CANNY'S EDGE DETECTOR

Shravani Rakshe
New York University
spr4123@nyu.edu

Niharika Krishnan
New York University
nk2982@nyu.edu

ABSTRACT:

The Canny edge detector is an edge detection operator that uses a multi-stage algorithm to detect a range of edges in images. The stages include:

1. Gaussian Smoothing
2. Horizontal Gradient, Vertical Gradient, Gradient Magnitude, Gradient Angle
3. Non-Maxima Suppression
4. Simple Thresholding

GITHUB LINK: https://github.com/niharikakrishnan/Canny-Edge-Detector

IMPLEMENTATION:

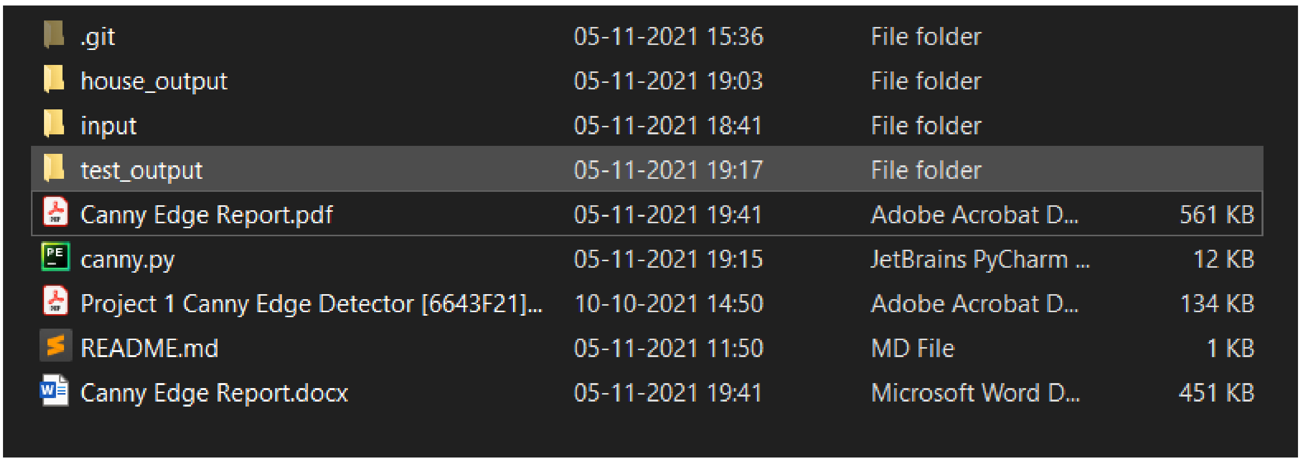
Assumes the pre-requisite environment with Python3 and necessary libraries (opencv & numpy) are already installed. If not, please install using:

```
pip install opencv-python
pip install numpy
```

Ensure the input image is in the same path as the source code file, ***canny.py***

STEPS:

1. Open terminal window and change directory to where the solution code, ***canny.py*** is located



2. Run the ***Command: python3 canny.py -i "[path_to_input_filename]"***

Note: Some machines have a different python path setup, in such cases, please use:

Command: python canny.py -i "[path_to_input_filename]"

Example: python3 canny.py -i "house.bmp"

```
D:\NYU\Computer Science\Computer Vision\Canny-Edge-Detector>python canny.py -i "input/house.bmp"
```

3. Canny Edge Detector Solution is implemented with print statements showing the completion of functions

```
D:\NYU\Computer Science\Computer Vision\Canny-Edge-Detector>python canny.py -i "input/house.bmp"
Running Canny Edge Detector!
Applying Gaussian Smoothing to input image
Convoluting image of shape (225, 225) with kernel size of (7, 7)
Computing Horizontal Gradient of the Smoothened Image.
Computing Horizontal Gradient.
Convoluting image of shape (225, 225) with kernel size of (3, 3)
Computing Vertical Gradient.
Convoluting image of shape (225, 225) with kernel size of (3, 3)
Computing Gradient Magnitude.
Computing Gradient Angle.
Computing Gradient Direction.
Applying Non Maxima Suppression
Applying Simple Thresholding
Canny Edge Detector Implemented and output images stored in the directory!
```

4. The output images will be stored in a directory named ***filename_output*** present at the same location

Example: house_output

.git	05-11-2021 15:36	File folder	
house_output	05-11-2021 19:03	File folder	
input	05-11-2021 18:41	File folder	
test_output	05-11-2021 15:36	File folder	
Canny Edge Report.docx	05-11-2021 19:02	Microsoft Word D...	374 KB
canny.py	05-11-2021 19:08	JetBrains PyCharm ...	12 KB
Project 1 Canny Edge Detector [6643F21]...	10-10-2021 14:50	Adobe Acrobat D...	134 KB
README.md	05-11-2021 11:50	MD File	1 KB

SOURCE CODE:

```
from math import degrees, pi
import numpy as np
import argparse
import cv2
import os

def convolution(image, mask):
'''
Function to perform convolution of an image with a mask using matrix multiplication. In cases
where the mask goes outside of the image border, it is considered as undefined and replaced
with zeroes. Region of interest surrounding every reference pixel is computed and multiplied
with the mask.
:param image: a grayscale image of size N X M
:param mask: a mask/kernel of size N X N
:return convoluted_image: image after convolution with size N X M, same as output image
'''
       #Getting the number of rows and columns of the image and the mask using .shape method
       image_row, image_col = image.shape
       mask_row, mask_col = mask.shape
       print("Convoluting image of shape {} with kernel size of {}".format(image.shape,
mask.shape))
```

```
        #Initialising the convoluted 2D array with zeros
        convoluted_image = np.zeros(image.shape)

        #Defining the number of rows and columns that will be undefined based on the mask
dimensions
        add_row = int(mask_row - 1) // 2
        add_col = int(mask_col - 1) // 2

        #Initialising a 2D array with zeros along with extra rows and columns to handle the
undefined values
        modified_image = np.zeros((image_row + (2 * add_row), image_col + (2 * add_col)))

        modified_image_row, modified_image_col = modified_image.shape

        #Defining the region of interest for the input image
        modified_image[add_row: modified_image_row - add_row, add_col:modified_image_col -
add_col] = image

        #Matrix multiplication - Convoluting image with kernel
        for row in range(1,image_row-1):
                for col in range(1,image_col-1):
                        #Using sliding window concept for maxtrix multiplication of kernel and
region of interest of input image
                        convoluted_image[row, col] = np.sum(mask * modified_image[row : row +
mask_row, col : col + mask_col])

        return convoluted_image

def gaussian_smoothing(image, mask):
'''
Function to perform Gaussian Smoothing with a 7 x 7 mask. Region of interest surrounding
every reference pixel is computed by multiplying with the mask.
:param image: a grayscale image of size N X M
:param mask: a mask/kernel of size 7 X 7
:return gaussian_image: image after smoothing
'''
        print("Applying Gaussian Smoothing to input image")

        #Getting the number of rows and columns of the image using .shape method
        image_row, image_col = image.shape

        #Applying Gaussian Smoothing to the input image
        gaussian_image = convolution(image, mask)

        #Normalising the gaussian smoothened image by using the sum of total pixels
        for row in range(image_row):
                for col in range(image_col):
                        if abs(gaussian_image[row, col]) > 255:
                                gaussian_image[row, col] = abs(gaussian_image[row, col]) / 140

        return gaussian_image

def gradient_operation(image, edge_filter):
'''
Function that uses Prewitt's operator to compute the horizontal gradient, vertical gradient.
This is followed by the computation of gradient magnitude using the square root of the sum of
squares of horizontal and vertical gradient.
Gradient angle is computed using the tan inverse of vertical and horizontal gradient.
:param image: a grayscale image after Gaussian smoothing
:param edge_filter: Prewitt's operator edge filter
```

```
:return horizontal_gradient: Normalized horizontal gradient
:return vertical_gradient: Normalized vertical gradient
:return gradient_magnitude: Normalized gradient magnitude image
:return gradient_direction: Gradient angle of the image
'''
       #Getting the number of rows and columns of the image using .shape method
       image_row, image_col = image.shape

       print("Computing Horizontal Gradient.")
       #Computing the horizontal gradient by convoluting the input image with prewitt's
horizontal edge filter
       horizontal_gradient = convolution(image, edge_filter)

       print("Computing Vertical Gradient.")
       #Transforming and flipping the horizontal gradient edge_filter to calculate vertical
gradient
       #Output: [[1,1,1], [0,0,0], [-1,-1,-1]]
       vertical_edge_filter = np.flip(edge_filter.T, axis=0)

       #Computing the vertical gradient by convoluting the input image with prewitt's vertical
edge filter
       vertical_gradient = convolution(image, vertical_edge_filter)

       print("Computing Gradient Magnitude.")
       #Using the formula, gradient magnitude = Square Root of Squares of Horizontal and
Vertical Gradient
       gradient_magnitude = np.sqrt(np.square(horizontal_gradient) +
np.square(vertical_gradient))

       #Normalising the Gradient Magnitude by dividing it with (3*root(2))
       for row in range(image_row):
              for col in range(image_col):
                     if abs(gradient_magnitude[row, col]) > 255:
                            gradient_magnitude[row, col] = abs(gradient_magnitude[row, col]) /
3*(np.sqrt(2))

       print("Computing Gradient Angle.")
       #Calculating gradient angle -> tan inverse (vertical gradient/horizontal gradient) in
radians
       gradient_angle = np.arctan2(vertical_gradient, horizontal_gradient)

       print("Computing Gradient Direction.")
       #Converting gradient angle from radians to degree which returns in the range of -180 to
180. Required to perform non-maxima suppression
       gradient_direction = np.rad2deg(gradient_angle)

       #Adding 180 degrees to the gradient angle for converting the range from 0 360.
       gradient_direction += 180

       return horizontal_gradient, vertical_gradient, gradient_magnitude, gradient_direction

def non_maxima_suppression(gradient_magnitude, gradient_direction):
'''
Function to scan along the image gradient direction, and if pixels are not part of the local
maxima they are set to zero. This has the effect of suppressing all image information that is
not part of local maxima.
:param gradient_magnitude: square root of sum of squares of horizontal and vertical gradient
:param gradient_direction: the matrix that has the gradient angle at each pixel location
:return nms_output: Normalized gradient magnitude image after non-maxima suppression
'''
```

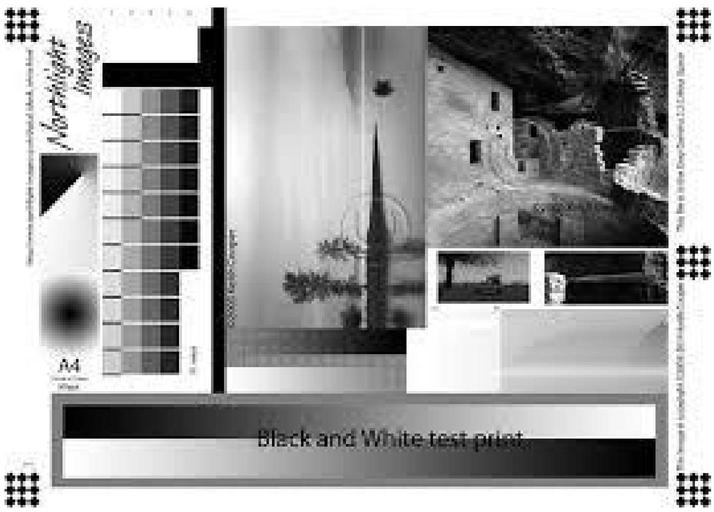
```
	print("Applying Non Maxima Suppression")

	gradient_row, gradient_col = gradient_magnitude.shape
	#Getting the number of rows and columns of the gradient magnitude using .shape method
	nms_output = np.zeros(gradient_magnitude.shape)

	#Applying non maxima suppression to all pixels other than the border
	for row in range(1, gradient_row-1):
		for col in range(1, gradient_col-1):
			angle = gradient_direction[row, col]

			#Mapping to Sector 0, hence compare with left and right pixels
			if (0 <= angle < 22.5) or (157.5 <= angle < 202.5) or (337.5 <= angle <=
360):
				before_pixel = gradient_magnitude[row, col - 1]
				after_pixel = gradient_magnitude[row, col + 1]

			#Mapping to Sector 1, hence compare with upper right and lower left pixels
			elif (22.5 <= angle < 67.5) or (202.5 <= angle < 247.5):
				before_pixel = gradient_magnitude[row + 1, col - 1]
				after_pixel = gradient_magnitude[row - 1, col + 1]

			#Mapping to Sector 2, hence compare with upper and lower pixels
			elif (67.5 <= angle < 112.5) or (247.5 <= angle < 292.5):
				before_pixel = gradient_magnitude[row - 1, col]
				after_pixel = gradient_magnitude[row + 1, col]

			#Mapping to Sector 3, hence compare with upper left and lower right pixels
			else:
				before_pixel = gradient_magnitude[row - 1, col - 1]
				after_pixel = gradient_magnitude[row + 1, col + 1]

			#If the centre pixel is strictly greater than the neighbouring pixels, we
use the gradient magnitude value, else zero
			if ((gradient_magnitude[row, col] > before_pixel) and
(gradient_magnitude[row, col] > after_pixel)):
				nms_output[row, col] = gradient_magnitude[row, col]

	return nms_output

def simple_thresholding(image):
'''
Function to compute binary edge maps using simple thresholding for thresholds chosen at
the 25th, 50th and 75th percentiles
:param image: the image after non maxima suppression
:return tresholding_output: Binary edge maps using simple thresholding for thresholds chosen
at the 25th, 50th and 75th percentiles
'''
	print("Applying Simple Thresholding")

	#Getting the number of rows and columns of the image using .shape method
	image_row, image_col = image.shape

	#Using the percentile function to calculate the threshold value at 25th, 50th and 75th
percentile
	threshold_25 = np.percentile(list(set(image.flatten())), 25)
	threshold_50 = np.percentile(list(set(image.flatten())), 50)
	threshold_75 = np.percentile(list(set(image.flatten())), 75)
	threshold = {"threshold_25": threshold_25, "threshold_50": threshold_50,
"threshold_75": threshold_75}
```

```
    thresholding_output = {}

    #Applying simple thresholding with threshold values at 25, 50 and 75th percentile
    #If pixel intensity is strictly greater than the threshold value, we assign it with a
value of 255 (white)
    for threshold_key, threshold_value in threshold.items():

        #Initializing output image array with zeroes
        output = np.zeros(image.shape)
        for row in range(image_row):
            for col in range(image_col):
                if image[row, col] > threshold_value:
                    output[row, col] = 255
        thresholding_output[threshold_key] = output

    return thresholding_output

if __name__ == '__main__':
    print("Running Canny Edge Detector!")
    ap = argparse.ArgumentParser()
    ap.add_argument("-i", "--image", required=True, help="Path to the image")
    args = vars(ap.parse_args())

    #Reading and opening input image and converting to grayscale
    frame = cv2.imread(args['image'])
    image = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    #Creating path to write output images of Canny Edge Detector
    folder, fname_with_extension = os.path.split(args['image'])
    fname , extension = os.path.splitext(fname_with_extension)
    path = str(fname) + "_output"
    access = 0o755

    #Checking whether the specified output path exists or not
    isExist = os.path.exists(path)

    #Creating a new directory if it does not already exist
    if not isExist:
        os.makedirs(path,access)
        print("Output directory created.")

    #Defining 7 x 7 Gaussian mask as mentioned in the Project requirement
    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]], dtype='int')

    #Applying Gaussian smoothing to input image with given mask
    gaussian_image = gaussian_smoothing(image, mask)
    cv2.imwrite(path + "/"+str(fname)+"_GaussianSmoothing.bmp", gaussian_image)

    #Defining Prewitt's Edge Operator
    edge_filter = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]], dtype='int')

    #Applying gradient operation to compute horizontal gradient, vertical gradient,
```

```
gradient magnitude and gradient angle
    horizontal_gradient, vertical_gradient, gradient_magnitude, gradient_direction = 
gradient_operation(image, edge_filter)
    cv2.imwrite(path + "/"+str(fname)+"_HorizontalGradient.bmp", horizontal_gradient)
    cv2.imwrite(path + "/"+str(fname)+"_VerticalGradient.bmp", vertical_gradient)
    cv2.imwrite(path + "/"+str(fname)+"_GradientMagnitude.bmp", gradient_magnitude)

    #Applying Non Maxima Suppression to the gradient magnitude
    nonmaxima_image = non_maxima_suppression(gradient_magnitude, gradient_direction)
    cv2.imwrite(path + "/"+str(fname)+"_NonMaximaSuppression.bmp", nonmaxima_image)

    #Applying Simple thresholding with thresholds chosen at 25th, 50th and 75th percentile
    threshold_image = simple_thresholding(nonmaxima_image)
    for threshold_name, threshold_value in threshold_image.items():
        cv2.imwrite(path + "/"+str(fname)+ "_"+threshold_name+".bmp", threshold_value)

    print("Canny Edge Detector implemented and output images stored in the directory!")
```

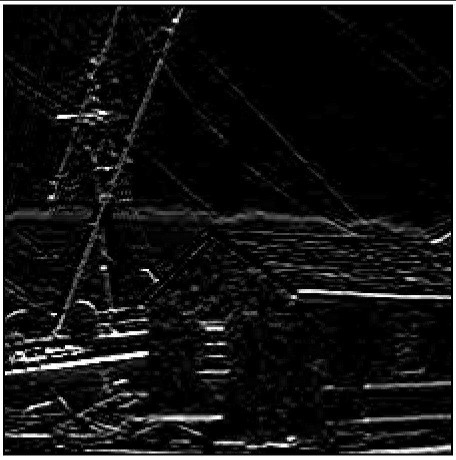
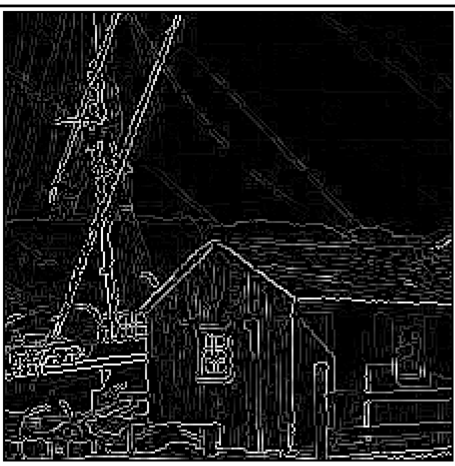
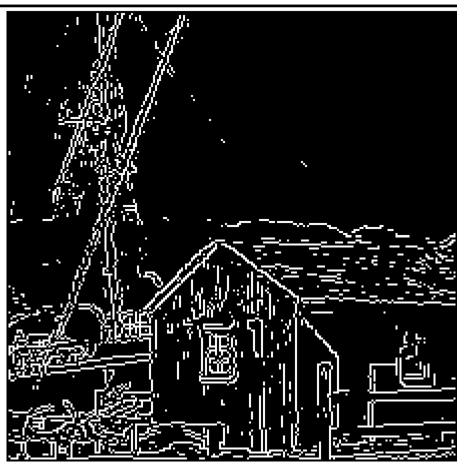
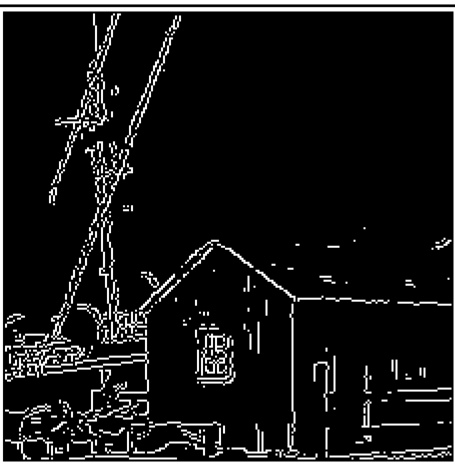
INPUT IMAGES:

house.bmp

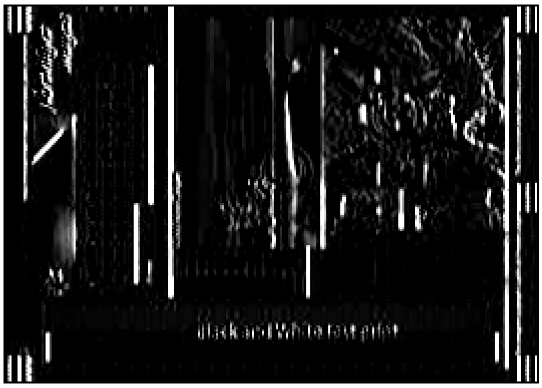
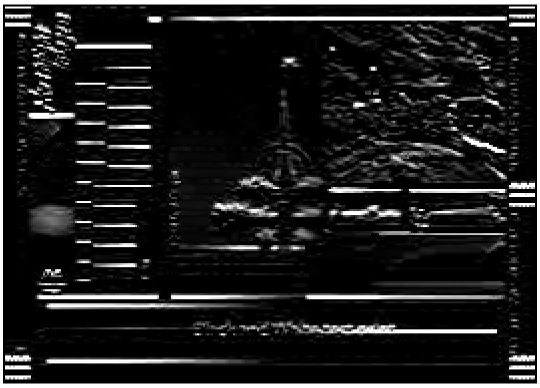
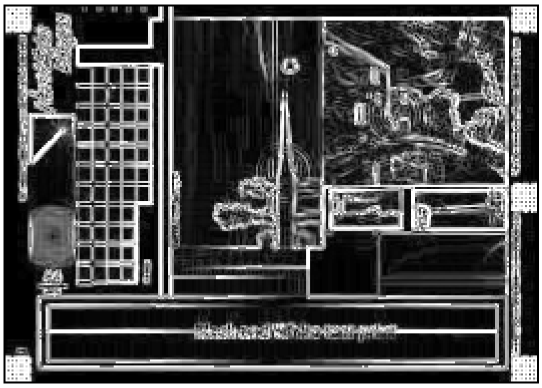
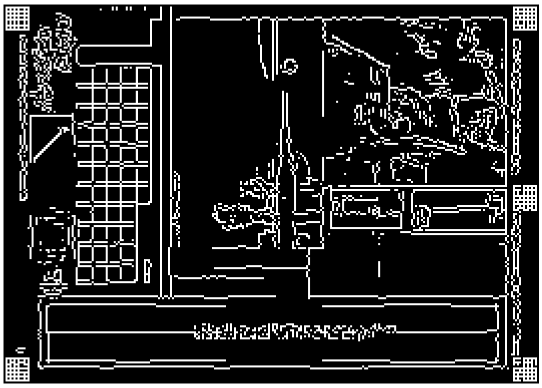


test_patterns.bmp

OUTPUT IMAGE:

GAUSSIAN SMOOTHING	HORIZONTAL GRADIENT
	
VERTICAL GRADIENT	**GRADIENT MAGNITUDE**
	
NON MAXIMA SUPPRESSION	**THRESHOLDING AT 25^{TH} PERCENTILE**
	
THRESHOLDING AT 50^{TH} PERCENTILE	**THRESHOLDING AT 75^{TH} PERCENTILE**
	

OUTPUT IMAGES:

GAUSSIAN SMOOTHING	HORIZONTAL GRADIENT
	
VERTICAL GRADIENT	**GRADIENT MAGNITUDE**
	
NON MAXIMA SUPPRESSION	**THRESHOLDING AT 25TH PERCENTILE**
	
THRESHOLDING AT 50TH PERCENTILE	**THRESHOLDING AT 75TH PERCENTILE**