.**

**Output Images**

House.bmp

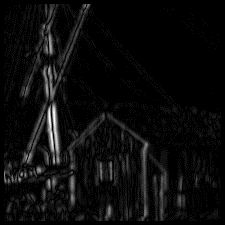
Gaussian Smoothing



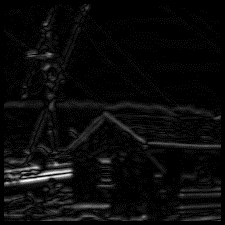
Gradient Magnitude



Gradient x



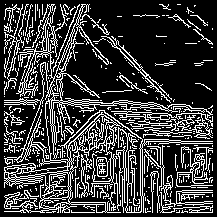
Gradient y



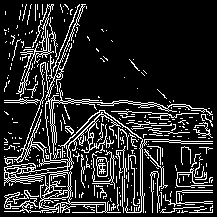
Non Max



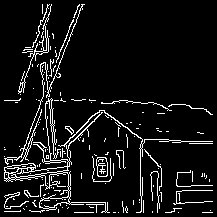
Threshold-25th Percentile



Threshold-50th Percentile



Threshold-75th Percentile

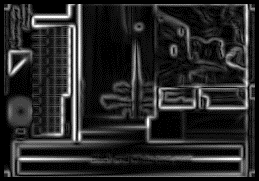


Test Patterns.bmp

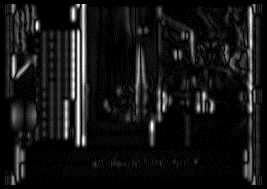
Gaussian Smoothing



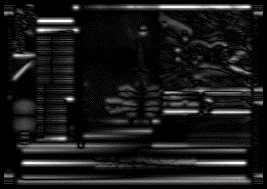
Gradient Magnitude



Gradient x



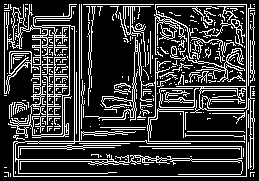
Gradient y



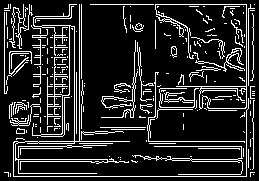
Non Max



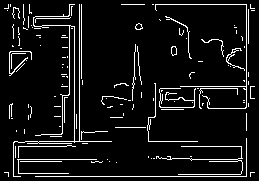
Threshold-25th Percentile



Threshold-50th Percentile



Threshold-75th Percentile



**Source Code:**

Canny.py

####################################################

## Developed by: Eashan Kaushik & Srijan Malhotra ##

## Project Start: 8th October 2021                ##

####################################################

import os

# cv2 is just used for reading images

import cv2

import numpy as np

import math

import matplotlib.pyplot as plt

from PIL import Image as im

import datetime

import math

import copy

import shutil

# The Convolution module is developed by us

from convolution import SeConvolve

# image name to save image after gray scale conversion

GRSC\_PATH = 'grsc.PNG'

class CannyEdgeDetector:

    def \_\_init\_\_(self, image\_path):

        # Path of the image, on which canny detector will be applied

        self.image\_path = image\_path

        # Read Image

        self.image\_read()

        # Gaussian Kernel used for smoothing

        self.\_gaussian\_kernel = 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]])

        # Gradient x and y operations Prewitt's Operators

        self.\_convolution\_matrix\_gx = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])

        self.\_convolution\_matrix\_gy = np.array([[1, 1, 1], [0, 0, 0], [-1, -1, -1]])

        # Output of step 1

        self.\_smoothed\_image = None

        # Output of step 2

        self.\_gradient\_x = None

        self.\_gradient\_y = None

        self.\_gradient\_x\_norm = None

        self.\_gradient\_y\_norm = None

        # Output of step 3

        self.\_magnitude = None

        self.\_magnitude\_norm = None

        # Angle Output

        self.\_angle = None

        self.\_edge\_angle = None

        # Output of step 4

        self.\_non\_max\_output = None

        # Output of step 5

        self.\_threshold\_output\_25 = None

        self.\_threshold\_output\_50 = None

        self.\_threshold\_output\_75 = None

    #######################

    ## Getter and Setter ##

    #######################

    @property

    def gaussian\_kernel(self):

        return self.\_gaussian\_kernel

    @property

    def convolution\_matrix\_gx(self):

        return self.\_convolution\_matrix\_gx

    @property

    def convolution\_matrix\_gy(self):

        return self.\_convolution\_matrix\_gy

    @property

    def image\_matrix(self):

        return self.\_image\_matrix

    @property

    def smoothed\_image(self):

        return self.\_smoothed\_image

    @property

    def gradient\_x(self):

        return self.\_gradient\_x

    @property

    def gradient\_y(self):

        return self.\_gradient\_y

    @property

    def magnitude(self):

        return self.\_magnitude

    @property

    def angle(self):

        return self.\_angle

    @property

    def edge\_angle(self):

        return self.\_edge\_angle

    @property

    def non\_max\_output(self):

        return self.\_non\_max\_output

    @property

    def threshold\_output\_25(self):

        return self.\_threshold\_output\_25

    @property

    def threshold\_output\_50(self):

        return self.\_threshold\_output\_50

    @property

    def threshold\_output\_75(self):

        return self.\_threshold\_output\_75

    #######################

    #######################

    # This function reads the path provided to it and calls the convert\_to\_matrix

    def image\_read(self):

        # self.\_image\_matrix = cv2.imread(self.image\_path, cv2.IMREAD\_GRAYSCALE)

        src = cv2.imread(self.image\_path)

        self.img = cv2.cvtColor(src, cv2.COLOR\_BGR2GRAY)

        now\_time = datetime.datetime.now().strftime("%d%m%Y%H%M%S")

        plt.imsave('gray-op/' + now\_time + '-' + GRSC\_PATH, self.img, cmap='gray')

        self.covert\_to\_matrix('gray-op/' + now\_time + '-' + GRSC\_PATH)

    # This function converts the image to numpy matrix

    def covert\_to\_matrix(self, path):

        gsrc = cv2.imread(path, 0)

        matrix = list()

        for row\_index in range(0, gsrc.shape[0]):

            row = []

            for column\_index in range(0, gsrc.shape[1]):

                pixel = gsrc.item(row\_index, column\_index)

                row.append(pixel)

            matrix.append(row)

        self.\_image\_matrix = np.array(matrix)

    # Main procedure - calls different functions to compute edge detection

    def canny\_detector(self):

        # Gaussian Smoothing - Step1

        self.gaussian\_smoothing()

        # Gradient Operation Prewitt - Step 2 and 3

        self.gradient\_operation()

        # Non-Max Supression - Step4

        self.non\_max\_suppression()

        # Thresholding - Step5

        self.thresholding()

        # Generating output

        self.generate\_output()

    # Step 1: Gaussian Smoothing

    def gaussian\_smoothing(self):

        # Convolution done on the image\_matrix

        smoothing = SeConvolve(self.\_image\_matrix, self.\_gaussian\_kernel)

        \_, self.\_smoothed\_image = smoothing.convolution()

    # Step 2: Gradient Operation

    def gradient\_operation(self):

        # Convolution done on the image\_matrix w.r.t gradient x

        gradient\_x = SeConvolve(self.\_smoothed\_image, self.\_convolution\_matrix\_gx, mode='gradient')

        self.\_gradient\_x, self.\_gradient\_x\_norm = gradient\_x.convolution()

        # Convolution done on the image\_matrix w.r.t gradient y

        gradient\_y = SeConvolve(self.\_smoothed\_image, self.\_convolution\_matrix\_gy, mode='gradient')

        self.\_gradient\_y, self.\_gradient\_y\_norm = gradient\_y.convolution()

        # We compute gradient magnitude, gradient angle and edge angle

        self.\_magnitude, self.\_magnitude\_norm = self.calcuate\_magnitude(self.\_gradient\_x, self.\_gradient\_y)

        self.\_angle, self.\_edge\_angle = self.calculate\_angle(self.\_gradient\_x, self.\_gradient\_y)

    # Step 3: Magnitude computation

    # This function calculates the gradient magnitude

    def calcuate\_magnitude(self, gradient\_x, gradient\_y):

        height, width = gradient\_x.shape

        # After gaussing smoothing and gradient computation we have lost a total of 8 rows and 8 columns

        magnitude = np.zeros((height - 8, width - 8))

        # looping over the desired matrix

        for i in range(4,height - 4):

            for j in range(4,width - 4):

                # gradient calculated using root(gx\*\*2 + gy\*\*2)

                temp = (gradient\_x[i, j] \*\* 2) + (gradient\_y[i, j] \*\* 2)

                magnitude[i - 4, j - 4] = math.sqrt(temp)

        # Nomralization of Magnitude

        magnitude\_norm = magnitude / 360.624

        # same size as original image

        magnitude = np.pad(magnitude, 4, mode='constant')

        magnitude\_norm = np.pad(magnitude\_norm, 4, mode='constant')

        return magnitude, magnitude\_norm

    def calculate\_angle(self, gradient\_x, gradient\_y):

        height, width = gradient\_x.shape

        # After gaussing smoothing and gradient computation we have lost a total of 8 rows and 8 columns

        angle = np.zeros((height - 8, width - 8))

        edge\_angle = np.zeros((height - 8, width - 8))

        # looping over the desired matrix

        for i in range(4,height - 4):

            for j in range(4,width - 4):

                if gradient\_x[i, j]  != 0:

                    # gradient angle computed using tan-1(gy/gx)

                    angle[i - 4, j - 4] = math.degrees(math.atan((gradient\_y[i, j] / gradient\_x[i, j])))

                    edge\_angle[i - 4, j - 4] = angle[i - 4, j - 4] + 90

        # same size as original image

        angle = np.pad(angle, 4, mode='constant')

        edge\_angle = np.pad(angle, 4, mode='constant')

        return angle, edge\_angle

    # Step 4: Non-Maxima Suppression

    def non\_max\_suppression(self):

        angle = self.\_angle

        magnitude = copy.deepcopy(self.\_magnitude)

        height, width = magnitude.shape

        # looping over the desired matrix

        for i in range(4,height - 4):

            for j in range(4,width - 4):

                # this code calculates the sector the pixel belongs to according to gradient angle

                if angle[i, j] < 0:

                    current\_sector = self.sector(angle[i, j] + 360)

                else:

                    current\_sector = self.sector(angle[i, j])

                # this code returns which pixel we should compare with according to sector

                check\_one, check\_two = self.check(current\_sector, i, j)

                check\_one\_x, check\_one\_y = check\_one

                check\_two\_x, check\_two\_y = check\_two

                # non max suppression

                if not(magnitude[i, j] > magnitude[check\_one\_x, check\_one\_y] and magnitude[i, j] > magnitude[check\_two\_x, check\_two\_y]):

                    magnitude[i, j] = 0

        self.\_non\_max\_output = magnitude

    # this function returns the sector value according to angle

    def sector(self, angle):

        # for sector 0

        if((0 <= angle <= 22.5) or (337.5 < angle <= 360) or (157.5 < angle <= 202.5)):

            return '0'

        # for sector 1

        elif((67.5 >= angle > 22.5) or (247.5 >= angle > 202.5)):

            return '1'

        # for sector 2

        elif((112.5 >= angle > 67.5) or (292.5 >= angle > 247.5)):

            return '2'

        # for sector 3

        elif((157.5 >= angle > 112.5) or (337.5>= angle > 292.5)):

            return '3'

        else:

            print('wtf')

        # return '0'

    # this function returns which pixel we should compare with according to sector

    def check(self, current\_sector, current\_i, current\_j):

        if(current\_sector == '0'):

            return ((current\_i,current\_j-1), (current\_i,current\_j+1))

        elif(current\_sector == '1'):

            return ((current\_i-1,current\_j+1), (current\_i+1,current\_j-1))

        elif(current\_sector == '2'):

            return ((current\_i-1,current\_j), (current\_i+1,current\_j))

        elif(current\_sector == '3'):

            return ((current\_i-1,current\_j-1), (current\_i+1,current\_j+1))

        else:

            print('wtf')

    # Step 5 Thresholding

    def thresholding(self):

        temp\_magnitude = copy.deepcopy(self.\_non\_max\_output)

        temp\_magnitude = temp\_magnitude[4: temp\_magnitude.shape[0] - 4, 4: temp\_magnitude.shape[1] - 4].flatten()

        temp\_magnitude = temp\_magnitude[temp\_magnitude != 0]

        # output according to 25th Percentile

        magnitude = copy.deepcopy(self.\_non\_max\_output)

        percentile = np.percentile(temp\_magnitude, 25)

        self.\_threshold\_output\_25 = self.threshold(magnitude, percentile)

        # output according to 50th Percentile

        magnitude = copy.deepcopy(self.\_non\_max\_output)

        percentile = np.percentile(temp\_magnitude, 50)

        self.\_threshold\_output\_50 = self.threshold(magnitude, percentile)

        # output according to 75th Percentile

        magnitude = copy.deepcopy(self.\_non\_max\_output)

        percentile = np.percentile(temp\_magnitude, 75)

        self.\_threshold\_output\_75 = self.threshold(magnitude, percentile)

    # this function performs thresholding

    def threshold(self, magnitude, T):

        height, width = magnitude.shape

        # looping over the desired matrix

        for i in range(4,height - 4):

            for j in range(4,width - 4):

                if magnitude[i,j] < T:

                    magnitude[i,j] = 0

                else:

                    magnitude[i, j] = 1

        return magnitude

    # this function is used to save .PNG images of results

    def generate\_output(self):

        name = self.image\_path.split('.')[0].split('/')[-1]

        name = 'Output/' + name

        if os.path.isdir(name):

            shutil.rmtree(name)

        os.mkdir(name)

        # Output of step 1

        plt.imsave(name + '/' + 'GaussianSmoothing.bmp', self.\_smoothed\_image, cmap='gray')

        # Output of step 2

        plt.imsave(name + '/' + 'GradientX.bmp', self.\_gradient\_x\_norm, cmap='gray')

        plt.imsave(name + '/' + 'GradientY.bmp', self.\_gradient\_y\_norm, cmap='gray')

        # Output of step 3

        plt.imsave(name + '/' + 'GradientMagnitude.bmp', self.\_magnitude\_norm, cmap='gray')

        # Output of Step 4

        plt.imsave(name + '/' + 'Non-Max.bmp', self.\_non\_max\_output, cmap='gray')

        # Output of step 5

        plt.imsave(name + '/' + 'T25.bmp', self.\_threshold\_output\_25, cmap='gray')

        plt.imsave(name + '/' + 'T50.bmp', self.\_threshold\_output\_50, cmap='gray')

        plt.imsave(name + '/' + 'T75.bmp', self.\_threshold\_output\_75, cmap='gray')

Convolution.py

####################################################

## Developed by: Eashan Kaushik & Srijan Malhotra ##

## Project Start: 29th October 2021               ##

####################################################

import numpy as np

class SeConvolve:

    def \_\_init\_\_(self, image\_matrix, kernel, mode='smoothing'):

        # input image matrix

        self.image\_matrix = image\_matrix

        # inpur kernel

        self.kernel = kernel

        # output after convolution with the kerne

        self.\_output = None

        # normalized output

        self.\_output\_norm = None

        # mode smoothing or gradient

        self.mode = mode

    #######################

    ## Getter and Setter ##

    #######################

    @property

    def output(self):

        return self.\_output

    @property

    def output\_norm(self):

        return self.\_output\_norm

    #######################

    #######################

    def convolution(self):

      height, width = self.image\_matrix.shape

      # if mode == smoothing

      if self.mode == 'smoothing':

        self.\_output = np.zeros((height - 6, width - 6))

        # code to perform gaussian smoothing

        # looping over the desired matrix

        for i in range(3,height-3):

          for j in range(3,width-3):

            # martix multiplication for convolution

            # 7 x 7 gaussian smoothing leads to loss of 3 rows and 3 columns and thats why we start from i-3 and j-3

            self.\_output[i - 3,j - 3] = (self.kernel[0, 0] \* self.image\_matrix[i - 3, j - 3]) + (self.kernel[0, 1] \* self.image\_matrix[i - 3, j - 2]) + (self.kernel[0, 2] \* self.image\_matrix[i - 3, j - 1]) + \

            (self.kernel[0, 3] \* self.image\_matrix[i - 3, j]) + (self.kernel[0, 4] \* self.image\_matrix[i - 3, j + 1]) + (self.kernel[0, 5] \* self.image\_matrix[i - 3, j + 2]) + \

            (self.kernel[0, 6] \* self.image\_matrix[i - 3, j + 3]) + (self.kernel[1, 0] \* self.image\_matrix[i - 2, j - 3]) + (self.kernel[1, 1] \* self.image\_matrix[i - 2, j - 2]) + \

            (self.kernel[1, 2] \* self.image\_matrix[i - 2, j - 1]) + (self.kernel[1, 3] \* self.image\_matrix[i - 2, j]) + (self.kernel[1, 4] \* self.image\_matrix[i - 2, j + 1]) + \

            (self.kernel[1, 5] \* self.image\_matrix[i - 2, j + 2]) + (self.kernel[1, 6] \* self.image\_matrix[i - 2, j + 3]) + (self.kernel[2, 0] \* self.image\_matrix[i - 1, j - 3]) + \

            (self.kernel[2, 1] \* self.image\_matrix[i - 1, j - 2]) + (self.kernel[2, 2] \* self.image\_matrix[i - 1, j - 1]) + (self.kernel[2, 3] \* self.image\_matrix[i - 1, j]) + \

            (self.kernel[2, 4] \* self.image\_matrix[i - 1, j + 1]) + (self.kernel[2, 5] \* self.image\_matrix[i - 1, j + 2]) + (self.kernel[2, 6] \* self.image\_matrix[i - 1, j + 3]) + \

            (self.kernel[3, 0] \* self.image\_matrix[i, j - 3]) + (self.kernel[3, 1] \* self.image\_matrix[i, j - 2]) + (self.kernel[3, 2] \* self.image\_matrix[i, j - 1]) + \

            (self.kernel[3, 3] \* self.image\_matrix[i, j]) + (self.kernel[3, 4] \* self.image\_matrix[i, j + 1]) + (self.kernel[3, 5] \* self.image\_matrix[i, j + 2]) + \

            (self.kernel[3, 6] \* self.image\_matrix[i, j + 3]) + (self.kernel[4, 0] \* self.image\_matrix[i + 1, j - 3]) + (self.kernel[4, 1] \* self.image\_matrix[i + 1, j - 2]) + \

            (self.kernel[4, 2] \* self.image\_matrix[i + 1, j - 1]) + (self.kernel[4, 3] \* self.image\_matrix[i + 1, j]) + (self.kernel[4, 4] \* self.image\_matrix[i + 1, j + 1]) + \

            (self.kernel[4, 5] \* self.image\_matrix[i + 1, j + 2]) + (self.kernel[4, 6] \* self.image\_matrix[i + 1, j + 3]) + (self.kernel[5, 0] \* self.image\_matrix[i + 2, j - 3]) + \

            (self.kernel[5, 1] \* self.image\_matrix[i + 2, j - 2]) + (self.kernel[5, 2] \* self.image\_matrix[i + 2, j - 1]) + (self.kernel[5, 3] \* self.image\_matrix[i + 2, j]) + \

            (self.kernel[5, 4] \* self.image\_matrix[i + 2, j + 1]) + (self.kernel[5, 5] \* self.image\_matrix[i + 2, j + 2]) + (self.kernel[5, 6] \* self.image\_matrix[i + 2, j + 3]) + \

            (self.kernel[6, 0] \* self.image\_matrix[i + 3, j - 3]) + (self.kernel[6, 1] \* self.image\_matrix[i + 3, j - 2]) + (self.kernel[6, 2] \* self.image\_matrix[i + 3, j - 1]) + \

            (self.kernel[6, 3] \* self.image\_matrix[i + 3, j]) + (self.kernel[6, 4] \* self.image\_matrix[i + 3, j + 1]) + (self.kernel[6, 5] \* self.image\_matrix[i + 3, j + 2]) + \

            (self.kernel[6, 6] \* self.image\_matrix[i + 3, j + 3])

      # if mode == gradient

      elif self.mode == 'gradient':

        # code to find the gradients

        self.\_output = np.zeros((height - 8, width - 8))

        # looping over the desired matrix

        for i in range(4,height - 4):

          for j in range(4,width - 4):

            # martix multiplication for gradient computation

            # prewit convolution after gaussian smoothing leads to loss of 4 rows and 4 columns thats why we start from i-4 and j-4

            self.\_output[i - 4,j - 4] = (self.kernel[0, 0] \* self.image\_matrix[i - 1, j - 1]) + (self.kernel[0, 1] \* self.image\_matrix[i - 1, j]) + (self.kernel[0, 2] \* self.image\_matrix[i - 1, j + 1]) + \

                      (self.kernel[1, 0] \* self.image\_matrix[i, j - 1]) + (self.kernel[1, 1] \* self.image\_matrix[i, j]) + (self.kernel[1, 2] \* self.image\_matrix[i, j + 1]) + (self.kernel[2, 0] \* self.image\_matrix[i + 1, j - 1]) + \

                      (self.kernel[2, 1] \* self.image\_matrix[i + 1, j]) + (self.kernel[2, 2] \* self.image\_matrix[i + 1, j + 1])

      # we call the normalize function to normalize the output

      self.normalize()

      return self.\_output, self.\_output\_norm

    # normalize

    def normalize(self):

      if self.mode == 'smoothing':

        # normalize using sum of all values

        self.\_output\_norm = self.\_output / 140

        self.\_output\_norm = np.pad(self.\_output\_norm, 3, mode='constant')

      elif self.mode == 'gradient':

        # normalize using sum of absolute values

        temp\_output = np.absolute(self.\_output)

        self.\_output\_norm = temp\_output / 3

        self.\_output\_norm = np.pad(self.\_output\_norm, 4, mode='constant')