Purdue University

ECE 661 Computer Vision

HW #4

Student Name:

Khalil Mustafa Khalil "Ahmad Yousef "

PUID:0020148553

Instructor: Prof. Avi Kak

Section #: 1

October 16, 2008

1. Problem Description:

The goal of this homework is to learn how to estimate the homography between two images and to eliminate the manual selection of the correspondences (pixels) between images by using the RANSAC algorithm, which automatically through iterative procedure selects some of them. RANSAC uses NCC (Normalized Cross Correlation) or SSD (Sum of Squared Difference) to match landmark features (Harris Corner) in one image with similar features in another image of the same scene that is not registered with the first image. RANSAC algorithm then randomly selects a set of putative correspondences (Minimum of 4) from the matched corner lists found using NCC or SSD for both images. Now the following steps are performed:

- The randomly selected putative correspondences are recursively checked for co-linearity. So, if the correspondences pixels are non-collinear the algorithm continues, otherwise a new set of putative correspondences is selected again.
- The selected non-collinear correspondences ($\vec{x}'$, $\vec{x}$) are then normalized. The reason for this step is that DLT (Direct Linear Transformation algorithm used to compute the homography [$A_{2n\times 9}h_{9\times 1} = 0$] from the set of correspondences $n \geq 4$) is not invariant under similarity transformations (choices of scale and coordinate origin). Thus with presence of noise the computed solution (homography) will diverge from the correct results. Therefore, to insure better results (matrix A has exactly rank = 8 with presence of noise) and to make sure that DLT is invariant to similarity transformations, Algorithm 4.2 in the book for data normalization is used which is widely known as Hartley algorithm. Data Normalization involves the following:
    - We choose to scale the coordinates so that the average distance of a point (corner) x from the origin is equal to $\sqrt{2}$. This means that the "average" point is equal to $(1,1,1)^T$.
    - The points are translated so that their centroid is at the origin (0, 0).
    - The points are then scaled so that the average distance from the origin is equal to $\sqrt{2}$.
    - This transformation is applied to each of the two images independently (T, T').
- The normalized correspondences are used by DLT algorithm (applying SVD either on Equation 4.1 or on 4.3 in the book based on the number of correspondences) to compute the homography, noting that what we are computing is the homography between the normalized correspondences ($\tilde{\vec{x}}' = T'\vec{x}' \xleftrightarrow{H\_norm} \tilde{\vec{x}} = T\vec{x}$) not between original pixels ($\vec{x}' \xleftrightarrow{H} \vec{x}$). Therefore the homography H_{norm} between normalized pixels is computed first then the homography between original pixels H is given by $H = T'^{-1}H_{norm}T$.
- Having the homography, RANSAC comes to the picture to make sure that this estimated Homography computed in previous step is as correct as possible by insuring that no outliers (mismatched points) has originally participated in the homography computation (all the used points are only inliers). To do so, the estimated homography H is applied to both images, so that we transform the first image pixels (corners) by the multiplication of H into the second image pixels or correspondents and transform the second image pixels by the multiplication of H^{1} into the first image correspondents. Now, what we have is the original correspondences and the transformed correspondences, which should be equal if our homography is correct and accurate. This is the key to judge the correctness of the estimated homography. Simply, we compute the distance threshold (Algorithm 4.3 in the book: sum of the square of Euclidean distance) between every correspondent and its transform and then check if it is below a certain threshold t determined by table 4.2 in the book. So, if correspondent pair is below t, we add the correspondent pixels to the consensus set S holding only inliers correspondences and if not we simply ignore these pixels and repeat the process for all the correspondences.
- Now, we check the size of the consensus set. So, if the number of the inliers in the consensus set is greater than certain threshold, which is defined adaptively as the maximum number of inliers we have so far and

can be initially one, then we re-estimate the homography using all points in S and terminate by applying the obtained homography to our images. If below the threshold, we have to start all previous steps from the beginning.

- We repeat this process N trials (This number is updated adaptively according to equation 4.18 in the book so that the probability that at least one of the random samples of S points is free from outliers is 0.99) after which the largest consensus set S_i is selected and the homography is re-estimate using all points in the subset S_i.
- All the previous steps are summarized in the following flowchart Table 1.

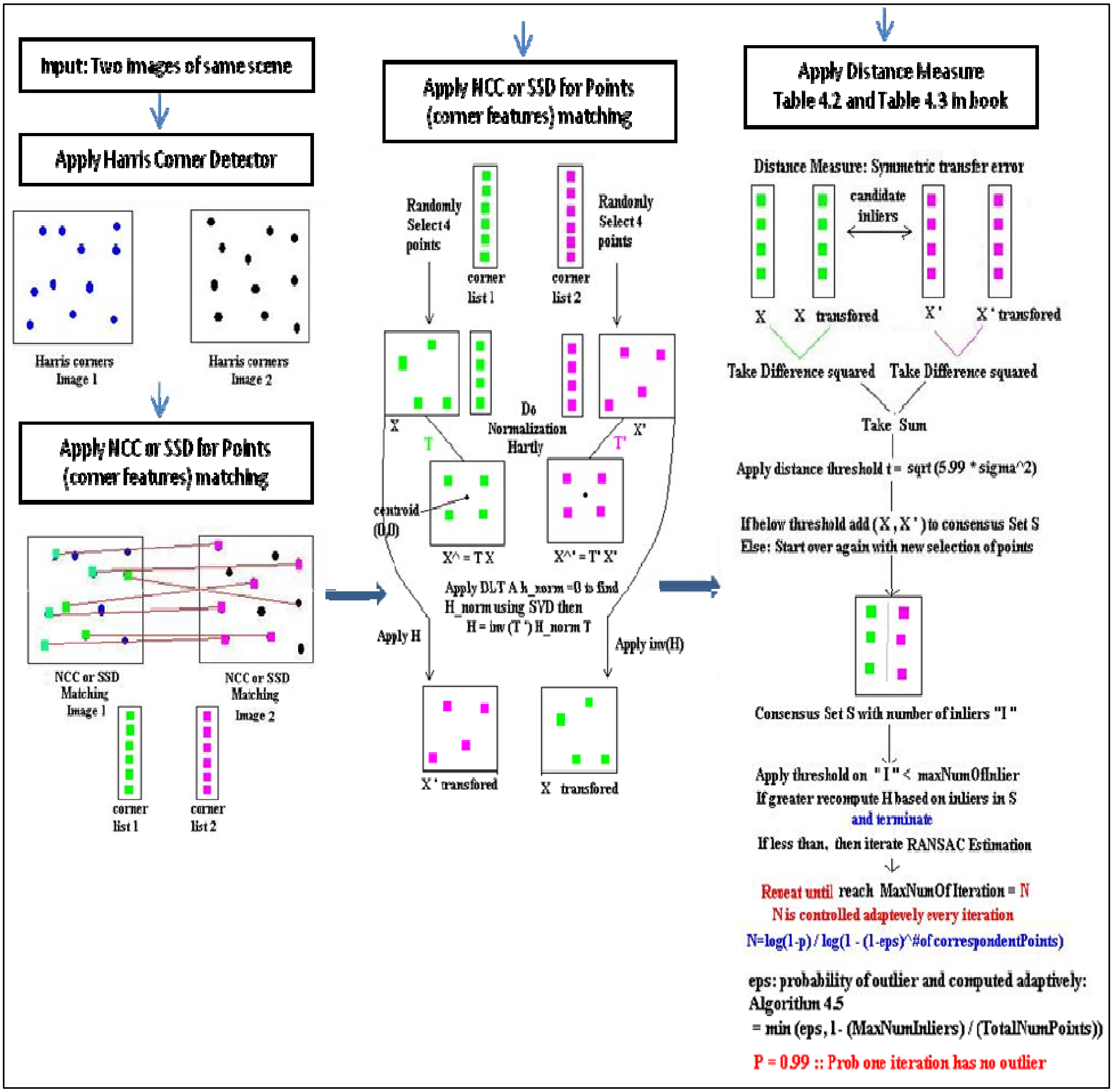

Table 1: RANSAC Algorithm Flowchart

2. Implementation

2.1 Input:

Input is two images of the same scene that have small differences of rotation and translation between them. See Figure 1 below.

Figure 1: Two images of the same scene with small roational angle

2.2: Harris Corner Detection (HW3)

Corner points are extracted using Harris corner detector applied to both images see Figure 2 and Figure 3 below.

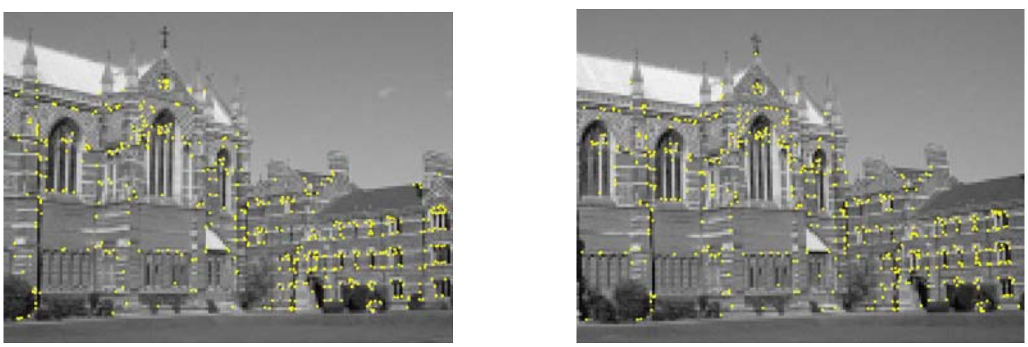

Figure 2: Detected Harris corners images

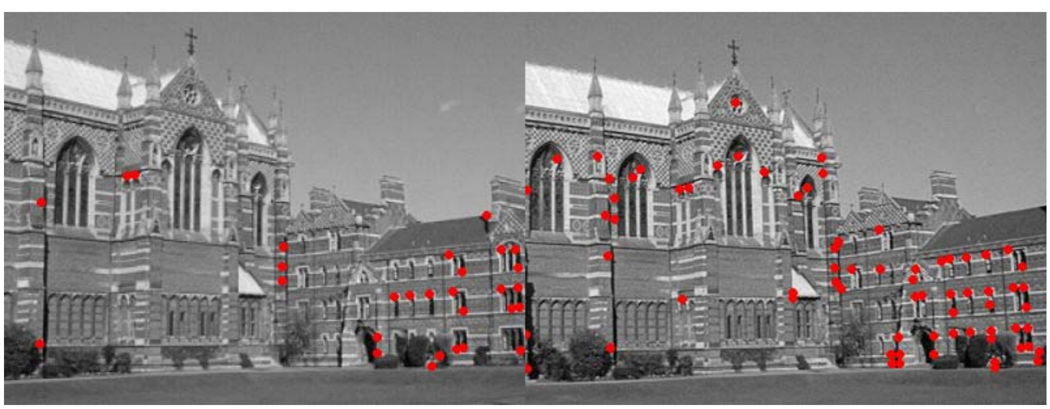

Figure 3: Detected corners superimposed on a combined image of the two images of the same scene

2.3 NCC and SSD Similarity Matching (HW3):

Putative correspondences between the set of extracted corners in the first image and the ones in the second image are computed and highlighted by comparing image neighborhoods around the corner points using both NCC and SSD. See Figure 4 for the output results of this part and Figure 5 for abstract implementation of NCC.

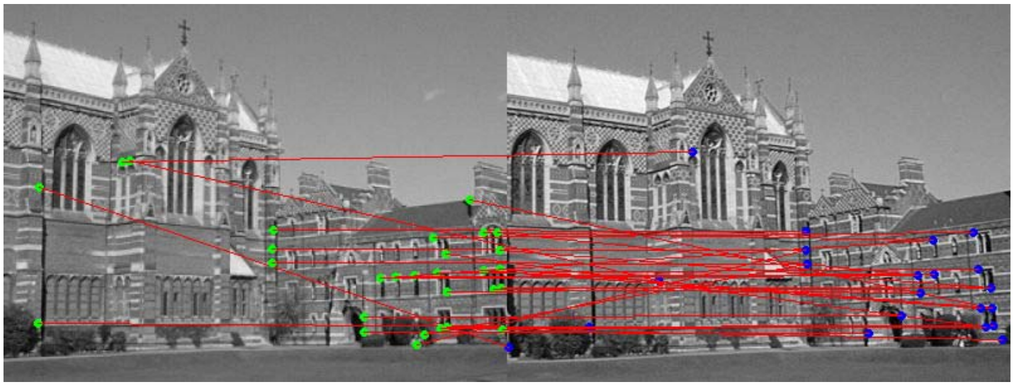

Figure 4: The result of correspondence matching using NCC. With tight threshold

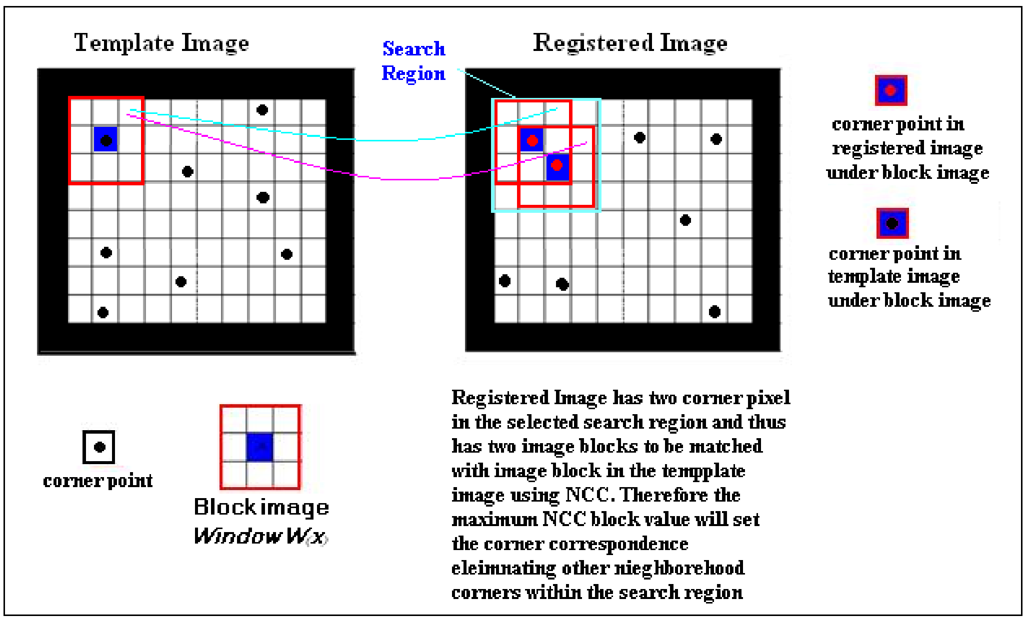

Figure 5: NCC Implementation Abstract

2.4 RANSAC:

RANSAC algorithm is used to estimate the homography between two images by automatically selecting the correspondences (inliers) through an iterative procedure. Algorithm 4.6 is implemented which actually demonstrates the RANCSAC steps and is showed in Table 1. In this section, more elaboration is presented to clarify the RANSAC operations.

Algorithm 4.6:
Compute the 2D homography between two images. Algorithm

(i) **Interest points:** Compute interest points in each image.
(ii) **Putative correspondences:** Compute a set of interest point matches based on proximity and similarity of their intensity neighbourhood.
(iii) **RANSAC robust estimation:** Repeat for N samples, where N is determined adap-tively as in algorithm 4.5:

(a) Select a random sample of 4 correspondences and compute the homography H.
(b) Calculate the distance $d\pm$ for each putative correspondence.
(c) Compute the number of inliers consistent with H by the number of correspondences for which $d_{\perp} = \sqrt{5.99}\sigma$ pixels.

Choose the H with the largest number of inliers. In the case of ties choose the solution that has the lowest standard deviation of inliers.

(iv) **Optimal estimation:** re-estimate H from all correspondences classified as inliers, by minimizing the ML cost function (4.8-p95) using the Levenberg-Marquardt algorithm of section A6.2(p600).
(v) **Guided matching:** Further interest point correspondences are now determined using the estimated H to define a search region about the transferred point position.
The last two steps can be iterated until the number of correspondences is stable.

2.4.1 Co-linearity Check

All points (corners) that are selected randomly are checked for co-linearity. The following procedure is followed:

- Based on the number of correspondences we have selected initially or we have in each RANSAC iteration say it n, then points are recursively checked in triples (n choose 3) times (all possible combination in which order doesn't matter), due to the fact that any two points are collinear (i.e. form a line) , thus a third point is conlinear if it is on the line. So basically, from two points we take a cross product to form a line then we take the dot product between the line and the third point. If dot product is zero or within threshold distance μ, then the triple is collinear. If not, the triple is not collinear. This process is repeated for all non-repeated possible triples (regardless the order)
- .μ is choosen 0.5.
- See Figure 6.

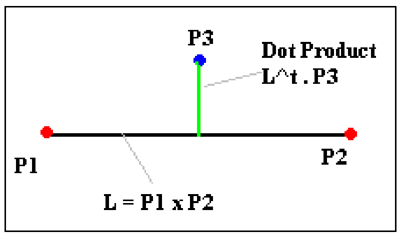

Figure 6 Colinerarity check

2.4.2 Data Normalization

Algorithm 4.2 in the book for data normalization is used which is also known as Hartley algorithm [1].

Algorithm 4.2:

Given $n > 4$ 2D to 2D point correspondences { $x_i \leftrightarrow x_i'$ }, determine the 2D homography matrix H such that $x_i' = Hx_i$.
Algorithm
(i) **Normalization of** x: Compute a similarity transformation T, consisting of a translation and scaling, that takes points x to a new set of points $\tilde{x}_i$ such that the centroid of the points $\tilde{x}_i$ is the coordinate origin $(0, 0)^T$, and their average distance from the origin is $\sqrt{2}$.
(ii) **Normalization of** x' : Compute a similar transformation T' for the points in the second image, transforming points x_i' to $\tilde{x}'$.
(iii) **DLT:** Apply algorithm 4.1(p91) to the correspondences $\tilde{x}_i \leftrightarrow \tilde{x}_i'$ to obtain a homography H.
(iv) **Denormalization:** Set $H = T'^{-1}\tilde{H}T$.

The transformation matrices (T, T') follows the same form used in Hartly paper referenced below.

[1] Wojciech Chojnacki, Michael J. Brooks, Anton van den Hengel, Darren Gawley, "Revisiting Hartley's Normalized Eight-Point Algorithm," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 9, pp. 1172-1177, Sept., 2003

$$T=\begin{bmatrix} s^{-1} & & -s^{-1}\overline{m}_1 \\ & s^{-1} & -s^{-1}\overline{m}_2 \\ 0 & 0 & 1 \end{bmatrix}, T'=\begin{bmatrix} s'^{-1} & & -s'^{-1}\overline{m}'_1 \\ & s'^{-1} & -s'^{-1}\overline{m}'_2 \\ 0 & 0 & 1 \end{bmatrix}$$

Where $(\overline{m}_1,\overline{m}_2),(\overline{m}'_1,\overline{m}'_2)$ be the centroids pixels of domain and range image repectively and computated by averaging the *x* and *y* coordinates of all selected points in the domain and range image separtly. s,s' be scalar quantities of domain and range image repectivel and computed as follows: $s=\left(\frac{1}{2n}\sum_{i=1}^{n}\left\|m_i-\overline{m}\right\|^2\right)^{0.5}=\left(\frac{1}{2n}\sum_{i=1}^{n}(m_{1,i}-\overline{m}_1)^2+(m_{2,i}-\overline{m}_2)^2\right)^{0.5}$, $m_{1,i},m_{2,i}$ are corner coordinate.

$$s'=\left(\frac{1}{2n}\sum_{i=1}^{n}\left\|m'_i-\overline{m}'\right\|^2\right)^{0.5}=\left(\frac{1}{2n}\sum_{i=1}^{n}(m'_{1,i}-\overline{m}'_1)^2+(m'_{2,i}-\overline{m}'_2)^2\right)^{0.5}$$

After doing normalization based on the following transformations : $\widetilde{x}_i=Tx_i$, $\widetilde{x}'_i=T'x'_i$, $\widetilde{x}'_i=\widetilde{H}\widetilde{x}_i$. We can compute the estimated Homography *H*: $x'=Hx\ \Rightarrow H=T'^{-1}\,\widetilde{H}T$ and that's what we want.

2.4.3 Computing Homography *H* (DLT Algorithm 4.2):

Equation 4.1 or 4.3 in the book is used based on the number of correspondences *n* we have. Thus, if number of correspondences is equal four (i.e. *n* =4), then equation 4.1 is used and want to solve $A_{3n\times 9}h_{9\times 1}=0$, where the solution $h_{9\times 1}$ is the null space of *A* (*A* has rank =8), which is also the last column vector of orthogonal matrix *V* obtained from singular value decomposition SVD of $A=U\sum V^T$. This result is obtained based on the derivation made in the class, where the original problem was to minimize: $\min_{h,\|h\|_2=1}\left\|A_{3n\times 9}h_{9\times 1}\right\|_2$ which corresponds to minimum eigenvalue.

Equation 4.1: $\begin{bmatrix} 0^{\mathsf T} & -w'_i\mathrm{x}_i^{\mathsf T} & y'_i\mathrm{x}_i^{\mathsf T} \\ w'_i\mathrm{x}_i^{\mathsf T} & 0^{\mathsf T} & -x'_i\mathrm{x}_i^{\mathsf T} \\ -y'_i\mathrm{x}_i^{\mathsf T} & x'_i\mathrm{x}_i^{\mathsf T} & 0^{\mathsf T} \end{bmatrix}\begin{pmatrix}\mathrm{h}^1\\ \mathrm{h}^2\\ \mathrm{h}^3\end{pmatrix}=0$

Equation 4.3: $\begin{bmatrix} 0^{\mathsf T} & -w'_i\mathrm{x}_i^{\mathsf T} & y'_i\mathrm{x}_i^{\mathsf T} \\ w'_i\mathrm{x}_i^{\mathsf T} & 0^{\mathsf T} & -x'_i\mathrm{x}_i^{\mathsf T} \end{bmatrix}\begin{pmatrix}\mathrm{h}^1\\ \mathrm{h}^2\\ \mathrm{h}^3\end{pmatrix}=0$

If number of correspondences *n* is greater than 4, then equation 4.3 is used and want to solve $A_{2n\times 9}h_{9\times 1}=0$, where the solution $h_{9\times 1}$ is the null space of *A* obtained using SVD (last column vector of orthogonal matrix *V*).

2.4.4 Distance Measure.

Distance measure is used to judge the correctness of the estimated homography. Simply, we compute the distance threshold following Algorithm 4.3 in the book which is known as Symmetric transfer error *i.e.* the sum of the square of Euclidean distance between every correspondent and its transform.

Symmetric transfer error = *derror*= $\sum_i d(x_i, H^{-1}x_i')^2 + d(x_i', Hx_i)^2$

Check if *derror* is below a certain threshold $t = \sqrt{5.99}\sigma$ determined by table 4.2 in the book (σ is chosen 3). So, if correspondent pair is below *t*, we add the correspondent pixels to the consensus set *S* holding only inliers correspondences and if not we simply ignore these pixels and repeat the process for all correspondences. Now we check the size of the consensus set. So, if the number of the inliers in the consensus set is greater than certain threshold, which is defined adaptively as the maximum number of inliers we have so far and is considered initially one, then we re-estimate the homography using all points in *S* and terminate by applying the obtained homography to our images. If below the threshold, we have to start RANSAC algorithm Again.

2.4.5 RANSAC Parameters Selection:

There is no key secret to set the RANSAC parameters e.g. *N* (number of trials of RANSAC), thresholds and others, but we are guided by setting them to the extent we have the best results. Therefore, results obtained in this homework are based on trial and error of setting the parameters with the consifderation (e.g. Tables 4.2 and 4.3 and algorithm 4.5 in the book).

Results:

Figure 7: Input images

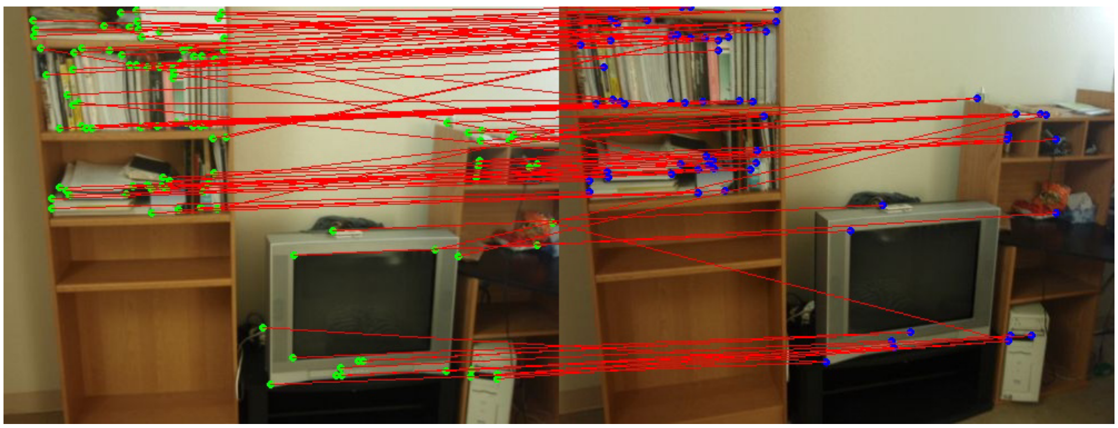

Figure 8: The result of correspondence matching using NCC (102 correspondences, threshold -2)

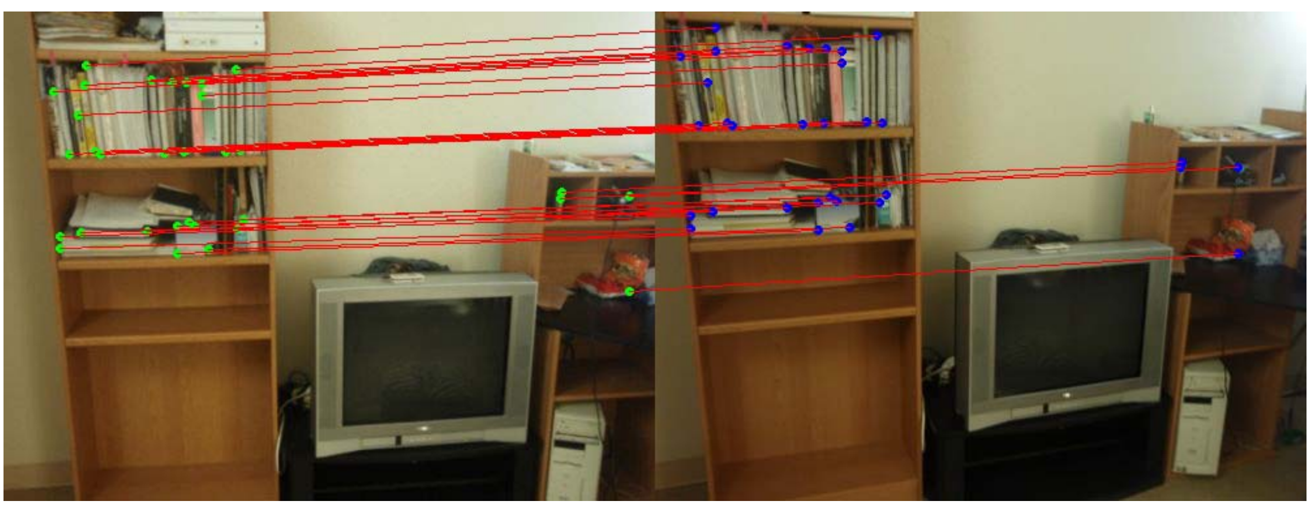

Figure 9: Inlier matching (32 correspondences)

Figure 10: Final Results without Homography Refinement (First Image, First Image with Homography, Second Image) with N=71 , eps=0.5 trials=71

Figure 11: Final Results with Homography Refinement (First Image, First Image with Homography, Second Image)

Figure 12: Input images

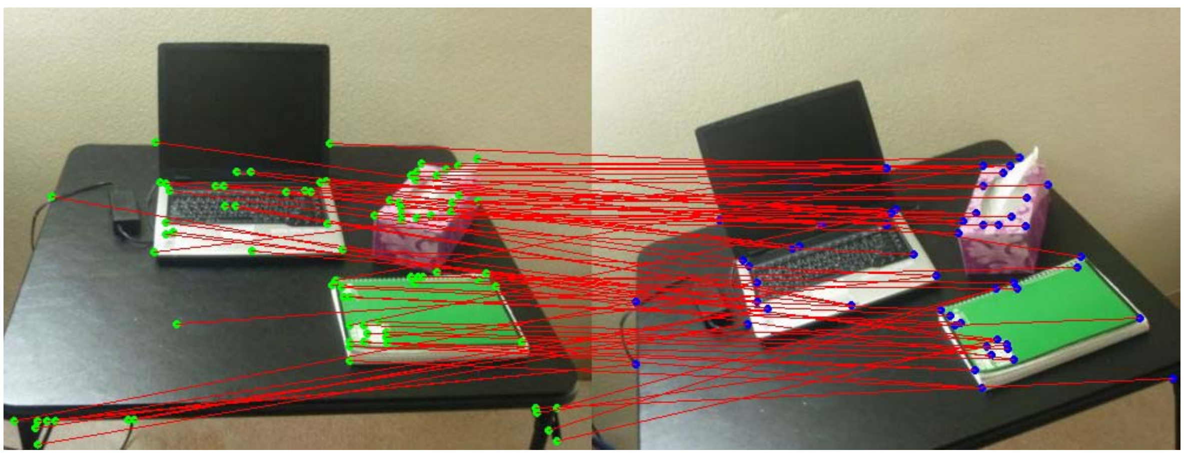

Figure 13: The result of correspondence matching using NCC (83 correspondences, threshold -2)

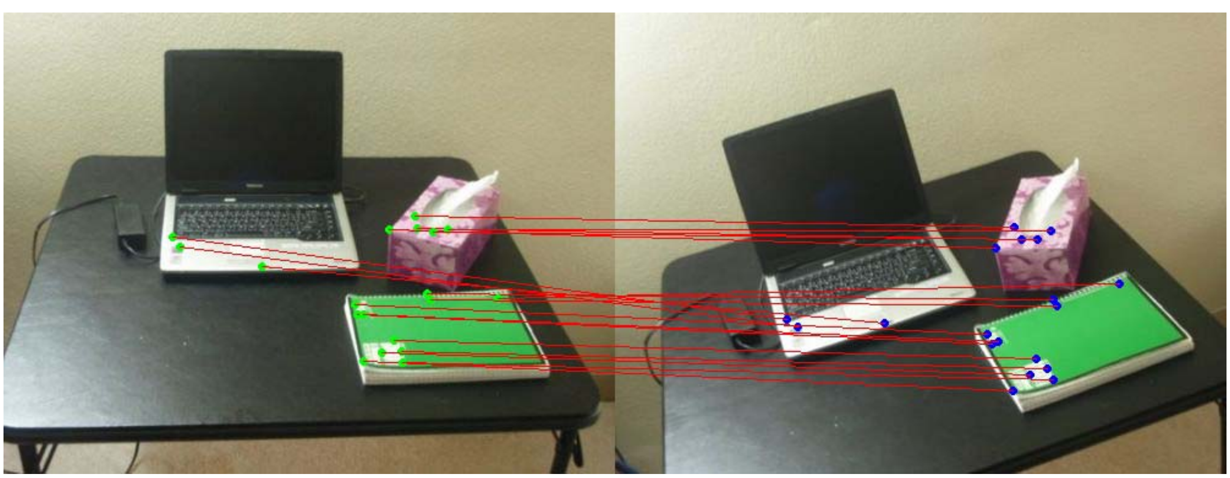

Figure 14: Inlier matching (18 correspondences)

Figure 15: Final Results without Homography Refinement (First Image, First Image with Homography, Second Image) with N=71 , eps=0.5 trials=71

Figure 16: Final Results with Homography Refinement (First Image, First Image with Homography, Second Image)

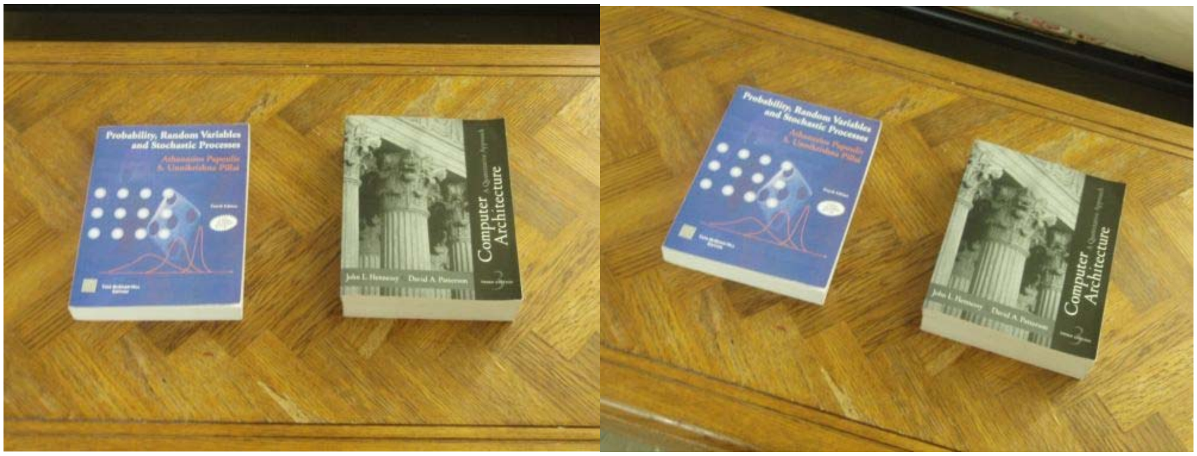

Figure 17: Input images

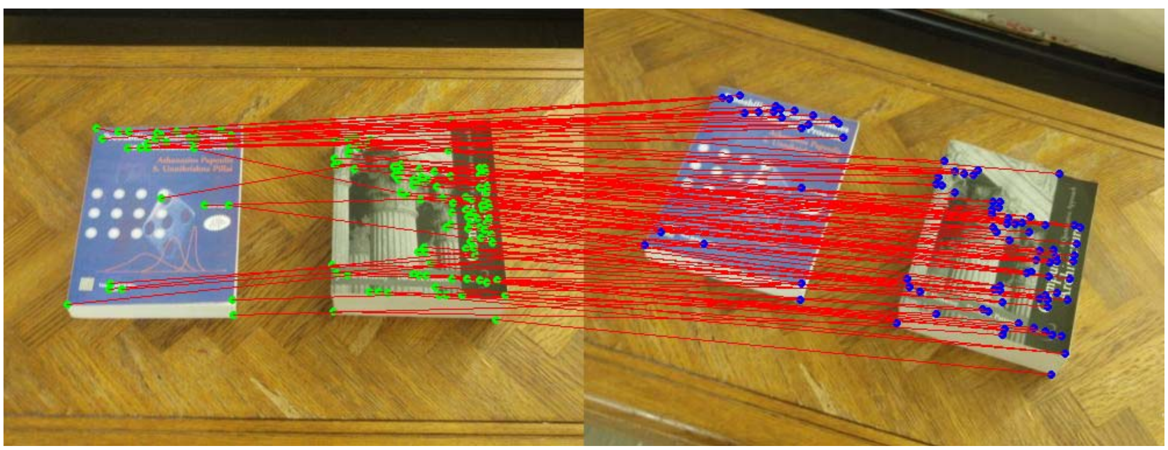

Figure 18: The result of correspondence matching using NCC (123 correspondences, threshold -2)

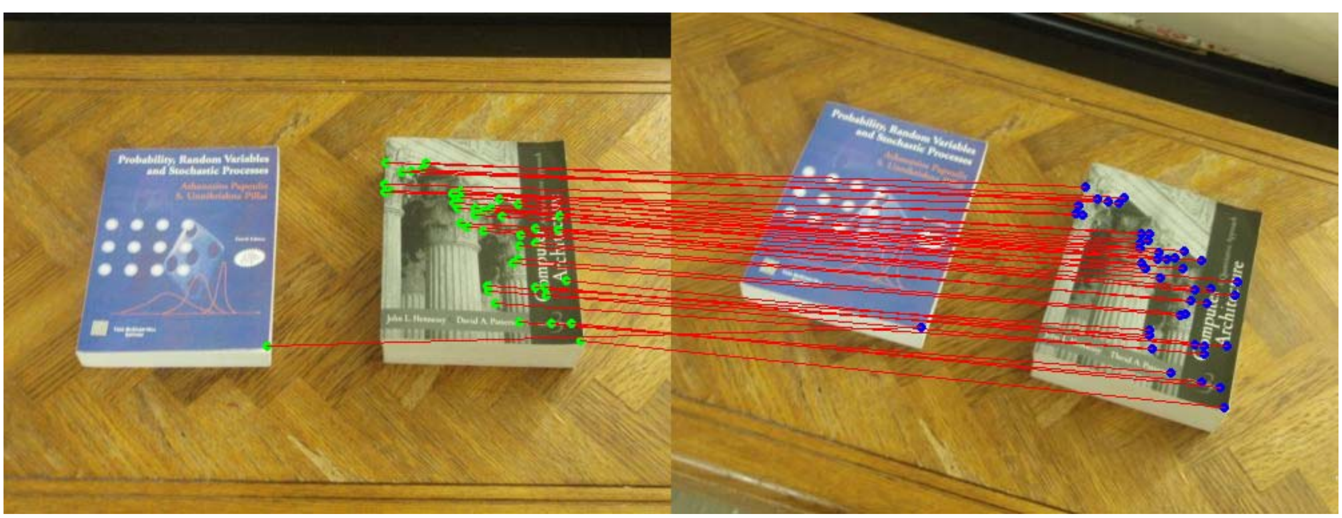

Figure 19: Inlier matching (43 correspondences)

Figure 20: Final Results without Homography Refinement (First Image, First Image with Homography, Second Image) with N=71 , eps=0.5 trials=71

Figure 21: Final Results with Homography Refinement (First Image, First Image with Homography, Second Image)

Figure 22: Input images

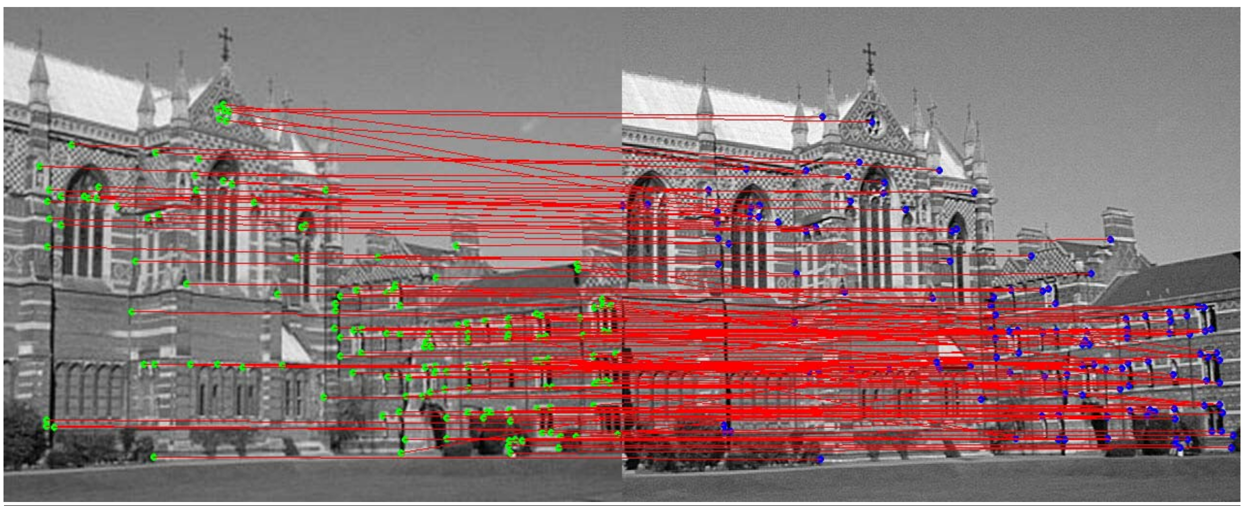

Figure 23: The result of correspondence matching using NCC (145 correspondences, threshold -1.5)

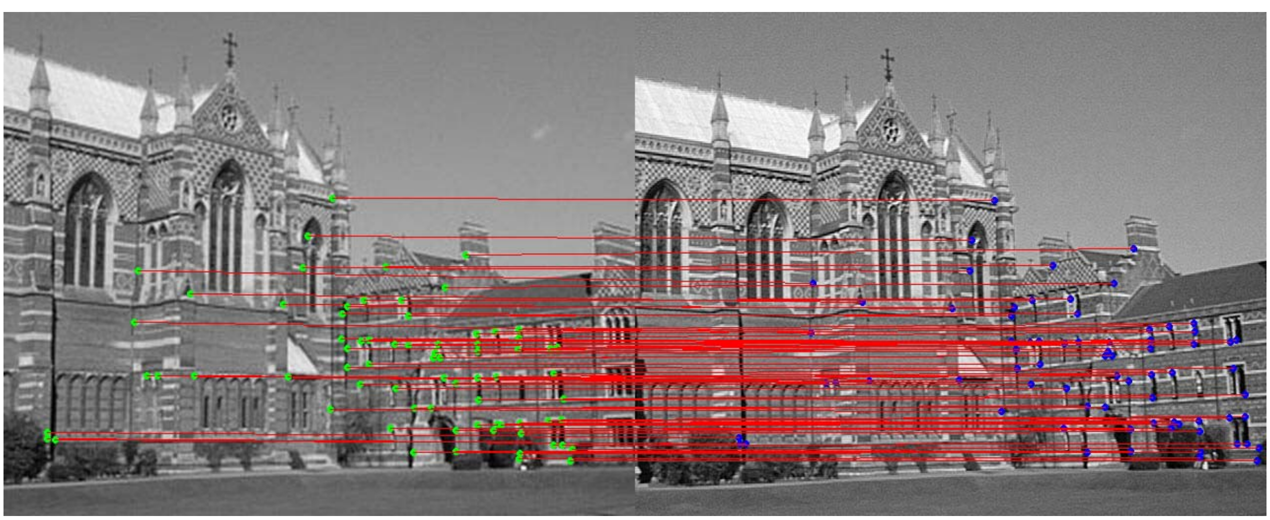

Figure 14: Inlier matching (73 correspondences)

Figure 25: Final Results without Homography Refinement (First Image, First Image with Homography, Second Image) with N=69   eps=0.496552   trials=69

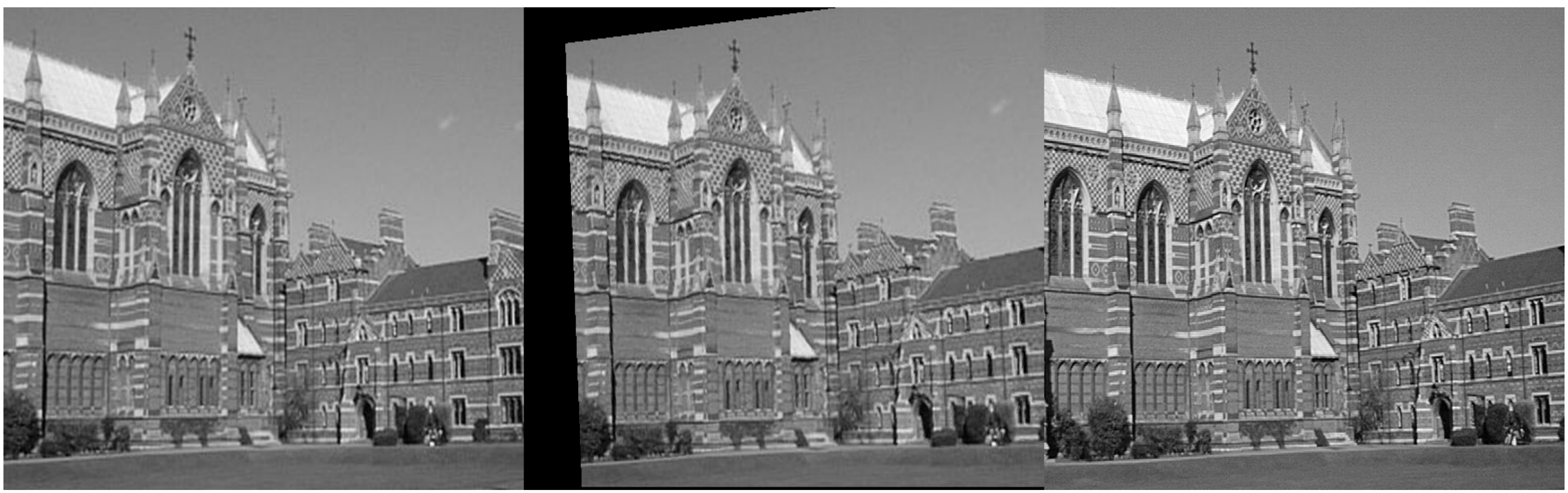

Figure 26: Final Results with Homography Refinement (First Image, First Image with Homography, Second Image)

Code has 7 files:

hw3.cpp, control.h , control.cpp, Harris.cpp, matchNCC.cpp, matchSSD.cpp and RANSAC.cpp)

```
//***************************************************************************************************
// control.h                          Due: Thursday October 16, 2008
// Holds all definitions and function declerations
//
// Prepared By: Khalil Yousef
// Computer Vision ECE 661, Prof. Avi Kak
//***************************************************************************************************
#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <cv.h>
#include <cxcore.h>
#include <highgui.h>
#include <time.h>
#include <cstdlib>

#define HARRIS_THRESHOLD 7000          // make threshold tight by increasing this amount
#define SSD_THRESHOLD 500000           // make threshold tight by decreasing this amount
#define NCC_THRESHOLD -2.0             // make threshold tight by increasing this amount
#define WINDOW_OPTION CV_BLUR          // Smoothing windoe Option used in cvSmooth command
#define PARAM1 3                       // Smoothing Window parameter
#define PARAM2 3                       // smoothing Window parameter
#define RANGE_NEIGHBOR 6
#define KABA 0.04                      // Hariss Equaion factor
#define WINDOWSIZE_NCC 25              // NCC Image block window size
#define SEARCH_RANGE 256               // NCC Matching Search region in Registered Image
#define MAX_NUM_CORNER 2000            // Maximum Number of corners to be found by NCC or SSD
#define WINDOWSIZE_SSD 25              // SSD Image block window size
#define OUTPUT_MODE 1                  // 1: Horizontal View  2: Vertical view
#define COLINEARITY_THRESHOLD 0.5      // point on line means dot product zero or within this threshold
#define MaxNumOfCorresp 1000           // Max num of Inlier correspondencies
#define MinNumOfCorresp  4;            // S:= min # of data element needed to form an estimate H
#define SIGMA sqrt(9)                  // Distance threshold param based on Gaussian
                                       // Distribution of noise of points
#define DISTANCE_THRESHOLD sqrt(5.99*pow(SIGMA,2))   // based on table 4.2 page 119-book
#define MaxRANSACtrialNum  1000        // MAX number of RANSAC Trials , could be inf initially
#define p 0.99                         // Probability at least one trial have no outlier
#define e 0.5                          // Worst case Probabiity of outlier According to the book

// Corner List structure
typedef struct{
        int len;
        int regImgPositionI[MAX_NUM_CORNER];
        int regImgPositionJ[MAX_NUM_CORNER];
        int tempImgPositionI[MAX_NUM_CORNER];
        int tempImgPositionJ[MAX_NUM_CORNER];
}CorspMap;

//Image processing tool
void Array2CvMat(float *arr, CvMat *cvArr, int row, int column);
void CvMat2Array(CvMat *cvArr, float *arr, int row, int column);
void CvImageCopyFloat2Uchar(IplImage *src, IplImage *dst);
void CvImageCopyUchar2Float(IplImage *src, IplImage *dst);
void InitializeImage(IplImage *image);
void CombineTwoImages(IplImage *image1, IplImage *image2,IplImage *outImage);
void CombineThreeImages(IplImage *image1, IplImage *image2,IplImage *image3, IplImage *outImage);
void WriteImage(IplImage *image, char *imagee);
void MakeImageBlock(IplImage *block, IplImage *image,int centPosI, int centPosJ);
void ApplyHomography(IplImage *inImage, IplImage *outImage, CvMat *H);
void ApplyInvHomography(IplImage *inImage, IplImage *outImage, CvMat *H);

//Harris
void HarrisCornerDectect(IplImage *inImage, IplImage *cornerMap);
void Sobel(IplImage *inImage, IplImage *outImage,CvMat *kernel1st, CvMat *kernel2nd);
void FindLocalMaxPoint(IplImage *inImage, IplImage *map, int range);
```

```
void HarrisResults(IplImage *inImage1, IplImage *inImage2,IplImage *cornerMap1, IplImage *cornerMap2);
void MarkCornerPoints(IplImage *image, IplImage *cornerMap);

//NCC
void  match_NCC(IplImage *tempImg, IplImage *regImg,IplImage *tempCorMapImg, IplImage
*regCorMapImg,CorspMap *corspMap);
bool  NCCcheck(IplImage *registeredImg, IplImage *templateBlock,IplImage *regCorMapImg,int cornerPosI,
int cornerPosJ,int *pCorspPosI, int *pCorspPosJ,int searchRange);
float NCC(IplImage *block1, IplImage *block2);
void  UpdateCorrespMap(CorspMap *corspMap, int tempCorPosI, int tempCorPosJ,int regCorPosI, int
regCorPosJ);
void  InitializeCorspMap(CorspMap *corspMap);
void  NCCResults(IplImage *inImage1, IplImage *inImage2, CorspMap *corspMap,char *outputName, int
NCCstep);
void  DrawCorrespLine(IplImage *templateImage, IplImage *registeredImg,IplImage *outImage, CorspMap
*corspMap);

//SSD
void match_SSD(IplImage *tempImg, IplImage *regImg,IplImage *tempCorMapImg, IplImage
*regCorMapImg,CorspMap *SSDcorspMap);
bool SSDcheck(IplImage *registeredImg, IplImage *templateBlock,IplImage *regCorMapImg,int cornerPosI, int
cornerPosJ,int *pCorspPosI, int *pCorspPosJ);
void SSDResults(IplImage *tempImg, IplImage *regImg, CorspMap *SSDcorspMap,char *outputName);
float SSD(IplImage *block1, IplImage *block2);

//RANSAC
void CopyCorspMap(CorspMap *dst, CorspMap *src);
void RANSACresults(IplImage *tempImg,IplImage *H_tempImg, IplImage *regImg,char *outputName, int step);
void HomograhyRefinement(CorspMap *inlierMap, CvMat *H);
void RANSAC(CorspMap *corspMap, CorspMap *inlierMap,CvMat *H);
void ComputeHomography(float templatePosiitons[][2], float registeredPosiitons[][2],int numOfCorresp,
CvMat *H);
void DataNormalization(int numOfPositions, float positions[][2], CvMat *T);
void DistanceMeasure(CvMat *H, CorspMap *corspMap, CorspMap *inlierMap);
bool CollinearityCheck(float points[][2], int numOfPoints);
bool IsColinear(CvMat *p1, CvMat *p2, CvMat *p3);



//*********************************************************************************************
// HW4.cpp                              Due: Thursday October 16, 2008
// 1. detect corner points of 2 input images.
// 2. using the corner points, finds correspondences of the 2 images or Image registration
//         a. Using NCC
//         b. Using SSD
// 3. Estimate homography and remove projectivity distortion using
//              RANSAC
//
// Prepared By: Khalil Yousef
// Computer Vision ECE 661, Prof. Avi Kak
//**********************************************************************************************

#include "control.h"   // Holds All Declerations

int main()
{
        //************************************get input images************************
        // decleration of input images of same scene
        IplImage *tempImg = 0, *regImg = 0;
        int select=1;
        char *templateImg;
        char *rigesteredImg;
        printf("Please select your pairs image (1,2,3,4)\n");
        scanf("%d",&select);
        switch(select)
        {
                case(1): templateImg ="test1_1.jpg";rigesteredImg= "test1_2.jpg"; break;
                case(2): templateImg ="test2_1.jpg";rigesteredImg = "test2_2.jpg";break;
                case(3): templateImg ="test3_1.jpg";rigesteredImg="test3_2.jpg";break;
                default: templateImg ="sample_a.jpg";rigesteredImg = "sample_b.jpg";break;
        }
```

```
// load input images
tempImg = cvLoadImage(templateImg, 1);
if(!tempImg)
{printf("Could not load image file: %s\n", templateImg); exit(0);}
regImg = cvLoadImage(rigesteredImg, 1);
if(!regImg){printf("Could not load image file: %s\n", rigesteredImg);exit(0);}

long StartTime, FinishTime, RunTimeHarris, RunTimeNCC, RunTimeSSD, RunTimeRANSAC;
srand(time(0));

//*************************************Harris corner detector**************************
StartTime=clock();
IplImage *cornerMap1 = 0, *cornerMap2 = 0;          // Corner images

cornerMap1 = cvCreateImage(cvSize(tempImg->width, tempImg->height), IPL_DEPTH_8U, 1);
cornerMap2 = cvCreateImage(cvSize(tempImg->width, tempImg->height), IPL_DEPTH_8U, 1);

// Apply Harris corner detection algorithm into 2 images
HarrisCornerDectect(tempImg, cornerMap1);
HarrisCornerDectect(regImg, cornerMap2);
FinishTime=clock();
RunTimeHarris=(FinishTime-StartTime)/ CLK_TCK;

// Show Harris Corners results
HarrisResults(tempImg, regImg, cornerMap1, cornerMap2);

//*********************************** Similarity Matching******************************
//**************************** find correspondences using NCC *************************
//Define array structure of coresponding corner points (holds coordinates)
StartTime=clock();
CorspMap NCCcorspMap;
InitializeCorspMap(&NCCcorspMap); // initialize correspondence map
match_NCC(tempImg, regImg, cornerMap1, cornerMap2, &NCCcorspMap);
FinishTime=clock();
RunTimeNCC=(FinishTime-StartTime)/ CLK_TCK;
NCCResults(tempImg, regImg, &NCCcorspMap, "result_NCC.jpg",1);
printf("Number of NCC Correspondences = %d\n", NCCcorspMap.len);

//**************************** find correspondences using SSD *************************
StartTime=clock();
CorspMap SSDcorspMap;
InitializeCorspMap(&SSDcorspMap); // initialize correspondence map
InitializeCorspMap(&SSDcorspMap); // initialize correspondence map
match_SSD(tempImg, regImg, cornerMap1, cornerMap2, &SSDcorspMap);
FinishTime=clock();
RunTimeSSD=(FinishTime-StartTime)/ CLK_TCK;
SSDResults(tempImg, regImg, &SSDcorspMap, "result_SSD.jpg");
printf("Number of SSD Correspondences = %d\n", SSDcorspMap.len);



//**************************** Estimate Homography using RANSAC *************************
//****************************************find inliers*********************************
StartTime=clock();
CorspMap inlierMap;
InitializeCorspMap(&inlierMap); // initialize inliers map

// creat result image after applying the homography to tempImg and then compare the result with
// regImg

IplImage *H_tempImg = 0;
H_tempImg = cvCreateImage(cvSize(tempImg->width, tempImg->height), IPL_DEPTH_8U, tempImg-
>nChannels);

CvMat *H = cvCreateMat(3, 3, CV_32FC1);             // Homography Matrix

RANSAC(&NCCcorspMap, &inlierMap, H);

// plot inliers matching  and thier numbers
NCCResults(tempImg, regImg, &inlierMap, "result_step3_inliers.jpg",2);
```

```
        // Apply Homography H and plot Results
        // transform tempImg using H to get regImg and compare it with original regImg
        ApplyHomography(tempImg, H_tempImg, H);
        RANSACresults(tempImg,H_tempImg, regImg,"result_Estimate1.jpg", 1);

        //***************************Refinement Recalculate Homography *************************
        IplImage *H_tempImg2 = 0;
        H_tempImg2 = cvCreateImage(cvSize(tempImg->width, tempImg->height), IPL_DEPTH_8U, tempImg-
        >nChannels);

        // recalculate the homoraphy and apply it again to the image
        HomograhyRefinement(&inlierMap, H);
        ApplyHomography(tempImg, H_tempImg2, H);
        RANSACresults(tempImg,H_tempImg2, regImg,"result_Estimate2.jpg",2);
        FinishTime=clock();
        RunTimeRANSAC=(FinishTime-StartTime)/ CLK_TCK;

        //**********************************************************************************
        // release the images and matrix
        cvReleaseImage(&tempImg);
        cvReleaseImage(&regImg);
        cvReleaseImage(&H_tempImg);
        cvReleaseImage(&H_tempImg2);
        cvReleaseMat(&H);

        printf("\n\nRuntime of the algorithms are:\n");
        cout<<"Harris \t\t="<<RunTimeHarris<<" sec\n";
        cout<<"NCC Matching \t="<<RunTimeNCC<<" sec\n";
        cout<<"SSD Matching \t="<<RunTimeSSD<<" sec\n";
        cout<<"RANSAC \t\t="<<RunTimeRANSAC<<" sec\n";

        return 0;
}


//*****************************************************************************************************
// Control.cpp                        Due: Thursday October 16, 2008
// General Image processing image
//
// Prepared By: Khalil Yousef
// Computer Vision ECE 661, Prof. Avi Kak
//*****************************************************************************************************

#include "control.h"

//#####################################################################################################
void CvImageCopyFloat2Uchar(IplImage *src, IplImage *dst)
{
        int i, j, k;
        float pixel;
        int height = src->height;
        int width = src->width;
        int channels = src->nChannels;
        int step = dst->widthStep;
        uchar *dstData = (uchar *)dst->imageData;
        float *srcData = (float *)src->imageData;

        // copy float precision image to uchar precision image
        for(i = 0; i < height; i++) for(j = 0; j < width; j++) for(k = 0; k < channels; k++)
        {
                pixel = srcData[i*step + j*channels + k];
                pixel = (pixel > 255 ? 255 : pixel);
                pixel = (pixel < 0 ? 0 : pixel);
                dstData[i*step + j*channels + k] = (uchar)pixel;
        }
}

//#####################################################################################################

void CvImageCopyUchar2Float(IplImage *src, IplImage *dst)
{
```

```
	int i, j, k;
	int height = src->height;
	int width = src->width;
	int channels = src->nChannels;
	int step = src->widthStep;
	float *dstData = (float *)dst->imageData;
	uchar *srcData = (uchar *)src->imageData;

	// copy uchar precision image to float precision image
	for(i = 0; i < height; i++)for(j = 0; j < width; j++)for(k = 0; k < channels; k++)
		dstData[i*step + j*channels + k]= (float)srcData[i*step + j*channels + k];

}

//#####################################################################################################

void Array2CvMat(float *arr, CvMat *cvArr, int row, int column)
{
	int i, j;
	for(i = 0; i < row; i++)for(j = 0; j < column; j++)
		cvmSet(cvArr, i, j, arr[i*column + j]);
}

//#####################################################################################################


void CvMat2Array(CvMat *cvArr, float *arr, int row, int column)
{
	int i, j;
	for(i = 0; i < row; i++)
		for(j = 0; j < column; j++)
			arr[i*column + j] = float(cvmGet(cvArr, i, j));
}

//#####################################################################################################

void InitializeImage(IplImage *image)
{
	int i, j, k;
	int height = image->height;
	int width = image->width;
	int channels = image->nChannels;
	int step = image->widthStep;
	uchar *imageData = (uchar *)image->imageData;
	for(i = 0; i < height; i++)for(j = 0; j < width; j++)for(k = 0; k < channels; k++)
		imageData[i*step + j*channels + k]= 0;
}

//#####################################################################################################

void InitializeCorspMap(CorspMap *corspMap)
{
	for(int i = 0; i < MAX_NUM_CORNER; i++)
	{
		corspMap->regImgPositionI[i] = 0;
		corspMap->regImgPositionJ[i] = 0;
		corspMap->tempImgPositionI[i] = 0;
		corspMap->tempImgPositionJ[i] = 0;
	}
	corspMap->len = 0;
}
//#####################################################################################################

void UpdateCorrespMap(CorspMap *corspMap, int domainPosI, int domainPosJ, int rangePosI, int rangePosJ)
{
	int len;
	len = corspMap->len;
	if(corspMap->len >= MAX_NUM_CORNER)
	{
		printf("UpdateCorrespMap called on a full corspMap\n");
		printf("Next positions of correspondences will be overwritten\n");
```

```
                printf("in the current correspondence \n");
                len = MAX_NUM_CORNER - 1;
        }
        corspMap->regImgPositionI[len] = rangePosI;
        corspMap->regImgPositionJ[len] = rangePosJ;
        corspMap->tempImgPositionI[len] = domainPosI;
        corspMap->tempImgPositionJ[len] = domainPosJ;
        corspMap->len = len + 1;
}
//#####################################################################################################

void CombineTwoImages(IplImage *image1, IplImage *image2,IplImage *outImage)
{
        int i, j, k;
        uchar *outImageData = 0, *image1Data = 0, *image2Data = 0;
        int height = image1->height;
        int width = image1->width;
        int step = image1->widthStep;
        int channels = image1->nChannels;
        int outWidth = outImage->width;
        int outHeight = outImage->height;
        int outStep = outImage->widthStep;
        if(outWidth == width * 2 && outHeight == height){}
        else
                if(outWidth == width && outHeight == height * 2){}
                else{
                        printf("image combining error\n");
                        exit(0);
                }
        outImageData = (uchar *)outImage->imageData;
        image1Data = (uchar *)image1->imageData;
        image2Data = (uchar *)image2->imageData;
        for(i = 0; i < outHeight; i++)for(j = 0; j < outWidth; j++) for(k = 0; k < channels; k++)
        {
                if(i < height && j < width)
                     outImageData[i*outStep + j*channels + k]= image1Data[i*step + j*channels + k];
                else
                if((i >= height && j < width))
                  outImageData[i*outStep + j*channels + k]= image2Data[(i-height)*step + j*channels + k];
                else
                if((i < height && j >= width))
                   outImageData[i*outStep + j*channels + k]= image2Data[i*step + (j-width)*channels + k];
                else
                {  printf("there is no i > height & j > width \n");
                   exit(0);
                }
        }
}
//########################################################################################

void WriteImage(IplImage *image, char *imageName)
{
        if(!cvSaveImage(imageName, image))
        {
                printf("Could not save: %s\n", imageName);
        }
}

//#####################################################################################################
void MakeImageBlock(IplImage *block, IplImage *image,int centPosI, int centPosJ)
{
        uchar *blockData = 0, *imageData = 0;
        int blockHeight, blockWidth, imageHeight, imageWidth;
        int blockStep, channels, imageStep;
        int i, j, k, posI, posJ;
        blockHeight = block->height;
        blockWidth = block->width;
        imageHeight = image->height;
        imageWidth = image->width;
        channels = block->nChannels;
        blockStep = block->widthStep;
```

```
        imageStep = image->widthStep;
        blockData = (uchar *)block->imageData;
        imageData = (uchar *)image->imageData;
        for(i = 0; i < blockHeight; i++)
        {
                for(j = 0; j < blockWidth; j++)
                {
                        for(k = 0; k < channels; k++)
                        {
                                posI = centPosI + i - blockHeight / 2;
                                posJ = centPosJ + j - blockWidth / 2;
                                posI = min(max(posI, 0), imageHeight - 1);
                                posJ = min(max(posJ, 0), imageWidth - 1);
                                blockData[i*blockStep + j*channels + k]= imageData[posI*imageStep +
                                posJ*channels + k];
                        }
                }
        }
}
//#####################################################################################################
// usage : DrawCorrespLine(image1, image2, outImage, corspMap);
// -------------------------------------------------------------
void DrawCorrespLine(IplImage *tempImg, IplImage *regImg,IplImage *outImage, CorspMap *corspMap)
{
        int a, b, i;
        uchar *outImageData = 0, *tempImgData = 0, *regImgData = 0;
        CvPoint rangePos;
        CvPoint domainPos;
        int height = regImg->height;
        int width = regImg->width;
        if(height == outImage->height && width * 2 == outImage->width)
                a = 1; b = 0;
        else
                if(height * 2 == outImage->height && width == outImage->width)
                        a = 0; b = 1;
                else
                {
                        printf("Error\n");
                        exit(0);
                }
        outImageData = (uchar *)outImage->imageData;
        tempImgData = (uchar *)tempImg->imageData;
        regImgData = (uchar *)regImg->imageData;
        CombineTwoImages(tempImg, regImg, outImage);
        for(i = 0; i < corspMap->len; i++)
        {
                rangePos = cvPoint(corspMap->regImgPositionJ[i] + width * a,corspMap->regImgPositionI[i] +
                height * b);
                domainPos = cvPoint(corspMap->tempImgPositionJ[i],corspMap->tempImgPositionI[i]);
                cvCircle(outImage, domainPos, 2, cvScalar(0, 255, 0), 2);
                cvCircle(outImage, rangePos, 2, cvScalar(255, 0, 0), 2);
                cvLine(outImage, domainPos, rangePos, cvScalar(0, 0, 255), 1);
        }
}

//#####################################################################################################

void MarkCornerPoints(IplImage *image, IplImage *cornerMap)
{
        int i, j;
        uchar *cornerMapData = 0;
        int height = cornerMap->height;
        int width = cornerMap->width;
        int mapStep = cornerMap->widthStep;
        cornerMapData = (uchar *)cornerMap->imageData;
        for(i = 0; i < height; i++)for(j = 0; j < width; j++)
                if(cornerMapData[i*mapStep + j] == 1)
                        cvCircle(image, cvPoint(j, i), 2, CV_RGB(255,0,0), 2, 8, 0);
}
//#####################################################################################################
```

```
void CopyCorspMap(CorspMap *dst, CorspMap *src)
{
	for(int i = 0; i < MAX_NUM_CORNER; i++)
	{
		dst->regImgPositionI[i] = src->regImgPositionI[i];
		dst->regImgPositionJ[i] = src->regImgPositionJ[i];
		dst->tempImgPositionI[i] = src->tempImgPositionI[i];
		dst->tempImgPositionJ[i] = src->tempImgPositionJ[i];
	}
	dst->len = src->len;
}

//##########################################################################################################

// This function transforms input image using H
// out(x') = in(x) can be viewed as:
// out(x') = in(inv(H)x') (given output image x' and H => result H^-1 x')
// If you have output image x' and H, it is better to use ApplyInvHomography function rather than
// ApplyHomography in order to get rid of black lines (i.e. interpolation in some sense)

void ApplyHomography(IplImage *inImage, IplImage *outImage, CvMat *H)
{
	int i, j, k;

	// get the input image data
	int height = inImage->height;
	int width = inImage->width;
	int step = inImage->widthStep;
	int channels = inImage->nChannels;
	uchar *inData = (uchar *)inImage->imageData;
	uchar *outData = (uchar *)outImage->imageData;

	CvMat *invH = cvCreateMat(3, 3, CV_32FC1);
	cvInvert(H, invH);
	float invh[9];
	// out(x') = in(inv(H)x')
	CvMat2Array(invH, invh, 3, 3);
	int ii, jj;
	float x1, x2, x3;
	for(i = 0; i < height-3; i++) for(j = 0; j < width; j++)for(k = 0; k < channels; k++)
	{
		// x : template,			x' : registered
		// i:=y direction , j:x direction
		// out(x') = in(inv(H)x')
		// i, j : x', ii, jj : x, x = invHx'
		x1 = invh[0] * j + invh[1] * i + invh[2];
		x2 = invh[3] * j + invh[4] * i + invh[5];
		x3 = invh[6] * j + invh[7] * i + invh[8];
		ii = min(height - 1, max(0, (int)(x2 / x3)));
		jj = min(width - 1, max(0, (int)(x1 / x3)));
		// make borders zero
		if(ii == 0 || ii == height -1 || jj == 0 || jj == width - 1)
			outData[i*step + j*channels + k] = 0;
		else
			outData[i*step + j*channels + k]= inData[ii*step + jj*channels + k];

	}
}

//##########################################################################################################

// out(Hx) = in(x)	=> Given input image x and H, then result is (Hx)
// If you have Input image x and H, it is better to use ApplyHomography function rather than
// ApplyInvHomography in order to get rid of black lines (i.e. interpolation in some sense)

void ApplyInvHomography(IplImage *inImage, IplImage *outImage, CvMat *H)
{
	int i, j, k;
	// get the input image data
	int height = inImage->height;
	int width = inImage->width;
```

```
        int step = inImage->widthStep;
        int channels = inImage->nChannels;
        uchar *inData = (uchar *)inImage->imageData;
        uchar *outData = (uchar *)outImage->imageData;

        float h[9];
        CvMat2Array(H, h, 3, 3);
        int ii, jj;
        float x1, x2, x3;
        for(i = 0; i < height-3; i++) for(j = 0; j < width; j++) for(k = 0; k < channels; k++)
        {
                // i, j : x, ii, jj : x', x' = Hx
                // x : template, x' : registered
                // out(Hx) = in(x)
                x1 = h[0] * j + h[1] * i + h[2];
                x2 = h[3] * j + h[4] * i + h[5];
                x3 = h[6] * j + h[7] * i + h[8];
                ii = min(height - 1, max(0, (int)(x2 / x3)));
                jj = min(width - 1, max(0, (int)(x1 / x3)));
                outData[ii*step + jj*channels + k]= inData[i*step + j*channels + k];
        }
}
//##########################################################################################

void CombineThreeImages(IplImage *image1, IplImage *image2,IplImage *image3, IplImage *outImage)
{
        int i, j, k;
        uchar *outImageData = 0, *image1Data = 0, *image2Data = 0 , *image3Data = 0;
        int height = image1->height;
        int width = image1->width;
        int step = image1->widthStep;
        int channels = image1->nChannels;
        int outWidth = outImage->width;
        int outHeight = outImage->height;
        int outStep = outImage->widthStep;
        if(outWidth == width * 3 && outHeight == height){}
        else
            if(outWidth == width && outHeight == height * 3){}
            else{
                   printf("image combining error\n");
                   exit(0);
                }
        outImageData = (uchar *)outImage->imageData;
        image1Data = (uchar *)image1->imageData;
        image2Data = (uchar *)image2->imageData;
        image3Data = (uchar *)image3->imageData;

        for(i = 0; i < outHeight; i++) for(j = 0; j < outWidth; j++)for(k = 0; k < channels; k++)
        {
            if(i < height && j < width)
                outImageData[i*outStep + j*channels + k]= image1Data[i*step + j*channels + k];
            else
            if(i < 2*height && j < width)
              outImageData[i*outStep + j*channels + k]= image2Data[(i-height)*step + j*channels + k];
            else
            if(i < height && j <2*width)
               outImageData[i*outStep + j*channels + k]= image2Data[i*step + (j-width)*channels + k];
            if(i >= (2*height) && i < 3*height &&j < width)
              outImageData[i*outStep + j*channels + k]= image3Data[(i-(2*height))*step + j*channels + k];
            else
            if(i < height && j >= (2*width)&& j < 3*width)
               outImageData[i*outStep + j*channels + k]= image3Data[i*step + (j-(2*width))*channels + k];
        }
}
```

```
//***************************************************************************************************
// Harris.cpp                        Due: Thursday October 16, 2008
// detect corner points of input image
//
// Prepared By: Khalil Yousef
// Computer Vision ECE 661, Prof. Avi Kak
//***************************************************************************************************

#include "control.h"

void HarrisCornerDectect(IplImage *inImage,IplImage *cornerMap)
{
        int i, j, k;
        int height, width, channels, step;
        height = inImage->height;
        width = inImage->width;
        channels = inImage->nChannels;
        step = inImage->widthStep;

        // define images that holds derivatives
        IplImage *tempImage = 0,*dxdxImg = 0, *dxdyImg = 0, *dydyImg = 0;
        float *dxdxData = 0, *dxdyData = 0, *dydyData = 0;
        float dxdx, dxdy, dydy;

        // Sobel derivative masks
        float dx[9] = {-1, 0, 1, -2, 0, 2, -1, 0, 1};
        float dy[9] = {1, 2, 1, 0, 0, 0, -1, -2, -1};
        float sMask[9] = {1, 1, 1, 1, 1, 1, 1, 1, 1};
        CvMat *Dx = cvCreateMat(3, 3, CV_32FC1); Array2CvMat(dx, Dx, 3, 3);
        CvMat *Dy = cvCreateMat(3, 3, CV_32FC1); Array2CvMat(dy, Dy, 3, 3);
        CvMat *window = cvCreateMat(3, 3, CV_32FC1);Array2CvMat(sMask, window, 3, 3);

        // Creat Derivative Images
        tempImage = cvCreateImage(cvSize(width, height), IPL_DEPTH_8U, channels);
        dxdxImg = cvCreateImage(cvSize(width, height), IPL_DEPTH_32F, channels);
        dxdyImg = cvCreateImage(cvSize(width, height), IPL_DEPTH_32F, channels);
        dydyImg = cvCreateImage(cvSize(width, height), IPL_DEPTH_32F, channels);
        dxdxData = (float *)dxdxImg->imageData;
        dxdyData = (float *)dxdyImg->imageData;
        dydyData = (float *)dydyImg->imageData;

        // Gaussian Smoothing to reduce noise
        cvSmooth(inImage, tempImage, CV_GAUSSIAN, 3, 0, 0);

        Sobel(tempImage, dxdxImg, Dx, Dx);
        Sobel(tempImage, dydyImg, Dy, Dy);
        Sobel(tempImage, dxdyImg, Dx, Dy);
        cvReleaseImage(&tempImage);

        //========================
        // Apply Harriss Algorithm
        //========================

        IplImage *eigenvalueImage = 0;
        eigenvalueImage = cvCreateImage(cvSize(width, height), IPL_DEPTH_32F, 1);
        float *eigenData = 0;
        eigenData = (float *)eigenvalueImage->imageData;
        int eigStep = cornerMap->widthStep;
        double slambda;
        float lambda1, lambda2, lambda3;
        CvMat *G = cvCreateMat(2, 2, CV_32FC1); // Harris Matrix
        CvMat *q = cvCreateMat(2, 2, CV_32FC1); // eigenvector of G
        CvMat *lambda = cvCreateMat(2, 1, CV_32FC1); // eigenvalue of G

        //    [sdxdx sdxdy]
        // G =[sdxdy sdydy]  summed by window to construct the matrix

        // Eigen Value Method
        // find small eigenvalues for each pixel
        for(i = 0; i < height; i++)
        {
```

```
			for(j = 0; j < width; j++)
			{
				for(k = 0; k < channels; k++)
				{
					dxdx = dxdxData[i*step + j*channels + k];
					dxdy = dxdyData[i*step + j*channels + k];
					dydy = dydyData[i*step + j*channels + k];

					cvmSet(G, 0, 0, dxdx);		cvmSet(G, 0, 1, dxdy);
					cvmSet(G, 1, 0, dxdy); cvmSet(G, 1, 1, dydy);

					// Schur Decomposition
					cvEigenVV(G, q, lambda);
					if(channels == 3)
					{
						if(k == 0)
							lambda1 = float(cvmGet(lambda, 1, 0)); // lambda for B
						else if(k == 1)
							lambda2 = float(cvmGet(lambda, 1, 0)); // lambda for G
						else
							lambda3 = float(cvmGet(lambda, 1, 0)); // lambda for R
					}
					else
					{
						// channels == 1
						lambda1 = float(cvmGet(lambda, 1, 0));
					}
				}
				if(channels == 3)
					slambda = pow(pow(lambda1,2) + pow(lambda1,2) + pow(lambda1,2), .5);
				else
					slambda = lambda1;
				eigenData[i*eigStep + j] = float(slambda / HARRIS_THRESHOLD);
			}
		}
		// fine local maximum corner points
		FindLocalMaxPoint(eigenvalueImage, cornerMap, RANGE_NEIGHBOR);

		// release images
		cvReleaseImage(&tempImage);
		cvReleaseImage(&dxdxImg);
		cvReleaseImage(&dxdyImg);
		cvReleaseImage(&dydyImg);
		cvReleaseImage(&eigenvalueImage);
		// release matrices
		cvReleaseMat(&Dx); cvReleaseMat(&Dy);
		cvReleaseMat(&window);
		cvReleaseMat(&G); cvReleaseMat(&q); cvReleaseMat(&lambda);
}


void Sobel(IplImage *inImage, IplImage *outImage,CvMat *kernel1st, CvMat *kernel2nd)
{
		int i, j, k;
		IplImage *f1Image = 0, *f2Image = 0, *tempImage = 0;
		float *f1Data = 0, *f2Data = 0, *outData;
		int height, width, step, channels;
		// create the output image
		height = inImage->height;
		width = inImage->width;
		channels = inImage->nChannels;
		step = inImage->widthStep;

		f1Image = cvCreateImage(cvSize(width, height), IPL_DEPTH_32F, channels);
		f2Image = cvCreateImage(cvSize(width, height), IPL_DEPTH_32F, channels);
		tempImage = cvCreateImage(cvSize(width, height), IPL_DEPTH_32F, channels);
		f1Data = (float *)f1Image->imageData;
		f2Data = (float *)f2Image->imageData;
		outData = (float *)outImage->imageData;

		CvImageCopyUchar2Float(inImage, tempImage); // copy input image to float precision image
```

```
        cvFilter2D(tempImage, f1Image, kernel1st, cvPoint(0, 0));
        cvFilter2D(tempImage, f2Image, kernel2nd, cvPoint(0, 0));
        for(i = 0; i < height; i++) for(j = 0; j < width; j++) for(k = 0; k < channels; k++)
                outData[i*step + j*channels + k]= f1Data[i*step + j*channels + k]* f2Data[i*step +
                j*channels + k];
        cvCopyImage (outImage, tempImage);
        // Smoothing
        cvSmooth(tempImage, outImage, WINDOW_OPTION, PARAM1, PARAM2);
        cvReleaseImage(&tempImage);
        cvReleaseImage(&f1Image);
        cvReleaseImage(&f2Image);
}


void FindLocalMaxPoint(IplImage *inImage, IplImage *map, int range)
{
        int r, sum, numOfNeighbor;
        int i, j, ii, jj, posI, posJ;
        float current;
        float *inData = 0;
        uchar *mapData = 0;
        int height = inImage->height;
        int width = inImage->width;
        int step = map->widthStep;
        r = range / 2;
        numOfNeighbor = (2*r + 1) * (2*r + 1);
        inData = (float *)inImage->imageData;
        mapData = (uchar *)map->imageData;
        for(i = 0; i < height; i++)
        {
                for(j = 0; j < width; j++)
                {
                        current = inData[i*step + j];
                        if(current < 1)
                                mapData[i*step + j] = false;
                        else
                        {
                                // check neighbors
                                sum = 0;
                                for(ii = -r; ii <= r; ii++)for(jj = -r; jj <= r; jj++)
                                {
                                        posI = min(max((i+ii), 0), height - 1);
                                        posJ = min(max((j+jj), 0), width - 1);
                                        sum += (current >= inData[posI*step + posJ]);
                                }
                                if(sum == numOfNeighbor)
                                        mapData[i*step + j] = 1;
                                else
                                        mapData[i*step + j] = 0;
                        }
                }
        }
}

void HarrisResults(IplImage *tempImg, IplImage *regImg,IplImage *cornerMap1, IplImage *cornerMap2)
{
        IplImage *outImage1 = 0, *outImage2 = 0, *outImage3 = 0;
        int height, width, channels;

        height = tempImg->height;
        width = tempImg->width;
        channels = tempImg->nChannels;
        outImage1 = cvCreateImage(cvSize(width, height), IPL_DEPTH_8U, channels);
        outImage2 = cvCreateImage(cvSize(width, height), IPL_DEPTH_8U, channels);
        cvCopyImage(tempImg, outImage1);
        cvCopyImage(regImg, outImage2);
        MarkCornerPoints(outImage1, cornerMap1);
        MarkCornerPoints(outImage2, cornerMap2);

        if(OUTPUT_MODE == 1) // horizontal
                outImage3 = cvCreateImage(cvSize(width * 2, height), IPL_DEPTH_8U, channels);
```

```
        else                                // vertical
                outImage3 = cvCreateImage(cvSize(width, height*2), IPL_DEPTH_8U, channels);
        CombineTwoImages(outImage1, outImage2, outImage3);

        // display the result image

        cvNamedWindow("output image", CV_WINDOW_AUTOSIZE);
        cvShowImage("output image", outImage3);
        cvWaitKey(0);
        cvDestroyWindow("output image");

        // write output image
        WriteImage(outImage1, "result corner1.jpg");
        WriteImage(outImage2, "result corner2.jpg");
        WriteImage(outImage3, "result_Harris_Corner.jpg");
        CombineTwoImages(tempImg, regImg, outImage3);
        WriteImage(outImage3, "inputs.jpg");
        cvReleaseImage(&outImage1);
        cvReleaseImage(&outImage2);
        cvReleaseImage(&outImage3);
        //printf("done\n");
}




//*************************************************************************************************
// matchNCC.cpp                              Due: Thursday October 16, 2008
// 1. detect corner points of 2 input images.
// 2. using the corner points, finds correspondences of the 2 images or Image registration
//
// Prepared By: Khalil Yousef
// Computer Vision ECE 661, Prof. Avi Kak
//*************************************************************************************************

#include "control.h"

// Matching based on NCC value (block1, block2)> threhold;

float NCC(IplImage *block1, IplImage *block2)
{
        int i, j, k;
        uchar *block1Data = 0, *block2Data = 0;
        int height = block1->height;
        int width = block1->width;
        int channels = block1->nChannels;
        int step = block1->widthStep;
        float mBlock1[3];float mBlock2[3];
        float varB1[3];        float varB2[3];
        double numerTerm[3];   double denomTerm[3];
        float ncc = 0;
        block1Data = (uchar *)block1->imageData;
        block2Data = (uchar *)block2->imageData;
        for(k = 0; k < channels; k++)
        {
                mBlock1[k] = 0; mBlock2[k] = 0;varB1[k] = 0;varB2[k] = 0;    numerTerm[k] = 0;
        }

        // calculate mean values
        for(i = 0; i < height; i++)    for(j = 0; j < width; j++)for(k = 0; k < channels; k++)
        {
                mBlock1[k] += (float)block1Data[i*step + j*channels + k];
                mBlock2[k] += (float)block2Data[i*step + j*channels + k];
        }

        // Make all channels block images means zero
        for(k = 0; k < channels; k++)
        {
                mBlock1[k] = mBlock1[k] / (height * width);
                mBlock2[k] = mBlock2[k] / (height * width);
        }
```

```
        for(i = 0; i < height; i++)for(j = 0; j < width; j++)for(k = 0; k < channels; k++)
        {
            numerTerm[k] += ((float)block1Data[i*step + j*channels + k]- mBlock1[k])*
                            ((float)block2Data[i*step + j*channels + k]- mBlock2[k]);
            varB1[k] += float(pow(((float)block1Data[i*step + j*channels + k]- mBlock1[k]), 2));
            varB2[k] += float(pow(((float)block2Data[i*step + j*channels + k]- mBlock2[k]), 2));

        }

        for(k = 0; k < channels; k++)
        {
                denomTerm[k] = pow(varB1[k]*varB2[k], 0.5);
                if(denomTerm[k] == 0)
                        ncc += 0;
                else
                        ncc += float(numerTerm[k] / denomTerm[k]);
        }

        // we can calculate NCC for color image blocks as the average of each color NCCs
        ncc = ncc / channels;
        return(ncc);
}

void match_NCC(IplImage *tempImg, IplImage *regImg,IplImage *tempCMap, IplImage *regCMap,CorspMap
*corspMap)
{
        int height, width, channels, mapStep;
        int i, j;
        int corspPosI = 0,corspPosJ = 0;
        int searchRange = SEARCH_RANGE;

        int blockHeight = WINDOWSIZE_NCC;
        int blockWidth = WINDOWSIZE_NCC;

        IplImage *tempBlock = 0;
        uchar *cornerMapData = 0;

        channels = tempImg->nChannels;
        mapStep = tempCMap->widthStep;
        height = tempImg->height;
        width = tempImg->width;

        tempBlock = cvCreateImage(cvSize(blockWidth, blockHeight), IPL_DEPTH_8U, channels);

        InitializeImage(tempBlock);
        cornerMapData = (uchar *)tempCMap->imageData;

        for(i = 0; i < height; i++)for(j = 0; j < width; j++)
        {
                if(cornerMapData[i*mapStep + j] == 1)
                {
                        MakeImageBlock(tempBlock, tempImg, i, j);

                        if(NCCcheck(regImg, tempBlock, regCMap, i, j,&corspPosI, &corspPosJ, searchRange))
                        {
                                UpdateCorrespMap(corspMap, i, j, corspPosI, corspPosJ);
                        }
                }
        }

        cvReleaseImage(&tempBlock);
}

bool NCCcheck(IplImage *regImg, IplImage *tempBlock,IplImage *regCMap,int cornerPosI, int cornerPosJ,int
*pCorspPosI, int *pCorspPosJ,int searchRange)
{
        IplImage *regBlock = 0;
        int channels, blockHeight, blockWidth;
        int i, j;
        float value;
```

```
	float maxNcc = float(NCC_THRESHOLD);
	int r = searchRange / 2;
	uchar *cornerMapData = 0;
	int mapStep = regCMap->widthStep;
	int height, width, iBegin, jBegin, iEnd, jEnd;
	height = regImg->height;
	width = regImg->width;
	blockHeight = tempBlock->height;
	blockWidth = tempBlock->width;
	channels = tempBlock->nChannels;
	regBlock = cvCreateImage(cvSize(blockWidth, blockHeight), IPL_DEPTH_8U, channels);
	InitializeImage(regBlock);

	cornerMapData = (uchar *)regCMap->imageData;

	iBegin = max(cornerPosI - r, 0);
	jBegin = max(cornerPosJ - r, 0);
	iEnd = min(cornerPosI + r + 1, height - 1);
	jEnd = min(cornerPosJ + r + 1, width - 1) ;
	for(i = iBegin; i < iEnd; i++)for(j = jBegin; j < jEnd; j++)
	{
		if(cornerMapData[i*mapStep + j] == 1) // corner point
		{
			MakeImageBlock(regBlock, regImg, i, j);
			value = NCC(regBlock, tempBlock);// calculate ncc
			if(value > maxNcc) // maximum NCC
			{
					maxNcc = value;
					*pCorspPosI = i;
					*pCorspPosJ = j;
			}
		}
	}
	cvReleaseImage(&regBlock);
	// if no corner in search range, do nothing
	if(maxNcc == NCC_THRESHOLD)
		return(false);
	else
		return(true);
}

void NCCResults(IplImage *tempImg, IplImage *regImg, CorspMap *corspMap,char *outputName, int NCCstep)
{
	IplImage *outImage = 0;
	int height, width, channels;
	height = tempImg->height;
	width = tempImg->width;
	channels = tempImg->nChannels;
	if(OUTPUT_MODE == 1) // horizontal
		outImage = cvCreateImage(cvSize(width * 2, height), IPL_DEPTH_8U, channels);
	else				// vertical
		outImage = cvCreateImage(cvSize(width, height  * 2), IPL_DEPTH_8U, channels);
	DrawCorrespLine(tempImg, regImg, outImage, corspMap);
	if(NCCstep ==1)
	{
		cvNamedWindow("output image NCC", CV_WINDOW_AUTOSIZE);
		cvShowImage("output image NCC", outImage);
		cvWaitKey(0);
		cvDestroyWindow("output image NCC");
	}
	else
	{
		cvNamedWindow("Inliers Matching between Template Image and registered Image",
		 CV_WINDOW_AUTOSIZE);
		cvShowImage("Inliers Matching between Template Image and registered Image", outImage);
		cvWaitKey(0);
		cvDestroyWindow("Inliers Matching between Template Image and registered Image");
	}
	WriteImage(outImage, outputName);
	cvReleaseImage(&outImage);
}
```

```
//*****************************************************************************************************
// matchSSD.cpp                              Due: Thursday October 16, 2008
// 1. detect corner points of 2 input images.
// 2. using the corner points, finds correspondences of the 2 images or Image registration
//
// Prepared By: Khalil Yousef
// Computer Vision ECE 661, Prof. Avi Kak
//*****************************************************************************************************

#include "control.h"

// Matching based on SSD value (block1, ALL registered image blocks2) < SSDthrehold;

void match_SSD(IplImage *tempImg, IplImage *regImg,IplImage *tempCMap, IplImage *regCMap,CorspMap
*corspMap)
{
        int height, width, channels, mapStep;
        int i, j;
        int corspPosI = 0,corspPosJ = 0;

        int blockHeight = WINDOWSIZE_SSD;
        int blockWidth = WINDOWSIZE_SSD;

        IplImage *tempBlock = 0;
        uchar *cornerMapData = 0;

        channels = tempImg->nChannels;
        mapStep = tempCMap->widthStep;
        height = tempImg->height;
        width = tempImg->width;

        tempBlock = cvCreateImage(cvSize(blockWidth, blockHeight), IPL_DEPTH_8U, channels);

        InitializeImage(tempBlock);
        cornerMapData = (uchar *)tempCMap->imageData;
        //int counter=0;

        for(i = 0; i < height; i++)for(j = 0; j < width; j++)
        {
                if(cornerMapData[i*mapStep + j] == 1)        // True corner for template image
                {
                        // for each template corner creat a window block around it
                        // and find the sum of squared difference between this template window block
                        // and all possible registered window blocks around registered corner points
                        // choose the min sum value

                        MakeImageBlock(tempBlock, tempImg, i, j);
                        if(SSDcheck(regImg, tempBlock, regCMap, i, j,&corspPosI, &corspPosJ))
                        {
                                //counter++;
                                UpdateCorrespMap(corspMap, i, j, corspPosI, corspPosJ);
                        }
                }
        }

        cvReleaseImage(&tempBlock);
        //printf("counter = %d\n",counter);
}


bool SSDcheck(IplImage *regImg, IplImage *tempBlock,IplImage *regCMap,int cornerPosI, int cornerPosJ,int
*pCorspPosI, int *pCorspPosJ)
{
        IplImage *regBlock = 0;
        int channels, blockHeight, blockWidth;
        int i, j;
        float value, minSSD = SSD_THRESHOLD;

        uchar *cornerMapData = 0;
        int mapStep = regCMap->widthStep;
```

```
	int height, width;

	height = regImg->height;
	width = regImg->width;
	blockHeight = tempBlock->height;
	blockWidth = tempBlock->width;
	channels = tempBlock->nChannels;

	regBlock = cvCreateImage(cvSize(blockWidth, blockHeight), IPL_DEPTH_8U, channels);
	InitializeImage(regBlock);

	cornerMapData = (uchar *)regCMap->imageData;

	// search all terget image blocks centerd at their corresponding corners
	// so search region is whole image
	for(i = 0; i < height; i++)for(j = 0; j < width; j++)
	{
		if(cornerMapData[i*mapStep + j] == 1) // corner point at registered image
		{
			MakeImageBlock(regBlock, regImg, i, j);
			value = SSD(regBlock, tempBlock);// calculate ncc
			if(value < minSSD) // maximum NCC
			{
				minSSD = value;
				*pCorspPosI = i;
				*pCorspPosJ = j;
			}
		}
	}
	cvReleaseImage(&regBlock);
	// if no corner in search range, do nothing
	if(minSSD == SSD_THRESHOLD)
		return(false);
	else
		return(true);
}

float SSD(IplImage *block1, IplImage *block2)
{
	int i, j, k;
	uchar *block1Data = 0, *block2Data = 0;
	int height = block1->height;
	int width = block1->width;
	int channels = block1->nChannels;
	int step = block1->widthStep;
	float ssd = 0;
	block1Data = (uchar *)block1->imageData;
	block2Data = (uchar *)block2->imageData;
	float sdifference[3];

	for(k = 0; k < channels; k++)
		sdifference[k] = 0;

	for(i = 0; i < height; i++)for(j = 0; j < width; j++)for(k = 0; k < channels; k++)
		sdifference[k] += float(pow(((float)block1Data[i*step + j*channels + k]-
				(float)block2Data[i*step + j*channels + k]),2));

	for(k = 0; k < channels; k++)
		ssd += float(sdifference[k]);

	// we can calculate ssd for color image blocks as the average of each color NCCs
	ssd = float(ssd / channels);
	//printf("SSD = %f\n",ssd);
	return(ssd);
}

void SSDResults(IplImage *tempImg, IplImage *regImg, CorspMap *corspMap,char *outputName)
{
	IplImage *outImage = 0;
	int height, width, channels;
	height = tempImg->height;
```

```
        width = tempImg->width;
        channels = tempImg->nChannels;
        if(OUTPUT_MODE == 1) // horizontal
                outImage = cvCreateImage(cvSize(width * 2, height), IPL_DEPTH_8U, channels);
        else                            //vertical
                outImage = cvCreateImage(cvSize(width, height  * 2), IPL_DEPTH_8U, channels);
        DrawCorrespLine(tempImg, regImg, outImage, corspMap);
        cvNamedWindow("output image SSD", CV_WINDOW_AUTOSIZE);
        cvShowImage("output image SSD", outImage);
        cvWaitKey(0);
        cvDestroyWindow("output image SSD");
        WriteImage(outImage, outputName);
        cvReleaseImage(&outImage);
}



//*****************************************************************************************************
// RANSAC.cpp                        Due: Thursday October 16, 2008
//
// Prepared By: Khalil Yousef
// Computer Vision ECE 661, Prof. Avi Kak
//****************************************************************************************************

#include "control.h"

//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<RANSAC>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
//<<<<<<<<<<<<<<<<<<<Function is to apply RANSAC to estimate homography>>>>>>>>>>>>>>>>>>

void RANSAC(CorspMap *corspMap, CorspMap *inlierMap,CvMat *H)
{
        float templatePositions[MaxNumOfCorresp][2], registeredPositions[MaxNumOfCorresp][2];
        int numOfCorresp = MinNumOfCorresp;   // s =4
        int index;                            // Position Index to be used in random selection of corners
        int trials = 0;                       // Initialization of RANSAC Trials
        int numOfInliers, totalNumOfPoints;   // Termenology Used in Algorithm 4.5 in the book
        int maxNumOfInliers = 1;              // Initial num of inlier corners
        float eps =e;                         // e:=Probability of data element is inlier = E=0.5
        int N = MaxRANSACtrialNum;            // MAX number of trials of RANSAC
        float outlierProb;                    // w:= Probability that a randomly selected
                                              // data element is oulier

        CorspMap estInlierMap;                // define a list of candidate inliers points
        CvMat *H_est = cvCreateMat(3, 3, CV_32FC1);

        while(N > trials)              // Start RANSAC trials
        {
                // pick randomly 4 corresponding points or data elements from the Corner list
                // obtained from NCC
                for(int i = 0; i < numOfCorresp; i++)
                {
                        index = rand() % corspMap->len; // select random positions of corner points
                        // I -> y, J -> x
                        templatePositions[i][1] = float(corspMap->tempImgPositionI[index]);
                        templatePositions[i][0] = float(corspMap->tempImgPositionJ[index]);
                        registeredPositions[i][1] = float(corspMap->regImgPositionI[index]);
                        registeredPositions[i][0] = float(corspMap->regImgPositionJ[index]);
                }

                // check whether samples are good or not.
                // if the selected samples are good, then do homography estimation
                // else reselect samples.
                // i.e check the selected correspondencies if they are colinear (want no any 3 points are
                // colinear)
                // Key: Any two points are colinear so they form a line
                // Thus take dot product between the line and the third point =>
                // dot product is zero or within margin threshold if they are colinear

                if(CollinearityCheck(templatePositions, numOfCorresp) &&
                CollinearityCheck(registeredPositions, numOfCorresp))
                {
```

```
                    // compute homography - returned is H which is an initial estimated Homography
                    ComputeHomography(templatePositions, registeredPositions, numOfCorresp, H_est);

                    // calculate the distance for each correspondences
                    // compute the number of inliers
                    InitializeCorspMap(&estInlierMap);
                    DistanceMeasure(H_est, corspMap, &estInlierMap);

                    // choose H with the largest number of inliears
                    // Apply Threshold on the Number of inliers
                    // if below threshold break and start loop again
                    // if above threshold => restimate homography and terminate, we are done!
                    // Algorithm 4.4 Page 118

                    numOfInliers = estInlierMap.len;
                    if(numOfInliers >= maxNumOfInliers)
                    {
                           maxNumOfInliers = numOfInliers;
                           CopyCorspMap(inlierMap, &estInlierMap);
                           cvCopy(H_est, H, 0);
                    }
                    // adaptive algorithm for determining the number of RANSAC samples
                    // Equation 4.18 page 119 in the book
                    // Also from textbook algorithm 4.5 and Algorithm 4.6 pages 121, 123
                    totalNumOfPoints = corspMap->len;
                    outlierProb = 1 - ((float)maxNumOfInliers / (float)totalNumOfPoints);
                    eps=min(eps,outlierProb);
                    N = int(log(1 - p) / log(1 - pow((1 - eps), numOfCorresp)));
                    trials = trials + 1;

                }

        }
        printf("Number of inliers = %d\n",inlierMap->len );
        printf("N=%d\t eps=%f\t trial=%d\n",N,eps,trials);
}


//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<CollinearityCheck>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
//<<<<<<<<<<<<<<<This function checks colinearity of all given points>>>>>>>>>>>>>>>>>>>>

bool CollinearityCheck(float points[][2], int numOfPoints)
{
        bool colinear = false;
        int i, j, k;
        CvMat *p1 = cvCreateMat(3, 1, CV_32FC1);
        CvMat *p2 = cvCreateMat(3, 1, CV_32FC1);
        CvMat *p3 = cvCreateMat(3, 1, CV_32FC1);
        i = 0;  j = i + 1;     k = j + 1;

        // check colinearity recursively for all combinations of points as triples
        while(true)
        {
               // set point vectors in homogenious representation
               cvmSet(p1, 0, 0, points[i][0]);              //Initially i=0
               cvmSet(p1, 1, 0, points[i][1]);
               cvmSet(p1, 2, 0, 1);

               cvmSet(p2, 0, 0, points[j][0]);              //Initially j=1
               cvmSet(p2, 1, 0, points[j][1]);
               cvmSet(p2, 2, 0, 1);

               cvmSet(p3, 0, 0, points[k][0]);
               cvmSet(p3, 1, 0, points[k][1]);              //Initially k=2
               cvmSet(p3, 2, 0, 1);

               // check linearity
               colinear = IsColinear(p1, p2, p3) || colinear;

               // update point index to check all possibility of 3 points (4 choose 3=4)
               // cases in recuirsive order
```

```
		// L(p1,p2),p3		i=0, j=1, k=2
		// L(p1,p2),p4		i=0, j=1, k=3
		// L(p1,p3),p4		i=0, j=2, k=3
		// L(p2,p3),p4		i=1, j=2, k=3

		if(k < numOfPoints - 1)			// select forth point as p3 i.e. make k=2+1=3
			k += 1;
		else
		{
			if(j < numOfPoints - 2)
			{
				j += 1;		k = j + 1;
			}
			else
			{
				if(i < numOfPoints - 3)
				{
					i += 1;		j = i + 1;		k = j + 1;
				}
				else
					break;
			}
		}
	}
	return(!colinear);
}

//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<IsColinear>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

//<<<<This function checks the colinearity of the given 3 points p1, p2, and p3>>>>>>>>>>
//<<<<<<<<<<<<<<<<<<<<<<<If these are colinear, it returns True>>>>>>>>>>>>>>>>>>>>>>>>>

bool IsColinear(CvMat *p1, CvMat *p2, CvMat *p3)
{
	float ColinDistance;
	CvMat *lineP1P2 = cvCreateMat(3, 1, CV_32FC1);

	// Form colinear line becasue any two points are colinear
	cvCrossProduct(p1, p2, lineP1P2);

	// Check third point if it on the colinear line or not
	// For a point to be on the line dot product (distance) is zero or within margin
	ColinDistance = float(cvDotProduct(lineP1P2, p3));

	// release matrices
	cvReleaseMat(&lineP1P2);

	//printf("colieniarity distance = %f\n",ColinDistance);

	//					*P3
	//			p1	|	p2
	//		<--*------------*-->
	// points are colinear if:
	//{-COLINEARITY_THRESHOLD < ColinDistance < COLINEARITY_THRESHOLD}

	if((ColinDistance < COLINEARITY_THRESHOLD)&&(ColinDistance > -COLINEARITY_THRESHOLD))
		return(true);		// return yes they are colinear
	else
		return(false);
}


//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ComputeHomography>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// This function calculate the homography, H, using the set of given pairs of correspondences.
// Before computing the homography, data normalization will be performed.
// Then, it solve Ah = 0 using SVD to get H.

void ComputeHomography(float templatePositions[][2], float registeredPositions[][2],int numOfCorresp,
CvMat *H)
{
```

```
int column = 9, row;
float x, y, w, xp, yp, wp;

switch(numOfCorresp)
{
	case 4:				// # of correspondencies is = 4
		row = 3;		// eq 4.1 :
		break;
	case 0:case 1:case 2:case 3:	// # of correspondencies is < 4
		printf("Need more correspondence points! for computing H.\n");
		exit(0);
		break;
	default:			// # of correspondencies is > 4
		row = 2;		// eq 4.3
		break;

}

// normalization
CvMat *T = cvCreateMat(3, 3, CV_32FC1);
CvMat *Tp = cvCreateMat(3, 3, CV_32FC1);

DataNormalization(numOfCorresp, templatePositions, T);
DataNormalization(numOfCorresp, registeredPositions, Tp);

// After Data Normalization we have T, Tp and we have points xx'_vec xx vect
// the relation between them (xx'_vec=Htemp * xx_vec)
// At the end we want (x'_vec = H * x_vec)
// => x'_vec  = H * x_vec = [inv(T')* Htemp * T]* x_vec

// Now what we will find is "Htemp" from (xx'_vec=Htemp * xx_vec) correspondeinces
// with the following termonology xp_vec= Htemp * x_vec

float *a;	//float a[27];
//dynamic array of size (row * column) = 27 for equation 4.1 and 18 for eqaution 4.3;
a=(float *)malloc((row * column)* sizeof *a);
CvMat *A = cvCreateMat(numOfCorresp * row, column, CV_32FC1);

for(int i = 0; i < numOfCorresp; i++)
{
	//Get Homogenious representation for correspondencies
	x = templatePositions[i][0];		xp = registeredPositions[i][0];
	y = templatePositions[i][1];		yp = registeredPositions[i][1];
	w = 1;						wp = 1;

	// set Ai and repeat it for n correspondincies
	// [0, 0, 0, -wp*x, -wp*y, -wp*w, yp*x, yp*y, yp*w]
	// [wp*x, wp*y, wp*w, 0, 0, 0, -xp*x, -xp*y, -xp*w]

	// If row(==2)
	a[0]=0;a[1]=0;a[2]=0;			a[3]=-wp*x;a[4]=-wp*y;a[5]=-wp*w;
	a[6]=yp*x;a[7]=yp*y;a[8]=yp*w;		a[9]=wp*x;a[10]=wp*y;a[11]=wp*w;
	a[12]=0;a[13]=0;a[14]=0;		a[15]=-xp*x;a[16]=-xp*y;a[17]=-xp*w;
	if(row == 3) // add one more equation
	{
		a[18]=-yp*x;a[19]=-yp*y;a[20]=-yp*w;  a[21]=xp*x;a[22]=xp*y;a[23]=xp*w;
		a[24]=0;a[25]=0;a[26]=0;
		// [-yp*x, -yp*y, -yp*w, xp*x, xp*y, xp*w, 0, 0, 0]
	}

	// assemble Ai into a matrix A
	for(int m = 0; m < row; m++)
		for(int n = 0; n < column; n++)
			cvmSet(A, m + i*row, n, a[m*column + n]);
}

// calculate Htemp
float h_temp[9];
CvMat *Htmp = cvCreateMat(3, 3, CV_32FC1);
CvMat *D = cvCreateMat(numOfCorresp*row, column, CV_32FC1);
CvMat *U = cvCreateMat(numOfCorresp*row, numOfCorresp*row, CV_32FC1);
```

```
	CvMat *V = cvCreateMat(column, column, CV_32FC1);
	cvSVD(A, D, U, V);	// A = U D V
	//cvSVD(A, D, U, V, CV_SVD_U_T|CV_SVD_V_T); // A = U D V^T

	// Homography or the solution is the last column of V (9 by 9)
	for(i = 0; i < column; i++)
		h_temp[i] = float(cvmGet(V,i,column-1));

	Array2CvMat(h_temp, Htmp, 3, 3);		//Note: Htmp = Tp * H * invT

	// Compute Homography "H": x'_vec  = H * x_vec = [inv(T')* Htemp * T]* x_vec
	// denormalization Step:  H = invTp * Htmp * T
	CvMat *invTp = cvCreateMat(3, 3, CV_32FC1);
	cvInvert(Tp, invTp);
	CvMat *temp = cvCreateMat(3, 3, CV_32FC1);
	cvMatMul(invTp, Htmp, temp);
	cvMatMul(temp,T, H);

	// release matrices
	cvReleaseMat(&T); cvReleaseMat(&Tp);
	cvReleaseMat(&A); cvReleaseMat(&Htmp);
	cvReleaseMat(&D); cvReleaseMat(&U); cvReleaseMat(&V);
	cvReleaseMat(&invTp); cvReleaseMat(&temp);
}

//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<DataNormalization>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// Perform Isotropic scaling in page 107 in the book
// This function normalizes x and returns the similarity transform, T.
// The centroid of x will be transformed into (0,0).
// The average distance of normalized x will be sqrt(2).

// Basically there are two different methods of data normalization.
// The first method [26. Z. Zhang , Determining the epipolar geometry and its uncertainty:
// a review. International Journal of Computer Vision 27 2 (1998), pp. 161–198.
// View Record in Scopus | Cited By in Scopus (0)26] normalizes the data between [-1,1].
// The second was proposed by Hartley [42] and is based on two transformations.
// First, the points are translated so that their centroid is placed at the origin.
// Then, the points are scaled so that the mean of the distances of the points to the
// origin is Image.
// It has been proved that Hartley's method gives more accurate results than normalizing
// between [-1,1].
// 1- All points are translated so that thier centroid is transformed to origin (0,0)
// 2- All points are scaled so that the average distance between every point and the
//    origin is sqrt(2)
// Main purpose is to make sure that A has rank 8

void DataNormalization(int numOfPositions, float positions[][2], CvMat *T)
{
	int i;
	float sumI = 0, sumJ = 0, meanI = 0, meanJ = 0;
	float squareDist = 0, sumDist = 0, meanDist = 0;
	float x, y,w, xx, yy, ww;

	// calculate the centroid
	for(i = 0; i < numOfPositions; i++)
	{
		sumI += positions[i][0];
		sumJ += positions[i][1];
	}

	// Now Centroid of all points (meanI, meanJ) is considered the origin (0,0)
	meanI = sumI / numOfPositions;
	meanJ = sumJ / numOfPositions;

	// calculate the average distance of every point from origin using Frobinius norm
	// ||pointPos - mean||_F= sqrt[(posi - mean_i)^2 + (posj - mean_j)^2]
	// repaeat this process for all points and sum them to take the
	// average distnaces i.e. divide by numOfPositions

	for(i = 0; i < numOfPositions; i++)
```

```
	{
		squareDist += float(pow(positions[i][0] - meanI, 2)+ pow(positions[i][1] - meanJ, 2));

	}

	// final average distances of all points
	meanDist = float(pow(squareDist / (2*numOfPositions),0.5));

	// set the similarity transform;
	float scale = 1/ meanDist;

	// xx_vec = T  * x_vec   for template image corners positions
	// xx'_vec= T' * x'_vec  for registered image corners positions in the second call
	// Want at the end Homography = inv(T')* Htemp * T
	// where xx'_vec = Htemp * xx_vec
	//       x'_vec  = H * x_vec = [inv(T')* Htemp * T]* x_vec

	float t[9] = {scale ,   0  , -scale * meanI, // [x]   [xx]
	                0   , scale, -scale * meanJ  //*[y] = [yy]
	               0   ,   0  ,        1      }; // [1]   [ww]

	Array2CvMat(t, T, 3, 3);

	//Apply data normalization Transformation s.t.Average distance is sqrt(2) from origin
	for(i = 0; i < numOfPositions; i++)
	{
		x = float(positions[i][0]);
		y = float(positions[i][1]);
		w = 1;

		// Homogenious Representation
		xx = float(t[0] * x + t[1] * y + t[2] * w);
		yy = float(t[3] * x + t[4] * y + t[5] * w);
		ww = float(t[6] * x + t[7] * y + t[8] * w);

		// Retrun Back to Physical Representation
		xx = float(xx / ww);
		yy = float(yy / ww);

		positions[i][0] = xx;
		positions[i][1] = yy;
	}
}

//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<DistanceMeasure>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// Calculates distance of data using symmetric transfer error. Then, compute inliers
// that consist with H.

void DistanceMeasure(CvMat *H, CorspMap *corspMap, CorspMap *inlierMap)
{
	int i;
	int x, y,w, xp, yp, wp;
	float xT, yT, wT, xpT, ypT, wpT;
	CvMat *invH = cvCreateMat(3, 3, CV_32FC1);
	float sDist_xT_x, sDist_xpT_xp, sDistSum;
	cvInvert(H, invH);

	// use sum of squared Gaussian variable or distance as distance measure mentioned
	// in the book page 118 and distance threshold is selected based on table 4.2 in page
	// 119 in the book too.
	// I -> y, J -> x             Be careful with indices

	for(i = 0; i < corspMap->len; i++)
	{
		x = corspMap->tempImgPositionJ[i];
		y = corspMap->tempImgPositionI[i];
		w=1;

		// calculate (xpT = H * x) in the template imgae
		xpT = float(cvmGet(H, 0, 0) * x + cvmGet(H, 0, 1) * y+ cvmGet(H, 0, 2)* w);
```

```
			ypT = float(cvmGet(H, 1, 0) * x + cvmGet(H, 1, 1) * y+ cvmGet(H, 1, 2)* w);
			wpT = float(cvmGet(H, 2, 0) * x + cvmGet(H, 2, 1) * y+ cvmGet(H, 2, 2)* w);

			//Physical Representation of estimated registered image corners xpT:Transformed
			xpT = float(xpT / wpT);
			ypT = float(ypT / wpT);

			xp = corspMap->regImgPositionJ[i];
			yp = corspMap->regImgPositionI[i];
			wp=1;

			// calculate (xT = H^(-1) * xp)       in the registered image
			xT = float(cvmGet(invH, 0, 0) * xp + cvmGet(invH, 0, 1) * yp+ cvmGet(invH, 0, 2)* wp);
			yT = float(cvmGet(invH, 1, 0) * xp + cvmGet(invH, 1, 1) * yp+ cvmGet(invH, 1, 2)* wp);
			wT = float(cvmGet(invH, 2, 0) * xp + cvmGet(invH, 2, 1) * yp+ cvmGet(invH, 2, 2)* wp);

			//Physical Representation
			xT = float(xT / wT);
			yT = float(yT / wT);

			// Symmetric Transfer error page 95 in the book inpired from Algorithm 4.6 page 123
			// and page 124
			// calculate the square distance (square of Ecludian Distance between two points)
			sDist_xT_x = float(pow(xT - x, 2) + pow(yT - y, 2));
			sDist_xpT_xp = float(pow(xpT - xp, 2) + pow(ypT - yp, 2));

			sDistSum = float(sDist_xT_x + sDist_xpT_xp);

			if(sDistSum < DISTANCE_THRESHOLD)
			{
				UpdateCorrespMap(inlierMap, y, x, yp, xp);
				// posI -> y, posJ -> x              Be careful with indices
			}
		}
		// release matrices
		cvReleaseMat(&invH);
}

//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<Homhography Refinement>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void HomograhyRefinement(CorspMap *inlierMap, CvMat *H)
{
		int i;
		int numOfCorresp = inlierMap->len;
		float templatePositions[MaxNumOfCorresp][2], registeredPositions[MaxNumOfCorresp][2];
		for(i = 0; i < numOfCorresp; i++)
		{
			// I -> y, J -> x
			templatePositions[i][1] = float(inlierMap->tempImgPositionI[i]);
			templatePositions[i][0] = float(inlierMap->tempImgPositionJ[i]);
			registeredPositions[i][1] = float(inlierMap->regImgPositionI[i]);
			registeredPositions[i][0] = float(inlierMap->regImgPositionJ[i]);
		}
		ComputeHomography(templatePositions, registeredPositions, numOfCorresp, H);

}

//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<Output results>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

void RANSACresults(IplImage *tempImg,IplImage *H_tempImg, IplImage *regImg,char *outputName, int step)
{

		int height, width, channels;
		height = H_tempImg->height;
		width = H_tempImg->width;
		channels = H_tempImg->nChannels;

		IplImage * resultImage;

		// create the result images : 2 images in the same output image
		//if(OUTPUT_MODE == 1) // horizontal
		//	resultImage = cvCreateImage(cvSize(width * 2, height), IPL_DEPTH_8U, channels);
```

```
	//else				//vertical
	//	resultImage = cvCreateImage(cvSize(width, height  * 2), IPL_DEPTH_8U, channels);

		resultImage = cvCreateImage(cvSize((width *3), height), IPL_DEPTH_8U, channels);

	//	resultImage = cvCreateImage(cvSize(width, height  * 3), IPL_DEPTH_8U, channels);

	//CombineTwoImages(H_tempImg, regImg, resultImage);
	CombineThreeImages(tempImg,H_tempImg, regImg, resultImage);

	// display the result image

	if (step ==1)
	{
		cvNamedWindow("Template Image with homography VS. Registered Image", CV_WINDOW_AUTOSIZE);
		cvShowImage("Template Image with homography VS. Registered Image", resultImage);
		cvWaitKey(0);
		cvDestroyWindow("Template Image with homography VS. Registered Image");
	}
	else // step=2
	{
		cvNamedWindow("Template Image with refined homography VS. Registered Image",
		CV_WINDOW_AUTOSIZE);
		cvShowImage("Template Image with refined homography VS. Registered Image", resultImage);
		cvWaitKey(0);
		cvDestroyWindow("Template Image with refined homography VS. Registered Image");
	}

	// write output image
	WriteImage(resultImage, outputName);
	char name[80];
	sprintf(name, "transformed_%s", outputName);
	WriteImage(H_tempImg, name);
	cvReleaseImage(&resultImage);
	cvReleaseImage(&H_tempImg);
}
```