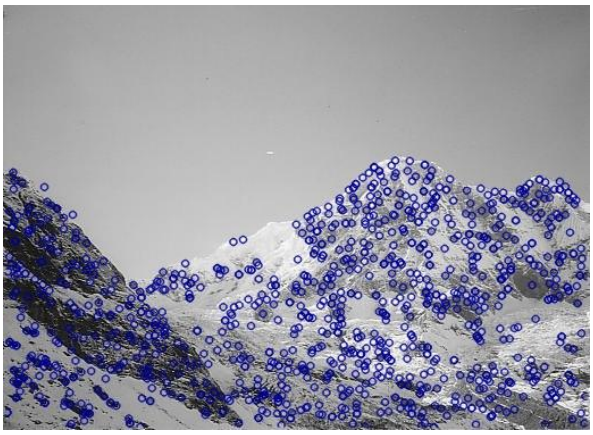
CVIP - Project 2

1 IMAGE FEATURES AND HOMOGRAPHY

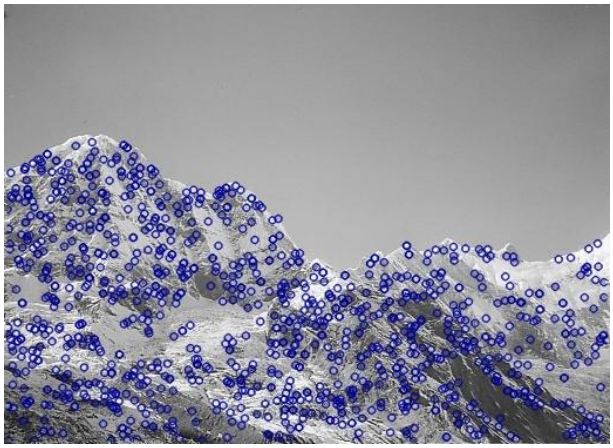
IN THIS TASK WE FIND THE IMAGE FEATURES AND TRY TO WARP THE TWO IMAGE WITH HOMOGRAPHY. ANY TWO IMAGES OF THE SAME PLANAR SURFACE IN SPACE ARE RELATED BY A HOMOGRAPHY

1.1

SIFT (Scale Invariant Feature Transform) is one of the widely used feature detector and discriptors. In this task we extract the SIFT features for the given two images. The SIFT function is called with the OpenCV libraries and source. After the detect and compute function call, the key-points and descriptors are stored. Those points are drawn on the image. The output is shown below



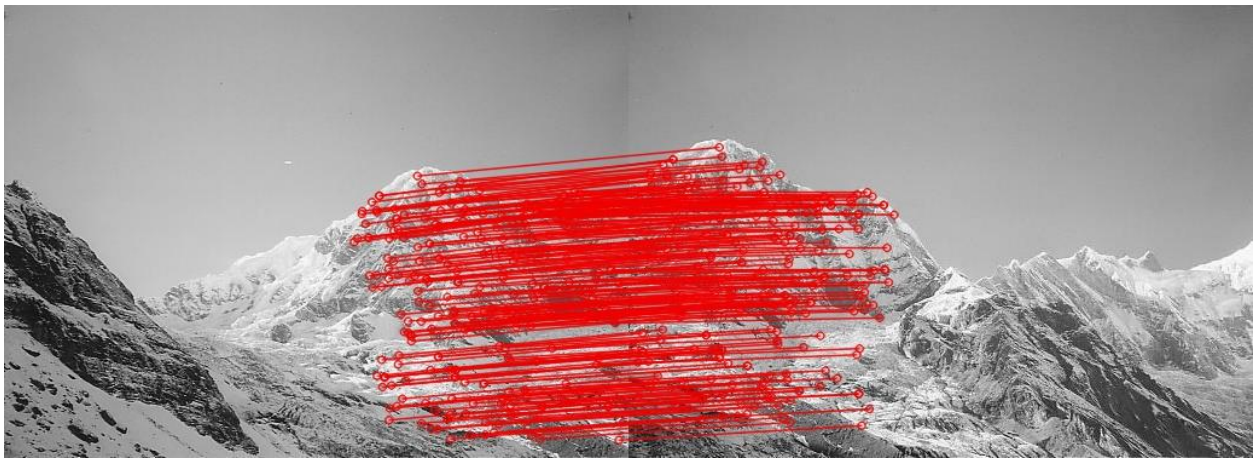
SIFT – Mountian1



SIFT – Mountain2

1.2 MATCH IMAGE

The keep points are matched with k means clustering of the both the images and the matching keep points are drawn.



Matching image with k-means

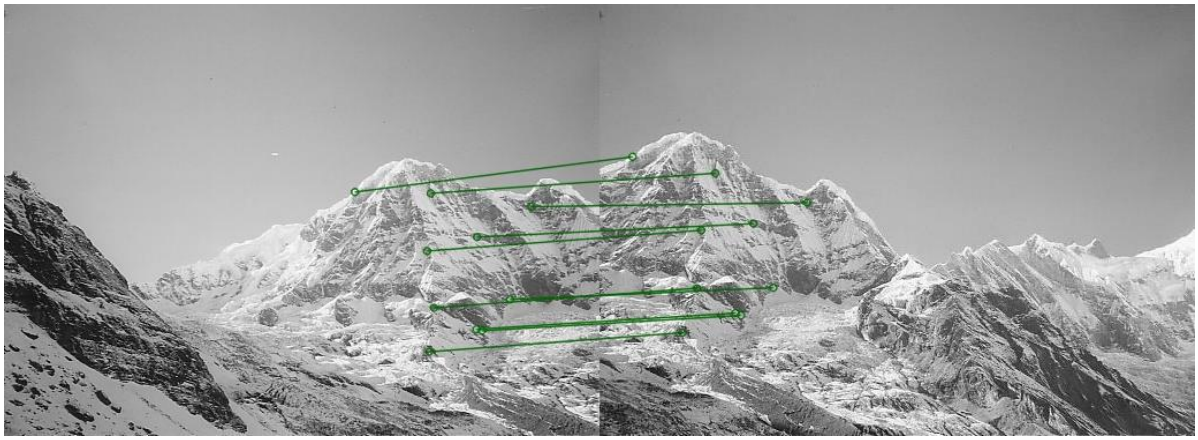
1.3 HOMOGRAPHY MATRIX

The homography matrix with first image to the second image.

```
[ 1.58930230e+00 -2.91559040e-01 -3.95969265e+02 ]
[ 4.49423930e-01 1.43110916e+00 -1.90613988e+02 ]
[ 1.21265043e-03 -6.28729364e-05 1.00000000e+00 ]
```

1.4 DRAW KEYPOINTS FOR RANDOM 10 INLINERS

Random 10 numbers are selected in the range of 255. Only those which match with the random 10 numbers already selected are matched and selected.



1.5 WARP IMAGES

'Warpperspective' function is used to warp the two images given.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
UBIT = 'jayanthk'
np.random.seed(sum([ord(c) for c in UBIT]))
import random
# Reading the images in grey scale
img1 = cv2.imread("D:\\sem1\\cvip\\proj2\\mountain1.jpg", 0)
img2 = cv2.imread("D:\\sem1\\cvip\\proj2\\mountain2.jpg", 0)
img1_color = cv2.imread("D:\\sem1\\cvip\\proj2\\mountain1.jpg", 1)
img2_color= cv2.imread("D:\\sem1\\cvip\\proj2\\mountain2.jpg", 1)
# calling the SIFT function
sift = cv2.xfeatures2d.SIFT_create()
#computing the sift features for img1 and img2
(kp1, d1) = sift.detectAndCompute(img1, None)
(kp2, d2) = sift.detectAndCompute(img2, None)
#drawing keypoints
sift1 = cv2.drawKeypoints(img1, kp1, color=(150, 0, 0), outImage=np.array([]))
sift2 = cv2.drawKeypoints(img2, kp2, color=(150, 0, 0), outImage=np.array([]))
match_c = 10
print("# keypoints: {}, descriptors: {}".format(len(kp1), d1.shape))
print("# keypoints: {}, descriptors: {}".format(len(kp2), d2.shape))
cv2.imwrite("task1 sift1.jpg", sift1)
cv2.imwrite("task1 sift2.jpg", sift2)
FLANN_INDEX_KDTREE = 0
#index parameters and search parameters with FLANN
indexp= dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
searchp = dict(checks=50)
flann = cv2.FlannBasedMatcher(indexp, searchp)
matches = flann.knnMatch(d1, d2, k=2)
g = []
#knn matcher with the condition mentioned
for m, n in matches:
    if m.distance < 0.75 * n.distance:
        g.append(m)
# Fucntion to find inliners - random selection
if (len(g) > match_c):
    #source points
    src_pt = np.float32([kp1[m.queryIdx].pt for m in g]).reshape(-1, 1, 2)
    # destiantion points
    dst_pt = np.float32([kp2[m.trainIdx].pt for m in g]).reshape(-1, 1, 2)
    H, mask = cv2.findHomography(src_pt, dst_pt, cv2.RANSAC, 5.0)
    matchesMask = mask.ravel().tolist()
    Mask1 = (mask.ravel()==1).tolist()
    #list to the random points
    list_t = []
```

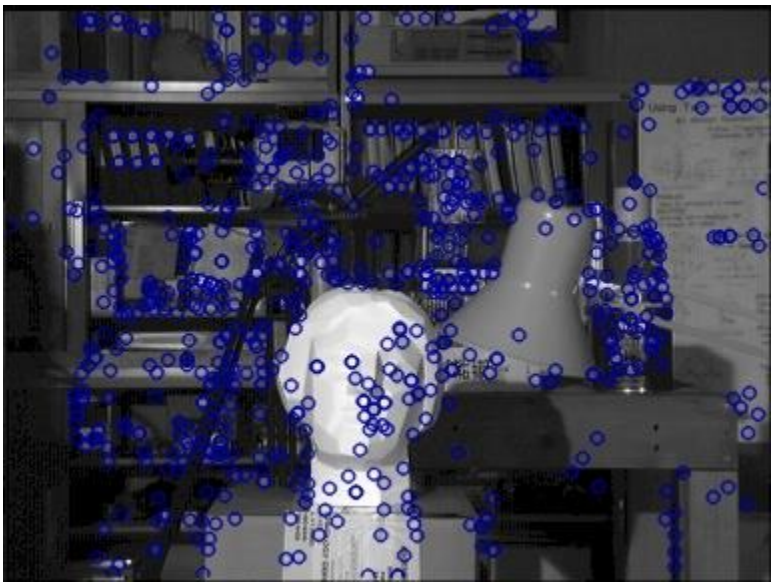
```
#randomly generated points for selecting the inliers
   for i in range(0, 10):
      x = random.randint(0, 244)
      list_t.append(x)
   for i in range(0,len(Mask1)):
      if (i in list_t):
         y =0
      else:
         Mask1[i]=False
   h, w = img1.shape
   pts = np.float32([[0, 0], [0, h - 1], [w - 1, h - 1], [w - 1, 0]]).reshape(-1, 1, 2)
   dst = cv2.perspectiveTransform(pts, H)
#function for wrapping the image - mostly made up of fucntion call
def warpImages(img1, img2, H):
   rows1, cols1 = img1.shape[:2]
   rows2, cols2 = img2.shape[:2]
   list_of_points_1 = np.float32([[0, 0], [0, rows1], [cols1, rows1], [cols1, 0]]).reshape(-1, 1, 2)
   temp_points = np.float32([[0, 0], [0, rows2], [cols2, rows2], [cols2, 0]]).reshape(-1, 1, 2)
   list_of_points_2 = cv2.perspectiveTransform(temp_points, H)
   list_of_points = np.concatenate((list_of_points_1, list_of_points_2), axis=0)
   [x_min, y_min] = np.int32(list_of_points.min(axis=0).ravel() - 0.5)
   [x_max, y_max] = np.int32(list_of_points.max(axis=0).ravel() + 0.5)
   translation_dist = [-x_min, -y_min]
   H_translation = np.array([[1, 0, translation_dist[0]], [0, 1, translation_dist[1]], [0, 0, 1]])
   output_img = cv2.warpPerspective(img2, H_translation.dot(H), (x_max - x_min, y_max - y_min))
   output_img[translation_dist[1]:rows1 + translation_dist[1], translation_dist[0]:cols1 + translation_dist[0]] =
img1
   return output_img
warp = warpImages(img2_color, img1_color, H)
draw_params = dict(matchColor=(0, 0, 240),
               singlePointColor=None,
               matchesMask=matchesMask,
               flags=2)
params1 = dict(matchColor=(0, 100, 0),
               singlePointColor=None,
               matchesMask=Mask1,
               flags=2)
img3 = cv2.drawMatches(img1, kp1, img2, kp2, g, None, **draw_params)
cv2.imwrite("warp.jpg", warp)
cv2.imwrite("task1_matches_knn.jpg", img3)
img4 = cv2.drawMatches(img1_color, kp1, img2_color, kp2, g, None, **params1)
cv2.imwrite("task1_matches.jpg", img4)
print("The homography matrix is")
print(H)
```

2 EPIPOLAR GEOMETRY

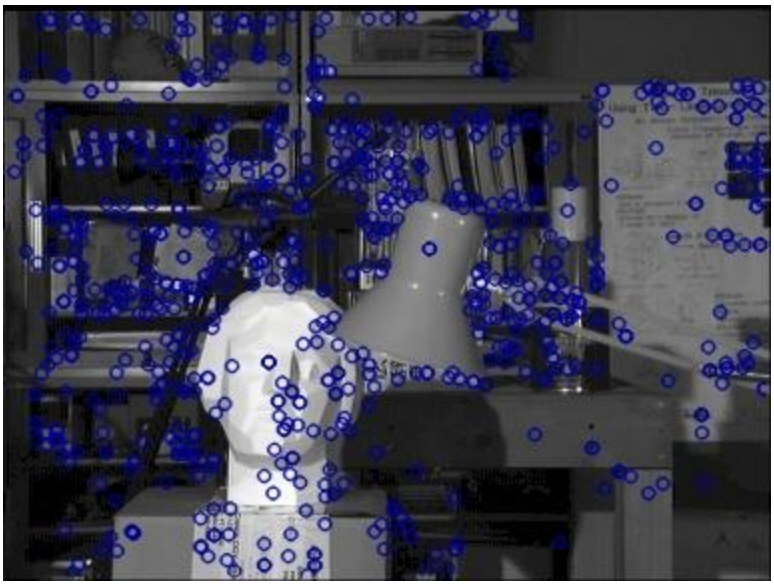
Epipolar geometry is the geometry of stereo vision. When two cameras view a 3D scene from two distinct positions, there are a number of geometric relations between the 3D points and their projections onto the 2D images that lead to constraints between the image points. These relations are derived based on the assumption that the cameras can be approximated by the pinhole camera model.

2.1 SIFT AND KNN FOR THE GIVEN IMAGE

The task 1.1 and 1.2 is repeated for both the images given, which gives us two SIFT images and one match image.



SIFT 1 Image



SIFT 2 Image

2.2 FUNDAMENTAL MATRIX

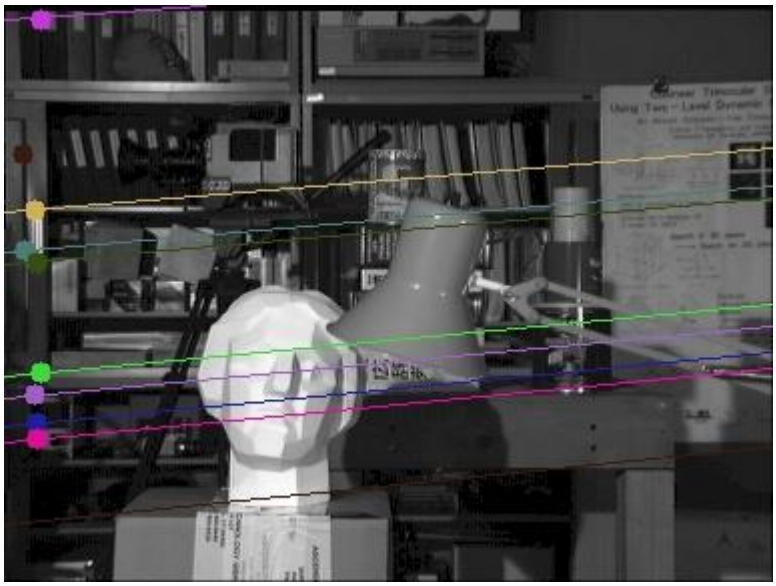
Fundamental Matrix with RANSAC.

```
[ 2.58938824e-06 3.00178724e-05 2.05540959e-02]
[ 2.36594625e-06 1.58586353e-05 2.80750023e-01]
[-2.40895761e-02 -2.87750686e-01 1.00000000e+00]
```

2.3 RANDOM 10 INLINERS

Randomly 10 inliners have been selected and is drawn on the image and is shown in figure below. The color of both the lines are not same as the colors have been randomized.

Epi image left



Epi image right

2.4 DISPARITY IMAGE

Disparity refers to the distance between two corresponding points in the left and right image of a stereo pair. The Disparity map is computed with the two images using libraries and the output is shown as below. The image below which is written using imwrite is shown as pic1 and the image plotted with pyplot is pic2. The program also plots the disparity image

Pic1

Pic 2

2.5 CODE

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
UBIT = 'jayanthk'
np.random.seed(sum([ord(c) for c in UBIT]))
import random

img1 = cv2.imread("D:\\sem1\\cvip\\proj2\\tsucuba_left.png", 0)
img2 = cv2.imread("D:\\sem1\\cvip\\proj2\\tsucuba_right.png", 0)
sift = cv2.xfeatures2d.SIFT_create()
(kp1, d1) = sift.detectAndCompute(img1, None)
(kp2, d2) = sift.detectAndCompute(img2, None)
sift_img1 = cv2.drawKeypoints(img1, kp1, color=(150,0,0), outImage=np.array([]))
sift_img2 = cv2.drawKeypoints(img2, kp2, color=(150,0,0), outImage=np.array([]))
print("# keypoints: {}, descriptors: {}".format(len(kp1), d1.shape))
print("# keypoints: {}, descriptors: {}".format(len(kp2), d2.shape))
cv2.imwrite("D:\\sem1\\cvip\\proj2\\task2_sift1.jpg", sift_img1)
cv2.imwrite("D:\\sem1\\cvip\\proj2\\task2_sift2.jpg", sift_img2)
FLANN_INDEX_KDTREE = 0
index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
search_params = dict(checks = 50)
flann = cv2.FlannBasedMatcher(index_params, search_params)
matches = flann.knnMatch(d1,d2,k=2)
g = []
for m,n in matches:
   if m.distance < 0.75*n.distance:
      g.append(m)
src_pt = np.float32([ kp1[m.queryIdx].pt for m in g ])
dst_pt = np.float32([ kp2[m.trainIdx].pt for m in g ])
F, mask = cv2.findFundamentalMat(src_pt, dst_pt, cv2.RANSAC,5.0)
matchesMask = mask.ravel().tolist()
draw_params = dict(matchColor = (150,0,0), # draw matches in green color
                   singlePointColor = None,
                  matchesMask = matchesMask,
                  flags = 2)
img3 = cv2.drawMatches(img1,kp1,img2,kp2,g,None,**draw_params)
cv2.imwrite("D:\\sem1\\cvip\\proj2\\task2_matches_knn.jpg", img3)
FLANN_INDEX_KDTREE = 0
index_p = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
search_p = dict(checks = 50)
flann = cv2.FlannBasedMatcher(index_p, search_p)
matches = flann.knnMatch(d1,d2,k=2)
print("Fundamental Matrix ")
print(F)
stereoMatcher = cv2.StereoBM_create()
stereoMatcher.setMinDisparity(16)
```

```
stereoMatcher.setNumDisparities(112)
stereoMatcher.setBlockSize(17)
stereoMatcher.setSpeckleRange(32)
stereoMatcher.setSpeckleWindowSize(120)
stereo = stereoMatcher.compute(img1, img2)
cv2.imwrite("D:\\sem1\\cvip\\proj2\\task2_disparity.jpg", stereo)
#showing plot of the disparity
plt.imshow(stereo)
src_pt = np.int32(src_pt)
dst_pt = np.int32(dst_pt)
pts1 = src_pt
pts2 = dst_pt
pts1 = pts1[mask.ravel()==1]
pts2 = pts2[mask.ravel()==1]
def drawlines(img1,img2,lines,pts1,pts2):
    r,c = img1.shape
    img1 = cv2.cvtColor(img1,cv2.COLOR_GRAY2BGR)
    img2 = cv2.cvtColor(img2,cv2.COLOR_GRAY2BGR)
    for r,pt1,pt2 in zip(lines,pts1,pts2):
        color = tuple(np.random.randint(0,255,3).tolist())
        x0,y0 = map(int, [0, -r[2]/r[1] ])
        x1,y1 = map(int, [c, -(r[2]+r[0]*c)/r[1] ])
        img1 = cv2.circle(img1,tuple(pt1),5,color,-1)
        img2 = cv2.circle(img2,tuple(pt2),5,color,-1)
return img1,img2
list = []
for i in range(0,11):
    y = random.randint(0,271)
    list.append(y)
lines1 = cv2.computeCorrespondEpilines(pts2.reshape(-1,1,2), 2,F)
lines_re = lines1.reshape(-1,3)
lines_co = np.copy(lines1)
count=0
for i in list:
    lines_co[count,:] = lines1[i,:]
line2_co = np.copy(lines11[0:10])
img5,img6 = drawlines(img1,img2,line2_co,pts1,pts2)
# epilines corresponding to points in left image (first image) and drawing its lines on right image
lines2 = cv2.computeCorrespondEpilines(pts1.reshape(-1,1,2), 1,F)
lines2 = lines2.reshape(-1,3)
lines2_co= np.copy(lines2)
count=0
for i in list:
    lines2_co[count,:] = lines2_co[i,:]
line2_rco= np.copy(lines2_co[0:10])
img3,img4 = drawlines(img2,img1,line2_rco,pts2,pts1)
```

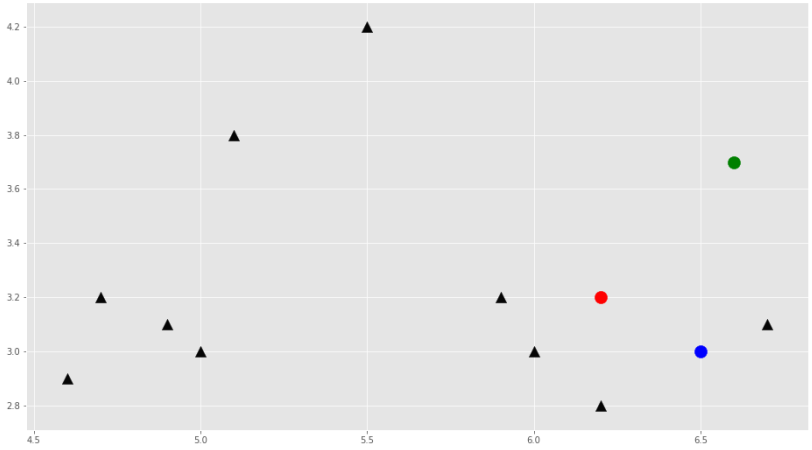
```
cv2.imwrite("D:\\sem1\\cvip\\proj2\\task2_epi1_left.jpg", img5)
cv2.imwrite("D:\\sem1\\cvip\\proj2\\task2_epi2_right.jpg", img3)
```

3 K-MEANS CLUSTERING

Implement K-Means clustering algorithm using only numpy with Euclidean distance.

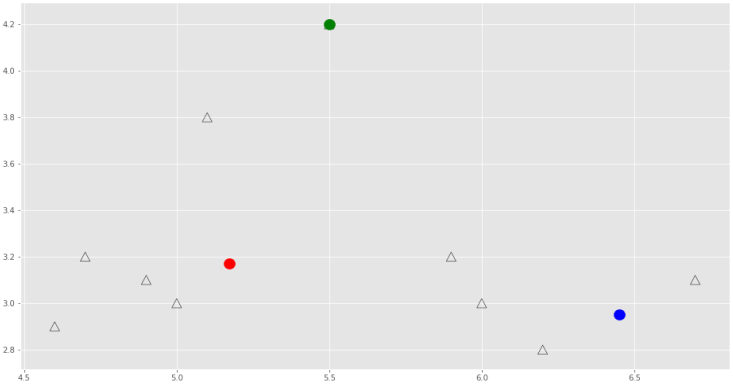
3.1 CLASSIFY N = 10 SAMPLES WITH THE NEAREST MEU POINTS

The Classification vector and classification plot meu(i = 1; 2; 3).



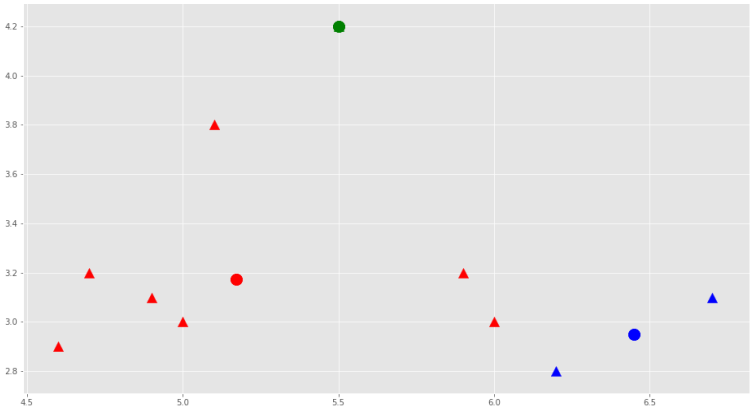
3.2 UPDATED MEU

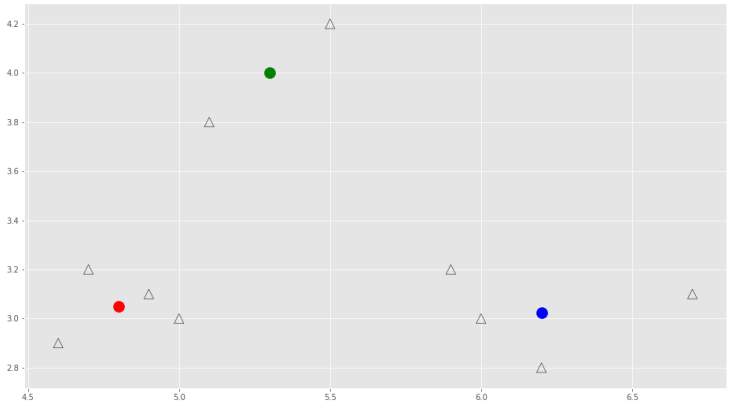
Plot with the updated MEU.



3.3 PLOTS FOR SECOND ITERATION

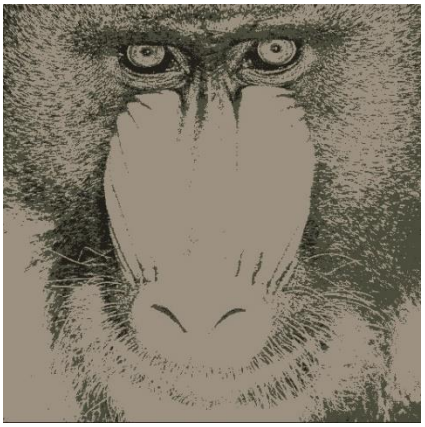
Plots after second iteration.





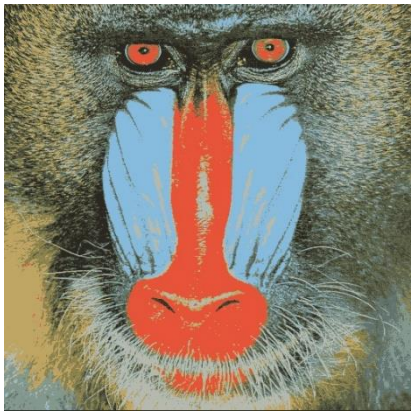
3.4 COLOR QUANTIZATION

Color quantization or color image quantization is quantization applied to color spaces; it is a process that reduces the number of distinct colors used in an image, usually with the intention that the new image should be as visually similar as possible to the original image.
We have applied k-means to for image quantization. The images are saved for k = {3,5,10,20}

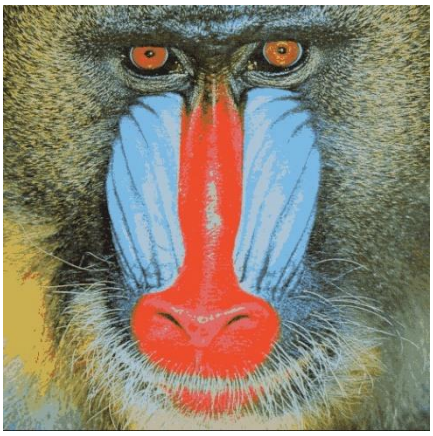


K= 3

K=5



K=10



K=20

3.5 CODE

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
UBIT = 'jayanthk'
np.random.seed(sum([ord(c) for c in UBIT]))
import matplotlib.pyplot as plt
from matplotlib import style
import matplotlib
plt.rcParams['figure.figsize'] = (16, 9)
plt.style.use('ggplot')
colors = ['r', 'g', 'b', 'y', 'c', 'm']
MatchThreshold = 10
y1 = [5.9, 4.6, 6.2, 4.7, 5.5, 5.0, 4.9, 6.7, 5.1, 6.0]
y2 = [3.2, 2.9, 2.8, 3.2, 4.2, 3.0, 3.1, 3.1, 3.8, 3.0]
X = np.array(list(zip(y1, y2)))
plt.scatter(f1, f2, c='black', s=7)
# Euclidean Distance Caculator
def dist(a, b, ax=1):return np.linalg.norm(a - b, axis=ax)
# Number of clusters
k = 3
# X  and Y coordinates of random centroids
C_x = [6.2, 6.6, 6.5]
C_y = [3.2, 3.7, 3.0]
C = np.array(list(zip(C_x, C_y)), dtype=np.float32)
plt.scatter(f1, f2, c='#050505', marker='^', s=150)
plt.scatter(C[0, 0], C[0, 1], marker='o', s=200, c=colors[0])
plt.scatter(C[1, 0], C[1, 1], marker='o', s=200, c=colors[1])
plt.scatter(C[2, 0], C[2, 1], marker='o', s=200, c=colors[2])
plt.savefig('task3 iter1 a.jpg')
C_old = np.zeros(C.shape)
clusters = np.zeros(len(X))
error = dist(C, C_old, None)
colors = ['r', 'g', 'b', 'y', 'c', 'm']
c = 0
while error != 0:
    c = c + 1
    for i in range(len(X)):
        distances = dist(X[i], C)
        cluster = np.argmin(distances)
        clusters[i] = cluster
    C_old = np.copy(C)
    for i in range(k):
    points = [X[j] for j in range(len(X)) if clusters[j] == i]
        C[i] = np.mean(points, axis=0)
    fig, b = plt.subplots()
    b.scatter(f1, f2, marker='^', s=150, edgecolor='black', facecolor='none')
    b.scatter(C[0, 0], C[0, 1], marker='o', s=200, c=colors[0])
    b.scatter(C[1, 0], C[1, 1], marker='o', s=200, c=colors[1])
```

```
b.scatter(C[1, 0], C[1, 1], marker='o', s=200, c=colors[1])
b.scatter(C[2, 0], C[2, 1], marker='o', s=200, c=colors[2])
plt.savefig('task3_iter{}_b.png'.format(c))
error = dist(C, C_old, None)
   fig, a = plt.subplots()
   if(c>1):
      break
   for i in range(3):
      points = np.array([X[j] for j in range(len(X)) if clusters[j] == i])
      a.scatter(points[:, 0], points[:, 1], marker='^', s=150, c=colors[i])
      a.scatter(C[0, 0], C[0, 1], marker='o', s=200, c=colors[1])
      a.scatter(C[1, 0], C[1, 1], marker='o', s=200, c=colors[2])
     a.scatter(C[2, 0], C[2, 1], marker='o', s=200, c=colors[3])
     plt.savefig('task3_iter{}_a.png'.format(c+1))
if (error == 0):
    print(C)

class KMeans_custom():
          def _init_(self, num_clusters, tolerance=0.001, epoch=25, centroids={}):
      self.tolerance = tolerance
      self.epochs = epoch
      self.centroids = centroids
      self.num_clusters = num_clusters
   def predict(self, data):
      cl= list()
      for point in data:
         distances = [np.linalg.norm(point - self.centroids[centroid]) for centroid in self.centroids]          c
cl.append(distances.index(min(distances)))
      return np.array(cl)
   def fit(self, data):
      if self.centroids == {}:
         for i in range(self.num_clusters):
            self.centroids[i] = data[i]
      for i in range(self.epochs):
         self.classifications = {}
         for clu in range(self.num_clusters):
            self.classifications[clu] = []
         for f in data:
            distances = [np.linalg.norm(f - self.centroids[centroid]) for centroid in self.centroids]
            classification = distances.index(min(distances))
self.classifications[classification].append(features)
         prev_centroid = dict(self.centroids)
         for classification in self.classifications:
            self.centroids[classification] = np.average(self.classifications[classification], axis=0)
         optimized = True
         for cent in self.centroids:
            original_centroid = prev_centroid[cent]
            current_centroid = self.centroids.get(cent)
            if np.sum(((original_centroid - current_centroid) * 100) * original_centroid) > self.tolerance:
                opti = False
         if opti:
           break
```

```
def recreate(codebook, labels, w, h):
    img = np.zeros((w, h, 3))
    img = np.array(img, dtype=np.float64)
    label_idx = 0
    for i in range(w):
        for j in range(h):
            img[i][j] = codebook[labels[label_idx]]
            label_idx += 1
    return img

def baboon(nc):
    imm = cv2.imread('D:\\sem1\\cvip\\proj2\\baboon.jpg')
    img_matrix = cv2.cvtColor(imm, cv2.COLOR_BGR2RGB)
    image = np.array(img_matrix, dtype=np.float64) / 255
    w, h, d = tuple(image.shape)
    image_array = np.reshape(image, (w * h, d))
    image_quantization = KMeans(num_clusters=nc, epoch=30)
    image_quantization.fit(image_array)
    lab = image_quantization.predict(image_array)
    print(image_quantization.centroids)
    imgn = recreate(image_quantization.centroids, lab, w, h)
matplotlib.image.imsave('task3_baboon_{}.png'.format(nc), imgn)

baboon(3)
```

4. REFERENCES

1) https://en.wikipedia.org/wiki/Color_quantization
2) https://en.wikipedia.org/wiki/Epipolar_geometry
3) https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_feature_homography/py_feature_homography.html
4) https://mubaris.com/posts/kmeans-clustering/
5) https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4712040/