

**ENGN - 4528**  
**COMPUTER VISION**

**CLAB - 2 REPORT**

Samyak Jain  
u6734495

08/03/2020

# CONTENTS

## PAGE

<b>TASK- 1 HARRIS CORNER DETECTOR</b> .....	3
1.1 OUTPUT FROM HARRIS CORNER .....	5
1.2 COMPARING WITH INBUILT FUNCTION .....	6
1.3 FACTORS AFFECTING HARRIS DETECTOR .....	7
<b>TASK- 2 K- MEANS CLUSTERING</b> .....	7
2.1 LAB COLOUR SPACE .....	8
2.2 OUTPUT FROM LAB IMAGES (3D) .....	8
2.3 OUTPUT FROM LAB IMAGES (5D) .....	9
2.4 K-MEANS ++ INITIALIZATION .....	10
2.5 OUTPUT FROM KMEANS++ .....	11
2.5.1 KMEANS++ ON RGB .....	11
2.5.2 KMEANS++ ON LAB(3D) .....	11
2.5.2 KMEANS++ ON LAB(5D) .....	11
<b>TASK- 3 EIGENFACES DECOMPOSITION</b> .....	13
3.1 READING IMAGES AS 2-D MATRIX .....	13
3.2 PCA ALGORITHM .....	13
1.3 RECOGNIZER FUNCTION .....	15
3.4 RECOGNIZER RESULTS .....	16
<b>REFERENCES</b> .....	19

# 1. TASK - 1: HARRIS CORNER DETECTOR

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

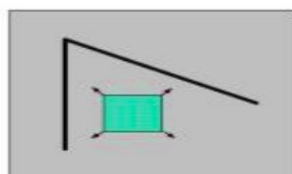
The “fspecial” function which is created in the harris.py file generates a 2-D gaussian filter taking shape of the filter needed and sigma as its parameters. This fspecial function uses the same formula which was used in our lab-1 assignment for creating the gaussian filter.

$$G(x,y) = \frac{1}{2\pi\sigma^2} \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}$$

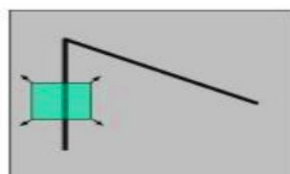
Here the fspecial function creates a 2-D gaussian filter of the shape of the maximum value between 1 and (np.floor(3 \* sigma) \* 2 + 1). Out of these two values if maximum value is 1 then the fspecial function would generate a filter of shape (1,1). Alternatively if the maximum value is given by (np.floor(3 \* sigma) \* 2 + 1) then the function would generate a 2-D filter of the value given by calculating the above operation.

```
def cornerness(img, window_size):  
    for y in range(height):  
        for x in range(width):  
            #summing over the derivatives of the image over all window  
            # creating the summed M matrix  
            Sxx = np.sum(Ix2[y:y + window_size , x : x+window_size])  
            Syy = np.sum(Iy2[y:y + window_size , x : x+window_size])  
            Sxy = np.sum(Ixy[y:y + window_size , x : x+window_size])  
            det= Sxx * Syy - (Sxy**2)  
            tr= Sxx + Syy  
            response = det - thresh * (tr ** 2)  
            R[y,x] = response  
    return R
```

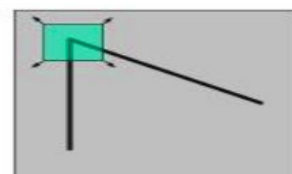
A corner in an image can be interpreted as a junction of two edges , where an edge is a sudden change in the image’s intensity or brightness. Shifting a window on any image on corner points should give a large change in intensity.




Flat region



Edge Region



Corner Region

$$E(u, v) = \sum_{x, y} w(x, y) [I(x+u, y+v) - I(x, y)]^2$$


This equation defines the sum of squared differences, where  $u$  and  $v$  are the  $x, y$  coordinates of our image in every  $3 \times 3$  window.  $I$  is the intensity value of the pixel. For corner detection we have to maximize this function. Applying Taylor expansion to the above equation we get :

$$E(u, v) \approx [u \quad v] \left( \sum \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \right) \begin{bmatrix} u \\ v \end{bmatrix}$$

Where the summed matrix of all the derivatives of the image ( $I_x, I_y, I_{xy}$ ) can be renamed as  $M$ . A score  $R$  is calculated for each window of image which can be given by the formula :

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

$$\det M = \lambda_1 \lambda_2$$

$$\text{trace } M = \lambda_1 + \lambda_2$$

The value  $R$  created by the equation above is the same as the output of our cornerness function where we have summed over the derivatives and applied the formula.

Now we proceed with our next part of the task that is to implement Non-Maximal Supression on our cornerness function. We know that the value  $R$  would be positive for corners, negative for edges and small for flat regions. So we implement NMS that finds out a 8-way local maxima for every window in the image. Our matrix  $R$  contains positive values for all those points where it sees a shift in intensity. But we only want those positive values which are the maximum in every window of the image and we want them to be detected only once. So we loop through every window in  $R$  and keep only those values which are greater than the mean of all positive values in that matrix  $R$ . This way we can ensure that small positive values which could be edges are not detected. We find out the corresponding coordinates of those values which are higher than mean and store them in a matrix, where each row is a coordinate where the corner was detected. Once we find these values we just mark these coordinates as red dots on the copy of the image and hence we can see the corners detected successfully.

```

def nonMaximalSupress(image,kernel_size)
    for x in range(0,M-2):
        for y in range(0,N-2):
            window = image[x:x+kernel_size, y:y+kernel_size]
            localMax = np.amax(window)
#Setting threshold to take only those values which are greater than
# mean of the R matrix
            if localMax > R[R>0].mean():
                maxCoord = list(np.unravel_index(np.argmax(window,
axis=None), window.shape))
                maxCoord[0]+=x
                maxCoord[1]+=y
                coord.append(maxCoord)
                local.append(localMax)
            coord_matrix = np.reshape(coord,(len(coord),2))
return coord_matrix

```

## 1.1 OUTPUT FROM OUR HARRIS CORNER DETECTOR



Fig. 4(a) Harris\_1



Fig. 4(b) Harris\_2



Fig. 4(c) Harris\_3

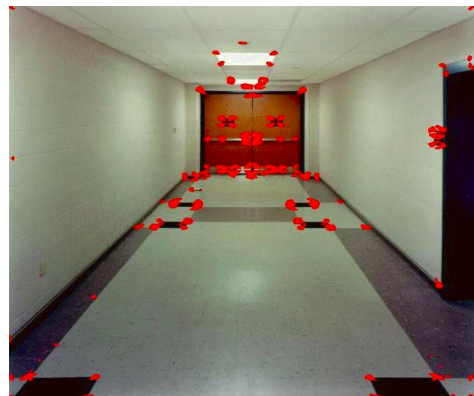


Fig. 4(d) Harris\_4

## 1.2 COMPARING OUTPUTS WITH INBUILT FUNCTION



Fig.5 (a) My Harris\_1



Fig.5(b) Inbuilt Harris\_1



Fig.5 (c) My Harris\_2



Fig.5(d) Inbuilt Harris\_2



Fig.5 (e) My Harris\_3



Fig.5(f) Inbuilt Harris\_3



Fig.5 (g) My Harris\_3



Fig.5(h) Inbuilt Harris\_3



## 1.3 FACTORS AFFECTING HARRIS CORNER DETECTOR

There are multiple factors that would affect the implementation of our harris corner detector. Firstly, the sigma value given for creating the gaussian filter would have a big role. Giving higher sigma ( $\sigma > 2$ ) would create more smoothed image and changing intensity where corners are detected would vary a lot. This would help in detecting corners more easily. Secondly, The empirical constant  $k$  would also affect the weights associated to the trace( $M$ ). Third and the most important, in our algorithm we have set the threshold to take only those points which are greater than the mean of the all positive values in  $R$ , but this may depend on the threshold set by the user. A lower threshold would detect points as corners which in reality might not be a corner point.

## TASK-2 : K-MEANS CLUSTERING

K-Means clustering is a method of vector quantisation. It aims to partition  $n$  data points into  $k$  clusters in which each cluster is represented by a centroid which is the mean of all data points belonging to a cluster.

```
def k_means(im,k):
    """input : takes an image and the number of clusters needed
       output : an image segmented into the k clusters
    """
    centroids= get_centroids(im,k)
    old_centroids=[]
    for i in range(20):
        old_centroids = centroids
        clusters = find_clusters(ima,centroids)
        centroids = new_cent(old_centroids,clusters)
    for x in range(h):
        for y in range(w):
            ima[x,y]=centroids[min_dist(ima[x,y],centroids)]
    return ima
```

The simple `k_means()` function takes the image and the number of clusters as input and returns the image segmented into  $k$  clusters. The `get_centroids()` function creates  $k$  random centroids at first. The `find_clusters()` returns a dictionary in which keys are clusters numbers and values are the pixels associated to that cluster. The `new_cent()` finds the new centre by calculating the mean of all pixels in the associated cluster. Whereas our `min_dist()` returns the index of the pixel from which minimum distance was found.

At last we replace all the pixels in the image by their corresponding centroids and hence it results in an image of  $k$  colour palettes.

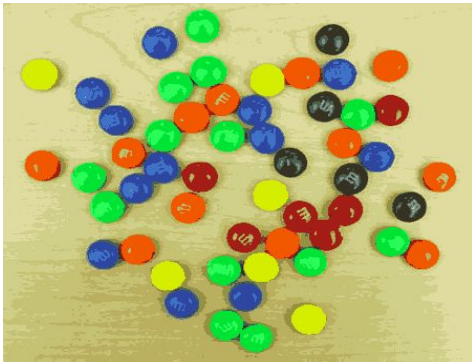


Fig.6(a)  $k = 15$ (random)



Fig.6(b)  $k = 15$ (random)

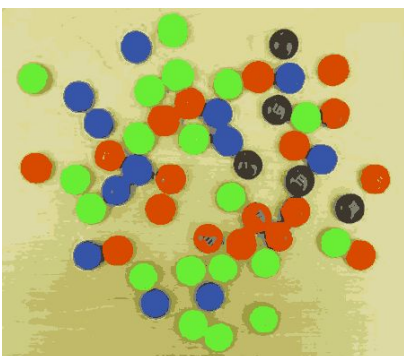
## 2.1 LAB COLOUR SPACE

To convert our image into  $L^*a^*b^*$  color space we have made use of the skimage 's color inbuilt package "rgb2lab" . It expresses color as three values:  $L^*$  for the lightness from black (0) to white (100),  $a^*$  from green (-128) to red (+128), and  $b^*$  from blue (-128) to yellow (+128). It is mainly designed to approximate human vision. Since it is not possible to display the lab colour space images, so after running our k-means algorithm on this colour space we revert the image back to RGB when plotting and then multiply all the pixels by 255 to save the LAB output.

```
for x in range(h):
    for y in range(w):
        ima[x,y]=centroids[min_dist(ima[x,y],centroids)]
plt.imshow(color.lab2rgb(ima))
pic = Image.fromarray((color.lab2rgb(ima)*255).astype('uint8'))
```

## 2.2 OUTPUT FROM LAB IMAGES WITHOUT COORDINATES

**K = 10**



**K =15**





Our next step is to convert our LAB colour space into a 5d vector where the 4th and 5th dimension would contain our pixel coordinates x and y . First we make 5d array containing all zeros . In the first 3 dimensions we append the our L\*,a\*and b\* pixels of our LAB image. Now after appending these pixels , our next step would be to append x and y pixel coordinates to our 4th and 5th dimension.

```
def make_lab5d(im):
    for x in range(height):
        for y in range(width):
            lab5d[x,y,0:3]=im[x,y,0:3]
            lab5d[x,y,3]=x
            lab5d[x,y,4]=y
    return lab5d
```

Now we run our k means algorithm on our Lab\_5d image vector. Again, it is not possible to display our image as a 5d vector . So before plotting we convert our Lab\_5d image' s L\*,a\*,b\* back to RGB colour space and multiply the pixels by 255 as we did it previously with our LAB image.

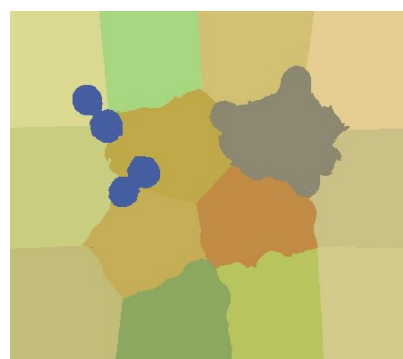
```
for x in range(h):
    for y in range(w):
        ima[x,y]=centroids[min_dist(ima[x,y],centroids)]
plt.imshow(color.lab2rgb(ima[:, :, 0:3]))
pic = Image.fromarray((color.lab2rgb(ima[:, :, 0:3])*255).astype('uint8'))
```

## 2.3 OUTPUT FROM LAB IMAGES WITH COORDINATES

**K = 10**



**K = 15**



## 2.4 K - MEANS++ INITIALIZATION

K-Means clustering starts with allocating random or arbitrary centers and then looks for better centers as the algorithm proceeds. As a result of randomisation our algorithm might get stuck in local minima as our initial set of centers might not be distributed over our data set. K-Means++ algorithm starts by allocating one random centre and then searches for other centers using the first one. Since it assigns center using the previous centres, this algorithm might give us centers that are distributed over our data. This algorithm is more likely to have less within cluster sum of square than normal k-means initialisation.

The k-means++ algorithm steps are as follows:

- Let  $D(x)$  be the shortest distance between a data point and the closest center we have already chosen.
- Initially pick a random center  $c_1$ , taken uniformly at random from data  $X$
- We take a new center  $c_i$  from  $x \in X$  with probability  $D(x)^2 / \sum_{x \in X} D(x)^2$
- Now we will repeat step 3, until we have  $k$  centers
- Then we will run our normal k-means algorithm.

```
def kpp(im1,K):
    for k in range(1,K):
        dist=[]
        for x in range(im1.shape[0]):
            for y in range(im1.shape[1]):
                for c in cent:
                    d.append(np.sum((im1[x,y]-c)**2))
                dist.append(min(d))
            d=[]
        prob=dist/np.sum(dist)
        cum_prob=np.cumsum(prob)
        random_sample=random.random()
        for j,l in enumerate(cum_prob):
            if random_sample < l:
                i=pixels[j]
                break
        cent.append(i)
    return cent
```

In our code we first initialize a random center and then implement the algorithm described above for all centers to be generated till  $k$ . Same way we find the probabilities and evaluate the cumsum probabilities. Then we create a random number from 0 to 1 and check for the number in cumsum probabilities that is just greater than this randomly generated number and assign its corresponding pixel as our new center.

## 2.5 OUTPUT FROM K-MEANS++ ALGORITHM

### 2.5.1 K-MEANS ++ ON RGB IMAGE



Fig.7(a) K =10 with random



Fig. 7(b) K =10 with K-means ++

### 2.5.2 K-MEANS ++ ON LAB IMAGE



Fig.8(a) K =10 with random



Fig. 8(b) K =10 with K-means ++

### 2.5.3 K-MEANS ++ ON LAB WITH COORDINATES (5D)



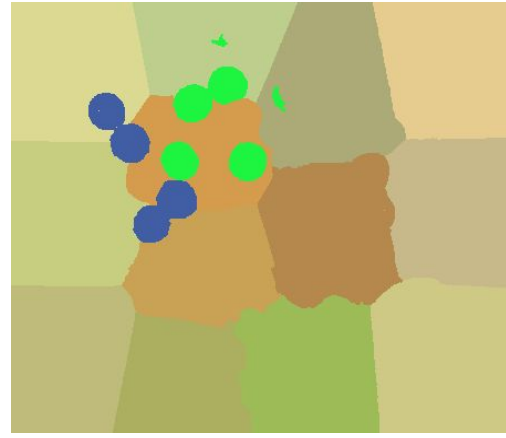
Fig.9(a) K = 10 with random



Fig. 9(c) K =10 with K-means++



**Fig .10(a) K = 10 with random**



**Fig. 10(b) K =10 with K-means++**



**Fig .11(a) K = 10 with random**



**Fig. 11(b) K =10 with K-means++**



**Fig .12(a) K = 15 with random**



**Fig. 12(b) K =10 with K-means++**

After implementing the k-Means++ algorithm, we notice that computation time for k-means++ is a lot more than standard k means since it searches for new centers by using the first randomly selected centroid. The outputs from our kmeans ++ are somewhat better or equal to outputs from standard algorithm . When the initialization from the standard algorithm is bad , then kmeans++ would give a better output as we can see in **Fig.10(a)** and **Fig.10(b)**.

## TASK -3 EIGENFACES DECOMPOSITION

The eigenfaces is a set of name given to the eigen vectors that is used to recognise faces in computer vision. The basic idea of this method is to treat every image as a vector and construct a low dimensional linear subspace that best explains the variations in the set of images. After applying this method faces are recognized by the nearest neighbour method :



$$\mathbf{y}_1 \dots \mathbf{y}_n$$

$$k = \underset{k}{\operatorname{argmin}} \left\| \mathbf{y}_k^T - \mathbf{x} \right\|$$

The method of eigen faces is sensitive to scale, lightning, pose and facial expression. So it is very important to align and center all our images before we start applying the method. It can be seen as a form of data normalisation, just as we normalise or scale any feature vectors by zero centering or apply min-max normalisation to get the inputs in the range 0-1 before putting through a machine learning model, same way align and centering the images before is a very important step.

### 3.1 READING THE FACE IMAGES AS 2-D MATRIX

```
face_vector = []
for file in os.listdir("Yale-FaceA/trainingset"):
    if file.endswith('.png'):
        face_image=
cv2.cvtColor(cv2.imread("Yale-FaceA/trainingset/"+file),cv2.COLOR_RGB2GRAY
)
        face_image = face_image.flatten()
        face_vector.append(face_image)
face_vector = np.asarray(face_vector)
A_pp = face_vector.transpose()
```

The first step is to make sure that we treat every image as a vector. With the help of python's os package we have made a 2-D matrix of images where every column of A\_pp is image vector .

### 3.2 PCA ALGORITHM

The first step in PCA algorithm is to find the mean of all the images (all N data points in a column). The mean image of our yale dataset looks something like this:





**Fig.13 Mean Image**

Since our images in 2-D space has 45k rows and 135 columns using  $A * A^T$  for the computation of covariance and eigen value decomposition will result in a large matrix. Computation complexity and time complexity would be very huge for these computations. Instead, we have taken the formula  $A^T A$  for our computation which will result in a (135,135) matrix and hence has low computation complexity as well as time saving.

After computing the covariance matrix, our next step would be to compute the top k eigenvectors. For computing eigenvectors we have taken the help of numpy's `linalg.eig()` function. Our next step is to select top k eigenvectors and visualize them. We have taken  $k=15$ . Our top 15 eigenvectors are displayed below:



Since the covariance matrix we created above is symmetric , the eigen vectors ( $u_1, u_2, u_3 \dots u_N$ ) of this matrix will form a basis which means any vector ( $x - \bar{x}$ ) can be written as a linear combination of the eigen vectors:

$$x - \bar{x} = b_1 u_1 + b_2 u_2 + \dots + b_N u_N = \sum_{i=1}^N b_i u_i$$

To reduce further dimensionality we will only keep terms that correspond to K largest eigen values. The linear transformation  $R^N \rightarrow R^K$  which performs dimensionality reduction is given by:

$$\begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_K \end{bmatrix} = \begin{bmatrix} u_1^T \\ u_2^T \\ \dots \\ u_K^T \end{bmatrix} (x - \bar{x}) = U^T (x - \bar{x})$$

As the last step of your PCA algorithm we find out the weights associated with each eigenfaces by multiplying the normalised faces with the eigenfaces that we computed. This results in a (135,15) matrix where each row consists of weights of the all training images associated to the 15 eigenfaces.

### 3.3 RECOGNIZER FUNCTION

```
def recognizer(filename):
    test_img = np.array(Image.open("Yale-FaceA/testset/"+filename))
    test_img = test_img.flatten()
    test_img = test_img.reshape(231*195,1)
    test_normalized_face_vector = test_img - reshaped_mean
    test_weight =
np.transpose(test_normalized_face_vector).dot(np.transpose(eigen_faces))

    index = (np.linalg.norm(test_weight - weights, axis=1))
    min_3 = np.argsort(index)[:3]
```

Our Recognizer function takes the filename of the test set as an argument and returns the index of the closest 3 pictures in our training set. First we read our test image as an array and reshape it to a 1-D column vector. Next we determine the projection of the test image onto the basis spanned by our top k eigen faces. From these projections again we find out the weights corresponding to the top 15 eigenfaces against our test image. Implementing a nearest neighbour search over all our images in the training set with the help of numpy's `linalg.norm()` function we return the corresponding indices of the minimum 3 distances which were calculated.

The images on these indices in our training set are likely to be the closest match to the testing image according to our algorithm.

### 3.4 RECOGNIZER RESULTS



**Correct Predictions = 3/3**

**Accuracy = 100%**



**Correct Predictions = 3/3**

**Accuracy = 100%**



**Correct Predictions = 3/3**

**Accuracy = 100%**



**Correct Predictions = 3/3**

**Accuracy = 100%**

Test Image



**Correct Predictions = 1/3**

**Accuracy = 33.33%**

Test Image



**Correct Predictions = 3/3**

**Accuracy = 100%**

Test Image



**Correct Predictions = 3/3**

**Accuracy = 100%**

Test Image



**Correct Predictions = 1/3**

**Accuracy = 33.33%**



Test Image



**Correct Predictions = 3/3**

**Accuracy = 100%**

Test Image



**Correct Predictions = 3/3**

**Accuracy = 100%**

**Overall Accuracy of our model = 86.66%**

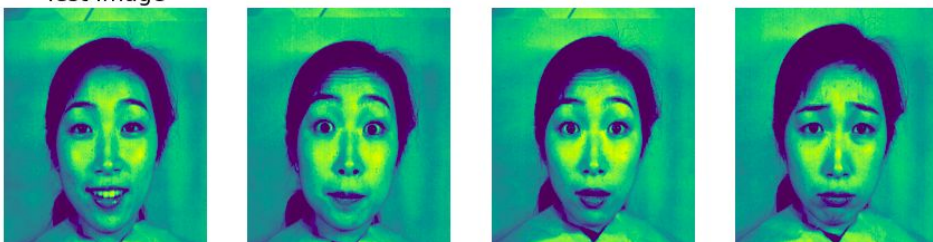
Since for personal reasons I have not taken my image for testing purposes .I have used the JAFFE( The Japanese Female Facial Expression ) database. After realigning and centering the images we split 9 images into the training set(not included in the training set yet) and one image into the testing set. After reading the image in the testing the output of our 3 closest images were as follows :

Test Image



After including rest 9 images of my face into our training set then running our recognizer function , the 3 closest images to our test image were as follows :

Test Image





We observe that before adding our faces into the training set , the algorithm shows different faces that are close to my image in the test set . But after adding the rest of our 9 images into the training set , the algorithm predicts the closest 3 images detected as my own image which shows that our algorithm is working relatively well.

## REFERENCES

- [1] JAFFE Dataset : Michael J. Lyons, Shigeru Akamatsu, Miyuki Kamachi, Jiro Gyoba.  
Coding Facial Expressions with Gabor Wavelets, 3rd IEEE International Conference on Automatic Face and Gesture Recognition, pp. 200-205 (1998).  
<http://doi.org/10.1109/AFGR.1998.670949>  
Open access content available at: <https://zenodo.org/record/3430156>
  
- [2] <http://ilpubs.stanford.edu:8090/778/1/2006-13.pdf>
  
- [3] <https://en.wikipedia.org/wiki/Eigenface>