ECE 661: Homework 3

Kai Chi, CHAN

1 Outline of the feature selection algorithm

In this homework, Harris corner detector is selected for feature selection. The basic idea of the detector is that we should easily recognize the feature point by looking through a small window. Also, shifting a window in any direction should give a large change in intensity. The algorithm of Harris corner detector used in this homework is shown as follow.

1. Find the x derivative $I_x(x, y)$ and y derivative $I_y(x, y)$ of an input image by Sobel operator

   $I_x(x, y) = I(x-1, y-1) + 2I(x-1, y) + I(x-1, y+1) - I(x+1, y-1) - 2I(x+1, y) - I(x+1, y+1)$

   $I_y(x, y) = I(x-1, y-1) + 2I(x, y-1) + I(x+1, y-1) - I(x-1, y+1) - 2I(x, y+1) - I(x+1, y+1)$,

   where $I(x, y)$ is the image intensity at pixel (x, y)

2. Compute the covariance matrix within a $M_H \times M_H$ window $C = \begin{bmatrix} \sum {I_x}^2 & \sum I_x I_y \\ \sum I_x I_y & \sum {I_y}^2 \end{bmatrix}$

3. Compute the eigenvalues λ_1 and λ_2 of the covariance matrix C by SVD

4. Compute the corner response $R = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2$, where $k = 0.04$ in this homework

5. Find the points with large corner response ($R > threshold$) (non-maximum suppression)

6. Take the points of local maxima of R

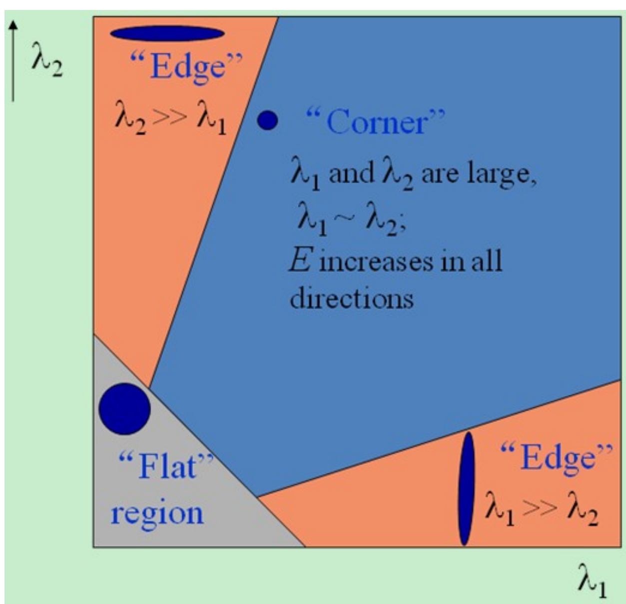


Figure 1: Relationship between eigenvalues and features

Figure 1 shows the relationship between eigenvalues and the corresponding features. R depends only on the eigenvalues of C. R is large for a corner, negative with large magnitude for an edge and $|R|$ is small for a flat region.

To find the corner points, a threshold of the corner response is set for every image. If the response R is larger than the threshold, the corresponding point is regarded as a corner.

Since corners may be very close to each other, non-maximum suppression is carried out. It is used to suppress the corners with weaker response and let the stronger one survive. As a result, only the local maxima of R within a window will be chosen as the features.

2 Description of the feature matching for NCC and SSD

To match a feature, a feature descriptor is created for each feature. In this homework, the feature descriptor is the brightness of each pixel in a window around the point of interest (feature). Formally, the feature descriptor at a feature point $\mathbf{x}$ is $l(\mathbf{x}) = \{I(\tilde{\mathbf{x}}) | \tilde{\mathbf{x}} \in W(\mathbf{x})\}$, where $W(\mathbf{x})$ is a $m \times m$ window around $\mathbf{x}$.

Now, matching two features is done by matching their feature descriptors. Due to noise in images, there will not be an exact match of two feature descriptor. So, we want to minimize the discrepancy measure between their feature descriptors.

To measure the discrepancy, the following discrepancy measures are used.

1. Sum of squared differences (SSD)

$$\frac{1}{m^2} \sum_{i=1}^{m^2} (I_{im1}^i - I_{im2}^i)^2,$$

where I_{im1}^i and I_{im2}^i are the ith pixel intensities within the $m \times m$ window in the first and second images respectively.

2. Normalized cross-correlation (NCC)

$$\frac{\sum_{i=1}^{m^2} (I_{im1}^i - m_{im1})(I_{im2}^i - m_{im2})}{\sqrt{(\sum_{i=1}^{m^2} (I_{im1}^i - m_{im1})^2)(\sum_{i=1}^{m^2} (I_{im2}^i - m_{im2})^2)}},$$

where m_{im1} and m_{im2} are the means of all pixel intensities within the $m \times m$ window in the first and second images respectively.

In SSD method, when two feature descriptors are similar, the SSD will be small. Otherwise, the SSD will be large. For each feature in the first image, exhaustive search is carried out to find the feature in the second image such that the SSD is minimized.

However, given a feature in the first image, the corresponding feature in the second image may not be detected by the Harris corner detector. In this case, the feature in the first image will be rejected if the smallest SSD value is larger than a predefined threshold, th_{ssd}.

Also, some features may not be unique enough. This makes the first and second smallest SSD values close to each other. Clearly, we don't want these features because the probability of getting a wrong match is high. As a result, if $\frac{firstsmallestSSD}{secondsmallestSSD} > ratio_{SSD}$, then the feature will be deleted. The $ratio_{SSD}$ is tuned for each image.

In NCC method, the range of NCC is between -1 and 1. When two feature descriptors are similar, the NCC will be very close to 1. For each feature in the first image, exhaustive search is carried out to find the feature in the second image such that the NCC is maximized.

However, given a feature in the first image, the corresponding feature in the second image may not be detected by the Harris corner detector. In this case, the feature in the first image will be rejected if the largest NCC value is smaller than a predefined threshold, , th_{ncc}.

Also, some features may not be unique enough. This makes the first and second largest NCC values close to each other. As a result, if $\frac{secondlargestNCC}{firstlargestNCC} > ratio_{NCC}$, then the feature will be deleted. The $ratio_{NCC}$ is tuned for each image.

3 Comparison of SSD and NCC on real images

When applying SSD and NCC to real images, the SSD is not invariant to scalings and shifts in image intensities ($I \mapsto \alpha I + \beta$, scaling plus offset), which often occurs in practice between images and is caused by changing lighting conditions over time. However, NCC is invariant to this kind of affine mapping of the intensity values.

Meanwhile, SSD is often preferred when there is small variation in intensity between images because it is a more sensitive measure than NCC and is computationally cheaper.

4 Setting the matching thresholds

In this homework, there are many parameters that need to be tuned. I tuned the parameters such that the number of correct matches is large and the number of mismatch is low. The parameters are summarized in the following tables. The values of the parameters for each image is shown in Table 2.

Table 1: Parameter description

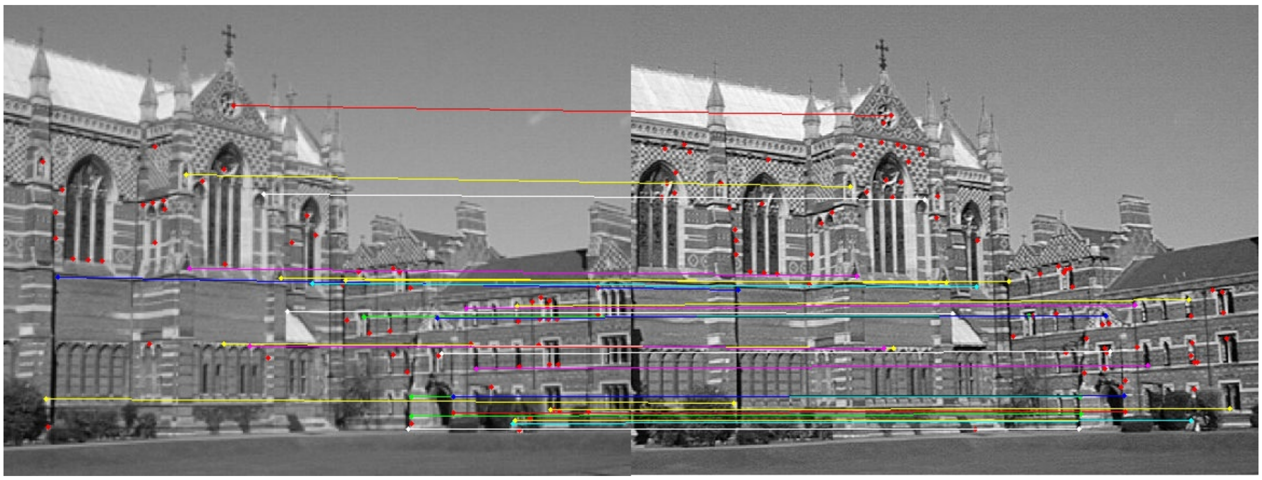
Parameter	Description
M_S	Window size of Sobel operator
M_H	Window size of Harris corner detector
th_R	Threshold of the corner response
W_R	Window size of the local maximum of the corner response
m_{ssd}	Window size of the feature descriptor using SSD
m_{ncc}	Window size of the feature descriptor using NCC
th_{ssd}	Threshold of the largest SSD value considered to be a corner
R_{ssd}	Ratio of $\frac{firstsmallestSSD}{secondsmallestSSD}$
th_{ncc}	Threshold of the smallest NCC value considered to be a corner
R_{ncc}	Ratio of $\frac{secondlargestNCC}{firstlargestNCC}$

Table 2: Parameter setting

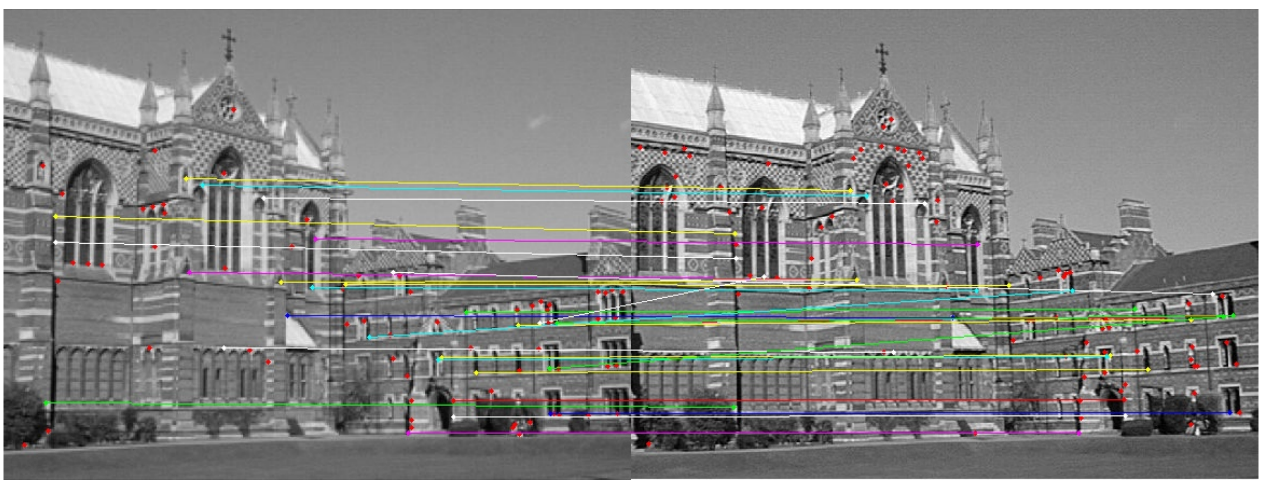
	M_S	M_H	th_R	W_R	m_{ssd}	m_{ncc}	th_{ssd}	R_{ssd}	th_{ncc}	R_{ncc}
sample_a.jpg	3	5	5000000000	5	51	13	1200	0.9	0.77	0.8
sample_b.jpg	3	5	9000000000	5	51	13	1200	0.9	0.77	0.8
table1.jpg	3	7	700000000	7	51	13	500	0.8	0.82	0.8
table2.jpg	3	7	700000000	7	51	13	500	0.8	0.82	0.8
cooker1.jpg	3	7	700000000	7	51	13	500	0.8	0.82	0.8
cooker2.jpg	3	7	700000000	7	51	13	500	0.8	0.82	0.8

5 Matching result

For each image, the number (A/B) in the caption means the (number of correct matches/number of total matches).

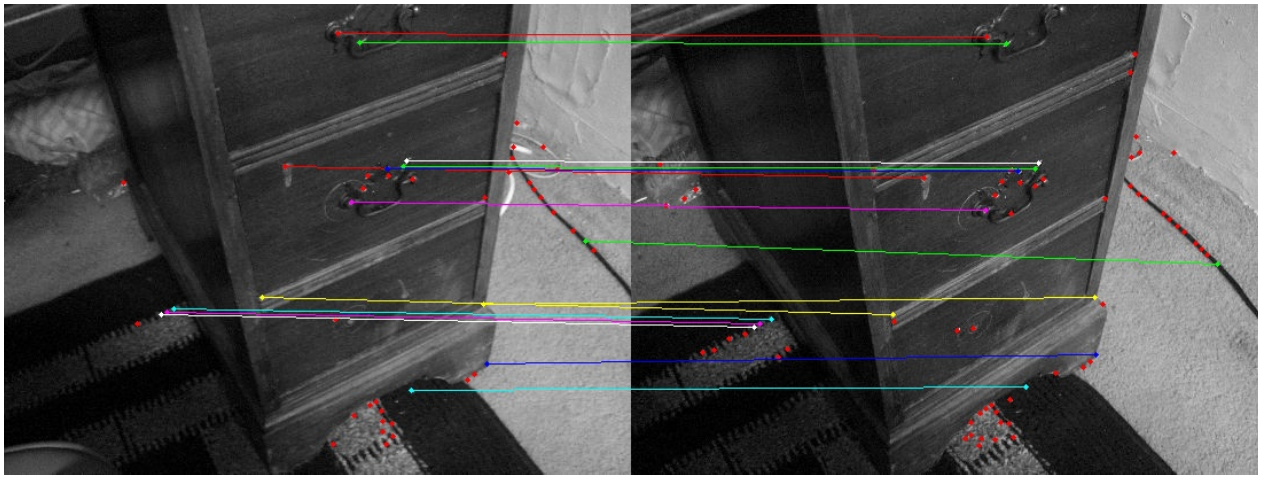


(a) SSD (24/26)

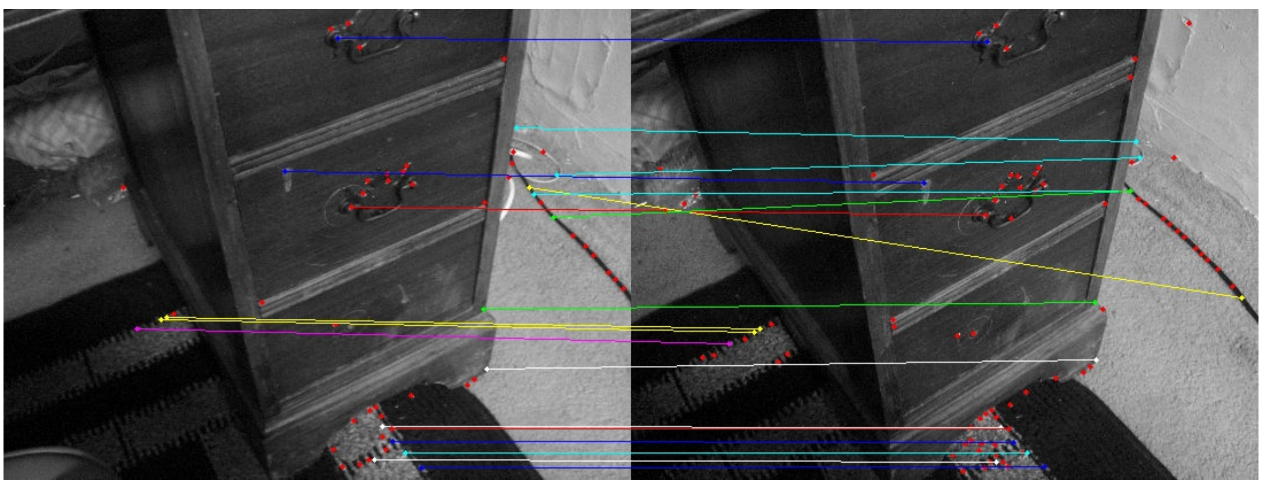


(b) NCC (23/28)

Figure 2: sample_a.jpg, sample_b.jpg

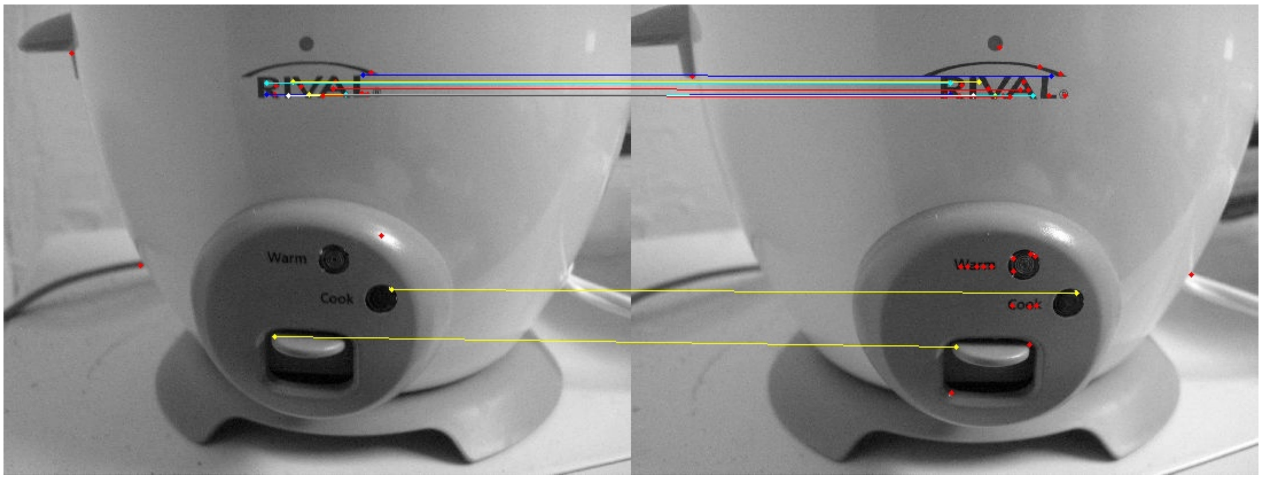


(a) SSD (15/15)

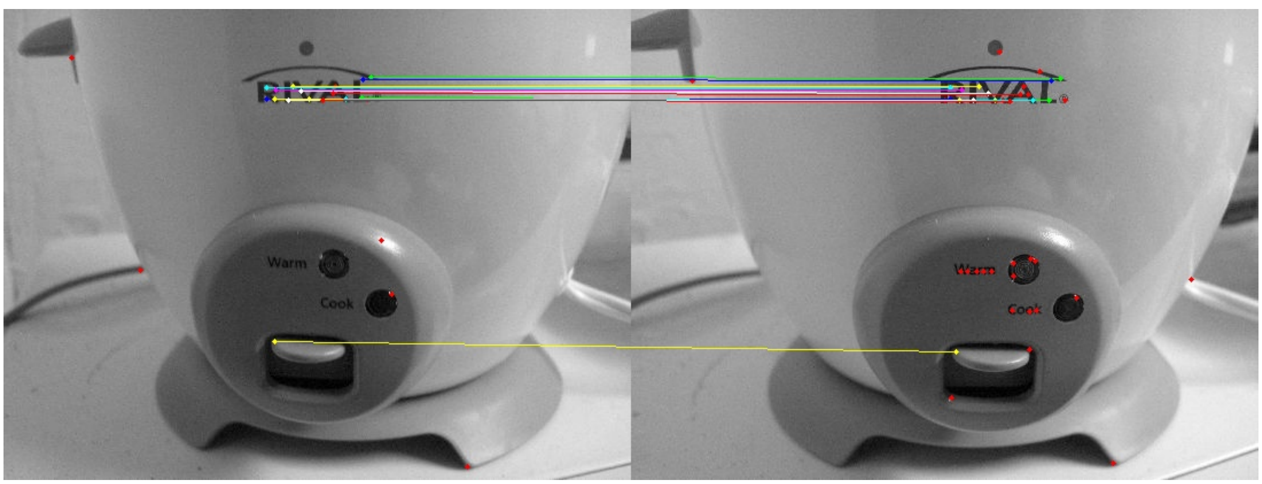


(b) NCC (17/19)

Figure 3: table1.jpg, table2.jpg



(a) SSD (11/11)



(b) NCC (15/15)

Figure 4: cooker1.jpg, cooker2.jpg

6 Source code

```
// EE661_HW3.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <cv.h>
#include <highgui.h>

#define MASK_SIZE 5 //sample-a
//#define MASK_SIZE 7 //others
#define SIZE 13 //ncc
//#define SIZE 51 //ssd
#define MAX 100

CvScalar cs[8] = {cvScalar(0, 0, 255), cvScalar(0, 255, 0), cvScalar(255, 0, 0),
          cvScalar(0, 255, 255), cvScalar(0, 255, 255), cvScalar(255, 255, 0),
                  cvScalar(255, 0, 255), cvScalar(255, 255, 255)};

typedef struct feature_descriptor{
  CvPoint pt[MAX];
  CvMat* m[MAX];
  int no;
  int match[MAX];
  double score[MAX];
} feDe;

CvMat* HarrisCornerDetector(IplImage* img)
{
  IplImage* img_x=0, *img_y=0;
  int height,width,step,channels,depth;
  uchar *data, *data_x, *data_y;

  // get the image data
  height    = img->height;
  width     = img->width;
  depth     = img->depth;
  step      = img->widthStep;
  channels  = img->nChannels;
  data      = (uchar *)img->imageData;
  //printf("Processing a %dx%d image with %d channels\n",height,width,channels);

  img_x = cvCreateImage(cvSize(width, height), depth, channels);
  img_y = cvCreateImage(cvSize(width, height), depth, channels);

  // Get X-derivative
  cvSobel(img, img_x, 1, 0);

  // Get Y-derivative
  cvSobel(img, img_y, 0, 1);

  data_x = (uchar *)img_x->imageData;
  data_y = (uchar *)img_y->imageData;

  CvMat* w = cvCreateMat(MASK_SIZE, MASK_SIZE, CV_64FC1);
  CvMat* h = cvCreateMat(img->height, img->width, CV_64FC3);
  CvMat* M = cvCreateMat(2, 2, CV_64FC1);
  CvMat* M_U = cvCreateMat(2, 2, CV_64FC1);
  CvMat* M_D = cvCreateMat(2, 2, CV_64FC1);
  CvMat* M_V = cvCreateMat(2, 2, CV_64FC1);
  CvMat* r = cvCreateMat(img->height, img->width, CV_64FC1);
  CvMat* r1 = cvCreateMat(img->height, img->width, CV_64FC1);

  cvZero(h);
  cvZero(r);

  for (int i=0; i<w->rows; i++)
    for (int j=0; j<w->cols; j++)
      cvmSet(w, i, j, 1.0/MASK_SIZE);
```

```

  for (int i=MASK_SIZE/2; i<img->height-MASK_SIZE/2; i++)
  {
    for (int j=MASK_SIZE/2; j<img->width-MASK_SIZE/2; j++)
    {
      CvScalar s=cvGet2D(h,i,j);
      for (int w_x=-MASK_SIZE/2; w_x<MASK_SIZE/2+1; w_x++)
      {
        for (int w_y=-MASK_SIZE/2; w_y<MASK_SIZE/2+1; w_y++)
        {
          s.val[0]+=cvmGet(w, w_y+MASK_SIZE/2, w_x+MASK_SIZE/2)*data_x[(i+w_y)*step+(j+w_x)
              ]*data_x[(i+w_y)*step+(j+w_x)];
          s.val[1]+=cvmGet(w, w_y+MASK_SIZE/2, w_x+MASK_SIZE/2)*data_x[(i+w_y)*step+(j+w_x)
              ]*data_y[(i+w_y)*step+(j+w_x)];
          s.val[2]+=cvmGet(w, w_y+MASK_SIZE/2, w_x+MASK_SIZE/2)*data_y[(i+w_y)*step+(j+w_x)
              ]*data_y[(i+w_y)*step+(j+w_x)];
        }
      }
      cvSet2D(h, i, j, s);
      cvmSet(M, 0, 0, s.val[0]);
      cvmSet(M, 0, 1, s.val[1]);
      cvmSet(M, 1, 0, s.val[1]);
      cvmSet(M, 1, 1, s.val[2]);
      cvSVD(M, M_D, M_U, M_V);

      double lambda1 = cvmGet(M_D, 0, 0);
      double lambda2 = cvmGet(M_D, 1, 1);

      cvmSet(r, i, j, lambda1*lambda2 - 0.04*(lambda1+lambda2)*(lambda1+lambda2));
    }
  }

  cvCopy(r,r1);

  for (int i=MASK_SIZE/2; i<img->height-MASK_SIZE/2; i++)
  {
    for (int j=MASK_SIZE/2; j<img->width-MASK_SIZE/2; j++)
    {
      for (int w_x=-MASK_SIZE/2; w_x<MASK_SIZE/2+1; w_x++)
      {
        for (int w_y=-MASK_SIZE/2; w_y<MASK_SIZE/2+1; w_y++)
        {
          if (cvmGet(r, i, j)<cvmGet(r, i+w_y, j+w_x))
          {
            cvmSet(r1, i, j, 0);
          }
        }
      }
    }
  }

  return r1;
}

CvMat* neighbour(IplImage *img, CvPoint x, int size)
{
  CvMat* m = cvCreateMat(size, size, CV_64FC1);

  int step;
  uchar *data;

  // get the image data
  step      = img->widthStep;
  data      = (uchar *)img->imageData;

  for (int i=-size/2; i<=size/2; i++)
  {
    for (int j=-size/2; j<=size/2; j++)
    {
      cvmSet(m, i+size/2, j+size/2, data[(i+x.y)*step+(j+x.x)]);
```

```
    }
  }

  return m;
}

void getFD(IplImage* img,CvMat* r,feDe *fd, long long int thres)
{
  int count=0;
  for (int i=SIZE/2; i<img->height-SIZE/2; i++)
  {
    for (int j=SIZE/2; j<img->width-SIZE/2; j++)
    {
      if (cvmGet(r, i, j)>thres)
      {
        if (count>99)
          break;
        fd->no = count++;
        fd->pt[fd->no].x = j;
        fd->pt[fd->no].y = i;
        fd->m[fd->no] = neighbour(img, cvPoint(j, i), SIZE);
      }
    }
  }
  return;
}

void drawFeatures(IplImage *img, feDe fd)
{
  for (int i=0; i<=fd.no; i++)
  {
    //if (fd.match[i]>0)
      cvCircle(img, fd.pt[i], 0, cvScalar(0, 0, 255), 5);
  }

  return;
}

double ncc(CvMat* I1, CvMat* I2)
{
  double I1_mean = cvMean(I1);
  double I2_mean = cvMean(I2);

  CvMat* I1_mean_matrix = cvCreateMat(SIZE, SIZE, CV_64FC1);
  CvMat* I2_mean_matrix = cvCreateMat(SIZE, SIZE, CV_64FC1);
  CvMat* I12 = cvCreateMat(SIZE, SIZE, CV_64FC1);
  CvMat* I1_sq = cvCreateMat(SIZE, SIZE, CV_64FC1);
  CvMat* I2_sq = cvCreateMat(SIZE, SIZE, CV_64FC1);

  cvAddS(I1, cvScalar(-I1_mean), I1_mean_matrix);
  cvAddS(I2, cvScalar(-I2_mean), I2_mean_matrix);

  cvMul(I1_mean_matrix, I2_mean_matrix, I12);

  cvMul(I1_mean_matrix, I1_mean_matrix, I1_sq);
  cvMul(I2_mean_matrix, I2_mean_matrix, I2_sq);

  return cvSum(I12).val[0]/sqrt(cvSum(I1_sq).val[0]*cvSum(I2_sq).val[0]);
}

double ssd(CvMat* I1, CvMat* I2)
{
  CvMat* I12 = cvCreateMat(SIZE, SIZE, CV_64FC1);
  CvMat* I12_sq = cvCreateMat(SIZE, SIZE, CV_64FC1);

  cvSub(I1, I2, I12);
  cvMul(I12, I12, I12_sq);

  return cvSum(I12_sq).val[0]/(SIZE*SIZE);
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
  IplImage* img = 0, *img1 = 0;
  feDe fd, fd1;

  // load an image
  img=cvLoadImage("sample_a.jpg",0);
  img1=cvLoadImage("sample_b.jpg",0);
  //img=cvLoadImage("img1.jpg",0);
  //img1=cvLoadImage("img2.jpg",0);
  //img=cvLoadImage("table1.jpg",0);
  //img1=cvLoadImage("table2.jpg",0);
  //img=cvLoadImage("cooker1.jpg",0);
  //img1=cvLoadImage("cooker2.jpg",0);


  CvMat* r = cvCreateMat(img->height, img->width, CV_64FC1);
  CvMat* r1 = cvCreateMat(img->height, img->width, CV_64FC1);

  r = HarrisCornerDetector(img);
  r1 = HarrisCornerDetector(img1);

  fd.no = -1;
  fd1.no = -1;
  //sample-a
  getFD(img, r, &fd,    5000000000);
  getFD(img1, r1, &fd1, 9000000000);
  //others
  //getFD(img, r, &fd,     700000000);
  //getFD(img1, r1, &fd1, 700000000);
  printf("%d %d\n", fd.no, fd1.no);

  /*//SSD
  int ssd_thres;
  double ratio;
  ssd_thres = 1200; //sample-a
  ratio = 0.9; //sample-a
  //ssd_thres = 500; //others
  //ratio = 0.8;

  double score, score2;
  for (int i=0; i<=fd.no; i++)
  {
    fd.match[i] = -1;
    score = 10000000000;
    score2 = -100;

    for (int j=0; j<=fd1.no; j++)
    {
      double value = ssd(fd.m[i], fd1.m[j]);
      if (score>value && value<ssd_thres)
      {
        score2 = score;
        fd.match[i] = j;
        score = value;
        fd.score[i] = score;
        fd1.match[j] = i;
        fd1.score[j] = score;
      }
    }
    if (score>0 && score2>0)
      if (score/score2>ratio)
      {
        fd1.match[fd.match[i]] = -1;
        fd1.score[fd.match[i]] = -1;
        fd.match[i] = -1;
        fd.score[i] = -1;
      }
  }*/
```

```
//NCC
double thres = 0.77; //sample -a
double ratio = 0.8;
//double thres = 0.82; //others
//double ratio = 0.8;//others
double score, score2;
for (int i=0; i<=fd.no; i++)
{
  fd.match[i] = -1;
  score = -100;//ncc
  score2 = -100;
  for (int j=0; j<=fd1.no; j++)
  {
    double value = ncc(fd.m[i], fd1.m[j]);
    if (score<value && value >thres)
    {
      score2 = score;
      fd.match[i] = j;
      score = value;
      fd.score[i] = score;
      fd1.match[j] = i;
      fd1.score[j] = score;
    }
  }
  if (score>0 && score2>0)
  {
    if (score2/score>ratio)
    {
      fd1.match[fd.match[i]] = -1;
      fd1.score[fd.match[i]] = -1;
      fd.match[i] = -1;
      fd.score[i] = -1;
    }
  }
}


IplImage *img_h = cvCreateImage(cvSize(img->width*2, img->height), img->depth, 3);
cvSetImageROI( img_h, cvRect(0, 0, img->width, img->height) );
cvCvtColor(img, img_h, CV_GRAY2BGR);
drawFeatures(img_h, fd);
cvResetImageROI(img_h);

cvSetImageROI( img_h, cvRect(img->width, 0, img->width, img->height) );
cvCvtColor(img1, img_h, CV_GRAY2BGR);
drawFeatures(img_h, fd1);
cvResetImageROI(img_h);

// create a window
cvNamedWindow("mainWin");

int count = 0;

for (int i=0; i<=fd.no; i++)
{
  if (fd.match[i]>=0)
  {
    cvCircle(img_h,fd.pt[i], 0, cs[i%8],5);
    cvCircle(img_h,cvPoint(fd1.pt[fd.match[i]].x+img->width, fd1.pt[fd.match[i]].y), 0, cs[i%8],5);
    cvLine(img_h, fd.pt[i], cvPoint(fd1.pt[fd.match[i]].x+img->width, fd1.pt[fd.match[i]].y), cs[i%8]);
    count++;
  }
}

printf("count:%d\n",count);

cvSaveImage("result.jpg",img_h);
```

```
  cvShowImage("mainWin", img_h );

  // wait for a key
  cvWaitKey(0);

  // release the image
  cvReleaseImage(&img );

  return 0;
}
```