ECE661 Computer Vision : HW6

Dae Woo Kim

November 9, 2010

1 Problem

In this homework, we implement Zhengyou Zhang's camera calibration procedure,. All five camera intrinsic parameters and six extrinsic parameters should be estimated based on image homography based method. In addition, radial distortion parameters should be estimated also.

2 Finding Feature Points

In this homework, we use checker board pattern. We place the pattern on model plane(where Z component of world corrdinate is zero). And then several picture are taken by changing the location and rotation angle of the camera. We can get world coordinate of these corner easily by ordiering the corner point from top to bottom and from left to right. There are 8 vertical lines and 10 horizontal lined resulting in 80 corner points. The corner point detection procedures for the captured image are:

1) Apply the Canny edge detector to the images

2) Use the Hough transform to fit straight lines to the Canny edges

3) Group the lines into horizontal and vertical line sets.

4) Sort each set of lines (e.g. top to bottom for horizontal and left to right for vertical).

5) If two or more lines are very close then they probably correspond to the same line in the physical world.

In this case we should choose one of them and remove the rest.

6) Define the corners as the intersections of the tted straight lines.

7) Use a Harris corner detector to find the strongest nearby corner to each of the intersections from 6 and use this as the true corner location.

This step is particularly important if there is a lot of radial distortion.

4) Define the corners as the intersections of these fitted lines by calculating the cross product of the homogenous representation of the lines.

3 Estimating Camera Parameters

A 2D point is denoted by $\mathrm{m} = [u, v]^T$. A 3D point is denoted by $\mathrm{M} = [X, Y, Z]^T$. We use $\tilde{\mathrm{x}}$ to denote the augmented vector by adding 1 as the last element: $\tilde{\mathrm{m}} = [u, v, 1]^T$ and $\tilde{\mathrm{M}} = [X, Y, Z, 1]^T$. A camera is modeled by the usual pinhole: the relationship between a 3D point M and its image projection m is given by

$$s\tilde{\mathrm{m}} = \mathrm{A}\,[\mathrm{R}\,\mathrm{t}]\,\tilde{\mathrm{M}} \tag{1}$$

where s is an arbitrary scale factor, $[\mathrm{R}\,\mathrm{t}]$, called the extrinsic parameters, is the rotation and translation which relates the world coordinate system to the camera coordinate system, and A, called the camera intrinsic matrix, is given by

$$A = \begin{bmatrix} \alpha & \gamma & u_0 \\ 0 & \beta & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2}$$

with $(u0\, v0)$ the coordinates of the principal point, $\alpha$and β the scale factors in image u and v axes, and γ the parameter describing the skewness of the two image axes. Without loss of generality, we assume the model plane is on $\mathrm{Z} = 0$ of the world coordinate system. Let's denote the ith column of the rotation matrix R by r_i. From (1), we have

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathrm{A}\begin{bmatrix} r_1 & r_2 & r_3 & t \end{bmatrix}\begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix} \tag{3}$$

$$= \mathrm{A}\begin{bmatrix} r_1 & r_2 & t \end{bmatrix}\begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} \tag{4}$$

By abuse of notation, we still use M to denote a point on the model plane, but $\mathrm{M} = [X, Y]^T$ since Z is always equal to 0. In turn,$\mathrm{M} = [X, Y, 1]^T$. Therefore, a model point M and its image m is related by a homography H:

$$s\tilde{\mathrm{m}} = \mathrm{H}\tilde{\mathrm{M}} \quad \text{with } \mathrm{H} = \mathrm{A}\begin{bmatrix} r_1 & r_2 & t \end{bmatrix} \tag{5}$$

As is clear, the3×3 matrix H is defined up to a scale factor. Given an image of the model plane, an homography can be estimated. Let's denote it by $\mathrm{H} = \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix}$. From (2), we have

$$\begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix} = \lambda \mathrm{A}\begin{bmatrix} r_1 & r_2 & t \end{bmatrix} \tag{6}$$

where λ is an arbitrary scalar. Using the knowledge that r_1 and r_2 are orthonormal, we have

$$\mathrm{h}_1^{\mathrm{T}}\mathrm{A}^{-\mathrm{T}}\mathrm{A}^{-1}\mathrm{h}_2 = 0 \tag{7}$$

$$\mathrm{h}_1^{\mathrm{T}}\mathrm{A}^{-\mathrm{T}}\mathrm{A}^{-1}\mathrm{h}_1 = \mathrm{h}_2^{\mathrm{T}}\mathrm{A}^{-\mathrm{T}}\mathrm{A}^{-1}\mathrm{h}_2 \tag{8}$$

These are the two basic constraints on the intrinsic parameters, given one homography. Because a homography has 8 degrees of freedom and there are 6 extrinsic parameters (3 for rotation and 3 for translation), we can only obtain 2 constraints on the intrinsic parameters.

Let $B = A^{-T}A^{-1} = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{12} & B_{22} & B_{23} \\ B_{13} & B_{23} & B_{33} \end{bmatrix}$. Note that B is symmetric, defined by a 6D vector

$$\mathrm{b} = \begin{bmatrix} B_{11} & B_{12} & B_{22} & B_{13} & B_{23} & B_{33} \end{bmatrix}^T \tag{9}$$

Let the ith column vector of H be $\mathrm{h_i} = [h_{i1}, h_{i2}, h_{i3}]^T$. Then, we have

$$\mathrm{h_i^T B h_j} = \mathrm{v_{ij}^T b} \tag{10}$$

with

$$v_{ij} = \begin{bmatrix} h_{i1}h_{j1}, & h_{i1}h_{j2} + h_{i2}h_{j1}, & h_{i2}h_{j2}, & h_{i3}h_{j1} + h_{i1}h_{j3}, & h_{i3}h_{j2} + h_{i2}h_{j3}, & h_{i3}h_{j3} \end{bmatrix}^T \tag{11}$$

Therefore, the two fundamental constraints (3) and (4), from a given homography, can be rewritten as 2 homogeneous equations in b:

$$\begin{bmatrix} v_{12}^T \\ (v_{11} - v_{22})^T \end{bmatrix} \mathrm{b} = 0 \tag{12}$$

If n images of the model plane are observed, by stacking n such equations as (8) we have

Vb = 0 ; (9)

$$\mathrm{Vb} = 0 \tag{13}$$

where V is a 2×6matrix. If $n \geqq 3$, we will have in general a unique solution b defined up to a scale factor. Once b is estimated, we can compute all camera intrinsic matrix A. Once A is known, the extrinsic parameters for each image is readily computed. From (2), we have

r1 = ¸A¡1h1 r2 = ¸A¡1h2 r3 = r1 £ r2 t = ¸A¡1h3

$$\mathrm{r}_1 = \lambda \mathrm{A}^{-1}\mathrm{h}_1 \tag{14}$$

$$\mathrm{r}_2 = \lambda \mathrm{A}^{-1}\mathrm{h}_2 \tag{15}$$

$$\mathrm{r}_3 = \mathrm{r}_1 \times \mathrm{r}_2 \tag{16}$$

$$\mathrm{t} = \lambda \mathrm{A}^{-1}\mathrm{h}_3 \tag{17}$$

with $\lambda = 1/\left\|\mathrm{A}^{-1}\mathrm{h}_1\right\| = 1/\left\|\mathrm{A}^{-1}\mathrm{h}_2\right\|$.

As the radial distortion is expected to be small, one would expect to estimate the other five intrinsic parameters reasonable well by simply ignoring distortion. One strategy is then to estimate radial distortion parameter $k_1$and k_2 after having estimated the other parameters, which will give us the ideal pixel coordinates (u, v). Then, we have two equations for each point in each image:

$$\begin{bmatrix} (u-u_0)(x^2+y^2) & (u-u_0)(x^2+y^2)^2 \\ (v-v_0)(x^2+y^2) & (v-v_0)(x^2+y^2)^2 \end{bmatrix} \begin{bmatrix} k_1 \\ k_2 \end{bmatrix} = \begin{bmatrix} \breve{u}-u \\ \breve{v}-v \end{bmatrix} \tag{18}$$

where $(\breve{u}, \breve{v})$is the corresponding real observed image coordinates.

Given m points in n images, we can stack all equations together to obtain in total $2mn$ equations, or in matrix form as Dk = d, where $\mathrm{k} = [k_1, k_2]^T$. The linear least-squares solution is given by

$$\mathrm{k} = (\mathrm{D}^{\mathrm{T}}\mathrm{D})^{-1}\mathrm{D}^{\mathrm{T}}\mathrm{d} \tag{19}$$

We can refine this closed-form solution through maximum likelihood inference. We are given n images of a model plane and there are m points on the model plane. Assume that the image points are corrupted by independent and identically distributed noise. The maximum likelihood estimate can be obtained by minimizing the following functional:

$$\sum_{i=0}^{n}\sum_{j=0}^{m} \left\|\mathrm{m}_{\mathrm{ij}} - \hat{\mathrm{m}}(\mathrm{A}, \mathrm{k}_1, \mathrm{k}_2, \mathrm{R}_{\mathrm{i}}, \mathrm{t}_{\mathrm{i}}, \mathrm{M}_{\mathrm{j}})\right\|^2 \tag{20}$$

3.1 Summary

The recommended calibration procedure is as follows:

1. Print a pattern and attach it to a planar surface;

2. Take a few images of the model plane under different orientations by moving either the plane or the camera;

3. Detect the feature points in the images;

4. Estimate the five intrinsic parameters and all the extrinsic parameters using the closed-form solution. Estimate the coefficients of the radial distortion by solving the linear least-squares

6. Refine all parameters by minimizing (20).

4 Results

4.1 ECE661_hw6_images.zip

(a) Intrinsic camera matrix K as well as R and t for the first image (P1010001s.jpg)

internal params A = 745.696388 1.023360 324.479212
0 743.531573 240.117608
0 0 1

radial distortion k1, k2 = -0.161717 0.831244

external params [r1 r2 r3 t] =

P1010001.jpg : 0.999505 -0.013192 0.028549 -131.659114
0.014339 0.999083 -0.040332 -162.538052
-0.027991 0.040722 0.998778 854.667262

(b) For at least 4 of the images show the identfied corners in green and the reprojected corners in red.

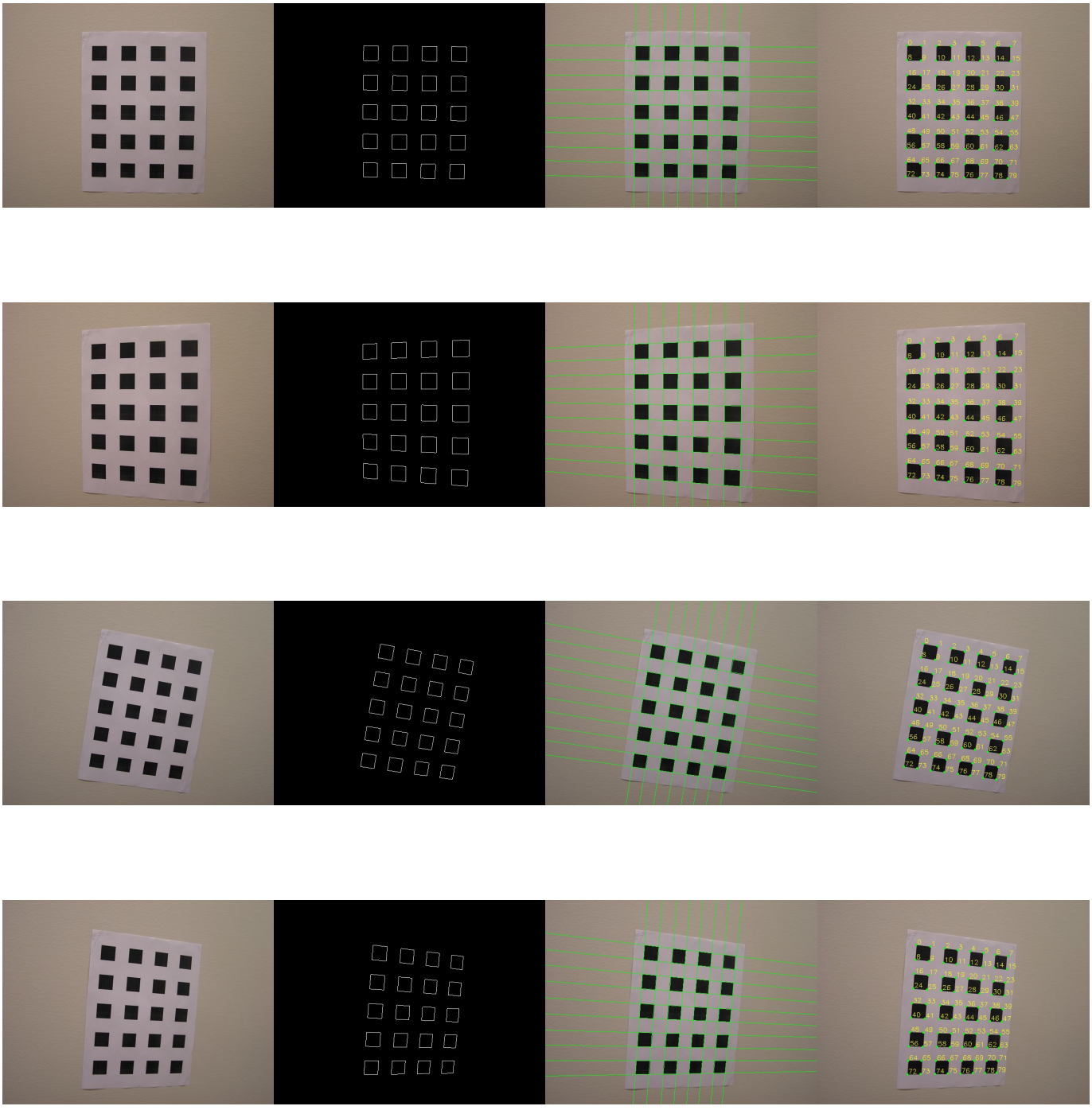

fig 1. Results of Corner Feature Finding Procedure for the P1010001.jpg (first row), P1010002.jpg (second row), P1010003.jpg (third row), P1010004.jpg (fourth row). First column is input image. Second column is detected edge, third is fitted and merged lines and fourth column is images of founded final corner points.

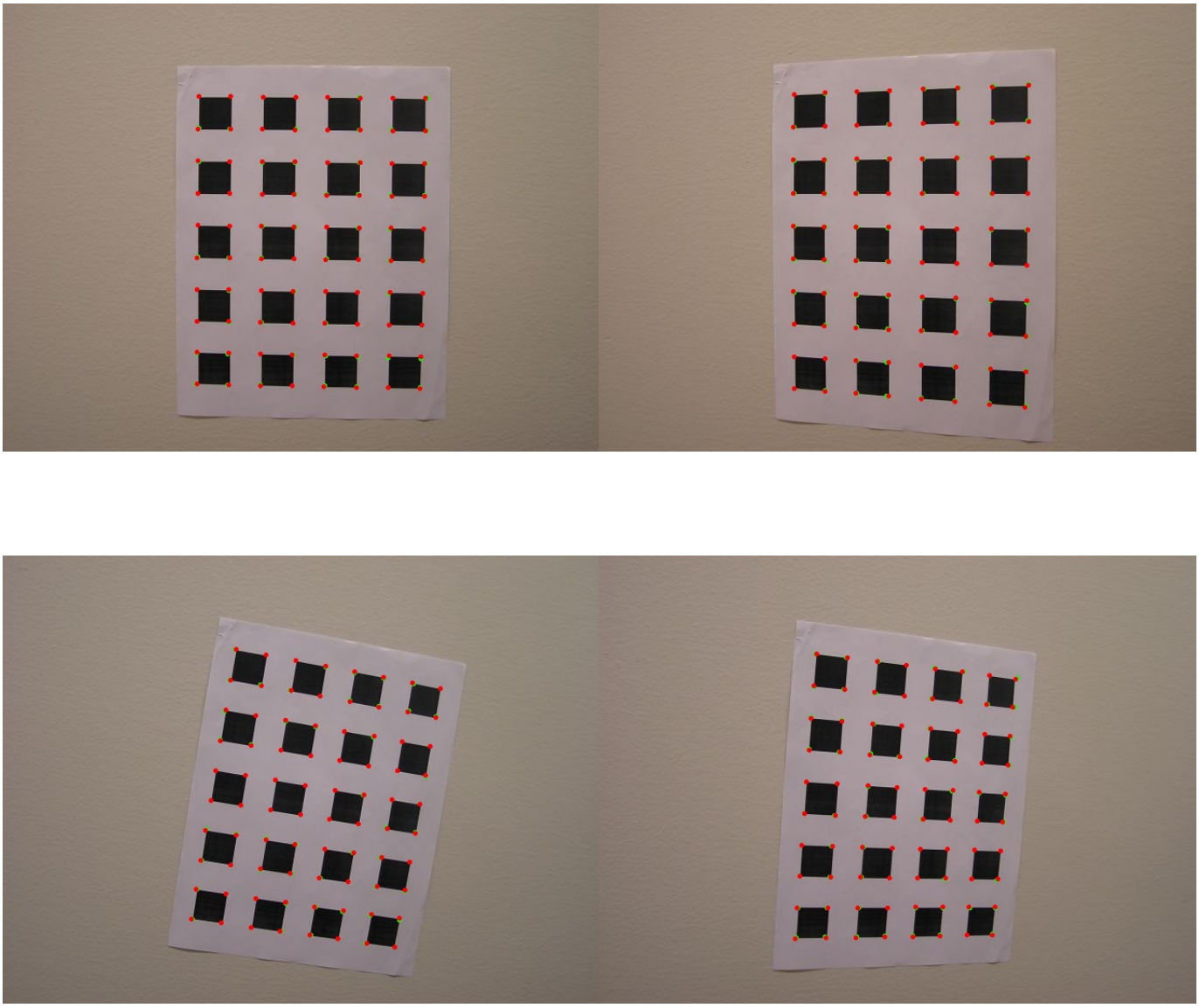

fig 2. Identfied corners(green) and the reprojected corners(red) after Calibration for the P1010001.jpg (first row left), P1010002.jpg (first row right), P1010003.jpg (second row left) and P1010004.jpg (second row right).

4.2 Images taken by my own camera

(a) Intrinsic camera matrix K as well as R and t for the first image (P001.jpg)

internal params A = 859.242272 -3.156311 311.587763

0 860.907451 211.594528

0 0 1

radial distortion k1, k2 = -0.067643 1.078421

external params [r1 r2 r3 t] =

P001.jpg : 0.941036 -0.019079 0.337769 -161.331850

0.036824 0.998253 -0.046207 -168.937930

-0.336297 0.055920 0.940094 1368.850215

(b) For at least 4 of the images show the identied corners in green and the reprojected corners in red.

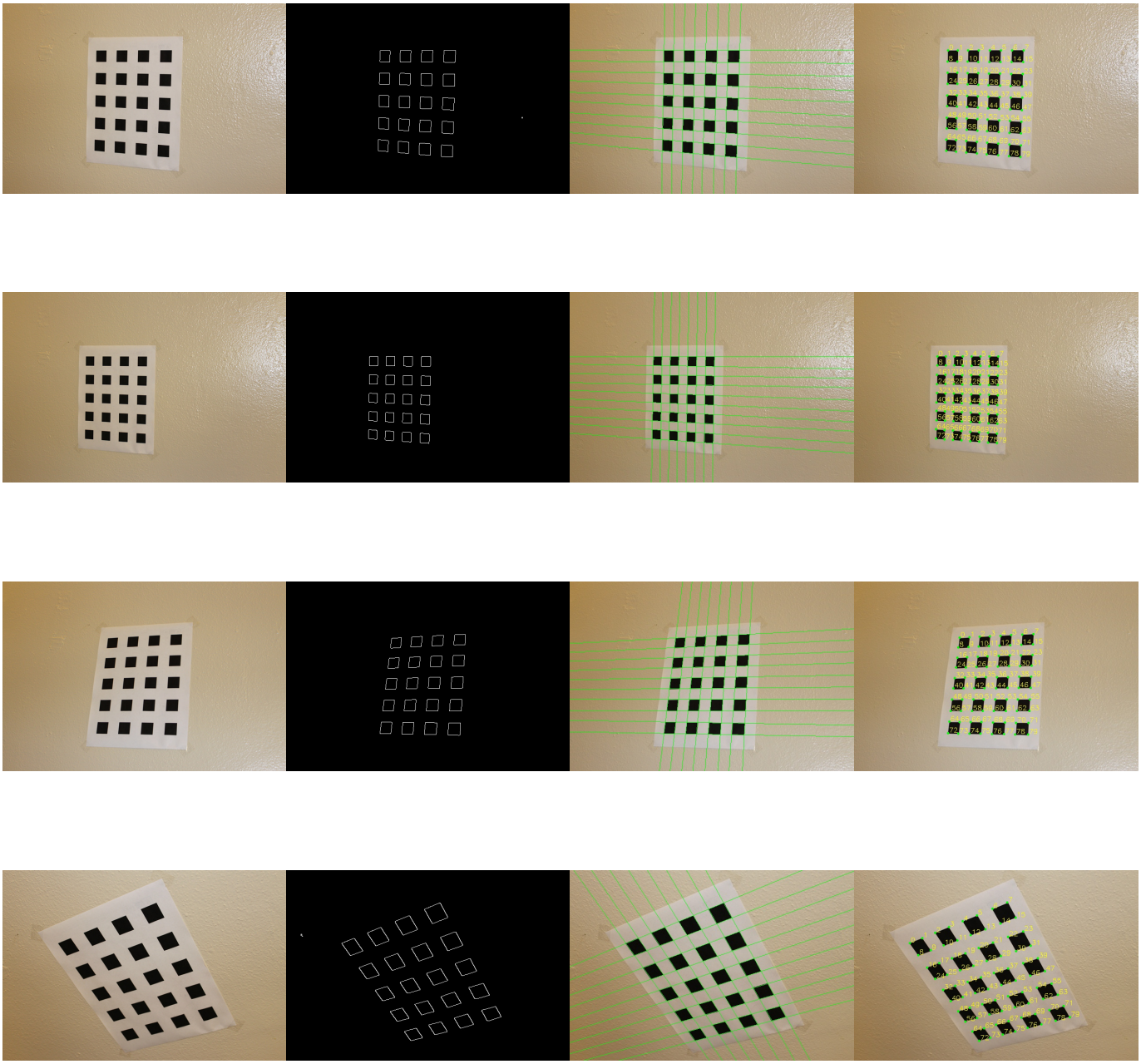

fig 3. Results of Corner Feature Finding Procedure for the P001.jpg (first row), P002.jpg (second row), P003.jpg (third row), P004.jpg (fourth row). First column is input image. Second column is detected edge, third is fitted and merged lines and fourth column is images of founded final corner points.

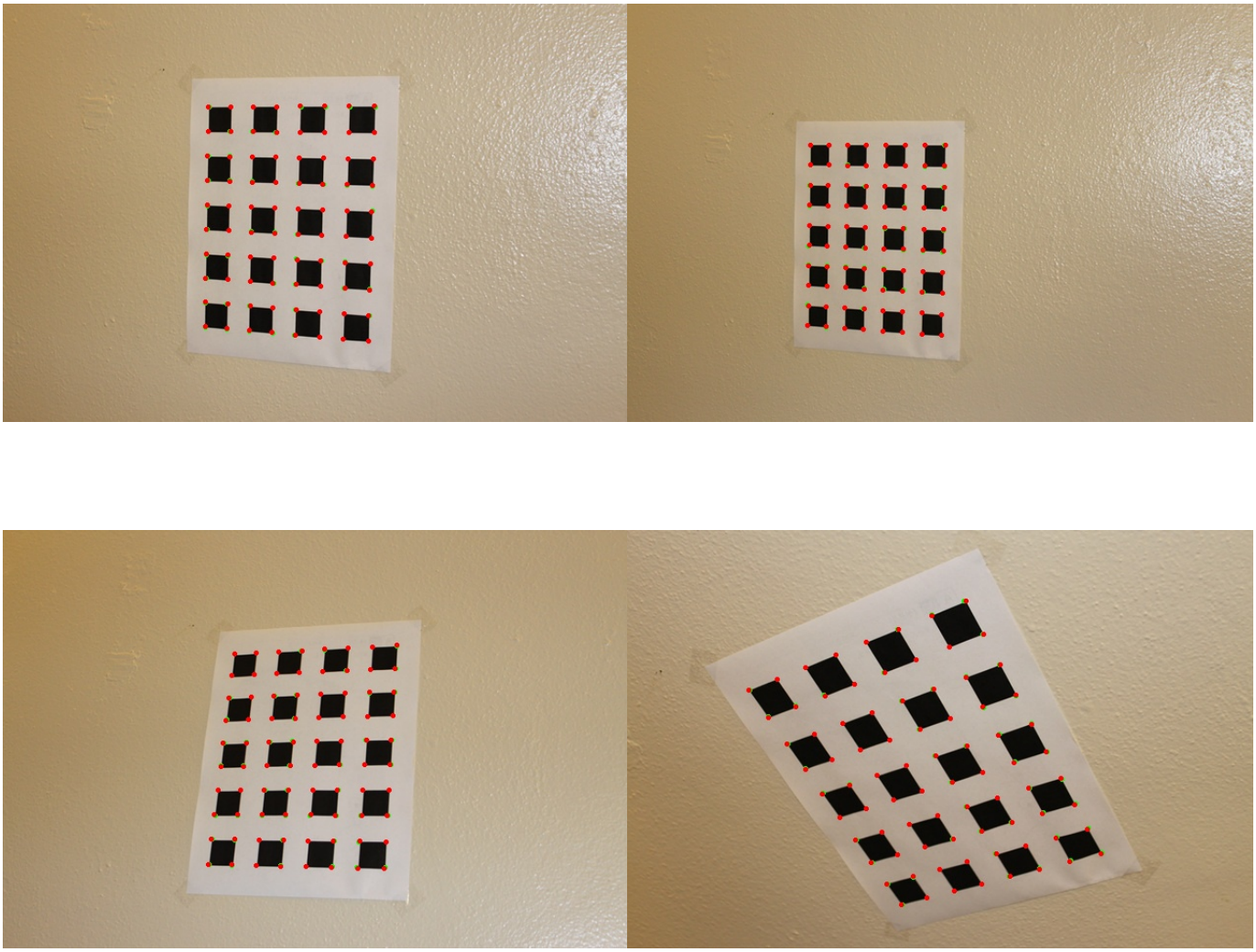

fig 4. Identfied corners(green) and the reprojected corners(red) after Calibration for the P001.jpg (first row left), P002.jpg (first row right), P003.jpg (second row left) and P004.jpg (second row right).

5. Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "cv.h"
#include "cxcore.h"
#include "highgui.h"
// For check program execution time check
#include <windows.h>
#include <time.h>
#include "levmar.h"

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

#define WINSIZE_CORNERFIND      3
#define TH_CORNERDIST           25
#define K_CONST                         0.04
#define TH_CORNERPOINT          100

#define TH_MATCH_NCC            0.00 //0.63(sample), 0.85(chair), 0.70(bed)
#define WINSIZE_NCC                     35
#define MAX_CORNERS                     10000

#define TH_DISTANCE                     10 // threshold of distance in weighted RANSAC // 30

#define MAX_ITER_NUM            50 // max iteration number for minimization
#define ERROR_POW2                      1e-7 // max iteration number for minimization

#define OPTIMIZE_DLT            0       // DLT
#define OPTIMIZE_GD                     1       // Gradient_Descent
#define OPTIMIZE_GN                     2       // Gauss_Newton
#define OPTIMIZE_LM                     3       // Levenberg_Marquardt
#define OPTIMIZE_DL                     4       // Dog_Leg

#define MAX_NUM_CORNER          80
#define MAX_IMG_NUM                     10
#define NUM_HORIZONTAL          8
#define NUM_VERTICAL            10
#define WORLD_SCALE                     40

typedef struct _CornerData
{
        int num;
        CvPoint pt[MAX_CORNERS];
} CornerData;
```

```
typedef struct _CornerPair
{
	int num;
	CvPoint ptx[MAX_CORNERS];
	CvPoint ptxp[MAX_CORNERS];
	float sm_val[MAX_CORNERS];
} CornerPair;

typedef struct{
	int num;
	// intrinsic parameters
	double alpha;
	double beta;
	double u0;
	double v0;
	double gamma;
	double k1;
	double k2;
	// extrinsic parameters
	double r1[MAX_IMG_NUM][3];
	double r2[MAX_IMG_NUM][3];
	double r3[MAX_IMG_NUM][3];
	double v[MAX_IMG_NUM][3];
	double t[MAX_IMG_NUM][3];
}CameraPara;

typedef struct{
	int		num;
	int		num_pair[MAX_IMG_NUM];
	double	H[MAX_IMG_NUM][3][3];
}HMatrix;

typedef struct {
	int num_inlier;
	int *index;
	CvPoint2D64f *ptx, *ptxp;
}PtsPairs;

IplImage *imga, *imgb;
CornerPair cnpair, incnpair;
CornerData cndata_x, cndata_xp;
char imgprefix[50] = "P10100";
char imgname[50], filename[50];

//-------------------------------------------------------------//
// Error_n_Jacobian
//-------------------------------------------------------------//
void Error_n_Jacobian(CvMat *error, CvMat *jacob, CornerPair incnpair, CvMat *h)
```

```
{
        double  xhat, yhat, what, what_pow2, htmp[9];
        double ptxx, ptxy, ptxpx, ptxpy;
        int i;

        cvZero(error);
        cvZero(jacob);
        htmp[0] = cvmGet(h,0,0);
        htmp[1] = cvmGet(h,1,0);
        htmp[2] = cvmGet(h,2,0);
        htmp[3] = cvmGet(h,3,0);
        htmp[4] = cvmGet(h,4,0);
        htmp[5] = cvmGet(h,5,0);
        htmp[6] = cvmGet(h,6,0);
        htmp[7] = cvmGet(h,7,0);
        htmp[8] = cvmGet(h,8,0);
        for (i=0; i<incnpair.num;i++){
                ptxx = (double)incnpair.ptx[i].x;
                ptxy = (double)incnpair.ptx[i].y;
                ptxpx = (double)incnpair.ptxp[i].x;
                ptxpy = (double)incnpair.ptxp[i].y;
                xhat =   htmp[0]*ptxx+htmp[1]*ptxy+htmp[2];
                yhat =   htmp[3]*ptxx+htmp[4]*ptxy+htmp[5];
                what =   htmp[6]*ptxx+htmp[7]*ptxy+htmp[8];
                what_pow2 = what*what;
#ifdef DEBUG
                if(i==0){
                        printf("xhat[%d] = %f, yhat[%d] = %f, what[%d]
= %f₩n",i,xhat,i,yhat,i,what);
                        printf("incnpair.ptxp[%d].x = %f, xhat/what[%d]
= %f₩n",i,ptxpx,i,xhat/what);
                        printf("incnpair.ptxp[%d].y = %f, yhat/what[%d]
= %f₩n",i,ptxpy,i,yhat/what);
                }
#endif
                // error vector
                cvmSet(error,i*2,0,ptxpx - xhat/what);
                cvmSet(error,i*2+1,0,ptxpy - yhat/what);
#ifdef DEBUG
                if(i==0){
                        printf("error[%d] = %f₩n",i*2,cvmGet(error,i*2,0));
                        printf("error[%d] = %f₩n",i*2+1,cvmGet(error,i*2+1,0));
                }
#endif
                // Jacobian matrix
                cvmSet(jacob,i*2,0,-ptxx/what);
                cvmSet(jacob,i*2,1,-ptxy/what);
                cvmSet(jacob,i*2,2,-1./what);
```

```
                cvmSet(jacob,i*2,6,ptxx*xhat/what_pow2);
                cvmSet(jacob,i*2,7,ptxy*xhat/what_pow2);
                cvmSet(jacob,i*2,8,xhat/what_pow2);
                cvmSet(jacob,i*2+1,3,-ptxx/what);
                cvmSet(jacob,i*2+1,4,-ptxy/what);
                cvmSet(jacob,i*2+1,5,-1./what);
                cvmSet(jacob,i*2+1,6,ptxx*yhat/what_pow2);
                cvmSet(jacob,i*2+1,7,ptxy*yhat/what_pow2);
                cvmSet(jacob,i*2+1,8,yhat/what_pow2);
#ifdef DEBUG
                if(i==0){
                        printf("jacob[%d][%d] = %f₩n",i*2,0,cvmGet(jacob,i*2,0));
                        printf("jacob[%d][%d] = %f₩n",i*2,1,cvmGet(jacob,i*2,1));
                        printf("jacob[%d][%d] = %f₩n",i*2,2,cvmGet(jacob,i*2,2));
                        printf("jacob[%d][%d] = %f₩n",i*2,3,cvmGet(jacob,i*2,3));
                        printf("jacob[%d][%d] = %f₩n",i*2,4,cvmGet(jacob,i*2,4));
                        printf("jacob[%d][%d] = %f₩n",i*2,5,cvmGet(jacob,i*2,5));
                        printf("jacob[%d][%d] = %f₩n",i*2,6,cvmGet(jacob,i*2,6));
                        printf("jacob[%d][%d] = %f₩n",i*2,7,cvmGet(jacob,i*2,7));
                        printf("jacob[%d][%d] = %f₩n",i*2,8,cvmGet(jacob,i*2,8));
                        printf("jacob[%d][%d] = %f₩n",i*2+1,0,cvmGet(jacob,i*2+1,0));
                        printf("jacob[%d][%d] = %f₩n",i*2+1,1,cvmGet(jacob,i*2+1,1));
                        printf("jacob[%d][%d] = %f₩n",i*2+1,2,cvmGet(jacob,i*2+1,2));
                        printf("jacob[%d][%d] = %f₩n",i*2+1,3,cvmGet(jacob,i*2+1,3));
                        printf("jacob[%d][%d] = %f₩n",i*2+1,4,cvmGet(jacob,i*2+1,4));
                        printf("jacob[%d][%d] = %f₩n",i*2+1,5,cvmGet(jacob,i*2+1,5));
                        printf("jacob[%d][%d] = %f₩n",i*2+1,6,cvmGet(jacob,i*2+1,6));
                        printf("jacob[%d][%d] = %f₩n",i*2+1,7,cvmGet(jacob,i*2+1,7));
                        printf("jacob[%d][%d] = %f₩n",i*2+1,8,cvmGet(jacob,i*2+1,8));
                }
#endif
        }
}

//-------------------------------------------------------------//
// Gradient_Descent
//-------------------------------------------------------------//
double Gradient_Descent(CvMat *H, CornerPair incnpair, double err_pow2)
{
        double tmp, alpha = 1e-11, ch0;
        CvMat *error = cvCreateMat(incnpair.num*2,1,CV_64FC1);
        CvMat *jacob = cvCreateMat(incnpair.num*2,9,CV_64FC1);
        CvMat *jacob_T = cvCreateMat(9,incnpair.num*2,CV_64FC1);
        CvMat *jacob_T_error = cvCreateMat(9,1,CV_64FC1);
        CvMat *h0 = cvCreateMat(9,1,CV_64FC1);
        CvMat *h1 = cvCreateMat(9,1,CV_64FC1);
        CvMat *hdiff = cvCreateMat(9,1,CV_64FC1);
```

```
        // Matrix H to Vector h0
        cvmSet(h0,0,0,cvmGet(H,0,0));
        cvmSet(h0,1,0,cvmGet(H,0,1));
        cvmSet(h0,2,0,cvmGet(H,0,2));
        cvmSet(h0,3,0,cvmGet(H,1,0));
        cvmSet(h0,4,0,cvmGet(H,1,1));
        cvmSet(h0,5,0,cvmGet(H,1,2));
        cvmSet(h0,6,0,cvmGet(H,2,0));
        cvmSet(h0,7,0,cvmGet(H,2,1));
        cvmSet(h0,8,0,cvmGet(H,2,2));

        for(int k=0;k<MAX_ITER_NUM;k++){
#ifdef DEBUG
                printf("h0 = %f₩n",cvmGet(h0,0,0));
                printf("h1 = %f₩n",cvmGet(h0,1,0));
                printf("h2 = %f₩n",cvmGet(h0,2,0));
                printf("h3 = %f₩n",cvmGet(h0,3,0));
                printf("h4 = %f₩n",cvmGet(h0,4,0));
                printf("h5 = %f₩n",cvmGet(h0,5,0));
                printf("h6 = %f₩n",cvmGet(h0,6,0));
                printf("h7 = %f₩n",cvmGet(h0,7,0));
                printf("h8 = %f₩n",cvmGet(h0,8,0));
#endif
                // Calculate Jr(h) and r(h)
                Error_n_Jacobian(error, jacob, incnpair, h0);
                // Jr(h)^T
                cvmTranspose(jacob, jacob_T);
                // Jr(h)^T*r(h)
                cvMatMul(jacob_T,error,jacob_T_error);
                // h1 = h0 -2.*alpha*jr(h)^T*r(h)
                cvScaleAdd(jacob_T_error, cvScalar(-2.*alpha), h0, h1 );
                // hdiff = h1 - h0 = -2.*alpha*jr(h)^T*r(h)
                cvSub(h1, h0, hdiff);
                // || h1- h0 ||^2 < ERROR_POW2
                tmp = cvDotProduct(hdiff,hdiff);
                printf("iter = %d, || hdiff ||^2 = %f₩n", k, tmp);
                if(tmp < err_pow2)
                        break;
                cvCopy(h1,h0,NULL);
                alpha *= 0.01;
        }
        // Vector h1 to Matrix H
        cvmSet(H,0,0,cvmGet(h1,0,0));
        cvmSet(H,0,1,cvmGet(h1,1,0));
        cvmSet(H,0,2,cvmGet(h1,2,0));
        cvmSet(H,1,0,cvmGet(h1,3,0));
        cvmSet(H,1,1,cvmGet(h1,4,0));
        cvmSet(H,1,2,cvmGet(h1,5,0));
```

```
        cvmSet(H,2,0,cvmGet(h1,6,0));
        cvmSet(H,2,1,cvmGet(h1,7,0));
        cvmSet(H,2,2,cvmGet(h1,8,0));

        ch0 = cvDotProduct(error, error);

        cvReleaseMat(&error);
        cvReleaseMat(&jacob);
        cvReleaseMat(&jacob_T);
        cvReleaseMat(&jacob_T_error);
        cvReleaseMat(&h0);
        cvReleaseMat(&h1);
        cvReleaseMat(&jacob);
        cvReleaseMat(&hdiff);

        return ch0;
}

//-------------------------------------------------------------//
// Gauss_Newton
//-------------------------------------------------------------//
double Gauss_Newton(CvMat *H, CornerPair incnpair, double err_pow2)
{
        double tmp, ch0;
        CvMat *error = cvCreateMat(incnpair.num*2,1,CV_64FC1);
        CvMat *jacob = cvCreateMat(incnpair.num*2,9,CV_64FC1);
        CvMat *jacob_T = cvCreateMat(9,incnpair.num*2,CV_64FC1);
        CvMat *jacob_T_error = cvCreateMat(9,1,CV_64FC1);
        CvMat *jacob_T_jacob = cvCreateMat(9,9,CV_64FC1);
        CvMat *inv_jacob_T_jacob = cvCreateMat(9,9,CV_64FC1);
        CvMat *invjTj_jTe = cvCreateMat(9,1,CV_64FC1);
        CvMat *h0 = cvCreateMat(9,1,CV_64FC1);
        CvMat *h1 = cvCreateMat(9,1,CV_64FC1);
        CvMat *hdiff = cvCreateMat(9,1,CV_64FC1);

        // Matrix H to Vector h0
        cvmSet(h0,0,0,cvmGet(H,0,0));
        cvmSet(h0,1,0,cvmGet(H,0,1));
        cvmSet(h0,2,0,cvmGet(H,0,2));
        cvmSet(h0,3,0,cvmGet(H,1,0));
        cvmSet(h0,4,0,cvmGet(H,1,1));
        cvmSet(h0,5,0,cvmGet(H,1,2));
        cvmSet(h0,6,0,cvmGet(H,2,0));
        cvmSet(h0,7,0,cvmGet(H,2,1));
        cvmSet(h0,8,0,cvmGet(H,2,2));

        for(int k=0;k<MAX_ITER_NUM;k++){
                // Calculate Jr(h) and r(h) : 2nX9, 2nX1
```

```
                Error_n_Jacobian(error, jacob, incnpair, h0);
                // Jr(h)^T : 9X2n
                cvmTranspose(jacob, jacob_T);
                // Jr(h)^T*r(h) : 9X1
                cvMatMul(jacob_T, error, jacob_T_error);
                // Jr(h)^T*Jr(h) : 9X9
                cvMatMul(jacob_T, jacob, jacob_T_jacob);
                // (Jr(h)^T*Jr(h))^-1 : 9X9
                cvInvert(jacob_T_jacob, inv_jacob_T_jacob);
                // (Jr(h)^T*Jr(h))^-1*jr(h)^T*r(h) : 9X1
                cvMatMul(inv_jacob_T_jacob, jacob_T_error, invjTj_jTe);
                // h1 = h0 -(Jr(h)^T*Jr(h))^-1*jr(h)^T*r(h)
                cvSub(h0, invjTj_jTe, h1);
                // hdiff = h1 - h0 = -(Jr(h)^T*Jr(h))^-1*jr(h)^T*r(h)
                        // cvSub(h1, h0, hdiff);
                // || h1- h0 ||^2 < ERROR_POW2
                        // tmp = cvDotProduct(hdiff,hdiff);
                tmp = cvDotProduct(invjTj_jTe, invjTj_jTe);
                printf("iter = %d, || hdiff ||^2 = %f₩n", k, tmp);
                if(tmp < err_pow2)
                        break;
                cvCopy(h1,h0,NULL);
        }
        // Vector h1 to Matrix H
        cvmSet(H,0,0,cvmGet(h1,0,0));
        cvmSet(H,0,1,cvmGet(h1,1,0));
        cvmSet(H,0,2,cvmGet(h1,2,0));
        cvmSet(H,1,0,cvmGet(h1,3,0));
        cvmSet(H,1,1,cvmGet(h1,4,0));
        cvmSet(H,1,2,cvmGet(h1,5,0));
        cvmSet(H,2,0,cvmGet(h1,6,0));
        cvmSet(H,2,1,cvmGet(h1,7,0));
        cvmSet(H,2,2,cvmGet(h1,8,0));

        ch0 = cvDotProduct(error, error);

        cvReleaseMat(&error);
        cvReleaseMat(&jacob);
        cvReleaseMat(&jacob_T);
        cvReleaseMat(&jacob_T_error);
        cvReleaseMat(&h0);
        cvReleaseMat(&h1);
        cvReleaseMat(&jacob);
        cvReleaseMat(&hdiff);

        return ch0;
}
```

```
//-------------------------------------------------------------//
// Levenberg_Marquardt
//-------------------------------------------------------------//
double Levenberg_Marquardt(CvMat *H, CornerPair incnpair, double err_pow2)
{
        double tmp, tmpmax, mu, tau = 1e-1, rhoLM, ch0, ch1;
        CvMat *error = cvCreateMat(incnpair.num*2,1,CV_64FC1);
        CvMat *jacob = cvCreateMat(incnpair.num*2,9,CV_64FC1);
        CvMat *ident = cvCreateMat(9,9,CV_64FC1);
        CvMat *jacob_T = cvCreateMat(9,incnpair.num*2,CV_64FC1);
        CvMat *jacob_T_error = cvCreateMat(9,1,CV_64FC1);
        CvMat *jacob_T_jacob = cvCreateMat(9,9,CV_64FC1);
        CvMat *jTj_mui = cvCreateMat(9,9,CV_64FC1);
        CvMat *inv_jTj_mui = cvCreateMat(9,9,CV_64FC1);
        CvMat *diag = cvCreateMat(9,1,CV_64FC1);
        CvMat *invjTjmui_jTe = cvCreateMat(9,1,CV_64FC1);
        CvMat *h0 = cvCreateMat(9,1,CV_64FC1);
        CvMat *h1 = cvCreateMat(9,1,CV_64FC1);
        CvMat *htmp = cvCreateMat(9,1,CV_64FC1);
        CvMat *deltah = cvCreateMat(9,1,CV_64FC1);
        CvMat *jTr_mudeltah = cvCreateMat(9,1,CV_64FC1);

        cvSetIdentity(ident);

        // Matrix H to Vector h0
        cvmSet(h0,0,0,cvmGet(H,0,0));
        cvmSet(h0,1,0,cvmGet(H,0,1));
        cvmSet(h0,2,0,cvmGet(H,0,2));
        cvmSet(h0,3,0,cvmGet(H,1,0));
        cvmSet(h0,4,0,cvmGet(H,1,1));
        cvmSet(h0,5,0,cvmGet(H,1,2));
        cvmSet(h0,6,0,cvmGet(H,2,0));
        cvmSet(h0,7,0,cvmGet(H,2,1));
        cvmSet(h0,8,0,cvmGet(H,2,2));

        // To set initial mu
        Error_n_Jacobian(error, jacob, incnpair, h0);
        // Jr(h)^T : 9X2n
        cvmTranspose(jacob, jacob_T);
        // Jr(h)^T*r(h) : 9X1
        cvMatMul(jacob_T, error, jacob_T_error);
        // Jr(h)^T*Jr(h) : 9X9
        cvMatMul(jacob_T, jacob, jacob_T_jacob);
        // diag(Jr(h)^T*Jr(h)) : 9X1
        cvGetDiag(jacob_T_jacob, diag, 0);
        // find max element of diag(Jr(h)^T*Jr(h))
        tmpmax = cvmGet(diag,0,0);
        for(int i=1;i<9;i++){
```

```
        tmpmax = max(tmpmax,cvmGet(diag,i,0));
}
// initial mu
mu = tau*tmpmax;
for(int k=0;k<MAX_ITER_NUM;k++){
        // Calculate Jr(h) and r(h) : 2nX9, 2nX1
        Error_n_Jacobian(error, jacob, incnpair, h0);
        /* ------- for calculate mu -------- */
        if(k!=0){
                // C(hk+1) = ||r(hk+1)||^2 = ||error||^2
                ch1 = cvDotProduct(error, error);
                // C(hk+1) - C(hk)
                ch0 = ch1 - ch0;
                // Jr(hk)^T*r(hk)-mu*deltahk
                cvScaleAdd(deltah, cvScalar(-1.*mu), jacob_T_error, jTr_mudeltah);
                // deltah^T*(Jr(hk)^T*r(hk)-mu*deltahk)
                tmp = cvDotProduct(deltah, jTr_mudeltah);
                // rhoLM = (C(hk+1) - C(hk))/ (deltah^T*(Jr(hk)^T*r(hk)-mu*deltahk))
                rhoLM = ch0/tmp;
                // if rhoLM > 0  then muk+1 = muk*max(1/3,1-(2*rhoLM-1)^3)
                // else muk+1 = 2*muk
                if(rhoLM >0)
                        mu = mu*max(1/3, pow(1-(2*rhoLM-1),3));
                else{
                        mu = 2.*mu;
                        cvCopy(htmp,h0,NULL); // restore previous value
                        // Calculate Jr(h) and r(h) again: 2nX9, 2nX1
                        Error_n_Jacobian(error, jacob, incnpair, h0);
                }
        }
        /* ------- for update hk -------- */
        // Jr(h)^T : 9X2n
        cvmTranspose(jacob, jacob_T);
        // Jr(h)^T*r(h) : 9X1
        cvMatMul(jacob_T, error, jacob_T_error);
        // Jr(h)^T*Jr(h) : 9X9
        cvMatMul(jacob_T, jacob, jacob_T_jacob);
        // Jr(h)^T*Jr(h)+mu*I
        cvScaleAdd(ident, cvScalar(mu), jacob_T_jacob, jTj_mui);
        // (Jr(h)^T*Jr(h)+mu*I)^-1 : 9X9
        cvInvert(jTj_mui, inv_jTj_mui);
        // (Jr(h)^T*Jr(h)+mu*I)^-1*jr(h)^T*r(h)
        cvMatMul(inv_jTj_mui, jacob_T_error, invjTjmui_jTe);
        // h1 = h0 -(Jr(h)^T*Jr(h)+mu*I)^-1*jr(h)^T*r(h)
        cvSub(h0, invjTjmui_jTe, h1);
        // hdiff = h1 - h0
        cvSub(h1, h0, deltah);
        // || h1- h0 ||^2 < ERROR_POW2
```

```
                tmp = cvDotProduct(deltah, deltah);
                printf("iter = %d, || hdiff ||^2 = %f₩n", k, tmp);
                if(tmp < err_pow2)
                        break;
                // C(hk) = ||r(hk)||^2 = ||error||^2
                ch0 = cvDotProduct(error, error);
                // hk backup
                cvCopy(h0,htmp,NULL);
                // hk update
                cvCopy(h1,h0,NULL);
        }
        // Vector h1 to Matrix H
        cvmSet(H,0,0,cvmGet(h1,0,0));
        cvmSet(H,0,1,cvmGet(h1,1,0));
        cvmSet(H,0,2,cvmGet(h1,2,0));
        cvmSet(H,1,0,cvmGet(h1,3,0));
        cvmSet(H,1,1,cvmGet(h1,4,0));
        cvmSet(H,1,2,cvmGet(h1,5,0));
        cvmSet(H,2,0,cvmGet(h1,6,0));
        cvmSet(H,2,1,cvmGet(h1,7,0));
        cvmSet(H,2,2,cvmGet(h1,8,0));

        ch0 = cvDotProduct(error, error);

        cvReleaseMat(&error);
        cvReleaseMat(&jacob);
        cvReleaseMat(&jacob_T);
        cvReleaseMat(&jacob_T_error);
        cvReleaseMat(&jacob_T_jacob);
        cvReleaseMat(&h0);
        cvReleaseMat(&h1);
        cvReleaseMat(&htmp);
        cvReleaseMat(&deltah);
        cvReleaseMat(&ident);
        cvReleaseMat(&jTj_mui);
        cvReleaseMat(&inv_jTj_mui);
        cvReleaseMat(&diag);
        cvReleaseMat(&invjTjmui_jTe);
        cvReleaseMat(&jTr_mudeltah);

        return ch0;
}

//-------------------------------------------------------------//
// Dog_Leg
//-------------------------------------------------------------//
double Dog_Leg(CvMat *H, CornerPair incnpair, double err_pow2)
{
```

```
int j=0;
double tmp, deltahGD_size, deltahGN_size;
double rhoDL, Delta = 1., ch0, ch1, jTe_size, jjTe_size;
double beta, a, b, c;
CvMat *error = cvCreateMat(incnpair.num*2,1,CV_64FC1);
CvMat *jacob = cvCreateMat(incnpair.num*2,9,CV_64FC1);
CvMat *ident = cvCreateMat(9,9,CV_64FC1);
CvMat *jacob_T = cvCreateMat(9,incnpair.num*2,CV_64FC1);
CvMat *jacob_T_error = cvCreateMat(9,1,CV_64FC1);
CvMat *jacob_T_jacob = cvCreateMat(9,9,CV_64FC1);
CvMat *h0 = cvCreateMat(9,1,CV_64FC1);
CvMat *h1 = cvCreateMat(9,1,CV_64FC1);
CvMat *hGN = cvCreateMat(9,1,CV_64FC1);
CvMat *hGD = cvCreateMat(9,1,CV_64FC1);
CvMat *deltah = cvCreateMat(9,1,CV_64FC1);
CvMat *deltah_T = cvCreateMat(1,9,CV_64FC1);
CvMat *deltahGD = cvCreateMat(9,1,CV_64FC1);
CvMat *deltahGN = cvCreateMat(9,1,CV_64FC1);
CvMat *dhGN_GD = cvCreateMat(9,1,CV_64FC1);
CvMat *dhGD_d_dhGN_GD = cvCreateMat(9,1,CV_64FC1);
CvMat *error_T = cvCreateMat(1,incnpair.num*2,CV_64FC1);
CvMat *error_T_jacob = cvCreateMat(1,9,CV_64FC1);
CvMat *dhT_jTj = cvCreateMat(1,9,CV_64FC1);
CvMat *dhT_jTj_dh = cvCreateMat(1,1,CV_64FC1);
CvMat *eTj_dh = cvCreateMat(1,1,CV_64FC1);
CvMat *jTj_mui = cvCreateMat(9,9,CV_64FC1);
CvMat *inv_jTj_mui = cvCreateMat(9,9,CV_64FC1);
CvMat *invjTjmui_jTe = cvCreateMat(9,1,CV_64FC1);
CvMat *jjTe = cvCreateMat(incnpair.num*2,1,CV_64FC1);
CvMat *htmp = cvCreateMat(9,1,CV_64FC1);

cvSetIdentity(ident);

// Matrix H to Vector h0
cvmSet(h0,0,0,cvmGet(H,0,0));
cvmSet(h0,1,0,cvmGet(H,0,1));
cvmSet(h0,2,0,cvmGet(H,0,2));
cvmSet(h0,3,0,cvmGet(H,1,0));
cvmSet(h0,4,0,cvmGet(H,1,1));
cvmSet(h0,5,0,cvmGet(H,1,2));
cvmSet(h0,6,0,cvmGet(H,2,0));
cvmSet(h0,7,0,cvmGet(H,2,1));
cvmSet(h0,8,0,cvmGet(H,2,2));

for(int k=0;k<MAX_ITER_NUM;k++){
        // Calculate Jr(h) and r(h) : 2nX9, 2nX1
        Error_n_Jacobian(error, jacob, incnpair, h0);
        /* ------- for calculate Delta -------- */
```

```
		if(k!=0){
			// C(hk+1) = ||r(hk+1)||^2 = ||error||^2
			ch1 = cvDotProduct(error, error);
			// C(hk+1) - C(hk)
			ch0 = ch1 - ch0;
			// deltah^T : 1X9
			cvmTranspose(deltah, deltah_T);
			// deltah^T*Jr(h)^T*Jr(h) : 1X9
			cvMatMul(deltah_T, jacob_T_jacob, dhT_jTj);
			// deltah^T*Jr(h)^T*Jr(h)*deltah : 1X1
			cvMatMul(dhT_jTj, deltah, dhT_jTj_dh);
			// r(h)^T*Jr(h) : 1X9
			cvMatMul(error_T, jacob, error_T_jacob);
			// r(h)^T*Jr(h)*deltah : 1X1
			cvMatMul(error_T_jacob, deltah, eTj_dh);
			// rhoLM = (C(hk+1) - C(hk))/
(deltah^T*Jr(h)^T*Jr(h)*deltah+2*r(h)^T*Jr(h)*deltah))
			tmp = cvmGet(dhT_jTj_dh,0,0) + 2*cvmGet(eTj_dh,0,0);
			rhoDL = ch0/tmp;
			// Delta update
			if(rhoDL<=0){
				Delta = Delta/4.;
				cvCopy(htmp,h0,NULL); // restore previous value
				// Calculate Jr(h) and r(h) again: 2nX9, 2nX1
				Error_n_Jacobian(error, jacob, incnpair, h0);
			}
			else if(rhoDL <= 1/4){
				Delta = Delta/4.;
			}
			else if((rhoDL>1/4)&&(rhoDL<=3/4))
				Delta = Delta;
			else
				Delta = 2.*Delta;
		}
		/* ------- for update hk -------- */
		// Jr(h)^T : 9X2n
		cvmTranspose(jacob, jacob_T);
		// Jr(h)^T*r(h) : 9X1
		cvMatMul(jacob_T, error, jacob_T_error);
		// Jr(h)^T*Jr(h) : 9X9
		cvMatMul(jacob_T, jacob, jacob_T_jacob);
		// r(h)^T : 1X2n
		cvmTranspose(error, error_T);
		// r(h)^T*Jr(h) : 1X9
		cvMatMul(error_T, jacob, error_T_jacob);
			/* hGN */
		// Jr(h)^T*Jr(h)
		cvScaleAdd(ident, cvScalar(0), jacob_T_jacob, jTj_mui);
```

```
// (Jr(h)^T*Jr(h)+mu*I)^-1 : 9X9
cvInvert(jTj_mui, inv_jTj_mui);
// (Jr(h)^T*Jr(h)+mu*I)^-1*jr(h)^T*r(h)
cvMatMul(inv_jTj_mui, jacob_T_error, invjTjmui_jTe);
// hGN = h0 -(Jr(h)^T*Jr(h)+mu*I)^-1*jr(h)^T*r(h)
cvSub(h0, invjTjmui_jTe, hGN);
// deltahGN = hGN - h0
cvSub(hGN, h0, deltahGN);
// || hGN- h0 ||
deltahGN_size = sqrt(cvDotProduct(deltahGN, deltahGN));
        /* hGD */
// ||Jr(h)^T*r(h)||^2
jTe_size = cvDotProduct(jacob_T_error, jacob_T_error);
// Jr(h)*Jr(h)^T*r(h) : 2nX1
cvMatMul(jacob, jacob_T_error, jjTe);
// ||Jr(h)*Jr(h)^T*r(h)||^2
jjTe_size = cvDotProduct(jjTe, jjTe);
// hGD = h0-(||Jr(h)^T*r(h)||^2/||Jr(h)*Jr(h)^T*r(h)||^2)*Jr(h)^T*r(h)
cvScaleAdd(jacob_T_error, cvScalar(-1.*jTe_size/jjTe_size), h0, hGD);
// deltahGD = hGD - h0
cvSub(hGD, h0, deltahGD);
// || hGD- h0 ||
deltahGD_size = sqrt(cvDotProduct(deltahGD, deltahGD));
        /* update h1 */
if(Delta >= deltahGN_size)
        // h1 = h0 + deltahGN
        cvAdd(h0,deltahGN,h1);
else if((Delta < deltahGN_size)&&(Delta >= deltahGD_size)){
        // deltahGN - deltahGD
        cvSub(deltahGN, deltahGD, dhGN_GD);
        // beta
        a = cvDotProduct(dhGN_GD, dhGN_GD);
        b = 2*cvDotProduct(deltahGD, dhGN_GD);
        c = cvDotProduct(deltahGD, deltahGD)-Delta*Delta;
        beta = (sqrt(b*b-4*a*c)-b)/(2*a);
        // h1 = h0 + deltahGD + beta*(deltahGN-deltahGD)
        cvScaleAdd(dhGN_GD, cvScalar(beta), deltahGD, dhGD_d_dhGN_GD);
        cvAdd(h0, dhGD_d_dhGN_GD, h1);
}
else
        // h1 = h0 + Delta/||deltahGD||*deltahGD
        cvScaleAdd(deltahGD, cvScalar(Delta/deltahGD_size), h0, h1);

// hdiff = h1 - h0
cvSub(h1, h0, deltah);
// || h1- h0 ||^2 < ERROR_POW2
tmp = cvDotProduct(deltah, deltah);
printf("iter = %d, || hdiff ||^2 = %f₩n", k, tmp);
```

```
        if(tmp < err_pow2)
                break;
        // C(hk) = ||r(hk)||^2 = ||error||^2
        ch0 = cvDotProduct(error, error);
        // hk backup
        cvCopy(h0,htmp,NULL);
        // hk update
        cvCopy(h1,h0,NULL);
}
// Vector h1 to Matrix H
cvmSet(H,0,0,cvmGet(h1,0,0));
cvmSet(H,0,1,cvmGet(h1,1,0));
cvmSet(H,0,2,cvmGet(h1,2,0));
cvmSet(H,1,0,cvmGet(h1,3,0));
cvmSet(H,1,1,cvmGet(h1,4,0));
cvmSet(H,1,2,cvmGet(h1,5,0));
cvmSet(H,2,0,cvmGet(h1,6,0));
cvmSet(H,2,1,cvmGet(h1,7,0));
cvmSet(H,2,2,cvmGet(h1,8,0));

ch0 = cvDotProduct(error, error);

cvReleaseMat(&error);
cvReleaseMat(&jacob);
cvReleaseMat(&jacob_T);
cvReleaseMat(&jacob_T_error);
cvReleaseMat(&jacob_T_jacob);
cvReleaseMat(&jjTe);
cvReleaseMat(&h0);
cvReleaseMat(&h1);
cvReleaseMat(&hGN);
cvReleaseMat(&hGD);
cvReleaseMat(&htmp);
cvReleaseMat(&deltahGN);
cvReleaseMat(&deltahGD);
cvReleaseMat(&ident);
cvReleaseMat(&jTj_mui);
cvReleaseMat(&inv_jTj_mui);
cvReleaseMat(&invjTjmui_jTe);
cvReleaseMat(&dhGN_GD);
cvReleaseMat(&dhGD_d_dhGN_GD);
cvReleaseMat(&error_T);
cvReleaseMat(&error_T_jacob);
cvReleaseMat(&deltah);
cvReleaseMat(&deltah_T);
cvReleaseMat(&dhT_jTj);
cvReleaseMat(&dhT_jTj_dh);
cvReleaseMat(&eTj_dh);
```

```
        return ch0;
}

//-------------------------------------------------------------//
// SobelFilter
//-------------------------------------------------------------//
void SobelFilter(IplImage *pimg, CvMat* lx, CvMat* ly)
{
        int dx[9] = {-1, 0, 1,
                                -1, 0, 1,
                                -1, 0, 1};
        int dy[9] = {-1, -1, -1,
                                 0,  0,  0,
                                 1,  1,  1};
        double xm[9] = {-1, 0, 1,
                                        -2, 0, 2,
                                        -1, 0, 1};
        double ym[9] = {1, 2, 1,
                                        0, 0, 0,
                                        -1, -2, -1};
        int width, height, x, y, k, m;
        double sumx, sumy;
        CvScalar imgval;
        CvMat sobelx = cvMat(3,3,CV_64FC1,xm);
        CvMat sobely = cvMat(3,3,CV_64FC1,ym);

        width = pimg->width;
        height = pimg->height;

        for(y=1; y<height-1; y++)
        for(x=1; x<width-1; x++){
                sumx = 0; sumy = 0;
                for(k=-1; k<=1; k++)
                for(m=-1; m<=1; m++){
                        imgval = cvGet2D(pimg,y+k,x+m);
                        sumx += imgval.val[0]*cvmGet(&sobelx,k+1,m+1);
                        sumy += imgval.val[0]*cvmGet(&sobely,k+1,m+1);
                }
                cvmSet(lx,y,x,(sumx));
                cvmSet(ly,y,x,(sumy));
        }
}
//-------------------------------------------------------------//
// Function : CornerCheck
//-------------------------------------------------------------//
void CornerCheck(int x, int y, int *cornerNo, CvPoint *corners,
                                double *cornerVal, double cvalue)
```

```
{
        int i, exist;
        double r;

        if(cvalue > TH_CORNERPOINT )
        {
                exist=0;
                for(i=0;i<*cornerNo;i++)
                {
                        r = sqrt((double)((corners[i].x-x)*(corners[i].x-x)
                                                        +(corners[i].y-y)*(corners[i].y-y)));
                        if(r < TH_CORNERDIST)
                        {
                                if( cornerVal[i] < cvalue)
                                {
                                        corners[i].x = x;
                                        corners[i].y = y;
                                        cornerVal[i]= cvalue;
                                }
                                exist=1;
                                break;
                        }
                }
                if(exist==0)
                {
                        corners[*cornerNo].x = x;
                        corners[*cornerNo].y = y;
                        if((*cornerNo) >= MAX_CORNERS)
                        {
                                printf("Too Many Corners!!");
                        }
                        else
                        {
                                cornerVal[*cornerNo]=cvalue;
                                (*cornerNo) = (*cornerNo) + 1;
                        }
                }
        }
}


//-----------------------------------------------------------//
// FindingCorners
//-----------------------------------------------------------//
void FindingCorners(IplImage *pimg,     CornerData *corners)
{
        int width, height, i,j,k,m;
        int num = 0;
```

```
        double cornerVal[MAX_CORNERS];
        double Ixx_sum, Iyy_sum, Ixy_sum;
        double cornerness;

        CvPoint curr_point;
        width = pimg->width;
        height = pimg->height;
        CvMat *Ix = cvCreateMat(height,width,CV_64FC1);
        CvMat *Iy = cvCreateMat(height,width,CV_64FC1);

        SobelFilter(pimg, Ix, Iy);
        corners->num = 0;
        for(i=WINSIZE_CORNERFIND;i<height-WINSIZE_CORNERFIND;i++)
        for(j=WINSIZE_CORNERFIND;j<width-WINSIZE_CORNERFIND;j++)
        {
                curr_point = cvPoint(j,i);
                Ixx_sum = 0.;
                Iyy_sum = 0.;
                Ixy_sum = 0.;
                cornerness = 0.;
                for(k=-WINSIZE_CORNERFIND;k<=WINSIZE_CORNERFIND;k++)
                for(m=-WINSIZE_CORNERFIND;m<=WINSIZE_CORNERFIND;m++)
                {
                        Ixx_sum += cvmGet(Ix,i+k,j+m)
                                        *cvmGet(Ix,i+k,j+m)/10000;
                        Iyy_sum += cvmGet(Iy,i+k,j+m)
                                        *cvmGet(Iy,i+k,j+m)/10000;
                        Ixy_sum += cvmGet(Ix,i+k,j+m)
                                        *cvmGet(Iy,i+k,j+m)/10000;
                }
                cornerness = (Ixx_sum*Iyy_sum - Ixy_sum*Ixy_sum)
                                - K_CONST*(Ixx_sum+Iyy_sum)*(Ixx_sum+Iyy_sum);
                if(i> (WINSIZE_CORNERFIND+1) && i<height-(WINSIZE_CORNERFIND+1)
                        && j>(WINSIZE_CORNERFIND+1) && j<width-(WINSIZE_CORNERFIND+1))
                       CornerCheck(j,i, &(corners->num), corners->pt, cornerVal, cornerness);

        }
        cvReleaseMat(&Ix);
        cvReleaseMat(&Iy);
}

//-------------------------------------------------------------//
// NCC
//-------------------------------------------------------------//
int NCC(IplImage *img_a, IplImage *img_b, CornerData *corners1,
                CornerData *corners2, CornerPair *cnpair)
{
        int width, height,i,j, k, m, ncc_num;
```

```
int ax, ay, bx, by, num;
double amean, bmean, numer;
double asum, bsum, diff_ij, max_val;
int check;
CvPoint max_point;
CvScalar imgval_a, imgval_b;
int *matchedidx = new int [corners1->num];

width= img_a->width;
height= img_a->height;
cnpair->num = corners1->num;
ncc_num = 0;
for(i=0;i<corners1->num;i++)
{
        ax = corners1->pt[i].x;
        ay = corners1->pt[i].y;
        max_val=-10000;
        cnpair->ptx[ncc_num].x = ax;
        cnpair->ptx[ncc_num].y = ay;
        for(j=0;j<corners2->num;j++)
        {
                bx = corners2->pt[j].x;
                by = corners2->pt[j].y;
                amean = 0.;
                bmean = 0.;
                num = 0;
                for(k=-WINSIZE_NCC; k<=WINSIZE_NCC;k++)
                for(m=-WINSIZE_NCC; m<=WINSIZE_NCC;m++)
                {
                        if((ay+k)<0 || (ay+k)>=height || (ax+m)<0 || (ax+m)>=width)
                                continue;
                        if((by+k)<0 || (by+k)>=height || (bx+m)<0 || (bx+m)>=width)
                                continue;
                        imgval_a = cvGet2D(img_a, ay+k, ax+m);
                        amean += imgval_a.val[0];
                        imgval_b = cvGet2D(img_b, by+k, bx+m);
                        bmean += imgval_b.val[0];
                        num++;
                }
                amean /= num;
                bmean /= num;
                asum = 0.;
                bsum = 0.;
                numer = 0.0;
                for(k=-WINSIZE_NCC; k<=WINSIZE_NCC;k++)
                for(m=-WINSIZE_NCC; m<=WINSIZE_NCC;m++)
                {
                        if((ay+k)<0 || (ay+k)>=height || (ax+m)<0 || (ax+m)>=width)
```

```
                                continue;
                        if((by+k)<0 || (by+k)>=height || (bx+m)<0 || (bx+m)>=width)
                                continue;
                        imgval_a = cvGet2D(img_a, ay+k, ax+m);
                        imgval_b = cvGet2D(img_b, by+k, bx+m);
                        numer += (imgval_a.val[0] - amean)
                                *(imgval_b.val[0] - bmean);
                        asum += (imgval_a.val[0] - amean)
                                *(imgval_a.val[0] - amean);
                        bsum += (imgval_b.val[0] - bmean)
                                *(imgval_b.val[0] - bmean);
                }
                diff_ij = numer/ sqrt(asum*bsum);
                if(diff_ij>max_val)
                {
                        max_val=diff_ij;
                        max_point.x = bx;
                        max_point.y = by;
                        matchedidx[ncc_num] = j;
                }
        }
        cnpair->sm_val[ncc_num]=(float)max_val;
        cnpair->ptxp[ncc_num].x = max_point.x;
        cnpair->ptxp[ncc_num].y = max_point.y;
        cnpair->ptx[ncc_num].x = ax;
        cnpair->ptx[ncc_num].y = ay;

        check = 0;
        for(j=0; j<ncc_num; j++){
                if(matchedidx[j] == matchedidx[ncc_num]){
                        if(cnpair->sm_val[j] < cnpair->sm_val[ncc_num]){
                                cnpair->sm_val[j] = cnpair->sm_val[ncc_num];
                                cnpair->ptx[j].x = cnpair->ptx[ncc_num].x;
                                cnpair->ptx[j].y = cnpair->ptx[ncc_num].y;
                        }
                        check = 1;
                        break;
                }
        }
        if(check == 0)
                ncc_num++;
    }
    cnpair->num = ncc_num;
    delete matchedidx;
    return ncc_num;
}

//-------------------------------------------------------------//
```

```
// ColinearCheck
//-------------------------------------------------------------//
bool ColinearCheck(int num, CvPoint2D64f *point)
{
	int i,j,k;
	bool colinear = false;
	double tmp;

	CvMat *point1 = cvCreateMat(3,1,CV_64FC1);
	CvMat *point2 = cvCreateMat(3,1,CV_64FC1);
	CvMat *point3 = cvCreateMat(3,1,CV_64FC1);
	CvMat *line = cvCreateMat(3,1,CV_64FC1);
	for(i=0; i<num-2; i++){
		cvmSet(point1,0,0,point[i].x);
		cvmSet(point1,1,0,point[i].y);
		cvmSet(point1,2,0,1);
		for(j=i+1; j<num-1; j++){
			cvmSet(point2,0,0,point[j].x);
			cvmSet(point2,1,0,point[j].y);
			cvmSet(point2,2,0,1);
			cvCrossProduct(point1, point2, line);
			for(k=j+1; k<num; k++){
				cvmSet(point3,0,0,point[k].x);
				cvmSet(point3,1,0,point[k].y);
				cvmSet(point3,2,0,1);
				tmp = cvDotProduct(point3, line);
				if(abs(tmp) < 0.01){
					colinear = true;
					break;
				}
			}
			if(colinear == true)
				break;
		}
		if(colinear == true)
			break;
	}

	cvReleaseMat(&point1);
	cvReleaseMat(&point2);
	cvReleaseMat(&point3);
	cvReleaseMat(&line);

	return colinear;
}


//-------------------------------------------------------------//
```

```
// CalculateHomography
//---------------------------------------------------------------//
void CalculateHomography(int num, CvPoint2D64f *point1,
                                                CvPoint2D64f *point2, CvMat *H)
{
        CvMat *A = cvCreateMat(2*num, 9, CV_64FC1);
        CvMat *U = cvCreateMat(2*num, 2*num, CV_64FC1);
        CvMat *D = cvCreateMat(2*num, 9, CV_64FC1);
        CvMat *V = cvCreateMat(9, 9, CV_64FC1);
        cvZero(A);

        for(int i=0; i<num; i++){
                cvmSet(A,2*i,3,-point1[i].x);
                cvmSet(A,2*i,4,-point1[i].y);
                cvmSet(A,2*i,5,-1);
                cvmSet(A,2*i,6,point2[i].y*point1[i].x);
                cvmSet(A,2*i,7,point2[i].y*point1[i].y);
                cvmSet(A,2*i,8,point2[i].y);
                cvmSet(A,2*i+1,0,point1[i].x);
                cvmSet(A,2*i+1,1,point1[i].y);
                cvmSet(A,2*i+1,2,1);
                cvmSet(A,2*i+1,6,-point2[i].x*point1[i].x);
                cvmSet(A,2*i+1,7,-point2[i].x*point1[i].y);
                cvmSet(A,2*i+1,8,-point2[i].x);
        }

        cvSVD(A, D, U, V, CV_SVD_U_T|CV_SVD_V_T);       // SVD

        cvmSet(H, 0, 0, cvmGet(V, 8, 0));
        cvmSet(H, 0, 1, cvmGet(V, 8, 1));
        cvmSet(H, 0, 2, cvmGet(V, 8, 2));
        cvmSet(H, 1, 0, cvmGet(V, 8, 3));
        cvmSet(H, 1, 1, cvmGet(V, 8, 4));
        cvmSet(H, 1, 2, cvmGet(V, 8, 5));
        cvmSet(H, 2, 0, cvmGet(V, 8, 6));
        cvmSet(H, 2, 1, cvmGet(V, 8, 7));
        cvmSet(H, 2, 2, cvmGet(V, 8, 8));

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


//---------------------------------------------------------------//
// CalculateInlierNumber
//---------------------------------------------------------------//
```

```
int CalculateInlierNumber(int num, CvPoint *point1, CvPoint *point2,
                                        CvMat *H, CvMat *inlier_mask, double *std_d)
{
        double curr_d, sum_d, mean_d, stdev_d;
        CvPoint tmp;
        int i, num_inlier;

        CvMat *x = cvCreateMat(3,1,CV_64FC1);
        CvMat *xp = cvCreateMat(3,1,CV_64FC1);
        CvMat *point = cvCreateMat(3,1,CV_64FC1);
        CvMat *invH = cvCreateMat(3,3,CV_64FC1);
        CvMat *distance = cvCreateMat(num, 1, CV_64FC1);

        cvInvert(H, invH);

        cvZero(inlier_mask);
        sum_d = 0;
        num_inlier = 0;
        for(i=0; i<num; i++){
                cvmSet(x,0,0,point1[i].x);
                cvmSet(x,1,0,point1[i].y);
                cvmSet(x,2,0,1);
                cvmSet(xp,0,0,point2[i].x);
                cvmSet(xp,1,0,point2[i].y);
                cvmSet(xp,2,0,1);
                // d(Hx, xp)
                cvMatMul(H, x, point);
                tmp.x = (int)(cvmGet(point,0,0)/cvmGet(point,2,0));
                tmp.y = (int)(cvmGet(point,1,0)/cvmGet(point,2,0));
                curr_d = (tmp.x-point2[i].x)*(tmp.x-point2[i].x)
                                + (tmp.y-point2[i].y)*(tmp.y-point2[i].y);
                // d(x, invH xp)
                cvMatMul(invH, xp, point);
                tmp.x = (int)((cvmGet(point,0,0)/cvmGet(point,2,0)));
                tmp.y = (int)((cvmGet(point,1,0)/cvmGet(point,2,0)));
                curr_d += (tmp.x-point1[i].x)*(tmp.x-point1[i].x)
                                + (tmp.y-point1[i].y)*(tmp.y-point1[i].y);
                if(curr_d < TH_DISTANCE){
                        cvmSet(distance,i,0,curr_d);
                        sum_d += curr_d;
                        cvmSet(inlier_mask,i,0,1);
                        num_inlier++;
                }
        }

        // standard deviation
        mean_d = sum_d/(double)num_inlier;
        stdev_d = 0;
```

```
        for(i=0; i<num; i++){
                if(cvmGet(inlier_mask,i,0) == 1)
                        stdev_d += (cvmGet(distance,i,0)-mean_d)
                                        *(cvmGet(distance,i,0)-mean_d);
        }
        *std_d = stdev_d/(double) (num_inlier -1);

        cvReleaseMat(&x);
        cvReleaseMat(&xp);
        cvReleaseMat(&point);
        cvReleaseMat(&invH);
        cvReleaseMat(&distance);

        return num_inlier;
}


//-----------------------------------------------------------------//
// Normalize
//-----------------------------------------------------------------//
void Normalize(int num, CvPoint2D64f *point, CvMat *T)
{
        double scale, tx, ty, meanx, meany, tmp;
        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 += point[i].x;
                meany += point[i].y;
        }
        meanx = meanx/(double)num;
        meany = meany/(double)num;
        tmp = 0;
        for(i=0; i<num; i++){
                tmp += sqrt((point[i].x-meanx)*(point[i].x-meanx)
                                + (point[i].y-meany)*(point[i].y-meany));
        }
        tmp = tmp/(double)num;
        scale = sqrt(2.0)/tmp;
        tx = -scale * meanx;
        ty = -scale * meany;
        cvZero(T);
        cvmSet(T,0,0,scale);
        cvmSet(T,1,1,scale);
        cvmSet(T,0,2,tx);
        cvmSet(T,1,2,ty);
```

```
        cvmSet(T,2,2,1.0);

        for(i=0; i<num; i++){
                cvmSet(x,0,0,point[i].x);
                cvmSet(x,1,0,point[i].y);
                cvmSet(x,2,0,1.0);
                cvMatMul(T,x,xp);       // xp = T*x
                point[i].x = cvmGet(xp,0,0)/cvmGet(xp,2,0);
                point[i].y = cvmGet(xp,1,0)/cvmGet(xp,2,0);
        }
        cvReleaseMat(&x);
        cvReleaseMat(&xp);
}


//-------------------------------------------------------------//
// Weighted RANSAC algorithm
//-------------------------------------------------------------//
void weighted_RANSAC(CornerPair cnpair, CvMat *H, CvMat *inlier_mask)
{
        int i, j, N, sample_size, sample_count;
        int numinlier, max_numinlier;
        double curr_std_d, std_d;
        double prob, e, weight_sum, corr, uW;
        bool colinear;
        CvMat *curr_inlier_mask = cvCreateMat(cnpair.num,1,CV_64FC1);
        CvMat *curr_H = cvCreateMat(3,3,CV_64FC1);
        CvMat *Ta = cvCreateMat(3,3,CV_64FC1);
        CvMat *Tb = cvCreateMat(3,3,CV_64FC1);
        CvMat *invTb = cvCreateMat(3,3,CV_64FC1);
        CvMat *tmp_point = cvCreateMat(3,1,CV_64FC1);

        max_numinlier = -1;
        N = 500;
        sample_size = 4;
        sample_count = 0;
        prob = 0.99;

        CvPoint2D64f *curr_mpointa = new CvPoint2D64f[sample_size];
        CvPoint2D64f *curr_mpointb = new CvPoint2D64f[sample_size];
        int *curr_idx = new int[sample_size];

        weight_sum = 0;
        for (i=0;i<cnpair.num;i++){
                weight_sum += cnpair.sm_val[i];
        }
        while(N > sample_count){
                colinear = true;
```

```
        while(colinear == true){
                colinear = false;
                for(i=0; i<sample_size; i++){
                        // weighted sampling
                        uW = ((double)(rand()%1000))/1000.*weight_sum;
                        corr = 0;
                        for(j=0; j<cnpair.num; j++){
                                corr += cnpair.sm_val[j];
                                if(corr>=uW){
                                        curr_idx[i] = j;
                                        break;
                                }
                        }
                        for(j=0; j<i; j++){
                                if(curr_idx[i] == curr_idx[j]){
                                        colinear = true;
                                        break;
                                }
                        }
                        if(colinear == true) break;
                        curr_mpointa[i].x = (double)cnpair.ptx[curr_idx[i]].x;
                        curr_mpointa[i].y = (double)cnpair.ptx[curr_idx[i]].y;
                        curr_mpointb[i].x = (double)cnpair.ptxp[curr_idx[i]].x;
                        curr_mpointb[i].y = (double)cnpair.ptxp[curr_idx[i]].y;
                }
                // Colinear check
                if(colinear == false)
                        colinear = ColinearCheck(sample_size, curr_mpointa);
        }

        // Nomalized DLT
        Normalize(sample_size, curr_mpointa, Ta);
        Normalize(sample_size, curr_mpointb, Tb);

        // H = invTb * curr_H * Ta
        CalculateHomography(sample_size, curr_mpointa, curr_mpointb, curr_H);
        cvInvert(Tb, invTb);
        cvMatMul(invTb, curr_H, curr_H);
        cvMatMul(curr_H, Ta, curr_H);

        // Calculate the # of inliers
        numinlier = CalculateInlierNumber(cnpair.num, cnpair.ptx, cnpair.ptxp,

curr_H,curr_inlier_mask,&curr_std_d);
        // Update H
        if(numinlier > max_numinlier ||
                (numinlier == max_numinlier && curr_std_d < std_d))
        {
```

```
                        max_numinlier = numinlier;
                        std_d = curr_std_d;
                        cvCopy(curr_inlier_mask, inlier_mask);
                        cvCopy(curr_H, H);
                }

                // update number N
                e = 1 - (double)numinlier / (double)cnpair.num;
                N = (int)(log(1-prob)/log(1-pow(1-e,sample_size)));
                sample_count++;
                printf("%d,%d ",cnpair.num, numinlier);
        }
        cvReleaseMat(&curr_H);
        cvReleaseMat(&curr_inlier_mask);
        cvReleaseMat(&Ta);
        cvReleaseMat(&Tb);
        cvReleaseMat(&invTb);
        cvReleaseMat(&tmp_point);
        delete curr_mpointa;
        delete curr_mpointb;
        delete curr_idx;
}

//-------------------------------------------------------------//
// Feature Points Find
//-------------------------------------------------------------//
//*********************************************************
// SortLines
//*********************************************************

void Sort_Lines(CvSeq* lines, int num)
{
        int i,j;
        float tmp0, tmpt1;
        float *line0, *line1;

        for(i = 0; i < num; i++){
                for(j = 0; j < num-i-1; j++){
                        line0 = (float*)cvGetSeqElem(lines,j);
                        line1 = (float*)cvGetSeqElem(lines,j+1);
                        if(line0[0] < line1[0]){
                                tmp0 = line0[0];
                                tmpt1 = line0[1];
                                line0[0] = line1[0];
                                line0[1] = line1[1];
                                line1[0] = tmp0;
                                line1[1] = tmpt1;
                        }
```

```
                }
        }
}


//*********************************************************
// Calculate_Boarder
//*********************************************************
void Calculate_Boarder(int width, int height, float rho,
                        float theta,  CvPoint *point0, CvPoint *point1)
{
        float co, si, ta;
        float th=0.001;

        co = cos(theta);
        si = sin(theta);
        ta = tan(theta);

        if( fabs(si) < th ){
                point0->x = cvRound(rho);
                point1->x = cvRound(rho);
                point0->y = 0;
                point1->y = height;
        }
        else if( fabs(co) < th ){
                point0->y = cvRound(rho);
                point1->y = cvRound(rho);
                point0->x = 0;
                point1->x = width;
        }
        else{
                point0->x = 0;
                point0->y = cvRound(rho/si);
                if(point0->y < 0){
                        point0->x = cvRound(rho/co);
                        point0->y = 0;
                }
                else if(point0->y > height){
                        point0->x = cvRound((point0->y - height)*ta);
                        point0->y = height;
                }
                point1->x = width;
                point1->y = cvRound(rho/si - width/ta);
                if(point1->y < 0){
                        point1->x = cvRound(rho/co);
                        point1->y = 0;
                }
                else if(point1->y > height){
```

```
                        point1->x = cvRound(-1.0 * ((height - rho/si) * ta));
                        point1->y = height;
                }
        }
}

//*********************************************************
// DrawLines
//*********************************************************
void DrawLines(IplImage *img, CvSeq *lines)
{
        int i;
        CvPoint point0, point1;
        float *line;
        int width = img->width;
        int     height = img->height;

        for( i = 0; i < lines->total; i++ )
        {
                line = (float*)cvGetSeqElem(lines,i);
                Calculate_Boarder(width, height, line[0], line[1], &point0, &point1);
                cvLine(img, point0, point1, CV_RGB(0,255,0), 1, 8, 0);
        }
}

//*********************************************************
// MergeLines
//*********************************************************
void MergeLines(CvSeq* lines, int width, int height, CvSeq* lines1,
                                CvSeq* lines2, int* num1, int* num2)
{
        int i,j, num_line = lines->total, num_newline, cnt;
        float rho_sum, theta_sum, *linex, *liney;
        CvMat *mask = cvCreateMat(1,num_line,CV_32FC1);
        CvMat *new_rho = cvCreateMat(1,num_line, CV_32FC1);
        CvMat *new_theta = cvCreateMat(1,num_line, CV_32FC1);
        double theta1, theta2;
        CvPoint2D32f point;

        cvZero(mask);
        num_newline = 0;
        for(i = 0; i < num_line; i++){
                if(cvmGet(mask,0,i) == 1)
                        continue;
                linex = (float*)cvGetSeqElem(lines,i);
                rho_sum = linex[0];
                theta_sum = linex[1];
                cvmSet(mask,0,i,1.0);
```

```
        cnt = 1;
        for(j = i+1; j < num_line; j++){
                liney = (float*)cvGetSeqElem(lines,j);
                if(pow(linex[0]-liney[0], (float)2.0) < 100
                        && pow(linex[1]-liney[1],(float)2.0) < 1e-2){
                        rho_sum += liney[0];
                        theta_sum += liney[1];
                        cvmSet(mask,0,j,1.0);
                        cnt++;
                }
        }
        cvmSet(new_rho,0,num_newline,(double)rho_sum/cnt);
        cvmSet(new_theta,0,num_newline,(double)theta_sum/cnt);
        num_newline++;
}
theta1 = cvmGet(new_theta, 0, 0);
i = 1;
while(pow(abs(theta1)-abs(cvmGet(new_theta, 0, i)), 2.0) < 5*1e-1){
        i++;
}
theta2 = cvmGet(new_theta, 0, i);
*num1 = 0;      *num2 = 0;
for(i = 0; i<num_newline; i++){
        if(pow(theta1-cvmGet(new_theta, 0, i), 2.0) < 5*1e-1
                || pow(theta1-(CV_PI+cvmGet(new_theta, 0, i)), 2.0) < 5*1e-1)
        {
                point = cvPoint2D32f(cvmGet(new_rho,0,i), cvmGet(new_theta,0,i));
                cvSeqPush(lines1, &point);
                (*num1)++;
        }
}
lines1->total = *num1;
for(i = 0; i<num_newline; i++){
        if(pow(theta2-cvmGet(new_theta, 0, i), 2.0) < 5*1e-1
                || pow(theta2-(CV_PI+cvmGet(new_theta, 0, i)), 2.0) < 5*1e-1)
        {
                point = cvPoint2D32f(cvmGet(new_rho,0,i), cvmGet(new_theta,0,i));
                cvSeqPush(lines2, &point);
                (*num2)++;
        }
}
lines2->total = *num2;

Sort_Lines(lines1, *num1);
Sort_Lines(lines2, *num2);

cvReleaseMat(&mask);
cvReleaseMat(&new_rho);
```

```
        cvReleaseMat(&new_theta);
}
//*********************************************************
//*********************************************************
void OrderLines(int width, int height, CvSeq* lines1,
                                        CvSeq* lines2, int num1, int num2)
{
        int i;
        CvSeq* tmp_line;
        float *line, *line_rho;
        float *line_the, *tmpline;

        if(num1 != 8 || num2 != 10){
                printf("error : line number should be 8 and 10₩n");
                exit(0);
        }

        line_rho = (float*)cvGetSeqElem(lines1,0);
        line_the = (float*)cvGetSeqElem(lines1,1);
        if(line_rho[0]/cos(line_rho[1]) > line_the[0]/cos(line_the[1])){
                tmp_line = cvCloneSeq(lines1);
                for(i=0; i<num1; i++){
                        line = (float*)cvGetSeqElem(lines1,i);
                        tmpline = (float*)cvGetSeqElem(tmp_line,num1-1-i);
                        line[0] = tmpline[0];
                        line[1] = tmpline[1];
                }
        }
        line_rho = (float*)cvGetSeqElem(lines2,0);
        line_the = (float*)cvGetSeqElem(lines2,1);
        if(line_rho[0]/sin(line_rho[1]) > line_the[0]/sin(line_the[1])){
                tmp_line = cvCloneSeq(lines2);
                for(i=0; i<num2; i++){
                        line = (float*)cvGetSeqElem(lines2,i);
                        tmpline = (float*)cvGetSeqElem(tmp_line,num2-1-i);
                        line[0] = tmpline[0];
                        line[1] = tmpline[1];
                }
        }
}
//*********************************************************
// FindCornerPoints
//*********************************************************
int FindCornerPoints(char *imgsavename, IplImage *img,
                                        CvPoint2D64f *cpoint)
{
        char text[10], savename[20];
        int i, j, num_corner, num1, num2, tmpnum, lineWidth = 1;
```

```
 int width = img->width, height = img->height;
 float *line1, *line2;
 double hscle = 0.5, vscle = 0.5;
 IplImage *img_gray, *img_edge, *img_tmp;
 CvMemStorage *storage = cvCreateMemStorage(0);
 CvSeq *lines = 0, *lines1 = 0, *lines2 = 0, *tmpline = 0;
 CvMat *pt1 = cvCreateMat(3,1,CV_64FC1);
 CvMat *pt2 = cvCreateMat(3,1,CV_64FC1);
 CvMat *lin1 = cvCreateMat(3,1,CV_64FC1);
 CvMat *lin2 = cvCreateMat(3,1,CV_64FC1);
 CvMat *point = cvCreateMat(3,1,CV_64FC1);
 CvPoint point1, point2;
 CvFont font;

cvInitFont(&font, CV_FONT_HERSHEY_SIMPLEX|CV_FONT_ITALIC, hscle, vscle, 0, lineWidth);
 img_gray = cvCreateImage(cvSize(width,height), IPL_DEPTH_8U, 1);
 cvCvtColor(img, img_gray, CV_BGR2GRAY);

 // Canny Edge Detection
 img_edge = cvCreateImage(cvSize(width, height), IPL_DEPTH_8U, 1);
 cvCanny(img_gray, img_edge, 50, 400, 3);
 sprintf(savename, "%s", imgsavename);
 strcat(savename, "_edge.jpg");
 cvSaveImage(savename, img_edge);

 // Line Fitting
 lines = cvHoughLines2(img_edge,storage,CV_HOUGH_STANDARD,1,CV_PI/180,50,0,0 );
 lines1 = cvCloneSeq(lines, storage);
 cvClearSeq(lines1);
 lines2 = cvCloneSeq(lines, storage);
 cvClearSeq(lines2);
 img_tmp = cvCloneImage(img);
 DrawLines(img_tmp,lines);
 sprintf(savename, "%s", imgsavename);
 strcat(savename, "_Fittedlines.jpg");
 cvSaveImage(savename, img_tmp);
 MergeLines(lines, width, height, lines1, lines2, &num1, &num2);

 // Line Ordering
 // lines1 : vertical, lines2 : Horizontal
 if(num1 > num2){
         tmpline = lines1;
         lines1 = lines2;
         lines2 = tmpline;
         tmpnum = num1;
         num1 = num2;
         num2 = tmpnum;
 }
```

```
        OrderLines(width, height, lines1, lines2, num1, num2);
        img_tmp = cvCloneImage(img);
        DrawLines(img_tmp,lines1);
        DrawLines(img_tmp,lines2);
        sprintf(savename, "%s", imgsavename);
        strcat(savename, "_Mergedlines.jpg");
        cvSaveImage(savename, img_tmp);

        // Finding Corners
        img_tmp = cvCloneImage(img);
        num_corner = 0;
        for(j=0; j<num2; j++){
                line2 = (float*)cvGetSeqElem(lines2,j);
                Calculate_Boarder(width, height, line2[0], line2[1], &point1, &point2);
                cvmSet(pt1, 0, 0, point1.x);
                cvmSet(pt1, 1, 0, point1.y);
                cvmSet(pt1, 2, 0, 1.0);
                cvmSet(pt2, 0, 0, point2.x);
                cvmSet(pt2, 1, 0, point2.y);
                cvmSet(pt2, 2, 0, 1.0);
                cvCrossProduct(pt1, pt2, lin2);
                for(i=0; i<num1; i++){
                        line1 = (float*)cvGetSeqElem(lines1,i);
                        Calculate_Boarder(width, height, line1[0], line1[1], &point1,
&point2);
                        cvmSet(pt1, 0, 0, point1.x);
                        cvmSet(pt1, 1, 0, point1.y);
                        cvmSet(pt1, 2, 0, 1.0);
                        cvmSet(pt2, 0, 0, point2.x);
                        cvmSet(pt2, 1, 0, point2.y);
                        cvmSet(pt2, 2, 0, 1.0);
                        cvCrossProduct(pt1, pt2, lin1);
                        cvCrossProduct(lin2, lin1, point);
                        cpoint[num_corner].x = cvmGet(point,0,0)/cvmGet(point,2,0);
                        cpoint[num_corner].y = cvmGet(point,1,0)/cvmGet(point,2,0);
                        cvCircle(img_tmp, cvPoint((int)cpoint[num_corner].x,
(int)cpoint[num_corner].y),
                                                1, cvScalar(0, 255, 0), 2, 8, 0);
                        sprintf(text, "%d", num_corner);
                        cvPutText(img_tmp, text, cvPoint((int)cpoint[num_corner].x+2,
                                        (int)cpoint[num_corner].y-2), &font, cvScalar(0, 255,
255));
                        num_corner++;
                }
        }
        sprintf(savename, "%s", imgsavename);
        strcat(savename, "_corners.jpg");
        cvSaveImage(savename, img_tmp);
```

```
        cvReleaseImage(&img_gray);
        cvReleaseImage(&img_edge);
        cvReleaseImage(&img_tmp);
        cvReleaseMat(&lin1);
        cvReleaseMat(&lin2);
        cvReleaseMat(&pt1);
        cvReleaseMat(&pt2);
        cvReleaseMat(&point);

        return num_corner;
}

//*********************************************************
// EstimateHomography
//*********************************************************
void EstimateHomography(char *imgname, IplImage *img, CvPoint2D64f *worldpts,
                                        CvPoint2D64f *imgpts, CvMat *H, PtsPairs *ptspairs)
{
        int height, width, step, channels;
        int num, num_matched;
        int i, count;
        CornerPair cnpair, incnpair;

        height = img->height;
        width = img->width;
        step = img->widthStep;
        channels = img->nChannels;

        // Finding Corner Points
        printf("ch1 %s",imgname);
        num = FindCornerPoints(imgname, img, imgpts);
        printf("ch2 %d",num);
        if(num != MAX_NUM_CORNER)
                printf("error in corner detection₩n");
        num_matched = num;

        // weighted RANSAC algorithm
        CvMat *inlier_mask = cvCreateMat(num_matched,1,CV_64FC1);
        cnpair.num = num_matched;
        for(i=0;i<num_matched;i++){
                cnpair.ptx[i].x = (int)worldpts[i].x;
                cnpair.ptx[i].y = (int)worldpts[i].y;
                cnpair.ptxp[i].x = (int)imgpts[i].x;
                cnpair.ptxp[i].y = (int)imgpts[i].y;
                cnpair.sm_val[i] = 1;
        }
        weighted_RANSAC(cnpair, H, inlier_mask);
```

```
        printf("ch3 ");

        ptspairs->num_inlier = 0;
        incnpair.num = 0;
        count = 0;
        for(i=0; i<num_matched; i++){
                if(cvmGet(inlier_mask,i,0) == 1){
                        incnpair.ptx[count].x = (int)worldpts[i].x;
                        incnpair.ptx[count].y = (int)worldpts[i].y;
                        incnpair.ptxp[count].x = (int)imgpts[i].x;
                        incnpair.ptxp[count].y = (int)imgpts[i].y;
                        incnpair.num++;
                        ptspairs->ptx[count].x = worldpts[i].x;
                        ptspairs->ptx[count].y = worldpts[i].y;
                        ptspairs->ptxp[count].x = imgpts[i].x;
                        ptspairs->ptxp[count++].y = imgpts[i].y;
                        ptspairs->num_inlier++;

                }
        }
        printf("number of inlier: %d₩n",ptspairs->num_inlier);

        // Estimate H
        CalculateHomography(incnpair.num, ptspairs->ptx, ptspairs->ptxp, H);
        printf("ch4 ");

        // Levenberg_Marquardt
        Levenberg_Marquardt(H, incnpair, ERROR_POW2);

        cvReleaseMat(&inlier_mask);
}



//-----------------------------------------------------------//
// Camera Calibration
//-----------------------------------------------------------//
//*********************************************************
// GetVijFromH
//*********************************************************
void GetVijFromH(int i, int j, CvMat *H, CvMat *vij) {
        cvmSet(vij,0,0,cvmGet(H, 0, i-1)*cvmGet(H, 0, j-1));
        cvmSet(vij,1,0,cvmGet(H, 0, i-1)*cvmGet(H, 1, j-1)
                        + cvmGet(H, 1, i-1)*cvmGet(H, 0, j-1));
        cvmSet(vij,2,0,cvmGet(H, 1, i-1)*cvmGet(H, 1, j-1));
        cvmSet(vij,3,0,cvmGet(H, 2, i-1)*cvmGet(H, 0, j-1)
                        + cvmGet(H, 0, i-1)*cvmGet(H, 2, j-1));
        cvmSet(vij,4,0,cvmGet(H, 2, i-1)*cvmGet(H, 1, j-1)
```

```
                        + cvmGet(H, 1, i-1)*cvmGet(H, 2, j-1));
        cvmSet(vij,5,0,cvmGet(H, 2, i-1)*cvmGet(H, 2, j-1));

}
//*********************************************************
// VectNormalize
//*********************************************************
void VectNormalize(CvMat *v, double norm)
{
        for(int i=0; i<v->rows; i++)
                cvmSet(v, i, 0, cvmGet(v, i, 0)/norm);
}
//*********************************************************
// CvMatToVector
//*********************************************************
void CvMatToVector(double *v, CvMat *mat, int num)
{
        for(int i=0; i<num; i++) v[i] = cvmGet(mat, i, 0);
}
//*********************************************************
// VToRotation
//*********************************************************
void VToRotation(double vx, double vy, double vz, CvMat *R)
{
        int i;
        double theta, wx, wy, wz;
        double Wdata[9];

        theta = sqrt(pow(vx,2) + pow(vy,2) + pow(vz,2));
        wx = vx/theta;
        wy = vy/theta;
        wz = vz/theta;
        Wdata[0] = 0;   Wdata[1] = -wz; Wdata[2] = wy;
        Wdata[3] = wz;  Wdata[4] = 0;   Wdata[5] = -wx;
        Wdata[6] = -wy; Wdata[7] = wx;  Wdata[8] = 0;

        CvMat W = cvMat(3, 3, CV_64FC1, Wdata);
        CvMat* tmp = cvCreateMat(3, 3, CV_64FC1);

        cvZero(R);
        for(i=0; i<3; i++)
                cvmSet(R,i,i,1.0);
        cvmScale(&W,tmp,sin(theta));
        cvAdd(R,tmp,R);
        cvMatMul(&W,&W,tmp);
        cvmScale(tmp,tmp,1-cos(theta));
        cvAdd(R,tmp,R);
```

```
        cvReleaseMat(&tmp);
}
//*********************************************************
// CamCalibGetImgPoint
//*********************************************************
void CamCalibGetImgPoint(CvPoint2D64f X, double *para,
                                double Xtr[2], int idx, int isRadial)
{
        double alpha = para[0];
        double beta = para[1];
        double gamma = para[2];
        double u0 = para[3];
        double v0 = para[4];
        double k1 = para[5];
        double k2 = para[6];
        double vx = para[7+6*idx];
        double vy = para[7+6*idx+1];
        double vz = para[7+6*idx+2];
        double tx = para[7+6*idx+3];
        double ty = para[7+6*idx+4];
        double tz = para[7+6*idx+5];
        CvMat* R = cvCreateMat(3, 3, CV_64FC1);
        CvMat* ptX = cvCreateMat(3, 1, CV_64FC1);
        CvMat* ptx = cvCreateMat(3, 1, CV_64FC1);
        double x,y,u,v;
        double tmp;
        CvMat* A = cvCreateMat(3, 3, CV_64FC1);

        cvZero(A);
        cvmSet(A,2,2,1.0);
        cvmSet(A,0,0,alpha);
        cvmSet(A,0,1,gamma);
        cvmSet(A,0,2,u0);
        cvmSet(A,1,1,beta);
        cvmSet(A,1,2,v0);
        // V to R
        VToRotation(vx,vy,vz,R);
        // R = [r1,r2,t]
        cvmSet(R,0,2,tx);
        cvmSet(R,1,2,ty);
        cvmSet(R,2,2,tz);
        cvMatMul(A,R,A);
        // X
        cvmSet(ptX,0,0,X.x);
        cvmSet(ptX,1,0,X.y);
        cvmSet(ptX,2,0,1.0);
        // x = RX, u = ARX
        cvMatMul(R,ptX,ptx);
```

```
        x = cvmGet(ptx,0,0)/cvmGet(ptx,2,0);
        y = cvmGet(ptx,1,0)/cvmGet(ptx,2,0);
        u = u0 + alpha*x + gamma*y;
        v = v0 + beta*y;
        if(!isRadial){
                Xtr[0] = u;
                Xtr[1] = v;
        }
        else{
                tmp = pow(x,2.0)+pow(y,2.0);
                Xtr[0] = u + (u-u0)*(k1*tmp+k2*pow(tmp,2.0));
                Xtr[1] = v + (v-v0)*(k1*tmp+k2*pow(tmp,2.0));
        }
}
//*********************************************************
// CameraCalibrationErrorFunc
//*********************************************************
static void CameraCalibrationErrorFunc(double *para, double *tran_x,
                                                                        int m, int n, void
*adata)
{
        int i;
        PtsPairs * pair = (PtsPairs *)adata;
        for(i=0; i<pair->num_inlier; i++){
                CamCalibGetImgPoint(pair->ptx[i], para, tran_x+i*2, pair->index[i],1);
        }
}
//*********************************************************
// CamCalibInternal
//*********************************************************
void CamCalibInternal(CameraPara *camparam, HMatrix allH, CvMat *A)
{
        int i,j,k, num_image = allH.num;
        double lambda, b[6];
        CvMat *H = cvCreateMat(3,3,CV_64FC1);
        CvMat *V = cvCreateMat(2*num_image, 6, CV_64FC1);
        CvMat *Up = cvCreateMat(2*num_image,2*num_image,CV_64FC1);
        CvMat *Dp = cvCreateMat(2*num_image,6,CV_64FC1);
        CvMat *Vp = cvCreateMat(6,6,CV_64FC1);
        CvMat *v12 = cvCreateMat(6,1,CV_64FC1);
        CvMat *v11 = cvCreateMat(6,1,CV_64FC1);
        CvMat *v22 = cvCreateMat(6,1,CV_64FC1);

        // intrinsic parameters
        for(i=0; i<num_image; i++){
                for(j=0; j<3; j++)
                for(k=0; k<3; k++)
                        cvmSet(H,j,k,allH.H[i][j][k]);
```

```
                GetVijFromH(1, 2, H, v12);
                GetVijFromH(1, 1, H, v11);
                GetVijFromH(2, 2, H, v22);
                for(j=0; j<2; j++){
                        for(k=0; k<6; k++){
                                cvmSet(V,2*i, k, cvmGet(v12,k,0));
                                cvmSet(V,2*i+j,k, cvmGet(v11,k,0)-cvmGet(v22,k,0));
                        }
                }
        }
        // solve Vb = 0
        cvSVD(V, Dp, Up, Vp, CV_SVD_U_T|CV_SVD_V_T);
        for(i=0; i<6; i++)
                b[i] = cvmGet(Vp, 5, i);
        // b = [B11 B12 B22 B13 B23 B33]^T
        camparam->v0 = (b[1]*b[3] - b[0]*b[4])/(b[0]*b[2] - pow(b[1], 2.0));
        lambda = b[5] - (pow(b[3], 2.0) + camparam->v0*(b[1]*b[3] - b[0]*b[4]))/b[0];
        camparam->alpha = sqrt(lambda/b[0]);
        camparam->beta = sqrt((lambda*b[0])/(b[0]*b[2] - pow(b[1], 2.0)));
        camparam->gamma = -b[1]*pow(camparam->alpha, 2.0)*camparam->beta/lambda;
        camparam->u0 = (camparam->gamma*camparam->v0)/camparam->beta - (b[3]*pow(camparam-
>alpha, 2.0))/lambda;

        cvZero(A);
        cvmSet(A,2,2,1.0);
        cvmSet(A,0,0,camparam->alpha);
        cvmSet(A,0,1,camparam->gamma);
        cvmSet(A,0,2,camparam->u0);
        cvmSet(A,1,1,camparam->beta);
        cvmSet(A,1,2,camparam->v0);

        cvReleaseMat(&H);
        cvReleaseMat(&V);
        cvReleaseMat(&Vp);
        cvReleaseMat(&Up);
        cvReleaseMat(&v11);
        cvReleaseMat(&v12);
        cvReleaseMat(&v22);
        cvReleaseMat(&Dp);
}
//*********************************************************
// CamCalibExternal
//*********************************************************
void CamCalibExternal(CameraPara *camparam, HMatrix allH, CvMat *A)
{
        int i,j,k, num_image = allH.num;
        double norm, trace, theta, tmp, rr;
        CvMat *r1 = cvCreateMat(3,1,CV_64FC1);
```

```
CvMat *r2 = cvCreateMat(3,1,CV_64FC1);
CvMat *r3 = cvCreateMat(3,1,CV_64FC1);
CvMat *h1 = cvCreateMat(3,1,CV_64FC1);
CvMat *h2 = cvCreateMat(3,1,CV_64FC1);
CvMat *h3 = cvCreateMat(3,1,CV_64FC1);
CvMat *t = cvCreateMat(3,1,CV_64FC1);
CvMat *R = cvCreateMat(3,3,CV_64FC1);
CvMat *H = cvCreateMat(3,3,CV_64FC1);
CvMat *Q = cvCreateMat(3,3,CV_64FC1);
CvMat *invA = cvCreateMat(3,3,CV_64FC1);
CvMat *QU = cvCreateMat(3,3,CV_64FC1);
CvMat *trQU = cvCreateMat(3,3,CV_64FC1);
CvMat *QD = cvCreateMat(3,3,CV_64FC1);
CvMat *QV = cvCreateMat(3,3,CV_64FC1);

// Extrinsic parameters
cvInvert(A,invA);
for(i=0; i<num_image; i++){
        for(j=0; j<3; j++){
                cvmSet(h1,j,0,allH.H[i][j][0]/allH.H[i][2][2]);
                cvmSet(h2,j,0,allH.H[i][j][1]/allH.H[i][2][2]);
                cvmSet(h3,j,0,allH.H[i][j][2]/allH.H[i][2][2]);
        }
        cvMatMul(invA, h1, r1);
        norm = sqrt(cvDotProduct(r1, r1));
        VectNormalize(r1, norm);
        cvMatMul(invA, h2, r2);
        VectNormalize(r2, norm);
        cvCrossProduct(r1,r2,r3);
        cvMatMul(invA, h3, t);
        VectNormalize(t, norm);
        for(j=0; j<3; j++){
                cvmSet(Q,j,0,cvmGet(r1,j,0));
                cvmSet(Q,j,1,cvmGet(r2,j,0));
                cvmSet(Q,j,2,cvmGet(r3,j,0));
        }
        cvSVD(Q, QD, QU, QV, CV_SVD_U_T|CV_SVD_V_T);
        cvTranspose(QU, trQU);
        cvMatMul(trQU, QV, R);
        rr = cvDet(R);
        for(j=0; j<3; j++){
                camparam->r1[i][j] = cvmGet(R,j,0);
                camparam->r2[i][j] = cvmGet(R,j,1);
                camparam->r3[i][j] = cvmGet(R,j,2);
        }
        CvMatToVector(camparam->t[i], t, 3);
        trace = (cvTrace(R)).val[0];
        theta = acos((trace - 1) / 2.0);
```

```
                tmp = theta/(2*sin(theta));
                camparam->v[i][0] = tmp*(cvmGet(R, 2, 1)-cvmGet(R, 1, 2));
                camparam->v[i][1] = tmp*(cvmGet(R, 0, 2)-cvmGet(R, 2, 0));
                camparam->v[i][2] = tmp*(cvmGet(R, 1, 0)-cvmGet(R, 0, 1));
                for(j=0; j<3; j++)
                for(k=0; k<3; k++)
                        cvmSet(H,j,k,allH.H[i][j][k]/allH.H[i][2][2]);
                for(j=0; j<3; j++){
                        cvmSet(R,j,2,camparam->t[i][j]);
                }
                cvMatMul(A,R,R);
        }

        cvReleaseMat(&h1);
        cvReleaseMat(&h2);
        cvReleaseMat(&h3);
        cvReleaseMat(&invA);
        cvReleaseMat(&R);
        cvReleaseMat(&t);
        cvReleaseMat(&Q);
        cvReleaseMat(&QD);
        cvReleaseMat(&QU);
        cvReleaseMat(&QV);
        cvReleaseMat(&r1);
        cvReleaseMat(&r2);
        cvReleaseMat(&r3);
        cvReleaseMat(&H);
}

//*********************************************************
// CamCalibRadial
//*********************************************************
void CamCalibRadial(CameraPara *camparam, HMatrix allH,
                                        CvMat *A, PtsPairs ptspairs)
{
        int i,j,k;
        int num_image = allH.num;
        int num_pair = ptspairs.num_inlier;
        int index;
        double u,v,x,y;
        double tmp1, tmp2, tmp3;
        double alpha = camparam->alpha;
        double beta = camparam->beta;
        double gamma = camparam->gamma;
        double u0 = camparam->u0;
        double v0 = camparam->v0;
        CvMat *solk = cvCreateMat(2,1,CV_64FC1);
        CvMat *DT = cvCreateMat(2,2*num_pair,CV_64FC1);
```

```
CvMat *tmp = cvCreateMat(2,2,CV_64FC1);
CvMat *R = cvCreateMat(3,3,CV_64FC1);
CvMat *D = cvCreateMat(2*num_pair,2,CV_64FC1);
CvMat *d = cvCreateMat(2*num_pair,1,CV_64FC1);
CvMat *ptX = cvCreateMat(3,1,CV_64FC1);
CvMat *ptx = cvCreateMat(3,1,CV_64FC1);

index = 0;
for(i=0; i<num_image; i++){
        // set H
        for(j=0; j<3; j++)
        for(k=0; k<3; k++)
                cvmSet(R,j,k,allH.H[i][j][k]);
        for(j=0; j<allH.num_pair[i]; j++){
                cvmSet(ptX,0,0,ptspairs.ptx[index].x);
                cvmSet(ptX,1,0,ptspairs.ptx[index].y);
                cvmSet(ptX,2,0,1.0);
                cvMatMul(R,ptX,ptx);
                u = cvmGet(ptx,0,0)/cvmGet(ptx,2,0);
                v = cvmGet(ptx,1,0)/cvmGet(ptx,2,0);
                x = (u-u0)/alpha;
                y = (v-v0)/beta;
                cvmSet(d,2*index, 0, ptspairs.ptxp[index].x-u);
                cvmSet(d,2*index+1,0, ptspairs.ptxp[index].y-v);
                tmp1 = u - u0;
                tmp2 = v - v0;
                tmp3 = pow(x,2.0)+pow(y,2.0);
                cvmSet(D,2*index, 0,tmp1*tmp3);
                cvmSet(D,2*index, 1,tmp1*pow(tmp3,2.0));
                cvmSet(D,2*index+1,0,tmp2*tmp3);
                cvmSet(D,2*index+1,1,tmp2*pow(tmp3,2.0));
                index++;
        }
}
if(index != num_pair){
        printf("error CamCalibRadial!!");
        exit(0);
}
cvSolve(D,d,solk,CV_SVD);
camparam->k1 = cvmGet(solk,0,0);
camparam->k2 = cvmGet(solk,1,0);

cvReleaseMat(&D);
cvReleaseMat(&solk);
cvReleaseMat(&R);
cvReleaseMat(&d);
cvReleaseMat(&ptX);
cvReleaseMat(&ptx);
```

```
}
//*********************************************************
// InitLMParam
//*********************************************************
void InitLMParam(CameraPara *camparam, double *para, int num_image)
{
	para[0] = camparam->alpha;
	para[1] = camparam->beta;
	para[2] = camparam->gamma;
	para[3] = camparam->u0;
	para[4] = camparam->v0;
	para[5] = camparam->k1;
	para[6] = camparam->k2;
	for(int i=0; i<num_image; i++){
		para[7+6*i] = camparam->v[i][0];
		para[7+6*i+1] = camparam->v[i][1];
		para[7+6*i+2] = camparam->v[i][2];
		para[7+6*i+3] = camparam->t[i][0];
		para[7+6*i+4] = camparam->t[i][1];
		para[7+6*i+5] = camparam->t[i][2];
	}
}
//*********************************************************
// StoreLMParam
//*********************************************************
void StoreLMParam(CameraPara *camparam, double *para, int num_image)
{
	int i,j;
	CvMat *R = cvCreateMat(3,3,CV_64FC1);
	camparam->alpha = para[0];
	camparam->beta = para[1];
	camparam->gamma = para[2];
	camparam->u0 = para[3];
	camparam->v0 = para[4];
	camparam->k1 = para[5];
	camparam->k2 = para[6];
	for(i=0; i<num_image; i++){
		camparam->v[i][0] = para[7+6*i];
		camparam->v[i][1] = para[7+6*i+1];
		camparam->v[i][2] = para[7+6*i+2];
		camparam->t[i][0] = para[7+6*i+3];
		camparam->t[i][1] = para[7+6*i+4];
		camparam->t[i][2] = para[7+6*i+5];
		VToRotation(camparam->v[i][0], camparam->v[i][1], camparam->v[i][2], R);
		for(j=0; j<3; j++){
			camparam->r1[i][j] = cvmGet(R,j,0);
			camparam->r2[i][j] = cvmGet(R,j,1);
			camparam->r3[i][j] = cvmGet(R,j,2);
```

```
                }
        }
        cvReleaseMat(&R);
}
//*********************************************************
// CamCalibRefineParams
//*********************************************************
void CamCalibRefineParams(CameraPara *camparam,
                                                HMatrix allH, PtsPairs ptspairs)
{
        int i,numimg = allH.num, ret;
        double opts[LM_OPTS_SZ], info[LM_INFO_SZ];

        opts[0]=LM_INIT_MU;
        opts[1]=1E-12;
        opts[2]=1E-12;
        opts[3]=1E-15;
        opts[4]=LM_DIFF_DELTA;
        void (*err)(double *p, double *hx, int m, int n, void *adata);
        int LM_m = (5+2+numimg*6);
        int LM_n = 2*ptspairs.num_inlier;
        double *ptx = (double *)malloc(LM_n*sizeof(double));
        double *para = (double*)malloc(LM_m*sizeof(double));

        InitLMParam(camparam, para, numimg);
        for(i=0; i<ptspairs.num_inlier; i++){
                ptx[2*i] = ptspairs.ptxp[i].x;
                ptx[2*i+1] = ptspairs.ptxp[i].y;
        }
        err = CameraCalibrationErrorFunc;
        ret = dlevmar_dif(err, para, ptx, LM_m, LM_n, 1000, opts, info, NULL, NULL,
        &ptspairs);
        printf("Distortion: %f %f₩n", info[0], info[1]);

        StoreLMParam(camparam, para, numimg);
}
//*********************************************************
// CameraCalibration
//*********************************************************
void CameraCalibration(CameraPara *camparam, HMatrix allH,
                                        CvMat *A, PtsPairs ptspairs)
{
        CamCalibInternal(camparam, allH, A);
        CamCalibExternal(camparam, allH, A);
        CamCalibRadial(camparam, allH, A, ptspairs);
        CamCalibRefineParams(camparam, allH, ptspairs);
}
//*********************************************************
```

```
// PrintCamParams
//*********************************************************
void PrintCamParams(char *filename, CameraPara camparam, int num_image)
{
        int i,j;
        printf("internal params A = ₩n");
        printf("        %f %f %f₩n", camparam.alpha, camparam.gamma, camparam.u0);
        printf("        0 %f %f₩n", camparam.beta, camparam.v0);
        printf("        0 0 1₩n");
        printf("radial distortion k1, k2 = ₩n");
        printf("        %f %f₩n", camparam.k1, camparam.k2);
        printf("external params [r1 r2 r3 t] = ₩n");
        for(i=0; i<num_image; i++){
                printf(" %s : ₩n", filename);
                for(j=0; j<3; j++)
                        printf(" %f %f %f %f₩n", camparam.r1[i][j], camparam.r2[i][j],
                                                                        camparam.r3[i][j],
camparam.t[i][j]);
        }
}

//-------------------------------------------------------------//
// Main
//-------------------------------------------------------------//
void main(int argc, char *argv[])
{
        int i,j,k, num_image,num_pair;
        int index[MAX_NUM_CORNER*MAX_IMG_NUM];
        IplImage *img, *img_tmp, *img_gray, *img_trans;

        CameraPara camparam;
        CvMat *ptX = cvCreateMat(3,1,CV_64FC1);
        CvMat *ptx = cvCreateMat(3,1,CV_64FC1);
        CvMat *H = cvCreateMat(3,3,CV_64FC1);
        CvMat *invH = cvCreateMat(3,3,CV_64FC1);
        CvMat *A = cvCreateMat(3,3,CV_64FC1);
        HMatrix allH;

        CvPoint2D64f worldpts[MAX_NUM_CORNER];
        CvPoint2D64f imgpts[MAX_NUM_CORNER*MAX_IMG_NUM];
        PtsPairs ptspair_all, ptspairs;
        CvPoint2D64f pntset1[MAX_NUM_CORNER];
        CvPoint2D64f pntset2[MAX_NUM_CORNER];
        CvPoint2D64f pntset1all[MAX_NUM_CORNER*MAX_IMG_NUM];
        CvPoint2D64f pntset2all[MAX_NUM_CORNER*MAX_IMG_NUM];
        CvPoint2D64f pntset1tr[MAX_NUM_CORNER*MAX_IMG_NUM];

        ptspair_all.ptx = pntset1all;
```

```
ptspair_all.ptxp = pntset2all;
ptspairs.ptx = pntset1;
ptspairs.ptxp = pntset2;
ptspair_all.index = index;
double *para,trans_pts[2];

if(argc >= 3){
        sprintf(imgprefix, "%s", argv[1]);
        num_image = atoi(argv[2]);
}
else if(argc == 2){
        sprintf(imgprefix, "%s", argv[1]);
        num_image = 10;
}
else{
        num_image = 10;
}
para = new double[7+num_image*6];

printf("m0 ");

/* ---------------------------------------------------------------- */
/* Finding Feature Points and Estimate Homography                   */
/* ---------------------------------------------------------------- */
// specify pattern image corner points coordinate
for(j=0; j<NUM_VERTICAL; j++){
        for(i=0; i<NUM_HORIZONTAL; i++){
                worldpts[j*NUM_HORIZONTAL+i].x = (double)i*WORLD_SCALE;
                worldpts[j*NUM_HORIZONTAL+i].y = (double)j*WORLD_SCALE;
        }
}

allH.num = num_image;
num_pair = 0;
printf("m1 ");

for(i=0; i<num_image; i++){
        sprintf(imgname, "%s%02d", imgprefix,i+1);
        sprintf(filename, "%s%02d.jpg", imgprefix,i+1);
        printf("%s %s %s₩n", imgprefix, imgprefix, filename);
        img = cvLoadImage(filename);
        printf("m2 ");
        if(!img){
                printf("Load error: %s₩n", filename);
                exit(0);
        }
        // Estimate Homography
        EstimateHomography(imgname, img, worldpts,
```

```
                                        &imgpts[i*MAX_NUM_CORNER], H, &ptspairs);
        printf("m3 ");

        for(j=0; j<3; j++){
                for(k=0; k<3; k++)
                        printf("%f ", cvmGet(H,j,k)/cvmGet(H,2,2));
                printf("₩n");
        }
        printf("₩n");

        for(j=0; j<3; j++)
        for(k=0; k<3; k++)
                allH.H[i][j][k] = cvmGet(H,j,k);

        for(j=0; j<ptspairs.num_inlier; j++){
                pntset1all[num_pair].x = pntset1[j].x;
                pntset1all[num_pair].y = pntset1[j].y;
                pntset2all[num_pair].x = pntset2[j].x;
                pntset2all[num_pair].y = pntset2[j].y;
                index[num_pair++] = i;
        }
        allH.num_pair[i] = ptspairs.num_inlier;

        CvMat *check = cvCreateMat(img->height, img->width, CV_64FC1);
        cvInvert(H,invH);
        printf("m4 ");
        img_trans = cvCloneImage(img);

        cvReleaseMat(&check);
}
ptspair_all.num_inlier = num_pair;

/* ---------------------------------------------------------------- */
/* Camera Calibration                                               */
/* ---------------------------------------------------------------- */
// Solve closed-form solutions
CameraCalibration(&camparam, allH, A, ptspair_all);
printf("m5 ");
PrintCamParams(filename, camparam, num_image);

/* ---------------------------------------------------------------- */
/* Project point to image plane                                     */
/* ---------------------------------------------------------------- */
InitLMParam(&camparam, para, num_image);
printf("m9 ");
for(i=0; i<num_image; i++){
        sprintf(filename, "%s%02d.jpg", imgprefix,i+1);
        printf("%s ₩n", filename);
```

```
                img = cvLoadImage(filename);
                if(!img){
                        printf("Load error: %s₩n", filename);
                        exit(0);
                }
                img_tmp = cvCloneImage(img);
                for(j=0; j<NUM_VERTICAL; j++){
                        for(k=0; k<NUM_HORIZONTAL; k++){
                                CamCalibGetImgPoint(worldpts[j*NUM_HORIZONTAL+k], para,
trans_pts, i, 1);
                                // identified corners - green
                                cvCircle(img,

cvPoint((int)imgpts[i*MAX_NUM_CORNER+j*NUM_HORIZONTAL+k].x,

(int)imgpts[i*MAX_NUM_CORNER+j*NUM_HORIZONTAL+k].y),
                                                1, cvScalar(0, 255, 0), 2, 8, 0);
                                // reprojected corners - red
                                cvCircle(img, cvPoint((int)trans_pts[0], (int)trans_pts[1]),
                                                1, cvScalar(0, 0, 255), 2, 8, 0);
                        }
                }
                sprintf(filename, "%s%02d_refine_proj.jpg", imgprefix,i+1);
                cvSaveImage(filename, img);
        }
        printf("m10 ");

        cvReleaseMat(&A);
        cvReleaseImage(&img);
        cvReleaseImage(&img_trans);
        cvReleaseImage(&img_tmp);

}
```