**Assignment 4 (60/40 points)**

**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.

#Importing the Packages

#Running them

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

#Loading and Reading the image

Firework = r'C:/Users/admin/Desktop/fireworks.jfif'

#Reading the specified path for the image

img\_fire = cv2.imread(Firework)

#Converting BGR to RGB

img\_conv = cv2.cvtColor(img\_fire , cv2.COLOR\_BGR2RGB)

#Plotting the image

plt.imshow(img\_conv)



#Grayscale image

gry\_firework = cv2.imread(Firework, 0)

#Robert's Edge Filter

rob\_img = filters.roberts(gry\_firework)

#Canny's Edge Filter

can\_img = feature.canny(gry\_firework)

can\_img = feature.canny(gry\_firework, sigma = 3)

#Sobel's Edge Filter

sobel\_img = filters.sobel(gry\_firework)

#Prewitt's Edge Filter

prewitt\_img = filters.prewitt(gry\_firework)

#The blurred images are plotted

#Rows

#Columns

rows = 2 ; cols = 2

axes = []

fig = plt.figure()

images = [rob\_img, can\_img, sobel\_img, prewitt\_img]

no\_of\_edges = ['Roberts', 'Canny', 'Sobel', 'Prewitt']

for i in range(rows \* cols):

img\_ = images[i]

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

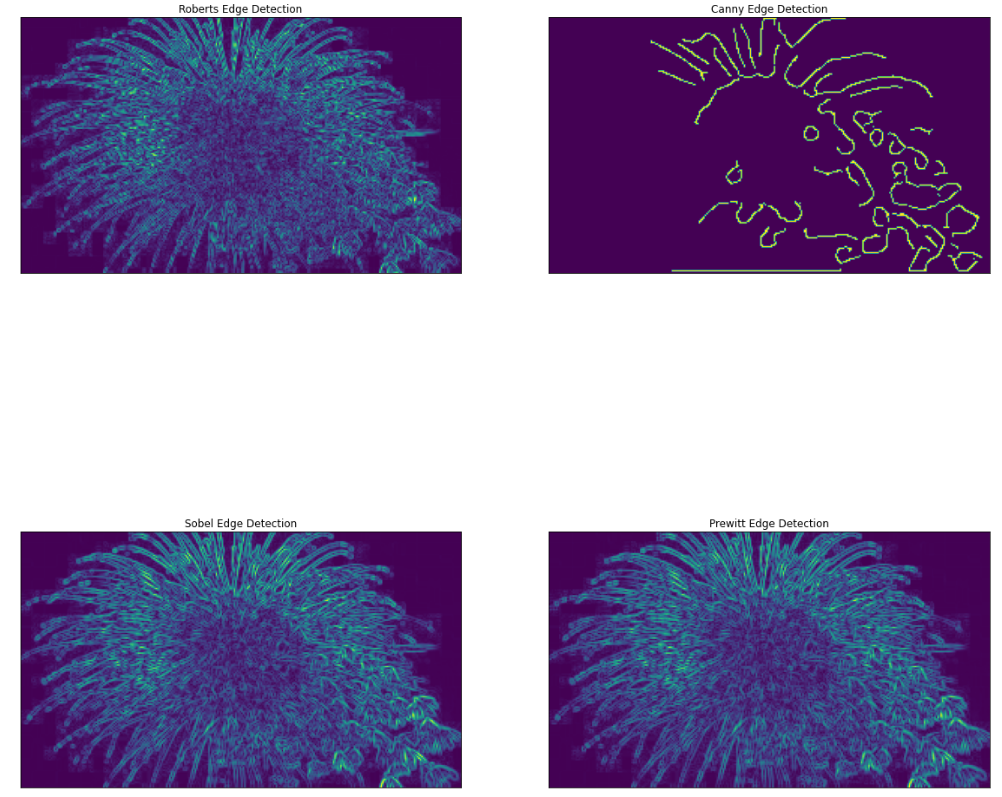
axes.append(fig.add\_subplot(rows, cols, i + 1))

subplot\_title = ("{0} Edge Detection".format(no\_of\_edges[i]))

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

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

plt.imshow(img\_)



In comparison with the original image, the sobel is doing well in comparison of other two filters. The Canny Edge Filter did pretty badly as we can see the edges being faded off and missing a lot of information in the image. Also, Sobel Edge detection has some edges that 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?

#7x7 is created and plotted

x1 = cv2.getDerivKernels(1, 0, 7) ; y1= cv2.getDerivKernels(0, 1, 7)

#Filtered Image

sobel\_x\_7x7 = cv2.Sobel(gry\_firework, -1, 1, 0, 7) ; sobel\_y\_7x7 = cv2.Sobel(gry\_firework, -1, 0, 1, 7)

#7x7 is combined for X and Y Sobel

comb\_sobel = sobel\_x\_7x7 + sobel\_y\_7x7

#7x7 Sobel Operators are plotted

#Rows

#Columns

rows = 2 ; cols = 2

axes = []

fig = plt.figure()

image\_types = [sobel\_img, sobel\_x\_7x7, sobel\_y\_7x7, comb\_sobel]

no\_of\_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)

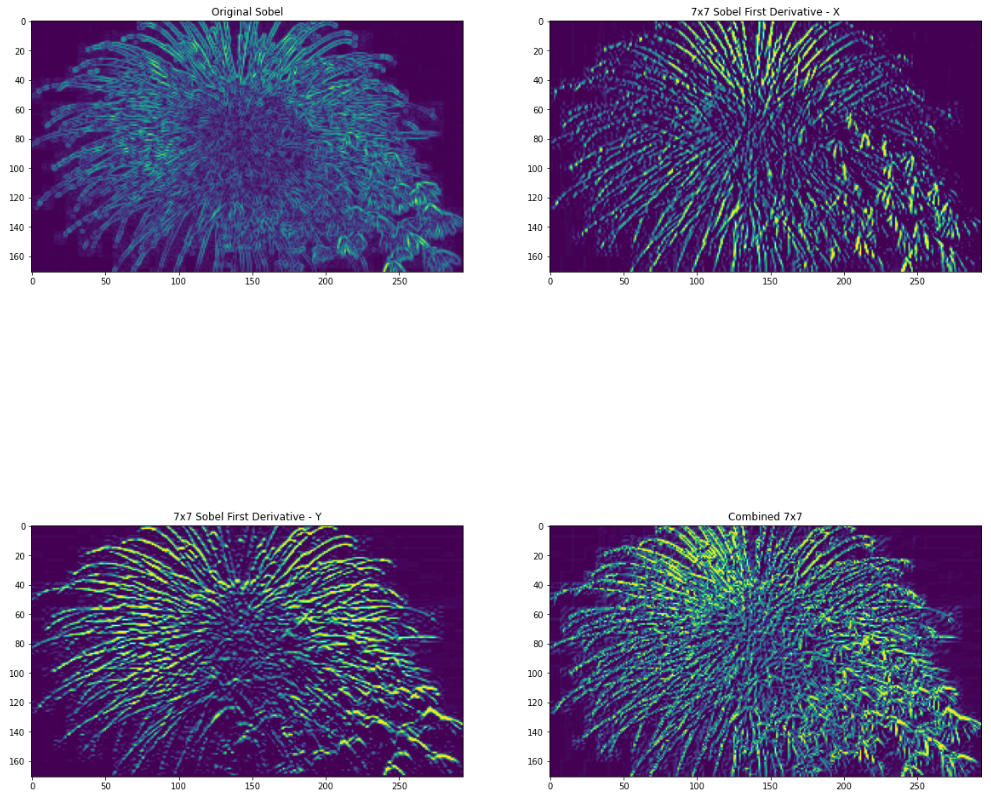
axes.append(fig.add\_subplot(rows, cols, i + 1))

subplot\_title = ("{0}".format(no\_of\_edges[i]))

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

#Plotting the image

plt.imshow(img)



The 7x7 filter here, the combined filter detects both the horizontal and vertical edges and by comparison the original Sobel edges are brighter and easier to detect.

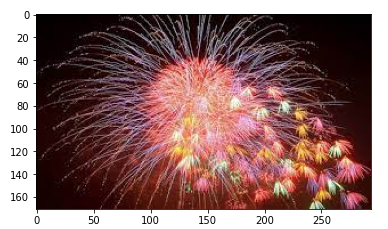
**Histogram-based segmentation (20)**

Implement histogram based segmentation on your image as follows:

1. Show your image.

#Show image

plt.imshow(img\_conv)



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

#Pixel frequencies

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

cdf = frequencies\_img.cumsum()

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

#The histogram is plotted

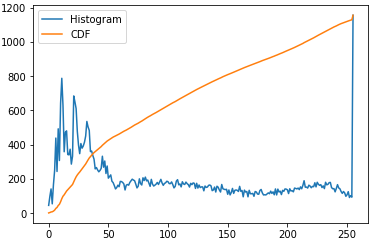
plt.plot(frequencies\_img, label = 'Histogram')

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

plt.legend()

#Image is plotted

plt.show()



From this plot, the brighter pixels are in the first and after that we have a stable line.

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

#Creating the threshold for 25 and 229 for lower and upper end

ret, low\_thresh = cv2.threshold(img\_fire, 25, 255, cv2.THRESH\_BINARY)

ret, high\_thresh = cv2.threshold(img\_fire, 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.)

#The images are plotted

plt.clf()

fig = plt.figure()

#20, 20

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

pt1 = fig.add\_subplot(1,2,1)

pt1.set\_title('Low Threshold')

pt1.imshow(low\_thresh)

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

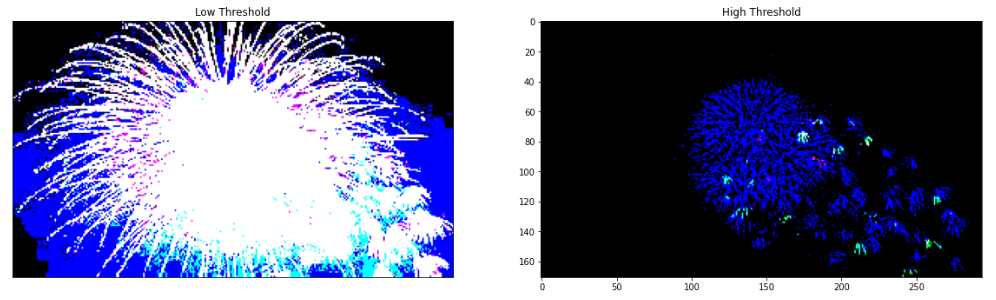
pt2 = fig.add\_subplot(1,2,2)

pt2.set\_title('High Threshold')

pt2.imshow(high\_thresh)

#The image is plotted

plt.show()



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

#Histograms Thresholding is done

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

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

#CDF

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()

#The new historgrams are plotted

plt.clf()

fig = plt.figure()

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

pt1 = fig.add\_subplot(1,2,1)

pt1.set\_title('Low Threshold')

pt1.plot(cdf\_low, label = 'Histogram')

pt1.plot(normalized\_cdf\_low, label = 'CDF')

pt1.legend()

pt2 = fig.add\_subplot(1,2,2)

pt2.set\_title('High Threshold')

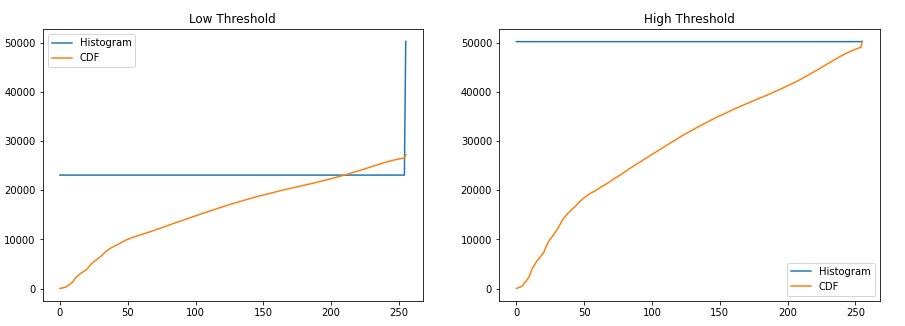
pt2.plot(cdf\_high, label = 'Histogram')

pt2.plot(normalized\_cdf\_high, label = 'CDF')

pt2.legend()

#The image is plotted

plt.show()



**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)

#Using the pil we are reading the image

Firework\_pil = Image.open(Firework)

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

#Applying the gaussian noise on the image

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

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

#Applying the Salt and pepper on the image

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

salt\_and\_pepper\_Firework = salt\_and\_pepper.augment\_image(Firework\_arr)

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

#Plotting noise plots

#Rows and Cols

rows = 2 ; cols= 2

axes = []

pic=plt.figure()

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

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

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

Tow\_= images[i]

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

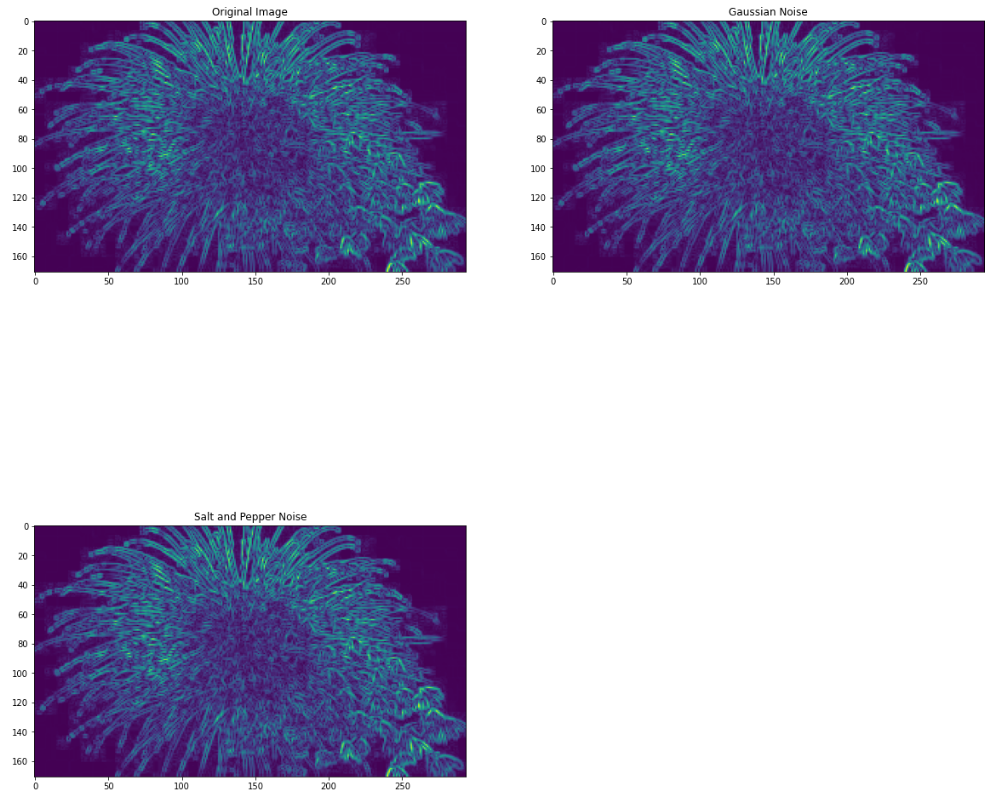
axes.append(pic.add\_subplot(rows, cols, i + 1))

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

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

#Plotting the image

plt.imshow(img\_)



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.

#Creating the Filters

filter\_sizes = [3, 5, 7]

#Applying the Gaussian

for filter\_ in filter\_sizes:

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

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

#Applying the Salt and pepper

for filter\_ in filter\_sizes:

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

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

#Applying the Gaussian plots

#Rows

#Columns

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)

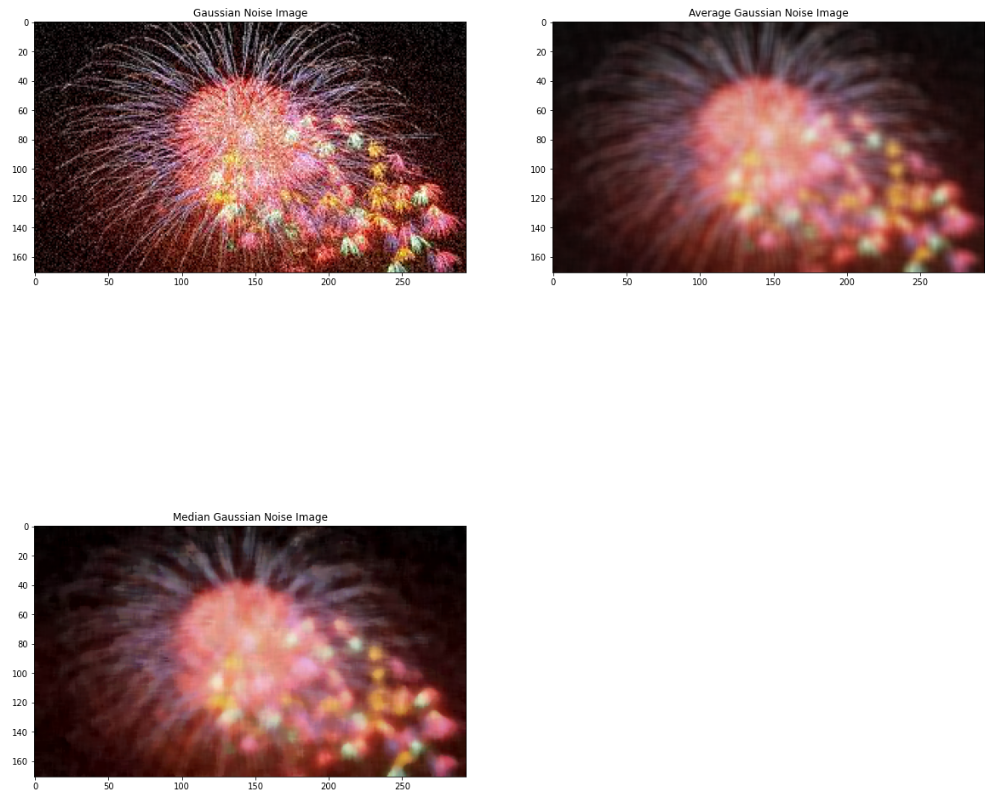
axes.append(fig.add\_subplot(rows, cols, i + 1))

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

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

#Plotting Image

plt.imshow(img\_)



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).

#Applying the Salt and pepper plots

#Rows

#Columns

rows = 2 ; cols = 2

axes = []

fig = plt.figure()

images = [salt\_and\_pepper\_Firework, 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\_)

