# CLab-2 Report

**ENGN6528**

Zhiyuan Huang

u6656110

**27/9/2020**

# 1 Hrris corner detector

1.

```
#########################################################################
# Task: Compute the Harris Cornerness
#########################################################################
row, col = bw.shape # get the shape of the image
res = np.zeros((row, col)) # record which points are corners
R = np.zeros((row, col)) # The cornerness value of every pixel
rmax = 0
for i in range(row):
    for j in range(col):
        # calculate the M matrix for every pixel in the image
        M = np.array([[Ix2[i, j], Ixy[i, j]], [Ixy[i, j], Iy2[i, j]]], dtype=np.float64)
        # find the R value based on the determinant and trace of M
        R[i, j] = np.linalg.det(M) - k * np.power(np.trace(M), 2)
        # Find the maximum
        if R[i, j] > rmax:
            rmax = R[i, j]

#########################################################################
# Task: Perform non-maximum suppression and
#       thresholding, return the N corner points
#       as an Nx2 matrix of x and y coordinates
#########################################################################
for i in range(1, row - 1):
    for j in range(1, col - 1):
        # Here we use thresh * rmax as threshold, if R is greater , it is a corner
        # We choose the pixel as corner only if it is biggest in its neighbour.
        # We can ignore duplicate detections. This size of window can be adjusted.
        if R[i, j] > thresh * rmax and R[i, j] == np.amax(R[i-1 : i+2, j-1 : j+2]):
            res[i, j] = 1

cx, cy = np.where(res == 1)
plt.plot(cy, cx, 'r+')
bw = plt.imread('Harris_2.pgm')
plt.imshow(bw)
plt.show()
```

Figure 1: The Harris corner detection implementation with appropriate comments.

```
# Detect corners using built-in function cv2.cornerHarris()
bw = plt.imread('Harris_1.jpg')
img = cv2.imread('Harris_1.jpg')
bw = cv2.cvtColor(bw, cv2.COLOR_RGB2GRAY)
bw = np.float32(bw)
dst = cv2.cornerHarris(bw, 2, 3, 0.04)
#result is dilated for marking the corners, not important
dst = cv2.dilate(dst,None)
# Threshold for an optimal value, it may vary depending on the image.
img[dst>0.01*dst.max()]=[255,0,0]
plt.imshow(img, 'gray')
plt.show()
```

Figure 2: Harris corner with in-built function

2.

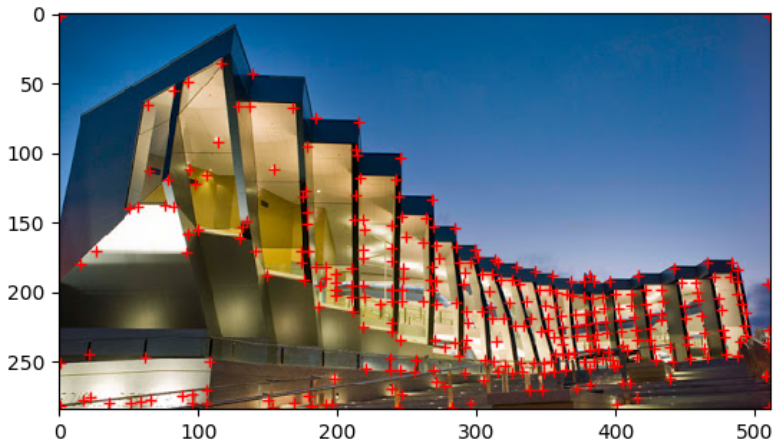



Figure 3: First image corner detection using my detector

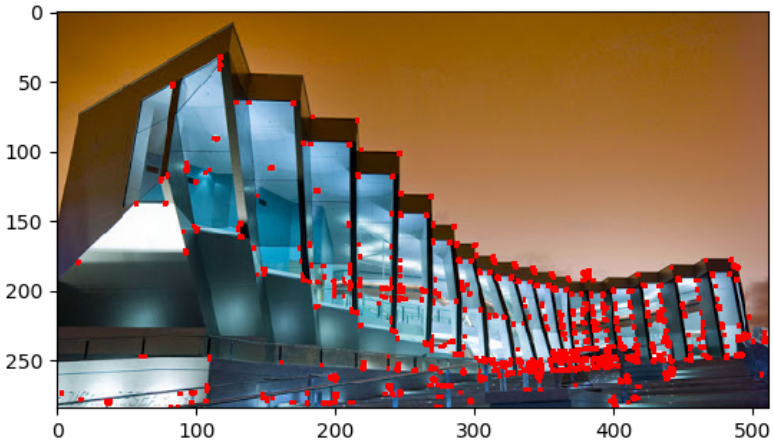



Figure 4: First image corner detection using build-in detector

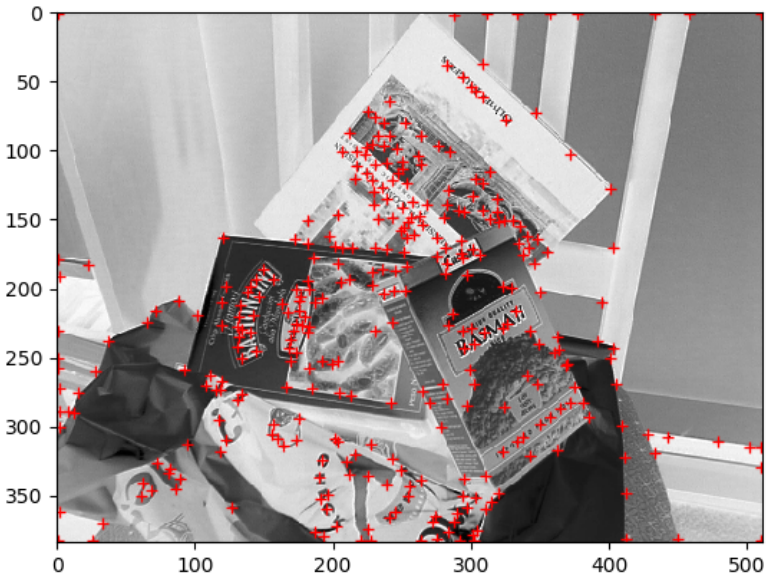



Figure 5: Second image corner detection using my detector

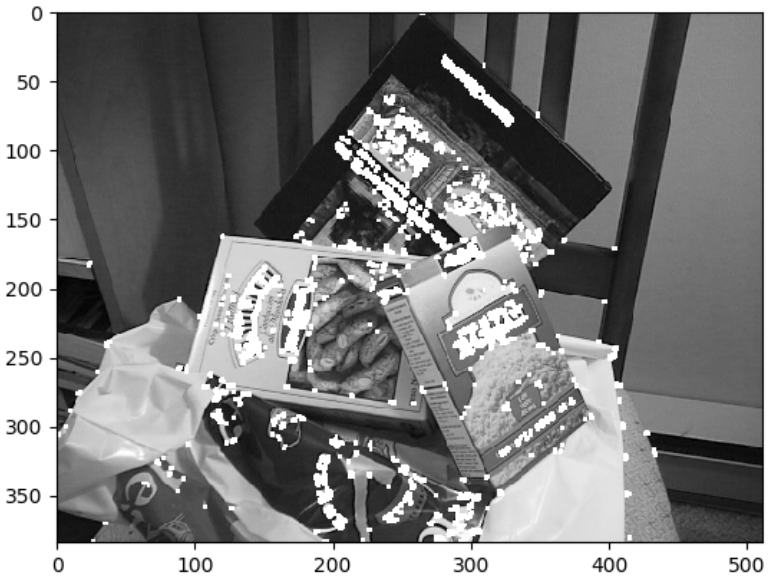



Figure 6: Second image corner detection using build-in detector

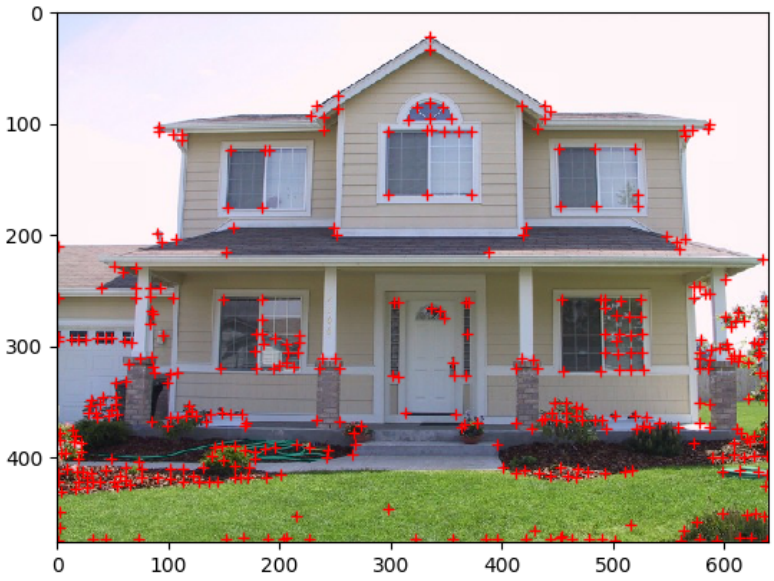



Figure 7: Third image corner detection using my detector

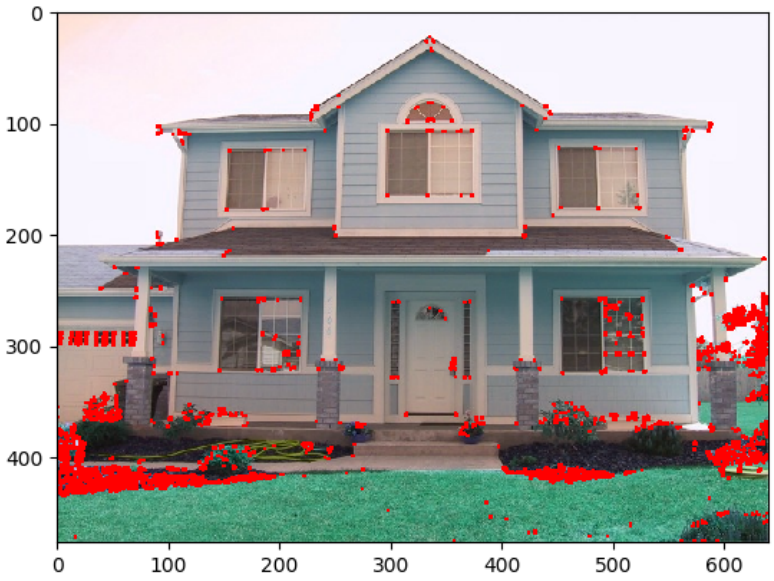



Figure 8: Third image corner detection using build-in detector

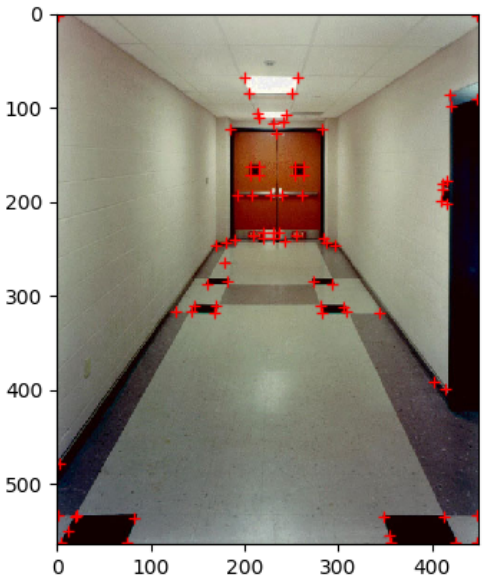



Figure 9: Fourth image corner detection using my detector

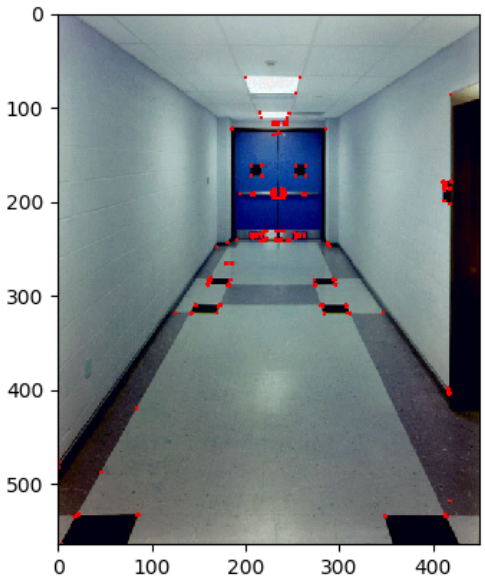



Figure 10: Fourth image corner detection using build-in detector

5.

Comparison: From the images above we can see that my implementation of Harris corner detector could effectively detect corners in the images. The results are pretty accurate. Almost the same as the build-in Harris corner detector.

Discussion: Based on the implementation, we can easily conclude that threshold could affect the performance, If threshold is small, some non-corner area could be treated as corners. While a large threshold could ignore some corners. Hence, it is important to choose proper threshold.

Another fact that could affect the performance is the size of non-maximum suppression window. Again, the size should be chosen carefully to make the

detection clear and accurate.

# 2 Task 2 K-means clustering and color image segmentation

1. The K-means clustering implementation is shown in following figures.

```
def my_kmeans(img, k):
    row, col, clr = img.shape
    # initialize centroids randomly
    centroids = {
        i: img[np.random.randint(0, row), np.random.randint(0, col)]
        for i in range(k)
    }
    # Initialize centroids using kmeans++
    centroids = kmeansplus(img, k)
    changed = True
    res = None
    # Iterate until centroids don't change
    while changed:
        changed = False
        print(centroids)
        # assign all the points to its nearest centroid
        res, mean = assign(img, centroids)
        # Check whether centroids changed
        for i in range(k):
            cent = mean[i] / (len(res[i]) + 1)
            # Calculate the norm of previous centroids and new centroids
            # The results less than 1 is acceptable
            if np.linalg.norm(np.asarray(centroids[i]) - np.asarray(cent)) > 1:
                centroids[i] = cent
                changed = True
    return centroids, res
```

```python
def assign(img, centroids):
    row, col, clr = img.shape
    cluster = len(centroids)
    res = {
        i: []
        for i in range(cluster)
    }
    mean = {
        i: np.zeros((clr, ))
        for i in range(cluster)
    }
    # Iterate through all pixels in the image
    # assign it to its nearest centroid cluster
    for i in range(row):
        for j in range(col):
            minimum = 0
            idx = 0
            for k in centroids.keys():
                dist = np.linalg.norm(img[i, j] - centroids[k])
                if minimum == 0 or dist < minimum:
                    minimum = dist
                    idx = k
            res[idx].append((i, j))
            mean[idx] += img[i, j]
    return res, mean
```

Figure 11: K-means clustering implementation

2

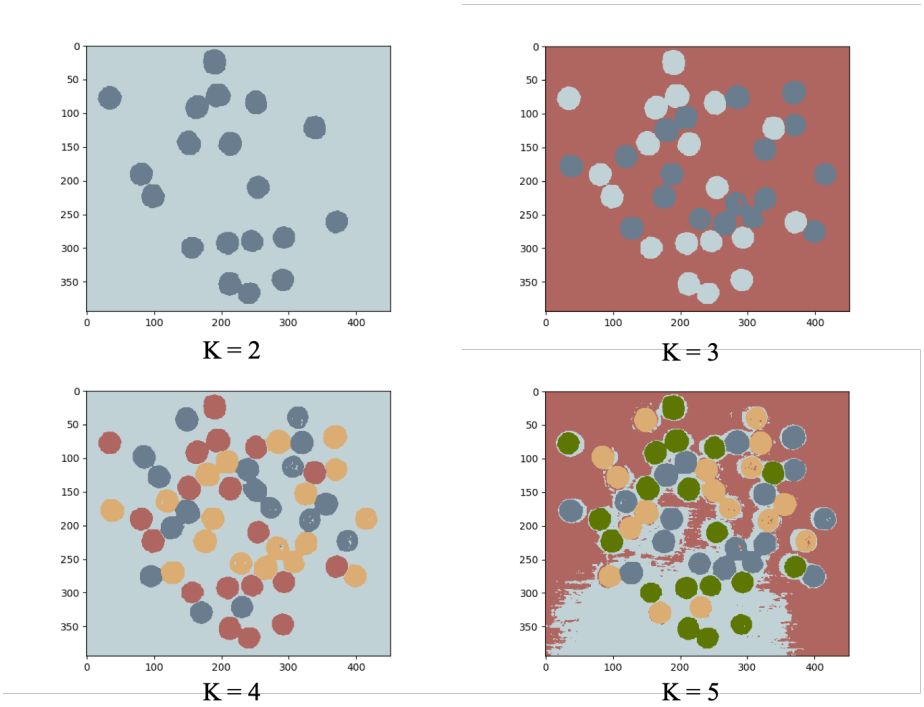



Figure 12: Apply k-means to the image with different k values without coordinates

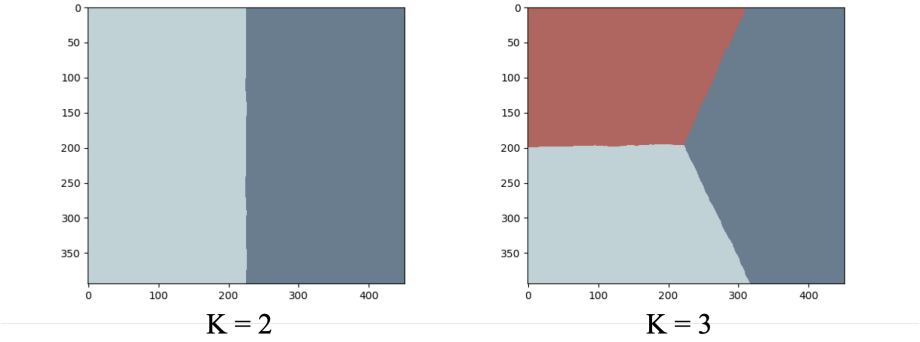



Figure 13: Apply k-means to the image with different k values with coordinates

If we don't use coordinates, we can get pretty good segmentation. However, if we add coordinates, the image was just divided evenly according to coordinates. The reason I think is because the value of the coordinates have more impact than $L^*$, $a^*$ and $b^*$. $a^*$ and $b^*$ are often less than 20 while coordinates can be up to 400 since the image size is around 400 * 400. That is the reason why we get the evenly divided results. Moreover, I think using coordinates doesn't make too much sense in this case. Since the pixel location has nothing to do with its clustering. For instance, the background, which is the table, is shown every where in the image. However, no matter where it is, it should be in same cluster. The coordinates shouldn't have impact on its clustering.

The reason why we use $L^*$, $a^*$ and $b^*$ instead of RGB is because brightness plays an important role in pixel clustering. It is better if we don't ignore its variation in brightness.

3.

```python
def kmeansplus(img, k):
    row, col, clr = img.shape
    # Initialize the first centroid randomly
    first = img[np.random.randint(0, row), np.random.randint(0, col)]
    centroids = dict()
    centroids[0] = first
    dist = np.zeros((row, col))
    # Find the remaining k - 1 centroids
    for _ in range(k - 1):
        total = 0.0
        for i in range(row):
            for j in range(col):
                # For a pixel in the image, find its nearest centroid
                # Then record the distance and sum them
                dist[i, j] = get_closest_dist(img[i, j], centroids)
                total += dist[i, j]
        # We multiply the total distance by a random number between 0 ~ 1
        # Then iterate through data points to find which pixel locates there
        # This pixel is chosen as the next centroid
        total *= np.random.random()
        centers = len(centroids)
        for m in range(row):
            for n in range(col):
                total -= dist[m, n]
                if total > 0:
                    continue
                centroids[_ + 1] = img[m, n]
                break
            if centers == len(centroids) + 1:
                break
    return centroids
```

Figure 14: Initialize centroids using k-means++ algorithm

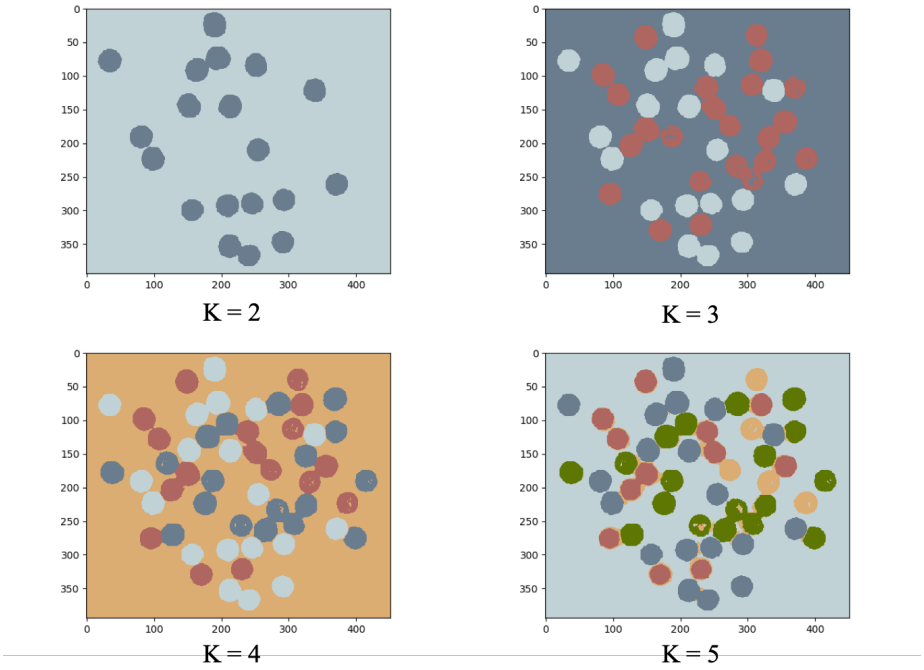



Figure 15: K-means++ color segmentation with different k values

From the figures above we can see that the results compared to using kmeans are pretty similar. I only run several times. If I run thousands of times on same image, k-means++ may be more stable than k-means.

The k-means++ algorithm could improve the convergence time. For small k, the convergence time doesn't vary much. However, when k increases, k-means++ gradually becomes faster than k-means. For the pepper image when k is 4, the convergence time using my k-means implementation is about 72 seconds, while the convergence time using k-means++ is only 52 seconds. However, I have to say that the result is not stable for k-means algorithm because it highly

relies on the centroids initialization. But I would say that for most of time, k-means++ is stable and reliable.

Figure below are the results from another image using k-means++ with different k value.

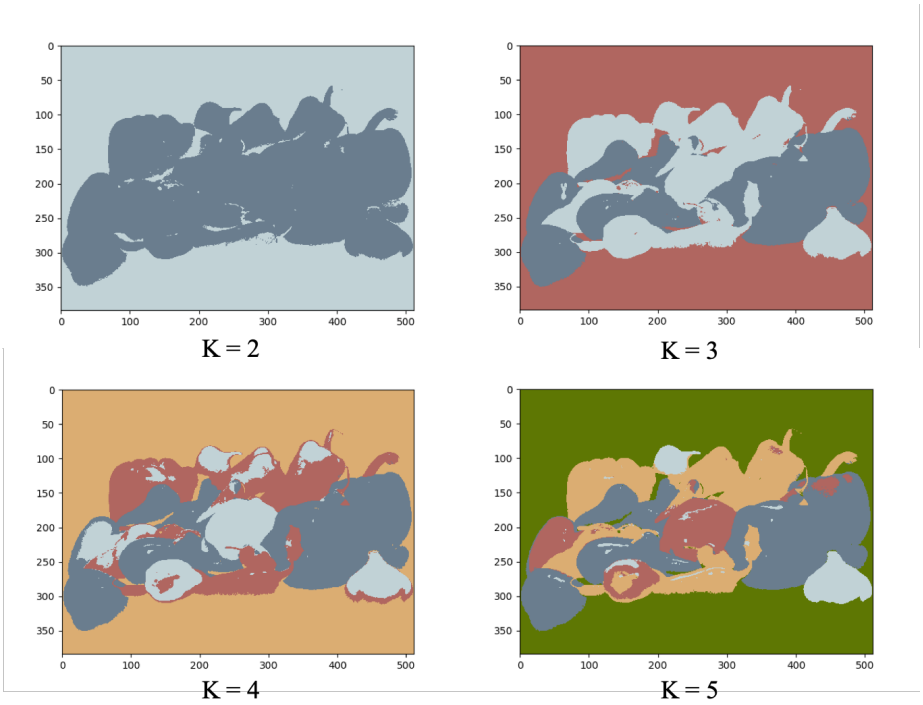



Figure 16: K-means++ color segmentation with different k values

# 3 Task 3 Face recognition using Eigenface

1. Alignment is very important in eigenfaces because we need to find the linear combination of a face using eigenfaces. We have to make sure that nose, mouth and eyes are in the same coordinates, otherwise we may be looking for a nose

with the linear combination of cheeks or something else, that is definitely not accurate.

2.

(a) Here is the code to load the image.

```
def loadImg(files, num):
    faces = np.mat(np.zeros((num, 231 * 195)))
    j = 0
    for i in os.listdir(files):
        img = cv2.imread(files + i, 0)
        faces[j, :] = np.mat(img).flatten()
        j += 1
    return faces
```

Figure 17: Load the image from a specific directory

This is the mean face.

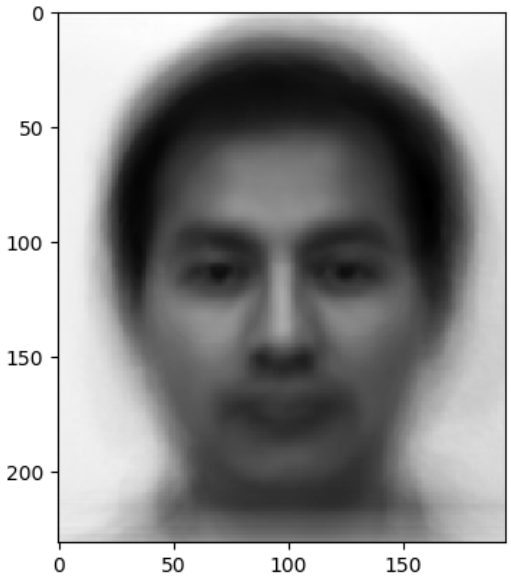



Figure 18: The mean face

(b) Here is the PCA implementation.

```
def eigenface(faces):
    # Calculate the mean face from the data set
    mean = np.mean(faces, axis=1)
    # Normalize the train set
    trainset = faces - mean
    # Calculate the eigenvectors and eigenvalues
    S = trainset.T * trainset
    eigval, eigvec = np.linalg.eigh(np.mat(S))
    # 95% of the variance
    v = np.sum(eigval) * 0.95
    pc = 0
    # Sort the eigenvalues in descending order
    sortedInd = np.argsort(-eigval)
    sum = 0
    # Find how many eigenvalues consists 95% of variance
    for i in range(len(eigval)):
        sum += eigval[sortedInd[i]]
        if sum > v:
            pc = i + 1
            break

    # Make k to be 12
    pc = 12
    eigfaces = dict()
    # Find the first k eigenvectors of covariance matrix
    eigvec = trainset * eigvec[:, sortedInd[0 : pc]]
    # reshape eigenfaces and plot them
    for i in range(pc):
        eigfaces[i] = np.reshape(eigvec[:, i], (231, 195))
        plt.imshow(eigfaces[i], 'gray')
        plt.show()
    return mean, eigvec, trainset
```

Figure 19: PCA implementation

A faster way to compute eigenvalues and eigenvectors are to compute the

eigenvalues of $T^T T$ instead of the covariance matrix S (T is the training set matrix we get from first step). In this case, the covariance matrix is huge while $T^T T$ is much smaller. We can prove that their eigenvalues are same. Their eigenvectors can be calculated by multiplying T.

(c) In this case, at least 59 principal components are needed to capture 95% variance.

(d) These are the eigenfaces.

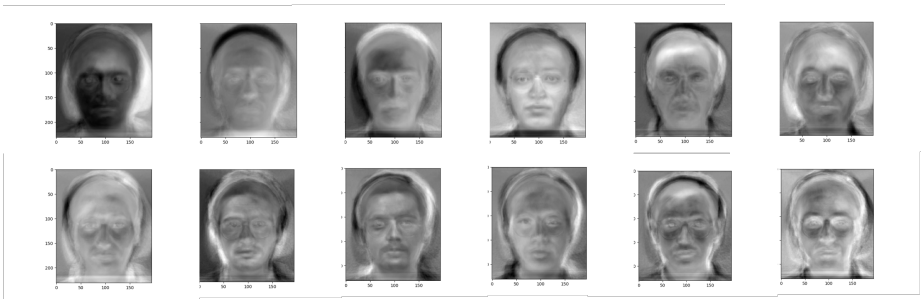


Figure 20: Eigenfaces

(e)

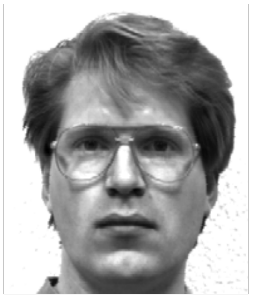
Here are the results:

**Test image**

**Three most similar image in training set**

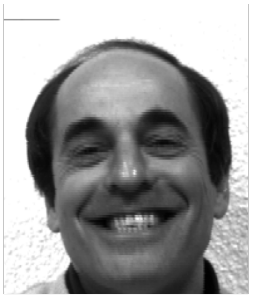
**Test image**

**Three most similar image in training set**



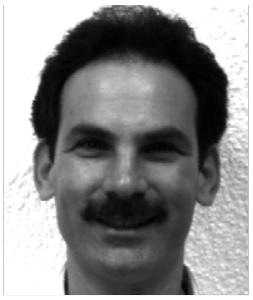
**Test image**

**Three most similar image in training set**

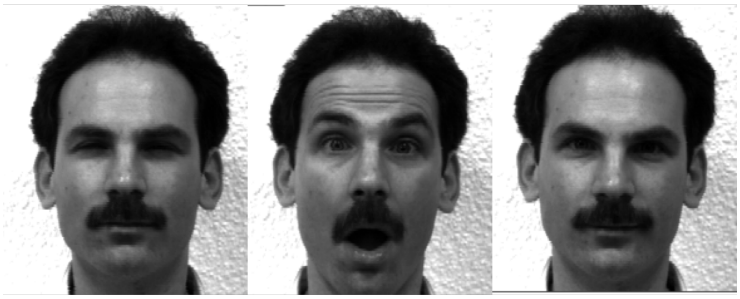


**Test image**

**Three most similar image in training set**



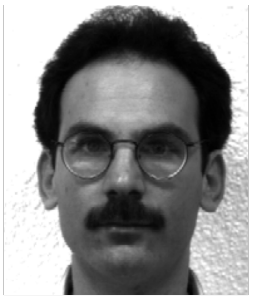
**Test image**



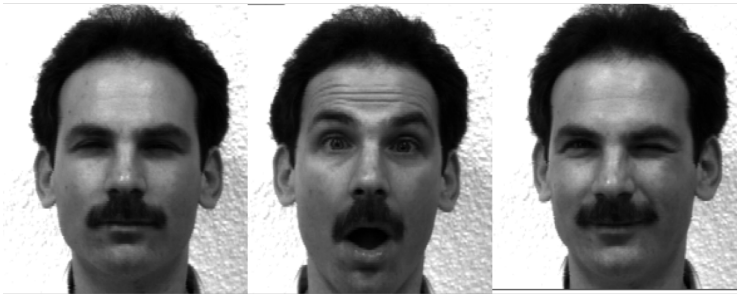
**Three most similar image in training set**

**Test image**

**Three most similar image in training set**



**Test image**



**Three most similar image in training set**

**Test image**

**Three most similar image in training set**

**Test image**

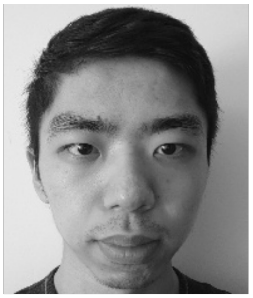
**Three most similar image in training set**

**Test image**

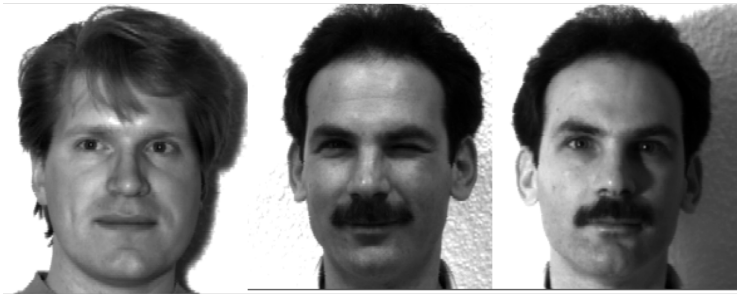
**Three most similar image in training set**

In the results, the left image is the most similar image, the middle one is the second. If we only see the most similar image, there is only one that is the wrong person. If we consider three images, one person gets wrong for the second and third similar image, another one gets wrong only for the most similar image. I would see accurate is high. Most of the recognition is correct.

(f) Here is my own face and three most similar faces if I don't add my own faces into the training set.

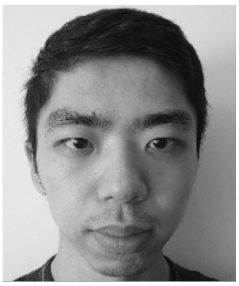


**Test image**

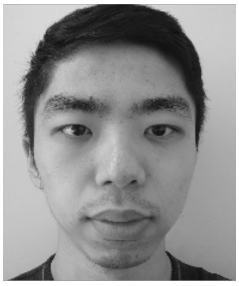
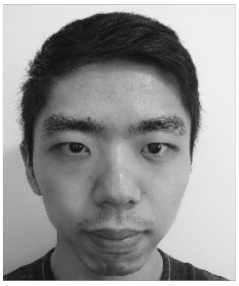
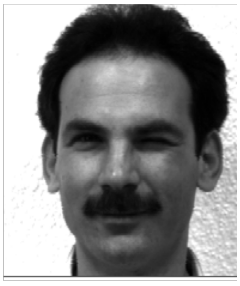


**Three most similar image in training set**

(g) If we add the my faces in the training set. Here is the result.



**Test image**

**Three most similar image in training set**