CVIP -Project2

5th November 2018

Pratik Appaso Vagyani
Graduate Student,
Department of Computer Science & Engineering
SUNY Buffalo,
pratikap@buffalo.edu
UBIT No:50288741

OVERVIEW

Project consist of 3 tasks:

1. Image Features and Homography
2. Epipolar Geometry
3. K-means Clustering

Image Features and Homography

Source code:

```
import numpy as np
import cv2
UBIT = 'pratikap'; np.random.seed(sum([ord(c) for c in UBIT]))
MOUNTAIN1 = r"data\mountain1.jpg"
MOUNTAIN2 = r"data\mountain2.jpg"


class ImageFeaturesHomography:

   def task1(self, input_img, result_img):
       gray_scale_img = cv2.cvtColor(input_img, cv2.COLOR_BGR2GRAY)
       sift = cv2.xfeatures2d.SIFT_create()
       key_points = sift.detect(gray_scale_img, None)
       result_img_mat = cv2.drawKeypoints(input_img,key_points,color=(0,255,255), outImage=np.array([]),
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
       cv2.imwrite(result_img, result_img_mat)

   def task2(self,image_1, image_2,result_img):
       img1 = cv2.cvtColor(image_1, cv2.COLOR_BGR2GRAY)
       img2 = cv2.cvtColor(image_2, cv2.COLOR_BGR2GRAY)
       detector = cv2.xfeatures2d.SIFT_create()
       key_points1, descript1 = detector.detectAndCompute(img1, None)
       key_points2, descript2 = detector.detectAndCompute(img2, None)

       matcher = cv2.DescriptorMatcher_create(cv2.DescriptorMatcher_FLANNBASED)
       knn_matches = matcher.knnMatch(descript1, descript2, k=2)

       good_matches = []
       for m, n in knn_matches:
           if m.distance < 0.7 * n.distance:
```

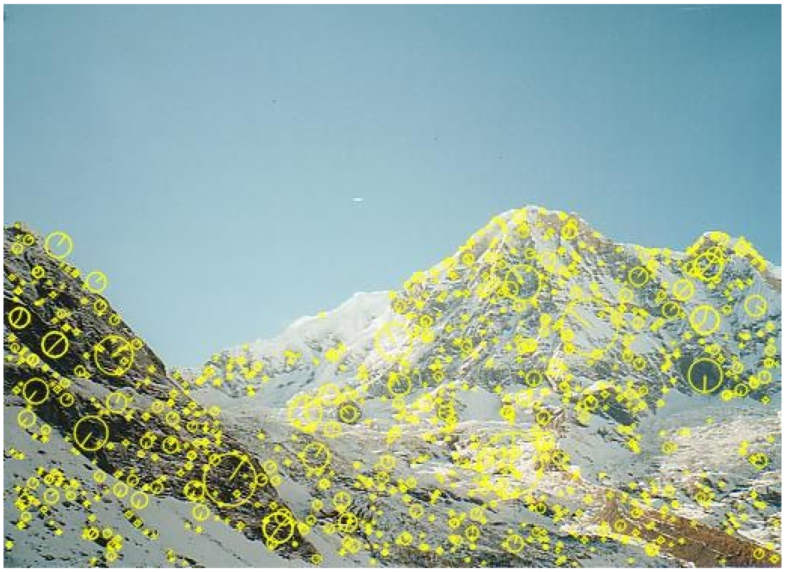
```python
                good_matches.append(m)

        img_matches = np.empty((max(img1.shape[0], img2.shape[0]), img1.shape[1] + img2.shape[1], 3), dtype=np.uint8)
        cv2.drawMatches(image_1, key_points1, image_2, key_points2, good_matches, img_matches,
                       flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
        cv2.imwrite(result_img, img_matches)
        return key_points1, key_points2, good_matches

    def task3(self, key_points1, key_points2, good_matches):
        src_points = np.empty((len(good_matches), 2), dtype=np.float32)
        dest_points = np.empty((len(good_matches), 2), dtype=np.float32)
        for i in range(len(good_matches)):
            src_points[i, 0] = key_points1[good_matches[i].queryIdx].pt[0]
            src_points[i, 1] = key_points1[good_matches[i].queryIdx].pt[1]
            dest_points[i, 0] = key_points2[good_matches[i].trainIdx].pt[0]
            dest_points[i, 1] = key_points2[good_matches[i].trainIdx].pt[1]
        H, mask = cv2.findHomography(src_points, dest_points, cv2.RANSAC)
        print(H)
        return H, mask, src_points, dest_points

    def task4(self, img1, img2, key_points1, key_points2, good_matches, mask, out_file):
        matchesMask = mask.ravel().tolist()
        new_matches = []
        new_good = []
        for i in np.random.randint(0, len(matchesMask)-1, 10):
            new_matches.append(matchesMask[i])
            new_good.append(good_matches[i])

        result = cv2.drawMatches(img1, key_points1, img2, key_points2, new_good, None,matchColor=(0,255,255),
matchesMask=new_matches, flags=2)
        cv2.imwrite(out_file, result)

    def task5(self,img1, img2, homography, out_file):
        #warp1 = cv2.warpPerspective(img1, homography, (img2.shape[1], img2.shape[0]),img2)
        #warp2 = cv2.warpPerspective(img2, homography, (img1.shape[1], img1.shape[0]),img1)
        #cv2.imwrite(out_file, warp)
        rows1, cols1 = img1.shape[:2]
        rows2, cols2 = img2.shape[:2]

        lp1 = np.float32([[0, 0], [0, rows1], [cols1, rows1], [cols1, 0]]).reshape(-1, 1, 2)
        temp = np.float32([[0, 0], [0, rows2], [cols2, rows2], [cols2, 0]]).reshape(-1, 1, 2)

        lp2 = cv2.perspectiveTransform(temp, homography)
        lp = np.concatenate((lp1, lp2), axis=0)

        [x_min, y_min] = np.int32(lp.min(axis=0).ravel() - 0.5)
        [x_max, y_max] = np.int32(lp.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]])

        result = cv2.warpPerspective(img1, H_translation.dot(homography), (x_max - x_min, y_max - y_min))
        result[translation_dist[1]:rows1 + translation_dist[1], translation_dist[0]:cols1 + translation_dist[0]] = img2
        cv2.imwrite(out_file, result)

    def start(self):
        img1 = cv2.imread(MOUNTAIN1)
        img2 = cv2.imread(MOUNTAIN2)

        self.task1(img1, r"data\task1_sift1.jpg")
        self.task1(img2, r"data\task1_sift2.jpg")

        key_points1, key_points2, good_matches = self.task2(img1, img2, r'data\task1_matches_knn.jpg')
        H, mask,_,_ =self.task3(key_points1, key_points2, good_matches)
        self.task4(img1, img2, key_points1,key_points2, good_matches,mask,r"data\task1_matches.jpg")
        self.task5(img1, img2,H,r"data\task1pano.jpg")
```

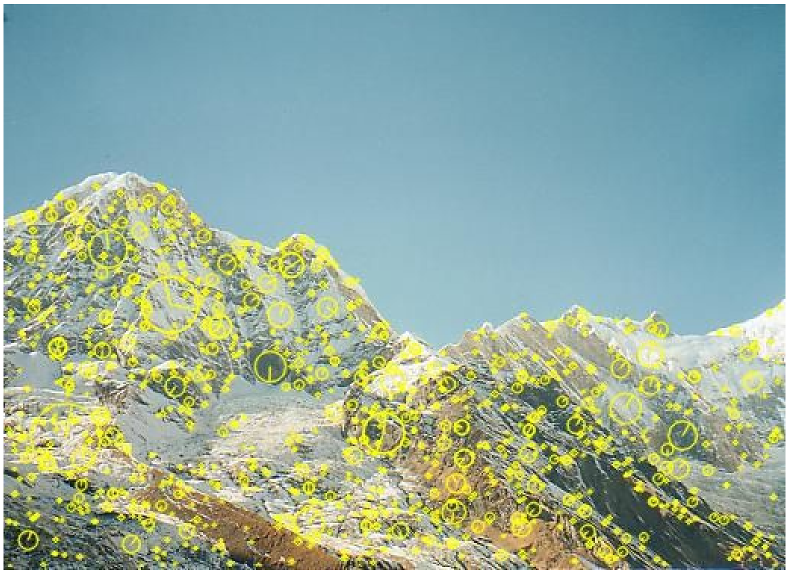
```
def main():
   ifh = ImageFeaturesHomography()
   ifh.start()


if __name__ == '__main__':
   main()
```

Result:

1.1 SIFT feature:

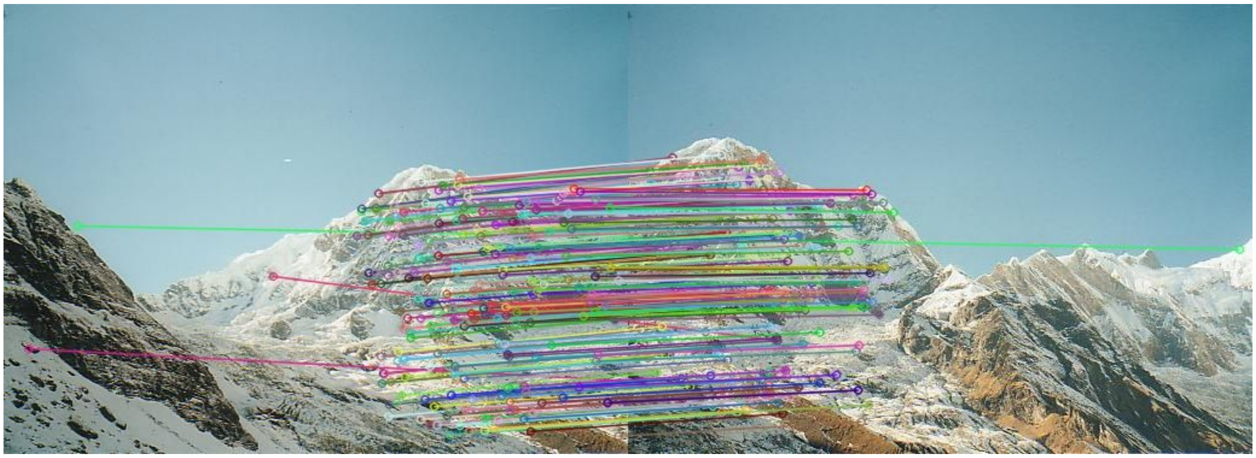
1. Create SIFT using `cv2.xfeatures2d.SIFT_create()`
2. Detect keypoints using gray-scale image
3. Draw keypoints on original image using `cv2.drawKeypoints`





1.2 match keypoints using knn:

1. Use SIFT and get key points & descriptor of both images using `sift.detectAndCompute()`
2. Create a flann based matcher and find knn matches `matcher.knnMatch(descript1, descript2, k=2)`
3. Find good matches with threshold of 0.75 and draw good matches using `cv2.drawMatches`

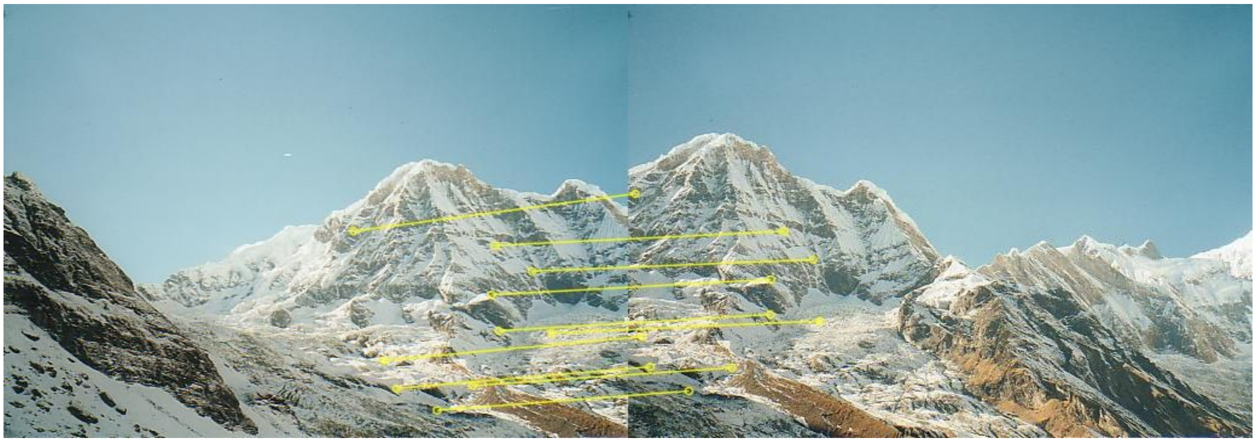


1.3 Homography matrix H:

```
[[ 1.59127326e+00 -2.92907219e-01 -3.96218005e+02]
 [ 4.51154863e-01  1.43105483e+00 -1.90949351e+02]
 [ 1.21978996e-03 -6.82082094e-05  1.00000000e+00]]
```

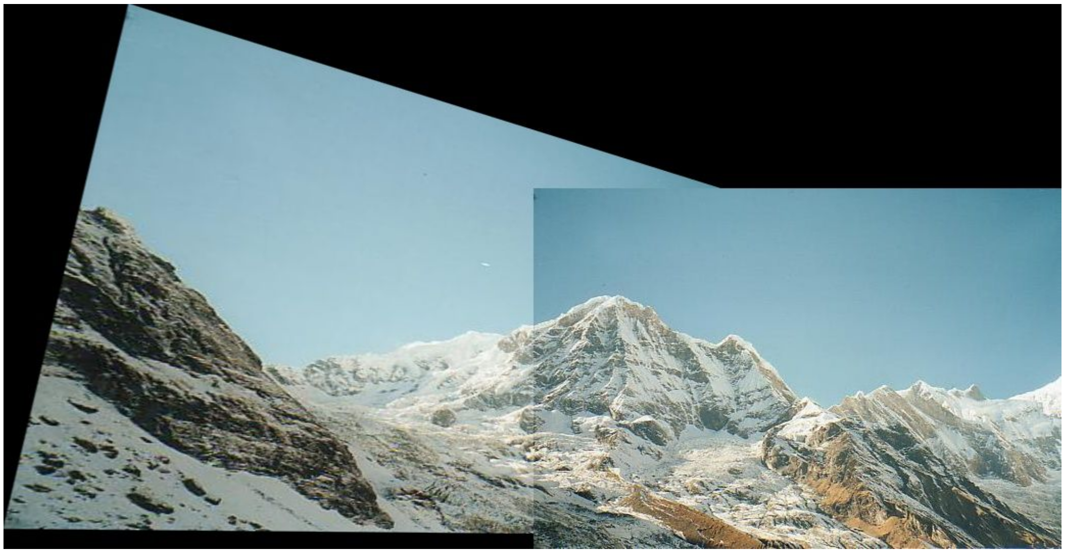
1.4

1. Use data from 1.3 `key_points1, key_points2, good_matches, mask`
2. Get inlier using `mask.ravel().tolist()`
3. Randomly select 10 inlier and drawMatches on image



1.5

1. Get corners of both the images and calculate perspective transform of right image with homography matrix obtained in 1.3.
2. Dot product of homography translation with the homography matrix
3. Compute warp image of the left image using the dot product



Epipolar Geometry

Source code:

```
import numpy as np
import cv2
from ImageFeaturesHomography import ImageFeaturesHomography
UBIT = 'pratikap'; np.random.seed(sum([ord(c) for c in UBIT]))


class EpipolarGeometry:
   def task1(self, img1,img2, output1,output2,output3):
       ifh = ImageFeaturesHomography()
       ifh.task1(img1,output1)
       ifh.task1(img2,output2)
       ifh.task2(img1, img2, output3)

   def task2(self, img_1, img_2):
       img1 = cv2.cvtColor(img_1, cv2.COLOR_BGR2GRAY)
       img2 = cv2.cvtColor(img_2, cv2.COLOR_BGR2GRAY)

       #detector = cv2.xfeatures2d_SURF.create(hessianThreshold=400)
       sift = cv2.xfeatures2d.SIFT_create()
       key_points1, descript1 = sift.detectAndCompute(img1, None)
       key_points2, descript2 = sift.detectAndCompute(img2, None)
       #matcher = cv2.DescriptorMatcher_create(cv2.DescriptorMatcher_FLANNBASED)
       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(des1, des2, k=2)
       knn_matches = flann.knnMatch(descript1, descript2, k=2)
       # -- Filter matches using the Lowe's ratio test
       points1 = []
       points2 = []
```

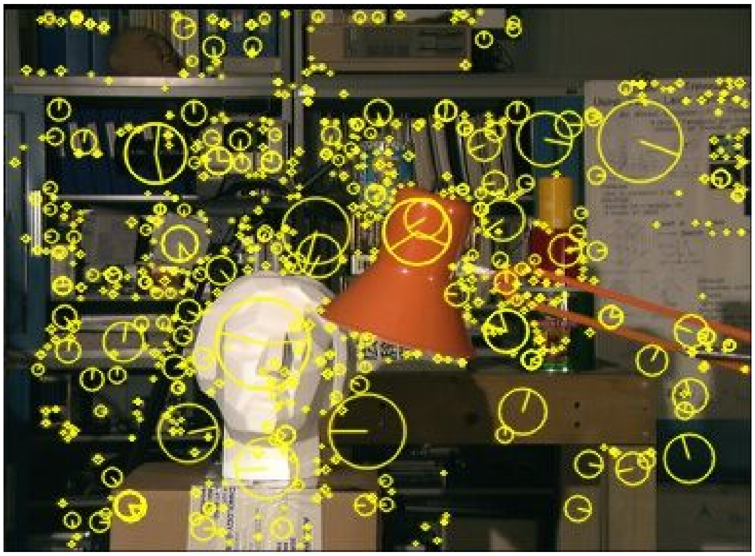
```
        for m, n in knn_matches:
            if m.distance < 0.75 * n.distance:
                points1.append(key_points1[m.queryIdx].pt)
                points2.append(key_points2[m.trainIdx].pt)
        points1 = np.int32(points1)
        points2 = np.int32(points2)
        F, mask = cv2.findFundamentalMat(points1, points2, cv2.RANSAC)
        print(F)
        return F, mask, points1, points2

    def task3(self, F, mask,img1,img2, points1, points2):
        pts1 = points1[mask.ravel() == 1]
        pts2 = points2[mask.ravel() == 1]
        n_pts1 = []
        n_pts2 = []
        for i in np.random.randint(0, len(pts1)-1, 10):
            n_pts1.append(pts1[i])
            n_pts2.append(pts2[i])

        pts1 = np.int32(n_pts1)
        pts2 = np.int32(n_pts2)

        lines1 = cv2.computeCorrespondEpilines(pts2.reshape(-1, 1, 2), 2, F)
        lines1 = lines1.reshape(-1, 3)
        img5, img6 = self.draw_lines(img1, img2, lines1, pts1, pts2)
        lines2 = cv2.computeCorrespondEpilines(pts1.reshape(-1, 1, 2), 1, F)
        lines2 = lines2.reshape(-1, 3)
        img3, img4 = self.draw_lines(img2, img1, lines2, pts2, pts1)
        cv2.imwrite(r'data\task2_epi_right.jpg',img5)
        cv2.imwrite(r'data\task2_epi_left.jpg',img3)

    def draw_lines(self, img_1, img_2, lines, pts1, pts2):
        r, c= cv2.cvtColor(img_1,cv2.COLOR_BGR2GRAY).shape
        img1 = cv2.cvtColor(cv2.cvtColor(img_1,cv2.COLOR_BGR2GRAY), cv2.COLOR_GRAY2BGR)
        img2 = cv2.cvtColor(cv2.cvtColor(img_2,cv2.COLOR_BGR2GRAY), cv2.COLOR_GRAY2BGR)
        clr = np.array([10,50,255])
        for r, pt1, pt2 in zip(lines, pts1, pts2):
            clr = np.add(clr,np.array([20,15,-20]))
            color = tuple(clr.tolist())
            x0, y0 = map(int, [0, -r[2] / r[1]])
            x1, y1 = map(int, [c, -(r[2] + r[0] * c) / r[1]])
            img1 = cv2.line(img1, (x0, y0), (x1, y1), color, 1)
            img1 = cv2.circle(img1, tuple(pt1), 5, color, -1)
            img2 = cv2.circle(img2, tuple(pt2), 5, color, -1)
        return img1, img2

    def task4(self,img1,img2):
        img_l = cv2.cvtColor(img1,cv2.COLOR_BGR2GRAY)
        img_r = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
        stereo = cv2.StereoBM_create(numDisparities=64, blockSize=11)
        disparity = stereo.compute(img_l, img_r)
        #import matplotlib.pyplot as plt
        cv2.imwrite(r'data\\task2_disparity.jpg',disparity)

    def start(self):
        img_left = cv2.imread(r'data\tsucuba_left.png')
        img_right = cv2.imread(r'data\tsucuba_right.png')
        self.task1(img_left,img_right,r'data\task2_sift1.jpg',r'data\task2_sift2.jpg',r'data\task2_matches_knn.jpg')
        print("--" * 5, "task 2.1 completed", "--" * 5)
        F, mask, points1, points2 = self.task2(img_left, img_right)
        print("--" * 5, "task 2.2 completed", "--" * 5)
        self.task3(F, mask,img_left,img_right, points1, points2)
        print("--" * 5, "task 2.3 completed", "--" * 5)
        img_left = cv2.imread(r'data\tsucuba_left.png')
        img_right = cv2.imread(r'data\tsucuba_right.png')
        self.task4(img_left, img_right)
        print("--" * 5, "task 2.4 completed", "--" * 5)
```

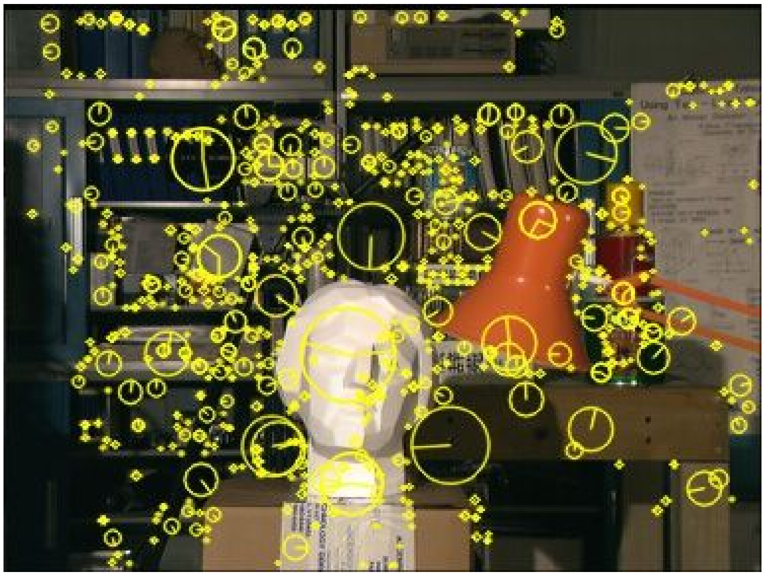
```
def main():
    eg = EpipolarGeometry()
    eg.start()


if __name__ == "__main__":
    main()
```

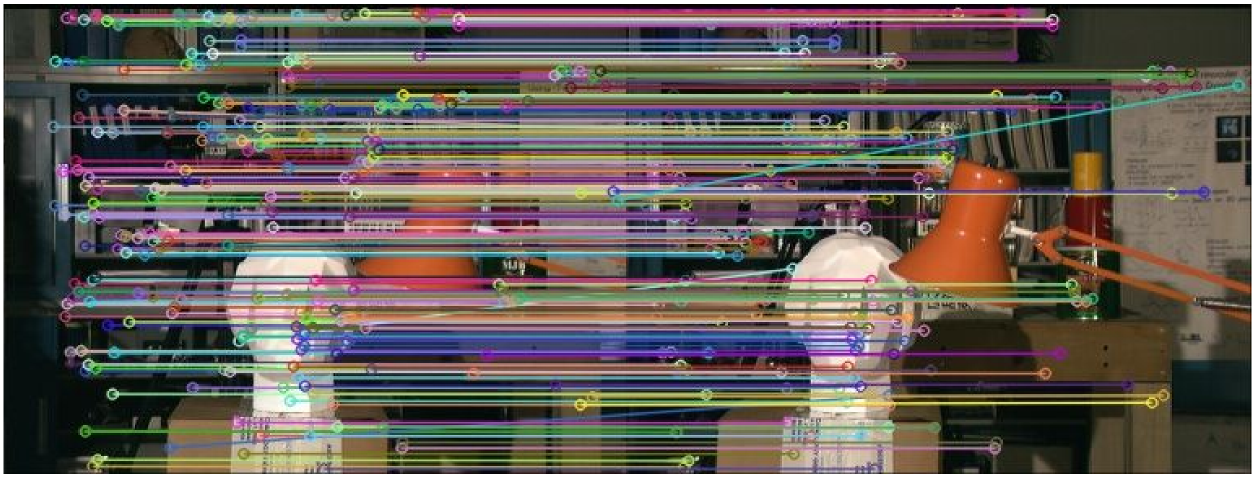
Result

2.1

1. Called functions 1.1, 1.2 of task1 with images of task2







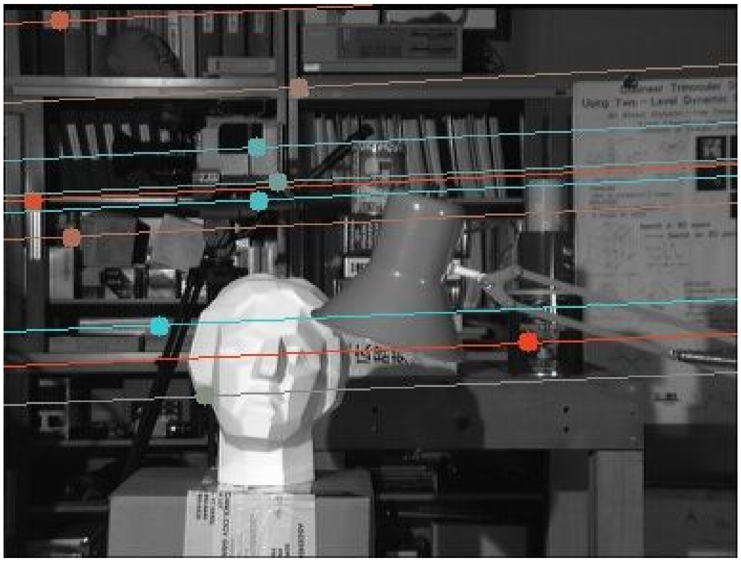
2.2 Fundamental matrix F:

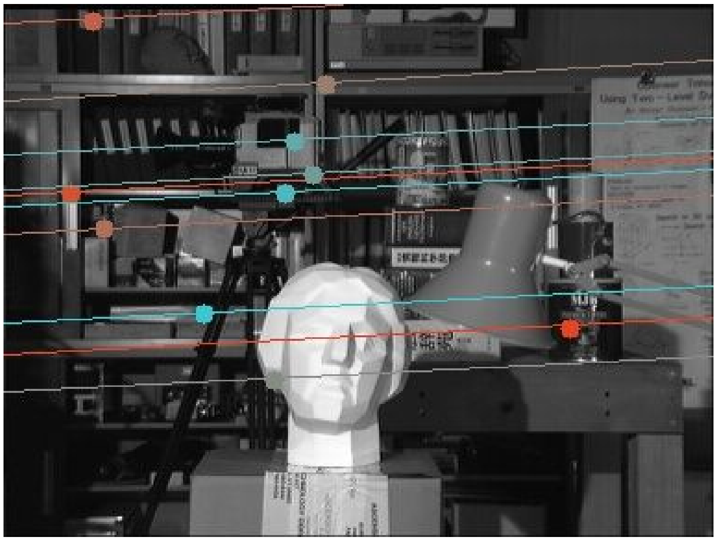
1. Get keypoints of both images , list them on basis of best matches
2. Calculate fundamental matrix using `cv2.findFundamentalMat(points1, points2, cv2.RANSAC)`

```
[[-2.12607354e-06 -8.10713687e-05  7.47530309e-02]
 [ 4.60726414e-05  3.79326900e-05  1.32728554e+00]
 [-7.52042326e-02 -1.32608913e+00  1.00000000e+00]]
```

2.3

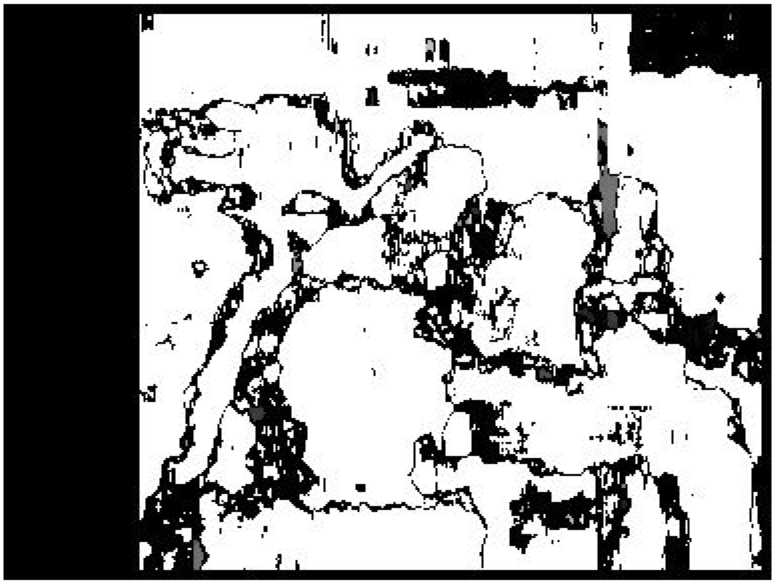
1. Use F, mask,point1,point2 from task 2.2
2. select inliers using the mask.ravel function
3. Select 10 random points.
4. Compute gray scaled images and call the computeCorrespondingEpilines and using fundamental matrix
5. Draw lines for left image key points on the right and vice versa a circle depicting the key point and line corresponds the match.





2.4

1. create object of stereo using the cv2.stereoBM_create .
2. compute disparity using `stereo.compute(img_l, img_r)`



K-means Clustering

Source code:

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


class KMeansClustering:

   def ecludien_distance(self, p, q):
       return math.sqrt(sum([(a - b) ** 2 for a, b in zip(p, q)]))

   def generate_random_mu(self,shape,k):
       x = np.random.randint(0, 255, k)
       y = np.random.randint(0, 255, k)
       z = np.random.randint(0, 255, k)
       mu = []
       for x_, y_, z_ in zip(x,y,z):
           mu.append([z_,y_,z_])
       return mu

   def encode_mu(self,mu):
       return "_".join([str(x) for x in mu])

   def decode_mu(self, mu):
       return [float(a) for a in mu.split("_")]

   def task1(self, mu_list, point_list, color=['r', 'g', 'b'], out_file=r'data\\task3_iter1_a.jpg'):
       plt.clf()
       cluster_map = {}
       for mu in mu_list:
           cluster_map[self.encode_mu(mu)] = []

       for point in point_list:
           min_dis = None
           min_mu = None
           for mu in mu_list:
               dst = self.ecludien_distance(mu, point)
               if min_dis is None or dst < min_dis:
                   min_mu = mu
                   min_dis = dst
           cluster_map[self.encode_mu(min_mu)].append(point)
           plt.text(point[0]-0.11,point[1]-0.11,str(point[0])+","+str(point[1]))
       if out_file is not None:
           for i, cluster in enumerate(cluster_map):
               points = np.array(cluster_map[cluster])
               x = [float(cluster.split("_")[0])]
               y = [float(cluster.split("_")[1])]
               plt.scatter(x, y, c=color[i],edgecolors=color[i], marker="o",s=90)
               plt.text(x[0]-0.11, y[0]-0.11, str(x[0])[:3]+","+str(y[0])[:3])
               plt.scatter(points[:, 0], points[:, 1], c=color[i], edgecolors=color[i], marker="^", s=50)
           plt.savefig(out_file)
           plt.clf()
       classification_vector=[]
       for point in point_list:
           for c, mu in enumerate(cluster_map):
               if self.encode_mu(point) in [self.encode_mu(x) for x in cluster_map[mu]]:
                   classification_vector.append(c+1)
                   break
       print(classification_vector)
       return cluster_map
```

```
def task2(self, cluster_map, color=['r', 'g', 'b'], out_file=r'data\\task3_iter1_b.jpg'):
    new_cluster = []
    for i, cluster in enumerate(cluster_map):
        x = np.average(np.array(cluster_map[cluster])[:, 0])
        y = np.average(np.array(cluster_map[cluster])[:, 1])
        new_cluster.append([x, y])
        plt.scatter([x], [y], c=color[i], edgecolors=color[i], marker="o", s=90)
        plt.text(x - 0.11, y - 0.11, str(x)[:3] + "," + str(y)[:3])
        points = np.array(cluster_map[cluster])
        for pt in points:
            plt.text(pt[0] - 0.11, pt[1] - 0.11, str(pt[0]) + "," + str(pt[1]))
        plt.scatter(points[:, 0], points[:, 1], c=color[i], edgecolors=color[i], marker="^", s=50)
    plt.savefig(out_file)
    plt.clf()
    print(new_cluster)
    return new_cluster

def task3(self,cluster, points, color=['r', 'g', 'b'], out_file=r'data\\task3_iter2_a.jpg'):
    cluster_map = self.task1(cluster, points, color, out_file)
    self.task2(cluster_map,out_file=r'data\\task3_iter2_b.jpg')

def cluster(self, mu_list, img):
    cluster_map = {}
    for mu in mu_list:
        cluster_map[self.encode_mu(mu)] = []

    for p,row in enumerate(img):
        for q,pixel in enumerate(row):
            min_dis = None
            min_mu = None
            for mu in mu_list:
                dst = self.ecludien_distance(mu, pixel)
                if min_dis is None or (dst is not None and dst < min_dis):
                    min_mu = mu
                    min_dis = dst
            cluster_map[self.encode_mu(min_mu)].append([p,q])
    return cluster_map

def average_mu(self, img,cluster_map):
    new_cluster = {}
    new_mu = []
    for mu in cluster_map:
        r = 0
        g = 0
        b = 0
        for pixel in cluster_map[mu]:
            r += img[pixel[0]][pixel[1]][0]
            g += img[pixel[0]][pixel[1]][1]
            b += img[pixel[0]][pixel[1]][2]

        cluster_size = len(cluster_map[mu])

        if cluster_size > 0:
            r = float(r/cluster_size)
            g = float(g/cluster_size)
            b = float(b/cluster_size)
            new_mu.append([r,g,b])
            new_cluster[self.encode_mu([r,g,b])] = cluster_map[mu]
        else:
            new_mu.append(self.decode_mu(mu))
            new_cluster[mu] = cluster_map[mu]
    return new_mu, new_cluster

def task4(self, img_name, k):
    img = cv2.imread(img_name)
    new_img = cv2.imread(img_name)
    print(img.shape)
```

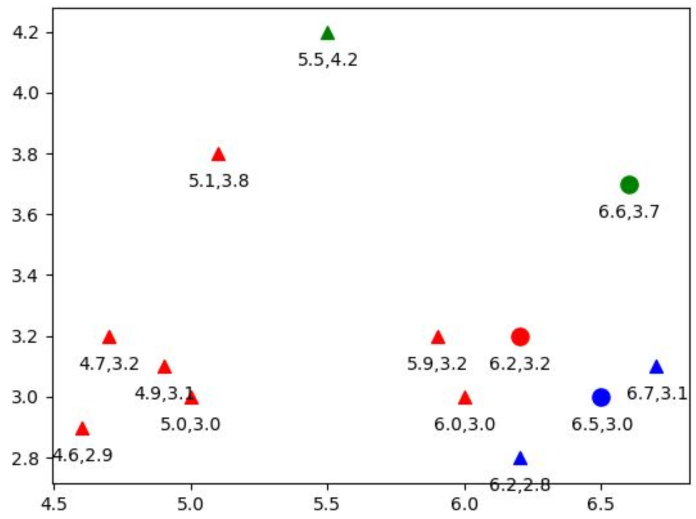
```
       mu_list = self.generate_random_mu(img.shape, k)

       cnt = 50
       while cnt > 0:
           cnt -= 1
           cluster_map = self.cluster(mu_list, img)
           new_mu, cluster_map = self.average_mu(img,cluster_map)
           flag = 0
           for o_m, n_m in zip(mu_list,new_mu):
               dis = self.ecludien_distance(o_m,n_m)
               if abs(o_m[0] - n_m[0]) < 2 and abs(o_m[1] - n_m[1]) < 2 and abs(o_m[2] - n_m[2]) < 2:
                   flag += 1

           mu_list = new_mu
           if flag >= len(mu_list)/3:
               break
       print(flag,cluster_map.keys())
       for i,cluster in enumerate(cluster_map):
           clu = self.decode_mu(cluster)
           #clu = np.dot([20,20,20],i*3)
           print(clu)
           for pixel in cluster_map[cluster]:
               new_img[pixel[0]][pixel[1]] = clu

       cv2.imwrite(r'data\task3_baboon_'+str(k)+'.jpg', new_img)

   def start(self):
       point_list = [[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]]
       MU_point = [[6.2, 3.2], [6.6, 3.7], [6.5, 3.0]]
       cluster_map = self.task1(MU_point, point_list)
       print("--"*5,"task 3.1 completed","--"*5)
       new_mu = self.task2(cluster_map)
       print("--" * 5, "task 3.2 completed", "--"*5)
       self.task3(new_mu,point_list)
       print("--" * 5, "task 3.3 completed", "--"*5)
       # color quantization
       self.task4(r"data\baboon.jpg", 3)
       self.task4(r"data\baboon.jpg", 5)
       self.task4(r"data\baboon.jpg", 10)
       self.task4(r"data\baboon.jpg", 20)
       print("--" * 5, "task 3.4 completed", "--" * 5)


def main():
   kmc = KMeansClustering()
   kmc.start()


if __name__ == '__main__':
   main()
```

Result

3.1 Classify points

1. Create dictionary with mu as key
2. Classify points closer to mu and append in dictionary[mu]
3. Plot mu and points in cluster with respective color of mu

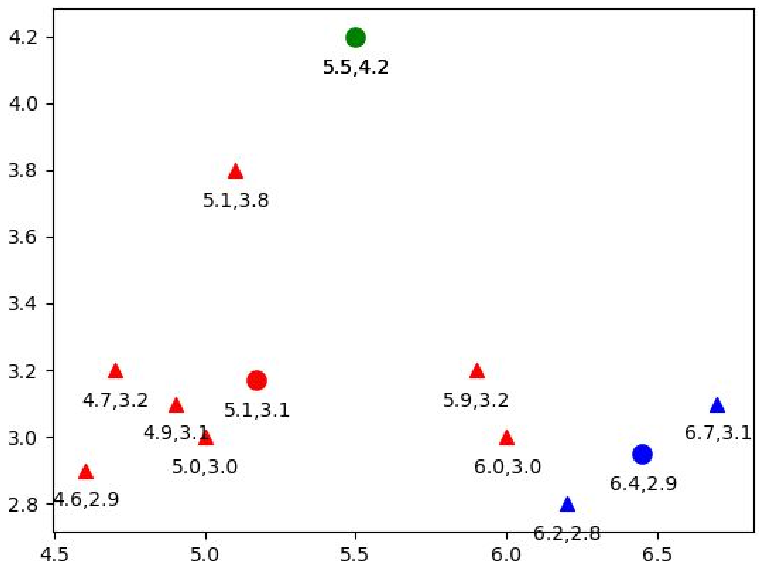
Classification vector: [1, 1, 3, 1, 2, 1, 1, 3, 1, 1]



3.2 Recompute mu

1. Use dictionary(hashTable) from 3.1 and average x,y value of all points in dictionary[mu] for each mu
2. Append avg x,yas new mu

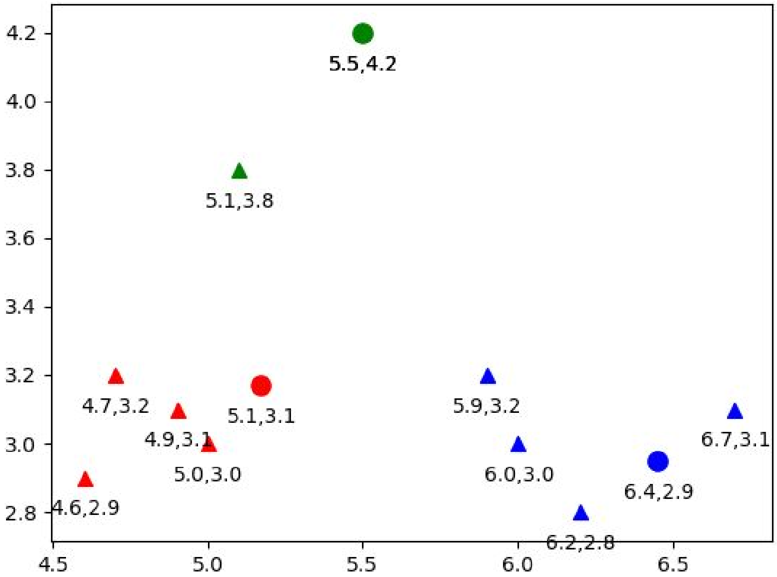
Updated mu : [[5.171428571428572, 3.1714285714285713], [5.5, 4.2], [6.45, 2.95]]



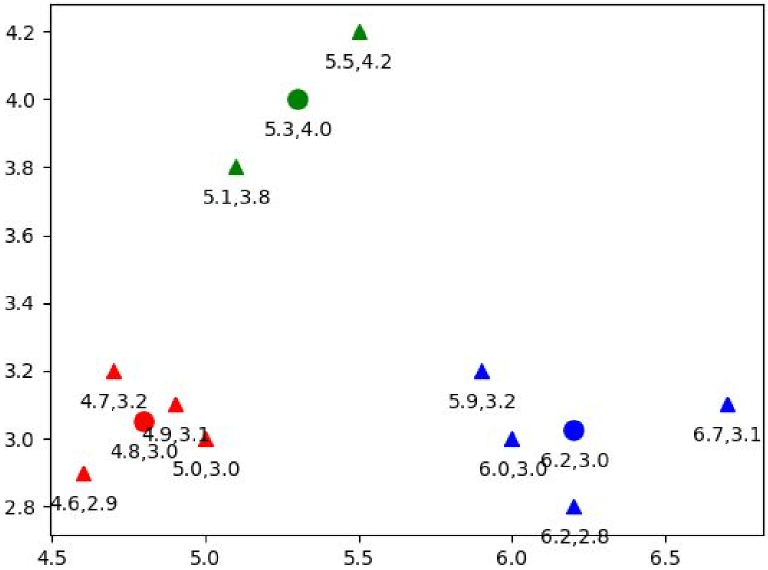
3.3 Second iteraion

1. Repeat steps from 3.1 & 3.2 (as second iteration) with new mu from 3.2

Classification vector : [3, 1, 3, 1, 2, 1, 1, 3, 2, 3]

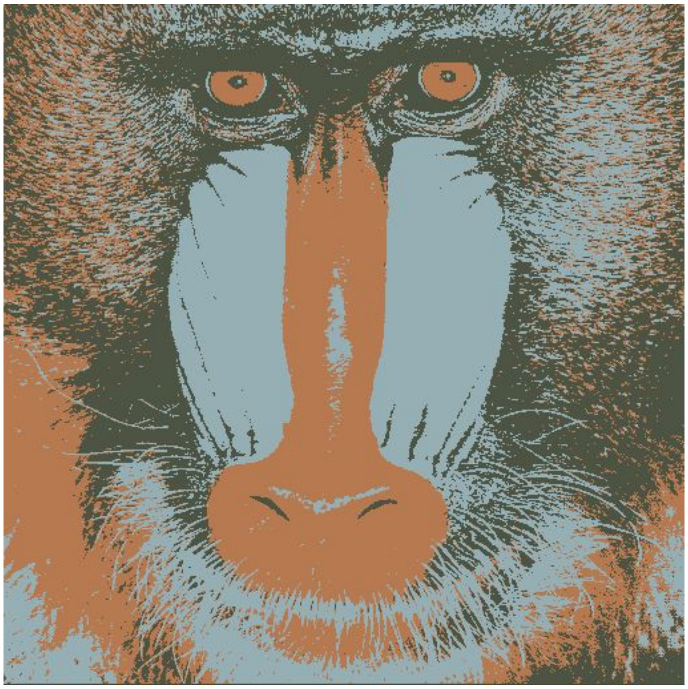


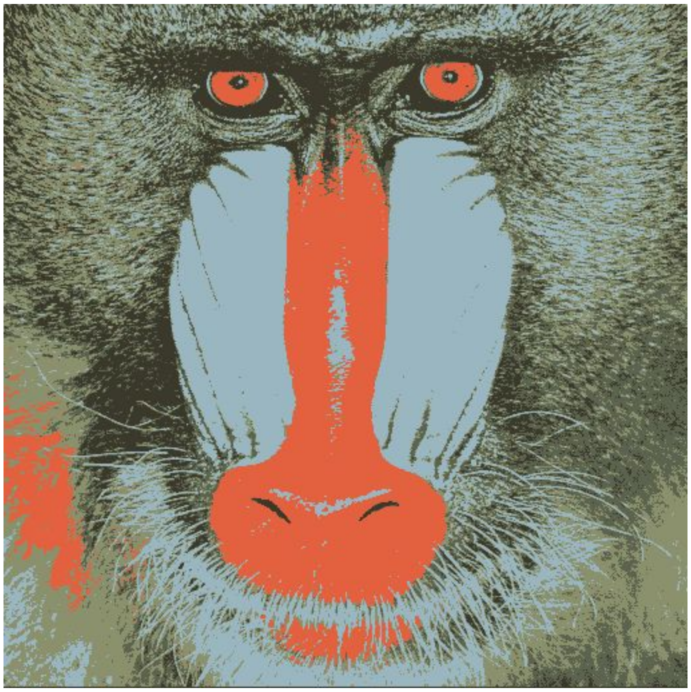
Updated mu : [[4.800000000000001, 3.05], [5.3, 4.0], [6.2, 3.025]]

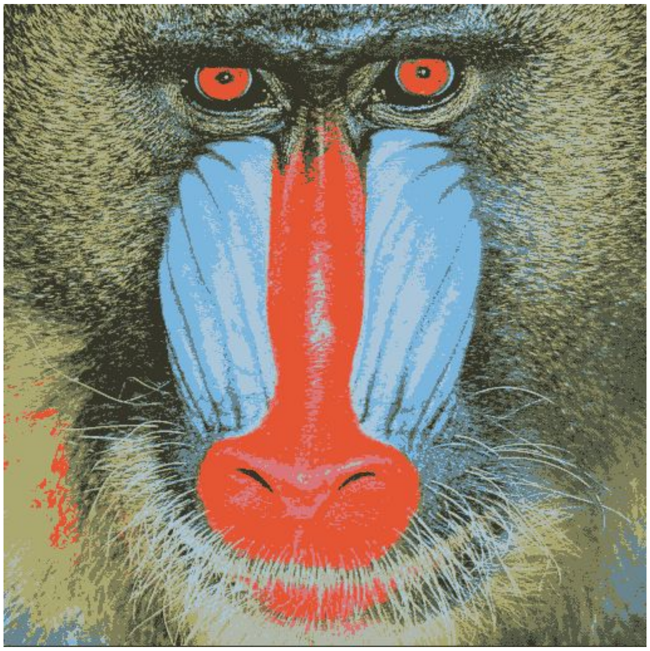


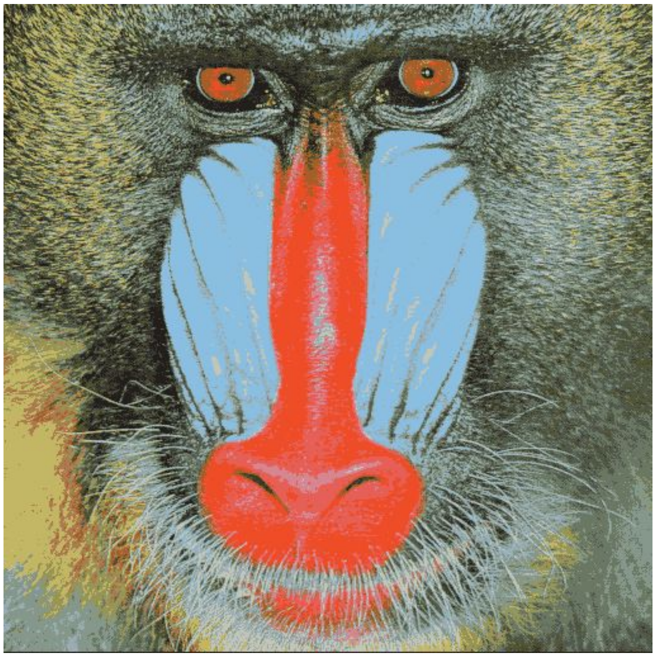
3.4 Color Quantization

1. Create list of random mu
2. Cluster each pixel with its rgb value to rgb value of mu
3. Update mu with averaging rgb value from clusters formed in step2.
4. Repeat 2,3 till mu gets stable
5. Update pixels in each cluster with its rgb value of mu.









Reference:

1. https://docs.opencv.org/3.4/d7/dff/tutorial_feature_homography.html
2. http://www.cs.toronto.edu/~jepson/csc420/notes/epiPolarGeom.pdf
3. https://docs.opencv.org/3.4.3/da/de9/tutorial_py_epipolar_geometry.html
4. https://www.programcreek.com/python/example/89407/cv2.FM_RANSAC
5. https://www.kaggle.com/asymptote/homography-estimate-stitching-two-imag
6. https://matplotlib.org/api/markers_api.html#module-matplotlib.markers