CLab-2 Report

ENGN4528

Ernest Kwan

u6381103

18/05/2020

Contents

Task-1: Harris Corner Detector

```python
"""
CLAB Task-1: Harris Corner Detector
Your name (Your uniID): Ernest Kwan (u6381103)
"""

import numpy as np
import cv2
import imageio
import matplotlib.pyplot as plt

def conv2(img, conv_filter):
    # flip the filter
    f_siz_1, f_size_2 = conv_filter.shape
    conv_filter = conv_filter[range(f_siz_1 - 1, -1, -1), :][:, range(f_siz_1 - 1, -1, -1)]
    pad = (conv_filter.shape[0] - 1) // 2
    result = np.zeros((img.shape))
    img = np.pad(img, ((pad, pad), (pad, pad)), 'constant', constant_values=(0, 0))
    filter_size = conv_filter.shape[0]
    for r in np.arange(img.shape[0] - filter_size + 1):
        for c in np.arange(img.shape[1] - filter_size + 1):
            curr_region = img[r:r + filter_size, c:c + filter_size]
            curr_result = curr_region * conv_filter
            conv_sum = np.sum(curr_result)
            result[r, c] = conv_sum

    return result

def fspecial(shape=(3, 3), sigma=0.5):
    m, n = [(ss - 1.) / 2. for ss in shape]
    y, x = np.ogrid[-m:m + 1, -n:n + 1]
    h = np.exp(-(x * x + y * y) / (2. * sigma * sigma))
    h[h < np.finfo(h.dtype).eps * h.max()] = 0
    sumh = h.sum()
    if sumh != 0:
        h /= sumh
    return h
```

```python
# Parameters, add more if needed
sigma = 2
thresh = 0.01
k = 0.04

img = cv2.imread('Harris_1.jpg')
bw = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Derivative masks
dx = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])
dy = dx.transpose()
# computer x and y derivatives of image
Ix = conv2(bw, dx)
Iy = conv2(bw, dy)

g = fspecial((max(1, np.floor(3 * sigma) * 2 + 1),
max(1, np.floor(3 * sigma) * 2 + 1)), sigma)

Iy2 = conv2(np.power(Iy, 2), g)
Ix2 = conv2(np.power(Ix, 2), g)
Ixy = conv2(Ix * Iy, g)

######################################################################
# Task: Compute the Harris Cornerness
######################################################################
def cornerness(Ix2, Iy2, Ixy, k=0.04):
    """Compute the Harris cornerness given the image derivatives Ix2, Iy2 and Ixy

    Returns:
    R : the Harris cornerness for every pixel
    """
    det_M = Ix2 * Iy2 - Ixy * Ixy
    trace_M = Ix2 + Iy2
    R = det_M - k * (trace_M ** 2)
    return R

R = cornerness(Ix2, Iy2, Ixy, k)


######################################################################
```

```python
# Task: Perform non-maximum suppression and
#       thresholding, return the N corner points
#       as an Nx2 matrix of x and y coordinates
######################################################################
def thresholding_and_nms(R, thresh, neighbour_dist=3):
    """Apply thresholding on the Harris corners to only keep values above
    thresh * max value.
    Then perform non-maximum suppression in 3x3 (default) neighbourhoods.

    Returns:
    out: a list of coordinates of the corners
    """
    thresh = thresh * R.max()
    r = neighbour_dist // 2
    out = []
    for i in range(r, R.shape[0]-r-1):
        for j in range(r, R.shape[1]-r-1):
            if R[i, j] > thresh and R[i-r:i+r+1, j-r:j+r+1].max() == R[i,j]:
                out.append([i, j])
    return out

corners = thresholding_and_nms(R, thresh, 3)
corners = np.array(corners)
img = cv2.imread('Harris_1.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
# mark crosses for visualisation
for c in corners:
    img[c[0]-1, c[1]-1] = [0,255,0]
    img[c[0]-1, c[1]+1] = [0,255,0]
    img[c[0], c[1]] = [0,255,0]
    img[c[0]+1, c[1]+1] = [0,255,0]
    img[c[0]+1, c[1]-1] = [0,255,0]


plt.figure(figsize=(8,6))
plt.imshow(img)
plt.show()

# Compare with opencv's built-in function
harris = cv2.cornerHarris(bw, 2, 3, k)
corners = thresholding_and_nms(harris, thresh, 3)
corners = np.array(corners)
img = cv2.imread('Harris_1.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
# mark crosses for visualisation
for c in corners:
    img[c[0]-1, c[1]-1] = [0,255,0]
    img[c[0]-1, c[1]+1] = [0,255,0]
    img[c[0], c[1]] = [0,255,0]
    img[c[0]+1, c[1]+1] = [0,255,0]
    img[c[0]+1, c[1]-1] = [0,255,0]

plt.figure(figsize=(8,6))
plt.imshow(img)
plt.show()
```

3. In block 5 a Gaussian filter is generated. A Gaussian window is used for Harris corner detection to give a smoothed squared derivative. This is to reduce noise as it may negatively impact corner detection.

   In block 7, we first compute the Harris cornerness using the mathematical equation

   $$R = \det M - k(\operatorname{trace} M)^2 \ ,$$

   where M is a 2x2 matrix computed from image derivatives:

   $$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

   The functions of image derivatives are already given so it is straightforward to calculate the determinant and trace of the matrix:

   $$\det M = I_x^2 I_y^2 - (I_x I_y)^2$$

   $$\operatorname{trace} M = I_x^2 + I_y^2$$

   After we obtain the Harris cornerness for every pixel, we perform non-maximum suppression and thresholding. We only keep values above a certain threshold and the points of local maxima. This filters out uncertain points and points that are too close to each other. The threshold is set to be 0.01 of the maximum value. This gives a proportional threshold that would work with different images. Then we use a 3x3 sliding window to get the local maxima values. Lastly the corners are returned to a list, and a small green cross is marked for every corner on the image for visualisation.

4. The results of applying our function on the four test images are displayed below.

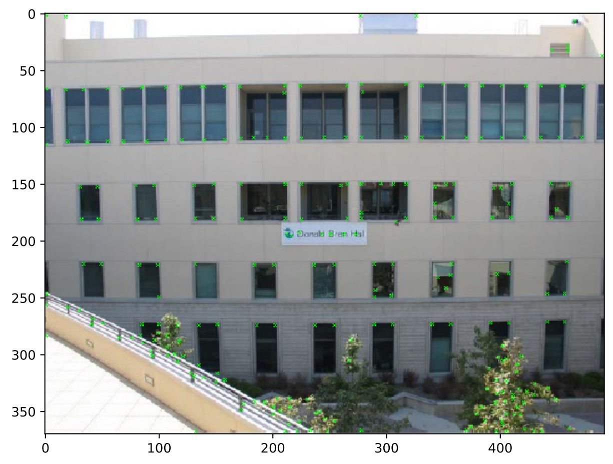


Figure 1.1a

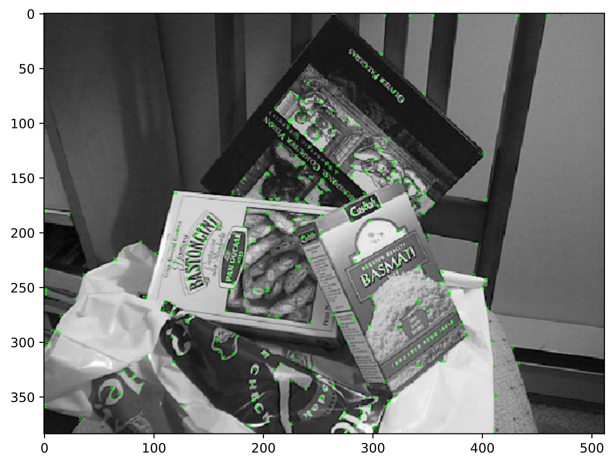


Figure 1.1b

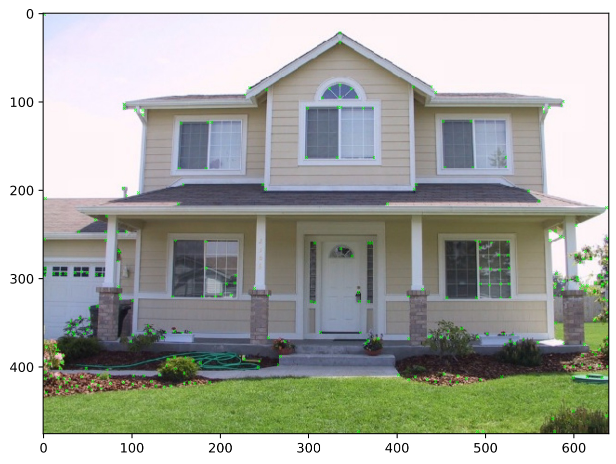


Figure 1.1c

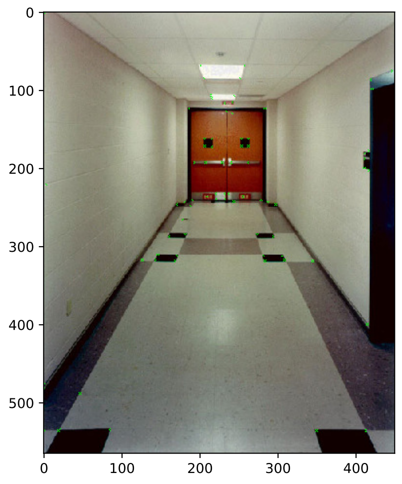


Figure 1.1d

5. The results of applying the built-in function cv2.cornerHarris() on the four test images are displayed below.

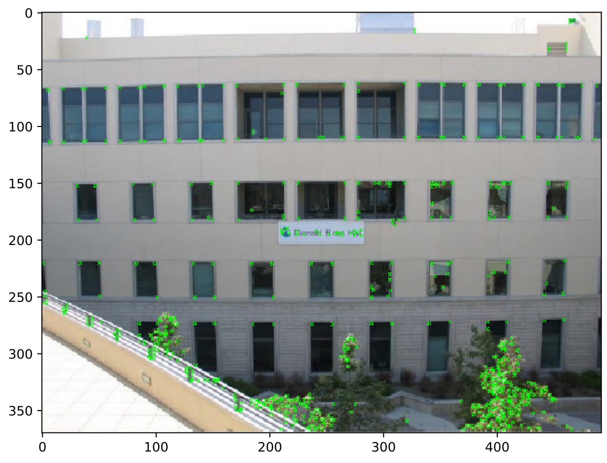


Figure 1.2a

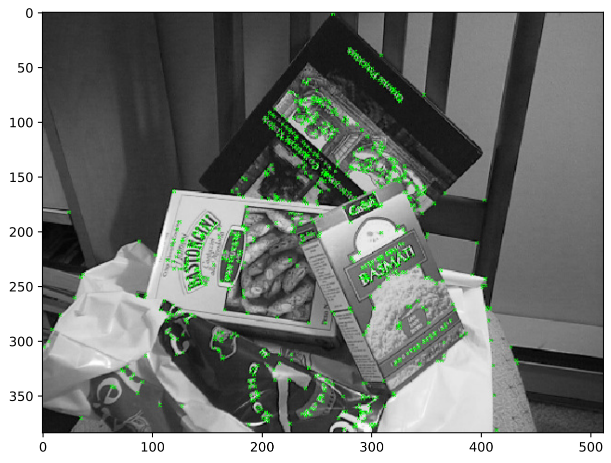


Figure 1.2b

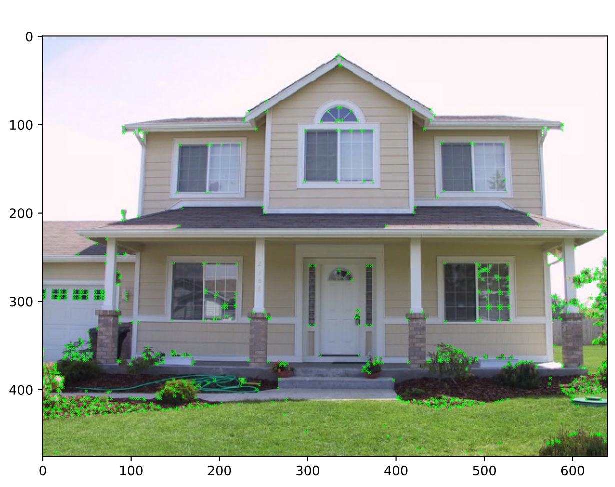


Figure 1.2c

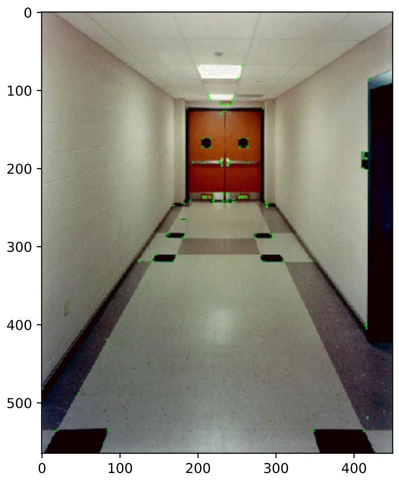


Figure 1.2d

The same parameters are used and the same thresholding and non-maximum suppression function is applied. We can see that the results are similar. However, the opencv's implementation of Harris corner seems to be able to find more corners even with non-maximum suppression. One of the factors may be the difference in the derivative filters. This will result in a different gradient covariance matrix. The window function is another factor. The size of the window and the sigma of the Gaussian function will affect the way it looks at the image. Lastly, different thresholds and constants (k) may be used to affect the number of corners returned.

Task-2: K-Means Clustering and Color Image Segmentation

1. To implement the K-means function, first we pick a value of k and use k random data points to be the initial cluster centroids.

   ```
   k = 4
   rand = np.random.choice(X.shape[0], k, replace=False)
   ```

   where `X` is the array of data points to be processed. A similarity measure between data points needs to be defined and Euclidean distance is used here. In the assignment step, we calculate the distances between data points and centroids and assign each data point to the closest centroid.

   ```
   dist = np.linalg.norm(X[i]-C, axis=1)
   S[i] = np.argmin(dist)
   ```

   where `C` are the cluster centroids and `S[i]` indicates which cluster data point `i` is assigned to. Then we re-compute the centres of the clusters using the mean of the assigned data points in the update step. `C[i]` is the new value of the ith cluster.

   ```
   C[i] = X[np.where(S==i)].mean(axis=0)
   ```

   The two steps are repeated until there is no further change.

   ```
   while(i>0):
           i -= 1
           S = assignment_step(C, X)
           C = update_step(C, X, S)
           new_S = assignment_step(C, X)
           if(np.array_equal(new_S, S)): # converge, return early
               return C, new_S
       return C, new_S
   ```

   To demonstrate our function, some data points are generated using `np.random.multivariate_normal` such that there appears to be 4 Gaussian distributions. The results of applying our K-means function on the dataset are shown below.

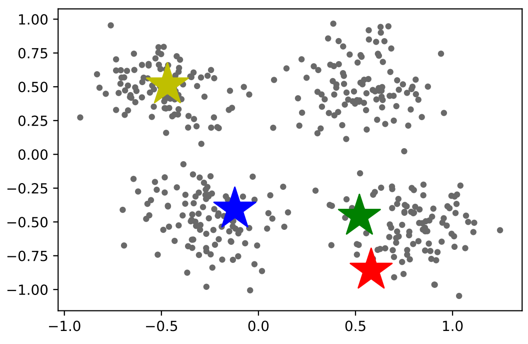


Figure 2.1a: Initial clusters

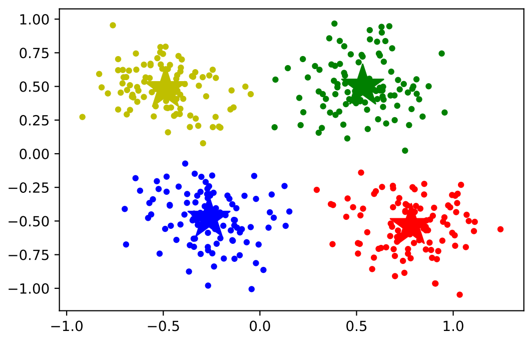


Figure 2.1b: After perfoming K-means clustering

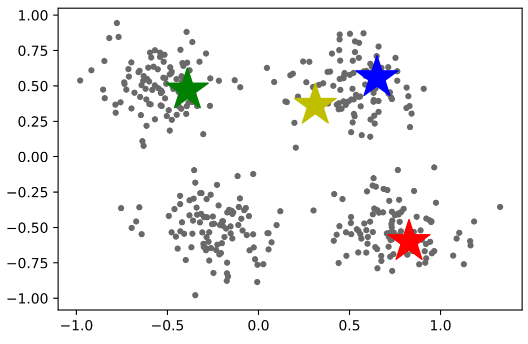


Figure 2.1c: Initial clusters

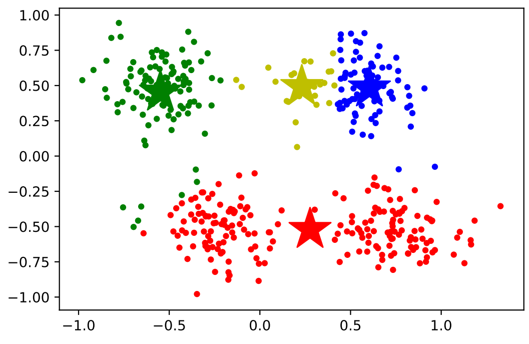


Figure 2.1d: After performing K-means clustering

As we can see the results are dependent on the initial locations of the cluster centroids. In Figures 2.1c and 2.1d, because the yellow cluster is initially very close to the blue cluster, they ended up both in the top right corner and the red cluster covered all the data points on the bottom.

2. We apply the K-means function to colour image segmentation on the two images below. The pixels are represented in L*a*b* colour space[1]. Some functions from part 1 are modified for our task. For example, the functions work for 3-dimensional arrays instead of 2 and the condition for k-means to converge is changed. Instead of converging when there is no change at all, we allow it to converge if there is less than 1% difference. If not, it will not converge if 1 pixel gets updated and the run time will be extremely long.

[1]`https://en.wikipedia.org/wiki/CIELAB_color_space`

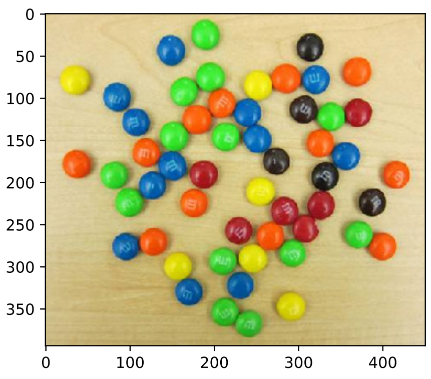


Figure 2.2a: mandm.png

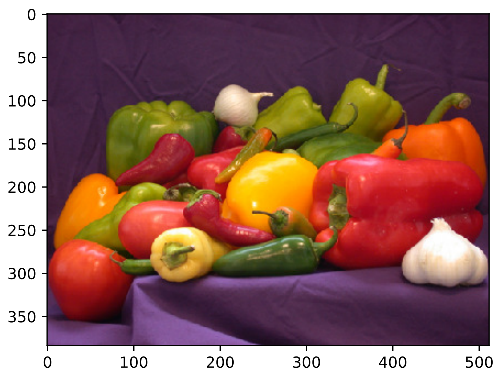


Figure 2.2b: peppers.png

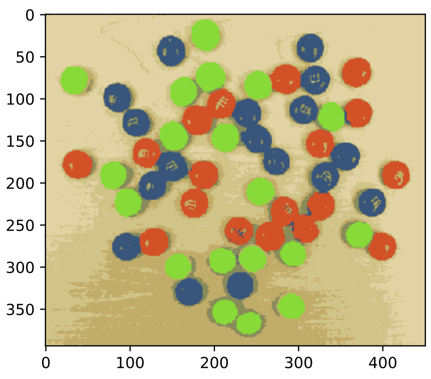


Figure 2.2c: 3-D vectors with k = 7

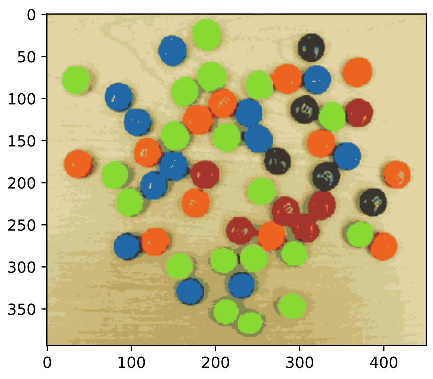


Figure 2.2d: 3-D vectors with k = 10

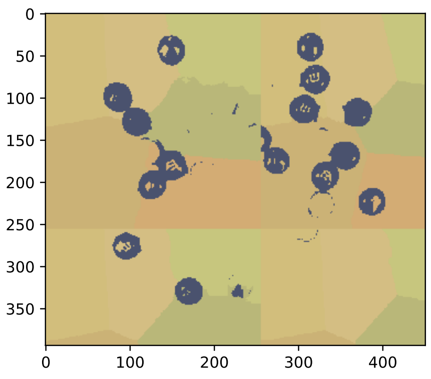


Figure 2.2e: 5-D vectors with k = 7

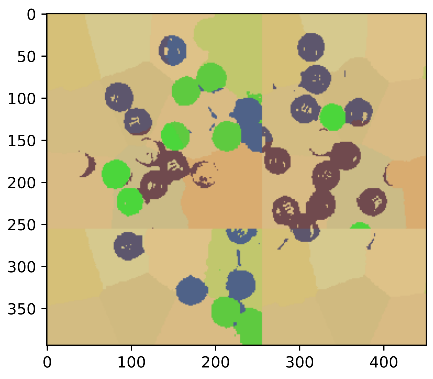


Figure 2.2f: 5-D vectors with k = 10

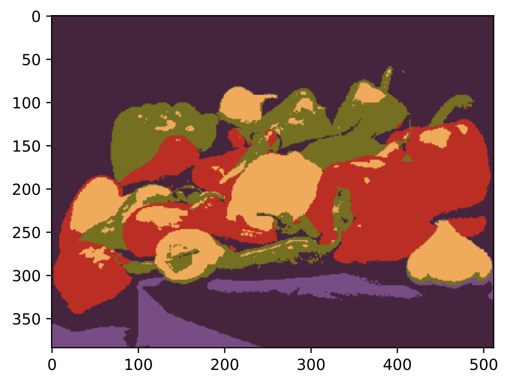


Figure 2.2g: 3-D vectors with k = 5

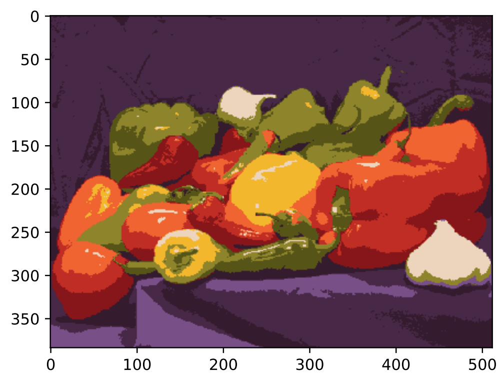


Figure 2.2h: 3-D vectors with k = 10

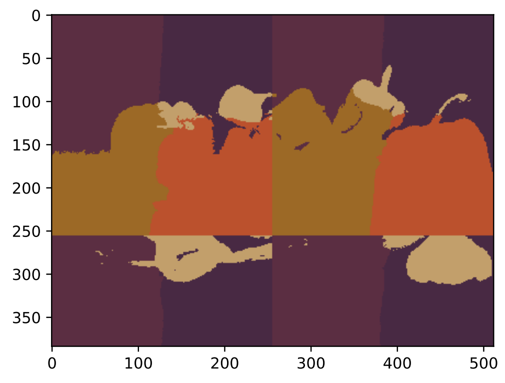


Figure 2.2i: 5-D vectors with k = 5

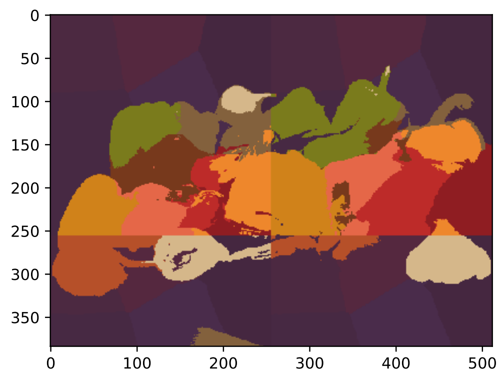


Figure 2.2j: 5-D vectors with k = 15

First, consider the results of 3-D vectors. Since there are many colours in mandm.png which are close to each other, and some of the colours are similar (e.g. red and orange), the k-means algorithm was not able to differentiate all the M&M's when k is 7, which is approximately how many different colours there are (red, orange, yellow, green, blue, black and light brown). If we increase k to 10, it will successfully identify all the M&M's colours. The case is similar with peppers.png. It is a similar case for peppers.png. We can see that the yellow pepper and the garlic are put into the same segment when k = 5. When k = 10, everything looks clearly separated.

When 5 dimensions are used for the feature vectors, the k-means algorithm will take the pixel coordinates into account. That means it will try to put pixels that are near each other in a cluster as the average is calculated with the coordinates as well. In our case, it is not ideal, since the colours can be scattered all over the image. k needs to be a much higher number to separate everything, otherwise the clusters will be mixed with different colours. The pixel coordinates can only be included when the colours are more distinct, for example an image with a clear blue sky, green fields and few objects with the same colour in different locations.

3. As seen in part 1, the K-means algorithm is sensitive to initialisation. Thus, K-means++ is introduced to improve the performance of K-means by generating better initial cluster centres. Instead of choosing all the centres by random data points, it only chooses the first centre by random. Then the centres are chosen with a probability proportional to the squared distance to the nearest centre. That makes the centres highly unlikely to be very close to each other. As a result, the clusters will be far away from each other and the k-means algorithm will be generally less likely to converge to a suboptimal local minimum. Although the initialisation will take more time, the authors show that the k-means++ algorithm is O(log k) competitive to the optimal k-means solution. The number of iterations it takes to converge before and after using k-means++ were recorded. The results are shown in the tables below.

Table 2.1: Number of iterations for k-means to converge on mandm.png

mandm.png	3-D vectors		5-D vectors	
k	standard k-means	k-means++	standard k-means	k-means++
7	18	7	16	13
10	30	24	16	8
15	39	19	28	7
20	45	42	18	13

Table 2.2: Number of iterations for k-means to converge on peppers.png

peppers.png	3-D vectors		5-D vectors	
k	standard k-means	k-means++	standard k-means	k-means++
5	10	12	10	7
7	15	10	12	8
10	18	9	17	15
15	16	24	11	11

As we can see, k-means++ increases the convergence speed for almost all cases.

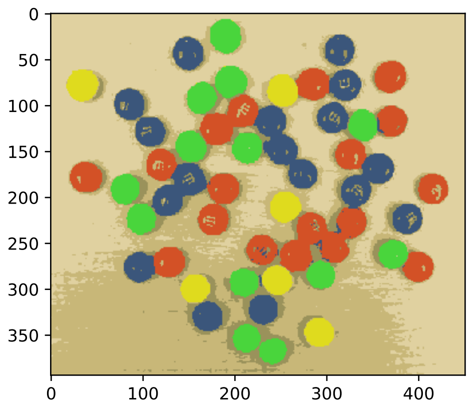


Figure 2.3a: 3-D vectors with k = 7

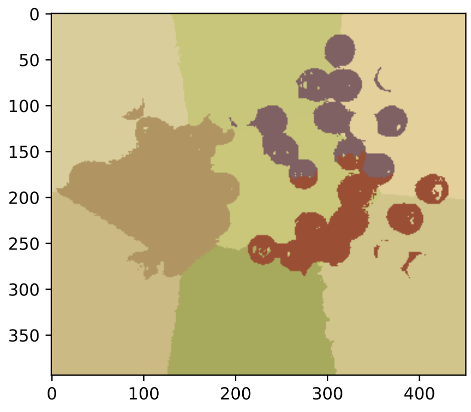


Figure 2.3b: 5-D vectors with k = 10

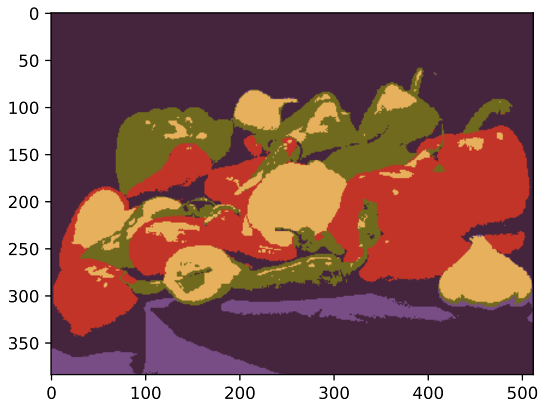


Figure 2.3c: 3-D vectors with k = 5

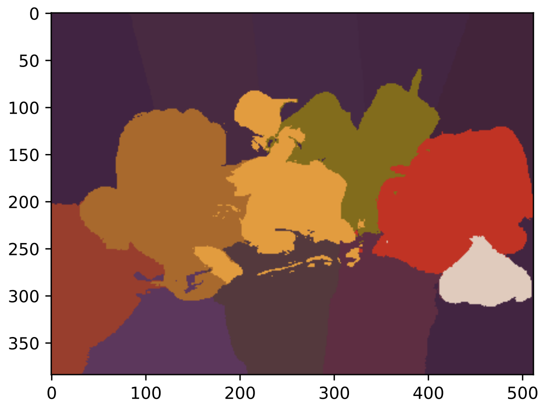


Figure 2.3d: 5-D vectors with k = 15

Similar to the standard k-means algorithm, 3-D vectors give much better results compared to 5-D vectors for our images. Notice that in Figure 2.3d the algorithm managed to roughly split the image into segments with the average colour of the area. This is an improvement from Figure 2.2j.

Task-3: Face Recognition using Eigenface

1. Alignment needs to be done on the images for Eigenface so that the eyes, nose and mouth are the approximately the same position. The PCA uses the average of the images and the difference from it. It recognises faces using the relative positions of the facial features. If the images are not aligned, the average will not have much meaning and the difference will not reflect actual difference in facial features.

2. Since we are dealing with a lot of dimensions for images, the eigendecomposition of high dimensional matrices is computationally expensive. On the other hand, the number of data points is much smaller since there are way less images (135) than dimensions ($195 \times 231 = 45045$), i.e. $N \ll D$. Thus, instead of
$$C = \frac{1}{N} A A^T \in \mathbb{R}^{D \times D},$$
we can have the covariance matrix as
$$C = \frac{1}{N} A^T A \in \mathbb{R}^{N \times N},$$
where AA^T and $A^T A$ have the same eigenvalues. If u_i is an eigenvector of $A^T A$, then $v_i = Au_i$ is an eigenvector of AA^T. Note that v_i then needs to be normalised.

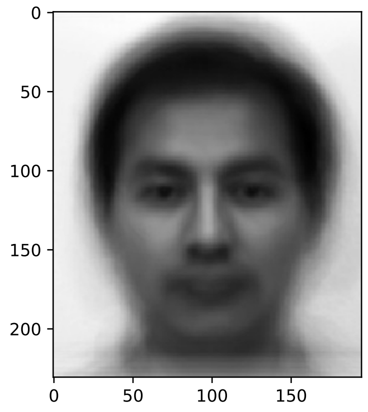
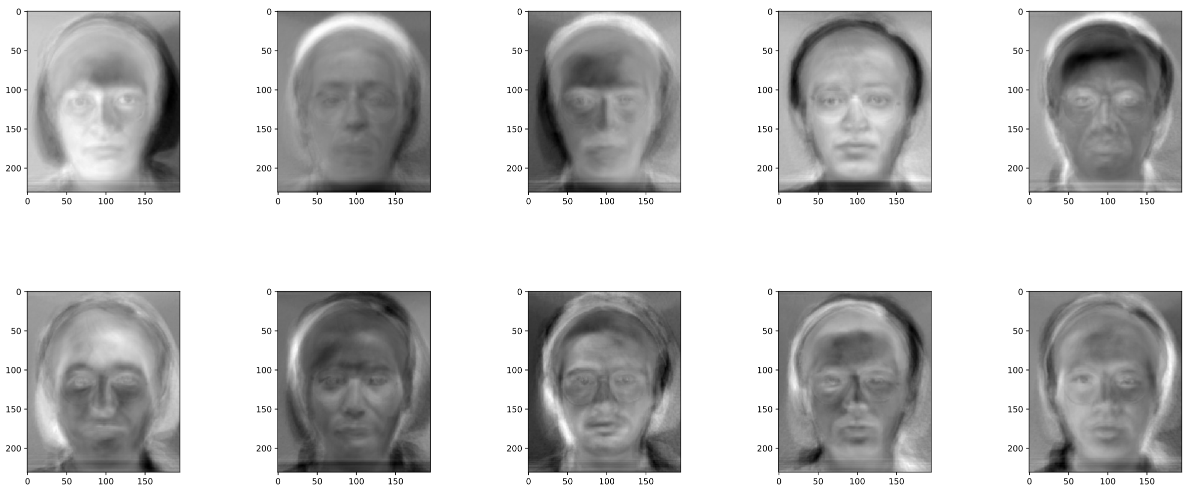
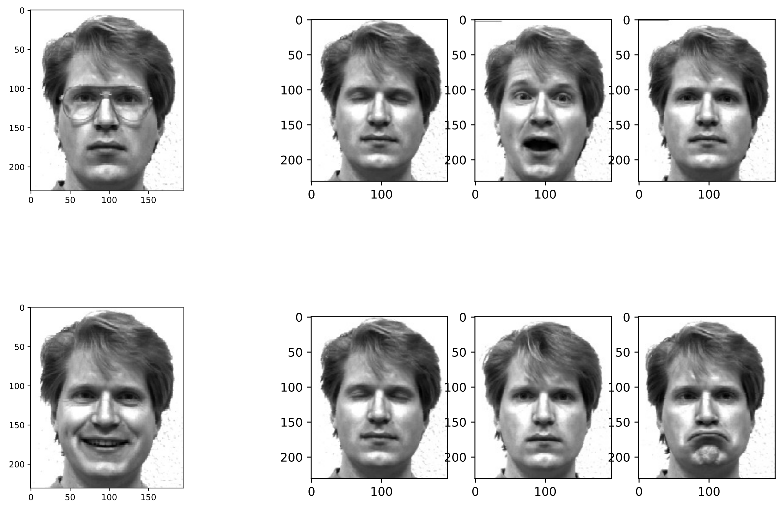


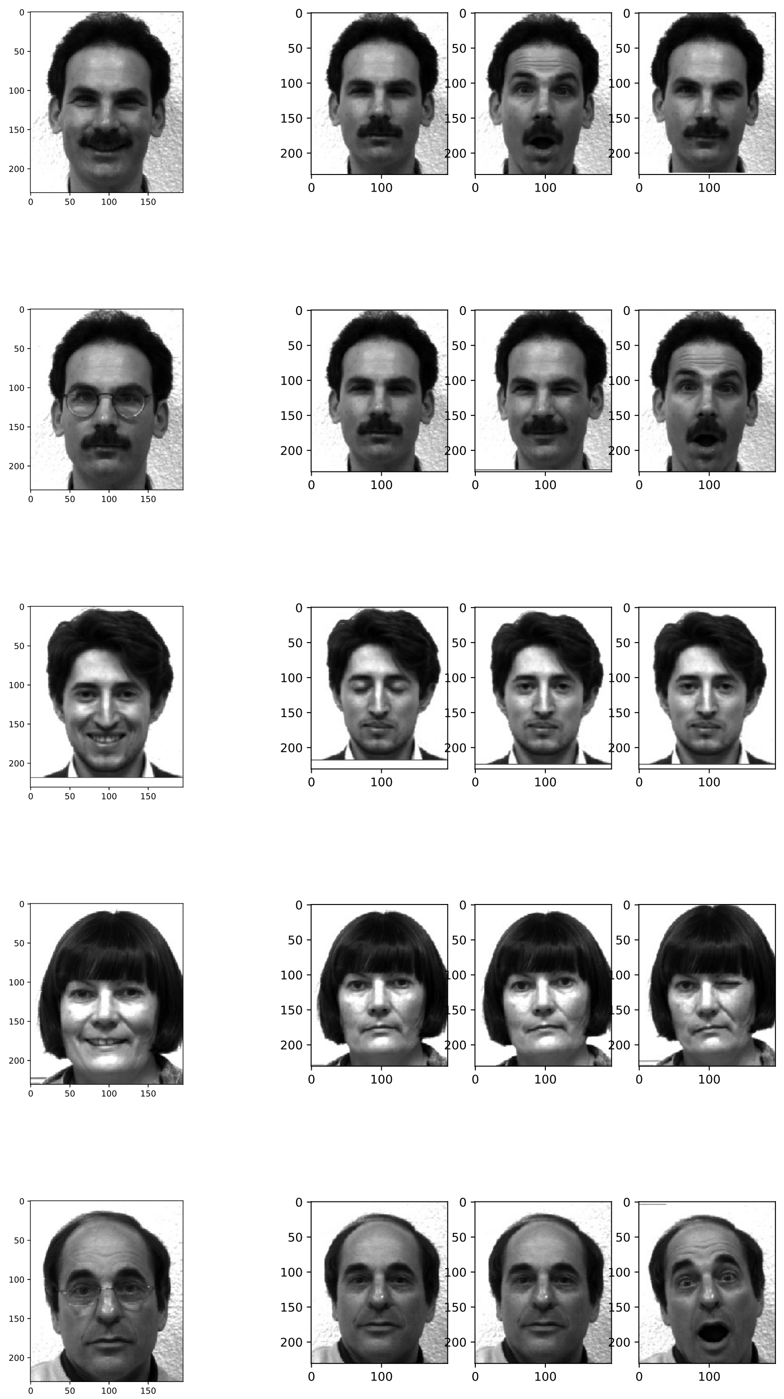
Figure 3.1: The average face

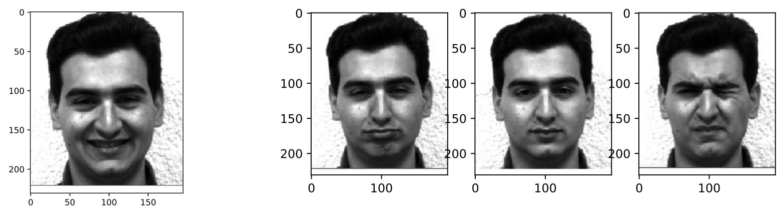
3. After implementing PCA with the above method, we get the top 10 eigenfaces as follows. Each of the eigenface represents a fairly different face. If 15 is used, it might represent all the 15 different subjects' faces.

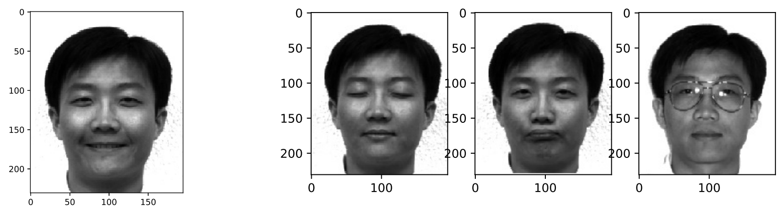


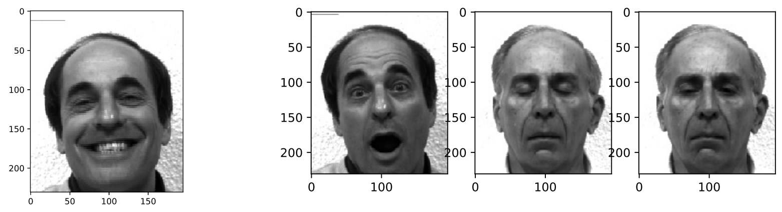
4. We project the images in the test set onto the basis spanned by the top 10 eigenfaces and find the 3 images with the least differences. The algorithm successfully identify the person for most images. The results are displayed below where the test set images are on the left and the 3 closest images are on the right.





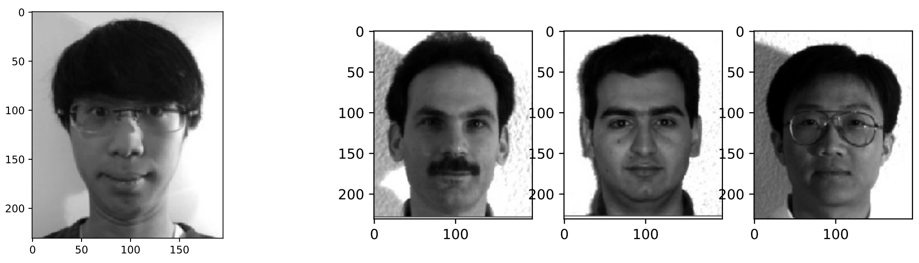






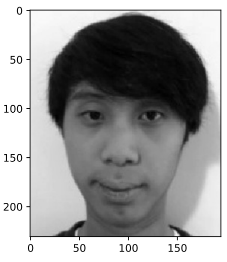
For the last image, the result is incorrect as it returned images of another person for the second and third nearest images. It was probably due to the fact that these 2 persons look alike and both have little hair. Overall, the performance is good as it returned the same person for all other test images, with an overall accuracy of $28/30 = 93.3\%$.

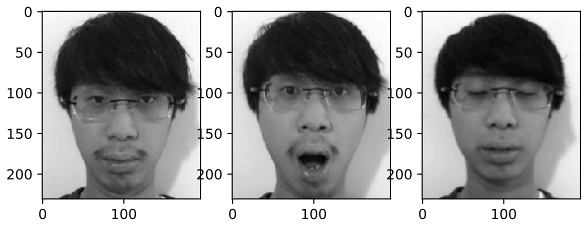
5. We run the recogniser on one of my own images and get the following results.



Since it has never seen this image before, it will be harder for it to return images of the same person. The top 3 nearest images are of 3 different people. Notice that two of them have a right light just like my photo.

6. Now we add 9 of my face images into the training set and repepat the experiment. The following 3 images are calculated as the closest images.





It managed to recognise my face and returned 3 of my other images, which shows good capability. To my surprise, it did not return other images of me with my eye glasses off, since I have a few images of me with and without glasses. In addition, the second closest image is me with my mouth open, which I thought would have a higher difference. One of the reasons might be my images are not perfectly aligned. Another factor might be the lighting of these images. The algorithm found them to be similar since the similar location of the shadows are taken into account.