

# Face Recognition with OpenCV2

Philipp Wagner  
<http://www.bytefish.de>

February 6, 2012

## 1 Introduction

[OpenCV \(Open Source Computer Vision\)](#) is a popular computer vision library started by [Intel](#) in 1999. The cross-platform library sets its focus on real-time image processing and includes patent-free implementations of the latest computer vision algorithms. In 2008 [Willow Garage](#) took over support and OpenCV 2.3.1 now comes with a programming interface to C, C++, [Python](#) and [Android](#). OpenCV is released under a BSD license, so it is used in academic and commercial projects such as [Google Streetview](#).

This document is the guide I've wished for, when I was working myself into face recognition. It helps you with installing OpenCV2 on your machine and explains you how to build a project on Windows and Linux. Two face recognition algorithms are prototyped with [GNU Octave/MATLAB](#) and implemented with the OpenCV2 C++ API. All concepts are explained in detail, but a basic knowledge of C++ is assumed. I've decided to leave the C++ implementation details out (as I am afraid they confuse people) and provide you with examples how to use the projects. [MinGW](#) (the GCC port for Windows) is used as the C/C++ compiler for Windows, because it works great with OpenCV2 and comes under terms of a public license (please see [mingw.org/license](http://mingw.org/license) for details). If someone writes a similar guide for Microsoft Visual Studio 2008/2010, I would be happy to add it to the document.

You don't need to copy and paste the code snippets, the code has been pushed into my github repository:

- [github.com/bytefish](https://github.com/bytefish)
- [github.com/bytefish/facerecognition\\_guide](https://github.com/bytefish/facerecognition_guide)
- [github.com/bytefish/opencv](https://github.com/bytefish/opencv)

All code is released under a [BSD license](#), so feel free to use it for your projects.

## 2 Installation Guide

This installation guide explains how to install the software for this document. [CMake](#) is used as build system for the examples, [MinGW \(Minimalist GNU for Windows\)](#) is used as the compiler for Windows and OpenCV2 is compiled from source. There are binaries for OpenCV2 already, so why is it useful to build it from source at all? Your architecture may not be supported by the binaries, your toolchain may differ or the OpenCV version in your repository may not be the latest. Please note: You can always use the binaries supplied by WillowGarage or the binaries supplied by your distribution if they work for you.

The following guide was tested on Microsoft Windows XP SP3 and Ubuntu 10.10.

### 2.1 Installing CMake

[CMake](#) is an open-source, cross-platform build system. It manages the build process in a compiler-independent manner and is able to generate native build environments to compile the source code ([Make](#), [Apple Xcode](#), [Microsoft Visual Studio](#), [MinGW](#), ...). Projects like [OpenCV](#), [KDE](#) or [Blender 3D](#) recently switched to CMake due to its flexibility. The CMake build process itself is controlled by

configuration files, placed in the source directory (called `CMakeLists.txt`). Each `CMakeLists.txt` consists of CMake commands in the form of `COMMAND(arguments...)`, that describe how to include header files, build libraries and executables. Please see the [CMake Documentation](#) for a list of available commands. A Windows installer is available at [cmake.org/resources/software.html](http://cmake.org/resources/software.html) (called `cmake-<version>-win32-x86.exe`). Make sure to select "Add CMake to the system PATH for all users" during setup or manually add it, so you can use `cmake`, `ccmake` and the `cmake-gui` from command line (see [Microsoft Support: How To Manage Environment Variables in Windows XP](#) for details). Linux users should check the repository of their distribution, because the CMake binaries are often available already. If CMake is not available one can build it from source by:

```
./bootstrap
make
make install
```

Or install generic Linux binaries (called `cmake-<version>-<os>-<architecture>.sh`):

```
sudo sh cmake-<version>-<os>-<architecture>.sh --prefix=/usr/local
```

## 2.2 Installing MinGW

MinGW (Minimalist GNU for Windows) is a port of the [GNU Compiler Collection \(GCC\)](#) and can be used for the development of native [Microsoft Windows](#) applications. The easiest way to install MinGW is to use the automated mingw-get-installer [from sourceforge.net/projects/mingw/files/Automated MinGW Installer/mingw-get-inst/](http://sourceforge.net/projects/mingw/files/Automated%20MinGW%20Installer/mingw-get-inst/) (called `mingw-get-inst-20101030.exe` at time of writing this). If the path to the download changes, please navigate there from [mingw.org](http://mingw.org).

Make sure to select "C++ Compiler" in the *Compiler Suite* dialog during setup. Since MinGW doesn't add its binaries to the Windows PATH environment, you'll need to manually add it. The MinGW Page says: *Add c:\MinGW\bin to the PATH environment variable by opening the System control panel, going to the Advanced tab, and clicking the Environment Variables button. If you currently have a Command Prompt window open, it will not recognize the change to the environment variables; you will need to open a new Command Prompt window to get the new PATH.*

Linux users should install [gcc](#) and [make](#) (or a build tool supported by CMake) from the repository of their distribution. In Ubuntu the `build-essential` package contains all necessary tools to get started, in Fedora and SUSE you'll need to install it from the available development tools.

## 2.3 Building OpenCV

Please skip this section if you are using the OpenCV binaries supplied by WillowGarage or your distribution. To build OpenCV you'll need CMake (see section 2.1), a C/C++ compiler (see section 2.2) and the OpenCV source code. At time of writing this, the latest OpenCV sources are available at <http://sourceforge.net/projects/opencvlibrary/>. I've heard the OpenCV page will see some changes soon, so if the sourceforge isn't used for future versions anymore navigate from the official page: <http://opencv.willowgarage.com>.

In this guide I'll use [OpenCV 2.3.0](#) for Windows and [OpenCV 2.3.1](#) for Linux. If you need the latest Windows version download the [superpack](#), which includes binaries and sources for Windows.

### Create the build folder

First of all extract the source code to a folder of your choice, then open a terminal and `cd` into this folder. Then create a folder `build`, where we will build OpenCV in:

```
mkdir build
cd build
```

### Build OpenCV in Windows

Now we'll create the Makefiles to build OpenCV. You need to specify the path you want to install OpenCV to (e.g. `c:/opencv`), preferably it's not the source folder. Note, that CMake expects a slash (`/`) as path separator. So if you are using Windows you'll now write:

```
cmake -G "MinGW Makefiles" -D:CMAKE_BUILD_TYPE=RELEASE -D:BUILD_EXAMPLES=1 -D:
    CMAKE_INSTALL_PREFIX=C:/opencv ..
mingw32-make
mingw32-make install
```

Usually CMake is good at guessing the parameters, but there are a lot more options you can set (for Qt, Python, ..., see [WillowGarage's Install Guide](#) for details). It's a good idea to use the `cmake-gui` to see and set the available switches. For now you can stick to the Listing, it works fine for Windows and Linux.

Better get a coffee, because OpenCV takes a while to compile! Once it is finished and you've decided to build dynamic libraries (assumed in this installation guide), you have to add the `bin` path of the installation to Windows `PATH` variable (e.g. `C:/opencv/bin`). If you don't know how to do that, see [Microsoft Support: How To Manage Environment Variables in Windows XP](#) for details.

## Build OpenCV in Linux

Creating the Makefiles in Linux is (almost) similar to Windows. Again choose a path you want to install OpenCV to (e.g. `/usr/local`), preferably it's not the source folder.

```
1 cmake -D CMAKE_BUILD_TYPE=RELEASE -D BUILD_EXAMPLES=1 -D CMAKE_INSTALL_PREFIX=/usr/
    local ..
2 make
3 sudo make install
```

## Sample CMakeLists.txt

Once CMake is installed a simple `CMakeLists.txt` is sufficient for building an OpenCV project:

```
# set the minimum cmake version
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)
# project name
PROJECT(hello_opencv)
# you probably need to set this
SET(OpenCV_DIR /path/to/your/opencv/installation)
# finds OpenCV
FIND_PACKAGE(OpenCV REQUIRED)
# build the executable from main.cpp
ADD_EXECUTABLE(hellocv main.cpp)
# link against the opencv libraries
TARGET_LINK_LIBRARIES(hellocv ${OpenCV_LIBS})
```

To build the project one would simply do (assuming we're in the folder with `CMakeLists.txt`):

```
# create build directory
mkdir build
# ... and cd into
cd build
# generate platform-dependent makefiles
cmake ..
# build the project
make
# run the executable
./hellocv
```

Or if you are on Windows with MinGW you would do:

```
mkdir build
cd build
cmake -G "MinGW Makefiles" ..
mingw32-make
```

## 3 Face Recognition

Face recognition is an easy task for humans. Experiments in [6] have shown, that even one to three day old babies are able to distinguish between known faces. So how hard could it be for a computer?

It turns out we know little about human recognition to date. Are inner features (eyes, nose, mouth) or outer features (head shape, hairline) used for a successful face recognition? How do we analyze an image and how does the brain encode it? It was shown by [David Hubel](#) and [Torsten Wiesel](#), that our brain has specialized nerve cells responding to specific local features of a scene, such as lines, edges, angles or movement. Since we don't see the world as scattered pieces, our visual cortex must somehow combine the different sources of information into useful patterns. Automatic face recognition is all about extracting those meaningful features from an image, putting them into a useful representation and performing some kind of classification on them.

Face recognition based on the geometric features of a face is probably the most intuitive approach to face recognition. One of the first automated face recognition systems was described in [9]: marker points (position of eyes, ears, nose, ...) were used to build a feature vector (distance between the points, angle between them, ...). The recognition was performed by calculating the euclidean distance between feature vectors of a probe and reference image. Such a method is robust against changes in illumination by its nature, but has a huge drawback: the accurate registration of the marker points is complicated, even with state of the art algorithms. Some of the latest work on geometric face recognition was carried out in [4]. A 22-dimensional feature vector was used and experiments on large datasets have shown, that geometrical features alone don't carry enough information for face recognition.

The Eigenfaces method described in [14] took a holistic approach to face recognition: A facial image is a point from a high-dimensional image space and a lower-dimensional representation is found, where classification becomes easy. The lower-dimensional subspace is found with Principal Component Analysis, which identifies the axes with maximum variance. While this kind of transformation is optimal from a reconstruction standpoint, it doesn't take any class labels into account. Imagine a situation where the variance is generated from external sources, let it be light. The axes with maximum variance do not necessarily contain any discriminative information at all, hence a classification becomes impossible. So a class-specific projection with a Linear Discriminant Analysis was applied to face recognition in [3]. The basic idea is to minimize the variance within a class, while maximizing the variance between the classes at the same time (Figure 1).

Recently various methods for a local feature extraction emerged. To avoid the high-dimensionality of the input data only local regions of an image are described, the extracted features are (hopefully) more robust against partial occlusion, illumination and small sample size. Algorithms used for a local feature extraction are Gabor Wavelets ([16]), Discrete Cosinus Transform ([5]) and Local Binary Patterns ([1, 11, 12]). It's still an open research question how to preserve spatial information when applying a local feature extraction, because spatial information is potentially useful information.

### 3.1 Eigenfaces

The problem with the image representation we are given is its high dimensionality. Two-dimensional  $p \times q$  grayscale images span a  $m = pq$ -dimensional vector space, so an image with  $100 \times 100$  pixels lies in a 10,000-dimensional image space already. That's way too much for any computations, but are all dimensions really useful for us? We can only make a decision if there's any variance in data, so what we are looking for are the components that account for most of the information. The Principal Component Analysis (PCA) was independently proposed by [Karl Pearson](#) (1901) and [Harold Hotelling](#) (1933) to turn a set of possibly correlated variables into a smaller set of uncorrelated variables. The idea is that a high-dimensional dataset is often described by correlated variables and therefore only a few meaningful dimensions account for most of the information. The PCA method finds the directions with the greatest variance in the data, called principal components.

#### 3.1.1 Algorithmic Description

Let  $X = \{x_1, x_2, \dots, x_n\}$  be a random vector with observations  $x_i \in \mathbb{R}^d$ .

1. Compute the mean  $\mu$

$$\mu = \frac{1}{n} \sum_{i=1}^n x_i \tag{1}$$

2. Compute the the Covariance Matrix  $S$

$$S = \frac{1}{n} \sum_{i=1}^n (x_i - \mu)(x_i - \mu)^T \quad (2)$$

3. Compute the eigenvalues  $\lambda_i$  and eigenvectors  $v_i$  of  $S$

$$Sv_i = \lambda_i v_i, i = 1, 2, \dots, n \quad (3)$$

4. Order the eigenvectors descending by their eigenvalue. The  $k$  principal components are the eigenvectors corresponding to the  $k$  largest eigenvalues.

The  $k$  principal components of the observed vector  $x$  are then given by:

$$y = W^T(x - \mu) \quad (4)$$

where  $W = (v_1, v_2, \dots, v_k)$ . The reconstruction from the PCA basis is given by:

$$x = Wy + \mu \quad (5)$$

The Eigenfaces method then performs face recognition by:

1. Projecting all training samples into the PCA subspace (using Equation 4).
2. Projecting the query image into the PCA subspace (using Listing 5).
3. Finding the nearest neighbor between the projected training images and the projected query image.

Still there's one problem left to solve. Imagine we are given 400 images sized  $100 \times 100$  pixel. The Principal Component Analysis solves the covariance matrix  $S = XX^T$ , where  $size(X) = 10000 \times 400$  in our example. You would end up with a  $10000 \times 10000$  matrix, roughly  $0.8GB$ . Solving this problem isn't feasible, so we'll need to apply a trick. From your linear algebra lessons you know that a  $M \times N$  matrix with  $M > N$  can only have  $N - 1$  non-zero eigenvalues. So it's possible to take the eigenvalue decomposition  $S = X^T X$  of size  $N \times N$  instead:

$$X^T X v_i = \lambda_i v_i \quad (6)$$

and get the original eigenvectors of  $S = XX^T$  with a left multiplication of the data matrix:

$$XX^T(Xv_i) = \lambda_i(Xv_i) \quad (7)$$

The resulting eigenvectors are orthogonal, to get orthonormal eigenvectors they need to be normalized to unit length. I don't want to turn this into a publication, so please look into [7] for the derivation and proof of the equations.

### 3.1.2 Example

It's always useful to prototype algorithms before implementing them with OpenCV, because this gives you an idea what the solution looks like. I use [GNU Octave/MATLAB](#) in this document, although I recently switched to [Python](#) with [NumPy](#) and [matplotlib](#). OpenCV2 uses [NumPy](#) arrays since OpenCV 2.3, so all algorithms using [NumPy](#) play fine with OpenCV's Python bindings. A full-blown [GNU Octave/MATLAB](#) and Python environment is available at <https://github.com/bytcfish/facerec>, including:

- Preprocessing
  - Histogram Equalization
  - Local Binary Patterns
  - TanTriggs Preprocessing [13]
- Feature Extraction
  - Eigenfaces [14]

- Fisherfaces [3]
- Local Binary Patterns Histograms [1]
- Classifier
  - k-Nearest Neighbor Model (with various metrics)
  - Support Vector Machine [15]
- Cross Validation
  - k-fold CV
  - Leave One Out CV
  - Leave One Subject Out CV

I don't want to do a toy example here. We are doing face recognition, so you'll need some face images. You can either create your own database or start with one of the available databases, [face-rec.org/databases](http://face-rec.org/databases) gives an up-to-date overview. Three interesting databases are<sup>1</sup>:

**AT&T Facedatabase** The AT&T Facedatabase, sometimes also known as *ORL Database of Faces*, contains ten different images of each of 40 distinct subjects. For some subjects, the images were taken at different times, varying the lighting, facial expressions (open / closed eyes, smiling / not smiling) and facial details (glasses / no glasses). All the images were taken against a dark homogeneous background with the subjects in an upright, frontal position (with tolerance for some side movement).

**Yale Facedatabase A** The AT&T Facedatabase is good for initial tests, but it's a fairly easy database. The Eigenfaces method already has a 97% recognition rate, so you won't see any improvements with other algorithms. The Yale Facedatabase A is a more appropriate dataset for initial experiments, because the recognition problem is harder. The database consists of 15 people (14 male, 1 female) each with 11 grayscale images sized  $320 \times 243$  pixel. There are changes in the light conditions (center light, left light, right light), facial expressions (happy, normal, sad, sleepy, surprised, wink) and glasses (glasses, no-glasses).

Bad news is it's not available for public download anymore, because the original server seems to be down. You can find some sites mirroring it ([like the MIT](#)), but I can't make guarantees about the integrity. If you need to crop and align images yourself, read my notes at [byte-fish.de/blog/fisherfaces](http://byte-fish.de/blog/fisherfaces).

**Extended Yale Facedatabase B** The Extended Yale Facedatabase B contains 2414 images of 38 different people in its cropped version. The focus is on extracting features that are robust to illumination, the images have almost no variation in emotion/occlusion/... I personally think, that this dataset is too large for the experiments I perform in this document, you better use the [AT&T Facedatabase](#). A first version of the Yale Facedatabase B was used in [3] to see how the Eigenfaces and Fisherfaces method (section 4.5) perform under heavy illumination changes. [10] used the same setup to take 16128 images of 28 people. The Extended Yale Facedatabase B is the merge of the two databases, which is now known as Extended Yalefacedatabase B.

The face images need to be stored in a folder hierarchy similar to `<database name>/<subject name>/<filename>.<ext>`. The [AT&T Facedatabase](#) already comes in such a hierarchy:

```
philipp@mango:~/facerec/data/at$ tree
.
|-- README
|-- s1
|   |-- 10.pgm
|   |-- 1.pgm
|   |-- 2.pgm
|   |-- 3.pgm
|   |-- ...
|-- s2
|   |-- 10.pgm
|   |-- 1.pgm
|   |-- 2.pgm
|   |-- 3.pgm
```

<sup>1</sup>Parts of the description are quoted from [face-rec.org](http://face-rec.org).

```
| |-- ...
...
|-- s40
| |-- 10.pgm
| |-- 1.pgm
| |-- 2.pgm
| |-- 3.pgm
| |-- ...
|
```

First we'll need a function to list the files for a given path. This can be done with a function similar to Listing 1.

Listing 1: [src/m/list\\_files.m](#)

```
function L = list_files(path_fn)
% get information about given path_fn
L = dir(path_fn);
% ... ignore . and ..
L = L(3:length(L));
% ... turn into a cell array
L = struct2cell(L);
% ... and only keep the filenames.
L = L(1,:);
end
```

The function in Listing 2 can be used to read in the images for each subfolder of a given directory. Each directory is given a unique (integer) label, you probably want to store the folder name as well. The function returns the images as a data matrix and the corresponding classes, the width and height of the images (we'll need this in later code). This function is really basic and there's much to enhance, but it does its job.

Listing 2: [src/m/read\\_images.m](#)

```
function [X y width height] = read_images(path_fn)
% get files for a given path
folder = list_files(path_fn);
% initialize the empty return values
X=[];
y=[];
width=0;
height=0;
% start counting with class index 1
classIdx = 1;
% for each file...
for i=1:length(folder)
    subject = folder{i};
    % ... get files in this subdir
    images = list_files([path_fn, filesep, subject]);
    % ... ignore a file or empty folder
    if(length(images) == 0)
        continue;
    end
    % ... for each image
    for j=1:length(images)
        % ... get the absolute path
        filename = [path_fn, filesep, subject, filesep, images{j}];
        % ... read the image
        T = double(imread(filename));
        % ... get the image information
        [height width channels] = size(T);
        % ... and grayscale if it's a color image
        if(channels == 3)
            T = 0.2989 * T(:, :, 1) + 0.5870 * T(:, :, 2) + 0.1140 * T(:, :, 3);
        end
        % ... reshape into a row vector and append to data matrix
        X = [X; reshape(T,1,width*height)];
        % ... append the corresponding class to the class vector
        y = [y, classIdx];
    end
    % ... increase the class index
    classIdx = classIdx + 1;
end
```

```

    end % ... for-each folder.
end

```

We want to plot some data, so we need a method to turn data into a representation [GNU Octave/MATLAB](#) understands. The image data is expected as unsigned integer values in range  $[0, 255]$ , so we need a function to normalize the data first (Listing 3):

Listing 3: [src/m/normalize.m](#)

```

function X = normalize(X, l, h)
    minX = min(X(:));
    maxX = max(X(:));
    %% Normalize to [0...1].
    X = X - minX;
    X = X ./ (maxX - minX);
    %% Scale to [low...high].
    X = X .* (h-l);
    X = X + l;
end

```

Listing 4 then turns the image into the expected representation, by first normalizing the values to a range between  $[0, 255]$  and then casting the matrix to unsigned integer values.

Listing 4: [src/m/toGrayscale.m](#)

```

function Y = toGrayscale(X, width, height)
    Y = normalize(X, 0, 255);
    if(nargin==3)
        Y = reshape(Y, height, width);
    end
    Y = uint8(Y);
end

```

Translating the PCA from the algorithmic description of section 3.1.1 to [GNU Octave/MATLAB](#) is almost trivial. Don't copy and paste from this document, the source code is available in folder [src/m](#). Listing 5 implements the Principal Component Analysis given by Equation 1, 2 and 3. It also implements the inner-product PCA formulation, which occurs if there are more dimensions than samples. You can shorten this code, I just wanted to point out how it works.

Listing 5: [src/m/pca.m](#)

```

function [W, mu] = pca(X, y, k)
    [n,d] = size(X);
    mu = mean(X);
    Xm = X - repmat(mu, rows(X), 1);
    if(n>d)
        C = Xm'*Xm;
        [W,D] = eig(C);
        % sort eigenvalues and eigenvectors
        [D, i] = sort(diag(D), 'descend');
        W = W(:,i);
        % keep k components
        W = W(:,1:k);
    else
        C = Xm*Xm';
        %C = cov(Xm');
        [W,D] = eig(C);
        % multiply with data matrix
        W = Xm'*W;
        % normalize eigenvectors
        for i=1:n
            W(:,i) = W(:,i)/norm(W(:,i));
        end
        % sort eigenvalues and eigenvectors
        [D, i] = sort(diag(D), 'descend');
        W = W(:,i);
        % keep k components
        W = W(:,1:k);
    end
end

```



The observations are given by row, so the projection in Equation 4 needs to be rearranged a little:

Listing 6: [src/m/project.m](#)

```
function Y = project(W, X, mu)
    if(nargin<3)
        Y = X*W;
    else
        Y = (X-repmat(mu, rows(X), 1))*W;
    end
end
```

The same applies to the reconstruction in Equation 5:

Listing 7: [src/m/reconstruct.m](#)

```
function X = reconstruct(W, Y, mu)
    if(nargin<3)
        X = Y * W';
    else
        X = Y * W' + repmat(mu, rows(Y), 1);
    end
end
```

Now that everything is defined it's time for the fun stuff. The face images are read with Listing 2 and then a full PCA (see Listing 5) is performed. I'll use the great [matplotlib](#) library for plotting in [Python](#), please install it if you haven't done already.

Listing 8: [src/m/example\\_eigenfaces.m](#)

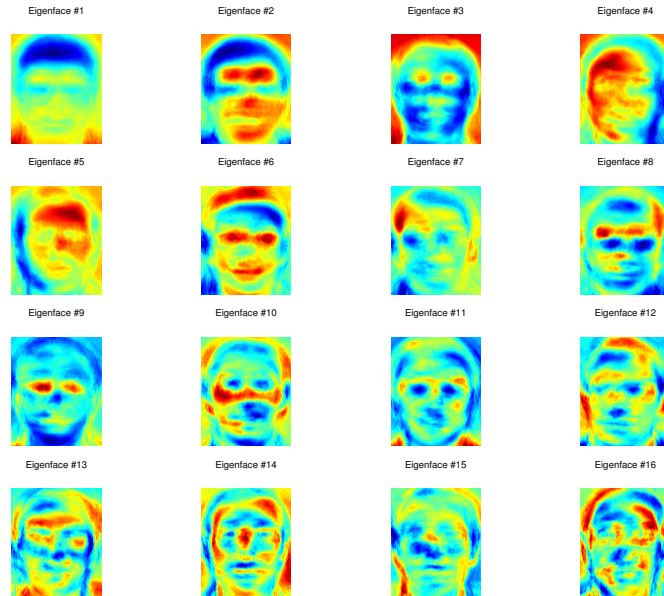
```
% load function files from subfolders aswell
addpath (genpath ('.'));
% read images
[X,y,w,h] = read_images('/home/philipp/facerec/data/at');
% n - number of samples
% d - dimensionality
[n,d] = size(X);
% perform a full pca
[W,mu] = pca(X,y,n);
```

That's it already. Pretty easy, no? Each principal component has the same length as the original image, thus it can be displayed as an image. These ghostly looking faces are called the *Eigenfaces*, that's where the Eigenfaces method got its name from. We'll do a subplot for the first at most 16 Eigenfaces.

Listing 9: [src/m/example\\_eigenfaces.m](#)

```
% plot eigenfaces
figure; hold on;
title('Eigenfaces (AT&T Facedatabase)');
for i=1:min(16,n)
    subplot(4,4,i);
    eigenface_i = toGrayscale(W(:,i), w, h);
    imshow(eigenface_i);
    colormap(jet(256));
    title(sprintf('Eigenface #%i', i));
end
```

I've used a colormap, so you can see how the grayscale values are distributed within the specific Eigenfaces. You can see, that the Eigenfaces do not only encode facial features, but also the illumination in the images (see the left light in Eigenface #4, right light in Eigenfaces #5):

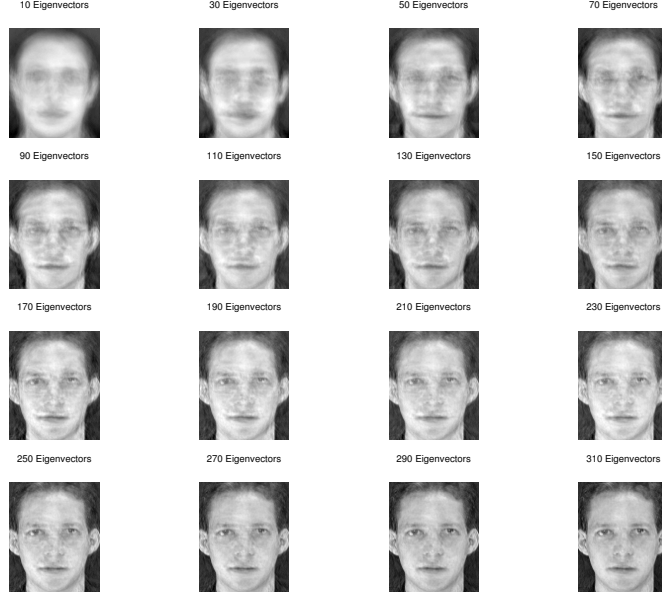


We've already seen in Equation 5, that we can reconstruct a face from its lower dimensional approximation. So let's see how many Eigenfaces are needed for a good reconstruction. I'll do a subplot with 10, 30, ..., 310 Eigenfaces:

Listing 10: [src/m/example\\_eigenfaces.m](#)

```
%plot eigenfaces reconstruction
steps = 10:20:min(n,320);
Q = X(1,:); % first image to reconstruct
figure; hold on;
title('Reconstruction (AT&T Facedatabase)');
for i=1:min(16, length(steps))
    subplot(4,4,i);
    numEvs = steps(i);
    P = project(W(:,1:numEvs), X(1,:), mu);
    R = reconstruct(W(:,1:numEvs),P,mu);
    comp = toGrayscale(R, w, h);
    imshow(comp);
    title(sprintf('%i Eigenvectors', numEvs));
end
```

10 Eigenvectors are obviously not sufficient for a good image reconstruction, 50 Eigenvectors may already be sufficient to encode important facial features. You'll get a good reconstruction with approximately 300 Eigenvectors for the AT&T Facedatabase. There are rule of thumbs how many Eigenfaces you should choose for a successful face recognition, but it heavily depends on the input data. [17] is a good point to start researching for this.



The k-Nearest Neighbor matching is left out for this example. Please see the GNU Octave/MATLAB code at <https://github.com/bytefish/facerec/tree/master/m> to see how it is implemented, it's all there.

## 3.2 Fisherfaces

The Linear Discriminant Analysis was invented by the great statistician [Sir R. A. Fisher](#), who successfully used it for classifying flowers in his 1936 paper *The use of multiple measurements in taxonomic problems* [8]. But why do we need another dimensionality reduction method, if the Principal Component Analysis (PCA) did such a good job?

The PCA finds a linear combination of features that maximizes the total variance in data. While this is clearly a powerful way to represent data, it doesn't consider any classes and so a lot of discriminative information may be lost when throwing components away. Imagine a situation where the variance is generated by an external source, let it be the light. The components identified by a PCA do not necessarily contain any discriminative information at all, so the projected samples are smeared together and a classification becomes impossible.

In order to find the combination of features that separates best between classes the Linear Discriminant Analysis maximizes the ratio of between-classes to within-classes scatter. The idea is simple: same classes should cluster tightly together, while different classes are as far away as possible from each other. This was also recognized by [Belhumeur](#), [Hespanha](#) and [Kriegman](#) and so they applied a Discriminant Analysis to face recognition in [3].

### 3.2.1 Algorithmic Description

Let  $X$  be a random vector with samples drawn from  $c$  classes:

$$X = \{X_1, X_2, \dots, X_c\} \quad (8)$$

$$X_i = \{x_1, x_2, \dots, x_n\} \quad (9)$$

The scatter matrices  $S_B$  and  $S_W$  are calculated as:

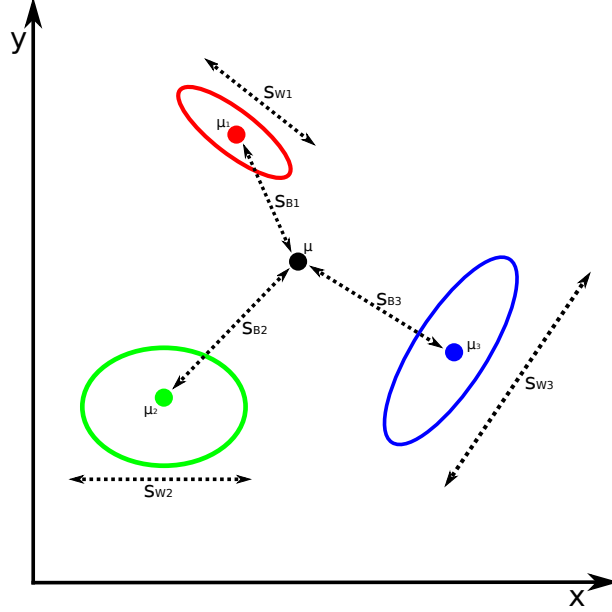


Figure 1: This figure shows the scatter matrices  $S_B$  and  $S_W$  for a 3 class problem.  $\mu$  represents the total mean and  $[\mu_1, \mu_2, \mu_3]$  are the class means.

$$S_B = \sum_{i=1}^c N_i (\mu_i - \mu)(\mu_i - \mu)^T \quad (10)$$

$$S_W = \sum_{i=1}^c \sum_{x_j \in X_i} (x_j - \mu_i)(x_j - \mu_i)^T \quad (11)$$

, where  $\mu$  is the total mean:

$$\mu = \frac{1}{N} \sum_{i=1}^N x_i \quad (12)$$

And  $\mu_i$  is the mean of class  $i \in \{1, \dots, c\}$ :

$$\mu_i = \frac{1}{|X_i|} \sum_{x_j \in X_i} x_j \quad (13)$$

Fisher's classic algorithm now looks for a projection  $W$ , that maximizes the class separability criterion:

$$W_{opt} = \arg \max_W \frac{|W^T S_B W|}{|W^T S_W W|} \quad (14)$$

Following [3], a solution for this optimization problem is given by solving the General Eigenvalue Problem:

$$\begin{aligned} S_B v_i &= \lambda_i S_W v_i \\ S_W^{-1} S_B v_i &= \lambda_i v_i \end{aligned} \quad (15)$$

There's one problem left to solve: The rank of  $S_W$  is at most  $(N - c)$ , with  $N$  samples and  $c$  classes. In pattern recognition problems the number of samples  $N$  is almost always smaller than the dimension of the input data (the number of pixels), so the scatter matrix  $S_W$  becomes singular (see [2]). In [3] this was solved by performing a Principal Component Analysis on the data and projecting the

samples into the  $(N - c)$ -dimensional space. A Linear Discriminant Analysis was then performed on the reduced data, because  $S_W$  isn't singular anymore.

The optimization problem can be rewritten as:

$$W_{pca} = \arg \max_W |W^T S_T W| \quad (16)$$

$$W_{fld} = \arg \max_W \frac{|W^T W_{pca}^T S_B W_{pca} W|}{|W^T W_{pca}^T S_W W_{pca} W|} \quad (17)$$

The transformation matrix  $W$ , that projects a sample into the  $(c - 1)$ -dimensional space is then given by:

$$W = W_{fld}^T W_{pca}^T \quad (18)$$

One final note: Although  $S_W$  and  $S_B$  are symmetric matrices, the product of two symmetric matrices is not necessarily symmetric. so you have to use an eigenvalue solver for general matrices. OpenCV's `cv::eigen` only works for symmetric matrices in its current version; since eigenvalues and singular values aren't equivalent for non-symmetric matrices you can't use a Singular Value Decomposition (SVD) either.

### 3.2.2 Example

Translating the Linear Discriminant Analysis to [GNU Octave/MATLAB](#) is almost trivial again, see Listing 11. For projecting and reconstructing from the basis you can use the functions from Listing 6 and 7.

Listing 11: [src/m/lda.m](#)

```
function [W, mu] = lda(X,y,k)
% dimension of observation
[n,d] = size(X);
% number of classes
labels = unique(y);
C = length(labels);
% allocate scatter matrices
Sw = zeros(d,d);
Sb = zeros(d,d);
% total mean
mu = mean(X);
% calculate scatter matrices
for i = 1:C
    Xi = X(find(y == labels(i)),:); % samples for current class
    n = rows(Xi);
    mu_i = mean(Xi); % mean vector for current class
    Xi = Xi - repmat(mu_i, n, 1);
    Sw = Sw + Xi'*Xi;
    Sb = Sb + n * (mu_i - mu)'*(mu_i - mu);
end
% solve general eigenvalue problem
[W, D] = eig(Sb, Sw);
% sort eigenvectors
[D, i] = sort(diag(D), 'descend');
W = W(:,i);
% keep at most (c-1) eigenvectors
W = W(:,1:k);
end
```

The functions to perform a PCA (Listing 5) and LDA (Listing 11) are now defined, so we can go ahead and implement the Fisherfaces from Equation 18.

Listing 12: [src/m/fisherfaces.m](#)

```
function [W, mu] = fisherfaces(X,y,k)
% number of samples
N = rows(X);
% number of classes
labels = unique(y);
```

```

c = length(labels);
if(nargin < 3)
    k = c-1;
end
k = min(k,(c-1));
% get (N-c) principal components
[Wpca, mu] = pca(X, y, (N-c));
[Wlda, mu_lda] = lda(project(Wpca, X, mu), y, k);
W = Wpca*Wlda;
end

```

For this example I am going to use the Yale Facedatabase A, just because the plots are nicer. Each Fisherface has the same length as an original image, thus it can be displayed as an image. We'll again load the data, learn the Fisherfaces and make a subplot of the first 16 Fisherfaces.

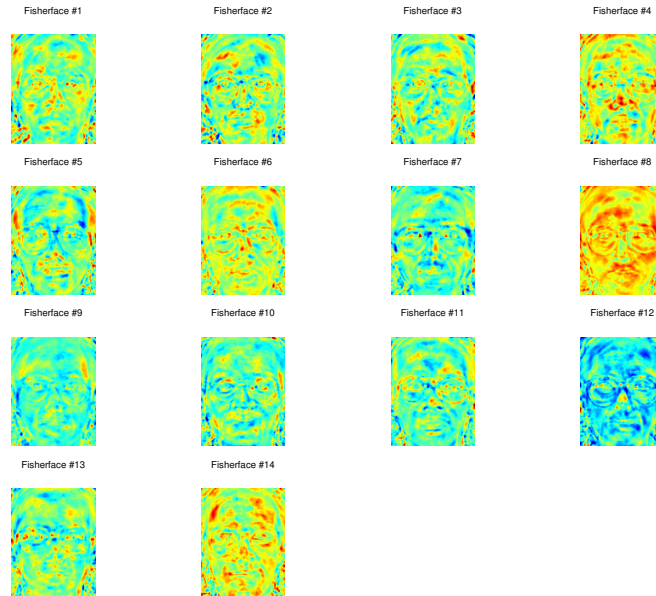
Listing 13: [src/m/example\\_fisherfaces.m](#)

```

% load function files from subfolders aswell
addpath (genpath ('.'));
% for plotting
db_name = 'Yale Facedatabase';
% read images
[X,y,w,h] = read_images('/home/philipp/facerec/data/yalefaces_recognition');
% n - number of samples
% d - dimensionality
[n,d] = size(X);
% get the unique classes
c = unique(y);
% compute the fisherfaces
[W,mu] = fisherfaces(X,y);
% plot fisherfaces
figure; hold on;
title(sprintf('Fisherfaces %s', db_name));
for i=1:min(16,length(c)-1)
    subplot(4,4,i);
    fisherface_i = toGrayscale(W(:,i), w, h);
    imshow(fisherface_i);
    colormap(jet(256));
    title(sprintf('Fisherface #%i', i));
end

```

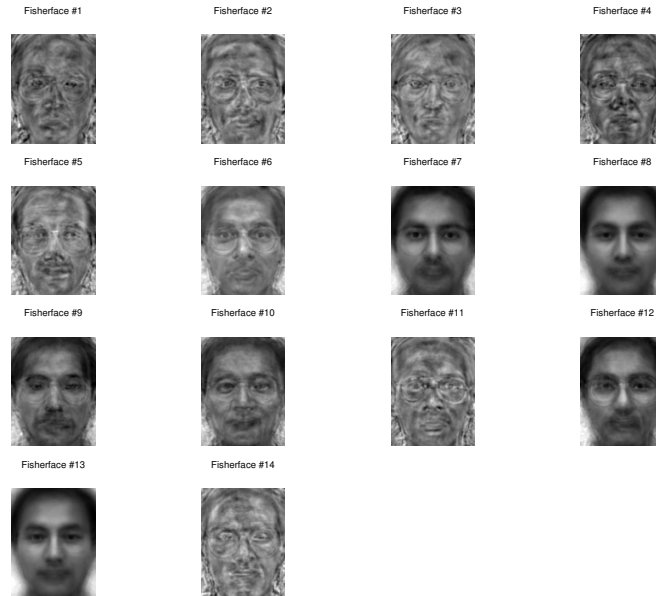
The Fisherfaces method learns a class-specific transformation matrix, so they do not capture illumination as obviously as the Eigenfaces method. The Discriminant Analysis instead finds the facial features to discriminate between the persons. It's important to mention, that the performance of the Fisherfaces heavily depends on the input data as well. Practically said: if you learn the Fisherfaces for well-illuminated pictures only and you try to recognize faces in bad-illuminated scenes, then method is likely to find the wrong components (just because those features may not be predominant on bad illuminated images). This is somewhat logical, since the method had no chance to learn the illumination.



The Fisherfaces allow a reconstruction of the projected image, just like the Eigenfaces did. But since we only identified the features to distinguish between subjects, you can't expect a nice approximation of the original image. We can rewrite Listing 10 for the Fisherfaces method into Listing 14, but this time we'll project the sample image onto each of the Fisherfaces instead. So you'll have a visualization, which features each Fisherface describes.

Listing 14: [src/m/example.fisherfaces.m](#)

```
steps = 1:min(16,length(c)-1);
Q = X(1,:); % first image to reconstruct
figure; hold on;
title(sprintf('Fisherfaces Reconstruction %s', db_name));
for i=1:min(16, length(steps))
    subplot(4,4,i);
    numEv = steps(i);
    P = project(W(:,numEv), X(1,:), mu);
    R = reconstruct(W(:,numEv),P,mu);
    comp = toGrayscale(R, w, h);
    imshow(comp);
    title(sprintf('Fisherface #%i', numEv));
end
```



The k-Nearest Neighbor matching is left out for this example. Please see the GNU Octave/MATLAB code at <https://github.com/bytefish/facerec/tree/master/m> to see how it is implemented, it's all there.

## 4 Face Recognition with OpenCV

The C++ API of OpenCV2 closely resembles the GNU Octave/MATLAB code we've written in section 3.1.2 and 3.2.2. If you know how to implement it in MATLAB, you won't have a great problem translating it to OpenCV2. I personally think you don't need a book to learn about the OpenCV2 C++ API. There's a lot of documentation coming with OpenCV, just have a look into the `doc` folder of your OpenCV installation. The easiest way to get started is the *OpenCV Cheat Sheet (C++)* (`opencv_cheatsheet.pdf`), because it shows you how to use all the functions with examples. The *The OpenCV Reference Manual* (`opencv2refman.pdf`) is the definite guide to the API (500+ pages). Of course, you'll need a book or other literature for understanding computer vision algorithms - I can't give an introduction to this here.

### 4.1 Downloading and Building the Source Code

Only a high-level description of the implementation is given, because pasting the complete code doesn't make any sense. I wanted to explain some implementation details, but I am afraid it would confuse people. The source code is available at <http://www.github.com/bytefish/opencv>:

- **Eigenfaces:** <https://github.com/bytefish/opencv/tree/master/eigenfaces>
- **Fisherfaces:** <https://github.com/bytefish/opencv/tree/master/lda>

If you want to clone both projects with `git` then issue:

```
git clone git@github.com:bytefish/opencv.git
```

However, if you don't have `git` on your system you can download both projects as zip or tarball:

- **zip** <https://github.com/bytefish/opencv/zipball/master>
- **tar** <https://github.com/bytefish/opencv/tarball/master>

Building the demo executables is then as simple as (assuming you are in a projects folder):



```
mkdir build
cd build
cmake ..
make
./eigenfaces /path/to/csv.ext
```

Or if you are on Windows with MinGW you would do:

```
mkdir build
cd build
cmake -G "MinGW Makefiles" ..
mingw32-make
eigenfaces.exe /path/to/csv.ext
```

Please make sure to read the *README.markdown* coming with each project!

## 4.2 Reading in the face images

OpenCV can read data from various video sources and image types, you can start your research with the documentation on [Reading and Writing Images and Video](#). I needed to read images from different folders for a project and I don't know a simpler approach than reading from a CSV file (if you know a simpler approach, please ping me):

```
void read_csv(const string& filename, vector<Mat>& images, vector<int>& labels) {
    std::ifstream file(filename.c_str(), ifstream::in);
    if(!file)
        throw std::exception();
    std::string line, path, classlabel;
    // for each line
    while (std::getline(file, line)) {
        // get current line
        std::stringstream liness(line);
        // split line
        std::getline(liness, path, ';');
        std::getline(liness, classlabel);
        // push pack the data
        images.push_back(imread(path,0));
        labels.push_back(atoi(classlabel.c_str()));
    }
}
```

Some people had questions about the usage of the executables, mainly concerned with reading the images and the corresponding labels from a CSV file. Basically all the CSV file needs to contain are lines composed of a *filename* followed by a ; followed by the *label* (as integer number), making up a line like this: `/path/to/image;0`. So if the AT&T Facedatabase is extracted to `/home/philipp/facerec/data/at` the CSV file has to look like this:

```
/home/philipp/facerec/data/at/s1/1.pgm;0
/home/philipp/facerec/data/at/s1/2.pgm;0
[...]
/home/philipp/facerec/data/at/s2/1.pgm;1
/home/philipp/facerec/data/at/s2/2.pgm;1
[...]
/home/philipp/facerec/data/at/s40/1.pgm;39
/home/philipp/facerec/data/at/s40/2.pgm;39
```

Think of the label as the subject (the person) you want to recognize. You don't need to take care about the order of the labels, just make sure the same subjects (persons) belong to the same (unique) label.<sup>2</sup> I'll now show the definition of the classes and a source code listing, that shows how to use the classes. I think the code answers most of the questions already.

## 4.3 Eigenfaces

### 4.3.1 Definiton

<sup>2</sup>The CSV file for the AT&T Database comes with this document.

```

class Eigenfaces {
public:
    /// Initialize an empty Eigenfaces model with num_components = 0
    /// and dataAsRow = true.
    Eigenfaces();
    /// Initialize a Eigenfaces model for num_components and dataAsRow.
    Eigenfaces(int num_components, bool dataAsRow = true);
    /// compute the eigenfaces for data (given in src) and labels, keep
    /// num_components principal components. Pass dataAsRow = true, if
    /// the observations are given by row, false if given by column.
    Eigenfaces(const vector<Mat>& src,
               const vector<int>& labels,
               int num_components = 0,
               bool dataAsRow = true);
    /// compute the eigenfaces for data (given in src) and labels, keep
    /// num_components principal components. Pass dataAsRow = true, if
    /// the observations are given by row, false if given by column.
    Eigenfaces(const Mat& src,
               const vector<int>& labels,
               int num_components = 0,
               bool dataAsRow = true);
    /// compute the eigenfaces for data (given in src) and labels
    void compute(const vector<Mat>& src, const vector<int>& labels);
    /// compute the eigenfaces for data (given in src) and labels
    void compute(const Mat& src, const vector<int>& labels);
    /// get a prediction for a given a sample
    int predict(const Mat& src);
    /// project a sample
    Mat project(const Mat& src);
    /// reconstruct a sample
    Mat reconstruct(const Mat& src);
    /// getter
    Mat eigenvectors() const;
    Mat eigenvalues() const;
    Mat mean() const;
};

```

### 4.3.2 Example

```

// ...
// include the eigenfaces
#include "eigenfaces.hpp"
// ...
int main(int argc, char *argv[]) {
    // the samples and corresponding labels (classes/subjects/...)
    vector<Mat> images;
    vector<int> labels;
    // read in images and labels
    string fn_csv = string("/path/to/your/csv.ext");
    // read in the images
    try {
        read_csv(fn_csv, images, labels);
    } catch(exception& e) {
        cerr << "Error opening file \"" << fn_csv << "\"." << endl;
        exit(1);
    }
    // get width and height of the samples
    int width = images[0].cols;
    int height = images[0].rows;
    // get a test sample
    Mat testSample = images[images.size()-1];
    int testLabel = labels[labels.size()-1];
    // ... and delete it from training samples
    images.pop_back();
    labels.pop_back();
    // num_components eigenfaces
    int num_components = 80;
    // compute the eigenfaces
    Eigenfaces eigenfaces(images, labels, num_components);
}

```

```

// get a prediction (recognize a face)
int predicted = eigenfaces.predict(testSample);
cout << "actual=" << testLabel << " / predicted=" << predicted << endl;
// see the reconstruction with num_components
Mat p = eigenfaces.project(images[0].reshape(1,1));
Mat r = eigenfaces.reconstruct(p);
// see the reconstruction with num_components eigenfaces
imshow("original", images[0]);
imshow("reconstruction", toGrayscale(r.reshape(1, height)));
// get the eigenvectors
Mat W = eigenfaces.eigenvectors();
// show first 10 eigenfaces
for(int i = 0; i < 10; i++) {
    Mat ev = W.col(i).clone();
    imshow(num2str(i), toGrayscale(ev.reshape(1, height)));
}
// ...
}

```

## 4.4 Fisherfaces

### 4.4.1 Definition

```

class Fisherfaces {
public:
    ///! Initialize an empty Eigenfaces model with num_components = 0
    /// and dataAsRow = true.
    Fisherfaces();
    ///! Initialize a Fisherfaces model for num_components and dataAsRow.
    Fisherfaces(int num_components,
        bool dataAsRow = true);
    ///! compute the fisherfaces for data (given in src) and labels, keep
    /// num_components discriminants. Pass dataAsRow = true, if the
    /// observations are given by row, false if given by column.
    Fisherfaces(const vector<Mat>& src,
        const vector<int>& labels,
        int num_components = 0,
        bool dataAsRow = true);
    ///! compute the fisherfaces for data (given in src) and labels, keep
    /// num_components discriminants. Pass dataAsRow = true, if the
    /// observations are given by row, false if given by column.
    Fisherfaces(const Mat& src,
        const vector<int>& labels,
        int num_components = 0,
        bool dataAsRow = true);
    ///! compute the fisherfaces for data (given in src) and labels
    void compute(const Mat& src, const vector<int>& labels);
    ///! compute the fisherfaces for data (given in src) and labels
    void compute(const vector<Mat>& src, const vector<int>& labels);
    ///! get a prediction for a given a sample
    int predict(const Mat& src);
    ///! project a sample
    Mat project(const Mat& src);
    ///! reconstruct a sample
    Mat reconstruct(const Mat& src);
    /// getter
    Mat eigenvectors() const;
    Mat eigenvalues() const;
    Mat mean() const;
};

```

### 4.4.2 Example

```

// ...
// include the fisherfaces header
#include "fisherfaces.hpp"
// ...

```

```

int main(int argc, char *argv[]) {
    // the samples and corresponding labels (classes/subjects/...)
    vector<Mat> images;
    vector<int> labels;
    // read in images and labels
    string fn_csv = string("/path/to/your/csv.ext");
    // read in the images
    try {
        read_csv(fn_csv, images, labels);
    } catch(exception& e) {
        cerr << "Error opening file \"" << fn_csv << "\"." << endl;
        exit(1);
    }
    // get width and height
    int width = images[0].cols;
    int height = images[0].rows;
    // get test instances
    Mat testSample = images[images.size()-1];
    int testLabel = labels[labels.size()-1];
    // ... and delete last element
    images.pop_back();
    labels.pop_back();
    // build the Fisherfaces model
    subspace::Fisherfaces model(images, labels);
    // test model
    int predicted = model.predict(testSample);
    cout << "predicted class = " << predicted << endl;
    cout << "actual class = " << testLabel << endl;
    // get the eigenvectors
    Mat W = model.eigenvectors();
    // show first 10 fisherfaces
    for(int i = 0; i < 10; i++) {
        Mat ev = W.col(i).clone();
        imshow(num2str(i), toGrayscale(ev.reshape(1, height)));
    }
}

```

## 4.5 Linear Discriminant Analysis

The Fisherfaces method includes a Linear Discriminant Analysis, so you get this class for free. Please read section , which explains why you can't use this method directly on image data.

### 4.5.1 Definition

```

class LinearDiscriminantAnalysis {
public:
    //! Initialize an empty LDA model with num_components = 0
    // and dataAsRow = true.
    LinearDiscriminantAnalysis();
    //! Initialize a LDA model for num_components and dataAsRow.
    LinearDiscriminantAnalysis(int num_components, bool dataAsRow = true);
    //! compute the LDA for data (given in src) and labels, keep
    // num_components discriminants. Pass dataAsRow = true, if the
    // observations are given by row, false if given by column.
    LinearDiscriminantAnalysis(const Mat& src,
        const vector<int>& labels,
        int num_components = 0,
        bool dataAsRow = true);
    //! compute the LDA for data (given in src) and labels, keep
    // num_components discriminants. Pass dataAsRow = true, if the
    // observations are given by row, false if given by column.
    LinearDiscriminantAnalysis(const vector<Mat>& src,
        const vector<int>& labels,
        int num_components = 0,
        bool dataAsRow = true);
    //! compute the LDA for data (given in src) and labels
    void compute(const Mat& src, const vector<int>& labels);
    //! compute the LDA for data (given in src) and labels

```

```

void compute(const vector<Mat>& src, const vector<int>& labels);
///! project a sample
Mat project(const Mat& src);
///! reconstruct a sample
Mat reconstruct(const Mat& src);
///! getter
Mat eigenvectors() const;
Mat eigenvalues() const;
};

```

#### 4.5.2 Example

This example is the OpenCV C++ implementation of the tutorial at [http://bytefish.de/wiki/pca\\_lda\\_with\\_gnu\\_octave](http://bytefish.de/wiki/pca_lda_with_gnu_octave). The values found by GNU Octave are reported in the comments. If you want to work through the example yourself, the GNU Octave/MATLAB code is [given on the wiki page](#).

```

// ...
// include the fisherfaces header
#include "fisherfaces.hpp"
// ... or
// #include "subspace.hpp"

int main(int argc, char *argv[]) {
    // example taken from: http://www.bytefish.de/wiki/pca_lda_with_gnu_octave
    double d[11][2] = {
        {2, 3},
        {3, 4},
        {4, 5},
        {5, 6},
        {5, 7},
        {2, 1},
        {3, 2},
        {4, 2},
        {4, 3},
        {6, 4},
        {7, 6}};
    int c[11] = {0,0,0,0,0,1,1,1,1,1,1};
    // convert into OpenCV representation
    Mat _data = Mat(11, 2, CV_64FC1, d).clone();
    vector<int> _classes(c, c + sizeof(c) / sizeof(int));
    // perform the lda
    subspace::LinearDiscriminantAnalysis lda(_data, _classes);
    // GNU Octave finds the following Eigenvalue:
    // octave> d
    // d =
    // 1.5195e+00
    //
    // Eigen finds the following Eigenvalue:
    // [1.519536390756363]
    //
    // Since there's only 1 discriminant, this is correct.
    cout << "Eigenvalues:" << endl << lda.eigenvalues() << endl;
    // GNU Octave finds the following Eigenvectors:
    // octave:13> V(:,1)
    // V =
    //
    // 0.71169 -0.96623
    // -0.70249 -0.25766
    //
    // Eigen finds the following Eigenvector:
    // [0.7116932742510111;
    // -0.702490343980524 ]
    //
    cout << "Eigenvectors:" << endl << lda.eigenvectors() << endl;
    // project a data sample onto the subspace identified by LDA
    Mat x = _data.row(0);
    cout << "Projection of " << x << ": " << endl;
    cout << lda.project(x) << endl;
}

```

## 5 Conclusion

This document explained and implemented the Eigenfaces [14] and the Fisherfaces [3] method with GNU Octave/MATLAB and OpenCV2. CMake was used as a cross-platform build system, which makes it really easy to build platform-dependent Makefiles. All code in this document is put under a BSD license, so feel free to use it for your academic and commercial projects. More (maybe) here:

- <http://www.bytefish.de>
- <http://www.github.com/bytefish>

## References

- [1] AHONEN, T., HADID, A., AND PIETIKAINEN, M. Face Recognition with Local Binary Patterns. *Computer Vision - ECCV 2004* (2004), 469–481.
- [2] A.K. JAIN, S. J. R. Small sample size effects in statistical pattern recognition: Recommendations for practitioners. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 13, 3 (1991), 252–264.
- [3] BELHUMEUR, P. N., HESPAÑA, J., AND KRIEGMAN, D. Eigenfaces vs. fisherfaces: Recognition using class specific linear projection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 19, 7 (1997), 711–720.
- [4] BRUNELLI, R., AND POGGIO, T. Face recognition through geometrical features. In *European Conference on Computer Vision (ECCV)* (1992), pp. 792–800.
- [5] CARDINAUX, F., SANDERSON, C., AND BENGIO, S. User authentication via adapted statistical models of face images. *IEEE Transactions on Signal Processing* 54 (January 2006), 361–373.
- [6] CHIARA TURATI, VIOLA MACCHI CASSIA, F. S., AND LEO, I. Newborns face recognition: Role of inner and outer facial features. *Child Development* 77, 2 (2006), 297–311.
- [7] DUDA, R. O., HART, P. E., AND STORK, D. G. *Pattern Classification (2nd Edition)*, 2 ed. November 2001.
- [8] FISHER, R. A. The use of multiple measurements in taxonomic problems. *Annals Eugen.* 7 (1936), 179–188.
- [9] KANADE, T. *Picture processing system by computer complex and recognition of human faces*. PhD thesis, Kyoto University, November 1973.
- [10] LEE, K.-C., HO, J., AND KRIEGMAN, D. Acquiring linear subspaces for face recognition under variable lighting. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)* 27, 5 (2005).
- [11] MATURANA, D., MERY, D., AND SOTO, A. Face recognition with local binary patterns, spatial pyramid histograms and naive bayes nearest neighbor classification. *2009 International Conference of the Chilean Computer Science Society (SCCC)* (2009), 125–132.
- [12] RODRIGUEZ, Y. *Face Detection and Verification using Local Binary Patterns*. PhD thesis, École Polytechnique Fédérale De Lausanne, October 2006.
- [13] TAN, X., AND TRIGGS, B. Enhanced local texture feature sets for face recognition under difficult lighting conditions. *IEEE Transactions on Image Processing* 19 (2010), 1635–650.
- [14] TURK, M., AND PENTLAND, A. Eigenfaces for recognition. *Journal of Cognitive Neuroscience* 3 (1991), 71–86.
- [15] VAPNIK, V. *Statistical Learning Theory*. John Wiley and Sons, New York, 1998.

- [16] WISKOTT, L., FELLOUS, J.-M., KRÜGER, N., AND MALSBURG, C. V. D. Face recognition by elastic bunch graph matching. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE* 19 (1997), 775–779.
- [17] ZHAO, W., CHELLAPPA, R., PHILLIPS, P., AND ROSENFELD, A. Face recognition: A literature survey. *Acm Computing Surveys (CSUR)* 35, 4 (2003), 399–458.