CSE 573 - Project 2

Amlan Gupta (#50288686)

November 2018

1 Image Features and Homography

```
def perform_sift(img):

        sift_img = np.zeros(img.shape)


        sift = cv2.xfeatures2d.SIFT_create()

        kp, desc = sift.detectAndCompute(img, None)
        sift_img = cv2.drawKeypoints(img, kp, sift_img)

        return sift_img, kp, desc

def find_knn_match(img_1, img_2, kp_1, kp_2, desc_1, desc_2):

        bf = cv2.BFMatcher()
        matches = bf.knnMatch(desc_1,desc_2, k=2)

        # Apply ratio test
        good_matches = []
        pts1 = []
        pts2 = []
        for m,n in matches:
            if m.distance < 0.75*n.distance:
                good_matches.append(m)
                pts2.append(kp_2[m.trainIdx].pt)
                pts1.append(kp_1[m.queryIdx].pt)


        knn_matched_img = np.zeros(img_1.shape)
        knn_matched_img = cv2.drawMatches(img_1, kp_1, img_2, kp_2, good_matches, knn_matched_img, flags=2)
        return knn_matched_img, good_matches, pts1, pts2


def find_homography_and_match_images(good_matches, kp_1, kp_2, img_1, img_2):

        src_pts = np.float32([ kp_1[m.queryIdx].pt for m in good_matches ]).reshape(-1,1,2)
        dst_pts = np.float32([ kp_2[m.trainIdx].pt for m in good_matches ]).reshape(-1,1,2)

        M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
        matchesMask = mask.ravel()
        good_matches_np = np.asarray(good_matches)


        matchesMask = matchesMask[mask.ravel()==1]
        good_matches_np = good_matches_np[mask.ravel()==1]

        randIndx = np.random.randint(low=0, high=good_matches_np.shape[0], size=10)

        good_matches_np = good_matches_np[randIndx]
```

```
        matchesMask = matchesMask[randIndx]


        draw_params = dict(matchColor = (0, 0, 255),
                    singlePointColor = None,
                    matchesMask = matchesMask.tolist(),
                    flags = 2)


        matches_img = cv2.drawMatches(img_1, kp_1, img_2, kp_2, good_matches_np.tolist(), None, **draw_params)
        return M, matches_img



def do_image_stitching(img_1, img_2, M):

        (h1, w1) = img_1.shape[:2]
        (h2, w2) = img_2.shape[:2]

        #remap the coordinates of the projected image onto the panorama image space
        top_left = np.dot(M,np.asarray([0,0,1]))
        top_right = np.dot(M,np.asarray([w2,0,1]))
        bottom_left = np.dot(M,np.asarray([0,h2,1]))
        bottom_right = np.dot(M,np.asarray([w2,h2,1]))

        #normalize
        top_left = top_left/top_left[2]
        top_right = top_right/top_right[2]
        bottom_left = bottom_left/bottom_left[2]
        bottom_right = bottom_right/bottom_right[2]



        pano_left = int(min(top_left[0], bottom_left[0], 0))
        pano_right = int(max(top_right[0], bottom_right[0], w1))
        W = pano_right - pano_left

        pano_top = int(min(top_left[1], top_right[1], 0))
        pano_bottom = int(max(bottom_left[1], bottom_right[1], h1))
        H = pano_bottom - pano_top

        size = (W, H)

        # offset of first image relative to panorama
        X = int(min(top_left[0], bottom_left[0], 0))
        Y = int(min(top_left[1], top_right[1], 0))
        offset = (-X, -Y)

        panorama = np.zeros((size[1], size[0]), np.uint8)

        (ox, oy) = offset

        translation = np.matrix([
                                        [1.0, 0.0, ox],
                                        [0, 1.0, oy],
                                        [0.0, 0.0, 1.0]
                                        ])


        M = translation * M

        cv2.warpPerspective(img_1, M, size, panorama)

```

```
        panorama[oy:h1+oy, ox:ox+w1] = img_2

        return panorama


def image_feature_and_homography(mountain_1_img, mountain_2_img):

        # task 1.1

        kp_mountain_1_img, kp_1, desc_1 = perform_sift(mountain_1_img)
        write_image(kp_mountain_1_img, 'task1_sift1.jpg')


        kp_mountain_2_img, kp_2, desc_2 = perform_sift(mountain_2_img)
        write_image(kp_mountain_2_img, 'task1_sift2.jpg')


        # task 1.2

        knn_matched_img, good_matches, _, _ = find_knn_match(mountain_1_img, mountain_2_img,
        kp_1, kp_2, desc_1, desc_2)
        write_image(knn_matched_img, 'task1_matches_knn.jpg')


        #task 1.3, task 1.4

        h_matrix, task1_matches = find_homography_and_match_images(good_matches, kp_1, kp_2,
        mountain_1_img, mountain_2_img)
        print(h_matrix)
        write_image(task1_matches, 'task1_matches.jpg')

        # task 1.5
        panorama = do_image_stitching(mountain_1_img, mountain_2_img, h_matrix)
        write_image(panorama,'task1_pano.jpg')




def main():

        mountain_1_img = cv2.imread(SOURCE_FOLDER + "mountain1.jpg", 0)
        mountain_2_img = cv2.imread(SOURCE_FOLDER + "mountain2.jpg", 0)

        image_feature_and_homography(mountain_1_img, mountain_2_img)



main()
```

1. **Given two images mountain1.jpg and mountain2.jpg, extract SIFT features and draw the keypoints for both images. Include the resulting two images (task1_sift1.jpg, task1_sift2.jpg) in the report.**

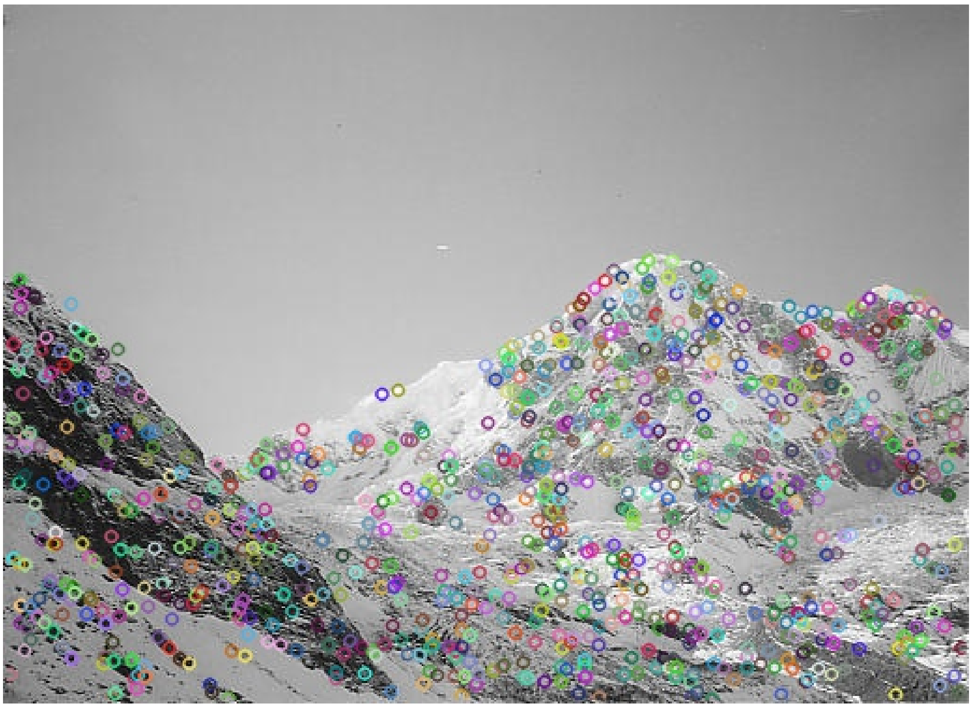


Figure 1: Keypoints for **mountain1.jpg**

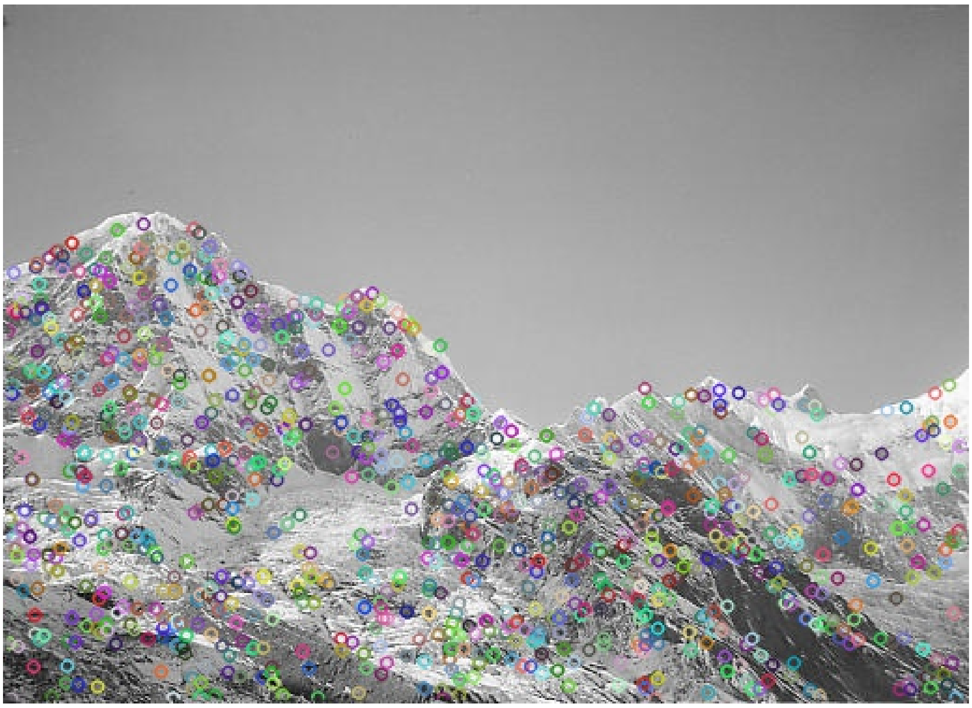


Figure 2: Keypoints for **mountain2.jpg**

2. **Match the keypoints using k-nearest neighbour (k=2), i.e., for a keypoint in the left image, finding the best 2 matches in the right image. Filter good matches satisfy m.distance <0.75 n.distance, where m is the first match and n is the second match. Draw the match image using cv2.drawMatches for all matches (your match image should contain both inliers and outliers). Include the result image (task1_matches_knn.jpg) in the report.**

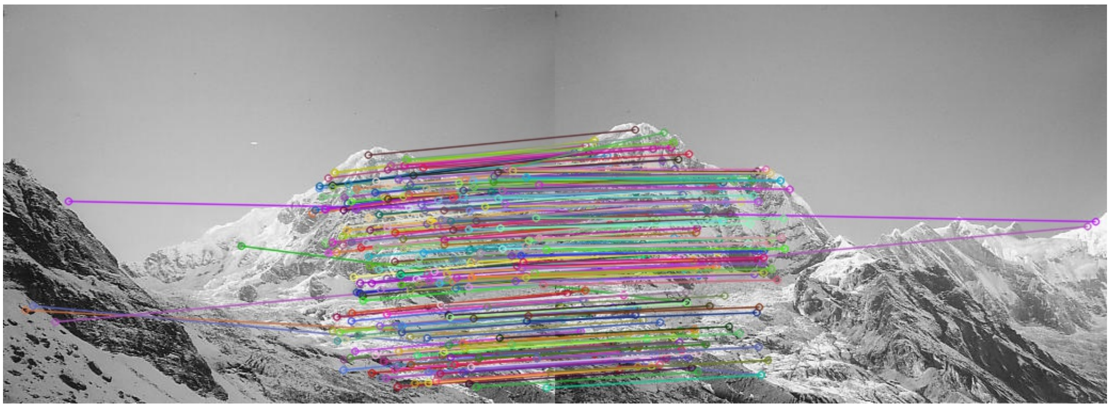


Figure 3: Matched keypoints using k-nearnest neighbour (k =2)

3. **Compute the homography matrix H (with RANSAC) from the first image to the second image. Include the matrix values in the report.**

$$\begin{bmatrix} 1.589302 & -0.291559 & -395.969265 \\ 0.449424 & 1.431109 & -190.613988 \\ 0.001213 & -0.000063 & 1.000000 \end{bmatrix}$$

4. **Draw the match image for around 10 random matches using only inliers. Include the result image (task1_matches.jpg) in the report.**

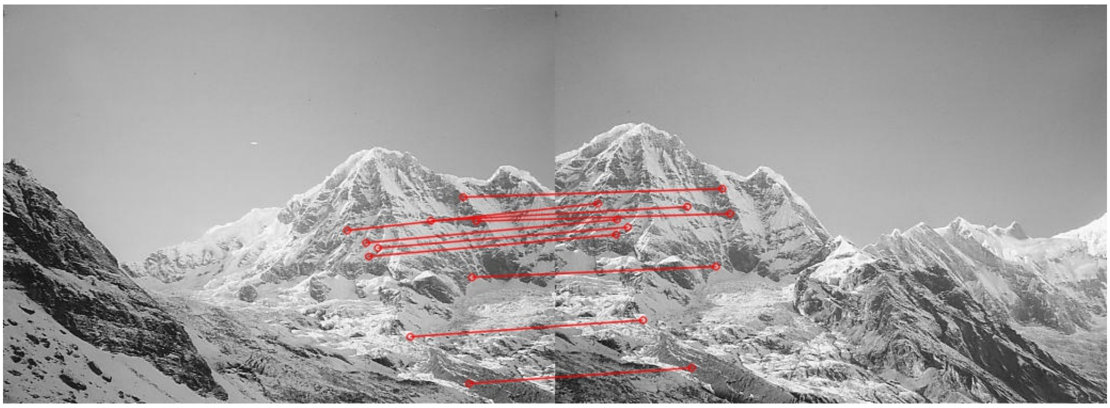


Figure 4: 10 random matches using only inliers

5. **Warp the first image to the second image using H. The resulting image should contain all pixels in mountain1.jpg and mountain2.jpg. Include the result image (task1_pano.jpg) in the report.**

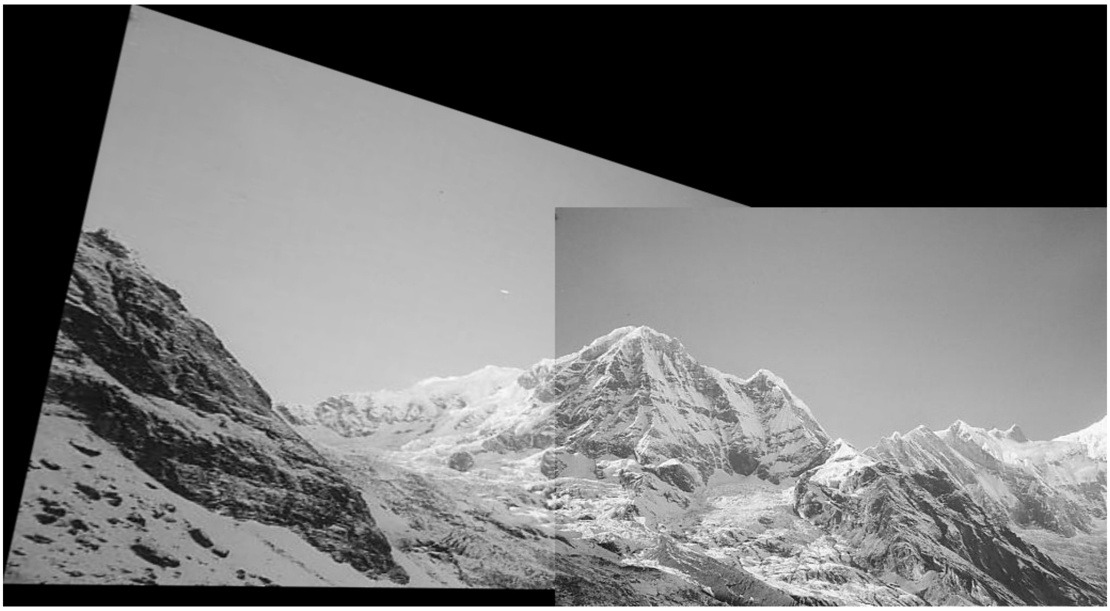


Figure 5: Warped image using Homography matrix

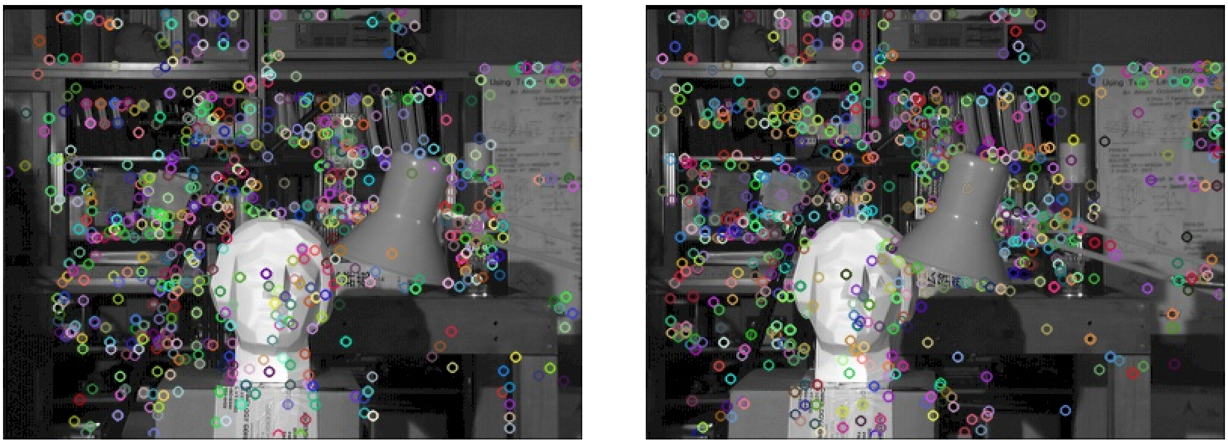
2 Epipolar Geometry

```
def perform_sift(img):

        sift_img = np.zeros(img.shape)


        sift = cv2.xfeatures2d.SIFT_create()

        kp, desc = sift.detectAndCompute(img, None)
        sift_img = cv2.drawKeypoints(img, kp, sift_img)

        return sift_img, kp, desc

def find_knn_match(img_1, img_2, kp_1, kp_2, desc_1, desc_2):

        bf = cv2.BFMatcher()
        matches = bf.knnMatch(desc_1,desc_2, k=2)

        # Apply ratio test
        good_matches = []
        pts1 = []
        pts2 = []
        for m,n in matches:
            if m.distance < 0.75*n.distance:
                good_matches.append(m)
                pts2.append(kp_2[m.trainIdx].pt)
                pts1.append(kp_1[m.queryIdx].pt)


        knn_matched_img = np.zeros(img_1.shape)
        knn_matched_img = cv2.drawMatches(img_1, kp_1, img_2, kp_2, good_matches, knn_matched_img, flags=2)
        return knn_matched_img, good_matches, pts1, pts2

```

```python


def drawlines(img1,img2,lines,pts1,pts2, color):
        r,c = img1.shape
        img1 = cv2.cvtColor(img1,cv2.COLOR_GRAY2BGR)
        img2 = cv2.cvtColor(img2,cv2.COLOR_GRAY2BGR)
        for r,pt1,pt2,colr in zip(lines,pts1,pts2,color):
                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), tuple(colr),1)
                img1 = cv2.circle(img1,tuple(pt1),5,tuple(colr),-1)
                img2 = cv2.circle(img2,tuple(pt2),5,tuple(colr),-1)
        return img1,img2

def find_disparity_map(imgL, imgR):

        window_size = 3
        min_disp = 16
        num_disp = 64-min_disp
        stereo = cv2.StereoSGBM_create(minDisparity = min_disp,
                numDisparities = num_disp,
                blockSize = 9,
                P1 = 8*3*window_size**2,
                P2 = 32*3*window_size**2,
                disp12MaxDiff = 1,
                uniquenessRatio = 10,
                speckleWindowSize = 100,
                speckleRange = 32
        )

        disp = stereo.compute(imgL, imgR).astype(np.float32) / 16.0
        disp = (disp-min_disp)/num_disp

        disp = disp*300


        write_image(disp,'task2_disparity.jpg')


def epipolar_geometry(tsucuba_left_img, tsucuba_right_img):

        # task 2.1

        kp_tsucuba_left_img, kp_1, desc_1 = perform_sift(tsucuba_left_img)
        write_image(kp_tsucuba_left_img, 'task2_sift1.jpg')

        kp_tsucuba_right_img, kp_2, desc_2 = perform_sift(tsucuba_right_img)
        write_image(kp_tsucuba_right_img, 'task2_sift2.jpg')

        # task 2.2, 2.3

        knn_matched_img, good_matches, pts1, pts2 = find_knn_match(tsucuba_left_img, tsucuba_right_img,
                        kp_1, kp_2, desc_1, desc_2)

        pts1 = np.int32(pts1)
        pts2 = np.int32(pts2)
        F, mask = cv2.findFundamentalMat(pts1,pts2,cv2.FM_RANSAC)
        pts1 = pts1[mask.ravel()==1]
        pts2 = pts2[mask.ravel()==1]
        print(F)


        randIndx = np.random.randint(low=0, high=pts1.shape[0], size=10)
```

```
        pts1 = pts1[randIndx]
        pts2 = pts2[randIndx]


        color = np.random.randint(0,255, size=(10, 3)).tolist()

        lines1 = cv2.computeCorrespondEpilines(pts2.reshape(-1,1,2), 2, F)
        lines1 = lines1.reshape(-1,3)
        tsucuba_left_ep , _ = drawlines(tsucuba_left_img, tsucuba_right_img, lines1, pts1, pts2, color)

        write_image(tsucuba_left_ep, 'task2_epi_left.jpg')

        lines2 = cv2.computeCorrespondEpilines(pts1.reshape(-1,1,2), 1, F)
        lines2 = lines2.reshape(-1,3)
        tsucuba_right_ep, _ = drawlines(tsucuba_right_img,tsucuba_left_img,lines2, pts2, pts1, color)

        write_image(tsucuba_right_ep, 'task2_epi_right.jpg')

        # task 2.4

        find_disparity_map(tsucuba_left_img, tsucuba_right_img)




def main():

        tsucuba_left_img = cv2.imread(SOURCE_FOLDER + "tsucuba_left.png", 0)
        tsucuba_right_img = cv2.imread(SOURCE_FOLDER + "tsucuba_right.png", 0)

        epipolar_geometry(tsucuba_left_img, tsucuba_right_img)




main()
```

1. **Given two images tsucuba_left.png and tsucuba_right.png, do the same process for Task 1.1 and 1.2. Include the three images (2 for task 1.1 and 1 for task 1.2) (task2_sift1.jpg, task2_sift2.jpg, task2_matches_knn.jpg) in the report.**



(a) for tsucuba_left.png

(b) for tsucuba_right.png

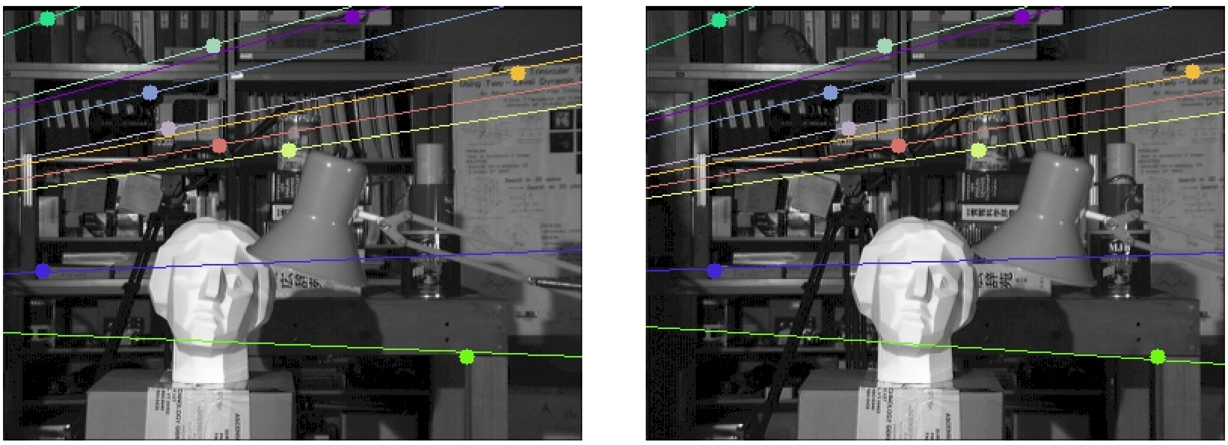
Figure 6: Detected Keypoints

Figure 7: Matched keypoints using k-nearnest neighbour (k =2)

2. **Computer the fundamental matrix F (with RANSAC). Include the matrix values in the report.**

$$\begin{bmatrix} 3.88435667e^{-06} & -2.28e^{-04} & 4.41240231e^{-02} \\ 2.42e^{-04} & -1.90e^{-05} & 9.95e^{-02} \\ -4.58e^{-02} & -1.02e^{-01} & 1 \end{bmatrix}$$

3. **Randomly select 10 inlier match pairs. For each keypoint in the left image, compute the epiline and draw on the right image. For each keypoint in the right image, compute the epiline and draw on the left image [Using different colors for different match pairs, but the same color for epilines on the left and right images with the same match pair.] Include two images (task2_epi_right.jpg, task2_epi_left.jpg) with epilines in the report.**



(a) task2_epi_right.jpg

(b) task2_epi_left.jpg

Figure 8: Epilines

4. **Compute the disparity map for tsucuba_left.png and tsucuba_right.png. Include the disparity image (task2_disparity.jpg) in the report.**

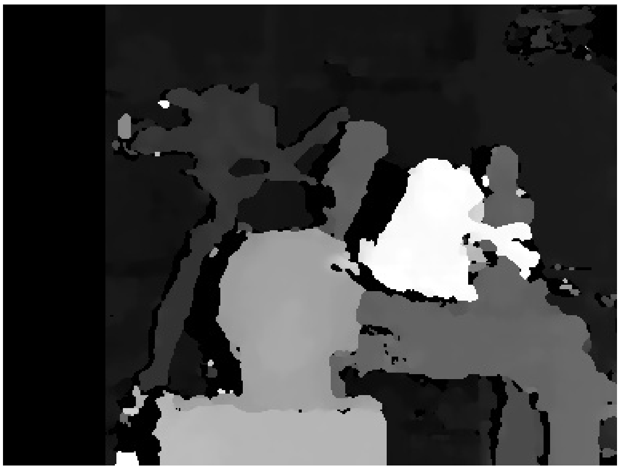


Figure 9: Disparity map for tsucuba_left.png and tsucuba_right.png

3 K-means Clustering

```
def measure_euclidean_distance(pt1 , pt2):
    dis = math.sqrt(((pt1[0] - pt2[0])**2) + ((pt1[1] - pt2[1])**2))
    return dis

def calculate_distances_from_centroids(mu, mu_c, X, it_n):
    cluster_c = []

    for pt in X:

        isFirst = True
        for m, mc in zip(mu,mu_c):
            if isFirst == True:
                d = measure_euclidean_distance(pt,m)
                c = mc
                isFirst = False
            elif (measure_euclidean_distance(pt,m) < d):
                d = measure_euclidean_distance(pt,m)
                c = mc

        cluster_c.append(c)

    fig = plt.figure()
    ax = fig.add_subplot(111)
    plt.scatter(X[:,0], X[:,1], marker= "^", facecolors="None", edgecolors= cluster_c)
    plt.scatter(mu[:,0], mu[:,1], c= mu_c)
    for xy in zip(mu[:,0], mu[:,1]):
            ax.annotate('(%.2f, %.2f)' % xy, xy=xy, textcoords='data')
    # plt.show()
    plt.savefig(OUTPUT_FOLDER + 'task3_iter'+str(it_n+1)+'_a.jpg')
    plt.clf()

    return np.asarray(cluster_c)

def vector_classification():

        X = np.array([
        [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 = np.array([[6.2, 3.2], [6.6, 3.7], [6.5, 3.0]])
        mu_c = ['r','g','b']

        for i in range(2):

            cluster_c = calculate_distances_from_centroids(mu, mu_c, X, i)

            print(cluster_c)

            clusters = []
            for mc in zip(mu_c):
                clusters.append(X[cluster_c == mc])

            mu = []
            for clus in clusters:
                mu.append(np.mean(clus, axis=0))

            mu = np.asarray(mu)
            print(mu)


            fig = plt.figure()
            ax = fig.add_subplot(111)
            plt.scatter(X[:,0], X[:,1], marker= "^", facecolors="None", edgecolors= cluster_c)
            plt.scatter(mu[:,0], mu[:,1], c= mu_c)
            for xy in zip(mu[:,0], mu[:,1]):
                    ax.annotate('(%.2f, %.2f)' % xy, xy=xy, textcoords='data')
            plt.savefig(OUTPUT_FOLDER + 'task3_iter'+str(i+1)+'_b.jpg')
            plt.clf()



def measure_euclidean_distance_3d(color1 , color2):
    try:
        dis = ((color1[0] - color2[0])**2) + ((color1[1] - color2[1])**2) + ((color1[2] - color2[2])**2)
    except:
        print(color1, color2)
    return dis


def calculate_distances_from_centroids_3d(mu, mu_c, image):
    cluster_c = np.zeros([image.shape[0],image.shape[1]])

    h, w, l = image.shape


    for i in range(h):
        for j in range(w):
            pixel = image[i][j]

            isFirst = True
            for m, mc in zip(mu,mu_c):
                if isFirst == True:
                    d = measure_euclidean_distance_3d(pixel,m)
```

```
                    c = mc
                    isFirst = False
                elif (measure_euclidean_distance_3d(pixel,m) < d):
                    d = measure_euclidean_distance_3d(pixel,m)
                    c = mc

            cluster_c[i][j] = c


    return np.asarray(cluster_c)


def color_quantization(image):


        k = [3, 5, 10, 20]
        # k = [20]

        for kval in k:

            mu = np.random.randint(0,255, size=(kval, 3))
            mu_indx = np.random.randint(0,image.shape[0], size=(kval, 2))
            mu = []

            for i in range(mu_indx.shape[0]):
                    mu.append(image[mu_indx[i][0]][mu_indx[i][1]])
            mu = np.asarray(mu).astype(float)

            mu_c = np.arange(kval)


            for zz in range(30):

                cluster_c = calculate_distances_from_centroids_3d(mu, mu_c, image)

                h, w, l = image.shape
                clusters = []


                for mc in zip(mu_c):
                    clustered_img_np = []
                    for i in range(h):
                        for j in range(w):
                            if(cluster_c[i][j] == mc):
                                clustered_img_np.append(image[i][j])
                    clusters.append(np.asarray(clustered_img_np))

                mu = []
                for clus in clusters:
                    c_mean = np.nanmean(clus, axis=0)
                    mu.append(c_mean)

                mu = np.asarray(mu)

                h, w, l = image.shape
                output_img = np.zeros([h,w,l])

                for i in range(h):
                    for j in range(w):
                        index =int(cluster_c[i][j])
                        output_img[i][j] = mu[index]

                op = output_img.astype(int)
```

```
                print(zz)
            write_image(op, 'task3_baboon_'+ str(kval) +'.jpg')
                # print('boboon for ' + str(kval) + 'k means generated!')



def k_means_clustering(image):

        #task 3.1, 3.2, 3.3
        vector_classification()

        #task 3.4
        color_quantization(image)



def main():


        baboon_img = cv2.imread(SOURCE_FOLDER + "baboon.jpg")
        k_means_clustering(baboon_img)

        print('DONE!!')




main()
```

1. **Classify N = 10 samples according to nearest μ_i(i = 1, 2, 3). Plot the results by coloring the empty triangles in red, blue or green. Include the classification vector and the classification plot (task3_iter1_a.jpg) in the report.**

   The classification vector is [**r r b r g r r b r r**]

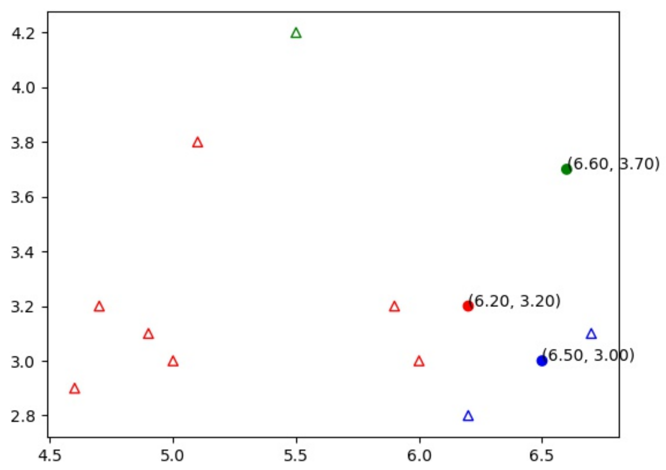


Figure 10: Classification plot for N =10 samples

2. **Recompute μ_i. Plot the updated μ_i in solid circle in red, blue, and green respectively. Include the updated μ_i values and the plot in the report (task3_iter1_b.jpg).**

   The value of updated μ_i in first iteration is:

$$\begin{bmatrix} 5.17142857 & 3.17142857 \\ 5.5 & 4.2 \\ 6.45 & 2.95 \end{bmatrix}$$

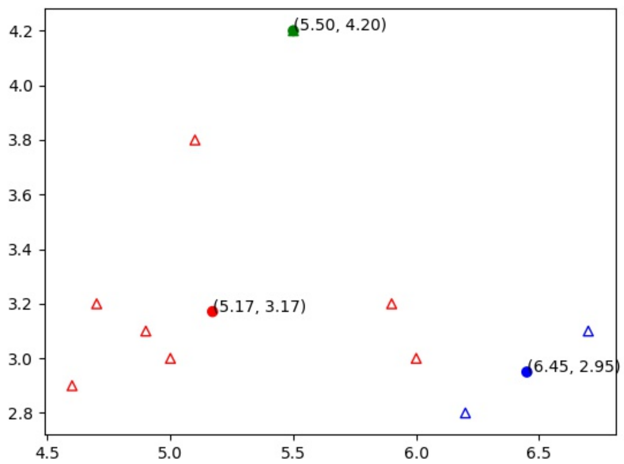


Figure 11: Updated μ_i in plot

3. **For a second iteration, plot the classification plot and updated μ_i plot for the second iteration. Include the classification vector and updated μ_i values and these two plots (task3_iter2_a.jpg, task3_iter2_b.jpg) in the report.**

   Classification vector for second iteration: [**b r b r g r r b g b**]

   The value of updated μ_i in second iteration is:

   $$\begin{bmatrix} 4.8 & 3.05 \\ 5.3 & 4.0 \\ 6.2 & 3.025 \end{bmatrix}$$

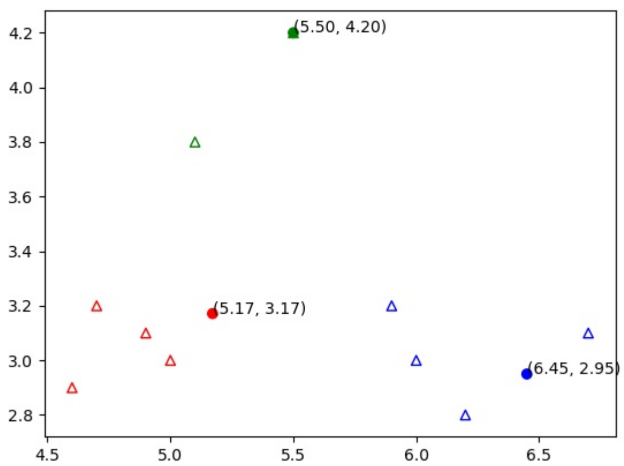


Figure 12: task3_iter2_a.jpg

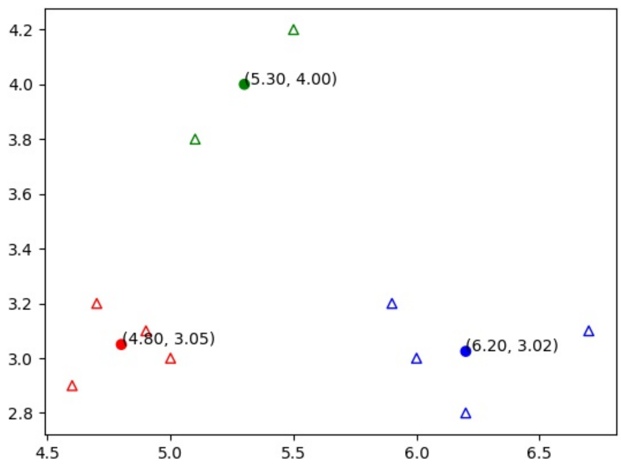


Figure 13: task3_iter2_b.jpg

4. **Apply k-means to image color quantization. Using only k colors to represent the image baboon.jpg. Include the color quantized images for k = 3, 5, 10, 20 (task3_baboon_3.jpg, task3_baboon_5.jpg, task3_baboon_10.jpg, task3_baboon_20.jpg**

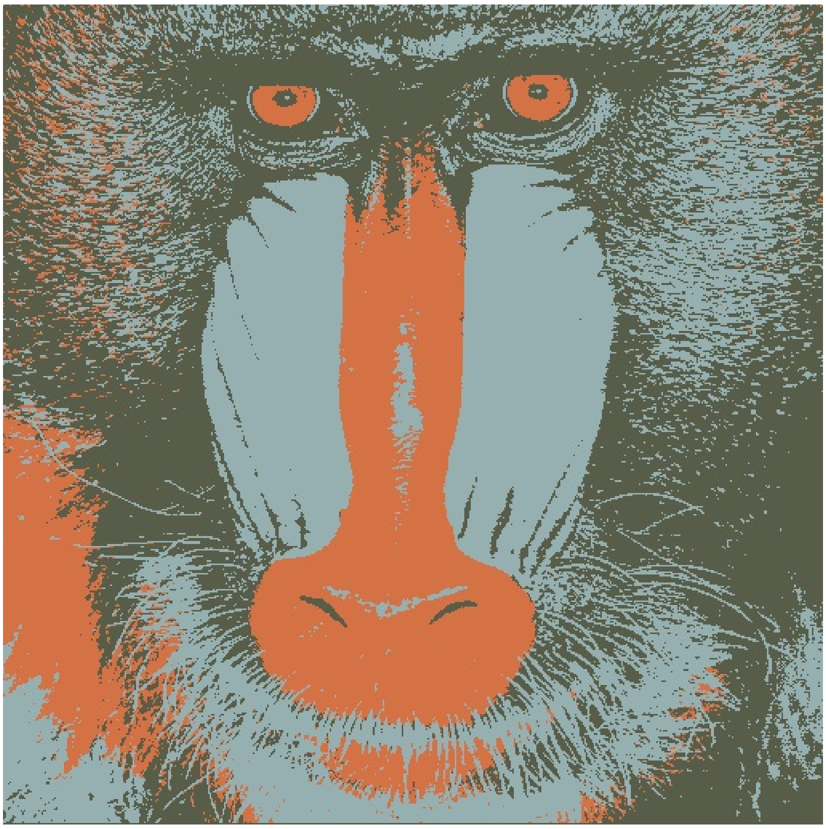


Figure 14: Color quantied baboon image for k =3

Figure 15: Color quantied baboon image for k =5

Figure 16: Color quantied baboon image for k =10

Figure 17: Color quantied baboon image for k =20