

# Automatic Computation of a Homography by RANSAC Algorithm

Rong Zhang

## 1 Problem

In this homework, we consider automatic computation of the image homography by a robust estimator - the Random Sample Consensus (RANSAC) algorithm. Different from what we have done in HW1 and HW2, the goal of this homework is to eliminate the manual selection of the corresponding pixels, i.e., no prior information is required.

The automatic computing of the homography includes two steps. The first step is to obtain interest points and determine putative correspondences, while the second one is to estimate the homography and the correspondences which are consistent with this estimate by RANSAC algorithm.

## 2 Determine Putative Correspondences

From HW3, we know how to detect corner points based on gradient information by Harris algorithm or smallest eigenvalue method. We also applied the Normalized Cross Correlation (NCC) criterion for corner points matching. In this homework, the corner detection and feature matching methods in HW3 are used for determining an initial point correspondence set. Note that these putative correspondences measured by the NCC criterion are not always perfect matches. We will show that in the result section that some of the correspondences are not true in reality. However, the RANSAC algorithm applied later will be used to eliminate these mismatches.

Given an image  $I(\mathbf{X}) = I(x, y)$  where  $I$  indicates the intensity and  $\mathbf{X} = (x, y)$  is the pixel horizontal and vertical position, the corner detection and NCC matching are summarized as follows.

### 1. Corner detection

- (a) choose a window  $W$  with fixed size, e.g.,  $5 \times 5$ , and a threshold  $T \in \mathbb{R}$
- (b) compute the image gradient  $\nabla I = (I_x, I_y) = (g_x * I, g_y * I)$ , for example, using the sobel operator

(c) at all pixels within the window  $W$  centered at  $(x, y)$ , compute the  $2 \times 2$  matrix

$$G = \begin{pmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{pmatrix}, \quad (1)$$

(d) smallest eigenvalue method: if the smallest singular value  $\min(\sigma_i(G))$ ,  $i = (1, 2)$  is greater than  $T$ , the pixel  $(x, y)$  is regarded as a corner point candidate.

(e) check uniqueness condition: if the pixel  $(x, y)$  has greatest  $\min(\sigma_i(G))$  in a neighborhood, mark  $(x, y)$  as a corner point and eliminate the other false alarmed candidate in the neighborhood.

2. NCC matching: for each corner point  $\mathbf{X}_i$ , ( $i = 1, 2, \dots, N_1$ ),

(a) compute the NCC value with all the feature point  $\mathbf{X}'_j$  ( $j = 1, 2, \dots, N_2$ ) in the camera picture by

$$NCC(i, j) = \frac{\sum_W (I_1(\tilde{\mathbf{X}}) - \bar{I}_1)(I_2(\tilde{\mathbf{X}}') - \bar{I}_2)}{\sqrt{\sum_{W(\mathbf{X}_i)} (I_1(\tilde{\mathbf{X}}) - \bar{I}_1)^2 \sum_{W(\mathbf{X}'_j)} (I_2(\tilde{\mathbf{X}}') - \bar{I}_2)^2}}, \quad (2)$$

where  $\tilde{\mathbf{X}} \in W(\mathbf{X}_i)$ ,  $\tilde{\mathbf{X}}' \in W(\mathbf{X}'_j)$ ,  $\bar{I}_1$  and  $\bar{I}_2$  are the mean intensities of all the pixels within the window and  $N$  is the total number of these pixels.

(b) determine the matched pointed  $\mathbf{X}'_{j^*}$  by,

$$j_* = \arg \max_{j=(1,2,\dots,N_2)} (NCC(i, j)). \quad (3)$$

(c) if there is another correspondence  $(i', j_*)$  determined before, choose  $i_*$  as the matched point for  $j_*$  by

$$i_* = \arg \max_{\hat{i}=(i,i')} (NCC(\hat{i}, j_*)), \quad (4)$$

i.e., create one-to-one correspondence.

### 3 RANSAC for a Homography

#### 3.1 Direct Linear Transformation (DLT)

Given a scene image  $\vec{X}$  (homogeneous representation) and a projective transformation  $H$ , we have the camera image  $\vec{X}' = H\vec{X}$ . If we write  $\vec{X}_i = (x_i, y_i, w_i)^T$ ,  $\vec{X}'_i = (x'_i, y'_i, w'_i)^T$  and

$H = \begin{pmatrix} \vec{h}_1^T \\ \vec{h}_2^T \\ \vec{h}_3^T \end{pmatrix}$ , where  $\vec{h}_i^T$  is a row vector denoting the  $i$ th row of  $H$ , we have

$$\begin{pmatrix} \vec{0}^T & -w'_i \vec{X}_i^T & y'_i \vec{X}_i^T \\ w'_i \vec{X}_i^T & \vec{0}^T & -x'_i \vec{X}_i^T \end{pmatrix} \begin{pmatrix} \vec{h}_1 \\ \vec{h}_2 \\ \vec{h}_3 \end{pmatrix} = \vec{0}, \quad (5)$$

This can be written as

$$A_i \vec{h} = \vec{0}, \quad (6)$$

where  $A_i$  is a  $2 \times 9$  matrix and  $\vec{h}$  is  $9 \times 1$ .

Therefore,  $n$  ( $n \geq 4$ ) correspondences will provide a matrix  $A$  with dimension  $2n \times 9$  and the solution of  $h$  can be written as

$$\begin{aligned} & \text{minimize } \|A\vec{h}\| \quad \text{s.t. } \|h\| = 1 \\ & \text{is equivalent to} \\ & \text{minimize } \vec{h}^T A^T A \vec{h} \quad \text{s.t. } \|h\| = 1. \end{aligned}$$

The solution is the eigenvector of  $A^T A$  associated with the smallest eigenvalue.

From the textbook, we know that the DLT algorithm can lead to unstable results because of the numerical calculation, inaccurate point correspondences and so on. This motivates a robust algorithm, such as RANSAC.

### 3.2 Normalized DLT

For the numerical calculation issues in the DLT algorithm, a normalization process should be applied. This step is very important for less well conditioned problems such as DLT.

Given  $n \geq 4$  point correspondences  $\vec{X}_i$  and  $\vec{X}'_i$ , a similarity transformation  $T_1$

$$T_1 = \begin{pmatrix} s & 0 & t_x \\ 0 & s & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

which consists of a translation and scaling will take points  $\vec{X}_i$  to a new set of points  $\vec{\hat{X}}_i = T_1 \vec{X}_i$  such that the centroid of the new points has the coordinate  $(0, 0)^T$ , and their average distance from the origin is  $\sqrt{2}$ .

Suppose  $\vec{\hat{X}}_i = (x_i, y_i, 1)^T$ , we have

$$\vec{\hat{X}}_i = T_1 \vec{X}_i = \begin{pmatrix} sx_i + t_x \\ sy_i + t_y \\ 1 \end{pmatrix} = \begin{pmatrix} \hat{x}_i \\ \hat{y}_i \\ 1 \end{pmatrix}. \quad (7)$$

The centroid of  $\vec{X}_i$  is  $\begin{pmatrix} \hat{x} \\ \hat{y} \\ 1 \end{pmatrix} = \begin{pmatrix} s\bar{x} + t_x \\ s\bar{y} + t_y \\ 1 \end{pmatrix} = (0, 0, 1)^T$  where the  $\bar{x}$  denote the mean.

Therefore,  $t_x = -s\bar{x}$  and  $t_y = -s\bar{y}$ .

The average distance between  $\vec{X}_i$  and origin is

$$\begin{aligned} & \frac{1}{n} \sum_i \sqrt{\hat{x}_i^2 + \hat{y}_i^2} \\ &= \frac{1}{n} \sum_i \sqrt{(sx_i - s\bar{x})^2 + (sy_i - s\bar{y})^2} \\ &= \frac{s}{n} \sum_i \sqrt{(x_i - \bar{x})^2 + (y_i - \bar{y})^2} \\ &= \sqrt{2}. \end{aligned} \tag{8}$$

Therefore,  $s$  can be computed as

$$s = \frac{\sqrt{2}}{\frac{1}{n} \sum_i \sqrt{(x_i - \bar{x})^2 + (y_i - \bar{y})^2}}. \tag{9}$$

Similarly, a similarity transformation  $T_2$  will take points  $\vec{X}'_i$  to a new set of points  $\vec{X}_i = T_2 \vec{X}'_i$ . Apply DLT to the correspondences  $\vec{X}_i$  and  $\vec{X}'_i$ , we obtain a homography  $\hat{H}$ . Since

$$\vec{X}_i = H \vec{X}_i = T_2^{-1} \vec{X}'_i = T_2^{-1} \hat{H} \vec{X}'_i = T_2^{-1} \hat{H} T_1 \vec{X}_i, \tag{10}$$

we have the desired homography  $H = T_2^{-1} \hat{H} T_1$ .

### 3.3 RANSAC Algorithm

After we get  $n$  putative correspondences, the RANSAC robust estimation is used in computing 2D homography.

1. initialize number of estimation  $N = 500$ , threshold  $T\_DIST$ ,  $MAX\_inlier = -1$ ,  $MIN\_std = 10e5$  and  $p = 0.99$ .
2. for  $i$ th ( $i = 1 : N$ ) estimation
  - (a) randomly choose 4 correspondences
  - (b) check whether these points are colinear, if so, redo the above step
  - (c) compute the homography  $H_{curr}$  by normalized DLT from the 4 points pairs
  - (d) for each putative correspondence, calculate distance  $d_i = d(\vec{X}'_i, H_{curr} \vec{X}_i) + d(\vec{X}_i, H_{curr}^{-1} \vec{X}'_i)$  by the above  $H_{curr}$
  - (e) compute the standard deviation of the inlier distance  $curr\_std$
  - (f) count the number of inliers  $m$  which has the distance  $d_i < T\_DIST$

- (g) if( $m > \text{MAX\_inlier}$  or ( $m == \text{MAX\_inlier}$  and  $\text{curr\_std} < \text{MIN\_std}$ )) update best  $H = H_{\text{curr}}$  and record all the inliers
  - (h) update  $N$  by Algorithm 4.5: compute  $\epsilon = 1 - m/n$  and set  $N = \frac{\log(1-p)}{\log(1-(1-\epsilon)^4)}$
3. refinement: re-estimate  $H$  from all the inliers using the DLT algorithm.
  4. transform  $\vec{X}'$  by  $H^{-1}$  (i.e.,  $H^{-1}\vec{X}'$ ) to get the reconstructed scene image, compare to the original scene image.

## 4 Results

The results are showed in this section. We show the original images first, followed by the detected corner points (number of feature points are indicated in the figure captions). In the NCC result figure, lines with different colors shows correspondences for difference matched point pairs. The numbers of putative correspondences are indicated under the figure. In the RANSAC result figure, green points and green lines show the inliers and the red ones are the outliers. We can see from the figures that the RANSAC algorithm efficiently eliminate those inaccurate correspondences. The last figures are the original scene image and the transformed camera images using  $H$  obtained from the four correspondences and from all the inliers. The reconstructed scene image using  $H$  obtained from all the inliers is slightly closer to the original scene image. The mean square error (MSE) of between the transformed image and original scene image are listed in the figures. We can see that the transformed image under  $H$  obtained from all the inliers has a lower MSE value than that obtained from the 4 pairs.

For all the testing images, different parameters T\_SMALLEST\_EIG are used (indicated under the corner detection figures). The other parameters are the same

```
MAX_CORNERPOINT_NUM = 500
W_SIZE = 7
EUC_DISTANCE = 10
B_SIZE = 30
W_SIZE_MATCH = 30
T_DIST = 30
```



Figure 1: Original images (left:samplea.jpg; right: sampleb.jpg)

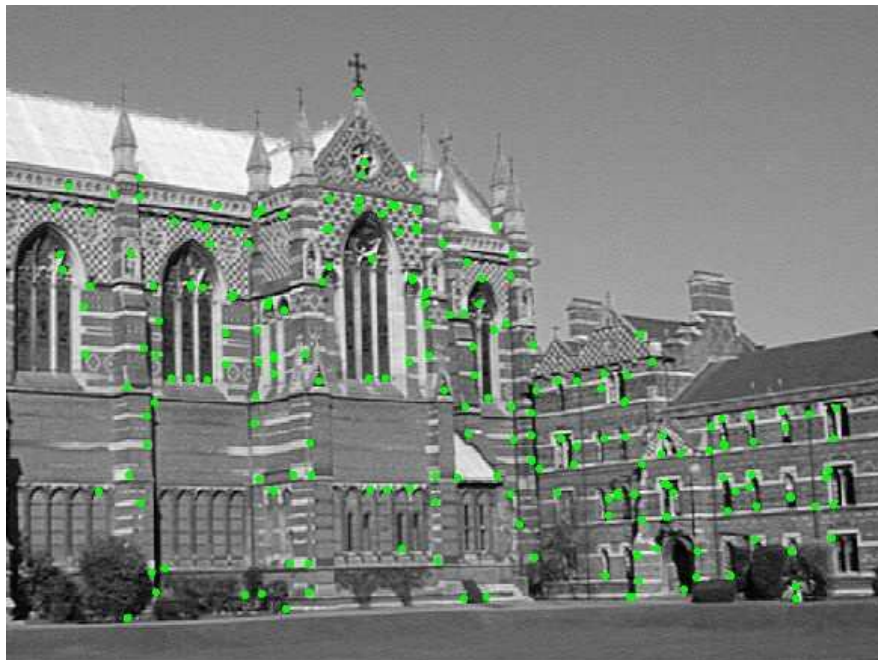
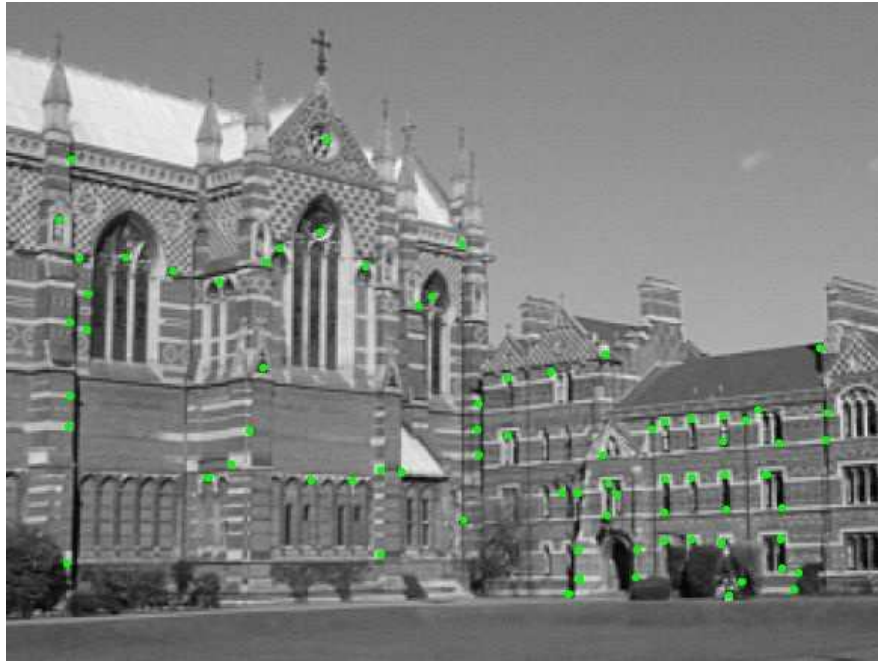


Figure 2: Detected corners  $T\_SMALLEST\_EIG = 60$  (number of corner points detected: 80 and 219)



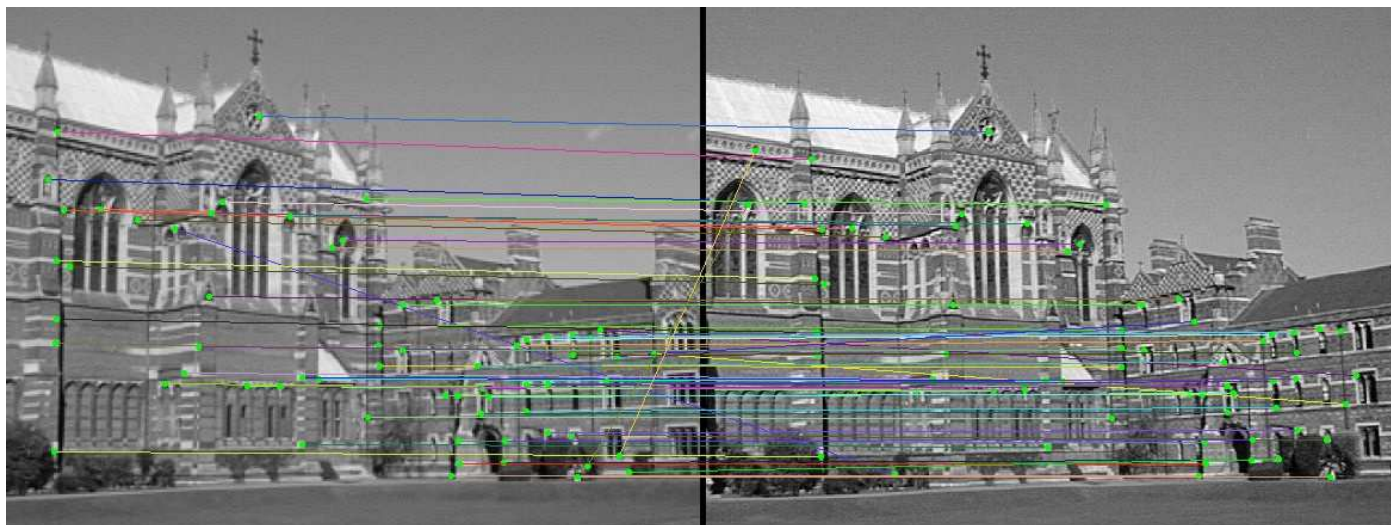


Figure 3: NCC matching (number of matched pairs: 64)

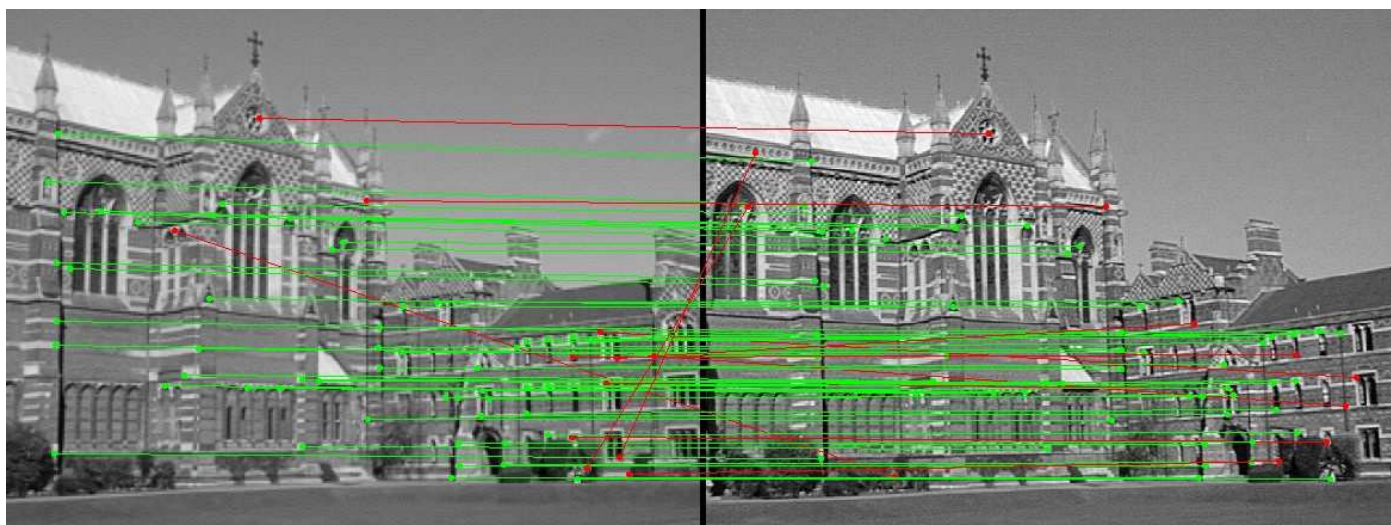


Figure 4: RANSAC results: green points and lines represent inliers and red ones are outliers (number of inlier: 52)



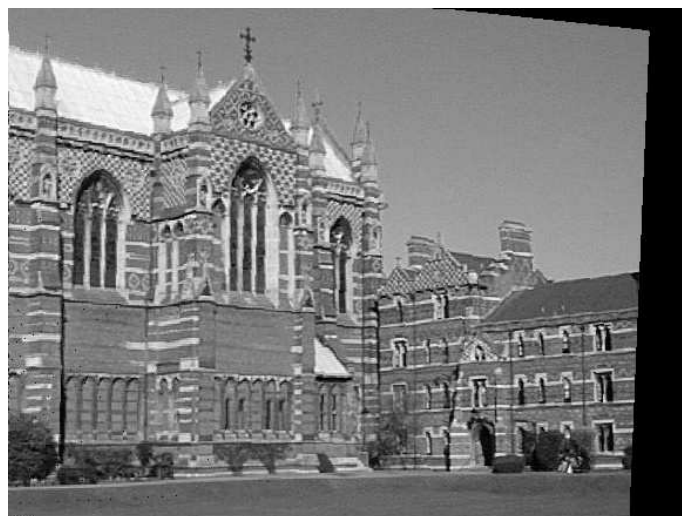
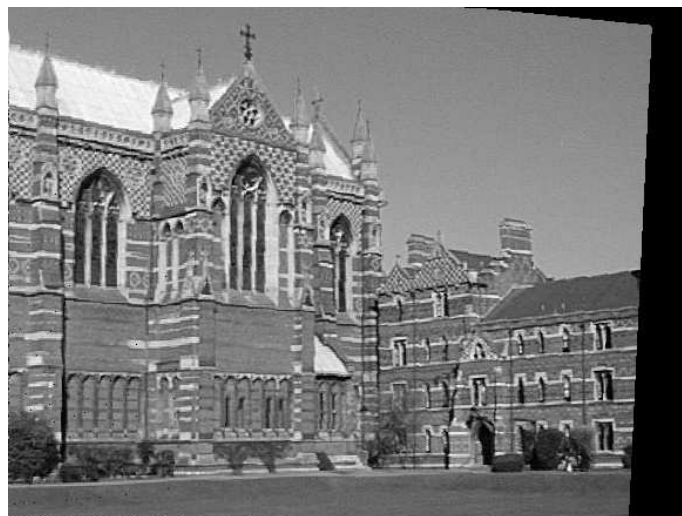


Figure 5: Up: original image samplea.jpg; middle: transformed sampleb.jpg (mse:691.00) by best  $H$  obtained by 4 corner point pairs; down: transformed sampleb.jpg (mse:551.03) by  $H$  calculated from all inliers (52 in all)



Figure 6: Original images (left:m1a.jpg; right: m1b.jpg)



Figure 7: Detected corners  $T\_SMALLEST\_EIG = 10$  (number of corner points detected: 68 and 64)



Figure 8: NCC matching (number of matched pairs: 54)

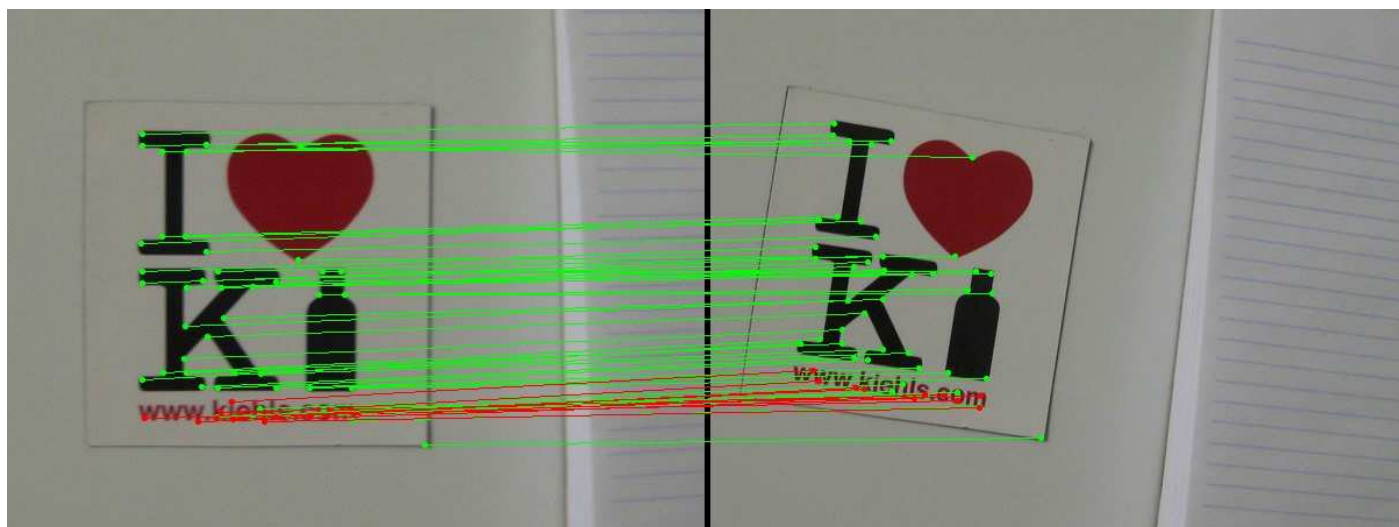


Figure 9: RANSAC results: green points and lines represent inliers and red ones are outliers (number of inlier: 44)



Figure 10: Up: original image m1a.jpg; middle: transformed m1b.jpg (mse:85.71) by best H obtained by 4 corner point pairs; down: transformed m1b.jpg (mse:55.78) by H calculated from all inliers (44 in all)





Figure 11: Original images (left:m2a.jpg; right: m2b.jpg)



Figure 12: Detected corners  $T\_SMALLEST\_EIG = 80$  (number of corner points detected: 46 and 48)

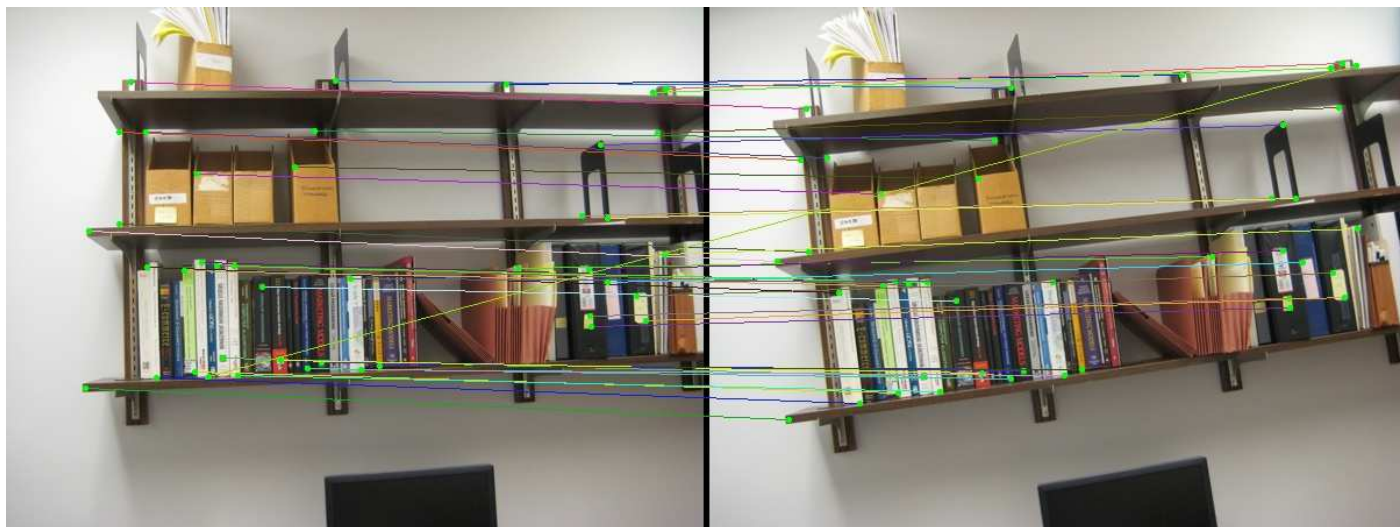


Figure 13: NCC matching (number of matched pairs: 42)

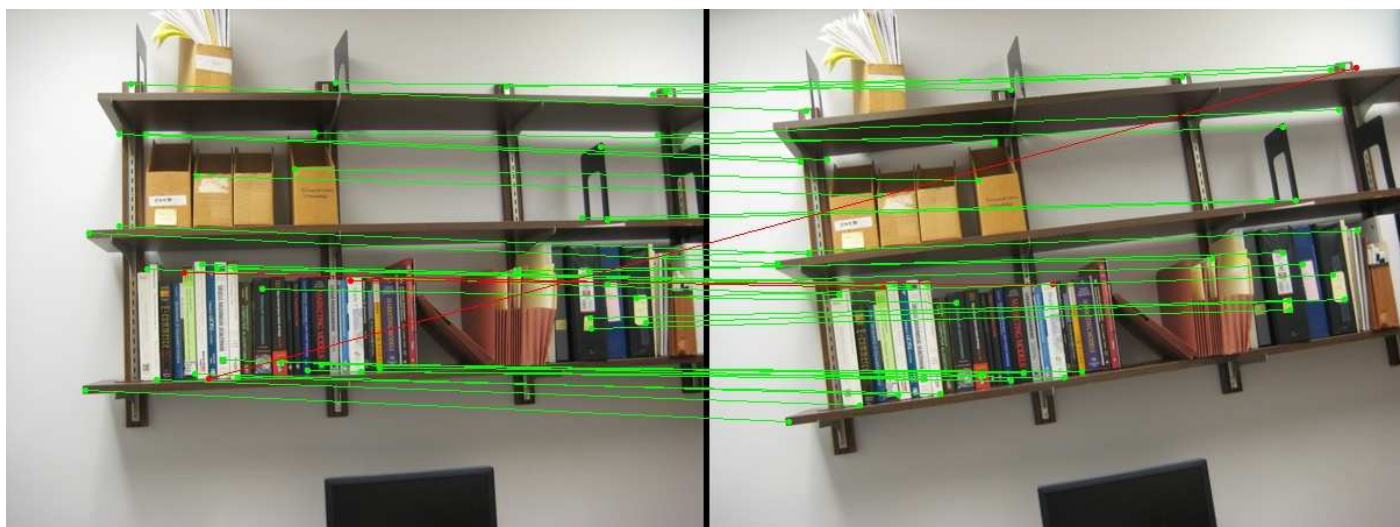


Figure 14: RANSAC results: green points and lines represent inliers and red ones are outliers (number of inlier: 39)



Figure 15: Up: original image m2a.jpg; middle: transformed m2b.jpg (mse:184.07) by best H obtained by 4 corner point pairs; down: transformed m2b.jpg (mse:150.35) by H calculated from all inliers (39 in all)



```

//*****
// RANSAC algorithm:
//   Smallest eigenvalue method is used
//   for corner detection; NCC is used for
//   similarity measure
//*****
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <cv.h>
#include <highgui.h>

#define CLIP2(minv, maxv, value) (min(maxv, max(minv, value)))

#define MAX_CORNERPOINT_NUM 500 // max number of detected corner pts
#define T_SMALLEST_EIG 60 // thres. for the smallest eigenvalue method
#define W_SIZE 7 // window size used in corner detection
#define EUC_DISTANCE 10 // thres. for Euclidean distance for uniqueness_corner
#define B_SIZE 30 // size for excluding boundary pixel
#define W_SIZE_MATCH 30 // window size used in NCC
#define T_DIST 30 // thres. for distance in RANSAC algorithm

//*****
// Compute gradient based on Sobel operator
// input: image
// output: gradient_x, gradient_y
//*****
void Gradient_Sobel(IplImage *img, CvMat* I_x, CvMat* I_y){
    int width = img->width;
    int height = img->height;
    int i,j,ii,jj;
    double valuex, valuey;
    CvScalar curpixel;
    // the sobel operator below is already flipped
    // for the convolution process
    double sobel_xdata [] = {1,0,-1,2,0,-2,1,0,-1};
    double sobel_ydata [] = {-1,-2,-1,0,0,0,1,2,1};
    CvMat sobel_x = cvMat(3,3,CV_64FC1,sobel_xdata);
    CvMat sobel_y = cvMat(3,3,CV_64FC1,sobel_ydata);

    for(i=0; i<height; i++) //for each row
    for(j=0; j<width; j++){ //for each column
        // convolution
        valuex = 0;
        valuey = 0;
        for(ii=-1; ii<=1; ii++)
        for(jj=-1; jj<=1; jj++){
            if(i+ii < 0 || i+ii >= height || j+jj < 0 || j+jj >= width)
                continue;
            curpixel = cvGet2D(img,i+ii,j+jj);
            valuex += curpixel.val[0]*cvmGet(&sobel_x,ii+1,jj+1);
            valuey += curpixel.val[0]*cvmGet(&sobel_y,ii+1,jj+1);
        }
        cvmSet(I_x,i,j,(valuex));
        cvmSet(I_y,i,j,(valuey));
    }
}

```



```

//*****
// exclude those false alarmed corners in a small neighborhood
// i.e., store only the corner pts with greatest NCC value
// input: CvPoint *corner (pts_queue)
//         int num (ttl number of pts in queue)
//         double *corner_cost (NCC values of the pts in queue)
//         CvPoint curr_point (candidate to be put in queue)
//         double curr_cost (NCC value of curr_point)
// output: updated corner, corner_cost
// return ttl number of pts in queue
//*****
int Corner_Uniqueness(CvPoint *corner, int num, double *corner_cost, CvPoint
curr_point, double curr_cost){
    int i,j;
    int idxnum = 0, newidx;
    int *idx;
    int isNeighbor = 0;
    idx = (int*) malloc(sizeof(int)* num); // to record the neighborhood corner
point should be deleted

    if(num == 0){ // the first point
        // add curr_point into queue
        corner[num] = cvPoint(curr_point.x, curr_point.y);
        corner_cost[num++] = curr_cost;
    }else{
        // compare the curr_point with the points in queue
        for(i=0; i<num; i++){
            // if the Euclidean Distance is small (within the neighborhood)
            if(sqrt(pow(curr_point.x-corner[i].x,2.0)+pow(curr_point.y-
corner[i].y,2.0)) < EUC_DISTANCE){
                isNeighbor = 1;
                if(corner_cost[i] < curr_cost) // more accurate corner
detected

                    idx[idxnum++] = i;
            }
        }
        if(idxnum > 0){
            // delete the false alarm points
            corner[idx[0]] = cvPoint(curr_point.x, curr_point.y);;
            corner_cost[idx[0]] = curr_cost;
            // more than one false alarm points detected
            if(idxnum > 1){
                // start from the 2nd point
                newidx = idx[1];
                for(i=1; i<idxnum; i++){
                    for(j=idx[i]+1; j<min(idx[min(i+1, idxnum)], num);
j++){
                        corner[newidx] = cvPoint(corner[j].x,
corner[j].y);
                        corner_cost[newidx++] = corner_cost[j];
                    }
                }
            }
            num -= idxnum;
            num++;
        }else if(isNeighbor == 0){
            // add curr_point into queue
            corner[num] = cvPoint(curr_point.x, curr_point.y);;

```

```

        corner_cost[num++] = curr_cost;
    }
}
delete idx;
return num;
}

//*****
// Corner detection
// input: img
// output: corner (detected corner pts)
// return the total number of detected corner pts
//*****
int DetectCorner(IplImage *img, CvPoint *corner){
    int num = 0;
    int i,j,ii,jj;
    int height = img->height;
    int width = img->width;
    int wsize;
    double g11,g12,g22;
    double corner_cost[MAX_CORNERPOINT_NUM];
    double curr_cost;
    CvPoint curr_point;
    CvMat *G = cvCreateMat(2,2,CV_32FC1);
    CvMat *U = cvCreateMat(2,2,CV_32FC1);
    CvMat *V = cvCreateMat(2,2,CV_32FC1);
    CvMat *D = cvCreateMat(2,2,CV_32FC1);

    // set window size
    if(W_SIZE%2 == 0){
        printf("error for window size\n");
        return 0;
    }else
        wsize = (W_SIZE-1)/2;

    // compute the gradient I_x, I_y
    CvMat *I_x = cvCreateMat(height,width,CV_64FC1);
    CvMat *I_y = cvCreateMat(height,width,CV_64FC1);
    Gradient_Sobel(img, I_x, I_y);

    double factor = 10000;
    // check each pixel
    // exclude the boundary
    for(i=B_SIZE; i<height-B_SIZE; i++){
        for(j=B_SIZE; j<width-B_SIZE; j++){
            curr_point = cvPoint(j,i);
            g11 = 0;
            g12 = 0;
            g22 = 0;
            for(ii=-wsize; ii<=wsize; ii++){
                for(jj=-wsize; jj<=wsize; jj++){
                    if(i+ii < 0 || i+ii >= height || j+jj < 0 || j+jj >= width)
                        continue;
                    g11 += pow(cvmGet(I_x,i+ii,j+jj),2.0)/factor;
                    g12 += cvmGet(I_x,i+ii,j+jj)*cvmGet(I_y,i+ii,j+jj)/factor;
                    g22 += pow(cvmGet(I_y,i+ii,j+jj),2.0)/factor;
                }
            }
            cvmSet(G,0,0,g11);

```

```

        cvmSet(G,0,1,g12);
        cvmSet(G,1,0,g12);
        cvmSet(G,1,1,g22);

        // Smallest eigenvalue method
        // SVD The flags cause U and V to be returned transposed (does not work
well without the transpose flags).
        // Therefore, in OpenCV,  $S = U^T D V$ 
        cvSVD(G, D, U, V, CV_SVD_U_T|CV_SVD_V_T);

        curr_cost = cvmGet(D,1,1);
        if(curr_cost > T_SMALLEST_EIG)
            num = Corner_Uniqueness(corner, num, corner_cost, curr_point,
curr_cost);

        if(num >= MAX_CORNERPOINT_NUM){
            printf("error. MAX_CORNERPOINT_NUM reached!");
            return -1;
        }
    }

    cvReleaseMat(&G);
    cvReleaseMat(&U);
    cvReleaseMat(&V);
    cvReleaseMat(&D);
    cvReleaseMat(&I_x);
    cvReleaseMat(&I_y);
    return num;
}

//*****
// Similarity measure based on NCC
// input:  img1, img2, (images)
//         p1, p2 (detected corner pts for each image x and x')
//         num1, num2 (ttl number of detected pts for each image)
// output: m1, m2 (matched pairs)
// return the total number of matched pairs
//*****
int CornerPointMatching_NCC(IplImage *img1, IplImage *img2, CvPoint *p1, int num1,
CvPoint *p2, int num2, CvPoint2D64f *m1, CvPoint2D64f *m2){
    int i,j,ii,jj,idx;
    double cur_value;
    double MAX_value;
    int cur_x, cur_y, match_x, match_y;
    double mean1, mean2;
    int available_num;
    CvScalar intensity;
    double tmp1, tmp2;
    double v1, v2, v3;
    double *nccvalues = new double[num1];
    int *matchedidx = new int [num1];
    int check = 0;

    int height = img1->height;
    int width = img1->width;

    idx = 0;
    for(i=0; i<num1; i++){

```

```

// for each point in p1, find a match in p2
MAX_value = -10000;
cur_x = p1[i].x;
cur_y = p1[i].y;
m1[idx].x = (double)cur_x;
m1[idx].y = (double)cur_y;

for(j=0; j<num2; j++){
    match_x = p2[j].x;
    match_y = p2[j].y;
    available_num = 0;
    mean1 = 0; mean2 = 0;
    for(ii=-W_SIZE_MATCH; ii<W_SIZE_MATCH; ii++){
        for(jj=-W_SIZE_MATCH; jj<W_SIZE_MATCH; jj++){
            if(cur_y+ii < 0 || cur_y+ii >= height || cur_x+jj < 0
|| cur_x+jj >=width)
                continue;
            intensity = cvGet2D(img1, cur_y+ii, cur_x+jj);
            mean1 += intensity.val[0];
            intensity = cvGet2D(img2, match_y+ii, match_x+jj);
            mean2 += intensity.val[0];
            available_num++;
        }
    }
    mean1 /= available_num;
    mean2 /= available_num;

    v1 = 0; v2 = 0; v3 = 0;
    for(ii=-W_SIZE_MATCH; ii<W_SIZE_MATCH; ii++){
        for(jj=-W_SIZE_MATCH; jj<W_SIZE_MATCH; jj++){
            if(cur_y+ii < 0 || cur_y+ii >= height || cur_x+jj < 0
|| cur_x+jj >=width)
                continue;
            intensity = cvGet2D(img1, cur_y+ii, cur_x+jj);
            tmp1 = intensity.val[0] - mean1;
            intensity = cvGet2D(img2, match_y+ii, match_x+jj);
            tmp2 = intensity.val[0] - mean2;
            v1 += tmp1*tmp2;
            v2 += pow(tmp1, 2.0);
            v3 += pow(tmp2, 2.0);
        }
    }
    cur_value = v1 / sqrt(v2*v3);
    if(cur_value > MAX_value)
    {
        // a better match
        MAX_value = cur_value;
        nccvalues[idx] = cur_value;
        m2[idx].x = (double)match_x;
        m2[idx].y = (double)match_y;
        matchedidx[idx] = j;
    }
}
check = 0;
for(j=0; j<idx; j++){
    if(matchedidx[j] == matchedidx[idx]){

```

```

        if(nccvalues[j] < nccvalues[idx]){
            nccvalues[j] = nccvalues[idx];
            m1[j].x = m1[idx].x;
            m1[j].y = m1[idx].y;
        }
        check = 1;
        break;
    }
}
if(check == 0)
    idx++;
}
delete nccvalues;
delete matchedidx;
return idx;
}

```

```

//*****
// Check colinearity of a set of pts
// input:  p (pts to be checked)
//         num (ttl number of pts)
// return true if some pts are coliner
//         false if not
//*****
bool isColinear(int num, CvPoint2D64f *p){
    int i,j,k;
    bool iscolinear;
    double value;
    CvMat *pt1 = cvCreateMat(3,1,CV_64FC1);
    CvMat *pt2 = cvCreateMat(3,1,CV_64FC1);
    CvMat *pt3 = cvCreateMat(3,1,CV_64FC1);
    CvMat *line = cvCreateMat(3,1,CV_64FC1);

    iscolinear = false;
    // check for each 3 points combination
    for(i=0; i<num-2; i++){
        cvmSet(pt1,0,0,p[i].x);
        cvmSet(pt1,1,0,p[i].y);
        cvmSet(pt1,2,0,1);
        for(j=i+1; j<num-1; j++){
            cvmSet(pt2,0,0,p[j].x);
            cvmSet(pt2,1,0,p[j].y);
            cvmSet(pt2,2,0,1);
            // compute the line connecting pt1 & pt2
            cvCrossProduct(pt1, pt2, line);
            for(k=j+1; k<num; k++){
                cvmSet(pt3,0,0,p[k].x);
                cvmSet(pt3,1,0,p[k].y);
                cvmSet(pt3,2,0,1);
                // check whether pt3 on the line
                value = cvDotProduct(pt3, line);
                if(abs(value) < 10e-2){
                    iscolinear = true;
                    break;
                }
            }
        }
        if(iscolinear == true) break;
    }
}

```



```

        if(iscolinear == true) break;
    }
    cvReleaseMat(&pt1);
    cvReleaseMat(&pt2);
    cvReleaseMat(&pt3);
    cvReleaseMat(&line);
    return iscolinear;
}

//*****
// Compute the homography matrix H
// i.e., solve the optimization problem  $\min ||Ah||=0$  s.t.  $||h||=1$ 
// where A is  $2n \times 9$ , h is  $9 \times 1$ 
// input: n (number of pts pairs)
//        p1, p2 (coresponded pts pairs x and x')
// output:  $3 \times 3$  matrix H
//*****
void ComputeH(int n, CvPoint2D64f *p1, CvPoint2D64f *p2, CvMat *H){
    int i;
    CvMat *A = cvCreateMat(2*n, 9, CV_64FC1);
    CvMat *U = cvCreateMat(2*n, 2*n, CV_64FC1);
    CvMat *D = cvCreateMat(2*n, 9, CV_64FC1);
    CvMat *V = cvCreateMat(9, 9, CV_64FC1);

    cvZero(A);
    for(i=0; i<n; i++){
        // 2*i row
        cvmSet(A,2*i,3,-p1[i].x);
        cvmSet(A,2*i,4,-p1[i].y);
        cvmSet(A,2*i,5,-1);
        cvmSet(A,2*i,6,p2[i].y*p1[i].x);
        cvmSet(A,2*i,7,p2[i].y*p1[i].y);
        cvmSet(A,2*i,8,p2[i].y);
        // 2*i+1 row
        cvmSet(A,2*i+1,0,p1[i].x);
        cvmSet(A,2*i+1,1,p1[i].y);
        cvmSet(A,2*i+1,2,1);
        cvmSet(A,2*i+1,6,-p2[i].x*p1[i].x);
        cvmSet(A,2*i+1,7,-p2[i].x*p1[i].y);
        cvmSet(A,2*i+1,8,-p2[i].x);
    }

    // SVD
    // The flags cause U and V to be returned transposed
    // Therefore, in OpenCV,  $A = U^T D V$ 
    cvSVD(A, D, U, V, CV_SVD_U_T|CV_SVD_V_T);

    // take the last column of  $V^T$ , i.e., last row of V
    for(i=0; i<9; i++)
        cvmSet(H, i/3, i%3, cvmGet(V, 8, i));

    cvReleaseMat(&A);
    cvReleaseMat(&U);
    cvReleaseMat(&D);
    cvReleaseMat(&V);
}

```

```

//*****
// Compute number of inliers by computing distance under a particular H
// distance = d(Hx, x') + d(invH x', x)
// input: num (number of pts pairs)
//         p1, p2 (corresponded pts pairs x and x')
//         H (the homography matrix)
// output: inlier_mask (masks to indicate pts of inliers in p1, p2)
//         dist_std (std of the distance among all the inliers)
// return: number of inliers
//*****
int ComputeNumberOfInliers(int num, CvPoint2D64f *p1, CvPoint2D64f *p2, CvMat *H,
CvMat *inlier_mask, double *dist_std){
    int i, num_inlier;
    double curr_dist, sum_dist, mean_dist;
    CvPoint2D64f tmp_pt;
    CvMat *dist = cvCreateMat(num, 1, CV_64FC1);
    CvMat *x = cvCreateMat(3,1,CV_64FC1);
    CvMat *xp = cvCreateMat(3,1,CV_64FC1);
    CvMat *pt = cvCreateMat(3,1,CV_64FC1);
    CvMat *invH = cvCreateMat(3,3,CV_64FC1);

    cvInvert(H, invH);

    // check each correspondence
    sum_dist = 0;
    num_inlier = 0;
    cvZero(inlier_mask);
    for(i=0; i<num; i++){
        // initial point x
        cvmSet(x,0,0,p1[i].x);
        cvmSet(x,1,0,p1[i].y);
        cvmSet(x,2,0,1);
        // initial point x'
        cvmSet(xp,0,0,p2[i].x);
        cvmSet(xp,1,0,p2[i].y);
        cvmSet(xp,2,0,1);

        // d(Hx, x')
        cvMatMul(H, x, pt);
        tmp_pt.x = (int)(cvmGet(pt,0,0)/cvmGet(pt,2,0));
        tmp_pt.y = (int)(cvmGet(pt,1,0)/cvmGet(pt,2,0));
        curr_dist = pow(tmp_pt.x-p2[i].x, 2.0) + pow(tmp_pt.y-p2[i].y, 2.0);
        // d(x, invH x')
        cvMatMul(invH, xp, pt);
        tmp_pt.x = (int)(cvmGet(pt,0,0)/cvmGet(pt,2,0));
        tmp_pt.y = (int)(cvmGet(pt,1,0)/cvmGet(pt,2,0));
        curr_dist += pow(tmp_pt.x-p1[i].x, 2.0) + pow(tmp_pt.y-p1[i].y, 2.0);

        if(curr_dist < T_DIST){
            // an inlier
            num_inlier++;
            cvmSet(inlier_mask,i,0,1);
            cvmSet(dist,i,0,curr_dist);
            sum_dist += curr_dist;
        }
    }

    // Compute the standard deviation of the distance

```

```

mean_dist = sum_dist/(double)num_inlier;
*dist_std = 0;
for(i=0; i<num; i++){
    if(cvmGet(inlier_mask,i,0) == 1)
        *dist_std += pow(cvmGet(dist,i,0)-mean_dist,2.0);
}
*dist_std /= (double) (num_inlier -1);

cvReleaseMat(&dist);
cvReleaseMat(&x);
cvReleaseMat(&xp);
cvReleaseMat(&pt);
cvReleaseMat(&invH);
return num_inlier;
}

//*****
// finding the normalization matrix  $x' = T*x$ , where  $T=\{s,0,tx, 0,s,ty, 0,0,1\}$ 
// compute T such that the centroid of  $x'$  is the coordinate origin (0,0)T
// and the average distance of  $x'$  to the origin is  $\sqrt{2}$ 
// we can derive that  $tx = -scale*mean(x)$ ,  $ty = -scale*mean(y)$ ,
//  $scale = \sqrt{2}/(\sum(\sqrt{(xi-mean(x))^2+(yi-mean(y))^2})/n)$ 
// where n is the total number of points
// input: num (ttl number of pts)
//         p (pts to be normalized)
// output: T (normalization matrix)
//         p (normalized pts)
// NOTE: because of the normalization process, the pts coordinates should
//        has accuracy as "float" or "double" instead of "int"
//*****
void Normalization(int num, CvPoint2D64f *p, CvMat *T){
    double scale, tx, ty;
    double meanx, meany;
    double value;
    int i;
    CvMat *x = cvCreateMat(3,1,CV_64FC1);
    CvMat *xp = cvCreateMat(3,1,CV_64FC1);

    meanx = 0;
    meany = 0;
    for(i=0; i<num; i++){
        meanx += p[i].x;
        meany += p[i].y;
    }
    meanx /= (double)num;
    meany /= (double)num;

    value = 0;
    for(i=0; i<num; i++)
        value += sqrt(pow(p[i].x-meanx, 2.0) + pow(p[i].y-meany, 2.0));
    value /= (double)num;

    scale = sqrt(2.0)/value;
    tx = -scale * meanx;
    ty = -scale * meany;

    cvZero(T);
    cvmSet(T,0,0,scale);
}

```

```

    cvmSet(T,0,2,tx);
    cvmSet(T,1,1,scale);
    cvmSet(T,1,2,ty);
    cvmSet(T,2,2,1.0);

    //Transform x' = T*x
    for(i=0; i<num; i++){
        cvmSet(x,0,0,p[i].x);
        cvmSet(x,1,0,p[i].y);
        cvmSet(x,2,0,1.0);
        cvMatMul(T,x,xp);
        p[i].x = cvmGet(xp,0,0)/cvmGet(xp,2,0);
        p[i].y = cvmGet(xp,1,0)/cvmGet(xp,2,0);
    }

    cvReleaseMat(&x);
    cvReleaseMat(&xp);
}

//*****
// RANSAC algorithm
// input: num (ttl number of pts)
//         m1, m2 (pts pairs)
// output: inlier_mask (indicate inlier pts pairs in (m1, m2) as 1; outlier: 0)
//         H (the best homography matrix)
//*****
void RANSAC_homography(int num, CvPoint2D64f *m1, CvPoint2D64f *m2, CvMat *H,
CvMat *inlier_mask){
    int i,j;
    int N = 1000, s = 4, sample_cnt = 0;
    double e, p = 0.99;
    int numinlier, MAX_num;
    double curr_dist_std, dist_std;
    bool iscolinear;
    CvPoint2D64f *curr_m1 = new CvPoint2D64f[s];
    CvPoint2D64f *curr_m2 = new CvPoint2D64f[s];
    int *curr_idx = new int[s];

    CvMat *curr_inlier_mask = cvCreateMat(num,1,CV_64FC1);
    CvMat *curr_H = cvCreateMat(3,3,CV_64FC1);
    CvMat *T1 = cvCreateMat(3,3,CV_64FC1);
    CvMat *T2 = cvCreateMat(3,3,CV_64FC1);
    CvMat *invT2 = cvCreateMat(3,3,CV_64FC1);
    CvMat *tmp_pt = cvCreateMat(3,1,CV_64FC1);

    // RANSAC algorithm (reject outliers and obtain the best H)
    srand(134);
    MAX_num = -1;
    while(N > sample_cnt){
        // for a randomly chosen non-colinear correspondances
        iscolinear = true;
        while(iscolinear == true){
            iscolinear = false;
            for(i=0; i<s; i++){
                // randomly select an index
                curr_idx[i] = rand()%num;
                for(j=0; j<i; j++){
                    if(curr_idx[i] == curr_idx[j]){

```

```

        iscolinear = true;
        break;
    }
}
if(iscolinear == true) break;
curr_m1[i].x = m1[curr_idx[i]].x;
curr_m1[i].y = m1[curr_idx[i]].y;
curr_m2[i].x = m2[curr_idx[i]].x;
curr_m2[i].y = m2[curr_idx[i]].y;
}
// Check whether these points are colinear
if(iscolinear == false)
    iscolinear = isColinear(s, curr_m1);
}
// Normalized DLT
Normalization(s, curr_m1, T1); //curr_m1 <- T1 * curr_m1
Normalization(s, curr_m2, T2); //curr_m2 <- T2 * curr_m2

// Compute the homography matrix H = invT2 * curr_H * T1
ComputeH(s, curr_m1, curr_m2, curr_H);
cvInvert(T2, invT2);
cvMatMul(invT2, curr_H, curr_H); // curr_H <- invT2 * curr_H
cvMatMul(curr_H, T1, curr_H); // curr_H <- curr_H * T1

// Calculate the distance for each putative correspondence
// and compute the number of inliers
numinlier =
ComputeNumberOfInliers(num,m1,m2,curr_H,curr_inlier_mask,&curr_dist_std);

// Update a better H
if(numinlier > MAX_num || (numinlier == MAX_num && curr_dist_std <
dist_std)){
    MAX_num = numinlier;
    cvCopy(curr_H, H);
    cvCopy(curr_inlier_mask, inlier_mask);
    dist_std = curr_dist_std;
}

// update number N by Algorithm 4.5
e = 1 - (double)numinlier / (double)num;
N = (int)(log(1-p)/log(1-pow(1-e,s)));
sample_cnt++;
}

// Optimal estimation using all the inliers
delete curr_m1, curr_m2, curr_idx;
cvReleaseMat(&curr_H);
cvReleaseMat(&T1);
cvReleaseMat(&T2);
cvReleaseMat(&invT2);
cvReleaseMat(&tmp_pt);
cvReleaseMat(&curr_inlier_mask);
}

int main(int argc, char *argv[])
{
    IplImage *img_1=0, *img_2=0, *gimg_1=0, *gimg_2=0;

```

```

IplImage *img_show0, *img_show1, *img_show2, *img_interp, *img_scene;
uchar *data_img1, *data_interp, *data_scene;
int height, width, step, channels;
int num_1, num_2, num_matched, num_inlier;
int i,j,k;
int ttlw, ttlh;
CvPoint newmatched;
CvPoint cornerp1[MAX_CORNERPOINT_NUM];
CvPoint cornerp2[MAX_CORNERPOINT_NUM];
CvPoint2D64f matched1[MAX_CORNERPOINT_NUM];
CvPoint2D64f matched2[MAX_CORNERPOINT_NUM];
// NOTE: because of the normalization process, the pts coordinates
// should has accurcy as "float" or "double" instead of "int"

if(argc<3){
    printf("Usage: main <image-file-name>\n\7");
    exit(0);
}

// load the color image1 and image2
img_1 = cvLoadImage(argv[1]);
data_img1 = (uchar *)img_1->imageData;
if(!img_1){
    printf("Could not load image file: %s\n",argv[1]);
    exit(0);
}
img_2 = cvLoadImage(argv[2]);
if(!img_2){
    printf("Could not load image file: %s\n",argv[2]);
    exit(0);
}
height    = img_1->height;
width     = img_1->width;
step      = img_1->widthStep;
channels   = img_1->nChannels;

// create gray scale image
gimg_1 = cvCreateImage(cvSize(width,height), IPL_DEPTH_8U, 1);
gimg_2 = cvCreateImage(cvSize(img_2->width,img_2->height), IPL_DEPTH_8U, 1);
cvCvtColor(img_1, gimg_1, CV_BGR2GRAY);
cvCvtColor(img_2, gimg_2, CV_BGR2GRAY);
cvSmooth(gimg_1, gimg_1, CV_GAUSSIAN, 3, 3, 0);
cvSmooth(gimg_2, gimg_2, CV_GAUSSIAN, 3, 3, 0);

// detect corner
// corner points are stored in CvPoint cornerp1 and cornerp2
num_1 = DetectCorner(gimg_1, cornerp1);
num_2 = DetectCorner(gimg_2, cornerp2);
printf("number of corner points detected: %d %d\n", num_1, num_2);

// feature matching by NCC
// matched pairs are stored in CvPoint2D64f matched1 and matched2
num_matched = CornerPointMatching_NCC(gimg_1, gimg_2, cornerp1, num_1, cornerp2,
num_2, matched1, matched2);
printf("number of matched pairs: %d \n", num_matched);

// generate a new image displaying the two images
ttlw = 5+width+img_2->width;

```



```

ttlh = max(height,img_2->height);
// img_show1 is the image showing the two images together
// with the corner point correspondence
img_show1 = cvCreateImage(cvSize(ttlw,ttlh), IPL_DEPTH_8U, 3);
cvZero(img_show1);
for(i=0; i<ttlh; i++){
    for(j=0; j<ttlw; j++){
        if(i<height && j<width)
            cvSet2D(img_show1,i,j,cvGet2D(img_1,i,j));
        else if(i<height && j>=width+5 && j<ttlw)
            cvSet2D(img_show1,i,j,cvGet2D(img_2,i,j-width-5));
    }
}
// img_show2 is the image showing the two images together
// and indicating the inliers and outliers
img_show2 = cvCloneImage(img_show1);

// generate corner detection results
// img_show0 shows the original images with detected corner points
img_show0 = cvCloneImage(img_1);
for(i=0; i<num_1; i++)
    cvCircle(img_show0, cornerp1[i], 1, CV_RGB(0,255,0), 2, 8, 0);
cvSaveImage("cornerp1.jpg", img_show0);
img_show0 = cvCloneImage(img_2);
for(i=0; i<num_2; i++)
    cvCircle(img_show0, cornerp2[i], 1, CV_RGB(0,255,0), 2, 8, 0);
cvSaveImage("cornerp2.jpg", img_show0);

// generate img_show1
for(i=0; i<num_matched; i++){
    newmatched.x = (int)matched2[i].x + width + 5;
    newmatched.y = (int)matched2[i].y;
    cvLine(img_show1, cvPoint((int)matched1[i].x,(int)matched1[i].y),
newmatched, CV_RGB(rand()%255,rand()%255,rand()%255), 1, 8, 0);
    cvCircle(img_show1, cvPoint((int)matched1[i].x,(int)matched1[i].y),1,
CV_RGB(0,255,0), 2, 8, 0);
    cvCircle(img_show1, newmatched, 1, CV_RGB(0,255,0), 2, 8, 0);
}
cvSaveImage("NCC_result.jpg", img_show1);

// RANSAC algorithm
CvMat *H = cvCreateMat(3,3,CV_64FC1);
CvMat *invH = cvCreateMat(3,3,CV_64FC1);
CvMat *inlier_mask = cvCreateMat(num_matched,1,CV_64FC1);
RANSAC_homography(num_matched, matched1, matched2, H, inlier_mask);

num_inlier = 0;
for(i=0; i<num_matched; i++){
    newmatched.x = (int)matched2[i].x + width + 5;
    newmatched.y = (int)matched2[i].y;
    if(cvmGet(inlier_mask,i,0) == 1){
        // green points and lines show the inliers' correspondence
        cvLine(img_show2, cvPoint((int)matched1[i].x,(int)matched1[i].y),
newmatched, CV_RGB(0,255,0),1, 8, 0);
        cvCircle(img_show2, cvPoint((int)matched1[i].x,(int)matched1[i].y), 1,
CV_RGB(0,255,0), 2, 8, 0);
        cvCircle(img_show2, newmatched, 1, CV_RGB(0,255,0), 2, 8, 0);
        num_inlier++;
    }
}

```

```

    }else{
        // red points and lines show the outliers' correspondence
        cvLine(img_show2, cvPoint((int)matched1[i].x,(int)matched1[i].y),
newmatched, CV_RGB(255,0,0),1, 8, 0);
        cvCircle(img_show2, cvPoint((int)matched1[i].x,(int)matched1[i].y), 1,
CV_RGB(255,0,0), 2, 8, 0);
        cvCircle(img_show2, newmatched, 1, CV_RGB(255,0,0), 2, 8, 0);
    }
}
printf("number of inlier: %d\n",num_inlier);
cvSaveImage("RANSAC_result.jpg", img_show2);

// reconstructed homography using the best H computed from 4 pairs of points
CvMat *check = cvCreateMat(height, width, CV_64FC1);
CvMat *ptxp = cvCreateMat(3,1,CV_64FC1);
CvMat *ptx = cvCreateMat(3,1,CV_64FC1);
int curpi, curpj, count;
double mse = 0, msetmp;
int pixnum;
cvZero(check);
cvInvert(H, invH);
img_scene = cvCloneImage(img_2);
cvZero(img_scene);
for (i=0; i<img_1->height; i++){ //y - ver
    for (j=0; j<img_1->width; j++){ //x - hor
        // set X_a
        cvmSet(ptxp,0,0,(double)j);
        cvmSet(ptxp,1,0,(double)i);
        cvmSet(ptxp,2,0,1.0);
        // compute X
        cvMatMul(invH, ptxp, ptx);
        curpi = CLIP2(0, height-1, (int)(cvmGet(ptx,1,0)/cvmGet(ptx,2,0)));
        curpj = CLIP2(0, width-1, (int)(cvmGet(ptx,0,0)/cvmGet(ptx,2,0)));

        cvSet2D(img_scene,curpi,curpj,cvGet2D(img_2,i,j));
        cvmSet(check,curpi,curpj,1);
    }
}
//interpolation
img_interp = cvCloneImage(img_scene);
data_interp = (uchar *)img_interp->imageData;
data_scene = (uchar *)img_scene->imageData;
mse = 0;
pixnum = 0;
for (i=1; i<height-1; i++){ //y - ver
    for (j=1; j<width-1; j++){ //x - hor
        msetmp = 0;
        if(cvmGet(check,i,j) == 0){
            count = (cvmGet(check,i-
1,j)==1)+(cvmGet(check,i+1,j)==1)+(cvmGet(check,i,j-1)==1)+(cvmGet(check,i,j+1)==1);
            if(count != 0){
                for (k=0; k<channels; k++){
                    data_interp[i*step+j*channels+k] =
(int)((data_scene[(i-
1)*step+j*channels+k]+data_scene[(i+1)*step+j*channels+k]+data_scene[i*step+(j-
1)*channels+k]+data_scene[i*step+(j+1)*channels+k])/count);
                    msetmp += pow(data_img1[i*step+j*channels+k]-
data_interp[i*step+j*channels+k], 2.0);

```



```

        msetmp = 0;
        if(cvmGet(check,i,j) == 0){
            count = (cvmGet(check,i-
1,j)==1)+(cvmGet(check,i+1,j)==1)+(cvmGet(check,i,j-1)==1)+(cvmGet(check,i,j+1)==1);
            if(count != 0 ){
                for (k=0; k<channels; k++){
                    data_interp[i*step+j*channels+k] =
(int)((data_scene[(i-
1)*step+j*channels+k]+data_scene[(i+1)*step+j*channels+k]+data_scene[i*step+(j-
1)*channels+k]+data_scene[i*step+(j+1)*channels+k])/count);
                    msetmp += pow(data_img1[i*step+j*channels+k]-
data_interp[i*step+j*channels+k], 2.0);
                }
                mse += msetmp/channels;
                pixnum++;
            }
        }else{
            for (k=0; k<channels; k++)
                msetmp += pow(data_img1[i*step+j*channels+k]-
data_interp[i*step+j*channels+k], 2.0);
            mse += msetmp/channels;
            pixnum++;
        }
    }
}
mse /= pixnum;
printf("pixnum:%d mse is %f\n", pixnum, mse);
cvSaveImage("scene_inliers.jpg",img_interp);

// release
delete inlierp1, inlierp2;
cvReleaseMat(&H);
cvReleaseMat(&invH);
cvReleaseMat(&ptx);
cvReleaseMat(&ptxp);
cvReleaseMat(&check);
cvReleaseMat(&inlier_mask);
cvReleaseImage(&img_1);
cvReleaseImage(&img_2);
cvReleaseImage(&gimg_1);
cvReleaseImage(&gimg_2);
cvReleaseImage(&img_show0);
cvReleaseImage(&img_show1);
cvReleaseImage(&img_show2);
cvReleaseImage(&img_scene);
cvReleaseImage(&img_interp);
return 0;
}

```