Clab-2 Report

ENGN4528

Thao Pham

u7205329

02/05/2021

1 Task-1: Harris Corner Detector

Given that we were provided with codes to apply differential operators to an image, our task now involves two separate steps: 1) compute Harris cornerness, and 2) perform non-maximum suppression and thresholding.

1.1 Compute Harris cornerness

In order to find the cornerness of each pixel, we first find its second-order matrix as follows:

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

In this paper, the window function $w(x,y)$ used is an isotropic Gaussian distribution of size 5-by-5 and $\sigma = 1$. To find each $\sum_{x,y} w(x,y) I_x^2$, we first take the Hadamard product between each 5-by-5 window of I_x^2 and the window function and take the sum of all elements of the resulting matrix. Following this, we calculate the corner response measure where

$$R = det(M) - \alpha trace(M)^2$$

```python
def compute_Harris_cornerness(img, alpha, Ix2, Iy2, Ixy):
    h,w = img.shape

    Rmatrix = np.zeros(img.shape)
    window = fspecial((5,5), 1) # window function

    for i in range(2,w-2):
        for j in range(2,h-2):

            # Find the second-order matrix of each pixel
            Ix2_window = np.multiply(Ix2[j-2:j+3, i-2:i+3], window)
            Iy2_window = np.multiply(Iy2[j-2:j+3, i-2:i+3], window)
            Ixy_window = np.multiply(Ixy[j-2:j+3, i-2:i+3], window)

            sum_Ix2 = np.sum(Ix2_window)
            sum_Iy2 = np.sum(Iy2_window)
            sum_Ixy = np.sum(Ixy_window)

            M = np.array([[sum_Ix2, sum_Ixy], [sum_Ixy, sum_Iy2]])

            # Response Value
            r = np.linalg.det(M) - alpha*(np.trace(M)**2)

            # Save the response value to a new matrix
            Rmatrix[j,i] = r

    return Rmatrix
```

1.2 Non-maximum suppression and Thresholding

We now proceed to perform thresholding to filter out pixels that are potential corners which is done from line 5 to line 8 below. Following this, we add the pixel with the highest response value to the `final-corners` list. We then proceed to find corners that are 8-way local maxima in the following fashion:
For each corner c_i in `corners-sorted`, check whether or not the corner is within a radius of 8 from any existing corner in `final-corners`. If yes, we will dismiss the corner. If not, we will add c_i to the `final-corners` list and proceed to the next corner c_{i+1}.

```
def marking_corners(img, response_matrix, t, d):
    corners = {}
    h, w = img.shape

    # Thresholding
    for j in range(h):
        for i in range(w):
            if response_matrix[j,i] > t:
                corners[(j,i)] = response_matrix[j,i]

    # Sort the corners w.r.t to their response values in descending manner
    corners_sorted = sorted(corners.items(), key=lambda x: x[1], reverse=True)

    # Create the final_corners list in preparation for NMS
    final_corners = [corners_sorted[0]]

    # Non-maximum suppression
    for potential_corner in corners_sorted:
        within_radius = False
        for chosen_corner in final_corners:
            yp, xp = potential_corner[0]
            y_chosen, x_chosen = chosen_corner[0]

            # If a potential corner is within a radius of 8 of any existing chosen corners,
            # we will dismiss it
            if (np.abs(xp - x_chosen) <= d) and (np.abs(yp - y_chosen) <= d):
                within_radius = True
                break

        # If not in conflict with any chosen corners, we will add it to the final list of corners
        if not within_radius:
            final_corners.append(potential_corner)

    # Return a Nx2 matrix of the pixel coordinates
    yc, xc = [], []
    for coord, Rval in final_corners:
        y,x = coord
        yc.append(y)
        xc.append(x)

    mat = np.column_stack((yc, xc))
    return mat
```

Finally, the full process for each image can be done as follows. Note that the chosen empirical constant is 0.04 and the threshold for determining a corner is 1% of the maximum response value.

```
# Calculate the image derivatives (code given)
def calculate_derivs(img):
    dx = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])
    dy = dx.transpose()

    Ix, Iy = conv2(img, dx), conv2(img, dy)
    sigma = 2

    # This line underneath is to create a Gaussian 2D distribution of size 13 x 13
    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)
    return Ix2, Iy2, Ixy

# Getting the image derivatives
Ix2, Iy2, Ixy = calculate_derivs(img)

# Calculate the corner response matrix
alpha = 0.04
response_matrix = compute_Harris_cornerness(img, alpha, Ix2, Iy2, Ixy)
threshold = np.max(response_matrix) * 0.01

# NMS and thresholding and getting the final corners
M = marking_corners(img, response_matrix, threshold, 8)
ys, xs = M[:, 0], M[:, 1]
```

1.3 Results and Discussion

Figure 1 (left column) shows the result (in red dots) of our detector system tested on four test images while the results (in yellow dots) yielded from Python's built-in function `cv2.cornerHarris()` are shown on the right column. We see that the algorithm has done a fairly good job in identifying major corners such as most outer corners of the window frames in `Harris-1.jpg` or the main skeleton of the house in `Harris-3.jpg`. The algorithm still struggled, however, when distinguishing between patterns and actual corners. Therefore, we see many red dots in highly textured areas such as grass fields or texts. The detector also missed many true corners across all test images.

Comparing to the result achieved by `cv2.cornerHarris()`, we found that both methods obtained very similar corners. Both also missed many of the smaller panes within the window frames. Furthermore, our manual detector found more false positives, especially in the textured regions as mentioned above. A number of factors that may affect the performance of the detector include the level of noise in the reference image, the implementation of finding image derivatives, the window function and empirical constant used in finding the response values, and lastly, the implementation of non-maximum suppression.

For `Harris-5.jpg`, we should not get any corners since this is a case of a vertical edge. That is, if we hover a window and move up and down across the image, we won't see any intensity change. When investigating the matrix of response values, we should see a large majority of response values are zero (corresponds to the flat area) and certain areas of negative values and large magnitude (corresponds to the vertical edge). This is true in our case as displayed in the heat map of the response matrix in Figure 2. (b).

In our case, however, four corners were identified. After a closer inspection, we found that these misclassifications were introduced after the image derivatives were found. We believe that this is due to the padding method during cross-correlation between the differential operators and the Gaussian smoothing filter. A small white horizontal strip is visible at the bottom of the image's second-derivative (Figure 2 c).

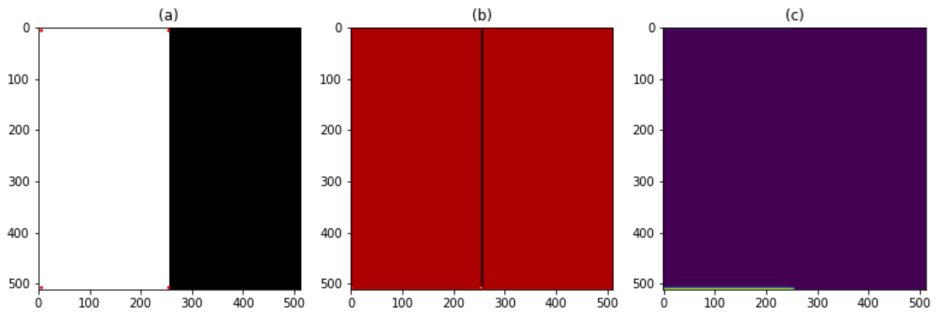

Figure 2: a) `Harris-5.jpg` with detected corners b) Response values matrix c) Image second-derivative in the y-direction

For `Harris-6.jpg`, the algorithm detected numerous "corners" across the image in combination with the true corner. This large number of misclassifications is due to the image being badly corrupted with noise. Thus, movements across any direction result in high variations in intensities. A solution to this problem is to pre-process the image by applying another layer of Gaussian filter. Figure 3 (b) shows the image smoothed out by a Gaussian kernel of size 7-by-7 and $\sigma = 3$. Note that the kernel size and sigma chosen need to be specific to the reference image.

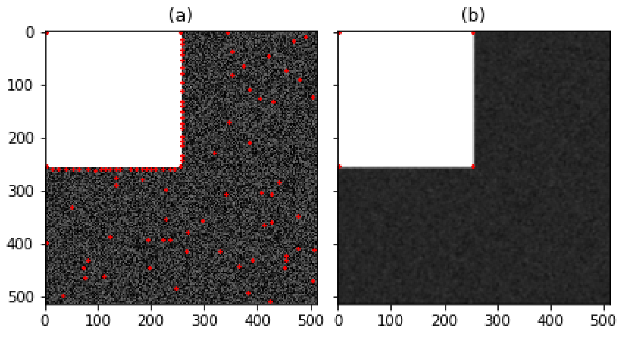

Figure 3: a) `Harris-6.jpg` with detected corners b) Pre-processed `Harris-6.jpg` with detected corners

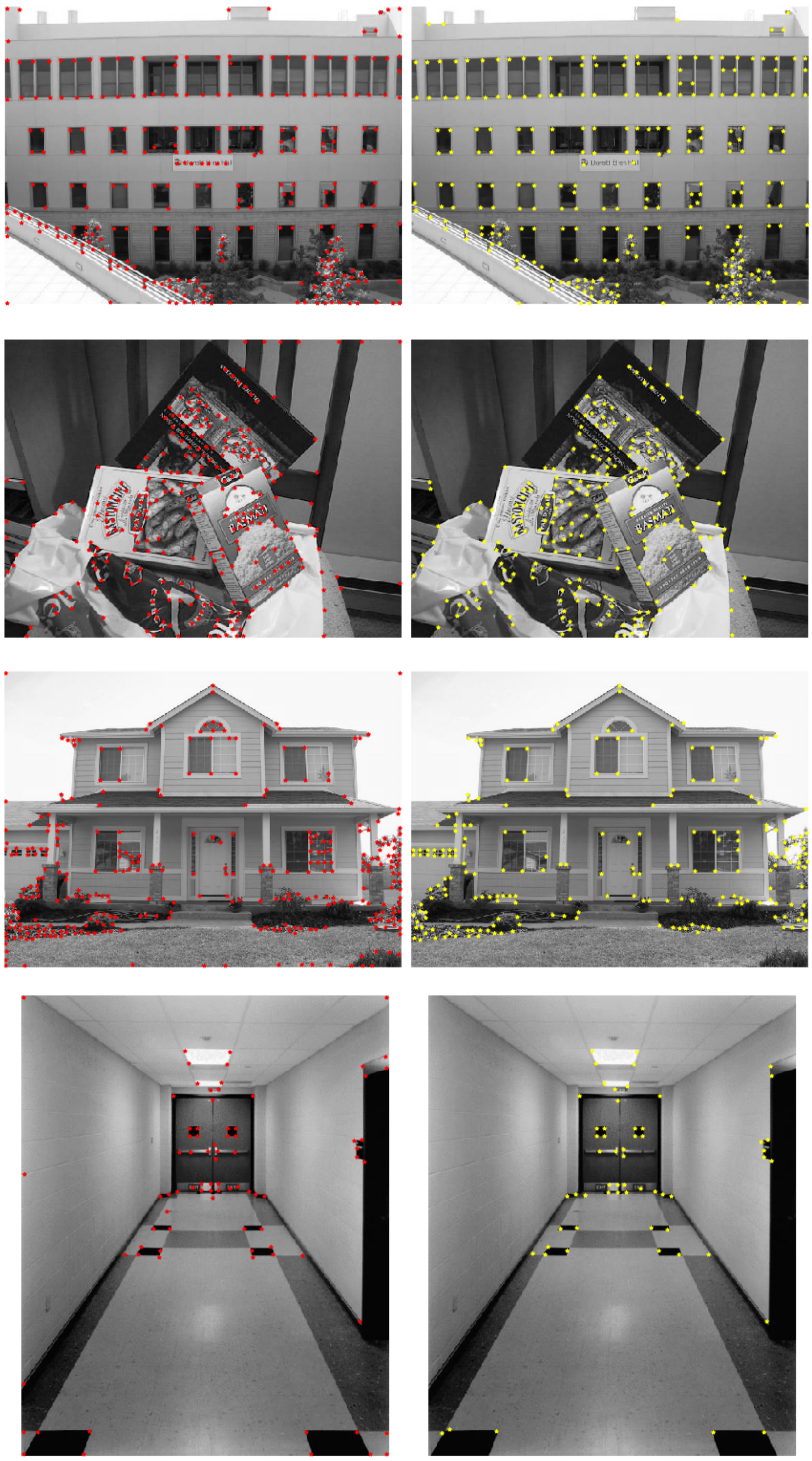
Figure 1: Harris Corner Detector applied on four test images manually (left column) and with `cv2.cornerHarris()` (right column)

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

We will discuss the result yielded from applying the k-mean algorithm for colour image segmentation for the M&M and Pepper image, respectively.

2.1 M&M image

2.1.1 With different number of clusters

In this section, we will perform colour image segmentation on a 5-dimensional feature space (L, a*, b* for colour and x, y for spatial representation) (denote 5DKM) and investigate the effects for a different number of clusters. Note that the similarity between each data point and the cluster centres is determined by the Euclidean distance between them.

With K = 7, it can be seen that the background was partitioned into 7 regions with no individual candy formed in the foreground. As K increases to 20, a number of candies had started taking shapes and their colours correctly identified (e.g. blue and red). At K = 40, more M&M candies were separated from the background. However, more iterations are required for the segmentation to be meaningful.

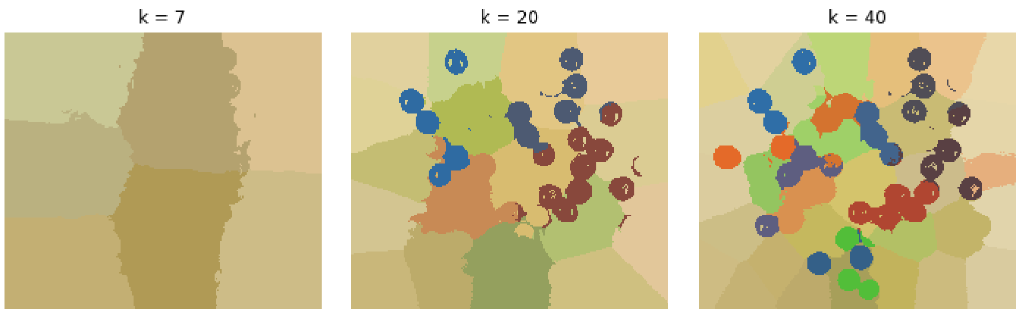

Figure 4: 5DKM on `MandM.png` with different number of clusters

2.1.2 With and without pixel coordinates

We now evaluate segmentation results done on different feature spaces - with (5DKM) and without pixel coordinates (3DKM). It can be seen that when we only consider the colour similarity, the segmentation result became meaningful at K = 7, where the background and foreground were separated clearly and the majority of the candies colours have been recognised while this was not the case in 5DKM.

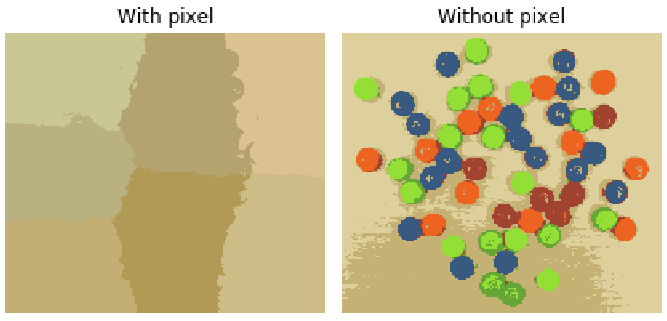

Figure 5: 5DKM and 3DKM on `MandM.png` with K = 7

2.2 Pepper Image

2.2.1 With different number of clusters

It appears that the 5DKM method worked considerably better on the `Peppers.png`. Similar to above, the algorithm partitioned the image into several major regions at k = 7. At k = 20, we can informatively tell the foreground from the background. At k = 40, differ from above, the segmented image is now comprehensible as most of the fruits have their shapes clearly outlined and colours accurately identified. Shades are also slowly introduced.

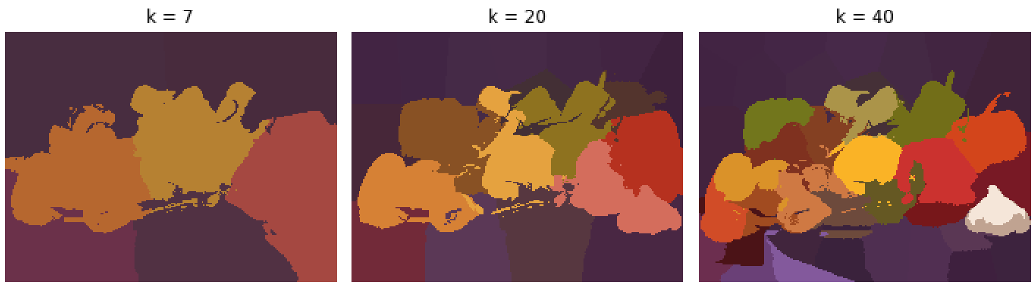

Figure 6: 5DKM on `Peppers.png` with different number of clusters

2.2.2 With and without pixel coordinates

Consistent with the result shown in `MandM.png`, the segmentation process based on only the colour features reached a more meaningful result at a smaller K for `Peppers.png`. At K = 7, the result post-partitioned closely resembles the original image. Shades and highlights are also introduced showing signs of over-segmentation.

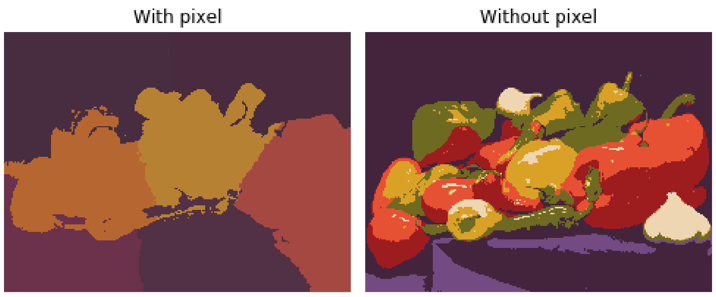

Figure 7: 5DKM and 3DKM on `Peppers.png` with K = 7

2.3 K-means++

In the above section, the initialisation method used is rather naive since we set random data points from the dataset to be the initial cluster representatives. The method is not optimal since the chosen centroids can be next to each other (in terms of coordinates) or have very similar colours. These poor starting centers can increase computation time significantly. Hence, we will implement a different initialisation method called **K-means++** that deliberately finds data points that are far away from each other, thus in general, leads the process to converge faster. The overall workflow was illustrated by Arthur, D. and Vassilvitskii, S. (2016) as follows:

1. Select a random data point as the first cluster representative.
2. For each data point x_i, compute the distance D(x) between x_i and the closest cluster center among the chosen ones.
3. Compute the weighted probability distribution where

$$p(x_i) = \frac{D(x_i)^2}{\sum D(x_i)^2}$$

4. Choose a new centroid based on this new probability distribution.
5. Repeat the above 3 steps until all k cluster representatives have been selected.

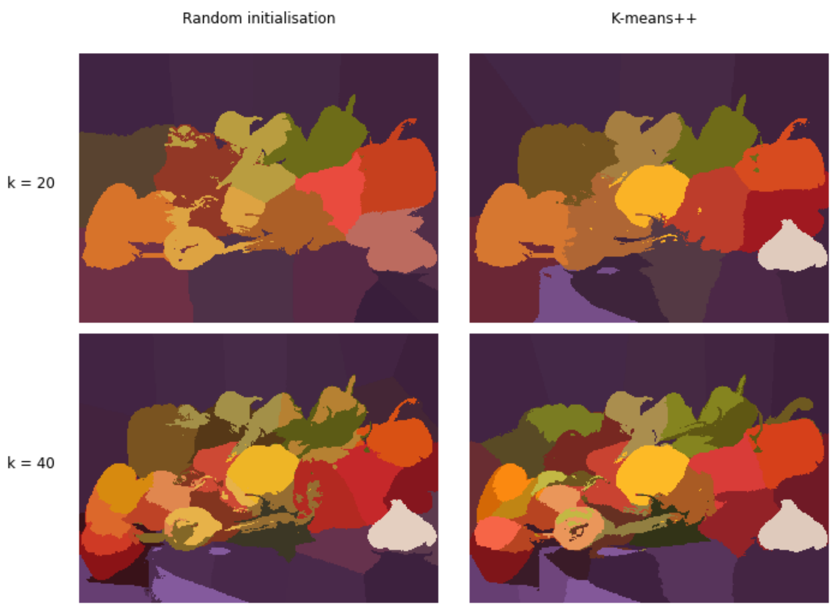

Figure 8: Segmented results using different initialisation methods and different number of clusters

The segmented results obtained from the two mentioned methods are shown in Figure 8. With a fixed number of clusters, it can be seen that the results are not significantly different. However, as mentioned in the discussion above, the main advantage of K-means++ lays in the reduced computation time. Indeed, for K = 20, the clustering algorithm converged after 184.36 seconds when K-means++ was used, earlier than that of the random method (198.86 seconds). The difference is more significant when we increase K to 40 (394.32 seconds to 526.08 seconds). Note that the elapsed time reported did not include the initialisation time.

3 Task-3: Face Recognition using Eigenface

Alignment: Given that PCA is built upon the idea of capturing maximal variance and projecting original data onto that lower-dimensional subspace, it is particularly sensitive to alignment and scaling. Variation in the position of the eyes or changes in the background can easily alter the coefficients as we perform reduction of dimensionality.

3.1 Train an Eigenface recognition system

For data preparation, we first turn each training image of size 195×231 into a 45045-dimensional vector. Following this, we yield a data matrix of size 45045×135 where each column represents a training image. By taking the average across all columns, we obtain the mean face image as can be seen in Figure 9.

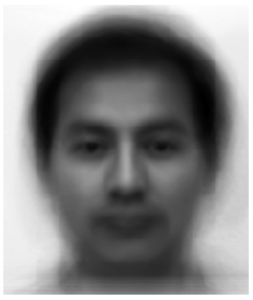

Figure 9: Mean face image.

Since each image vector is very large, the covariance matrix AA^T is of size 45045×45045 and thus performing eigen-decomposition directly on this matrix will be extremely time- and memory-consuming. Instead, we will decompose the 135×135 matrix A^TA. Let $\mathbf{v_i}$ be an eigenvector of A^TA, by definition

$$A^TA\mathbf{v_i} = \lambda\mathbf{v_i}$$

left-multiply both sides by A

$$AA^T(A\mathbf{v_i}) = \lambda(A\mathbf{v_i})$$

3.2 Top 15 Eigenfaces

For every eigenvector $\mathbf{v_i}$, we obtain an eigenvector $\mathbf{u_i} = A\mathbf{v_i}$ for the covariance matrix AA^T. Each of the newly-obtained vectors $\mathbf{u_i}$ represents certain types of variation among our training data. By ordering the eigenvalues in a descending manner, the eigenvectors corresponding to the top k eigenvalues capture the highest variances in our training set. Figure 10 shows these top 15 eigenvectors normalised.

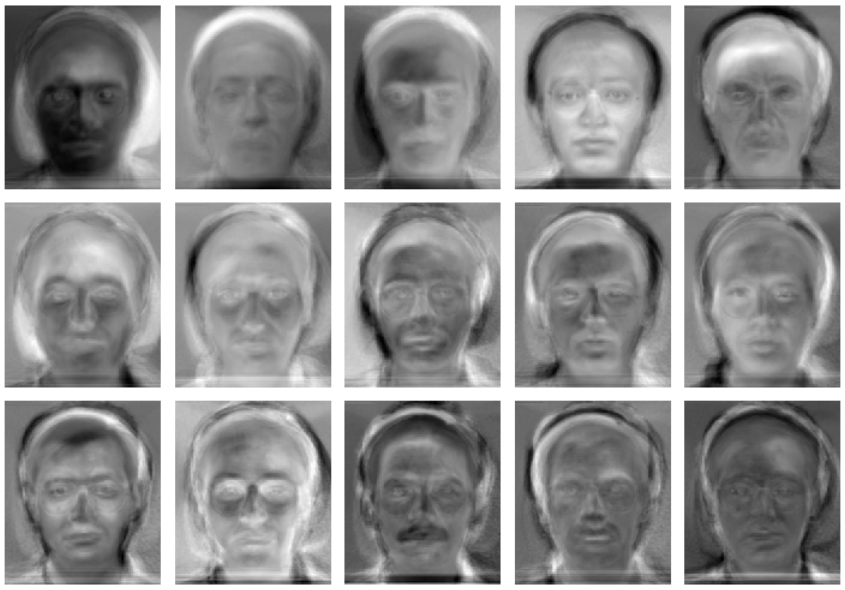

Figure 10: Top 15 eigenfaces

3.3 Nearest Neighbour Search

To proceed with face recognition, we project all training images into this new lower-dimensional subspace spanned by the above 15 eigenfaces using the following equation:

$$\hat{x}_{train} = \mu + w_1\mathbf{u_1} + w_2\mathbf{u_2} + ... + w_{15}\mathbf{u_{15}}$$

where μ is the mean image vector and each coefficient w_i is determined by the dot product

$$w_i = \mathbf{u_i}^T(\mathbf{x}_{train} - \mathbf{u_i})$$

For each test image that we wish to match, we also perform dimensionality reduction and yield $\hat{x}_{test}$. The distance between $\hat{x}_{test}$ and each of $\hat{x}_{train}$ will then be computed. Finally, the three training images that give the minimum distances will be returned as the most similar images. The results are shown in Figure 11.

It is evident that our face detector has performed as desired as it correctly identified 9 out of our 10 test images. For the test person #3, however, 2 of 3 closest images are of a different person. While this is a misclassification, we do see a high facial similarity between the two participants (particularly the hair line, eyebrows, and overall bone structure). We can increase the recognition accuracy by adding more participants of different gender and background to the training set.

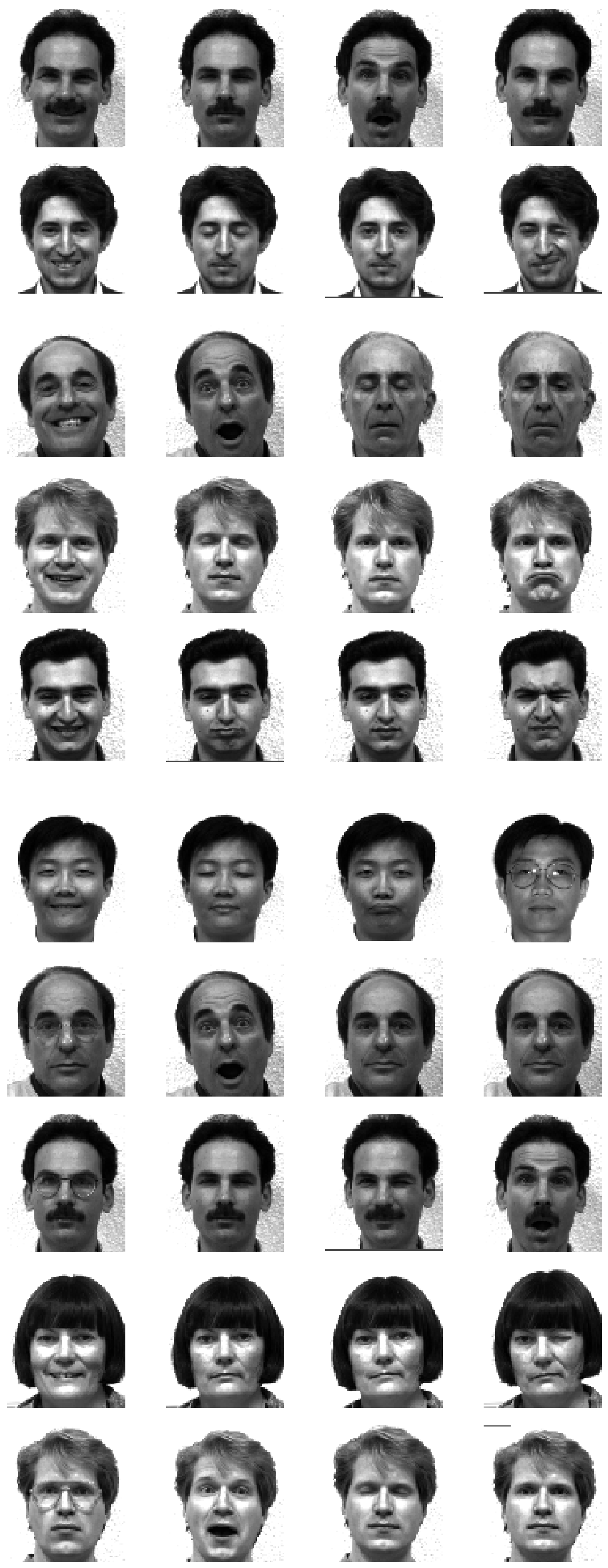
Figure 11: Top three similar face images for each reference image (leftmost) in the test dataset

3.4 Face Recognition with Unseen Test Image

The face recognition system was then tested with a new test image. We then repeat the entire PCA process with 9 additional images in the training data. It can be seen in Figure 12 that the algorithm has correctly identified the new person added to the dataset.

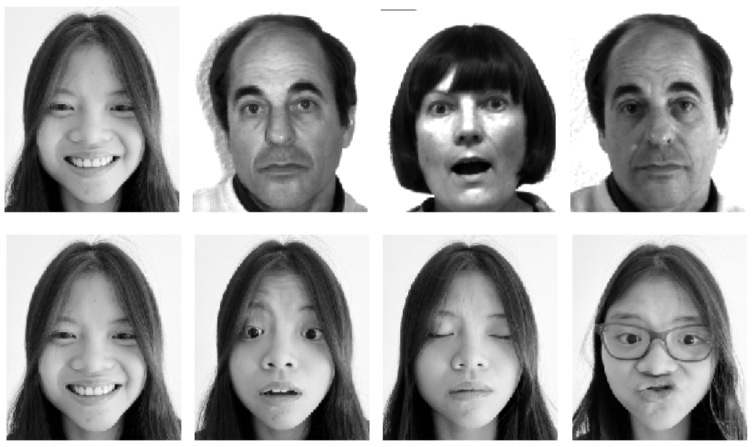
Figure 12: Face Recognition on new test image before (upper row) and after (lower row) expanding the training set

4 References

Arthur, D. & Vassilvitskii, S. (2016). k-means++: The Advantages of Careful Seeding. Stanford University. California, US. Retrieved from: http://ilpubs.stanford.edu:8090/778/1/2006-13.pdf