Project 4 - Image Stitching Tool using SIFT

Siddharth Roheda and Prathamesh Prabhudesai

April 22, 2016

Contents

1 Introduction

This project focuses on the use of the principles of homography in order to stitch two images. Images considered in this project have at least 40% overlap. Often, we want to click images which do not fit into the entire frame of the camera, and it is not practical to take it from a large distance, since, minor details may be lost. Image stitching can be of a great value in such a case. We use two different approaches to achieve this. The first method involves manual selection of the points of correspondence in the two images, followed by stitching of images using the Homography matrix. In the second method we implement automatic detection of the correspondence points using a modified version of the SIFT descriptor. Again, the Homography matrix is used to stitch the images.

Any two images having the same planar surface can be related by a homography matrix, assuming a pinhole camera. The application of the homography matrix to find corresponding points can be seen in figure 1.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Figure 1: Application of the homography matrix to find corresponding points in image

Here, x and y are the coordinates of a point in the first image that correspond with the point with coordinates x' and y' in the second image. The objective now is to find the homography, H, and map all the pixels in the first image onto the second image, so that the two images get stitched. This has been addressed in the approach section. Images used in this project can be seen in figure 2.

The rest of the report is divided into four sections, namely, the Approach, Results, Extra Work, and Conclusion sections. The Approach section, which is the major section of this report, describes the problem mathematically and explains how to go from two separate overlapping images to forming the homography matrix, followed by, using this matrix to get a stitched image. The next section presents and discusses the results obtained from the simulations which were run using IFS. The section on Extra work talks about the work done by the authors, that might be in addition to what was assigned. Finally, the Conclusion section talks about what was learned from the project and concludes it.

Figure 2: Images of EB II clicked by the authors

2 Approach

2.1 Selecting Correspondences

Before we can actually develop the homography matrix, or form the stitched image, a certain number of correspondences between the two images must be determined. At least 10-12 correspondences should be usually selected in order to get satisfactory results.

2.1.1 Manual Selection of Correspondences

As the first step, we select the corresponding points manually, and concentrate on implementing the homography matrix in order to stitch the image. pixels that are expected to be extracted by a feature detector are selected as correspondences and the coordinates are stored in two separate matrices. The correspondences are usually selected as corners in the images.

2.1.2 Automatic Selection of Correspondences

In order to make selection of correspondences automatic, we find interest points in the two images using a modified version of the Scale Invariant Feature Transform. The first step is to form a scale space using the Difference of Gaussian (DoG) approach. In order to do this, a Gaussian scale space is first formed, and successive frames are subtracted. This leads to a DoG scale space which is a very good approximation of the Laplacian of Gaussian. The DoG approach is illustrated in figure 3. The ratio of scales is selected as $\sqrt{2}$ as in [1].The Difference of Gaussian can be illustrated in form of an equation as follows:

$$D(x, y, \sigma) = f(x, y, k\sigma) - f(x, y, \sigma) \tag{1}$$

SIFT actually uses a pyramid scale space in order to speed up the process, but since images used in this project are rather small, it is not necessary to use the pyramid scale space.

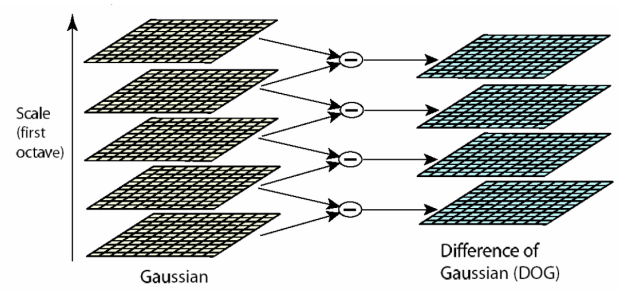

Figure 3: Creating the DoG scale space [3]

Once we have the scale space, the next step is to find the interest points. We can find interest points by looking for local extrema points, in both scale and space, i.e. the triple $[x, y, \sigma]^T$ is an interest point if $D(x, y, \sigma)$ is a local extrema. This is illustrated in figure 4.

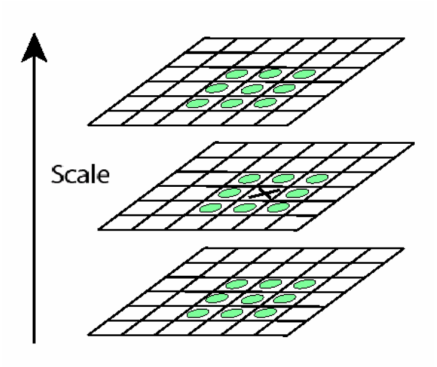

Figure 4: Looking for extrema points in scale and space [3]

Now we have the position of the interest points along with the scale. The next step is to identify the dominant direction in the neighborhood of the interest points. To do this, the gradient is calculated in the neighborhood of the interest point. The scale of the interest point is used to determine the size of the kernel, which was selected as 3σ. A histogram with 16 bins was then created, and the gradient angle of each of the pixels in the neighborhood of the interest point was distributed in these bins. The gradient magnitude of each pixel was used to increment the bins, rather than just increment by 1 [2].The histogram was then smoothed and the peak was found. The bin corresponding to this peak is treated as the dominant direction in the neighborhood.

The development of the descriptor is where our method slightly differs from the SIFT approach and so this is further discussed in Extra Work (section 4.1). The descriptor is then used to match extrema points in the two images, and find correspondences. Euclidean distance was used in this case to match the descriptors.

The matching results in mostly correct correspondences. Since the images are clicked by a human, and are likely to be shifted slightly, or rotated by a small angle, the correspondences may be off by a few rows, but not too many. So, we can put a constraint like, the difference between the row number of the correspondence from image 1 and image 2 should not be more than about 20 or 30. This largely helps to remove incorrect correspondences.

2.2 Developing the Homography Matrix

Now that we have selected the correspondences, the next step is to develop the homography matrix, H. Given 'n' correspondences, a 2x8 matrix (figure 5) is formed for each correspondence and all the matrices are then stacked to form a matrix of size 2nx8, call this matrix A.

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

Figure 5: Matrix relating corresponding points

Create a 2x1 matrix, $D = [-y' \; x']^T$, for every correspondence. These matrices are now stacked together to form a 2nx1 matrix.

Next, we find the 8x1 matrix, h, using equation 2.

$$h = (A^T A)^{-1} A^T D \tag{2}$$

$$h_{8\times 1} = (A^T_{8\times 2n} A_{2n\times 8})^{-1}_{8\times 8} A^T_{8\times 2n} D_{2n\times 1} \tag{3}$$

Equation 3 displays the expected sizes for the matrices in the equation, considering 'n' correspondences were used.

The matrix h is of the form, $h = [h_1 \; h_2 \; h_3 \; h_4 \; h_5 \; h_6 \; h_7 \; h_8]^T$. From this, the Homography matrix, H, can be created as can be seen in figure 6.

$$H = \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & 1.0 \end{bmatrix}$$

Figure 6: Homography matrix, H

2.3 Using the Homography Matrix to Stitch Images

The homography matrix obtained in the previous subsection can be used in order to stitch the two partially overlapping images. If **x** is the vector with coordinates

of a point in the first image, the corresponding point, **x'** in the second image can be found using equation 4.

$$x' = Hx \tag{4}$$

Each point in the first image can now be mapped into the second image, to form the stitched image. It is required to increase the size of the second image, in order to make sure that all the points are mapped inside the dimensions of the image.

3 Results

3.1 Manual Selection of Correspondences and Stitching

Figure 7 shows the manually selected set of correspondences. It makes sense to select corners as correspondences as these are expected to be interest points. Figure 8 shows the result of stitching the two images using the Homography matrix, which is formed using the selected correspondences.

Figure 7: Manually selected corresponding points. Red points indicate correspondences in the first image, and green points indicate correspondences in second image

3.2 Automatic Selection of Correspondences

3.2.1 Scale Space

Figure 9 shows four levels in the Difference of Gaussian scale space. This is a very close approximation to the Laplacian of Gaussian and works much faster. The original image was first blurred at 10 levels of scale, having ratio of $\sqrt{2}$, and then successive frames were subtracted.

Figure 8: Stitching using manually selected correspondences

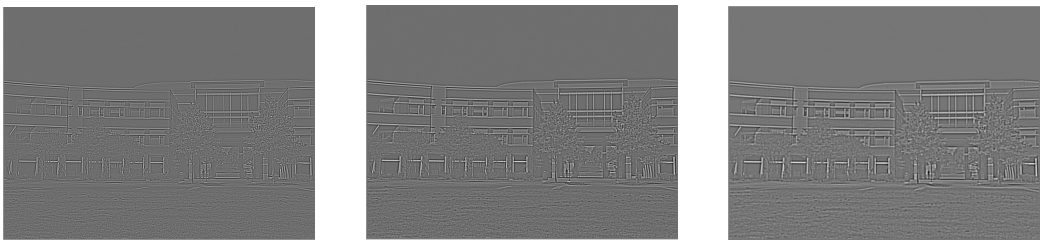

Figure 9: The Difference of Gaussian Scale Space ($\sigma_1 - \sigma_2$)

3.2.2 Selected Correspondences

The local extrema points detected in scale and space can be seen in figure 10. These are the triples $[x, y, \sigma]^T$, where, $D(x, y, \sigma)$ is a local maximum. There are a lot of points detected here. This may be because of a small neighborhood being selected for extrema detection. These points are largely reduced when matching between the descriptors obtained from the points of two images are compared. Each descriptor in the first image is compared with every other descriptor in the second image. After matching the descriptors we end up with a few corresponding points, which can be seen in figure 11.

3.2.3 Stitching of images

In figure 12, the result of stitching using the correspondences found above can be seen. The result is satisfactory, even though the rows are slightly off.

4 Extra Work

4.1 Modified SIFT Descriptor

Once we have the principal direction corresponding to the neighborhood of each interest point, we go ahead and develop a descriptor which is then used to match

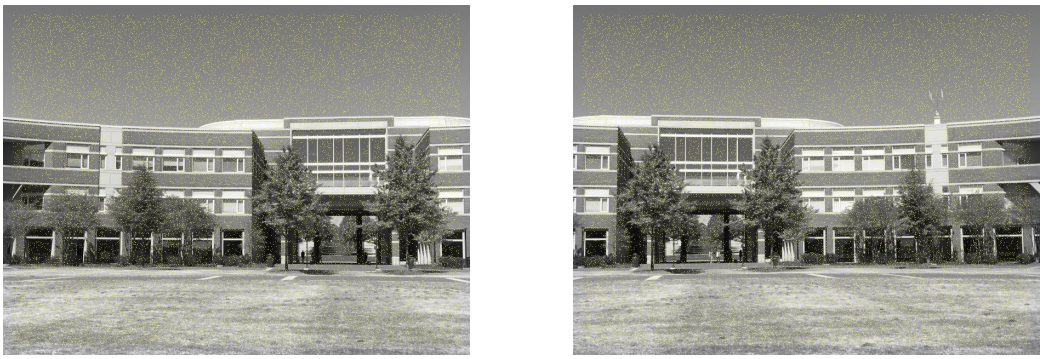

Figure 10: All the extrema points (marked yellow) detected using DoG approach in SIFT

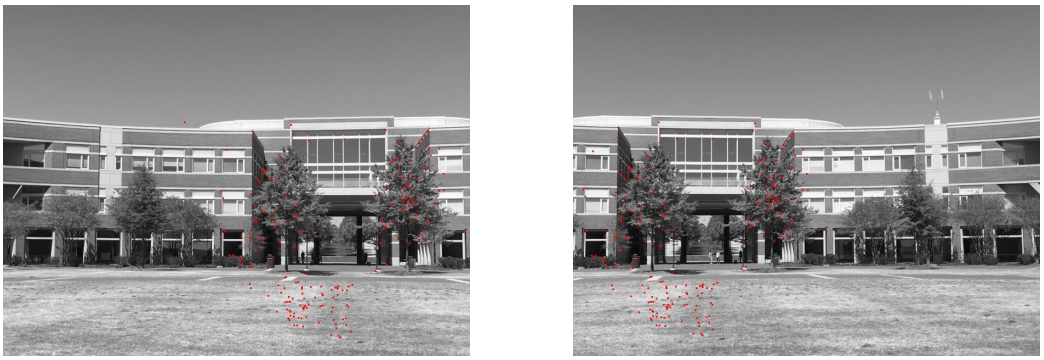

Figure 11: Points matched using the descriptor discussed in Extra Work section

Figure 12: Stitching using SIFT to detect correspondences

the extrema points in the two images.

To do this, we first select a 7x7 neighborhood with the interest point at the center of the neighborhood. All the gradients in the neighborhood are then made relative to the dominant direction by subtracting the dominant direction from each

of them. The resulting gradient values are then formed into a single vector which has 49 elements, instead of the 128 elements in SIFT. This makes the formation of the descriptor much more simpler, faster, and also gives satisfactory results for our application.

This descriptor may not be very robust to large rotations or scaling, but works well for small rotations. The result for one such case is discussed in the next section.

4.2 Stitching Rotated Images (5-10 degrees)

Figure 13 shows the original images, of which the second image is slightly rotated by about 5-7 degrees. In such a case, it is expected that the images can still be matched since the descriptor used is invariant to rotation. The result of implementing the automatic stitching on these images can be seen in Figure 14.

Figure 13: The image on the right is deliberately clicked such that it is rotated by about 5-7 degrees

Figure 14: Result of using the SIFT approach to stitch a slightly rotated image

5 Conclusion

In this project we learned how to stitch multiple images with corresponding points. The implementation here is for two images, and can be easily extended to more than two. We also implemented the concepts of SIFT to find correspondences in the image automatically, and developed a simpler version of the SIFT descriptor which works satisfactorily well for this application. The descriptor was found to work well for small rotations of about 5 to 10 degrees.

6 References

[1] W. Snyder and Q. Hairong, Fundamental Principles of Computer Vision.
[2] D. Lowe, "Object Recognition from Local Scale - Invariant Features."
[3]"Microsoft PowerPoint - cs664-6-features.ppt - cs664-6-features.pdf." [Online]. Available: http://www.cs.cornell.edu/courses/cs664/2008sp/handouts/cs664-6-features.pdf. [Accessed: 21-Apr-2016].

7 Appendix (C++ Implementation using IFS libraries)

Code for correspondence matching using SIFT and stitching

```
#include <iostream>
#include <stdlib.h>
#include <ifs.h>
#include <flip.h>
#include <math.h>

using namespace std;

#define PI 3.1416
#define MAX(X, Y) (((X) > (Y)) ? (X) : (Y))
#define MIN(X, Y) (((X) > (Y)) ? (Y) : (X))
#define ISODD(X) ((X%2) ? (X) : (X+1))

// Function Prototypes

float ** create_kernel(float sigma,int size);

IFSHDR * kernelOperation(IFSHDR * image,int maxrow,int maxcol,
float ** kernel,int kernrows,int kerncols,int scale);

void diff_of_gauss(IFSHDR * img, int maxrow,int maxcol,float sigma1,
float sigma2,IFSHDR * imgdog);
```

```
IFSHDR * zoomout(IFSHDR * image, int maxrow, int maxcol, int scale);

void create_octave(IFSHDR * image,IFSHDR * octave_image_dog1,
IFSHDR * octave_image_dog2,IFSHDR * octave_image_dog3,
IFSHDR * octave_image_dog4,IFSHDR * octave_image_dog5,
IFSHDR * octave_image_dog6,IFSHDR * octave_image_dog7,
IFSHDR * octave_image_dog8,int maxrow,int maxcol);

IFSHDR * find_extrema(IFSHDR * octave_image_dog1,
IFSHDR * octave_image_dog2,IFSHDR * octave_image_dog3,
IFSHDR * octave_image_dog4,IFSHDR * octave_image_dog5,
IFSHDR * octave_image_dog6,IFSHDR * octave_image_dog7,
IFSHDR * octave_image_dog8,int maxrow,int maxcol);

void gradient(IFSHDR * imdx,IFSHDR * imdy,
IFSHDR * g,int maxrow,int maxcol);

void magnitude(IFSHDR * imdx,IFSHDR * imdy,
IFSHDR * m,int maxrow,int maxcol);

void descriptor(IFSHDR * image, IFSHDR * extrema_image,
int maxrow,int maxcol,float ** desc_image);

int histogram(int scale,int index,IFSHDR * m,IFSHDR * g,
float ** desc_image,int maxrow,int maxcol,IFSHDR * extrema_image);

void matching(float ** desc_image1,float ** desc_image2,
int ** correspond1,int ** correspond2,int desc_row1,
int desc_row2, float threshold);

double ** create_matA(double ** extrema_mat1, double ** extrema_mat2,
int num_points);

double ** create_vecD(double ** extrema_mat2, int num_points);

void transpose_mat(double ** A, int rows, int cols,
double ** A_transpose);

void mat_mul(double ** A, double ** B, int r1, int c1, int r2,
int c2, double ** AmulB);

double Determinant(double **a,int n);
```

```
void CoFactor(double **a,int n,double **b);

void Transpose(double **a,int n);

double ** inverse(double ** A, int n);

// main

int main()
{
// Input Images and  Dimensions

IFSIMG image1,image2;
float value;
image1 = ifspin((char *)"small1.ifs");
image2 = ifspin((char *)"small2.ifs");

int * len1, * len2, maxrow1,maxcol1,maxrow2,maxcol2;
len1 = ifssiz(image1); len2 = ifssiz(image2);
maxrow1 = len1[2]; maxcol1 = len1[1]; maxrow2 = len2[2]; maxcol2 = len2[1


IFSIMG color1,color2;
color1 = ifspin((char *)"image1/c1.ifs");
color2 = ifspin((char *)"image1/c2.ifs");

// Image 1 Scale Space

IFSIMG octave1_image1_dog1,octave1_image1_dog2,octave1_image1_dog3,
octave1_image1_dog4,octave1_image1_dog5,octave1_image1_dog6,
octave1_image1_dog7,octave1_image1_dog8;
octave1_image1_dog1 = ifscreate((char *)"float",len1,IFS_CR_ALL,0);
octave1_image1_dog2 = ifscreate((char *)"float",len1,IFS_CR_ALL,0);
octave1_image1_dog3 = ifscreate((char *)"float",len1,IFS_CR_ALL,0);
octave1_image1_dog4 = ifscreate((char *)"float",len1,IFS_CR_ALL,0);
octave1_image1_dog5 = ifscreate((char *)"float",len1,IFS_CR_ALL,0);
octave1_image1_dog6 = ifscreate((char *)"float",len1,IFS_CR_ALL,0);
octave1_image1_dog7 = ifscreate((char *)"float",len1,IFS_CR_ALL,0);
octave1_image1_dog8 = ifscreate((char *)"float",len1,IFS_CR_ALL,0);
create_octave(image1,octave1_image1_dog1,octave1_image1_dog2,
octave1_image1_dog3,octave1_image1_dog4,octave1_image1_dog5,
octave1_image1_dog6,octave1_image1_dog7,octave1_image1_dog8,maxrow1,maxco

// Image 2 Scale Space
```

```
IFSIMG octave1_image2_dog1,octave1_image2_dog2,octave1_image2_dog3,
octave1_image2_dog4,octave1_image2_dog5,octave1_image2_dog6,
octave1_image2_dog7,octave1_image2_dog8;
octave1_image2_dog1 = ifscreate((char *)"float",len2,IFS_CR_ALL,0);
octave1_image2_dog2 = ifscreate((char *)"float",len2,IFS_CR_ALL,0);
octave1_image2_dog3 = ifscreate((char *)"float",len2,IFS_CR_ALL,0);
octave1_image2_dog4 = ifscreate((char *)"float",len2,IFS_CR_ALL,0);
octave1_image2_dog5 = ifscreate((char *)"float",len2,IFS_CR_ALL,0);
octave1_image2_dog6 = ifscreate((char *)"float",len2,IFS_CR_ALL,0);
octave1_image2_dog7 = ifscreate((char *)"float",len2,IFS_CR_ALL,0);
octave1_image2_dog8 = ifscreate((char *)"float",len2,IFS_CR_ALL,0);
create_octave(image2,octave1_image2_dog1,octave1_image2_dog2,
octave1_image2_dog3,octave1_image2_dog4,octave1_image2_dog5,
octave1_image2_dog6,octave1_image2_dog7,
octave1_image2_dog8,maxrow2,maxcol2);

IFSIMG extrema_image1;
extrema_image1 = ifscreate((char *)"float",len1,IFS_CR_ALL,0);
extrema_image1 = find_extrema(octave1_image1_dog1,
octave1_image1_dog2,octave1_image1_dog3,octave1_image1_dog4,
octave1_image1_dog5,octave1_image1_dog6,octave1_image1_dog7,
octave1_image1_dog8,maxrow1,maxcol1);
ifsfree(octave1_image1_dog1,IFS_FR_ALL);
ifsfree(octave1_image1_dog2,IFS_FR_ALL);
ifsfree(octave1_image1_dog3,IFS_FR_ALL);
ifsfree(octave1_image1_dog4,IFS_FR_ALL);
ifsfree(octave1_image1_dog5,IFS_FR_ALL);
ifsfree(octave1_image1_dog6,IFS_FR_ALL);
ifsfree(octave1_image1_dog7,IFS_FR_ALL);
ifsfree(octave1_image1_dog8,IFS_FR_ALL);

IFSIMG extrema_image2;
extrema_image2 = ifscreate((char *)"float",len2,IFS_CR_ALL,0);
extrema_image2 = find_extrema(octave1_image2_dog1,octave1_image2_dog2,
octave1_image2_dog3,octave1_image2_dog4,octave1_image2_dog5,
octave1_image2_dog6,octave1_image2_dog7,octave1_image2_dog8,
maxrow2,maxcol2);
ifsfree(octave1_image2_dog1,IFS_FR_ALL);
ifsfree(octave1_image2_dog2,IFS_FR_ALL);
ifsfree(octave1_image2_dog3,IFS_FR_ALL);
ifsfree(octave1_image2_dog4,IFS_FR_ALL);
ifsfree(octave1_image2_dog5,IFS_FR_ALL);
ifsfree(octave1_image2_dog6,IFS_FR_ALL);
```

```
ifsfree(octave1_image2_dog7,IFS_FR_ALL);
ifsfree(octave1_image2_dog8,IFS_FR_ALL);

cout << "\nNow plotting points....\n";

IFSIMG allpoints1,allpoints2;
int threedlen1[4] = {3,maxcol1,maxrow1,3};
int threedlen2[4] = {3,maxcol2,maxrow2,3};
allpoints1 = ifscreate((char *)"float",threedlen1,IFS_CR_ALL,0);
allpoints2 = ifscreate((char *)"float",threedlen2,IFS_CR_ALL,0);

for(int row = 0; row<maxrow1; row++)
{
for(int col = 0; col<maxcol1; col++)
{
if(ifsfgp(extrema_image1,row,col) > 0)
{
ifsfpp3d(allpoints1,0,row,col,255);
ifsfpp3d(allpoints1,1,row,col,255);
ifsfpp3d(allpoints1,2,row,col,0);
}
else
{
value = ifsfgp(image1,row,col);
ifsfpp3d(allpoints1,0,row,col,value);
ifsfpp3d(allpoints1,1,row,col,value);
ifsfpp3d(allpoints1,2,row,col,value);
}
}
}

for(int row = 0; row<maxrow2; row++)
{
for(int col = 0; col<maxcol2; col++)
{
if(ifsfgp(extrema_image2,row,col) > 0)
{
ifsfpp3d(allpoints2,0,row,col,255);
ifsfpp3d(allpoints2,1,row,col,255);
ifsfpp3d(allpoints2,2,row,col,0);
}
else
{
value = ifsfgp(image2,row,col);
```

```
ifsfpp3d(allpoints2,0,row,col,value);
ifsfpp3d(allpoints2,1,row,col,value);
ifsfpp3d(allpoints2,2,row,col,value);
}
}
}

ifspot(allpoints1,(char *)"sift_image1.ifs");
ifspot(allpoints2,(char *)"sift_image2.ifs");

int desc_row1;
desc_row1 = ifsfgp(extrema_image1,0,0);
float ** desc_image1 = new float *[desc_row1];
for(int i = 0;i<desc_row1;i++) desc_image1[i] = new float[51];

int desc_row2;
desc_row2 = ifsfgp(extrema_image2,0,0);
float ** desc_image2 = new float *[desc_row2];
for(int i = 0;i<desc_row2;i++) desc_image2[i] = new float[51];

descriptor(image1, extrema_image1,maxrow1,maxcol1,desc_image1);
descriptor(image2, extrema_image2,maxrow2,maxcol2,desc_image2);

ifsfree(extrema_image1,IFS_FR_ALL);
ifsfree(extrema_image2,IFS_FR_ALL);

int ** correspond1 = new int *[2];
for(int i=0;i<2;i++) correspond1[i] = new int[desc_row1];

int ** correspond2 = new int *[2];
for(int i=0;i<2;i++) correspond2[i] = new int[desc_row1];

matching(desc_image1,desc_image2,correspond1,correspond2,
desc_row1,desc_row2,500);

int ** faithful_match = new int *[desc_row1];
for(int i=0;i<desc_row1;i++) faithful_match[i] = new int[4];

cout << "\nNow plotting matched points....\n";

IFSIMG faith1,faith2;
faith1 = ifscreate((char *)"float",threedlen1,IFS_CR_ALL,0);
faith2 = ifscreate((char *)"float",threedlen2,IFS_CR_ALL,0);
```

```
for(int row = 0; row<maxrow1; row++)
        {
                for(int col = 0; col<maxcol1; col++)
                {
                                value = ifsfgp(image1,row,col);
                                ifsfpp3d(faith1,0,row,col,value);
                                ifsfpp3d(faith1,1,row,col,value);
                                ifsfpp3d(faith1,2,row,col,value);
                }
        }


for(int row = 0; row<maxrow2; row++)
        {
                for(int col = 0; col<maxcol2; col++)
                {
                                value = ifsfgp(image2,row,col);
                                ifsfpp3d(faith2,0,row,col,value);
                                ifsfpp3d(faith2,1,row,col,value);
                                ifsfpp3d(faith2,2,row,col,value);
                }
        }

int num_points=-1;

for(int i=0;i<desc_row1;i++)
{
        if((abs(correspond1[0][i] - correspond2[0][i]) < 5)
    && (abs(correspond1[1][i] - correspond2[1][i]) > 370)
    && (abs(correspond1[1][i] - correspond2[1][i]) < 380) )
        {
                if(correspond1[1][i] > correspond2[1][i])
                {
      num_points++;
        faithful_match[num_points][0] = correspond1[0][i];
        faithful_match[num_points][1] = correspond1[1][i];
        faithful_match[num_points][2] = correspond2[0][i];
        faithful_match[num_points][3] = correspond2[1][i];
                }
        }
}
int row1,col1,row2,col2;
cout << "\nnumber of points: "<< num_points;
for(int i=0;i<num_points;i++)
```

```
{
        row1 = faithful_match[i][0];
        col1 = faithful_match[i][1];
        row2 = faithful_match[i][2];
        col2 = faithful_match[i][3];
        cout <<"\nPainting...";
        for(int j=-1;j<2;j++)
{
                for(int k=-1;k<2;k++)
{
ifsfpp3d(faith1,0,row1+j,col1+k,255);
ifsfpp3d(faith1,1,row1+j,col1+k,0);
ifsfpp3d(faith1,2,row1+j,col1+k,0);
ifsfpp3d(faith2,0,row2+j,col2+k,255);
ifsfpp3d(faith2,1,row2+j,col2+k,0);
ifsfpp3d(faith2,2,row2+j,col2+k,0);
}
}
}

ifspot(faith1,(char *)"matchedpoints1.ifs");
ifspot(faith2,(char *)"matchedpoints2.ifs");


double ** extrema_mat1 = new double * [2];
for(int i = 0; i<2; i++) extrema_mat1[i] = new double[num_points];

double ** extrema_mat2 = new double * [2];
for(int i = 0; i<2; i++) extrema_mat2[i] = new double[num_points];

for(int i=0;i<num_points;i++)
{
extrema_mat1[0][i] = (double) faithful_match[i][0];
extrema_mat1[1][i] = (double) faithful_match[i][1];
extrema_mat2[0][i] = (double) faithful_match[i][2];
extrema_mat2[1][i] = (double) faithful_match[i][3];
}

double ** D = new double * [2*num_points];
for(int i = 0; i<2*num_points; i++) D[i] = new double[1];

double ** A = new double * [2*num_points];
for(int i = 0; i<2*num_points; i++) A[i] = new double[8];
```

```
double ** AT = new double * [8];
for(int i = 0; i<8; i++) AT[i] = new double [2*num_points];

double ** ATA = new double * [8];
for(int i = 0; i<8; i++) ATA[i]= new double [8];

double ** ATAinv;

double ** ATAinvAT = new double * [8];
for(int i = 0; i<8; i++) ATAinvAT[i]= new double [2*num_points];

double ** h = new double * [8];
for(int i = 0; i<8; i++) h[i]= new double [1];

double ** H = new double * [3];
for(int i = 0; i<3; i++) H[i]= new double [3];

A = create_matA(extrema_mat1, extrema_mat2, num_points);

D = create_vecD(extrema_mat2, num_points);

transpose_mat(A, 2*num_points, 8, AT);

mat_mul(AT,A,8,2*num_points,2*num_points,8,ATA);

ATAinv = inverse(ATA,8);

mat_mul(ATAinv,AT,8,8,8,2*num_points,ATAinvAT);

mat_mul(ATAinvAT,D,8,2*num_points,2*num_points,1,h);

H[0][0] = h[0][0]; H[0][1] = h[1][0]; H[0][2] = h[2][0];
H[1][0] = h[3][0]; H[1][1] = h[4][0]; H[1][2] = h[5][0];
H[2][0] = h[6][0]; H[2][1] = h[7][0]; H[2][2] = 1;

int maxH = 0;
cout<<"\nThe H Matrix: ";
for(int i=0;i<3;i++)
{
        cout<<endl;
        for(int j=0;j<3;j++)
        {
                cout<< H[i][j] << "\t";
                if(maxH < H[i][j])
```

```
                        maxH = H[i][j];
        }
}

int maxcoldiff = 0;
int mincol = 10000;
int correscol;
for(int i=0;i<num_points;i++)
{
        value = faithful_match[i][1] - faithful_match[i][3];
        if(maxcoldiff < value)
        {
                maxcoldiff = value;
        }
        if(mincol > faithful_match[i][1])
        {
                mincol = faithful_match[i][1];
                correscol = faithful_match[i][3];
        }
}

int stitch_row,stitch_col;

stitch_row = MAX(maxrow1,maxrow2);
stitch_col = maxcol1 + maxcoldiff;
int stitchlen[3] = {2,stitch_col,stitch_row};
int colorstitchedlen[4] = {3,stitch_col,stitch_row,3};

IFSIMG stitched_image,colorstitched;
stitched_image = ifscreate((char *)"float",stitchlen,IFS_CR_ALL,0);
colorstitched = ifscreate((char *)"float",colorstitchedlen,IFS_CR_ALL,0);

for (int row = 0; row<maxrow1; row++)
{
        for(int col = 0; col<maxcol1; col++)
        {
        ifsfpp(stitched_image, row, col,ifsfgp(image1, row, col));
        for(int frame=0;frame <3;frame++)
                {
ifsfpp3d(colorstitched,frame,row,col,ifsfgp3d(color1,frame,row,col));
                }
        }
}
```

```
int i = 0;
for (int row = 0; row<maxrow2; row++)
{
        for(int col = mincol-1; col<stitch_col; col++)
        {
                value = ifsfgp(image2,row,i);
                ifsfpp(stitched_image,row,col,value);
                for(int frame=0;frame <3;frame++)
                        {
ifsfpp3d(colorstitched,frame,row,col,ifsfgp3d(color2,frame,row,i));
                        }
                i++;
        }
        i = 0;
}

int p1,p2;

for (int row = 0; row<maxrow2; row++)
{
        for(int col = 0; col<maxcol2; col++)
        {
                p[0][0] = 0; p[1][0] = 0; p[2][0] = 0;
                q[0][0] = row; q[1][0] = col; q[2][0] = 1;

                mat_mul(H,q,3,3,3,1,p);

                p1 = round(p[0][0]);
                p2 = round(p[1][0]);

                cout << "\t--->\tGoing in!!";
ifsfpp(stitched_image,row+p1,stitch_col+p2-1, ifsfgp(image2,row,col));
        }
}

ifspot(stitched_image,(char *) "stitched_image.ifs");
ifspot(colorstitched,(char *) "stitched_image_color.ifs");
cout <<"\nStitched Image saved as stitched_image.ifs\n"<<endl;

return 1;
}

// Function Definitions
```

```
// Create Kernel
float ** create_kernel(float sigma,int size)
{
        float sum = 0;

        float ** kernel = new float*[size];
        for(int i=0;i<size;i++)
                kernel[i] = new float[size];

        for(int i=0;i<size;i++)
        {
                for(int j=0;j<size;j++)
                {
kernel[i][j] = exp(-1*((i - floor(size/2))*(i - floor(size/2))
+ (j - floor(size/2))*(j - floor(size/2)))
/(2*sigma*sigma))/(sqrt(2*PI)*sigma);
                        sum = sum + kernel[i][j];
                }
        }

        for(int i=0;i<size;i++)
        {
                for(int j=0;j<size;j++)
                {
                        kernel[i][j] = kernel[i][j] / sum;
                }
        }

        return kernel;
}

//Operate Kernel
IFSHDR * kernelOperation(IFSHDR * image,
int maxrow,int maxcol,float ** kernel,
int kernrows,int kerncols,int scale)
{
        IFSIMG operatedImage;
        int len[3] = {2,maxcol,maxrow};
        operatedImage = ifscreate((char *)"float",len,IFS_CR_ALL,0);
        int row,col,i,j;
        float value;
        for(row=0;row<maxrow;row++)
                for(col=0;col<maxcol;col++)
                {
```

```
                        value = ifsfgp(image,row,col);
                        ifsfpp(operatedImage,row,col,value);
                }

for(row=(kernrows-1)/2;row<(maxrow-((kernrows-1)/2));row++)
{
for(col=(kerncols-1)/2;col<(maxcol-((kerncols-1)/2));col++)
{
value = 0;
for(i=-(kernrows-1)/2;i<=(kernrows-1)/2;i++)
{
for(j=-(kerncols-1)/2;j<=(kerncols-1)/2;j++)
{
value = value + ifsfgp(image,row+i,col+j) *
kernel[i+(kernrows-1)/2][j+(kerncols-1)/2];
}
}
ifsfpp(operatedImage,row,col,value/scale);
}
}

return operatedImage;
}

void diff_of_gauss(IFSHDR * img, int maxrow,int maxcol,
float sigma1,float sigma2, IFSHDR * imgdog)
{
IFSIMG img1;
IFSIMG img2;
int len[3] = {2,maxcol,maxrow};
float value = 0;

img1 = ifscreate((char *)"float",len,IFS_CR_ALL,0);
img2 = ifscreate((char *)"float",len,IFS_CR_ALL,0);
float ** kernel1;
float ** kernel2;

int kernel_size;
kernel_size = floor(MAX(sigma1,sigma2)*3);

if(kernel_size%2 == 0) kernel_size = kernel_size + 1;
cout << kernel_size << "\t";
kernel1 = create_kernel(sigma1,kernel_size);
kernel2 = create_kernel(sigma2,kernel_size);
```

```
img1 = kernelOperation(img,maxrow,maxcol,
kernel1,kernel_size,kernel_size,1);
img2 = kernelOperation(img,maxrow,maxcol,
kernel2,kernel_size,kernel_size,1);

for(int i=0;i < maxrow;i++)
{
for(int j = 0;j<maxcol;j++)
{
value = ifsfgp(img1,i,j) - ifsfgp(img2,i,j);
ifsfpp(imgdog,i,j,value);
}
}
free(kernel1);
free(kernel2);
ifsfree(img1,IFS_FR_ALL);
ifsfree(img2,IFS_FR_ALL);
}

IFSHDR * zoomout(IFSHDR * image, int maxrow, int maxcol, int scale)
{
IFSIMG scaled;
int len[3] = {2,ceil(maxcol/scale),ceil(maxrow/scale)};
scaled = ifscreate((char *)"float",len,IFS_CR_ALL,0);
float value=0;
int p=0; int q=0;
while((p<ceil(maxrow/scale)) && (q<ceil(maxcol/scale)))
{
        for(int i = 0; i< (maxrow-1); i=i+scale)
                {
                        for(int j = 0;j< (maxcol-1);j=j+scale)
                        {
value = (ifsfgp(image,i,j) + ifsfgp(image,i+1,j) +
ifsfgp(image,i,j+1) + ifsfgp(image,i+1,j+1))*0.25;
                                ifsfpp(scaled,p,q,value);
                                q++;
                        }
                        p++;
                        q = 0;
                }
        }
        return scaled;
}
```

```
void create_octave(IFSHDR * image,IFSHDR * octave_image_dog1,
IFSHDR * octave_image_dog2,IFSHDR * octave_image_dog3,
IFSHDR * octave_image_dog4,IFSHDR * octave_image_dog5,
IFSHDR * octave_image_dog6,IFSHDR * octave_image_dog7,
IFSHDR * octave_image_dog8,int maxrow,int maxcol)
{
        diff_of_gauss(image,maxrow,maxcol,0.707,1,octave_image_dog1);
        diff_of_gauss(image,maxrow,maxcol,1,1.414,octave_image_dog2);
        diff_of_gauss(image,maxrow,maxcol,1.414,2,octave_image_dog3);
        diff_of_gauss(image,maxrow,maxcol,2,2.828,octave_image_dog4);
        diff_of_gauss(image,maxrow,maxcol,2.828,4,octave_image_dog5);
        diff_of_gauss(image,maxrow,maxcol,4,5.656,octave_image_dog6);
        diff_of_gauss(image,maxrow,maxcol,5.656,8,octave_image_dog7);
        diff_of_gauss(image,maxrow,maxcol,8,11.312,octave_image_dog8);
}

IFSHDR * find_extrema(IFSHDR * octave_image_dog1,
IFSHDR * octave_image_dog2,IFSHDR * octave_image_dog3,
IFSHDR * octave_image_dog4,IFSHDR * octave_image_dog5,
IFSHDR * octave_image_dog6,IFSHDR * octave_image_dog7,
IFSHDR * octave_image_dog8,int maxrow,int maxcol)
{
        IFSIMG extremaimg;
        int len[3] = {2,maxcol,maxrow};
        extremaimg = ifscreate((char *)"float",len,IFS_CR_ALL,0);
        int count = 0;
        float extrema = 0;
        float maxima = -100000;
        float minima = 100000;
        float *** combined_mat = new float **[8];
        for(int i=0;i<8;i++)
                combined_mat[i] = new float * [maxrow];
        for(int i =0;i<8;i++)
                for(int j=0;j<maxrow;j++)
                        combined_mat[i][j] = new float [maxcol];

        for(int row=0;row<maxrow;row++)
        {
                for(int col=0;col<maxcol;col++)
                {
combined_mat[0][row][col] = ifsfgp(octave_image_dog1,row,col);
combined_mat[1][row][col] = ifsfgp(octave_image_dog2,row,col);
combined_mat[2][row][col] = ifsfgp(octave_image_dog3,row,col);
```

```
combined_mat[3][row][col] = ifsfgp(octave_image_dog4,row,col);
combined_mat[4][row][col] = ifsfgp(octave_image_dog5,row,col);
combined_mat[5][row][col] = ifsfgp(octave_image_dog6,row,col);
combined_mat[6][row][col] = ifsfgp(octave_image_dog7,row,col);
combined_mat[7][row][col] = ifsfgp(octave_image_dog8,row,col);
                }
        }

for(int scale=1;scale<=6;scale++)
{
for(int row=1;row<maxrow-1;row++)
{
for(int col=1;col<maxcol-1;col++)
{
for(int k=-1;k<2;k++)
{
for(int i=-1;i<2;i++)
{
for(int j=-1;j<2;j++)
{
if((maxima < combined_mat[scale+k][row+i][col+j]) || (minima > combined_m
{
if(maxima < combined_mat[scale+k][row+i][col+j])
maxima = combined_mat[scale+k][row+i][col+j];
if(minima > combined_mat[scale+k][row+i][col+j])
minima = combined_mat[scale+k][row+i][col+j];
extrema = combined_mat[scale+k][row+i][col+j];
}
}
}
}

if(extrema == combined_mat[scale][row][col])
{
ifsfpp(extremaimg,row,col,scale);
count++;
}
extrema = 0;
maxima = -100000;
minima = 100000;
}
}
}
free(combined_mat);
```

```
for(int i=0;i<17;i++) for(int j=0;j<maxcol;j++)
ifsfpp(extremaimg,i,j,0);
for(int i=maxrow-18;i<maxrow;i++)
for(int j=0;j<maxcol;j++)
ifsfpp(extremaimg,i,j,0);
for(int i=0;i<17;i++) for(int j=0;j<maxrow;j++)
ifsfpp(extremaimg,j,i,0);
for(int i=maxcol-18;i<maxcol;i++)
for(int j=0;j<maxrow;j++)
ifsfpp(extremaimg,j,i,0);
count = count - 34*(maxrow+maxcol) + 68;
ifsfpp(extremaimg,0,0,count);

return extremaimg;
}

void descriptor(IFSHDR * image, IFSHDR * extrema_image,
int maxrow,int maxcol,float ** desc_image)
{
int len[3] = {2,maxcol,maxrow};
int kernel_size;
IFSIMG img1,img2,img3,img4,img5,img6;
img1 = ifscreate((char *)"float",len,IFS_CR_ALL,0);
img2 = ifscreate((char *)"float",len,IFS_CR_ALL,0);
img3 = ifscreate((char *)"float",len,IFS_CR_ALL,0);
img4 = ifscreate((char *)"float",len,IFS_CR_ALL,0);
img5 = ifscreate((char *)"float",len,IFS_CR_ALL,0);
img6 = ifscreate((char *)"float",len,IFS_CR_ALL,0);
float ** kernel1;
kernel_size = floor(1*3);
if(kernel_size%2 == 0) kernel_size = kernel_size + 1;
kernel1 = create_kernel(1,kernel_size);
img1 = kernelOperation(image,maxrow,maxcol,
kernel1,kernel_size,kernel_size,1);

float ** kernel2;
kernel_size = floor(1.414*3);
if(kernel_size%2 == 0) kernel_size = kernel_size + 1;
kernel2 = create_kernel(1.414,kernel_size);
img2 = kernelOperation(image,maxrow,maxcol,
kernel2,kernel_size,kernel_size,1);

float ** kernel3;
kernel_size = floor(2*3);
```

```
if(kernel_size%2 == 0) kernel_size = kernel_size + 1;
kernel3 = create_kernel(2,kernel_size);
img3 = kernelOperation(image,maxrow,maxcol,
kernel3,kernel_size,kernel_size,1);

float ** kernel4;
kernel_size = floor(2.828*3);
if(kernel_size%2 == 0) kernel_size = kernel_size + 1;
kernel4 = create_kernel(2.828,kernel_size);
img4 = kernelOperation(image,maxrow,maxcol,kernel4,kernel_size,kernel_siz
float ** kernel5;
kernel_size = floor(4*3);
if(kernel_size%2 == 0) kernel_size = kernel_size + 1;
kernel5 = create_kernel(4,kernel_size);
img5 = kernelOperation(image,maxrow,maxcol,kernel5,kernel_size,kernel_siz

float ** kernel6;
kernel_size = floor(5.656*3);
if(kernel_size%2 == 0) kernel_size = kernel_size + 1;
kernel6 = create_kernel(5.656,kernel_size);
img6 = kernelOperation(image,maxrow,maxcol,
kernel6,kernel_size,kernel_size,1);

free(kernel1); free(kernel2); free(kernel3);
free(kernel4); free(kernel5); free(kernel6);

IFSIMG imdx,imdy,g1,g2,g3,g4,g5,g6,m1,m2,m3,m4,m5,m6;
imdx = ifscreate((char *)"float",len,IFS_CR_ALL,0);
imdy = ifscreate((char *)"float",len,IFS_CR_ALL,0);
g1 = ifscreate((char *)"float",len,IFS_CR_ALL,0);
g2 = ifscreate((char *)"float",len,IFS_CR_ALL,0);
g3 = ifscreate((char *)"float",len,IFS_CR_ALL,0);
g4 = ifscreate((char *)"float",len,IFS_CR_ALL,0);
g5 = ifscreate((char *)"float",len,IFS_CR_ALL,0);
g6 = ifscreate((char *)"float",len,IFS_CR_ALL,0);
m1 = ifscreate((char *)"float",len,IFS_CR_ALL,0);
m2 = ifscreate((char *)"float",len,IFS_CR_ALL,0);
m3 = ifscreate((char *)"float",len,IFS_CR_ALL,0);
m4 = ifscreate((char *)"float",len,IFS_CR_ALL,0);
m5 = ifscreate((char *)"float",len,IFS_CR_ALL,0);
m6 = ifscreate((char *)"float",len,IFS_CR_ALL,0);

        fldx(img1,imdx);
        fldx(img1,imdy);
```

```
        magnitude(imdx,imdy,m1,maxrow,maxcol);
        gradient(imdx,imdy,g1,maxrow,maxcol);

        fldx(img2,imdx);
        fldx(img2,imdy);
        magnitude(imdx,imdy,m2,maxrow,maxcol);
        gradient(imdx,imdy,g2,maxrow,maxcol);

        fldx(img3,imdx);
        fldx(img3,imdy);
        magnitude(imdx,imdy,m3,maxrow,maxcol);
        gradient(imdx,imdy,g3,maxrow,maxcol);

        fldx(img4,imdx);
        fldx(img4,imdy);
        magnitude(imdx,imdy,m4,maxrow,maxcol);
        gradient(imdx,imdy,g4,maxrow,maxcol);

        fldx(img5,imdx);
        fldx(img5,imdy);
        magnitude(imdx,imdy,m5,maxrow,maxcol);
        gradient(imdx,imdy,g5,maxrow,maxcol);

        fldx(img6,imdx);
        fldx(img6,imdy);
        magnitude(imdx,imdy,m6,maxrow,maxcol);
        gradient(imdx,imdy,g6,maxrow,maxcol);

        ifsfree(imdx,IFS_FR_ALL); ifsfree(imdy,IFS_FR_ALL);
        ifsfree(img1,IFS_FR_ALL); ifsfree(img2,IFS_FR_ALL);
    ifsfree(img3,IFS_FR_ALL); ifsfree(img4,IFS_FR_ALL);
    ifsfree(img5,IFS_FR_ALL); ifsfree(img6,IFS_FR_ALL);

int index = 0;
index = histogram(1,index,m1,g1,desc_image,maxrow,maxcol,extrema_image);
index = histogram(2,index,m2,g2,desc_image,maxrow,maxcol,extrema_image);
index = histogram(3,index,m3,g3,desc_image,maxrow,maxcol,extrema_image);
index = histogram(4,index,m4,g4,desc_image,maxrow,maxcol,extrema_image);
index = histogram(5,index,m5,g5,desc_image,maxrow,maxcol,extrema_image);
index = histogram(6,index,m6,g6,desc_image,maxrow,maxcol,extrema_image);

}

void gradient(IFSHDR * imdx,IFSHDR * imdy,
```

```
IFSHDR * g,int maxrow,int maxcol)
{
int angle;

for(int row=0;row<maxrow;row++)
{
for(int col=0;col<maxcol;col++)
{
angle = atan2(ifsfgp(imdy,row,col),ifsfgp(imdx,row,col));
angle = angle*180/PI;
angle = ((angle+360)%360);
angle = round(angle/22.5);
angle = angle*22.5;
ifsfpp(g,row,col,angle);
}
}
}

void magnitude(IFSHDR * imdx,IFSHDR * imdy,
IFSHDR * m,int maxrow,int maxcol)
{
float mag;
for(int row=0;row<maxrow;row++)
{
for(int col=0;col<maxcol;col++)
{
mag = ifsfgp(imdx,row,col)*ifsfgp(imdx,row,col) +
ifsfgp(imdy,row,col)*ifsfgp(imdy,row,col);
mag = sqrt(mag);
ifsfpp(m,row,col,mag);
}
}
}

int histogram(int scale,int index,IFSHDR * m,IFSHDR * g,
float ** desc_image,int maxrow,int maxcol,
IFSHDR * extrema_image)
{
int hist[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
int position;
int principal;

for(int row=0;row<maxrow;row++)
{
```

```
for(int col=0;col<maxcol;col++)
{
if(ifsfgp(extrema_image,row,col) == scale)
{
for(int i=row-3;i<=row+3;i++)
{
for(int j=col-3;j<=col+3;j++)
{
position = (int) (ifsfgp(g,i,j)/22.5);
hist[position] += ifsfgp(m,i,j);
}
}
int maximum = -10000;
for(int i=0;i<16;i++)
{
if(hist[i] > maximum)
{
maximum = hist[i];
position = i;
}
}
principal = position*22.5;
for(int i=0;i<16;i++) hist[i] = 0;
position = 0;
for(int i=row-3;i<=row+3;i++)
{
for(int j=col-3;j<=col+3;j++)
{
desc_image[index][position] = ifsfgp(g,i,j) - principal;
position++;
}
}
desc_image[index][49] = row;
desc_image[index][50] = col;
index++;
}
}
}
return index;
}

void matching(float ** desc_image1,float ** desc_image2,
int ** correspond1,int ** correspond2,int desc_row1,
int desc_row2, float threshold)
```

```
{
float distance;
int position;
int mindistance;

for(int k = 0; k<desc_row1; k++)
{
position = desc_row2 - 1;
mindistance = threshold;
for(int i=0;i<desc_row2; i++)
{
distance = 0;
for(int j = 0; j<49; j++)
{
distance += (desc_image1[k][j] - desc_image2[i][j])*
(desc_image1[k][j] - desc_image2[i][j]);
}
distance = sqrt(distance);
if(distance < mindistance && distance > 0)
{
mindistance = distance;
position = i;
}
}
correspond1[0][k] = desc_image1[k][49];
correspond1[1][k] = desc_image1[k][50];
correspond2[0][k] = desc_image2[position][49];
correspond2[1][k] = desc_image2[position][50];
}
}

double ** create_matA(double ** extrema_mat1,
double ** extrema_mat2, int num_points)
{
int k = 0;
double ** A = new double * [2*num_points];
for(int i = 0; i<2*num_points; i++) A[i] = new double[8];
for(int i = 0; i<2*num_points-1; i = i+2)
{
k = floor(i/2);
A[0+i][0] = 0; A[0+i][1] = 0; A[0+i][2] = 0; A[0+i][5] = -1;
A[1+i][2] = 1; A[1+i][3] = 0; A[1+i][4] = 0; A[1+i][5] = 0;
A[0+i][3] = -1*extrema_mat1[0][k];
A[0+i][4] = -1*extrema_mat1[1][k];
```

```
A[0+i][6] = extrema_mat1[0][k]*extrema_mat2[1][k];
A[0+i][7] = extrema_mat1[1][k]*extrema_mat2[1][k];
A[1+i][0] = extrema_mat1[0][k]; A[1+i][1] = extrema_mat1[1][k];
A[1+i][6] = -1*extrema_mat1[0][k]*extrema_mat2[0][k];
A[1+i][7] = -1*extrema_mat1[1][k]*extrema_mat2[0][k];
}
return A;
}

double ** create_vecD(double ** extrema_mat2, int num_points)
{
        double ** D = new double * [2*num_points];
        for(int i = 0; i<2*num_points; i++) D[i] = new double[1];
        int k =0;

        for(int i = 0; i<2*num_points-1; i = i+2)
        {
                k = floor(i/2);
                D[0+i][0] = -1*extrema_mat2[1][k];
                D[1+i][0] = extrema_mat2[0][k];
        }
        return D;
}

void transpose_mat(double ** A, int rows,
int cols, double ** A_transpose)
{
for (int row = 0; row<rows; row++)
for(int col = 0; col<cols; col++)
A_transpose[col][row] = A[row][col];
}

void mat_mul(double ** A, double ** B,
int r1, int c1, int r2, int c2,
double ** AmulB)
{
for(int i=0;i<r1;i++) for(int j=0;j<c2;j++) AmulB[i][j] = 0;
if (c1 != r2)
cout<<"\n"<<"matrix dimension mismatch"<<endl;
else
for(int i=0; i<r1; ++i)
for(int j=0; j<c2; ++j)
for(int k=0; k<c1; ++k)
AmulB[i][j]+=A[i][k]*B[k][j];
```

```
}

double Determinant(double **a,int n)
{
int i,j,j1,j2;
double det = 0;
double **m = NULL;

if (n < 1) {
   } else if (n == 1) {
      det = a[0][0];
   } else if (n == 2) {
      det = a[0][0] * a[1][1] - a[1][0] * a[0][1];
   } else {
      det = 0;
      for (j1=0;j1<n;j1++) {
         m = new double * [n-1];
         for (i=0;i<n-1;i++)
            m[i] = new double [n-1];
         for (i=1;i<n;i++) {
            j2 = 0;
            for (j=0;j<n;j++) {
               if (j == j1)
                  continue;
               m[i-1][j2] = a[i][j];
               j2++;
            }
         }
         det += pow(-1.0,j1+2.0) * a[0][j1] * Determinant(m,n-1);
         for (i=0;i<n-1;i++)
            free(m[i]);
         free(m);
      }
   }
   return(det);
}

void CoFactor(double **a,int n,double **b)
{
   int i,j,ii,jj,i1,j1;
   double det;
   double **c;

   c = new double * [n-1];
```

```
    for (i=0;i<n-1;i++)
      c[i] = new double [n-1];

    for (j=0;j<n;j++) {
       for (i=0;i<n;i++) {

          // Form the adjoint a_ij
          i1 = 0;
          for (ii=0;ii<n;ii++) {
             if (ii == i)
                continue;
             j1 = 0;
             for (jj=0;jj<n;jj++) {
                if (jj == j)
                   continue;
                c[i1][j1] = a[ii][jj];
                j1++;
             }
             i1++;
          }

          det = Determinant(c,n-1);

          b[i][j] = pow(-1.0,i+j+2.0) * det;
       }
    }
    for (i=0;i<n-1;i++)
                  free(c[i]);
    free(c);
}

void Transpose(double **a,int n)
{
    int i,j;
    double tmp;
    for (i=1;i<n;i++) {
       for (j=0;j<i;j++) {
          tmp = a[i][j];
          a[i][j] = a[j][i];
          a[j][i] = tmp;
       }
    }
}
```

```
double ** inverse(double ** A, int n)
{
        double ** B = new double * [n];
        for(int i = 0; i<n; i++)
        {
                B[i] = new double[n];
        }
        double det;
        det = Determinant(A,n);
        CoFactor(A, n, B);
        Transpose(B,n);

        for(int i = 0; i<n; i++)for(int j = 0; j<n; j++)
        {
                B[i][j] = B[i][j]/det;
        }
        return B;
}
```