18-551, Fall 2009
Group 1
Find MY FACE!
Saurabh Sanghvi (saurabh.sanghvi@gmail.com)
Ping-Hsiu Hsieh (phsieh@andrew.cmu.edu)
Maxwell Jordan (maxjordan@gmail.com)
1. Problem & Introduction

Face recognition systems can be very useful in online photo albums such as Facebook. On Facebook, users can post albums from vacations and activities with themselves and their friends. The users can then manually choose squares around the person in the picture and identify who that person is if that person is also a Facebook friend. On Facebook, this process is called “Tagging” a friend. The process of tagging your friends when you upload a whole album of photos online can be long and boring, but it makes sharing photos with your friends much easier. The motivation of our project is to automate this process. However, images posted online can be arbitrary. They could be taken in complex scenarios where the background of an image is not a uniform color. Therefore, a working face detection algorithm must first be implemented before any recognition step can be performed.

For the face detection part of our project, our goal is to detect faces that can have either indoor or outdoor lighting conditions. Out of the many variations that faces posted in online albums can have, different positions of light sources is one. For the face recognition part of our project, our goal is to recognize the same face image with different positions of the light source.

Since this can be a very challenging task, we will impose some limitations on the input images that we will be using:

Limitations with respect to face detection:
1) Faces are frontal.
2) Faces are roughly of the size 60x60.

Limitations with respect to face recognition:
1) The total number of Facebook friends that will be recognized is 20.

Our system will take color images in JPEG format of size 604x453 as our input images since this is most common size of the Facebook images. The input format is chosen because when the user tries to download a Facebook image from a browser, the default file format is JPEG.

2. Novelty

Novel Part of Our Face Detection Algorithm

Several past projects in 18-551 have done face detection. Most of the face detection algorithms we saw that were used in 551 were based on Group 2 Spring 2005 - "Face Detection for Mobile Device." Their algorithm consists of the following steps. The first step, we will refer to as “skin detection”, involves thresholding color pixel values for skin tones. Then binary morphological processing is applied to get rid of any false detection from the skin detection. After morphological processing, blob coloring is done to label each region (skin blob) of the input image that is potentially a face with a unique number. In this step, it is often necessary to also compute different properties such as width, height, and mass (area) of each skin blob. The last step, blob rejection, is to further throw out skin blobs that are most likely not faces. Since the past projects also had limitations on the size of the faces that they would try to detect, the blob rejection is often done by looking at the mass, width/height ratio, and height/width ration of the skin blobs.

Their algorithm is based on the assumption that faces in the input image are far from one and another, so that each individual face results in an individual skin blob. It fails when two faces are very close to each other in the input image that the detected skin blobs of the two faces are labeled as only one blob during blob coloring.

Figure 1 and Figure 2 illustrate when the algorithm would fail. [Note: Throughout the report, the binary images we show use the following convention: the white pixels are what we are interested, and the black pixels are background.] Two faces are present in Figure 1, but after the skin detection step, blob coloring would only find one blob. As a result, the face detection algorithm would either identify that blob as only one face if the skin blob happens to pass the
blob rejection scheme or reject that blob as a face if the skin blob does not pass the blob rejection scheme.

The novel part of the face detection algorithm that we will be using involves searching for eyes in the detected skin blobs. We will increase the face detection success rate by checking how many eye pairs exist in the detected skin blobs. For example, in Figure 1, we would find two faces instead of just one because we found two eye pairs in that skin blob.

Eye detection is also important because it could be used to register the face images detected so that the images are aligned with training images in the recognition part.

**Novel Part of Implementation on DSK**

We implemented blob coloring iteratively (as supposed to recursively) in C on the DSK. This has not been done in past projects. Most of the reports we saw stated that they used the Matlab command bwlabel() on the PC side.

**3. Algorithms**

*Face Detection*

![Diagram of face detection process]

**A. Color Compensation**

We need the image to be properly white balanced and for the colors to appear as accurately as possible. We will later use these colors to detect skin tones as the first step of face detection, hence the importance of this step. Different light sources emit different color light. The human eye is exceptional at understanding image data irrelevant to light color, however, computer vision algorithms have trouble compensating for such differences. We will use grey point analysis and attempt to shift the individual channel histograms to be centered around the grey point. The idea is that each channel (Red, Green, Blue) of a correctly white balanced image has an average grey value. If there is a strong red cast to an image, the red channel will be shifted towards 255 and have an off center average. We will correct for this by subtracting the difference between the mean value and the grey point value.

It is important to note that for our application, we will more often than not, not require white balancing. It is well known that white balancing and image without knowledge of the image capture device cannot result in a perfect solution. Also, since most images posted on Facebook have been taken with modern digital cameras, additional white balancing may not be necessary and could be potentially damaging if applied. We will therefore have to threshold our white balancing algorithm to not alter an already balanced image. As we go through the different color channels if we find that one channel is particularly removed from the grey point while the others are not, we know there is a definite color shift of one channel. If all the channels are similarly shifted off the grey point, then the shift is constant and not one channel is particularly dominant. If this is the case we will not apply our color balancing scheme.

**B. Skin Detection**

We use an elliptical skin model proposed by Hsu to detect skin pixels in a transformed YCbCr color space [2]. According to Hsu, looking at pixels only in the Cb Cr subspace results in many false positives. He proposes a nonlinear
transformation of the Cr, Cb space into a Cr’, Cb’ color space where the skin pixels in this transformed YCr’Cb’ are luma (Y) independent. Hsu’s experiment shows that skin pixels can be detected using an ellipse in the Cr’Cb’ plane. If the pixel in the Cr’Cb’ plane lies inside the ellipse, then we will say it’s skin; otherwise the pixel is not skin.

To use the transformed YCbCr color space, we first need to transform RGB channels of the input image to YCbCr channels. We use the equation provided in Hsu’s paper: [2]

\[
\begin{bmatrix}
Y \\
Cb \\
Cr
\end{bmatrix} = \frac{1}{256} \begin{bmatrix}
65 & 129.057 & 25.064 \\
-37.945 & -74.494 & 112.439 \\
\end{bmatrix} \begin{bmatrix}
R \\
G \\
B
\end{bmatrix} + \begin{bmatrix}
16 \\
128 \\
128
\end{bmatrix}
\]

Equation 1 [2]

Notice that the resultant pixel values in Y, Cb, Cr will be not be integer numbers as the pixel values in the R, G, B channels that we started with. We tried rounding the result into integer values, but the skin detection result was worse than the result if we had just kept the Y, Cb, and Cr values as floating point numbers. This is the reason why we have to keep the pixel values for Y, Cb, and Cr on the DSK as arrays of floats.

Next, we will transform the Cb and Cr pixels values to get Cb’ and Cr’ pixel values. Once we have the Cb’ and Cr’ pixel values, we will be able to tell if a given pixel is skin or not. Therefore, the result of the skin detection algorithm is a binary image where value 1 represents a skin pixel and value 0 represents non-skin. We will refer to this binary image that resulted directly from Hsu’s elliptical skin model as the “original_skin_map” throughout the rest of the report.

The complete set of equations used to transform the Cb and Cr pixels values to get Cb’ and Cr’ pixel and the ellipse used to threshold skin pixel values can be found in Hsu’s paper[2]. We will highlight the reason that Hsu uses the Cr’, Cb’ color space.

Figure 3 shows the plot of skin pixels that Hsu sampled in the YCbCr color space. Notice that the skin pixel is dependent on Y. To correctly threshold skin pixels in this color space using Hsu’s skin model would require examining Y value, Cb value, and Cr value of any given input pixel.
Figure 4 The nonlinear transformation of YCbCr color space (a) The transformed YCbCr color space (b) a 2D projection of (a) in the Cb-Cr subspace, in which the elliptical skin model is overlaid on the skin cluster [2].

Figure 4(a) is the exact same skin model, but in the YCb’Cr’ color space. Notice that now the skin pixels become independent of Y. Since it is more like a cylinder now. To correctly threshold skin pixels in this color space using Hsu’s skin model would only require examining Cb’ and Cr’ values of any given input pixel. Hsu uses an ellipse on the Cb’ Cr’ plane to threshold skin pixel values. The ellipse is drawn in green in Figure 4(b).

C. Imfill + Morphological Processing

The purpose of this step is to remove any skin blob in the binary image, original skin map, that is smaller than a disk of radius 25. (Since the size of faces in the input images is 60 x 60.) In other words, binary morphological opening (erosion then dilation) needs to be applied on original skin map.

There is, however, a problem if we apply binary erosion directly onto original skin map. The problem is that if any holes exist inside the region that is supposed to be face. It is possible that erosion would simply wipe out the whole skin region. Figure 5 is an example of such problem. In this example, we break a single erosion with a disk structuring element of radius 25 into 5 steps where each step is a single erosion with a disk structuring element of radius 5.

Figure 5 Problem if binary erosions applied directly on original skin map
(a) Original Image; (b) Result of skin detection on a; (c) b after one binary erosion with a disk of radius 5; (d) b after two binary erosions with disk of radius 5; (e) b after three binary erosions with disk of radius 5; (f) b after four binary erosions with disk of radius 5; (g) b after five binary erosions with disk of radius 5.

The problem in Figure 5 is that the original skin map of the input image has holes inside the skin region that is actually of interest to us (a face). And because of the holes, the face blob is completed gone by the time the 4th erosion. This is bad because even after binary dilation is done on (g), there will be absolutely no way to recover the face blob.
The solution to this problem would be to fill in any holes that are inside an enclosed skin blob. Matlab has a command imfill() to do so, therefore, we will refer to our implementation of this operation as “imfill” throughout the rest of the report. In Figure 6, we will show how imfill solves the problem given illustrated in Figure 5.

![Image](a) (b) (c) (d) (e) (f) (g) (h)

Figure 6 Applying binary erosion on imfilled original_skin_map.
(a) Original image; (b) result of skin detection on a; (c) b after imfill; (d) c after one binary erosion with disk of radius 5; (e) c after two binary erosions with a disk of radius 5; (f) c after three binary erosions with a disk of radius 5; (g) c after four binary erosions with a disk of radius 5; (h) c after five binary erosions with a disk of radius 5

As shown in Figure 6 (h), the face blob now does not just disappear after 5 times of erosion. If binary dilation is applied on (h), the proper size of the face blob can then be obtained. But we will first explain how we implemented imfill.

**Imfill**

Imfill is simple once you have a working blob coloring function that is able to find out the maximum and the minimum coordinates of each blob in both the horizontal and vertical directions. (i.e: x\_max, x\_min, y\_max, y\_min)

Up until this point, we have not talked about our implementation of blob coloring. But we will first explain how we implemented imfill in C on the DSK, assuming that we already have blob coloring working. We will use an example to illustrate how it works. We will be looking at a binary image of the size 7 x 6 (Figure 7(a)). We can see that inside the blob in Figure 7(a), there are two holes. We will show how these two holes can be filled.

We will index the rows and columns starting from zero, so MAX\_COL=5 and MAX\_ROW=6. If blob coloring is applied to label Figure7 (a), the result would be 1 blob found with x\_min=1, x\_max=4, y\_min=1, y\_max=5.
Figure 7 (a) a 7x6 binary image that we want to perform imfill on. (b) Blob color result of (a). (c) Inverted version of (a). (d) Blob color result of (c). (e) Because blob#1 is connected to the edge, blob#1 is left untouched. (f) because blob#1 is NOT connected to the edge, we remove blob#2. (g) Because blob #3 is not connected to the edge, we remove blob 3. (h) End result of imfill.
The first step in imfill is to invert the binary image that you want to fill in. By inverting, we mean changing the 1s to 0s and vice versa. Then you want to perform blob coloring on the inverted image. As shown in Figure 7(d), the inverted version has a total of three blobs.

What we do next is go through each blob found in the inverted image, and check if the blob is connected to the edge of the image. As long as one of the following 4 conditions is met, the blob is connected to the edge:

1) x_min==0 (first column)
2) y_min==0 (first row)
3) x_max==MAX_COL (last column)
4) y_max==MAX_ROW (last row)

If a blob is not connected to the edge, then we just kill the blob by filling in zeros for all pixels that were labeled as the same blob number. Once all blobs have been examined, the last step is to invert the resultant image again. Figure 7 (h) is the end result of the imfill operation on Figure 7(a).

**Morphological Processing**

Binary morphological processing is what we used to remove any artifacts that are smaller than the face size we are expecting. Skin detection is great however; hands, arms, and legs will come through. There are also issues with wooden floors or sandy beaches, which can come through skin detection (Figure 8). A method to combat these issues is morphological processing with a structuring element equivalent to the minimum face size.

We will specifically apply morphological opening which is erosion followed by dilation. It is also important to note that before we begin any morphological processing we use imfill. This is necessary to avoid removing faces. To understand this, consider what the erosion operation does. Given a structuring element, say a disk with diameter 25, we will pass the center of the structuring element through every pixel in the binary image. We define the disk as a 50x50 matrix enclosing a diameter 50 circle of 1s with zero everywhere else in the matrix (around the corners). Passing this disk through, we take the minimum value of the multiplication of the structuring element and the relevant pixels of the image. We write the result to the location of the center of the structuring element. Dilation is a similar process except we are concerned with the maximum of the surrounding values. Essentially, with erosion, if there exists any 0 values within the radius of the structuring element, a 0 is written as the result. This is why imfill is necessary, because if an eye or a bright forehead are not detected as skin and retain a 0 value, we could remove an entire face with erosion.

Now that we understand erosion and dilation, we will apply them in a specific order to first remove all the noise in the image. By first doing erosion we can remove hands, t-shirt designs and anything that is not at least as large as our structuring element. The downside to erosion, is that we will remove up to the size of the radius from any faces we want to keep. For this reason, we will use dilation using the same structuring element. By doing dilation second, we will only enlarge objects larger than our minimum face size, and by using the same sized structuring element, we will restore them to their original size. From Figure 9 we can see that all we have remaining are three face blobs and three false positives. We will attempt to remove the false positives later on by doing eye detection. For the remaining part of the report, we will refer to the imfilled original skin_map after morphological opening as the **face_map**, such as Figure 9.
D. Blob Coloring

Once we have the binary image, **face_map**, that tells us all the potential face regions, we do blob coloring on the **face_map**. Later on when we locate the eye blobs, we can check if the two eye blobs are within the same face blob. Blob coloring is also used later to label eye blobs on another binary image, but we will first explain how we wrote blob coloring iteratively in C. Note: We implemented the blob coloring considering the 8-connected neighborhood just because we wanted to consider pixels that are diagonally connected as in the same blob. The other way would be 4-connected neighborhood.

Blob coloring involves scanning the binary input image that you want to label using the filter shown in Figure 13.

‘X’ is the center of the kernel. Each time the ‘X’ position is place on a new pixel in the input image, we first check whether the input pixels in position ‘X’ is a foreground pixel (foreground pixels are pixels of interest). If not, we just move on to the next pixel. If so, we know that the pixel at ‘X’ must be labeled. To decide which blob number we label the pixel at position ‘X’, we want to check if ‘A’, ‘B’, ‘C’, and ‘D’ are foreground pixels. If none of the neighbors are foreground pixels, we just label pixel at ‘X’ with the next sequential blob number.

For example, assuming we have scanned though the first row and the first pixel in the second row of Figure 14. The kernel is now placed at where ‘X’ is labeled. Since pixel ‘X’ is a foreground pixel, we know that pixel
needs to be labeled. And because none of the neighboring pixels ‘A’, ‘B’, ‘C’, ‘D’ is a foreground pixel, we will just label pixel ‘X’ with blob number 1.

On the other hand, if any of the 4 neighboring pixels is a foreground pixel, we would want to label the center pixel with the lowest blob number that the neighboring pixels were labeled with.

For example, assuming we have scanned through all pixels up until where ‘X’ is labeled in Figure 16. The kernel is now placed where ‘X’ is labeled. Since pixel ‘X’ is a foreground pixel, we know that pixel needs to be labeled. And because neighbors ‘A’ (labeled with blob#1) and ‘C’ (labeled with blob#2) are foreground pixels, we will just label pixel ‘X’ with 1. (Because min(1,2) = 1)

At this point, it becomes apparent that blob#1 and blob#2 are actually the same blob. The way to solve this problem is to keep another array that keeps track of what the numbers that we used in the blob colored result really means. We will call this array **blob_num_record**. The **index** of **blob_num_record** is the blob number we used to label blobs in the blob colored result. The **value** is the real blob number that the index refers to. For example, before we label pixel ‘X’, we really did not know that blob#1 and blob#2 are the same blob. So **blob_num_record** would just look like Figure 18 (value 0 just means that the index was never used to label a blob). But after pixel ‘X’ is labeled, we realized that blob#1 and blob#2 are actually the same blob. So **blob_num_record** would become what’s shown Figure 19.
This summarizes how we wrote the blob coloring algorithm iteratively in C on the DSK. The last question remained unanswered is how big the array `blob_num_record` should be. It turns out that given a binary image of size M x N, we will never find more blobs than (M*N/4).

**E. Eye Map Construction**

Hsu’s paper describes two EyeMap equations to find out where eyes are in a colored image, EyeMapL and EyeMapC.

EyeMapL is the greyscale dilated Y divided by the grey-scale eroded Y where ⊕ is the greyscale dilation operation and ⊖ is the greyscale erosion and the function $g_\sigma$ is the structuring element to be used in dilation and erosion. Hsu used a 3D hemispherical structuring element to perform the greyscale dilation and erosion. The purpose of $\text{EyeMapL}$ is “to emphasize brighter and darker pixels in the luma component around eye regions” [2]. In the luminance channel, it is evident that the properties of the eye and the nature of the concave eye socket present unique characteristics. Immediately outside the eye socket is a region of relatively bright luminance, then as you come into the eye socket the luminance drops around the edge of the eye then spikes up on the white of the eye then drops again around the pupil. Using this property of drastic changes in luminance we can construct EyeMapL.

$$\text{EyeMapL} = \frac{Y(x, y) \oplus g_\sigma(x, y)}{Y(x, y) \ominus g_\sigma(x, y) + 1}$$

Equation 2 [2]

$$(f \oplus g_\sigma)(x, y) = \text{Max}\{f(x - c, y - r) + g(c, r)\} \quad (x - c, y - r) \in F, \quad (c, r) \in G$$

Equation 3 [2]

$$(f \ominus g_\sigma)(x, y) = \text{Min}\{f(x - c, y - r) + g(c, r)\} \quad (x - c, y - r) \in F, \quad (c, r) \in G$$

Equation 4 [2]

In grayscale dilation and erosion, each neighbor has an associated height. A dilation output pixel is computed by first adding each of the neighbor input pixels with its corresponding height, and then taking the maximum (dilation) or minimum (erosion.)

Figure 20 is an example of a hemispherical structure element when $\sigma = 1$. Notice the center of the structuring element has a height of zero and the further away from the center, the more negative the height is. This shows the idea that we are weighting the pixels closer to the center more heavily than pixels further away from the center.
Detailed equations that describes the structuring element can be found in [2].

An important note is that EyeMapL is constructed in potential face regions of the input image. Face regions are pixels with value 1 in the binary image, face_map. Before erosion and dilation is done, we AND the Y-channel of the input image with face_map, so that in face regions, the Y-channel values are kept. For pixels that are non-face, we fill in the average value of the Y-channel values that are in the face region. Figure 21 summarizes how to construct EyeMapL. Note: EyeMapL is a gray-scale image that has high values around the eye regions.

Hsu also describes EyeMapC, which focuses entirely on the chrominance channels of the image [2]. EyeMapC is defined as:

$$E_{Eye\, Map\, C} = \frac{1}{3} \left\{ (C_b^2) + (\tilde{C}_r)^2 + (C_b/C_r) \right\}$$

Equation 5 [2]

where $C_b^2$, $(\tilde{C}_r)^2$, and $C_b/C_r$ are all normalized to [0,255] and $\tilde{C}_r$ is the negative of $C_r$ ($255-C_r$). The idea behind EyeMapC is based on the observation that high $C_b$ and low $C_r$ values are found around the eyes. According to Hsu, the resultant EyeMapC should be histogram equalized to produce better result. However, for our project, we ended up dropping EyeMapC from our process. We found that EyeMapC removed many false positives that the luminance based EyeMapL produced but also removed some actual eyes. We decided it would be best to get as many potential eyes from EyeMapL and remove any false positives through blob detection and statistical analysis.
F. Eye Blob Thresholding

To turn the gray-scale image EyeMapL into a binary image that shows where the eye blobs are, we devise the following threshold scheme:

For each face blob found in face_map, we look at all the pixel values in EyeMapL and compute the median and standard deviation. For each face blob, we are only interested in the upper percentage of pixel values. We will calculate this threshold dynamically per each face blob by using median + standard deviation as the threshold value. This is better than using a fixed number to threshold because a fixed number may work for some faces but not others while a dynamic threshold will work for a wider range of faces.

![Diagram of Eye Blob Thresholding](image)

**Figure 21**

![Image of three face blobs in an EyeMap](image)

**Figure 22 Three face blobs in an EyeMap. We will check median value and standard deviation of each individual face blob to establish thresholding levels per face blob.**
We will call the binary image created by thresholding EyeMapL, the “\textit{thresholded\_eye\_map}”.

G. Eye Blob Rejection Scheme
Not all the eye blobs found from thresholding EyeMapL are really all eyes. Hence we need to reject those blobs that are most likely not eyes. To find the properties of all the eye blobs in \textit{thresholded\_eye\_map}, we need to apply blob coloring on \textit{thresholded\_eye\_map}.

Here is the blob rejection schemes we used. All of the number we chose come from the training images that we will be describing in the following section.

a. Throw away eye blobs that are smaller than 29 pixels
b. Throw away eye blobs that are bigger than 437 pixels
c. Throw away eye blobs that have height/width ratio \( \geq 2 \)
d. Throw away eye blobs that have width/height ratio \( > 3.25 \)
e. Throw away eye blobs that have density \( < 0.4 \) where density is calculated by forming a bounding box around the eye blob based on height and width of the potential eye and checking what percentage of the bounding box is the blob.
f. Throw away eye blobs that exist in the lowest one-third of the face blob
   \( \rightarrow \) these are mostly false detection of the mouth.

H. Forming Eye Pairs
We form eye pairs using another rejection scheme. We basically go through each potential eye pair that have both eyes in the same face blob, and form eye pairs only if the following conditions are met:

a. Distance between two eye blobs must be: \( \text{distance} \geq 30 \&\& \text{distance} \leq 62 \)
b. Horizontal distance \( \geq 10 \)
c. Vertical distance \( \leq 30 \)
d. \( \text{mass of the bigger eye blob/mass of the smaller eye blob} \leq 3 \)
   \( \rightarrow \) This is based on the assumption that two eye blobs have roughly the same mass.

I. Final Face Rejection: Rotate, Crop, and Density Check
Going through each eye pair individually we will rotate and crop both the original image and the \textit{face\_map}. The \textit{face\_map} is important because we will use it as a final check to reject false faces.
For image rotation it is important to note that the indexing of images occurs on the fourth quadrant of the Cartesian plane. As you move out column wise from the top left corner of an image, the X coordinates increase. As you move down row wise through an image the Y coordinate value increases. This relationship places you in the fourth quadrant. In order to perform a pure rotation on an image we must rotate the image about the origin. To center an image about the origin, we will translate it by half its width on the X axis and half its height on the Y axis. This translation will result in the scenario described in Figure 24. After we have centered the image we will use a simple rotation matrix to find the coordinates of the rotated image. The angle of rotation will be computed using the \( \arctan(\Delta Y/\Delta X) \) where \( \Delta Y \) and \( \Delta X \) denote the difference in X and Y coordinates between each eye in the eye pair. We will also check which eye is higher on the Y axis. If the right eye is higher on the Y axis we will perform a negative rotation to swing the left eye up relative to the right eye. Each translated XY coordinate will have associated RGB values and a rotated XY coordinate. When we have found all the rotated XY coordinates, we must find the largest and smallest coordinates. Since the image we are rotating is rectangular, any rotation will bring the corners out, so the old max and min height might be less than the new necessitating a larger matrix. Therefore we will have to allocate a new, larger matrix to contain the rotated image as seen in Figure 25. Since we are using a rotation matrix, the resulting indices of the rotated coordinates may not be integer values if we have rotated by an angle of anything other than a multiple of ninety degrees. To correct for this, we will also have to perform image interpolation. We will take average values around a radius of the target pixel from the original image. In our implementation we utilized C# built in image processing library as it could produce a higher quality interpolation scheme than we could. After we have rotated and interpolated the image, we will translate it back to the fourth quadrant.

As we perform these operations on the original image, we also must keep track of the eye pairs. This is important for the final square crop of the face. We will build the crop proportionally based on the distance between the eye pairs to match the dimensions of the PIE database. We will also perform these same operations on the skin mask to perform a final density check. This check can confirm that the eye pair is located on a single face and encompasses an entire face.

Consider Figure 26 where the found eye pair includes an eye on two different faces. By using the similarly rotated and cropped \textbf{face\_map} we can perform a density check on the \textbf{face\_map}. An actual face should be made entirely of skin so we can remove a low density \textbf{face\_map} from our valid eye pair list. Figure 27 shows a proper eye pair with a dense \textbf{face\_map}. Finally the cropped face will be passed onto face recognition.
Figure 27 Image of a correct eye pair and associated rotated face and skin crop.

**Face Recognition**

J. Face Recognition using Eigenfaces

An image of size N×N can be converted to a 1D vector of length $N^2$. Now we can think of the image as a point in a $N^2$-dimensional space. A collection of images with the same size, then, maps to a collection of points in this space. Images of faces, being similar in overall configuration, will not be randomly distributed in this huge space and therefore can be described by a lower dimensional subspace. Principal component analysis (PCA) is used to find vectors that best account for the distribution of face images within the entire image space. These vectors define the subspace of the face images, which is called the “face space.” Each vector of length $N^2$, describes a N×N image, and is a linear combination of the original face image. These vectors are, in fact the eigenvectors of the covariance matrix corresponding to the original face images. Because they are face-like, they are also called the “eigenfaces.”

If we are using M images $\Gamma_1, \Gamma_2, \ldots, \Gamma_M$ in the training set, then the average of the set is defined by:

$$\Psi = \frac{1}{M} \sum_{n=1}^{M} \Gamma_n$$

Equation 6 [3]

Then each image differs from the average by the vector

$$\Phi_i = \Gamma_i - \Psi$$

Equation 7 [3]

This set (the difference between each image and the average) of large vectors is then subject to principal component analysis, which seeks a set of M orthogonal vector, $u_k$, that best describes the distribution of the data. The kth vector, $u_k$, is chosen such that

$$\lambda_k = \frac{1}{M} \sum_{n=1}^{M} (u_k^T \Phi_n)^2$$

Equation 8 [3]

is maximum subject to

$$u_l^T u_k = \delta_{lk} = \begin{cases} 1 & \text{if } l = k \\ 0, \text{ otherwise} \end{cases}$$

Equation 9 [3]

The vectors $u_k$ and scalars $\lambda_k$ are the eigenvectors and eigenvalues, respectively of the covariance matrix.
where \( A = [\Phi_1 \Phi_2 \ldots \Phi_M] \) and each of the \( \Phi_i \) is a column vector.

The covariance matrix, however, will be of size \( N^2 \times N^2 \), which is not feasible for any reasonable sizes of an image.

It turns out we can just calculate the eigenvectors and the eigenvalues of the correlation matrix, which is only of size \( M \times M \) and the results will be the same as calculating from the covariance matrix.

The correlation matrix is \( L = A^\top A \) and it has \( M \) eigenvectors \( V_{ik} \) \( (k=1, \ldots, M) \). These vectors determine the linear combinations of the \( M \) training images to form the eigenfaces \( U_i \). If we decide to keep all \( M \) eigenvectors, then

\[
U_i = \sum_{k=1}^{M} V_{ik} \Phi_k , \quad i = 1, \ldots, M
\]

Each image in the training is now a point in \( M \)-dimensional space. What we do next is to classify whether a test image is a person already in the training set by projecting the test image onto the face space and calculating the minimum Euclidean distance between the test image point in the \( M \)-dimensional space with all \( M \) training image points.

\[
w_k = \mu_i^\top (\Gamma - \Psi) \quad \text{for } k = 1, \ldots, M'
\]

\[
D^\top = [w_1 \ w_2 \ \ldots \ w_M]
\]

\[
E_k = ||D_k - D_k||
\]

If the minimum distance \( E_k \) with a certain training image point is smaller than a rejection value \( R \), then the test image is classified as a “known face”. All training images and test images are in grey scale and have been normalized to have zero mean and unit variance.

The top three most principle components which translate to the three eigenvectors with highest eigenvalues are subject to illumination variation. Eliminating these three eigenvectors significantly improves the recognition results and removes some degree of illumination variation [4]. The results section will give more insight into the decision to get rid of the top three principle components and the number of eigenvectors used in our project.

Figure 28 shows the data flow for recognition in our project. All training is done on the PC side but projection, comparison, and classification of test images are done on the DSK.
4. Database & Results

Detection

For the detection part, we used 7 indoor images and 7 outdoor images as the training set. These images are primary used to fine tune the eye blob rejection scheme and also for us to see if the skin model has trouble detection skins indoor vs. outdoor. We chose images that have faces roughly of the size 60 x 60.

<table>
<thead>
<tr>
<th>Indoor Images</th>
<th>Faces Found</th>
<th>False Detections</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>68.jpg</td>
<td>2/2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>75.jpg</td>
<td>1/2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>82.jpg</td>
<td>2/2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>107.jpg</td>
<td>2/2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>174.jpg</td>
<td>2/2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>200.jpg</td>
<td>2/2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>204.jpg</td>
<td>3/3</td>
<td>1</td>
<td>1 eye-pair is OFF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outdoor Images</th>
<th>Faces Found</th>
<th>False Detections</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>35.jpg</td>
<td>2/2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>63.jpg</td>
<td>3/3</td>
<td>0</td>
<td>1 eye-pair is OFF</td>
</tr>
<tr>
<td>98.jpg</td>
<td>2/2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>147.jpg</td>
<td>3/3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>152.jpg</td>
<td>2/2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>168.jpg</td>
<td>2/3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>175.jpg</td>
<td>3/3</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Recognition

Database:

For recognition we used the PI database which consisted of 64 different people with 24 different illuminations created by a moving light source. Each of these images is 60 by 60 pixels and cropped tightly around face to avoid including hair and ears. Figure 29 shows the 24 different illuminations for one person from PIE database.

![Figure 29 24 Illuminations of same person in PIE database](image)

Training Set: For our recognition we trained 20 different people because this is average amount of people tagged in a Facebook album. For each person we used 5 different training images, the images boxed in red in Figure 29 show which images were selected for training. Most Facebook users have at least 100 tagged images so finding 5 training images is a conservative number that provides the necessary results. For each person we selected a dark and light image and three medium images with light from left, light from right and light from center. The total number of training images is 100 (20 people with 5 training images each). The goal of using 5 training images is to handle all the different positions of illumination. After this training was done there were 19 images per person in training set that were untouched as well as 44 other people in the PIE database to test rejection. In order to decide which eigenvectors to keep and determine accuracy, 5 images corresponding to dark, light, medium left, medium right, and medium center were tested. The final untouched testing set consists of 14 images per person in the training set and all 24 images for the 44 other people in PIE database not in training.

Results:

In our evaluation of PCA algorithm, and achieving the highest accuracy we first conducted tests to determine which eigenvectors to keep and identify what data was found in these eigenvectors. Belhumeur [4] mentions throwing out the top three principle components to deal with illumination variation but does not provide any reasoning in his paper as to why that is the case.

<table>
<thead>
<tr>
<th>Image Size</th>
<th>Number Eigenvectors</th>
<th>Eigenvectors Skipped</th>
<th>Training Im/Person</th>
<th># of people</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>7 out of 100</td>
<td>Skip Top Component</td>
<td>5</td>
<td>20</td>
<td>45.00%</td>
</tr>
<tr>
<td>60</td>
<td>7 out of 100</td>
<td>Skip Top2 Components</td>
<td>5</td>
<td>20</td>
<td>81.66%</td>
</tr>
</tbody>
</table>
Figure 30 Table of Results for recognition

<table>
<thead>
<tr>
<th>Skip top3 Components</th>
<th>85.00%</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 out of 100</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Skip top 4 components</th>
<th>81.66%</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 out of 100</td>
<td></td>
</tr>
</tbody>
</table>

Figure 30 shows that skipping the top 3 principle components results in the best accuracy. The figure also shows that the accuracy significantly jumps up after skipping the top 2 principle components. Figure 31 shows that there is no data in the top component and the 2nd principle component still contains no resemblance to a face. The 3rd highest component is first eigenface to resemble a face but still contains illumination variation. In addition when reconstruction of images was computed using the eigenfaces, if the top 3 components were used the reconstructed image still showed effects of illumination. However, if the top 3 components were removed, most of the illumination effects could no longer be seen.

In addition to getting rid of the top 3 components it is also necessary to determine which other eigenvectors to keep. The idea of PCA is to keep the minimum number of top components that will still accurately recognize and reconstruct an image. The following chart shows how the total number of eigenvectors used affect accuracy. As the number of eigenvectors used increases the accuracy increases but the relationship is decreasing so after 7 eigenvectors the accuracy growth is minimal. We decided to use 7 eigenvectors because it gives similar accuracy to using more eigenvectors and also allows for quicker computation and speed. The energy shown on Figure 33 is calculated by removing the top 3 principle components, renormalizing the picture and then calculating the percentage of energy issued of the remaining eigenvectors. When we use 7 eigenvectors we have 54.1% of the total energy.
Our results were based on using 60 by 60 pixels and 20 people in training set. Figure 34 shows that increasing the resolution to 100 by 100 pixels does not significantly improve accuracy. In addition increasing the number of training images per person slightly lowers accuracy as seen in figure below. Finally PCA is not shift invariant and Figure 35 shows that 1 pixel shifts still receive high accuracy rates but shifting by 3 pixels significantly lowers accuracy.

<table>
<thead>
<tr>
<th>Image Size</th>
<th>Number Eigenvectors</th>
<th>Skipped eigenvectors</th>
<th>Training Im/Persn</th>
<th># of people</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>60 x 60</td>
<td>26/200</td>
<td>4</td>
<td>5</td>
<td>40</td>
<td>77.50%</td>
</tr>
<tr>
<td>60 x 60</td>
<td>26/200</td>
<td>3</td>
<td>5</td>
<td>40</td>
<td>82.50%</td>
</tr>
</tbody>
</table>
The results from our paper match very closely with the results from Belhumeur’s paper. This paper also received its best accuracy results after removing top 3 principle components and had an error rate of 15.3%. The paper used Harvard and Yale databases which do not have as harsh light illumination as the PIE database. In addition 10 eigenvectors were used and 30 training images per person. [4] Figure 36 highlights the results of the paper and is very consistent with our recognition results.

<table>
<thead>
<tr>
<th>Shift Variations</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left 1 Pixels</td>
<td>80%</td>
</tr>
<tr>
<td>Left 2 Pixels</td>
<td>73.33%</td>
</tr>
<tr>
<td>Left 3 Pixels</td>
<td>58.33%</td>
</tr>
<tr>
<td>Right 1 Pixels</td>
<td>80%</td>
</tr>
<tr>
<td>Right 2 Pixels</td>
<td>66.67%</td>
</tr>
<tr>
<td>Right 3 Pixels</td>
<td>46.67%</td>
</tr>
<tr>
<td>Up 1 Pixels</td>
<td>66.67%</td>
</tr>
<tr>
<td>Up 2 Pixels</td>
<td>30%</td>
</tr>
<tr>
<td>Up 3 Pixels</td>
<td>15%</td>
</tr>
<tr>
<td>Up 1 Pixels</td>
<td>68.33%</td>
</tr>
<tr>
<td>Up 2 Pixels</td>
<td>38.33%</td>
</tr>
<tr>
<td>Up 3 Pixels</td>
<td>23.33%</td>
</tr>
</tbody>
</table>
5. Problems and Errors

Face Detection:

From the result, what we observed is that Hsu’s elliptical skin model has no problem detecting faces whether the setting is indoor or outdoor. Skin detection will fail if a face is underexposed.

For Figure 38, we failed to detect the face on the right because the face was not exposed to enough light. As we can see from the original skin map of that image, we lose parts of the face. And because of this the erosion step erases the whole face.

For Figure 40, we detected 4 false faces because a region of the background that is actually not skin got through Hsu’s elliptical skin model.
EyeMapL sometimes finds eye brows, mouth and nostrils.

For Figure 41, we detected 1 false face, because we treated an eye and a nostril as an eye-pair.

![Image](image1.png)

Figure 41 (a) Original image (b) the thresholded_eye_blob_map after eye blob rejection scheme

This algorithm also does not work for faces with glasses on.

![Image](image2.png)

Figure 42 (a) Original image (b) the EyeMap (c) the thresholded_eye_blob

As shown in Figure 42, EyeMapL has very high values around the frame.

Errors:

The majority of the recognition errors were caused by dark images which have no data. Since we were using average face spaces, dark images were recognized incorrectly as the darkest person in the training set. Dark images accounted for 70.8% of the errors. In addition to these errors other errors occurred on people whose facespaces overlapped other facespaces. This occurs because a person’s training space was not distinct and different test images were falsely identified as other people because they were closer to edge of another facespace. Figure 43 shows different test images of the same person being falsely identified as the wrong person. Overlapping facespaces is a known problem of PCA, and algorithms like Fisherface work at improving this issue. [4]
When using Euclidean distance, a test image is compared to all training sets to determine which set it is a minimum distance away from. However, if this distance is larger than a certain value then the test image shall be rejected as not being part of the training set. To compute our rejection threshold we computed our training sets and then took each individual training image and computed its Euclidean distance from its training set. Averaging these values together our rejection threshold was computed to be $1.8 \times 10^6$. Introducing rejection into analysis lowers accuracy by 4% because some images are identified correctly but fall outside of the rejection threshold. However, if the threshold is increased then images not part of the training set get identified as people in the training set. This tradeoff is a known problem of PCA and why techniques like Fisherface and LDA shall be used to better handle rejection and false classification errors. [4]

In the future to improve recognition results and the effects of illumination in dark images, the Fisherface algorithm should be used. A drawback of eigenface classification is that it takes into account differentiation between people but also within the images of the same person. Algorithms like Fisherface work better for classification because it maximizes the ratio of the determinant between-class and within-class of the projected samples. [4] Belhumeur also notes that Fisherface performs with better accuracy and computation time and in the future our recognition algorithm should be moved to use Fisherface.

6. Memory & Speed

Face Detection

The operation that gives us the most trouble in terms of speed was trying to remove any artifacts smaller than a disk of radius 25 from imfilled original_skin_map. Our original idea was to erode with a disk of radius 25 and then dilate with a disk of radius of 25. This was too slow on the DSK because the structuring element is 51x51. Each output pixel requires 2601 reads on the input image and 2601 reads on the kernel. And there is a total of 223,262 (403 x 554) output pixels to generate.

Then we decided to use a much smaller structuring element of radius 5 and do the operation 5 times. This made the kernel smaller (11x11). We also tried paging in 42 rows of the input image to internal memory produce 32 rows of output each time. This helped, but was still very slow. One erosion operation took 822,911,205 cycles = 3.66 sec when –o3 optimization level was used. The problem turned out to be that we completely ignored the fact that there are only 32 A/B registers on the DSK, so Code Composer was unable to find the pipeline schedule.

After the demo, Prof. Casasent’s advice was to use a smaller kernel. So we ended up using a 3x3 kernel (disk of radius 1) and turn one operation into 25 operations instead. We profiled one operation and it took only 1,345,172 cycles = 5.98 msec.
Since the erosion step needs to be done 25 times and the dilation step needs to be done 25 times, the whole binary morphological opening should take $5.98 \text{sec} \times 25 = 149.5 \text{msec}$.

We also profiled how long 25 operations took. 25 operations took a total of $31,736,697 \text{cycles} = 141 \text{msec}$.

![3x3 kernel with structuring element of a disk with radius 1.](image)

Here is the analysis on the number of cycles for one erosion operation using a 3x3 kernel (disk of radius 1):

For each iteration of the outer two for loops, this is what we are doing:

a. We are copying 42 rows from input_img (external memory) and write 42 rows onto wkspace (internal memory) each time for 10 times using DMA.

b. When each chunk in step a. arrived in wkspace, we proceed with the following:

i. We are accessing 5 elements in wkspace (internal memory) -> a total of 5 reads on internal memory per output pixel. (but four of the reads can be reused next time)

ii. For each output pixel to be generated, a total of 5 AND operations on internal memory need to be made.

iii. After the AND operations are done, the result is stored onto outspace (internal memory). -> one memory write onto internal memory per output pixel.

c. Once 40 rows of output pixels have been generated in outspace (internal memory), we are copying 40 rows from outspace (internal memory) and writing the 40 rows from outspace (internal memory) and writing the 40 rows onto output_img (external memory) each time for 10 times using DMA.

Cycles for DMA transfer in:

10 DMA transfers of 42 rows of input pixels:

$$10 \times (42 \times 604)/4 \times (2.25 \text{cycles/word}) = 142,695 \text{ cycles}$$

Cycles for DMA transfer out:

10 DMA transfers of 40 rows of output pixels:

$$10 \times (40 \times 604)/4 \times (2.25 \text{cycles/word}) = 135,900 \text{ cycles}$$

For each output pixel we are accessing internal memory 5 times.

Hence the total number of cycles taken for one erosion operation is:

$$5 \times (1.5 \text{cycles}) \times (554 \times 403 \text{pixels}) + 142,695 + 135,900 = 1,953,060 \text{ cycles} = 8.68 \text{msec}$$

We did not use the TI image library to perform the 3x3 erosion or dilation because the limit of TI’s functions was that the number of columns in the image needs to be a multiple of 8. Our image has 604 columns, which is not a multiple of 8.

Below is the summary of the memory usage on the DSK.

<table>
<thead>
<tr>
<th>What’s stored</th>
<th>Size (#of bytes)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>input_r-channel</td>
<td>273612</td>
<td>Each pixel value is stored as an unsigned char (1 byte)</td>
</tr>
<tr>
<td>input g-channel</td>
<td>273612</td>
<td>Each pixel value is stored as an unsigned char (1 byte)</td>
</tr>
</tbody>
</table>
For the detection program, code size on the DSK was **26,624 bytes**.

**Recognition:**

For Recognition on the DSK we need to calculate the Projected Test Image and then the Minimum Euclidean distance from each of the Projected Test Images. Since our pictures are each 60 by 60 pixels we need 3600 bytes in internal memory. To calculate the average image we need 3600 floats which require 14400 bytes which we also place in internal memory. Computing the facespace requires 7 eigenfaces of 3600 floats each which equals 100,800 bytes. This cannot fit in internal memory but is not big enough to page in smaller chunks. The set of 20 projected training images with 7 dimensions requires 140 floats or 560 bytes which can be stored in internal memory. Computing the facespace requires 7 floats which takes 28 bytes in internal memory. Due to the lack of size and memory constraint paging does not help optimization. However unrolling matrix multiplications and Euclidean distance calculations help optimize the recognition algorithm. Overall the recognition parts are all significantly faster than detection. The following figure provides profiling statistics for all of the recognition functions.

<table>
<thead>
<tr>
<th>Recognition Profiling Times</th>
<th>Pre Optimization</th>
<th>Internal Memory</th>
<th>Loop Unrolling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculate Mean Centered Image</td>
<td>106675= 0.469 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Projected Test Image</td>
<td>2452914=10.793 ms</td>
<td>2445660= 10.761 ms</td>
<td></td>
</tr>
<tr>
<td>Euclidean Distance</td>
<td>17025= 0.0749 ms</td>
<td>16627= 0.0732 ms</td>
<td>12162= 0.0535 ms</td>
</tr>
</tbody>
</table>

**Figure 45**

7. **Demo**

**Detection**

For the detection demo, we will use 30 images, 15 indoor and 15 outdoor of the standard Facebook size (604x453). Using our GUI we will be able to display all the intermediary steps utilized in our face detection process. We will be able to show:

- `original_skin_map`
- `imfilled original_skin_map`
- `imfilled original_skin_map` after erosion
- `face_map`
- `EyeMapL`
- `thresholded_eye_blobs` obtained from EyeMapL
- `thresholded_eye_blobs` after eye blob rejection scheme
Final detected faces and eye locations along with cropped and rotated 60x60 results

**Recognition**
For our demo we have created a set of 20 trained people known as friends. The demo allows you to test recognition on these people with untouched test images for each person. In addition the GUI will allow you to recognize non-friends which should be rejected as a non-friend. Once the TA’s have played this part they can add any non-friend to be part of the trained set. After training the new non-friend, the non-friend should now be recognized correctly and not rejected as a non-friend. The demo will display the name of the person recognized or a message saying that the image is not a friend. The GUI displays all of the images so TA’s can pick users that look similar to each other.

**Final Demo**
Detected and recognize faces in image downloaded from Facebook.
## 8. Final Work Schedule

<table>
<thead>
<tr>
<th>Week</th>
<th>Maxwell</th>
<th>Ping-Hsiu</th>
<th>Saurabh</th>
</tr>
</thead>
<tbody>
<tr>
<td>8/24</td>
<td>Research on Face Detection algorithms</td>
<td>Research on Skin Model</td>
<td>Research on Face Recognition algorithms</td>
</tr>
<tr>
<td>8/31</td>
<td>Implement Hsu’s Elliptical Skin Model in Matlab.</td>
<td>Help Max with implementation of Hsu’s Elliptical Skin Model in Matlab</td>
<td>Research on PCA, obtain PIE database.</td>
</tr>
<tr>
<td>9/7</td>
<td>Research on morphological processing, blob coloring, imfill. Try them on the detected skin in Matlab</td>
<td>Implement YCbCr to RGB in C, implement YCbCr nonlinear transform, implement Hsu’s elliptical skin model in C</td>
<td>Research on Fisherface</td>
</tr>
<tr>
<td>9/14</td>
<td>Implement EyeMapC and EyeMapL in Matlab.</td>
<td>Help Max with morphological processing and eye map constructions in Matlab</td>
<td>Implement PCA training code in Matlab</td>
</tr>
<tr>
<td>9/21</td>
<td>Implement Mouth Map in Matlab.</td>
<td>Implement morphological processing in C</td>
<td>Implement PCA recognition code in Matlab</td>
</tr>
<tr>
<td>9/28</td>
<td>Figuring out how to threshold the values from the EyeMap to obtain the eye blobs.</td>
<td>Implement EyeMapL in C</td>
<td>Evaluate the PCA recognition result from the PIE images with different resolutions.</td>
</tr>
<tr>
<td>10/5</td>
<td>Help Peter with implementation of blob coloring in C</td>
<td>Implement blob coloring in C</td>
<td>Analyze the recognition result with different number of eigenvectors kept in PCA</td>
</tr>
<tr>
<td>10/12</td>
<td>Start devising the eye blob rejection scheme</td>
<td>Implement imfill in C</td>
<td>Analyze the recognition result with top 3 eigenvectors kept and thrown away.</td>
</tr>
<tr>
<td>10/19</td>
<td>Help Peter with eye blob thresholding in C</td>
<td>Implement eye blob thresholding (calculate median, std deviation) in C</td>
<td>Implement PCA recognition code in C</td>
</tr>
<tr>
<td>10/26</td>
<td>Try the eye detection result with and without EyeMapC</td>
<td>Fixing Bug in the blob coloring C code…</td>
<td>Put PCA recognition C code on the DSK</td>
</tr>
<tr>
<td>11/2</td>
<td>Implement image rotation and image cropping in C#</td>
<td>Help Saurabh with implementation of PCA recognition in C</td>
<td>Optimize the PCA recognition C code on the DSK</td>
</tr>
<tr>
<td>11/16</td>
<td>Research on how to use OpenCV read in RGB channels of images and work on PC side code to send pixels to the DSK side.</td>
<td>Use the training images for detection part to find the most optimized blob rejection scheme for thresholded eye blobs, combine all previous C code and put it on the DSK</td>
<td>Implement the Recognition part of GUI</td>
</tr>
<tr>
<td>11/23</td>
<td>Implement the Detection part of GUI</td>
<td>Work with Max to write GUI for detection part, Work with Saurabh to write GUI for recognition part</td>
<td>Add more friends in the training set for demo</td>
</tr>
</tbody>
</table>
9. References


