**­­­CSC 481**

**Assignment 4**

**Hithesh Shanmugam**

**Edge Detection (10)**

Use the edge function to generate results for Roberts, Canny, Sobel, and Prewitt operators on an image of your choice. Note also that the various edge functions support a number of parameters – feel free to explore those to get more interesting results. State which operator gives the best performance and why you think so. Notice also how this problem is different than the one using filters that I asked you to do for Assignment 3.

import matplotlib.pyplot as plt

import numpy as np

import math

import cv2

import os

import PIL

from PIL import Image

import matplotlib.image as mpimg

from imgaug import augmenters as iaa

import skimage

from skimage import filters, feature

#Load and display image

path\_of\_img = r'C:\Users\sures\OneDrive - DePaul University\Desktop\firework.jpg'

image = cv2.imread(path\_of\_img)

#Convert to rgb

img\_process = cv2.cvtColor(image , cv2.COLOR\_BGR2RGB)

plt.imshow(img\_process)

# #Hide axes

plt.xticks([]), plt.yticks([])

plt.show()

**Output:**



#Grayscale image

gray\_image = cv2.imread(path\_of\_img, 0)

rgb2gray(img\_process)

#Roberts

roberts\_img = filters.roberts(gray\_image)

#Canny

canny\_img1 = feature.canny(gray\_image)

canny\_img2 = feature.canny(gray\_image, sigma = 3)

#Sobel

sobel\_img = filters.sobel(gray\_image)

#Prewitt

prewitt\_img = filters.prewitt(gray\_image)

#Plotting blurred images

rows = 2

cols = 2

axes = []

fig = plt.figure()

images = [roberts\_img, canny\_img1, sobel\_img, prewitt\_img]

edges = ['Roberts', 'Canny', 'Sobel', 'Prewitt']

for i in range(rows \* cols):

img\_ = images[i]

fig.set\_size\_inches(20, 20)

axes.append(fig.add\_subplot(rows, cols, i + 1))

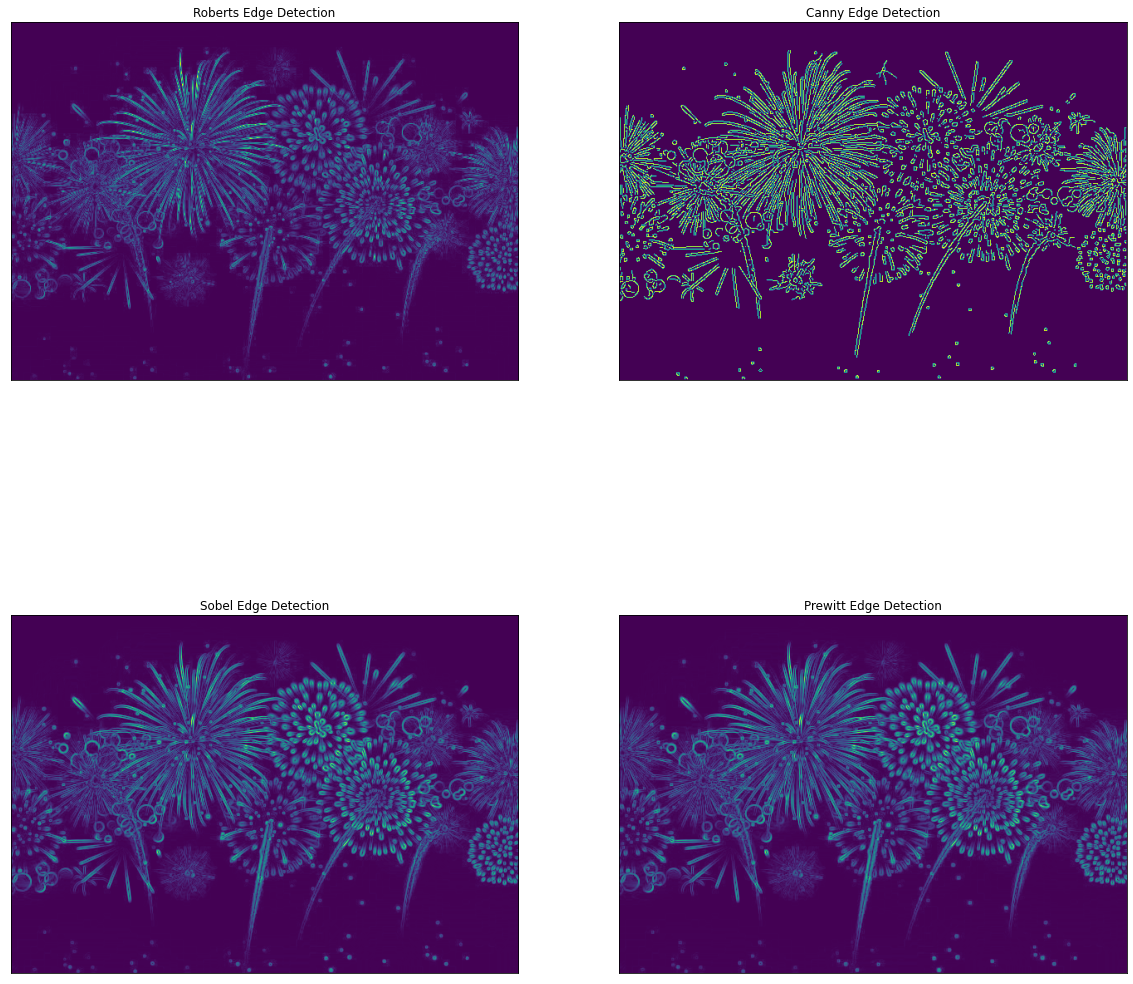
subplot\_title = ("{0} Edge Detection".format(edges[i]))

axes[-1].set\_title(subplot\_title)

plt.xticks([]), plt.yticks([])

plt.imshow(img\_)

**Output:**

****

By comparing with the original image, I feel like the Sobel was better comparing with other filters because the edges where the lights in the bottom left the edge in canny was a lot of mess where it has some edges inside it too but others didn’t have that and also with Sobel edge detection some edges are brighter than in the Prewitt and Robert’s.

**Edge Filter (10)**

Design a 7x7 “Sobel” operator and filter your image from the “Edge Detection” task above with your filter. The main idea behind the design of a proper Sobel-ish operator is to model the Gaussian derivate in one direction and the Gaussian in the perpendicular direction:

http://campar.in.tum.de/twiki/pub/Chair/HaukeHeibelGaussianDerivatives/_MathModePlugin_3da151a11410133ca6af39a948c3651c.png

In this formulation, σ becomes a parameter of the filter; you can choose whatever you like, although choosing 1 makes the math much simpler. *x* and *y* are the distance from the center pixel of your filter. Aside from the size of your filter, how does it differ from the standard Sobel operator? How are the image results different than what you saw applying the standard Sobel operator?

#Create 7x7

x = cv2.getDerivKernels(1, 0, 7)

y = cv2.getDerivKernels(0, 1, 7)

#Filter image

sobelx\_7x7 = cv2.Sobel(gray\_image, -1, 1, 0, 7)

sobely\_7x7 = cv2.Sobel(gray\_image, -1, 0, 1, 7)

#Combined 7x7

combined\_sobel = sobelx\_7x7 + sobely\_7x7

#Plotting sobel operators

rows = 2

cols = 2

axes = []

fig = plt.figure()

image\_types = [sobel\_img, sobelx\_7x7, sobely\_7x7, combined\_sobel]

edges = ['Original Sobel', '7x7 Sobel First Derivative - X', '7x7 Sobel First Derivative - Y', 'Combined 7x7']

for i in range(rows \* cols):

img = image\_types[i]

fig.set\_size\_inches(20, 20)

axes.append(fig.add\_subplot(rows, cols, i + 1))

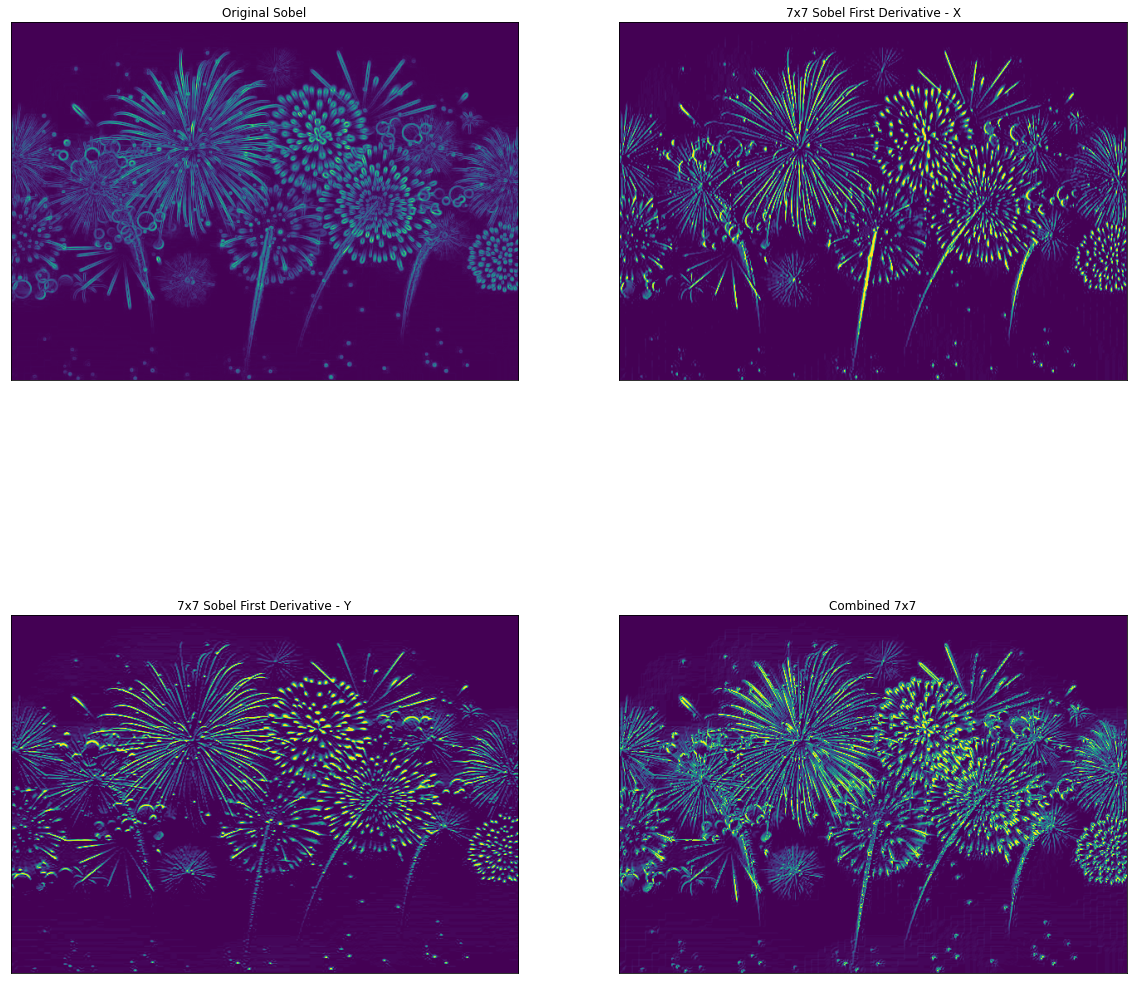
subplot\_title = ("{0}".format(edges[i]))

axes[-1].set\_title(subplot\_title)

plt.xticks([]), plt.yticks([])

plt.imshow(img)

**Output:**

****

Here in the 7 x 7 filter the combined filter detects both the horizontal and vertical edges and comparing with the original Sobel the edges are brighter and easy to detect.

**Histogram-based segmentation (20)**

Implement histogram based segmentation on your image as follows:

1. Show your image.

#Show image

plt.xticks([]), plt.yticks([])

plt.imshow(img\_process)

**Output:**

****

1. Display the histogram and identify the peaks of your histogram with the “objects” that they correspond to.

#Pixel frequencies

frequencies = cv2.calcHist([img\_process], [0], None, [256], [0,256])

#CDF

cdf = frequencies.cumsum()

cdf\_norm = cdf \* frequencies.max()/cdf.max()

#Plot histogram

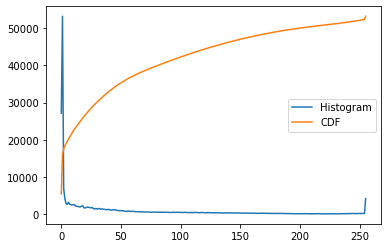
plt.plot(frequencies, label = 'Histogram')

plt.plot(cdf\_norm, label = 'CDF')

plt.legend()

plt.show()

**Output:**

****

In this image the brighter pixels are in the first and after that we have a stable line. The brightest pixel would be the brightest firework in the sky which it has the white pixels in it.

1. Specify the ranges that you will use to identify the binary objects.

#Create thresholds for 25 on the low end and 229 on the high end

ret, low\_threshold = cv2.threshold(image, 25, 255, cv2.THRESH\_BINARY)

ret, high\_threshold = cv2.threshold(image, 229, 255, cv2.THRESH\_BINARY)

1. Show the identified objects as binary images for each range. (Remember to scale the images for display so that objects can be seen.)

# Plot images

plt.clf()

fig = plt.figure()

fig.set\_size\_inches(20, 20)

ax1 = fig.add\_subplot(1,2,1)

ax1.set\_title('Low Threshold')

ax1.imshow(low\_threshold)

plt.xticks([]), plt.yticks([])

ax2 = fig.add\_subplot(1,2,2)

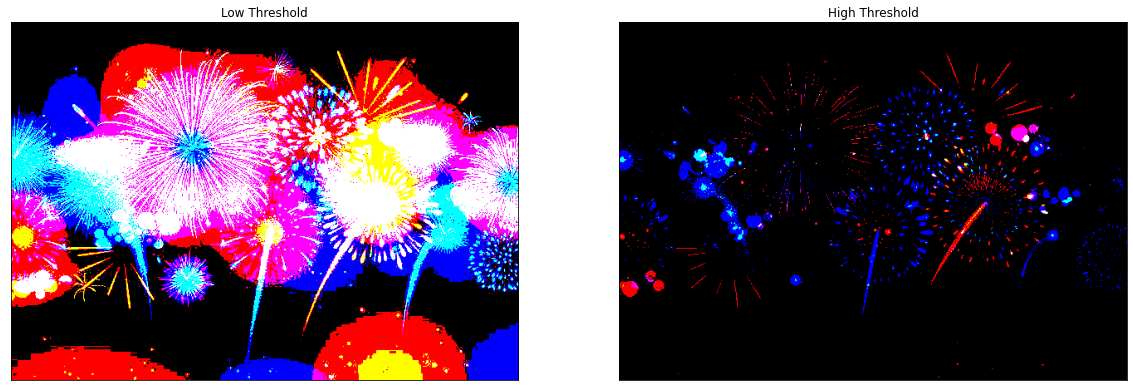
ax2.set\_title('High Threshold')

ax2.imshow(high\_threshold)

plt.xticks([]), plt.yticks([])

plt.show()

**Output:**

****

1. Finally construct the histogram-based segmented image, by combining the binary images.

#Thresh histograms

histogram\_low = cv2.calcHist([low\_threshold], [0], None, [256], [0,256])

histogram\_high = cv2.calcHist([high\_threshold], [0], None, [256], [0,256])

#Thresh CDFs

cdf\_low = histogram\_low.cumsum()

normalized\_cdf\_low = cdf \* histogram\_low.max()/cdf.max()

cdf\_high = histogram\_high.cumsum()

normalized\_cdf\_high = cdf \* histogram\_high.max()/cdf.max()

#Plot new histograms

plt.clf()

fig = plt.figure()

fig.set\_size\_inches(15, 5)

ax1 = fig.add\_subplot(1,2,1)

ax1.set\_title('Low Threshold')

ax1.plot(cdf\_low, label = 'Histogram')

ax1.plot(normalized\_cdf\_low, label = 'CDF')

ax1.legend()

ax2 = fig.add\_subplot(1,2,2)

ax2.set\_title('High Threshold')

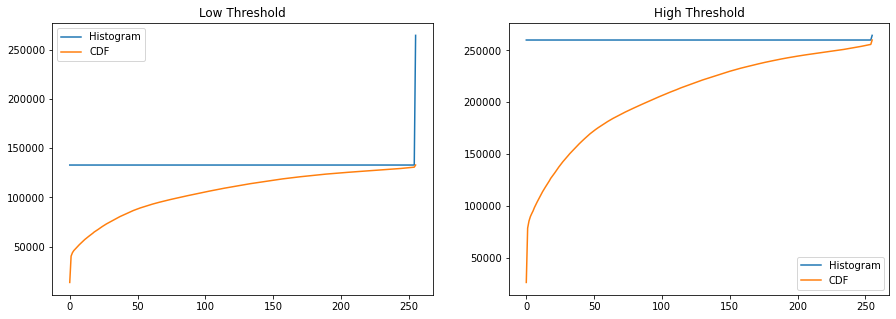
ax2.plot(cdf\_high, label = 'Histogram')

ax2.plot(normalized\_cdf\_high, label = 'CDF')

ax2.legend()

plt.show()

**Output:**

****

**481 Students: Noise reduction** ***(*20/0)**

1. On an image of your choice, use the imnoise function to generate two noise corrupted images as follows: (the article at <https://stackoverflow.com/questions/14435632/impulse-gaussian-and-salt-and-pepper-noise-with-opencv> show one way to do this with Python and openCV)

#Read image with pil

img\_pil = Image.open(path\_of\_img)

img\_arr = np.asarray(img\_pil)

#Gaussian noise

gauss\_aug = iaa.AdditiveGaussianNoise(loc = 0, scale = 0.1 \* 255)

gauss\_img = gauss\_aug.augment\_image(img\_arr)

#Salt and pepper

salt\_and\_pepper = iaa.SaltAndPepper(p = 0.05)

salt\_and\_pepper\_img = salt\_and\_pepper.augment\_image(img\_arr)

1. Use subplot to display the original image and the two noise corrupted images.

#Plotting noise plots

rows = 2

cols = 2

axes = []

fig = plt.figure()

images = [img\_process, gauss\_img, salt\_and\_pepper\_img]

edges = ['Original Image', 'Gaussian Noise', 'Salt and Pepper Noise']

for i in range((rows \* cols) - 1):

img\_ = images[i]

fig.set\_size\_inches(20, 20)

axes.append(fig.add\_subplot(rows, cols, i + 1))

subplot\_title = ("{0}".format(edges[i]))

axes[-1].set\_title(subplot\_title)

plt.xticks([]), plt.yticks([])

plt.imshow(img\_)

**Output:**

****

1. Use the function fspecial to design averaging filters of size (3x3), (5,5), and (7x7). Use subplot to display the noise\_saltAndpepper image and the three averaged filtered results. Do the same for the noise\_gaussian image.

*#Filters*

filter\_sizes **=** [3, 5, 7]

*#Gaussian*

**for** filter\_ **in** filter\_sizes:

gauss\_avg\_img **=** cv2**.**blur(gauss\_img, (filter\_, filter\_))

gauss\_med\_img **=** cv2**.**medianBlur(gauss\_img, filter\_)

*#Salt and pepper*

**for** filter\_ **in** filter\_sizes:

s\_p\_avg\_img **=** cv2**.**blur(salt\_and\_pepper\_img, (filter\_, filter\_))

s\_p\_med\_img **=** cv2**.**medianBlur(img2, filter\_)

*#Gaussian plots*

rows **=** 2

cols **=** 2

axes **=** []

fig **=** plt**.**figure()

images **=** [gauss\_img, gauss\_avg\_img, gauss\_med\_img]

edges **=** ['Gaussian Noise Image', 'Average Gaussian Noise Image', 'Median Gaussian Noise Image']

**for** i **in** range((rows **\*** cols) **-** 1):

img\_ **=** images[i]

fig**.**set\_size\_inches(20, 20)

axes**.**append(fig**.**add\_subplot(rows, cols, i **+** 1))

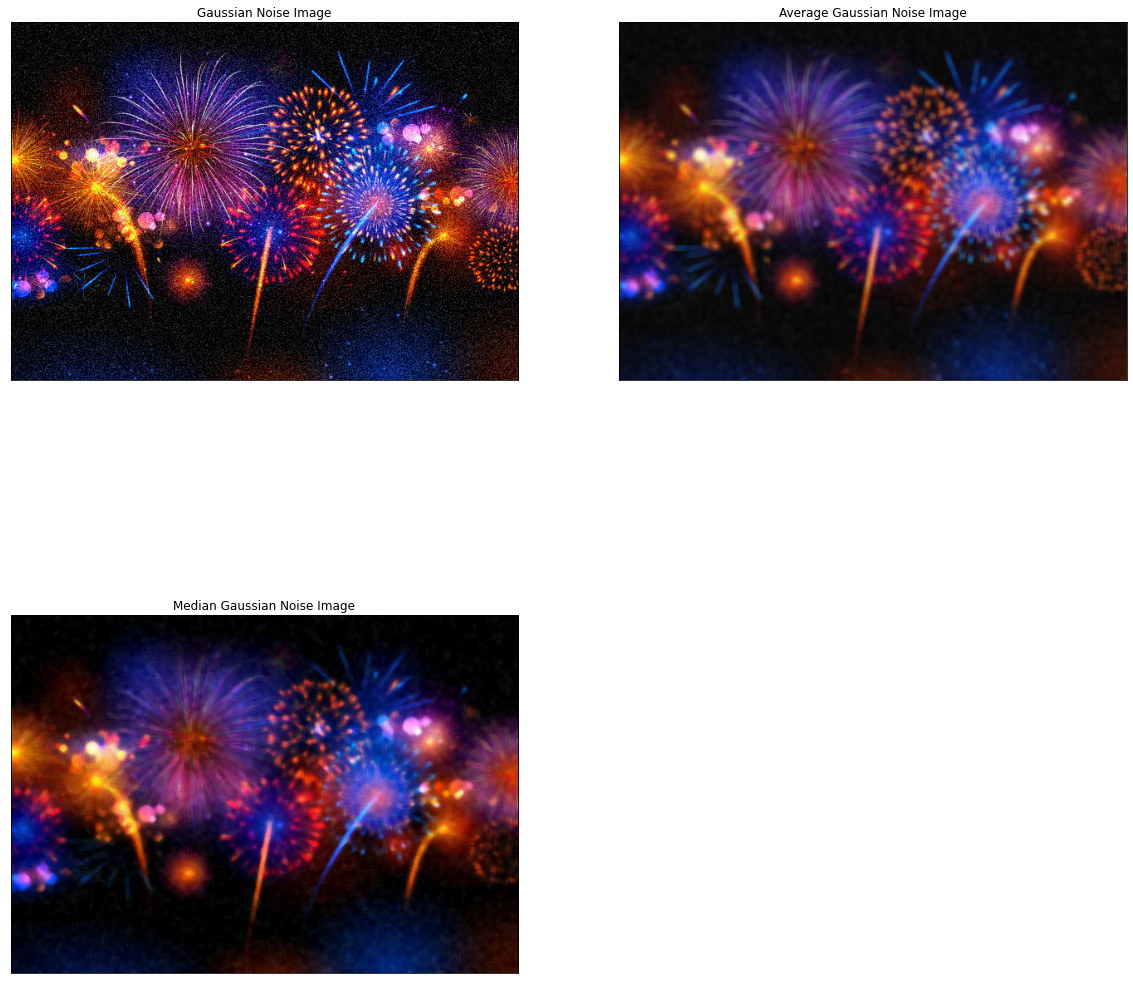
subplot\_title **=** ("{0}"**.**format(edges[i]))

axes[**-**1]**.**set\_title(subplot\_title)

plt**.**xticks([]), plt**.**yticks([])

plt**.**imshow(img\_)

**Output:**

****

1. Use the medfilt2 function to perform median filtering on the noise\_saltAndpepper image. Design the median filters to work with window sizes of (3x3), (5x5), and (7x7). Use your filters on the noise\_gaussian image also and display as in part c).

*#Salt and pepper plots*

rows **=** 2

cols **=** 2

axes **=** []

fig **=** plt**.**figure()

images **=** [salt\_and\_pepper\_img, s\_p\_avg\_img, s\_p\_med\_img]

edges **=** ['Salt and Pepper Noise Image', 'Average Salt and Pepper Noise Image', 'Median Salt and Pepper Noise Image']

**for** i **in** range((rows **\*** cols) **-** 1):

img\_ **=** images[i]

fig**.**set\_size\_inches(20, 20)

axes**.**append(fig**.**add\_subplot(rows, cols, i **+** 1))

subplot\_title **=** ("{0}"**.**format(edges[i]))

axes[**-**1]**.**set\_title(subplot\_title)

plt**.**xticks([]), plt**.**yticks([])

plt**.**imshow(img\_)

**Output:**

