Abstract

Face recognition presents a challenging problem in the field of image analysis and computer vision, and as such has received a great deal of attention over the last few years because of its many applications in various domains. Face recognition techniques can be broadly divided into three categories based on the face data acquisition methodology: methods that operate on intensity images; those that deal with video sequences; and those that require other sensory data such as 3D information or infra-red imagery. In this paper, an overview of some of the well-known methods in each of these categories is provided and some of the benefits and drawbacks of the schemes mentioned therein are examined. Furthermore, a discussion outlining the incentive for using face recognition, the applications of this technology, and some of the difficulties plaguing current systems with regard to this task has also been provided. This paper also mentions some of the most recent algorithms developed for this purpose and attempts to give an idea of the state of the art of face recognition technology.
تصميم وتنفيذ نظام التعرف على الوجه
بناء على واجهة برمجة تطبيقات الحواسيب

المدرس المساعد سيف سعد احمد
كلية التراث الجامعة
قسم علوم الحاسبات

المستخلص

فكرة البحث مبنية على أساس التعرف على معالم الوجه باعتبارها مشكلة صعبة في مجال تحليل الصور وتطبيقات الكمبيوتر، وعلى هذا النحو قد تلقى قدرا كبيرا من الاهتمام على مدى السنوات القليلة الماضية بسبب العديد من التطبيقات في مختلف المجالات. ويمكن تقسيم تقنيات التعرف على الوجه بشكل عام إلى ثلاث فئات بناءً على منهجية جمع البيانات (أساليب التي تعمل على كثافة الصور، وأساليب تتعامل مع تسلسل الفيديو، وتلك التي تتطلب بيانات الحسية الأخرى مثل المعلومات 3D أو الصور بالأشعة تحت الحمراء). في هذا البحث سيتم توفير لمحة عامة عن بعض الأساليب المعروفة في كل من هذه الفئات و يتم فحص بعض من مزايا وعيوب المخططات المذكورة فيه. علاوة على ذلك، تناول البحث أيضا مناقشة الخطوط العريضة المحفزة لاستخدام التعرف على الوجه، وتطبيقات هذه التكنولوجيا، و بعض الصعوبات التي تعاني منها النظام الحالية فيما يتعلق بهذه المهمة. وتطرق البحث أيضا إلى بعض أحدث الخوارزميات التي وضعت لهذا الغرض ومحاولات اعطاء فكرة عن حالة من اللفن من حالات تقنية التعرف على الوجه.

1.1 Introduction

Humans have been using physical characteristics such as face, voice, etc. to recognize each other for thousands of years. With new advances in technology, biometrics has become an emerging technology for recognizing individuals using their biological traits. Now, biometrics is becoming part of day to day life, where in a person is recognized by his/her personal biological characteristics. Our goal is to develop an inexpensive security surveillance system, which will be able to detect and identify facial and body characteristics in adverse weather conditions. There are many factors which influence this type of methods i.e.
lighting condition background noise, fog and rain. A particular attention is given to face recognition. Face recognition refers to an automated or semi-automated process of matching facial images. Many techniques are available to apply face recognition one of them is Principle Component Analysis (PCA). PCA is a way of identifying patterns in data and expressing the data in such a way to highlight their similarities and differences.

Humans often use faces to recognize individuals and advancements in computing capability over the past few decades now enabled similar recognitions automatically. Early face recognition algorithms used simple geometric models, but the recognition process has now matured into a science of sophisticated mathematical representation and matching processes. Major advancements and initiative in the past ten to fifteen years have propelled face recognition technology into the spotlight. Face recognition can be used for both verification and identification (open –set and closed-set).

1.2 Problem Statement

- Almost all the face databases have frontal faces. Hence, for matching we require the frontal face image or one should have to generate the frontal face features from the non-frontal face image.
- Linear Object Class assumptions ensures that there exists a linear transformation which relates feature vector of two different posed images.[1]
- In this work we have estimated this transformation and generate frontal face features from the features of posed image.

The face recognition problem can be formulated as follows: Given an input face image and a database of face images of known individuals, how can we verify or determine the identity of the person in the input image.
1.3 Aims and Objectives

The goal of this project is to create a prototype to reduce the amount of climacteric factors to provide facial recognition and facial tracking in daylight surveillance and night vision surveillance for wide areas. Also, it implements pan and tilt support to give the ability to rotate the cameras by software control. The aim of face recognition is to identify or verify one or more persons from still images or video images of a scene using a stored database of faces.

This project is a step towards developing a face recognition system which can recognize static images. It can be modified to work with dynamic images. In that case the dynamic images received from the camera can first be converted into the static one's and then the same procedure can be applied on them. But then there are lots of other things that should be considered. Like distance between the camera and the person, magnification factor, view [top, side, front] etc.

Objectives:

1) To recognize a sample face from a set of given faces.

2) Use of Principal Component Analysis [Using Eigen face approach].

3) Use a simple approach for recognition and compare it with Eigen face approach.

4) Suggest which one is better and why. It may happen that in some cases latter may work better than former approach and vice versa.

2.1 Introduction

This chapter explains how a face can be tracked. As with earlier examples, I'll grab frames from the webcam, and draw them rapidly onto a panel. At the same time, a detector analyzes the frames to find a face and highlight it in the panel. The application, called Face Tracker is shown in Figure 1.
The tracker draws a yellow rectangle around the face, and red crosshairs centered inside the rectangle.

The detection code is fast when there's a face present in the image (around 40ms), but may take substantially longer to decide there's no face (as much as 200ms). Two important aspects of the coding are finding ways to speed up the detection, and making sure that lengthy detection processing don't slow down the rest of the program (in particular, the rendering of successive images onto the panel).

The next chapter will extend the processing to recognize the tracked face. The distinction between face detection and recognition is that recognition returns a name for the face.

Detection is carried out by a Haar classifier, pre-trained to find facial features (when viewed front-on). The classifier's training requires a great deal of time, but thankfully I can skip that stage because I'm using a face classifier that's already part of OpenCV.

2.2 Face Detection

The FaceDetection.java example described in this section reads an image from a file, locates all the faces in the picture. It draws yellow rectangles around them,
then writes the modified image out to a new JPEG file. Figure 2 shows an
eexample image before and after the faces have been identified.

![Figure 2. Finding Faces in an Image.](image)

The face detection uses the same Haar classifier that I'll be employing later on in
my tracker application, and Figure 2 highlights some of the strengths and
weaknesses of the approach. Multiple faces can be found easily and quickly, but
only if a face is almost level and almost completely visible. For instance, the
classifier failed to label the men at the left and right edges of the image because
too much of their faces are missing or obscured. Also, a Haar classifier can often
return false positives – highlighted areas which are not faces. This can be seen in
the right hand image of Figure 2, where a crinkled shirt elbow is misidentified.

The code for FaceDetection.java:

```java

public class FaceDetection {
```
private static final int SCALE = 2;

// scaling factor to reduce size of input image

// cascade definition for face detection private static final String CASCADE_FILE = "haarcascade_frontalface_alt.xml";

private static final String OUT_FILE = "markedFaces.jpg";

public static void main(String[] args){
    if (args.length != 1) {  System.out.println("Usage: run FaceDetection <input-file>"); return;}

    // preload the opencv_objdetect module to work around a known bug

    Loader.load(opencv_objdetect.class);

    // load an image

    System.out.println("Loading image from " + args[0]); IplImage origImg = cvLoadImage(args[0]);

    // convert to grayscale

    IplImage grayImg =cvCreateImage(cvGetSize(origImg), IPL_DEPTH_8U, 1);
    cvCvtColor(origImg, grayImg, CV_BGR2GRAY);

    // scale the grayscale (to speed up face detection) IplImage smallImg =
    IplImage.create(grayImg.width()/SCALE, grayImg.height()/SCALE, IPL_DEPTH_8U, 1);
    cvResize(grayImg, smallImg, CV_INTER_LINEAR);

    // equalize the small grayscale cvEqualizeHist(smallImg, smallImg);

    // create temp storage, used during object detection
CvMemStorage storage = CvMemStorage.create();

// instantiate a classifier cascade for face detection
CvHaarClassifierCascade cascade = new CvHaarClassifierCascade(cvLoad(CASCADE_FILE));
System.out.println("Detecting faces...");

CvSeq faces = cvHaarDetectObjects(smallImg, cascade, storage,
1.1, 3, CV_HAAR_DO_CANNY_PRUNING);

cvClearMemStorage(storage);

// draw thick yellow rectangles around all the faces
int total = faces.total();
System.out.println("Found " + total + " face(s)" );
for (int i = 0; i < total; i++) {
CvRect r = new CvRect(cvGetSeqElem(faces, i));
cvRectangle(origImg, cvPoint(r.x()*SCALE, r.y()*SCALE ),cvPoint( (r.x() + r.width())*SCALE, (r.y() + r.height())*SCALE ),CvScalar.YELLOW, 6, CV_AA, 0);
// undo image scaling when calculating rect coordinates
}

if (total > 0) {
System.out.println("Saving marked-faces version of " + args[0] + " in " + OUT_FILE); cvSaveImage(OUT_FILE, origImg);
}

} // end of main()

// end of FaceDetection class

The image preprocessing consists of three steps: conversion of the color input image to gray scale (necessary for the subsequent equalization and Haar detection), scaling to reduce the size of the image (and thereby reduce the detection time), and gray scale equalization. Equalization examines the image's range of gray scale values and widens them to cover more of the total range.
from black to white. The result is an image with larger contrasts between similarly shaded areas, which makes object detection easier later on.

2.3 The Face Tracker

My tracker application (see Figure 1) captures webcam snaps with JavaCV's FrameGrabber and then performs face detection using code similar to the previous section. The class diagrams for the application are given in Figure 4.

![Class Diagrams for the FaceTracker Application.](image)

I won't bother explaining the top-level FaceTracker class – it's a standard JFrame which creates the FacePanel object, and a button. Pressing the button, labeled as "Save Face", makes FacePanel save the currently highlighted face (i.e. the sub image inside the yellow rectangle) to a file. The Face Panel class spends much of it's time inside a threaded loop which repeatedly grabs an image from the webcam and draws it onto the panel until the window is closed. FacePanel differs from similar panel classes in earlier examples in one important way. Since face detection is such a time consuming process, it is farmed out to a separate thread that uses a mixture of Java 2D and JavaCV. The rest of this chapter will describe these aspects in more detail.

2.4 Initializing the Detector
When cvHaarDetectObjects() is eventually called, it has two prerequisites that I can deal with at start-up time: I load the classifier's XML file, and create dynamic storage which will be allocated as the function progresses. This occurs in the initDetector() method, called from FacePanel's constructor:

// globals

// classifier for face detection

private static final String FACE_CASCADE_FNM = "haarcascade_frontalface_alt.xml";

// "haarcascade_frontalface_alt2.xml"

private CvHaarClassifierCascade classifier;

private CvMemStorage storage;

private CanvasFrame debugCanvas;

private void initDetector() {

    // instantiate a classifier cascade for face detection
    classifier = new CvHaarClassifierCascade(cvLoad(FACE_CASCADE_FNM));
    if (classifier.isNull()) { System.out.println("Could not load: " + FACE_CASCADE_FNM); System.exit(1); }

    storage = CvMemStorage.create();

    // create storage used during object detection
    // debugCanvas = new CanvasFrame("Debugging Canvas");

} // end of initDetector()

The code for the creation of a debugCanvas object is commented out. It was used during debugging to show the intermediate stages in an image's
transformation. CanvasFrame is a useful way of quickly displaying an image without creating additional GUI elements in the Swing application.

2.5 The Display Loop

The FacePanel() constructor invokes a thread which starts the webcam display loop inside run(). The method is similar to what we've seen before, except when it passes the snapped image to trackFace() for processing (shown in bold in the following):

    // globals

    private static final int DELAY = 100;
    // time (ms) between redraws of the panel

    private static final int CAMERA_ID = 0;

    private static final int DETECT_DELAY = 500;
    // time (ms) between each face detection

    private static final int MAX_TASKS = 4;
    // max no. of tasks that can be waiting to be executed

    private IplImage snapIm = null; private volatile boolean isRunning;
    // used for the average ms snap time information

    private int imageCount = 0; private long totalTime = 0;
    // used for thread that executes the face detection

    private AtomicInteger numTasks;
    // used to record number of detection tasks

    private long detectStartTime = 0;

    public void run() { FrameGrabber grabber = initGrabber(CAMERA_ID); if (grabber == null) return;long duration;
isRunning = true; while (isRunning) {
    long startTime = System.currentTimeMillis();
    snapIm = picGrab(grabber, CAMERA_ID);

    if (((System.currentTimeMillis() - detectStartTime) > DETECT_DELAY) &&
        (numTasks.get() < MAX_TASKS)) trackFace(snapIm); imageCount++;

    repaint(); duration = System.currentTimeMillis() - startTime;
    totalTime += duration; if (duration < DELAY) {
        try {
            Thread.sleep(DELAY - duration);
        } catch (Exception ex) {}
    }
}

Face detection, even after various speed optimizations, can still take a 200ms to fail to find anything. Such a lengthy delay would severely affect run()'s display loop, which is meant to draw a new image onto the panel roughly every DELAY (100) ms. I get around that problem by utilizing a separate thread to execute the work inside trackFace() (see below for details), allowing the display loop to progress without delay. Altogether, FaceTracker utilizes three threads – the GUI event thread, a thread containing the display loop in run(), and a face detection thread inside trackFace().

Due to the time-consuming nature of trackFace()'s work, I limit its call frequency to once every DETECT_DELAY (500) ms. I'll explain the other part of the if-test around the trackFace() call – the test of the numTasks atomic integer – shortly.

trackFace()'s threaded nature means that its call in run() will return almost immediately, before the detection task has been completed. As a consequence, the duration calculated inside run() doesn't include detection time.

2.6 Tracking a Face in a Thread

The simplest way of implementing a threaded detection task is to fire off a new thread each time an image needs to be analyzed. This is almost certainly not a
good idea since we don't know whether the underlying OpenCV library (i.e. the C code inside OpenCV's DLLs) is capable of dealing with multiple detection tasks being carried out at the same time.

It's hard to test OpenCV's robustness in the face of concurrency, since any problems depend on how multiple calls overlap in their use of global data structures, DLLs, and the underlying OS. It's better to avoid the problem altogether by enforcing a

restriction that only one detection task can execute inside the detection thread at a time; pending tasks will have to queue up to wait their turn.

Since Java 5, it's been easy to create threads with this kind of behavior, by using an ExecutorService object to manage a single threaded executor:

// global

private ExecutorService executor;

// in the FacePanel() constructor

executor = Executors.newSingleThreadExecutor();

The factory method, Executors.newSingleThreadExecutor(), creates an executor consisting of a single worker thread taking tasks off an unbounded queue one at a time. Tasks are guaranteed to execute sequentially, and no more than one task will be active at any given time.

One way of improving this execution scenario is to limit the length of the task queue, since we don't want an unbounded number of detection tasks waiting to be processed. The queue length can be limited by using an atomic integer as a counter to record the number of tasks currently on the queue:

// globals
private static final int MAX_TASKS = 4;

// max no. of tasks that can be waiting to be executed private AtomicInteger numTasks;

// used to record number of detection tasks

// in the FacePanel constructor()

numTasks = new AtomicInteger(0);

numTasks is atomic since both the webcam display and detection threads are able to modify it. I don't want problems to arise if they try to manipulate the integer at the same time.

The second half of the if-test around the call to trackFace() implements the bounded queue requirement:

if (((System.currentTimeMillis() - detectStartTime) > ETECT_DELAY) && (numTasks.get() < MAX_TASKS)) trackFace(im);

trackFace() is only called if there are less than MAX_TASKS (4) tasks associated with the executor – one running and three waiting.

2.7 Detecting a Face

The face detection code inside trackFace() is in a run() method. It's invocation is added to the executor's queue as a pending task when trackFace() is called:

// globals

private IplImage grayIm; private volatile boolean saveFace = false;

// set by the "Save Face" button

private void trackFace(final IplImage img){

grayIm = scaleGray(img); numTasks.getAndIncrement();
// increment no. of tasks before entering queue

executor.execute(new Runnable() {
    public void run() {
        detectStartTime = System.currentTimeMillis(); CvRect rect =
        findFace(grayIm); if (rect != null) {
            setRectangle(rect); if (saveFace) {
                clipSaveFace(img); saveFace = false; }
        }
    }
});
long detectDuration = System.currentTimeMillis() - detectStartTime;
System.out.println("detect time: "+detectDuration + "ms");
umTasks.getAndDecrement();

// decrement no. of tasks since finished

The hard work of face detection by the Haar classifier is hidden away in
findFace(), which returns a single JavaCV rectangle object. This information is
stored by setRectangle() for later rendering onto the panel, and the clipped face
is saved if the "Save Face" button has been pressed.

The task counter, numTasks, is incremented outside the run() method since I
want to record the number of tasks queuing as well as the one currently
executing. However the counter is decremented at the end of the task (the last
line of run()).

The Haar classifier requires a grayscale image, which is generated by
scaleGray() before the thread starts. scaleGray() also reduces the image's size, to
speed up the processing, and equalizes it. The code is very similar to that
performed in the earlier Face Detector example.

The time taken by the classifier is printed to standard output. On my slow test
machine, finding a face usually took 20-50ms while failing to find one could take
between 70-200ms. These times indicate that I could increase the detection
activation frequency which is currently set at once every DETECT_DELAY
(500) ms.
2.8 Finding a Face

The findFace() method calls the Haar classifier, and extracts a single rectangle from the result.

```java
private CvRect findFace(IplImage grayIm) {

    // Haar classification
    CvSeq faces = cvHaarDetectObjects(grayIm, classifier, storage,
          1.1, 1, CV_HAAR_DO_ROUGH_SEARCH |
          CV_HAAR_FIND_BIGGEST_OBJECT);

    /* speed things up by searching for only a single, 
       largest face subimage */ int total = faces.total(); if (total == 0) {
       System.out.println("No faces found"); return null;}

    else if (total > 1) System.out.println("Multiple faces detected (" + total 
       + "); using the first"); else System.out.println("Face detected");

    CvRect rect = new CvRect(cvGetSeqElem(faces, 0)); //get rectangle
    cvClearMemStorage(storage);

    return rect; // end of findFace()

    The arguments of the cvHaarDetectObjects() call are a little different from those 
    in my earlier FaceDetection.java example. The fifth argument sets the number 
    of overlapping detections needed before a region is deemed to contain an object 
    to only 1 (it was 3 in Face Detection). Reducing this value increases processing 
    speed, but increases the chance of negative hits.

    The final argument is an OR'ed combination of
    CV_HAAR_DO_ROUGH_SEARCH and
CV_HAAR_FIND_BIGGEST_OBJECT which signals that only the largest object need be returned, and that a faster search is preferred. Canny pruning isn't included since it interacts unfavorably with the rough search setting.

During debugging, it was useful to display the JavaCV image utilized in findFace(). I added the following lines at the start of the method:

```java
// show the grayscale debugCanvas.showImage(grayIm);
debugCanvas.waitKey(0);
```

2.9 Saving a Rectangle

setRectangle() extracts the face rectangle's coordinates ((x, y), width, height) from the JavaCV data structure and stores them in a Java Rectangle object. In the process, the data is enlarged, so it has the same scale as the original snapped image.

```java
// globals
private static final int IM_SCALE = 4; private static final int SMALL_MOVE = 5;

private Rectangle faceRect; // holds coords of highlighted face
private void setRectangle(CvRect r) { synchronized(faceRect) { int xNew =
  r.x()*IM_SCALE; int yNew = r.y()*IM_SCALE; int widthNew =
  r.width()*IM_SCALE; int heightNew = r.height()*IM_SCALE;
  // calculate movement of new rectangle compared to previous one int xMove =
  (xNew + widthNew/2) - (faceRect.x + faceRect.width/2); int yMove = (yNew +
  heightNew/2) - (faceRect.y + faceRect.height/2);
  // report movement only if it is 'significant' if ((Math.abs(xMove)>
  SMALL_MOVE) || (Math.abs(yMove) > SMALL_MOVE))
  System.out.println("Movement (x,y): (" +xMove + "," + yMove + ")");
```
faceRect.setRect( xNew, yNew, widthNew, heightNew);

}} // end of setRectangle()

Perhaps the most mysterious aspect of setRectangle() is its use of a synchronized block. It's present because there's a possibility that faceRect can be used in two threads at the same time. The face detection thread calls setRectangle() and clipSaveFace() which both access faceRect, while the Java GUI thread needs it for drawing.

The movement of the current face rectangle compared to the last one is calculated in setRectangle(), but not used elsewhere in the application. I included the code since such information would be useful in more complex face tracking applications.

2.10 Rendering the Highlighted Face

Figure 4 shows a typical rendering of the highlighted face in Face Tracker.

![Figure4. A Highlighted Face.](image)

The panel contains four elements: the webcam image in the background, a yellow rectangle, a red crosshairs image, and statistics written at the bottom left corner.

All rendering is done through calls to the panel's paintComponent():
private IplImage snapIm = null; // current webcam snap

public void paintComponent(Graphics g) { super.paintComponent(g);
 Graphics2D g2 = (Graphics2D) g; g2.setFont(msgFont);

 // draw the image, stats, and detection rectangle if (snapIm != null) {
 g2.setColor(Color.YELLOW); g2.drawImage(snapIm.getBufferedImage(), 0, 0, this); String statsMsg = String.format("Snap Avg. Time: %.1f ms", 
 ((double) totalTime / imageCount)); g2.drawString(statsMsg, 5, HEIGHT - 10);
 // write statistics in bottom-left corner drawRect(g2); } else { // no image yet
 g2.setColor(Color.BLUE); g2.drawString("Loading from camera " + 
 CAMERA_ID + "...", 5, HEIGHT-10); } // end of paintComponent()

drawRect() is in charge of drawing the yellow rectangle and the crosshairs:

// global private Rectangle faceRect; // coords of the highlighted face

private void drawRect(Graphics2D g2) { synchronized(faceRect) { if
 (faceRect.width == 0) return; // draw a thick yellow rectangle
 g2.setColor(Color.YELLOW); g2.setStroke(new BasicStroke(6));
 g2.drawRect(faceRect.x, faceRect.y, faceRect.width, faceRect.height); int
 xCenter = faceRect.x + faceRect.width/2; int yCenter = faceRect.y +
 faceRect.height/2; drawCrosshairs(g2, xCenter, yCenter);
 }

 drawRect() uses a synchronized block for the same reason as setRectangle() earlier – I don't want its access to the rectangle information to be affected by other threads.

Standard Java2D code is used to draw the rectangle, replacing my earlier use of JavaCV's cvRectangle() in FaceDetection.java.
drawCrosshairs() draws a pre-loaded PNG image (see Figure 5) so it's centered at the given coordinates.

Figure 5. The Crosshairs Image.

Chapter Three

3.1. GUI Face Recognizer Applications

In the last chapter I developed software that could detect and track a face as it moved in front of a webcam. The next step, and the topic of this chapter, is to attach a name to the face, to recognize it using a technique called *Eigen faces*.

The outcome is the GUI application shown in Figure 1.

Figure 1 A GUI Face Recognizer Application.

When the user presses the "Recognize Face" button (at the bottom of the window in Figure 1), the currently selected face is compared against a set of
images that the recognizer code has been trained against. The name associated with closest matching training image is reported in a text field (and in yellow text in the image pane), along with a distance measure represented the 'closeness' of the match.

The recognition process relies on a preceding training phase involving multiple facial images, with associated names. Typical training images are shown in Figure 2.

![Training Images](image.png)

**Figure 2 Training Images.**

It's important that the training images all be cropped and orientated in a similar way, so that the variations between the images are caused by facial differences rather than differences in the background or facial position. There should be uniformity in the images' size, resolution, and brightness. It's useful to include several pictures of the same face showing different expressions, such as smiling and frowning. The name of each person is encoded in each image's filename. For instance, I'm represented by three
image files, called "andrew1.png", "andrew2.png", and "andrew4.png".

The training process creates Eigen faces which are composites of the training images which highlight elements that distinguish between faces. Typical Eigen faces generated by the training session are shown in Figure 3.

![Eigen faces](image)

**Figure 3** Some Eigen faces.

Due to their strange appearance, Eigen faces are sometimes called ghost faces. Each training image can be represented by a weighted sequence of Eigen faces, as depicted in Figure 4.

![Weighted sequence](image)
Figure 4. A Training Image as a Weights Sequence of Eigen faces. The idea is that a training image can be decomposed into the weighted sum of multiple Eigen faces, and all the weights stored as a sequence.

Not all Eigen faces are equally important – some will contain more important facial elements for distinguishing between images. This means that it is not usually necessary to use all the generated Eigen faces to recognize a face. This allows an image to be represented by a smaller weights sequence (e.g. using only three weights instead of six in Figure 4). The downside is that less weights means that less important facial elements will not be considered during the recognition process.

Another way of understanding the relationship between Eigen faces and images is that each image is positioned in a multi-dimensional Eigen space; whose axes are the Eigen faces (see Figure 5)

![Figure 5 an Image in Eigen space.](image-url)
The weights can now be viewed as the image's coordinates in the Eigen space. Due to my poor drawing skill, I've only shown three axes in Figure 5, but if there are six weights, and then there should be six orthogonal axes (one for each Eigen face).

After the training phase comes face recognition. A picture of a new face is decomposed into Eigen faces, with a weight assigned to each one denoting its importance (in the same way as in Figure 4). The resulting weights sequence is compared with each of the weights sequences for the training images, and the name associated with the 'closest' matching training image is used to identify the new face.

Alternatively, we can explain the recognition stage in terms of Eigen spaces: the new image is positioned in the Eigen space, with its own coordinates (weights). Then a distance measure (often Euclidean distance) is used to find the closest training image (see Figure 6).

Figure 6. Recognizing an Image in Eigen space.
Although I've been using faces in my explanation, there's nothing preventing this technique being applied to other kinds of pictures, in which case it's called *Eigen image* recognition. Eigen imaging is best applied to objects that have a regular structure made up of varying subcomponents. Faces are a good choice because they all have a very similar composition (two eyes, a nose and mouth), but with variations between the components (e.g. in size and shape).

Recognizing a New Image

The other major part of my modified version of Java faces is the code to reading in a new image (new data), and deciding which of the training images it most closely resembles. The top-level class is Face Recognizer, and uses many of the same support classes as Build Eigen Faces. The classes are shown in Figure 7.

![Figure 7 Classes Used by face Recognizer.](image)

Face Recognizer is quite separate from Build Eigen Faces, but they share information via the Eigen. Cache file. face Recognizer starts by loading the cache, and converts it back into a Face Bundle object. It loads the new
image using methods from the FileUtils and ImageUtils classes.

The face Recognizer constructor uses the calcWeights() method in Face Bundle to create weights for the training images. This is done now rather than at build-time because it’s only during face recognition that the user supplies the number of Eigen faces that will be used during the recognition process. Face Bundle. calcWeights() is listed below:

// in the FaceBundle class
public double[][] calcWeights(int numEFs)
{
    Matrix2D imsMat = new Matrix2D(imageRows);  // training images

    Matrix2D facesMat = new Matrix2D(eigenFaces);
    Matrix2D facesSubMatTr =
    facesMat.getSubMatrix(numEFs).transpose();

    Matrix2D weights = imsMat.multiply(facesSubMatTr);
    return weights.toArray();}  // end of calcWeights()

The required number of Eigen faces is supplied in the numEFs variable, and the resulting weights array is calculated by multiplying the images matrix to a sub matrix of the Eigen faces. The multiplication is illustrated in Figure 8 (recall that we are assuming that an image is N x N pixels big, and there are M training images).