ECE661 HW5

Chad Aeschliman

10/23/08

1 Problem Statement

The problem is to determine the homography between two similar images. RANSAC is to be used to prune a crude set of correspondences down to good set and to provide an initial homography. Levenberg-Marquardt minimization is then used to modify this homography in order to minimize the reprojection error.

2 Solution

The implementation of the initial feature detection and correspondence matching as well as the RANSAC refinement closely follow the textbook and were covered in preceding homeworks so they will not be discussed again. The output of RANSAC is an initial guess at the homography between the two images as well as a set of good correspondences (the set of inliers) between the images. Given this information, the idea is to refine the estimated homography until it in some sense optimally explains the correspondences. Levenberg-Marquardt (LM) minimization can be used to do this.

The first step is to define a measure of optimality (cost function). The definition for the cost was taken from the textbook (Equation 4.8):

$$C = \sum_i d(\mathbf{x}_i, \hat{\mathbf{x}}_i)^2 + d(\acute{\mathbf{x}}_i, \hat{\mathrm{H}}\hat{\mathbf{x}}_i)^2$$

where $(\mathbf{x}_i, \acute{\mathbf{x}}_i)$ is a corresponding pair of points from RANSAC, the parameter $\hat{\mathbf{x}}_i$ is a floating point in the domain image, $\hat{\mathrm{H}}$ is an estimate of the homography between the images, and the distance function $d(\cdot)$ is taken to be the euclidean distance.

Note that C is minimized by adjusting the floating points $\hat{\mathbf{x}}_i$ and the estimated homography $\hat{\mathrm{H}}$. Intuitively, during the optimization each floating point and its corresponding point (determined by applying the estimated homography $\hat{\mathrm{H}}$) attempt to move as close as possible to one of the corresonding pairs identified by RANSAC. This is achieved by adjusting the location of each floating point and also $\hat{\mathrm{H}}$. The advantage of this cost function is that it methodically handles the error in the coordinates of the identified correspondences in both images. This is in contrast to the symmetric transfer error cost function which assumes that the coordinates of the correspondences are known exactly in the first and then second image in turn.

LM was used to minimize the cost function using the open source c/c++ implementation *lmfit* [1]. This implementation computes a numerical approximation of the Jacobian, which greatly simplifies the code and is still fast (for the images which were tested , LM converged within about one second). Hence, the implementation is primarily limited to just the definition of a cost function which parses a vector of parameters and returns the x and y distances between each floating correspondence and the associated RANSAC correspondence.

3 Results

The solution method outlined in the preceding section was tested on several pairs of images. For each pair, in order to evaluate the accuracy of the estimated homography each image was remapped using either the

[1] http://www.messen-und-deuten.de/lmfit/index.html

homography or its inverse to match the other image. The magnitude of the pixel by pixel difference was then computed. Several figures are included at the end of this report which show the remapped images and the difference images. For each figure, the top two images are the input images, in the middle are the remapped images (placed under the image they should match), and the bottom images are the difference images. Note that some pairs of images have significant brightness differences which tend to dominate the difference images and makes them difficult to analyze.

4 Code

4.1 hw5.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include <limits.h>
#include "opencv/cv.h"
#include "opencv/highgui.h"
#include "lmfit/lmmin.h"
#include "hw5.h"

#define VERBOSE_PRINTING 0

// utility function which prints out a matrix
void printMatrix(CvMat* M, const char* name)
{
    int indent_size;
    int i, j;
    int rows = M->rows;
    int cols = M->cols;

    // print the matrix name
    indent_size = printf("%s = ", name);

    // print out the matrix
    for (i = 0; i < rows; i++)
    {
        // start of a row
        if (i==0)
            printf("[");
        for (j = 0; j < cols; j++)
        {
            if (j > 0)
            {
                printf(",");
            }
            printf("%11.5lg", cvmGet(M, i, j));
        }
        if (i==rows-1)
            printf("]\n");
        else
            printf("\n");

        // indent the next line
        for (j = 0; j < indent_size; j++)
        {
            printf(" ");
        }
    }
    printf("\n");
}

// This is the function which is to be minimized. In particular,
// the LM algorithm attempts to force the square of the entries
// in fvec to 0.
void lm_evaluate_custom(double *par, int m_dat, double *fvec,
          void *data, int *info)
{
    int i;
    CvMat* H = cvCreateMat(3,3,CV_64FC1);;
    CvMat* image1_coord = cvCreateMat(3,1,CV_64FC1);
    CvMat* image2_coord = cvCreateMat(3,1,CV_64FC1);
    optimization_data *opt_data = (optimization_data *)data;
    int number_of_inliers = opt_data->number_of_inliers;
```

```
    // first fill in the homography with the parameters in par
    for (i=0;i<9;i++)
    {
        cvmSet(H,i/3,i%3,par[i]);
    }

    // compute the distance between the estimated "true" coordinates in image1
    // and image2 and the original coordinates
    cvmSet(image2_coord,2,0,1.0);
    for (i=0;i<number_of_inliers; i++)
    {
        double dx = par[9+2*i]-opt_data->inlier_set2[i].x;
        double dy = par[9+2*i+1]-opt_data->inlier_set2[i].y;
        fvec[4*i] = dx;
        fvec[4*i+1] = dy;
        cvmSet(image2_coord,0,0,par[9+2*i]);
        cvmSet(image2_coord,1,0,par[9+2*i+1]);
        cvMatMul(H,image2_coord,image1_coord);
        dx = cvmGet(image1_coord,0,0)/cvmGet(image1_coord,2,0)-opt_data->inlier_set1[i].x;
        dy = cvmGet(image1_coord,1,0)/cvmGet(image1_coord,2,0)-opt_data->inlier_set1[i].y;
        fvec[4*i+2] = dx;
        fvec[4*i+3] = dy;
    }
}

// Prints out some status information during the optimization
void lm_print_custom(int n_par, double *par, int m_dat, double *fvec,
            void *data, int iflag, int iter, int nfev)
{
    if (iflag == 0)
    {
        printf("starting minimization\n");
    }
    else if (iflag == -1)
    {
        printf("terminated after %d evaluations\n", nfev);
    }

#if VERBOSE_PRINTING
    int i;
    printf("  errors: ");
    for (i = 0; i < m_dat; ++i)
        printf(" %12g", fvec[i]);
    printf("\n");
#endif

}

// First computes an initial set of correspondences between two images
// using Harris corner detection and NCC.  Then refines these correspondences
// using RANSAC.  Finally, uses all of the inliers identified by RANSAC
// to compute a homography between the images using Levenberg-Marquardt.
int main( int argc, char** argv )
{
    char* filename1;
    char* filename2;

    IplImage* image1;
    IplImage* image2;

    int i;

    int number_of_correspondences;
    CvPoint corners1[MAX_NUM_CORNERS];
    CvPoint corners2[MAX_NUM_CORNERS];

    optimization_data data; //defined in hw5.h

    CvMat *ransac_H = cvCreateMat(3,3,CV_64FC1);
    CvMat *invH = cvCreateMat(3,3,CV_64FC1);

    // attempt to read in the image files
    if (argc >= 3)
    {
        filename1 = argv[1];
        filename2 = argv[2];
    }
    else
    {
        printf("Usage: hw5 filename1 filename2");
        return 1;
```

```
}
if((image1 = cvLoadImage(filename1,1)) == 0)
{
   printf("Could not read file 1!");
   return 2;
}
if((image2 = cvLoadImage(filename2,1)) == 0)
{
   printf("Could not read file 2!");
   return 2;
}

// compute some correspondences using the Harris corner detector and
// NCC using a simplified version of the code from hw3 (code is in
// compute_correspondences.c).
compute_base_correspondences(image1,image2,
                              corners1, corners2,
                              &number_of_correspondences);
printf("%d base corresponences:\n",number_of_correspondences);

// run RANSAC on these base correspondences to get an inlier set
// and an initial guess for the homography (code is in ransac.c)
compute_ransac_correspondences(corners1, corners2, number_of_correspondences,
                               data.inlier_set1, data.inlier_set2, &data.number_of_inliers,
                                   ransac_H);
printf("%d inlier corresponences:\n",data.number_of_inliers);
printMatrix(ransac_H, "Ransac_H");

// With an initial homography from RANSAC and a set of inliers, we now
// need to run the LM algorithm to refine this homography.
CvMat* input_coord = cvCreateMat(3,1,CV_64FC1);
CvMat* output_coord = cvCreateMat(3,1,CV_64FC1);
cvmSet(input_coord,2,0,1.0);
int n_p = 9+2*data.number_of_inliers;
double *p = malloc(n_p*sizeof(double));
for (i=0;i<9;i++)
{
   p[i] = cvmGet(ransac_H,i/3,i%3);
}
cvInvert(ransac_H,invH,CV_LU);
for (i=0;i<data.number_of_inliers;i++)
{
   cvmSet(input_coord,0,0,data.inlier_set1[i].x);
   cvmSet(input_coord,1,0,data.inlier_set1[i].y);
   cvMatMul(invH,input_coord,output_coord);
   p[9+2*i] = cvmGet(output_coord,0,0)/cvmGet(output_coord,2,0);
   p[9+2*i+1] = cvmGet(output_coord,1,0)/cvmGet(output_coord,2,0);
}

// auxiliary settings:
lm_control_type control;
lm_initialize_control(&control);
control.maxcall = 200000;
control.ftol = 1.0e-16;
control.xtol = 1.0e-16;
control.gtol = 1.0e-16;
control.stepbound = 10.0;

// perform the Levenberg-Marquardt minimization using the lmfit library
lm_minimize(4*data.number_of_inliers, n_p, p, lm_evaluate_custom, lm_print_custom,
  &data, &control);
printf("\nLM Exit String: %s\n\n",lm_infmsg[control.info]);

// form the error minimizing homography and print it out
CvMat* minimized_H = cvCreateMat(3,3,CV_64FC1);;
for (i=0;i<9;i++)
{
   cvmSet(minimized_H,i/3,i%3,p[i]);
}
free(p);
printMatrix(minimized_H,"Minimized_H");

// use the new homography to map the range image to the domain and
// vice versa.
// compute a corrected range image by applying H^-1 to a grid of points in
// the world coordinate system
cvInvert(minimized_H,invH,CV_LU);
IplImage* corrected_image = cvCreateImage(cvGetSize(image1),8,3);
cvZero(corrected_image);
int j,k;
for (i=0; i<corrected_image->width; i++)
```

```
{
   cvmSet(input_coord,0,0,(double)i);
   for (j=0; j<corrected_image->height; j++)
   {
      double xi,yi,fx,fy;
      cvmSet(input_coord,1,0,(double)j);

      // compute the associated image coordinate
      cvMatMul(invH,input_coord,output_coord);
      xi = cvmGet(output_coord,0,0)/cvmGet(output_coord,2,0);
      yi = cvmGet(output_coord,1,0)/cvmGet(output_coord,2,0);

      // if outside of the image then move on
      if (xi<0||yi<0||xi>=(image2->width-1)||yi>=(image2->height-1))
      {
         continue;
      }

      // compute the fractional component of the image coord.
      fx = xi - (int)xi;
      fy = yi - (int)yi;

      // compute the pixel value using linear interpolation
      for (k=0;k<3;k++)
      {
         double value = 0;
         value += (1.0-fx)*(1.0-fy)*((uchar*)(image2->imageData + image2->widthStep*(int)yi))
              [((int)xi)*3+k];
         value += (1.0-fx)*fy*((uchar*)(image2->imageData + image2->widthStep*(int)(yi+1)))[((
              int)xi)*3+k];
         value += fx*(1.0-fy)*((uchar*)(image2->imageData + image2->widthStep*(int)yi))[((int)(
              xi+1))*3+k];
         value += fx*fy*((uchar*)(image2->imageData + image2->widthStep*(int)(yi+1)))[((int)(xi
              +1))*3+k];
        ((uchar*)(corrected_image->imageData + corrected_image->widthStep*j))[i*3+k] = value;
      }
   }
}

// save corrected image
char new_filename[FILENAME_MAX];
strcpy(new_filename,filename2);
sprintf(new_filename+strlen(new_filename)-4,"_r2d_new.png");
cvSaveImage(new_filename,corrected_image);

// create and save the difference image
IplImage* difference_image = cvCreateImage(cvGetSize(image1),8,3);
cvAbsDiff(image1, corrected_image, difference_image);
strcpy(new_filename,filename2);
sprintf(new_filename+strlen(new_filename)-4,"_diff_r2d_new.png");
cvSaveImage(new_filename,difference_image);
cvReleaseImage(&corrected_image);
cvReleaseImage(&difference_image);

// now do the other direction
corrected_image = cvCreateImage(cvGetSize(image2),8,3);
cvZero(corrected_image);
cvmSet(input_coord,2,0,1.0);
for (i=0; i<corrected_image->width; i++)
{
   cvmSet(input_coord,0,0,(double)i);
   for (j=0; j<corrected_image->height; j++)
   {
      double xi,yi,fx,fy;
      cvmSet(input_coord,1,0,(double)j);

      // compute the associated image coordinate
      cvMatMul(minimized_H,input_coord,output_coord);
      xi = cvmGet(output_coord,0,0)/cvmGet(output_coord,2,0);
      yi = cvmGet(output_coord,1,0)/cvmGet(output_coord,2,0);

      // if outside of the image then move on
      if (xi<0||yi<0||xi>=(image1->width-1)||yi>=(image1->height-1))
      {
         continue;
      }

      // compute the fractional component of the image coord.
      fx = xi - (int)xi;
      fy = yi - (int)yi;
```

```
        // compute the pixel value using linear interpolation
        for (k=0;k<3;k++)
        {
          double value = 0;
          value += (1.0-fx)*(1.0-fy)*((uchar*)(image1->imageData + image1->widthStep*(int)yi))
               [((int)xi)*3+k];
          value += (1.0-fx)*fy*((uchar*)(image1->imageData + image1->widthStep*(int)(yi+1)))[((
               int)xi)*3+k];
          value += fx*(1.0-fy)*((uchar*)(image1->imageData + image1->widthStep*(int)yi))[((int)(
               xi+1))*3+k];
          value += fx*fy*((uchar*)(image1->imageData + image1->widthStep*(int)(yi+1)))[((int)(xi
               +1))*3+k];
         ((uchar*)(corrected_image->imageData + corrected_image->widthStep*j))[i*3+k] = value;
        }
      }
    }

    // save corrected image
    strcpy(new_filename,filename2);
    sprintf(new_filename+strlen(new_filename)-4,"_d2r_new.png");
    cvSaveImage(new_filename,corrected_image);

    // create and save the difference image
    difference_image = cvCreateImage(cvGetSize(image2),8,3);
    cvAbsDiff(image2, corrected_image, difference_image);
    strcpy(new_filename,filename2);
    sprintf(new_filename+strlen(new_filename)-4,"_diff_d2r_new.png");
    cvSaveImage(new_filename,difference_image);

    return 0;
}
```

4.2 hw5.h

```
#ifndef HW5_H_
#define HW5_H_

#define MAX_NUM_CORNERS 2000

void compute_base_correspondences(IplImage* image1, IplImage* image2,
       CvPoint corners1[MAX_NUM_CORNERS], CvPoint corners2[MAX_NUM_CORNERS],
       int *number_of_correspondences);

void compute_ransac_correspondences(CvPoint corners1[MAX_NUM_CORNERS],
       CvPoint corners2[MAX_NUM_CORNERS],
       int number_of_correspondences,
       CvPoint inlier_set1[MAX_NUM_CORNERS],
       CvPoint inlier_set2[MAX_NUM_CORNERS],
       int *number_of_inliers,
       CvMat *best_H);

typedef struct {
    int number_of_inliers;
    CvPoint inlier_set1[MAX_NUM_CORNERS];
    CvPoint inlier_set2[MAX_NUM_CORNERS];
} optimization_data;

#endif /* HW5_H_ */
```

4.3 compute_correspondences.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include "opencv/cv.h"
#include "opencv/highgui.h"
#include "hw5.h"

#define DIR_X 0
#define DIR_Y 1

#define W 5
#define THRESHOLD_EIG 25

#define MATCH_W 9
#define THRESHOLD_NCC 0.7

// compute the 3x3 Sobel gradient of a grayscale image
```

```
void computeSobelGradient(IplImage* input, IplImage* output, int direction)
{
   short sobel[3][3];
   int i,j,i2,j2;
   short temp;

   // fill in sobel matrix
   if (direction==DIR_X)
   {
      sobel[0][0] = -1;
      sobel[1][0] = -2;
      sobel[2][0] = -1;
      sobel[0][1] = 0;
      sobel[1][1] = 0;
      sobel[2][1] = 0;
      sobel[0][2] = 1;
      sobel[1][2] = 2;
      sobel[2][2] = 1;
   }
   else
   {
      sobel[0][0] = -1;
      sobel[0][1] = -2;
      sobel[0][2] = -1;
      sobel[1][0] = 0;
      sobel[1][1] = 0;
      sobel[1][2] = 0;
      sobel[2][0] = 1;
      sobel[2][1] = 2;
      sobel[2][2] = 1;
   }

   // now convolve
   cvZero(output);
   for (i=1;i<(input->width - 1);i++)
   {
      for (j=1;j<(input->height - 1);j++)
      {
         temp = 0;
         for (i2=-1;i2<=1;i2++)
         {
            for (j2=-1;j2<=1;j2++)
            {
               temp += sobel[j2+1][i2+1]*(short)((uchar*)(input->imageData + input->widthStep*(j+
                  j2)))[i+i2];
            }
         }
         ((short*)(output->imageData + output->widthStep*j))[i] = temp;
      }
   }
}

// find the corners of an image using Harris method
int find_corners(IplImage* dx, IplImage* dy, short* corners_x, short* corners_y, int*
   corners_value)
{
   int i,j,i2,j2;

   // compute a Gaussian window of the appropriate size
   double sigma = (W*0.5 - 1)*0.3 + 0.8;
   double inv_sigma = 1.0/sigma;
   int kernel[W][W];
   for (i=-W/2;i<(W+1)/2;i++)
   {
      for (j=-W/2;j<(W+1)/2;j++)
      {
         kernel[j+W/2][i+W/2] = (int)(W*2*inv_sigma*exp(-0.5*inv_sigma*inv_sigma*(i*i+j*j)));
      }
   }


   // sum squared gradients over neighborhoods and identify corners.
   int sum_dx2, sum_dy2, sum_dxy;
   double test_value;
   unsigned short corner_count = 0;
   for (i=W/2;i<(dx->width-W/2);i++)
   {
      for (j=W/2;j<(dx->height-W/2);j++)
      {
         // compute local squared gradient sum
         sum_dx2 = 0;
```

```c
            sum_dy2 = 0;
            sum_dxy = 0;
            for (i2=-W/2;i2<(W+1)/2;i2++)
            {
                for (j2=-W/2;j2<(W+1)/2;j2++)
                {
                    short single_dx = ((short*)(dx->imageData + dx->widthStep*(j+j2)))[i+i2];
                    short single_dy = ((short*)(dy->imageData + dy->widthStep*(j+j2)))[i+i2];
                    // Note: scale by 1/(W^2) to ensure no overflow
                    sum_dx2 += kernel[j2+W/2][i2+W/2]*single_dx*single_dx/(W*W*kernel[W/2][W/2]);
                    sum_dy2 += kernel[j2+W/2][i2+W/2]*single_dy*single_dy/(W*W*kernel[W/2][W/2]);
                    sum_dxy += kernel[j2+W/2][i2+W/2]*single_dx*single_dy/(W*W*kernel[W/2][W/2]);
                }
            }
            sum_dx2 = sum_dx2/(256);
            sum_dy2 = sum_dy2/(256);
            sum_dxy = sum_dxy/(256);

            // now compute whether or not this is a corner
            // and decide if we should keep it.
            double trace = sum_dx2+sum_dy2;
            test_value = 0.5*(trace - sqrt(trace*trace - 4*(sum_dx2*sum_dy2 - sum_dxy*sum_dxy)));
            if (test_value > THRESHOLD_EIG)
            {
                // meets the threshold, now check its neighbors
                char should_use = 1;
                int index = -1;
                for (i2=0;i2<corner_count;i2++)
                {
                    if (i-corners_x[i2] <= W/2 && corners_x[i2]-i <= W/2 &&
                        j-corners_y[i2] <= W/2 && corners_y[i2]-j <= W/2)
                    {
                        // the corner with index i2 is a near neighbor so compare
                        if (test_value > corners_value[i2])
                        {
                            // replace the other corner
                            should_use = 1;
                            index = i2;
                            break;
                        }
                        else
                        {
                            // don't use it
                            should_use = 0;
                            break;
                        }
                    }
                }
                if (should_use)
                {
                    // check if we are replacing a neighboring corner
                    if (index < 0)
                    {
                        // add as a new corner if there is room, otherwise we drop it
                        if (corner_count < MAX_NUM_CORNERS)
                        {
                            index = corner_count;
                            corner_count++;
                        }
                    }
                    // store the corner
                    if (index >= 0)
                    {
                        corners_x[index] = i;
                        corners_y[index] = j;
                        corners_value[index] = test_value;
                    }
                }
            }
        }
    }
    return corner_count;
}


// compute the normalized cross correlation of two matrices
double compare_squares_ncc(CvMat* template, CvMat* subimage, double template_norm, double
    subimage_norm)
{
    int i,j;
    int temp_sum = 0;
```

```
    int mean_template = 0;
    int mean_subimage = 0;
    for (i=0;i<template->cols;i++)
    {
        for (j=0;j<template->rows;j++)
        {
            mean_template += ((uchar*)(template->data.ptr + template->step*j))[i];
        }
    }
    mean_template = mean_template/(template->rows*template->cols);
    for (i=0;i<template->cols;i++)
    {
        for (j=0;j<template->rows;j++)
        {
            mean_subimage += ((uchar*)(subimage->data.ptr + subimage->step*j))[i];
        }
    }
    mean_subimage = mean_subimage/(subimage->rows*subimage->cols);
    for (i=0;i<template->cols;i++)
    {
        for (j=0;j<template->rows;j++)
        {
            temp_sum += (((uchar*)(template->data.ptr + template->step*j))[i]-mean_template) * (((
                uchar*)(subimage->data.ptr + subimage->step*j))[i]-mean_subimage);
        }
    }
    return ((double)temp_sum)/(template_norm*subimage_norm);
}


void compute_base_correspondences(IplImage* image1, IplImage* image2, CvPoint corners1[
    MAX_NUM_CORNERS], CvPoint corners2[MAX_NUM_CORNERS], int *number_of_correspondences)
{
    int n, i, j;
    short corners_x[2][MAX_NUM_CORNERS];
    short corners_y[2][MAX_NUM_CORNERS];
    int num_corners[2];
    int match_index_ncc[MAX_NUM_CORNERS];
    int match_count = 0;

    for (n=0;n<2;n++)
    {
        IplImage* image;
        IplImage* gray;

        // convert image to grayscale
        if (n==0)
        {
            image = image1;
        }
        else
        {
            image = image2;
        }
        gray = cvCreateImage( cvGetSize(image), IPL_DEPTH_8U, 1 );
        cvCvtColor( image, gray, CV_BGR2GRAY );

        // compute gradient in the x and y directions
        IplImage* dx = cvCreateImage( cvGetSize(gray), IPL_DEPTH_16S, 1 );
        IplImage* dy = cvCreateImage( cvGetSize(gray), IPL_DEPTH_16S, 1 );
        computeSobelGradient(gray,dx,DIR_X);
        computeSobelGradient(gray,dy,DIR_Y);
        cvReleaseImage(&gray);

        // find the corners
        int corners_value[MAX_NUM_CORNERS];
        num_corners[n] = find_corners(dx, dy, corners_x[n], corners_y[n], &corners_value[n]);
        cvReleaseImage(&dx);
        cvReleaseImage(&dy);
    }

    // determine correspondences
    double template_norm, best_match_ncc;
    int best_match_index_ncc;
    for (i=0;i<num_corners[0];i++)
    {
        CvMat* sub_image;
        CvMat* template;

        // assume no match
        match_index_ncc[i] = -1;
```

```
        // check that the square is inside the image
        if (corners_x[0][i] < MATCH_W/2 ||
            corners_y[0][i] < MATCH_W/2 ||
            corners_x[0][i]+MATCH_W/2 >= image1->width ||
            corners_y[0][i]+MATCH_W/2 >= image1->height)
        {
            continue;
        }

        // get the template sub image
        template = cvCreateMat(MATCH_W,MATCH_W,CV_8UC1);
        cvGetSubRect(image1,template,cvRect(corners_x[0][i]-MATCH_W/2,corners_y[0][i]-MATCH_W/2,
            MATCH_W,MATCH_W));
        template_norm = sqrt(compare_squares_ncc(template,template,1.0,1.0));

        // loop over all possible matches
        best_match_ncc = 0.0;
        for (j=0;j<num_corners[1];j++)
        {
            double match_ncc;

            // check that the square is inside the image
            if (corners_x[1][j] < MATCH_W/2 ||
                corners_y[1][j] < MATCH_W/2 ||
                corners_x[1][j]+MATCH_W/2 >= image2->width ||
                corners_y[1][j]+MATCH_W/2 >= image2->height)
            {
                continue;
            }

            // get the comparison sub_image rectangle
            sub_image = cvCreateMat(MATCH_W,MATCH_W,CV_8UC1);
            cvGetSubRect(image2,sub_image,cvRect(corners_x[1][j]-MATCH_W/2,corners_y[1][j]-MATCH_W/2,
                MATCH_W,MATCH_W));
            double sub_image_norm = sqrt(compare_squares_ncc(sub_image,sub_image,1.0,1.0));

            // perform the matching
            match_ncc = compare_squares_ncc(template, sub_image, template_norm, sub_image_norm);
            cvReleaseMat(&sub_image);

            // compare the results and update if necessary
            if (match_ncc > best_match_ncc)
            {
                best_match_ncc = match_ncc;
                best_match_index_ncc = j;
            }
        }

        // compare best matches to threshold and store
        if (best_match_ncc >= THRESHOLD_NCC)
        {
            match_index_ncc[i] = best_match_index_ncc;
            corners1[match_count].x = corners_x[0][i];
            corners1[match_count].y = corners_y[0][i];
            corners2[match_count].x = corners_x[1][best_match_index_ncc];
            corners2[match_count].y = corners_y[1][best_match_index_ncc];
            match_count++;
        }
        cvReleaseMat(&template);
        if (match_count==MAX_NUM_CORNERS)
        {
            break;
        }
    }
    *number_of_correspondences = match_count;
}
```

4.4 ransac.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>
#include <float.h>
#include "opencv/cv.h"
#include "opencv/highgui.h"
#include "hw5.h"

#define INLIER_THRESHOLD 1.5
```

```
#define PROBABILITY_REQUIRED 0.99

// Using a base set of correspondences (contained in corners1 and corners2),
// computes various homographies from randomly generated data until one
// is found which produces enough inliers to have a high probability of
// being correct.  The inlier correspondences and the estimated (unrefined)
// homography are returned.
void compute_ransac_correspondences(CvPoint corners1[MAX_NUM_CORNERS],
      CvPoint corners2[MAX_NUM_CORNERS],
      int number_of_correspondences,
      CvPoint best_inlier_set1[MAX_NUM_CORNERS],
      CvPoint best_inlier_set2[MAX_NUM_CORNERS],
      int *number_of_inliers,
      CvMat *best_H)
{
   int i, j;
   int N, sample_count;

   N = INT_MAX;
   sample_count = 0;
   int max_inliers = 0;
   double best_variance = 0;
   CvPoint inlier_set1[MAX_NUM_CORNERS];
   CvPoint inlier_set2[MAX_NUM_CORNERS];
   CvMat *H = cvCreateMat(3,3,CV_64FC1);
   CvMat* image1_coord = cvCreateMat(3,1,CV_64FC1);
   CvMat* image2_coord = cvCreateMat(3,1,CV_64FC1);
   while (N > sample_count)
   {
      CvPoint points1[4];
      CvPoint points2[4];

      // get 4 random correpondences
      i = 0;
      while (i<4)
      {
         int index = rand()%number_of_correspondences;

         // check for duplicate point
         int duplicate = 0;
         for (j=0;j<i;j++)
         {
            if (points1[j].x==corners1[index].x && points1[j].y==corners1[index].y)
            {
               duplicate = 1;
               break;
            }
         }
         if (duplicate)
         {
            continue;
         }

         // add correspondence to list
         points1[i].x = corners1[index].x;
         points1[i].y = corners1[index].y;
         points2[i].x = corners2[index].x;
         points2[i].y = corners2[index].y;
         i++;
      }

      // set up the problem as a matrix equation
      double sol_matrix[64];
      double sol_vector[8];
      for (i=0;i<4;i++)
      {
         sol_matrix[2*i*8+0] = points2[i].x;
         sol_matrix[2*i*8+1] = points2[i].y;
         sol_matrix[2*i*8+2] = 1;
         sol_matrix[2*i*8+3] = 0;
         sol_matrix[2*i*8+4] = 0;
         sol_matrix[2*i*8+5] = 0;
         sol_matrix[2*i*8+6] = -points1[i].x*points2[i].x;
         sol_matrix[2*i*8+7] = -points1[i].x*points2[i].y;

         sol_matrix[(2*i+1)*8+0] = 0;
         sol_matrix[(2*i+1)*8+1] = 0;
         sol_matrix[(2*i+1)*8+2] = 0;
         sol_matrix[(2*i+1)*8+3] = points2[i].x;
         sol_matrix[(2*i+1)*8+4] = points2[i].y;
         sol_matrix[(2*i+1)*8+5] = 1;
```

```
    sol_matrix[(2*i+1)*8+6] = -points1[i].y*points2[i].x;
    sol_matrix[(2*i+1)*8+7] = -points1[i].y*points2[i].y;

    sol_vector[2*i] = points1[i].x;
    sol_vector[2*i+1] = points1[i].y;
}

// solve the problem and copy the solution into H
CvMat *temp = cvCreateMat(8,1,CV_64FC1);
CvMat A;
CvMat B;
cvInitMatHeader(&A,8,8,CV_64FC1,sol_matrix,CV_AUTOSTEP);
cvInitMatHeader(&B,8,1,CV_64FC1,sol_vector,CV_AUTOSTEP);
cvSolve(&A, &B, temp, CV_LU);
for (i=0;i<8;i++)
{
    cvmSet(H,i/3,i%3,cvmGet(temp,i,0));
}
cvmSet(H,2,2,1.0);
cvReleaseMat(&temp);

// H should map points from image 1 into image 2. The H can be
// checked by computing the backprojection error for each point
// correspondence
int num_inliers = 0;
double sum_distance = 0;
double sum_distance_squared = 0;
for (i=0;i<number_of_correspondences;i++)
{
    // first compute the distance between the original coordinate and the backprojected
    // corresponding coordinate
    cvmSet(image2_coord,0,0,corners2[i].x);
    cvmSet(image2_coord,1,0,corners2[i].y);
    cvmSet(image2_coord,2,0,1.0);
    cvMatMul(H,image2_coord,image1_coord);
    double dx = ((double)cvmGet(image1_coord,0,0)/(double)cvmGet(image1_coord,2,0))-corners1[
        i].x;
    double dy = ((double)cvmGet(image1_coord,1,0)/(double)cvmGet(image1_coord,2,0))-corners1[
        i].y;
    double distance = sqrt(dx*dx + dy*dy);

    // compare this distance to a threshold to determine if it is an inlier
    if (distance<INLIER_THRESHOLD)
    {
        // it is an inlier so add it to the inlier set
        inlier_set1[num_inliers].x = corners1[i].x;
        inlier_set1[num_inliers].y = corners1[i].y;
        inlier_set2[num_inliers].x = corners2[i].x;
        inlier_set2[num_inliers].y = corners2[i].y;
        num_inliers++;
        sum_distance += distance;
        sum_distance_squared += distance*distance;
    }
}

// check if this is the best H yet (most inliers, lowest variance in the event
// of a tie)
if (num_inliers >= max_inliers)
{
    // compute variance in case of a tie
    double mean_distance = sum_distance/((double)num_inliers);
    double variance = sum_distance_squared/((double)num_inliers-1.0) - mean_distance*
        mean_distance*(double)num_inliers/((double)num_inliers-1.0);
    if ((num_inliers > max_inliers) || (num_inliers==max_inliers && variance < best_variance)
        )
    {
        // this is the best H so store its information
        best_variance = variance;
        if (best_H)
            cvReleaseMat(&best_H);
        best_H = cvCloneMat(H);
        max_inliers = num_inliers;
        for (i=0;i<num_inliers;i++)
        {
            best_inlier_set1[i].x = inlier_set1[i].x;
            best_inlier_set1[i].y = inlier_set1[i].y;
            best_inlier_set2[i].x = inlier_set2[i].x;
            best_inlier_set2[i].y = inlier_set2[i].y;
        }
    }
}
```

```
        // update N and sample_count using algorithm 4.5
        sample_count++;
        if (num_inliers > 0)
        {
            double epsilon = 1.0 - ((double)num_inliers)/((double)number_of_correspondences);
            double inv_epsilon = 1.0 - epsilon;
            N = (int)(log(1.0-PROBABILITY_REQUIRED)/log(1.0-(inv_epsilon*inv_epsilon*inv_epsilon*
                 inv_epsilon)));
        }
    }
    *number_of_inliers = max_inliers;
}
```

Figure 1: Simple window matching. From top to bottom: input images, remapped images, difference images.

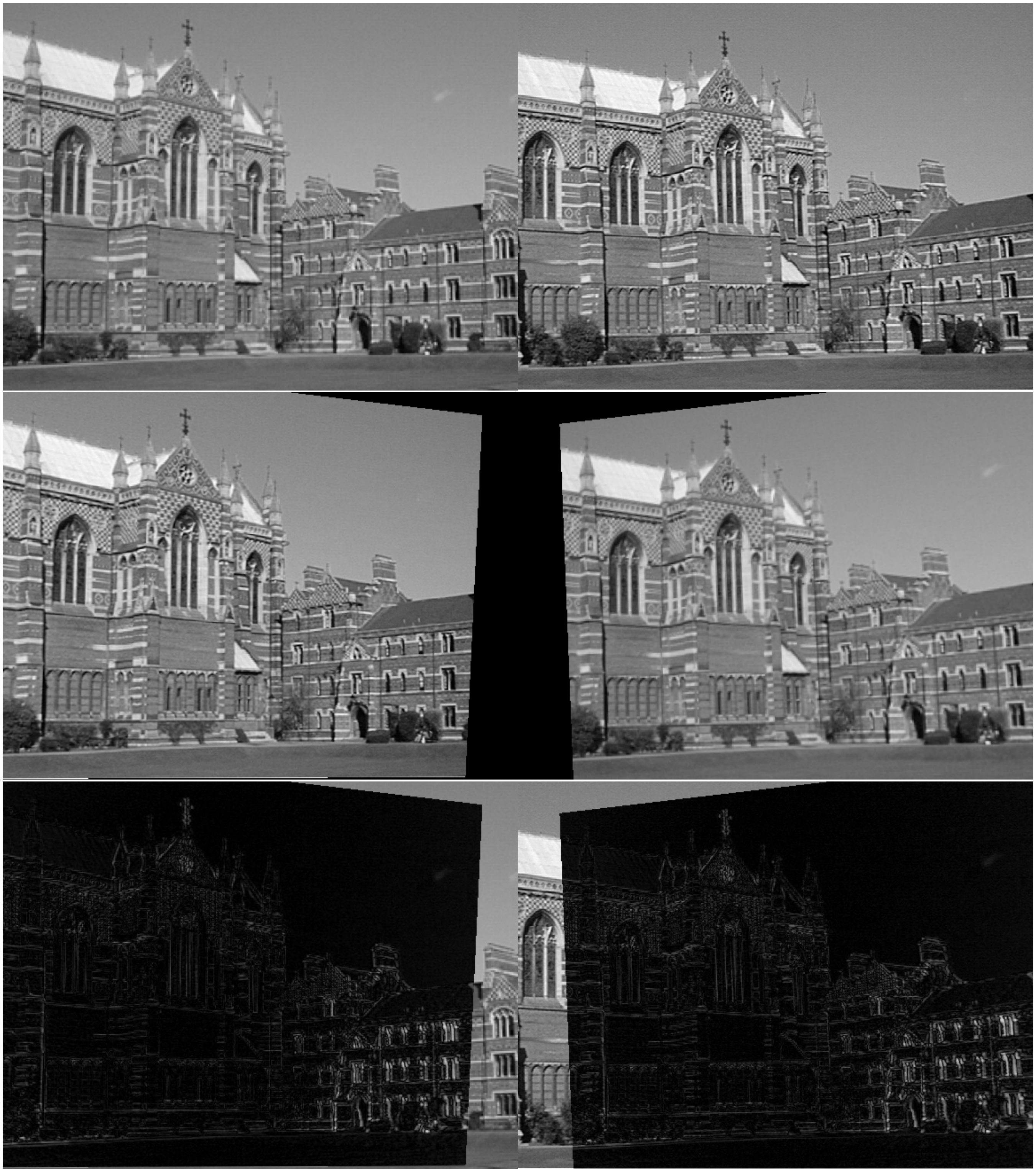

Figure 2: Sample image matching. From top to bottom: input images, remapped images, difference images.

Figure 3: Simple dart board matching. From top to bottom: input images, remapped images, difference images.

Figure 4: Difficult dart board matching. From top to bottom: input images, remapped images, difference images.