|  |
| --- |
| **Homography and Epipolar Geometry** |

**Dhayanidhi Gunasekaran**

University at Buffalo

*dhayanid@buffalo.edu*

**Abstract**

To perform the image feature matching tasking such as Homography, Epipolar geometry and K-Means Clustering for the given images.

**1 Image Features and Homography**

Homography is explained as the concept of relating any two images which are in the same planar surface in the space.

**1.1 Task Objective**

To detect the features matches in the given images and to draw the matched inliers and warp the two images over one another based on the homography matrix.

|  |  |
| --- | --- |
|  |  |

Figure 1: Input Images for Homography

**1.2 Procedure**

The implementation is done using python, opencv and numpy libraries.

1. Images are read using imread function.
2. Image features are extracted using SIFT feature extraction method.
3. The keypoint features extracted from both the input images are compared and the one with greater match are used for homography.
4. The keypoint matching is done by FlannBasedMatcher.
5. Homography matrix is computed.
6. Randomly 10 inliers are generated and plotted in the original images
7. Image 1 are warped on to the left image so that, it will result in a panaroma image.

**1.3 Code**

UBIT = "dhayanid";

import numpy as np;

np.random.seed(sum([ord(c) for c in UBIT]))

import cv2 as cv

import matplotlib as plt

# Read input images

mount2\_color = cv.imread("data/mountain2.jpg")

mount2 = cv.imread("data/mountain2.jpg", cv.IMREAD\_GRAYSCALE)

mount1\_color = cv.imread("data/mountain1.jpg")

mount1 = cv.imread("data/mountain1.jpg", cv.IMREAD\_GRAYSCALE)

# extracting sift features

sift = cv.xfeatures2d.SIFT\_create()

key1, desc1 = sift.detectAndCompute(mount1, None)

key2, desc2 = sift.detectAndCompute(mount2, None)

# drawing the extracted key points in input image

task1\_sift1 = cv.drawKeypoints(mount1\_color, key1, mount1)

task1\_sift2 = cv.drawKeypoints(mount2\_color, key2, mount2)

# writing the sift image

cv.imwrite("task1\_sift1.jpg", task1\_sift1)

cv.imwrite("task1\_sift2.jpg", task1\_sift2)

# feature matching

# flann based matcher is used to get the matches betweent the keypoints of two images with k=2

index = dict(algorithm=0, trees=5)

search = dict()

flann = cv.FlannBasedMatcher(index, search)

feature\_match = flann.knnMatch(desc1, desc2, k=2)

key\_point\_match = []

# this loop gets all the good matches which satisfies the given condition

for x, y in feature\_match:

if x.distance < 0.75 \* y.distance:

key\_point\_match.append(x)

# this line randomly selects 10 good matches and return it to the variable

#random\_key\_point = np.random.choice(key\_point\_match, 10)

task1\_matches\_knn = cv.drawMatches(

mount1\_color, key1, mount2\_color, key2, key\_point\_match, mount2)

# random points are drawn and printed in a output file

cv.imwrite("task1\_matches\_knn.jpg", task1\_matches\_knn)

# Homography matrix computation

if len(key\_point\_match) > 10:

src\_pts = np.float32(

[key1[m.queryIdx].pt for m in key\_point\_match]).reshape(-1, 1, 2)

target\_pts = np.float32(

[key2[m.trainIdx].pt for m in key\_point\_match]).reshape(-1, 1, 2)

Matrix, mask = cv.findHomography(src\_pts, target\_pts, cv.RANSAC, 5.0)

print("Homography matrix")

print(Matrix)

# convert the mask to a list

inliers = mask.ravel().tolist()

# get random 10 values from the inliers/matches

random\_inliers = np.random.choice(inliers, 10)

random\_key\_point2 = np.random.choice(key\_point\_match, 10)

height, width = mount2.shape

draw\_parameters = dict(matchColor=(

255, 0, 0), singlePointColor=None, matchesMask=random\_inliers.tolist(), flags=2)

# draw the matched parameters with respect to both th eimages

task1\_matches = cv.drawMatches(

mount1\_color, key1, mount2\_color, key2, random\_key\_point2, None, \*\*draw\_parameters)

cv.imwrite("task1\_matches.jpg", task1\_matches)

M = np.float32([[1,0,517],[0,1,374],[0,0,1]])

dest = cv.warpPerspective(mount1\_color,np.matmul(M,Matrix),(mount2\_color.shape[1] + mount1\_color.shape[1],mount1\_color.shape[0] +mount2\_color.shape[0]))

dest[374:748, 517:1034] = mount2\_color

cv.imwrite("task1\_pano.jpg", dest)

**1.4 Output Images**

**1.4.1 SIFT Features**

|  |  |
| --- | --- |
|  |  |

Figure 2: SIFT Features of the Input Images.

**1.4.2 Feature match using KNN (inliers and outliers)**



Figure 3: Feature match using KNN

**1.4.3 Homography matrix**

The computed Homography matrix is as follows.

[[ 1.58930258e+00 -2.91559627e-01 -3.95969243e+02]

[ 4.49424370e-01 1.43110804e+00 -1.90613924e+02]

[ 1.21265246e-03 -6.28766581e-05 1.00000000e+00]]

**1.4.4 Random 10 inlier Matches**

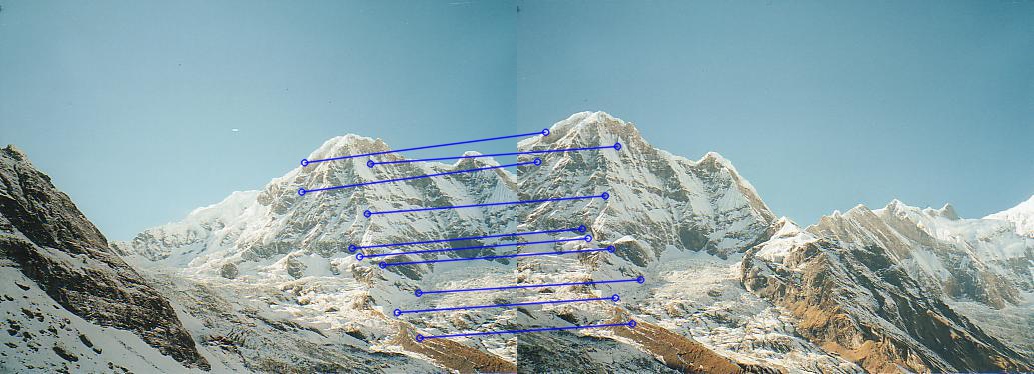


Figure 4: Feature match using KNN

**1.4.5 Panaroma Image**



Figure 5: Panaroma Image

**2 Epipolar Geometry**

Epipolar geometry is a concept in stereo vision, in which two cameras view a same 3D image from two distinct positions. The geometric relation between the images and their projections leads to constraints between the image points.

**2.1 Task Objective**

To extract the SIFT features and compute the fundamental Matrix. To select 10 random match pairs and to draw the line on both the images. Match points obtained from image 1 are to be drawn in image 2 and vice versa. Finally, to compute the disparity image between the left and right image.

|  |  |
| --- | --- |
|  |  |

Figure 6: Input images for Task 2

**2.2 Code**

UBIT = "dhayanid";

import numpy as np;

np.random.seed(sum([ord(c) for c in UBIT]))

import cv2 as cv

from matplotlib import pyplot as plt

def get\_Max(matrix):

largest\_num = matrix[0][0]

for row\_idx, row in enumerate(matrix):

for col\_idx, num in enumerate(row):

if num > largest\_num:

largest\_num = num

return largest\_num

def Normalise\_Matrix(Matrix):

MAX\_VALUE = get\_Max(Matrix)

row = len(Matrix)

col = len(Matrix[0])

for i in range(row):

for j in range(col):

Matrix[i][j] = (Matrix[i][j]/MAX\_VALUE)\*255

return Matrix

#this funciton returns the 10 elements out of the given array

def gettenelem(random\_array):

count = 0

random\_ten = []

for i in range(len(random\_array)):

count+=1

if count<10:

random\_ten.append(random\_array[i])

return np.array(random\_ten)

#this function draws the set of lines in the given color image

#color\_count is used to show the same color for the same point pair in left and right image

def draw\_epiline(img1,lines,pts1):

row,col,d = img1.shape

color\_count = 0

for row,pt1 in zip(lines,pts1):

color\_count += 17

B=color\_count

G=color\_count + 100

R=color\_count +31

color\_list= [B,G,R]

color = tuple(color\_list)

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

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

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

img1 = cv.circle(img1,tuple(pt1.flatten()),5,color,-1)

return img1

# Read input images

left\_img\_color = cv.imread("data/tsucuba\_left.png")

left\_img = cv.imread("data/tsucuba\_left.png", cv.IMREAD\_GRAYSCALE)

right\_img\_color = cv.imread("data/tsucuba\_right.png")

right\_img = cv.imread("data/tsucuba\_right.png", cv.IMREAD\_GRAYSCALE)

# extracting sift features

sift = cv.xfeatures2d.SIFT\_create()

key1, desc1 = sift.detectAndCompute(left\_img, None)

key2, desc2 = sift.detectAndCompute(right\_img, None)

# drawing the extracted key points in input image

task2\_sift1 = cv.drawKeypoints(left\_img\_color, key1, left\_img)

task2\_sift2 = cv.drawKeypoints(right\_img\_color, key2, right\_img)

# writing the sift image

cv.imwrite("task2\_sift1.jpg", task2\_sift1)

cv.imwrite("task2\_sift2.jpg", task2\_sift2)

# feature matching

# flann based matcher is used to get the matches betweent the keypoints of two images with k=2

index = dict(algorithm=0, trees=5)

search = dict()

flann = cv.FlannBasedMatcher(index, search)

feature\_match = flann.knnMatch(desc1, desc2, k=2)

key\_point\_match = []

# this loop gets all the good matches which satisfies the given condition

for x, y in feature\_match:

if x.distance < 0.75 \* y.distance:

key\_point\_match.append(x)

# this line randomly selects 10 good matches and return it to the variable

#random\_key\_point = np.random.choice(key\_point\_match, 10)

task2\_matches\_knn = cv.drawMatches(

left\_img\_color, key1, right\_img\_color, key2, key\_point\_match, right\_img)

# random points are drawn and printed in a output file

cv.imwrite("task2\_matches\_knn.jpg", task2\_matches\_knn)

src\_pts = np.float32([key1[m.queryIdx].pt for m in key\_point\_match]).reshape(-1,1,2)

target\_pts = np.float32([key2[m.trainIdx].pt for m in key\_point\_match]).reshape(-1,1,2)

#fundamental matrix computation

#pts1 = np.int32(src\_pts)

#pts2 = np.int32(target\_pts)

pts1 = src\_pts

pts2 = target\_pts

F\_Matrix, mask = cv.findFundamentalMat(pts1, pts2, cv.RANSAC,5.0)

print("Fundamental Matrix")

print(F\_Matrix)

# We select only inlier points

pts1 = pts1[mask.ravel()==1]

pts2 = pts2[mask.ravel()==1]

#get random 10 points in the list and compute the line for each points in pts2

# draw the computed epilines over the left image

random\_ten\_pts2 = gettenelem(pts2)

lines1 = cv.computeCorrespondEpilines(np.array(random\_ten\_pts2), 2,F\_Matrix).reshape(-1,3)

epi\_left = draw\_epiline(left\_img\_color,lines1,pts1)

#get random 10 points in the list and compute the line for each points pts1

# draw the computed epilines over the right image

random\_ten\_pts1 = gettenelem(pts1)

lines2 = cv.computeCorrespondEpilines(np.array(random\_ten\_pts1), 1,F\_Matrix).reshape(-1,3)

epi\_right = draw\_epiline(right\_img\_color,lines2,pts2)

cv.imwrite("task2\_epi\_left.jpg",epi\_left)

cv.imwrite("task2\_epi\_right.jpg",epi\_right)

stereo = cv.StereoBM\_create(numDisparities=80, blockSize=25)

disparity = stereo.compute(left\_img,right\_img)

disparity = Normalise\_Matrix(disparity)

cv.imwrite("task2\_disparity.jpg",disparity)

**2.3 Output**

**2.3.1 SIFT and KNN matched features**

|  |  |
| --- | --- |
|  |  |

Figure 7: SIFT Features

****

Figure 8: KNN matches

**2.3.2 Fundamental Matrix**

The computed Fundamental Matrix is as follows.

[[ 2.54494551e-06 1.55828977e-05 2.20461309e-02]

[ 1.45536087e-05 1.49451509e-05 2.65976425e-01]

[-2.54111087e-02 -2.72823100e-01 1.00000000e+00]]

**2.3.3 Epilines**

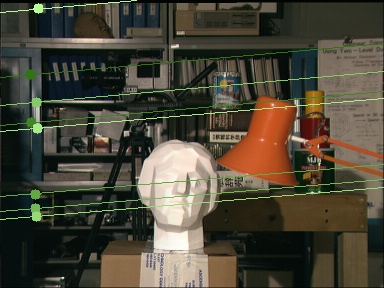
****

Figure 9: Epilines on Left image

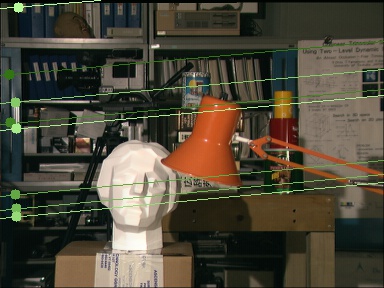
****

Figure 10: Epilines on Right image

**2.3.4 Disparity**

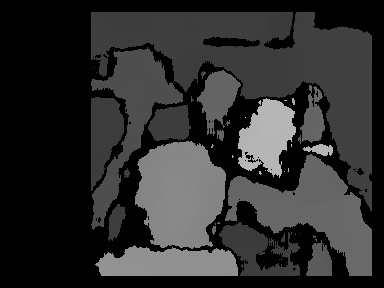
****

Figure 11: Disparity image

**3 K-Means Clustering**

K-Means clustering aims to group all the data points based on the distance between the clusters. Each data point is computed with a Euclidean distance between all the available clusters and aligned with the shortest distance. And the clusters are then computed with the average of all the points aligned to it and recentered. This process is repeated until the center r of the clusters remain unchanged.

**3.1 Task Objective**

Given an matrix with 10 points and 3 centers. To perform K means clustering for all the data points and align the points to each center and compute the coordinates for the centre. Plot the same. Also given with an image for performing color quantization clustering for the centers 3,5,10,20.

**3.2 Code**

UBIT = "dhayanid"

import numpy as np

np.random.seed(sum([ord(c) for c in UBIT]))

import math

from matplotlib import pyplot as plt

import cv2 as cv

def computeEuclideanDist(a, b):

x1 = a[0]

y1 = a[1]

x2 = b[0]

y2 = b[1]

distance = math.sqrt(((x1-x2)\*\*2)+((y1-y2)\*\*2))

return round(distance, 3)

# this function returns the classification vector

def computedistanceAndClassify(x,red,green,blue):

class\_vector = []

for i in range(len(x)):

lista = []

red\_dist = computeEuclideanDist(x[i], red)

green\_dist = computeEuclideanDist(x[i], green)

blue\_dist = computeEuclideanDist(x[i], blue)

lista.append(red\_dist)

lista.append(green\_dist)

lista.append(blue\_dist)

min\_value = min(lista)

if min\_value == red\_dist:

class\_vector.append('r')

elif min\_value == blue\_dist:

class\_vector.append('b')

elif min\_value == green\_dist:

class\_vector.append('g')

return class\_vector

# calculate the average x and y value ofall the points in the cluster

def computeNewCentroid(x,code):

sum\_x = 0

sum\_y = 0

count = 0

for a in x:

if a[2] == code:

count += 1

sum\_x += a[0]

sum\_y += a[1]

avg\_x = sum\_x/count

avg\_y = sum\_y/count

return [avg\_x,avg\_y]

# center locations

red = [6.2, 3.2]

green = [6.6, 3.7]

blue = [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]]

cvectr = computedistanceAndClassify(x,red,green,blue)

print("classification vector")

print(cvectr)

for i in range(len(cvectr)):

x[i].append(cvectr[i])

plt.figure()

for point in x:

plt.scatter(point[0], point[1], c=point[2], marker='^')

plt.scatter(6.2, 3.2, c='r', marker='o')

plt.scatter(6.6, 3.7, c='g', marker='o')

plt.scatter(6.5, 3.0, c='b', marker='o')

plt.savefig("task3\_iter1\_a.jpg")

red\_new\_centroid = computeNewCentroid(x,'r')

blue\_new\_centroid = computeNewCentroid(x,'b')

green\_new\_centroid = computeNewCentroid(x,'g')

plt.figure()

for point in x:

plt.scatter(point[0], point[1], c=point[2], marker='^')

plt.scatter(red\_new\_centroid[0],red\_new\_centroid[1], c='r', marker='o')

plt.scatter(blue\_new\_centroid[0],blue\_new\_centroid[1], c='b', marker='o')

plt.scatter(green\_new\_centroid[0],green\_new\_centroid[1], c='g', marker='o')

plt.savefig("task3\_iter1\_b.jpg")

plt.figure()

cvectr1 = computedistanceAndClassify(x,red\_new\_centroid,green\_new\_centroid,blue\_new\_centroid)

print("classification vector after iteration 1")

print(cvectr1)

x1= []

for i in range(len(cvectr1)):

arr = []

arr.append(x[i][0])

arr.append(x[i][1])

arr.append(cvectr1[i])

x1.append(arr)

plt.figure()

for point in x1:

plt.scatter(point[0], point[1], c=point[2], marker='^')

plt.scatter(red\_new\_centroid[0],red\_new\_centroid[1], c='r', marker='o')

plt.scatter(blue\_new\_centroid[0],blue\_new\_centroid[1], c='b', marker='o')

plt.scatter(green\_new\_centroid[0],green\_new\_centroid[1], c='g', marker='o')

plt.savefig("task3\_iter2\_a.jpg")

red\_new\_centroid\_1 = computeNewCentroid(x1,'r')

blue\_new\_centroid\_1 = computeNewCentroid(x1,'b')

green\_new\_centroid\_1 = computeNewCentroid(x1,'g')

plt.figure()

for point in x1:

plt.scatter(point[0], point[1], c=point[2], marker='^')

plt.scatter(red\_new\_centroid\_1[0],red\_new\_centroid\_1[1], c='r', marker='o')

plt.scatter(blue\_new\_centroid\_1[0],blue\_new\_centroid\_1[1], c='b', marker='o')

plt.scatter(green\_new\_centroid\_1[0],green\_new\_centroid\_1[1], c='g', marker='o')

plt.savefig("task3\_iter2\_b.jpg")

img = cv.imread("data/baboon.jpg")

print(img.shape)

# compute euclidean distance of 3d image

def computeEuclideanDist3d(a, b):

x1 = a[0]

y1 = a[1]

z1 = a[2]

x2 = b[0]

y2 = b[1]

z2 = b[2]

distance = math.sqrt(((x1-x2)\*\*2)+((y1-y2)\*\*2)+((z1-z2)\*\*2))

return round(distance, 1)

#find the minimum value of the list and returns the min value and the centroid

def find\_centroid(a):

temp\_list = []

for i in range(len(a)):

temp\_list.append(a[i][0])

min\_value = min(temp\_list)

for i in range(len(a)):

if min\_value == a[i][0]:

centroid = a[i][1]

return centroid

# this function returns the classification vector

def computedistanceAndClassifyGeneric(x,centroids\_Array):

class\_vector = []

lista = []

for center in centroids\_Array:

lista.append([computeEuclideanDist3d(x,center),center[3]])

centroid = find\_centroid(lista)

if len(x)<4:

x.append(centroid)

else:

x[3] = centroid

return x

# calculate the average x and y value ofall the points in the cluster

def computeNewCentroidGeneric(x,code):

sum\_x = 0

sum\_y = 0

sum\_z = 0

count = 0

for row in range(len(x)):

if x[row][3] == code:

count += 1

sum\_x += x[row][0]

sum\_y += x[row][1]

sum\_z += x[row][2]

avg\_x = round(sum\_x/count, 1)

avg\_y = round(sum\_y/count, 1)

avg\_z = round(sum\_z/count, 1)

return [avg\_x,avg\_y,avg\_z,code]

def getcentroid\_value(img\_coord,final\_centroid):

for i in final\_centroid:

if img\_coord[3] == i[3]:

arr=[]

arr.append(math.floor(i[0]))

arr.append(math.floor(i[1]))

arr.append(math.floor(i[2]))

return arr

def adjust\_centroid(centroid\_array,img\_wvector):

new\_centroid =[]

for x in centroid\_array:

new\_centroid.append(computeNewCentroidGeneric(img\_wvector,x[3]))

return new\_centroid

def classify(centroid\_array,img\_wvector):

for row in range(len(img\_wvector)):

img\_wvector[row]= computedistanceAndClassifyGeneric(img\_wvector[row],centroid\_array)

new\_centroid = adjust\_centroid(centroid\_array,img\_wvector)

print(new\_centroid)

if np.array\_equal(new\_centroid,centroid\_array) != True:

centroid\_array = new\_centroid

classify(centroid\_array,img\_wvector)

else:

print("centroid computed")

return centroid\_array,img\_wvector

def trainKmeans(img,k):

centroid\_array = []

img\_w = img

img\_wvector = []

for row in range(len(img\_w)):

for col in range(len(img\_w[0])):

arr = []

arr.append(img\_w[row][col][0])

arr.append(img\_w[row][col][1])

arr.append(img\_w[row][col][2])

#arr.append(random.randint(0,k-1))

img\_wvector.append(arr)

rand\_index = np.random.choice(len(img\_wvector), k)

for i in range(k):

arr = []

j = rand\_index[i]

for value in img\_wvector[j]:

arr.append(value)

arr.append(i)

centroid\_array.append(arr)

final\_centroid,final\_wvector = classify(centroid\_array,img\_wvector)

count =0

for row in range(len(img\_w)):

for col in range(len(img\_w[0])):

img\_w[row][col] = getcentroid\_value(final\_wvector[count],final\_centroid)

count +=1

return img\_w

task3\_baboon\_3=trainKmeans(img,3)

cv.imwrite("task3\_baboon\_3.jpg",task3\_baboon\_3)

task3\_baboon\_5=trainKmeans(img,5)

cv.imwrite("task3\_baboon\_5.jpg",task3\_baboon\_5)

task3\_baboon\_10=trainKmeans(img,10)

cv.imwrite("task3\_baboon\_10.jpg",task3\_baboon\_10)

task3\_baboon\_20=trainKmeans(img,20)

cv.imwrite("task3\_baboon\_20.jpg",task3\_baboon\_20)

**3.3 Output**

**3.3.1 Classification iteration1**

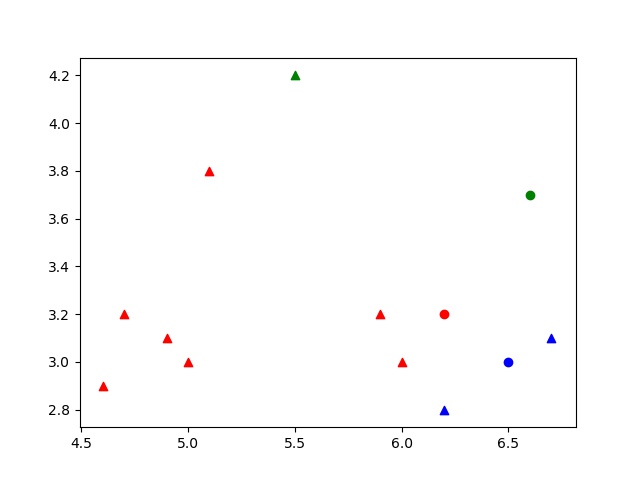
****

Figure 12: iteration 1-classificaiton

**3.3.2 Computing centers**

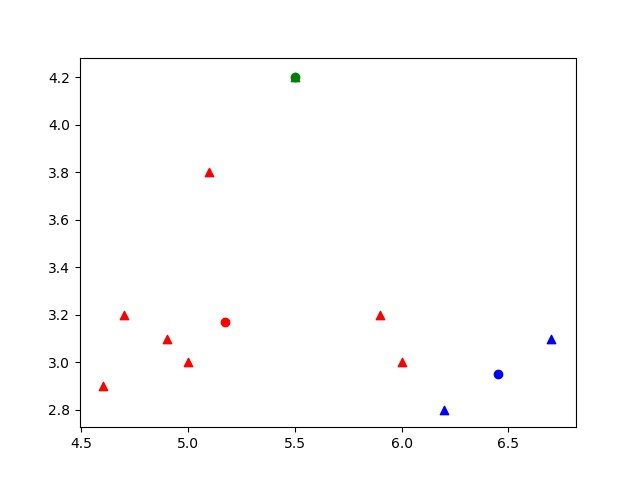
****

Figure 13: iteration 1- Computing centers

**3.3.3 Iteration 2**

|  |  |
| --- | --- |
|  |  |

Figure 14: iteration 2-Classification and Computing centers

**3.3.4 Color Quantization clustering**

****

Figure 15: Cluster 3



Figure 16: Cluster 5



Figure 17: Cluster 10



Figure 18: Cluster 20

**4 Conclusion**

All the three tasks implemented in the project provided the intuitive understanding of the Homography, Epipolar geometry and K - Means Clustering.

**References**

[1] Richard Szeliski (2010), Computer Vision: Algorithms and Applications

[2] <https://docs.opencv.org/2.4/doc/tutorials/>

[3] <https://docs.python.org/3/>