University of Nevada, Reno

CS 485 — Computer Vision

Assignment #3

Joshua Gleason

Instructor:
Dr. George Bebis

March 24, 2011

Contents

1 Introduction

This program demonstrates a method for normalizing a group of face images with hand labeled features. This normalization method attempts to map five points to a set of five predefined locations using an Affine transform. Because each point consists of two parts (x and y) a total of 10 equations are known while the Affine transformation matrix only has 6 parameters. This leads to an over-determined system which very likely has no solution. By using a method known as Singular Value Decomposition(SVD), the solution which yields the least Mean Square Error(MSE) can be computed efficiently.

A method for correcting non-uniform illumination in the resulting face images was also implemented. This method involves solving for the coefficients of Eq. 1 with respect to every pixel in the image. This means for an NxM image, there are 4 unknown and NM equations. Because this system is also over-determined, SVD was used to calculate the least MSE coefficients.

$$f(x, y) = ax + by + cxy + d \tag{1}$$

2 Results

The results of the normalization process were very favorable (Fig. 1). While the points in the images could not be mapped exactly to the desired locations, Fig. 2 shows that the features were all very close. The white circles have a radius of 5 pixels and the black circles have a radius of 4 pixels. Fig. 2 shows that no feature was more than 9 pixels away from its desired position and in most cases the features were no more than 3 pixels away.

The light correction function calculated using SVD can be seen in Fig. 3. The final results of the light correction can be seen in Fig. 4. While this method seemed to work well in most cases, it seemed to have trouble when a large amount of dark background was in an image. The black background seemed to overpower the function and cause it to try and lighten this area while darkening bright area causing an odd shadowing effect.

3 Conclusion

By implementing these methods I learned a great deal about using SVD for solving over-determined systems. I also learned a great deal about Affine transforms and how to map the pixels using an inverse to ensure no holes are left in the transformed image.

Originally I was confused about why an iterative method is used here rather than directly mapping each image to the final locations. The reason I came up with is that this iterative method not only aligns the images to the final location but also to each other. By using this iterative method even if the final location points are chosen poorly, all the faces should be mapped to approximately the same locations.

The light correction part was also an interesting concept, it shows how a relatively simple function like Eq. 1 can be used to correct major intensity inconsistencies in an image.

4 Images

Figure 1: Results of face normalization.

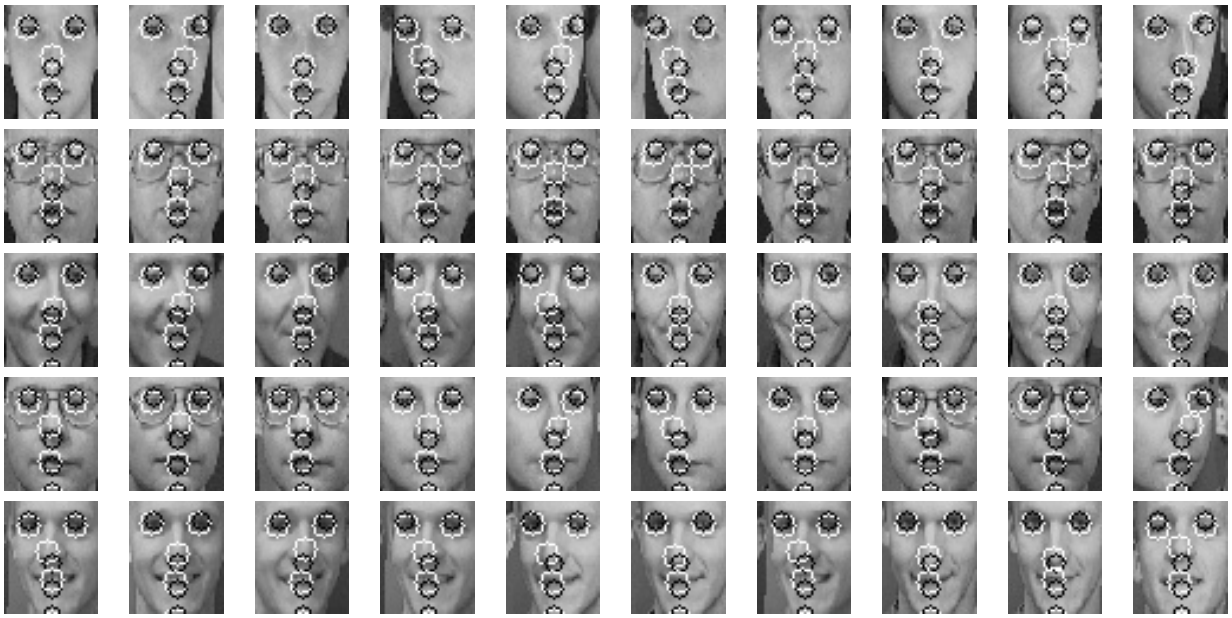

Figure 2: Black circles are desired locations, white circles are actual locations.

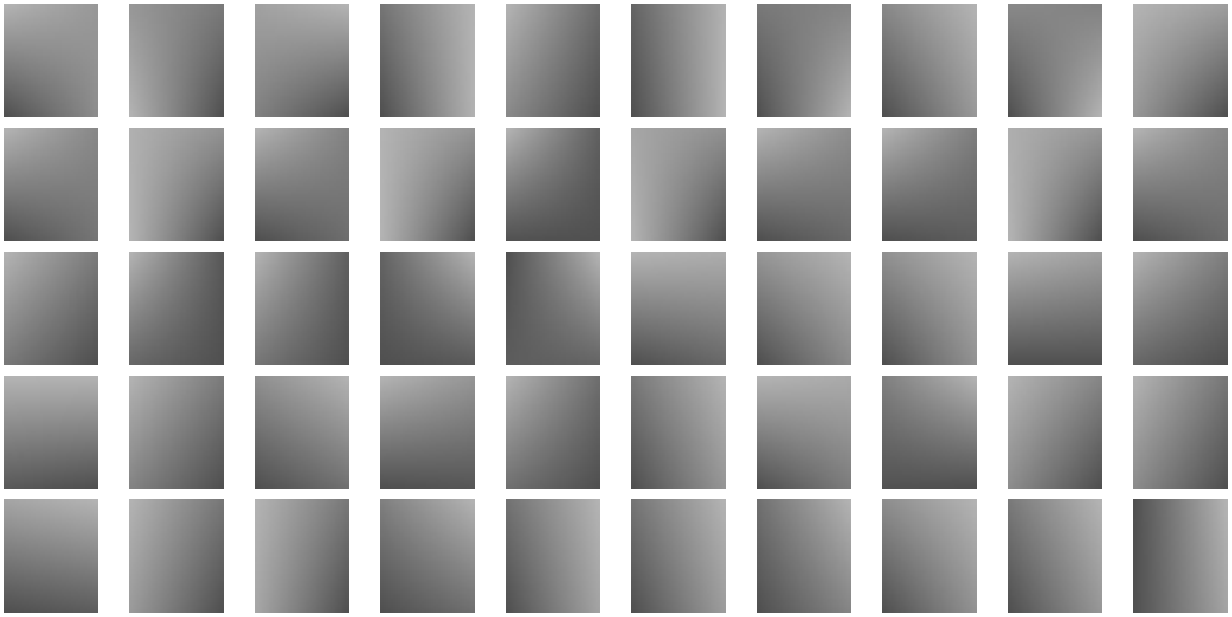

Figure 3: Light normalization functions.

Figure 4: Light normalization face images.

5 Source Code

5.1 funcs.h

```
using namespace cv;

struct Face
{
  Face( Point lEye, Point rEye, Point nose, Point mouth, Point chin, string _fname ) :
    F(5,2,CV_32FC1), fname(_fname)
  {
    F.at<float>(0,0) = lEye.x;
    F.at<float>(0,1) = lEye.y;
    F.at<float>(1,0) = rEye.x;
    F.at<float>(1,1) = rEye.y;
    F.at<float>(2,0) = nose.x;
    F.at<float>(2,1) = nose.y;
    F.at<float>(3,0) = mouth.x;
    F.at<float>(3,1) = mouth.y;
    F.at<float>(4,0) = chin.x;
    F.at<float>(4,1) = chin.y;
  }

  string getFilename() const
  {
    return fname;
  }

  Point getLeftEye() const
  {
    return Point(F.at<float>(0,0),F.at<float>(0,1));
  }

  Point getRightEye() const
  {
    return Point(F.at<float>(1,0),F.at<float>(1,1));
  }

  Point getNose() const
  {
    return Point(F.at<float>(2,0),F.at<float>(2,1));
  }

  Point getMouth() const
  {
    return Point(F.at<float>(3,0),F.at<float>(3,1));
  }

  Point getChin() const
  {
    return Point(F.at<float>(4,0),F.at<float>(4,1));
  }

  void setFilename( string _fname )
  {
    fname = _fname;
  }

  void setLeftEye( Point p )
  {
    F.at<float>(0,0)=p.x;
    F.at<float>(0,1)=p.y;
  }

  void setRightEye( Point p )
  {
    F.at<float>(1,0)=p.x;
    F.at<float>(1,1)=p.y;
  }

  void setNose( Point p )
```

```
  {
    F.at<float>(2,0)=p.x;
    F.at<float>(2,1)=p.y;
  }

  void setMouth( Point p )
  {
    F.at<float>(3,0)=p.x;
    F.at<float>(3,1)=p.y;
  }

  void setChin( Point p )
  {
    F.at<float>(4,0)=p.x;
    F.at<float>(4,1)=p.y;
  }

  Mat F;
  string fname;
};
```

5.2 main.cc

```
#include <cv.h>
#include <highgui.h>
#include <cvaux.h>
#include <iostream>
#include <fstream>
#include <vector>
#include <iomanip>
#include "face.h"

using namespace std;
using namespace cv;

vector<Face> readFeatures(string fname)
{
  ifstream fin(fname.c_str());

  vector<Face> faces;

  if ( !fin.good() )
    return faces;

  float x[5],y[5];
  string imageloc;

  while (fin.good())
  {
    for ( int i = 0; i < 5; i++ )
      fin >> x[i] >> y[i];
    fin >> imageloc;
    if ( fin.good() )
      faces.push_back(
        Face(Point(x[0], y[0]),
             Point(x[1], y[1]),
             Point(x[2], y[2]),
             Point(x[3], y[3]),
             Point(x[4], y[4]),
             imageloc));

  }
}

Mat readFinalLocations(string fname)
{
  ifstream fin(fname.c_str());

  Mat F(5,2,CV_32FC1,Scalar(-1));

  if ( !fin.good() )
    return F;
```

```
  fin >> F.at<float>(0,0)
      >> F.at<float>(0,1)
      >> F.at<float>(1,0)
      >> F.at<float>(1,1)
      >> F.at<float>(2,0)
      >> F.at<float>(2,1)
      >> F.at<float>(3,0)
      >> F.at<float>(3,1)
      >> F.at<float>(4,0)
      >> F.at<float>(4,1);

  return F;
}

Mat getPMatrix(Face& face)
{
  Mat ret = Mat(5,3,CV_32FC1);

  for ( int i = 0; i < 5; i++ )
  {
    for ( int j = 0; j < 2; j++ )
      ret.at<float>(i,j)=face.F.at<float>(i,j);
    ret.at<float>(i,2)=1.0;
  }

  return ret;
}

// c1 = [a11,a12,b1]^T
// c2 = [a21,a22,b2]^T
Mat buildInvAffine(Mat& c1, Mat& c2)
{
  float a11 = c1.at<float>(0,0),
        a12 = c1.at<float>(1,0),
        b1 = c1.at<float>(2,0),
        a21 = c2.at<float>(0,0),
        a22 = c2.at<float>(1,0),
        b2 = c2.at<float>(2,0);
  float mult = 1.0/(a11*a22-a21*a12);
  Mat inv = Mat(3,3,CV_32FC1);


  inv.at<float>(0,0) = a22*mult;
  inv.at<float>(0,1) = -a21*mult;
  inv.at<float>(0,2) = 0;
  inv.at<float>(1,0) = -a12*mult;
  inv.at<float>(1,1) = a11*mult;
  inv.at<float>(1,2) = 0;
  inv.at<float>(2,0) = (a12*b2-a22*b1)*mult;
  inv.at<float>(2,1) = (a21*b1-a11*b2)*mult;
  inv.at<float>(2,2) = (a11*a22-a21*a12)*mult;

  return inv;
}

void applyTrans( const Mat& img, Mat &newImg, const Mat& T )
{
  Mat point = Mat(1,3,CV_32FC1), current = Mat(1,3,CV_32FC1);
  current.at<float>(0,2) = 1.0;
  float newR, newC;
  for ( int r = 0; r < newImg.rows; r++ )
    for ( int c = 0; c < newImg.cols; c++ )
    {
      current.at<float>(0,0) = (float)c;
      current.at<float>(0,1) = (float)r;
      point = current*T;
      newC = point.at<float>(0,0);
      newR = point.at<float>(0,1);

      //if ( (int)newR >= img.rows || (int)newR < 0 ||
      //     (int)newC >= img.cols || (int)newC < 0 )
      //
```

```
      //
      //  newImg.at<uchar>(r,c) = img.at<uchar>(
      //  ((int)newR<0 ? 0 : ((int)newR>img.rows ? img.rows-1 : (int)newR )),
      //  ((int)newC<0 ? 0 : ((int)newC>img.cols ? img.cols-1 : (int)newC )));
      //else
        newImg.at<uchar>(r,c) = img.at<uchar>((int)newR,(int)newC);
    }
}

float getDiff(const Mat& F1, const Mat& F2)
{
  Mat diff = F1-F2;
  return (float)(norm(diff));
}

int main(int argc, char *argv[])
{
  cout << "Make sure the following directories exist" << endl
       << "./results/" << endl << "./results/circles/" << endl
       << "./results/light/" << endl << "./results/light/faces/" << endl;

  // check arguments
  if ( argc < 4 )
  {
    cout << "Please enter the filename of the faces feature locations followed by"
         << "the filename containing the final face locations and a threshold" << endl;
    return -1;
  }

  float thresh = atof(argv[3]);

  // initialize variables
  vector<Face> faces = readFeatures(argv[1]), faces2 = readFeatures(argv[1]);
  Mat F_final = readFinalLocations(argv[2]);
  Mat c1, c2, T, P, F_bar, F_bar_prime, F_bar_prev, F_i_prime, F_tot, px, py;
  Mat img, newImg = Mat(48,40,CV_8UC1);

  // holds the seperate inverse affine transforms for each face
  vector<Mat> T_inv;

  // set all of them to initialize as Identity matrix
  for ( int i = 0; i < (int)faces.size(); i++ )
  {
    T_inv.push_back(Mat(3,3,CV_32FC1));
    setIdentity(T_inv[i]);
  }
  // initialize P as a 5x3 matrix
  P = Mat(5,3,CV_32FC1);

  // initialize F_bar to first image
  F_bar = faces[0].F.clone();

  do {
    // make copy of F_bar for later
    F_bar_prev = F_bar.clone();

    // set P
    for ( int i = 0; i < 5; i++ )
    {
      P.at<float>(i,0) = F_bar.at<float>(i,0);
      P.at<float>(i,1) = F_bar.at<float>(i,1);
      P.at<float>(i,2) = 1.0;
    }

    // set px and py
    px = F_final.col(0).clone();
    py = F_final.col(1).clone();

    // solve for transform
    solve(P,px,c1,DECOMP_SVD);
    solve(P,py,c2,DECOMP_SVD);

    // build transform matrix
```

```
  T = Mat(3,2,CV_32FC1);
  for ( int i = 0; i < 3; i++ )
  {
    T.at<float>(i,0) = c1.at<float>(i,0);
    T.at<float>(i,1) = c2.at<float>(i,0);
  }

  // apply on F_bar (P)
  F_bar_prime = P*T;

  // Update F_bar by setting equal to F_bar_prime
  F_bar = F_bar_prime.clone();

  // zero matrix
  F_tot = Mat(5,2,CV_32FC1,Scalar(0.0));

  // For every face image F_i use SVD to align F_i with F_bar to make F_i_prime
  for ( int index = 0; index < (int)faces.size(); index++ )
  {
    // set P
    P = getPMatrix(faces[index]);

    px = F_bar.col(0).clone();
    py = F_bar.col(1).clone();

    // perform SVD
    solve(P,px,c1,DECOMP_SVD);
    solve(P,py,c2,DECOMP_SVD);

    // build the inverse Affine
    T_inv[index] *= buildInvAffine(c1,c2);

    // build transform matrix
    for ( int i = 0; i < 3; i++ )
    {
      T.at<float>(i,0) = c1.at<float>(i,0);
      T.at<float>(i,1) = c2.at<float>(i,0);
    }

    // apply on F_i to get F_i_prime
    F_i_prime = P*T;

    // keep running total
    F_tot += F_i_prime;

    // set F_i to F_i_prime
    faces[index].F = F_i_prime.clone();
  }

  // update F by average the values of F_i for each image
  F_bar = F_tot/(int)faces.size();

} while ( getDiff(F_bar,F_bar_prev) > thresh );

// show results
for ( int index = 0; index < (int)faces.size(); index++ )
{
  // read image
  img = imread(faces[index].fname,0);

  // apply transform
  applyTrans(img,newImg,T_inv[index]);

  Mat circleImg = newImg.clone();

  // draw circles
  for ( int i = 0; i < 5; i++ )
  {
    circle(circleImg,Point(faces[index].F.at<float>(i,0),faces[index].F.at<float>(i,1)),
      5,Scalar(255,255,255),1);
    circle(circleImg,Point(F_final.at<float>(i,0),F_final.at<float>(i,1)),
      4,Scalar(0,0,0),1);
  }
```

```
    // write images with circles
    ostringstream sout;
    sout << "./results/circles/" << index << ".jpg";
    imwrite(sout.str(),circleImg);

    // write before light normalization
    sout.str("");
    sout << "./results/" << index << ".jpg";
    imwrite(sout.str(),newImg);

    // light normalization

    Mat B = Mat(newImg.rows*newImg.cols,1,CV_32FC1);
    Mat A = Mat(newImg.rows*newImg.cols,4,CV_32FC1);
    Mat X; // [a b c d]
    int i, v;
    int min, max;
    max = min = (int)newImg.at<uchar>(0,0);
    // normalize light
    for ( int y = 0; y < newImg.rows; y++ )
      for ( int x = 0; x < newImg.cols; x++ )
      {
        v = (int)newImg.at<uchar>(y,x);
        if ( max < v )
          max = v;
        if ( min > v )
          min = v;

        i = y*newImg.cols+x;
        B.at<float>(i,0) = (float)newImg.at<uchar>(y,x);
        A.at<float>(i,0) = x;
        A.at<float>(i,1) = y;
        A.at<float>(i,2) = x*y;
        A.at<float>(i,3) = 1;
      }
    solve(A,B,X,DECOMP_SVD);

    float a = X.at<float>(0,0);
    float b = X.at<float>(1,0);
    float c = X.at<float>(2,0);
    float d = X.at<float>(3,0);

    Mat model = Mat(newImg.rows, newImg.cols, CV_32FC1);

    for ( int y = 0; y < model.rows; y++ )
      for ( int x = 0; x < model.cols; x++ )
        model.at<float>(y,x) = a*x+b*y+c*x*y+d;

    Mat saveModel;
    normalize(model, saveModel, 80, 180, CV_MINMAX, CV_8UC1);

    Mat correct;
    newImg.convertTo(correct,CV_32FC1);

    correct -= model;

    normalize(correct, newImg, min, max, CV_MINMAX, CV_8UC1);

    // save
    sout.str("");
    sout << "./results/light/faces/" << index << ".jpg";
    imwrite(sout.str(),newImg);
    sout.str("");
    sout << "./results/light/" << index << ".jpg";
    imwrite(sout.str(),saveModel);
  }

  // end
  return 0;
}
```