# Assignment 2 – Edge Detection

- Ashrith Chandramouli (aschandr@syr.edu)

**Brief instructions**

The results for this assignment are generated from the python script – edge_detection.py

The script does not require any parameters to run and can be run using the command

" python edge_detection.py" from the command prompt in the directory of the script. The script is developed on Python 3.7.2 and I have included the following libraries which are necessary to run the script.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage,signal
from math import sqrt,pi,degrees,atan
import cv2
```

The results of this script are generated on a window. The results may take an average of 10-15 seconds to process and appear for each test image. Closing the window will then initiate processing for the next test image.

All the output images have been included in the zip file for reference. The result image may appear to be losing some data but this is due to the size of the image displayed. Zooming in on the images displayed after running the script will display it clearly. For the purposes of simpler demonstration, I have not included certain test cases which experiment with the threshold and standard deviation values. Uncommenting green lines below will run the cases

```
if __name__ == "__main__":

    img = cv2.imread("Flowers.jpg",0)
    canny_edge_detector(img,1,10,20)
    img1 = cv2.imread("Syracuse_01.jpg",0)
    canny_edge_detector(img1,1,10,20)
    img2 = cv2.imread("Syracuse_02.jpg",0)
    # canny_edge_detector(img2,1,5,20)
    # canny_edge_detector(img2,2,5,20)
    # canny_edge_detector(img2,3,5,20)
    # canny_edge_detector(img2,1,20,40)
    canny_edge_detector(img2,1,10,20)
    img3 = cv2.imread("seattle.jpg",0)
    canny_edge_detector(img3,1,10,20)
```

The following pages of the report will explain my process and observations.

**Task 1) Implement the CANNY EDGE detection algorithm**

**CANNY ENHANCER**

The canny enhancer algorithm has 4 parts in of itself

Noise removal by smoothing – I've made use of Gaussian smoothing with a sigma value of 3

Gradient values – The gradient matrices on x and y axes represented by jx and jy respectively. I've used a 1d kernel for convolution as shown by kernel_jx and kernel_jy

Edge strength / intensity image – The strength value determined by the gradient values are captured in the es matrix

Edge orientation image – The direction of the edge normal in degrees is stored in the eo matrix

```
#Performs image smoothing and creates gradient, strength and orientation grids
#Takes in the image and sigma value for smoothing as parameters and returns ed
ge strength and orientation images
def canny_enhancer(img, sigma):
    #Gaussian smoothing
    smooth_img = gaussian_filter(img,sigma)
    kernel_jx = [[-1,0,1]]
    kernel_jy = [[-1],[0],[1]]
    #Gradients on x and y co-ordinates
    jx = signal.convolve2d(smooth_img,kernel_jx,boundary='fill',mode='same')
    jy = signal.convolve2d(smooth_img,kernel_jy,boundary='fill',mode='same')

    #Edge strength
    es = np.hypot(jx, jy)
    #Edge orientation
    eo = np.degrees(np.arctan2(jy, jx))

    return np.array(es,dtype=np.int32),np.array(eo,dtype=np.int32)
```

**NONMAX SUPPRESSION**

The nonmax suppression algorithm reduces the size of an edge to 1 pixel. This process involves the edge orientation matrix extensively. The thinning of edges is performed as follows. The neighbours of a pixel are determined by discretizing the edge orientation values for that pixel. If the edge strength of the pixel is less than its neighbours then the intensity value of the pixel is set to 0. If it is greater than both the neighbours, we retain the original value of the pixel. This process helps us better visualize the resulting image.

The corresponding code is as shown below

```
#Performs thinning of edges to 1 pixel
#Takes in the strength and orientation matrices and returns the strength matri
x with 1 pixel edges
def nonmax_suppression(strength,orientation):
    intensity = np.copy(strength)
    for i in range(1,len(intensity)-1):
        for j in range(1,len(intensity[0])-1):
            directions = positions(orientation[i][j])
            x1 = directions[0][0]
            y1 = directions[0][1]
            x2 = directions[1][0]
            y2 = directions[1][1]
            neighbor1 = strength[i+x1][j+y1]
            neighbor2 = strength[i+x2][j+y2]
            if strength[i][j] < neighbor1 or strength[i][j] < neighbor2:
                intensity[i][j] = 0
            else:
                intensity[i][j] = strength[i][j]

    return intensity
```

```
#Obtains the neigboring matrix element position based on the edge normal value
#Takes in the edge normal value in degrees and outputs the neigboring pixel po
sitions
def positions(num):
    if 0 <= num < 22.5 or 157.5 <= num <= 180:
        return ((0,-1),(0,1))
    elif 22.5 <= num < 67.5:
        return ((1,-1),(-1,1))
    elif 67.5 <= num < 112.5:
        return ((1,0),(-1,0))
    else:
        return ((1,1),(-1,-1))
```

The function positions as shown above performs the discretization using the edge orientation which is necessary to find the neighbours for a pixel.

x1,y1 and x2,y2 are the locations of the neighbours along the direction of the edge normal.

**HYSTERESIS THRESHOLD**

This process involves chaining the weak and strong pixels on an edge together after thinning so we can identify the edge clearly. For performing the chaining, I've made use of a Depth First Search process which recursively chains neighbouring pixels if they're above the threshold level. A high and low threshold levels have been used. Any pixel value above the high threshold is considered a strong pixel, any pixel between the low and high values is considered weak and anything below the low threshold is discarded. For every pixel having a value equal to or higher than the high threshold, we perform chaining to link the other pixels to form an edge. The direction we follow while chaining is perpendicular to the edge normal of the pixel. It is implemented in the code as shown below

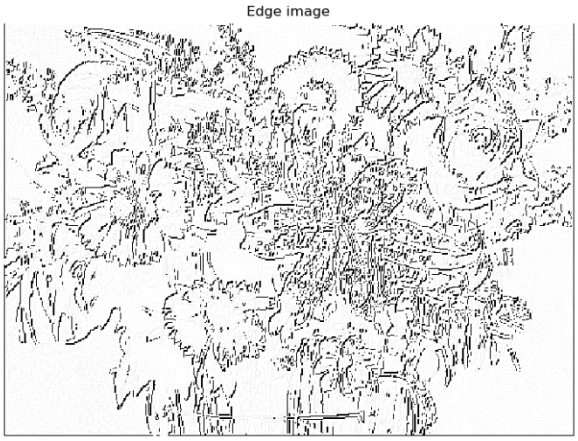
```
#Performs hysteresis thresh
#Takes in the image, orientation matrix, low and high threshold values as para
meters
def hysteresis_threshold(img,ori,low = 5, high = 20):
    for i in range(len(img)):
        for j in range(len(img[0])):
            visited = set()
            if img[i][j] >= high:
                img[i][j] = 255
                chaining(img,i,j,ori,low,visited)
```

```
#Performs edge tracking
#Takes in the image, starting location of pixel, orientation matrix, low thres
hold and a visited set for dfs
#Performs edge linking traversing in the direction parallel to edge normal
def chaining(img,i,j,ori,low,visited):
    visited.add((i,j))
    #90 is added to get the perpendicular value
    directions = positions((ori[i][j]+90)%180)
    x1 = directions[0][0]
    y1 = directions[1][1]
    x2 = directions[1][0]
    y2 = directions[1][1]
    while (x1,y1) not in visited and x1 >= 0 and x1 < len(img) and y1 >=0 and
y1 < len(img[0]) and img[x1][y1] >= low:
        img[x1][y1] = 255
        chaining(img,x1,y1,ori,low,visited)
    while (x2,y2) not in visited and x2 >= 0 and x2 < len(img) and y2 >=0 and
y2 < len(img[0]) and img[x2][y2] >= low:
        img[x2][y2] = 255
        chaining(img,x2,y2,ori,low,visited)
```

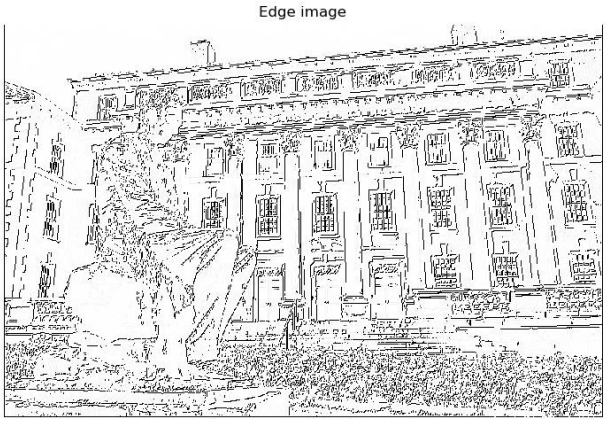
x1,y1 and x2,y2 in the code above show the neighbouring pixels in the direction perpendicular to the edge normal.

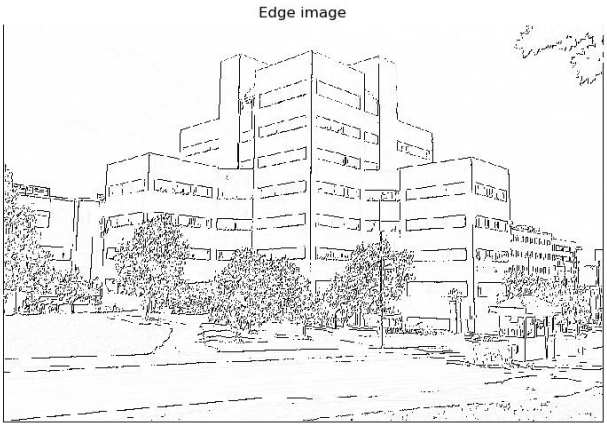
**Task 2) Test algorithm on images**

Below are the results of running the canny edge detection processes on the images given






A better view of the results can be found in the result of the script. Zooming into the image will show better results as we lose edge details as the size of the image in the window decreases.

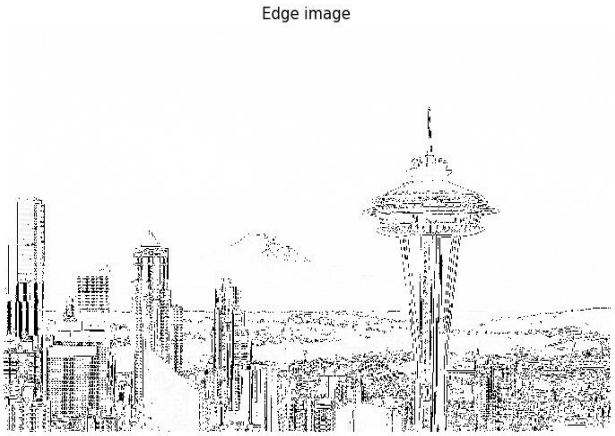
**Observations with varied values for sigma and thresholds**

By experimenting with sigma values 1,2 and 3 and with low, high thresholds (20,40), (10,20) and (5,10), I came up with the following

- As we increase the value of sigma, we lose details on the edges and the edge becomes lighter. This is because the intensity/strength values from gradients decrease due to the blurring effect.
- As we increase the threshold values, we lose a lot of the pixels which actually constitute an edge and hence the edge gets lighter or is completely gone. This is because the image may contain weak edges which get filtered by the high threshold values. As we decrease the threshold values, the pixels which do not constitute an actual edge appear and the image gets cluttered by these pixels.

Uncommenting the tests on the main function will run these tests on Syracuse_02.jpg

Task 3) Test algorithm on a favourite image


I’ve used a picture of the Seattle skyline here