ECE661 Computer Vision Homework 9

Epipolar Geometry and 3D Scene Reconstruction

Rong Zhang

12/04/2008

1 Problem

In this homework, we consider 3D projective reconstruction of the scene structure from a pair of stereo image. The fundamental matrix F is estimated from $n \geq 8$ point correspondences first. We then compute the epipoles $\vec{e}$ and $\vec{e}'$, and camera matrices P and P'. The triangulation approach is used to estimate scene structure $\vec{X}_i$. The LM optimization approach is then used to refine the camera matrix P' and scene structure $\vec{X}_i$. The last step in this homework is 3D scene reconstruction. Image rectification is applied in order to set epipoles to infinity and epipolar lines are parallel, which makes easier the interest-point selection for reconstruction.

2 Epipolar Geometry

Given any pair of matching point $\{\vec{x}_i, \vec{x}_i'\}$, the epipolar geometry constraint is

$$\vec{x}'^T F \vec{x} = 0. \tag{1}$$

For each pair, we have

$$A_i \vec{f} = \begin{bmatrix} x'x & x'y & x' & y'x & y'y & y' & x & y & 1 \end{bmatrix} \vec{f} = 0 \tag{2}$$

$$\tag{3}$$

where $\vec{f} = (f_{11}, f_{12}, ..., f_{33})^T$ is the 9-vector made up of the elements in F.

Therefore, if we have $n \geq 8$ point pairs, we are able to estimate F by solving the linear equations

$$\begin{aligned} &min \left\| A\vec{f} \right\| \\ &subject\ to \left\| \vec{f} \right\| = 1. \end{aligned} \tag{4}$$

From epipolar geometry, we have the gold standard approach for estimating F from the point correspondences as follows,

1. Compute an initial rank 2 estimation of F using normalized linear approach based on $n \geq 8$ point pairs $\{\vec{x}_i, \vec{x}'_i\}$.
   (a) Normalize n point pairs: $\hat{\vec{x}}_i = T\vec{x}_i$ and $\hat{\vec{x}}'_i = T'\vec{x}'_i$.
   (b) Linear solution: determine $\hat{F}'$ from equation 2;
   (c) Apply rank constraint: $det(F) = 0$ using SVD approach.
   (d) Denormalization: set $F = T'^T \hat{F}' T$
2. Compute an initial estimate of $\{\hat{\vec{x}}_i, \hat{\vec{x}}'_i\}$ as follows
   (a) Compute epipoles $\vec{e}$ and $\vec{e}'$ by $F\vec{e} = 0$, $\vec{e}'^T F = 0$, i.e., the right null vector and left null vector.
   (b) Set $P = [I|\vec{0}]$, $P' = [[\vec{e}']_x F|\vec{e}'] = [M|e']$.
   (c) Estimate $\hat{\vec{X}}_i$ by triangulation method.
   Assume $P = [\vec{p}^1, \vec{p}^2, \vec{p}^3]$ and $P' = [\vec{p}'^1, \vec{p}'^2, \vec{p}'^3]$, we have

$$\begin{aligned} x_i(\vec{p}^{3T}\vec{X}_i) - \vec{p}^{1T}\vec{X}_i = 0 \\ y_i(\vec{p}^{3T}\vec{X}_i) - \vec{p}^{2T}\vec{X}_i = 0 \\ x'_i(\vec{p}'^{3T}\vec{X}_i) - \vec{p}'^{1T}\vec{X}_i = 0 \\ y'_i(\vec{p}'^{3T}\vec{X}_i) - \vec{p}'^{2T}\vec{X}_i = 0 \end{aligned} \tag{5}$$

$$\tag{6}$$

   which can be written as $A\vec{X}_i = 0$ where

$$A = \begin{matrix} x_i\vec{p}^{3T} - \vec{p}^{1T} \\ y_i\vec{p}^{3T} - \vec{p}^{2T} \\ x'_i\vec{p}'^{3T} - \vec{p}'^{1T} \\ y'_i\vec{p}'^{3T} - \vec{p}'^{2T} \end{matrix} . \tag{7}$$

$$\tag{8}$$

   $\vec{X}_i$ can be solved by SVD approach.
3. using LM optimization, minimize the cost function

$$\sum_i d(\vec{x}_i, \hat{\vec{x}}_i)^2 + d(\vec{x}'_i, \hat{\vec{x}}'_i)^2 \tag{9}$$

   over F and $\vec{X}_i$. The total number of parameters is $3n + 12$: $3n$ for the n 3D points $\vec{X}_i$ and 12 for the camera matrix P'. Note that $\hat{\vec{x}}_i = P\vec{X}_i$ and $\hat{\vec{x}}'_i = P'\vec{X}_i$
4. update F and others, $\vec{e}$, $\vec{e}'$, P, P', $\vec{X}_i$.

3 Image Rectification

In order to reconstruct the 3D scene as much as possible, we need to establish interest-point pairs as much as possible and as accurate as possible. From the previous section, we have estimated the camera matrices P and P', we will use equation 7 to estimate the corresponding world coordinate of image $\vec{x}_i$ and $\vec{x}'_i$.

From epipolar geometry, we know that the two image point pair are on the epipolar line $\vec{l}$ and $\vec{l'}$ respectively. For more efficient and accurate point correspondences, we transform the images pair to make the epipoles locate at infinity. In that case, epipolar lines are parallel and in fact parallel to the x coordinate.

The image rectification process is as follows

1. for H'
   (a) Compute T that send the image center into origin
   (b) Compute R that rotate epipole to $[f, 0, 1]^T$
   (c) Compute G that send $[f, 0, 1]^T$ to $[f, 0, 0]^T$
   (d) Compute T_2 that reserve the original image center
   (e) Set $H' = T_2GRT$
2. for H
   (a) Compute $H_0 = H'M$
   (b) Set $H_a = [a\ b\ c;\ 0\ 1\ 0;\ 0,\ 0,\ 1]$
   (c) Minimize

$$\sum_i (a\hat{x}_i + b\hat{y}_i + c - \hat{x}'_i)^2 \tag{10}$$

   where $\vec{\hat{x}}_i = H_0\vec{x}_i$ and $\vec{\hat{x}}'_i = H'\vec{x}'_i$;
   (d) Set $H = H_aH_0$

4 Results

The results are showed in the following figures. 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.

Figure 1. Original image pair

Figure 2. Manually selected 12 point pairs and the estimated epipolar lines

Figure 3. Rectified images

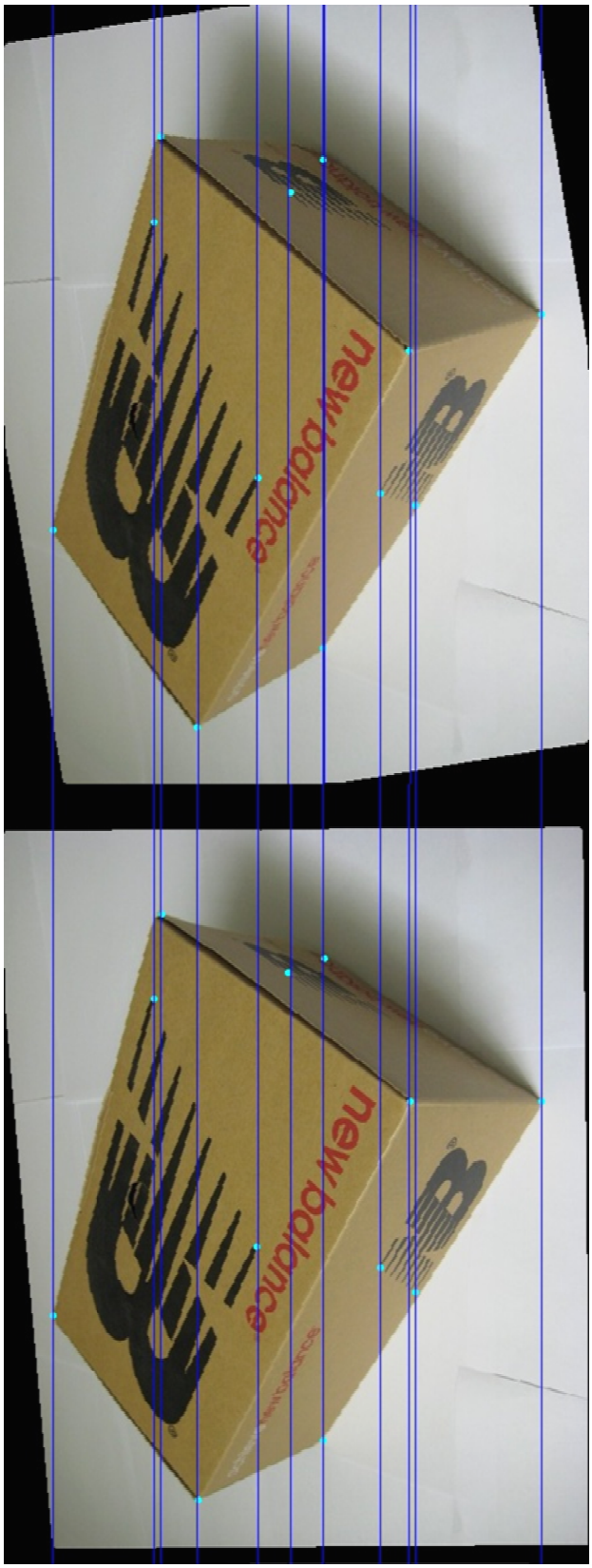

Figure 4. Image point pairs and epipolar lines after image rectification

(the epipolar lines are parallel to the x-axis; l and l’ almost has the same y coordinate)

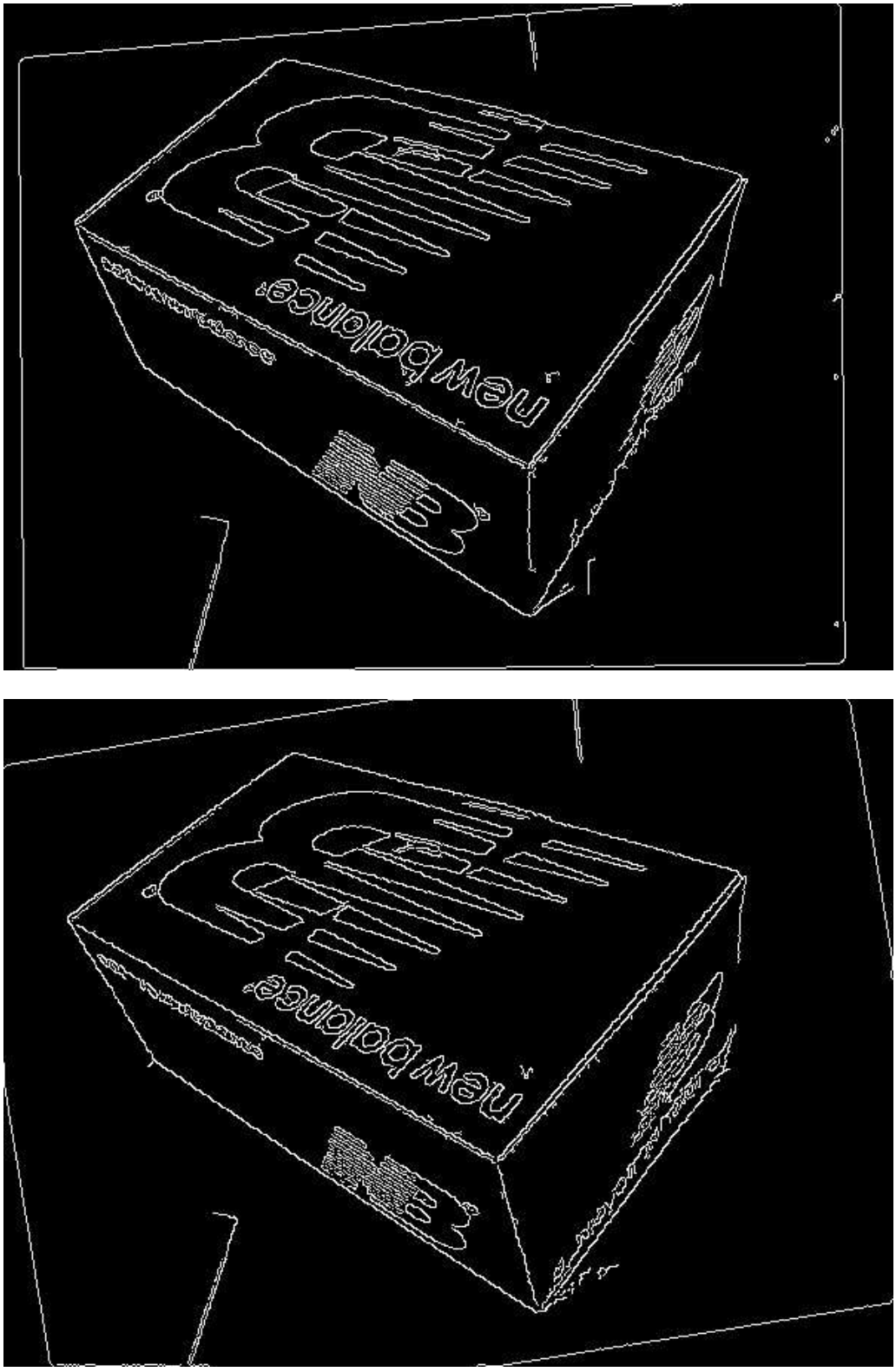

Figure 5. Edge detection by Canny filter after image rectification

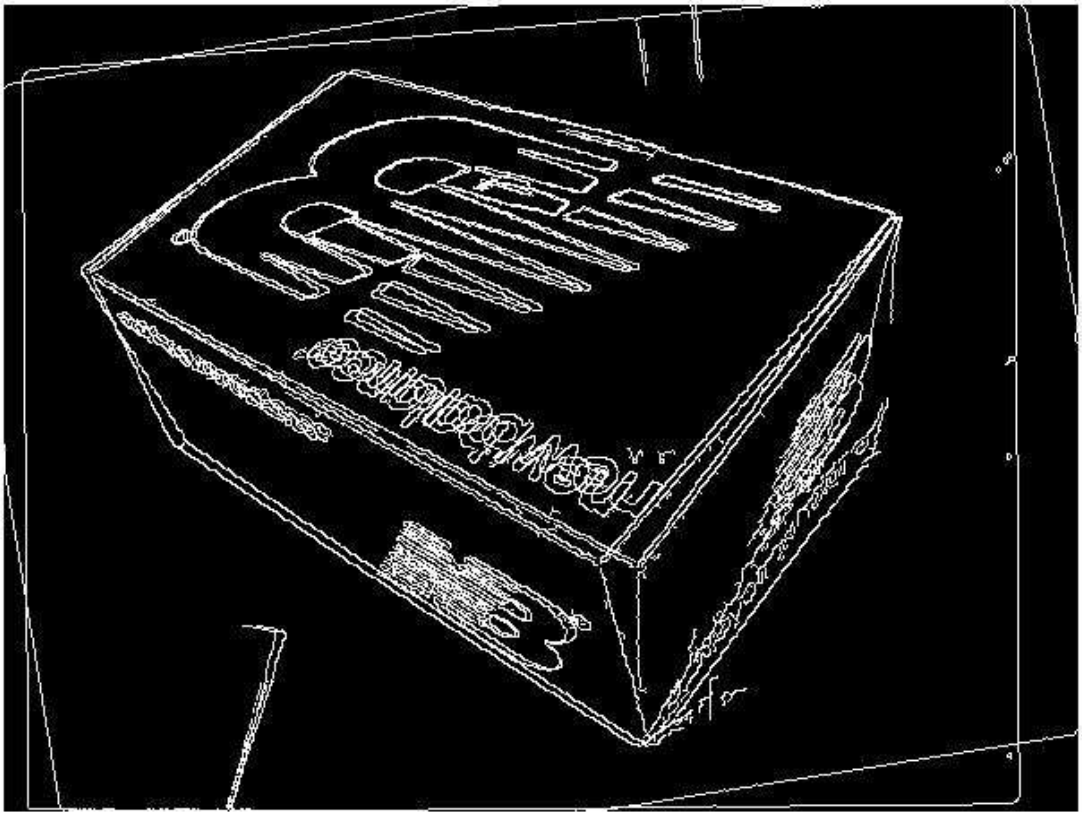

Figure 6. Combined detected edges of the image pair in Fig. 5

Figure 7. Matched point pairs based on NCC value

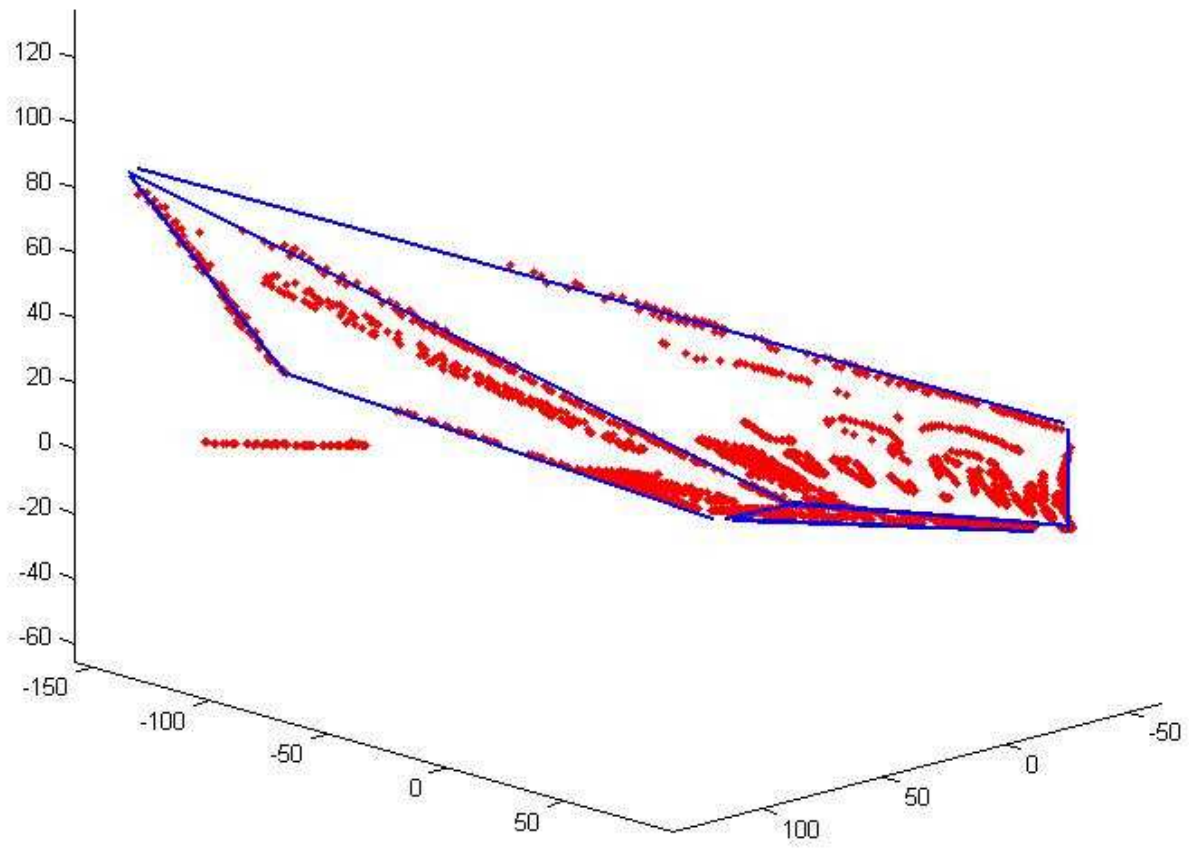

Figure 8. 3D reconstruction (up to a projective distortion)

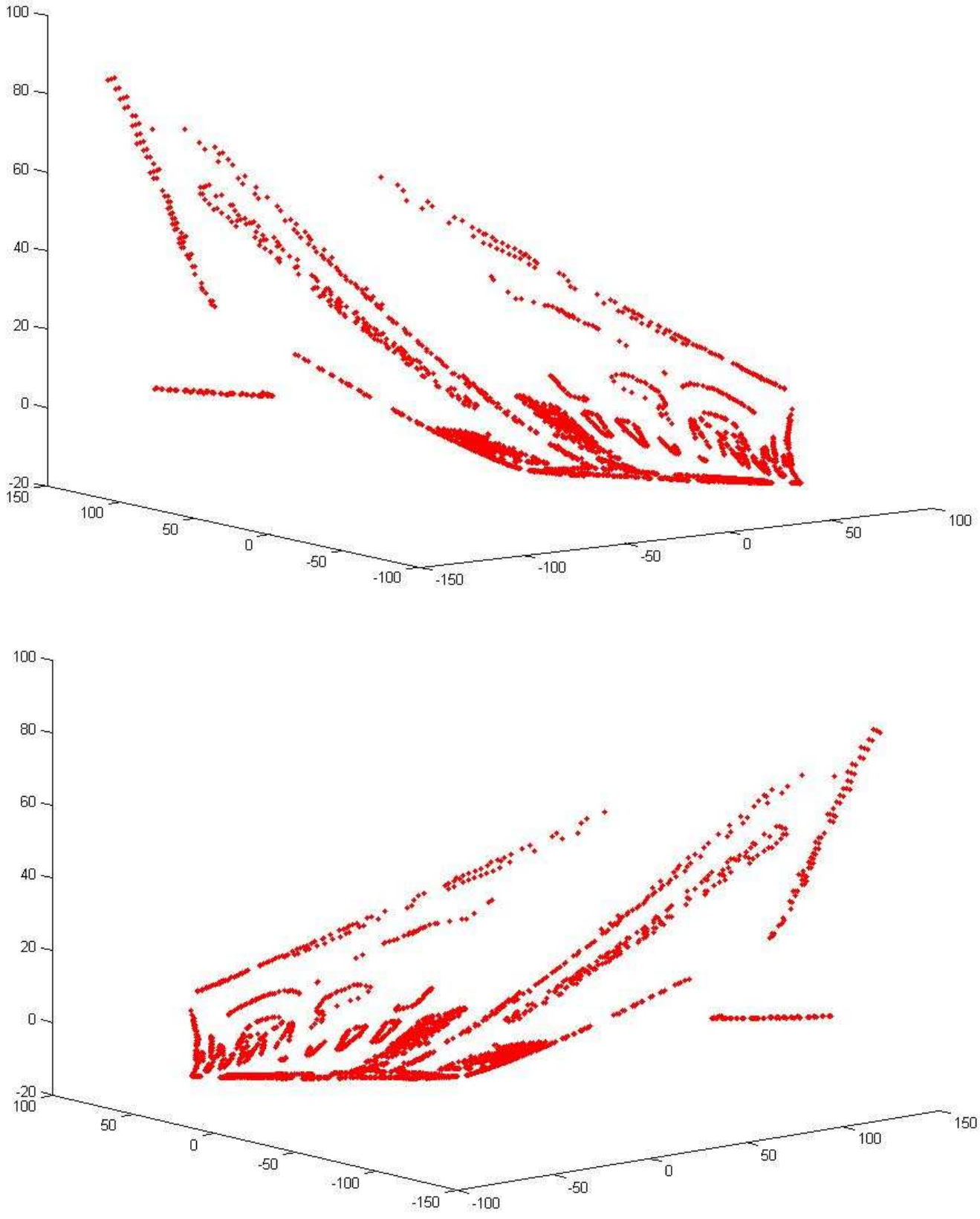

Figure 9. Other views: 3D reconstruction (up to a projective distortion)

F matrix is :

0.000002 0.000002 -0.014789

-0.000014 0.000003 0.073648

0.007414 -0.069132 1.000000

epipoles:

5413.950708 595.097372 1.000000

23697.520834 4745.031761 1.000000

P' is

35.180805 -328.033883 4744.958113 23697.520834

-175.699038 1638.258731 -23697.535623 4745.031761

-0.340943 0.051072 1815.458644 1.000000

distortion: 25.902560 6.652391

LM algorithm iterations: 221.000000

After LM optimization F is

0.000002 0.000001 -0.012333

-0.000011 0.000000 0.073191

0.005217 -0.068102 1.000000

P' is

25.020724 -322.105783 4746.186035 23837.988689

-123.611703 1614.406138 -23702.776411 4010.456416

-0.279521 0.005330 1793.366418 0.469050

H' is

0.974582 0.160595 -30.444607

-0.165849 0.979616 52.889146

-0.000019 -0.000003 1.000000

H is

0.827458 0.007485 13.367957

-0.078080 0.911814 39.503072

-0.000156 0.000004 1.000000

```
//*********************************************************
// 3D reconstruction
// * Epipolar geometry
// * Image rectification
// * 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 MAX_INTEREST_PT 500
#define numptpair 12
#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

typedef struct {
  CvPoint2D64f *p1, *p2;
  CvPoint3D64f *X;
  int num;
}PtsPairs;

void PrintCvMat(CvMat *A){
      int i,j;
      for(i=0; i<A->rows; i++){
            for(j=0; j<A->cols; j++)
                  printf("%f ", cvmGet(A,i,j));
            printf("\n");
      }
      printf("\n");
}

// distortion = d(x,PX)+d(x',P'X)
void DistFunc(int ithpoint, double h[], double tran_x[])
{
      int i,j;
      CvMat *P   = cvCreateMat(3,4,CV_64FC1);
      CvMat *Pp  = cvCreateMat(3,4,CV_64FC1);
      CvMat *X  = cvCreateMat(4,1,CV_64FC1);
      CvMat *xtmp = cvCreateMat(3,1,CV_64FC1);
      CvMat *xptmp = cvCreateMat(3,1,CV_64FC1);

      // Set P
      cvZero(P);
      cvmSet(P,0,0,1);
      cvmSet(P,1,1,1);
      cvmSet(P,2,2,1);
```

```
	// Set P'
	for(i=0; i<3; i++)
		for(j=0; j<4; j++)
			cvmSet(Pp,i,j,h[i*4+j]);

	// Set X
	for(j=0; j<3; j++)
		cvmSet(X,j,0,h[12+ithpoint*3+j]);
	cvmSet(X,3,0,1);

	cvMatMul(P,X,xtmp);
	tran_x[0] = cvmGet(xtmp,0,0)/cvmGet(xtmp,2,0);
	tran_x[1] = cvmGet(xtmp,1,0)/cvmGet(xtmp,2,0);
	cvMatMul(Pp,X,xptmp);
	tran_x[2] = cvmGet(xptmp,0,0)/cvmGet(xptmp,2,0);
	tran_x[3] = cvmGet(xptmp,1,0)/cvmGet(xptmp,2,0);

	cvReleaseMat(&P);
	cvReleaseMat(&Pp);
	cvReleaseMat(&X);
	cvReleaseMat(&xtmp);
	cvReleaseMat(&xptmp);
}

static void CalculateDistFunc(double *p, double *tran_x, int m, int n, void *adata)
{
	int i;
	PtsPairs * pair;
	pair = (PtsPairs *)adata;
	for(i=0; i<pair->num; i++)
		DistFunc(i, p, tran_x+i*4);
}

//*********************************************************************
// 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);
}

//*****************************************************************
//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* ch = NULL){
        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);
        CvMat *check;

        if(ch == NULL)
              check = cvCreateMat(height,width,CV_64FC1);

        cvZero(*img_xp);
        for (i=0; i<height; i++){       //y - ver
              for (j=0; j<width; j++){   //x - hor
                    if(j==248 && i==82)
                          int rr=0;
                // 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));
                if(check != NULL)
                      cvmSet(check,curpi,curpj,1);
              }
        }
        InterpolateImage(img_xp,check);
        InterpolateImage(img_xp,check);
        if(ch == NULL)
              cvReleaseMat(&check);
        // Release
        cvReleaseMat(&ptx);
        cvReleaseMat(&ptxp);
}

//********** estimate the fundamental matrix **********
```

```
// input: points correspondences pt1 & pt2
//        number of point pairs: num
// output: F is a 3*3 matrix with the rank constraint
//*************************************************
void EstFundamentalMatrix(int num, CvPoint2D64f *pt1, CvPoint2D64f *pt2, CvMat *F){
      int i;
      CvMat *A = cvCreateMat(num,9,CV_64FC1);
      CvMat *Ftmp = cvCreateMat(3,3,CV_64FC1);// for rank constraint
      CvMat *T1 = cvCreateMat(3,3,CV_64FC1);  // for normalization
      CvMat *T2 = cvCreateMat(3,3,CV_64FC1);  // for normalization
      CvMat *T2T = cvCreateMat(3,3,CV_64FC1);
      CvMat *D = cvCreateMat(num,9,CV_64FC1);
      CvMat *U = cvCreateMat(num,num,CV_64FC1);
      CvMat *V = cvCreateMat(9,9,CV_64FC1);
      CvMat *DD = cvCreateMat(3,3,CV_64FC1);
      CvMat *UU = cvCreateMat(3,3,CV_64FC1);
      CvMat *VV = cvCreateMat(3,3,CV_64FC1);
      CvMat *UUT = cvCreateMat(3,3,CV_64FC1);
      CvPoint2D64f *p1 = new CvPoint2D64f[num];
      CvPoint2D64f *p2 = new CvPoint2D64f[num];

      for(i=0; i<num; i++){
            p1[i].x = pt1[i].x;
            p1[i].y = pt1[i].y;
            p2[i].x = pt2[i].x;
            p2[i].y = pt2[i].y;
      }

      // normalization
      Normalization(num, p1, T1); //p1 <- T1 * p1
      Normalization(num, p2, T2); //p2 <- T2 * p2

      // Set A
      for(i=0; i<num; i++){
            // each points correspondances create an equation
            cvmSet(A,i,0,p2[i].x*p1[i].x);
            cvmSet(A,i,1,p2[i].x*p1[i].y);
            cvmSet(A,i,2,p2[i].x);
            cvmSet(A,i,3,p2[i].y*p1[i].x);
            cvmSet(A,i,4,p2[i].y*p1[i].y);
            cvmSet(A,i,5,p2[i].y);
            cvmSet(A,i,6,p1[i].x);
            cvmSet(A,i,7,p1[i].y);
            cvmSet(A,i,8,1);
      }
      // solve min{Af} s.t. |f|=1
      // 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(Ftmp,i/3,i%3,cvmGet(V,8,i));

      // Apply rank constraint
      // Ftmp = UU^T DD VV
      cvSVD(Ftmp, DD, UU, VV, CV_SVD_U_T|CV_SVD_V_T);
      cvmSet(DD,2,2,0); // DD = diag{s,t,0}
      cvTranspose(UU, UUT);
      cvMatMul(UUT, DD, Ftmp);
      cvMatMul(Ftmp, VV, Ftmp);
```

```
	// Denormalization
	// Set F = T2_trans * Ftmp * T1
	cvTranspose(T2, T2T);
	cvMatMul(T2T, Ftmp, Ftmp);
	cvMatMul(Ftmp, T1, F);
	cvScale(F,F,1/cvmGet(F,2,2));

	printf("F matrix is :\n");
	PrintCvMat(F);

	// Release
	cvReleaseMat(&A);
	cvReleaseMat(&Ftmp);
	cvReleaseMat(&D);
	cvReleaseMat(&U);
	cvReleaseMat(&V);
	cvReleaseMat(&DD);
	cvReleaseMat(&UU);
	cvReleaseMat(&VV);
	cvReleaseMat(&UUT);
	cvReleaseMat(&T1);
	cvReleaseMat(&T2);
	cvReleaseMat(&T2T);
	delete p1,p2;
}

//******* Compute Epipoles ******
// Fe = 0 & e'^T*F = 0
// input: Fundamental matrix F
// output: e & e'  (3*1 vector)
//*******************************
void ComputeEpipoles(CvMat *F, CvMat *e, CvMat *ep) {
	int i;
	CvMat *D = cvCreateMat(3,3,CV_64FC1);
	CvMat *U = cvCreateMat(3,3,CV_64FC1);
	CvMat *V = cvCreateMat(3,3,CV_64FC1);

	// F = U^T D V
	cvSVD(F, D, U, V, CV_SVD_U_T|CV_SVD_V_T);
	// e =  the last column of V^T, i.e., last row of V
	// ep = the last column of U^T, i.e., last row of U
	for(i = 0; i < 3; i++){
		cvmSet(e , i, 0, cvmGet(V, 2, i) / cvmGet(V, 2, 2));
		cvmSet(ep, i, 0, cvmGet(U, 2, i) / cvmGet(U, 2, 2));
	}
	printf("epipoles: \n");
	PrintCvMat(e);
	PrintCvMat(ep);

	// Release
	cvReleaseMat(&D);
	cvReleaseMat(&U);
	cvReleaseMat(&V);
}

//******* vector v to [v]x *********
// input: vector d*1
// output: matrix d*d [v]x
//*********************************
void Vector2SkewMatrix(CvMat *v, CvMat *vx){
```

```
	cvZero(vx);
	cvmSet(vx,0,1,-cvmGet(v,2,0));
	cvmSet(vx,0,2, cvmGet(v,1,0));
	cvmSet(vx,1,0, cvmGet(v,2,0));
	cvmSet(vx,1,2,-cvmGet(v,0,0));
	cvmSet(vx,2,0,-cvmGet(v,1,0));
	cvmSet(vx,2,1, cvmGet(v,0,0));
}

//******** Set camera matrices ******************
// P = [I,0^T], Pp = [SF,e'] (Pp = [[e']xF,e'] used)
// input: Fundamental matrix F
//        epipole e'
// output: Camera matrices 3*4 P, Pp
//***********************************************
void ComputeCameraMatrices(CvMat *F, CvMat *ep, CvMat *P, CvMat *Pp){
	int i,j;
	CvMat *epx  = cvCreateMat(3,3,CV_64FC1);  //[ep]x
	CvMat *tmp  = cvCreateMat(3,3,CV_64FC1);

	cvZero(P);
	cvZero(Pp);

	// Set P
	cvmSet(P,0,0,1);
	cvmSet(P,1,1,1);
	cvmSet(P,2,2,1);

	// Set Pp
	// Set epx
	Vector2SkewMatrix(ep,epx);
	cvMatMul(epx,F,tmp);
	for(i=0; i<3; i++)
		for(j=0; j<3; j++)
			cvmSet(Pp,i,j,cvmGet(tmp,i,j));
	for(i=0; i<3; i++)
		cvmSet(Pp,i,3,cvmGet(ep,i,0));

	printf("P' is \n");
	PrintCvMat(Pp);

	// Release
	cvReleaseMat(&epx);
	cvReleaseMat(&tmp);
}


//******** Compute Epipolar Lines ******************
// l' = Fx   l = F^T x'
// input: Fundamental matrix F
//        image point x
//        indicator: left image or right image (x or x')
// output: intersection points of the epipole line
//         l or l' with image boundary: intersectp1 & intersectp2
//***************************************************
void ComputeEpipolarLine(int width, int height, CvMat *F, CvPoint2D64f p, int isRight,
CvPoint2D64f *intersectp1, CvPoint2D64f *intersectp2) {
	int i;
	CvPoint2D64f inter_bd[4];
	int mask[4];
```

```
	CvMat *x = cvCreateMat(3, 1, CV_64FC1);
	CvMat *line = cvCreateMat(3, 1, CV_64FC1);
	cvmSet(x,0,0,p.x);
	cvmSet(x,1,0,p.y);
	cvmSet(x,2,0,1);

	if(isRight) // left image x right image epipolar line l'
		cvMatMul(F,x,line);
	else{ // right image x' left image epipolar line l
		CvMat *FT = cvCreateMat(3, 3, CV_64FC1);
		cvTranspose(F,FT);
		cvMatMul(FT,x,line);
		cvReleaseMat(&FT);
	}

	inter_bd[0].x = -cvmGet(line,2,0)/cvmGet(line,0,0);   // intersect with hor
(x,0,1)
	inter_bd[0].y = 0;
	inter_bd[1].x = -(cvmGet(line,2,0)+cvmGet(line,1,0)*height)/cvmGet(line,0,0);
// intersect with hor (x,height,1)
	inter_bd[1].y = height;
	inter_bd[2].x = 0;
	inter_bd[2].y = -cvmGet(line,2,0)/cvmGet(line,1,0);   // intersect with ver
(0,y,1)
	inter_bd[3].x = width;
	inter_bd[3].y = -(cvmGet(line,2,0)+cvmGet(line,0,0)*width)/cvmGet(line,1,0);
// intersect with ver (width,y,1)

	for(i=0; i<2; i++)
		mask[i] = (inter_bd[i].x>=0 && inter_bd[i].x<width)?1:0;
	for(i=2; i<4; i++)
		mask[i] = (inter_bd[i].y>=0 && inter_bd[0].y< height)?1:0;

	i=0;
	while(mask[i]==0)
		i++;
	intersectp1->x = inter_bd[i].x;
	intersectp1->y = inter_bd[i].y;

	i++;
	while(mask[i]==0)
		i++;
	intersectp2->x = inter_bd[i].x;
	intersectp2->y = inter_bd[i].y;
}

// apply xnew = Hx
void TransformationByH(CvMat *H, CvPoint2D64f *x, CvPoint2D64f *xnew){
	CvMat *ptx = cvCreateMat(3,1,CV_64FC1);
	CvMat *ptxnew = cvCreateMat(3,1,CV_64FC1);

	cvmSet(ptx,0,0,x->x);
	cvmSet(ptx,1,0,x->y);
	cvmSet(ptx,2,0,1);
	cvMatMul(H,ptx,ptxnew);
	xnew->x = cvmGet(ptxnew,0,0)/cvmGet(ptxnew,2,0);
	xnew->y = cvmGet(ptxnew,1,0)/cvmGet(ptxnew,2,0);

	cvReleaseMat(&ptx);
	cvReleaseMat(&ptxnew);
```

```
}

//************* Image Rectification ***************
// compute the matrix H' such that H'e' = [1 0 0]'
// and then compute the H=HaH0 for left image to
// minimize the disparity
// input:  e', P, P', xi, xi'
// output: matrix H' = T2GRT and H=HaH0 and F_new
//*************************************************
void ImageRectification(int width, int height, int num, CvPoint2D64f *x, CvPoint2D64f
*xp, CvMat *ep, CvMat *P, CvMat *Pp, CvMat *H, CvMat *Hp, CvMat *F){

      if(cvmGet(ep,2,0) == 0){
            printf("no need to rectification, error!\n");
            exit(0);
      }
      int i;
      int centerx, centery;
      double alpha, f;
      CvMat *G = cvCreateMat(3,3,CV_64FC1);
      CvMat *R = cvCreateMat(3,3,CV_64FC1);
      CvMat *T = cvCreateMat(3,3,CV_64FC1);
      CvMat *T2 = cvCreateMat(3,3,CV_64FC1);
      CvMat *invP = cvCreateMat(4,3,CV_64FC1);
      CvMat *M = cvCreateMat(3,3,CV_64FC1);
      CvMat *H0 = cvCreateMat(3,3,CV_64FC1);
      CvMat *Ha = cvCreateMat(3,3,CV_64FC1);
      CvMat *A = cvCreateMat(num,3,CV_64FC1);
      CvMat *b = cvCreateMat(num,1,CV_64FC1);
      CvMat *t = cvCreateMat(3,1,CV_64FC1);
      CvMat *HpinvT = cvCreateMat(3,3,CV_64FC1);
      CvMat *Hinv = cvCreateMat(3,3,CV_64FC1);
      CvPoint2D64f xnew, xpnew;

      //*************** for H' ***************
      centerx = (int)width/2;   //x0
      centery = (int)height/2;  //y0
      // Set T = [1 0 -x0; 0 1 -y0; 0 0 1] s.t. G[x0,y0,1]'=[0 0 1]'
      cvZero(T);
      cvmSet(T,0,0,1);
      cvmSet(T,1,1,1);
      cvmSet(T,2,2,1);
      cvmSet(T,0,2,-centerx);
      cvmSet(T,1,2,-centery);

      // ep: [epx epy 1]' is mapped (by G) into [epx-x0 epy-y0 1]'
      // Set rotation matrix R = [cos(a) -sin(a) 0; sin(a) cos(a) 0; 0 0 1]
      // s.t. R[epx-x0, epy-y0, 1]'=[f 0 1]'
      // cos(a)*(epx-x0)-sin(a)*(epy-y0)=f
      // sin(a)*(epx-x0)+cos(a)*(epy-y0)=0
      alpha = atan(-(cvmGet(ep,1,0)/cvmGet(ep,2,0)-
centery)/(cvmGet(ep,0,0)/cvmGet(ep,2,0)-centerx));
      f = cos(alpha)*(cvmGet(ep,0,0)/cvmGet(ep,2,0)-centerx)-
sin(alpha)*(cvmGet(ep,1,0)/cvmGet(ep,2,0)-centery);
      cvZero(R);
      cvmSet(R,0,0,cos(alpha));
      cvmSet(R,0,1,-sin(alpha));
      cvmSet(R,1,0,sin(alpha));
      cvmSet(R,1,1,cos(alpha));
      cvmSet(R,2,2,1);
```

```
// Set G = [1 0 0; 0 1 0; -1/f 0 1];
cvZero(G);
cvmSet(G,0,0,1);
cvmSet(G,1,1,1);
cvmSet(G,2,2,1);
cvmSet(G,2,0,-1/f);

// H' = GRT
// H' will send e' to [f 0 0]'
cvMatMul(G,R,Hp);
cvMatMul(Hp,T,Hp);

// preserve center point
xnew.x = centerx;
xnew.y = centery;
TransformationByH(Hp, &xnew, &xpnew);
// T2 = [1 0 centerx-xpnew.x; 0 1 centery-xpnew.y; 0 0 1];
cvZero(T2);
cvmSet(T2,0,0,1);
cvmSet(T2,0,2,centerx-xpnew.x);
cvmSet(T2,1,1,1);
cvmSet(T2,1,2,centery-xpnew.y);
cvmSet(T2,2,2,1);
cvMatMul(T2,Hp,Hp);

printf("H' is \n");
cvScale(Hp,Hp,1/cvmGet(Hp,2,2));
PrintCvMat(Hp);

//*************** for H ***************
// M = P'P+
cvPseudoInv(P, invP);
cvMatMul(Pp, invP, M);

//cvScale(M,M,10
// H0 = H'M
cvMatMul(Hp, M, H0);

// for Ha solve linear equations
for(i=0; i<num; i++){
      // transform x': H'x'
      TransformationByH(Hp,&xp[i],&xpnew);
      // transform x:  H0x
      TransformationByH(H0,&x[i],&xnew);
      // one equation for each point pair
      cvmSet(A,i,0,xnew.x);
      cvmSet(A,i,1,xnew.y);
      cvmSet(A,i,2,1);
      cvmSet(b,i,0,xpnew.x);
}
cvSolve(A,b,t,CV_SVD);

// Set Ha = [a b c; 0 1 0; 0 0 1];
cvZero(Ha);
cvmSet(Ha,0,0,cvmGet(t,0,0));
cvmSet(Ha,0,1,cvmGet(t,1,0));
cvmSet(Ha,0,2,cvmGet(t,2,0));
cvmSet(Ha,1,1,1);
cvmSet(Ha,2,2,1);
```

```
	// H = HaH0
	cvMatMul(Ha,H0,H);

	printf("H is \n");
	cvScale(H,H,1/cvmGet(H,2,2));
	PrintCvMat(H);

	// update F = H'^{-t} F H^{-1}
	HpinvT = cvCreateMat(3,3,CV_64FC1);
	cvInvert(H, Hinv);
	cvInvert(Hp, HpinvT);
	cvTranspose(HpinvT, HpinvT);
	cvMatMul(HpinvT, F, F);
	cvMatMul(F,Hinv,F);
	cvScale(F,F,1/cvmGet(F,2,2));

	printf("H'-t is\n");
	cvScale(HpinvT, HpinvT, 1/cvmGet(HpinvT,2,2));
	PrintCvMat(HpinvT);
	cvScale(Hinv, Hinv, 1/cvmGet(Hinv,2,2));
	printf("H-1 is\n");
	PrintCvMat(Hinv);

	// Release
	cvReleaseMat(&G);
	cvReleaseMat(&R);
	cvReleaseMat(&T);
	cvReleaseMat(&T2);
	cvReleaseMat(&invP);
	cvReleaseMat(&M);
	cvReleaseMat(&H0);
	cvReleaseMat(&Ha);
	cvReleaseMat(&A);
	cvReleaseMat(&b);
	cvReleaseMat(&t);
	cvReleaseMat(&HpinvT);
	cvReleaseMat(&Hinv);
}

void Reconstruct3DPt(int num, CvPoint2D64f *x, CvPoint2D64f *xp, CvMat *P, CvMat *Pp,
CvPoint3D64f *X){
	int i,j;
	CvMat *A  = cvCreateMat(4,4,CV_64FC1);
	CvMat *D  = cvCreateMat(4,4,CV_64FC1);
	CvMat *U  = cvCreateMat(4,4,CV_64FC1);
	CvMat *V  = cvCreateMat(4,4,CV_64FC1);

	for(i=0; i<num; i++){ // for each correspondence
		for(j=0; j<4; j++){ // for each col
			cvmSet(A,0,j,x[i].x *cvmGet(P,2,j) - cvmGet(P,0,j));
			cvmSet(A,1,j,x[i].y *cvmGet(P,2,j) - cvmGet(P,1,j));
			cvmSet(A,2,j,xp[i].x*cvmGet(Pp,2,j) - cvmGet(Pp,0,j));
			cvmSet(A,3,j,xp[i].y*cvmGet(Pp,2,j) - cvmGet(Pp,1,j));

		}
		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
		X[i].x = cvmGet(V,3,0)/cvmGet(V,3,3);
		X[i].y = cvmGet(V,3,1)/cvmGet(V,3,3);
		X[i].z = cvmGet(V,3,2)/cvmGet(V,3,3);
```

```
			//printf("%f %f %f\n", X[i].x, X[i].y, X[i].z);
		}
		cvReleaseMat(&A);
		cvReleaseMat(&D);
		cvReleaseMat(&U);
		cvReleaseMat(&V);
}

// LM algorithm
// para
// output: updated F, e, e', Pp
void LMOptimizationPara(int num, CvPoint2D64f *p1, CvPoint2D64f *p2, CvMat *Pp,
CvPoint3D64f *X, CvMat *e, CvMat *ep, CvMat *F){

		int i,j;
		CvMat *epx = cvCreateMat(3,3,CV_64FC1);  //[ep]x
		CvMat *M = cvCreateMat(3,3,CV_64FC1);
		CvMat *D = cvCreateMat(3,3,CV_64FC1);
		CvMat *U = cvCreateMat(3,3,CV_64FC1);
		CvMat *V = cvCreateMat(3,3,CV_64FC1);
		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 = 12+3*num, LM_n = 4*num;
		double *x = (double *)malloc(LM_n*sizeof(double));
		double *p = (double*)malloc(LM_m*sizeof(double));

		PtsPairs ptspairs;
		ptspairs.num = num;
		ptspairs.p1 = p1;
		ptspairs.p2 = p2;
		ptspairs.X = X;

		for(i=0; i<3; i++)
			for(j=0; j<4; j++)
				p[4*i+j] = cvmGet(Pp,i,j);

		for(i=0; i<num; i++){
			p[12+i*3] = X[i].x;
			p[12+i*3+1] = X[i].y;
			p[12+i*3+2] = X[i].z;
		}
		for(i=0; i<num; i++){
			j = i<<2;
			x[j]   = ptspairs.p1[i].x;
			x[j+1] = ptspairs.p1[i].y;
			x[j+2] = ptspairs.p2[i].x;
			x[j+3] = ptspairs.p2[i].y;
		}
		err = CalculateDistFunc;
		ret = dlevmar_dif(err, p, x, 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]);

		// update parameters
```

```
	for(i=0; i<3; i++)
		for(j=0; j<4; j++)
			cvmSet(Pp,i,j,p[4*i+j]);

	for(i=0; i<num; i++){
		X[i].x = p[12+i*3];
		X[i].y = p[12+i*3+1];
		X[i].z = p[12+i*3+2];
	}

	// update e'
	for(i=0; i<3; i++)
		cvmSet(ep,i,0,cvmGet(Pp,i,3));

	// update F
	Vector2SkewMatrix(ep,epx);
	for(i=0; i<3; i++)
		for(j=0; j<3; j++)
			cvmSet(M,i,j,cvmGet(Pp,i,j));
	cvMatMul(epx,M,F);
	cvScale(F,F,1/cvmGet(F,2,2));

	// update e
	// F = U^T D V
	cvSVD(F, D, U, V, CV_SVD_U_T|CV_SVD_V_T);
	// e =  the last column of V^T, i.e., last row of V
	// ep = the last column of U^T, i.e., last row of U
	for(i = 0; i < 3; i++)
		cvmSet(e , i, 0, cvmGet(V, 2, i) / cvmGet(V, 2, 2));

	printf("After LM optimization F is \n");
	PrintCvMat(F);

	cvReleaseMat(&epx);
	cvReleaseMat(&M);
	cvReleaseMat(&D);
	cvReleaseMat(&U);
	cvReleaseMat(&V);
}

// generate a new image displaying the two images
void ShowTwoImage(IplImage *img1, IplImage *img2, IplImage **img_re){
	int i,j;
	cvZero(*img_re);
	for(i=0; i<(*img_re)->height; i++){
		for(j=0; j<(*img_re)->width; j++){
		  if(i<img1->height && j<img1->width)
			cvSet2D(*img_re,i,j,cvGet2D(img1,i,j));
		  else
			cvSet2D(*img_re,i,j,cvGet2D(img2,i,j-img1->width));
		}
	}
}

//*****************************************************************
// Similarity measure based on NCC
// input:  img1, img2, (images)
//         p1, p2 (feature pts for each image  x and x')
//         num1, num2 (ttl number of detected pts for each image)
// output: m1, m2 (matched pairs)
```

```
// return the total number of matched pairs
//*************************************************************
int InterestedPointMatching_NCC(IplImage *img1, IplImage *img2, CvPoint *p1, int num1,
CvPoint *p2, int num2, CvPoint2D64f  *m1, CvPoint2D64f  *m2){
      int i,j,ii,jj,idx;
      double cur_value;
      double MAX_value;
      int cur_x, cur_y, match_x, match_y;
      double mean1, mean2;
      int available_num;
      CvScalar intensity;
      double tmp1, tmp2;
      double v1, v2, v3;
      double *nccvalues = new double[num1];
      int *matchedidx = new int [num1];
      int check = 0;

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

      idx = 0;
      for(i=0; i<num1; i++){
            // for each point in p1, find a match in p2
            MAX_value = -10000;
            cur_x = p1[i].x;
            cur_y = p1[i].y;
            m1[idx].x = (double)cur_x;
            m1[idx].y = (double)cur_y;

            for(j=0; j<num2; j++){
                  if(pow(cur_x-p2[j].x, 2.0) > 400)
                        continue;
                  match_x = p2[j].x;
                  match_y = p2[j].y;
                  available_num = 0;
                  mean1 = 0; mean2 = 0;
                  for(ii=-W_SIZE_MATCH; ii<W_SIZE_MATCH; ii++){
                        for(jj=-W_SIZE_MATCH; jj<W_SIZE_MATCH; jj++){
                              if(cur_y+ii < 0 || cur_y+ii >= height || cur_x+jj < 0
|| cur_x+jj >=width)
                                    continue;
                              intensity = cvGet2D(img1, cur_y+ii, cur_x+jj);
                              mean1 += intensity.val[0];
                              intensity = cvGet2D(img2, match_y+ii, match_x+jj);
                              mean2 += intensity.val[0];
                              available_num++;
                        }
                  }
                  mean1 /= available_num;
                  mean2 /= available_num;

                  v1 = 0; v2 = 0; v3 = 0;
                  for(ii=-W_SIZE_MATCH; ii<W_SIZE_MATCH; ii++){
                        for(jj=-W_SIZE_MATCH; jj<W_SIZE_MATCH; jj++){
                              if(cur_y+ii < 0 || cur_y+ii >= height || cur_x+jj < 0
|| cur_x+jj >=width)
                                    continue;
                              intensity = cvGet2D(img1, cur_y+ii, cur_x+jj);
                              tmp1 = intensity.val[0] - mean1;
                              intensity = cvGet2D(img2, match_y+ii, match_x+jj);
```

```
                        tmp2 = intensity.val[0] - mean2;
                        v1 += tmp1*tmp2;
                        v2 += pow(tmp1, 2.0);
                        v3 += pow(tmp2, 2.0);

                    }
                }
                cur_value = v1 / sqrt(v2*v3);
                if(cur_value > MAX_value)
                {
                    // a better match
                    MAX_value = cur_value;
                    nccvalues[idx] = cur_value;
                    m2[idx].x = (double)match_x;
                    m2[idx].y = (double)match_y;
                    matchedidx[idx] = j;
                }

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

        // bubble sorting
        double tmpm1x, tmpm1y, tmpm2x, tmpm2y;
        for(i = 0; i < idx; i++){
            for(j = 0; j < idx-i-1; j++){
                if(m1[j].x > m1[j+1].x){
                    // swap
                    tmpm1x = m1[j].x;
                    tmpm1y = m1[j].y;
                    tmpm2x = m2[j].x;
                    tmpm2y = m2[j].y;
                    m1[j].x = m1[j+1].x;
                    m1[j].y = m1[j+1].y;
                    m2[j].x = m2[j+1].x;
                    m2[j].y = m2[j+1].y;
                    m1[j+1].x = tmpm1x;
                    m1[j+1].y = tmpm1y;
                    m2[j+1].x = tmpm2x;
                    m2[j+1].y = tmpm2y;
                }
            }
        }
        delete nccvalues;
        delete matchedidx;
        return idx;
}
```

```
int main(int argc, char *argv[])
{
	int i,j,k;
	IplImage *img1=0, *img2=0, *tmpimg1=0, *tmpimg2;
	IplImage *gimg=0, *edgeimg1, *edgeimg2, *imgre;
	int width, height;
	CvMat *F  = cvCreateMat(3,3,CV_64FC1); // Fundamental matrix F
	CvMat *e  = cvCreateMat(3,1,CV_64FC1); // e
	CvMat *ep = cvCreateMat(3,1,CV_64FC1); // e'
	CvMat *P  = cvCreateMat(3,4,CV_64FC1); // P
	CvMat *Pp = cvCreateMat(3,4,CV_64FC1); // P'
	CvMat *H  = cvCreateMat(3,3,CV_64FC1); // H  rectification matrix
	CvMat *Hp = cvCreateMat(3,3,CV_64FC1); // H'
	CvMat *invH  = cvCreateMat(3,3,CV_64FC1);
	CvMat *invHp = cvCreateMat(3,3,CV_64FC1);

	double p1x[] = {41,94,385,501,536,206,385,265,470,225,246,490};
	double p1y[] = {123,230,417,249,125,14,312,184,114,311,283,220};
	double p2x[] = {58,103,330,474,512,227,319,241,448,201,214,454};
	double p2y[] = {107,213,421,274,154,23,316,185,139,302,276,246};

	CvPoint2D64f p1[numptpair];
	CvPoint2D64f p2[numptpair];
	CvPoint2D64f pt1, pt2, ptrans;
	CvPoint3D64f X[numptpair];

	CvPoint interesPt1[MAX_INTEREST_PT];
	CvPoint interesPt2[MAX_INTEREST_PT];
	CvPoint2D64f matchedPt1[MAX_INTEREST_PT];
	CvPoint2D64f matchedPt2[MAX_INTEREST_PT];
	CvPoint2D64f transPt1[MAX_INTEREST_PT];
	CvPoint2D64f transPt2[MAX_INTEREST_PT];
	CvPoint3D64f reconstX[MAX_INTEREST_PT];
	int numpt1, numpt2, matchednum;
	bool nextline;
	FILE *fid;
	int colorR, colorG, colorB;

	// Load images
	if(argc < 3){
		printf("Usage:\n");
		exit(0);
	}
	img1 = cvLoadImage(argv[1]);
	if(!img1){
		printf("Could not load image file: %s\n", argv[1]);
		exit(0);
	}
	img2 = cvLoadImage(argv[2]);
	if(!img2){
		printf("Could not load image file: %s\n", argv[2]);
		exit(0);
	}
	width  = img1->width;
	height = img1->height;

	// ****************************
	// estimate fundamental matrix F
	// ****************************
```

```
	// initialize point correspondences
	for(i=0; i<numptpair; i++){
		p1[i].x = p1x[i];
		p1[i].y = p1y[i];
		p2[i].x = p2x[i];
		p2[i].y = p2y[i];
	}
	EstFundamentalMatrix(numptpair, p1, p2, F);
	// Compute e e', P P'
	ComputeEpipoles(F,e,ep);
	ComputeCameraMatrices(F,ep,P,Pp);

	// output pts pairs and associated epipolar lines
	tmpimg1 = cvCloneImage(img1);
	tmpimg2 = cvCloneImage(img2);
	for(i=0; i<numptpair; i++){
		// for left image
		cvCircle(tmpimg1, cvPoint((int)p1[i].x, (int)p1[i].y), 2,
CV_RGB(0,255,255), 2, 8, 0);
		ComputeEpipolarLine(width, height, F, p2[i], 0, &pt1, &pt2);
		cvLine(tmpimg1, cvPoint((int)pt1.x,(int)pt1.y),
cvPoint((int)pt2.x,(int)pt2.y), CV_RGB(0,0,255), 1 , 8, 0 );
		// for right image
		cvCircle(tmpimg2, cvPoint((int)p2[i].x, (int)p2[i].y), 2,
CV_RGB(0,255,255), 2, 8, 0);
		ComputeEpipolarLine(width, height, F, p1[i], 1, &pt1, &pt2);
		cvLine(tmpimg2, cvPoint((int)pt1.x,(int)pt1.y),
cvPoint((int)pt2.x,(int)pt2.y), CV_RGB(0,0,255), 1 , 8, 0 );
	}
	cvSaveImage("ptpair1.jpg", tmpimg1);
	cvSaveImage("ptpair2.jpg", tmpimg2);
	// 3D reconstruction initialze
	Reconstruct3DPt(numptpair, p1, p2, P, Pp, X);

	// ******************************
	// LM optimization for 3n+12 paras
	// ******************************
	LMOptimizationPara(numptpair, p1, p2, Pp, X, e, ep, F);
	PrintCvMat(Pp);
	PrintCvMat(F);

	// ******************************
	// Image Rectification
	// i.e., map epipoles into (1,0,0)'
	// ******************************
	ImageRectification(width, height, numptpair, p1, p2, ep, P, Pp, H, Hp, F);
	tmpimg1 = cvCloneImage(img1);
	tmpimg2 = cvCloneImage(img2);
	Trans_Image(&img1, &tmpimg1, H);
	Trans_Image(&img2, &tmpimg2, Hp);
	cvSaveImage("Rectified1.jpg", tmpimg1);
	cvSaveImage("Rectified2.jpg", tmpimg2);
	for(i=0; i<numptpair; i++){
		// for left image
		TransformationByH(H,&p1[i],&ptrans);
		cvCircle(tmpimg1, cvPoint((int)ptrans.x, (int)ptrans.y), 2,
CV_RGB(0,255,255), 2, 8, 0);
		TransformationByH(Hp,&p2[i],&ptrans);
		ComputeEpipolarLine(width, height, F, ptrans, 0, &pt1, &pt2);
```

```
            cvLine(tmpimg1, cvPoint((int)pt1.x,(int)pt1.y),
cvPoint((int)pt2.x,(int)pt2.y), CV_RGB(0,0,255), 1 , 8, 0 );
            // for right image
            TransformationByH(Hp,&p2[i],&ptrans);
            cvCircle(tmpimg2, cvPoint((int)ptrans.x, (int)ptrans.y), 2,
CV_RGB(0,255,255), 2, 8, 0);
            TransformationByH(H,&p1[i],&ptrans);
            ComputeEpipolarLine(width, height, F, ptrans, 1, &pt1, &pt2);
            cvLine(tmpimg2, cvPoint((int)pt1.x,(int)pt1.y),
cvPoint((int)pt2.x,(int)pt2.y), CV_RGB(0,0,255), 1 , 8, 0 );
      }
      imgre = cvCreateImage(cvSize(2*width,height), IPL_DEPTH_8U, 3);
      ShowTwoImage(tmpimg1, tmpimg2, &imgre);
      cvSaveImage("ImageRectified.jpg", imgre);

      // ****************************
      // find point correspondences
      // then 3D reconstruction
      // ****************************
      // edge detection
      Trans_Image(&img1, &tmpimg1, H);
      Trans_Image(&img2, &tmpimg2, Hp);
      gimg = cvCreateImage(cvSize(width,height), IPL_DEPTH_8U, 1);
      cvCvtColor(tmpimg1, gimg, CV_BGR2GRAY);
      edgeimg1 = cvCreateImage(cvSize(width, height), IPL_DEPTH_8U, 1);
      edgeimg2 = cvCreateImage(cvSize(width, height), IPL_DEPTH_8U, 1);
      cvCanny(gimg, edgeimg1, 20, 100, 3);
      cvSaveImage("edge1.jpg", edgeimg1);
      cvCvtColor(tmpimg2, gimg, CV_BGR2GRAY);
      cvCanny(gimg, edgeimg2, 20, 100, 3);
      cvSaveImage("edge2.jpg", edgeimg2);

      fid = fopen("3Dpoints.txt","wt");
      cvInvert(H,invH);
      cvInvert(Hp,invHp);
      // exclude boundary pixels
      for(i=40; i<height-40; i++){
            // for each row
            nextline = false;
            numpt1 = 0;
            numpt2 = 0;
            for(j=40; j<width-40; j++){
                  if(cvGet2D(edgeimg1,i,j).val[0] == 255){
                        interesPt1[numpt1].x = j;
                        interesPt1[numpt1++].y = i;
                  }
                  if(cvGet2D(edgeimg2,i,j).val[0] == 255){
                        interesPt2[numpt2].x = j;
                        interesPt2[numpt2++].y = i;
                  }
            }
            //for(j=0; j<numpt1; j++){
            //    cvCircle(tmpimg1, cvPoint((int)interesPt1[j].x,
(int)interesPt1[j].y), 1, CV_RGB(255,0,0), 2, 8, 0);
            //}
            //for(j=0; j<numpt2; j++){
            //    cvCircle(tmpimg2, cvPoint((int)interesPt2[j].x,
(int)interesPt2[j].y), 1, CV_RGB(255,0,0), 2, 8, 0);
            //}
```

```
            // NCC matching
            matchednum = InterestedPointMatching_NCC(tmpimg1, tmpimg2, interesPt1,
numpt1, interesPt2, numpt2, matchedPt1, matchedPt2);
            for(j=0; j<matchednum-1; j++){
                  if(matchedPt2[j].x > matchedPt2[j+1].x){
                        nextline = true;
                        break;
                  }
            }
            if(nextline)
                  continue;

            for(j=0; j<matchednum; j++){
                  colorR = rand()%255;
                  colorG = rand()%255;
                  colorB = rand()%255;
                  cvCircle(tmpimg1, cvPoint((int)matchedPt1[j].x,
(int)matchedPt1[j].y), 1, CV_RGB(colorR,colorG,colorB), 2, 8, 0);
                  cvCircle(tmpimg2, cvPoint((int)matchedPt2[j].x,
(int)matchedPt2[j].y), 1, CV_RGB(colorR,colorG,colorB), 2, 8, 0);
            }

            // 3D reconstruction (save to file)
            // back to the original image
            for(j=0; j<matchednum; j++){
                  TransformationByH(invH,&matchedPt1[j],&transPt1[j]);
                  TransformationByH(invHp,&matchedPt2[j],&transPt2[j]);
            }
            // compute 3D world coordinate (up to a projective distortion)
            Reconstruct3DPt(matchednum, transPt1, transPt1, P, Pp, reconstX);
            // output 3D world point to file
            for(j=0; j<matchednum; j++){
                  fprintf(fid, "%f %f %f \n",reconstX[j].x, reconstX[j].y,
reconstX[j].z);
            }
      }
      cvSaveImage("tmp1.jpg", tmpimg1);
      cvSaveImage("tmp2.jpg", tmpimg2);

      // release
      fclose(fid);
      cvReleaseMat(&F);
      cvReleaseMat(&e);
      cvReleaseMat(&ep);
      cvReleaseMat(&P);
      cvReleaseMat(&Pp);
      cvReleaseMat(&H);
      cvReleaseMat(&Hp);
      cvReleaseMat(&invH);
      cvReleaseMat(&invHp);
      cvReleaseImage(&img1);
      cvReleaseImage(&img2);
      cvReleaseImage(&tmpimg1);
      cvReleaseImage(&tmpimg2);
      cvReleaseImage(&gimg);
      cvReleaseImage(&edgeimg1);
      cvReleaseImage(&edgeimg2);
      cvReleaseImage(&imgre);
      return 0;
}
```