Project 4

ECE 763 – Computer Vision

Image Stitching using Correspondences and Homography

By:

Suraj Shanbhag

Ujan Sengupta

CONTENTS

INTRODUCTION

One of the most intriguing problems in the domain of Computer Vision is to match and stitch two contiguous (or somehow related) scenes or images. If such a match is achieved, it implies that the computer is able to interpret and stitch images which, barring some overlap, are quite different. This has wide ranging implications as it is a nascent yet significant step in equipping a machine with cognitive capabilities.

Being able to stitch different images to construct a contiguous environment is an important aspect of many practical applications of Computer Vision. From autonomous robot navigation and high-res photo mosaics to medical imaging and forensic science, image matching helps find patterns in different images that may be inconspicuous to the human eye. In the case of robot navigation, for example, stitching different images which are discrete for the robot (although humans understand them to be contiguous) helps construct a map for the robot to navigate. Thus, making sure that the correct images are stitched to each other is vital in ensuring the correct understanding of the environment of the robot.

Although not aiming for something quite that ambitious, this project confines the scope to finding a match between two images with some overlap (around 40%). Those images are then to be stitched, providing us a continuous image with, hopefully, no aberrations.

The aim is to find a set of definite correspondence points between two images (since they're both presumed to have at least 40% overlap) and to use those correspondence points to construct a matrix (referred to as the H matrix). Once the H matrix is found, simple matrix algebra can be applied to calculate all the correspondence points and then stitch the images.

The project is divided into two parts. The first part involves finding a certain number of correspondences manually and the second part involves finding the correspondence points using an interest point detector and local area descriptors. Once the points are found, the H matrix is constructed and the images are stitched together by finding the correspondences for every pixel of either of the images to the other image.

THEORY

The process of image stitching consists of three main parts, i.e. image registration, image calibration and image blending. In this project, we focus primarily on the first two. Since blending involves improving the aesthetics of the stitched image, it is not germane to the aim of this assignment.

However, before we start the process, we first need to do feature detection. Feature detection essentially finds interesting points in the two images and tries to find the correspondences existing between the two. For this project, the correspondences are found manually in Part 1 and using an interest point detector and a local area descriptor in Part 2. Corresponding points are the set of points which are common between the two images.

Image registration will be further elaborated in the Approach section of Part 2, as it is relevant to that segment of the project.

Image Calibration:

The aim of image calibration is to position the two images on the panosphere (in this case, the final stitched image) such that a perfect positioning of the overlapped regions is achieved, often using geometric optimizations.

Frequently, something known as alignment may be necessary to transform an image to match the view point of the image it is being composted with. Alignment refers to a change in the coordinates system so that it adopts a new coordinate system which outputs an image matching the required viewpoint.

An image may go through different transformations like pure translation, pure rotation, translation + rotation + scaling, an affine transformation or a projective transform.

For this project, we first consider a reference picture and then, rotating through a small angle, we consider the second picture (which has roughly 40% overlap with the previous one). This procedure and our understanding of the problem somewhat estimates a projective transform.

Now, projective transformation can be mathematically described as

$$x' = H * x$$

- where x is points in the old coordinate system, x' is the corresponding points in the transformed image and H is the homography matrix. However, for our purpose, we consider x to be the coordinates of the left image and x' to be the coordinates of the right image.

We need to use the data we possess, i.e. the corresponding sets of image coordinates, to derive the H matrix which has 8 parameters or degrees of freedom. It is the 3x3 form of the vector **h**, which can be computed using the Singular Value Decomposition of matrix A.

A is constructed using the coordinates of correspondences and h is the one dimensional vector of the 9 elements of the reshaped homography matrix.

Also, **h** is in the null space of A.

That is,

$$A * h = 0$$

Now, doing SVD of A gives us:

$$A = U * S * V^T$$

This implies that h is the column in V corresponding to the smallest singular vector.

Once we find the **h** vector and resultantly, the H matrix, we can use it to calculate where a particular pixel in the left image shall go in the right image. We could also calculate H^{-1} and find out the vice-versa.

Once we calculate the pixel values, the images can be stitched by placing points of one image on to the other. This might need the size of the resultant image to be bigger than the approximate sum of sizes of the constituent images as we don't know where the pixels might end up after multiplying with the homography matrix.

PART 1

Approach:-

First, we manually found a list of 10 correspondences between the two images that we clicked, and listed it in the form

x1, y1||x1', y1'
x2, y2||x2', y2'
x3, y3||x3', y3'
x4, y4||x4', y4'
.
.
.

Where x1, y1 are the pixel coordinates in the left image and x1', y1' are the pixel coordinates in the right image.

Next, for each correspondence, we construct the 2x8 matrix –

$$\begin{bmatrix} 0 & 0 & 0 & -x_i & -y_i & -1 & x_i y_i' & y_i y_i' \\ x_i & y_i & 1 & 0 & 0 & 0 & -x_i x_i' & -y_i x_i' \end{bmatrix}$$

We then stacked the ten 2x8 matrices into a 20x8 matrix and called it A.
For each correspondence, we found the 2x1 vector $[-y_i' \; x_i']^T$ and stacked them to form a 20x1 vector named D.

We finally constructed the 8x1 **h** vector using-

$$\mathbf{h} = (A^T A)^{-1} A^T D$$

We rewrote the **h** vector as a 3x3 matrix H, appending a 1 at the last position (3rd row, 3rd column) in the matrix.

Now, we could have also calculated H^{-1} to do the pixel calculation the other way around to fill up the black spots we got as a result of following this method. However, we decided to switch x1, y1||x1', y1' to x1',y1'||x1,y1. Doing this copies the pixels of the right image to the left one. This eliminates the need of finding H^{-1} to eliminate the black spots.

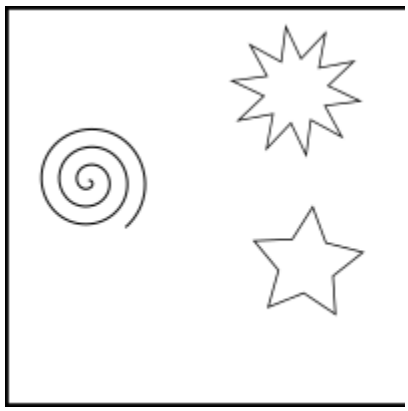
We ought to make clear that the number of correspondence points we took manually is directly related to how good our output should look and 10 points is the minimum that we could have (and did) use.

Images:-

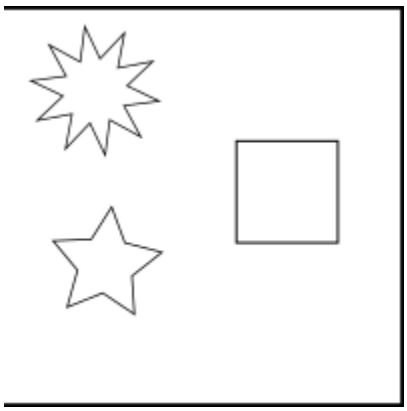
The images we have considered in this project (the non-synthetic ones) are actually color images in RGB. However, since finding correspondences and other image operations are easier in grayscale when a single frame of the image is considered, we have presented all our findings in grayscale. The conversion to color is a simple matter of running the same algorithm on three different frames and adding them to a 3 dimensional ifsImage.

1) **Synthetic Image:**

   The first image we considered was made using synthetically, using a computer software (Inkscape).



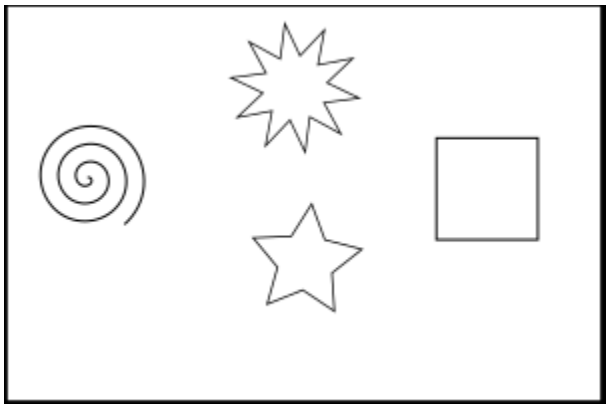
Left Side



Right Side

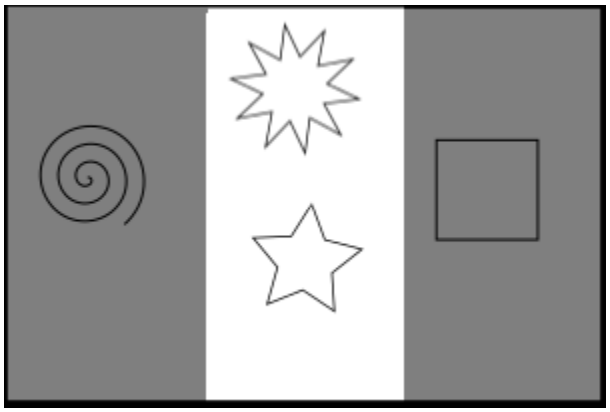
We can see that the two central figures feature as the overlap between the two images. Since this is a synthetic image, finding the correspondences manually was fairly simple as there was no noise and the image features were prominent.

OUTPUT:



And the overlap is shown with the help of an addition operation in the algorithm, in which every point in the overlap has its brightness summed over both images.

OVERLAP:



So, as we can see from the above images, the algorithm works quite well for synthetic images. We get a near perfect match and the stitching is done seamlessly.

2) Water Meter:

For this image, we took a couple of pictures of the water meter near one of our apartments and tried to stitch the images.

Left Side

Right Side

As we can see, there are two aspects of this image that might cause a problem with our algorithm:

a) The angles that the two images have been taken at are slightly different and there appears to be a horizontal skew.
b) The object is not planar and since our algorithm is only valid for planar surfaces (since we're using the Homogrpaphy Matrix instead of the Fundamental Matrix), this might be an issue.

OUTPUT:

OVERLAP:

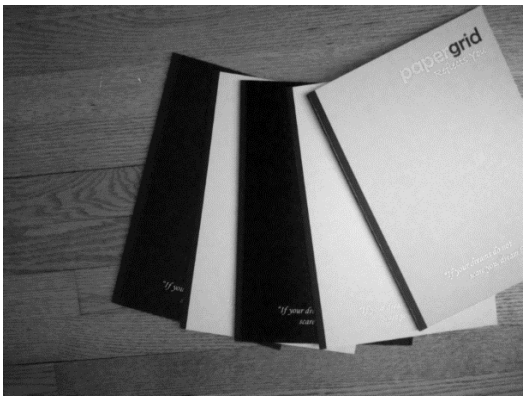
There are certain distinctive features that are noticeable from the above images:

I. The black regions in the outer extremities of the image frame is due to the fact that the final image onto which both the constituent images were stitched, had dimensions thrice that of either of the constituent images. This was done to avoid the pixel coordinates arising from the homography calculation from going out of bounds. The removal of the black regions is a matter of simple thresholding but they've been shown here deliberately to explain their presence. The image shown here actually had an even larger black background, which has been cropped to improve viewability.

II. There is a distinct line in the output image that shows where the two images get merged. Also, in the overlap image, the first circular meter can be seen as a hazy juxtaposition of the same object that has been viewed from different angles. Both of these issues arise due to the fact that the two pictures were taken from slightly different angles and that the object/surface in question is not planar but actually comprehensively 3 dimensional.

In order to check whether our algorithm worked for objects that are highly planar and not necessarily large structures viewed from a distance, we decided to test it on two different arrangements of books, with a varying degree of interest points. The two arrangements are named, rather unimaginatively, Books-1 and Books-2.

3) Books – 1:



Left Side

Right Side

We can see that the right image is slightly rotated, since the actual arrangement of the books is more appropriately depicted in the left image.

OUTPUT:

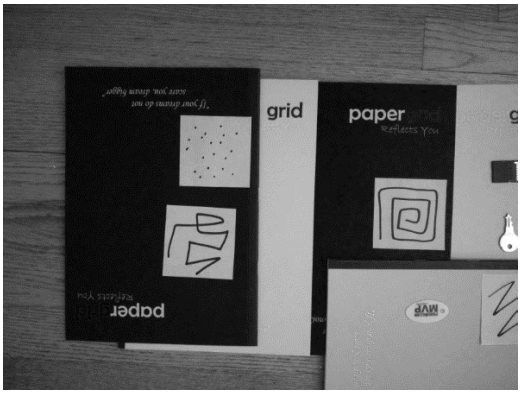
OVERLAP:

One interesting observation that we can make from this image is that the algorithm can be said to be somewhat rotation invariant. Although the right hand side image has been rotated, the overlapped parts were matched in the correct manner and the right image was stitched to the left one keeping the rotation in mind.

This is, however, highly dependent on the correspondence points that we choose to construct the A matrix. If the points are localized to a small neighborhood, the rotation invariance would not be comprehensive and we would see visible aberrations in the output image. However, since we chose points that were placed far apart from each other (although only 10 of them), the H matrix was able to take into account the orientation of the extremities of the correspondence points.

4) Books – 2

This arrangement of books is slightly different from the previous one and we drew several shapes on post-its and stuck them on the covers so as to increase the number of feature points that we could use to find correspondences.

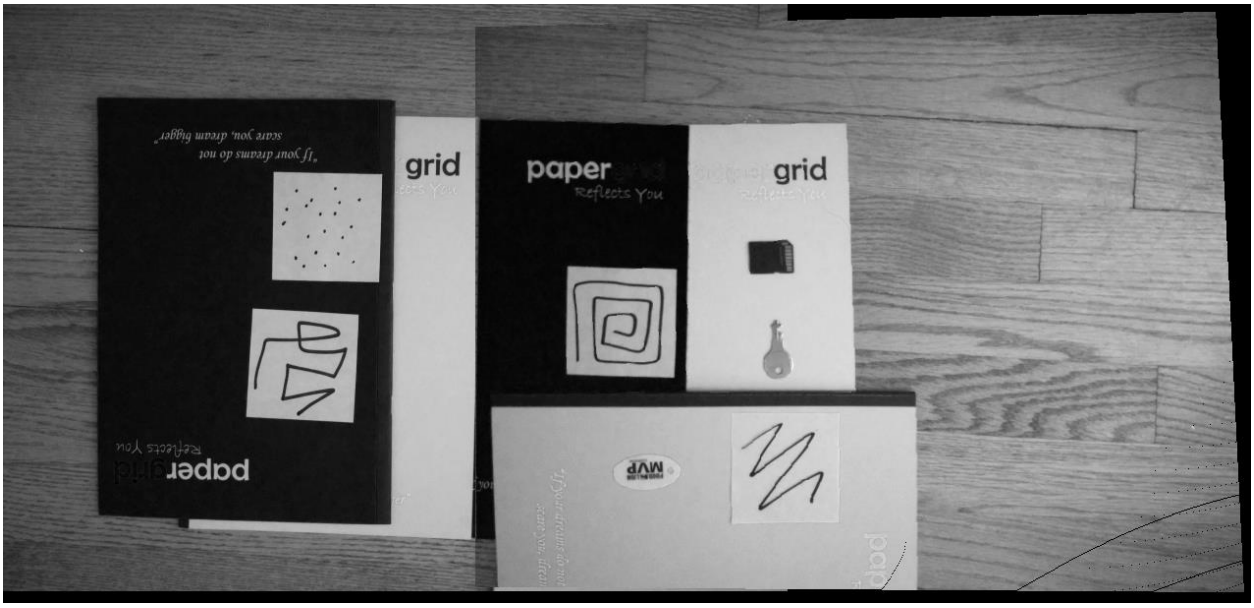


Left Side

Right Side

We can see that there are a number of different objects (a key, an MVP tag, a mini-SD memory card, etc.) placed on top of the books, so as to make this image different from the previous one in terms of internal difference in orientation and feature points.

OUTPUT:



OVERLAP:

We can see that to do the stitching, the right image is rotated by a small amount and adjusted by translating it downwards to fit it with the left image. The image is stitched quite well but we can see that the overlap image shows slight haziness at certain points.

This is due to the fact that the calculations to find the pixel coordinates in the final image is done with floating point numbers and the coordinates are actually integers without any decimal component. So, while converting from **float** to **int**, some of the accuracy in terms of image information is lost (i.e. a particular pixel can be placed one coordinate away from its intended position).

PART 2

Approach:

In this section of the project, we delve into feature detection and image registration, the two essential algorithms we use to find correspondence points automatically. Once we find the correspondence points, the rest of the process is the same as elaborated in the Theory section and Part 1.

For our implementation, we have used the SIFT interest Operator and the SIFT descriptor to do feature detection and finding correspondences.

1) Feature Detection:

   Feature detection is necessary to automatically find correspondences between images. Robust correspondences are required in order to estimate the necessary transformation to align an image with the image it is being composted on. Corners, blobs, Harris corners and Difference of Gaussian of Harris corners (DoG) are good features since they are repeatable and distinct.

   For this project, we used the SIFT interest operator to perform feature detection. The following are the steps we took:

   a) Blur the image with a Derivative of Gaussian kernel with 5 different values of **σ** (i.e. 0.707, 1.0, 1.414, 2.0 and 2.83)**,** producing 5 differently blurred images. Call them, say, im1, im2, im3, im4 and im5.
   b) Subtract im2 from im1, im3 from im2 and so on, creating 4 different Difference of Gaussian (DoG) images.
   c) We now look for a points which are the local extrema in their own scale and in one scale above and below. Once we have such a point, we check through the DoG scale space to see if it is an extremum in all the DoG images. If it is, it is denoted as an interest point.

      This theoretical algorithm is implemented in code by creating a temporary image (of the same size as the image we're working on) to hold the interest points.

For every local extremum in a particular DoG image, the brightness of the corresponding pixel in the temporary image is incremented by a certain value, say 20.
We do this for the four DoG images, and this gives us a temporary image where some pixels are very bright, some which are moderately bright and others which are quite dim. Though the brightest pixels are definitely interest points, but a threshold is used to designate a certain brightness, above which all points are interest points and below which, we don't consider the pixels.

After following the above methodology and implementing it in code, we get an image with the all the interest points in the image. Next, we use our implementation of the SIFT descriptor to describe the neighborhoods of the interest points, following which, the neighborhoods from the two images are matched to each other. If a match is found, the interest point to which the neighborhood belongs is adjudged to be a correspondence point, and its coordinates are recorded.

2) Image Registration:

The process of image registration involves matching feature vectors in a set of images. To do that, direct alignment methods (like template matching) can be used to search for image alignments that minimize the sum of absolute differences between neighborhood pixels. Or, a more detailed approach using gradient directions can be used to find more accurate neighborhoods.

A brief explanation of the SIFT descriptor is as follows:

- The SIFT descriptor uses gradients to describe the neighborhood of an interest point.
- The orientations are sampled in the said neighborhood and a 16x16 array of samples is extracted and the dominant direction is subtracted from all 256 orientations to create a set of directions relative to a local coordinate system (which is defined by the dominant direction).
- We then divide the 16x16 neighborhood into 16 4x4 sub-regions, for which we create a histogram of directions. The histogram has 8 bins. So, for each interest point, we have a 16x8 = 128 element vector.

- To compare two interest points, we just compare their respective 128 length vectors using minimum distance classification. If the comparison gave us a good result, i.e. if the interest points are adjudged to be correspondences, we make a list of the two points in an array (created to hold the list of correspondences).

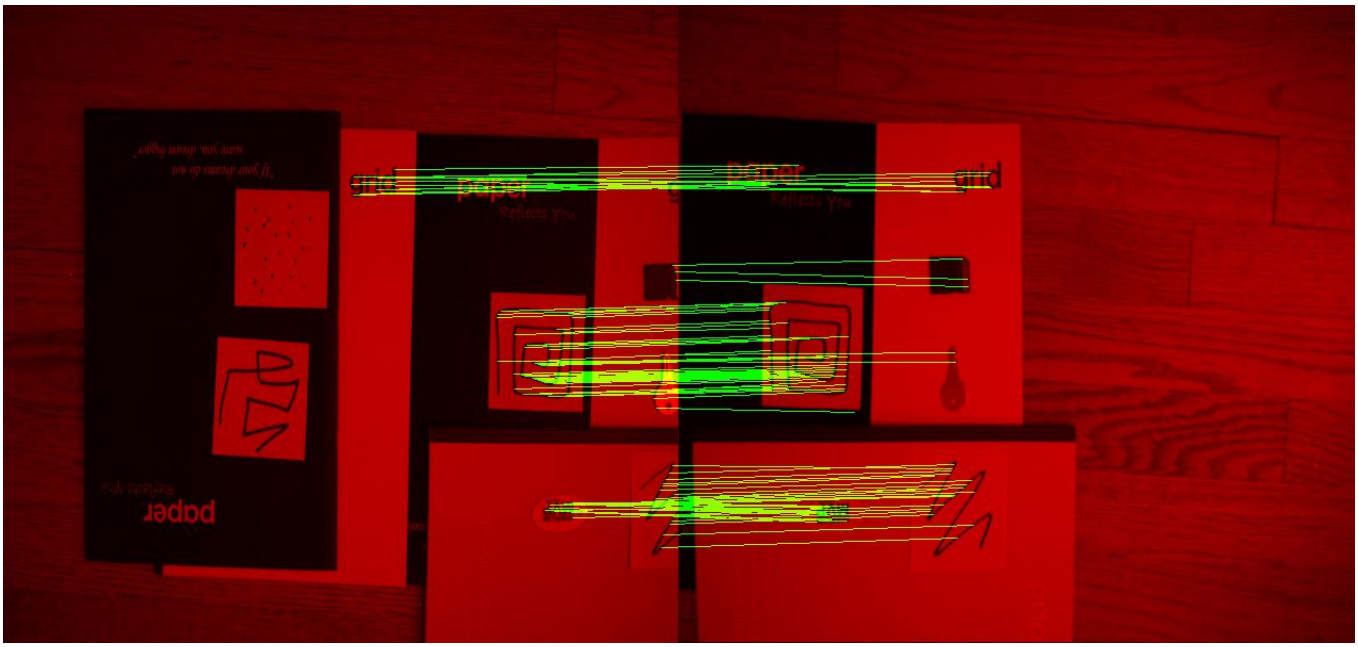
In order to derive an accurate H matrix from a multitude of correspondence points that we got using the above method, we use a technique called Random Sample Consensus (or RANSAC).

3) RANSAC:
   In our implementation of the RANSAC algorithm, we create an H matrix using a set of random points from the list of correspondences. Having found an H matrix, it is then used to calculate the correspondence pixels on the left image for every pixel on the right image. Since we already have the list of our corresponding interest points, we check if multiplying with the H matrix gives us the same result as the ones in the list.
   In the first run, we store the H matrix that we get using the above method. We then run the algorithm for an arbitrary (say 150) number of iterations. Each time, we check if the H matrix gives us a higher number of correspondence points that the previous one. If it does, we store the new H matrix. If it doesn't we discard the new H matrix and persist with the previously stored one.
   Although not extremely accurate, this algorithm is designed to give a probabilistically accurate result with an increase in the number of iterations.
   It should be noted that this is not the exact implementation of RANSAC. RANSAC uses the concept of inliers and outliers to probabilistically determine which points accurately correspond and which don't.

Images:

For Part 2, the images we stitched did not compare with the images we got in the first part. This was due to the fact that our implementation of the RANSAC algorithm was not able to provide us with an H matrix that would give an accurately stitched image. Hence, they've been omitted from this report.

However, to check the accuracy of our implementation of the SIFT descriptor, we designed an algorithm to place the two images side-by-side and to draw lines between every pair of corresponding points.

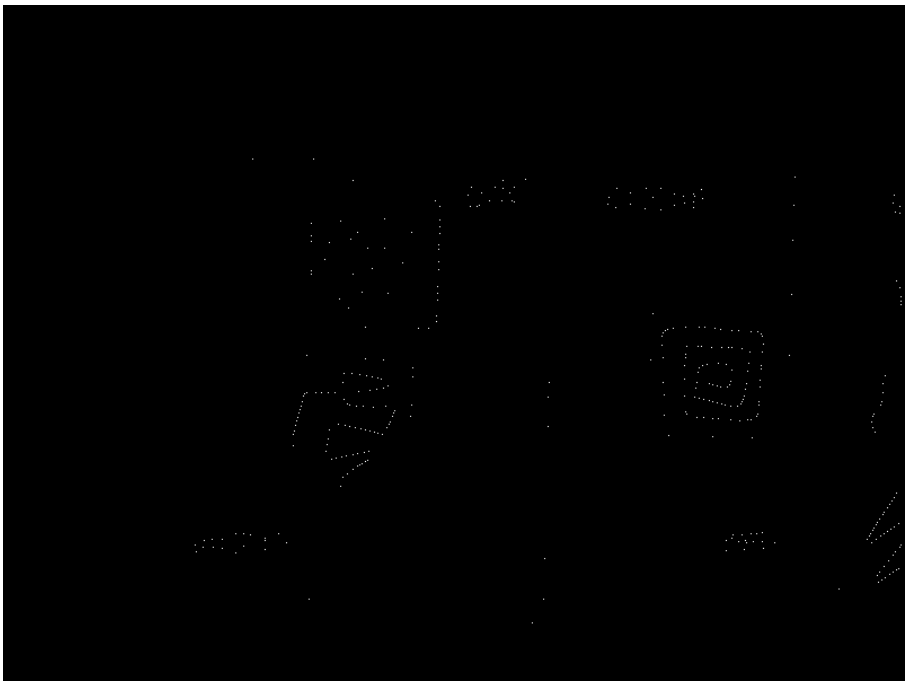


The image is tinted red so as to show the lines in green (thus providing a bright contrast)

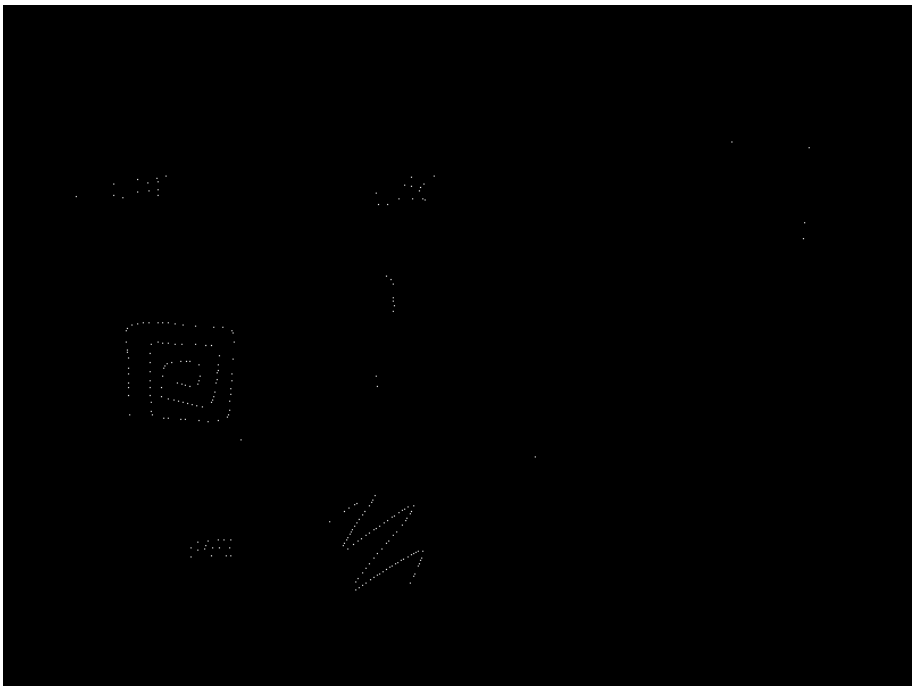
If we look carefully at the bottom-most post-it and the design we've drawn on it, we can find that the left and the right images have interest points that correspond extremely well. This shows that our implementation of the SIFT descriptor was accurate.

It is worth mentioning at this point that Template Matching techniques were also employed to find matches between corresponding interest points in the two images. This approach failed because of a rather high number of false matches that were found. Since sizeable chunks of both the images have similar composition, template matching would require large template neighborhoods, which would in turn, take a toll on the accuracy of matches.

The set of interest points we achieved for the left and right images are shown below.



LEFT



RIGHT

CONCLUSION

This project was designed to test us on our understanding of camera projections, interest operators, local area descriptors and feature vectors. However, having done extensive research on how to implement those algorithms in code has perhaps increased our learnings beyond the scope of what the project was initially designed for.

From our results, we can conclusively say that the first part of the project provided us with some highly accurate stitched images, especially given that some of the image pairs were slightly rotated). As for the image registration in the second part, our implementation of the SIFT descriptor provided us with accurate matches between corresponding interest points of the two images.

The stitched images produced by the algorithm in the second part were not nearly as good as those in the first part. We suspect that the primary reason for that was that our implementation of the RANSAC algorithm did not provide us with an accurate H matrix, despite running for over a hundred iterations. Since it's a probabilistic algorithm, we can only hope to improve it by running it for an even higher number of iterations to get an accurate result.

Our code is attached in the next part.

CODE

Appendix

Code for the PART 2 is after the complete code for PART 1

PART 1

Code Structure

Structure of the Directory:

Project:

- bin
  - main.out
- build
  - Makefile
- hdr
  - ***includeheaders.h*** *(inclusion of iostream,stdio,flip.h,ifs.h,math.h)*
  - ***main.h*** *(inclusion of includeheaders.h, commonFunctions.h)*
  - ***IFS2D.h*** *(IFS2D custom class to handle IFS images)*
  - ***IFS3D.h*** *(IFS3D custom class to handle 3D IFS images)*
- obj
  - commonFunctions.o
  - main.o
  - IFS2D.o
  - IFS3D.o

- output
  - stitched3.ifs (stitched image)
  - stitched4.ifs (to find the quality of stitching)
- src
  - commonFunctions.cpp
  - main.cpp
  - IFS2D.cpp (only header is pasted below)
  - IFS3D.cpp

The content of each file is printed in the following sections.

This code Pertains to the PART 1 of the project

```
1.  // main.cpp
2.
3.
4.  // my own header
5.  #include "../hdr/main.h"
6.
7.
8.  using namespace std;
9.  // constants
10. #define offset 100
11.
12. // function declarations
13.
14. //void transpose(int **srcarray,int **tararray,int rows,int cols);
15.
16. // global declarations
17. void printmatrix(float **matrix,int rows ,int col);
18. void printmatrixd(double **matrix,int rows ,int col);
19. void getHmatrix(float **Amatrix,float **Dmatrix,float **Hmatrix,int NumPts);
20. void getDmatrix(char filename[],float **Dmatrix);
21. void getAmatrix(char filename[],float **Amatrix);
22. void getCorPoint(float **Xmatrix,float **X_matrix,float **Hmatrix);
23. int getNumPoints(char filename[]);
24.
25. std::vector<int>  Corpx;
26. std::vector<int>  Corpx_;
27. std::vector<int>  Corpy;
28. std::vector<int>  Corpy_;
29. /*************************************************************/
30.
31. int main(int argc,char *argv[])
32. {
33.     int numPts=getNumPoints(argv[1]);
34.     float **Amatrix=matrix(1,numPts*2,1,8);
35.     float **Dmatrix=matrix(1,numPts*2,1,1);
36.     float **Hmatrix=matrix(1,3,1,3);
37.     getAmatrix(argv[1],Amatrix);
38.     printmatrix(Amatrix,numPts,8);
39.     getDmatrix(argv[1],Dmatrix);
40.     printmatrix(Dmatrix,numPts,1);
41.     getHmatrix(Amatrix,Dmatrix,Hmatrix,numPts);
42.     printmatrix(Hmatrix,3,3);
43.
44.     for(int index =0;index<Corpx.size();index++)
45.     {
46.         float **Xmatrix;
47.         float **Xdmatrix;
48.         Xmatrix=matrix(1,3,1,1);
49.         Xdmatrix=matrix(1,3,1,1);
50.         Xmatrix[1][1]=Corpx.at(index);
```

```
        Xmatrix[2][1]=Corpy.at(index);
        Xmatrix[3][1]=1;
        getCorPoint(Xmatrix,Xdmatrix,Hmatrix);
        //cout<<(int)Xdmatrix[1][1]<<" "<<(int)Xdmatrix[2][1]<<" "<<Xdmatrix[3][1]<<"\n";
        //cout<<Dmatrix[(index*2)+2][1]<<" "<<-Dmatrix[(index*2)+1][1]<<" 1\n";
    }

    /* stitch */
    IFS2D LeftImg (argv[2]);
    IFS2D RightImg (argv[3]);
    IFS2D StitchedImg3 ((char *)"float",LeftImg.getRows()*2,LeftImg.getColumns()*2);
    IFS2D StitchedImg4 ((char *)"float",LeftImg.getRows()*2,LeftImg.getColumns()*2);

    for(int row=0;row < LeftImg.getRows();row++)
    {
        for(int col=0; col <LeftImg.getColumns();col++)
        {
            StitchedImg3.putPixel(row+offset,col,LeftImg.getPixel(row,col));
            StitchedImg4.putPixel(row+offset,col,LeftImg.getPixel(row,col));
        }
        cout<<".";
    }
    double **dHmatrix=dmatrix(1,3,1,3);
    double **dInvHmatrix=dmatrix(1,3,1,3);
    cout<<"\n";
    for(int r=1;r<=3;r++)
    {
        for(int c=1;c<=3;c++)
        {
            dHmatrix[r][c]=Hmatrix[r][c];
            cout<<" "<<dHmatrix[r][c];
        }
        cout<<"\n";
    }
    cout<<ifsinverse(dHmatrix,dInvHmatrix,3);
    cout<<"\n";
    for(int r=1;r<=3;r++)
    {
        for(int c=1;c<=3;c++)
        {
            Hmatrix[r][c]=(float)dInvHmatrix[r][c];
            cout<<Hmatrix[r][c]<<" ";
        }
        cout<<"\n";
    }
    for(int row=0;row < StitchedImg3.getRows();row++)
    {
        for(int col=0; col < StitchedImg3.getColumns();col++)
        {
                            float **Xmatrix;
                            float **Xdmatrix;
                            Xmatrix=matrix(1,3,1,1);
                            Xdmatrix=matrix(1,3,1,1);
                            Xdmatrix[1][1]=row;
                            Xdmatrix[2][1]=col;
                            Xdmatrix[3][1]=1;
                            getCorPoint(Xdmatrix,Xmatrix,Hmatrix);
                            int rtrow=(int)Xmatrix[1][1];
                            int rtcol=(int)Xmatrix[2][1];
                            //cout<<row<<" "<<rtrow<<" "<<col<<" "<<rtcol<<"\n";
```

```
                        if(rtrow > 0 && rtrow < RightImg.getRows() && rtcol > 0 && rtcol < RightImg.getColumns())
                        {
                            StitchedImg3.putPixel(row+(offset),col,StitchedImg3.getPixel(row+offset,col)+ RightImg.getPixel(rtrow,rtcol));
                            StitchedImg4.putPixel(row+(offset),col,RightImg.getPixel(rtrow,rtcol));
                        }
                    }
                    cout<<".";

                }
                StitchedImg3.WritetoDisk((char *)"./../output/Stitched3.ifs");
                StitchedImg4.WritetoDisk((char *)"./../output/Stitched4.ifs");

                return 0;
            }
            void getCorPoint(float **Xmatrix,float **X_matrix,float **Hmatrix)
            {
                ifsmatmult(Hmatrix,Xmatrix,X_matrix,3,3,3,1);
                X_matrix[1][1]=X_matrix[1][1]/X_matrix[3][1];
                X_matrix[2][1]=X_matrix[2][1]/X_matrix[3][1];
                X_matrix[3][1]=X_matrix[3][1]/X_matrix[3][1];
            }

            int getNumPoints(char filename[])
            {
                std::fstream myfile(filename, std::ios_base::in);
                int a;
                int read=0;
                int count=0;
                std::vector<int> Corpx;
                /* get all the manual correspondences */
                while (myfile >> a)
                {
                    switch(read)
                    {
                        case 0:
                            Corpx.push_back(a);
                            break;
                        case 1:
                            break;
                        case 2:
                            break;
                        case 3:
                            break;
                        default:
                            break;
                    }
                    read++;
                    read=read%4;
                }
                return Corpx.size();
            }
            void getAmatrix(char filename[],float **Amatrix)
            {
                std::fstream myfile(filename, std::ios_base::in);
                int a;
                int read=0;
                /* get all the manual correspondences */
                while (myfile >> a)
                {
```

```
                    switch(read)
                    {
                        case 0:
                            Corpx.push_back(a);
                            break;
                        case 1:
                            Corpy.push_back(a);
                            break;
                        case 2:
                            Corpx_.push_back(a);
                            break;
                        case 3:
                            Corpy_.push_back(a);
                            break;
                        default:
                            break;
                    }
                    read++;
                    read=read%4;
                }
                for(int row_A=0;row_A < Corpx_.size();row_A++)
                {
                    int xi=Corpx.at((row_A));
                    int yi=Corpy.at((row_A));
                    int xi_=Corpx_.at((row_A));
                    int yi_=Corpy_.at((row_A));
                    int index=row_A*2+1;
                    Amatrix[index][1]=0;
                    Amatrix[index][2]=0;
                    Amatrix[index][3]=0;
                    Amatrix[index][4]=-xi;
                    Amatrix[index][5]=-yi;
                    Amatrix[index][6]=-1;
                    Amatrix[index][7]=xi*yi_;
                    Amatrix[index][8]=yi*yi_;

                    Amatrix[index+1][1]=xi;
                    Amatrix[index+1][2]=yi;
                    Amatrix[index+1][3]=1;
                    Amatrix[index+1][4]=0;
                    Amatrix[index+1][5]=0;
                    Amatrix[index+1][6]=0;
                    Amatrix[index+1][7]=-xi*xi_;
                    Amatrix[index+1][8]=-yi*xi_;
                }

            }
            void getDmatrix(char filename[],float **Dmatrix)
            {
                int a;
                std::fstream myfile(filename, std::ios_base::in);
                int read=0;
                /* get all the manual correspondences */
                int index=1;
                while (myfile >> a)
                {
                    switch(read)
                    {
                        case 0:
                            break;
                        case 1:
```

```
                            break;
                        case 2:
                            Dmatrix[index+1][1]=a;
                            //cout<<a<<" "<<index<<" "<<Dmatrix[index][1]<<"\n";
                            index++;
                            break;
                        case 3:
                            Dmatrix[index-1][1]=-a;
                            //cout<<a<<" "<<index<<" "<<Dmatrix[index][1]<<"\n";
                            index++;
                            break;
                        default:
                            break;
                    }
                    read++;
                    read=read%4;
                }

            }
            void getHmatrix(float **Amatrix,float **Dmatrix,float **Hmatrix,int NumPts)
            {
                float **AmatrixT,**ATxA,**finvATxA,**iATAxAT,**iATAATxD;
                double **invATxA;
                double **dATxA;
                AmatrixT = matrix(1,8,1,NumPts);
                ATxA=matrix(1,8,1,8);
                dATxA=dmatrix(1,8,1,8);
                invATxA=dmatrix(1,8,1,8);
                finvATxA=matrix(1,8,1,8);
                iATAxAT=matrix(1,8,1,NumPts);
                iATAATxD=matrix(1,8,1,1);

                transpose(Amatrix,NumPts,8,AmatrixT);
                ifsmatmult(AmatrixT,Amatrix,ATxA,8,NumPts,NumPts,8);
                for(int r=1;r<=8;r++)
                {
                    for(int c=1;c<=8;c++)
                    {
                        dATxA[r][c]=ATxA[r][c];
                    }
                }
                ifsinverse(dATxA,invATxA,8);
                for(int r=1;r<=8;r++)
                {
                    for(int c=1;c<=8;c++)
                    {
                        finvATxA[r][c]=(float)invATxA[r][c];
                    }
                }
                ifsmatmult(finvATxA,AmatrixT,iATAxAT,8,8,8,NumPts);
                ifsmatmult(iATAxAT,Dmatrix,iATAATxD,8,NumPts,NumPts,1);
                //getchar()();

                /*
                      cout<<"Amatrix\n";
                      printmatrix(Amatrix,NumPts,8);

                      cout<<"Amatrix Transpose \n";
                      printmatrix(AmatrixT,8,NumPts);

                      cout<<"Amatrix x A transpose \n";
```

```
                        printmatrix(ATxA,8,8);

                        cout<<"Amatrix x A transpose in double\n";
                        printmatrixd(dATxA,8,8);// double format of AT * A

                        cout<<"inverse of Amatrix x A Transpose\n";
                        printmatrixd(invATxA,8,8);// inverse of AT * A

                        cout<<"float format of inverse \n";
                        printmatrix(finvATxA,8,8);// float format of inverse of AT * A

                        cout<<"inverse x A transpose\n";
                        printmatrix(iATAxAT,8,NumPts);

                        cout<<"Dmatrix\n";
                        printmatrix(Dmatrix,NumPts,1);

                        cout<<"H\n";
                        printmatrix(iATAATxD,8,1);
                        */
                    Hmatrix[1][1]=iATAATxD[1][1];
                    Hmatrix[1][2]=iATAATxD[2][1];
                    Hmatrix[1][3]=iATAATxD[3][1];
                    Hmatrix[2][1]=iATAATxD[4][1];
                    Hmatrix[2][2]=iATAATxD[5][1];
                    Hmatrix[2][3]=iATAATxD[6][1];
                    Hmatrix[3][1]=iATAATxD[7][1];
                    Hmatrix[3][2]=iATAATxD[8][1];
                    Hmatrix[3][3]=1;


                }
                void printmatrix(float **matrix,int rows ,int col)
                {
                    cout<<"\n***************************************\n";
                    for(int r=0;r<rows;r++)
                    {
                        cout<<r<<"\t";
                        for(int c=0;c<col;c++){
                            cout<<matrix[r+1][c+1]<<"\t";
                        }
                        cout<<"\n";
                    }

                }
                void printmatrixd(double **matrix,int rows ,int col)
                {
                    cout<<"\n***************************************\n";
                    for(int r=0;r<rows;r++)
                    {
                        cout<<r<<"\t";
                        for(int c=0;c<col;c++){
                            cout<<matrix[r+1][c+1]<<" ";
                        }
                        cout<<"\n";
                    }

                }
```

IFS2D.h

```
#ifndef _includeheaders_
#define _includeheaders_
#include "includeheaders.h"
#endif

class IFS2D
{
    private:
        IFSIMG img;
        int *len;
        int rows;
        int columns;
        float **ptr;
    protected:

    public:
        IFS2D(char[]);
        IFS2D(IFSIMG);
        IFS2D(const IFS2D&);
        IFS2D(char[],int , int);

        int* getSize(void);
        int getRows(void);
        int getColumns(void);
        float** getDataPtr(void);
        float getPixel(int,int);
        void putPixel(int,int,float);

        float getMax(void);
        float getMin(void);
        float getMean(void);
        float getMedian(void);
        float getMode(void);
        float getVariance(void);
        void getHistogram(void);
        void getHistogramRow(int);
        void getHistogramColumn(int);

        void add(IFS2D,IFS2D);
        void subtract(IFS2D,IFS2D);
        void multiply(IFS2D,IFS2D);
        void divide(IFS2D,IFS2D,float);
        void add(IFS2D);
        void subtract(IFS2D);
        void multiply(IFS2D);
        void divide(IFS2D,float);
        void powerOf(float);
        void square(void);

        void copyImg(IFS2D);
        void threshold(float);
        void scale(float,float);
        void applyKernel(float[]);
        void applyKernelXY(float[]);
        void Gauss(float);
```

```
56.         void DOG(float,int,int);
57.         void DOGx(float);
58.         void DOGy(float);
59.         void DOGx2(float);
60.         void DOGy2(float);
61.         void Laplacian(float);
62.         void QuadraticVariation();
63.
64.         void GradientMagnitude(float);
65.         void GradientDirection(float);
66.
67.         void SavetoText(char[]);
68.         int WritetoDisk(char[]);
69.         void SubSample(void);
70.         void getNeighborhood(int,int,float*);
71. };
```

PART 2

Code Structure

Structure of the Directory:

Project:

- bin
  - main.out
- build
  - Makefile
- hdr
  - ***includeheaders.h*** *(inclusion of iostream,stdio,flip.h,ifs.h,math.h)*
  - ***main.h*** *(inclusion of includeheaders.h, commonFunctions.h)*
  - ***FIndHMatrix.h***
  - ***IFS2D.h*** *(IFS2D custom class to handle IFS images)*
  - ***IFS3D.h*** *(IFS3D custom class to handle 3D IFS images)*
- obj
  - commonFunctions.o
  - main.o
  - IFS2D.o
  - IFS3D.o

- output
  - stitched3.ifs (stitched image)
  - stitched4.ifs (to find the quality of stitching)
- src
  - commonFunctions.cpp
  - main.cpp
  - FindHMatrix.cpp **(RANSAC)**
  - IFS2D.cpp (only header is pasted below)

- IFS3D.cpp

PART 2

Main.cpp

```
1.  // main.cpp
2.
3.
4.  // my own header
5.  #include "../hdr/main.h"
6.
7.
8.  using namespace std;
9.  // constants
10. #define Scale 1
11. #define ErrCheck 10
12. #define NeiSiz 20
13. #define Diffthreshold 10
14. #define offset 100
15. //#define threshold 10
16. typedef std::vector<int> IntPoints8;
17. typedef std::vector<IntPoints8> Sector16;
18. typedef std::vector<Sector16> imgVector;
19. // function declarations
20.     template <typename T>
21. string ntos ( T Number ,string name)
22. {
23.     ostringstream ss;
24.     ss << Number;
25.     string path="./../output/"+name+"_"+ss.str()+".ifs";
26.     return path;
27. }
28. void pause(int set)
29. {
30.     static int count=0;
31.     cout<<"\nstopped at :"<<count;
32.     if(set==1)
33.     {
34.         cout<<"\n Press enter to continue :";
35.         getchar();
36.     }
37.     count++;
38. }
39. void FindIntPt(std::vector<long>& Pts,IFS2D src_img,int filecount);
40. void FindTemplateCorrespondences(IFS2D img1,IFS2D img2,std::vector<long>& img1_IntPt,std::vector<long>& img2_I
    ntPt);
41. void FindSIFTCorrespondences(IFS2D img1,IFS2D img2,std::vector<long>& img1_IntPt,std::vector<long>& img2_IntPt
    );
```

```
int CheckIfMax(int r,int c,int sigma);
float compareMin(float arr_1[],float arr_2[],float arr_3[]);
inline void GetNeighbor(IFS2D src_img,float neighbor[NeiSiz*2][NeiSiz*2],int rowInd,int colInd);
float CompareNeighborPercent(float neighbor1[NeiSiz*2][NeiSiz*2],float neighbor2[NeiSiz*2][NeiSiz*2]);
float CompareNeighborSum(float neighbor1[NeiSiz*2][NeiSiz*2],float neighbor2[NeiSiz*2][NeiSiz*2]);
void getFeatureVector(IFS2D Gradimg,int rowInd,int colInd,std::vector< std::vector < std::vector <int> > >& IM
GVector,int index);
void matchVectors(imgVector& img1Vector,imgVector& img2Vector,int img1Pts,int img2Pts);
void DrawLine(IFS2D img,int x0,int y0,int x1,int y1);
void Line( IFS2D &img,const float x1, const float y1, const float x2, const float y2);
void Line3D( IFS3D &img,const float x1, const float y1, const float x2, const float y2);
void DrawCorrespondences(IFS2D img1,IFS2D img2,char filename[]);
void getCorPoint(float **Xmatrix,float **X_matrix,float **Hmatrix);

// global declarations
const float Sigma[]={0.707*Scale,1.0*Scale,1.414*Scale,2.0*Scale,2.83*Scale};
float threshold=0;
float PercentThreshold=0;
//const float Sigma[]={2.828,4.0,5.656,8.0,11.32};

/*************************************************************/

int main(int argc,char *argv[])
{
    IFS2D LeftImg (argv[1]);
    IFS2D RightImg (argv[2]);
    IFS2D StitchedImg ((char *)"float",LeftImg.getRows()*2,RightImg.getRows()*2);
    IFS2D StitchedImg2 ((char *)"float",LeftImg.getRows()*2,RightImg.getRows()*2);
    threshold=atof(argv[3]);
    PercentThreshold=atof(argv[4]);
    std::vector<long> img1_IntPt;
    std::vector<long> img2_IntPt;

    FindIntPt(img1_IntPt,LeftImg,1);
    FindIntPt(img2_IntPt,RightImg,2);

    cout<<"\nNumber of img1 Interest Pts :"<<img1_IntPt.size();
    cout<<"\nNumber of img2 Interest Pts :"<<img2_IntPt.size();

    LeftImg.WritetoDisk((char *)"./../output/firstnormal.ifs");
    RightImg.WritetoDisk((char *)"./../output/secondnormal.ifs");
    //FindTemplateCorrespondences(LeftImg,RightImg,img1_IntPt,img2_IntPt);
    FindSIFTCorrespondences(LeftImg,RightImg,img1_IntPt,img2_IntPt);
    DrawCorrespondences(LeftImg,RightImg,(char *)"./../output/BookInput.txt");
    float **Hmatrix=matrix(1,3,1,3);
    FindHMatrix((char *)"./../output/BookInput.txt",Hmatrix);

    for(int row=0;row < LeftImg.getRows();row++)
    {
        for(int col=0; col <LeftImg.getColumns();col++)
        {
            StitchedImg.putPixel(row+offset,col,LeftImg.getPixel(row,col));
            StitchedImg2.putPixel(row+offset,col,LeftImg.getPixel(row,col));
        }
    }
    for(int row=0;row < StitchedImg.getRows();row++)
    {
        for(int col=0; col < StitchedImg.getColumns();col++)
        {
                        float **Xmatrix;
                        float **Xdmatrix;
```

```
                            Xmatrix=matrix(1,3,1,1);
                            Xdmatrix=matrix(1,3,1,1);
                            Xdmatrix[1][1]=row;
                            Xdmatrix[2][1]=col;
                            Xdmatrix[3][1]=1;
                            getCorPoint(Xdmatrix,Xmatrix,Hmatrix);
                            //cout<<"right "<<(int)Xmatrix[1][1]<<" "<<(int)Xmatrix[2][1]<<" left "<<(int)Xdmatrix[1][1]<<" "<<(int)Xdmatrix[2][1]<<"\n";
                            int rtrow=(int)Xmatrix[1][1];
                            int rtcol=(int)Xmatrix[2][1];
                            if(rtrow > 0 && rtrow < RightImg.getRows() && rtcol > 0 && rtcol < RightImg.getColumns())
                            {
                                StitchedImg.putPixel(row+(offset),col,StitchedImg.getPixel(row+offset,col)+ RightImg.getPixel(rtrow,rtcol));
                                StitchedImg2.putPixel(row+(offset),col,RightImg.getPixel(rtrow,rtcol));
                            }
                        }
                    }
                    pause(1);
                    StitchedImg.WritetoDisk((char *)"./../output/Stitched.ifs");
                    pause(1);
                    StitchedImg2.WritetoDisk((char *)"./../output/Stitched2.ifs");
                    return 0;
                }
                void getCorPoint(float **Xmatrix,float **X_matrix,float **Hmatrix)
                {
                    ifsmatmult(Hmatrix,Xmatrix,X_matrix,3,3,3,1);
                    X_matrix[1][1]=X_matrix[1][1]/X_matrix[3][1];
                    X_matrix[2][1]=X_matrix[2][1]/X_matrix[3][1];
                    X_matrix[3][1]=X_matrix[3][1]/X_matrix[3][1];
                }
                void DrawCorrespondences(IFS2D img1,IFS2D img2,char filename[])
                {
                    IFS2D finalImg ((char *)"float",img2.getRows(),img1.getColumns()+img2.getColumns());
                    IFS3D final3D ((char *)"float",img2.getRows(),img1.getColumns()+img2.getColumns(),3);

                    for(int r=0;r<finalImg.getRows();r++)
                    {
                        for(int c=0;c<finalImg.getColumns();c++)
                        {
                            if(c>img1.getColumns())
                            {
                                finalImg.putPixel(r,c,img2.getPixel(r,c));
                                final3D.putPixel(r,c,0,img2.getPixel(r,c));
                            }
                            else
                            {
                                finalImg.putPixel(r,c,img1.getPixel(r,c));
                                final3D.putPixel(r,c,0,img1.getPixel(r,c));
                            }

                        }

                    }
                    std::fstream myfile(filename,std::ios_base::in);
                    int a;
                    int read=0;
                    /* get all the manual correspondences */
                    std::vector<int>  AllCorpx;
                    std::vector<int>  AllCorpx_;
```

```
160.                    std::vector<int>  AllCorpy;
161.                    std::vector<int>  AllCorpy_;
162.                    while (myfile >> a)
163.                    {
164.                        switch(read)
165.                        {
166.                            case 0:
167.                                AllCorpx.push_back(a);
168.                                break;
169.                            case 1:
170.                                AllCorpy.push_back(a);
171.                                break;
172.                            case 2:
173.                                AllCorpx_.push_back(a);
174.                                break;
175.                            case 3:
176.                                AllCorpy_.push_back(a);
177.                                break;
178.                            default:
179.                                break;
180.                        }
181.                        read++;
182.                        read=read%4;
183.                    }
184.                    for(int index=0;index<AllCorpx.size();index++)
185.                    {
186.                        cout<<"\n"<<AllCorpx[index]<<" "<<AllCorpy[index]<<" "<<AllCorpx_[index]<<" "<<AllCorpy
    _[index]+img1.getColumns();
187.                        Line(finalImg,AllCorpx_[index],AllCorpy_[index],AllCorpx[index],AllCorpy[index]+img1.ge
    tColumns());
188.                        Line3D(final3D,AllCorpx_[index],AllCorpy_[index],AllCorpx[index],AllCorpy[index]+img1.g
    etColumns());
189.                        cout<<".";
190.                    }
191.                    finalImg.WritetoDisk((char *)"./../output/final.ifs");
192.                    final3D.WritetoDisk((char *)"./../output/final.ifs");
193.                }
194.                void FindSIFTCorrespondences(IFS2D img1,IFS2D img2,std::vector<long>& img1_IntPt,std::vector<lo
    ng>& img2_IntPt)
195.                {
196.                    ofstream outputFile("./../output/BookInput.txt");
197.                    IFS2D img1Grad (img1);
198.                    IFS2D img2Grad (img2);
199.                    img1.GradientDirection(1);
200.                    img2.GradientDirection(2);
201.
202.                    int img1_NumPt=img1_IntPt.size();/* Number of INterest points in Image 1*/
203.                    int img2_NumPt=img2_IntPt.size();/* NUmber of INterest points in Image 2 */
204.                    imgVector img1Vector(img1_NumPt,Sector16(16,IntPoints8(8,0)));
205.                    imgVector img2Vector(img1_NumPt,Sector16(16,IntPoints8(8,0)));
206.
207.                    /* Get the feaure vector for img 1*/
208.                    cout<<"\nGetting Feature Vectors of Img1";
209.                    for(int index=0 ;index != img1_NumPt ; index++)
210.                    {
211.                        int img_cols = img1.getColumns();
212.                        int rowInd = img1_IntPt.at( index ) / img_cols;
213.                        int colInd = img1_IntPt.at( index ) % img_cols;
214.                        getFeatureVector(img1Grad,rowInd,colInd,img1Vector,index);
215.                    }
216.                    cout<<"\nGetting Feature Vectors of Img2";
```

```
                for(int index=0 ;index != img2_NumPt ; index++)
                {
                    int img_cols = img2.getColumns();
                    int rowInd = img2_IntPt.at( index ) / img_cols;
                    int colInd = img2_IntPt.at( index ) % img_cols;
                    getFeatureVector(img1Grad,rowInd,colInd,img2Vector,index);
                }
                cout<<"\nFinding Best Match";
                std::vector<int> img1MatchPt(img1_NumPt,-1);
                std::vector<int> img2MatchPt(img2_NumPt,-1);
                int countPts=0;
                int img1Cols = img1.getColumns();
                int img2Cols = img2.getColumns();
                for(int img1Index=0 ;img1Index != img1_NumPt ; img1Index++)
                {
                    int img1Row = img1_IntPt[img1Index] / img1Cols;
                    int img1Col = img1_IntPt[img1Index] % img1Cols;
                    int minerror=PercentThreshold;
                    int minerrorIndex=-1;
                    for(int img2Index=0 ;img2Index != img2_NumPt ; img2Index++)
                    {
                        int error=0;
                        int img2Row = img2_IntPt[img2Index] / img2Cols;
                        int img2Col = img2_IntPt[img2Index] % img2Cols;
                        if(img2MatchPt[img2Index]== -1 && (abs(img1Row-
   img2Row) < Diffthreshold) && (img1Col > (img1Cols/2)) && (img2Col < (img2Cols/2)))
                        {
                            for(int Rindex=0;Rindex<16;Rindex++)
                            {
                                for(int Cindex=0;Cindex<8;Cindex++)
                                {
                                    int img1Val=img1Vector[img1Index][Rindex][Cindex];
                                    int img2Val=img2Vector[img2Index][Rindex][Cindex];
                                    error+= abs((int)sqrt((img1Val*img1Val)-(img2Val*img2Val)));
                                }
                            }
                            if(minerror>error)
                            {
                                minerrorIndex=img2Index;
                                minerror=error;
                            }
                        }
                    }
                    if(minerrorIndex != -1)
                    {
                        int img2Row = img2_IntPt[minerrorIndex] / img2Cols;
                        int img2Col = img2_IntPt[minerrorIndex] % img2Cols;
                        img1MatchPt[img1Index] = minerrorIndex;
                        img2MatchPt[minerrorIndex] = img1Index;
                        cout<<"\n"<<img2Row<<"\t"<<img2Col<<"\t"<<img1Row<<"\t"<<img1Col;
                        outputFile<<"\n"<<img2Row<<"\t"<<img2Col<<"\t"<<img1Row<<"\t"<<img1Col;
                        //cout<<" "<<minerror<<" "<<minerrorIndex;;
                        countPts++;
                    }
                }
                cout<<"\n$$$"<<countPts;

            }
            void DrawLine(IFS2D img,int x0,int y0,int x1,int y1)
            {
                float deltax = x1 - x0;
```

```
277.                 float deltay = y1 - y0;
278.                 float error = -1.0;
279.                 float deltaerr = fabs(deltay / deltax);    // Assume deltax != 0 (line is not vertical),
280.                 // note that this division needs to be done in a way that preserves the fractional part
281.                 int y = y0;
282.                 for(int x=x0;x<(x1-1);x++)
283.                 {
284.                     img.putPixel(x,y,img.getMax()+50);
285.                     error = error + deltaerr;
286.                     if(error >= 0.0)
287.                     {
288.                         y = y + 1;
289.                         error = error - 1.0;
290.                     }
291.                 }
292.             }
293.             void Line3D( IFS3D &img,float x1, float y1, float x2,  float y2)
294.             {
295.                 //Bresenham's line algorithm
296.                 const bool steep = (fabs(y2 - y1) > fabs(x2 - x1));
297.                 if(steep)
298.                 {
299.                     std::swap(x1, y1);
300.                     std::swap(x2, y2);
301.                 }
302.
303.                 if(x1 > x2)
304.                 {
305.                     std::swap(x1, x2);
306.                     std::swap(y1, y2);
307.                 }
308.
309.                 const float dx = x2 - x1;
310.                 const float dy = fabs(y2 - y1);
311.
312.                 float error = dx / 2.0f;
313.                 const int ystep = (y1 < y2) ? 1 : -1;
314.                 int y = (int)y1;
315.                 const int maxX = (int)x2;
316.                 for(int x=(int)x1; x<maxX; x++)
317.                 {
318.                     if(steep)
319.                     {
320.                         img.putPixel(y,x,1,255);
321.                     }
322.                     else
323.                     {
324.                         img.putPixel(x,y,1,255);
325.                     }
326.
327.                     error -= dy;
328.                     if(error < 0)
329.                     {
330.                         y += ystep;
331.                         error += dx;
332.                     }
333.                 }
334.             }
335.             void Line( IFS2D &img,float x1, float y1, float x2,  float y2)
336.             {
337.                 //Bresenham's line algorithm
```

```
338.                 const bool steep = (fabs(y2 - y1) > fabs(x2 - x1));
339.                 if(steep)
340.                 {
341.                     std::swap(x1, y1);
342.                     std::swap(x2, y2);
343.                 }
344.
345.                 if(x1 > x2)
346.                 {
347.                     std::swap(x1, x2);
348.                     std::swap(y1, y2);
349.                 }
350.
351.                 const float dx = x2 - x1;
352.                 const float dy = fabs(y2 - y1);
353.
354.                 float error = dx / 2.0f;
355.                 const int ystep = (y1 < y2) ? 1 : -1;
356.                 int y = (int)y1;
357.                 const int maxX = (int)x2;
358.                 float **data=img.getDataPtr();
359.                 for(int x=(int)x1; x<maxX; x++)
360.                 {
361.                     if(steep)
362.                     {
363.                         data[y][x]=255;
364.                     }
365.                     else
366.                     {
367.                         data[x][y]=255;
368.                     }
369.
370.                     error -= dy;
371.                     if(error < 0)
372.                     {
373.                         y += ystep;
374.                         error += dx;
375.                     }
376.                 }
377.             }
378.             void getFeatureVector(IFS2D Gradimg,int rowInd,int colInd,imgVector& IMGVector,int index)
379.             {
380.
381.                 /* Get the feaure vector for img 1*/
382.                 float neighbor16x16[16][16];
383.                 int img_cols = Gradimg.getColumns();
384.                 int img_rows = Gradimg.getRows();
385.                 float **dataptr= Gradimg.getDataPtr();
386.                 int neighborrow=0;
387.                 float DominantDirection=0.0;
388.                 float DominantDirectioncount=0.0;
389.                 std::vector<float> histogramVal;
390.                 std::vector<float> histogramCount;
391.                 /* For each point  get the 16x16 */
392.                 std::vector<float> Directionlist;
393.                 for(int row = rowInd-8 ; row != rowInd+8 ; row++)
394.                 {
395.                     int neighborcol=0;
396.                     for(int col = colInd-8 ; col != colInd+8 ; col++)
397.                     {
398.                         if(row >= 0 && row < img_rows && col >= 0 && col < img_cols)
```

```
                        {
                            neighbor16x16[neighborrow][neighborcol]=dataptr[row][col];
                            Directionlist.push_back(neighbor16x16[neighborrow][neighborcol]);
                        }
                        else
                        {
                            neighbor16x16[neighborrow][neighborcol]=0;
                        }
                        neighborcol++;
                    }
                    neighborrow++;
                }/* for get gthe nneighbor */
                std::sort (Directionlist.begin(),Directionlist.end());
                /* For each point  Compute the histogram*/
                for(int listind=0;listind < (int)Directionlist.size();listind++)
                {
                    int count=0;
                    int flag=0;
                    while(count < (int)histogramCount.size())
                    {
                        if(histogramVal.at(count) == Directionlist.at(listind))
                        {
                            flag=count;
                            break;
                        }
                        else
                        {
                            count++;
                        }
                    }
                    if(flag == 0)
                    {
                        histogramVal.push_back(Directionlist.at(listind));
                        histogramCount.push_back(1);
                    }
                    else
                    {
                        histogramCount.back()=histogramCount.back()+1;
                    }
                }/* get the histogram */

                /* Find the dominant Direction */
                for(int listind=0;listind < (int)histogramCount.size();listind++)
                {
                    if(DominantDirectioncount < histogramCount.at(listind))
                    {
                        DominantDirection = histogramVal.at(listind);
                        DominantDirectioncount = histogramCount.at(listind);
                    }
                }
                /* ind the direction relative to dominant direction */
                for(int row=0;row < neighborrow ; row++)
                {
                    for(int col=0;col < neighborrow ; col++)
                    {
                        neighbor16x16[row][col]=neighbor16x16[row][col]-DominantDirection;
                    }
                }
                /* quantize the direction into 8 direcction */
                for(int row=0;row < neighborrow ; row++)
                {
```

```
460.                     for(int col=0;col < neighborrow ; col++)
461.                     {
462.                         float degrees=neighbor16x16[row][col];
463.                         if(degrees < 0)
464.                         {
465.                             degrees=degrees+360;
466.                         }
467.                         if(degrees <= 22.5)
468.                         {
469.                             degrees=0;
470.                         }
471.                         else if(degrees <= 67.5 && degrees > 22.5)
472.                         {
473.                             degrees=1;
474.                         }
475.                         else if(degrees <= 112.5 && degrees > 67.5)
476.                         {
477.                             degrees=2;
478.                         }
479.                         else if(degrees <= 157.5 && degrees > 112.5)
480.                         {
481.                             degrees=3;
482.                         }
483.                         else if(degrees <= 202.5 && degrees > 157.5)
484.                         {
485.                             degrees=4;
486.                         }
487.                         else if(degrees <= 247.5 && degrees > 202.5)
488.                         {
489.                             degrees=5;
490.                         }
491.                         else if(degrees <= 292.5 && degrees > 247.5)
492.                         {
493.                             degrees=6;
494.                         }
495.                         else if(degrees <= 337.5 && degrees > 292.5)
496.                         {
497.                             degrees=7;
498.                         }
499.                         if(degrees > 337.5)
500.                         {
501.                             degrees=0;
502.                         }
503.                         neighbor16x16[row][col]=degrees;
504.                     }
505.                 }
506.                 /* compute histogram for each 4x4 */
507.                 int vectorCount=0;
508.                 for(int row=0;row<4;row++)
509.                 {
510.                     for(int col=0;col<4;col++)
511.                     {
512.                         int val=(int)neighbor16x16[row][col];
513.                         IMGVector[index][vectorCount][val]++;
514.
515.                     }
516.                     for(int col=4;col<8;col++)
517.                     {
518.                         int val=(int)neighbor16x16[row][col];
519.                         IMGVector[index][vectorCount+1][val]++;
520.                     }
```

```
                    for(int col=8;col<12;col++)
                    {
                        int val=(int)neighbor16x16[row][col];
                        IMGVector[index][vectorCount+2][val]++;
                    }
                    for(int col=12;col<16;col++)
                    {
                        int val=(int)neighbor16x16[row][col];
                        IMGVector[index][vectorCount+3][val]++;
                    }
                }
                vectorCount=vectorCount+4;

                for(int row=4;row<8;row++)
                {
                    for(int col=0;col<4;col++)
                    {
                        int val=(int)neighbor16x16[row][col];
                        IMGVector[index][vectorCount][val]++;
                    }
                    for(int col=4;col<8;col++)
                    {
                        int val=(int)neighbor16x16[row][col];
                        IMGVector[index][vectorCount+1][val]++;
                    }
                    for(int col=8;col<12;col++)
                    {
                        int val=(int)neighbor16x16[row][col];
                        IMGVector[index][vectorCount+2][val]++;
                    }
                    for(int col=12;col<16;col++)
                    {
                        int val=(int)neighbor16x16[row][col];
                        IMGVector[index][vectorCount+3][val]++;
                    }

                }
                vectorCount=vectorCount+4;

                for(int row=8;row<12;row++)
                {
                    for(int col=0;col<4;col++)
                    {
                        int val=(int)neighbor16x16[row][col];
                        IMGVector[index][vectorCount][val]++;
                    }
                    for(int col=4;col<8;col++)
                    {
                        int val=(int)neighbor16x16[row][col];
                        IMGVector[index][vectorCount+1][val]++;
                    }
                    for(int col=8;col<12;col++)
                    {
                        int val=(int)neighbor16x16[row][col];
                        IMGVector[index][vectorCount+2][val]++;
                    }
                    for(int col=12;col<16;col++)
                    {
                        int val=(int)neighbor16x16[row][col];
                        IMGVector[index][vectorCount+3][val]++;
                    }
```

```
582.
583.                     }
584.
585.                     vectorCount=vectorCount+4;
586.                     for(int row=12;row<16;row++)
587.                     {
588.                         for(int col=0;col<4;col++)
589.                         {
590.                             int val=(int)neighbor16x16[row][col];
591.                             IMGVector[index][vectorCount][val]++;
592.                         }
593.                         for(int col=4;col<8;col++)
594.                         {
595.                             int val=(int)neighbor16x16[row][col];
596.                             IMGVector[index][vectorCount+1][val]++;
597.                         }
598.                         for(int col=8;col<12;col++)
599.                         {
600.                             int val=(int)neighbor16x16[row][col];
601.                             IMGVector[index][vectorCount+2][val]++;
602.                         }
603.                         for(int col=12;col<16;col++)
604.                         {
605.                             int val=(int)neighbor16x16[row][col];
606.                             IMGVector[index][vectorCount+3][val]++;
607.                         }
608.
609.                     }
610.                 }
611.                 void FindTemplateCorrespondences(IFS2D img1,IFS2D img2,std::vector<long>& img1_IntPt,std::vecto
    r<long>& img2_IntPt)
612.                 {
613.
614.                     ofstream outputFile("./../output/BookInput.txt");
615.                     int img1_NumPt=img1_IntPt.size();/* Number of INterest points in Image 1*/
616.                     int img2_NumPt=img2_IntPt.size();/* NUmber of  INterest points in IMaged 2 */
617.
618.                     //
619.                     /* For each point in image 1 find the best match in Image 2 */
620.                     float maxever=0.0;
621.                     float img1_neighbor[NeiSiz*2][NeiSiz*2];
622.                     float img2_neighbor[NeiSiz*2][NeiSiz*2];
623.                     std::vector<int> matched;
624.                     for(int img2_index=0 ; img2_index != img2_NumPt ; img2_index++)
625.                     {
626.                         matched.push_back(0);
627.                     }
628.                     int maxcount=0;
629.                     for(int img1_index=0 ;img1_index != img1_NumPt ; img1_index++)
630.                     {
631.                         int img1_cols = img1.getColumns();
632.                         int img1_rowInd = img1_IntPt.at( img1_index ) / img1_cols;
633.                         int img1_colInd = img1_IntPt.at( img1_index ) % img1_cols;
634.                         GetNeighbor(img1,img1_neighbor,img1_rowInd,img1_colInd);
635.                         /*float maxPercent=0;
636.                           int maxPercentRow=0;
637.                           int maxPercentCol=0;
638.                           int maxPercentInd=0;
639.                           int maxPercentnewRow=0;
640.                           int maxPercentnewCol=0;
641.                           int maxPercentnewInd=0;*/
```

```
642.                        float minSum=FLT_MAX;
643.                        int minSumRow=0;
644.                        int minSumCol=0;
645.                        int minSumInd=0;
646.                        int minSumnewRow=0;
647.                        int minSumnewCol=0;
648.                        int count=0;
649.                        int img2_cols = img2.getColumns();
650.                        int img2_rows = img2.getRows();
651.                        float **dataptr=img2.getDataPtr();
652.                        for(int img2_index=0 ; img2_index != img2_NumPt ; img2_index++)
653.                        {
654.                            int img2_rowInd = 0;
655.                            int img2_colInd = 0;
656.                            img2_rowInd = img2_IntPt.at( img2_index ) / img2_cols;
657.                            img2_colInd = img2_IntPt.at( img2_index ) % img2_cols;
658.                            if(matched.at(img2_index) == 0 && abs(img2_rowInd-
    img1_rowInd) < 10 && img1_colInd > img1_cols/2 && img2_colInd < img2_cols/2)
659.                            {
660.                                count++;
661.                                for(int newRowInd = img2_rowInd-
    ErrCheck;newRowInd != img2_rowInd+ErrCheck ; newRowInd++)
662.                                {
663.                                    for(int newColInd = img2_colInd-
    ErrCheck;newColInd != img2_colInd+ErrCheck ; newColInd++)
664.                                    {
665.                                        /*************************************************************************
    ****************/
666.                                        int NeighborRow=0;
667.                                        for(int row = newRowInd-NeiSiz ; row != newRowInd + NeiSiz ; row++)
668.                                        {
669.                                            int NeighborCol=0;
670.                                            for(int col = newColInd-NeiSiz ; col != newColInd+NeiSiz ; col++)
671.                                            {
672.                                                if(row >= 0 && row <(int)img2_rows && col >= 0 && col < (int)im
    g2_cols)
673.                                                {
674.                                                    img2_neighbor[NeighborRow][NeighborCol]=dataptr[row][col];

675.                                                }
676.                                                else
677.                                                {
678.                                                    img2_neighbor[NeighborRow][NeighborCol]=-1.0;
679.                                                }
680.                                                NeighborCol++;
681.                                            }
682.                                            NeighborRow++;
683.                                        }
684.                                        /*************************************************************************
    *******************************/
685.                                        float sum=CompareNeighborSum(img1_neighbor,img2_neighbor);
686.                                        if(minSum > sum)
687.                                        {
688.                                            minSum=sum;
689.                                            minSumnewRow=newRowInd;
690.                                            minSumnewCol=newColInd;
691.                                            minSumRow=img2_rowInd;
692.                                            minSumCol=img2_colInd;
693.                                            minSumInd=img2_index;
694.                                        }
695.                                        /*if(maxPercent < percent)
```

```
                                        {
                                        maxPercent=percent;
                                        maxPercentnewRow=newRowInd;
                                        maxPercentnewCol=newColInd;
                                        maxPercentnewInd=img2_index;
                                        maxPercentRow=img2_rowInd;
                                        maxPercentCol=img2_colInd;
                                        maxPercentInd=img2_index;
                                        }*/


                                    }
                                }
                            }
                        }
                        /*if(maxever < maxpercent)
                          {
                          maxever=maxPercent;
                          }*/
                        /*if(maxPercent > PercentThreshold)
                          {
                          maxcount++;
                          cout<<"\n index:"<<img1_index<<"/"<<img1_NumPt;
                          cout<<"\tIMG1 "<<img1_rowInd<<",:"<<img1_colInd;
                          cout<<"\tIMG2 "<<maxPercentnewRow<<",:"<<maxPercentnewCol;
                          cout<<"\tIMG2 "<<maxPercentRow<<",:"<<maxPercentCol;
                          cout<<"\tMatch :"<<(100*maxPercent)<<"% max:"<<maxever<<"\tcount :"<<count<<"$$$"<<ma
    xcount<<"\n";
                        //cout<<img1_rowInd<<" "<<img1_colInd<<" "<<maxPercentRow<<" "<<maxPercentCol<<"\n";

                        // outputFile<<maxSumnewRow<<" "<<maxSumnewCol<<" "<<img1_rowInd<<" "<<img1_colInd<<"\n
    ";
                         *    matched.at(maxSumInd)=0;
                         img1.putPixel(img1_rowInd,img1_colInd,255);
                         img2.putPixel(maxSumnewRow,maxSumnewCol,255);
                         }*/
                        if(minSum < PercentThreshold*10000)
                        {
                            maxcount++;

                            cout<<"\n index:"<<img1_index<<"/"<<img1_NumPt;
                            cout<<"\tIMG1 "<<img1_rowInd<<",:"<<img1_colInd;
                            cout<<"\tIMG2 "<<minSumnewRow<<",:"<<minSumnewCol;
                            cout<<"\tIMG2 "<<minSumRow<<",:"<<minSumCol;
                            cout<<"\tMatch :"<<(100*minSum)<<"% max:"<<maxever<<"\tcount :"<<count<<"$$$"<<maxc
    ount<<"\n";
                            //cout<<img1_rowInd<<" "<<img1_colInd<<" "<<minSumRow<<" "<<minSumCol<<"\n";

                            outputFile<<minSumnewRow<<" "<<minSumnewCol<<" "<<img1_rowInd<<" "<<img1_colInd<<"\
    n";
                            matched.at(minSumInd)=0;
                            img1.putPixel(img1_rowInd,img1_colInd,255);
                            img2.putPixel(minSumnewRow,minSumnewCol,255);
                        }
                        //cout << '\a';
                        //X
                        cout<<minSum<<"\n";
                        minSum=FLT_MAX;

                        cout<<".";

```

```
753.                 }
754.                 img1.WritetoDisk((char *)"./../output/output1.ifs");
755.                 img2.WritetoDisk((char *)"./../output/output2.ifs");
756.             }
757.
758.             float CompareNeighborSum(float neighbor1[NeiSiz*2][NeiSiz*2],float neighbor2[NeiSiz*2][NeiSiz*2
    ])
759.             {
760.                 float sum=0;
761.                 for(int row=0 ; row != NeiSiz*2 ; row++)
762.                 {
763.                     for(int col = 0 ; col != NeiSiz*2 ; col++)
764.                     {
765.                         if(neighbor1[row][col] != -1.0 && neighbor2[row][col] != -1.0)
766.                         {
767.                             float val=neighbor1[row][col] - neighbor2[row][col];
768.                             sum+=val*val;
769.                         }
770.
771.                     }
772.                 }
773.                 return sum;
774.
775.
776.             }
777.             float CompareNeighborPercent(float neighbor1[NeiSiz*2][NeiSiz*2],float neighbor2[NeiSiz*2][NeiS
    iz*2])
778.             {
779.                 float totalCount=0;
780.                 float matches=0;
781.                 for(int row=0 ; row != NeiSiz*2 ; row++)
782.                 {
783.                     for(int col = 0 ; col != NeiSiz*2 ; col++)
784.                     {
785.                         if(neighbor1[row][col] != -1.0 && neighbor2[row][col] != -1.0)
786.                         {
787.                             totalCount++;
788.                             if(fabs(neighbor1[row][col] - neighbor2[row][col]) < Diffthreshold)
789.                             {
790.                                 matches++;
791.                             }
792.                         }
793.
794.                     }
795.                 }
796.                 return matches/totalCount;
797.
798.
799.             }
800.             inline void GetNeighbor(IFS2D src_img,float neighbor[NeiSiz*2][NeiSiz*2],int rowInd,int colInd)

801.             {
802.                 int NeighborRow=0;
803.                 float **dataptr=src_img.getDataPtr();
804.                 for(int row=rowInd-NeiSiz;row != rowInd+NeiSiz;row++)
805.                 {
806.                     int NeighborCol=0;
807.                     for(int col=colInd-NeiSiz;col != colInd+NeiSiz;col++)
808.                     {
809.                         if(row >= 0 && row <(int)src_img.getRows() && col >= 0 && col < (int)src_img.getCol
    umns())
```

```
                            {
                                //neighbor[NeighborRow][NeighborCol]=src_img.getPixel(row,col);
                                neighbor[NeighborRow][NeighborCol]=dataptr[row][col];
                            }
                            else
                            {
                                neighbor[NeighborRow][NeighborCol]=-1.0;
                            }
                            // cout<<"\nrowInd :"<<rowInd<<"\tcolInd:"<<colInd<<"\trow:"<<row<<"\tcol:"<<col;
                            // pause(0);
                            NeighborCol++;
                        }
                        NeighborRow++;
                    }
                }


                void FindIntPt(std::vector<long>& Pts,IFS2D src_img,int filecount)
                {
                    IFS2D Scaledsrc (src_img);
                    IFS2D Output ((char *)"float",src_img.getRows(),src_img.getColumns());
                    std::vector<IFS2D> ScaleSpace;
                    std::vector<IFS2D> DoG;

                    int index=0;
                    for(int i=0;i<5 && Scaledsrc.getRows()>25;i++)
                    {
                        ScaleSpace.push_back(IFS2D(Scaledsrc));
                        ScaleSpace[i].Gauss(Sigma[i%5]);
                        if(i%5 != 0)
                        {
                            DoG.push_back(ScaleSpace[i]);
                            DoG[index].subtract(ScaleSpace[i-1]);
                            char *path=(char *)ntos(i,"DoG").c_str();
                            //DoG[index].threshold(DoG[index].getMax());
                            std::cout<<"\nDOG "<<index<<" "<<Sigma[i%5]<<"\t"<<DoG[index].getMax()<<"  "<<DoG[i
    ndex].getRows();
                            DoG[index].WritetoDisk(path);
                            index++;

                        }
                        if(i!=0 && ((i-4)%5) == 0)
                        {
                            Scaledsrc.SubSample();
                        }
                        //char *pathi=(char *)ntos(i,"Scale").c_str();
                        //ScaleSpace[i].WritetoDisk(pathi);
                    }
                    std::vector<int> LocalMinRow;
                    std::vector<int> LocalMinCol;
                    std::vector<int> LocalMinScale;
                    int count=0;
                    for(int scale=1;scale<=2;scale++)
                    {
                        for(int r=1;r<DoG[0].getRows()-2;r++)
                        {
                            for(int c=1;c<DoG[0].getColumns()-2;c++)
                            {
                                float HigherNeighbor[9];
                                float CurrentNeighbor[9];
                                float LowerNeighbor[9];
```

```
                                std::vector<float> VOXEL;
                                // get the neighborhood in the higher scale
                                DoG[scale-1].getNeighborhood(r,c,HigherNeighbor);
                                // get the neighborhood in the current scale
                                DoG[scale].getNeighborhood(r,c,CurrentNeighbor);
                                // get the neighborhood in the lower scale
                                DoG[scale+1].getNeighborhood(r,c,LowerNeighbor);

                                float CurVal=DoG[scale].getPixel(r,c);
                                for(int index=0;index<9;index++)
                                {
                                    VOXEL.push_back(HigherNeighbor[index]);
                                    if(index!=4) VOXEL.push_back(CurrentNeighbor[index]);
                                    VOXEL.push_back(LowerNeighbor[index]);

                                }
                                int minflag=0;
                                int maxflag=0;
                                for(int index=0;index!=(int)VOXEL.size();index++)
                                {
                                    if(VOXEL.at(index)<CurVal)
                                    {
                                        maxflag++;
                                    }
                                    if(VOXEL.at(index)>CurVal)
                                    {
                                        minflag++;
                                    }
                                }
                                if((maxflag==26 || minflag==26 ) && CurVal > threshold)
                                {
                                    //cout<<"*";
                                    Output.putPixel(r,c,Output.getPixel(r,c)+10);
                                    if(Output.getPixel(r,c) == 20)
                                        count++;
                                }
                            }

                        }

                    }
                    char *path=(char *)ntos(filecount,"POI_").c_str();
                    Output.WritetoDisk(path);
                    int pix=0;
                    int cols=Output.getColumns();
                    for(int r=0;r<Output.getRows();r++)
                    {
                        for(int c=0;c<Output.getColumns();c++)
                        {
                            if(Output.getPixel(r,c)>0)
                            {
                                Pts.push_back((cols*r)+c);
                                //cout<<"\n r:"<<r<<" c:"<<c<<" val:"<<(cols*r)+c<<" cols:"<<cols;
                                pix++;
                            }
                        }
                    }
                    //cout<<"####"<<pix<<"####";
                }
```

FindHMatrix.cpp

```
1.  // my own header
2.  #include "../hdr/FindHMatrix.h"
3.
4.
5.  #define offset 100
6.  void printmatrix(float **matrix,int rows ,int col);
7.  void printmatrixd(double **matrix,int rows ,int col);
8.  void getHmatrix(float **Amatrix,float **Dmatrix,float **Hmatrix,int NumPts);
9.  void getDmatrix(float **Dmatrix);
10. void getAmatrix(float **Amatrix);
11. void getCorPoint(float **Xmatrix,float **X_matrix,float **Hmatrix);
12. int getNumPoints(char filename[]);
13. float CheckCorrespondence(float **Xmatrix,float **X_matrix,float **Hmatrix);
14. std::vector<int>  Corpx(10,0);
15. std::vector<int>  Corpx_(10,0);
16. std::vector<int>  Corpy(10,0);
17. std::vector<int>  Corpy_(10,0);
18.
19. void FindHMatrix(char argv[],float **HMatrix)
20. {
21.     int numPts=getNumPoints(argv);
22.     float **Amatrix=matrix(1,20*2,1,8);
23.     float **Dmatrix=matrix(1,20*2,1,1);
24.     float **TempHmatrix=matrix(1,3,1,3);
25.     //float **Hmatrix=matrix(1,3,1,3);
26.     /* Get all points */
27.     std::fstream myfile(argv, std::ios_base::in);
28.     int a;
29.     int read=0;
30.     /* get all the manual correspondences */
31.     std::vector<int>  AllCorpx;
32.     std::vector<int>  AllCorpx_;
33.     std::vector<int>  AllCorpy;
34.     std::vector<int>  AllCorpy_;
35.     while (myfile >> a)
36.     {
37.         switch(read)
38.         {
39.             case 0:
40.                 AllCorpx.push_back(a);
41.                 break;
42.             case 1:
43.                 AllCorpy.push_back(a);
44.                 break;
45.             case 2:
46.                 AllCorpx_.push_back(a);
47.                 break;
48.             case 3:
49.                 AllCorpy_.push_back(a);
50.                 break;
51.             default:
52.                 break;
```

```
        }
        read++;
        read=read%4;
    }
    int iterations=150;
    int maxCount=0;
    while(iterations !=0)
    {
        for(int PtNum=0;PtNum<10;PtNum++)
        {
            int randPt=rand()%numPts;
            Corpx[PtNum]=AllCorpx[randPt];
            Corpy[PtNum]=AllCorpy[randPt];
            Corpx_[PtNum]=AllCorpx_[randPt];
            Corpy_[PtNum]=AllCorpy_[randPt];
            //std::cout<<"\n"<<Corpx[PtNum]<<"\t"<<Corpy[PtNum]<<"\t"<<Corpx_[PtNum]<<"\t"<<Corpy_[PtNum]<<"\n
";
        }
            getAmatrix(Amatrix);
            //printmatrix(Amatrix,20,8);
            getDmatrix(Dmatrix);
            //printmatrix(Dmatrix,20,1);
            getHmatrix(Amatrix,Dmatrix,TempHmatrix,20);
            //printmatrix(TempHmatrix,3,3);
            int count=0;
            for(int index =0;index<Corpx.size();index++)
            {
                float **Xmatrix;
                float **Xdmatrix;
                float **Xddmatrix;
                Xmatrix=matrix(1,3,1,1);
                Xdmatrix=matrix(1,3,1,1);
                Xddmatrix=matrix(1,3,1,1);
                Xmatrix[1][1]=Corpx[index];
                Xmatrix[2][1]=Corpy[index];
                Xmatrix[3][1]=1;
                Xdmatrix[1][1]=Corpx_[index];
                Xdmatrix[2][1]=Corpy_[index];
                Xdmatrix[3][1]=1;
                float val=CheckCorrespondence(Xdmatrix,Xddmatrix,TempHmatrix);
                int xval=abs((int)(Xddmatrix[1][1]-Xmatrix[1][1]));
                int yval=abs((int)(Xddmatrix[2][1]-Xmatrix[2][1]));
                std::cout<<"\n"<<Xmatrix[1][1]<<" "<<Xmatrix[2][1];
                std::cout<<"\n"<<Xdmatrix[1][1]<<" "<<Xdmatrix[2][1];
                std::cout<<"\n"<<Xddmatrix[1][1]<<" "<<Xddmatrix[2][1];
                std::cout<<"\n**************************************";
                std::cout<<"\n"<<val;
                val=xval+yval;
                                if(val < 20)
                                {
                                    count++;
                                }
                            }
                            if(count > maxCount)
                            {
                                HMatrix[1][1]=TempHmatrix[1][1];
                                HMatrix[1][2]=TempHmatrix[1][2];
                                HMatrix[1][3]=TempHmatrix[1][3];
                                HMatrix[2][1]=TempHmatrix[2][1];
                                HMatrix[2][2]=TempHmatrix[2][2];
                                HMatrix[2][3]=TempHmatrix[2][3];
```

```
                            HMatrix[3][1]=TempHmatrix[3][1];
                            HMatrix[3][2]=TempHmatrix[3][2];
                            HMatrix[3][3]=TempHmatrix[3][3];
                            maxCount=count;
                        }
                        std::cout<<"\n"<<count<<"__"<<iterations;
                        iterations--;
                    }
                    printmatrix(HMatrix,3,3);
                }
                float CheckCorrespondence(float **Xmatrix,float **X_matrix,float **Hmatrix)
                {
                    ifsmatmult(Hmatrix,Xmatrix,X_matrix,3,3,3,1);
                    X_matrix[1][1]=X_matrix[1][1]/X_matrix[3][1];
                    X_matrix[2][1]=X_matrix[2][1]/X_matrix[3][1];
                    X_matrix[3][1]=X_matrix[3][1]/X_matrix[3][1];

                    return 0;

                }
                void getAmatrix(float **Amatrix)
                {
                    for(int row_A=0;row_A < (int)Corpx_.size();row_A++)
                    {
                        int xi=Corpx.at((row_A));
                        int yi=Corpy.at((row_A));
                        int xi_=Corpx_.at((row_A));
                        int yi_=Corpy_.at((row_A));
                        int index=row_A*2+1;
                        Amatrix[index][1]=0;
                        Amatrix[index][2]=0;
                        Amatrix[index][3]=0;
                        Amatrix[index][4]=-xi;
                        Amatrix[index][5]=-yi;
                        Amatrix[index][6]=-1;
                        Amatrix[index][7]=xi*yi_;
                        Amatrix[index][8]=yi*yi_;

                        Amatrix[index+1][1]=xi;
                        Amatrix[index+1][2]=yi;
                        Amatrix[index+1][3]=1;
                        Amatrix[index+1][4]=0;
                        Amatrix[index+1][5]=0;
                        Amatrix[index+1][6]=0;
                        Amatrix[index+1][7]=-xi*xi_;
                        Amatrix[index+1][8]=-yi*xi_;
                    }

                }
                void getDmatrix(float **Dmatrix)
                {
                    /* get all the manual correspondences */
                    int index=0;
                    for(int row_A=0;row_A < (int)Corpx_.size();row_A++)
                    {
                        Dmatrix[index+1][1]=Corpx_.at(row_A);
                        Dmatrix[index][1]=Corpy_.at(row_A);
                        index+=2;
                    }

                }
```

```
174.            void getHmatrix(float **Amatrix,float **Dmatrix,float **Hmatrix,int NumPts)
175.            {
176.                float **AmatrixT,**ATxA,**finvATxA,**iATAxAT,**iATAATxD;
177.                double **invATxA;
178.                double **dATxA;
179.                AmatrixT = matrix(1,8,1,NumPts);
180.                ATxA=matrix(1,8,1,8);
181.                dATxA=dmatrix(1,8,1,8);
182.                invATxA=dmatrix(1,8,1,8);
183.                finvATxA=matrix(1,8,1,8);
184.                iATAxAT=matrix(1,8,1,NumPts);
185.                iATAATxD=matrix(1,8,1,1);
186.
187.                transpose(Amatrix,NumPts,8,AmatrixT);
188.                ifsmatmult(AmatrixT,Amatrix,ATxA,8,NumPts,NumPts,8);
189.                for(int r=1;r<=8;r++)
190.                {
191.                    for(int c=1;c<=8;c++)
192.                    {
193.                        dATxA[r][c]=ATxA[r][c];
194.                    }
195.                }
196.                ifsinverse(dATxA,invATxA,8);
197.                for(int r=1;r<=8;r++)
198.                {
199.                    for(int c=1;c<=8;c++)
200.                    {
201.                        finvATxA[r][c]=(float)invATxA[r][c];
202.                    }
203.                }
204.                ifsmatmult(finvATxA,AmatrixT,iATAxAT,8,8,8,NumPts);
205.                ifsmatmult(iATAxAT,Dmatrix,iATAATxD,8,NumPts,NumPts,1);
206.                Hmatrix[1][1]=iATAATxD[1][1];
207.                Hmatrix[1][2]=iATAATxD[2][1];
208.                Hmatrix[1][3]=iATAATxD[3][1];
209.                Hmatrix[2][1]=iATAATxD[4][1];
210.                Hmatrix[2][2]=iATAATxD[5][1];
211.                Hmatrix[2][3]=iATAATxD[6][1];
212.                Hmatrix[3][1]=iATAATxD[7][1];
213.                Hmatrix[3][2]=iATAATxD[8][1];
214.                Hmatrix[3][3]=1;
215.
216.
217.            }
218.            int getNumPoints(char filename[])
219.            {
220.                std::fstream myfile(filename, std::ios_base::in);
221.                int a;
222.                int read=0;
223.                int count=0;
224.                std::vector<int> Corpx;
225.                /* get all the manual correspondences */
226.                while (myfile >> a)
227.                {
228.                    switch(read)
229.                    {
230.                        case 0:
231.                            Corpx.push_back(a);
232.                            break;
233.                        case 1:
234.                            break;
```

```
                        case 2:
                            break;
                        case 3:
                            break;
                        default:
                            break;
                    }
                    read++;
                    read=read%4;
                }
                return Corpx.size();
            }
            void printmatrix(float **matrix,int rows ,int col)
            {
                std::cout<<"\n*****************************************\n";
                for(int r=0;r<rows;r++)
                {
                    std::cout<<r<<"\t";
                    for(int c=0;c<col;c++){
                        std::cout<<matrix[r+1][c+1]<<"\t";
                    }
                    std::cout<<"\n";
                }

            }
            void printmatrixd(double **matrix,int rows ,int col)
            {
                std::cout<<"\n*****************************************\n";
                for(int r=0;r<rows;r++)
                {
                    std::cout<<r<<"\t";
                    for(int c=0;c<col;c++){
                        std::cout<<matrix[r+1][c+1]<<" ";
                    }
                    std::cout<<"\n";
                }

            }
```