ECE661 Computer Vision Homework 2

Image Rectification: Remove Projective and Affine Distortions

Rong Zhang

1 Problem

In this homework, projective and affine distortions are removed from camera images using methods different from HW#1 approach. Two methods are evaluated.

2 Method 1

The first method is a two-step approach, in which the projective distortion is removed first and subsequently the affine distortions.

2.1 Remove Projective Distortion

Since parallel lines remain parallel under affine distortion, we can recover the affine properties from images by the transformation matrix H_1 that maps the vanishing line back into the line at infinity $\vec{l}^{\infty}$.

Given two pairs of image lines $\vec{l}^{(1)}$, $\vec{l}^{(2)}$, $\vec{m}^{(1)}$ and $\vec{m}^{(2)}$, where $\vec{l}^{(1)} \parallel \vec{l}^{(2)}$ and $\vec{m}^{(1)} \parallel \vec{m}^{(2)}$. In a perspective distorted image, these two sets of parallel lines will intersect at points $\vec{p}^{(1)}$ and $\vec{p}^{(2)}$. The line formed by connecting $\vec{p}^{(1)}$ and $\vec{p}^{(2)}$ is the vanishing line $\vec{l} = (l_1, l_2, l_3)^T$. We have in homogeneous coordinates,

$$\begin{aligned} \vec{p}^{(1)} &= \vec{l}^{(1)} \times \vec{l}^{(2)}, \\ \vec{p}^{(2)} &= \vec{m}^{(1)} \times \vec{m}^{(2)}, \\ \vec{l} &= \vec{p}^{(1)} \times \vec{p}^{(2)}. \end{aligned} \tag{1}$$

Therefore, by appling transformation H_1, where

$$H_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ l_1 & l_2 & l_3 \end{pmatrix}, \tag{2}$$

the vanishing line will be mapped into the line at infinity $\vec{l}^{\infty} = (0, 0, 1)^T$. This can be verified as,

$$H_1^{-T} = \begin{pmatrix} 1 & 0 & -l_1/l_3 \\ 0 & 1 & -l_2/l_3 \\ 0 & 0 & 1/l_3 \end{pmatrix}, \tag{3}$$

and $H_1^{-T}\vec{l} = (0, 0, 1)^T$.

In summary, three steps should be applied,

1. choose two sets of image lines which are physically parallel,
2. find the vanishing line $\vec{l} = (l_1, l_2, l_3)^T$ using the above two set of lines,
3. form matrix H_1 and apply H_1 to camera images X_c, i.e., $X_a = H_1 X_c$, where X_a is the affinely rectified image.

2.2 Remove Affine Distortion

Now we get the affinely rectified image X_a, we want to find the affine transform matrix

$$H_2 = \begin{pmatrix} A & \vec{t} \\ \vec{0} & 1 \end{pmatrix}, \tag{4}$$

such that $X_a = H_2 X_s$, where X_s is the scene image in the real world.

Suppose we have a pair of physically orthogonal lines, $\vec{l} \perp \vec{m}$. Let $\vec{l}', \vec{m}'$ be the transformed lines under affine transformation H_2 (eg, $\vec{l}' = H_2^{-T}\vec{l}$), i.e., lines $\vec{l}', \vec{m}'$ are from the affinely rectified image X_a. By orthogonality, we know that

$$(l_1/l_3, l_2/l_3)(m_1/m_3, m_2/m_3)^T = 0, \tag{5}$$

then

$$l_1 m_1 + l_2 m_2 = \vec{l}^T C_{\infty}^* \vec{m} = 0, \tag{6}$$

where $C_{\infty}^* = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$ is the dual degenerate conic. Since $C_{\infty}^{*'} = H_2 C_{\infty}^* H_2^T$, we have

$$\vec{l}^T C_{\infty}^* \vec{m} = \vec{l}'^T H_2 H_2^{-1} C_{\infty}^{*'} H_2^{-T} H_2^T \vec{m}' = \vec{l}'^T C_{\infty}'^* \vec{m}' = 0. \tag{7}$$

Therefore,

$$\begin{aligned}\vec{l}'^T C'^*_\infty \vec{m}' &= \vec{l}'^T H_2 C^*_\infty H_2^T \vec{m} \\ &= \vec{l}'^T \begin{pmatrix} A & \vec{t} \\ \vec{0} & 1 \end{pmatrix} \begin{pmatrix} I & \vec{0} \\ \vec{0} & 1 \end{pmatrix} \begin{pmatrix} A^T & \vec{0} \\ \vec{t}^T & 1 \end{pmatrix} \vec{m}' \\ &= \vec{l}'^T \begin{pmatrix} AA^T & \vec{0} \\ \vec{0} & 0 \end{pmatrix} \vec{m}'.\end{aligned} \tag{8}$$

We have

$$(l'_1, l'_2) AA^T (m'_1, m'_2)^T = 0. \tag{9}$$

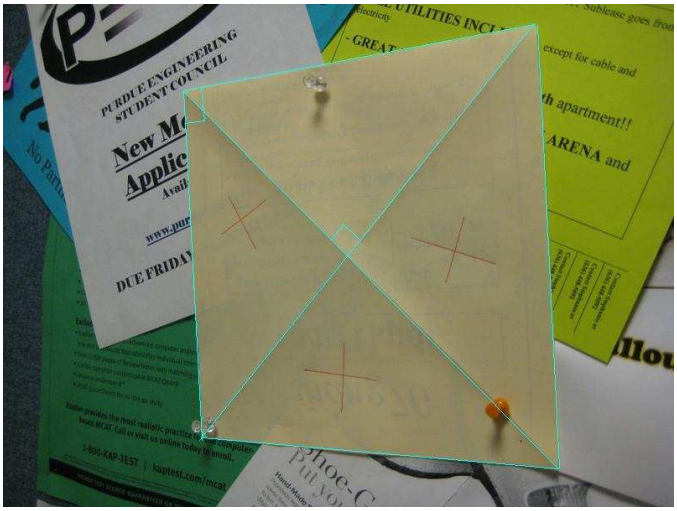
In order to get A, let $S = AA^T$, where $S = \begin{pmatrix} s_{11} & s_{12} \\ s_{12} & 1 \end{pmatrix}$ because S is symmetric. Note that the last element of S is set as 1 considering the scale problem.

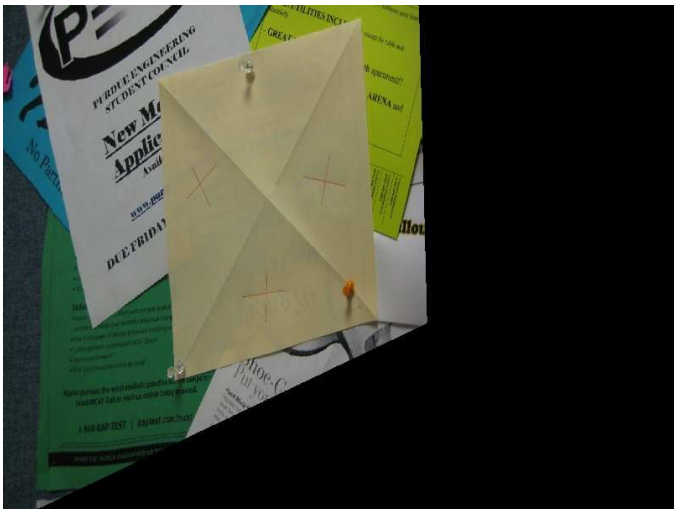
$$(l'_1 m'_1, l'_1 m'_2 + l'_2 m'_1) \begin{pmatrix} s_{11} \\ s_{12} \end{pmatrix} = -l'_2 m'_2 \tag{10}$$

Therefore, a pair of orthogonal lines provides a equation. We need two pairs of orthogonal lines $\vec{l}^{(1)} \perp \vec{m}^{(1)}$ and $\vec{l}^{(2)} \perp \vec{m}^{(2)}$ to solve the matrix S which has two unknown parameters. Note that these two pairs of lines should be non-parallel, otherwise, the two equations will be equivalent.

Since the symmetric matrix S can be written as $S = UDU^T$, where $U^{-1} = U^T$, we can get $A = U\sqrt{D}U^T$. Now, H_2 is available and the restored scene image is $X = H_2^{-1} X_a$.

3 Result





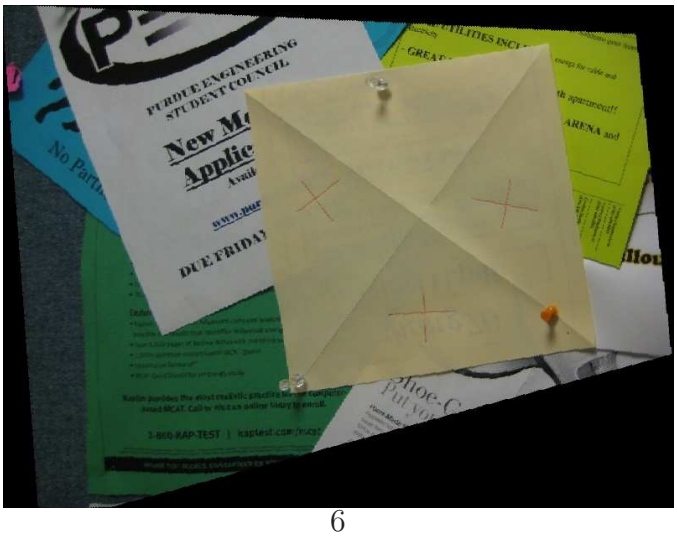


Figure 1: pic1.jpg

Figure 2: pic2.jpg

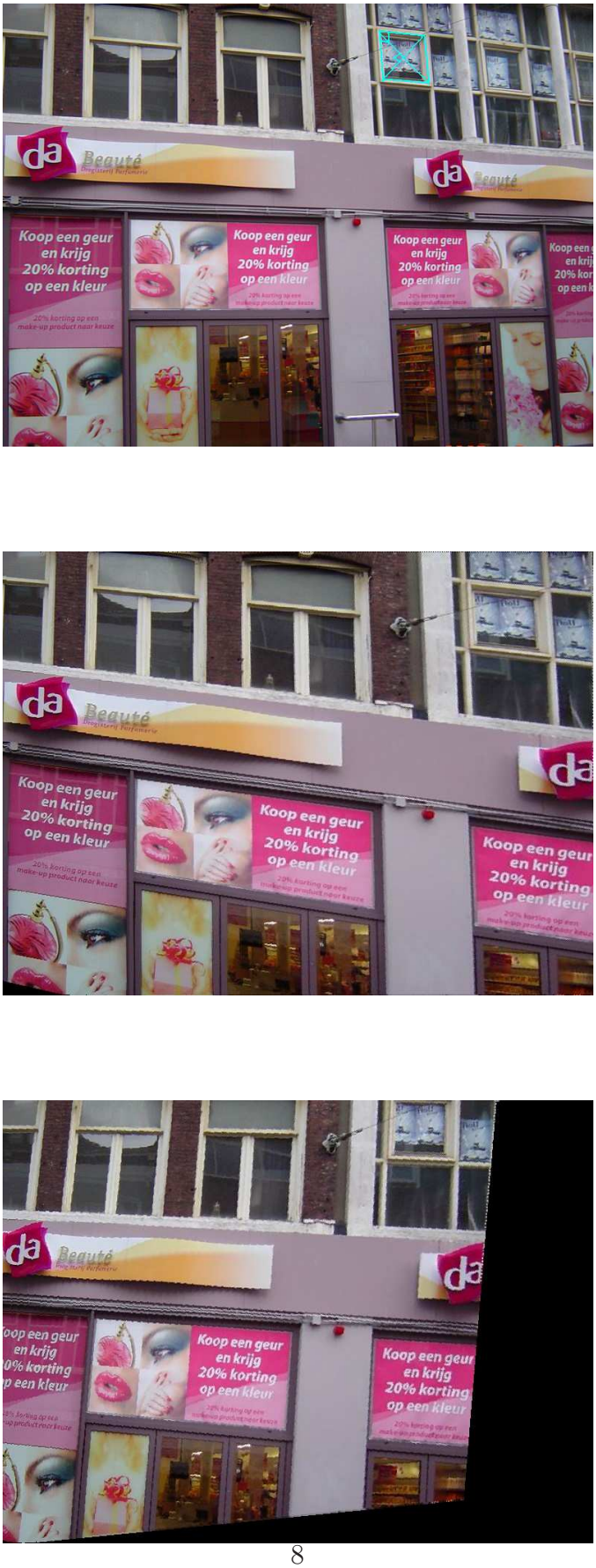
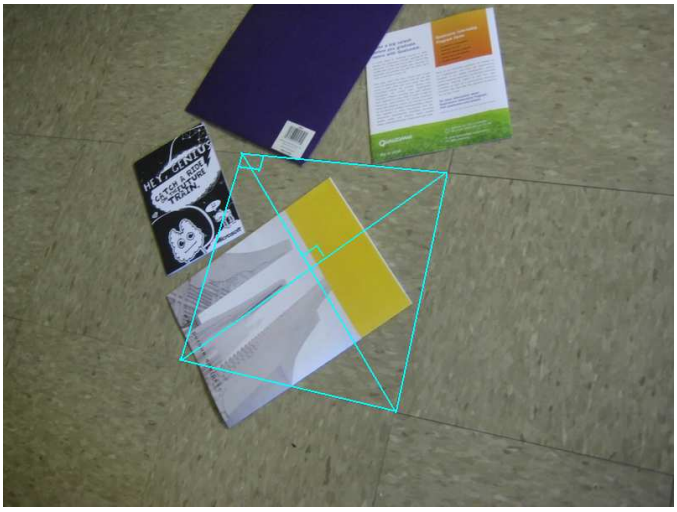


Figure 3: pic3.jpg



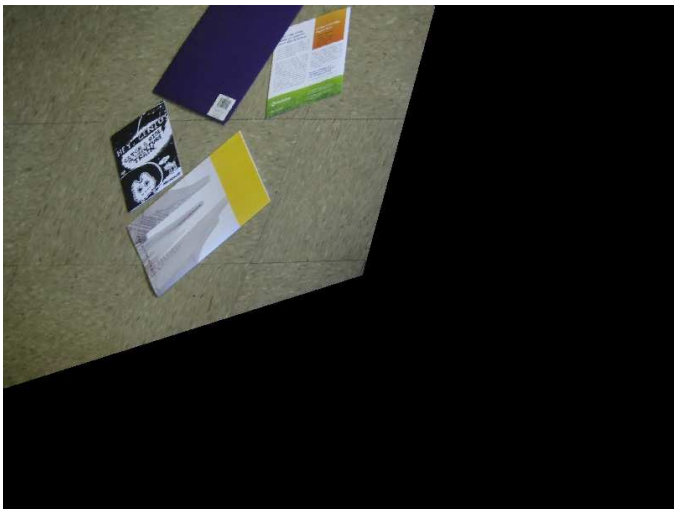


Figure 4: pic4.jpg

Figure 5: sample.jpg

4 Method 2

The second method is the metric rectification using C_∞^*. As stated in the textbook, the dual degenerate conic C_∞^* contains all the information for a metric rectification. It determines both the projective and affine components of a projective transformation H and leaves only similarity distortions.

Suppose we have a pair of physically orthogonal lines, $\vec{l} \perp \vec{m}$. Since

$$C_\infty^* = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{11}$$

and $(l_1, l_2)(m_1, m_2)^T = 0$, we can write

$$\vec{l}^T C_\infty^* \vec{m} = 0. \tag{12}$$

Under projective transformation H, we have $\vec{l'} = H^{-T}\vec{l}$, $\vec{m}' = H^{-T}\vec{m}$ and $C_\infty'^* = HC_\infty^* H^T$. From equation 7, we know that

$$\vec{l'}^T C_\infty'^* \vec{m}' = 0. \tag{13}$$

Suppose $C_\infty'^*$ has the general conic form

$$C_\infty'^* = \begin{pmatrix} a & b/2 & d/2 \\ b/2 & c & e/2 \\ d/2 & e/2 & f \end{pmatrix}, \tag{14}$$

we have

$$\begin{aligned} \vec{l'}^T C_\infty'^* \vec{m}' &= (l_1', l_2', l_3') \begin{pmatrix} a & b/2 & d/2 \\ b/2 & c & e/2 \\ d/2 & e/2 & f \end{pmatrix} \begin{pmatrix} m_1' \\ m_2' \\ m_3' \end{pmatrix} \\ &= \begin{pmatrix} l_1'm_1', \frac{l_1'm_2'+l_2'm_1'}{2}, l_2'm_2', \frac{l_1'm_3'+l_3'm_1'}{2}, \frac{l_2'm_3'+l_3'm_2'}{2}, l_3'm_3' \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \end{pmatrix} = 0. \end{aligned} \tag{15}$$

Considering scale problem, the conic has five degrees of freedom, i.e., we can let $f = 1$. Therefore, five pairs of orthogonal lines will provide five equations. The dual conic in the distorted image $C_\infty'^*$ is determined, which can be further decomposed into

$$C_\infty'^* = UDU^T. \tag{16}$$

Theoretically, the matrix D should have a element equals to zero. However, in practice, $C_\infty'^*$ is usually solved from the five equations by least mean square method for example. It

usually have the last eigenvalue λ_3 not exactly equals to zero but rather small compared to the other eigenvalues. Let

$$D = diag(\sqrt{\lambda_1}, \sqrt{\lambda_2}, 10)diag(1, 1, \lambda_3/100)diag(\sqrt{\lambda_1}, \sqrt{\lambda_2}, 10) \quad (17)$$

$$\approx diag(\sqrt{\lambda_1}, \sqrt{\lambda_2}, 10)C_\infty^* diag(\sqrt{\lambda_1}, \sqrt{\lambda_2}, 10), \quad (18)$$

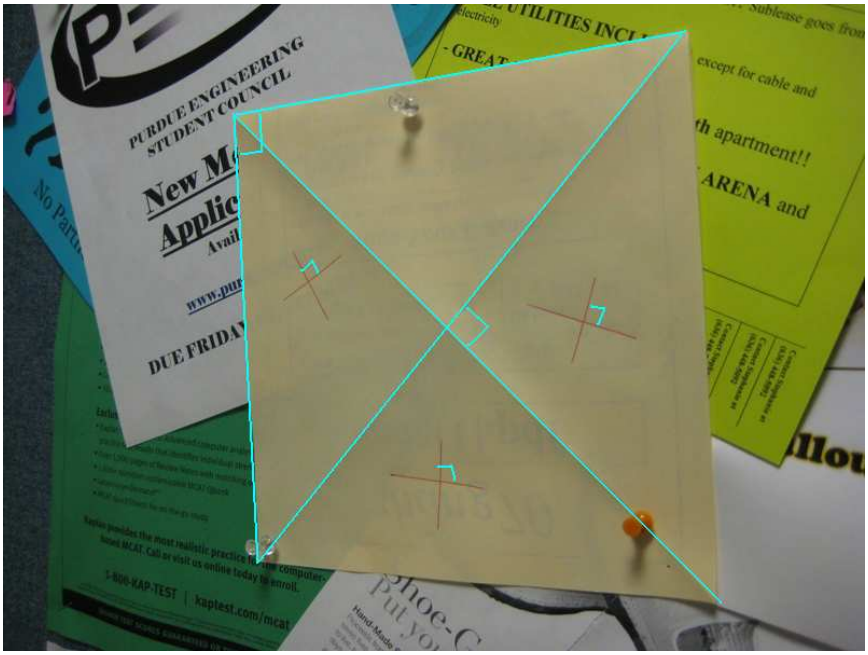
we get $H = UD$. Note that in the above equation, the eigenvalue λ_3 is scaled by a factor $1/100$ which will make $diag(1, 1, \lambda_3/100)$ more close to C_∞^*. By applying $X = H^{-1}X_c$ pointwise, the distorted image is rectified to a similarity distorted one.

5 Results

6 Discussion

The first method uses two steps to rectify the distorted image while the second method is a one-step approach. As we can seen from the results, both methods remove the projective and affine distortions while preserving the similarity. The first method is more robust by separating into two steps, which involves only solving a 2-dimentional function.

The one-step method need to solving equations with 5 degrees of freedom where numerical stability causes problem. For example, the decomposition of $C_\infty'^*$ usually have the last eigenvalue not exactly equals to zero and therefore the rectification results may not be as good as the first method. This method is so sensitive to the choice of orthogonal line pairs that a single change of the selected pixel position value may degrade the reconstruction performance (may also cause flipped images).



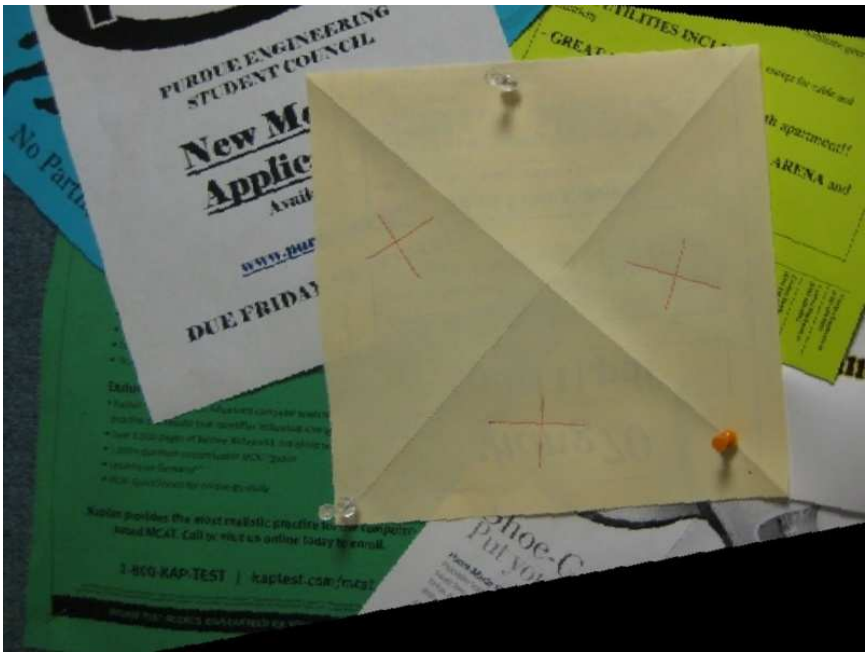


Figure 6: pic1.jpg

Figure 7: pic2.jpg

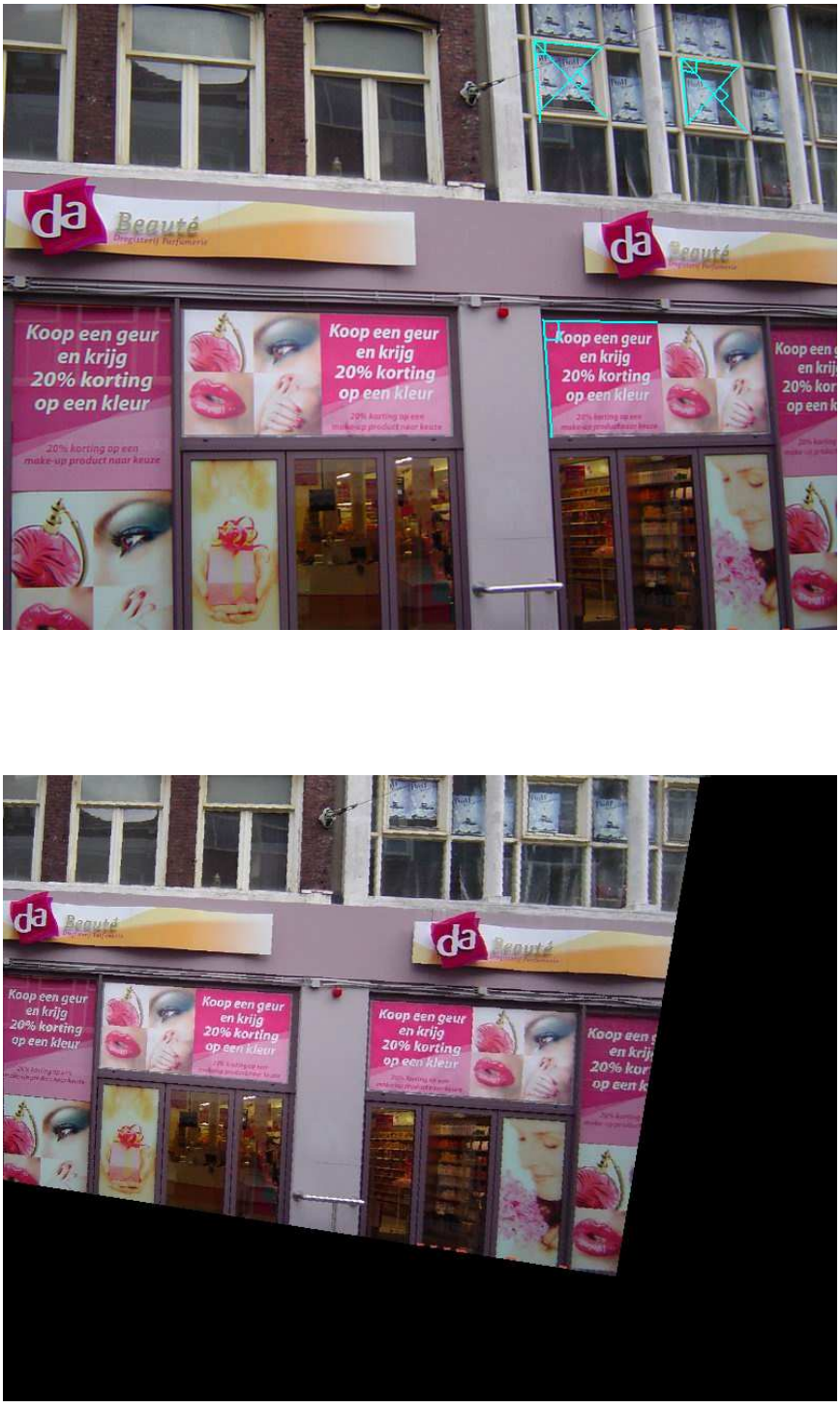


Figure 8: pic3.jpg

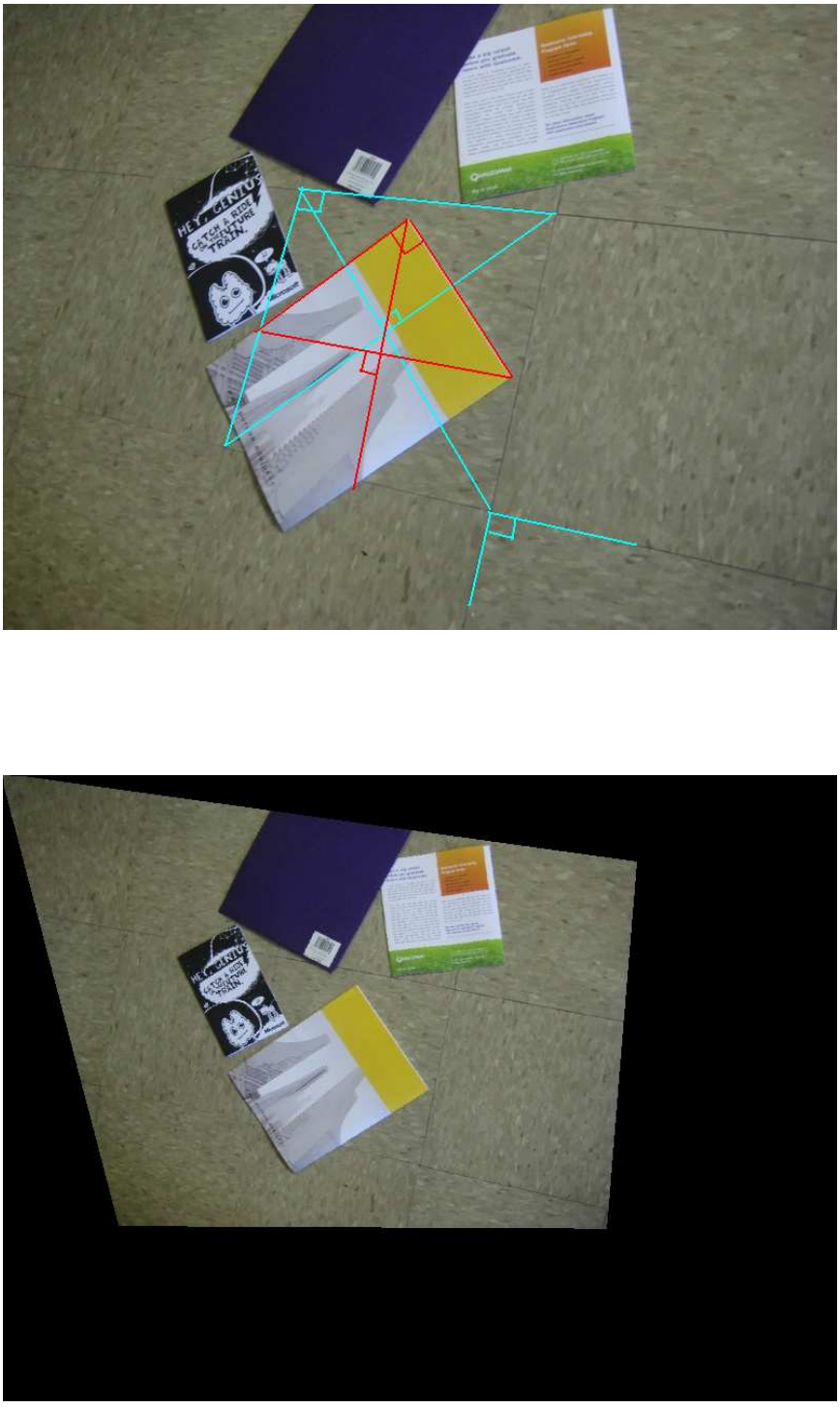


Figure 9: pic4.jpg

Figure 10: sample.jpg

Code for method 1

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <cv.h>
#include <highgui.h>

#define CLIP2(minv, maxv, value) (min(maxv, max(minv, value)))

CvMat* cvmGetCol(CvMat * src, int iCol){
  int i;
  int nRow = src->rows;
  CvMat *curCol = cvCreateMat(nRow,1,src->type);
  for(i=0; i<nRow; i++)
      cvmSet(curCol, i, 0, cvmGet(src, i, iCol));
  return curCol;
}

int main(int argc, char *argv[])
{
  IplImage *img_in = 0, *img_affine = 0, *img_scene = 0, *img_interp = 0;;
  int height, width, step, channels;
  uchar *data_in, *data_affine, *data_scene, *data_interp;
  int i,j,k;
  double distp_x, distp_y;
  int curpi, curpj;

  if(argc<2){
    printf("Usage: main <image-file-name>\n\7");
    exit(0);
  }

  // X_c is the camera image, X_a is the affinely rectified image, X is the true
image
  // X_a = H_1*X_c  X_a = H_2*X
  // load the distorted image X_c
  img_in = cvLoadImage(argv[1]);
  if(!img_in){
    printf("Could not load image file: %s\n",argv[1]);
    exit(0);
  }
  height    = img_in->height;
  width     = img_in->width;
  step      = img_in->widthStep;
  channels  = img_in->nChannels;
  data_in   = (uchar *)img_in->imageData;
  printf("Processing a %d x %d image with %d channels\n",height,width,channels);
  CvMat *check = cvCreateMat(height, width, CV_64FC1);
  cvZero(check);

  // allocate the output image and initialize
  img_affine  = cvCreateImage(cvSize(width, height), IPL_DEPTH_8U, channels);
  img_scene     = cvCreateImage(cvSize(width, height), IPL_DEPTH_8U, channels);
  cvZero(img_affine);
  cvZero(img_scene);
  data_affine = (uchar *)img_affine->imageData;
```

```
  data_scene  = (uchar *)img_scene->imageData;

  // remove projective distortion
  CvMat * vetex = cvCreateMat(3,4,CV_64FC1);
  // get the coordinate values of four vetexes of a square
  // up-left, up-right, down-left, down-right
  for(i=0; i<4; i++)
  {
        printf("input the %d th point's coordinates (x',y') ", i);
        scanf("%f %f", &distp_x, &distp_y);
        distp_x = tmpxp[i];
        distp_y = tmpyp[i];
        cvmSet(vetex,0,i,distp_x);
        cvmSet(vetex,1,i,distp_y);
        cvmSet(vetex,2,i,1);
        printf("\n");
  }

  //************** Affine Rectification *****************
  CvMat *l1 = cvCreateMat(3,1,CV_64FC1);
  CvMat *l2 = cvCreateMat(3,1,CV_64FC1);
  CvMat *m1 = cvCreateMat(3,1,CV_64FC1);
  CvMat *m2 = cvCreateMat(3,1,CV_64FC1);

  CvMat *v1 = cvCreateMat(3,1,CV_64FC1);
  CvMat *v2 = cvCreateMat(3,1,CV_64FC1);

  CvMat *vanishL = cvCreateMat(3,1,CV_64FC1);

  cvCrossProduct(cvmGetCol(vetex,0), cvmGetCol(vetex,1), l1);
  cvCrossProduct(cvmGetCol(vetex,2), cvmGetCol(vetex,3), l2);
  cvCrossProduct(cvmGetCol(vetex,0), cvmGetCol(vetex,2), m1);
  cvCrossProduct(cvmGetCol(vetex,1), cvmGetCol(vetex,3), m2);

  cvCrossProduct(l1, l2, v1);
  cvCrossProduct(m1, m2, v2);

  cvCrossProduct(v1, v2, vanishL);

  // normalize vanishing line
  // in order to map the distorted image back to the image window
  double scale = 1.0;
  cvmSet(vanishL,0,0,cvmGet(vanishL,0,0)/cvmGet(vanishL,2,0)*scale);
  cvmSet(vanishL,1,0,cvmGet(vanishL,1,0)/cvmGet(vanishL,2,0)*scale);
  cvmSet(vanishL,2,0,1.0*scale);

  double H1data[] =
{1,0,0,0,1,0,cvmGet(vanishL,0,0),cvmGet(vanishL,1,0),cvmGet(vanishL,2,0)};
  CvMat H1 = cvMat(3,3,CV_64FC1, H1data);

  // transform X_a = H_1*X_c
  CvMat * ptx  = cvCreateMat(3,1,CV_64FC1);
  CvMat * ptxp = cvCreateMat(3,1,CV_64FC1);

  cvmSet(&H1,2,2,1.0);
  for (i=0; i<height; i++){      //y - ver
        for (j=0; j<width; j++){   //x - hor
              // set X_c
```

```
                cvmSet(ptxp,0,0,(double)j);
                cvmSet(ptxp,1,0,(double)i);
                cvmSet(ptxp,2,0,1.0);
                // compute X_a
                cvMatMul(&H1, ptxp, ptx);
               curpi = CLIP2(0, height-1, (int)(cvmGet(ptx,1,0)/cvmGet(ptx,2,0)));
                curpj = CLIP2(0, width-1, (int)(cvmGet(ptx,0,0)/cvmGet(ptx,2,0)));

                cvSet2D(img_affine,curpi,curpj,cvGet2D(img_in,i,j));
                cvmSet(check,curpi,curpj,1);
          }
    }

    // output reconstructed affine image
    img_interp  = cvCloneImage(img_affine);
    data_interp = (uchar *)img_interp->imageData;
    //interpolation
    double count;
    for (i=1; i<height-1; i++){      //y - ver
          for (j=1; j<width-1; j++){   //x - hor
                if(cvmGet(check,i,j) == 0){
                      count = (cvmGet(check,i-1,j)==1)+(cvmGet(check,i+1,j)==1)+
                               (cvmGet(check,i,j-1)==1)+(cvmGet(check,i,j+1)==1)+
                                 (cvmGet(check,i-1,j-1)==1)+(cvmGet(check,i-
1,j+1)==1)+
                                 (cvmGet(check,i+1,j-
1)==1)+(cvmGet(check,i+1,j+1)==1);
                      if(count != 0){
                            for (k=0; k<channels; k++)
                                 data_interp[i*step+j*channels+k] = (int)
                                 ((data_affine[(i-
1)*step+j*channels+k]+data_affine[(i+1)*step+j*channels+k]
                                +data_affine[i*step+(j-
1)*channels+k]+data_affine[i*step+(j+1)*channels+k]
                                 +data_affine[(i-1)*step+(j-
1)*channels+k]+data_affine[(i-1)*step+(j+1)*channels+k]
                                 +data_affine[(i+1)*step+(j-
1)*channels+k]+data_affine[(i+1)*step+(j+1)*channels+k])/count);

                      }
                }
          }
    }

    img_affine  = cvCloneImage(img_interp);
    if(!cvSaveImage("affine.jpg",img_affine))
          printf("Could not save file\n");

    //************** Metric Rectification *****************
    // transform points by H1
    CvMat *pt = cvCreateMat(3,1,CV_64FC1);
    for(i=0; i<4; i++)
    {
          cvMatMul(&H1, cvmGetCol(vetex,i), pt);
          cvmSet(vetex,0,i,(int)(cvmGet(pt,0,0)/cvmGet(pt,2,0)));
          cvmSet(vetex,1,i,(int)(cvmGet(pt,1,0)/cvmGet(pt,2,0)));
          cvmSet(vetex,2,i,1.0);
    }
```

```
  cvCrossProduct(cvmGetCol(vetex,0), cvmGetCol(vetex,1), l1);
  cvCrossProduct(cvmGetCol(vetex,0), cvmGetCol(vetex,2), m1);
  cvCrossProduct(cvmGetCol(vetex,0), cvmGetCol(vetex,3), l2);
  cvCrossProduct(cvmGetCol(vetex,2), cvmGetCol(vetex,1), m2);

  double l11, l12, m11, m12, l21, l22, m21, m22;
  l11 = cvmGet(l1,0,0); l12 = cvmGet(l1,1,0);
  l21 = cvmGet(l2,0,0); l22 = cvmGet(l2,1,0);
  m11 = cvmGet(m1,0,0); m12 = cvmGet(m1,1,0);
  m21 = cvmGet(m2,0,0); m22 = cvmGet(m2,1,0);

  // M*x = b
  double Mdata[] = {l11*m11, l11*m12+l12*m11, l21*m21, l21*m22+l22*m21};
  double bdata[] = {-l12*m12, -l22*m22};
  CvMat M = cvMat(2,2,CV_64FC1,Mdata);
  CvMat b = cvMat(2,1,CV_64FC1,bdata);
  CvMat *x = cvCreateMat(2,1,CV_64FC1);
  cvSolve(&M,&b,x);

  // Set matrix S
  double Sdata[] = {cvmGet(x,0,0), cvmGet(x,1,0), cvmGet(x,1,0), 1.0};
  CvMat S = cvMat(2,2,CV_64FC1, Sdata);
  // SVD S=UDV_T
  CvMat* U  = cvCreateMat(2,2,CV_64FC1);
  CvMat* D  = cvCreateMat(2,2,CV_64FC1);
  CvMat* V  = cvCreateMat(2,2,CV_64FC1);
  cvSVD(&S, D, U, V, CV_SVD_U_T|CV_SVD_V_T);
  //The flags cause U and V to be returned transposed (does not work well
without the transpose flags).
  //Therefore, in OpenCV, S = U^T D V
  CvMat* U_T   = cvCreateMat(2,2,CV_64FC1);
  CvMat* sqrtD = cvCreateMat(2,2,CV_64FC1);
  CvMat* A     = cvCreateMat(2,2,CV_64FC1);
  cvPow(D, sqrtD, 0.5);
  cvTranspose(U, U_T);
  cvMatMul(U_T, sqrtD, A);
  cvMatMul(A, V, A);

  // Set H2
  double t[] = {0, 0};
  double H2data[] = {cvmGet(A,0,0),cvmGet(A,0,1),t[0],
cvmGet(A,1,0),cvmGet(A,1,1),t[1], 0,0,1};
  CvMat H2 = cvMat(3,3,CV_64FC1, H2data);
  CvMat *invH2 = cvCreateMat(3,3,CV_64FC1);
  cvInvert(&H2, invH2);

  cvZero(check);
  for (i=0; i<height; i++){      //y - ver
        for (j=0; j<width; j++){   //x - hor
              // set X_a
              cvmSet(ptxp,0,0,(double)j);
              cvmSet(ptxp,1,0,(double)i);
              cvmSet(ptxp,2,0,1.0);
              // compute X
              cvMatMul(invH2, ptxp, ptx);
             curpi = CLIP2(0, height-1, (int)(cvmGet(ptx,1,0)/cvmGet(ptx,2,0)));
              curpj = CLIP2(0, width-1, (int)(cvmGet(ptx,0,0)/cvmGet(ptx,2,0)));
```

```
                cvSet2D(img_scene,curpi,curpj,cvGet2D(img_affine,i,j));
                cvmSet(check,curpi,curpj,1);
          }
    }

    // output reconstructed scene image
    img_interp  = cvCloneImage(img_scene);
    data_interp = (uchar *)img_interp->imageData;
    //interpolation
    for (i=1; i<height-1; i++){       //y - ver
          for (j=1; j<width-1; j++){   //x - hor
                if(cvmGet(check,i,j) == 0){
                      count = (cvmGet(check,i-
1,j)==1)+(cvmGet(check,i+1,j)==1)+(cvmGet(check,i,j-
1)==1)+(cvmGet(check,i,j+1)==1);
                      if(count != 0 ){
                      for (k=0; k<channels; k++){
                            data_interp[i*step+j*channels+k] =
(int)((data_scene[(i-
1)*step+j*channels+k]+data_scene[(i+1)*step+j*channels+k]+data_scene[i*step+(j-
1)*channels+k]+data_scene[i*step+(j+1)*channels+k])/count);
                      }
                      }
                }
          }
    }
    if(!cvSaveImage("scene.jpg",img_interp))
          printf("Could not save file\n");

    // release the image
    cvReleaseImage(&img_in);
    cvReleaseImage(&img_affine);
    cvReleaseImage(&img_scene);
    cvReleaseImage(&img_interp);
    return 0;
}
```

Code for method 2

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <cv.h>
#include <highgui.h>

#define CLIP2(minv, maxv, value) (min(maxv, max(minv, value)))

CvMat* cvmGetCol(CvMat * src, int iCol){
  int i;
  int nRow = src->rows;
  CvMat *curCol = cvCreateMat(nRow,1,src->type);
  for(i=0; i<nRow; i++)
      cvmSet(curCol, i, 0, cvmGet(src, i, iCol));
  return curCol;
}

int main(int argc, char *argv[])
{
  IplImage *img_in = 0, *img_out = 0, *img_interp = 0;;
  int height, width, step, channels;
  uchar *data_in, *data_out, *data_interp;
  int i,j,k;
  double distp_x, distp_y;
  int curpi, curpj;
  double nunitdata[] = {-1, -1, -1};
  CvMat nunit = cvMat(3,1,CV_64FC1, &nunitdata);;

  if(argc<2){
    printf("Usage: main <image-file-name>\n\7");
    exit(0);
  }

  // X_c is the camera image, X is the true image
  // load the distorted image X_c
  img_in = cvLoadImage(argv[1]);
  if(!img_in){
    printf("Could not load image file: %s\n",argv[1]);
    exit(0);
  }
  height    = img_in->height;
  width     = img_in->width;
  step      = img_in->widthStep;
  channels  = img_in->nChannels;
  data_in   = (uchar *)img_in->imageData;
  printf("Processing a %d x %d image with %d channels\n",height,width,channels);

  CvMat *check = cvCreateMat(height, width, CV_64FC1); // used for interpolation
  cvZero(check);

  // allocate the output image and initialize
  img_out  = cvCreateImage(cvSize(width, height), IPL_DEPTH_8U, channels);
  cvZero(img_out);
  data_out = (uchar *)img_out->imageData;
```

```
  // get five pairs of orthogonal lines
  // for each lines, two points needed
  CvMat *A = cvCreateMat(5,5,CV_64FC1);
  CvMat *x = cvCreateMat(5,1,CV_64FC1);
  CvMat *b = cvCreateMat(5,1,CV_64FC1);
  CvMat *l = cvCreateMat(3,1,CV_64FC1);
  CvMat *m = cvCreateMat(3,1,CV_64FC1);

  CvMat * vetex = cvCreateMat(3,20,CV_64FC1);  // 4*5 columns representing 20
points
  for(i=0; i<5; i++){
        for(j=0; j<4; j++){
              printf("input the %d th point of the %d pair (x',y') ", j, i);
              scanf("%f %f", &distp_x, &distp_y);
              printf("\n");
              distp_x = tmpxp[i*4+j];
              distp_y = tmpyp[i*4+j];
              cvmSet(vetex,0,i*4+j,distp_x);
              cvmSet(vetex,1,i*4+j,distp_y);
              cvmSet(vetex,2,i*4+j,1);
        }

        // compute lines by cross product
        cvCrossProduct(cvmGetCol(vetex,i*4), cvmGetCol(vetex,i*4+1), l);
        cvCrossProduct(cvmGetCol(vetex,i*4+2), cvmGetCol(vetex,i*4+3), m);

        // create one equation
        cvmSet(A,i,0,cvmGet(l,0,0)*cvmGet(m,0,0));
        cvmSet(A,i,1,(cvmGet(l,0,0)*cvmGet(m,1,0)
                     +cvmGet(l,1,0)*cvmGet(m,0,0))/2.0);
        cvmSet(A,i,2,cvmGet(l,1,0)*cvmGet(m,1,0));
        cvmSet(A,i,3,(cvmGet(l,0,0)*cvmGet(m,2,0)
                     +cvmGet(l,2,0)*cvmGet(m,0,0))/2.0);
        cvmSet(A,i,4,(cvmGet(l,1,0)*cvmGet(m,2,0)
                     +cvmGet(l,2,0)*cvmGet(m,1,0))/2.0);
        cvmSet(b,i,0,-cvmGet(l,2,0)*cvmGet(m,2,0));
  }

  cvSolve(A,b,x);
  double Cdata[] = {cvmGet(x,0,0),     cvmGet(x,1,0)/2.0, cvmGet(x,3,0)/2.0,
                       cvmGet(x,1,0)/2.0, cvmGet(x,2,0),     cvmGet(x,4,0)/2.0,
                              cvmGet(x,3,0)/2.0, cvmGet(x,4,0)/2.0, 1};
  CvMat C = cvMat(3,3,CV_64FC1, &Cdata);

  // C is symmetirc
  CvMat* U  = cvCreateMat(3,3,CV_64FC1);  //eigenvector matrix
  CvMat* U_T  = cvCreateMat(3,3,CV_64FC1);
  CvMat* E  = cvCreateMat(3,1,CV_64FC1);  //eigenvalues

  cvEigenVV(&C, U, E);  //C=U^T D U
  if(cvmGet(E,0,0)*cvmGet(E,1,0)<0){
       printf("error! Please choose another set of lines");
  }
  if(cvmGet(E,0,0)<0 && cvmGet(E,1,0)<0){
        for (i=0;i<3;i++)
              for (j=0;j<3;j++)
                    cvmSet(&C,i,j,cvmGet(&C,i,j)*-1);
```

```
        cvEigenVV(&C, U, E);  //C=U^T D U
    }
    cvTranspose(U,U_T);

    CvMat * ss = cvCreateMat(3,3,CV_64FC1);
    cvZero(ss);
    for(i=0; i<2; i++)
        cvmSet(ss,i,i,sqrt(abs(cvmGet(E,i,0))));
    cvmSet(ss,2,2,10);
    //double ccdata[] = {1,0,0,0,1,0,0,0,cvmGet(E,2,0)/100};
    //CvMat cc  = cvMat(3,3,CV_64FC1, &ccdata);
    //note that C=U^T ss cc ss U

    cvMatMul(U_T,ss,U_T);
    cvTranspose(U_T,U); //now C = U^T cc U where cc is approximately the dual
degenerate conic

    //apply a Hs similarity transform in order to let the output in image window
    //which is okay because C = U^T cc U = U^T Hs cc Hs^T U (cc is invariant to Hs)
    double scale = 0.02;
    double angle = 0.3;
    double Hsdata[] = {scale*cos(angle), -scale*sin(angle), 0, scale*sin(angle),
scale*cos(angle), 0, 0, 0, 1};
    CvMat Hs = cvMat(3,3,CV_64FC1, &Hsdata);
    cvMatMul(U_T,&Hs,U_T);

    CvMat* invH   = cvCreateMat(3,3,CV_64FC1);
    cvInvert(U_T, invH);

    // transform X = invH*X_c
    CvMat * ptx  = cvCreateMat(3,1,CV_64FC1);
    CvMat * ptxp = cvCreateMat(3,1,CV_64FC1);
    for (i=0; i<height; i++){      //y - ver
        for (j=0; j<width; j++){   //x - hor
            // set X_c
            cvmSet(ptxp,0,0,(double)j);
            cvmSet(ptxp,1,0,(double)i);
            cvmSet(ptxp,2,0,1.0);
            // compute X_a
            cvMatMul(invH, ptxp, ptx);
           curpi = CLIP2(0, height-1, (int)(cvmGet(ptx,1,0)/cvmGet(ptx,2,0)));
            curpj = CLIP2(0, width-1, (int)(cvmGet(ptx,0,0)/cvmGet(ptx,2,0)));

            cvSet2D(img_out,curpi,curpj,cvGet2D(img_in,i,j));
            cvmSet(check,curpi,curpj,1);
        }
    }

    // output reconstructed affine image
    img_interp  = cvCloneImage(img_out);
    data_interp = (uchar *)img_interp->imageData;
    //interpolation
    double count;
    for (i=1; i<height-1; i++){      //y - ver
        for (j=1; j<width-1; j++){   //x - hor
            if(cvmGet(check,i,j) == 0){
                count = (cvmGet(check,i-1,j)==1)+(cvmGet(check,i+1,j)==1)+
                        (cvmGet(check,i,j-1)==1)+(cvmGet(check,i,j+1)==1)+
```

```
                                    (cvmGet(check,i-1,j-1)==1)+(cvmGet(check,i-
1,j+1)==1)+
                                    (cvmGet(check,i+1,j-
1)==1)+(cvmGet(check,i+1,j+1)==1);
                        if(count != 0){
                              for (k=0; k<channels; k++)
                                    data_interp[i*step+j*channels+k] = (int)
                                    ((data_out[(i-
1)*step+j*channels+k]+data_out[(i+1)*step+j*channels+k]
                                   +data_out[i*step+(j-
1)*channels+k]+data_out[i*step+(j+1)*channels+k]
                                    +data_out[(i-1)*step+(j-
1)*channels+k]+data_out[(i-1)*step+(j+1)*channels+k]
                                    +data_out[(i+1)*step+(j-
1)*channels+k]+data_out[(i+1)*step+(j+1)*channels+k])/count);

                        }
                  }
            }
      }

  if(!cvSaveImage("affine.jpg",img_interp))
        printf("Could not save file\n");

   // release the image
  cvReleaseImage(&img_in);
  cvReleaseImage(&img_out);
  cvReleaseImage(&img_interp);
  return 0;
}
```