Programming Assignment 1

Part 1

Functions

In my EE classes, I've used convolutions many times, so this is just based on something I've already built. I originally did the convolutions math in a more explicite way $outPic[i,j] = (myfilter * img[i:m+i, j:n+j]).sum()$, where i'm just performing element-wise multiplication, and then summing each point. I instead chose to run the cleaner, $outPic[i,j] = myfilter.flatten() @ img[i:m+i, j:n+j].flatten()$, way of "flattening"/vectorizing the sections of the image and then performing a dot product, which gives the same solution. The '@' is a shorthand for matrix multiplication, which on two vectors will do a dot product. I also just used the $np.pad$ function to zero pad, because it's not a main part of the algorithm.

- myConv: is a convolution function
  - takes in image, filter, pad size
  - zero pad the image
  - do:
    - section a 3×3 piece of the image
    - vectorize both filter and image section
    - dot product
    - move over by one pixel and repeat
- sobelFilt: takes in x and y DoG components
  - peforms $\sqrt{g_x^2 + g_y^2}$

```
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
import cv2
```

```
def myConv(imgOG, myfilter,padd=1):
    imgHeight,imgWidth = imgOG.shape
    m,n =  myfilter.shape
    img = np.pad(imgOG, ((padd,padd), (padd,padd)), mode='constant').copy()
```

```
    outPic = np.zeros((imgHeight,imgWidth))
    for i in range(imgHeight):
        for j in range(imgWidth):
            outPic[i,j] = myfilter.flatten() @ img[i:m+i,j:n+j].flatten()
            #outPic[i,j] = (myfilter * img[i:m+i,j:n+j]).sum()
    return np.abs(outPic)

def sobelFilt(Xgx,Xgy):
    return ((Xgx)**2 +(Xgy)**2)**(1/2)
```

Filters

All my filters declared. Two Gaussian Blurs and the 2 DoG components.

```
gausBlur1 = np.array([
    [1,2,1],
    [2,4,2],
    [1,2,1]])/16
gausBlur2 = np.array([
    [1,4,7,4,1],
    [4,16,26,16,4],
    [7,26,41,26,7],
    [4,16,26,16,4],
    [1,4,7,4,1]])/273

gx = np.array([
    [1,0,-1],
    [2,0,-2],
    [1,0,-1]])

gy = np.array([
    [1,2,1],
    [0,0,0],
    [-1,-2,-1]])
```

image 1

Just Loads the images and displays it.

```
img = cv2.imread('filter1_img.jpg',0)
plt.figure(figsize=(10,8))
plt.title("Original: Camera Man")
plt.imshow(img,cmap = "gray")
```

```
plt.imshow(img,cmap = "gray")
plt.show()
```

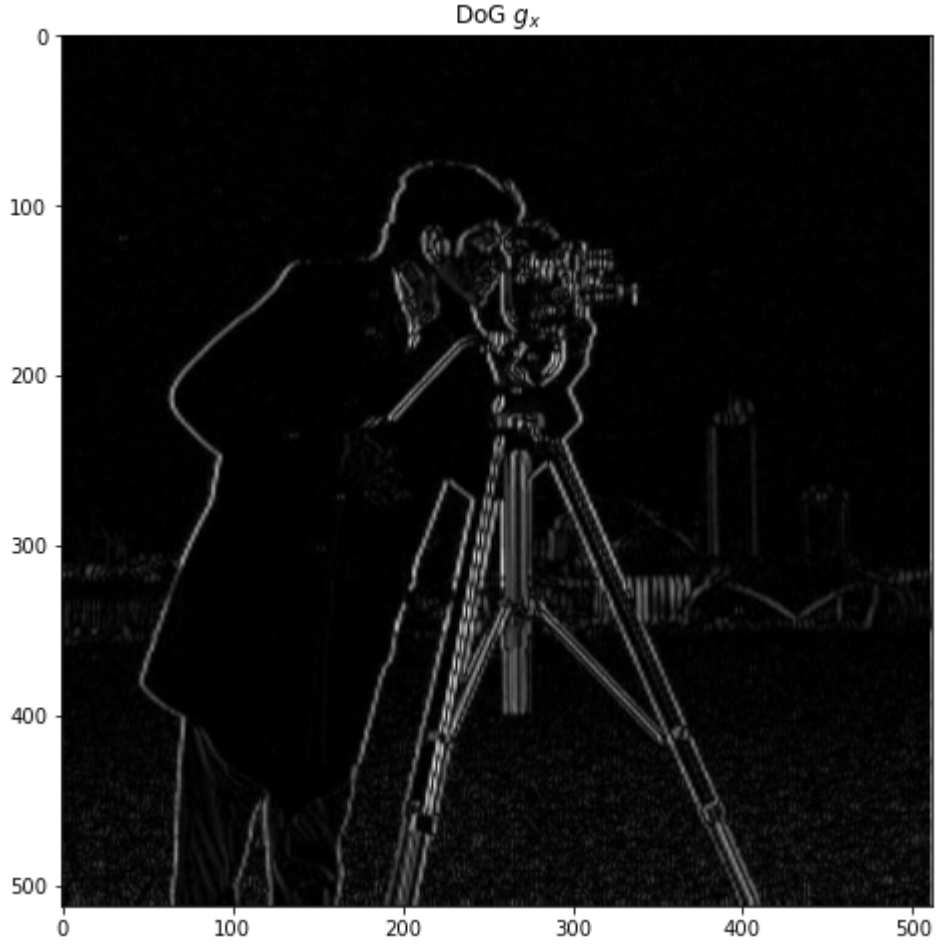
Apply filters

I apply all my filters. To save space, I just turned it into a list and called the name of the filter, the pad size and the title of the image. To apply the filter I simply convolve the filter with the image and display the new image.

```
filters = [[gausBlur1,1,"Gaussian $3 x 3$"],[gausBlur2,2,"Gaussian $5 x 5$"],[gx,1,"DoG $g_x$

for i in filters:
    oout = myConv(img,i[0],i[1])
    plt.figure(figsize=(10,8))
    plt.title(i[2])
    plt.imshow(oout,cmap = "gray")
    plt.show()
```


Gaussian 3x3
0
100
200
300
400
500
0
100
200
300
400
500


Gaussian 5x5
0
100
200
300
400
500
0
100
200
300
400
500


DoG g_x
0
100
200
300
400
500
0
100
200
300
400
500


DoG g_y
0
100
200
300
400

Sobel

The Sobel filter is just a quick preprossing step, where I add a gaussian blur, then I call my Sobel filter function and input the outputs of my applied DoG components. The exact same process is done for the next image.

```
# preprocessing before appling sobel filter
sob_pp = myConv(img,gausBlur2,2)

#appling sobel filter
oout = sobelFilt(myConv(sob_pp,gx,1),myConv(sob_pp,gy,1))
plt.figure(figsize=(10,8))
plt.title("Sobel")
plt.imshow(oout,cmap = "gray")
plt.show()
```

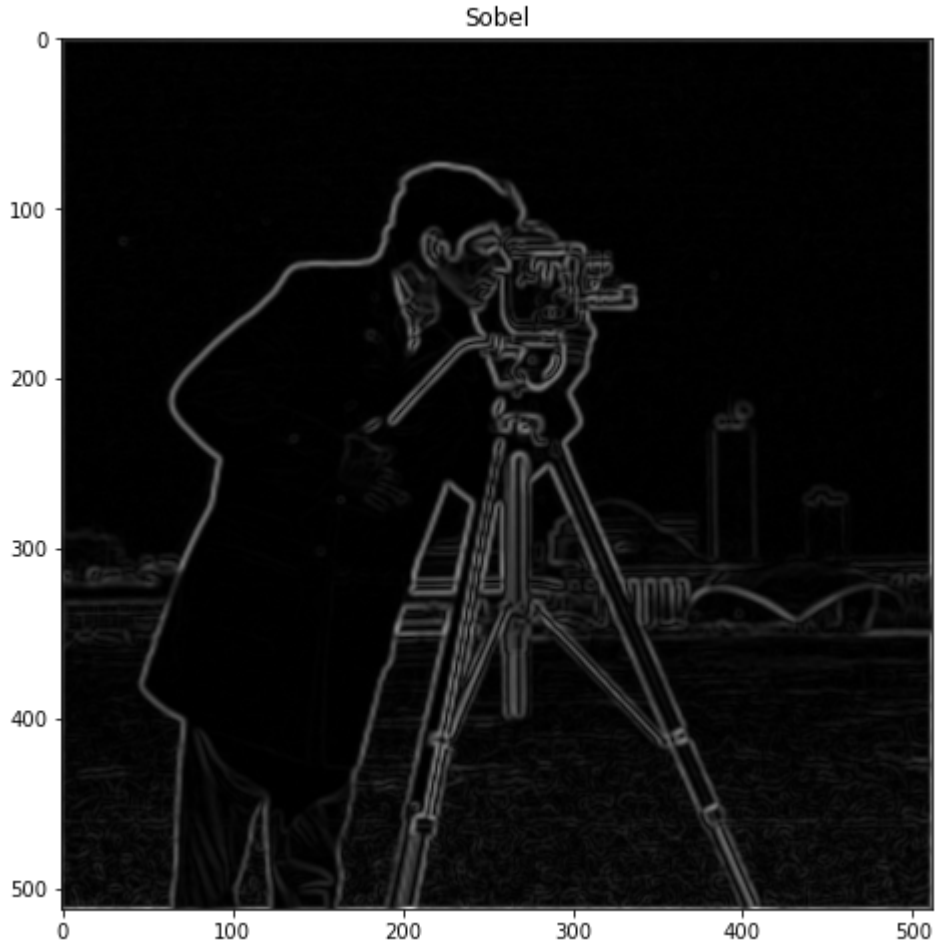
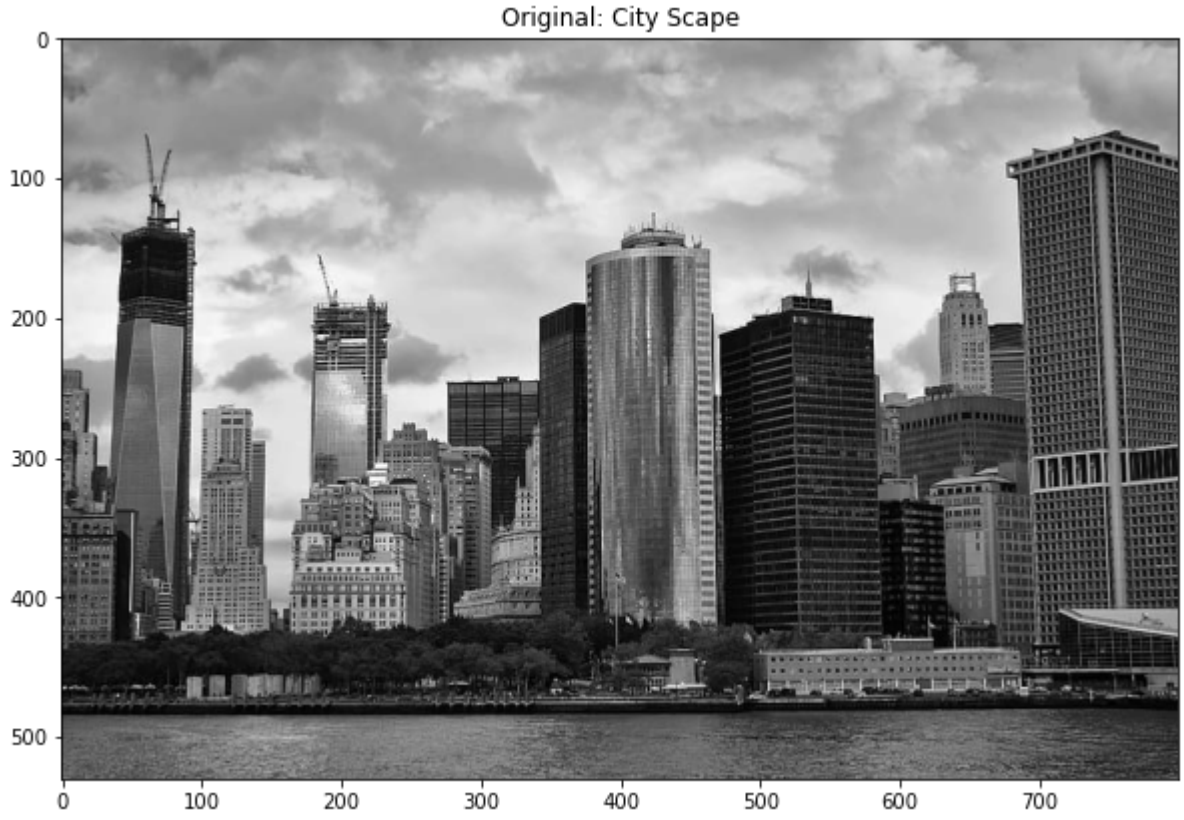


image 2

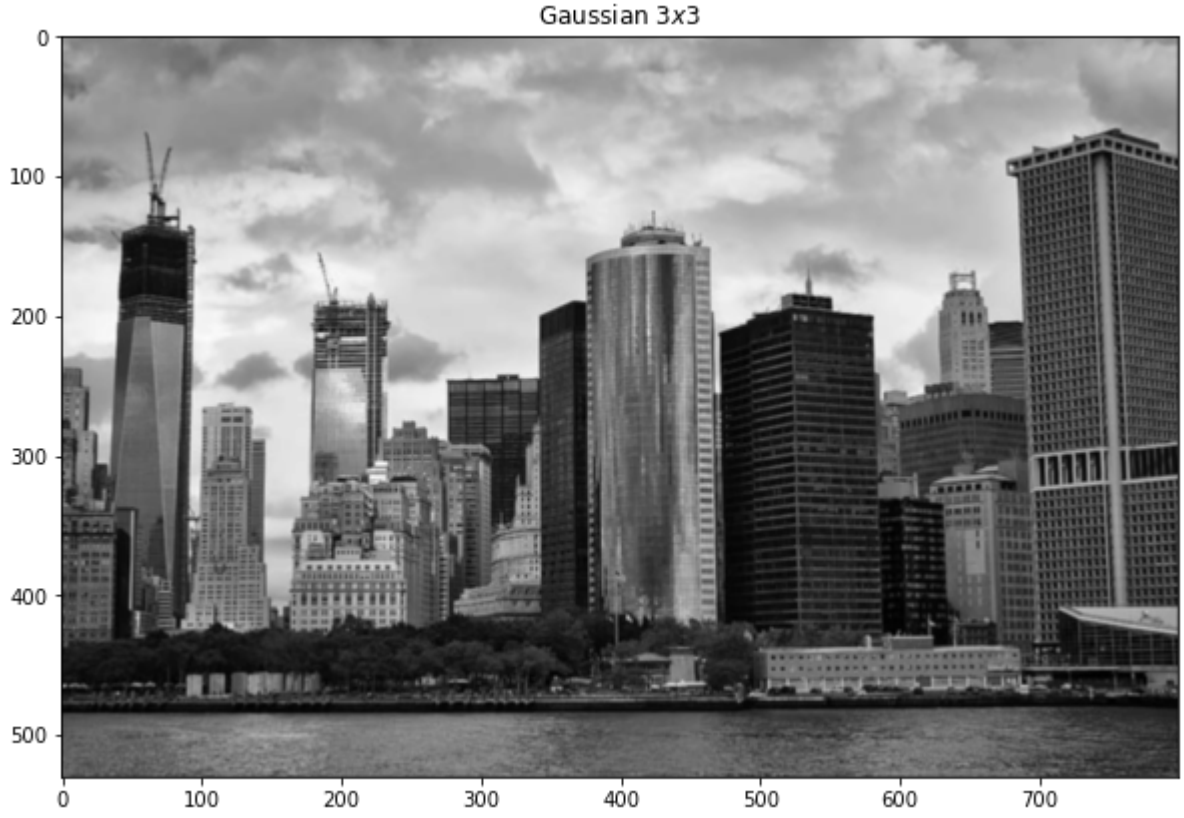
```
img = cv2.imread('filter2_img.jpg',0)
plt.figure(figsize=(10,8))
plt.title("Original: City Scape")
plt.imshow(img,cmap = "gray")
```

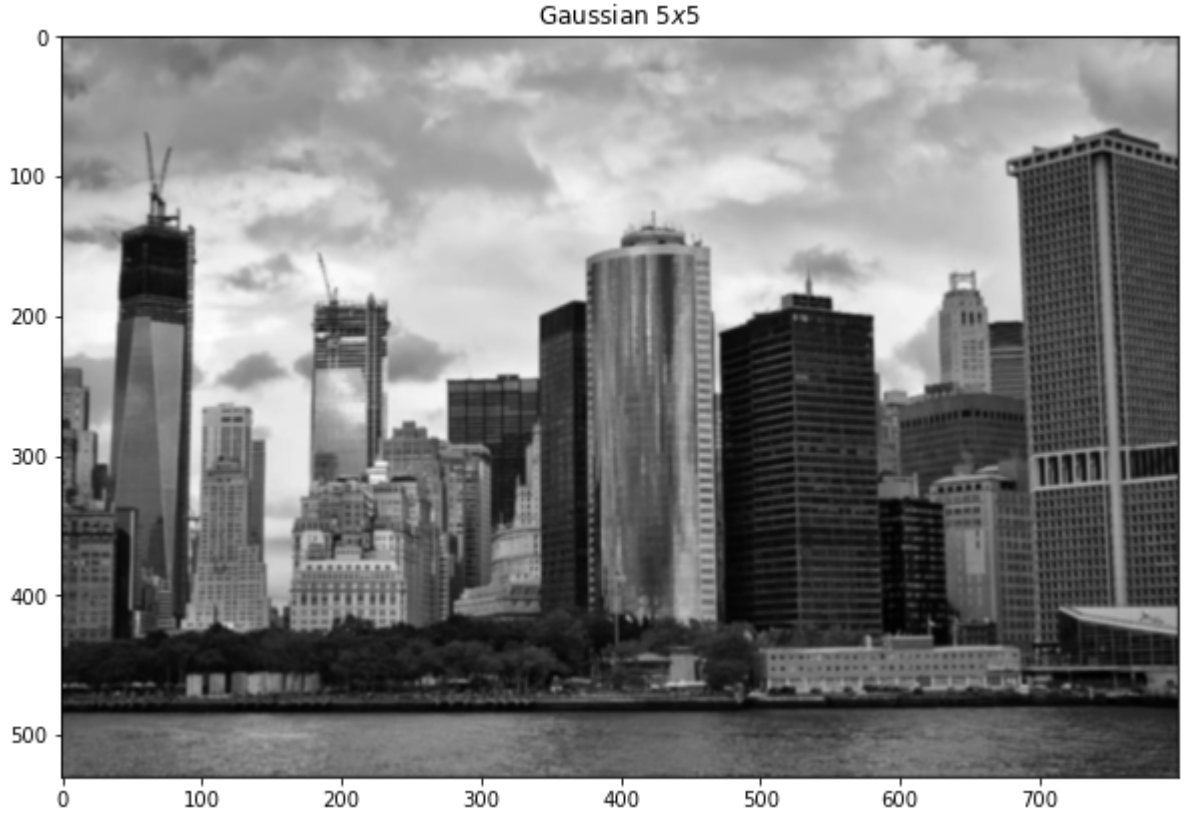
```
plt.show()
```

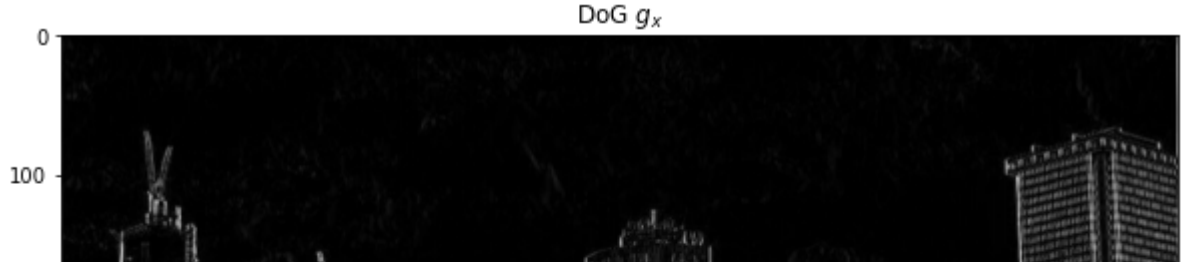


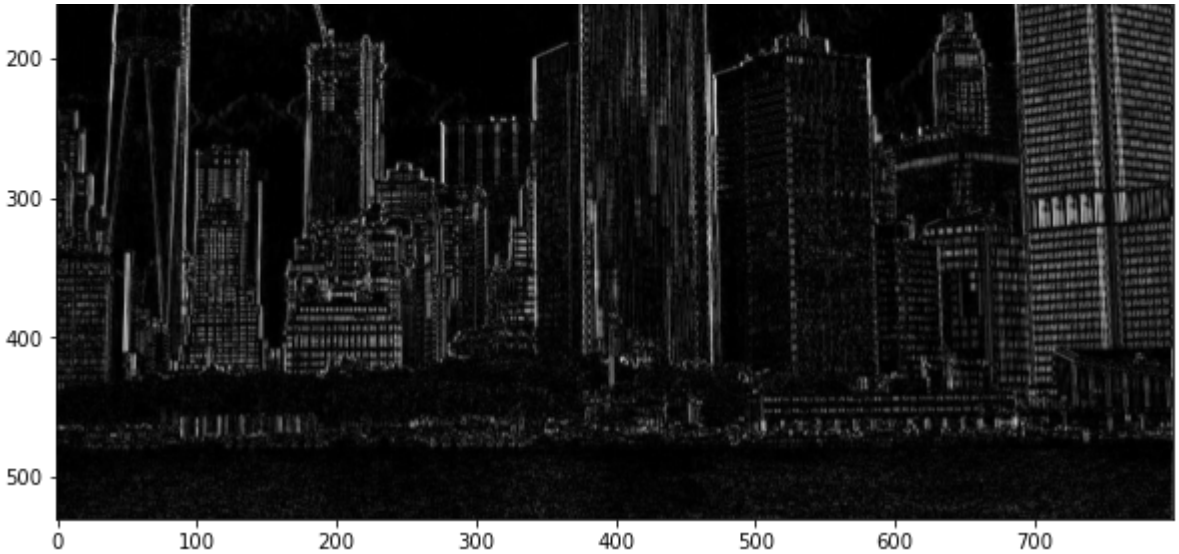
Apply Filters

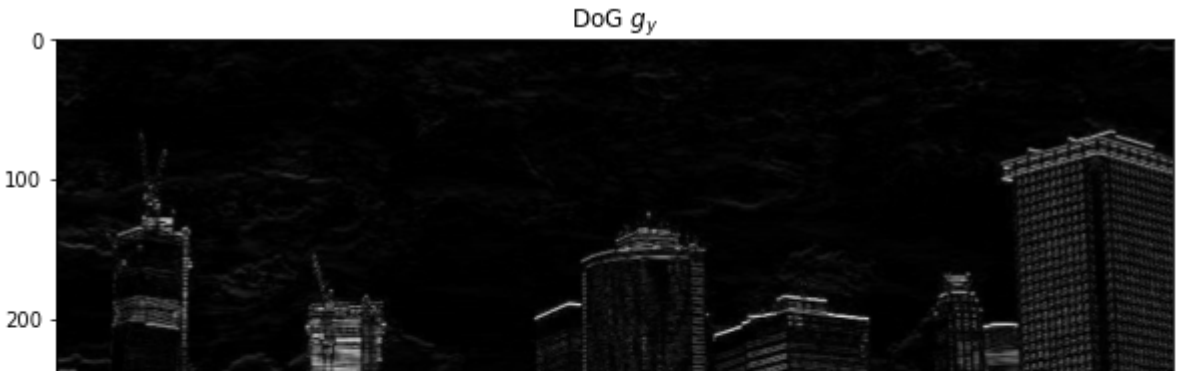
```
filters = [[gausBlur1,1,"Gaussian $3 x 3$"],[gausBlur2,2,"Gaussian $5 x 5$"],[gx,1,"DoG $g_x$

for i in filters:
    oout = myConv(img,i[0],i[1])
    plt.figure(figsize=(10,8))
    plt.title(i[2])
    plt.imshow(oout,cmap = "gray")
    plt.show()
```


Gaussian 3x3
0
100
200
300
400
500
0
100
200
300
400
500
600
700


Gaussian 5x5
0
100
200
300
400
500
0
100
200
300
400
500
600
700

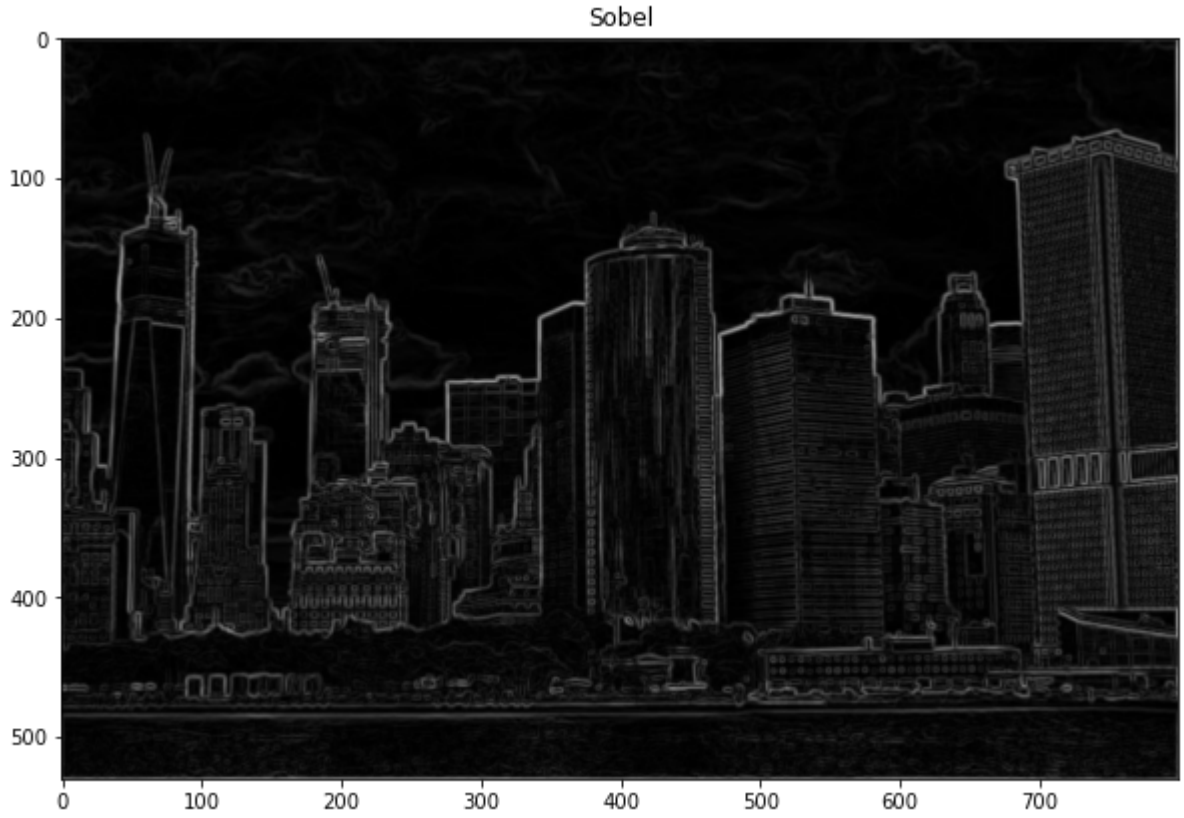

DoG g_x
0
100





Sobel

```
# preprocessing before appling sobel filter
sob_pp = myConv(img,gausBlur2,2)

#appling sobel filter

oout = sobelFilt(myConv(sob_pp,gx,1),myConv(sob_pp,gy,1))
plt.figure(figsize=(10,8))
plt.title("Sobel")
plt.imshow(oout,cmap = "gray")
plt.show()
```


Sobel
0
100
200
300
400
500
0
100
200
300
400
500
600
700

Part 2

K-Means

data load and true data

```
import numpy as np
#%matplotlib widget
import matplotlib.pyplot as plt
import cv2 as cv
import random
from tqdm import tqdm
```

K-Means algo

functions

I've used K-Means before in some of my EE classes, so this is just a slight alteration to something I've already built.

- myKmeans: is my K-Means algo
  - It starts by getting K random indexes
  - Turns the random indexes into the intial mean points
  - do:
    - cluster nearest points
    - remean
    - calculate error
    - repeat r times
  - return error, cluster points, means
- findBest: runs r times
  - initializes the starting error at infinity
  - runs my K-Means function r times
  - replaces the mean points and cluster set with the returned lowest error

```
#ps is random indices
def myKMeans(data,K,r):
    #start with random indexes
    ps = random.sample(range(len(data[:,1])),K)
```

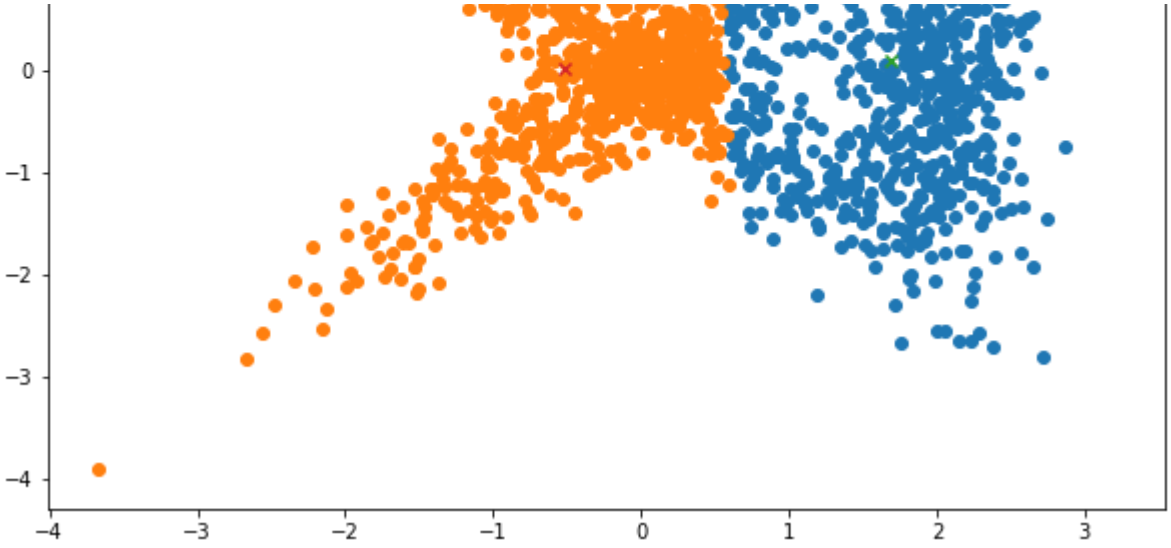
```
    ps = random.sample(range(len(data[:,1])), K)
    points=[]
    #turn indexes into points
    for k in ps:
        points.append(data[k])
    #run algo
    for x in tqdm(range(r),desc="K-Meaning Iter:"):
        clus =  np.array([])
        #cluster to closest point
        for i in data:
            dis = []
            for j in points:
                dis.append(np.linalg.norm(j-i))
            clus = np.append(clus,dis.index(min(dis)))
        #remean
        points = []
        err_p = np.zeros(K)
        for y in range(K):
            sing_p = data[np.where(clus == y)].mean(axis = 0)
            points.append(sing_p)
            err_p[y] = np.linalg.norm((data[np.where(clus == y)] - sing_p))
        #error
        err = (np.sum(err_p))
        #print("Error: ",err)
    return err,clus,points


def findBest(data,K,r):
    error = np.inf
    for i in range(r):
        err, temp_clus,temp_points = myKMeans(data,K,r)
        if err < error:
            error = err
            points = temp_points.copy()
            clus = temp_clus.copy()
    print("Error: ", error)
    return clus,points
```

Loading Cluster Data

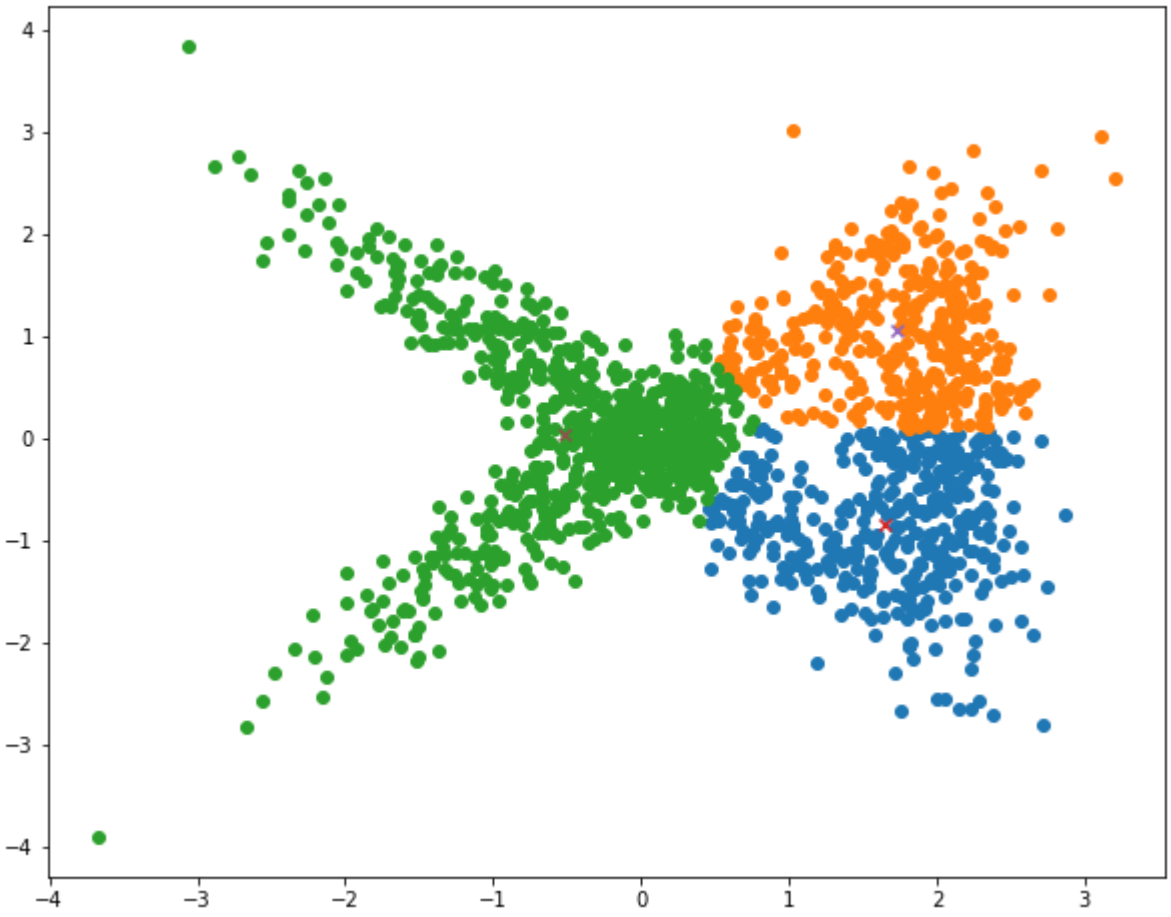
This is the cluster dataset. It loads the dataset obviously. As stated in the question prompt, the data contains 3 gaussian clusters, each with 500 points and aded to the file contiguiously. Using that information, I segmented the data into 3 equal sets of 500 hundred points, found the means and plotted the true means in order to see what was going on. Two of the means were very close together, which I assume will throw the K-Means Alogithm off when looking for $K = 3$. However this might make $K = 2$, plant one mean very close to those means that are close together.

Rather than just calling my functions 3 times with $K = 2, K = 3, K = 4$, I just used a for loop on a set of K values $K = [2, 3, 4]$. This was just for readability. The sum of the squarred errors for $K = 2$ was 66.7, for $K = 3$ was 64.0, and for $K = 4$ was 63.9.

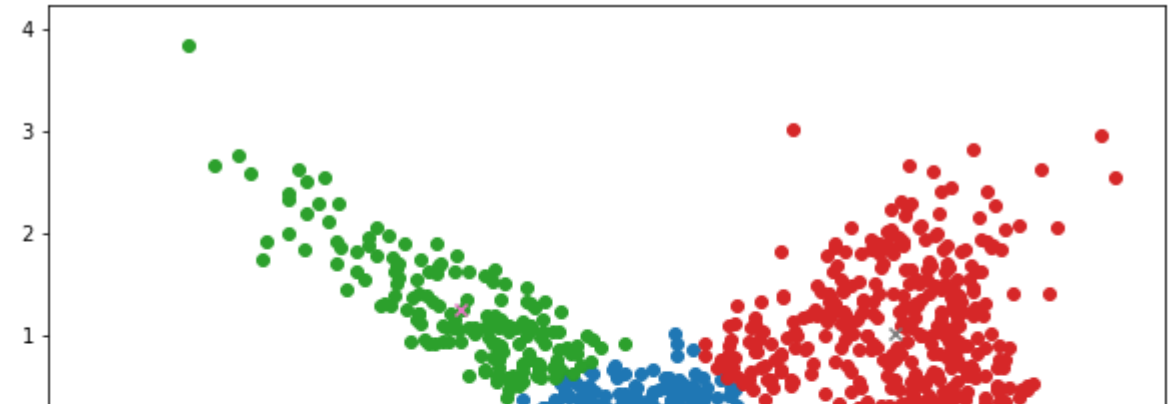
```
r= 10
data = np.loadtxt("510_cluster_dataset.txt")
print(data.shape)
plt.figure(figsize=(10,8))
plt.scatter(data[:,0],data[:,1])
truth= np.array([data[:500].mean(axis=0),data[500:1000].mean(axis=0),data[1000:1500].mean(axi
plt.scatter(truth[:,0],truth[:,1],color = 'black',marker='x')
plt.show()

for K in range(2,5,1):
    clus,points = findBest(data,K,r)


    plt.figure(figsize=(10,8))
    for y in range(K):
        d1 = data[np.where(clus == y)]
        plt.scatter(d1[:,0],d1[:,1],marker='o')

    for i in points:
        plt.scatter(i[0],i[1],marker='x')
    titl= "K = " + str(K)
    plt.title(titl)
plt.show()
```

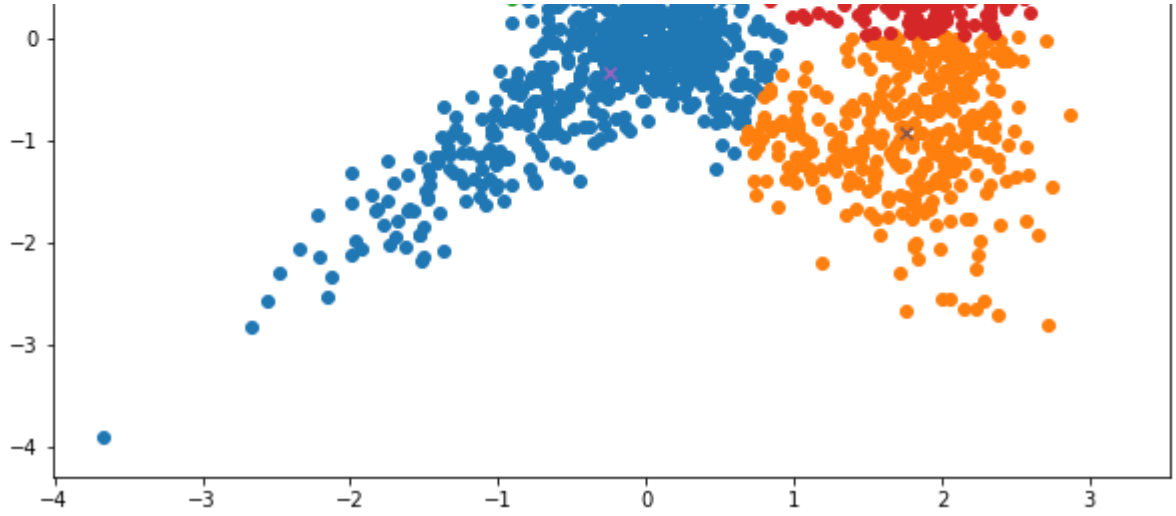


K = 3



K = 4




0
−1
−2
−3
−4
−4
−3
−2
−1
0
1
2
3

Part 2, (ii)

K = 5

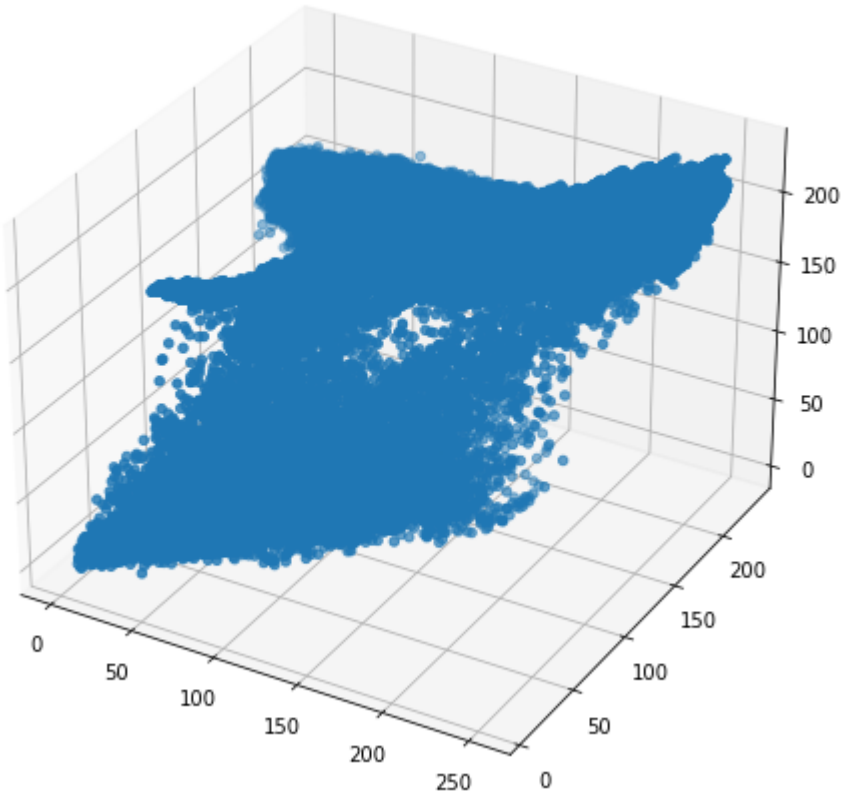
For this next portion, I shrank the size of images down to half the size. Even shrinking this down as much as I did, made my K-Means Alogorithm run for a few hours to complete ALL of the calculations. I could have made this go faster if I didn't choose $r = 10$. However, I choose $r = 10$ in order to get my K-Means to settle nicely. The biggest issue with this was the fact that it not only ran my K-Means algorithm for 10 iterations, it means I ran it 10 times and returning the K-Means with the lowest error.

I plotted the point in the color space, and segmented each cluster with a different color. As an added bonus, i also returned the original image recolored by the mean colors found by the algoithm. Someone in the study group mentioned they did this, and I thought it sounded cool, so I implemented it as well. I thought it was a good visual confirmation of what was going on. This process was repeated for the both images, and for $K = 5$ and $k = 10$.

```
img = cv.imread('Kmean_img1.jpg')
img = cv.cvtColor(img, cv.COLOR_BGR2RGB)
img = cv.resize(img, (0,0), fx=0.5, fy=0.5)
plt.figure(figsize=(10,8))
plt.title("Original: beach 1")
plt.imshow(img)
plt.show()
```


```
print(img.shape)
print(img.reshape(-1,3).shape)
img_dat = img.reshape(-1,3)
fig = plt.figure(figsize=(10,8))
ax = fig.add_subplot(projection='3d')
ax.scatter(img_dat[:,0], img_dat[:,1], img_dat[:,2], marker='o')
plt.show()
```

```
(336, 600, 3)
(201600, 3)
```

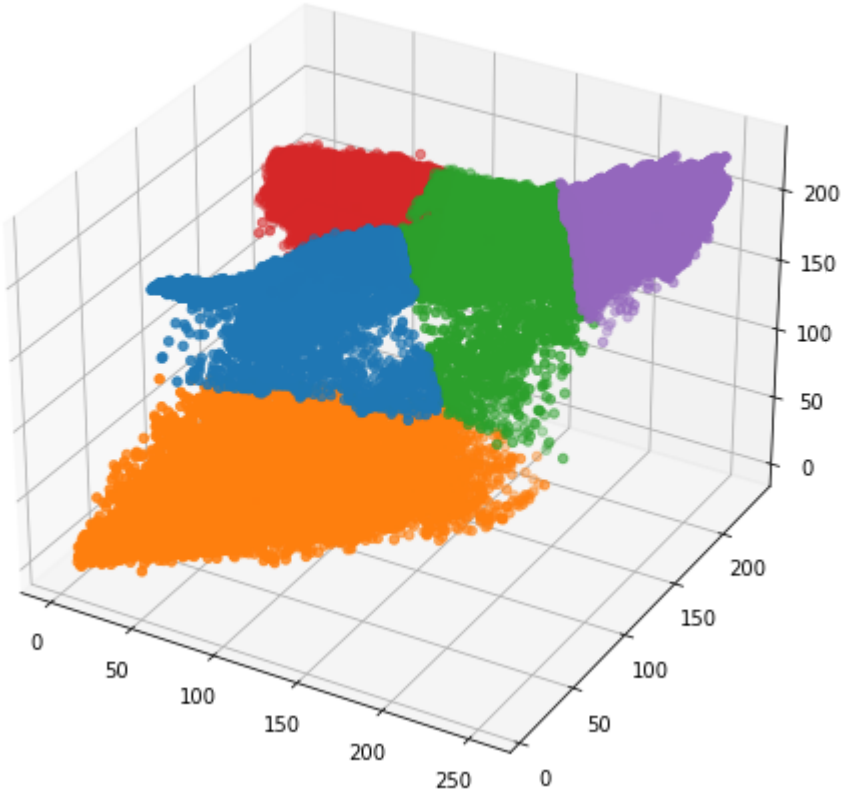



```
K = 5
r =10
clus,points = findBest(img_dat,K,r)
```

```
fig = plt.figure(figsize=(10,8))
ax = fig.add_subplot(projection='3d')
```

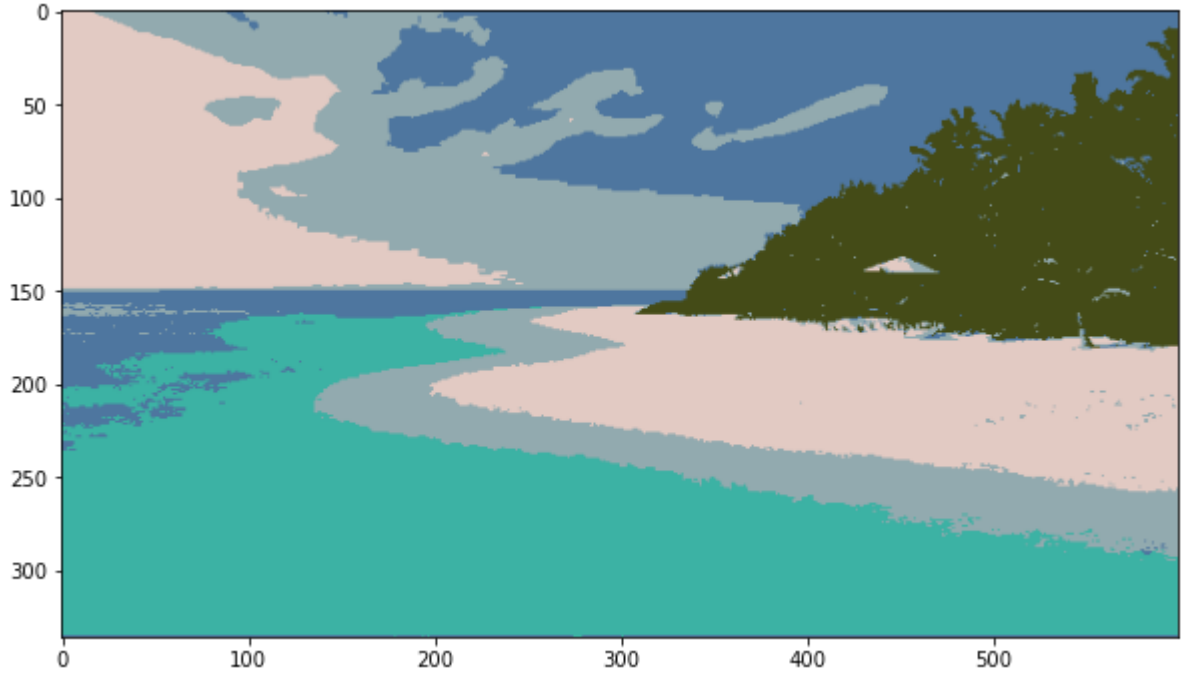
```
for y in range(K):
    d1 = img_dat[np.where(clus == y)]
    ax.scatter(d1[:,0],d1[:,1],d1[:,2],marker='o')

for i in points:
    ax.scatter(i[0],i[1],i[2],marker='x')
plt.show()
```

```
K-Meaning Iter:: 100%|██████████| 10/10 [05:31<00:00, 33.15s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [05:05<00:00, 30.51s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [05:05<00:00, 30.54s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [05:26<00:00, 32.68s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [05:28<00:00, 32.88s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [05:27<00:00, 32.71s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [05:17<00:00, 31.75s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [05:00<00:00, 30.03s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [05:05<00:00, 30.60s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [05:03<00:00, 30.32s/it]
```




```
new_pic = np.array([0,0,0])
for i in tqdm(clus,desc="re-color: "):
    new_pic = np.vstack((new_pic,points[int(i)]))
new_pic = new_pic[1:]
new_pic = np.reshape(new_pic,img.shape).astype(np.uint8)
plt.figure(figsize=(10,8))
plt.imshow(new_pic)
plt.show()
```

```
re-color: 100%|██████████| 201600/201600 [01:51<00:00, 1810.49it/s]
```

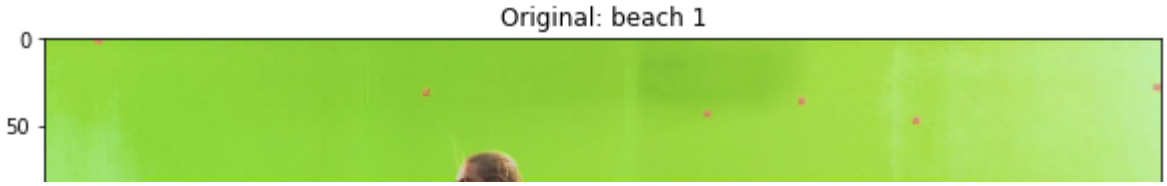


K = 5

Lord of the rings

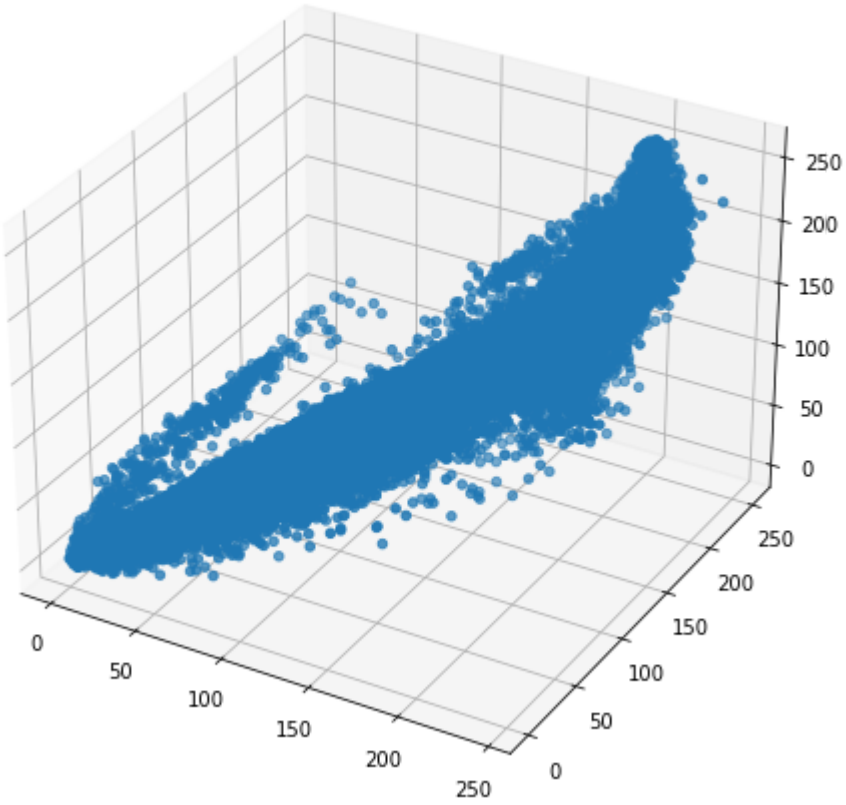
- the title is wrong, but I didn't want to rerun it. it took a few hours to run everything.

```
img = cv.imread('Kmean_img2.jpg')
img = cv.cvtColor(img, cv.COLOR_BGR2RGB)
img = cv.resize(img, (0,0), fx=0.5, fy=0.5)
plt.figure(figsize=(10,8))
plt.title("Original: beach 1")
plt.imshow(img)
plt.show()
```




```
print(img.shape)
print(img.reshape(-1,3).shape)
img_dat = img.reshape(-1,3)
fig = plt.figure(figsize=(10,8))
ax = fig.add_subplot(projection='3d')
ax.scatter(img_dat[:,0], img_dat[:,1], img_dat[:,2], marker='o')
plt.show()
```

```
(360, 640, 3)
(230400, 3)
```

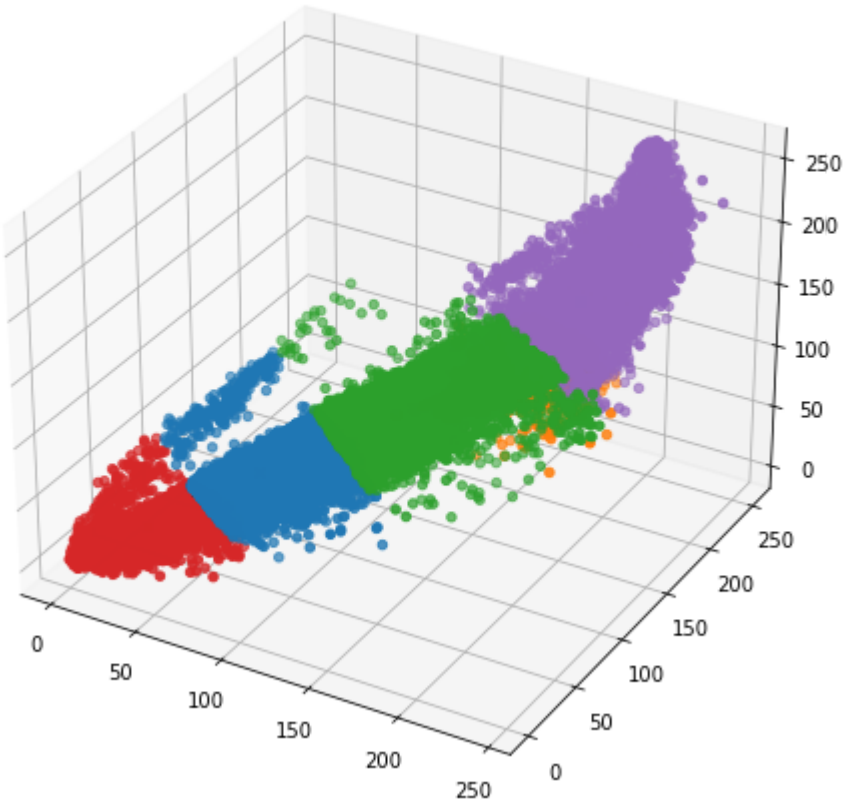



```
K = 5
r =10
clus,points = findBest(img_dat,K,r)
```

```
fig = plt.figure(figsize=(10,8))
ax = fig.add_subplot(projection='3d')
for y in range(K):
    d1 = img_dat[np.where(clus == y)]
    ax.scatter(d1[:,0],d1[:,1],d1[:,2],marker='o')
```

```
for i in points:
    ax.scatter(i[0],i[1],i[2],marker='x')
plt.show()
```

```
K-Meaning Iter:: 100%|██████████| 10/10 [06:53<00:00, 41.39s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [07:00<00:00, 42.05s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [06:53<00:00, 41.39s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [06:54<00:00, 41.44s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [07:00<00:00, 42.03s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [06:54<00:00, 41.47s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [07:00<00:00, 42.04s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [06:58<00:00, 41.82s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [06:54<00:00, 41.42s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [06:55<00:00, 41.52s/it]
```




```
new_pic = np.array([0,0,0])
for i in tqdm(clus,desc="re-color: "):
    new_pic = np.vstack((new_pic,points[int(i)]))
new_pic = new_pic[1:]
new_pic = np.reshape(new_pic,img.shape).astype(np.uint8)
plt.figure(figsize=(10,8))
plt.imshow(new_pic)
plt.show()
```

```
re-color: 100%|██████████| 230400/230400 [02:28<00:00, 1554.29it/s]
```

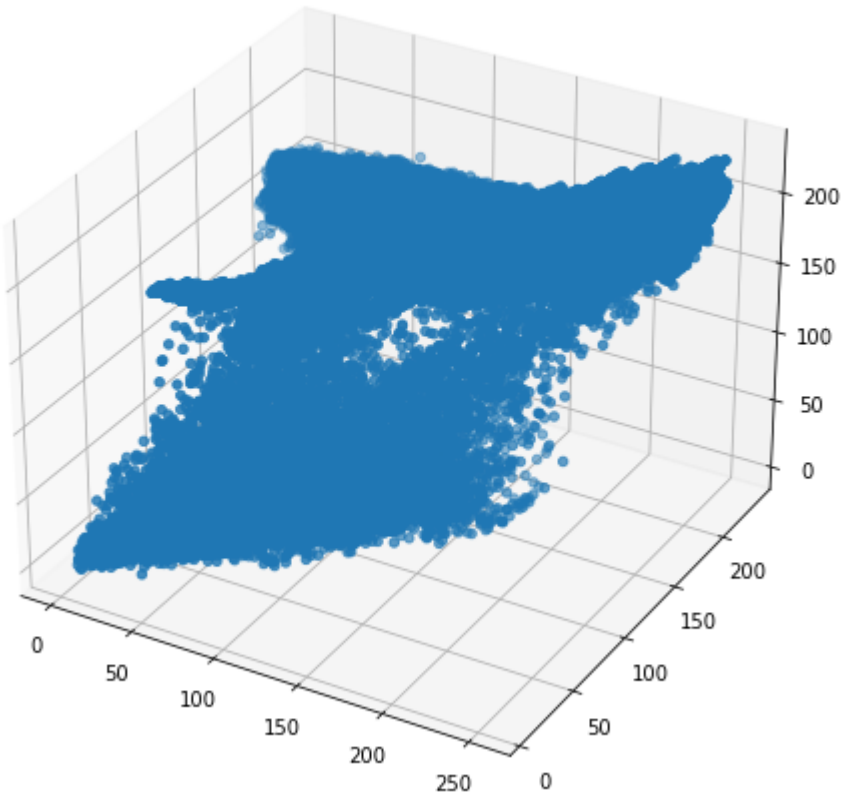
K=10

Beach

```
img = cv.imread('Kmean_img1.jpg')
img = cv.cvtColor(img, cv.COLOR_BGR2RGB)
img = cv.resize(img, (0,0), fx=0.5, fy=0.5)
plt.figure(figsize=(10,8))
plt.title("Original: beach 1")
plt.imshow(img)
plt.show()
```

```
print(img.shape)
print(img.reshape(-1,3).shape)
img_dat = img.reshape(-1,3)
fig = plt.figure(figsize=(10,8))
ax = fig.add_subplot(projection='3d')
ax.scatter(img_dat[:,0], img_dat[:,1], img_dat[:,2], marker='o')
plt.show()
```

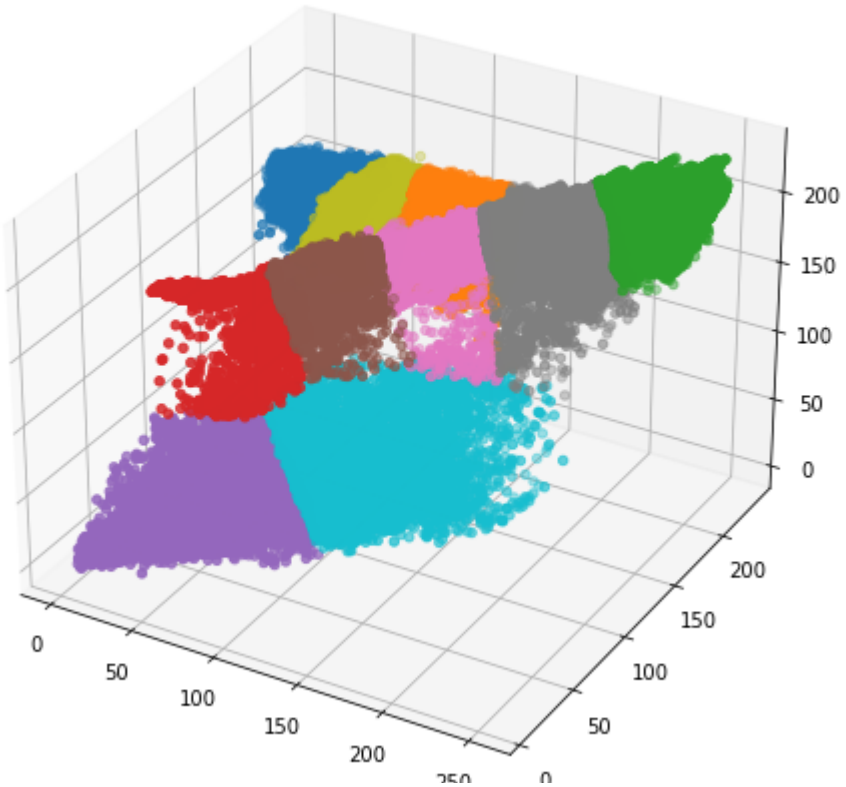
```
(336, 600, 3)
(201600, 3)
```




```
K = 10
r =10
clus,points = findBest(img_dat,K,r)
```

```
fig = plt.figure(figsize=(10,8))
ax = fig.add_subplot(projection='3d')
for y in range(K):
    d1 = img_dat[np.where(clus == y)]
    ax.scatter(d1[:,0],d1[:,1],d1[:,2],marker='o')

for i in points:
    ax.scatter(i[0],i[1],i[2],marker='x')
plt.show()
```

```
K-Meaning Iter:: 100%|██████████| 10/10 [05:34<00:00, 33.41s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [05:45<00:00, 34.54s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [05:39<00:00, 33.96s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [05:44<00:00, 34.43s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [05:38<00:00, 33.90s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [05:44<00:00, 34.42s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [05:38<00:00, 33.87s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [05:43<00:00, 34.38s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [05:37<00:00, 33.73s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [05:37<00:00, 33.74s/it]
```




```
new_pic = np.array([0,0,0])
for i in tqdm(clus,desc="re-color: "):
    new_pic = np.vstack((new_pic,points[int(i)]))
new_pic = new_pic[1:]
new_pic = np.reshape(new_pic,img.shape).astype(np.uint8)
plt.figure(figsize=(10,8))
plt.imshow(new_pic)
plt.show()
```

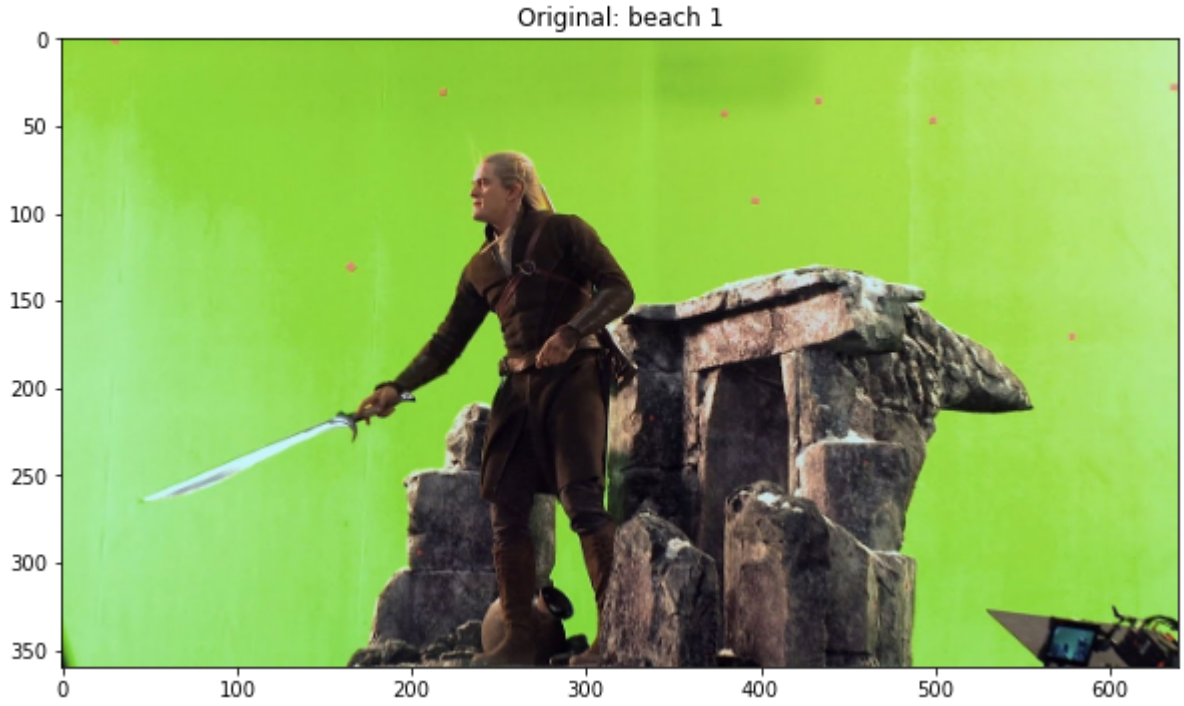
```
re-color: 100%|██████████| 201600/201600 [01:51<00:00, 1801.03it/s]
```

K = 10

Lord of the rings

- don't mind the bad title.... I discovered it after it had run for everal hours and I didn't want to re run it again to fix that.

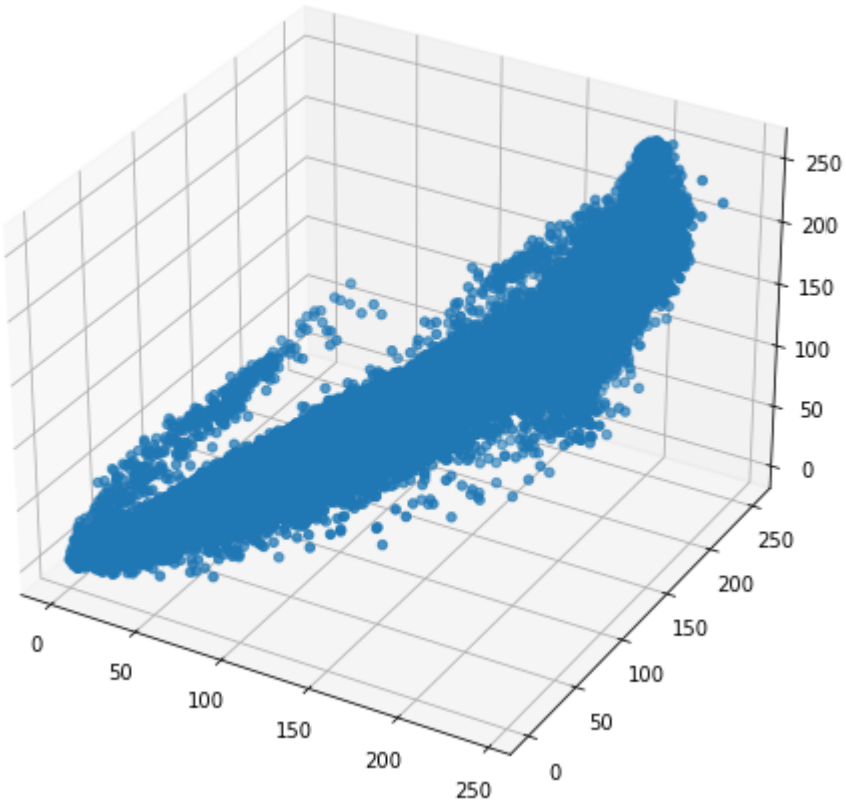
```
img = cv.imread('Kmean_img2.jpg')
img = cv.cvtColor(img, cv.COLOR_BGR2RGB)
img = cv.resize(img, (0,0), fx=0.5, fy=0.5)
plt.figure(figsize=(10,8))
plt.title("Original: beach 1")
plt.imshow(img)
plt.show()
```




```
print(img.shape)
print(img.reshape(-1,3).shape)
img_dat = img.reshape(-1,3)
```

```
img_dat = img.reshape(-1,3)
fig = plt.figure(figsize=(10,8))
ax = fig.add_subplot(projection='3d')
ax.scatter(img_dat[:,0], img_dat[:,1], img_dat[:,2], marker='o')
plt.show()
```

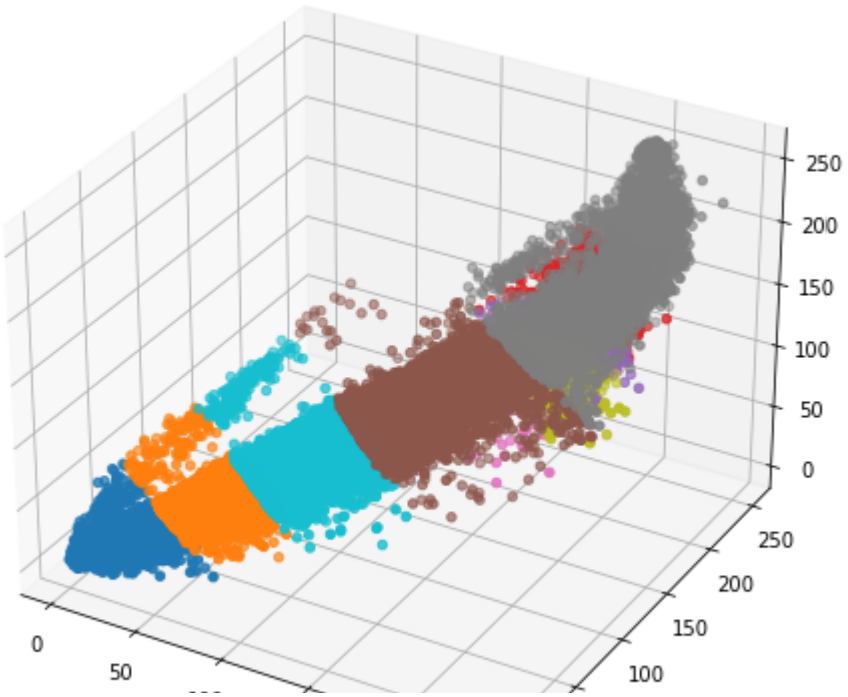
```
(360, 640, 3)
(230400, 3)
```




```
K = 10
r =10
clus,points = findBest(img_dat,K,r)
```

```
fig = plt.figure(figsize=(10,8))
ax = fig.add_subplot(projection='3d')
for y in range(K):
    d1 = img_dat[np.where(clus == y)]
    ax.scatter(d1[:,0],d1[:,1],d1[:,2],marker='o')

for i in points:
    ax.scatter(i[0],i[1],i[2],marker='x')
plt.show()
```

```
K-Meaning Iter:: 100%|██████████| 10/10 [07:42<00:00, 46.23s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [07:53<00:00, 47.31s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [07:53<00:00, 47.35s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [07:53<00:00, 47.38s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [07:56<00:00, 47.63s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [07:55<00:00, 47.56s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [07:56<00:00, 47.60s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [07:54<00:00, 47.48s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [07:55<00:00, 47.55s/it]
K-Meaning Iter:: 100%|██████████| 10/10 [07:57<00:00, 47.77s/it]
```




```
new_pic = np.array([0,0,0])
for i in tqdm(clus,desc="re-color: "):
    new_pic = np.vstack((new_pic,points[int(i)]))
new_pic = new_pic[1:]
new_pic = np.reshape(new_pic,img.shape).astype(np.uint8)
plt.figure(figsize=(10,8))
plt.imshow(new_pic)
plt.show()
```

Part3

- first part is my best attempt
- second part is using tools
    - so much shorter

Part 3 was a little harder than the others. I start with the original images, and turn them to black a white. Because the second image is so much much bigger, I shrunk the size down. This gave me better results than using the large second image. I then call the openCV SIFT function to get my keypoints for both images. On my first go, I use my keypoints fom image one to find the nearest neighbor on my points from image two. I then sort the list of nearest neightbors in order of smallest to lrgest distances. Then I take the top 10% and draw lines from one point to the other. It seems like a lot of the points are very close.

```
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
import cv2 as cv

img = cv.imread('SIFT1_img.jpg')
gray= cv.cvtColor(img,cv.COLOR_BGR2GRAY)
plt.figure(figsize=(10,8))
plt.imshow(gray,cmap = "gray")
plt.show()
```


```
sift = cv.SIFT_create()
kp = sift.detect(gray,None)
img1=cv.drawKeypoints(gray,kp,img)


# https://stackoverflow.com/questions/35884409/how-to-extract-x-y-coordinates-from-opencv-cv2
pts1 = cv.KeyPoint_convert(kp)
pts1 = np.unique(pts1,axis=0)
plt.figure(figsize=(10,8))
plt.imshow(img1,cmap = "gray")
plt.show()
```

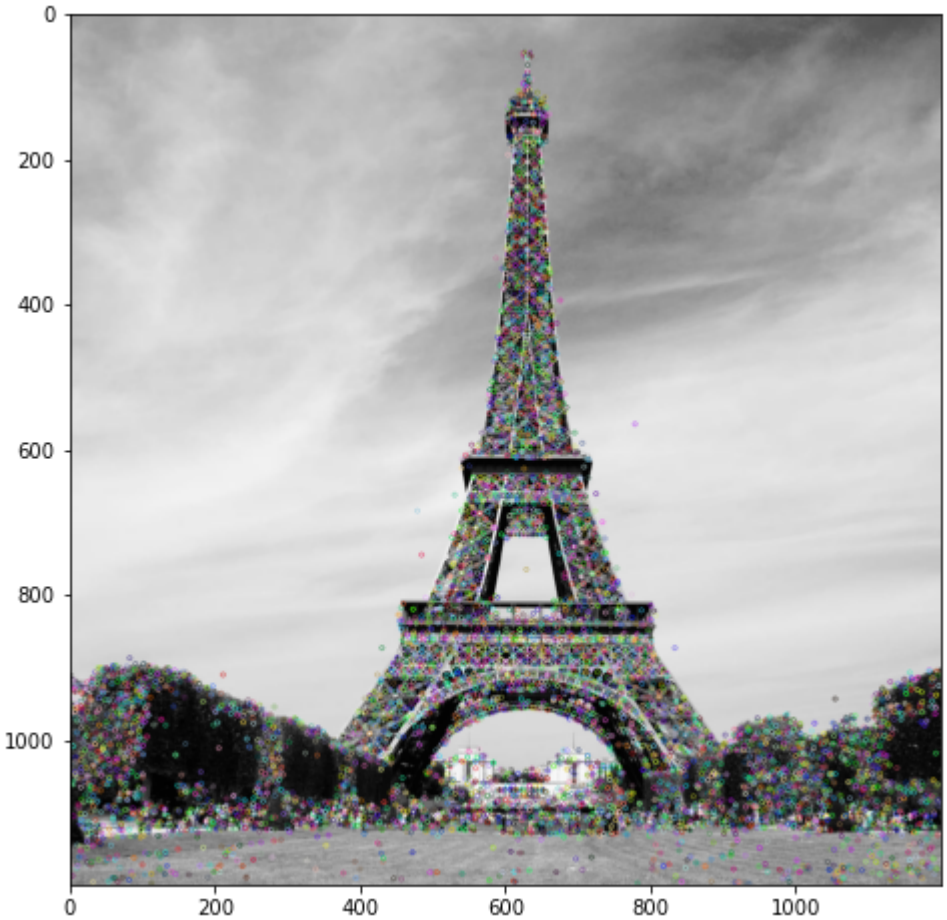
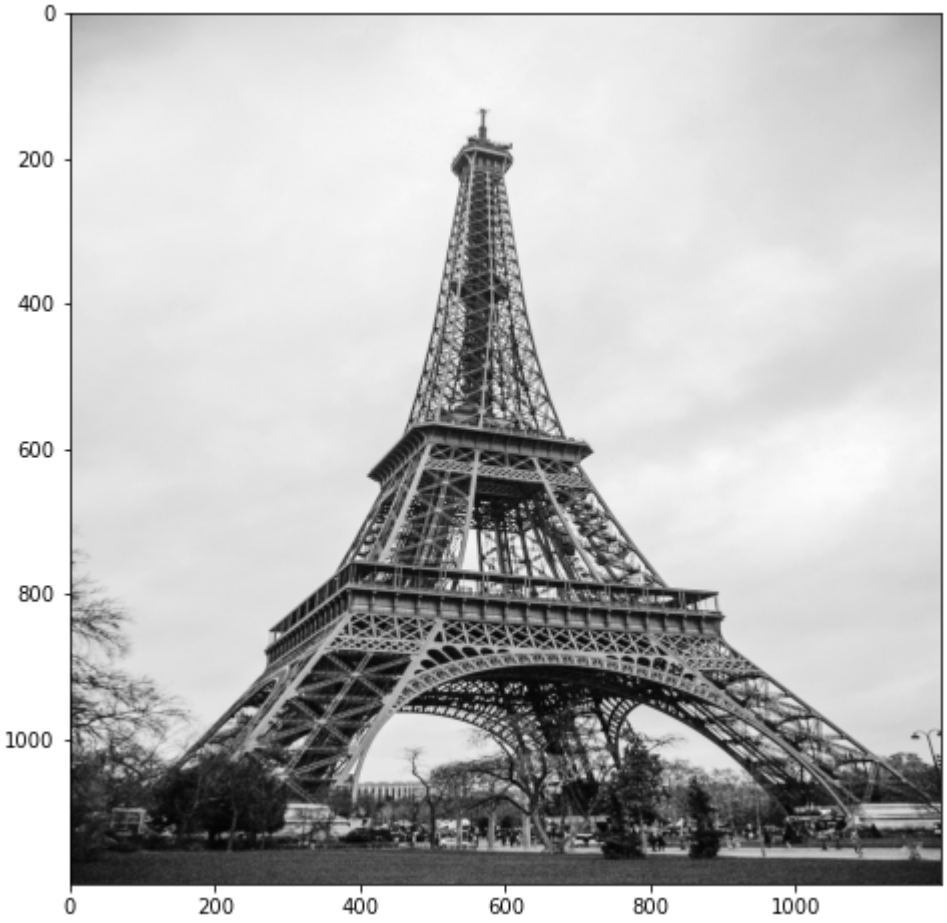


image 2

```
img = cv.imread('SIFT2_img.jpg')
dim = (img1.shape[0],img1.shape[1])
```

```
img = cv.resize(img,dim, interpolation = cv.INTER_AREA)
gray= cv.cvtColor(img,cv.COLOR_BGR2GRAY)
plt.figure(figsize=(10,8))
plt.imshow(gray,cmap = "gray")
plt.show()
```




```
sift = cv.SIFT_create()
kp = sift.detect(gray,None)
img2=cv.drawKeypoints(gray,kp,img)


# https://stackoverflow.com/questions/35884409/how-to-extract-x-y-coordinates-from-opencv-cv2
pts2 = cv.KeyPoint_convert(kp)
pts2 = np.unique(pts2,axis=0)
plt.figure(figsize=(10,8))
plt.imshow(img2,cmap = "gray")
plt.show()
```

Nearest Neighbor

```
I1 = np.array([0,0,0])
#print(pts1)
for i,t in enumerate(pts1):
    dist = (t - pts2)**2
    dist = np.sum(dist, axis=1)
    dist = np.sqrt(dist)
    I1 =  np.vstack((I1,[i,np.argmin(dist),np.amin(dist)]))
I1=I1[1:]
```

check matches I1

```
I1 = I1[np.argsort(I1[:, 2])]
```

```
I2 = np.array([0,0,0])
#print(pts1)
for i,t in enumerate(pts2):
    dist = (t - pts1)**2
    dist = np.sum(dist, axis=1)
    dist = np.sqrt(dist)
    I2 =  np.vstack((I2,[i,np.argmin(dist),np.amin(dist)]))
I2=I2[1:]
```

check matches l2

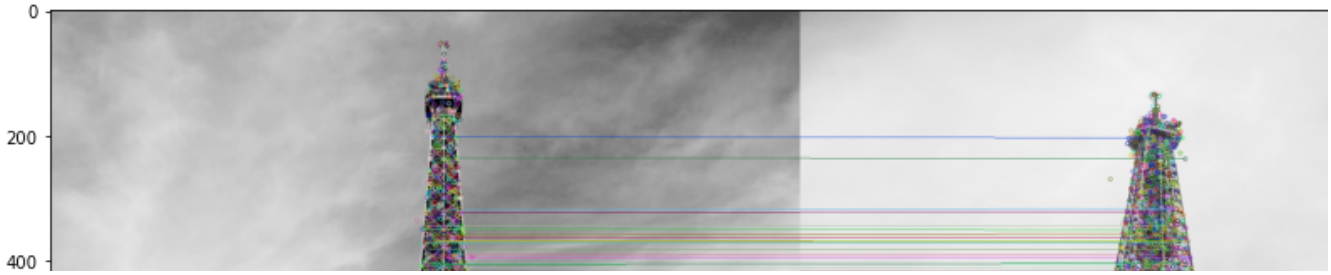
```
I2 = I2[np.argsort(I2[:, 2])]
```

lines

- drawing lines

```
m1 = np.zeros((img1.shape[0],img2.shape[1],3))
m2 = np.zeros((img2.shape[0]-img1.shape[0],img1.shape[1],3))

pts2[:,0] =  pts2[:,0] + img1.shape[0]
#pts2[:,1] =  pts2[:,1] + img1.shape[1]

#h_img1 = np.concatenate((m2, img2),axis=1)
#h_img2 = np.concatenate((img1,m2),axis=0)
#plt.imshow(h_img2)
#plt.show()
h_img = np.concatenate((img1,img2),axis = 1).astype(np.uint8)

for xx in range(int(len(pts2[:,1])*.1)):
    colors = np.random.random(size=3) * 256
    h_img = cv.line(h_img, tuple(pts1[int(I1[xx,0])]), tuple(pts2[int(I1[xx,1])]),colors,1)
plt.figure(figsize=(16,12))
plt.imshow(h_img,cmap="gray")
plt.show()
```

```
<ipython-input-9-0d3272ea5dc4>:15: DeprecationWarning: an integer is required (got type
  h_img = cv.line(h_img, tuple(pts1[int(I1[xx,0])]), tuple(pts2[int(I1[xx,1])]),colors,1
```

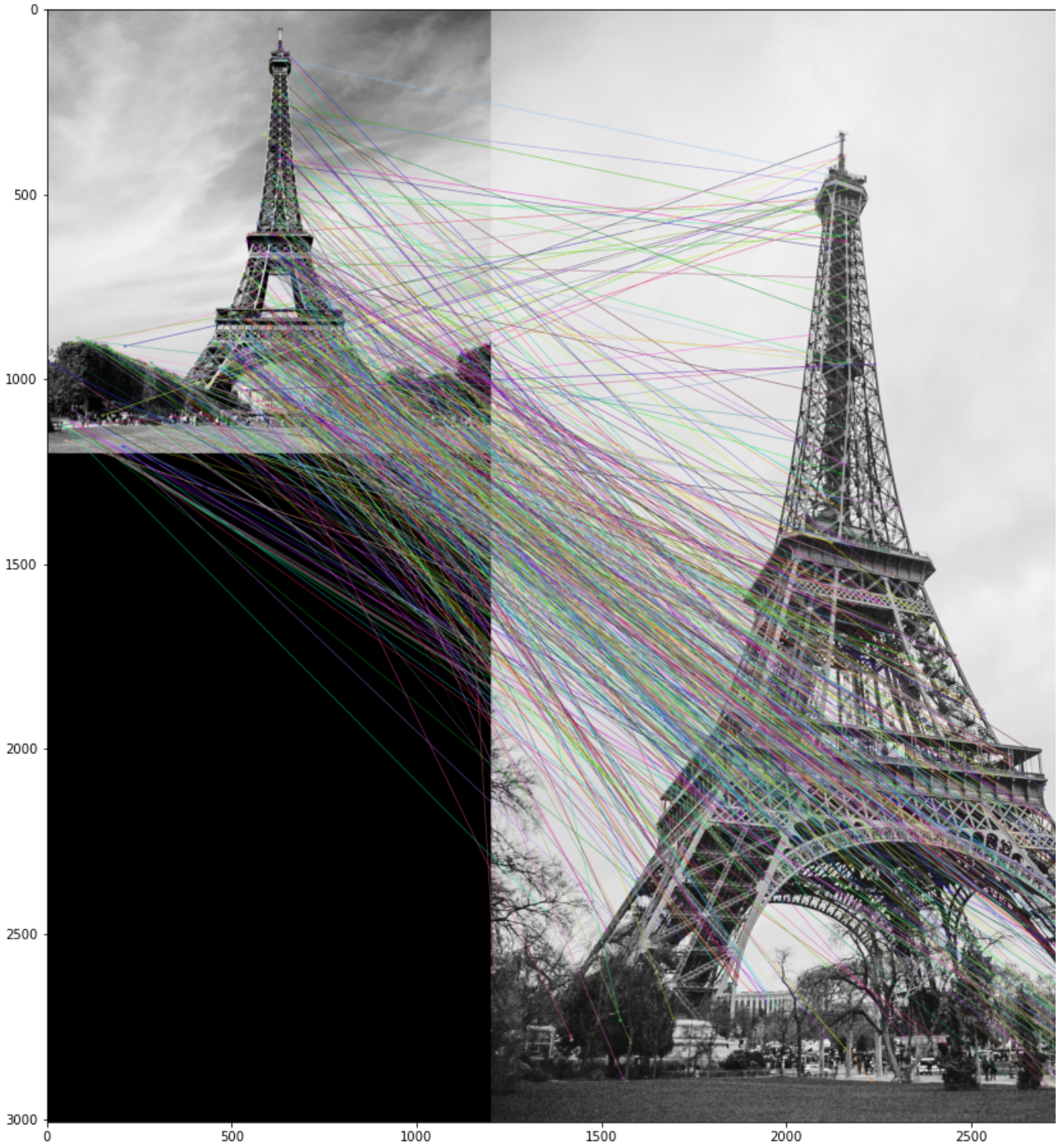


directly using the tools

- [open cv docs]

This was my second attempt. I wasn't happy with my original results. I didn't like that I had to shrink image2 down. So instead, I retried this assignment directly using the premade openCV tools. Obviously, this worked way better. The matcher was a lot faster, though I notice there are still some errors even in the top 10%. I assume that a lot of these bad matches could be filtered out. I did explicitely use the l2 norm in my mather as told in the instructions, though I assume I could use something better to get better results.

```
import numpy as np
import cv2 as cv
import matplotlib.pyplot as plt
img1 = cv.imread('SIFT1_img.jpg',cv.IMREAD_GRAYSCALE)
img2 = cv.imread('SIFT2_img.jpg',cv.IMREAD_GRAYSCALE)

sift = cv.SIFT_create()

kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)

bf = cv.BFMatcher(cv.NORM_L2)
matchList = bf.match(des1,des2)

matchList = sorted(matchList, key = lambda x:x.distance)

img3 = cv.drawMatches(img1,kp1,img2,kp2,matchList[:(len(matchList)//10)],None,flags=cv.DrawMa

plt.figure(figsize=(20,16))
plt.imshow(img3)
plt.show()
```


0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500