**CSE 573 Computer Vision and Image Processing**

**Project 2 Report**

**Name: - Chinmay Swami**

**Person No: - 50290385**

**UB Name: - chinmayp**

# Task 1: - Image Features and Homography

Used openCV official documents to implement the code. Also to resolve the pixel loss issue I referred to openCV thread below:

http://answers.opencv.org/question/144252/perspective-transform-without-crop/

Steps:-

1] Using SIFT generate Keypoints.

2] Match Keypoints using k-nearest neighbors.

3] Compute Homography matrix.

4] Draw the match image for around 10 random matches using only inliers.

5] Generate panorama by warping first image onto second image using homography matrix

## Code

import numpy as np;

import cv2

from functions import generateKeypoints,generateMatches

import random

# Task1.1

image1G = cv2.imread("Images/mountain1.jpg",cv2.IMREAD\_GRAYSCALE)

image2G = cv2.imread("Images/mountain2.jpg",cv2.IMREAD\_GRAYSCALE)

height1,width1 = image1G.shape

height2,width2 = image2G.shape

# Detect and compute keypoints for image 1 and image 2

keyPointImage1, descriptorsImage1, keyPointImage2, descriptorsImage2 = generateKeypoints(image1G, image2G)

img1 =cv2.drawKeypoints(image1G,keyPointImage1,image1,flags=cv2.DRAW\_MATCHES\_FLAGS\_DRAW\_RICH\_KEYPOINTS)

cv2.imwrite("task1\_sift1.jpg",img1)

img2 =cv2.drawKeypoints(image2G,keyPointImage2,image2,flags=cv2.DRAW\_MATCHES\_FLAGS\_DRAW\_RICH\_KEYPOINTS)

cv2.imwrite("task1\_sift2.jpg",img2)

# Task 1.2

goodMatches = generateMatches(descriptorsImage1, descriptorsImage2)

# Plot the matched keypoints that satisfied the threshold

img3 = cv2.drawMatches(image1G, keyPointImage1, image2G, keyPointImage2, goodMatches, None, flags=2)

cv2.imwrite("task1\_matches\_knn.jpg", img3)

# Task 1.3 & 1.4

sourcePoints = np.float32([keyPointImage1[m.queryIdx].pt for m in goodMatches])

destinationPoints = np.float32([keyPointImage2[m.trainIdx].pt for m in goodMatches])

homographyMatrix, inOut = cv2.findHomography(sourcePoints, destinationPoints, cv2.RANSAC, 5.0)

print("Homography matrix H is: \n",homographyMatrix)

matchedMask = inOut.ravel().tolist() # ravel is used to flatten the inOut array

# Calculate 10 random inliers

matchedMask2 = np.zeros(len(matchedMask))

matchedMask2 = matchedMask2.tolist()

count = 0

while count <= 10:

index = random.randint(0, len(matchedMask)-1)

if matchedMask[index] == 1:

matchedMask2[index] = 1

count += 1

# unique, counts = np.unique(matchedMask, return\_counts=True)

# print("\n Inliers and Outliers Stats: ", dict(zip(unique, counts)))

parameters = dict(matchColor = (0, 200, 0), # draw matches in black color

singlePointColor = None,

matchesMask = matchedMask2, # draw only inliers

flags = 2)

img5 = cv2.drawMatches(image1G, keyPointImage1, image2G, keyPointImage2, goodMatches, None, \*\*parameters)

cv2.imwrite("task1\_matches.jpg",img5)

# Code to find corners after transformation is applied

# Calculating the end points

endPoints = np.array([

[0,0],

[0,height1],

[width1, height1],

[width1,0]

])

corners = cv2.perspectiveTransform(np.float32([endPoints]), homographyMatrix)

# Find the bounding rectangle

x, y, boundedWidth, boundedHeight = cv2.boundingRect(corners)

# Translating -ve points which are causing loss of pixels into +Ve points

intermediateTransformationMatrix = np.array([

[ 1, 0, -x ],

[ 0, 1, -y ],

[ 0, 0, 1 ]

])

homographyMatrix = intermediateTransformationMatrix.dot(homographyMatrix)

print("\n New Homography matrix H is: \n",homographyMatrix)

result1 = cv2.warpPerspective(image1G, homographyMatrix, (boundedWidth, boundedHeight), flags=cv2.INTER\_CUBIC)

result1 = cv2.copyMakeBorder(result1,0,15,10,0,cv2.BORDER\_CONSTANT)

cv2.imwrite("image1wrap.jpg",result1)

boundedHeight, boundedWidth = result1.shape

# Creating a blank image

result = np.zeros((max(boundedHeight, height2), width2+width2), dtype=np.uint8)

result[:boundedHeight, :boundedWidth] = result1

result[boundedHeight-height2:, width1:] = image2G

cv2.imwrite("task1\_pano.jpg",result)

def generateKeypoints(image1,image2):

sift = cv2.xfeatures2d.SIFT\_create()

keyPoint1, descriptors1 = sift.detectAndCompute(image1, None)

keyPoint2, descriptors2 = sift.detectAndCompute(image2, None)

return keyPoint1,descriptors1,keyPoint2,descriptors2

def generateMatches(descriptor1,descriptor2):

bestMatcherObject = cv2.BFMatcher()

matchedKeypoints = bestMatcherObject.knnMatch(descriptor1, descriptor2, k=2)

# Filter good matches: Lowe's Ratio test to determine high quality features

goodMatches = []

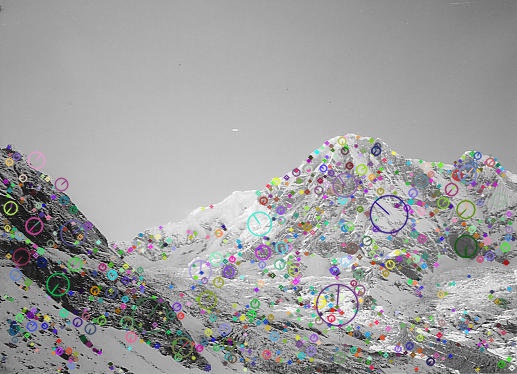
for i,(m, n) in enumerate(matchedKeypoints):

if m.distance < 0.75 \* n.distance:

goodMatches.append(m)

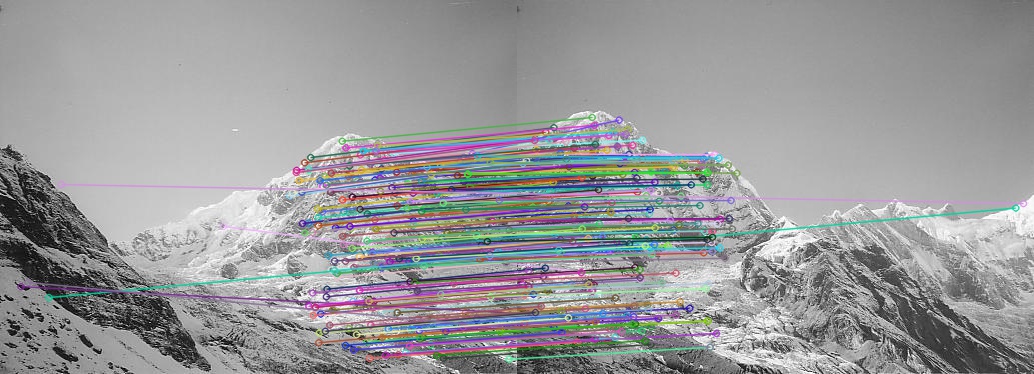
return goodMatches

## Output

******



***Figure 1.1:- Keypoints detected using SIFT***

******

***Figure 1.2:- Matched Keypoints using KNN’s***

***Homography matrix H before applying fix for pixel loss issue is:***

***[[ 1.58930420e+00 -2.91559554e-01 -3.95969728e+02]***

***[ 4.49424702e-01 1.43111081e+00 -1.90614367e+02]***

***[ 1.21265326e-03 -6.28716869e-05 1.00000000e+00]]***

***Homography matrix H after applying fix for pixel loss issue is:***

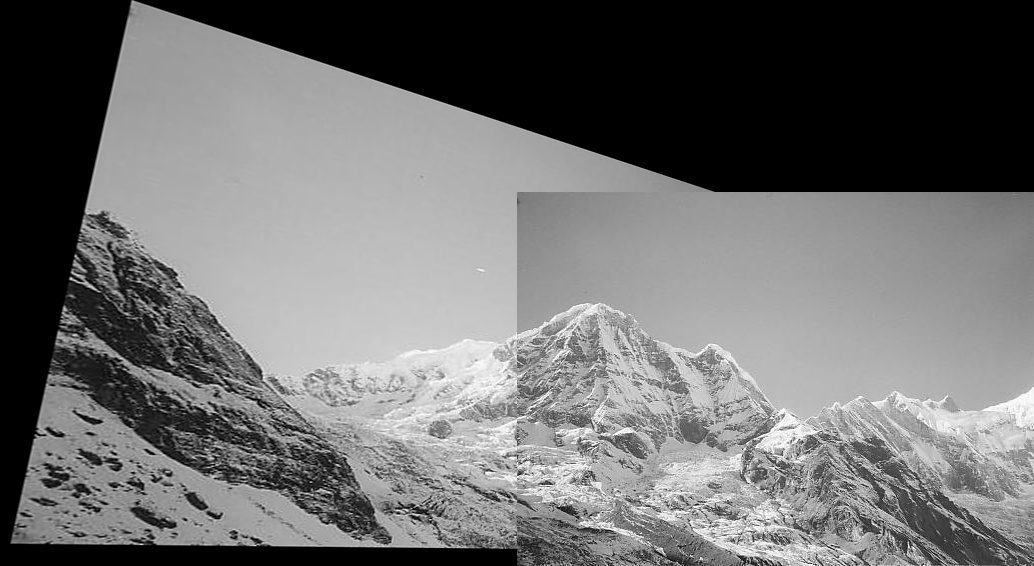
***[[ 2.21745859e+00 -3.24127088e-01 1.22030272e+02]***

***[ 6.81041475e-01 1.41910232e+00 3.85632623e-01]***

***[ 1.21265326e-03 -6.28716869e-05 1.00000000e+00]]***



***Figure 1.3: - Displaying randomly selected 10 inliers***



***Figure 1.3:- Final stitched image***

## Functions Created and their Discriptions

1] generateKeypoints (image1,image2): - This function is used to generate keypoints and its descriptors.

2] generateMatches (descriptor1,descriptor2):- This function is used to generate matches and filter good matches as per Lowe’s ratio test

# Task 2: - Epipolar Geometry

This task involved drawing epilines and disparity map

Steps:-

1] Repeat steps 1 and 2 from task1.

2] Compute Fundamental matrix using RANSAC.

3] compute epilines for 10 random inlier pairs

4] Computer Disparity map.

## Code

import numpy as np;

import cv2

from matplotlib import pyplot as plt

from functions import generateMatches, generateKeypoints, drawLines

import random

# Generate Fundamental matrix

fundamentalMatrix, inOut = cv2.findFundamentalMat(sourcePoints,destinationPoints,cv2.RANSAC)

print("Fundamental matrix: \n",fundamentalMatrix)

# Selecting only inliers

sourcePoints = sourcePoints[inOut.ravel()==1]

destinationPoints = destinationPoints[inOut.ravel()==1]

# Selecting 10 random points

sourcePointsSmall = []

destinationPointsSmall = []

for i in range(10):

index = random.randint(0,min(len(sourcePoints)-1,len(destinationPoints)-1))

sourcePointsSmall.append(sourcePoints[index])

destinationPointsSmall.append(destinationPoints[index])

sourcePointsSmall = np.asarray(sourcePointsSmall)

destinationPointsSmall = np.asarray(destinationPointsSmall)

# Find epilines corresponding to points in right image (second image) and

# drawing its lines on left image

lines1 = cv2.computeCorrespondEpilines(destinationPointsSmall.reshape(-1,1,2), 2,fundamentalMatrix)

lines1 = lines1.reshape(-1,3)

epiRight,img6 = drawLines(image1G,image2G,lines1,sourcePointsSmall,destinationPointsSmall)

# Find epilines corresponding to points in left image (first image) and

# drawing its lines on right image

lines2 = cv2.computeCorrespondEpilines(sourcePointsSmall.reshape(-1,1,2), 1,fundamentalMatrix)

lines2 = lines2.reshape(-1,3)

epiLeft,img4 = drawLines(image2G,image1G,lines2,destinationPointsSmall,sourcePointsSmall)

cv2.imwrite("task2\_epi\_right.jpg",epiRight)

cv2.imwrite("task2\_epi\_left.jpg",epiLeft)

# Task 1.5

stereo = cv2.StereoSGBM\_create(minDisparity=-20, numDisparities = 80,blockSize=17)

disparities = stereo.compute(image1G,image2G)

plt.imshow(disparities,'gray')

plt.imsave("task2\_disparity.png",disparities,cmap ='gray')

plt.show()

def drawLines(img1, img2, lines, sourcePoints, destinationPoints):

r, c = img1.shape

# This is done so that we can show color lines and points on the image

# creating list of colors

colors = ((109, 193, 120),

(149, 50, 148),

(196, 239, 3),

(92, 195, 53),

(28, 64, 91),

(127, 92, 87),

(90, 67, 49),

(81, 74, 60),

(128, 26, 209),

(147, 36, 145))

img1 = cv2.cvtColor(img1, cv2.COLOR\_GRAY2BGR)

img2 = cv2.cvtColor(img2, cv2.COLOR\_GRAY2BGR)

colorIndex = 0

for r, sourcePoints, destinationPoints in zip(lines, sourcePoints, destinationPoints):

# to draw line we need two points hence we calculate X0,Y0 and x1,y1

color = colors[colorIndex]

x0, y0 = map(int, [0, -r[2] / r[1]])

x1, y1 = map(int, [c, -(r[2] + r[0] \* c) / r[1]])

# Draw the line with the above points

img1 = cv2.line(img1, (x0, y0), (x1, y1), color, 1)

img1 = cv2.circle(img1, tuple(sourcePoints), 5, color, -1)

img2 = cv2.circle(img2, tuple(destinationPoints), 5, color, -1)

colorIndex += 1

return img1, img2

## Output

***Figure 2.1:- Keypoints detected using SIFT***



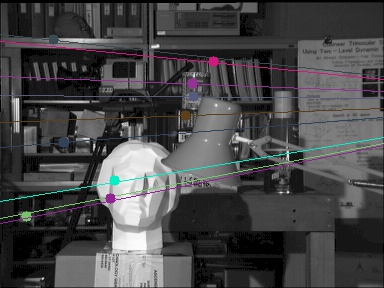
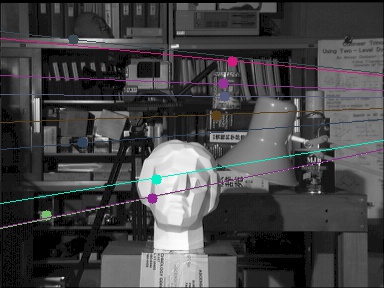
***Figure 2.2:- Matched Keypoints using KNN’s***

***Fundamental matrix:***

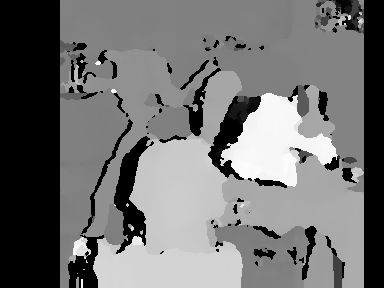
***[[-6.15358364e-07 -6.03703407e-04 6.05696120e-02]***

***[ 5.98543612e-04 -4.67460431e-05 -3.72753109e-01]***

***[-5.98189306e-02 3.68341287e-01 1.00000000e+00]]***

*** ***

***Figure 2.3:- Epilines***

******

***Figure 2.4:- Disparity Map***

## Functions Created and their Discriptions

Same as task 1

1] drawLines(img1, img2, lines, sourcePoints, destinationPoints): - Used to draw epi lines

# Task 3: - K-means Clustering

In this involved implementation of K-Means clustering and color quantization.

Since for color quantization task number of iterations are becoming huge I have added best result achieved by running program multiple times   
Steps:-

1] For given cluster centers classify the provided samples

2] Re compute new cluster centers

3] classify as per new cluster centers and again perform 1 more iteration

4] Apply K-means to perform color quantization

## Code

clusterCenters = [[6.2, 3.2], [6.6, 3.7], [6.5, 3.0]]

X = [[5.9, 3.2], [4.6, 2.9], [6.2, 2.8], [4.7, 3.2], [5.5, 4.2], [5.0, 3.0], [4.9, 3.1],[6.7, 3.1],[5.1, 3.8], [6.0, 3.0]]

XDistance = []

# Calculate the distances of points from clusters

XDistance = calculateDistance(clusterCenters, X, XDistance)

XClassified = []

print(len(XDistance), XDistance)

# Classify the points based on distance calculated earlier

XClassified = chooseCenteroid(XDistance,XClassified)

print(XClassified)

# Plot the cluesters

plotClusters(X, clusterCenters, XClassified, 'task3\_iter1\_a.png')

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Task 3.2:- Calculate new cluster centroids \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Calculate new centroids

clusterCenters = caculateNewMean(X,XClassified)

plotClusters(X, clusterCenters, XClassified,'task3\_iter1\_b.png')

XDistance = []

XDistance = calculateDistance(clusterCenters, X, XDistance)

XClassified = []

XClassified = chooseCenteroid(XDistance,XClassified)

print(XClassified)

# Classify the points based on distance calculated earlier

XClassified = chooseCenteroid(XDistance,XClassified)

plotClusters(X, clusterCenters, XClassified,'task3\_iter2\_a.png')

# \*\*\*\*\*\*\*\*\*\*\*\*\* Task 3.3 :- calculate euclidian disctances for new mean and generate clusters \*\*\*\*\*\*\*\*\*\*\*\*

# Calculate new centroids

clusterCenters = caculateNewMean(X,XClassified)

XDistance = []

# Calculate the distances of points from clusters

XDistance = calculateDistance(clusterCenters, X, XDistance)

XClassified = []

# Classify the points based on distance calculated earlier

XClassified = chooseCenteroid(XDistance,XClassified)

print(XClassified)

plotClusters(X,clusterCenters,XClassified,'task3\_iter2\_b.png')

# Task 3.4

kmeanFull()

def kmeanFull():

image = cv2.imread("Images/baboon.jpg")

imgHeight, imgWidth = image.shape[:2]

image = np.asarray(image,dtype=np.float64) / 255

noOfClusters = 3

clusterCenters = []

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Randomly generate cluster centers \*\*\*\*\*\*\*\*\*\*\*\*\*\*

for times in range(noOfClusters):

imageX = random.randint(0, 511)

imageY = random.randint(0, 511)

print(imageX, imageY)

clusterCenters.append(image[imageX][imageY])

# to get rid of the dtype info

clusterCenters = np.array(clusterCenters, dtype=np.float64) / 255

prevClusterCenters = np.asarray(clusterCenters)

print("cluster Centers \n", clusterCenters)

image = image.reshape((image.shape[0] \* image.shape[1], 3))

print("image Shape ", image.shape)

converge = False

iteration = 0

while (converge == False):

print("\*\*\*\*\*\*\*\*\*\*\* Iteration",iteration," \*\*\*\*\*\*\*\*\*\*\*")

iteration +=1

# \*\*\*\*\*\*\*\*\*\*\*\*\*\* Calculating euclidean distances \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

colorDistance = calculateColorDistance(noOfClusters, image, clusterCenters)

# print("colorDistance \n",colorDistance)

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Classify the colors to clusters \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ptsClassified = np.zeros((image.shape[0],noOfClusters))

ptsClassified = findCluster(colorDistance, ptsClassified, noOfClusters)

print("cluster element numbers :",np.sum(ptsClassified,axis=0))

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Calculate new Cluster centers \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

clusterCenters = reCalculateMean(ptsClassified, colorDistance, noOfClusters, clusterCenters)

converge = np.array\_equal(prevClusterCenters,clusterCenters)

print("New clusters: \n",clusterCenters)

if iteration == 15:

converge = True

for imgIndex in range(image.shape[0]):

for centers in range(noOfClusters):

if ptsClassified[imgIndex][centers] == 1:

image[imgIndex] = clusterCenters[centers]

print(image)

image = image.reshape((imgHeight,imgWidth,3))

colorDistance = colorDistance.reshape((imgHeight,imgWidth,noOfClusters))

image = image \* 255

image = image.astype(np.uint8)

colorDistance = colorDistance \* 255

colorDistance = colorDistance.astype(np.uint8)

cv2.imwrite("task3\_baboon\_5.jpg", image)

**Functions.py**

clusterCentersColor = ['Red', 'Green', 'Blue']

def calculateDistance(clusterCenters,X,XDistance):

for dataPts in range(len(X)):

XDistanceTemp = []

for centers in range(len(clusterCenters)):

# Center points are X0,Y0 and Datapoints are X1,Y1

d1 = ((clusterCenters[centers][0] - X[dataPts][0])\*\*2)

d2 = ((clusterCenters[centers][1] - X[dataPts][1])\*\*2)

distance = (d1 + d2)\*\*0.5

XDistanceTemp.append(distance)

XDistance.append(XDistanceTemp)

return XDistance

def chooseCenteroid(XDistance,XClassified):

for distances in range(len(XDistance)):

minValIndexLoc = XDistance[distances].index(min(XDistance[distances]))

if minValIndexLoc == 0: # belongs to cluster 1

XClassified.append('Red')

elif minValIndexLoc == 1: # belongs to cluster 2

XClassified.append('Green')

elif minValIndexLoc == 2: # belongs to cluster 3

XClassified.append('Blue')

return XClassified

def caculateNewMean(X,XClassified):

sumRedX = 0

sumRedY = 0

countRed = 0

sumBlueX = 0

sumBlueY = 0

countBlue = 0

sumGreenX = 0

sumGreenY = 0

countGreen = 0

for index in range(len(X)):

if XClassified[index] == 'Red': # belongs to cluster 1

sumRedX += X[index][0]

sumRedY += X[index][1]

countRed += 1

elif XClassified[index] == 'Blue': # belongs to cluster 2

sumBlueX += X[index][0]

sumBlueY += X[index][1]

countBlue += 1

elif XClassified[index] == 'Green': # belongs to cluster 3

sumGreenX += X[index][0]

sumGreenY += X[index][1]

countGreen += 1

newCenters = [[sumRedX/countRed, sumRedY/countRed],

[sumGreenX / countGreen, sumGreenY / countGreen],

[sumBlueX / countBlue, sumBlueY / countBlue]]

return newCenters

def plotClusters(X,clusterCenters,XClassified,fineName):

# Extract X & Y co-ordinates

arrayX = np.asarray(X)

arrayClusterCenters = np.asarray(clusterCenters)

xX = arrayX[:, :1].tolist()

yX = arrayX[:, 1:].tolist()

centerx = arrayClusterCenters[:, :1].tolist()

centery = arrayClusterCenters[:, 1:].tolist()

plt.scatter(centerx, centery, color=clusterCentersColor, marker="o")

for i in range(len(clusterCenters)):

plt.text(centerx[i][0] + 0.005, centery[i][0], s='(' + str(centerx[i][0]) + ',' + str(centery[i][0]) + ')')

for i in range(len(X)):

plt.scatter(xX[i], yX[i], color=XClassified[i], marker="^")

plt.text(xX[i][0] + 0.005, yX[i][0], s='(' + str(xX[i][0]) + ',' + str(yX[i][0]) + ')')

# plt.show()

plt.savefig(fineName)

plt.clf()

def calculateColorDistance(noOfClusters, image, clusterCenters):

colorDistance = np.zeros((image.shape[0], noOfClusters))

print("in function")

print(image.shape[0])

for imgX in range(image.shape[0]):

for centers in range(noOfClusters):

x = image[imgX]

y = clusterCenters[centers]

distance = np.linalg.norm(x-y)

colorDistance[imgX][centers] = distance

return colorDistance

def findCluster(colorDistance, ptsClassified, noOfClusters):

colorDistancecolumns = []

for imgX in range(colorDistance.shape[0]):

for centers in range(noOfClusters):

colorDistancecolumns.append(colorDistance[imgX][centers])

minValIndexLoc = colorDistancecolumns.index(min(colorDistancecolumns))

ptsClassified [imgX][minValIndexLoc] = 1

colorDistancecolumns = []

return ptsClassified

def reCalculateMean(ptsClassified, colorDistance, noOfClusters, clusterCenters):

oldClusterCenters = clusterCenters

clusterCenters = []

newMean = []

print(type(newMean))

clusterElementNo = np.sum(ptsClassified, axis=0)

for cluster in range(noOfClusters):

for imgX in range(colorDistance.shape[0]):

if ptsClassified[imgX][cluster] == 1:

# print(colorDistance[imgX])

newMean.append(colorDistance[imgX])

if not newMean:

newMean.append(oldClusterCenters[cluster])

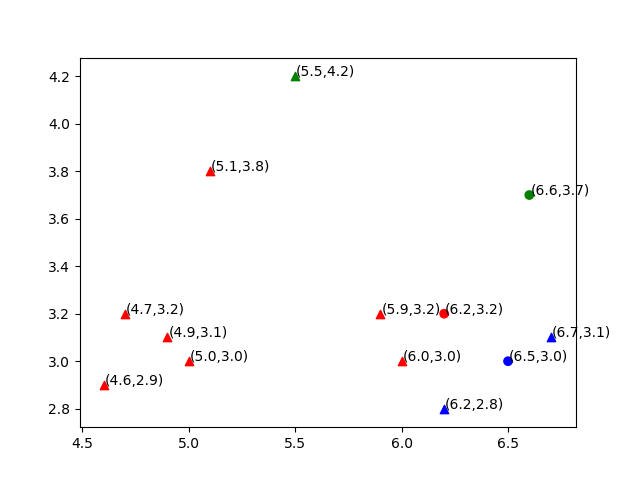
mean = np.mean(newMean,axis = 0)

clusterCenters.append(mean)

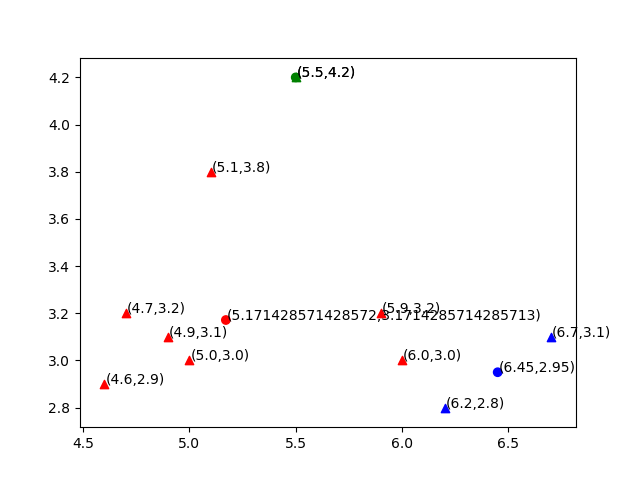
newMean = []

return clusterCenters

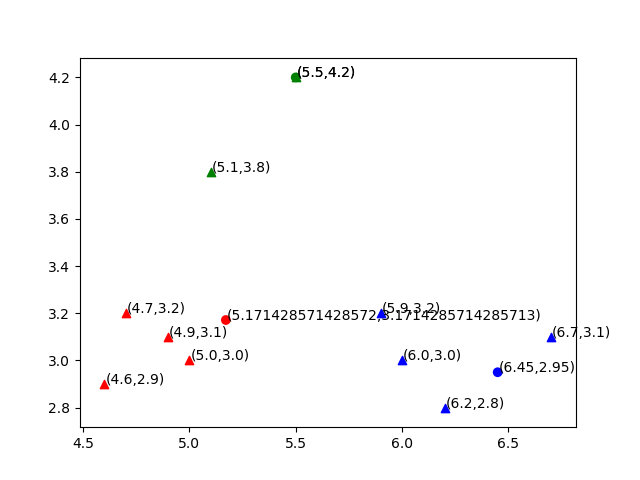
## Code SET A and Bonus part



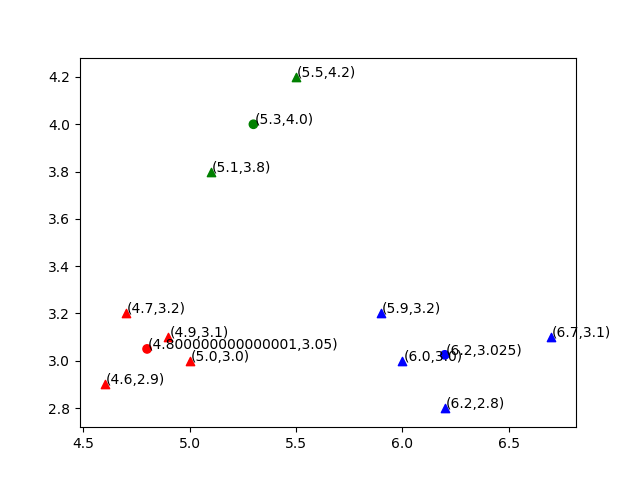
***Figure 3.1:- classified points as per given clusters***

******

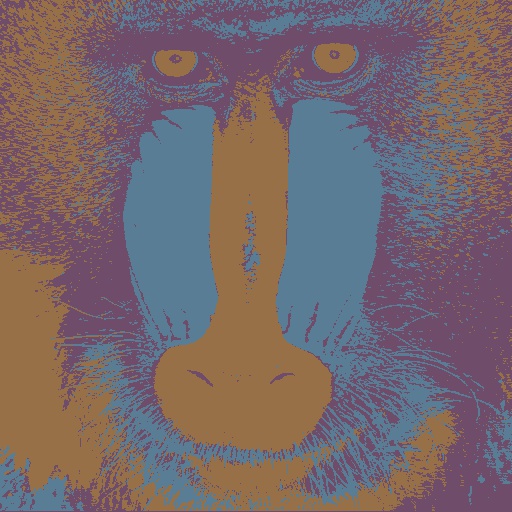
***Figure 3.2:- Recomputed cluster centers***

******

***Figure 3.3:- Reclassify as per new cluster centers***

******

***Figure 3.4:- Recompute cluster centers***

******

***Figure 3.5:- Color Quantization with 3 clusters.***