Seeing With OpenCV

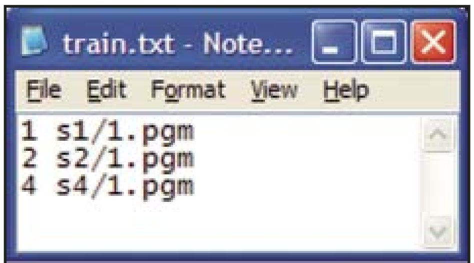
Implementing Eigenface

by Robin Hewitt

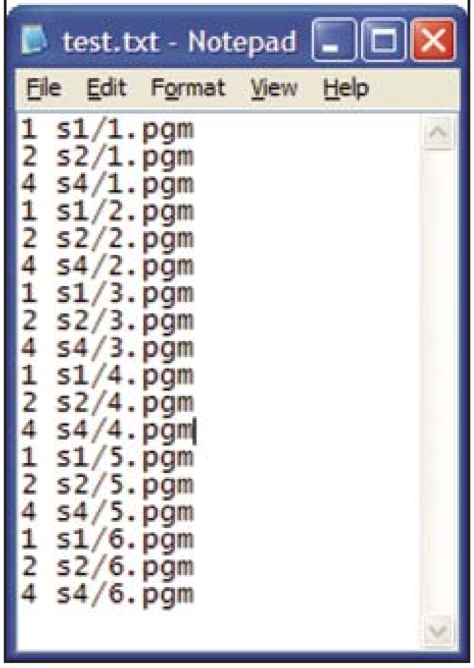
PART 5

Last month's article explained how the face recognition method called eigenface works. This month's article concludes both the OpenCV series and the eigenface topic with a detailed look at a complete program for implementing eigenface with OpenCV.

FIGURE 1. The two input files for the eigenface program: train.txt and test.txt. The paths in these input files point to images in the ORL face database.



```
1 s1/1.pgm
2 s2/1.pgm
4 s4/1.pgm
```



```
1 s1/1.pgm
2 s2/1.pgm
4 s4/1.pgm
1 s1/2.pgm
2 s2/2.pgm
4 s4/2.pgm
1 s1/3.pgm
2 s2/3.pgm
4 s4/3.pgm
1 s1/4.pgm
2 s2/4.pgm
4 s4/4.pgm
1 s1/5.pgm
2 s2/5.pgm
4 s4/5.pgm
1 s1/6.pgm
2 s2/6.pgm
4 s4/6.pgm
```

Here's a brief recap of last month's article explaining how eigenface works.

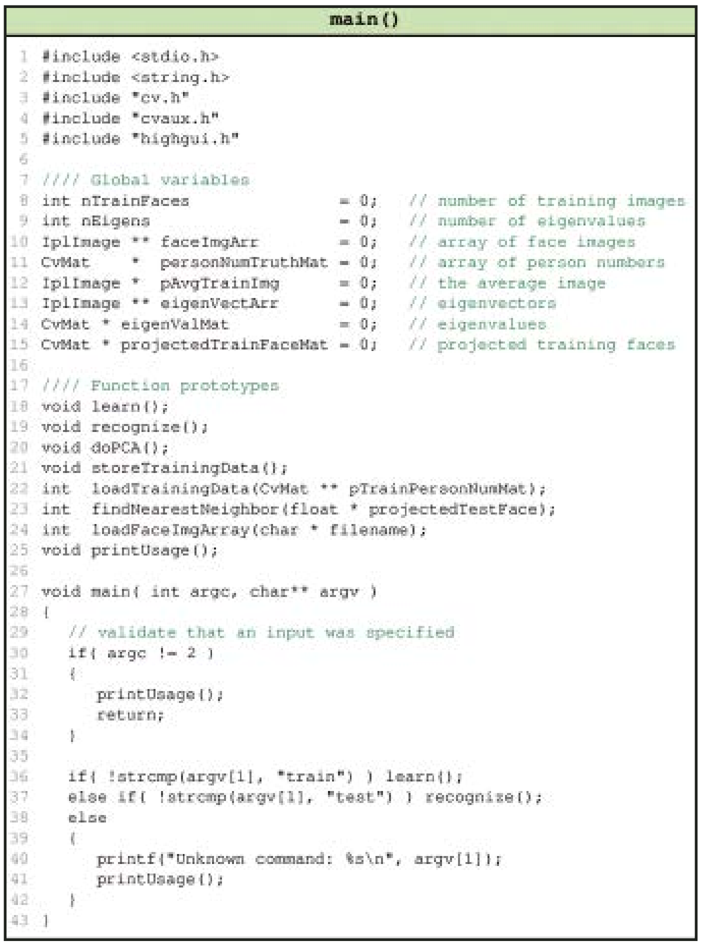
Eigenface consists of two phases: learning and recognition. In the learning phase, you give eigenface one or more face images for each person you want it to recognize. These images are called the training images. In the recognition phase, when you give eigenface a face image, it responds by telling you which training image is "closest" to the new face image.

Eigenface uses the training images to "learn" a face model. This face model is created by applying a method called Principal Components Analysis (PCA) to reduce the "dimensionality" of these images. Eigenface defines image dimensionality as the number of pixels in an image.

The lower dimensionality representation that eigenface finds during the learning phase is called a subspace. In the recognition phase, it reduces the dimensionality of the input image by "projecting" it onto the subspace it found during learning. "Projecting onto a subspace" means finding the closest point in that subspace. After the unknown face image has been projected, eigenface calculates the distance between it and each training image. Its output is a pointer to the closest training image. You can then look up which person eigenface identified.

Setting Up for Eigenface

In use, you'll probably want to combine eigenface with the face detection method presented in Part 2 of this series. To simplify the example code for this article, however, I'll be assuming you already have a set of training images and a set



```
#include <stdio.h>
#include <string.h>
#include "cv.h"
#include "cvaux.h"
#include "highgui.h"

//// Global variables
int nTrainFaces                 = 0;   // number of training images
int nEigens                     = 0;   // number of eigenvalues
IplImage ** faceImgArr          = 0;   // array of face images
CvMat    *  personNumTruthMat   = 0;   // array of person numbers
IplImage *  pAvgTrainImg        = 0;   // the average image
IplImage ** eigenVectArr        = 0;   // eigenvectors
CvMat * eigenValMat             = 0;   // eigenvalues
CvMat * projectedTrainFaceMat   = 0;   // projected training faces

//// Function prototypes
void learn();
void recognize();
void doPCA();
void storeTrainingData();
int  loadTrainingData(CvMat ** pTrainPersonNumMat);
int  findNearestNeighbor(float * projectedTestFace);
int  loadFaceImgArray(char * filename);
void printUsage();

void main( int argc, char** argv )
{
   // validate that an input was specified
   if( argc != 2 )
   {
      printUsage();
      return;
   }

   if( !strcmp(argv[1], "train") ) learn();
   else if( !strcmp(argv[1], "test") ) recognize();
   else
   {
      printf("Unknown command: %s\n", argv[1]);
      printUsage();
   }
}
```

FIGURE 2. The top-level source listing for the eigenface program.

of test images. These face images must all be *exactly the same size*.

For the examples in this article, I've used a free, publicly available face database — the Olivetti Research Lab's (ORL) Face Database. The URL is listed in the References and Resources sidebar.

To set up for using eigenface, unzip the ORL database in the same directory you'll use to run eigenface. This database contains 10 face images for each of 40 subjects. These are organized into 40 directories, named s1-s40. Each directory contains 10 images, named 1.pgm-10.pgm. All the ORL images are already the same size — 92 x 112 pixels.

You'll also need two input files: train.txt and test.txt. Figure 1 shows an example of these input files. Both files use the same format: person number, whitespace, path to image file.

You may have noticed that the first three test images are the same as the training images. These are useful test cases, because eigenface should always give the right answer when you ask it to recognize one of its training images. If it doesn't, you know you have some debugging to do!

To run the learning phase of this eigenface program, enter

```
eigenface train
```

at the command prompt. To run the recognition phase, enter

```
eigenface test
```

Source Code

Figures 2-10 contain the complete source listing for a basic eigenface program. To keep the presentation simple, I've omitted most of the error checking.

The Top-Level Listing

Figure 2 shows the top-level source listing for eigenface — a program to learn and recognize faces with OpenCV's eigenface methods. At line 4, it includes cvaux.h. Until now, we've only included cv.h and highgui.h. But we'll use two specialized functions for face recognition, `cvCalcEigenObjects()` and `cvEigenDecomposite()`, that are defined in cvaux.h.

The primary variables for eigenface are defined at lines 8-15. One of the datatypes here, `CvMat`, is one we haven't used before. This is OpenCV's matrix datatype. A matrix contains a table of data, arranged as rows and columns.

If you only need to hold data temporarily within your program, an ordinary C-style array is usually a little easier to use than `CvMat`. But the `CvMat` datatype can be nice when you want to take advantage of OpenCV functions for working with matrix data. The ones we'll use are OpenCV's persistence functions. With these, you can store matrix data with a single line of code. Reading it back into your programs later is just as easy. Here, I've used a C array for two variables (`faceImgArr` and `eigenVectArr`) and `CvMat` for several others.

The `main()` function simply reads the input string, then calls either the `learn()` method or the `recognize()` method. Figure 3 shows the `printUsage()` helper function.

FIGURE 3. The `printUsage()` helper function.

printUsage()

```
void printUsage()
{
   printf("Usage: eigenface <command>\n",
          "  Valid commands are\n"
          "    train\n"
          "    test\n");
}
```

The Learning Phase

Figure 4 shows the `learn()` function, which implements the learning phase as four steps:

1) Load the training data (line 6).

2) Do PCA on it to find a subspace (line 16).

3) Project the training faces onto the PCA subspace (lines 20-29).

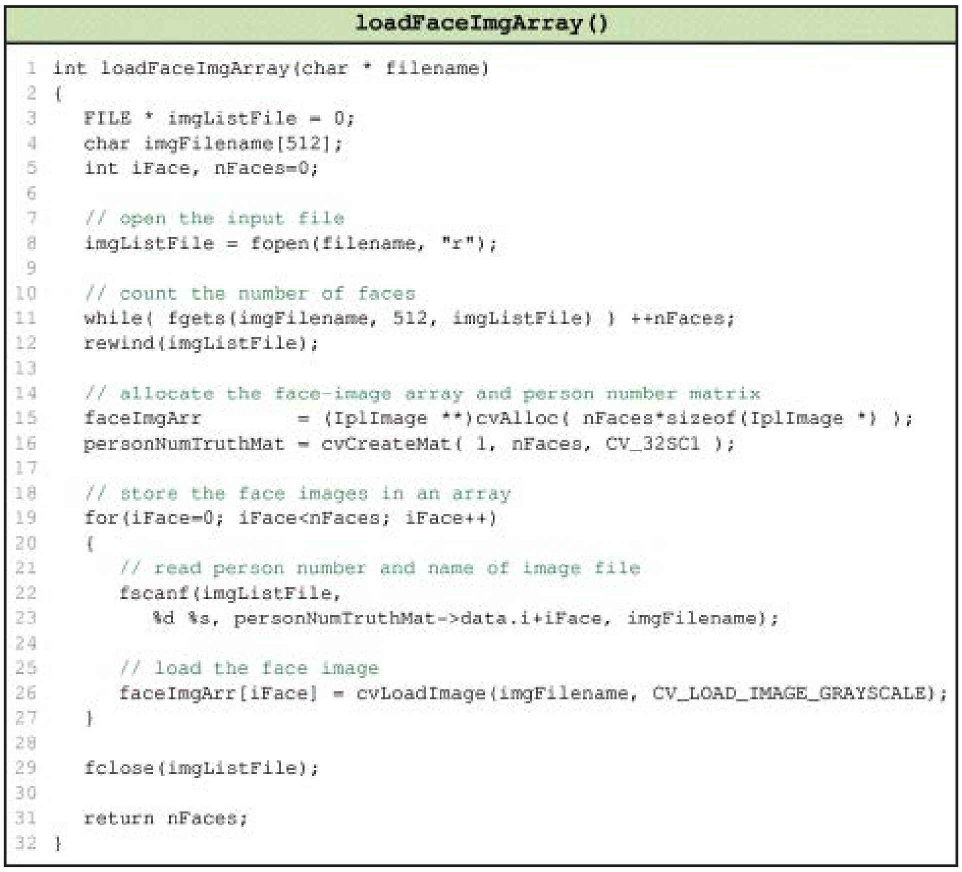
4) Save all the training information (line 32).
   a) Eigenvalues
   b) Eigenvectors
   c) The average training face image
   d) Projected faces
   e) Person ID numbers

The next four subsections analyze each

learn()

```
1  void learn()
2  {
3     int i;
4
5     // load training data
6     nTrainFaces = loadFaceImgArray("train.txt");
7     if( nTrainFaces < 2 )
8     {
9        fprintf(stderr,
10       "Need 2 or more training faces\n"
11       "Input file contains only %d\n", nTrainFaces);
12       return;
13    }
14
15    // do PCA on the training faces
16    doPCA();
17
18    // project the training images onto the PCA subspace
19    projectedTrainFaceMat = cvCreateMat(nTrainFaces, nEigens, CV_32FC1);
20    for(i=0; i<nTrainFaces; i++)
21    {
22       cvEigenDecomposite(
23          faceImgArr[i],
24          nEigens,
25          eigenVectArr,
26          0, 0,
27          pAvgTrainImg,
28          projectedTrainFaceMat->data.fl + i*nEigens);
29    }
30
31    // store the recognition data as an xml file
32    storeTrainingData();
33 }
```

FIGURE 4. The `learn()` function. This function implements the learning phase of eigenface.



```
loadFaceImgArray()

 1 int loadFaceImgArray(char * filename)
 2 {
 3    FILE * imgListFile = 0;
 4    char imgFilename[512];
 5    int iFace, nFaces=0;
 6
 7    // open the input file
 8    imgListFile = fopen(filename, "r");
 9
10    // count the number of faces
11    while( fgets(imgFilename, 512, imgListFile) ) ++nFaces;
12    rewind(imgListFile);
13
14    // allocate the face-image array and person number matrix
15    faceImgArr        = (IplImage **)cvAlloc( nFaces*sizeof(IplImage *) );
16    personNumTruthMat = cvCreateMat( 1, nFaces, CV_32SC1 );
17
18    // store the face images in an array
19    for(iFace=0; iFace<nFaces; iFace++)
20    {
21       // read person number and name of image file
22       fscanf(imgListFile,
23          %d %s, personNumTruthMat->data.i+iFace, imgFilename);
24
25       // load the face image
26       faceImgArr[iFace] = cvLoadImage(imgFilename, CV_LOAD_IMAGE_GRAYSCALE);
27    }
28
29    fclose(imgListFile);
30
31    return nFaces;
32 }
```

FIGURE 5. The `loadFaceImgArray()` function loads face images and person ID numbers for both the learning and recognition phases.

of these steps in detail.

Loading Face Images for Training or Test

The `loadFaceImgArray()` function (Figure 5) loads face images and person ID numbers for both the learning and recognition phases.

The face images — assumed here to be all the same size — are stored in the global variable `faceImgArr`; `loadFaceImgArray()` returns the number of face images loaded.

The person ID numbers are stored in a `CvMat` variable, `personNumTruthMat`. "Truth" here refers to the AI term, "ground truth." It means the values in this variable are the true (correct) values for each face image. During the learning phase, those are the only type of person ID numbers we have. But during the recognition phase, the program will have both ground truth values (specified in the file test.txt) and the output from eigenface. Having both allows us to evaluate how well eigenface does under varying conditions.

The `cvCreateMat()` function — called at line 16 — creates the `personNumTruthMat` variable. This function takes three parameters: the number of rows, the number of columns, and the datatype for the matrix. On a 32-bit operating system, the datatype for a matrix of int values is `CV_32SC1`. The `S` here stands for "signed," and `C1` indicates that the matrix has one channel. (A matrix can have up to four channels. Multiple channels allow you to add a third dimension to a matrix variable.)

The `CvMat` datatype is a `struct`, with the raw data stored in the `struct` element `data`; `data` is defined as a `union` (the definition is given in the CXCORE documentation), with `int` data accessed as `data.i`. Lines 22-23 show one way to access `CvMat` values — as offsets from the start of the data buffer.

Matrix rows are aligned to start on four-byte intervals. The in-memory row width, in bytes, is stored in `CvMat.step`. Since we're using a four-byte datatype with this matrix (and also, since it has only one row), we can ignore `CvMat.step`. But, if you create a matrix of, for example, char data, you may need to take the step size into account when you access the data elements.

Finding the PCA Subspace

The code to find the PCA subspace is in Figure 6. It calls the built-in OpenCV function for doing PCA, `cvCalcEigenObjects()`, at lines 27-36. The remainder of `doPCA()` creates the output variables that will hold the PCA results when `cvCalcEigenObjects()` returns.

At line 8, the number of eigenvalues is set to one less than the number of training images. (As explained last month, this is the maximum number of eigenvalues we can find.)

Lines 11-15 create the global image array `eigenVectArr`. When `cvCalcEigenObjects()` returns, each image in this array will hold one eigenvector, one "eigenface," in other words. Note that these are floating-point images, with data depth = `IPL_DEPTH_32F`.

At line 18, another matrix is created — `eigenValMat`. This matrix will hold the eigenvalues. The eigenvalues are floating-point numbers, and we only need one channel for this, so the matrix type is `CV_32FC1`. That gives us a one-channel matrix, with 32-bit, floating-point data values.

To do PCA, the dataset must first be "centered." For our face images, this means finding the average image — an image in which each pixel contains the average value for that pixel across all face images in the training set. The dataset is centered by subtracting the average face's pixel values from each training image.

You don't have to do that yourself. It happens inside `cvCalcEigenObjects()`. But you do need to hold onto the average image, because you'll need it later to project the data. So you'll need to allocate memory for the average image. The code for doing that is at line 21. Note that — like the eigenvectors — this is a floating-point image.

The last step before calling `cvCalcEigenObjects()` is to prepare a data structure called `CvTermCriteria`. The fields in this structure specify termination criteria for iterative algorithms such as PCA. You can read more about `CvTermCriteria` options in the CXCORE documentation. Here, we can simply tell it to compute each eigenvalue, then stop, since that's all we need.

FIGURE 6. doPCA() finds the PCA subspace using OpenCV's cvCalcEigenObjects() function.

```
doPCA()

 1 void doPCA()
 2 {
 3    int i;
 4    CvTermCriteria calcLimit;
 5    CvSize faceImgSize;
 6
 7    // set the number of eigenvalues to use
 8    nEigens = nTrainFaces-1;
 9
10    // allocate the eigenvector images
11    faceImgSize.width  = faceImgArr[0]->width;
12    faceImgSize.height = faceImgArr[0]->height;
13    eigenVectArr = (IplImage**)cvAlloc(sizeof(IplImage*) * nEigens);
14    for(i=0; i<nEigens; i++)
15       eigenVectArr[i] = cvCreateImage(faceImgSize, IPL_DEPTH_32F, 1);
16
17    // allocate the eigenvalue array
18    eigenValMat = cvCreateMat( 1, nEigens, CV_32FC1 );
19
20    // allocate the averaged image
21    pAvgTrainImg = cvCreateImage(faceImgSize, IPL_DEPTH_32F, 1);
22
23    // set the PCA termination criterion
24    calcLimit = cvTermCriteria( CV_TERMCRIT_ITER, nEigens, 1);
25
26    // compute average image, eigenvalues, and eigenvectors
27    cvCalcEigenObjects(
28       nTrainFaces,
29       (void*)faceImgArr,
30       (void*)eigenVectArr,
31       CV_EIGOBJ_NO_CALLBACK,
32       0,
33       0,
34       &calcLimit,
35       pAvgTrainImg,
36       eigenValMat->data.fl);
37 }
```

The code for that is at line 24.

Now that all the output variables are ready, we call cvCalcEigenObjects() to compute the PCA subspace for the training faces. The last parameter, eigenValMat->data.fl, is the pointer to the data values in eigenValMat. Here, we use the data.fl field, not data.i, since this matrix variable holds floating-point data.

Projecting the Training Faces

Now that you've found a subspace using PCA, you can convert the training images to points in this subspace. As explained last month, this step is called "projecting" the training image. The OpenCV function for this step is called cvEigenDecomposite() (Figure 4, line 22).

The OpenCV function names are, unfortunately, confusing. Not only is the projection function oddly named, but there's also a function named "EigenProjection" that doesn't project image data onto the subspace. In fact, it does the opposite. It restores (uncompresses) projected data, turning it back into the original image. The correct name for doing that is Reconstruction, not Projection!

You'll need a place to put the projected training images. Line 19 in Figure 4 creates a matrix for that purpose. The for loop, at lines 20-29, calls cvEigenDecomposite() once for each training image.

Saving the Learned Face Model

The small bit of extra effort to use OpenCV's CvMat datatype really pays off when it comes time to save the training data! Figure 7 shows the complete code for saving all the data for your learned face representation as an XML file using OpenCV's built-in persistence functions.

At line 7, the call to cvOpenFileStorage() opens an XML file named facedata.xml. The last parameter to this function controls the access mode. Here, it's CV_STORAGE_WRITE, which means to create (or overwrite) that file and open it for writing.

To write basic C-language data — integers, floating-point values, and strings — in XML format, you can use the function cvWrite<datatype>(). For example, the call to cvWriteInt() at line 10 writes the number of eigenvalues as <nEigens>2</nEigens>.

The really nice thing about using OpenCV's persistence functions is that it's just as easy to save complex datatypes, such as an image or matrix.

FIGURE 7. storeTrainingData() saves all the data for the learned face representation as an XML file using OpenCV's built-in persistence functions.

```
storeTrainingData()

 1 void storeTrainingData()
 2 {
 3    CvFileStorage * fileStorage;
 4    int i;
 5
 6    // create a file-storage interface
 7    fileStorage = cvOpenFileStorage( "facedata.xml", 0, CV_STORAGE_WRITE );
 8
 9    // store all the data
10    cvWriteInt( fileStorage, "nEigens", nEigens );
11    cvWriteInt( fileStorage, "nTrainFaces", nTrainFaces );
12    cvWrite(fileStorage, "trainPersonNumMat", personNumTruthMat, cvAttrList(0,0));
13    cvWrite(fileStorage, "eigenValMat", eigenValMat, cvAttrList(0,0));
14    cvWrite(fileStorage, "projectedTrainFaceMat", projectedTrainFaceMat, cvAttrList(0,0));
15    cvWrite(fileStorage, "avgTrainImg", pAvgTrainImg, cvAttrList(0,0));
16    for(i=0; i<nEigens; i++)
17    {
18       char varname[200];
19       sprintf( varname, "eigenVect_%d", i );
20       cvWrite(fileStorage, varname, eigenVectArr[i], cvAttrList(0,0));
21    }
22
23    // release the file-storage interface
24    cvReleaseFileStorage( &fileStorage );
25 }
```

```
recognize()

 1 void recognize()
 2 {
 3    int i, nTestFaces  = 0;         // the number of test images
 4    CvMat * trainPersonNumMat = 0;  // the person numbers during training
 5    float * projectedTestFace = 0;
 6
 7    // load test images and ground truth for person number
 8    nTestFaces = loadFaceImgArray("test.txt");
 9    printf("%d test faces loaded\n", nTestFaces);
10
11    // load the saved training data
12    if( !loadTrainingData( &trainPersonNumMat ) ) return;
13
14    // project the test images onto the PCA subspace
15    projectedTestFace = (float *)cvAlloc( nEigens*sizeof(float) );
16    for(i=0; i<nTestFaces; i++)
17    {
18       int iNearest, nearest, truth;
19
20       // project the test image onto the PCA subspace
21       cvEigenDecomposite(
22          faceImgArr[i],
23          nEigens,
24          eigenVectArr,
25          0, 0,
26          pAvgTrainImg,
27          projectedTestFace);
28
29       iNearest = findNearestNeighbor(projectedTestFace);
30       truth    = personNumTruthMat->data.i[i];
31       nearest  = trainPersonNumMat->data.i[iNearest];
32
33       printf("nearest = %d, Truth = %d\n", nearest, truth);
34    }
35 }
```

FIGURE 8. The `recognize()` function implements the recognition phase of the eigenface program.

Lines 12-15 add three matrices and an image to the same XML file. The built-in persistence functions save not only the row and column data, but all the header information, as well. Here's the XML that line 13 generates:

```
<eigenValMat type_id="opencv-
matrix">
 <rows>1</rows>
 <cols>2</cols>
 <dt>f</dt>
 <data>
 14279064. 9614034.</data></eigenVal
Mat>.
```

The second parameter to the `Write()` functions is a string. The string can be anything you like, but to ensure uniqueness — and for clarity's sake — it's usually a good idea to make it the same as your variable name.

When you've finished writing data, close the file and release the file storage as in line 24.

The Recognition Phase

Figure 8 shows the `recognize()` function, which implements the recognition phase of the eigenface program. It has just three steps. Two of them – loading the face images and projecting them onto the subspace — are already familiar.

As described above, the face images for recognition testing should be listed in a file named test.txt, using the same format as in train.txt. At line 8, the call to `loadFaceImgArray()` loads these into the `faceImgArr` and stores the ground truth for the person ID number in `personNumTruthMat`. This step is similar to line 6 of the `learn()` function in Figure 4. Here, the number of face images is stored in the local variable, `nTestFaces`.

We also need to load the global variable `nTrainFaces`, as well as most of the other training data — `nEigens`, `EigenVectArr`, `pAvgTrainImg`, and so on. The function `loadTrainingData()` in Figure 9 does that for us. Again, OpenCV's persistence functions make this step easy. To open file storage for reading, use the `CV_STORAGE_READ` flag. Then, simply call the appropriate `Read()` function for each variable. OpenCV locates and loads each data value in the XML file by name. When the variable is a `CvMat` type, OpenCV creates a new matrix for you automatically, then sets its data values.

The last parameter in the `Read()` function's interface is a default value. If a named variable is missing from the XML file, it will be set to the default. For pointer types — such as the matrices — it's a good idea to set the default to 0. You can then add a

```
loadTrainingData()

int loadTrainingData(CvMat ** pTrainPersonNumMat)
{
   CvFileStorage * fileStorage;
   int i;

   // create a file-storage interface
   fileStorage = cvOpenFileStorage( "facedata.xml", 0, CV_STORAGE_READ );
   if( !fileStorage )
   {
      fprintf(stderr, "Can't open facedata.xml\n");
      return 0;
   }

   nEigens = cvReadIntByName(fileStorage, 0, "nEigens", 0);
   nTrainFaces = cvReadIntByName(fileStorage, 0, "nTrainFaces", 0);
   *pTrainPersonNumMat = (CvMat *)cvReadByName(fileStorage, 0, "trainPersonNumMat", 0);
   eigenValMat  = (CvMat *)cvReadByName(fileStorage, 0, "eigenValMat", 0);
   projectedTrainFaceMat =
      (CvMat *)cvReadByName(fileStorage, 0, "projectedTrainFaceMat", 0);
   pAvgTrainImg = (IplImage *)cvReadByName(fileStorage, 0, "avgTrainImg", 0);
   eigenVectArr = (IplImage **)cvAlloc(nTrainFaces*sizeof(IplImage *));
   for(i=0; i<nEigens; i++)
   {
      char varname[200];
      sprintf( varname, "eigenVect_%d", i );
      eigenVectArr[i] = (IplImage *)cvReadByName(fileStorage, 0, varname, 0);
   }

   // release the file-storage interface
   cvReleaseFileStorage( &fileStorage );

   return 1;
}
```

FIGURE 9. OpenCV's persistence functions make it easy to load the saved training data from the XML file.

FIGURE 10. The findNearestNeighbor() function computes the distance from the projected test image to each projected training example to find the closest training image.

```
findNearestNeighbor()

1  int findNearestNeighbor(float * projectedTestFace)
2  {
3     double leastDistSq = DBL_MAX;
4     int i, iTrain, iNearest = 0;
5
6     for(iTrain=0; iTrain<nTrainFaces; iTrain++)
7     {
8        double distSq=0;
9
10       for(i=0; i<nEigens; i++)
11       {
12          float d_i =
13             projectedTestFace[i] -
14             projectedTrainFaceMat->data.fl[iTrain*nEigens + i];
15          distSq += d_i*d_i;
16       }
17
18       if(distSq < leastDistSq)
         {
19          leastDistSq = distSq;
20          iNearest = iTrain;
21       }
22    }
23
24    return iNearest;
25 }
```

validation check to make sure these pointers have a non-zero value before you use them. To simplify the example code, I've omitted these (and similar) validation steps from the loadTrainingData() function.

After all the data are loaded, the final step in the recognition phase is to project each test image onto the PCA subspace and locate the closest projected training image. The for loop at lines 16-34 of the recognize() function (Figure 8) implements this final step. The call to cvEigenDecomposite(), which projects the test image, is similar to the face-projection code in the learn() function.

As before, we pass it the number of eigenvalues (nEigens), and the array of eigenvectors (eigenVectArr). This time, however, we pass a test image, instead of a training image, as the first parameter. The output from cvEigenDecomposite() is stored in a local variable — projectedTestFace. Because there's no need to store the projected test image, I've used a C array for projectedTestFace, rather than an OpenCV matrix.

Finding the Nearest Neighbor

As last month's article explained, eigenface "recognizes" a face image by looking for the training image that's closest to it in the PCA subspace. Finding the closest training example in a learned subspace is a very common AI technique. It's called Nearest Neighbor matching.

Figure 10 shows the code for the findNearestNeighbor() function. It computes distance from the projected test image to each projected training example. The distance basis here is "Squared Euclidean Distance." As last month's column explained, to calculate Euclidean distance between two points, you'd add up the squared distance in each dimension, then take the square root of that sum. Here, we take the sum, but skip the square root step. The final result is the same, because the neighbor with the smallest distance also has the smallest squared distance, so we can save some computation time by comparing squared values.

The for loop at lines 6-22 computes the squared distance to each projected training image, and keeps track (at lines 18-21) of which training image is closest.

The return value is the index of the closest training image. In the recognize() function (Figure 8), this return value is used, at line 31, to look up the person ID number associated with the nearest training image.

Here is the print output from the recognize() function:

```
nearest = 1, Truth = 1
nearest = 2, Truth = 2
nearest = 4, Truth = 4
nearest = 1, Truth = 1
nearest = 2, Truth = 2
nearest = 4, Truth = 4
nearest = 1, Truth = 1
nearest = 2, Truth = 2
nearest = 4, Truth = 4
nearest = 1, Truth = 1
nearest = 2, Truth = 2
nearest = 4, Truth = 4
nearest = 1, Truth = 1
nearest = 2, Truth = 2
nearest = 4, Truth = 4
nearest = 1, Truth = 1
nearest = 2, Truth = 2
nearest = 1, Truth = 4
```

Not bad! We only have one mismatch: the last test image was misrecognized as Subject 1 instead of 4.

Improving Eigenface

Having a framework like this for training and testing will make it easier for you to add improvements to eigenface and to test their effects.

One of the first improvements you might want to add is to change the way distance is measured. The original eigenface paper used Euclidean distances between points, and that's the distance basis I've used in findNearestNeighbors(). But a different basis, called Mahalanobis distance (after its inventor), usually gives better results.

One of the things that happens when you project a face image onto the PCA subspace is that each dimension receives a certain amount of stretch. The amount of stretch isn't the same, though, in every direction. The directions that correspond to the largest eigenvalues get stretched far more than the directions associated with smaller eigenvalues. Because Euclidean distance ignores this stretching, using it to measure distance is approximately the same as using only one eigenvector and ignoring the rest!

It's easy to switch from Euclidean to Mahalanobis distance. Just change

References and Resources

- *OpenCV on Sourceforge*
  http://sourceforge.net/projects/opencvlibrary

- *Official OpenCV Usergroup*
  http://tech.groups.yahoo.com/group/OpenCV

- Turk, M., Pentland, A., *Face recognition using eigenfaces*, Proc. IEEE Conf. on Computer Vision and Pattern Recognition, 1991.

- *ORL Database*
  www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html

- Source code in this article can be downloaded from **www.cognotics.com/opencv/servo**.

line 15 in findNearestNeighbors() from

```
distSq += d_i*d_i;
```

to

```
distSq += d_i*d_i/eigenValMat->data.fl[i];
```

Switching to Mahalanobis distance eliminates the mismatch error mentioned above, bringing recognition accuracy up to 100% for these three subjects.

Where to Go From Here?

This article introduced several new OpenCV concepts. You can gain a deeper understanding of these from the OpenCV documentation. The persistence functions, the `CvTermCriteria struct`, and the `CvMat` datatype are described in detail in the CXCORE documentation. The eigenface functions are described in the CVAUX documentation. The CVAUX documentation isn't linked from the documentation index page, but you can find it in the documentation subdirectory named ref.

If you want to incorporate eigenface into a system that detects faces in live video, you'll first need to detect the face, then extract it into a separate image. Since each face image must be exactly the same size, the easiest way to do that is to define a standard size, say 50 x 50 pixels, ahead of time. Then, when you detect a face, you can use code like this to extract and resize it:

```
CvRect * pFaceRect = (CvRect*)cvGetSeqElem(pRectSeq, 0);
cvSetImageROI(pImg, *pFaceRect);
IplImage * pFaceImg =
   cvCreateImage( STD_SIZE, IPL_DEPTH_8U, 1 );
cvResize(pImg, pFaceImg, CV_INTER_AREA );
```

There are more capabilities built into OpenCV, and many, many more computer vision programs one can create using this library. I hope this short series of articles has given you a taste of what's possible with OpenCV, and perhaps motivated you to explore more of its capabilities.

Be seeing you! **SV**

STEER WINNING ROBOTS WITHOUT SERVOS!

Perform proportional speed, direction, and steering with only two Radio/Control channels for vehicles using two separate brush-type electric motors mounted right and left with our **mixing RDFR dual speed control**. Used in many successful competitive robots. Single joystick operation: up goes straight ahead, down is reverse. Pure right or left twirls vehicle as motors turn opposite directions. In between stick positions completely proportional. Plugs in like a servo to your Futaba, JR, Hitec, or similar radio. Compatible with gyro steering stabilization. Various volt and amp sizes available. The RDFR47E 55V 75A per motor unit pictured above.
www.vantec.com

VANTEC

Order at (888) 929-5055

Get 5 PCBs for $13 each in 5 days!

See our website for other offers on 2, 4, 6 layer boards.

"$13 each" offer is valid for these specs:

- 2 sided boards up to 20 sq" size
- FR-4, 0.062" thick
- 7 mil trace/space
- 15 mil min. finished hole size
- As many drill sizes as you want
- Free Tooling, Soldermask, Silkscreen

PCB Assembly Available!

Order Online at:
www.pcbfabexpress.com

PCB FAB EXPRESS
High Quality PCBs @ Low Impact Prices
email: **support@pcbfabexpress.com**
Phone: **408.522.1500**