Computer Vision
Homework #4

Data: Oct. 17, 2006

Donghun Kim (Dave Kim)
Robot Vision Lab

1. Problem Statement

Remove the outliers using by RANSAC robust estimation and Compute the homography with all correspondences classified as inliers. And Test image mosaicing using the homography.

2. Solving Methods

A. Corner Detection

i. Computer x and y derivatives of image (ex. Sobel operator)

$$I_x = G_\sigma^x * I \ , \quad I_y = G_\sigma^y * I$$

ii. Compute products of derivatives at every pixel

$$I_{x2} = I_x \cdot I_x \ , \quad I_{xy} = I_x \cdot I_y \ , \quad I_{y2} = I_y \cdot I_y$$

iii. Compute the sum of the products of derivatives at each pixel (Here, the size of W is 7 by 7.)

$$S_{x2} = G_W * I_{x2} \ , \quad S_{xy} = G_W * I_{xy} \ , \quad S_{y2} = G_W * I_{y2}$$

iv. Define at each pixel (x,y) the matrix

$$H(x,y) = \begin{bmatrix} S_{x2}(x,y) & S_{xy}(x,y) \\ S_{xy}(x,y) & S_{y2}(x,y) \end{bmatrix}$$

v. Compute the response(corner_value, R) of the detector at each pixel

① Harris's Equation (MODE_HARRIS_METHOD)

$R = Det(H) - k(Trace(H))^2$ where k =0.40

② Smallest Eigenvalue (MODE_EIGENVALUE_METHOD)

$R = the\ smallest\ eigenvalue$

vi. Threshold on value of R base on THRES_CORNERNESS or THRES_ EIGENVALUE_CORNERNESS

B. Matching using Normalized Cross Correlation

$$NCC(u,v) = \frac{\sum_{x,y}[f(x,y) - \overline{f}_{u,v}][t(x-u,y-v) - \overline{t}]}{\sqrt{\sum_{x,y}[f(x,y) - \overline{f}_{u,v}]^2 \sum_{x,y}[t(x-u,y-v) - \overline{t}]^2}}$$

Where $t(x,y)$ is the pixel value of the template image and $f(x,y)$ is the pixel value of the input image.

C. RANSAC robust estimation

i. Start from Putative correspondences by NCC

ii. Repeat for N samples, where N is determined adaptively.

① Select a random sample of 4 correspondences and compute the homography H with data normalization.

② Data Normalization

- Normalization of x : Compute similar transformation T to transform points x_i to $\tilde{x}_i$
- Normalization of x' : Compute similar transformation T' to transform points x_i' to $\tilde{x}_i'$
- DLT: Obtain a homography $\tilde{H}$
- Denormalization: Set $H = T'^{-1}\tilde{H}T$

③ Calculate the distance d from each putative correspondence.

$$d = d(x_i, \hat{x}_i)^2 + d(x_i', \hat{x}_i')^2$$

④ Compute the number of inliers consistent with H by the number of correspondence for which d < THRES_INLIER. (Set to 100)

iii. Choose the H with the largest number of inliers.

iv. Optimal estimation

① Re-estimate H from all correspondences classified as inliers by DLT.

v. Guided matching : Image Mosaicing

3. Test Results

A. Blue lines and yellow lines are detected by NCC which can make the outliers.

B. Yellow lines are inliers obtained by RANSAC.

C. Test 1 is to remove the outliers which can match by NCC

D. Test 2 is to give the image mosaicing using by a homography.

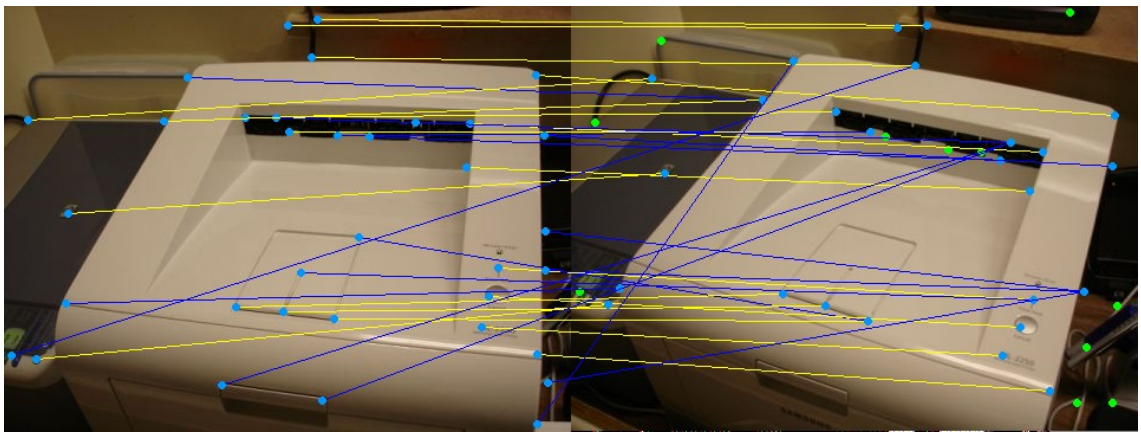

Fig 1. Test result for removing the outliers by RANSAC

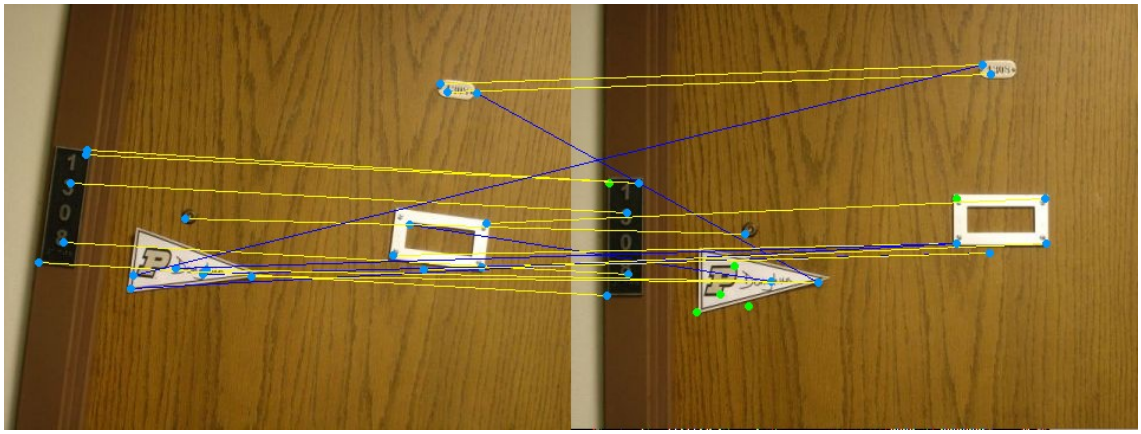

Fig 2. Test result for removing the outliers by RANSAC

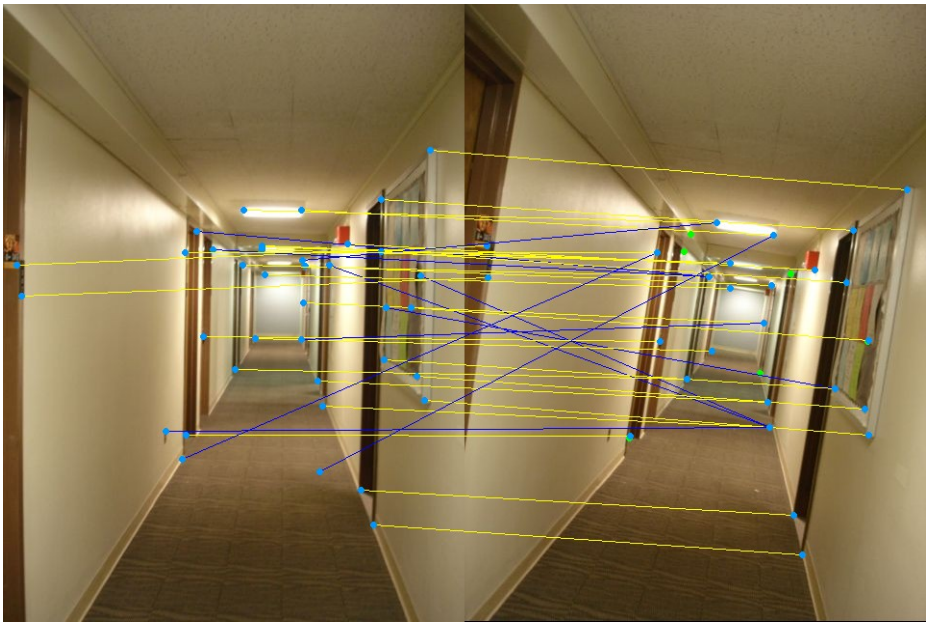

Fig 3. Test result for removing the outliers by RANSAC

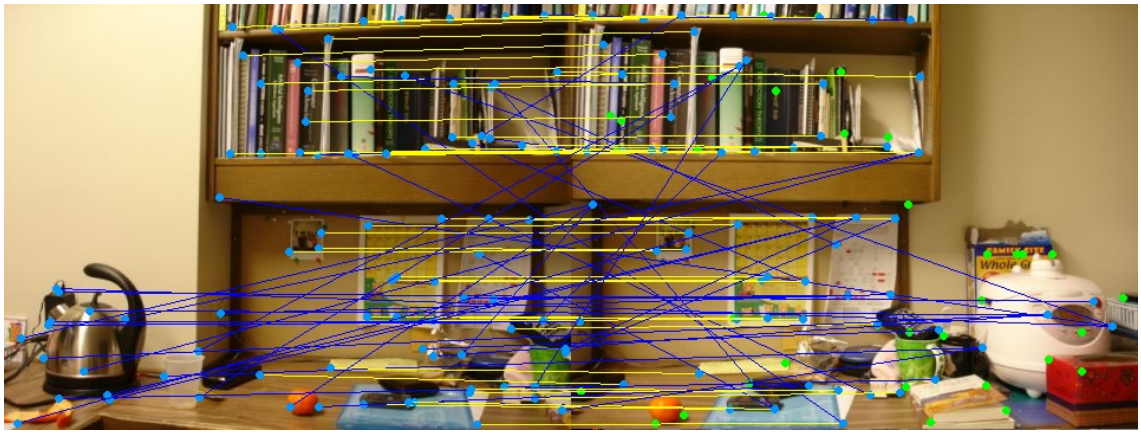

Fig 4. (a) Test result for removing the outliers by RANSAC

(b) Image Mosaicing by the homography

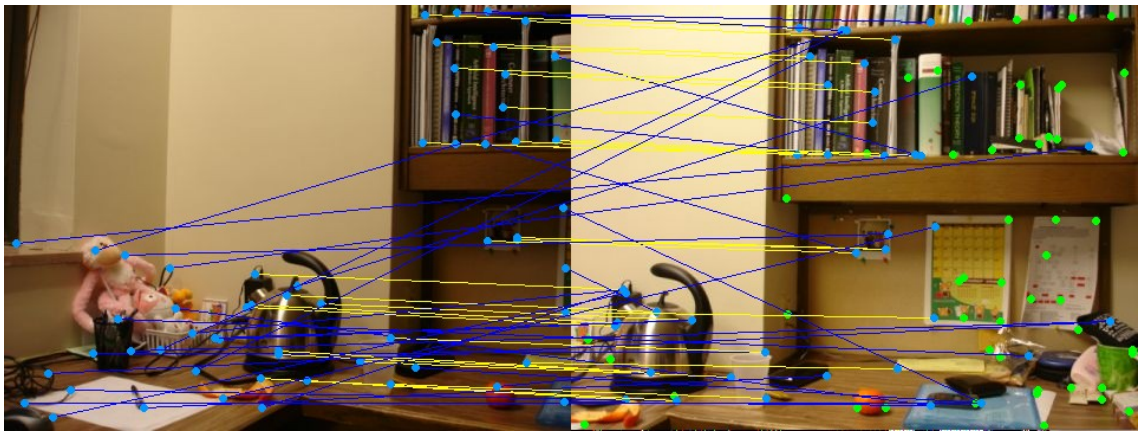

Fig 5. (a) Test result for removing the outliers by RANSAC

(b) Image Mosaicing by the homography

4. Source Code

// Main souce File

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#include "cv.h"
#include "cxcore.h"
#include "highgui.h"
#include "Homography.h"


void main(int argc, char *argv[])
{
	int num, mode, ox_size, oy_size, i,k, j;
	IplImage *ginImg1, *ginImg2, *wimage;
	CORNERINFO cnData, cnData2;
	MATPAIR cornerpair;
	IplImage *resultImg=0;
	IplImage *image, *image2;

	printf("Welcome to zava's Program.₩n");
	printf("========= Homework #4 ============₩n");
	printf("===== Automatic Homography ========₩n");
```

```
image     = cvLoadImage(".₩₩ImageData₩₩D2.jpg",-1);
image2    = cvLoadImage(".₩₩ImageData₩₩D3.jpg",-1);

ox_size=image->width;   oy_size=image->height;
ginImg1 = cvCreateImage(cvSize(ox_size,oy_size),IPL_DEPTH_8U,1);
ginImg2 = cvCreateImage(cvSize(ox_size,oy_size),IPL_DEPTH_8U,1);
resultImg= cvCreateImage(cvSize(ox_size,oy_size),IPL_DEPTH_8U,3);

// Convert color images to gray-level images and Smooth for damping noise
cvCvtColor(image,ginImg1,CV_BGR2GRAY);
cvCvtColor(image2,ginImg2,CV_BGR2GRAY);
cvSmooth(ginImg1, ginImg1, CV_GAUSSIAN, 3, 3, 0);
cvSmooth(ginImg2, ginImg2, CV_GAUSSIAN, 3, 3, 0);

// Extract corners for each image and Match the pairs
mode = MODE_EIGENVALUE_METHOD;     // or MODE_HARRIS_METHOD
NCC(ginImg1, ginImg2, &cnData, &cnData2, &cornerpair, mode, THRES_NCC);

// Robust Estimation using RANSAC
CvMat   *H= cvCreateMat(3,3,CV_MAT32F);
CvMat   *maxH= cvCreateMat(3,3,CV_MAT32F);
CvMat   *Ht= cvCreateMat(3,3,CV_MAT32F);
CvMat   *Htmp= cvCreateMat(3,3,CV_MAT32F);
CvMat   *T1= cvCreateMat(3,3,CV_MAT32F);
CvMat   *T2= cvCreateMat(3,3,CV_MAT32F);
CvMat   *T2_Inv= cvCreateMat(3,3,CV_MAT32F);
FLPOINT pin[MAX_CORNER_POINTS], mpin[MAX_CORNER_POINTS];

int       snum, inlierNo, max_inlier, loopNo, cnt, flag[MAX_CORNER_POINTS];
int       pos, coflag;
float     p,e, ProbOutlier;
uchar     *wdata, *idata, *idata2;
CvPoint Ptmp;

max_inlier=cnt=0;
```

```
loopNo=1000;    e =0.5;  p=0.99;
snum =4;
srand( (unsigned)time( NULL ) );

while(loopNo >cnt)
{
        //-------------- RANSAC ----------------------------//
        coflag=1;
        while(coflag)
        {
                for(i=0;i<4;i++)
                {
                        pos = rand() % cornerpair.matNo;
                        pin[i].x = cornerpair.cp[pos].x;
                        pin[i].y = cornerpair.cp[pos].y;
                        mpin[i].x = cornerpair.mp[pos].x;
                        mpin[i].y = cornerpair.mp[pos].y;
                }

                if( CheckColinearity(pin, snum)!=1 )
                        coflag=0;
        }
        Normalize(pin, snum, T1);
        Normalize(mpin, snum, T2);
        MakeHomographyMat(pin, mpin, snum, Ht);
        cvInvert(T2, T2_Inv);
        cvMatMul(T2_Inv,Ht,Htmp);
        cvMatMul(Htmp,T1,H);
        //-----------------------------------------------------//
        // Check inlier using the Estimated Homography
        EstimateInlier(&cornerpair, H, &inlierNo);

        if(max_inlier < inlierNo)
        {
                max_inlier=inlierNo;
                cvCopy(H, maxH, 0);
```

```
                memset(flag,0,MAX_CORNER_POINTS);
                for(k=0;k<cornerpair.matNo;k++)
                        flag[k]=cornerpair.flag[k];
        }

        ProbOutlier=1-(max_inlier/cornerpair.matNo);
        e = (e<ProbOutlier) ? e:ProbOutlier;
        loopNo = log(1-p)/log(1-pow((1-e),4));
        cnt++;
}

// Update as the maximum number of inliers
for(k=0,cnt=0;k<cornerpair.matNo;k++)
{
        cornerpair.flag[k]=flag[k];
        if(flag[k] ==1 )
        {
                cnt++;
                pin[cnt].x = cornerpair.cp[k].x;
                pin[cnt].y = cornerpair.cp[k].y;
                mpin[cnt].x = cornerpair.mp[k].x;
                mpin[cnt].y = cornerpair.mp[k].y;
        }
}

// Reestimate the Homography with inliers
MakeHomographyMat(pin, mpin, cnt, H);

// For Displaying the mosaicing image
wimage = cvCreateImage(cvSize(ox_size*2, oy_size),IPL_DEPTH_8U,3);
cvNamedWindow("Mosaicing", CV_WINDOW_AUTOSIZE);
ImageMosaicing(wimage, image, image2, H);
cvShowImage("Mosaicing",wimage);
cvSaveImage("hw4_mosaicing.jpg",wimage);

// For the display ------------------------------------------//
```

```
wdata   = (uchar *)wimage->imageData;
idata   = (uchar *)image->imageData;
idata2  = (uchar *)image2->imageData;

cvNamedWindow("Mapping Result", CV_WINDOW_AUTOSIZE);
for(i=0;i<oy_size;i++)
for(j=0;j<2*ox_size;j++)
{
        if(j<ox_size)
        {
                for(k=0;k<3;k++)
                    wdata[i*wimage->widthStep+j*wimage->nChannels+k] =
                        idata[i*image->widthStep+j*image->nChannels+k];
        }
        else
        {
                for(k=0;k<3;k++)
                    wdata[i*wimage->widthStep+j*wimage->nChannels+k] =
                        idata2[i*image->widthStep+j*image->nChannels+k];
        }
}

for(i=0;i<cnData.num;i++)
        cvCircle(wimage, cnData.cp[i], 2, CV_RGB(255,0,0), 2, 8, 0);
for( i=0 ; i<cnData2.num; i++)
{
        Ptmp.x=cnData2.cp[i].x+ox_size;
        Ptmp.y=cnData2.cp[i].y;
        cvCircle(wimage, Ptmp, 2, CV_RGB(0,255,0), 2, 8, 0);
}
for(i=0;i<cornerpair.matNo;i++)
{
        Ptmp.x = cornerpair.mp[i].x+ox_size;
        Ptmp.y = cornerpair.mp[i].y;

        if(cornerpair.flag[i]==1)
```

```
            {
                cvLine(wimage, cornerpair.cp[i], Ptmp,
                    CV_RGB(255,255,0),1,8,0);
            }
            else
            {
                cvLine(wimage, cornerpair.cp[i], Ptmp,
                 CV_RGB(0,0,255),1,8,0);
            }
            cvCircle(wimage, cornerpair.cp[i], 2, CV_RGB(0,150,255),2,8,0);
            cvCircle(wimage, Ptmp, 2, CV_RGB(0,150,255),2,8,0);
    }

    cvShowImage("Mapping Result",wimage);
    cvSaveImage("hw4_ransac.jpg",wimage);
    //-------------------------------------------------------------//

    cvWaitKey(0);

    cvDestroyWindow("Mosaicing");
    cvDestroyWindow("Mapping Result");
    cvReleaseMat(&Ht);
    cvReleaseMat(&Htmp);
    cvReleaseMat(&T1);
    cvReleaseMat(&T2);
    cvReleaseMat(&T2_Inv);
    cvReleaseImage(&wimage);
    cvReleaseImage(&resultImg);
    cvReleaseImage(&ginImg1);
    cvReleaseImage(&ginImg2);
}
```

// Source File for Corner Detection and NCC - “Homography.c”

```
#include <stdio.h>
#include <stdlib.h>
#include "cv.h"
```

```
#include "cxcore.h"
#include "highgui.h"
#include "Homography.h"

//-----------------------------------------------------------------//
// Function : CornerDetection
// IN           : int width, height
//                uchar *gin
//                int mode   ->   selcec Harris' eq. method or Eigenvale method
// OUT   : CvPoint *corners     -> Information of corner points
//           int *num                        -> The number of corner points
// Description:
//           Detect Corner using sobel operator.
//-----------------------------------------------------------------//
int CornerDetection(int width, int height, uchar *gin, CvPoint *corners,
                        int *num, int mode)
{
    int       i,j,k,m;
    int       ffsize, cnt, ret;
    double  *Gx, *Gy, CornerMap[MAX_CORNER_POINTS]={0};
    double  Gx2_sum, Gy2_sum, Gxy_sum;
    double  threshold, corner_value=0.0;
    int       flag=0;
    CvMat   *Cor = cvCreateMat(2,2,CV_MAT32F);
    CvMat   *D = cvCreateMat(2,2,CV_MAT32F);
    CvMat   *U = cvCreateMat(2,2,CV_MAT32F);
    CvMat   *V = cvCreateMat(2,2,CV_MAT32F);

    ffsize   = sizeof(double)*width*height;

    Gx = (double *)malloc(ffsize);
    Gy = (double *)malloc(ffsize);

    memset(Gx, 0, ffsize);
    memset(Gy, 0, ffsize);
```

```
// Find gradient of x & y axis
Sobel(width, height, gin, Gx, Gy);

cnt=0;
for(i=0;i<height;i++)
for(j=0;j<width;j++)
{
	Gx2_sum = Gy2_sum = corner_value= Gxy_sum = 0.0;

	for(k=-WS_CD;k<=WS_CD;k++)
	for(m=-WS_CD;m<=WS_CD;m++)
	{
		if((i+k)<0 || (i+k)>=height || (j+m)<0 || (j+m)>=width)
			continue;
	 Gx2_sum += Gx[(i+k)*width+(j+m)]*Gx[(i+k)*width+(j+m)]/10000;
	 Gy2_sum += Gy[(i+k)*width+(j+m)]*Gy[(i+k)*width+(j+m)]/10000;
	 Gxy_sum += Gx[(i+k)*width+(j+m)]*Gy[(i+k)*width+(j+m)]/10000;
	}

	// Method 1: Harris's eq. Method
	if(mode ==1)
	{
		corner_value = (Gx2_sum*Gy2_sum - Gxy_sum*Gxy_sum)
			- KAPA*(Gx2_sum+Gy2_sum)*(Gx2_sum+Gy2_sum);
		threshold = THRES_CORNERNESS;
	}
	// Method 2: Eigenvalue Method
	else
	{
		Cor->data.fl[0]=Gx2_sum;
		Cor->data.fl[1]=Gxy_sum;
		Cor->data.fl[2]=Gxy_sum;
		Cor->data.fl[3]=Gy2_sum;

		cvSVD(Cor, D, U, V,0);
		corner_value = D->data.fl[3];
```

```
                    threshold = THRES_EIGENVALUE_CORNERNESS;
            }
            if(i> (WS_CD+ 1) && i<height-(WS_CD+ 1)
                    && j>(WS_CD+ 1) && j<width-(WS_CD+ 1))
                    ret=VerifyCorner(j,i, &cnt, corners, CornerMap,
                                        corner_value, threshold);
    }

    (*num)= cnt;

    free(Gx);
    free(Gy);
    cvReleaseMat(&D);
    cvReleaseMat(&U);
    cvReleaseMat(&V);
    cvReleaseMat(&Cor);

    return (cnt);
}

//-----------------------------------------------------------------//
// Function : VerifyCorner
// IN          : int cx, cy        -> A corner point for verifying
//               int mode          -> Selcec Harris' eq. method or Eigenvale method
//               double cvalue  -> Corner-value to check
//               int vc_th         -> Threshold of a cornerness
// OUT       : double *CornerMap      -> Information of corner points
//               int *cornerNo             -> The number of corner points
// Description:
//    Verify that another corner is existed nearby. If there are corners
//    nearby, then choose a point with the highest corner-value.
//-----------------------------------------------------------------//
int VerifyCorner(int cx, int cy, int *cornerNo, CvPoint *corners,
                                    double *CornerMap, double cvalue, int vc_th)
{
    int i, px, py, dist, flag, cnt;
```

```
cnt = *cornerNo;
if(cvalue > vc_th )
{
	// verify that there are previous found corners nearby.
	flag=0;
	for(i=0;i<cnt;i++)
	{
		px=corners[i].x;py=corners[i].y;
		dist = sqrt((px-cx)*(px-cx)+(py-cy)*(py-cy));
		if(dist < THRES_VC_DIST)
		{
			if( CornerMap[i] < cvalue)
			{
				corners[i].x = cx;
				corners[i].y = cy;
				CornerMap[i]= cvalue;
			}

			flag=1;
			break;
		}
	}
	if(flag==0)
	{
		corners[cnt].x = cx;
		corners[cnt].y = cy;
		if(cnt >= MAX_CORNER_POINTS)
		{
			printf("over count!!");
		}
		else
		{
			CornerMap[cnt]=cvalue;
			cnt++;
		}
```

```
                (*cornerNo)= cnt;
                return 2;
        }

        return 1;
    }
    else
        return 0;
}

int Sobel( int xs, int ys, uchar *inImg, double *gx, double *gy)
{
    int dx[9] = {-1, 0, 1, -1, 0 , 1, -1, 0, 1};
    int dy[9] = {-1, -1, -1, 0, 0, 0, 1, 1, 1};
    int xmask[9] = {-1, 0, 1, -2, 0, 2, -1, 0, 1};
    int ymask[9] = {1, 2, 1, 0, 0, 0, -1, -2, -1};
    int lgx,lgy, i, j, k;

    for(i=0;i<ys; i++)
    for(j=0;j<xs;j++)
    {
        lgx=lgy=0;
        for(k=0;k<9;k++)
        {
            if((i+dy[k])<0 || (i+dy[k])>=ys || (j+dx[k])<0 ||
                (j+dx[k])>=xs)  continue;
            lgx += xmask[k]*inImg[(i+dy[k])*xs+j+dx[k]];
            lgy += ymask[k]*inImg[(i+dy[k])*xs+j+dx[k]];
        }
        lgx=abs(lgx);
        lgy=abs(lgy);
        if(lgx>255)     lgx=255;
        if(lgy>255)     lgy=255;
        gx[i*xs+j]= (double)lgx;
        gy[i*xs+j]= (double)lgy;
    }
```

```
    return 0;
}
//-------------------------------------------------------------//
// Function  : NCC
// IN        :  IplImage *ginImg1, *ginImg2      -> Two input images for matching
//              int mode       ->   select Harris' eq. method or Eigenvalue method
// OUT       :  CORNERINFO *corners1, *corners2 -> corner information
//              MATPAIR *cornerpair              -> Information of Matching pairs
// Description:
//              Normalized Cross-Correlation
//-------------------------------------------------------------//
int NCC(IplImage *ginImg1, IplImage *ginImg2, CORNERINFO *corners1,
        CORNERINFO *corners2, MATPAIR *cornerpair, int mode)
{
    Int       xs, ys,i,j, k, m;
    int       fnum, tnum, fx, fy, tx, ty, cnt, mcnt;
    float     fmean, tmean, numer, fsum, tsum, ncc_coef, max_val;
    uchar     *fdata, *tdata;
    CvPoint   maxpoint;

    xs= ginImg1->width;              ys= ginImg1->height;

    fdata=(uchar*)ginImg1->imageData;
    tdata=(uchar*)ginImg2->imageData;

    fnum=tnum=mcnt=0;
    CornerDetection(xs, ys, fdata, corners1->cp, &fnum, mode);
    CornerDetection(xs, ys, tdata, corners2->cp, &tnum, mode);
    corners1->num = fnum;
    corners2->num = tnum;
    cornerpair->num = fnum;

    for(i=0;i<fnum;i++)
    {
        fx = corners1->cp[i].x;
```

```
fy = corners1->cp[i].y;

max_val=-1;
for(j=0;j<tnum;j++)
{
        tx = corners2->cp[j].x;
        ty = corners2->cp[j].y;

        // Normalized Cross-corellation
        fmean= tmean=0.0; cnt=0;
        for(k=-WS_NCC; k<=WS_NCC;k++)
        for(m=-WS_NCC; m<=WS_NCC;m++)
        {
                if((fy+k)<0 || (fy+k)>=ys || (fx+m)<0 ||
                   (fx+m)>=xs)              continue;
                fmean += fdata[(fy+k)*xs+(fx+m)];
                tmean += tdata[(ty+k)*xs+(tx+m)];
                cnt++;
        }
        fmean /= cnt;     tmean /= cnt;
        fsum= tsum = numer= 0.0;
        for(k=-WS_NCC; k<=WS_NCC;k++)
        for(m=-WS_NCC; m<=WS_NCC;m++)
        {
                if((fy+k)<0 || (fy+k)>=ys || (fx+m)<0 ||
                        (fx+m)>=xs)      continue;
                numer += (fdata[(fy+k)*xs+(fx+m)] - fmean)
                        *(tdata[(ty+k)*xs+(tx+m)] - tmean);
                fsum  += (fdata[(fy+k)*xs+(fx+m)] - fmean)
                        *(fdata[(fy+k)*xs+(fx+m)] - fmean);
                tsum  += (tdata[(ty+k)*xs+(tx+m)] - tmean)
                        *(tdata[(ty+k)*xs+(tx+m)] - tmean);
        }
        ncc_coef = numer/ sqrt(fsum*tsum);
        if(ncc_coef>max_val)
        {
```

```
                                max_val=ncc_coef;
                                maxpoint.x = tx;
                                maxpoint.y = ty;
                        }
                }
                cornerpair->ncc_val[i]=max_val;
                cornerpair->mp[i].x = maxpoint.x;
                cornerpair->mp[i].y = maxpoint.y;
                cornerpair->cp[i].x = fx;
                cornerpair->cp[i].y = fy;
        }
        cornerpair->matNo = mcnt;
        return 0;
}
```

// Source File for Robust Estimation with RANSAC - “Homography.c”

```
//-------------------------------------------------------------------//
// Function : MakeHomographyMat
// IN           :          FLPOINT *pin    ->        n points
//                         int num                   ->        n
// OUT    :                CvMat *matout   ->        3x3 Matrix
// Description:
//                  Make 3x3 homography matrix from 2n x 9 matrix A
//-------------------------------------------------------------------//
int MakeHomographyMat(FLPOINT *pin, FLPOINT *cpin, int num, CvMat *matout)
{

        int i, j, index;
        CvMat *A = cvCreateMat(2*num, 9, CV_MAT32F);
        CvMat *U = cvCreateMat(2*num, 2*num, CV_MAT32F);
        CvMat *D = cvCreateMat(2*num, 9, CV_MAT32F);
        CvMat *V = cvCreateMat(9, 9, CV_MAT32F);

        for(i=0;i<num;i++)
        {
                index= i*18;
```

```
            A->data.fl[index+0]  = 0;
            A->data.fl[index+1]  = 0;
            A->data.fl[index+2]  = 0;
            A->data.fl[index+3]  = -1*pin[i].x;
            A->data.fl[index+4]  = -1*pin[i].y;
            A->data.fl[index+5]      = -1;
            A->data.fl[index+6]  = pin[i].x*cpin[i].y;
            A->data.fl[index+7]  = pin[i].y*cpin[i].y;
            A->data.fl[index+8]  = cpin[i].y;

            A->data.fl[index+9]  = pin[i].x;
            A->data.fl[index+10] = pin[i].y;
            A->data.fl[index+11] = 1;
            A->data.fl[index+12] = 0;
            A->data.fl[index+13] = 0;
            A->data.fl[index+14] = 0;
            A->data.fl[index+15] = -1*pin[i].x*cpin[i].x;
            A->data.fl[index+16] = -1*pin[i].y*cpin[i].x;
            A->data.fl[index+17] = -1*cpin[i].x;
    }

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

    for(i=0;i<9; i++)
    {
            matout->data.fl[i] = cvmGet(V, 9-1,i);
    }

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

    return 0;
}
```

```
//-----------------------------------------------------------------//
// Function : Normalize
// IN          :          FLPOINT *in              ->        n points
//                        int num                  ->        n
// OUT   :    CvMat *T                  ->        3x3 Similarity Matrix
// Description:
//                 Normalize about n points
//-----------------------------------------------------------------//
int Normalize(FLPOINT *in, int num, CvMat *T)
{
    int i;
    float tx, ty, dist, distSum, scale;

    tx=ty=0;
    for(i=0;i<num;i++)
    {
            tx += in[i].x;
            ty += in[i].y;
    }
    tx/=num;          ty/=num;

    distSum=0;
    for(i=0;i<num;i++)
    {
            dist = pow(tx-in[i].x, 2) +  pow(ty-in[i].y, 2);
            distSum += sqrt(dist);
    }
    distSum /= num;

    scale = sqrt(2)/distSum;

    T->data.fl[0] = scale;
    T->data.fl[1] = 0;
    T->data.fl[2] = -scale*tx;
    T->data.fl[3] = 0;
    T->data.fl[4] = scale;
```

```
    T->data.fl[5] = -scale*ty;
    T->data.fl[6] = 0;
    T->data.fl[7] = 0;
    T->data.fl[8] = 1;

    // data normalization
    CvMat *op = cvCreateMat(3,1,CV_MAT32F);
    CvMat *np = cvCreateMat(3,1,CV_MAT32F);
    for(i=0; i< num; i++)
    {
        op->data.fl[0] = in[i].x;
        op->data.fl[1] = in[i].y;
        op->data.fl[2] = 1;
        cvMatMul(T, op, np);
        in[i].x = np->data.fl[0]/np->data.fl[2];
        in[i].y = np->data.fl[1]/np->data.fl[2];
    }
    cvReleaseMat(&op);
    cvReleaseMat(&np);

    return 0;
}
//-------------------------------------------------------------//
// Function : EstimateInlier
// IN   :   MATPAIR *cornerpair   ->   n points
//          CvMat *H        ->   Given a Homography
// OUT  :   int *inlierNum         ->   The number of inliers
// Description:
//              Estimate the inliers only using a homography
//-------------------------------------------------------------//
int EstimateInlier(MATPAIR *cornerpair, CvMat *H, int *inlierNum)
{
    CvMat  *pin = cvCreateMat(3,1,CV_MAT32F);
    CvMat  *pout = cvCreateMat(3,1,CV_MAT32F);
    CvMat  *H_Inv = cvCreateMat(3,3,CV_MAT32F);
```

```
double   px1, py1, px2, py2, diff, diff1, diff2;
int                  i, cnt=0;

cvInvert(H, H_Inv);

for(i=0;i<cornerpair->matNo;i++)
{
        cornerpair->flag[i]=0;
        pin->data.fl[0]=cornerpair->cp[i].x;
        pin->data.fl[1]=cornerpair->cp[i].y;
        pin->data.fl[2]=1;
        cvMatMul(H,pin,pout);
        px1 = (pout->data.fl[0])/(pout->data.fl[2]);
        py1 = (pout->data.fl[1])/(pout->data.fl[2]);
        diff1= pow(px1-cornerpair->mp[i].x, 2)+
                        pow(py1-cornerpair->mp[i].y, 2);

        pin->data.fl[0]=cornerpair->mp[i].x;
        pin->data.fl[1]=cornerpair->mp[i].y;
        pin->data.fl[2]=1;
        cvMatMul(H_Inv,pin,pout);
        px2 = (pout->data.fl[0])/(pout->data.fl[2]);
        py2 = (pout->data.fl[1])/(pout->data.fl[2]);
        diff2= pow(px2-cornerpair->cp[i].x, 2)+
                        pow(py2-cornerpair->cp[i].y, 2);

        diff = diff1+ diff2;

        if( diff < THRES_INLIER )
        {
                cornerpair->flag[i] = 1;
                cnt++;
        }
}

(*inlierNum) = cnt;
```

```
    cvReleaseMat(&H_Inv);
    cvReleaseMat(&pout);
    cvReleaseMat(&pin);

    return 0;
}
//-------------------------------------------------------------------//
// Function : CheckColinearity
// IN    :  FLPOINT *pin   ->      n points
//          int num                ->      n
// OUT   :  return value    ->      colinear: 1, non-colinear: 0
// Description:
//              Check the colnearity with 3 points
//-------------------------------------------------------------------//
int CheckColinearity(FLPOINT *pin, int num)
{
    CvMat *m1 = cvCreateMat(3,1, CV_MAT32F);
    CvMat *m2 = cvCreateMat(3,1, CV_MAT32F);
    CvMat *m3 = cvCreateMat(3,1, CV_MAT32F);
    CvMat *mt = cvCreateMat(3,1, CV_MAT32F);
    float ft=0;
    int i,j,k;

    for(i=0;i<num-2;i++)
    for(j=i+1;j<num-1;j++)
    for(k=j+1;k<num;k++)
    {
        m1->data.fl[0]= pin[i].x;
        m1->data.fl[1]= pin[i].y;
        m1->data.fl[2]= 1;
        m2->data.fl[0]= pin[j].x;
        m2->data.fl[1]= pin[j].y;
        m2->data.fl[2]= 1;
        m3->data.fl[0]= pin[k].x;
        m3->data.fl[1]= pin[k].y;
```

```
                m3->data.fl[2]= 1;
                cvCrossProduct(m1,m2,mt);
                ft=cvDotProduct(mt,m3);

                if( abs(ft) ==0 )
                {
                        cvReleaseMat(&mt);
                        cvReleaseMat(&m3);
                        cvReleaseMat(&m2);
                        cvReleaseMat(&m1);
                        return 1;
                }
        }

        cvReleaseMat(&m3);
        cvReleaseMat(&m2);
        cvReleaseMat(&m1);
        return 0;
}
//-----------------------------------------------------------------//
// Function : ImageMosaicing
// IN      :    IplImage *image          ->      domain image
//              IplImage *image2         ->      range image
//              CvMat *H                 ->      Homography
// OUT    :    IplImage *wimage         ->      mosaicing image
// Description:
//                 Mosaic two images by a homography
//-----------------------------------------------------------------//
int ImageMosaicing(IplImage *wimage,IplImage *image,IplImage *image2,CvMat *H)
{
        int     ox_size, oy_size, i, j, k;
        int     step, channels, wstep, wchannels, px, py;
        uchar   *wdata, *idata, *idata2;
        CvMat   *pi = cvCreateMat(3,1, CV_MAT32F);
        CvMat   *po = cvCreateMat(3,1, CV_MAT32F);
        CvMat   *H_inv= cvCreateMat(3,3,CV_MAT32F);
```

```
ox_size = image->width;                    oy_size=image->height;
wstep   = wimage->widthStep;               wchannels = wimage->nChannels;
step    = image->widthStep;                channels = image->nChannels;
wdata   = (uchar *)wimage->imageData;
idata   = (uchar *)image->imageData;
idata2  = (uchar *)image2->imageData;

cvInvert(H, H_inv);

for(i=0;i<oy_size;i++)
for(j=0;j<2*ox_size;j++)
{
        if(j<ox_size)
        {
                for(k=0;k<3;k++)
                        wdata[i*wstep+j*wchannels+k] =
                                idata[i*step+j*channels+k];
        }
}

for(i=0;i<oy_size;i++)
for(j=0;j<ox_size;j++)
{
        pi->data.fl[0] = j;
        pi->data.fl[1] = i;
        pi->data.fl[2] = 1;
        cvMatMul(H_inv, pi, po);
        px = po->data.fl[0]/po->data.fl[2];
        py = po->data.fl[1]/po->data.fl[2];

        if(px>=2*ox_size || py>= oy_size || px<0 || py<0)
                continue;
        for(k=0;k<3;k++)
                wdata[py*wstep+px*wchannels+k] =
                                idata2[i*step+j*channels+k];
}
```

```
    // backprojection for interpolation
    for(i=0;i<oy_size;i++)
    for(j=0;j<2*ox_size;j++)
    {
            if(wdata[i*wstep+j*wchannels] == 0 &&
                wdata[i*wstep+j*wchannels+1] == 0   &&
                    wdata[i*wstep+j*wchannels+2] == 0)
            {
                    pi->data.fl[0]=j;
                    pi->data.fl[1]=i;
                    pi->data.fl[2]=1;
                    cvMatMul(H, pi, po);
                    px= po->data.fl[0]/po->data.fl[2];
                    py= po->data.fl[1]/po->data.fl[2];

                    if(px>=ox_size || px<0) continue;
                    if(py>=oy_size || py<0) continue;

                    for(k=0;k<3;k++)
                            wdata[i*wstep+j*wchannels+k] =
                                    idata2[py*step+px*channels+k];
            }
    }
    cvReleaseMat(&H_inv);
    cvReleaseMat(&pi);
    cvReleaseMat(&po);

    return 0;
}
```

// Header File for Corner Detection and NCC and Robust Estimation by RANSAC - “Homography.h”

```
#ifndef HOMOGRAPHY_H
#define HOMOGRAPHY_H

#define MODE_HARRIS_METHOD              1
```

```
#define MODE_EIGENVALUE_METHOD      2

// Definition for Extraction
#define WS_CD                       3
#define THRES_VC_DIST               25

#define KAPA                        0.04
#define THRES_CORNERNESS            50 //180
#define THRES_SVD_CORNERNESS        4//10//6

// Definition for Matching
#define THRES_NCC                   0.01//0.8//0.63
#define WS_NCC                      15//35
#define MAX_CORNER_POINTS           10000

#define NUM_ITERATION               50//50
#define THRES_INLIER                100//36   // 5x5
#define THRES_COLINEAR              0.5//0.4

typedef struct _MATPAIR
{
    int num;
    int matNo;
    int flag[MAX_CORNER_POINTS];
    CvPoint cp[MAX_CORNER_POINTS];
    CvPoint mp[MAX_CORNER_POINTS];
    float ncc_val[MAX_CORNER_POINTS];
} MATPAIR;

typedef struct _CORNER_INFO
{
    int num;
    CvPoint cp[MAX_CORNER_POINTS];
} CORNERINFO;
typedef struct _FLPOINTSET
{
```

```
    float x;
    float y;
} FLPOINT;

int Sobel( int xs, int ys, uchar *inImg, double *gx, double *gy);
int CornerDetection(int width, int height, uchar *gin, CvPoint *corners, int *num, int mode);
int VerifyCorner(int cx, int cy, int *cornerNo, CvPoint *corners, double *CornerMap, double cvalue, int vc_th);
int NCC(IplImage *ginImg1, IplImage *ginImg2, CORNERINFO *corners1, CORNERINFO *corners2, MATPAIR *cornerpair, int mode, int threshold);

// For Robust Estimation
int RANSAC(MATPAIR *cornerpair, CvMat *H, int *inlierNo);
int MakeHomographyMat(FLPOINT *pin, FLPOINT *cpin, int num, CvMat *matout);
int SelectPoints(MATPAIR *cornerpair, FLPOINT *pin, FLPOINT *cpin, int num);
int EstimateInlier(MATPAIR *cornerpair, CvMat *H, int *inlierNum);
int Normalize(FLPOINT *in, int num, CvMat *T);
int CheckColinearity(FLPOINT *pin, int num);
int ImageMosaicing(IplImage *wimage2, IplImage *image, IplImage *image2, CvMat *H);

#endif
```