# ECE 661 HW_4

Bharath Kumar Comandur J R

10/02/2012

## 1 Introduction

In this exercise we develop a Harris Corner Detector to extract interest points (such as corners) in a given image. We apply the algorithm on two different images of the same scene and try to establish correspondences between the interest points in the images. We use two metrics namely the NCC (Normalized Cross Correlation) and the SSD (Sum of Squared Differences) to establish correspondences. We also implement a SURF feature extractor to compare with the Harris Corner Detector.

## 2 Harris Corner Detector

### 2.1 Corner detection

To distinguish a corner point in an image, we need to select points where the gray-levels change significantly in at-least two different directions. The detection must be invariant to in-plane rotation of the image. The Harris Corner detector satisfies this invariance property. The steps of the algorithm are as follows

1. We calculate the derivative along the x and y directions using the Sobel operator. The image is convolved with two matrices

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad and \quad \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

for computing the x and y derivatives respectively.

2. A $5\sigma$X$5\sigma$ window is chosen around each pixel and the C matrix is computed as

$$C = \begin{bmatrix} \sum d_x^2 & \sum d_x d_y \\ \sum d_x d_y & \sum d_y^2 \end{bmatrix}$$

   All the pixels in the window are considered for the summation

3. If the point under consideration is a genuine corner, then the C matrix will have full rank. Hence we use the eigenvalues of C to determine if it is a corner point

4. At every pixel, we compute the Cornerstrength as given by the following expression

$$Cornerstrength = \lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2$$

   where k can be chosen as required. $\lambda_1 and\, \lambda_2$ are the eigenvalues of C. In practice, the trace of C equals $(\lambda_1 + \lambda_2)$ and the determinant of C equals $(\lambda_1\lambda_2)$. Hence a SVD of C is not required.

5. We set a threshold for the Cornerstrength to classify a point as a corner or not. Here it must be noted that many corner points might result very close to each other. Hence we use non-maximum suppression or in other words extract the points having a local maxima for Cornerstrength in a suitably sized window and classify them alone as corner points. The Cornerstrength is large for actual corners, negative for edges and has small values for flat regions.

## 2.2 Establishing correspondences between two images

We implement the Harris corner detector on two images of the same scene to extract interest points. Naturally we are interested in determining the invariance of our algorithm to in-plane rotation. More importantly we would like to establish correspondences between the interest points in the two images. We compare the gray levels in a window around a corner point with the gray levels around the corresponding pixel in the other image. We

use two metrics for this, namely the NCC (Normalized Cross Correlation) and the SSD (Sum of Squared Differences).

### 2.2.1 NCC

$$NCC = \frac{\sum\sum(f_1(i.j) - m_1)(f_2(i.j) - m_2)}{\sqrt{[\sum\sum(f_1(i.j) - m_1)^2]\,[\sum\sum(f_2(i.j) - m_2)^2]}}$$

where $f_1$ and $f_2$ refer to the image pixel intensity(gray-level) and $m_1$ and $m_2$are the average of the pixel intensities within a window in the first and second image respectively. NCC is computed between every pair of corresponding corner pixels. We observe that when two interest points have similar feature descriptors, the value of the NCC approaches 1. The NCC ranges between -1 and 1. For each interest point in the first image, we do a brute-force search on the interest points in the second image to maximize the NCC. Two kinds of issues might occur,

1. The Harris Corner Detector need not always detect the same interest point in the two images. So we set a threshold value for the largest NCC and ignore interest points where the largest NCC is less than this threshold value.

2. False correspondences. This occurs at points when the extracted features are not particularly unique. In such cases, it is logical to expect that the ratio of the maximum NCC value to the second largest NCC values is nearly 1. So we define a threshold for this ratio and ignore an interest point if

$$\frac{second maximum\, NCC}{maximum\, NCC} > ratiothresh_{NCC}$$

### 2.2.2 SSD

$$SSD = \sum_i \sum_j |f_1(i,j) - f_2(i,j)|^2$$

where $f_1$ and $f_2$ are the pixel intensities within a suitable window in the first and second image respectively. SSD is computed between every pair of corresponding corner pixels.

The SSD is extremely small for similar interest points. Similar to NCC, for each feature in the first image, we do a brute-force search on the interest points in the second

image to minimize the SSD. We again can face 2 problems

1. The Harris Corner Detector need not always detect the same interest point in the two images. So we set a threshold value for the smallest SSD and ignore interest points where the smallest SSD is greater than a threshold value.

2. False correspondences. This occurs at points when the extracted features are not particularly unique. In such cases, it is logical to expect that the ratio of the minimum SSD value to the second smallest SSD values is nearly 1. So we define a threshold for this ratio and ignore an interest point if

$$\frac{minimum\,SDD}{secondminimum\,SSD} > ratiothresh_{SSD}$$

## 2.3 Comparison of NCC and SSD

From the results we can observe the following

1. SSD involves less computations to calculate and is a more sensitive measure to small changes in intensity values between images.

2. Due to a variety of factors such as ambient conditions, there can be scaling and shifts in image intensities. The NCC remains invariant to such affine transformations, however the SSD does not exhibit such a behavior.

## 2.4 Parameters used

I have tuned the parameters in such a way to obtain sufficient number of correspondences between images and at the same time preserve accuracy of correspondences as much as possible.

Table 1 explains the symbols used in Table 2.

| Symbol | What it Denotes |
|---|---|
| $W_s$ | Dimension of window of Sobel operator |
| $W_H$ | Dimension of window for Harris Corner detection |
| $W_r$ | Dimension of window for non-maximum suppression |
| $W_{SSD}$ | Dimension of window for feature descriptor for SSD |
| $W_{NCC}$ | Dimension of window for feature descriptor for NCC |
| $T_{SSD}$ | Threshold for SSD |
| $r_{NCC}$ | Threshold for ratio of $\frac{secondmaximum\,NCC}{maximum\,NCC}$ |
| $T_R$ | Threshold for the corner strength |
| $r_{SSD}$ | Threshold for ratio of $\frac{minimum\,SDD}{secondminimum\,SSD}$ |
| $T_{NCC}$ | Threshold for NCC |
| $k$ | Cornerstrength parameter |

Table 1: Symbols and Description

Table 2 indicates the values of parameters used in simulations.

| Picture | $W_s$ | $W_H$ | $W_r$ | $W_{SSD}$ | $W_{NCC}$ | $T_{SSD}$ | $T_{NCC}$ | $T_R$ | $r_{SSD}$ | $r_{NCC}$ | $k$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| sample-a | 3 | 5 | 5 | 45 | 13 | 1250 | 0.7 | 5e+9 | 0.92 | 0.83 | 0.04 |
| sample-b | 3 | 5 | 5 | 45 | 13 | 1250 | 0.7 | 7e+9 | 0.92 | 0.83 | 0.04 |
| mypic1-a | 3 | 5 | 5 | 47 | 13 | 1500 | 0.7 | 2.2e+9 | 0.7 | 0.83 | 0.04 |
| mypic1-b | 3 | 5 | 5 | 47 | 13 | 1500 | 0.7 | 1e+9 | 0.7 | 0.83 | 0.04 |
| mypic2-a | 3 | 5 | 5 | 45 | 13 | 1500 | 0.83 | 2.2e+9 | 0.7 | 0.83 | 0.04 |
| mypic2-b | 3 | 5 | 5 | 45 | 13 | 1500 | 0.83 | 1e+9 | 0.7 | 0.83 | 0.04 |

Table 2: Values of Parameters

# 3 SURF

We apply the inbuilt OpenCV SURF (Speeded Up Robust Feature) descriptor and extractor on the pair of images and establish correspondences between the two. The theory behind SURF has been done in class, hence only the key points are mentioned.

Interest points and features are extracted at different scales. The main underlying concept is to maximize the determinant of the Hessian matrix which is given by

$$H(x,y,\sigma) = \begin{bmatrix} \frac{\partial^2}{\partial x^2} ff(x,y,\sigma) & \frac{\partial^2}{\partial x \partial y} ff(x,y,\sigma) \\ \frac{\partial^2}{\partial x \partial y} ff(x,y,\sigma) & \frac{\partial^2}{\partial y^2} ff(x,y,\sigma) \end{bmatrix}$$

where $\sigma$ refers to the scale and $ff(x,y,\sigma)$ refers to the Gaussian smoothed image. The cross-derivatives in the determinant help to discriminate interest points from edges.

Using the concept of Integral images, SURF does very fast computations at different scales. At each interest point, the dominant direction is determined and a descriptor is

computed with respect to this dominant direction. The descriptor at an interest point is calculated using a $20\sigma$X$20\sigma$ window around the point. A 64 element descriptor vector is created for each interest point, and these can be compared to find correspondences. The Euclidean distance between the descriptor vectors is usually used for this.

### 3.1 Performance of SURF vs Harris Corner Detector

- As expected, SURF detects many more interest points and better correspondences between the two images of the same scene. Scale space analysis , very fast computations and bigger descriptors for each interest point are some of the salient features of SURF.

- On the other hand, the Harris Corner Detector is useful for detecting corners in cases where scale space analysis is not a pre-requisite. It detects mostly corners and is sensitive to noise. One way to address this is to use some smoothing before using the Sobel operator.

- As we can observe, when we use NCC or SSD there might be a false correspondences, as the thresholds might not be exact and vary from image to image. Finding correspondences using metrics such as NCC and SSD is not as accurate as using the SURF feature descriptors.

- For SURF higher threshold for the Hessian reduces the number of correspondences obtained, however at the same time increases the accuracy of correspondences as illustrated.

## 4 Images

### 4.1 Harris Corner Detector

We present the images below. One pair of images was given along with the question, 2 other pair of images was taken by my digital camera. Ratio A/B in the image title refers to the number of correct matches out of total number of matches obtained.

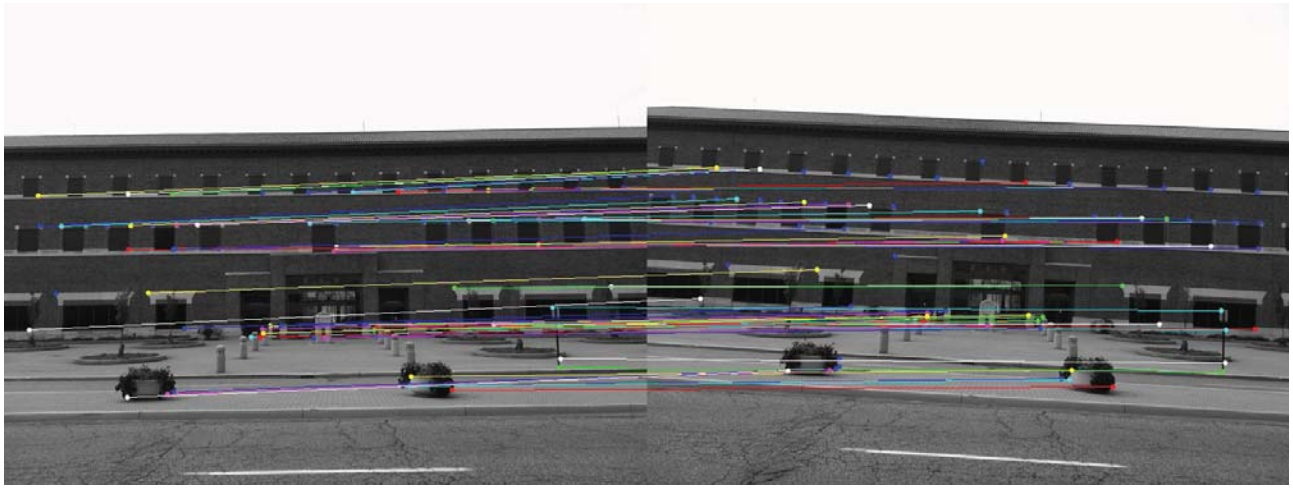

Figure 4.1: Sample a, Sample b, SSD, 62/62 correct matches

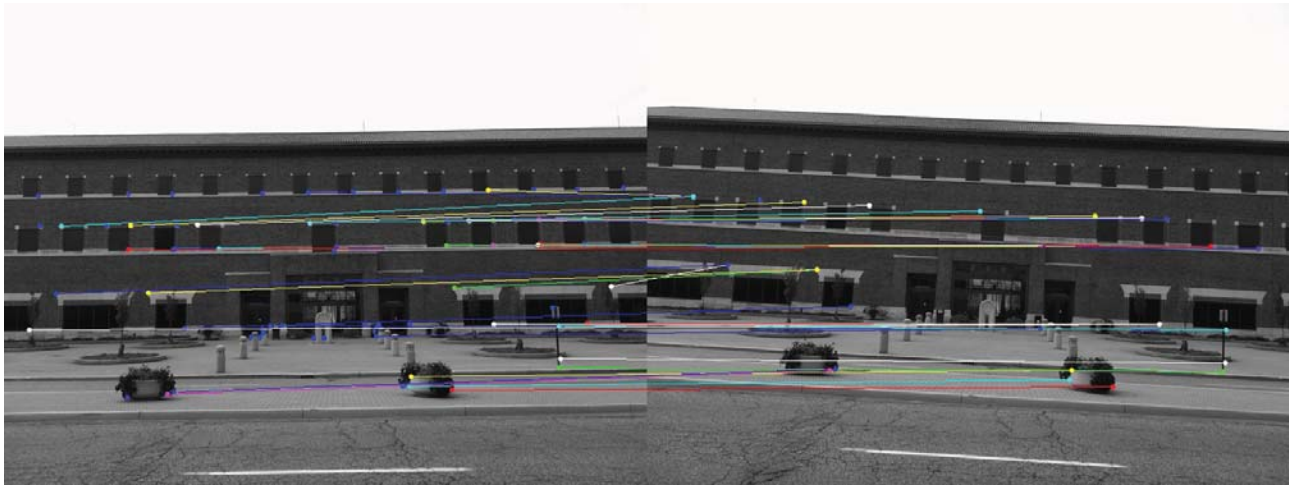

Figure 4.2: Sample a, Sample b, NCC, 32/34 correct matches

Figure 4.3: Mypic2, SSD, 51/53 correct matches

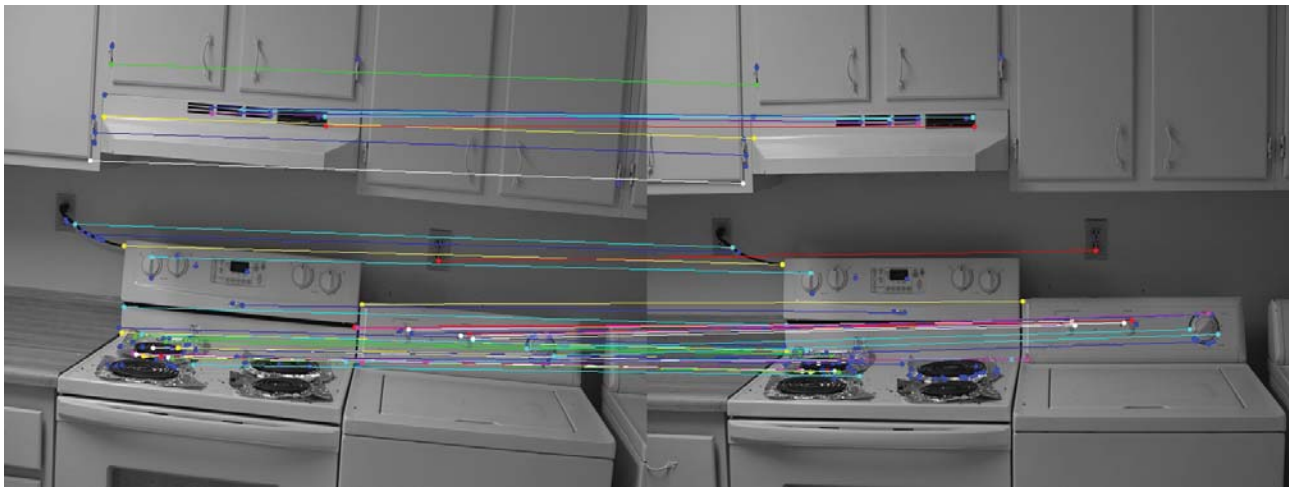

Figure 4.4: Mypic2, NCC, 48/48 correct matches

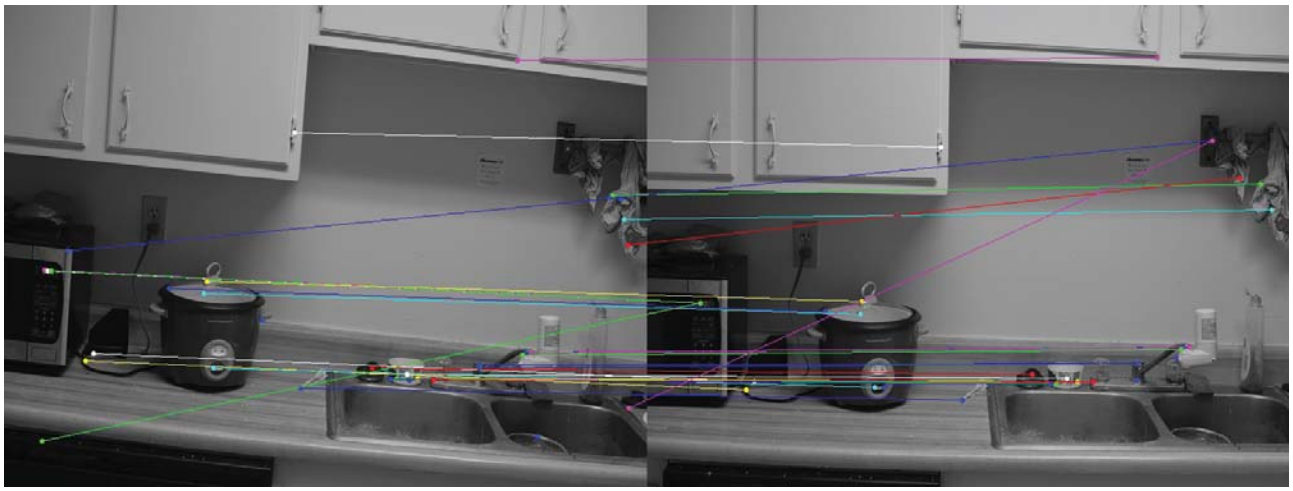

Figure 4.5: Mypic1, SSD, 25/29 correct matches

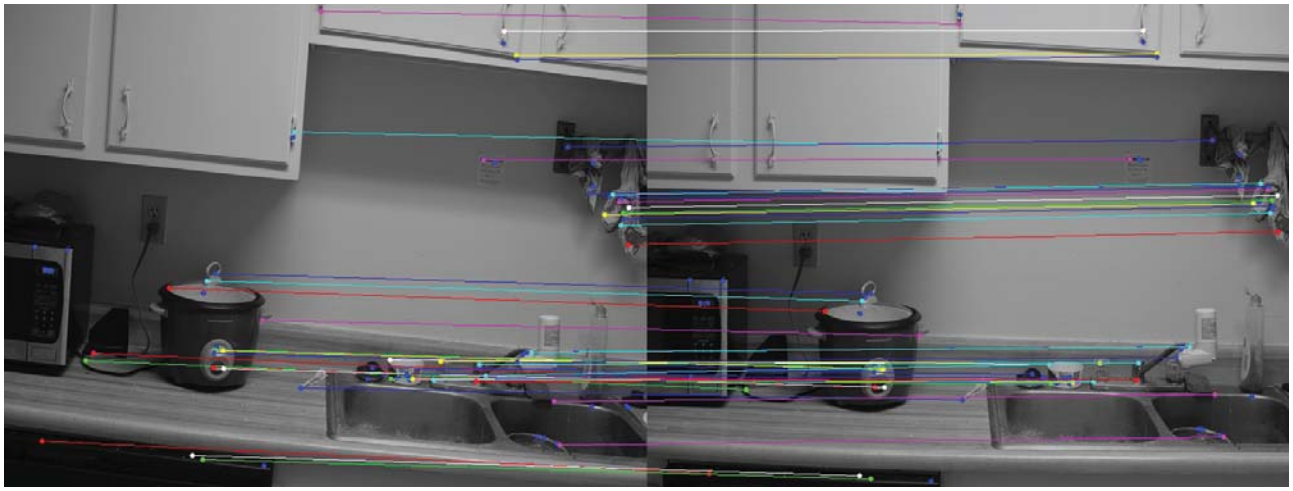

Figure 4.6: Mypic1, NCC, 49/49 correct matches

## 4.2 SURF

Below we present the results of the SURF algorithm.As we can see SURF detects far more interest points and establishes correspondences with more accuracy. For SURF we use a threshold for the key-point detector. If the hessian is smaller than the hessianThreshold, then those features are rejected by the detector. Therefore, the larger the value, fewer key-points are obtained. We run simulations for two values of Threshold on each image. We observe that for the smaller threshold, more interest points are retained but there are more false correspondences. For a larger threshold, there are fewer key-points but the correspondence accuracy is greater

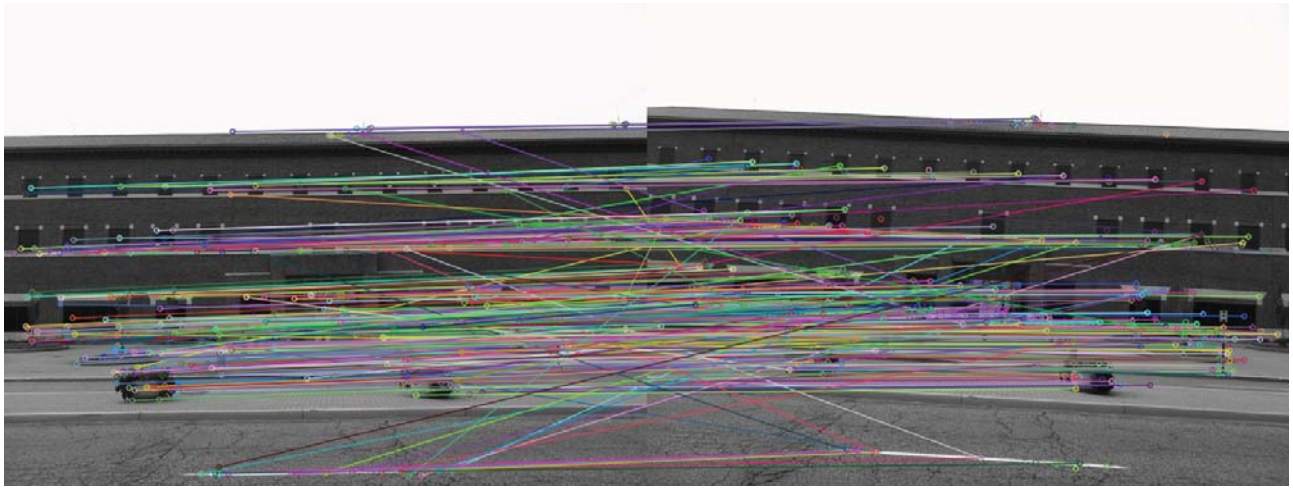

Figure 4.7: HessianThreshold- 1000, 414 matches

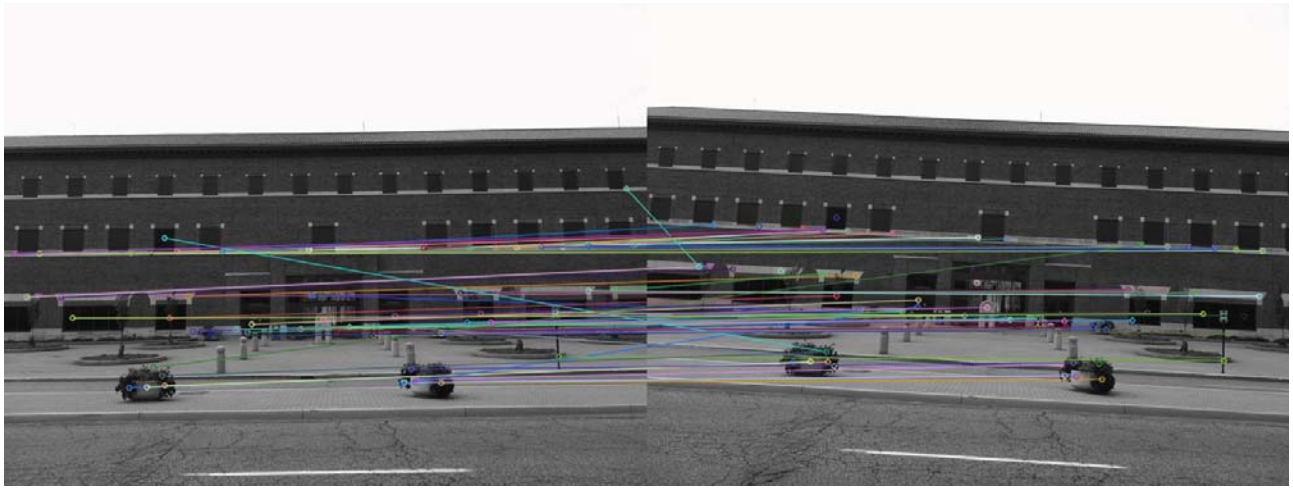

Figure 4.8: HessianThreshold- 4000 , correct matches- 69/72

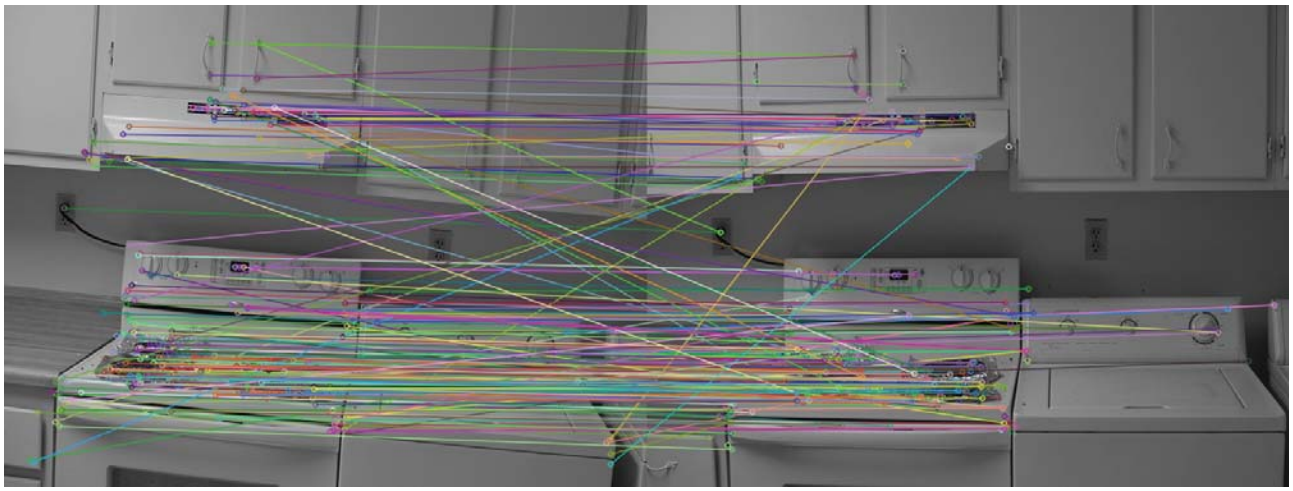

Figure 4.9: HessianThreshold- 1000 , matches- 290

Figure 4.10: HessianThreshold- 5000 , correct matches- 46/50

Figure 4.11: HessianThreshold- 500 , matches- 344

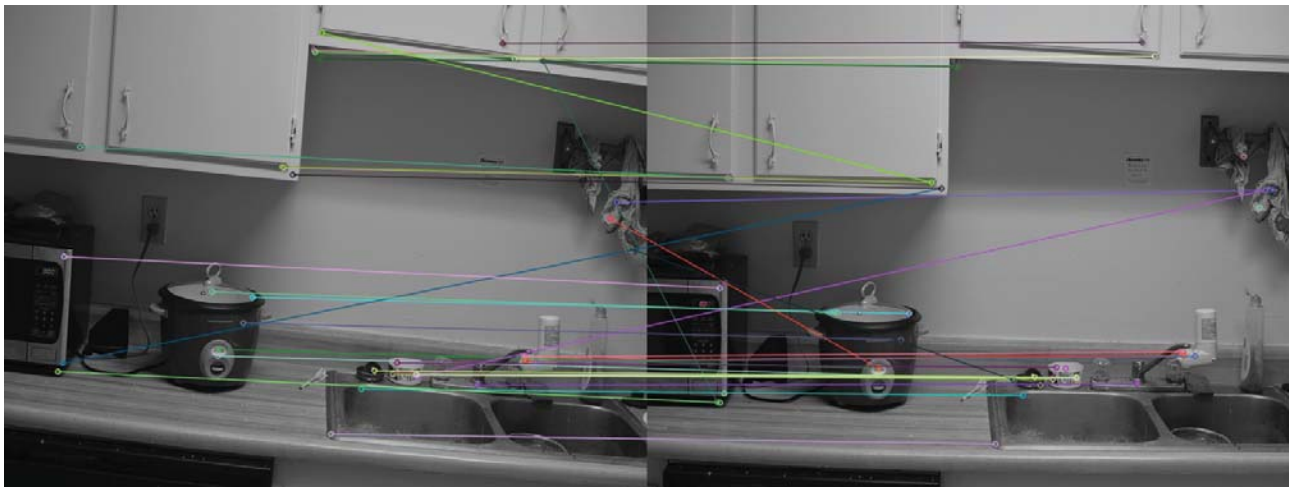

Figure 4.12: HessianThreshold- 3000 , correct matches- 32/36

# 5 Code

Below is the C++ code. Please note that the first main.cpp is for the Harris Corner Detector. The second main.cpp is for SURF extractor. Two separate main.cpp files were created for convenience and ease of understanding of the code. All simulations were done in MS Visual Studio 2010.

```
// This code is for Harris Corner Detector
#include <opencv2/core/core.hpp>
#include <opencv2\opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <iostream>
#include <math.h>

# define lim_fd 100

using namespace cv;
using namespace std;

//feature class
class descriptor {
```

```
public:
        double succ1[lim_fd];
        int fd_n;
        int hit_n[lim_fd];
        Point pt[lim_fd ];
        CvMat* Matr[lim_fd ];
};


int WIN_DIM=5; //dimensions for Harris cornerdetector window
int DIM_FM_SSD= 45;//window dimensions for SSD
int DIM_FM_NCC= 13; //window dimensions for NCC

//FUNCTION DECLARATIONS
CvMat* Harris(IplImage* orig_image1 );
CvMat* window(Point p, IplImage *orig_image1 , int dim );
void feature_extract ( descriptor *dcrip1 , CvMat*
   corner_strength1, long long int hold,  IplImage* orig_image1,
    int DIM);
void Drawtool( descriptor dcrip1, IplImage *orig_image1 );
void SSDC(descriptor *dcrip1, descriptor *dcrip2, int hold1,
   double decis1);
void NCCC(descriptor *dcrip3, descriptor *dcrip4,double hold2,
   double decis2);

int main (int argc, char *argv[] )
{
        descriptor dcrip1, dcrip2,dcrip3,dcrip4 ;
        int hold1 = 1500; //SSD threshold
        double decis1 = 0.7; //SSD ratio threshold
```

```
double hold2 = 0.7; //NCC theshold
double decis2 = 0.83;//NCC ratio threshold

char* name_file1,*name_file2;
IplImage* orig_image1 = 0, *orig_image2 = 0;
//read in input files
orig_image1 = cvLoadImage(argv[1] ,0);
orig_image2 = cvLoadImage(argv[2] ,0);

if(argc>=2)  //Checking input file
{
        name_file1=argv[1];
        name_file2=argv[2];
}
else
{
        cout<<"Wrong number of inputs"<<endl;
        system("pause");
        return 1;
}
if(orig_image1->height==0||orig_image1->width==0)
{
        cout<<"Invalid input file"<<endl;
        return 2;
}

CvMat* corner_strength1 = cvCreateMat(orig_image1->
   height,orig_image1->width,CV_64FC1);
CvMat* corner_strength2 = cvCreateMat(orig_image1->
   height,orig_image1->width,CV_64FC1);
```

```
//cornerstrength for sample images a and b of same scene
corner_strength1 = Harris(orig_image1);
corner_strength2 = Harris(orig_image2);

dcrip1.fd_n = -1;
dcrip2.fd_n = -1;
dcrip3.fd_n = -1;
dcrip4.fd_n = -1;

//extract features  . SSD and NCC
feature_extract( &dcrip1, corner_strength1, 2200000000,
   orig_image1, DIM_FM_SSD ) ;
feature_extract( &dcrip2, corner_strength2 ,2200000000,
   orig_image2 ,DIM_FM_SSD ) ;
feature_extract( &dcrip3, corner_strength1, 1000000000,
   orig_image1, DIM_FM_NCC ) ;
feature_extract( &dcrip4, corner_strength2 ,1000000000,
   orig_image2 ,DIM_FM_NCC ) ;

SSDC(&dcrip1,&dcrip2,hold1,decis1); // match
   correspondences for SSD
NCCC(&dcrip3,&dcrip4,hold2,decis2);// match
   correspondences for NCC

//output image for SSD
IplImage *new_image = cvCreateImage( cvSize(orig_image1
   ->width*2, orig_image1->height ), orig_image1->depth
   , 3);
```

```
cvSetImageROI( new_image , cvRect(0, 0, orig_image1->
    width , orig_image1->height) );
cvCvtColor(orig_image1 , new_image , CV_GRAY2BGR );
Drawtool(dcrip1 ,new_image );
cvResetImageROI( new_image );

cvSetImageROI( new_image , cvRect(orig_image1->width ,
    0, orig_image1->width , orig_image1->height ) );
cvCvtColor(orig_image2 , new_image , CV_GRAY2BGR );
Drawtool( dcrip2 ,new_image);
cvResetImageROI( new_image );

//output image for NCC
IplImage *new_image1 = cvCreateImage( cvSize(orig_image1
    ->width*2, orig_image1->height ), orig_image1->depth
    , 3);
cvSetImageROI( new_image1 , cvRect(0, 0, orig_image1->
    width , orig_image1->height) );
cvCvtColor(orig_image1 , new_image1 , CV_GRAY2BGR );
Drawtool(dcrip3 ,new_image1 );
cvResetImageROI( new_image1 );

cvSetImageROI( new_image1 , cvRect(orig_image1->width ,
    0, orig_image1->width , orig_image1->height ) );
cvCvtColor(orig_image2 , new_image1 , CV_GRAY2BGR );
Drawtool( dcrip4 ,new_image1);
cvResetImageROI( new_image1 );

Scalar Colorc[8] = {Scalar(255,0,255), Scalar
    (255,255,255), Scalar(0,255,0), Scalar(0,255,255),
```

```
    Scalar(255,0,0), Scalar(255,255,0), Scalar(0,0,255),
    Scalar(255,0,0)};
int counter1 = 0;
int counter2=0;

//draw correspondences as lines between interest points
for (int i =0; i <= dcrip1.fd_n; i++)
{
        if (dcrip1.hit_n [i] >=0)
        {
                cvCircle(new_image ,dcrip1.pt[i], 0,
                    Colorc[i%8] ,5);
                cvCircle(new_image , cvPoint( dcrip2.pt
                    [dcrip1.hit_n[i ]].x+orig_image1->
                    width, dcrip2.pt[dcrip1.hit_n[i]].y)
                    , 0, Colorc[i%8] ,5) ;
                cvLine (new_image , dcrip1.pt[i],
                    cvPoint( dcrip2.pt[dcrip1.hit_n[i]].
                    x+orig_image1->width, dcrip2.pt[
                    dcrip1.hit_n[i]].y), Colorc[i%8]) ;
                counter1 ++;
        }
}

for (int i =0; i <= dcrip3.fd_n; i++)
{
        if (dcrip3.hit_n [i] >=0)
        {
                cvCircle(new_image1 ,dcrip3.pt[i], 0,
                    Colorc[i%8] ,5);
```

```
                        cvCircle(new_image1 , cvPoint( dcrip4.
                            pt[dcrip3.hit_n[i ]].x+orig_image1->
                            width, dcrip4.pt[dcrip3.hit_n[i]].y)
                            , 0, Colorc[i%8] ,5) ;
                        cvLine (new_image1 , dcrip3.pt[i],
                            cvPoint( dcrip4.pt[dcrip3.hit_n[i]].
                            x+orig_image1->width, dcrip4.pt[
                            dcrip3.hit_n[i]].y), Colorc[i%8]) ;
                        counter2 ++;
                }
        }

        cout<<counter1<<endl<<counter2; //display number of
            matches for SSD and NCC respectively

        cvSaveImage("newssd.jpg ",new_image );
        cvSaveImage("newncc.jpg ",new_image1 );
        cvWaitKey(0) ;
        // release the image
        cvReleaseImage(& orig_image1 );
        system("pause");
        return 0;
}
//Harris corner detector function
CvMat* Harris(IplImage* orig_image1 )
{
         int sepr;
         uchar *pixel;
         sepr = orig_image1->widthStep;
         pixel = (uchar *)orig_image1->imageData;
```

```
    CvMat* C = cvCreateMat(2,2,CV_64FC1);
    IplImage* dx =0, *dy =0;
    CvMat* temp = cvCreateMat(orig_image1->height,
        orig_image1->width, CV_64FC3);
    CvMat* corner_strength1 = cvCreateMat(orig_image1->
        height, orig_image1->width, CV_64FC1);
    CvMat* corner_strength2 = cvCreateMat(orig_image1->
        height,orig_image1->width, CV_64FC1);
    cvZero(temp);
    cvZero(corner_strength1);

    dx = cvCreateImage(cvSize(orig_image1->width,
        orig_image1->height),orig_image1->depth,orig_image1
        ->nChannels);
    dy = cvCreateImage(cvSize(orig_image1->width,
        orig_image1->height),orig_image1->depth,orig_image1
        ->nChannels);

    //x and y derivatives using Sobel
    cvSobel(orig_image1,dx,1,0);
cvSobel(orig_image1,dy,0,1);

    uchar *dxpixel, *dypixel;
    dxpixel = (uchar *)dx->imageData ;
    dypixel = (uchar *)dy->imageData ;

    double norm=(double)1/WIN_DIM;

    //calculate C matrix
```

```
for (int row= WIN_DIM/2; row<orig_image1->height-
   WIN_DIM/2; row++)
{
        for (int col= WIN_DIM/2; col<orig_image1->width
           -WIN_DIM/2; col++)
        {
                CvScalar temp1= cvGet2D (temp,row,col);
                for (int stepx =- WIN_DIM/2; stepx <
                   WIN_DIM/2+1; stepx ++)
                {
                        for (int stepy =- WIN_DIM/2;
                           stepy < WIN_DIM/2+1; stepy
                           ++)
                        {
                                temp1.val[0]+= norm*
                                   dxpixel[(row+ stepy
                                   )*sepr+(col+ stepx)
                                   ]*dxpixel[(row+stepy
                                   )*sepr+(col+stepx)];
                                temp1.val[1]+= norm*
                                   dxpixel[(row+ stepy
                                   )*sepr+(col+ stepx)
                                   ]*dypixel[(row+stepy
                                   )*sepr+(col+stepx)];
                                temp1.val[2]+= norm*
                                   dypixel[(row+ stepy
                                   )*sepr+(col+ stepx)
                                   ]*dypixel[(row+stepy
                                   )*sepr+(col+stepx)];
                        }
```

```
                }
                cvSet2D(temp, row, col, temp1);
                cvmSet(C,0,0,temp1.val[0]);
                cvmSet(C,0,1,temp1.val[1]);
                cvmSet(C,1,0,temp1.val[1]);
                cvmSet(C,1,1,temp1.val[2]);
                CvScalar tr=cvTrace(C);
                double trac=tr.val[0];
                double det=cvDet(C);
                //compute corner strength for pizel
                    using trace and determinant of C to
                    avoid SVD
                cvmSet(corner_strength1,row,col, det -
                    0.04*(trac)*(trac));
        }
}

cvCopy (corner_strength1,corner_strength2);

//non maximum thresholding of cornerstrength
for(int row= WIN_DIM/2; row<orig_image1->height-WIN_DIM
    /2; row++)
{
        for (int col= WIN_DIM/2; col<orig_image1->width
            -WIN_DIM/2; col++)
        {
                for (int stepx =-WIN_DIM/2; stepx <
                    WIN_DIM/2+1; stepx++)
                {
```

```
                                for (int stepy =-WIN_DIM /2;
                                    stepy < WIN_DIM/2+1; stepy
                                    ++)
                                {
                                        if (cvmGet(
                                            corner_strength1,
                                            row, col)<cvmGet(
                                            corner_strength1,
                                            row+stepy, col+
                                            stepx))
                                        {
                                                cvmSet (
                                                    corner_strength2
                                                     , row, col,
                                                    0);
                                        }
                                }
                            }
                        }

                return corner_strength2;
}

//extract features
void feature_extract ( descriptor *dcrip1 , CvMat*
    corner_strength1, long long int hold,  IplImage* orig_image1,
     int  DIM)
 {
        int counter =0;
```

```
        for (int row= DIM/2; row<orig_image1->height-DIM/2; row
           ++)
        {
                for (int col= DIM/2; col<orig_image1->width-DIM
                   /2; col++)
                {
                        if (cvmGet(corner_strength1, row, col)>
                           hold)
                        {
                                if (counter >99)
                                        break ;
                                dcrip1->fd_n = counter ++;
                                dcrip1->pt[dcrip1->fd_n].x =
                                   col;
                                dcrip1->pt[dcrip1->fd_n].y =
                                   row;
                                dcrip1->Matr[dcrip1->fd_n] =
                                   window(Point(col,row),
                                   orig_image1,DIM);// feature
                                   graylevels in a window
                                   around the interest point
                        }
                }
        }
        return ;
 }

//window for extracting features
CvMat* window(Point p, IplImage *orig_image1 , int dim )
 {
```

```
        CvMat* Matr = cvCreateMat(dim , dim , CV_64FC1 );
        int sepr ;
        sepr = orig_image1->widthStep ;
        uchar *pixel ;
        pixel = (uchar *) orig_image1->imageData ;
        for (int row=-dim/2; row <= dim/2; row++)
        {
                for (int col=-dim/2; col <= dim/2; col++)
                {
                        cvmSet(Matr, row+ dim/2, col+ dim/2,
                           pixel[(row+p.y)*sepr+(col+p.x)]);//
                           window of pixels around interest
                           point
                }
        }
        return Matr;
}

//SSD function
void SSDC(descriptor *dcrip1, descriptor *dcrip2,int hold1,
   double decis1)
{
        CvMat* I = cvCreateMat(DIM_FM_SSD, DIM_FM_SSD , CV_64FC1
            );
        CvMat* J = cvCreateMat(DIM_FM_SSD , DIM_FM_SSD ,
           CV_64FC1 );
        double succ1 , succ2 ;
        for (int i =0; i <= dcrip1->fd_n; i ++)
        {
                dcrip1->hit_n [i] = -1;
```

```
succ1 = 10000000000;
succ2 = -100;

for (int j =0; j <= dcrip2->fd_n; j ++)
{
        cvSub(dcrip1->Matr[i], dcrip2->Matr[j],
           I );
        cvMul(I , I , J );

        double val= cvSum(J).val[0]/( DIM_FM_SSD
            * DIM_FM_SSD);

        if (val < hold1 && succ1 > val) //to
           threshold SSD  values
        {
                succ2 = succ1 ;
                dcrip1->hit_n[i] = j;
                succ1 = val ;
                dcrip1->succ1[i] = succ1 ;
                dcrip2->hit_n[j] = i;
                dcrip2->succ1[j] = succ1 ;
         }
 }
 if (succ2 >0 && succ1 >0)
 {
         if (succ1 /succ2 > decis1 ) // to
            prevent false correspondences when
            the feature is not unique
         {
```

```
                                dcrip2->hit_n[dcrip1->hit_n[i]]
                                    = -1;
                                dcrip2->succ1[dcrip1->hit_n[i]]
                                    = -1;
                                dcrip1->hit_n[i] = -1;
                                dcrip1->succ1[i] = -1;
                            }
                        }
                }
                return;
}

//drawing tool to circle interest points
void Drawtool(descriptor dcrip, IplImage *orig_image1 )
{
        for (int i =0; i <= dcrip.fd_n; i++)
        {
                cvCircle(orig_image1 , dcrip.pt[i], 0, cvScalar
                    (255, 0, 0), 5);
        }
        return ;
 }

//NCC function
void NCCC(descriptor *dcrip3, descriptor *dcrip4, double hold2 ,
   double decis2)
{
        CvMat* avg1matr = cvCreateMat(DIM_FM_NCC, DIM_FM_NCC,
            CV_64FC1);
```

```
CvMat* avg2matr = cvCreateMat(DIM_FM_NCC, DIM_FM_NCC,
    CV_64FC1);
CvMat* J = cvCreateMat(DIM_FM_NCC, DIM_FM_NCC, CV_64FC1
    );
CvMat* J1 = cvCreateMat(DIM_FM_NCC, DIM_FM_NCC,
    CV_64FC1);
CvMat* J2 = cvCreateMat(DIM_FM_NCC, DIM_FM_NCC,
    CV_64FC1);
double succ1, succ2 ;
for (int i =0; i <= dcrip3->fd_n; i++)
{
        dcrip3->hit_n[i] = -1;
        succ1 = -100; // NCC
        succ2 = -100;
        for(int j =0; j <= dcrip4->fd_n; j++)
        {
                double avg1 = cvAvg(dcrip3->Matr[i]).
                    val[0];
                double avg2 = cvAvg(dcrip4->Matr[j]).
                    val[0];

                cvAddS(dcrip3->Matr[i],cvScalar(-avg1),
                     avg1matr );
                cvAddS(dcrip4->Matr[j],cvScalar(-avg2),
                     avg2matr );

                cvMul(avg1matr, avg2matr, J);
                cvMul(avg1matr, avg1matr, J1 );
                cvMul(avg2matr, avg2matr, J2 );
```

```
                        double val = cvSum(J).val[0]/sqrt(cvSum
                            (J1).val[0]*cvSum(J2).val[0]) ; //
                            calculate NCC

                        if (val >hold2 && succ1 < val) //
                            threshold NCC
                        {
                                succ2 = succ1 ;
                                dcrip3->hit_n[i] = j;
                                succ1 = val ;
                                dcrip3->succ1[i] = succ1 ;
                                dcrip4->hit_n[j] = i;
                                dcrip4->succ1[j] = succ1 ;
                        }
                    }
                    if (succ2 >0 && succ1>0)
                    {
                            if ( succ2/succ1 > decis2 ) // to
                               prevent false correspondences where
                               features are not unique
                            {
                            dcrip4->hit_n[dcrip3->hit_n[i]] = -1;
                            dcrip4->succ1[dcrip3->hit_n[i]] = -1;
                            dcrip3->hit_n[i] = -1;
                            dcrip3->succ1[i] = -1;
                            }
                    }
            }
}
```

Below is the main.cpp for SURF algorithm.

```
// This is the code for SURF feature extractor
#include <stdio.h>
#include <iostream>
#include "opencv2/core/core.hpp"
#include <opencv2/nonfree/features2d.hpp>
#include <opencv2\legacy\legacy.hpp>
#include "opencv2/highgui/highgui.hpp"

using namespace cv;
using namespace std;

void SURFc(vector<KeyPoint>* vec_a, vector<KeyPoint>* vec_b ,
   vector<DMatch>* hit_n, Mat orig_image1, Mat orig_image2, int
   Hessianthreshold);

int main( int argc, char** argv )
{
        int Hessianthreshold = 4000; // Hessian threshold
        char* name_file1,*name_file2;
        Mat orig_image1, orig_image2;
        //load image files
        orig_image1 = imread(argv[1] ,CV_LOAD_IMAGE_GRAYSCALE);
        orig_image2 = imread(argv[2] ,CV_LOAD_IMAGE_GRAYSCALE);

        if(argc>=2)  //Checking input file
        {
                name_file1=argv[1];
                name_file2=argv[2];
        }
        else
```

```
{
        cout<<"Wrong number of inputs"<<endl;
        system("pause");
        return 1;
}
if(orig_image1.size().height==0||orig_image1.size().
   width==0||orig_image2.size().height==0||orig_image2.
   size().width==0)
{
        cout<<"Invalid input file"<<endl;
        return 2;
}

vector<KeyPoint> vec_a, vec_b;
vector<DMatch> hit_n;

SURFc(&vec_a, &vec_b ,&hit_n,  orig_image1, orig_image2,
   Hessianthreshold); // Function to extract SURF
   descriptors and matches

Mat new_image;
char correct_filename [200] ;
sprintf (correct_filename , "%s_correct.jpg", name_file1
    ) ;
//output
drawMatches(orig_image1, vec_a, orig_image2, vec_b,
   hit_n, new_image);
imwrite(correct_filename,new_image);
system("pause");
return 0;
```

```
  }

//SURF descriptor extractor and matching correspondences
void SURFc(vector<KeyPoint>* vec_a, vector<KeyPoint>* vec_b ,
   vector<DMatch>* hit_n, Mat orig_image1, Mat orig_image2, int
   Hessianthreshold)
{
        SurfFeatureDetector surf(Hessianthreshold);
        surf.detect(orig_image1,*vec_a);
        surf.detect(orig_image2, *vec_b);

        // extract descriptors
        SurfDescriptorExtractor fde;
        Mat dcript1, dcript2;
        fde.compute(orig_image1, *vec_a, dcript1);
        fde.compute(orig_image2, *vec_b, dcript2);

        //brute-force matching of descriptors
        BruteForceMatcher<L2<float> > corresp; //L2 norm
        corresp.match(dcript1, dcript2, *hit_n);
        cout<<hit_n->size();
        return;
}
```