Camera Calibration by Corner Detection

Evan Herbst

pd. 1

6 / 10 / 03

1 Abstract

The most annoying characteristic of calibrating digital measurement devices—and cameras in particular—is the time and effort required. Generally, cameras must be used to take several pictures of some set of orthogonal planes, fed some data from their packaging or otherwise helped along. This project aims to make calibrating a digital camera an easy task for the user by performing most calculations without input and using minimal input data in the form of pictures.

2 Introduction

Feature recognition in images is one way to calibrate a digital camera. The algorithm of which this project will make use, which was written by a senior researcher at Microsoft

Research, uses the measured locations of corresponding points in several images taken by a camera to find information about the camera itself. This project uses several prewritten modules to find features (in this case, easily detectable corners) in images, refine an estimate of their locations in 3-space, and calibrate the camera that took the images.

3 Procedure

The calibration process as a whole is comprised of several steps, each of which is at the moment encoded in its own routine. One goal of this project may eventually be to bring together all the steps into one massive calibration program.

3.1 Corner Detection

The first step is to take three or more (the more pictures are taken, the more accurate the calibration) pictures, with the camera being calibrated, of a model plane with easily detectable features (in this case, corners) from different angles. (It is still important to have taken a reasonable number of images to gather homography data; reducing the input needed is beyond the scope of this project.) The pictures are saved as images. Then a routine can search the pictures for the desired features (in this case, corners, which are useful because they are generally easy to find) and place their locations into a set of output files with corresponding coordinates listed. These files may need to be sorted to make sure their lists of points correspond.

This step is the reason this project can be fitted into the category of computer vision. Of

course the selection of algorithm for detecting feature points has an impact on the accuracy of the entire calibration. This project uses an algorithm modified from Harris' algorithm (elaborated on page 11 of [**??**]), which makes use of change in the image gradient over a window of a given size to estimate the position of a corner to (supposedly) an accuracy of less than a pixel.

The image gradient is the difference in value between the colors of adjacent pixels. For calibration, black and white images are most useful because comparisons need only be made in one dimension (gray). Corners are easily recognized as spots of very large image gradient, where one color gives way sharply to another. The biggest problem in finding corners in images at random is that some images may have sharp curves defined by lines but not intended to be the edges of objects, such as in the following picture:

3.2 Extrapolation

A series of corner detections is performed on each image taken, with the window size used to calculate the exact corner points varying. These results are then extrapolated down to an infinitesimal window size; in theory, this will produce the corner positions as exactly as they can be computed. In practice, a window size of one or three pixels squared seems to be better than zero for estimating positions. Since the coordinate adjuster does not require a certain window size, this doesn't matter too much, but it would be nice for the theory to fit experimental data–i.e., for the results given by the data from a window size of 0 to fit better than the results given by a window size of 1 pixel.

Different window sizes are used to calculate sets of corner points because the larger the window used, the more gradient data is given, but the smaller the window used, the more exact the analyzer can be. In addition, this type of routine should be able to be used on very small images that cannot support large window sizes. However, the smaller the window used, the less data is available, and at a window isze of about 1 pixel, there is *no* variation in image data, the gradient is useless and *no* routine can exactly find any corner based on zero data. This project does not attempt to answer the question of what window size is most useful in detecting corners.

4 Dehydrogenation

First I created a basic layout for my webpage. It has a JAVA menu on the left that has pull out tabs. This menu allows the user to login and access the authors through e-mail. The login link brings the user to a login page where they type in user name and password. The password is encoded and then searched for in the database. All of the passwords in the database are already encoded. Logging in brings up a diffrent JAVA menu that allows searching and retrieval of files from the MySQL database.

The search allows the user to look by filetype which is GIF, TIFF, or VRML. It also searches by filename and description. The description search takes a keyword from the user and looks for it in the descriptions of files in the database. The website can also be used to upload information from the user onto the database. This allows the databse to constantly expand as people add their files to it. An administrator would be expected to periodically

clean up the database to make sure it is organized. The last option allowed to the logged on user is the logout option. This clears the cookies so that someone cannot enter back onto the page from that computer without logging on again.

4.1 Coordinate Adjustment

The original point of this project concerned the possibility of adjusting the point coordinates given to the calibration routine. Since the points are probably not detected entirely correctly earlier in the process, it is likely that moving them slightly in some educated manner will improve the accuracy of the calibration. For instance, Mark has found from some informal tests that the error in point mapping after calibration by this method is related to the magnitude and direction of the image gradients at detected corner points; moving detected coordinates along the gradient might help fix that problem. Now that the calibrator has been written, it is the intent of this project to test various types of coordinate adjustments for efficacy in improving the accuracy of the resulting calibration. Although the image gradient is the most obvious tool for adjusting point positions, moving points a random distance in a random direction might also work as long as they are all moved the same way. However, the fact that they can only be moved in two directions and that their movements must somehow all be related are very limiting; there may be no perfect way to adjust them.

4.2 Homography

Homography is the geometrical relationship between two sets of points–in this case, sets of detected or measured corner points on planes in 3-space (see Figure [A]. Since planar homography is made up of linear relationships, the homographies this project finds can be represented by 3 x 3 matrices representing the transformations between sets of planes. This project finds the homographies between the 3-D plane of each image that was taken and the plane of the model points, which is (x, y, $z = 1$) and centered on (0, 0, 1), representing them as 3x3 transformation (rotation / translation) matrices of the form

$H = s * A * [R_1 R_2 | t]$

, where s is an arbitrary scalar and A is a matrix as specified below. R is the rotation transformation matrix between the two planes and t is the 3x1 translation vector.

Since a homography is made up of rotations and transformations from the model plane to an image plane, knowing the homography (based on relationships between individual pairs of points) tells us about the rotations and translations necessary to get from the origin to each image. Another part of the homography is the matrix containing information on the intrinsic camera parameters: the vertical and horizontal focal lengths of the camera, the amount of skew it introduces, and the point on any image, called the principal point of the camera, corresponding to the center of the camera lens. This matrix is known as the fundamental matrix of the camera, and is individual to the single camera being used.

4.3 Distortion

No camera lens is perfect. Through testing, it has been found that the most precise while still simple model of distortion introduced by a camera lens in general involves two parameters describing the radial distortion; tangenetial distortion is generally ignored. The two terms are defined by the equation

$\tilde{x} = x\{1 + [k_1(x^2 + y^2) + k_2(x^2 + y^2)^2]\}$

. k_1 is the stronger parameter. $\tilde{x}$ is a distorted point in an image (as captured by the camera), while x is the real point in the image plane. Later, x and y stand for individual coordinates in the point.

4.4 Optimization

Optimization is the process of minimzing a given function of an arbitrary number of parameters to achieve the best possible value for each parameter in the set to a high degree of accuracy. After all the parameters necessary for calibration have been estimated as above, the entire set is optimized based on the function given by the sum of ther errors between each detected point and the ideal (model) points given and homographized with the data above. While this ought to decrease the total error, it has been found during this project that optimization worsens the model greatly. This probably means that the cost function is not well-behaved, in which case there is not much that can be done about the problem. Thus the optimization has now been discarded; however, it is still useful in theory and useful to Zhang. The error is probably on the part of this project in using the

optimization incorrectly.

4.5 Error Checking

The last step in the loop is an error check. This routine calculates the errors, on the plane in which we have been working, between the points measured in Sec. 3.1 and the homographized points from the model file. The goal of this project is to have an accuracy such that the maximum error in 2-D is well under one pixel. If the result is not close to the desired value, the process continues with Sec. 3.1. At the moment, the maximum 2-D error is the number being used to decide how good a model is, rather than the average error.

5 Combination of Routines

At the moment, all of the above routines are written as separate programs. In future, I hope to combine them into one smaller and faster program. This will also allow me to add a layer of optimization around the last several steps (everything after the extrapolation) that uses the total or maximum pixel error given the parameters estimated as its cost function, and the positions of the corners as the optimized parameters in some way. In other words, the optimizer will change the positions of the detected corners slightly in some parameterized way (such as by a variable amount along the gradient) to try to minimize the error in final point locations. This step is just an extension of **Coordinate Adjustment** above.

6 Explanation of Terms

- ⋆ 2-D error – the distance, in the image plane, between the detected (real) image point and the model point that has been homographized and distorted by the computed model

- ⋆ 3-D error – the average distance, in three dimensions, between the detected (real) point and the ray projected by the camera toward the homographized and distorted model point

- ⋆ accuracy – the closeness of numerical results to actual data (*see precision*)

- ⋆ coordinate adjustment – actually moving the estimate of corner points that have been detected so that they make a nicer model of the homography

- ⋆ corner – a point in an image with a sharply defined image gradient (strong gradient value) (*see interest point*)

- ⋆ corner detection – searching images for corners and listing them in coordinated lists so that they can be analyzed

- ⋆ distortion – the movement of points from reality to an image due to an imperfect camera lens

- ⋆ extrapolation – interpretation of data for a result beyond the domain of previously obtained values

- $\star$ extrinsic parameters – parameters describing the relationships between the camera and the image planes

- $\star$ homography – the 3-D geometric relationship between two sets of points (in this case, points on planes)

- $\star$ image plane – the plane containing an image taken, not necessarily a simple plane like the model plane (*see image plane*)

- $\star$ interest point – any point in an image that is interesting for its image or gradient value (*see corner*)

- $\star$ intrinsic matrix – a matrix containing five parameters that describe the characteristics of a digital camera

- $\star$ intrinsic parameters – with reference to a digital camera, its focal length, skew and lens principal point

- $\star$ model plane – the plane containing the model points, measured by hand (in this case, the plane $z = 1$) (*see image plane*)

- $\star$ optimization – adjustment of a set of parameters for the minimization of a function of those parameters, called a "cost function"

- $\star$ parameter – any value that is changed by a function, such as an optimizer

- $\star$ precision – the ability to accurately reproduce results (*see accuracy*)

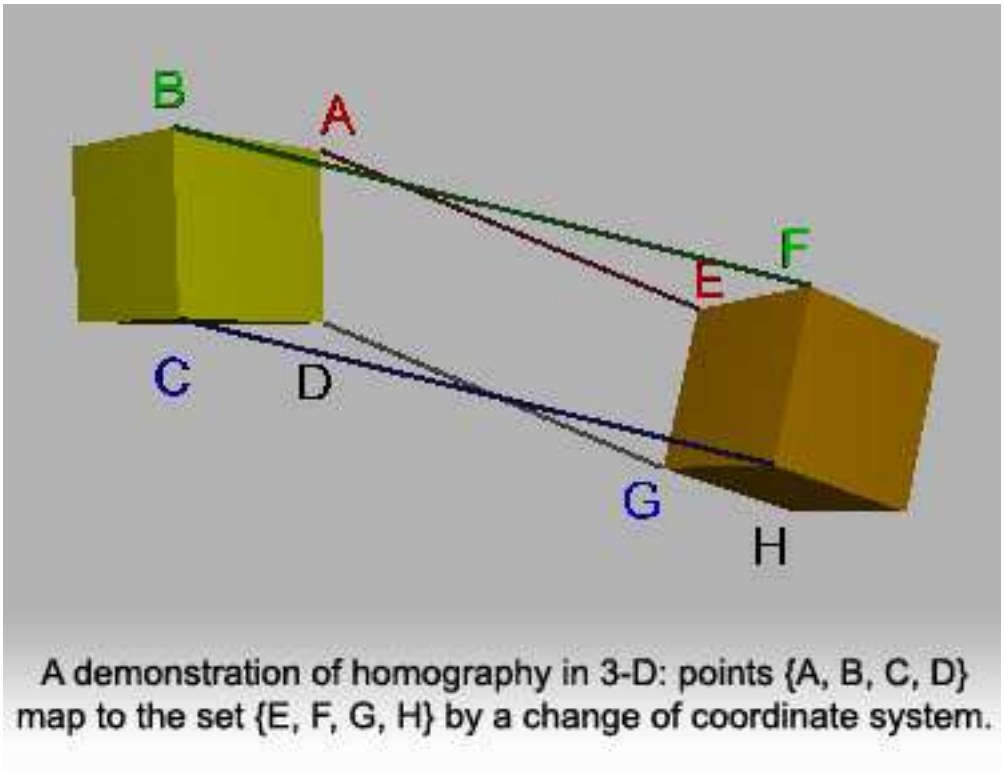

Figure 1:

- ⋆ radial distortion – distortion in the radial direction (away from the center of the lens)
- ⋆ tangential distortion – distortion in the tangential direction (parallel to a line through the center of the lens)
- ⋆ transformation – a series of translations, rotations and scales defining a geometric relationship between two objects or points

A Images

B Bibliography

I have used several papers to help me understand and code this project:

References

[1] Hartley, Richard. In Defence of the 8-Point Algorithm. Proceedings, Fifth International Conference on Computer Vision, 1995, p1064.

[2] Heikkila, Janne, and Olli Silven. A Four-step Camera Calibration Procedure with Implicit Image Correction. Proceedings, 1997 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, p1106 - 12.

[3] Livingston, Mark, and H. Harlyn Baker. Corner Detection and its Application to Camera Calibration. Unknown.

[4] Schmid, Ute. An article detailing various methods of corner detection in use. Unknown.

[5] Shoemake, Ken. Quaternions. University of Pennsylvania CIS Department technical report, 1994.

[6] Zhang, Zhengyou. A Flexible New Technique for Camera Calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, issue 11, p1330 - 34. 2000.

C Code

Prewritten code:

analyze.c: the extrapolator for data from the detector to small window sizes

detect.c: the corner detector that works on image data

matrix.c: matrix operations, useful for calculating Plessey operators

optimize.c: nonlinear multidimensional optimization

paraboloid.c: operations for fitting lines to parabolas

ppm.c: image manipulation operations

stax.c: statistix (get it?) keeping for both processes, useless practically

zerror3D.c: final statistics calculation for errors in corner point values calculated by Zhang's routine

and code written by me (included in full):

calibrate.cpp: my version of Zhang's algorithm, which takes in data from `analyze` and calibrates a camera

```
//calibrate.cpp: a rewritten Zhang's method that can be fooled with
//
//input: a model file, any number of data input files of following format:
//#pts
// X   Y   detectorStrength   imgGradX   imgGradY
//...
//(all files must have points in the same order)
//
//first line of model file should contain number of points to read
//
//output: the maximum 2-D error
//
/************************************
//output: a calibration info file of format:
//a c b u0 v0 k1 k2
//where a is horizontal focal length, b is vertical focal length, c is skew, k1 and
//
//R0
//t0
//
//R1
//t1
//etc.
************************************/

#include <stdlib.h>
#include <stdio.h>
#include <iomanip.h>
#include <string.h>
#include <typeinfo>
#include "rotations.h" //combines matrivec and quaternions nicely
#include "lmdif.c"
#include "zerror3d_newest_either.cpp" //error approximation

#define stricmp(a, b) strcasecmp(a, b)
#define square(a) (a) * (a)

double distance(vector& a, vector& b, int numDims)
{
if(a.length < numDims || b.length < numDims)
{
cout << "Error in distance(" << numDims << "): vectors have lengths " << a.length <
```

```
exit(0);
}
double sum = 0;
for(int i = 0; i < numDims; i++)
sum += square(a[i] - b[i]);
return sqrt(sum);
}

//must be global to be used by max2dError()
vector* modelCoords, **inputCoords, **inputCoordsCopy, **imageGradients; //must be
double** detectorValues;
vector** normalizedInputCoords; //filled by normalizeInputCoordinates()
vector* normalizedModelCoords;
matrix* imageNormalizations; //the Ts of normalizeInputCoordinates()
matrix modelNormalization;
int numImages, numPoints, curImgNum;
bool printOutput; //for max2dError
char* outputFilename; //ditto

#define modelPts normalizedModelCoords
#define imagePts normalizedInputCoords

//the cost function for the overarching optimization
void max2dError(int* numErrors, int* numParameters, double* parameters, double* cal

void normalizeInputCoordinates(int numInputFiles, int numPoints, const char* normDe
//values for normDescrip:
//"avgPointMagnitude"   = avg dist from centroid -> 1
//"xymax"               = normalize to max 1 in each of x- and y-directions
//"maxPointMagnitude"   = max dist from centroid -> 1
void estimateImageHomographies(int numInputFiles, int numPoints, matrix* imageHomog
void estimateIntrinsicParameters(int numInputFiles, int numPoints, matrix* imageHom
void estimateImageExtrinsicParameters(matrix& A, int numInputFiles, matrix* imageHo
void optimizeAllEstimatedParameters(int numInputFiles, int numPoints, matrix& A, ma
void estimateDistortionParameters(int numInputFiles, int numPoints, double* distort
void optimizeAll(int numInputFiles, int numPoints, matrix& A, matrix* imageRotation

//subroutines to lmdif():
//with estimateImageHomographies()
void calculatePointEstimateErrorsWithSimplifiedHomography(int* numPoints, int* esti
//with optimizeAllEstimatedParameters()
void calculatePointEstimateErrorsWithNonsimplifiedHomography(int* numValues, int* e
```

```
//with optimizeAll()
void calculatePointEstimateErrorsWithCompleteModel(int* numValues, int* estimateVec

int main(int argc, char* argv[])
{
if(argc < 5)
{
cout << "Error in calibrate: not enough inputs!\n";
exit(0);
}
//open and read model file
numImages = argc - 3;
FILE* modelFile = fopen(argv[1], "r");
if(!modelFile)
{
cout << "could not open model file\n";
exit(0);
}
fscanf(modelFile, "%d\n", &numPoints);
modelCoords = new vector[numPoints];
normalizedModelCoords = new vector[numPoints];
for(int i = 0; i < numPoints; i++)
{
modelCoords[i].resize(3);
normalizedModelCoords[i].resize(3);
fscanf(modelFile, "%lf %lf\n", &modelCoords[i][0], &modelCoords[i][1]);
modelCoords[i][2] = 1;
}
fclose(modelFile);
//open and read input files; make coordinate vectors of length 3 and homogeneous
inputCoords = new vector*[numImages];
inputCoordsCopy = new vector*[numImages];
imageGradients = new vector*[numImages];
detectorValues = new double*[numImages];
normalizedInputCoords = new vector*[numImages];
FILE* inputFile;
double trash;
//read in, making inputCoordsCopy the original for ease of code use below
for(int i = 0; i < numImages; i++)
{
inputFile = fopen(argv[2 + i], "r");
if(!inputFile)
```

```
{
cout << "could not open input file " << argv[2 + i] << endl;
exit(0);
}
fscanf(inputFile, "%lf\n", &trash);
if(trash != numPoints)
{
cout << "Error: number of points in input file " << argv[2 + i] << " (" << trash <<
exit(0);
}
inputCoords[i] = new vector[numPoints];
inputCoordsCopy[i] = new vector[numPoints];
imageGradients[i] = new vector[numPoints];
detectorValues[i] = new double[numPoints];
normalizedInputCoords[i] = new vector[numPoints];
for(int j = 0; j < numPoints; j++)
{
inputCoords[i][j].resize(3);
inputCoordsCopy[i][j].resize(3);
imageGradients[i][j].resize(3);
normalizedInputCoords[i][j].resize(3);
fscanf(inputFile, "%lf %lf %lf %lf %lf\n", &inputCoordsCopy[i][j][0], &inputCoordsC
inputCoordsCopy[i][j][2] = 1;
imageGradients[i][j][2] = 0; //so can be added to coords w/o changing w
}
}
outputFilename = new char[strlen(argv[argc - 1])];
strcpy(outputFilename, argv[argc - 1]);
//call lmdif, give it parameters to fool with detected point coordinates
//set up parameters
int numParams = 4, numValues = 4; //numValues is really 1 but must be >= numParams
double calculatedErrors[numValues], tempVec[numParams], params[numParams], fwdJacob
tempVec2[numParams], tempVec3[numParams], tempVec4[numParams], tempVec5[numValues],
maxCosineError = 1e-4, stepLength = 1e-3/*1e-14*/, factor = 1/*100*/;
int flag, numCalculatorFunctionCalls, permutationMatrixInfo[numParams], mode = 1, m
//fill initial estimates
params[0] = params[1] = params[2] = params[3] = 0; //for gradients in quadrants I,
printOutput = false;
cout << "main: calling lmdif...\n";
lmdifCode = lmdif(max2dError, &numValues, &numParams, params, calculatedErrors, &ma
&maxFuncEvals, &stepLength, tempVec, &mode, &factor, &printInfo, &flag, &numCalcula
fwdJacobian, &numValues, permutationMatrixInfo, outputData, tempVec2, tempVec3, tem
```

```
cout << "main: lmdif returns " << lmdifCode << endl;
cout << "main: info is " << flag << endl;
getchar();
//call cost function to print calibration info when optimized
printOutput = true;
max2dError(&numValues, &numParams, params, calculatedErrors, &flag);
cout << "max 2d error = " << calculatedErrors[0] << endl;
cout << "Done writing output\n";
return 0;
}

/////////////// end main ///////////////

//fills inputCoords from inputCoordsCopy with changes based on parameters given
void max2dError(int* numErrors, int* numParameters, double* parameters, double* cal
{
cout << "in max2dError: " << parameters[0] << ' ' << parameters[1] << ' ' << parame
//fill inputCoords based on parameters, using inputCoordsCopy as base values
int which;
for(int i = 0; i < numImages; i++)
for(int j = 0; j < numPoints; j++)
{
//cout << "0\n";
which = (imageGradients[i][j][0] < 0) ? ((imageGradients[i][j][1] < 0) ? 1 : 2) : (
/*cout << " 1\n";
inputCoordsCopy[i][j].print();
(imageGradients[i][j] * parameters[which]).print();
inputCoords[i][j].print();
cout << endl;
getchar();
(inputCoordsCopy[i][j] + (imageGradients[i][j] * parameters[which])).print();
cout << endl;*/
inputCoords[i][j] = inputCoordsCopy[i][j] + (imageGradients[i][j] * parameters[whic
//inputCoords[i][j].print();
//getchar();
}
//cout << "inputCoords initialized\n";
//normalize coord estimates for each image and model coords
normalizeInputCoordinates(numImages, numPoints, "avgPointMagnitude"); //fills norma
///// NOTE: as of 5/20, avgPointMagnitude is the best normalization format
//estimate homographies from the normalized model plane to each normalized image pl
//cout << "estimating image homographies\n";
```

```
matrix imageHomographies[numImages];
for(int i = 0; i < numImages; i++)
imageHomographies[i].resize(3, 3);
estimateImageHomographies(numImages, numPoints, imageHomographies);
//cout << "estimating intrinsic parameters\n";
//calculate intrinsic parameters: alpha, beta, gamma, lambda, lensCenterX, lensCent
vector intrinsicParameters(6);
estimateIntrinsicParameters(numImages, numPoints, imageHomographies, intrinsicParam
/*printf("intrinsic parameters are:\nalpha: %lf\nbeta: %lf\ngamma: %lf\nlambda: %lf
intrinsicParameters[0], intrinsicParameters[1], intrinsicParameters[2], intrinsicPa
//getchar();
//fill matrix A (camera intrinsic matrix) from intrinsic parameter values
matrix A(3, 3);
A[0][0] = intrinsicParameters[0];
A[0][1] = intrinsicParameters[2];
A[0][2] = intrinsicParameters[4];
A[1][0] = 0;
A[1][1] = intrinsicParameters[1];
A[1][2] = intrinsicParameters[5];
A[2][0] = 0;
A[2][1] = 0;
A[2][2] = 1;
//calculate extrinsic parameters for each image
matrix imageRotations[numImages];
vector imageTranslations[numImages];
for(int i = 0; i < numImages; i++)
{
imageRotations[i].resize(3, 3);
imageTranslations[i].resize(3);
}
estimateImageExtrinsicParameters(A, numImages, imageHomographies, imageRotations, i
/*for(int i = 0; i < 8; i++)
{
cout << "homography #" << i + 1 << ":\n";
imageHomographies[i].print();
}*/
// optimizeAllEstimatedParameters(numImages, numPoints, A, imageRotations, imageTra
//getchar();
//calculate distortion parameters
double distortionParameters[2]; //k1 and k2
double zhangDistortionParameters[2] = {-0.0654471, 1.70884};
estimateDistortionParameters(numImages, numPoints, distortionParameters, intrinsicP
```

```
//optimize everything together
// optimizeAll(numImages, numPoints, A, imageRotations, imageTranslations, distorti
/*matrix temp(3, 3);
for(int i = 0; i < numImages; i++)
{
cout << "\nfinal homography #" << i + 1 << ":\n";
for(int j = 0; j < 3; j++)
{
temp[j][0] = imageRotations[i][j][0];
temp[j][1] = imageRotations[i][j][1];
temp[j][2] = imageTranslations[i][j];
}
temp = A * temp;
temp.scale(1 / temp[2][2]);
temp.print();
}*/
if(printOutput) //we were called only to print info
{
//write output
FILE* outputFile = fopen(outputFilename, "w");
fprintf(outputFile, "%lf %lf %lf %lf %lf\n\n%lf %lf\n\n", intrinsicParameters[0], i
  distortionParameters[0], distortionParameters[1]);
printf("%lf %lf %lf %lf %lf\n\n%lf %lf\n\n", intrinsicParameters[0], intrinsicParam
  distortionParameters[0], distortionParameters[1]);
for(int i = 0; i < numImages; i++)
{
for(int j = 0; j < 3; j++)
{
for(int k = 0; k < 3; k++)
{
fprintf(outputFile, "%lf ", imageRotations[i][j][k]);
printf("%lf ", imageRotations[i][j][k]);
}
fprintf(outputFile, "\n");
printf("\n");
}
for(int j = 0; j < 3; j++)
{
fprintf(outputFile, "%lf ", imageTranslations[i][j]);
printf("%lf ", imageTranslations[i][j]);
}
fprintf(outputFile, "\n\n");
```

```
printf("\n\n");
}
fclose(outputFile);
}
else //we were called from lmdif
{
cout << "calling errorCalc...\n";
calculatedErrors[0] = errorCalc(numImages, numPoints, A, imageRotations, imageTrans
cout << "error: " << calculatedErrors[0] << endl;
for(int i = 1; i < (*numErrors); i++)
calculatedErrors[i] = calculatedErrors[0];
}
}

void normalizeInputCoordinates(int numInputFiles, int numPoints, const char* normDe
{ //puts results in normalizedInputCoords
vector centroid(3, 0);
double avgPointMagnitude = 0, maxPointMagnitude = 0, temp, xmax = 0, ymax = 0;
#define mNorm modelNormalization
mNorm.resize(3, 3);
//fill transformation matrix for model file
for(int j = 0; j < numPoints; j++)
centroid += modelCoords[j];
centroid.scale(1.0 / numPoints);
for(int j = 0; j < numPoints; j++)
{
if(abs(modelCoords[j][0] - centroid[0]) > xmax) xmax = abs(modelCoords[j][0] - cent
if(abs(modelCoords[j][1] - centroid[1]) > ymax) ymax = abs(modelCoords[j][1] - cent
}
//calculate average distance
for(int j = 0; j < numPoints; j++)
{
avgPointMagnitude += distance(centroid, modelCoords[j], 2);
if(distance(centroid, modelCoords[j], 2) > maxPointMagnitude) maxPointMagnitude = d
}
avgPointMagnitude /= numPoints;
//cout << "model normalization ";
//simulate matrix multiplication (T = s * t)
if(!stricmp(normDescrip, "avgPointMagnitude"))
{
mNorm[0][2] = -centroid[0] / avgPointMagnitude;
mNorm[1][2] = -centroid[1] / avgPointMagnitude;
```

```
mNorm[2][2] = 1;
mNorm[0][0] = mNorm[1][1] = 1 / avgPointMagnitude;
mNorm[0][1] = mNorm[1][0] = mNorm[2][0] = mNorm[2][1] = 0;
//cout << "(avgPointMagnitude = " << avgPointMagnitude << ")";
}
else if(!stricmp(normDescrip, "xymax"))
{
mNorm[0][2] = -centroid[0] / xmax;
mNorm[1][2] = -centroid[1] / ymax;
mNorm[2][2] = 1;
mNorm[0][0] = 1 / xmax;
mNorm[1][1] = 1 / ymax;
mNorm[0][1] = mNorm[1][0] = mNorm[2][0] = mNorm[2][1] = 0;
//cout << "(xmax = " << xmax << ", ymax = " << ymax << ")";
}
else if(!stricmp(normDescrip, "maxPointMagnitude"))
{
mNorm[0][2] = -centroid[0] / maxPointMagnitude;
mNorm[1][2] = -centroid[1] / maxPointMagnitude;
mNorm[2][2] = 1;
mNorm[0][0] = mNorm[1][1] = 1 / maxPointMagnitude;
mNorm[0][1] = mNorm[1][0] = mNorm[2][0] = mNorm[2][1] = 0;
//cout << "(maxPointMagnitude = " << maxPointMagnitude << ")";
}
//cout << ":\n";
//mNorm.print();
//transform points
temp = 0;
for(int j = 0; j < numPoints; j++)
{
normalizedModelCoords[j] = mNorm * modelCoords[j];
//normalizedModelCoords[j].print();
temp += sqrt(modelPts[j][0] * modelPts[j][0] + modelPts[j][1] * modelPts[j][1]);
//cout << "   distance: " << sqrt(modelPts[j][0] * modelPts[j][0] + modelPts[j][1]
}
//cout << "   total distance: " << temp << endl;
//getchar();
#undef mNorm
#define T imageNormalizations
T = new matrix[numInputFiles];
for(int i = 0; i < numInputFiles; i++)
{
```

```
//cout << "normalization #" << i + 1;
T[i].resize(3, 3);
//fill transformation matrix for file i
centroid[0] = centroid[1] = centroid[2] = avgPointMagnitude = maxPointMagnitude = x
for(int j = 0; j < numPoints; j++)
{
//inputCoords[i][j].print();
centroid += inputCoords[i][j];
}
centroid.scale(1.0 / numPoints);
for(int j = 0; j < numPoints; j++)
{
if(abs(inputCoords[i][j][0] - centroid[0]) > xmax) xmax = abs(inputCoords[i][j][0]
if(abs(inputCoords[i][j][1] - centroid[1]) > ymax) ymax = abs(inputCoords[i][j][1]
}
//cout << "average coord is (" << centroid[0] << ", " << centroid[1] << ")\n";
//calculate average distance
for(int j = 0; j < numPoints; j++)
{
avgPointMagnitude += distance(centroid, inputCoords[i][j], 2);
if(distance(centroid, inputCoords[i][j], 2) > maxPointMagnitude) maxPointMagnitude
}
avgPointMagnitude /= numPoints;
//simulate matrix multiplication (T = s * t)
if(!stricmp(normDescrip, "avgPointMagnitude"))
{
T[i][0][2] = -centroid[0] / avgPointMagnitude;
T[i][1][2] = -centroid[1] / avgPointMagnitude;
T[i][2][2] = 1;
T[i][0][0] = T[i][1][1] = 1 / avgPointMagnitude;
T[i][0][1] = T[i][1][0] = T[i][2][0] = T[i][2][1] = 0;
//cout << "(avgPointMagnitude = " << avgPointMagnitude << ")";
}
else if(!stricmp(normDescrip, "xymax"))
{
T[i][0][2] = -centroid[0] / xmax;
T[i][1][2] = -centroid[1] / ymax;
T[i][2][2] = 1;
T[i][0][0] = 1 / xmax;
T[i][1][1] = 1 / ymax;
T[i][0][1] = T[i][1][0] = T[i][2][0] = T[i][2][1] = 0;
//cout << "(xmax = " << xmax << ", ymax = " << ymax << ")";
```

```
}
else if(!stricmp(normDescrip, "maxPointMagnitude"))
{
T[i][0][2] = -centroid[0] / maxPointMagnitude;
T[i][1][2] = -centroid[1] / maxPointMagnitude;
T[i][2][2] = 1;
T[i][0][0] = T[i][1][1] = 1 / maxPointMagnitude;
T[i][0][1] = T[i][1][0] = T[i][2][0] = T[i][2][1] = 0;
//cout << "(maxPointMagnitude = " << maxPointMagnitude << ")";
}
//cout << ":\n";
//T[i].print();
//transform points
temp = 0;
for(int j = 0; j < numPoints; j++)
{
normalizedInputCoords[i][j] = T[i] * inputCoords[i][j];
temp += sqrt(imagePts[i][j][0] * imagePts[i][j][0] + imagePts[i][j][1] * imagePts[i
//imagePts[i][j].print();
//cout << "   distance: " << sqrt(square(imagePts[i][j][0]) + square(imagePts[i][j]
}
//cout << "\ntotal distance: " << temp << endl;
//getchar();
}
#undef T
}

//implement Zhang Appendix A, estimating image homographies from model plane to ima
//each homography matrix (one per image) is 3 x 3
//return estimates of homographies in imageHomographies
void estimateImageHomographies(int numInputFiles, int numPoints, matrix* imageHomog
{
int minValueIndex, lmdifCode;
matrix L(2 * numPoints, 9), lsvL(2 * numPoints, 2 * numPoints), rsvL(9, 9), temp(3,
vector x(9); //homography matrix values
vector svL(9); //assume we have at least 5 points detected
//parameters for lmdif()
double calculatedErrors[numPoints], tempVec[9], fwdJacobian[numPoints * 9], outputD
tempVec2[9], tempVec3[9], tempVec4[9], tempVec5[numPoints], maxFuncError = .001, ma
maxCosineError = 1e-4, stepLength = 1e-14, factor = 100;
int flag, numCalculatorFunctionCalls, permutationMatrixInfo[9], numParameters = 9,
matrix imageNormalizationInverse; //the inv(E) of finding H
```

```
for(int i = 0; i < numInputFiles; i++)
{
//obtain initial homography guess x, where Lx = 0
//fill up L
//each two rows are:  | ~M^T 0^T -u~M^T |
//                    | 0^T ~M^T -v~M^T |
//where ~M is the model point (x, y, 1) in world coords and (u, v) is the image poi
for(int j = 0; j < numPoints; j++)
{
L[2 * j][0] = modelPts[j][0];
L[2 * j][1] = modelPts[j][1];
L[2 * j][2] = 1;
L[2 * j][3] = L[2 * j][4] = L[2 * j][5] = 0;
L[2 * j][6] = -imagePts[i][j][0] * modelPts[j][0];
L[2 * j][7] = -imagePts[i][j][0] * modelPts[j][1];
L[2 * j][8] = -imagePts[i][j][0];
L[2 * j + 1][0] = L[2 * j + 1][1] = L[2 * j + 1][2] = 0;
L[2 * j + 1][3] = modelPts[j][0];
L[2 * j + 1][4] = modelPts[j][1];
L[2 * j + 1][5] = 1;
L[2 * j + 1][6] = -imagePts[i][j][1] * modelPts[j][0];
L[2 * j + 1][7] = -imagePts[i][j][1] * modelPts[j][1];
L[2 * j + 1][8] = -imagePts[i][j][1];
}
//find right singular vector of L assoc. with smallest singular value: this is x
L.svd(lsvL, svL, rsvL); //columns of V are eigenvectors w/ corresponding eigenvalue
//find index of minimum eigenvalue
minValueIndex = 0;
for(int j = 1; j < 9; j++)
if(svL[j] < svL[minValueIndex])
minValueIndex = j;
rsvL.transpose(); //so we can use its rows, rather than columns, as vectors (becaus
//minimize sum of squares of distances between detected image points and model poin
//number of estimated parameters (n) is 3 (for homography matrix)
//number of functions (m) is number of points (one distance per point): thus must h
curImgNum = i; //for calculatePointEstimateErrorsWithSimplifiedHomography()
lmdifCode = lmdif(calculatePointEstimateErrorsWithSimplifiedHomography, &numPoints,
&maxRelativeError, &maxCosineError, &maxFuncEvals, &stepLength, tempVec,
&mode, &factor, &printInfo, &flag, &numCalculatorFunctionCalls,
fwdJacobian, &numPoints, permutationMatrixInfo, outputData, tempVec2, tempVec3, tem
//cout << "lmdif returns " << lmdifCode << endl;
//cout << "info is " << flag << endl;
```

```
//insert values of x into imageHomographies (x is concatenated rows of H)
for(int j = 0; j < 3; j++)
for(int k = 0; k < 3; k++)
imageHomographies[i][j][k] = rsvL[minValueIndex][j * 3 + k] / rsvL[minValueIndex][8
//find H = inv(E) * H * D, where E is the image normalization and D is the model no
imageNormalizationInverse = imageNormalizations[i];
imageNormalizationInverse.invert();
imageHomographies[i] = imageNormalizationInverse * (imageHomographies[i] * modelNor
imageHomographies[i].scale(1 / imageHomographies[i][2][2]);
//cout << "end estimate of homography matrix " << i + 1 << ":\n";
//imageHomographies[i].print();
//getchar();
}
}

//form matrix V in Zhang section 3.1, SVD it,
//use elements of b to find six useful parameters (Appendix B)
void estimateIntrinsicParameters(int numInputFiles, int numPoints, matrix* imageHom
{
matrix V(2 * numInputFiles, 6);
//fill V with functions of the homography matrices
#define h(a, i, j) imageHomographies[a][j - 1][i - 1]
for(int i = 0; i < numInputFiles; i++)
{
//where column is indexed before row, and indices range from 1 to 3:
//v_ij = [ h_i1 * h_j1 , h_i1 * h_j2 + h_i2 * h_j1 , h_i2 * h_j2 , h_i3 * h_j1 + h_
//each two rows of V are:
// [ v_12        ]
// [ v_11 - v_22 ]
V[2 * i][0] = h(i, 1, 1) * h(i, 2, 1);
V[2 * i][1] = h(i, 1, 1) * h(i, 2, 2) + h(i, 1, 2) * h(i, 2, 1);
V[2 * i][2] = h(i, 1, 2) * h(i, 2, 2);
V[2 * i][3] = h(i, 1, 3) * h(i, 2, 1) + h(i, 1, 1) * h(i, 2, 3);
V[2 * i][4] = h(i, 1, 3) * h(i, 2, 2) + h(i, 1, 2) * h(i, 2, 3);
V[2 * i][5] = h(i, 1, 3) * h(i, 2, 3);
V[2 * i + 1][0] = h(i, 1, 1) * h(i, 1, 1) - h(i, 2, 1) * h(i, 2, 1);
V[2 * i + 1][1] = h(i, 1, 1) * h(i, 1, 2) + h(i, 1, 2) * h(i, 1, 1) - h(i, 2, 1) *
V[2 * i + 1][2] = h(i, 1, 2) * h(i, 1, 2) - h(i, 2, 2) * h(i, 2, 2);
V[2 * i + 1][3] = h(i, 1, 3) * h(i, 1, 1) + h(i, 1, 1) * h(i, 1, 3) - h(i, 2, 3) *
V[2 * i + 1][4] = h(i, 1, 3) * h(i, 1, 2) + h(i, 1, 2) * h(i, 1, 3) - h(i, 2, 3) *
V[2 * i + 1][5] = h(i, 1, 3) * h(i, 1, 3) - h(i, 2, 3) * h(i, 2, 3);
}
```

```
#undef h
//find right singular vector of V assoc. with smallest singular value: this is b
vector w(6);
matrix lsv(2 * numInputFiles, 2 * numInputFiles), rsv(6, 6);
V.svd(lsv, w, rsv); //columns of rsv are right singular vectors w/ corresponding si
//find index of minimum eigenvalue
int minValueIndex = 0;
for(int i = 1; i < 6; i++)
if(w[i] < w[minValueIndex])
minValueIndex = i;
#define b(i) rsv[i][minValueIndex] //use notation b for eigenvector corresponding t
/*cout << "printing estimated b\n";
for(int i = 0; i < 6; i++)
cout << b(i) << ' ';
cout << endl << endl;
getchar();*/
//calculate intrinsic parameters from vector b (Zhang Appendix B)
//intrinsic parameters: alpha, beta, gamma, lambda, lensCenterX (u0), lensCenterY (
/* v0     */ intrinsicParameters[5] = (b(1) * b(3) - b(0) * b(4)) / (b(0) * b(2) -
/* lambda */ intrinsicParameters[3] = b(5) - (square(b(3)) + intrinsicParameters[5]
/* alpha  */ intrinsicParameters[0] = sqrt(intrinsicParameters[3] / b(0));
/* beta   */ intrinsicParameters[1] = sqrt(intrinsicParameters[3] * b(0) / (b(0) *
/* gamma  */ intrinsicParameters[2] = -b(1) * square(intrinsicParameters[0]) * intr
/* u0     */ intrinsicParameters[4] = intrinsicParameters[2] * intrinsicParameters[
- b(3) * square(intrinsicParameters[0]) / intrinsicParameters[3];
#undef b
}

//Zhang section 3.1: get R, t from A, H
void estimateImageExtrinsicParameters(matrix& A, int numInputFiles, matrix* imageHo
{
//cout << "estimating R, t:\n";
matrix invA;
invA = A;
invA.invert();
//for SVDing R
matrix U(3, 3), V(3, 3);
vector w(3);
for(int i = 0; i < numInputFiles; i++)
{
//obtain initial guesses at rotation and translation matrices for each input file
imageRotations[i] = invA * imageHomographies[i]; //asign [r_1 r_2 t] = A^-1 * H
```

```
imageRotations[i].transpose();
//cout << "norm magnitudes in estimateExtrinsics: \n" << imageRotations[i][0].norm(
imageRotations[i].scale(1 / imageRotations[i][0].norm());
imageTranslations[i] = imageRotations[i][2];
imageRotations[i][2] = imageRotations[i][0].crossProduct(imageRotations[i][1]); //a
imageRotations[i].transpose();
//cout << "image # " << i + 1 << " translation:\n";
//imageTranslations[i].print();
//change R to a rotation-matrix format (Zhang appendix C): take SVD(R) = U*s*V^T, s
imageRotations[i].svd(U, w, V);
V.transpose();
imageRotations[i] = U * V;
//cout << "\nimage # " << i + 1 << " rotation:\n";
//imageRotations[i].print();
//getchar();
}
}

//Zhang section 3.2: minimize sum of squares of all points WITHOUT distortion to re
void optimizeAllEstimatedParameters(int numInputFiles, int numPoints, matrix& A, ma
{
//cout << "optimizing all parameters without distortion\n";
//parameters to lmdif()
int numValues = numPoints * numInputFiles, numParameters = 5 + 6 * numInputFiles;
double initialEstimate[numParameters], calculatedErrors[numValues], tempVec[numPara
tempVec2[numParameters], tempVec3[numParameters], tempVec4[numParameters], tempVec5
maxCosineError = 0, stepLength = 1e-8, factor = 100;
int flag, numCalculatorFunctionCalls, permutationMatrixInfo[numParameters], mode =
//load values from structures into lmdif array
initialEstimate[0] = A[0][0];
initialEstimate[1] = A[1][1];
initialEstimate[2] = A[0][1];
initialEstimate[3] = A[0][2];
initialEstimate[4] = A[1][2];
quaternion q;
double rodrigues[3]; //Rodrigues parameters: components of axis of rotation, scaled
for(int i = 0; i < numInputFiles; i++)
{
//glean three parameters related to the rotation axis and angle from a quaternion t
//cout << "in optimizeAll: image rotation #" << i + 1 << ": ";
//imageRotations[i].print();
q = imageRotations[i].createRotationQuaternion(); //q is [v sin theta, cos theta]
```

```
q.createRodrigues(rodrigues);
for(int j = 0; j < 3; j++)
{
initialEstimate[5 + i * 6 + j] = rodrigues[j];
initialEstimate[5 + i * 6 + 3 + j] = imageTranslations[i][j];
}
//cout << "\ntranslation: ";
//imageTranslations[i].print();
//getchar();
}
//minimize sum of squares of distances between detected image points and model poin
//s * ~m = A * [ r1 r2 t ] * ~M, where s is an arbitrary scalar
//number of estimated parameters (n) is 5 (intrinsic parameters) + 6 * number of fi
//number of functions (m) is number of points times number of files (one distance p
//cout << "calling lmdif...\n";
//cout << "parameters: ";
//for(int i = 0; i < 53; i++)
// cout << initialEstimate[i] << "   ";
//cout << "\n";
lmdifCode = lmdif(calculatePointEstimateErrorsWithNonsimplifiedHomography, &numValu
&maxRelativeError, &maxCosineError, &maxFuncEvals, &stepLength, tempVec, &mode, &fa
fwdJacobian, &numValues, permutationMatrixInfo, outputData, tempVec2, tempVec3, tem
//cout << "lmdif returns " << lmdifCode << endl;
//cout << "function called " << numCalculatorFunctionCalls << " times\n";
//cout << "info is " << flag << endl;
//cout << "parameters: ";
//for(int i = 0; i < 53; i++)
// cout << initialEstimate[i] << "   ";
//cout << "\n";
//put optimized values back into structures
A[0][0] = initialEstimate[0];
A[1][1] = initialEstimate[1];
A[0][1] = initialEstimate[2];
A[0][2] = initialEstimate[3];
A[1][2] = initialEstimate[4];
for(int i = 0; i < numInputFiles; i++)
{
for(int j = 0; j < 3; j++)
{
rodrigues[j] = initialEstimate[5 + i * 6 + j];
imageTranslations[i][j] = initialEstimate[5 + i * 6 + 3 + j];
}
```

```
//find rotation matrix from angle and axis of rotation by using a quaternion
q.createFromRodrigues(rodrigues);
imageRotations[i] = q.createRotationMatrix();
//cout << "later: rotation matrix #" << i + 1 << ":\n";
//imageRotations[i].print();
//getchar();
}
}

void estimateDistortionParameters(int numInputFiles, int numPoints, double* distort
{
//cout << "\nestimating distortion parameters\n";
matrix D(2 * numInputFiles * numPoints, 2), dTrans, dTransDInv;
vector k(2), d(2 * numInputFiles * numPoints), tempPoint(3), tempPoint2(3);
for(int i = 0; i < numInputFiles; i++)
{
//(u0, v0) are cameraIntrinsicParameters [4] and [5]
//(~u, ~v) are inputCoords
//fill D, d
for(int j = 0; j < numPoints; j++)
{
tempPoint = imageHomographies[i] * modelCoords[j]; //tempPoint is (u, v, ~)
tempPoint.scale(1 / tempPoint[2]);
//cout << "(u, v) = " << tempPoint[0] << ", " << tempPoint[1] << "; " << tempPoint[
tempPoint2 = imageNormalizations[i] * tempPoint; //tempPoint2 is (x, y, ~)
//cout << "(x, y) = " << tempPoint2[0] << ", " << tempPoint2[1] << "; " << tempPoin
D[i * numPoints * 2 + j * 2][0] = (tempPoint[0] - intrinsicParameters[4]) * (square
D[i * numPoints * 2 + j * 2][1] = (tempPoint[0] - intrinsicParameters[4]) * square(
D[i * numPoints * 2 + j * 2 + 1][0] = (tempPoint[1] - intrinsicParameters[5]) * (sq
D[i * numPoints * 2 + j * 2 + 1][1] = (tempPoint[1] - intrinsicParameters[5]) * squ
d[i * numPoints * 2 + j * 2] = inputCoords[i][j][0] - tempPoint[0];
d[i * numPoints * 2 + j * 2 + 1] = inputCoords[i][j][1] - tempPoint[1];
}
//getchar();
}
dTrans = D;
dTrans.transpose();
dTransDInv = dTrans * D;
dTransDInv.invert();
k = dTransDInv * (dTrans * d);
//cout << "distortionParameters initial estimate:  ";
//k.print();
```

```
//cout << endl;
//getchar();
distortionParameters[0] = k[0];
distortionParameters[1] = k[1];
}

//Zhang section 3.3: final optimization of all parameters including distortion
void optimizeAll(int numInputFiles, int numPoints, matrix& A, matrix* imageRotation
{
//cout << "optimizing all parameters with distortion\n";
//parameters to lmdif()
int numValues = numPoints * numInputFiles, numParameters = 7 + 6 * numInputFiles;
double initialEstimate[numParameters], calculatedErrors[numValues], tempVec[numPara
tempVec2[numParameters], tempVec3[numParameters], tempVec4[numParameters], tempVec5
maxCosineError = 1e-4, stepLength = 1e-14, factor = 100;
int flag, numCalculatorFunctionCalls, permutationMatrixInfo[numParameters], mode =
//load values from structures into lmdif array
initialEstimate[0] = A[0][0];
initialEstimate[1] = A[1][1];
initialEstimate[2] = A[0][1];
initialEstimate[3] = A[0][2];
initialEstimate[4] = A[1][2];
initialEstimate[5] = distortionParameters[0];
initialEstimate[6] = distortionParameters[1];
quaternion q;
double rodrigues[3]; //Rodrigues parameters: components of axis of rotation, scaled
for(int i = 0; i < numInputFiles; i++)
{
//glean three parameters related to the rotation axis and angle from a quaternion t
//cout << "in optimizeAll: image rotation #" << i + 1 << ":\n";
//imageRotations[i].print();
q.createFromRotationMatrix(imageRotations[i]); //q is [v sin theta, cos theta]
q.createRodrigues(rodrigues);
for(int j = 0; j < 3; j++)
{
initialEstimate[7 + i * 6 + j] = rodrigues[j];
initialEstimate[7 + i * 6 + 3 + j] = imageTranslations[i][j];
}
//printf("\nRodrigues #%i: %.8lf %.8lf %.8lf", i + 1, initialEstimate[7 + i * 6], i
getchar();
}
//minimize sum of squares of distances between detected image points and model poin
```

```
//s * ~m = A * [ r1 r2 t ] * ~M, where s is an arbitrary scalar
//number of estimated parameters (n) is 5 (intrinsic parameters) + 6 * number of fi
//number of functions (m) is number of points times number of files (one distance p
//cout << "calling lmdif...\n";
lmdifCode = lmdif(calculatePointEstimateErrorsWithCompleteModel, &numValues, &numPa
&maxRelativeError, &maxCosineError, &maxFuncEvals, &stepLength, tempVec,
&mode, &factor, &printInfo, &flag, &numCalculatorFunctionCalls,
fwdJacobian, &numValues, permutationMatrixInfo, outputData, tempVec2, tempVec3, tem
//cout << "lmdif returns " << lmdifCode << endl;
//cout << "function called " << numCalculatorFunctionCalls << " times\n";
//cout << "info is " << flag << endl;
//put optimized values back into structures
A[0][0] = initialEstimate[0];
A[1][1] = initialEstimate[1];
A[0][1] = initialEstimate[2];
A[0][2] = initialEstimate[3];
A[1][2] = initialEstimate[4];
distortionParameters[0] = initialEstimate[5];
distortionParameters[1] = initialEstimate[6];
for(int i = 0; i < numInputFiles; i++)
{
for(int j = 0; j < 3; j++)
{
rodrigues[j] = initialEstimate[7 + i * 6 + j];
imageTranslations[i][j] = initialEstimate[7 + i * 6 + 3 + j];
}
//find rotation matrix from angle and axis of rotation by using a quaternion
q.createFromRodrigues(rodrigues);
imageRotations[i] = q.createRotationMatrix();
}
}

/*subroutine fcn(m,n,x,fvec,iflag) */
/*         integer m,n,iflag */
/*         double precision x(n),fvec(m) */
/*         ---------- */
/*         calculate the functions at x and */
/*         return this vector in fvec. */
/*         ---------- */
/*         return */
/*         end */
```

```
/*         the value of iflag should not be changed by fcn unless */
/*         the user wants to terminate execution of lmdif. */
/*         in this case set iflag to a negative integer. */

/*       m is a positive integer input variable set to the number */
/*         of functions. */

/*       n is a positive integer input variable set to the number */
/*         of variables. n must not exceed m. */

/*       x is an array of length n. on input x must contain */
/*         an initial estimate of the solution vector. on output x */
/*         contains the final estimate of the solution vector. */

/*       fvec is an output array of length m which contains */
/*         the functions evaluated at the output x. */

//given: double** modelCoords; double*** inputCoords; int curImgNum
//compute value of first equation in Zhang appendix A (for use by estimateImageHomo
void calculatePointEstimateErrorsWithSimplifiedHomography(int* numPoints, int* esti
{ //homographyMatrixEstimates are the params being manipulated; calculatedErrors ar
vector homographizedModelPoint(3);
matrix estimatedHomographyMatrix(3, 3);
//fill homography matrix from estimates
for(int i = 0; i < 3; i++)
for(int j = 0; j < 3; j++)
estimatedHomographyMatrix[i][j] = homographyMatrixEstimates[i * 3 + j];
//calculate errors
for(int i = 0; i < (*numPoints); i++)
{
//inputCoords[curImgNum][i] is m_i
//modelCoords[i] is M_i
//homographizedModelPoint is ^m_i
//^m_i = 1 / (h_3 * M_i) * [[ h_1 * M_i ];[ h_2 * M_i ] [ unnec.]]
homographizedModelPoint = estimatedHomographyMatrix * normalizedModelCoords[i];
homographizedModelPoint.scale(1 / homographizedModelPoint[2]); //make z = 1
calculatedErrors[i] = distance(normalizedInputCoords[curImgNum][i], homographizedMo
/*cout << "\ninput coords: ";
normalizedInputCoords[curImgNum][i].print();
cout << "\nH * model coords: ";
homographizedModelPoint.print();*/
}
```

```
//getchar();
}

//for use by optimizeAllEstimatedParameters()
//calculate sum of squares of distances between detected image points and model poi
//s * ~m = A * [ r1 r2 t ] * ~M, where s is an arbitrary scalar
void calculatePointEstimateErrorsWithNonsimplifiedHomography(int* numValues, int* e
{
//cout << "in cPEEWNH...";
int numImages = ((*estimateVectorSize) - 5) / 6, numPoints = (*numValues) / numImag
matrix A(3, 3), imageTransformations[numImages], constructedHomographies[numImages]
vector homographizedModelPoint(3), imageTranslation(3);
quaternion q;
double rodrigues[3], temp;
//fill A
A[0][0] = parameterEstimates[0];
A[1][1] = parameterEstimates[1];
A[0][1] = parameterEstimates[2];
A[0][2] = parameterEstimates[3];
A[1][2] = parameterEstimates[4];
A[2][2] = 1;
A[1][0] = A[2][0] = A[2][1] = 0;
//cout << "A:";
//A.print();
for(int i = 0; i < numImages; i++)
{
imageTransformations[i].resize(3, 3);
constructedHomographies[i].resize(3, 3);
//find rotation matrix from angle and axis of rotation by using a quaternion
for(int j = 0; j < 3; j++)
{
rodrigues[j] = parameterEstimates[5 + i * 6 + j];
imageTranslation[j] = parameterEstimates[5 + i * 6 + 3 + j];
}
q.createFromRodrigues(rodrigues);
imageTransformations[i] = q.createRotationMatrix();
//cout << "rotation #" << i + 1 << ": ";
//imageTransformations[i].print();
for(int j = 0; j < 3; j++)
imageTransformations[i][j][2] = imageTranslation[j];
constructedHomographies[i] = A * imageTransformations[i];
//cout << "homography #" << i + 1 << ":";
```

```
//constructedHomographies[i].scale(1 / constructedHomographies[i][2][2]);
//constructedHomographies[i].print();
//getchar();
temp = 0;
//cout << "points and calculated points:\n";
for(int j = 0; j < numPoints; j++)
{
homographizedModelPoint = constructedHomographies[i] * modelCoords[j];
homographizedModelPoint.scale(1 / homographizedModelPoint[2]); //move to same xy-pl
//homographizedModelPoint.print();
//cout << "\n; ";
//inputCoords[i][j].print();
//getchar();
calculatedErrors[i * numPoints + j] = distance(homographizedModelPoint, inputCoords
temp += calculatedErrors[i * numPoints + j];
}
}
/*
cout << "\nparameters:\n";
for(int i = 0; i < 53; i++)
printf("%20.8lf", parameterEstimates[i]);
*/
//cout << "\ntotal error: " << temp << endl;
//getchar();
}

//called by optimizeAll() before main() writes output file
//calculates points allowing for distortion
void calculatePointEstimateErrorsWithCompleteModel(int* numValues, int* estimateVec
{
//cout << "in cPEEWCM... ";
int numImages = ((*estimateVectorSize) - 7) / 6, numPoints = (*numValues) / numImag
matrix A(3, 3), imageTransformations[numImages], constructedHomographies[numImages]
vector homographizedModelPoint(3), normalizedImagePoint(3), imageTranslation(3);
quaternion q;
double rodrigues[3], temp, distortionParameters[2];
//fill A
A[0][0] = parameterEstimates[0];
A[1][1] = parameterEstimates[1];
A[0][1] = parameterEstimates[2];
A[0][2] = parameterEstimates[3];
A[1][2] = parameterEstimates[4];
```

```
A[2][2] = 1;
A[1][0] = A[2][0] = A[2][1] = 0;
//extract distortion parameters
distortionParameters[0] = parameterEstimates[5];
distortionParameters[1] = parameterEstimates[6];
//cout << "distortion: " << distortionParameters[0] << ", " << distortionParameters
//cout << "A:";
//A.print();
for(int i = 0; i < numImages; i++)
{
imageTransformations[i].resize(3, 3);
constructedHomographies[i].resize(3, 3);
//find rotation matrix from angle and axis of rotation by using a quaternion
//cout << "Rodrigues:\n";
for(int j = 0; j < 3; j++)
{
rodrigues[j] = parameterEstimates[7 + i * 6 + j];
//cout << q[j] << ' ';
imageTranslation[j] = parameterEstimates[7 + i * 6 + 3 + j];
}
//cout << "\n";
q.createFromRodrigues(rodrigues);
//cout << "quaternion #" << i + 1 << ":\n";
//q.print();
imageTransformations[i] = q.createRotationMatrix();
//cout << "\nrotation #" << i + 1 << ":\n";
//rotationT.print();
//getchar();
for(int j = 0; j < 3; j++)
imageTransformations[i][j][2] = imageTranslation[j];
constructedHomographies[i] = A * imageTransformations[i];
//cout << "homography #" << i + 1 << ":";
//constructedHomographies[i].print();
//getchar();
temp = 0;
//cout << "points and calculated points:\n";
for(int j = 0; j < numPoints; j++) /// getting wrong numbers in homographizedModelP
{
homographizedModelPoint = constructedHomographies[i] * modelCoords[j];
homographizedModelPoint.scale(1 / homographizedModelPoint[2]); //move to same xy-pl
//cout << "\nideal: ";
//homographizedModelPoint.print();
```

```
normalizedImagePoint = imageNormalizations[i] * (constructedHomographies[i] * model
normalizedImagePoint.scale(1 / normalizedImagePoint[2]);
//cout << "; norm ideal: ";
//normalizedImagePoint.print();
//cout << endl;
for(int k = 0; k < 2; k++) //turn ideal coords into distorted ideal coords to compa
{
//cout << "  u - u0: " << homographizedModelPoint[k] - parameterEstimates[3 + k] <<
//cout << "  k1 term = " << parameterEstimates[5] << " * " << (square(normalizedIma
//cout << "  k2 term = " << parameterEstimates[6] << " * " << square(square(normali
homographizedModelPoint[k] += (homographizedModelPoint[k] - parameterEstimates[3 +
 * (parameterEstimates[5] * (square(normalizedImagePoint[0]) + square(normalizedIma
 + parameterEstimates[6] * square(square(normalizedImagePoint[0]) + square(normaliz
}
//inputCoords[i][j].print();
//cout << " : ";
//homographizedModelPoint.print(); ////////////////////// errors here
//cout << "\n";
//getchar();
calculatedErrors[i * numPoints + j] = distance(homographizedModelPoint, inputCoords
temp += calculatedErrors[i * numPoints + j]; //~u = u + (u - u0) * (k1(x^2 + y^2) +
} //where (~u, ~v) is inputCoords and (u, v) is the right side
}
//cout << "total error: " << temp << endl;
//getchar();
}
```

matrivec.h: my matrix and vector classes, more formal and easier to use than Mark's but based on them

```
//matrix.h: matrix manipulation routines for use in calibrate.cpp

#ifndef MATRIVEC_H
#define MATRIVEC_H

#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include "dsvdc.c"
```

```
void swap(double& a, double& b)
{
   double c = a;
   a = b;
   b = c;
}

class quaternion;
class matrix;

class vector
{
   public:

      vector() : length(0), values(NULL)
      {};
      vector(int elements);
vector(int elements, double fillVal);
      vector(double* dbls, int elements);
      ~vector()
      {};

const vector operator = (const vector v);
vector operator + (vector v);
void operator += (vector v);
vector operator - (vector v);
void operator -= (vector v);
vector operator * (double scalar);
void operator *= (double scalar);
matrix operator * (vector v);
vector operator * (matrix m);

const int getLength();
      const double& operator [] (int index) const;
      double& operator [] (int index);

void scale(double scalar);
void fill(double* dbls, int numElements);
      void resize(int newLen);

double norm();
```

```
double dotProduct(vector v);
vector crossProduct(vector v);

      void multiply(matrix m, vector& result);
void multiply(vector v, matrix& result);
void multiply(vector v, double& result);

      void print();


      int length;
      double* values;
};

class matrix
{
   public:

      matrix() : numRows(0), numCols(0), values(NULL)
      {};
      matrix(int rows, int cols);
matrix(int rows, int cols, double fillVal);
      matrix(vector* vals, int rows, int cols);
      ~matrix()
      {};

const matrix operator = (const matrix m);
matrix operator + (matrix m);
void operator += (matrix m);
matrix operator - (matrix m);
void operator -= (matrix m);
matrix operator * (double scalar);
void operator *= (double scalar);
matrix operator * (matrix m);
void operator *= (matrix m);
vector operator * (vector v);

const int getRows();
const int getCols();
      vector& operator [] (int index);
matrix submatrix(int row, int col, int rowSize, int colSize);
```

```
matrix identity(int size);

void multiply(matrix m, matrix& result);
      void multiply(vector v, vector& result);

void swapRows(int r1, int r2);

void fill(double datum);
void scale(double scalar);
      void transpose();
      void resize(int newRows, int newCols);
matrix augment(matrix m);
matrix rref();

double trace();
      double det();
bool isSymmetric();
int firstNonzeroColumn(int startCol, int startRow);
int firstNonzeroColumnEntry(int col);

void print();

bool invert(); //returns whether invertible
      int LUdecomp(int* index, double* d); //for square matrices
      int LUbacksub(int* index, double* b); //for square matrices
void svd(matrix& u, vector& w, matrix& v); //calls dsvdc()

quaternion createRotationQuaternion();


      int numRows, numCols;
      vector* values;
};

#include "matrivec.cpp"

#endif //MATRIVEC_H

//matrivec.cpp: vector and matrix manipulation routines for use in calibrate.cpp

vector::vector(int elements)
{
```

```
values = new double[length = elements];
}

vector::vector(int elements, double fillVal)
{
values = new double[length = elements];
for(int i = 0; i < length; i++)
values[i] = fillVal;
}

vector::vector(double* dbls, int elements)
{
values = new double[length = elements];
for(int i = 0; i < length; i++)
values[i] = dbls[i];
}

const vector vector::operator = (const vector v)
{
resize(length = v.length);
for(int i = 0; i < length; i++)
values[i] = v.values[i];
return v;
}

vector vector::operator + (vector v)
{
if(length != v.length)
{
cout << "Error in vector::operator +: vector lengths are not equal (" << length <<
exit(0);
}
vector temp(length);
for(int i = 0; i < length; i++)
temp[i] = values[i] + v.values[i];
return temp;
}

void vector::operator += (vector v)
{
if(length != v.length)
{
```

```
cout << "Error in vector::operator +=: vector lengths are not equal (" << length <<
exit(0);
}
for(int i = 0; i < length; i++)
values[i] += v.values[i];
}

vector vector::operator - (vector v)
{
if(length != v.length)
{
cout << "Error in vector::operator -: vector lengths are not equal (" << length <<
exit(0);
}
vector temp(length);
for(int i = 0; i < length; i++)
temp[i] = values[i] + v.values[i];
return temp;
}

void vector::operator -= (vector v)
{
if(length != v.length)
{
cout << "Error in vector::operator -=: vector lengths are not equal (" << length <<
exit(0);
}
for(int i = 0; i < length; i++)
values[i] -= v.values[i];
}

vector vector::operator * (double scalar)
{
vector temp(length);
for(int i = 0; i < length; i++)
temp[i] = values[i] * scalar;
return temp;
}

void vector::operator *= (double scalar)
{
for(int i = 0; i < length; i++)
```

```
values[i] *= scalar;
}

matrix vector::operator * (vector v)
{
matrix temp;
multiply(v, temp);
return temp;
}

vector vector::operator * (matrix m)
{
if(m.numRows != length || m.numCols != length)
{
cout << "Error in vector::operator * (matrix): vector size is " << length << " and
  << m.numCols << "!\n";
exit(0);
}
vector temp(length);
multiply(m, temp);
return temp;
}

const int vector::getLength()
{
return length;
}

const double& vector::operator [] (int index) const
{
return values[index];
}

double& vector::operator [] (int index)
{
return values[index];
}

void vector::scale(double scalar)
{
for(int i = 0; i < length; i++)
values[i] *= scalar;
```

```
}

void vector::fill(double* dbls, int numElements)
{
int minLen = min(length, numElements);
for(int i = 0; i < minLen; i++)
values[i] = dbls[i];
}

void vector::resize(int newLen)
{
if(!length)
{
values = new double[newLen];
length = newLen;
return;
}
int minLen = min(length, newLen);
double* newVals = new double[length = newLen];
for(int i = 0; i < minLen; i++)
newVals[i] = values[i];
values = newVals;
}

double vector::norm()
{
double sum = 0;
for(int i = 0; i < length; i++)
sum += values[i] * values[i];
return sqrt(sum);
}

double vector::dotProduct(vector v)
{
if(length != v.length)
{
cout << "Error in vector::dotProduct(): vector lengths are unequal (" << length <<
exit(0);
}
double sum = 0;
for(int i = 0; i < length; i++)
sum += values[i] * v.values[i];
```

```
return sum;
}

vector vector::crossProduct(vector v)
{
if(length != 3 || v.length != 3)
{
printf("Error in vector::crossProduct(): vector not of size 3!\n");
exit(0);
}
vector result(3);
result[0] = values[1] * v.values[2] - values[2] * v.values[1];
result[1] = values[2] * v.values[0] - values[0] * v.values[2];
result[2] = values[0] * v.values[1] - values[1] * v.values[0];
return result;
}

void vector::multiply(matrix m, vector& result) //horiz vector x matrix --> horiz v
{
if(length != m.numRows)
{
printf("Error in vec mult by matrix: vec num cols (%d) != matrix num rows (%d)!\n",
exit(0);
}
result.resize(m.numCols);
for(int i = 0; i < m.numCols; i++)
{
result[i] = 0;
for(int j = 0; j < m.numRows; j++)
result[i] += m[j][i] * values[j];
}
}

void vector::multiply(vector v, matrix& result)
{
result.resize(length, v.length);
for(int i = 0; i < length; i++)
for(int j = 0; j < v.length; j++)
result[i][j] = values[i] * v.values[j];
}

void vector::multiply(vector v, double& result)
```

```
{
result = dotProduct(v);
}

void vector::print()
{
printf("< ");
for(int i = 0; i < length; i++)
printf("%.8f  ", values[i]);
printf(" >");
}

/*****************************************************************************
/*****************************************************************************

matrix::matrix(int rows, int cols)
{
values = new vector[rows];
for(int i = 0; i < rows; i++)
values[i].resize(cols);
numRows = rows;
numCols = cols;
}

matrix::matrix(int rows, int cols, double fillVal)
{
values = new vector[rows];
for(int i = 0; i < rows; i++)
{
values[i].resize(cols);
for(int j = 0; j < cols; j++)
values[i][j] = fillVal;
}
numRows = rows;
numCols = cols;
}

matrix::matrix(vector* vals, int rows, int cols)
{
numRows = rows;
numCols = cols;
values = new vector[rows];
```

```
for(int i = 0; i < rows; i++)
values[i] = vals[i];
}

const matrix matrix::operator = (const matrix m)
{
if(numRows != m.numRows || numCols != m.numCols)
resize(m.numRows, m.numCols);
for(int i = 0; i < numRows; i++)
for(int j = 0; j < numCols; j++)
values[i][j] = m.values[i][j];
return m;
}

matrix matrix::operator + (matrix m)
{
if(numRows != m.numRows || numCols != numCols)
{
cout << "Error in matrix::operator +: matrices have different sizes (" << numRows <
  << " x " << m.numCols << ")!\n";
exit(0);
}
matrix temp(numRows, numCols);
for(int i = 0; i < numRows; i++)
for(int j = 0; j < numCols; j++)
temp[i][j] = values[i][j] + m.values[i][j];
return temp;
}

void matrix::operator += (matrix m)
{
if(numRows != m.numRows || numCols != m.numCols)
{
printf("Error in matrix::operator +=: matrices have sizes %d x %d and %d x %d (not
exit(0);
}
for(int i = 0; i < numRows; i++)
for(int j = 0; j < numCols; j++)
values[i][j] += m.values[i][j];
}

matrix matrix::operator - (matrix m)
```

```
{
if(numRows != m.numRows || numCols != numCols)
{
cout << "Error in matrix::operator -: matrices have different sizes (" << numRows <
  << " x " << m.numCols << ")!\n";
exit(0);
}
matrix temp(numRows, numCols);
for(int i = 0; i < numRows; i++)
for(int j = 0; j < numCols; j++)
temp[i][j] = values[i][j] - m.values[i][j];
return temp;
}

void matrix::operator -= (matrix m)
{
if(numRows != m.numRows || numCols != m.numCols)
{
printf("Error in matrix::operator -=: matrices have sizes %d x %d and %d x %d (not
exit(0);
}
for(int i = 0; i < numRows; i++)
for(int j = 0; j < numCols; j++)
values[i][j] -= m.values[i][j];
}

matrix matrix::operator * (double scalar)
{
matrix m = (*this);
m.scale(scalar);
return m;
}

void matrix::operator *= (double scalar)
{
scale(scalar);
}

matrix matrix::operator * (matrix m)
{
if(numCols != m.numRows)
{
```

```
printf("Error in matrix:: operator *=: columns of A do not equal rows of B (%d and
exit(0);
}
matrix temp(numRows, m.numCols);
multiply(m, temp);
return temp;
}

void matrix::operator *= (matrix m)
{
if(numCols != m.numRows)
{
printf("Error in matrix:: operator *=: columns of A do not equal rows of B (%d and
exit(0);
}
matrix temp(numRows, m.numCols);
multiply(m, temp);
(*this) = temp;
}

vector matrix::operator * (vector v)
{
if(v.length != numCols)
{
cout << "Error in matrix::* by vector: matrix has size " << numRows << " x " << num
exit(0);
}
vector temp(numRows);
for(int i = 0; i < numRows; i++)
temp.values[i] = v.dotProduct(values[i]);
return temp;
}

const int matrix::getRows()
{
return numRows;
}

const int matrix::getCols()
{
return numCols;
}
```

```
vector& matrix::operator [] (int index)
{
return values[index];
}

matrix matrix::submatrix(int row, int col, int rowSize, int colSize)
{
if(row + rowSize > numRows || col + colSize > numCols)
{
cout << "Error in matrix::submatrix: matrix has size  " << numRows << " x " << numC
  << ") to (" << row + rowSize - 1 << ", " << col + colSize - 1 << ") is requested!
exit(0);
}
matrix temp(rowSize, colSize);
for(int i = 0; i < rowSize; i++)
for(int j = 0; j < colSize; j++)
temp.values[i][j] = values[i + row][j + col];
return temp;
}

matrix matrix::identity(int size)
{
matrix temp(size, size, 0);
for(int i = 0; i < size; i++)
temp.values[i][i] = 1;
return temp;
}

void matrix::multiply(matrix m, matrix& result) //matrix x matrix
{
if(numCols != m.numRows)
{
printf("Error in matrix mult by matrix: 1's columns (%d) not equal to 2's rows (%d)
exit(0);
}
result.resize(numRows, m.numCols);
for(int i = 0; i < numRows; i++)
for(int j = 0; j < m.numCols; j++)
{
result.values[i][j] = 0;
for(int k = 0; k < numCols; k++)
```

```
result.values[i][j] += values[i][k] * m.values[k][j];
}
}

void matrix::multiply(vector v, vector& result) //matrix x vert vector
{
if(numCols != v.length)
{
printf("Error in matrix mult by vector: matrix columns (%d) not equal to vector len
exit(0);
}
result.resize(numRows);
for(int i = 0; i < numRows; i++)
result.values[i] = v.dotProduct(values[i]);
}

void matrix::fill(double datum)
{
for(int i = 0; i < numRows; i++)
for(int j = 0; j < numCols; j++)
values[i][j] = datum;
}

void matrix::scale(double scalar)
{
for(int i = 0; i < numRows; i++)
for(int j = 0; j < numCols; j++)
values[i][j] *= scalar;
}

void matrix::transpose()
{
matrix temp(numCols, numRows);
for(int i = 0; i < numRows; i++)
for(int j = 0; j < numCols; j++)
temp.values[j][i] = values[i][j];
delete[] values;
values = temp.values;
int i = numRows;
numRows = numCols;
numCols = i;
}
```

```
void matrix::resize(int newRows, int newCols)
{
if(!numRows && !numCols)
{
values = new vector[numRows = newRows];
for(int i = 0; i < numRows; i++)
values[i].resize(numCols = newCols);
return;
}
int minRows = min(numRows, newRows), minCols = min(numCols, newCols);
matrix temp(newRows, newCols);
for(int i = 0; i < minRows; i++)
for(int j = 0; j < minCols; j++)
temp.values[i][j] = values[i][j];
delete[] values;
values = temp.values;
numRows = newRows;
numCols = newCols;
}

void matrix::swapRows(int r1, int r2)
{
for(int i = 0; i < numCols; i++)
swap(values[r1][i], values[r2][i]);
}

matrix matrix::augment(matrix m)
{
if(m.numRows != numRows)
{
cout << "Error in matrix::augment: matrices have different row sizes (" << numRows
exit(0);
}
int cols = numCols;
matrix temp(numRows, numCols + m.numCols);
for(int i = 0; i < numRows; i++)
{
for(int j = 0; j < cols; j++)
{
temp.values[i][j] = values[i][j];
temp.values[i][j + cols] = m.values[i][j];
```

```
}
}
return temp;
}

matrix matrix::rref()
{
matrix temp = (*this);
int colNum = 0;
for(int i = 0; i < numRows; i++)
{
colNum = temp.firstNonzeroColumn(colNum, i);
if(colNum == -1) return temp;
if(temp.values[i][colNum] == 0)
temp.swapRows(i, temp.firstNonzeroColumnEntry(colNum));
temp[i].scale(1 / temp.values[i][colNum]);
for(int j = 0; j < numRows; j++)
if(j != i)
temp.values[j] -= temp.values[i] * temp.values[j][colNum];
colNum++;
}
return temp;
}

double matrix::trace() //trace: sum of elements on diagonal of square matrix
{
if(numRows != numCols)
{
printf("Error in matrix::trace(): matrix is not square (size is %d by %d)!\n", numR
exit(0);
}
double value = 0;
for(int i = 0; i < numRows; i++)
value += values[i][i];
return value;
}

double matrix::det() //determinant of square matrix
{
if(numRows != numCols)
{
printf("Error in matrix::det(): matrix is not square (size is %d by %d)!\n", numRow
```

```
exit(0);
}
if(numRows == 1) return values[0][0];
else if(numRows == 2) return values[0][0] * values[1][1] - values[1][0] * values[0]
else if(numRows == 3) //top-row expansion
{
return values[0][0] * (values[1][1] * values[2][2] - values[2][1] * values[1][2])
 - values[0][1] * (values[1][0] * values[2][2] - values[2][0] * values[1][2])
 + values[0][2] * (values[1][0] * values[2][1] - values[2][0] * values[1][1]);
}
else
{
printf("Error in matrix::det(): determinant not implemented for 4x4 or larger matri
exit(0);
}
}

bool matrix::isSymmetric()
{
if(numRows != numCols) return false;
for(int i = 1; i < numRows; i++)
for(int j = 0; j < i; j++)
if(values[i][j] != values[j][i])
return false;
return true;
}

int matrix::firstNonzeroColumn(int startCol, int startRow) //return col num
{
for(int i = startCol; i < numCols; i++)
for(int j = startRow; j < numRows; j++)
if(values[j][i] != 0)
return i;
return -1;
}

int matrix::firstNonzeroColumnEntry(int col) //return row num
{
for(int i = 0; i < numRows; i++)
if(values[i][col] != 0)
return i;
return -1;
```

```
}

void matrix::print()
{
printf("\n");
for(int i = 0; i < numRows; i++)
{
for(int j = 0; j < numCols; j++)
printf("%.8f  ", values[i][j]);
printf("\n");
}
}

/* Construct a unit quaternion from rotation matrix. Assumes matrix is
* used to multiply column vector on the left: v_new = r * v_old. Works
* correctly for right-handed coordinate system and right-handed rotations.
* Translation and perspective components ignored. */
//R = [  1 - 2(y^2 + z^2)  2(xy - wz)        2(xz + wy)         ] * a scalar
// [  2(xy + wz)        1 - 2(x^2 + z^2)  2(yz - wx)         ]
// [  2(xz - wy)        2(yz + wx)        1 - 2(x^2 + y^2)   ]
// (treat matrix as though it were homogeneous and 4 x 4)

// r[0][1] + r[1][0] = 4xy                know xy
// r[0][2] + r[2][0] = 4xz                know xz => y/z = xy/xz
// r[1][2] + r[2][1] = 4yz                know yz => y = sqrt(yz * y/z) => know x,
// r[0][1] = 2xy - 2wz                    know w
quaternion matrix::createRotationQuaternion() //calculate x, y, z, w of a rotation
{
if(numRows != 3 || numCols != 3)
{
cout << "Error in matrix::create rotation quaternion: matrix wrong size (" << numRo
exit(0);
}
quaternion q;
double xy = (values[0][1] + values[1][0]) / 4, xz = (values[0][2] + values[2][0]) /
q[Y] = sqrt(yz * ydz);
q[X] = xy / q[Y];
q[Z] = yz / q[Y];
q[W] = (values[0][1] - 2 * xy) / (2 * q[Z]);
q.normalize();
return q;
}
```

```
/* Replaces given matrix with its inverse.  Uses LU decomposition, then finds
inverse column-by-column.  Taken from "Numerical Recipies in C."  Currently
handles up to a 10x10 Matrix, although this can be easily changed.
Returns whether successful. */
bool matrix::invert()
{
if(numRows != numCols)
{
printf("Error in matrix::invert(): matrix is not square (size is %d by %d)!\n", num
exit(0);
}
int size = numRows;
(*this) = augment(identity(size));
rref();
(*this) = submatrix(0, size, size, size);
return true;
}

/*************************************************************************
**  Name:LUdecomp
**  Descrip: Replaces m with its LU decomposition.
**Taken from "Numerical Recipies in C," p.43
**  Returns: 0 on success, -1 on error.
*************************************************************************/
int matrix::LUdecomp(int* index, double* d)
{
int imax, i, j, k;
double big, dum, sum, temp;
vector vv(numRows);
(*d) = 1;
for(i = 0; i < numRows; i++)
{
big = 0.0;
for(j = 0; j < numRows; j++)
if((temp = fabs(values[j][i])) > big)
big = temp;
if(big == 0)
{
printf("LUdecomp: singular Matrix!\n");
return -1;
}
```

```
vv[i] = 1 / big;
}
for(j = 0; j < numRows; j++)
{
for(i = 0; i < j; i++)
{
sum = values[j][i];
for(k = 0; k < i; k++)
sum -= values[k][i] * values[j][k];
values[j][i] = sum;
}
big = 0;
for(i = j; i < numRows; i++)
{
sum = values[j][i];
for(k = 0; k < j; k++)
sum -= values[k][i] * values[j][k];
values[j][i] = sum;
if((dum = vv[i] * fabs(sum)) >= big)
{
big = dum;
imax = i;
}
}
if(j != imax)
{
for(k = 0; k < numRows; k++)
swap(values[k][imax], values[k][j]);
(*d) *= -1;
vv[imax] = vv[j];
}
index[j] = imax;
if(values[j][j] == 0)
{
printf("LUdecomp: hit singular pivot!\n");
return -1;
}
if(j != numRows - 1)
{
dum = 1 / values[j][j];
for(i = j + 1; i < numRows; i++)
values[j][i] *= dum;
```

```
}
}
return 0;
}

/*****************************************************************************
**  Name:LUbacksub
**  Descrip: Solves a set of linear equations mX = B where matrix m is the LU
**decomposition of the original matrix m.
**Taken from "Numerical Recipies in C," p.44
**  Returns: 0 on success, -1 on error.
*****************************************************************************/
int matrix::LUbacksub(int* index, double* b)
{
int i, j, ii = -1, ip;
double sum;
for(i = 0; i < numRows; i++)
{
ip = index[i];
sum = b[ip];
b[ip] = b[i];
if(ii > -1)
for(j = ii; j <= i - 1; j++)
sum -= values[j][i] * b[j];
else if(sum) ii = i;
b[i] = sum;
}
for(i = numRows - 1; i >= 0; i--)
{
sum = b[i];
for(j = i + 1; j < numRows; j++)
sum -= values[j][i] * b[j];
b[i] = sum / values[i][j];
}
return 0;
}

void matrix::svd(matrix& u, vector& w, matrix& v)
{
//initialize output
const int minDim = min(numRows, numCols);
u.resize(numRows, numRows);
```

```
w.resize(minDim);
v.resize(numCols, numCols);
//parameters to dsvdc:
//input
double* a = new double[numRows * numCols]; //will hold the data in *this
integer nRows = numRows, nCols = numCols;
double* workArray = new double[numRows];
integer jobCodes = 11; //return right sing vecs in V, left in U
//return
double* singularValues = new double[min(numRows + 1, numCols)];
double* returnValues = new double[numCols]; //not useful
double* leftSingularVectors = new double[numRows * numRows]; //not used if jobCodes
double* rightSingularVectors = new double[numCols * numCols];
integer returnErrorCode; //should be 0 if all OK
//initialize a
for(int i = 0; i < numCols; i++)
for(int j = 0; j < numRows; j++)
a[i * numRows + j] = values[j][i];
//call Linpack routine
/*cout << "parameters are: " << a << ", " << nRows << ", " << nCols << ", " << sing
  << ", " << returnValues << ", " << leftSingularVectors << ", " << rightSingularVe
  << ", " << workArray << ", " << jobCodes << ", " << returnErrorCode << endl;*/
dsvdc_(a, &nRows, &nRows, &nCols, singularValues, returnValues, leftSingularVectors
rightSingularVectors, &nCols, workArray, &jobCodes, &returnErrorCode);
//check for errors
if(returnErrorCode != 0)
{
printf("Error in matrix::svd(): SVD could not be computed!\n");
printf("%d singular values were not found\n", returnErrorCode);
exit(0);
}
//decode info
for(int i = 0; i < numRows; i++)
for(int j = 0; j < numRows; j++)
u[i][j] = leftSingularVectors[j * numRows + i];
for(int i = 0; i < minDim; i++)
w[i] = singularValues[i];
for(int i = 0; i < numCols; i++)
for(int j = 0; j < numCols; j++)
v[i][j] = rightSingularVectors[j * numCols + i];
}
```

quaternion.h: my quaternion class, written to interact with matrivec to make parameterization of rotations easy

```
#include <stdio.h>
#include "matrivec.h"

const int X = 0, Y = 1, Z = 2, W = 3;

class quaternion
{
public:

quaternion()
{};
quaternion(double a, double b, double c, double d)
{vals[X] = a; vals[Y] = b; vals[Z] = c; vals[W] = d;};
~quaternion()
{};

double norm();
double& operator [] (int index);

quaternion operator * (quaternion q);
void operator = (quaternion q);
void normalize();

matrix createRotationMatrix();
void makeRotationQuaternion(double x, double y, double z, double angle);
void createFromRotationMatrix(matrix& r);
void createRodrigues(double* rodrigues);
void createFromRodrigues(const double* rodrigues);

void print();

double vals[4];
};

quaternion quaternion::operator * (quaternion q)
{
quaternion qq;
```

```
qq.vals[W] = vals[W] * q.vals[W] - vals[X] * q.vals[X] - vals[Y] * q.vals[Y] - vals
qq.vals[X] = vals[W] * q.vals[X] + vals[X] * q.vals[W] + vals[Y] * q.vals[Z] - vals
qq.vals[Y] = vals[W] * q.vals[Y] + vals[Y] * q.vals[W] + vals[Z] * q.vals[X] - vals
qq.vals[Z] = vals[W] * q.vals[Z] + vals[Z] * q.vals[W] + vals[X] * q.vals[Y] - vals
return qq;
}

void quaternion::operator = (quaternion q)
{
for(int i = 0; i < 4; i++)
vals[i] = q.vals[i];
}

double& quaternion::operator [] (int index)
{
return vals[index];
}

double quaternion::norm()
{
return vals[X] * vals[X] + vals[Y] * vals[Y] + vals[Z] * vals[Z] + vals[W] * vals[W
}

void quaternion::makeRotationQuaternion(double x, double y, double z, double angle)
{
   double length, sinA;
   /* normalize vector */
   length = sqrt(x * x + y * y + z * z);
   /* if zero vector passed in, just return identity quaternion    */
   if(length < 1e-12)
   {
vals[X] = vals[Y] = vals[Z] = 0;
      vals[W] = 1;
      return;
   }
   sinA = sin(angle / 2);
   vals[W] = cos(angle / 2);
   vals[X] = sinA * x / length;
   vals[Y] = sinA * y / length;
   vals[Z] = sinA * z / length;
}
```

```
/*****************************************************************************
 * q_normalize-  normalize quaternion.  src and dest can be same
 *****************************************************************************/
void quaternion::normalize()
{
if(norm() < 1e-12)
{
cout << "Error in quaternion::norm: quaternion has norm 0!\n";
exit(0);
}
double normFactor = 1 / sqrt(vals[X] * vals[X] + vals[Y] * vals[Y] + vals[Z] * vals
for(int i = 0; i < 4; i++)
   vals[i] *= normFactor;
}

/* Construct rotation matrix from (possibly non-unit) quaternion.
* Assumes matrix is used to multiply column vector on the left:
* vnew = mat vold. Works correctly for right-handed coordinate system
* and right-handed rotations. */
//R = [  1 - 2(y^2 + z^2)  2(xy - wz)        2(xz + wy)         ] * a scalar
// [  2(xy + wz)        1 - 2(x^2 + z^2)  2(yz - wx)         ]
// [  2(xz - wy)        2(yz + wx)        1 - 2(x^2 + y^2)   ]
// (treat matrix as though it were homogeneous and 4 x 4)
matrix quaternion::createRotationMatrix()
{
normalize();
matrix r(3, 3);
r[X][X] = 1 - 2 * (vals[Y] * vals[Y] + vals[Z] * vals[Z]);
r[X][Y] = 2 * (vals[X] * vals[Y] + vals[W] * vals[Z]);
r[X][Z] = 2 * (vals[X] * vals[Z] - vals[W] * vals[Y]);
r[Y][X] = 2 * (vals[X] * vals[Y] - vals[W] * vals[Z]);
r[Y][Y] = 1 - 2 * (vals[X] * vals[X] + vals[Z] * vals[Z]);
r[Y][Z] = 2 * (vals[Y] * vals[Z] + vals[W] * vals[X]);
r[Z][X] = 2 * (vals[X] * vals[Z] + vals[W] * vals[Y]);
r[Z][Y] = 2 * (vals[Y] * vals[Z] - vals[W] * vals[X]);
r[Z][Z] = 1 - 2 * (vals[X] * vals[X] + vals[Y] * vals[Y]);
return r;
}

/* Construct a unit quaternion from rotation matrix. Assumes matrix is
* used to multiply column vector on the left: v_new = r * v_old. Works
* correctly for right-handed coordinate system and right-handed rotations.
```

```
* Translation and perspective components ignored. */
//R = [  1 - 2(y^2 + z^2)  2(xy - wz)        2(xz + wy)         ] * a scalar
// [  2(xy + wz)        1 - 2(x^2 + z^2)  2(yz - wx)         ]
// [  2(xz - wy)        2(yz + wx)        1 - 2(x^2 + y^2)   ]
// (treat matrix as though it were homogeneous and 4 x 4)

// r[0][1] + r[1][0] = 4xy                know xy
// r[0][2] + r[2][0] = 4xz                know xz => y/z = xy/xz
// r[1][2] + r[2][1] = 4yz                know yz => y = sqrt(yz * y/z) => know x,
// r[0][1] = 2xy - 2wz                    know w

void quaternion::createFromRotationMatrix(matrix& r)
{  //calculate x, y, z, w of a rotation quaternion from the corresponding rotation
if(r.numRows != 3 || r.numCols != 3)
{
cout << "Error in quaternion::create from rotation matrix: matrix wrong size ("
  << r.numRows << " x " << r.numCols << ")!\n";
exit(0);
}
double xy = (r[0][1] + r[1][0]) / 4, xz = (r[0][2] + r[2][0]) / 4, ydz = xy / xz, y
vals[Y] = sqrt(yz * ydz);
vals[X] = xy / vals[Y];
vals[Z] = yz / vals[Y];
vals[W] = (r[0][1] - 2 * xy) / (2 * vals[Z]);
}

void quaternion::createRodrigues(double* rodrigues) //q is [v sin theta, cos theta]
{
//theta is angle between vector being rotated and axis of rotation
double sinTheta = sqrt(vals[0] * vals[0] + vals[1] * vals[1] + vals[2] * vals[2]),
for(int i = 0; i < 3; i++)
rodrigues[i] = vals[i] * theta / sinTheta;
}

void quaternion::createFromRodrigues(const double* rodrigues)
{
//theta is angle between vector being rotated and axis of rotation
double theta = sqrt(rodrigues[0] * rodrigues[0] + rodrigues[1] * rodrigues[1] + rod
vals[3] = cos(theta);
for(int i = 0; i < 3; i++)
vals[i] = rodrigues[i] * sin(theta) / theta;
}
```

```
void quaternion::print()
{
printf("{ %.8lf, %.8lf, %.8lf, %.8lf }", vals[X], vals[Y], vals[Z], vals[W]);
}
```