ECE661 Computer Vision Homework 8

Camera Calibration

Rong Zhang

11/20/2008

1 Problem

In this homework, we implement the camera calibration algorithm described in Zhang's paper. All five of the camera intrinsic parameters and six of the extrinsic parameters are estimated based on image homography based method. In addition, radial distortion parameters k_1 and k_2 are also estimated.

2 Calibration Pattern and Fetures Points

We use a checkboard pattern for camera calibration in this homework. Suppose the pattern $\vec{X}$ is placed on model plane (Z component in world coordinate is zero) as illustrated below. Several picture are taken by changing the location and rotation angle of the camera as in Fig.1. The world coordinate of these corner points are easily obtained as shown below. For the perticular pattern used in this homework, there are 8 vertical lines and 10 horizontal lined resulting in 80 corner points.

In order to estimate the camera parameters, the homography matrices between the captured images and the calibration pattern on the model plane should be available. For image homography, the corner point detection procedures for the captured image are:

1. Apply Canny edge detector to the captured image $\vec{x}$.
2. Apply Hough line transformation to detect lines in the edge image.
3. Group the lines in previous step and order them from top to down and from left to right.
4. Define the corners as the intersections of these fitted lines by calculating the cross product of the homogenous representation of the lines.

Therefore, if the line is correctly ordered, we will get a ordered corner points as shown in Fig. 2-5. The point correspondences between the pattern and image are easily created. As shown in the results figures, after Hough transform, line grouping and ordering procedures are applied. Vertical line orders are indicated from color black gradually changing into blue while horizontal lines are indicated from black to white. The corner points are labeled from 0 to 79.

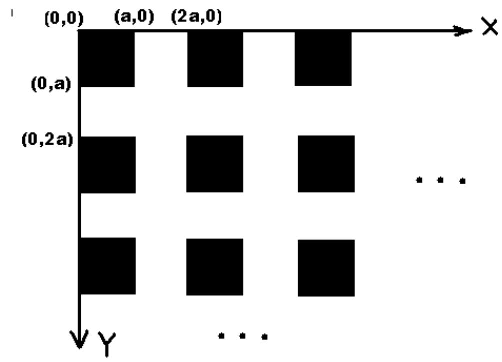


Figure 1: Calibration pattern in the model plane ($a = 40$ is used in my code)

3 Estimating Intrinsic and Extrinsic Parameters

From class, we know that the camera intrinsic parameters can be set in the matrix form

$$K = \begin{bmatrix} \alpha & \gamma & x_0 \\ 0 & \beta & y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

which has 5 degrees of freedom. If ω is the image of absolute conic, we have

$$\omega = K^{-T}K^{-1}. \tag{2}$$

Suppsoe the homography between the pattern and the image is $H = [\vec{h}_1, \vec{h}_2, \vec{h}_3]$, then we will have

$$\begin{aligned} \vec{h}_1^T \omega \vec{h}_2 &= 0 \\ \vec{h}_1^T \omega \vec{h}_1 &= h_2^T \omega \vec{h}_2. \end{aligned} \tag{3}$$

In Zhang's paper, closed form has been derived for each of the five unknown parameters in K given at least $n = 3$ homography matrices H_i, i.e., n images of the pattern with different camera rotation matrix R_i and camera translation t_i. The homography matrices are estimated from the RANSAC based algorithm with LM optimization which has been done in HW5.

Since

$$\begin{aligned}\vec{x} &= P\vec{X} = K[R,t]\vec{X} \\ &= K[\vec{r}_1,\vec{r}_2,\vec{r}_3,t][X,Y,0,1]^T \\ &= K[\vec{r}_1,\vec{r}_2,\vec{t}][X,Y,1]^T = H[X,Y,1]^T,\end{aligned} \tag{4}$$

where $H = K[r1, r2, t]$, we have

$$\begin{aligned}\vec{r}_1 &= \lambda K^{-1}\vec{h}1 \\ \vec{r}_2 &= \lambda K^{-1}\vec{h}2 \\ \vec{t} &= \lambda K^{-1}\vec{h}3,\end{aligned} \tag{5}$$

where λ is the normalization factor to make $\|r_i\| = 1$. Since R is the rotation matrix,

$$\vec{r}_3 = \vec{r}_1 \times \vec{r}_2. \tag{6}$$

4 Radial Distortion

Suppose (x, y) and $(\hat{x}, \hat{y})$ are the ideal and real normalized image coordinates, i.e.,

$$[xw, yw, w]^T = [\vec{r}_1,\vec{r}_2,\vec{t}][X,Y,1]^T. \tag{7}$$

We have

$$\begin{aligned}\hat{x} &= x + x[k_1(x^2+y^2) + k_2(x^2+y^2)^2] \\ \hat{y} &= y + y[k_1(x^2+y^2) + k_2(x^2+y^2)^2].\end{aligned} \tag{8}$$

Considering camera intrinsic matrix K, we have the ideal and real image (u, v) and $(\hat{u}\hat{v})$satisfying

$$\begin{aligned}\hat{u} &= u + (u - x_0)[k_1(x^2+y^2) + k_2(x^2+y^2)^2] \\ \hat{v} &= v + (v - y_0)[k_1(x^2+y^2) + k_2(x^2+y^2)^2].\end{aligned} \tag{9}$$

We therefore get 2 equations for each point pair

$$\begin{aligned}(u - x_0)(x^2+y^2)k_1 + (u - x_0)(x^2+y^2)^2 k_2 &= \hat{u} - u \\ (v - y_0)(x^2+y^2)k_1 + (v - y_0)(x^2+y^2)^2 k_2 &= \hat{v} - v.\end{aligned} \tag{10}$$

Given m points in n images, we have $2mn$ equations. The linear least square solution is given by

$$\vec{k} = (D^T D)^{-1} D^T \vec{d}. \tag{11}$$

Note that since the radial distortion effect is expected to be small, the parameters k_1 and k_2 can be estimated after having estimated the intrinsic and extrinsic parameters.

5 Refine Parameters based on LM algorithm

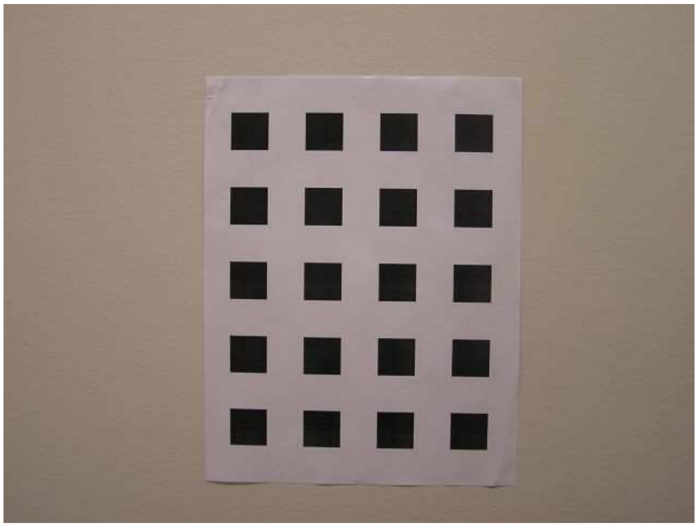
In the previous sections, the procedures for estimating intrinsic and extrinsic parameters are given. The estimated values are set as the initial values and they are further refined by LM optimization approach. The distortion function is given by

$$\sum_{i=1}^{n}\sum_{j=1}^{m}\left\|\vec{\hat{x}}_{ij} - \vec{x}(K, k_1, k_2 R_i, t_i, X_j)\right\|^2, \tag{12}$$

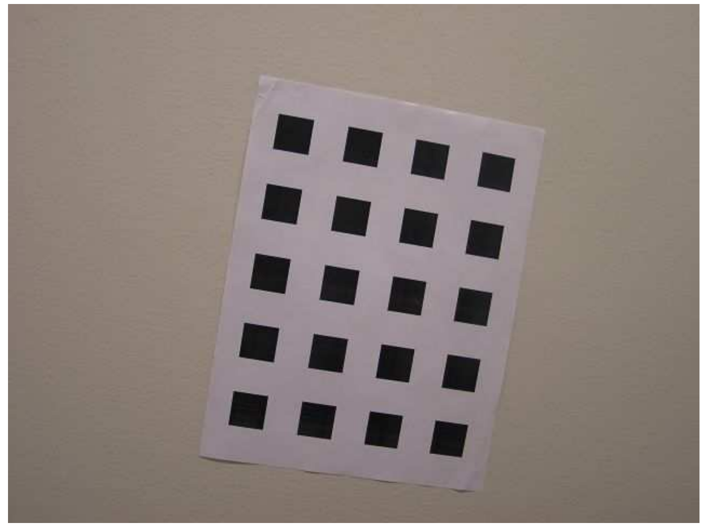
where $\vec{\hat{x}}_{ij}$ is the captured jth corner of the ith image.

Note that each rotation matrix R_i has 3 degrees of freedom. By Rodrigues formular [2], the rotation matrix R_i is converted to a 3-dim vector $\vec{v}$. Given the vector $\vec{v}$, the rotation matrix R_i is uniquely constructed. A detailed description can be found at http://en.wikipedia.org/wiki/Rotation_representation_(mathematics)

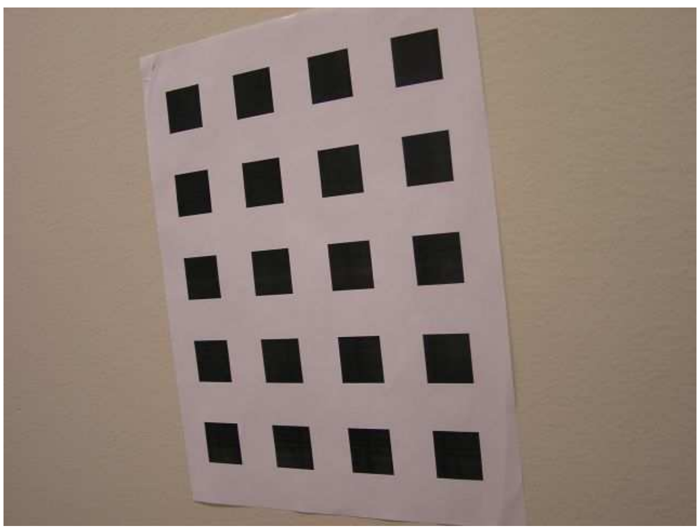
The LM algorithm used in this homework is the code provided at website http://www.ics.forth.gr/ lourakis/levmar/. For simplicity, the function *dlevmar_dif()* is used where the finite difference approximated Jacobian is used in stead of the analytical expression of Jacobian.



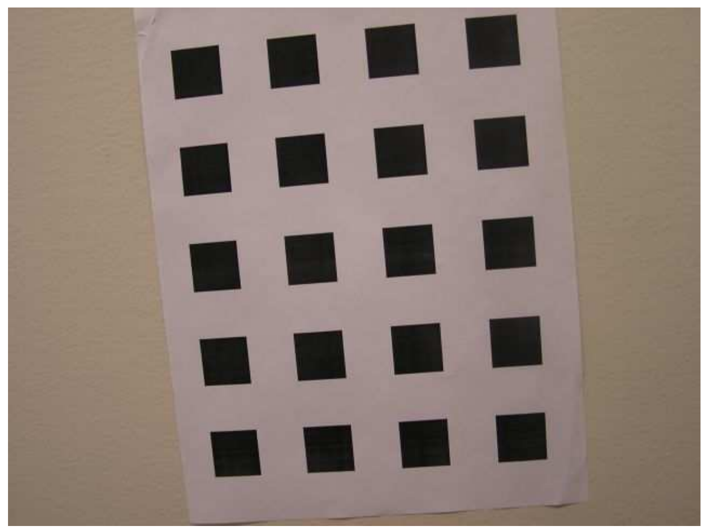
P1010001s.jpg



P1010003s.jpg



P1010053s.jpg



P1010062s.jpg

Figure 1. Original images

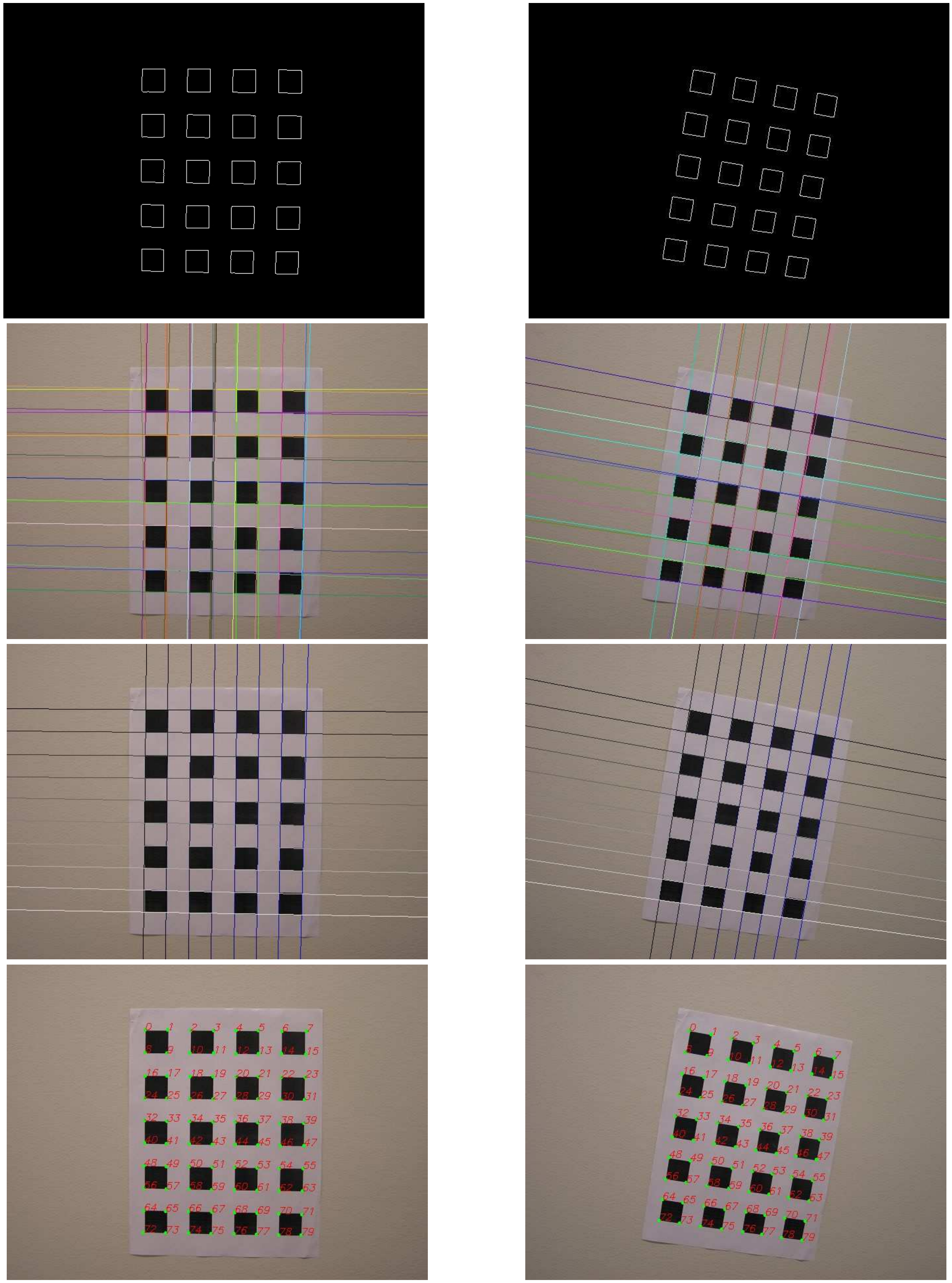


Figure 2. Results for edge detection, line detection, line grouping and ordering, and corner point identification (image P1010001s.jpg and P1010003s.jpg)

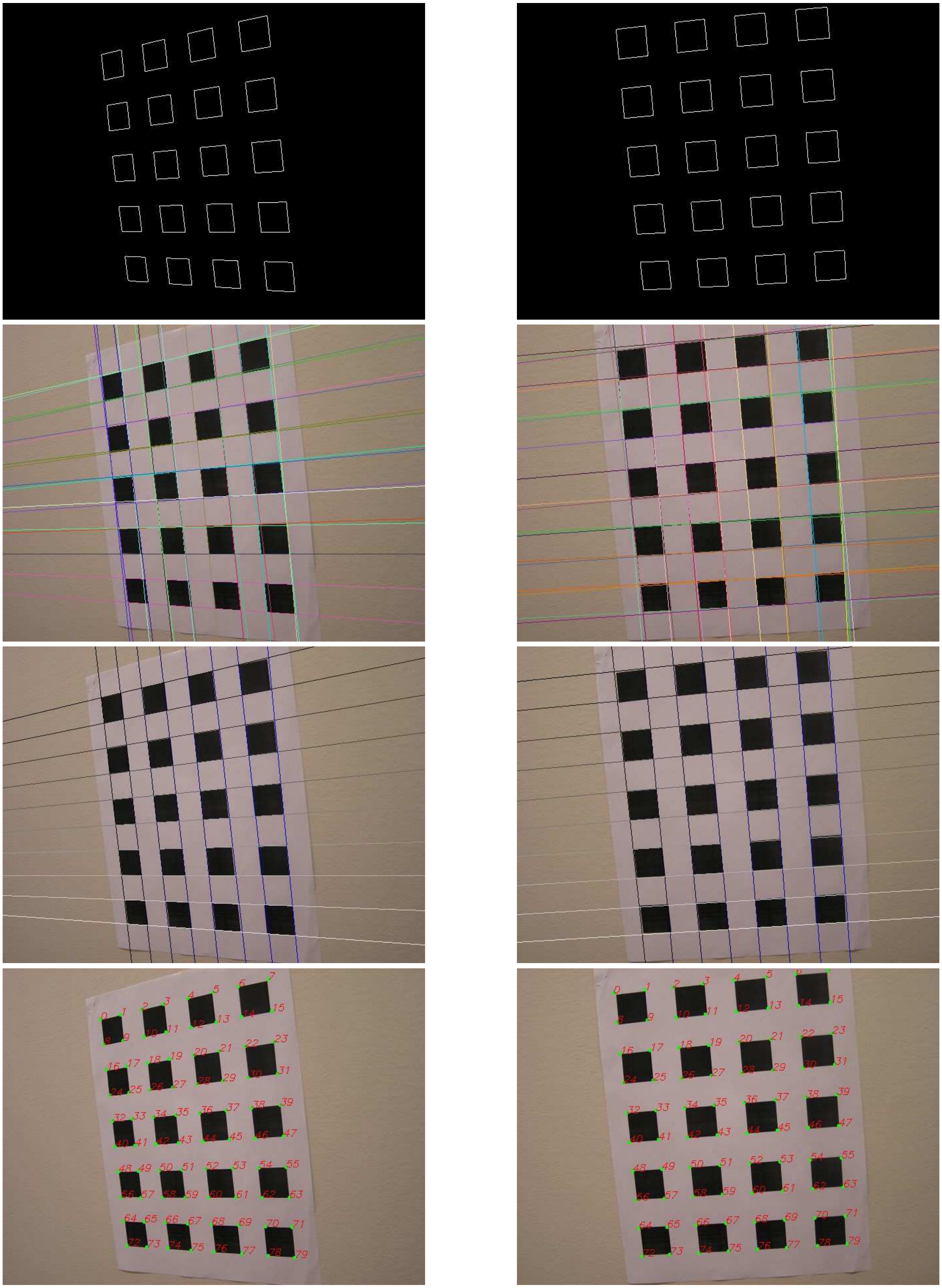


Figure 3. Results for edge detection, line detection, line grouping and ordering, and corner point identification (image P1010053s.jpg and P1010062s.jpg)

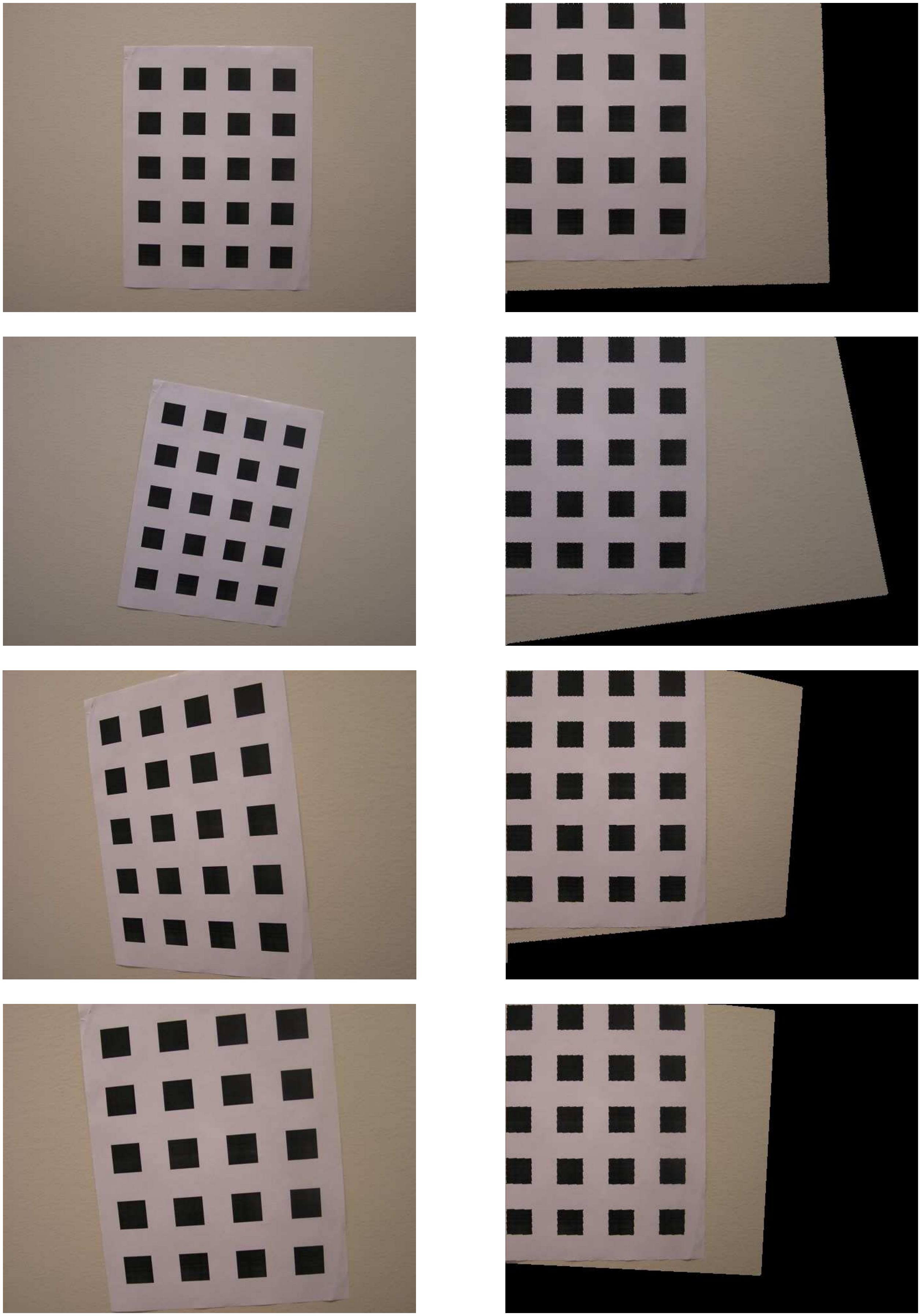


Figure 4. Original images and $invH*X' = (K[r1,r2,t])^{-1}*X'$

P1010001s.jpg

32 lines detected, grouped into 18

In RANSAC algorithm, number of inlier: 80

H:

0.854937 0.005353 210.110016

0.002704 0.876823 99.520031

-0.000038 0.000050 1.000000

P1010003s.jpg

26 lines detected, grouped into 18

In RANSAC algorithm, number of inlier: 80

H:

0.852833 -0.118102 249.682391

0.169351 0.826391 100.224727

0.000140 0.000060 1.000000

P1010053s.jpg

32 lines detected, grouped into 18

In RANSAC algorithm, number of inlier: 71

H:

0.641210 0.125579 147.541570

-0.227235 0.984017 76.878501

-0.000667 0.000067 1.000000

P1010062s.jpg

37 lines detected, grouped into 18

In RANSAC algorithm, number of inlier: 80

H:

1.090966 0.140303 146.969838

-0.122544 1.162723 37.299020

-0.000142 0.000122 1.000000

Camera Parameters (initial values):

intrinsic K:

725.112575 4.334241 305.365816

0 723.326805 247.560852

0 0 1

radial distortion para k1, k2:

0.016525 -0.282369

extrinsic [r1 r2 r3 t]:

image P1010001s.jpg:

0.999363 -0.015005 0.032387 -108.843538

0.016328 0.999026 -0.041001 -171.169713

-0.031740 0.041503 0.998634 836.334470

image P1010003s.jpg:

0.978897 -0.170780 -0.112231 -66.286548

0.163360 0.983910 -0.072344 -178.657158

0.122780 0.052483 0.991045 877.093189

image P1010053s.jpg

: 0.866134 0.101686 0.489359 -160.656447

-0.063659 0.993555 -0.093782 -175.310300

-0.495741 0.050076 0.867025 742.939378

image P1010062s.jpg

: 0.992975 0.083970 0.083366 -137.504240

-0.076903 0.993439 -0.084650 -184.446970

-0.089928 0.077644 0.992917 634.520476

LM algorithm:

distortion: before: 228.912255 after: 195.936303

iiterations: 102

Camera Parameters (after LM refinement)

intrinsic K:

726.959281 2.473622 311.722529

0 725.854245 241.317560

0 0 1

radial distortion para k1, k2:

-0.102392 0.184138

extrinsic [r1 r2 r3 t]:

image P1010001s.jpg:

0.999334 -0.014094 0.033661 -116.763909

0.015359 0.999174 -0.037630 -164.081443

-0.033103 0.038122 0.998725 835.760785

image P1010003s.jpg:

0.979669 -0.169604 -0.107162 -74.491437

0.163656 0.984560 -0.062120 -171.077477

0.116043 0.043320 0.992299 879.811613

image P1010053s.jpg:

0.869536 0.104232 0.482746 -167.569492

-0.067372 0.993372 -0.093132 -168.896434

-0.489253 0.048459 0.870794 740.558497

image P1010062s.jpg:

0.992465 0.085693 0.087582 -143.385429

-0.077847 0.992950 -0.089385 -178.652493

-0.094624 0.081894 0.992139 631.567595

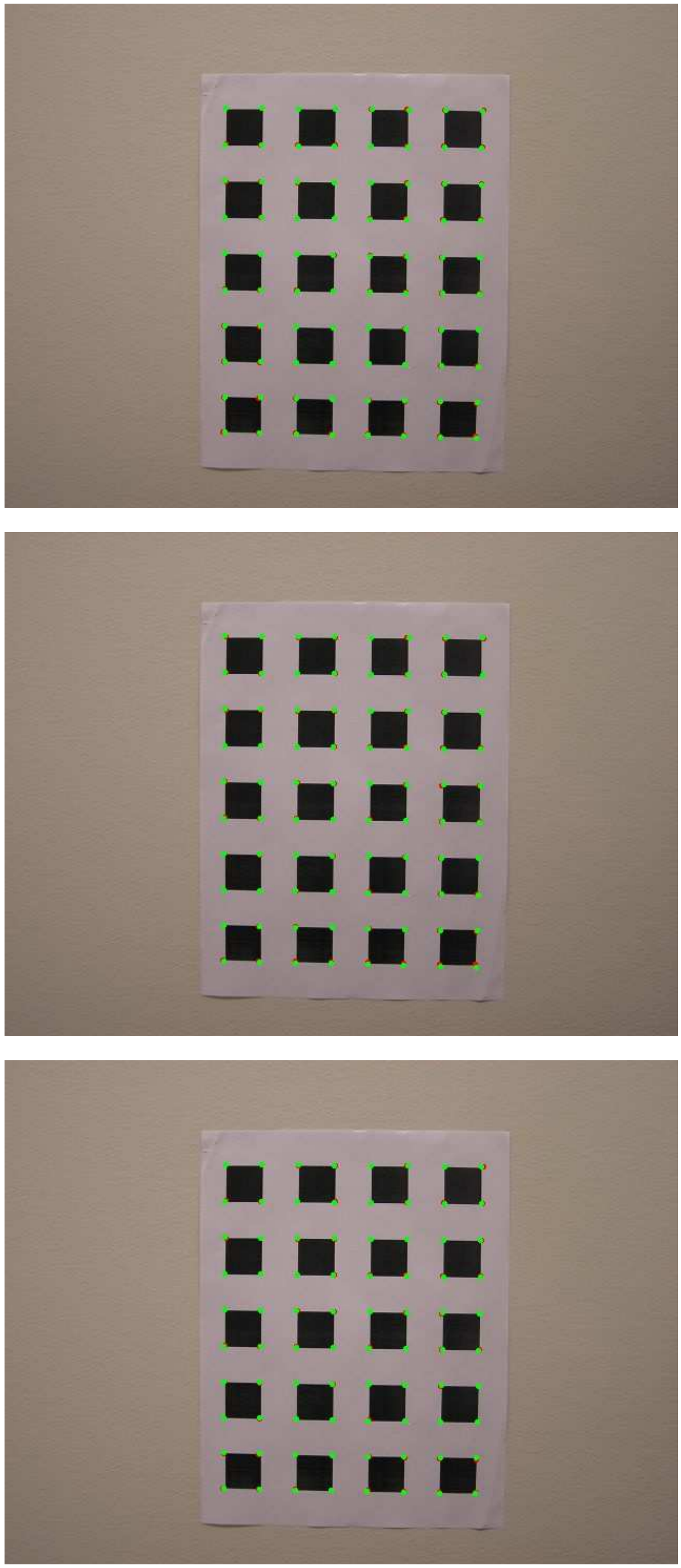


Figure 5. **Red pts:** detected corners in the image plane; **Green pts:** Projection of pattern corner points into image plane (1st row: with **initial** values of **K,R,t,k1,k2**; 2nd row: with **refined K,R,t**; 3rd row: with **refined K,R,t,k1,k2**)

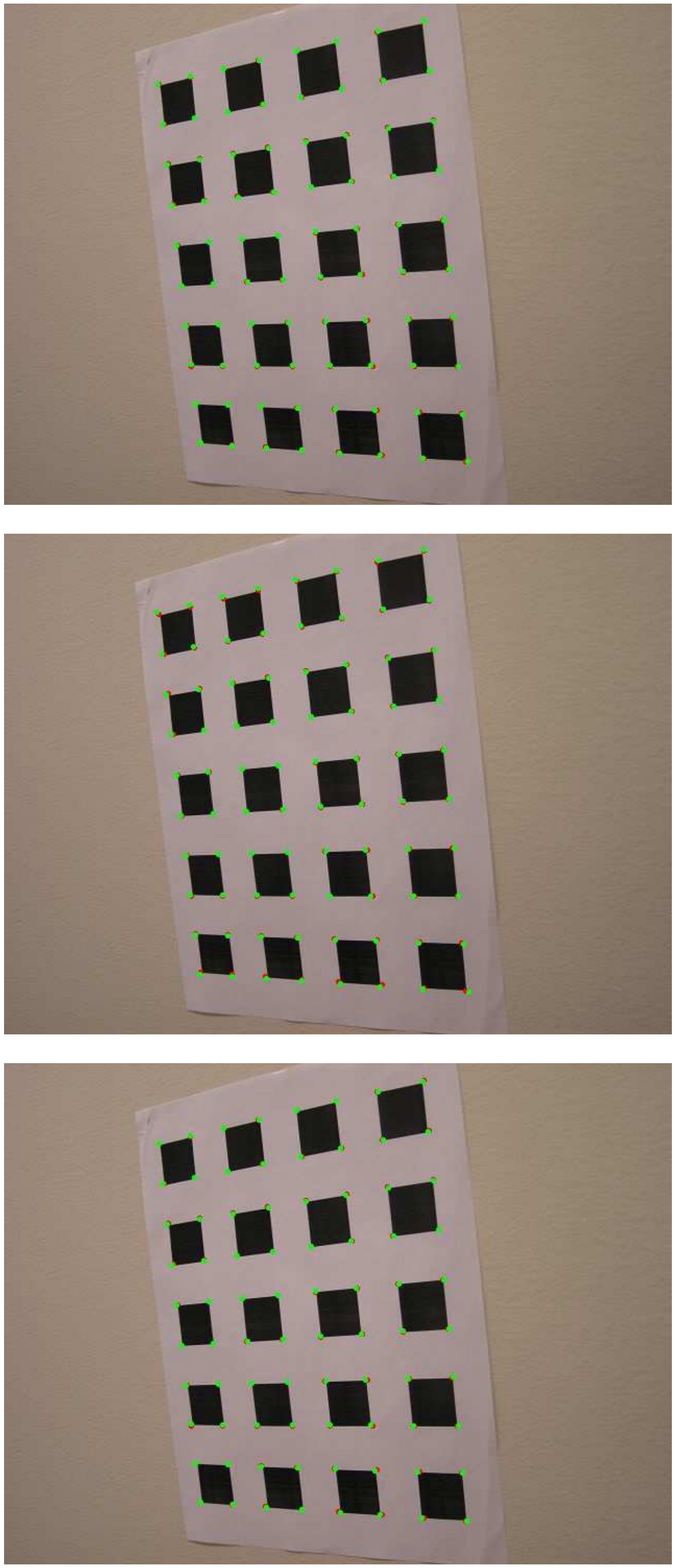


Figure 6. **Red pts:** detected corners in the image plane; **Green pts:** Projection of pattern corner points into image plane (1st row: with **initial** values of **K,R,t,k1,k2**; 2nd row: with **refined K,R,t**; 3rd row: with **refined K,R,t,k1,k2**)

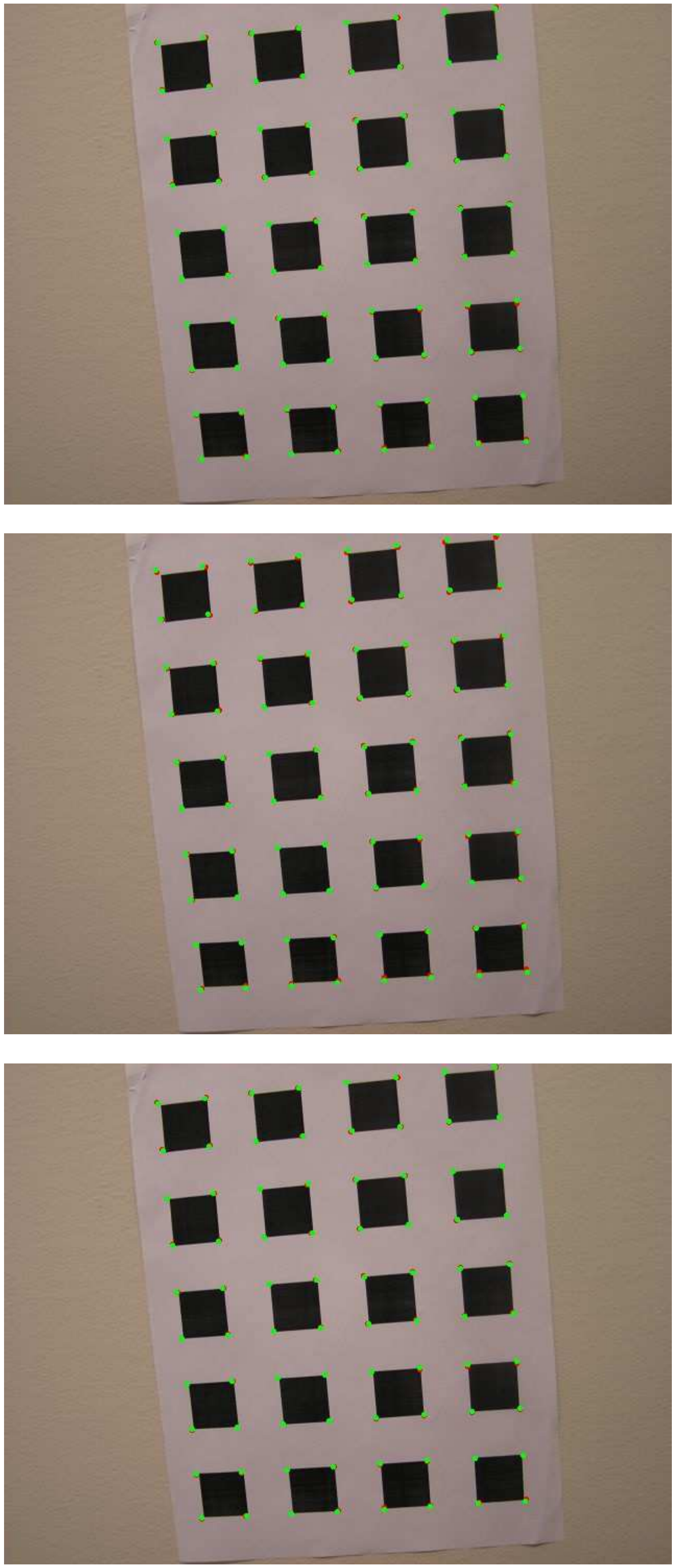


Figure 7. **Red pts:** detected corners in the image plane; **Green pts:** Projection of pattern corner points into image plane (1st row: with **initial** values of **K,R,t,k1,k2**; 2nd row: with **refined K,R,t**; 3rd row: with **refined K,R,t,k1,k2**)

```
//**************************************************************
// Camera Calibration:
//
// LM algorithm:
//   code from http://www.ics.forth.gr/~lourakis/levmar/
//   was used. A .lib file is created from the source code
//   provided. The main function used is dlevmar_dif()
//**************************************************************
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <cv.h>
#include <highgui.h>
// the following h files are from http://www.ics.forth.gr/~lourakis/levmar/
#include "misc.h"
#include "lm.h"

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

#define T_SMALLEST_EIG 10 // thres. for the smallest eigenvalue method
#define W_SIZE 7        // window size used in corner detection
#define EUC_DISTANCE 10 // thres. for Euclidean distance for uniquness_corner
#define B_SIZE 30       // size for excluding boundary pixel
#define W_SIZE_MATCH 30 // window size used in NCC
#define T_DIST 40       // thres. for distance in RANSAC algorithm

#define MAX_CORNERPOINT_NUM 80 // max number of detected corner pts
#define MAX_NUM_IMAGE 10
#define NUM_HOR 8    // checkboard pattern
#define NUM_VER 10   // checkboard pattern

typedef struct{
      int num;
      // intrinsic parameters
      double alphax, alphay;
      double x0;
      double y0;
      double skew;
      // extrinsic parameters
      double r1[MAX_NUM_IMAGE][3];
      double r2[MAX_NUM_IMAGE][3];
      double r3[MAX_NUM_IMAGE][3];
      double v[MAX_NUM_IMAGE][3];
      double t[MAX_NUM_IMAGE][3];
      // radial distortion
      double k1;
      double k2;
}CameraPara;

typedef struct{
      int num;
      double H[MAX_NUM_IMAGE][3][3];
      int numcorr[MAX_NUM_IMAGE];
}HomoMatrices;

/* global variables used by various homography estimation routines */
typedef struct {
  CvPoint2D64f *inlierp1, *inlierp2;
  int num_inlier;
  int *indicator;
}PtsPairs;
```

```
// X:p1;   Xp: p2    Xp = HX
// distortion = d(X,invH*Xp)+d(Xp,H*X)
// the following functions HomoDistFunc() and CalculateHomoDistFunc()
// are from the online example http://www.ics.forth.gr/~lourakis/homest/
void HomoDistFunc(CvPoint2D64f m1, CvPoint2D64f m2, double h[9], double tran_x[4])
{
      double
t1,t11,t13,t14,t15,t17,t18,t2,t20,t21,t23,t26,t28,t34,t4,t5,t53,t66,t68,t74,t8,t9;
    t1 = h[4];
    t2 = h[8];
    t4 = h[5];
    t5 = h[7];
    t8 = h[0];
    t9 = t8*t1;
    t11 = t8*t4;
    t13 = h[3];
    t14 = h[1];
    t15 = t13*t14;
    t17 = h[2];
    t18 = t13*t17;
    t20 = h[6];
    t21 = t20*t14;
    t23 = t20*t17;
    t26 = 1/(-t9*t2+t5*t11+t15*t2-t18*t5-t21*t4+t23*t1);
      t28 = m2.x;
      t34 = m2.y;
    t53 = 1/(-(t13*t5-t1*t20)*t26*t28-(-t5*t8+t21)*t26*t34+(-t9+t15)*t26);
    tran_x[0] = (-(t2*t1-t4*t5)*t26*t28+(t14*t2-t17*t5)*t26*t34-(t14*t4-t17*t1)*t26)*t53;
    tran_x[1] = ((t13*t2-t4*t20)*t26*t28-(t8*t2-t23)*t26*t34-(-t11+t18)*t26)*t53;
    t66 = m1.x;
    t68 = m1.y;
    t74 = 1/(t20*t66+t5*t68+t2);
    tran_x[2] = (t8*t66+t14*t68+t17)*t74;
    tran_x[3] = (t13*t66+t1*t68+t4)*t74;
}

static void CalculateHomoDistFunc(double *h, double *tran_x, int m, int n, void *adata)
{
      int i;
      PtsPairs * pair;
      pair = (PtsPairs *)adata;

      for(i=0; i<pair->num_inlier; i++)
            HomoDistFunc(pair->inlierp1[i], pair->inlierp2[i], h, tran_x+i*4);
}

// convert the 3-para [vx vy vz] of the Rodigrues
// fomular back into the 3x3 rotation matrix
void V2R(double vx, double vy, double vz, CvMat *R){
      int i;
      // theta = norm(v) because v=theta*w and norm(w)=1
      double theta  = sqrt(pow(vx, 2) + pow(vy, 2) + pow(vz, 2));
      double wx = vx/theta, wy = vy/theta, wz = vz/theta;
      double Wdata[9] = {0, -wz, wy,
                              wz, 0, -wx,
                         -wy, wx, 0};
      CvMat W = cvMat(3, 3, CV_64FC1, Wdata);
      CvMat* Mtmp = cvCreateMat(3, 3, CV_64FC1);

      //R = I + W*sin(theta) + W*W*(1-cos(theta);
      cvZero(R);
      for(i=0; i<3; i++)
            cvmSet(R,i,i,1.0);
```

```
	cvmScale(&W,Mtmp,sin(theta));
	cvAdd(R,Mtmp,R);
	cvMatMul(&W,&W,Mtmp);
	cvmScale(Mtmp,Mtmp,1-cos(theta));
	cvAdd(R,Mtmp,R);
	cvReleaseMat(&Mtmp);
}

void CameraCalibrationFunc(CvPoint2D64f X, double *para, double tran_X[2], int idx, int
isRadial = 1)
{
	double alphax = para[0], alphay = para[1];
	double skew = para[2];
	double x0 = para[3], y0 = para[4];
	double k1 = para[5], k2 = para[6];
	double vx = para[7+6*idx],   vy = para[7+6*idx+1], vz = para[7+6*idx+2];
	double tx = para[7+6*idx+3], ty = para[7+6*idx+4], 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,value;

	CvMat* K = cvCreateMat(3, 3, CV_64FC1);
	cvZero(K);
	cvmSet(K,2,2,1.0);
	cvmSet(K,0,0,alphax);
	cvmSet(K,0,1,skew);
	cvmSet(K,0,2,x0);
	cvmSet(K,1,1,alphay);
	cvmSet(K,1,2,y0);

	// V convert to R = [r1 r2 r3]
	V2R(vx,vy,vz,R);

	// Set R = [r1,r2,t]
	cvmSet(R,0,2,tx);
	cvmSet(R,1,2,ty);
	cvmSet(R,2,2,tz);

	cvMatMul(K,R,K);

	// Set X
	cvmSet(ptX,0,0,X.x);
	cvmSet(ptX,1,0,X.y);
	cvmSet(ptX,2,0,1.0);
	// x = RX   u = KRX
	cvMatMul(R,ptX,ptx);
	x = cvmGet(ptx,0,0)/cvmGet(ptx,2,0);
	y = cvmGet(ptx,1,0)/cvmGet(ptx,2,0);
	u = x0 + alphax*x + skew*y;
	v = y0 + alphay*y;
	if(isRadial){
		value = pow(x,2.0)+pow(y,2.0);
		tran_X[0] = u + (u-x0)*(k1*value+k2*pow(value,2.0));
		tran_X[1] = v + (v-y0)*(k1*value+k2*pow(value,2.0));
	}
	else{
		tran_X[0] = u;
		tran_X[1] = v;
	}
}
```

```
static void CalculateCameraCalibrationDistFunc(double *para, double *tran_x, int m, int
n, void *adata)
{
	int i;
	PtsPairs * pair;
	pair = (PtsPairs *)adata;
	for(i=0; i<pair->num_inlier; i++){
		CameraCalibrationFunc(pair->inlierp1[i], para, tran_x+i*2, pair-
>indicator[i]);
	}
}

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

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

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

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


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

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

```
            cvmSet(A,2*i+1,2,1);
            cvmSet(A,2*i+1,6,-p2[i].x*p1[i].x);
            cvmSet(A,2*i+1,7,-p2[i].x*p1[i].y);
            cvmSet(A,2*i+1,8,-p2[i].x);
      }

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

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

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

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

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

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

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

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

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

	cvInvert(H, invH);

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

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

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

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

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

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

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

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

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

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

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

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

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

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

	// RANSAC algorithm (reject outliers and obtain the best H)
	srand(134);
	MAX_num  = -1;
	while(N > sample_cnt){
		// for a randomly chosen non-colinear correspondances
		iscolinear = true;
		while(iscolinear == true){
			iscolinear = false;
			for(i=0; i<s; i++){
				// randomly select an index
				curr_idx[i] = rand()%num;
				for(j=0; j<i; j++){
					if(curr_idx[i] == curr_idx[j]){
						iscolinear = true;
						break;
					}
				}
				if(iscolinear == true) break;
				curr_m1[i].x = m1[curr_idx[i]].x;
				curr_m1[i].y = m1[curr_idx[i]].y;
				curr_m2[i].x = m2[curr_idx[i]].x;
				curr_m2[i].y = m2[curr_idx[i]].y;
```

```
            }
            // Check whether these points are colinear
            if(iscolinear == false)
                iscolinear = isColinear(s, curr_m1);
        }
        // Nomalized DLT
        Normalization(s, curr_m1, T1); //curr_m1 <- T1 * curr_m1
        Normalization(s, curr_m2, T2); //curr_m2 <- T2 * curr_m2

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

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

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

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

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

//*****************************************************************
//interpolation
// input: original image: img_ori,
//        mask: check (indicating pixel availability 1:yes; 0:no)
// output: interpolated image: img_ori
//         updated mask
//*****************************************************************
void InterpolateImage(IplImage** img_ori, CvMat *check){
    int i,j,k,count;
    int height = (*img_ori)->height, width = (*img_ori)->width;
    int channels = (*img_ori)->nChannels, step = (*img_ori)->widthStep;
    IplImage* img_interp = cvCloneImage(*img_ori);
    uchar *data_interp = (uchar *)(img_interp)->imageData;
    uchar *data_ori    = (uchar *)(*img_ori)->imageData;
    CvMat *check_avai  = cvCreateMat(height, width, CV_64FC1);

    cvCopy(check, check_avai);
    for (i=1; i<height-1; i++){      //y - ver
```

```
            for (j=1; j<width-1; j++){   //x - hor
                  if(cvmGet(check,i,j) == 0){
                        count = (cvmGet(check,i-
1,j)==1)+(cvmGet(check,i+1,j)==1)+(cvmGet(check,i,j-1)==1)+(cvmGet(check,i,j+1)==1);
                        if(count != 0 ){
                              for (k=0; k<channels; k++)
                                    data_interp[i*step+j*channels+k] =
(int)((data_ori[(i-
1)*step+j*channels+k]+data_ori[(i+1)*step+j*channels+k]+data_ori[i*step+(j-
1)*channels+k]+data_ori[i*step+(j+1)*channels+k])/count);
                              cvmSet(check_avai,i,j,1);
                        }
                  }
            }
      }
      cvCopy(check_avai, check);
      (*img_ori) = cvCloneImage(img_interp);

      // Release
      cvReleaseImage(&img_interp);
}

//********** transform images ***********
// input: img_x, H,
//         mask for interpolation: check
// output: img_xp, updated mask
//****************************************
void Trans_Image(IplImage** img_x, IplImage** img_xp, CvMat* H, CvMat* check){
      int i,j;
      int curpi,curpj;
      int height = (*img_x)->height;
      int width  = (*img_x)->width;
      CvMat *ptxp = cvCreateMat(3,1,CV_64FC1);
      CvMat *ptx  = cvCreateMat(3,1,CV_64FC1);

      cvZero(*img_xp);
      for (i=0; i<height; i++){      //y - ver
            for (j=0; j<width; j++){   //x - hor
              // set X_a
              cvmSet(ptx,0,0,(double)j);
              cvmSet(ptx,1,0,(double)i);
              cvmSet(ptx,2,0,1.0);
              // compute X
              cvMatMul(H, ptx, ptxp);
              curpi = CLIP2(0, height-1, (int)(cvmGet(ptxp,1,0)/cvmGet(ptxp,2,0)));
              curpj = CLIP2(0, width-1, (int)(cvmGet(ptxp,0,0)/cvmGet(ptxp,2,0)));

              cvSet2D(*img_xp,curpi,curpj,cvGet2D(*img_x,i,j));
              cvmSet(check,curpi,curpj,1);
            }
      }

      // Release
      cvReleaseMat(&ptx);
      cvReleaseMat(&ptxp);
}

//*********** Compute MSE of transformed image with ori image ***********
// input: img_x (X), homography matrix H; original X'
// output: write the transformed image under H (HX) into file "transfile"
//         write the error image = img_xp - HX into file "errfile"
// return: MSE of the error image
//************************************************************************
```

```
double ComputeMSE_Trans_Images(IplImage* img_x, IplImage* img_xp, CvMat* H, char*
transfile, char* errfile){
	int i,j,k;
	int height = img_x->height, width = img_x->width;
	int channels = img_x->nChannels, step = img_x->widthStep;
	double mse, msetmp, err;
	int pixnum;
	uchar *data_tr, *data_xp, *data_err;
	IplImage *img_tr, *img_err;
	CvMat *check = cvCreateMat(height, width, CV_64FC1);

	img_tr = cvCloneImage(img_x);
	//transform
	Trans_Image(&img_x, &img_tr, H, check);
	//interpolation
	InterpolateImage(&img_tr, check);
	data_tr = (uchar *)img_tr->imageData;

	img_err = cvCloneImage(img_tr);
	data_err  = (uchar *)(img_err)->imageData;
	data_xp = (uchar *)img_xp->imageData;
	mse = 0;
	pixnum = 0;
	// save error images (intensity I := I + 127 for display)
	for (i=1; i<height-1; i++){      //y - ver
		for (j=1; j<width-1; j++){   //x - hor
			msetmp = 0;
			for (k=0; k<channels; k++){ // for each channel
				if(cvmGet(check,i,j) == 1){ // available pixels
					err = data_xp[i*step+j*channels+k] -
data_tr[i*step+j*channels+k];
					data_err[i*step+j*channels+k] = (uchar)(127 + err);
					msetmp += pow(err, 2.0);
				}
				else
					data_err[i*step+j*channels+k] = 127;
			}
			if(cvmGet(check,i,j) == 1){
				mse += msetmp / channels;
				pixnum++;
			}
		}
	}
	mse /= pixnum;

	if(transfile != NULL)
		cvSaveImage(transfile,img_tr);
	if(errfile != NULL)
		cvSaveImage(errfile,img_err);

	cvReleaseMat(&check);
	cvReleaseImage(&img_tr);
	cvReleaseImage(&img_err);
	return mse;
}

void SortLines(CvSeq* lines, int num){
	int i,j;
	float tmprho, tmptheta;
	float *line1, *line2;

	// bubble sorting
	for(i = 0; i < num; i++){
```

```
		for(j = 0; j < num-i-1; j++){
			line1 = (float*)cvGetSeqElem(lines,j);
			line2 = (float*)cvGetSeqElem(lines,j+1);
			if(line1[0] < line2[0]){   //line[0] is rho
				// swap
				tmprho       = line1[0];
				tmptheta = line1[1];
				line1[0] = line2[0];
				line1[1] = line2[1];
				line2[0] = tmprho;
				line2[1] = tmptheta;
			}
		}
	}

}

void CalBorderPoint(float rho, float theta, int width, int height, CvPoint *pt1, CvPoint
*pt2){

	float a,b,c;

	a = cos(theta), b = sin(theta), c = tan(theta);
	if( fabs(b) < 0.001 ){
		pt1->x = pt2->x = cvRound(rho);
		pt1->y = 0;
		pt2->y = height;
	}else if( fabs(a) < 0.001 ){
		pt1->y = pt2->y = cvRound(rho);
		pt1->x = 0;
		pt2->x = width;
	}else{
		pt1->x = 0;
		pt1->y = cvRound(rho/b);
		if(pt1->y < 0){
			pt1->x = cvRound(rho / a);
			pt1->y = 0;
		}
		if(pt1->y >height){
			pt1->x = cvRound((pt1->y - height)*c);
			pt1->y = height;
		}

		pt2->x = width;
		pt2->y = cvRound(rho/b - width/c);

		if(pt2->y < 0){
			pt2->x = cvRound(rho/a);
			pt2->y = 0;
		}
		if(pt2->y > height){
			pt2->x = cvRound(-1.0 * ((height - rho/b) * c));
			pt2->y = height;
		}
	}
}

void PlotLines(IplImage *img, CvSeq *lines, int isorder = 0){
	int i;
	CvPoint pt1,pt2;
	float *line;
	int width = img->width, height = img->height;
```

```
	for( i = 0; i < lines->total; i++ )
	{
		line = (float*)cvGetSeqElem(lines,i);
		CalBorderPoint(line[0], line[1], width, height, &pt1, &pt2);
		if(isorder == 1)
			cvLine(img, pt1, pt2, CV_RGB(0,0,255/lines->total*i), 1 , 8, 0 );
		else if(isorder == 2)
			cvLine(img, pt1, pt2, CV_RGB(255/lines->total*i,255/lines->total*i,255/lines->total*i), 1 , 8, 0 );
		else
			cvLine(img, pt1, pt2, CV_RGB(rand()%255,rand()%255,rand()%255), 1 , 8, 0 );

	}
}

// Grouping Lines for those similar rho & theta
void GroupLines(CvSeq* lines, int width, int height, CvSeq* lineSet1, CvSeq* lineSet2, int* num1, int* num2){
	int i,j;
	int numlines = lines->total, numnewlines;
	float sumrho, sumtheta;
	float *linei, *linej;
	int count;
	CvMat *mask	= cvCreateMat(1,numlines,CV_32FC1);
	CvMat *newrho	= cvCreateMat(1,numlines, CV_32FC1);
	CvMat *newtheta = cvCreateMat(1,numlines, CV_32FC1);
	double theta1, theta2;
	double theta, rho;
	CvPoint2D32f pt;
	CvPoint pt1, pt2;

	/*for(i=0; i<numlines; i++){
		linei = (float*)cvGetSeqElem(lines,i);
		printf("%f %f n", linei[0], linei[1]);
		CalBorderPoint(linei[0], linei[1], width, height, &pt1, &pt2);
		theta = atan(double(pt1.x-pt2.x)/(pt2.y-pt1.y));
		rho = (pt1.x+pt2.x)/2.0*cos(theta)+(pt1.y+pt2.y)/2.0*sin(theta);
		linei[0] = (float)rho;
		linei[1] = (float)theta;
		printf("%f %f\n", rho, theta);
	}*/

	cvZero(mask);
	numnewlines = 0;
	for(i = 0; i < numlines; i++){
		if(cvmGet(mask,0,i) == 1)
			continue;
		linei = (float*)cvGetSeqElem(lines,i);
		sumrho	    = linei[0];
		sumtheta = linei[1];
		cvmSet(mask,0,i,1.0);

		count = 1;
		for(j = i+1; j < numlines; j++){
			linej = (float*)cvGetSeqElem(lines,j);
			if(pow(linei[0]-linej[0], (float)2.0) < 100 && pow(linei[1]-linej[1], (float)2.0) < 1e-2){
				sumrho	    += linej[0];
				sumtheta += linej[1];
				count++;
				cvmSet(mask,0,j,1.0);
			}
```

```
            }
            cvmSet(newrho,0,numnewlines,(double)sumrho/count);
            cvmSet(newtheta,0,numnewlines,(double)sumtheta/count);
            numnewlines++;
      }
      printf("%d %d\n", numlines, numnewlines);

      theta1 = cvmGet(newtheta, 0, 0);
      i = 1;
      while(pow(abs(theta1)-abs(cvmGet(newtheta, 0, i)), 2.0) < 5*1e-1){
            i++;
      }
      theta2 = cvmGet(newtheta, 0, i);

      // Set lineSet1, lineSet2
      *num1 = 0;
      *num2 = 0;
      for(i = 0; i<numnewlines; i++){
            if(pow(theta1-cvmGet(newtheta, 0, i), 2.0) < 5*1e-1 || pow(theta1-
(CV_PI+cvmGet(newtheta, 0, i)), 2.0) < 5*1e-1){
                  pt = cvPoint2D32f(cvmGet(newrho,0,i), cvmGet(newtheta,0,i));
                  cvSeqPush(lineSet1, &pt);
                  (*num1)++;
            }
      }
      lineSet1->total = *num1;

      for(i = 0; i<numnewlines; i++){
            if(pow(theta2-cvmGet(newtheta, 0, i), 2.0) < 5*1e-1 || pow(theta2-
(CV_PI+cvmGet(newtheta, 0, i)), 2.0) < 5*1e-1){
                  pt = cvPoint2D32f(cvmGet(newrho,0,i), cvmGet(newtheta,0,i));
                  cvSeqPush(lineSet2, &pt);
                  (*num2)++;
            }
      }
      lineSet2->total = *num2;

      // sort lines by value of rho
      SortLines(lineSet1, *num1);
      SortLines(lineSet2, *num2);

      cvReleaseMat(&mask);
      cvReleaseMat(&newrho);
      cvReleaseMat(&newtheta);
}

// lineSet1 should have 8 lines; lineSet2 should have 10 lines
void verifyLineOrder(int width, int height, CvSeq* lineSet1, CvSeq* lineSet2, int num1,
int num2){
      int i;
      float *line, *line0, *line1, *tline;
      CvSeq* tmpline;

      if(num1 != 8 || num2 != 10){
            printf("line number error\n");
            exit(0);
      }
      // for lineSet1
      // line[0]: rho; line[1]: theta
      line0 = (float*)cvGetSeqElem(lineSet1,0);
      line1 = (float*)cvGetSeqElem(lineSet1,1);
      if(line0[0]/cos(line0[1]) > line1[0]/cos(line1[1])){
            tmpline = cvCloneSeq(lineSet1);
```

```
            for(i=0; i<num1; i++){
                  line = (float*)cvGetSeqElem(lineSet1,i);
                  tline = (float*)cvGetSeqElem(tmpline,num1-1-i);
                  line[0] = tline[0];
                  line[1] = tline[1];
            }
      }
      // for lineSet2
      line0 = (float*)cvGetSeqElem(lineSet2,0);
      line1 = (float*)cvGetSeqElem(lineSet2,1);
      if(line0[0]/sin(line0[1]) > line1[0]/sin(line1[1])){
            tmpline = cvCloneSeq(lineSet2);
            for(i=0; i<num2; i++){
                  line = (float*)cvGetSeqElem(lineSet2,i);
                  tline = (float*)cvGetSeqElem(tmpline,num2-1-i);
                  line[0] = tline[0];
                  line[1] = tline[1];
            }
      }

}
//
int FindFeaturePoint(int imgidx, IplImage *img, CvPoint2D64f *pt){
      int i,j,numcorner;
      int width = img->width, height = img->height;
      int num1, num2, tmpnum;
      IplImage *gimg = 0;
      IplImage *edgeimg = 0, *tmpimg = 0;
      CvMemStorage *storage = cvCreateMemStorage(0);
      CvSeq *lines = 0, *lineSet1 = 0, *lineSet2 = 0, *tmpline = 0;
      float *line1, *line2;
      CvMat *p1   = cvCreateMat(3,1,CV_64FC1);
      CvMat *p2    = cvCreateMat(3,1,CV_64FC1);
      CvMat *ln1   = cvCreateMat(3,1,CV_64FC1);
      CvMat *ln2   = cvCreateMat(3,1,CV_64FC1);
      CvMat *point = cvCreateMat(3,1,CV_64FC1);
      CvPoint pt1, pt2;
      char label[10];
      char name[20];
      CvFont font;
      double hScale = .5;
      double vScale = .5;
      int lineWidth = 1;
      cvInitFont(&font, CV_FONT_HERSHEY_SIMPLEX|CV_FONT_ITALIC, hScale, vScale, 0,
lineWidth);

      // create gray scale image
      gimg = cvCreateImage(cvSize(width,height), IPL_DEPTH_8U, 1);
      cvCvtColor(img, gimg, CV_BGR2GRAY);
      //cvSmooth(gimg, gimg, CV_GAUSSIAN, 3, 3, 0);

      // edge detection
      edgeimg = cvCreateImage(cvSize(width, height), IPL_DEPTH_8U, 1);
      cvCanny(gimg, edgeimg, 50, 400, 3);
      sprintf(name, "%d", imgidx);
      strcat(name, "edge.jpg");
      cvSaveImage(name, edgeimg);

      // line fitting
      lines = cvHoughLines2(edgeimg,storage,CV_HOUGH_STANDARD,1,CV_PI/180,50,0,0 );
      lineSet1 = cvCloneSeq(lines, storage);
      cvClearSeq(lineSet1);
      lineSet2 = cvCloneSeq(lines, storage);
```

```
	cvClearSeq(lineSet2);

	// save results for Hough transform
	tmpimg = cvCloneImage(img);
	PlotLines(tmpimg,lines,0);
	sprintf(name, "%d", imgidx);
	strcat(name, "allines.jpg");
	cvSaveImage(name, tmpimg);

	GroupLines(lines, width, height, lineSet1, lineSet2, &num1, &num2);
	// feature points are labeled in raster scan order (assume # of hor. pts < # of
ver. pts)
	// i.e., num1 should < num2
	if(num1 > num2){
		// swap
		tmpline = lineSet1;
		lineSet1 = lineSet2;
		lineSet2 = tmpline;
		tmpnum = num1;
		num1 = num2;
		num2 = tmpnum;
	}
	verifyLineOrder(width, height, lineSet1, lineSet2, num1, num2);
	tmpimg = cvCloneImage(img);
	PlotLines(tmpimg,lineSet1,1);
	PlotLines(tmpimg,lineSet2,2);
	sprintf(name, "%d", imgidx);
	strcat(name, "lines.jpg");
	cvSaveImage(name, tmpimg);

	// find corners (intersection of lines)
	tmpimg = cvCloneImage(img);
	numcorner = 0;
	for(j=0; j<num2; j++){
		line2 = (float*)cvGetSeqElem(lineSet2,j);
		// get two pts in line 2
		CalBorderPoint(line2[0], line2[1], width, height, &pt1, &pt2);
		cvmSet(p1, 0, 0, pt1.x);
		cvmSet(p1, 1, 0, pt1.y);
		cvmSet(p1, 2, 0, 1.0);
		cvmSet(p2, 0, 0, pt2.x);
		cvmSet(p2, 1, 0, pt2.y);
		cvmSet(p2, 2, 0, 1.0);
		// get homo. rep. of line 2
		cvCrossProduct(p1, p2, ln2);
		for(i=0; i<num1; i++){
			line1 = (float*)cvGetSeqElem(lineSet1,i);
			// get two pts in line 1
			CalBorderPoint(line1[0], line1[1], width, height, &pt1, &pt2);
			cvmSet(p1, 0, 0, pt1.x);
			cvmSet(p1, 1, 0, pt1.y);
			cvmSet(p1, 2, 0, 1.0);
			cvmSet(p2, 0, 0, pt2.x);
			cvmSet(p2, 1, 0, pt2.y);
			cvmSet(p2, 2, 0, 1.0);
			// get homo. rep. of line 1
			cvCrossProduct(p1, p2, ln1);

			// get the intersection point
			cvCrossProduct(ln2, ln1, point);
			pt[numcorner].x   = cvmGet(point,0,0)/cvmGet(point,2,0);
			pt[numcorner].y = cvmGet(point,1,0)/cvmGet(point,2,0);
```

```
                  cvCircle(tmpimg, cvPoint((int)pt[numcorner].x, (int)pt[numcorner].y), 
1, cvScalar(0, 255, 0), 2, 8, 0);
                  sprintf(label, "%d", numcorner);
                  cvPutText(tmpimg, label, cvPoint((int)pt[numcorner].x, 
(int)pt[numcorner].y), &font, cvScalar(0, 0, 255));
                  numcorner++;
            }
      }
      sprintf(name, "%d", imgidx);
      strcat(name, "corners.jpg");
      cvSaveImage(name, tmpimg);

      // Release
      cvReleaseImage(&gimg);
      cvReleaseImage(&edgeimg);
      cvReleaseImage(&tmpimg);
      cvReleaseMat(&ln1);
      cvReleaseMat(&ln2);
      cvReleaseMat(&p1);
      cvReleaseMat(&p2);
      cvReleaseMat(&point);
      return numcorner;
}

//***********************************************************
// Compute the Homography Matrix
// 1. Canny edge detection & Hough transform
// 2. RANSAC
// 3. LM optimization
// input:   X: img_1     X': img_2   (X' = HX)
//          RANSACresultfile: file name for RANSAC result
// output:  H
//***********************************************************
void ComputeHomography(int imgidx, IplImage *img, CvPoint2D64f *patternp, CvPoint2D64f 
*cornerpts, CvMat *H, PtsPairs *ptspairs)
{
      int height, width, step, channels;
      int num, num_matched;
      int i,j,count;

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

      // detect corner
      // corner points are stored in CvPoint cornerp1 and cornerp2
      // note: these corner points are already stored in raster scan order
      // therefore, no feature matching method is applied
      num = FindFeaturePoint(imgidx, img, cornerpts);
      if(num != MAX_CORNERPOINT_NUM)
            printf("error in corner detection\n");
      num_matched = num;

      // RANSAC algorithm
      CvMat *inlier_mask = cvCreateMat(num_matched,1,CV_64FC1);
      RANSAC_homography(num_matched, patternp, cornerpts, H, inlier_mask);
    // count number of inliers
      ptspairs->num_inlier = 0;
      count = 0;
      for(i=0; i<num_matched; i++){
            if(cvmGet(inlier_mask,i,0) == 1){
                  ptspairs->inlierp1[count].x   = patternp[i].x;
```

```
            ptspairs->inlierp1[count].y   = patternp[i].y;
            ptspairs->inlierp2[count].x   = cornerpts[i].x;
            ptspairs->inlierp2[count++].y = cornerpts[i].y;
            ptspairs->num_inlier++;
        }
    }
    printf("number of inlier: %d\n",ptspairs->num_inlier);
    // Estimate H based on all the inlier points
    ComputeH(ptspairs->num_inlier, ptspairs->inlierp1, ptspairs->inlierp2, H);

    //IplImage *tmpimg;
    //tmpimg = cvCloneImage(img);
    //for(i=0; i<num_matched; i++)
    //    //if(cvmGet(inlier_mask,i,0) == 1)
    //          cvCircle(tmpimg, cvPoint((int)cornerpts[i].x,(int)cornerpts[i].y), 1,
CV_RGB(255,255,255), 2, 8, 0);
    //cvSaveImage("pt.jpg", tmpimg);
    //tmpimg = cvCloneImage(img);
    //CvMat *check = cvCreateMat(height, width, CV_64FC1);
    //CvMat *invH = cvCreateMat(3, 3, CV_64FC1);
    //cvInvert(H,invH);
    //Trans_Image(&img, &tmpimg, invH, check);
    //cvSaveImage("trans.jpg", tmpimg);
    //cvReleaseImage(&tmpimg);
    //cvReleaseMat(&check);
    //cvReleaseMat(&invH);

    // LM algorithm
    int 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; // relevant only if the finite difference Jacobian version
is used

    void (*err)(double *p, double *hx, int m, int n, void* adata);
    int LM_m = 9, LM_n = 4*ptspairs->num_inlier;
    double *x = (double *)malloc(LM_n*sizeof(double));
    double *p = (double*)malloc(9*sizeof(double));
    for(i=0; i<3; i++){
        j = 3*i;
        p[j]   = cvmGet(H,i,0);
        p[j+1] = cvmGet(H,i,1);
        p[j+2] = cvmGet(H,i,2);
    }
    for(i=0; i<ptspairs->num_inlier; i++){
        j = i<<2;
        x[j]   = ptspairs->inlierp1[i].x;
        x[j+1] = ptspairs->inlierp1[i].y;
        x[j+2] = ptspairs->inlierp2[i].x;
        x[j+3] = ptspairs->inlierp2[i].y;
    }
    err = CalculateHomoDistFunc;
    ret = dlevmar_dif(err, p, x, LM_m, LM_n, 1000, opts, info, NULL, NULL, (void
*)ptspairs); // no Jacobian
    //printf("distortion: %f %f\n", info[0], info[1]);
    //printf("LM algorithm iterations: %f \n", info[5]);

    // set H matrix
    for(i=0; i<3; i++){
        j = 3*i;
        cvmSet(H,i,0, p[j]);
        cvmSet(H,i,1, p[j+1]);
        cvmSet(H,i,2, p[j+2]);
```

```
	}

	// release
	cvReleaseMat(&inlier_mask);
}

// i: row number; j: col number
void SetElementVij(int i, int j, CvMat *H, CvMat *vij) {
	double hi1 = cvmGet(H, 0, i-1);
	double hi2 = cvmGet(H, 1, i-1);
	double hi3 = cvmGet(H, 2, i-1);
	double hj1 = cvmGet(H, 0, j-1);
	double hj2 = cvmGet(H, 1, j-1);
	double hj3 = cvmGet(H, 2, j-1);
	double vijdata[6] = {hi1*hj1, hi1*hj2+hi2*hj1, hi2*hj2,
	               hi3*hj1+hi1*hj3, hi3*hj2+hi2*hj3, hi3*hj3};

	for(int i=0; i<6; i++){
		cvmSet(vij,i,0,vijdata[i]);
		//printf("%f ", cvmGet(vij,i,0));
	}
	//printf("\n");
}

// vect is a col vector
void NormalizeVector(CvMat *v, double norm){
	int i;
	for(i=0; i<v->rows; i++)
		cvmSet(v, i, 0, cvmGet(v, i, 0)/norm);
}

void ColVector2CvMat(double *v, CvMat *vMat, int dim){
	int i;
	for(i=0; i<dim; i++)
		cvmSet(vMat, i, 0, v[i]);
}

void CvMat2ColVector(double *v, CvMat *vMat, int dim){
	int i;
	for(i=0; i<dim; i++)
		v[i] = cvmGet(vMat, i, 0);
}

void EstIntrinsicPara(CameraPara *camerap, HomoMatrices hms, CvMat *K){
	int i,j,k;
	int numimgs = hms.num;
	double lambda;
	CvMat *H = cvCreateMat(3,3,CV_64FC1);    // homography matrix
	CvMat *V = cvCreateMat(2*numimgs, 6, CV_64FC1);
	CvMat *v12 = cvCreateMat(6,1,CV_64FC1);
	CvMat *v11 = cvCreateMat(6,1,CV_64FC1);
	CvMat *v22 = cvCreateMat(6,1,CV_64FC1);
	// V = Up^T Dp Vp
	CvMat *Up = cvCreateMat(2*numimgs,2*numimgs,CV_64FC1);
	CvMat *Dp = cvCreateMat(2*numimgs,6,CV_64FC1);
	CvMat *Vp = cvCreateMat(6,6,CV_64FC1);
	double b[6]; // b = [B11 B12 B22 B13 B23 B33]^T

	//******************** intrinsic paras ********************
	// fill matrix V
	for(i=0; i<numimgs; i++){
		// set H
		for(j=0; j<3; j++)
```

```
                for(k=0; k<3; k++)
                      cvmSet(H,j,k,hms.H[i][j][k]);

          // two equations per image
          SetElementVij(1, 2, H, v12);
          SetElementVij(1, 1, H, v11);
          SetElementVij(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
    // V = Up^T Dp Vp
    cvSVD(V, Dp, Up, Vp, CV_SVD_U_T|CV_SVD_V_T);

    // take the last column of Vp^T, i.e., last row of Vp
    for(i=0; i<6; i++)
          b[i] = cvmGet(Vp, 5, i);

    // for camera parameters
    // b = [B11 B12 B22 B13 B23 B33]^T
    camerap->y0 = (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) + camerap->y0*(b[1]*b[3] - b[0]*b[4]))/b[0];
    camerap->alphax = sqrt(lambda/b[0]);
    camerap->alphay = sqrt((lambda*b[0])/(b[0]*b[2] - pow(b[1], 2.0)));
    camerap->skew   = -b[1]*pow(camerap->alphax, 2.0)*camerap->alphay/lambda;
    camerap->x0     = (camerap->skew*camerap->y0)/camerap->alphay - (b[3]*pow(camerap-
>alphax, 2.0))/lambda;

    // set intrinsic matrix K
    cvZero(K);
    cvmSet(K,2,2,1.0);
    cvmSet(K,0,0,camerap->alphax);
    cvmSet(K,0,1,camerap->skew);
    cvmSet(K,0,2,camerap->x0);
    cvmSet(K,1,1,camerap->alphay);
    cvmSet(K,1,2,camerap->y0);
    /*for(i=0; i<3; i++){
          for(j=0; j<3; j++)
                printf("%f ", cvmGet(K,i,j));
          printf("\n");
    }
    printf("\n");*/

    // Release
    cvReleaseMat(&H);
    cvReleaseMat(&V);
    cvReleaseMat(&v11);
    cvReleaseMat(&v12);
    cvReleaseMat(&v22);
    cvReleaseMat(&Dp);
    cvReleaseMat(&Vp);
    cvReleaseMat(&Up);
}

void EstExtrinsicPara(CameraPara *camerap, HomoMatrices hms, CvMat *K){
    int i,j,k;
    int numimgs = hms.num;
```

```
double norm, trace, theta, value1;
CvMat *invK = cvCreateMat(3,3,CV_64FC1);
CvMat *h1 = cvCreateMat(3,1,CV_64FC1);
CvMat *h2 = cvCreateMat(3,1,CV_64FC1);
CvMat *h3 = cvCreateMat(3,1,CV_64FC1);
CvMat *r1 = cvCreateMat(3,1,CV_64FC1);
CvMat *r2 = cvCreateMat(3,1,CV_64FC1);
CvMat *r3 = cvCreateMat(3,1,CV_64FC1);
CvMat *t  = cvCreateMat(3,1,CV_64FC1);
CvMat *Q  = cvCreateMat(3,3,CV_64FC1);
CvMat *QU = cvCreateMat(3,3,CV_64FC1);
CvMat *transQU = cvCreateMat(3,3,CV_64FC1);
CvMat *QD = cvCreateMat(3,3,CV_64FC1);
CvMat *QV = cvCreateMat(3,3,CV_64FC1);
CvMat *R  = cvCreateMat(3,3,CV_64FC1);
CvMat *H  = cvCreateMat(3,3,CV_64FC1);


// get extrinsic params
cvInvert(K,invK);
for(i=0; i<numimgs; i++){
      // set h1, h2, h3
      for(j=0; j<3; j++){
            cvmSet(h1,j,0,hms.H[i][j][0]/hms.H[i][2][2]);
            cvmSet(h2,j,0,hms.H[i][j][1]/hms.H[i][2][2]);
            cvmSet(h3,j,0,hms.H[i][j][2]/hms.H[i][2][2]);
      }
      // for r1
      cvMatMul(invK, h1, r1);
      norm = sqrt(cvDotProduct(r1, r1));
      NormalizeVector(r1, norm);
      // for r2
      cvMatMul(invK, h2, r2);
      NormalizeVector(r2, norm);
      // for r3
      cvCrossProduct(r1,r2,r3);
      // for t
      cvMatMul(invK, h3, t);
      NormalizeVector(t, norm);

      // refine rotation matrix
      // min||R - Q||_F, s.t R^T*R = I where Q = [r1, r2, r3]
      // Q = QU^T QD QV  => R = QU^T * QV
      // set Q
      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, transQU);
      cvMatMul(transQU, QV, R);


      double rr = cvDet(R);
      // Store r1, r2, r3, t
      for(j=0; j<3; j++){
            camerap->r1[i][j] = cvmGet(R,j,0);
            camerap->r2[i][j] = cvmGet(R,j,1);
            camerap->r3[i][j] = cvmGet(R,j,2);
      }
      CvMat2ColVector(camerap->t[i], t, 3);
```

```
		// Rodrigues Formula (representation rotation matrix R by three paras only)
		// R -> theta and w=[wx,wy,wz]' -> v = theta*w
		trace = (cvTrace(R)).val[0]; // cvTrace returns a CvScalar
		theta = acos((trace - 1) / 2.0);
		value1 = theta/(2*sin(theta));
		camerap->v[i][0] = value1*(cvmGet(R, 2, 1)-cvmGet(R, 1, 2));
		camerap->v[i][1] = value1*(cvmGet(R, 0, 2)-cvmGet(R, 2, 0));
		camerap->v[i][2] = value1*(cvmGet(R, 1, 0)-cvmGet(R, 0, 1));


		// H = K[r1 r2 t]
		for(j=0; j<3; j++)
			for(k=0; k<3; k++)
				cvmSet(H,j,k,hms.H[i][j][k]/hms.H[i][2][2]);
		/*printf("H is \n");
		for(j=0; j<3; j++){
			for(k=0; k<3; k++)
				printf("%f ", cvmGet(H,j,k));
			printf("\n");
		}
		printf("\n");*/

		for(j=0; j<3; j++){
			cvmSet(R,j,2,camerap->t[i][j]);
		}
		/*printf("K is \n");
		for(j=0; j<3; j++){
			for(k=0; k<3; k++)
				printf("%f ", cvmGet(K,j,k));
			printf("\n");
		}
		printf("\n");*/
		cvMatMul(K,R,R);
		/*printf("KR is \n");
		for(j=0; j<3; j++){
			for(k=0; k<3; k++)
				printf("%f ", cvmGet(R,j,k)/cvmGet(R,2,2));
			printf("\n");
		}
		printf("\n");*/
	}
	// Release
	cvReleaseMat(&invK);
	cvReleaseMat(&r1);
	cvReleaseMat(&r2);
	cvReleaseMat(&r3);
	cvReleaseMat(&t);
	cvReleaseMat(&h1);
	cvReleaseMat(&h2);
	cvReleaseMat(&h3);
	cvReleaseMat(&Q);
	cvReleaseMat(&QD);
	cvReleaseMat(&QU);
	cvReleaseMat(&QV);
	cvReleaseMat(&R);
	cvReleaseMat(&H);
}

void EstRadialDistPara(CameraPara *camerap, HomoMatrices hms, CvMat *K, PtsPairs
ptspairs){
	int i,j,k;
	int numimgs = hms.num;
	int numcorr = ptspairs.num_inlier;
```

```
	int idx;
	double alphax = camerap->alphax;
	double alphay = camerap->alphay;
	double skew = camerap->skew;
	double x0 = camerap->x0;
	double y0 = camerap->y0;
	double u,v,x,y;
	double value1, value2, value3;
	CvMat *R = cvCreateMat(3,3,CV_64FC1);
	CvMat *ptX = cvCreateMat(3,1,CV_64FC1);
	CvMat *ptx = cvCreateMat(3,1,CV_64FC1);
	CvMat *D  = cvCreateMat(2*numcorr,2,CV_64FC1);
	CvMat *d  = cvCreateMat(2*numcorr,1,CV_64FC1);
	CvMat *solk = cvCreateMat(2,1,CV_64FC1);

	CvMat *DT  = cvCreateMat(2,2*numcorr,CV_64FC1);
	CvMat *tmp  = cvCreateMat(2,2,CV_64FC1);

	idx = 0;
	for(i=0; i<numimgs; i++){
		// set H
		for(j=0; j<3; j++)
			for(k=0; k<3; k++)
				cvmSet(R,j,k,hms.H[i][j][k]);

		// for each point correspondence
		for(j=0; j<hms.numcorr[i]; j++){
			cvmSet(ptX,0,0,ptspairs.inlierp1[idx].x);
			cvmSet(ptX,1,0,ptspairs.inlierp1[idx].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);

			//printf("(%f %f), (%f %f)\n", 
u,v,ptspairs.inlierp2[idx].x,ptspairs.inlierp2[idx].y);
			x = (u-x0)/alphax;
			y = (v-y0)/alphay;
			value1 = u - x0;
			value2 = v - y0;
			value3 = pow(x,2.0)+pow(y,2.0);

			// add two equations
			cvmSet(D,2*idx,  0,value1*value3);
			cvmSet(D,2*idx,  1,value1*pow(value3,2.0));
			cvmSet(D,2*idx+1,0,value2*value3);
			cvmSet(D,2*idx+1,1,value2*pow(value3,2.0));
			cvmSet(d,2*idx,  0, ptspairs.inlierp2[idx].x-u);
			cvmSet(d,2*idx+1,0, ptspairs.inlierp2[idx].y-v);
			idx++;
		}
	}
	if(idx != numcorr){
		printf("error in radial distortion estimation");
		exit(0);
	}

	// solve k
	// k = (D^T * D)^-1 *D^T * d
	cvSolve(D,d,solk,CV_SVD);
	camerap->k1 = cvmGet(solk,0,0);
	camerap->k2 = cvmGet(solk,1,0);
```

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

void Camerapara2Para(CameraPara *camerap, double *para, int numimg){
	int i;
	// intrinsic paras
	para[0] = camerap->alphax;
	para[1] = camerap->alphay;
	para[2] = camerap->skew;
	para[3] = camerap->x0;
	para[4] = camerap->y0;
	// radial distortion paras
	para[5] = camerap->k1;
	para[6] = camerap->k2;
	// extrinsic paras
	for(i=0; i<numimg; i++){
		para[7+6*i]   = camerap->v[i][0];
		para[7+6*i+1] = camerap->v[i][1];
		para[7+6*i+2] = camerap->v[i][2];
		para[7+6*i+3] = camerap->t[i][0];
		para[7+6*i+4] = camerap->t[i][1];
		para[7+6*i+5] = camerap->t[i][2];
	}
}
void Para2Camerapara(CameraPara *camerap, double *para, int numimg){
	int i,j;
	CvMat *R = cvCreateMat(3,3,CV_64FC1);
	// intrinsic paras
	camerap->alphax = para[0];
	camerap->alphay = para[1];
	camerap->skew = para[2];
	camerap->x0 = para[3];
	camerap->y0 = para[4];
	// radial distortion paras
	camerap->k1 = para[5];
	camerap->k2 = para[6];
	// extrinsic paras
	for(i=0; i<numimg; i++){
		camerap->v[i][0] = para[7+6*i];
		camerap->v[i][1] = para[7+6*i+1];
		camerap->v[i][2] = para[7+6*i+2];
		camerap->t[i][0] = para[7+6*i+3];
		camerap->t[i][1] = para[7+6*i+4];
		camerap->t[i][2] = para[7+6*i+5];
		V2R(camerap->v[i][0], camerap->v[i][1], camerap->v[i][2], R);
		for(j=0; j<3; j++){
			camerap->r1[i][j] = cvmGet(R,j,0);
			camerap->r2[i][j] = cvmGet(R,j,1);
			camerap->r3[i][j] = cvmGet(R,j,2);
		}
	}
	cvReleaseMat(&R);
}

void RefineCameraCalibrationParas(CameraPara *camerap, HomoMatrices hms, PtsPairs
ptspairs){
```

```
	//*********** LM optimization ************
	int i,j;
	int numimg = hms.num;
	int 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; // relevant only if the finite difference Jacobian version
is used
	void (*err)(double *p, double *hx, int m, int n, void *adata);
	int LM_m = (5+2+numimg*6), LM_n = 2*ptspairs.num_inlier;
	double *ptx = (double *)malloc(LM_n*sizeof(double));
	double *para = (double*)malloc(LM_m*sizeof(double));

	// initialize parameters
	Camerapara2Para(camerap, para, numimg);

	for(i=0; i<ptspairs.num_inlier; i++){
		ptx[2*i]   = ptspairs.inlierp2[i].x;
		ptx[2*i+1] = ptspairs.inlierp2[i].y;
	}
	err = CalculateCameraCalibrationDistFunc;
	ret = dlevmar_dif(err, para, ptx, LM_m, LM_n, 1000, opts, info, NULL, NULL,
&ptspairs); // no Jacobian
	printf("distortion: %f %f\n", info[0], info[1]);
	printf("LM algorithm iterations: %f \n", info[5]);

	// store paras
	Para2Camerapara(camerap, para, numimg);
}


void CameraCalibration(CameraPara *camerap, HomoMatrices hms, CvMat *K, PtsPairs
ptspairs){

	//********************* intrinsic paras *********************
	EstIntrinsicPara(camerap, hms, K);

	//********************* extrinsic paras *********************
	EstExtrinsicPara(camerap, hms, K);

	//*********** estimate radial distortion para ***************
	EstRadialDistPara(camerap, hms, K, ptspairs);
}

void PrintOutParas(CameraPara camerap, int numimgs){
	int i,j;
	printf("intrinsic K: \n");
	printf("    %f %f %f\n", camerap.alphax, camerap.skew, camerap.x0);
	printf("    0 %f %f\n", camerap.alphay, camerap.y0);
	printf("    0 0 1\n");
	printf("radial distortion para k1, k2:\n");
	printf("    %f %f\n", camerap.k1, camerap.k2);
	printf("extrinsic [r1 r2 r3 t]:\n");
	for(i=0; i<numimgs; i++){
		printf("  image %d: \n", i);
		for(j=0; j<3; j++)
			printf("    %f %f %f %f\n", camerap.r1[i][j], camerap.r2[i][j],
camerap.r3[i][j], camerap.t[i][j]);
	}
}

int main(int argc, char *argv[])
{
```

```
	int i,j,k;
	double scale =  40;
	int numimgs;
	IplImage *img = 0, *tmpimg = 0;
	IplImage *gimg = 0;
	IplImage *img_tr;
	CvMat *H  = cvCreateMat(3,3,CV_64FC1);  // homography matrix
	CvMat *invH  = cvCreateMat(3,3,CV_64FC1);
	CvMat *ptX  = cvCreateMat(3,1,CV_64FC1);
	CvMat *ptx  = cvCreateMat(3,1,CV_64FC1);
	CvMat *K = cvCreateMat(3,3,CV_64FC1); // intrinsic matrix
	CameraPara camerap;
	HomoMatrices hms;
	CvPoint2D64f patternp[MAX_CORNERPOINT_NUM];
	CvPoint2D64f cornerpts[MAX_CORNERPOINT_NUM*MAX_NUM_IMAGE];
	int numcorr;
	PtsPairs ptspairsall;
	PtsPairs ptspairs;
	CvPoint2D64f ptsSet1[MAX_CORNERPOINT_NUM], ptsSet2[MAX_CORNERPOINT_NUM];
	CvPoint2D64f ptsSet1all[MAX_CORNERPOINT_NUM*MAX_NUM_IMAGE],
ptsSet2all[MAX_CORNERPOINT_NUM*MAX_NUM_IMAGE];
	CvPoint2D64f ptsSet1tr[MAX_CORNERPOINT_NUM*MAX_NUM_IMAGE];
	int indicator[MAX_CORNERPOINT_NUM*MAX_NUM_IMAGE];
	ptspairs.inlierp1 = ptsSet1;
	ptspairs.inlierp2 = ptsSet2;
	ptspairsall.inlierp1 = ptsSet1all;
	ptspairsall.inlierp2 = ptsSet2all;
	ptspairsall.indicator = indicator;
	double *para;
	double transpt[2];
	char name[30];

	numimgs = atoi(argv[argc-1]);
	if(numimgs != argc-2){
		printf("Usage: main <image-file-name>\n\7");
		exit(0);
	}

	// specify pattern image corner points coordinate
	for(j=0; j<NUM_VER; j++){
		for(i=0; i<NUM_HOR; i++){
			patternp[j*NUM_HOR+i].x = (double)i*scale;
			patternp[j*NUM_HOR+i].y = (double)j*scale;
		}
	}

	// Compute homography for each image and record pts pairs
	hms.num = numimgs;
	numcorr = 0;
	for(i=0; i<numimgs; i++){
		// load image
		img = cvLoadImage(argv[i+1]);
		if(!img){
			printf("Could not load image file: %s\n", argv[i+1]);
			exit(0);
		}
		// compute H matrix
		ComputeHomography(i, img, patternp, &cornerpts[i*MAX_CORNERPOINT_NUM], H,
&ptspairs);
		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");
		// put H matrix into HSet
		for(j=0; j<3; j++)
			for(k=0; k<3; k++)
				hms.H[i][j][k] = cvmGet(H,j,k);

		// store points correspondences
		for(j=0; j<ptspairs.num_inlier; j++){
			// copy current ptspairs to ptspairsall
			ptsSet1all[numcorr].x = ptsSet1[j].x;
			ptsSet1all[numcorr].y = ptsSet1[j].y;
			ptsSet2all[numcorr].x = ptsSet2[j].x;
			ptsSet2all[numcorr].y = ptsSet2[j].y;
			indicator[numcorr++] = i;
		}
		hms.numcorr[i] = ptspairs.num_inlier;

		// transform X by H
		CvMat *check = cvCreateMat(img->height, img->width, CV_64FC1);
		cvInvert(H,invH);
		img_tr = cvCloneImage(img);
		//transform
		Trans_Image(&img, &img_tr, invH, check);
		//interpolation
		InterpolateImage(&img_tr, check);
		InterpolateImage(&img_tr, check);
		sprintf(name, "%d", i);
		strcat(name, "trans.jpg");
		cvSaveImage(name, img_tr);
		cvReleaseMat(&check);
	}
	ptspairsall.num_inlier = numcorr;

	// Camera Calibration for intrinsic, extrinsic and radial dist. paras
	CameraCalibration(&camerap, hms, K, ptspairsall);
	PrintOutParas(camerap, numimgs);
	para = new double[7+numimgs*6];
	Camerapara2Para(&camerap, para, numimgs);
	for(i=0; i<numimgs; i++){
		// load image
		img = cvLoadImage(argv[i+1]);
		if(!img){
			printf("Could not load image file: %s\n", argv[i+1]);
			exit(0);
		}
		// for each pattern corner
		for(j=0; j<NUM_VER; j++){
			for(k=0; k<NUM_HOR; k++){
				CameraCalibrationFunc(patternp[j*NUM_HOR+k], para, transpt, i,
1);
				cvCircle(img,
cvPoint((int)cornerpts[i*MAX_CORNERPOINT_NUM+j*NUM_HOR+k].x,
					(int)cornerpts[i*MAX_CORNERPOINT_NUM+j*NUM_HOR+k].y), 1,
cvScalar(0, 0, 255), 2, 8, 0);
				cvCircle(img, cvPoint((int)transpt[0], (int)transpt[1]), 1,
cvScalar(0, 255, 0), 2, 8, 0);
			}
		}
		sprintf(name, "%d", i);
		strcat(name, "proj.jpg");
		cvSaveImage(name, img);
	}
```

```
	// refine calibration paras
	RefineCameraCalibrationParas(&camerap, hms, ptspairsall);
	printf("after LM refinement\n");
	PrintOutParas(camerap, numimgs);

	// Project corner pts in calibration pattern into the image plane
	Camerapara2Para(&camerap, para, numimgs);
	for(i=0; i<numimgs; i++){
		// load image
		img = cvLoadImage(argv[i+1]);
		if(!img){
			printf("Could not load image file: %s\n", argv[i+1]);
			exit(0);
		}
		tmpimg = cvCloneImage(img);
		// for each pattern corner
		for(j=0; j<NUM_VER; j++){
			for(k=0; k<NUM_HOR; k++){
				CameraCalibrationFunc(patternp[j*NUM_HOR+k], para, transpt, i,
1);
				cvCircle(img,
cvPoint((int)cornerpts[i*MAX_CORNERPOINT_NUM+j*NUM_HOR+k].x,
					(int)cornerpts[i*MAX_CORNERPOINT_NUM+j*NUM_HOR+k].y), 1,
cvScalar(0, 0, 255), 2, 8, 0);
				cvCircle(img, cvPoint((int)transpt[0], (int)transpt[1]), 1,
cvScalar(0, 255, 0), 2, 8, 0);
			}
		}
		sprintf(name, "%d", i);
		strcat(name, "proj_refine.jpg");
		cvSaveImage(name, img);
	}

	// release
	cvReleaseMat(&K);
	cvReleaseImage(&img);
	cvReleaseImage(&img_tr);
	cvReleaseImage(&tmpimg);
	return 0;
}
```