ECE 661 HW 2

Chad Aeschliman

2008-09-18

1 Problem

The problem is to determine the homography H up to similarity which maps a point or line from a plane in the world to the image plane $\acute{x} = Hx$. This is to be done using two different methods. For the first method, the projective error is first removed followed by the affine error using the provided coordinates of a rectangle and an additional orthogonal line pair. For the second method, five orthogonal line pairs are used to determine the image of the dual degenerate conic $\acute{C}^*_\infty$ and from this H is obtained directly up to similarity.

2 Solution

2.1 2-Step Method

The corner points of the provided rectangle can be used to obtain two sets of parallel lines (the top/bottom edges and left/right edges). Each set of parallel lines intersect at the image of an ideal point (a vanishing point). Thus the two sets of parallel lines provide two vanishing points. Computing the straight line through these points gives the vanishing line $\mathbf{l}_v$. To correct for the projection distortion, the vanishing line needs to be mapped to $\mathbf{l}_\infty = [0\ 0\ 1]^T$. This is done using the homography

$$H_p = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ l_1 & l_2 & l_3 \end{bmatrix} \tag{1}$$

where $\mathbf{l}_v = [l_1\ l_2\ l_3]^T$.

The second step is to remove the affine distortion. This is done by exploiting the invariance of cosines computed using the dual degenerate conic C^*_∞. In particular, for a pair of lines $\mathbf{l}$ and $\mathbf{m}$ which are orthogonal in the scene we have

$$\begin{aligned} 0 &= \acute{\mathbf{l}}^T H C^*_\infty H^T \acute{\mathbf{m}} \qquad (2) \\ &= \acute{\mathbf{l}}^T \acute{C}^*_\infty \acute{\mathbf{m}} \qquad (3) \end{aligned}$$

where $\acute{\mathbf{l}}$ and $\acute{\mathbf{m}}$ are the image of $\mathbf{l}$ and $\mathbf{m}$. Note that only an affine homography needs to be determined:

$$H_a = \begin{bmatrix} K & \mathbf{0} \\ \mathbf{0} & 1 \end{bmatrix} \tag{4}$$

Because of this $\acute{C}^*_\infty$ has the form

$$\acute{C}^*_\infty = \begin{bmatrix} S & \mathbf{0} \\ \mathbf{0} & 0 \end{bmatrix} \tag{5}$$

where $S = KK^T$. Because S has only two degrees of freedom (it is symmetric and homogeneous), two sets of orthogonal lines are required to solve for it. One corner of the supplied rectangle is used for one set of orthgonal lines and the other set is supplied separately. Note that two corners of the rectangle cannot be used because parallel lines differ only in the third element which is ignored when multiplying by $\acute{C}^*_\infty$. Once S is determined, an SVD decomposition is used to determine a suitable K, completing H_a. The overall mapping is given by the product of the individual mappings, $H = H_pH)a$.

2.2 C_∞^* Method

The previous method used the properties of C_∞^* only to correct for affine distortion, however, it can be used to solve for a general homography up to similarity. Let the homography be defined as

$$H = \begin{bmatrix} K & \mathbf{0} \\ \mathbf{v}^T & 1 \end{bmatrix} \tag{6}$$

Then it is easy to show that

$$\acute{C}_\infty^* = HC_\infty^* H^T = \begin{bmatrix} KK^T & K\mathbf{v} \\ \mathbf{v}^T K^T & \mathbf{v}\mathbf{v}^T \end{bmatrix} \tag{7}$$

Assuming that we can solve for $\acute{C}_\infty^*$ in the image, we can determine H by solving first for the 2×2 matrix K and then solving for $\mathbf{v}$.

To solve for $\acute{C}_\infty^*$, we first write it in the general form

$$\acute{C}_\infty^* = \begin{bmatrix} a & 0.5b & 0.5d \\ 0.5b & c & 0.5e \\ 0.5d & 0.5e & f \end{bmatrix} \tag{8}$$

We know that $\acute{\mathbf{l}}^T \acute{C}_\infty^* \acute{\mathbf{m}} = 0$ for images of orthogonal lines $\mathbf{l}$ and $\mathbf{m}$ (from the previous section). Plugging in the general form of $\acute{C}_\infty^*$ gives an equation of the form $0 = \mathbf{w}^T\mathbf{c}$ where the entries in $\mathbf{w}$ are given in the text and $\mathbf{c} = [a\ b\ c\ d\ e\ f]^T$. Stacking five such constraints in a matrix we are able to solve for $\mathbf{c}$ as the null vector giving $\acute{C}_\infty^*$.

2.3 Constructing the Corrected Image

To construct the corrected image, the goal is to set up a grid of evenly spaced points in the world plane with each point representing one pixel in the corrected image and then mapping this grid of points to the image using H. The extents of the grid are determined by mapping the corners of the image plane to the world plane using H^{-1}. The smallest and largest coordinates form the extents of the grid. The spacing of the grid points is chosen to set the resolution of the corrected image and the convention was chosen to preserve the width of the original image with the height adjusting as necessary.

In general a grid point does not map to a particular pixel in the original image so linear interpolation was used.

3 Results

9 pairs of original and corrected images for each method are included at the end after the code.

4 Code

4.1 2 step method

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

void premap_image(CvMat* invH, IplImage* image, int* new_height, double* scale_factor, double*
    trans_x, double* trans_y);
void map_image(CvMat* H, double scale_factor, double trans_x, double trans_y, IplImage* image,
    IplImage* corrected_image);

// utility function which prints out a matrix
void printMatrix(CvMat* M, const char* name)
{
   int indent_size;
   int i,j;
   int rows = M->rows;
   int cols = M->cols;

   // print the matrix name
   indent_size = printf("%s = ", name);

   // print out the matrix
```

```
    for (i=0;i<rows;i++)
    {
        // start of a row
        printf("[");
        for (j=0;j<cols;j++)
        {
            if (j>0)
                printf(",");
            printf("%11.5lg",cvmGet(M,i,j));
        }
        printf("]\n");

        // indent the next line
        for (j=0;j<indent_size;j++)
        {
            printf(" ");
        }
    }
    printf("\n");
}

// main function, reads in an image file and the coordinates
// of a rectangle and an additional orthogonal pair of lines
// in the image and then corrects the image using a two step
// process.
int main( int argc, char** argv )
{
    char* filename;

    IplImage* image;
    IplImage* perspective_corrected_image;
    IplImage* corrected_image;

    int i,j,k;
    double xi[8];
    double yi[8];

    // attempt to read in the image file
    if (argc >= 2)
    {
        filename = argv[1];
    }
    else
    {
        cvReleaseImage(&image);
        printf("Usage: hw2a filename");
        return 1;
    }
    if((image = cvLoadImage(filename,1)) == 0)
    {
        printf("Could not read the file!");
        return 2;
    }

    // read user input of the corners of a rectangle
    // and additional orthogonal line pair
    printf("Enter image coordinates for the upper left corner of a rectangle as x,y: ");
    scanf("%lf,%lf",&xi[0],&yi[0]);
    printf("Enter image coordinates for the upper right corner of a rectangle as x,y: ");
    scanf("%lf,%lf",&xi[1],&yi[1]);
    printf("Enter image coordinates for the lower right corner of a rectangle as x,y: ");
    scanf("%lf,%lf",&xi[2],&yi[2]);
    printf("Enter image coordinates for the lower left corner of a rectangle as x,y: ");
    scanf("%lf,%lf",&xi[3],&yi[3]);
    printf("Enter the first image coordinate for a line as x,y: ");
    scanf("%lf,%lf",&xi[4],&yi[4]);
    printf("Enter the second image coordinate for a line as x,y: ");
    scanf("%lf,%lf",&xi[5],&yi[5]);
    printf("Enter the first image coordinate for a line orthogonal to the previous line as x,y: ");
    scanf("%lf,%lf",&xi[6],&yi[6]);
    printf("Enter the second image coordinate for a line orthogonal to the previous line as x,y: ");
    scanf("%lf,%lf",&xi[7],&yi[7]);
    printf("\n");
```

```
// STEP 1: find the vanishing line using the rectangle
CvMat* temp_point0 = cvCreateMat(3,1,CV_64FC1);
CvMat* temp_point1 = cvCreateMat(3,1,CV_64FC1);
CvMat* temp_line0 = cvCreateMat(3,1,CV_64FC1);
CvMat* temp_line1 = cvCreateMat(3,1,CV_64FC1);
CvMat* vanishing_point0 = cvCreateMat(3,1,CV_64FC1);
CvMat* vanishing_point1 = cvCreateMat(3,1,CV_64FC1);
CvMat* vanishing_line = cvCreateMat(3,1,CV_64FC1);

for (j=0;j<2;j++)
{
   // first compute two "parallel" lines (parallel in the scene)
   for (i=0;i<2;i++)
   {
      int index1 = 2*i + j;
      int index2 = (index1 + 1)%4;
      cvmSet(temp_point0,0,0,xi[index1]);
      cvmSet(temp_point0,1,0,yi[index1]);
      cvmSet(temp_point0,2,0,1.0);
      cvmSet(temp_point1,0,0,xi[index2]);
      cvmSet(temp_point1,1,0,yi[index2]);
      cvmSet(temp_point1,2,0,1.0);

      // compute the straight line through the two corner points
      if (i==0)
         cvCrossProduct(temp_point0,temp_point1,temp_line0);
      else
         cvCrossProduct(temp_point0,temp_point1,temp_line1);
   }

   // now determine their intersection to get a vanishing point
   if (j==0)
      cvCrossProduct(temp_line0,temp_line1,vanishing_point0);
   else
      cvCrossProduct(temp_line0,temp_line1,vanishing_point1);
}
// now determine the vanishing line from the 2 vanishing points
cvCrossProduct(vanishing_point0,vanishing_point1,vanishing_line);
cvNormalize(vanishing_line,vanishing_line,1,0,CV_L2,NULL);
printMatrix(vanishing_line,"vanishing_line");

// STEP 2: determine the perspective correcting transform
CvMat* HP = cvCreateMat(3,3,CV_64FC1);
CvMat* invHP = cvCreateMat(3,3,CV_64FC1);

cvZero(invHP);
cvmSet(invHP,0,0,1.0);
cvmSet(invHP,1,1,1.0);
cvmSet(invHP,2,0,cvmGet(vanishing_line,0,0));
cvmSet(invHP,2,1,cvmGet(vanishing_line,1,0));
cvmSet(invHP,2,2,cvmGet(vanishing_line,2,0));
cvInvert(invHP,HP,CV_LU);

cvNormalize(HP,HP,1,0,CV_L2,NULL);
cvNormalize(invHP,invHP,1,0,CV_L2,NULL);

printMatrix(HP,"HP");
printMatrix(invHP,"invHP");


double xp[8];
double yp[8];

// STEP 3: determine the perspective corrected coordinates of the
// rectangle and orthogonal lines
CvMat* im_coord = cvCreateMat(3,1,CV_64FC1);
CvMat* pc_coord = cvCreateMat(3,1,CV_64FC1);
cvmSet(im_coord,2,0,1.0);
for (i=0;i<8;i++)
{
   cvmSet(im_coord,0,0,xi[i]);
   cvmSet(im_coord,1,0,yi[i]);
   cvMatMul(invHP,im_coord,pc_coord);
   xp[i] = cvmGet(pc_coord,0,0)/cvmGet(pc_coord,2,0);
```

```
    yp[i] = cvmGet(pc_coord,1,0)/cvmGet(pc_coord,2,0);
}


// STEP 4: determine the affine mapping
CvMat* sol_matrix = cvCreateMat(2,3,CV_64FC1);
for (j=0;j<2;j++)
{
    // first compute two "orthogonal" lines (orthogonal in the scene)
    for (i=0;i<2;i++)
    {
        int index1 = i*(j+1) + j*4;
        int index2 = index1 + 1;
        cvmSet(temp_point0,0,0,xp[index1]);
        cvmSet(temp_point0,1,0,yp[index1]);
        cvmSet(temp_point0,2,0,1.0);
        cvmSet(temp_point1,0,0,xp[index2]);
        cvmSet(temp_point1,1,0,yp[index2]);
        cvmSet(temp_point1,2,0,1.0);

        // compute the straight line through the two corner points
        if (i==0)
            cvCrossProduct(temp_point0,temp_point1,temp_line0);
        else
            cvCrossProduct(temp_point0,temp_point1,temp_line1);
    }
    // now add the appropriate row to the solution matrix
    printMatrix(temp_line0,"l0");
    printMatrix(temp_line1,"l1");
    double l1 = cvmGet(temp_line0,0,0);
    double l2 = cvmGet(temp_line0,1,0);
    double m1 = cvmGet(temp_line1,0,0);
    double m2 = cvmGet(temp_line1,1,0);
    cvmSet(sol_matrix,j,0,l1*m1);
    cvmSet(sol_matrix,j,1,l1*m2+l2*m1);
    cvmSet(sol_matrix,j,2,l2*m2);
}
printMatrix(sol_matrix,"sol_matrix");

// compute the null space of the solution matrix, this gives
// the entries of S = K*K
CvMat* V = cvCreateMat(3,3,CV_64FC1);
CvMat* temp_mat = cvCreateMat(2,3,CV_64FC1);
cvSVD(sol_matrix,temp_mat,NULL,V,0);
CvMat* S = cvCreateMat(2,2,CV_64FC1);
cvmSet(S,0,0,cvmGet(V,0,2));
cvmSet(S,1,0,cvmGet(V,1,2));
cvmSet(S,0,1,cvmGet(V,1,2));
cvmSet(S,1,1,cvmGet(V,2,2));
printMatrix(V,"V");
printMatrix(S,"S");

// determine K using an SVD
CvMat* U = cvCreateMat(2,2,CV_64FC1);
CvMat* D2 = cvCreateMat(2,2,CV_64FC1);
cvSVD(S,D2,U,NULL,CV_SVD_V_T);
CvMat* D = cvCreateMat(2,2,CV_64FC1);
cvPow(D2,D,0.5);
CvMat* UD = cvCreateMat(2,2,CV_64FC1);
cvMatMul(U,D,UD);
CvMat* K = cvCreateMat(2,2,CV_64FC1);
cvGEMM(UD,U,1.0,NULL,0,K,CV_GEMM_B_T);
printMatrix(U,"U");
printMatrix(D2,"D2");
printMatrix(D,"D");
printMatrix(K,"K");

// Compute the affine mapping
CvMat* HA = cvCreateMat(3,3,CV_64FC1);
CvMat* invHA = cvCreateMat(3,3,CV_64FC1);
cvZero(HA);
for (i=0;i<2;i++)
{
    for (j=0;j<2;j++)
```

```
        {
            cvmSet(HA,i,j,cvmGet(K,i,j));
        }
    }
    cvmSet(HA,2,2,1.0);
    cvInvert(HA,invHA,CV_LU);
    printMatrix(HA,"HA");
    printMatrix(invHA,"invHA");

    // create the full mapping by multiplying the affine and
    // perspective parts
    CvMat* H = cvCreateMat(3,3,CV_64FC1);
    CvMat* invH = cvCreateMat(3,3,CV_64FC1);
    cvMatMul(HP,HA,H);
    cvMatMul(invHA,invHP,invH);
    printMatrix(H,"H");
    printMatrix(invH,"invH");

    // draw lines on the image
    for (i=0;i<4;i++)
    {
        j = (i+1)%4;
        cvLine(image,cvPoint(xi[i],yi[i]),cvPoint(xi[j],yi[j]),CV_RGB(0,255,0),5,CV_AA,0);
    }
    for (i=4;i<8;i+=2)
    {
        j = i+1;
        cvLine(image,cvPoint(xi[i],yi[i]),cvPoint(xi[j],yi[j]),CV_RGB(255,0,0),5,CV_AA,0);
    }

    // map the image to correct the perspective distortion
    int new_height;
    double scale_factor, trans_x, trans_y;
    premap_image(invHP, image, &new_height, &scale_factor, &trans_x, &trans_y);
    perspective_corrected_image = cvCreateImage(cvSize(image->width,new_height),
                                    IPL_DEPTH_8U, 3);
    map_image(HP,scale_factor,trans_x, trans_y,image,perspective_corrected_image);


    // save perspective corrected image
    char new_filename[128];
    strcpy(new_filename,filename);
    sprintf(new_filename+strlen(filename)-4,"_pc.png");
    cvSaveImage(new_filename,perspective_corrected_image);

    // map the image to correct the affine distortion
    premap_image(invH, image, &new_height, &scale_factor, &trans_x, &trans_y);
    corrected_image = cvCreateImage(cvSize(image->width,new_height),
                                    IPL_DEPTH_8U, 3);
    map_image(H,scale_factor,trans_x, trans_y,image,corrected_image);

    // save affinely corrected image
    strcpy(new_filename,filename);
    sprintf(new_filename+strlen(filename)-4,"_corr.png");
    cvSaveImage(new_filename,corrected_image);

    // save original image with lines
    strcpy(new_filename,filename);
    sprintf(new_filename+strlen(filename)-4,"_lines.png");
    cvSaveImage(new_filename,image);

    // cleanup
    cvReleaseImage(&image);
    cvReleaseImage(&perspective_corrected_image);
    cvReleaseImage(&corrected_image);

    return 0;

}

void premap_image(CvMat* invH, IplImage* image, int* new_height, double* scale_factor, double*
    trans_x, double* trans_y)
{
    int i,j,k;
```

```
    // we need to determine the new bounds that the
    // image represents
    double max_x = -1.0e100;
    double max_y = -1.0e100;
    double min_x = 1.0e100;
    double min_y = 1.0e100;
    CvMat* world_coord = cvCreateMat(3,4,CV_64FC1);
    CvMat* image_coord = cvCreateMat(3,4,CV_64FC1);

    // pick some representative image coordinates
    cvmSet(image_coord,0,0,0);
    cvmSet(image_coord,1,0,0);
    cvmSet(image_coord,2,0,1);

    cvmSet(image_coord,0,1,0);
    cvmSet(image_coord,1,1,image->height-1);
    cvmSet(image_coord,2,1,1);

    cvmSet(image_coord,0,2,image->width-1);
    cvmSet(image_coord,1,2,0);
    cvmSet(image_coord,2,2,1);

    cvmSet(image_coord,0,3,image->width-1);
    cvmSet(image_coord,1,3,image->height-1);
    cvmSet(image_coord,2,3,1);

    // apply H^-1 to get the corresponding world coordinates
    cvMatMul(invH,image_coord,world_coord);

    // now check the bounds
    for (i=0;i<4;i++)
    {
        double w = cvmGet(world_coord,2,i);
        double real_x = cvmGet(world_coord,0,i)/w;
        double real_y = cvmGet(world_coord,1,i)/w;

        if (real_x < min_x)
            min_x = real_x;
        if (real_y < min_y)
            min_y = real_y;
        if (real_x > max_x)
            max_x = real_x;
        if (real_y > max_y)
            max_y = real_y;

    }


    // determine the correct size for the corrected image
    // assuming that the new width should match the old width
    *scale_factor = ((double)(image->width))/(max_x-min_x);
    *new_height = (int)((max_y-min_y)*(*scale_factor));
    *trans_x = min_x;
    *trans_y = min_y;
}

void map_image(CvMat* H, double scale_factor, double min_x, double min_y, IplImage* image, IplImage
    * corrected_image)
{
    int i,j,k;

    cvZero(corrected_image);

    // compute the new image by applying H to a grid of points in
    // the world coordinate system
    double step_size = 1.0/scale_factor;
    CvMat* input_coord = cvCreateMat(3,1,CV_64FC1);
    CvMat* output_coord = cvCreateMat(3,1,CV_64FC1);
    cvmSet(input_coord,2,0,1.0);
    for (i=0; i<corrected_image->width; i++)
    {
        cvmSet(input_coord,0,0,((double)i)*step_size+min_x);
        for (j=0; j<corrected_image->height; j++)
```

```
        {
          double xi,yi,fx,fy;
          cvmSet(input_coord,1,0,((double)j)*step_size+min_y);

          // compute the associated image coordinate
          cvMatMul(H,input_coord,output_coord);
          xi = cvmGet(output_coord,0,0)/cvmGet(output_coord,2,0);
          yi = cvmGet(output_coord,1,0)/cvmGet(output_coord,2,0);

          // if outside of the image then move on
          if (xi<0||yi<0||xi>=(image->width-1)||yi>=(image->height-1))
          {
            continue;
          }

          // compute the fractional component of the image coord.
          fx = xi - (int)xi;
          fy = yi - (int)yi;

          // compute the pixel value using linear interpolation
          for (k=0;k<3;k++)
          {
            double value = 0;
            value += (1.0-fx)*(1.0-fy)*((uchar*)(image->imageData + image->widthStep*(int)yi))[((
                int)xi)*3+k];
            value += (1.0-fx)*fy*((uchar*)(image->imageData + image->widthStep*(int)(yi+1)))[((int)
                xi)*3+k];
            value += fx*(1.0-fy)*((uchar*)(image->imageData + image->widthStep*(int)yi))[((int)(xi
                +1))*3+k];
            value += fx*fy*((uchar*)(image->imageData + image->widthStep*(int)(yi+1)))[((int)(xi+1)
                )*3+k];
           ((uchar*)(corrected_image->imageData + corrected_image->widthStep*j))[i*3+k] = value;
          }
        }
    }
}
```

4.2 C^*_∞ method

```
#include "cv.h"
#include "highgui.h"
#include <stdio.h>
#include <stdlib.h>
#define NUMBER_OF_PAIRS 5

void premap_image(CvMat* invH, IplImage* image, int* new_height, double* scale_factor, double*
    trans_x, double* trans_y);
void map_image(CvMat* H, double scale_factor, double trans_x, double trans_y, IplImage* image,
    IplImage* corrected_image);

// utility function which prints out a matrix
void printMatrix(CvMat* M, const char* name)
{
  int indent_size;
  int i,j;
  int rows = M->rows;
  int cols = M->cols;


  // print the matrix name
  indent_size = printf("%s = ", name);

  // print out the matrix
  for (i=0;i<rows;i++)
  {
    // start of a row
    if (i==0)
      printf("[");
    for (j=0;j<cols;j++)
    {
      if (j>0)
        printf(",");
      printf("%11.5lg",cvmGet(M,i,j));
```

```
        }
        if (i==rows-1)
            printf("]\n");
        else
            printf("\n");

        // indent the next line
        for (j=0;j<indent_size;j++)
        {
            printf(" ");
        }
    }
    printf("\n");
}

// main function, reads in an image file and the coordinates
// of a rectangle and an additional orthogonal pair of lines
// in the image and then corrects the image using a the
// degenerate dual conic.
int main( int argc, char** argv )
{
    char* filename;
    char new_filename[128];

    IplImage* image;
    IplImage* corrected_image;

    int i,j,k;
    double xi[4*NUMBER_OF_PAIRS];
    double yi[4*NUMBER_OF_PAIRS];

    // attempt to read in the image file
    if (argc >= 2)
    {
        filename = argv[1];
    }
    else
    {
        cvReleaseImage(&image);
        printf("Usage: hw2b filename");
        return 1;
    }
    if((image = cvLoadImage(filename,1)) == 0)
    {
        printf("Could not read the file!");
        return 2;
    }

    // read user input of the orthogonal lines
    for (i=0;i<NUMBER_OF_PAIRS;i++)
    {
        printf("Enter the first image coordinate for a line as x,y: ");
        scanf("%lf,%lf",&xi[4*i],&yi[4*i]);
        printf("Enter the second image coordinate for a line as x,y: ");
        scanf("%lf,%lf",&xi[4*i+1],&yi[4*i+1]);
        printf("Enter the first image coordinate for a line orthogonal to the previous line as x,y: "
            );
        scanf("%lf,%lf",&xi[4*i+2],&yi[4*i+2]);
        printf("Enter the second image coordinate for a line orthogonal to the previous line as x,y: 
            ");
        scanf("%lf,%lf",&xi[4*i+3],&yi[4*i+3]);
    }
    printf("\n");


    // STEP 1: determine the constraint matrix
    CvMat* sol_matrix = cvCreateMat(NUMBER_OF_PAIRS,6,CV_64FC1);
    CvMat* temp_point0 = cvCreateMat(3,1,CV_64FC1);
    CvMat* temp_point1 = cvCreateMat(3,1,CV_64FC1);
    CvMat* temp_line0 = cvCreateMat(3,1,CV_64FC1);
    CvMat* temp_line1 = cvCreateMat(3,1,CV_64FC1);
    for (j=0;j<NUMBER_OF_PAIRS;j++)
    {
        // first compute two "orthogonal" lines (orthogonal in the scene)
```

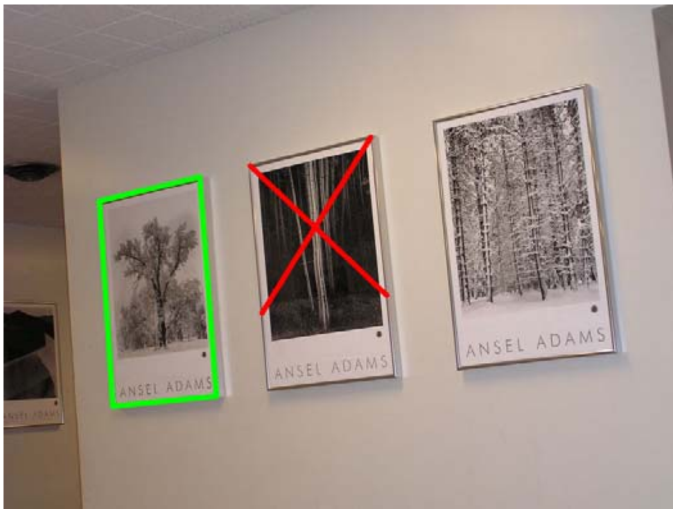
```
    for (i=0;i<2;i++)
    {
        int index1 = 2*i + 4*j;
        int index2 = index1 + 1;
        cvmSet(temp_point0,0,0,xi[index1]);
        cvmSet(temp_point0,1,0,yi[index1]);
        cvmSet(temp_point0,2,0,1.0);
        cvmSet(temp_point1,0,0,xi[index2]);
        cvmSet(temp_point1,1,0,yi[index2]);
        cvmSet(temp_point1,2,0,1.0);

        // compute the straight line through the two corner points
        if (i==0)
            cvCrossProduct(temp_point0,temp_point1,temp_line0);
        else
            cvCrossProduct(temp_point0,temp_point1,temp_line1);
    }
    cvNormalize(temp_line0,temp_line0,1,0,CV_L2,NULL);
    cvNormalize(temp_line1,temp_line1,1,0,CV_L2,NULL);
    printMatrix(temp_line0,"l");
    printMatrix(temp_line1,"m");
    // now add the appropriate row to the solution matrix
    double l1 = cvmGet(temp_line0,0,0);
    double l2 = cvmGet(temp_line0,1,0);
    double l3 = cvmGet(temp_line0,2,0);
    double m1 = cvmGet(temp_line1,0,0);
    double m2 = cvmGet(temp_line1,1,0);
    double m3 = cvmGet(temp_line1,2,0);
    cvmSet(sol_matrix,j,0,l1*m1);
    cvmSet(sol_matrix,j,1,0.5*(l1*m2+l2*m1));
    cvmSet(sol_matrix,j,2,l2*m2);
    cvmSet(sol_matrix,j,3,0.5*(l1*m3+l3*m1));
    cvmSet(sol_matrix,j,4,0.5*(l2*m3+l3*m2));
    cvmSet(sol_matrix,j,5,l3*m3);
}
printMatrix(sol_matrix,"sol_matrix");


// STEP 2: find the null space of sol_matrix and fill
//         in C*inf
CvMat* V = cvCreateMat(6,6,CV_64FC1);
CvMat* temp_mat = cvCreateMat(NUMBER_OF_PAIRS,6,CV_64FC1);
cvSVD(sol_matrix,temp_mat,NULL,V,0);
CvMat* Cstar_inf = cvCreateMat(3,3,CV_64FC1);
cvmSet(Cstar_inf,0,0,cvmGet(V,0,5));      // a
cvmSet(Cstar_inf,1,1,cvmGet(V,2,5));      // c
cvmSet(Cstar_inf,2,2,cvmGet(V,5,5));      // f
cvmSet(Cstar_inf,0,1,0.5*cvmGet(V,1,5)); // b/2
cvmSet(Cstar_inf,1,0,0.5*cvmGet(V,1,5)); // b/2
cvmSet(Cstar_inf,0,2,0.5*cvmGet(V,3,5)); // d/2
cvmSet(Cstar_inf,2,0,0.5*cvmGet(V,3,5)); // d/2
cvmSet(Cstar_inf,1,2,0.5*cvmGet(V,4,5)); // e/2
cvmSet(Cstar_inf,2,1,0.5*cvmGet(V,4,5)); // e/2
if (cvmGet(Cstar_inf,2,2) < 0)
{
    for (i=0;i<3;i++)
    {
        for (j=0;j<3;j++)
        {
            cvmSet(Cstar_inf,i,j,-1.0*cvmGet(Cstar_inf,i,j));
        }
    }
}
printMatrix(V,"V");
printMatrix(temp_mat,"singular_values");
printMatrix(Cstar_inf,"Cstar_inf");


// STEP 3: Determine K and v and then H
CvMat* S = cvCreateMat(2,2,CV_64FC1);
cvRepeat(Cstar_inf,S);
CvMat* U = cvCreateMat(2,2,CV_64FC1);
CvMat* D2 = cvCreateMat(2,2,CV_64FC1);
cvSVD(S,D2,U,NULL,0);
```

```
CvMat* D = cvCreateMat(2,2,CV_64FC1);
cvPow(D2,D,0.5);
CvMat* UD = cvCreateMat(2,2,CV_64FC1);
cvMatMul(U,D,UD);
CvMat* K = cvCreateMat(2,2,CV_64FC1);
cvGEMM(UD,U,1.0,NULL,0,K,CV_GEMM_B_T);
for (i=0;i<2;i++)
{
    for (j=0;j<2;j++)
    {
        cvmSet(K,i,j,1.0*cvmGet(K,i,j));
    }
}

CvMat* sol_vector = cvCreateMat(2,1,CV_64FC1);
cvmSet(sol_vector,0,0,1.0*cvmGet(Cstar_inf,0,2));
cvmSet(sol_vector,1,0,1.0*cvmGet(Cstar_inf,1,2));
CvMat* v = cvCreateMat(2,1,CV_64FC1);
cvSolve(K,sol_vector,v,CV_LU);
CvMat* H = cvCreateMat(3,3,CV_64FC1);
cvZero(H);
for (i=0;i<2;i++)
{
    for (j=0;j<2;j++)
    {
        cvmSet(H,i,j,cvmGet(K,i,j));
    }
    cvmSet(H,2,i,cvmGet(v,i,0));
}
cvmSet(H,2,2,2.0);
printMatrix(S,"S");
printMatrix(K,"K");
printMatrix(v,"v");
printMatrix(H,"H");

// check H by computing H C*inf H^T and comparing it to
// Cstar_inf
CvMat* Cstar = cvCreateMat(3,3,CV_64FC1);
cvZero(Cstar);
cvmSet(Cstar,0,0,1.0);
cvmSet(Cstar,1,1,1.0);
CvMat* HC = cvCreateMat(3,3,CV_64FC1);
cvMatMul(H,Cstar,HC);
CvMat* HCHT = cvCreateMat(3,3,CV_64FC1);
cvGEMM(HC,H,1.0,NULL,0,HCHT,CV_GEMM_B_T);
printMatrix(HCHT,"HCHT");
printf("%lg\n",cvmGet(Cstar_inf,2,2)/cvmGet(HCHT,2,2)); // this should be 1


// STEP 4: compute H^-1 and correct the image
CvMat* invH = cvCreateMat(3,3,CV_64FC1);
cvInvert(H,invH,CV_LU);
printMatrix(invH,"H^-1");

// draw lines on the image
int r = 0;
int b = 255;
for (i=0;i<(4*NUMBER_OF_PAIRS);i+=2)
{
    if (i%4==0)
    {
        r += 255/NUMBER_OF_PAIRS;
        b -= 255/NUMBER_OF_PAIRS;
    }
    j = i+1;
    cvLine(image,cvPoint(xi[i],yi[i]),cvPoint(xi[j],yi[j]),CV_RGB(r,0,b),5,CV_AA,0);
}

// map the image to correct the distortion
int new_height;
double scale_factor, trans_x, trans_y;
premap_image(invH, image, &new_height, &scale_factor, &trans_x, &trans_y);
corrected_image = cvCreateImage(cvSize(image->width,new_height),
                                IPL_DEPTH_8U, 3);
```

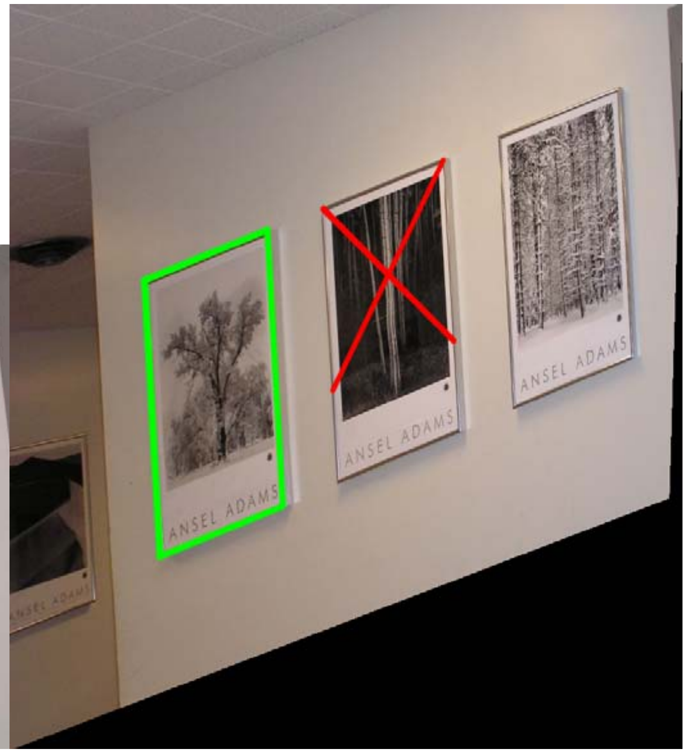
```
    map_image(H,scale_factor,trans_x, trans_y,image,corrected_image);

    // save corrected image
    strcpy(new_filename,filename);
    sprintf(new_filename+strlen(filename)-4,"_corr2.png");
    cvSaveImage(new_filename,corrected_image);

    // save original image with lines
    strcpy(new_filename,filename);
    sprintf(new_filename+strlen(filename)-4,"_lines2.png");
    cvSaveImage(new_filename,image);

    // cleanup
    cvReleaseImage(&image);
    //cvReleaseImage(&corrected_image);


    return 0;

}

void premap_image(CvMat* invH, IplImage* image, int* new_height, double* scale_factor, double*
     trans_x, double* trans_y)
{
    int i,j,k;

    // we need to determine the new bounds that the
    // image represents
    double max_x = -1.0e100;
    double max_y = -1.0e100;
    double min_x = 1.0e100;
    double min_y = 1.0e100;
    CvMat* world_coord = cvCreateMat(3,4,CV_64FC1);
    CvMat* image_coord = cvCreateMat(3,4,CV_64FC1);

    // pick some representative image coordinates
    cvmSet(image_coord,0,0,0);
    cvmSet(image_coord,1,0,0);
    cvmSet(image_coord,2,0,1);

    cvmSet(image_coord,0,1,0);
    cvmSet(image_coord,1,1,image->height-1);
    cvmSet(image_coord,2,1,1);

    cvmSet(image_coord,0,2,image->width-1);
    cvmSet(image_coord,1,2,0);
    cvmSet(image_coord,2,2,1);

    cvmSet(image_coord,0,3,image->width-1);
    cvmSet(image_coord,1,3,image->height-1);
    cvmSet(image_coord,2,3,1);

    // apply H^-1 to get the corresponding world coordinates
    cvMatMul(invH,image_coord,world_coord);

    // now check the bounds
    for (i=0;i<4;i++)
    {
        double w = cvmGet(world_coord,2,i);
        double real_x = cvmGet(world_coord,0,i)/w;
        double real_y = cvmGet(world_coord,1,i)/w;

        if (real_x < min_x)
            min_x = real_x;
        if (real_y < min_y)
            min_y = real_y;
        if (real_x > max_x)
            max_x = real_x;
        if (real_y > max_y)
            max_y = real_y;

    }
```

```
    // determine the correct size for the corrected image
    // assuming that the new width should match the old width
    *scale_factor = ((double)(image->width))/(max_x-min_x);
    *new_height = (int)((max_y-min_y)*(*scale_factor));
    *trans_x = min_x;
    *trans_y = min_y;
}

void map_image(CvMat* H, double scale_factor, double min_x, double min_y, IplImage* image, IplImage
    * corrected_image)
{
    int i,j,k;

    cvZero(corrected_image);

    // compute the new image by applying H to a grid of points in
    // the world coordinate system
    double step_size = 1.0/scale_factor;
    CvMat* input_coord = cvCreateMat(3,1,CV_64FC1);
    CvMat* output_coord = cvCreateMat(3,1,CV_64FC1);
    cvmSet(input_coord,2,0,1.0);
    for (i=0; i<corrected_image->width; i++)
    {
        cvmSet(input_coord,0,0,((double)i)*step_size+min_x);
        for (j=0; j<corrected_image->height; j++)
        {
            double xi,yi,fx,fy;
            cvmSet(input_coord,1,0,((double)j)*step_size+min_y);

            // compute the associated image coordinate
            cvMatMul(H,input_coord,output_coord);
            xi = cvmGet(output_coord,0,0)/cvmGet(output_coord,2,0);
            yi = cvmGet(output_coord,1,0)/cvmGet(output_coord,2,0);

            // if outside of the image then move on
            if (xi<0||yi<0||xi>=(image->width-1)||yi>=(image->height-1))
            {
                continue;
            }

            // compute the fractional component of the image coord.
            fx = xi - (int)xi;
            fy = yi - (int)yi;

            // compute the pixel value using linear interpolation
            for (k=0;k<3;k++)
            {
                double value = 0;
                value += (1.0-fx)*(1.0-fy)*((uchar*)(image->imageData + image->widthStep*(int)yi))[((
                    int)xi)*3+k];
                value += (1.0-fx)*fy*((uchar*)(image->imageData + image->widthStep*(int)(yi+1)))[((int)
                    xi)*3+k];
                value += fx*(1.0-fy)*((uchar*)(image->imageData + image->widthStep*(int)yi))[((int)(xi
                    +1))*3+k];
                value += fx*fy*((uchar*)(image->imageData + image->widthStep*(int)(yi+1)))[((int)(xi+1)
                    )*3+k];
               ((uchar*)(corrected_image->imageData + corrected_image->widthStep*j))[i*3+k] = value;
            }
        }
    }
}
```

5 2-Step Images



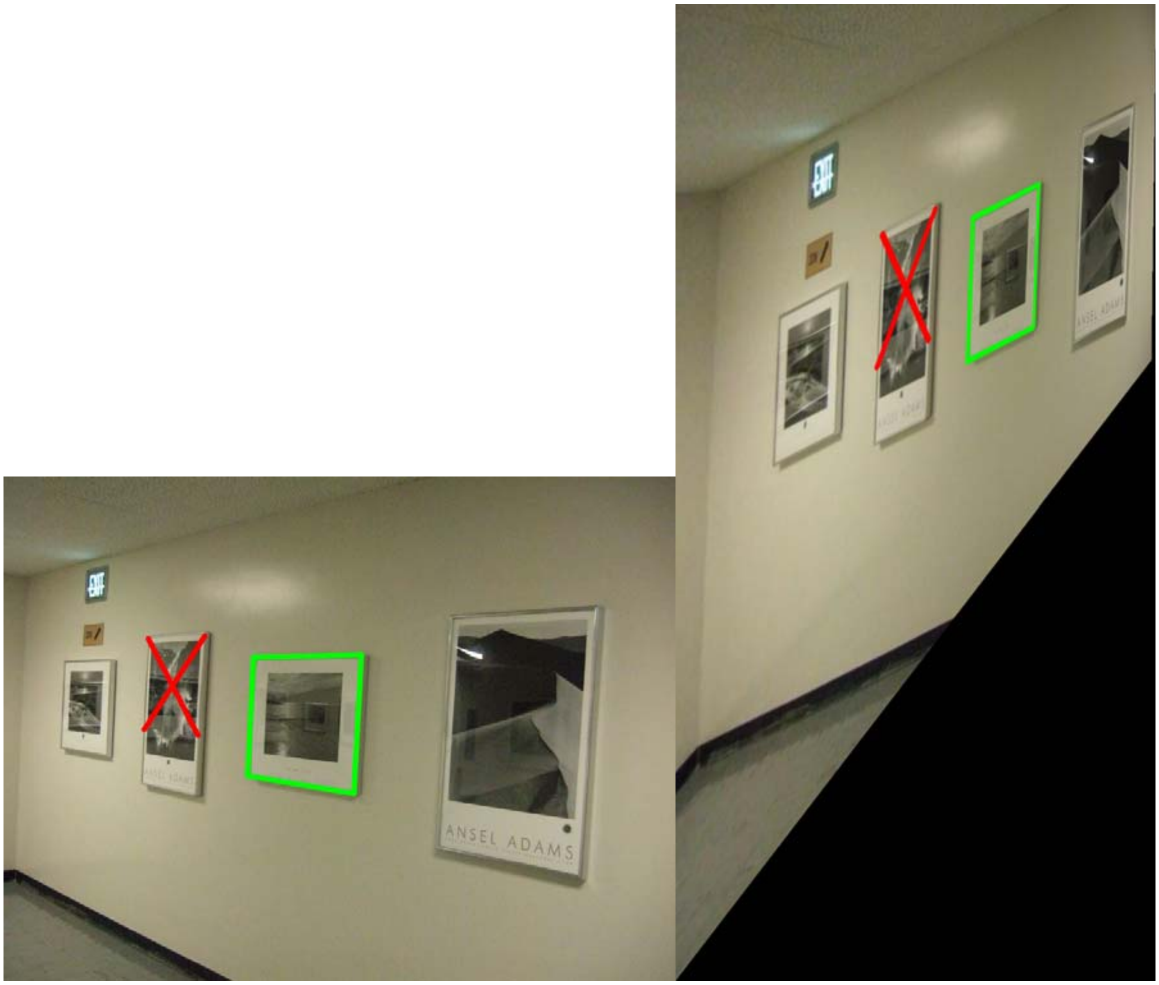
(a) Original with lines



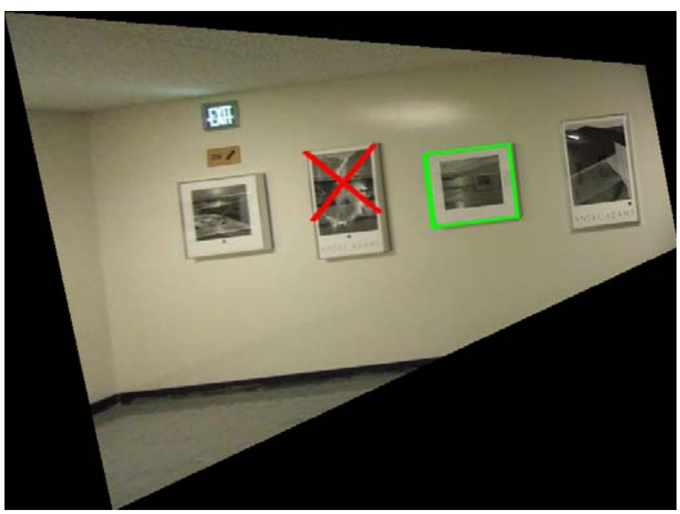
(b) Projective correction

(c) Affine correction

Figure 1: adams01 using 2-step method

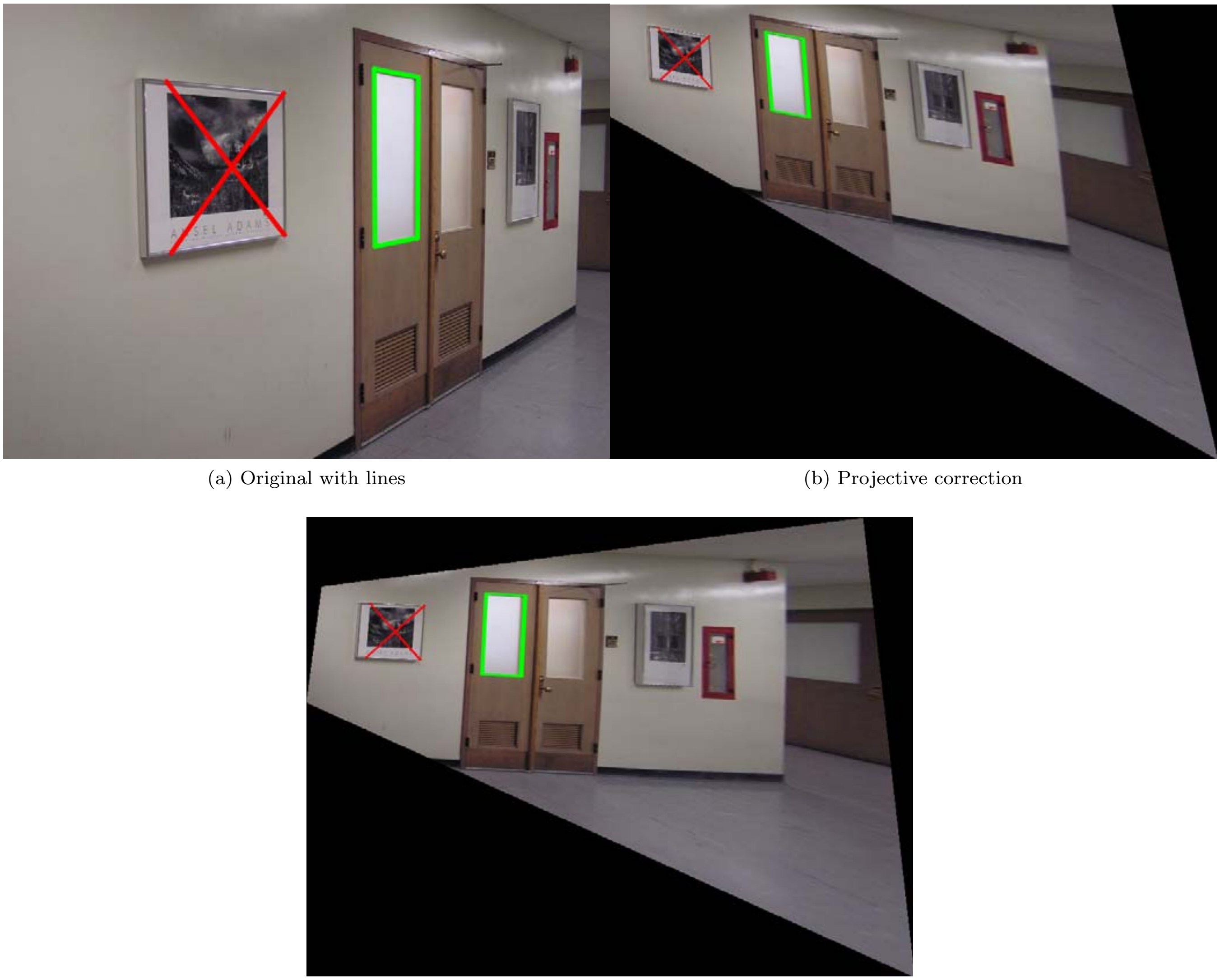


(a) Original with lines (b) Projective correction



(c) Affine correction

Figure 2: adams02 using 2-step method



(a) Original with lines

(b) Projective correction

(c) Affine correction

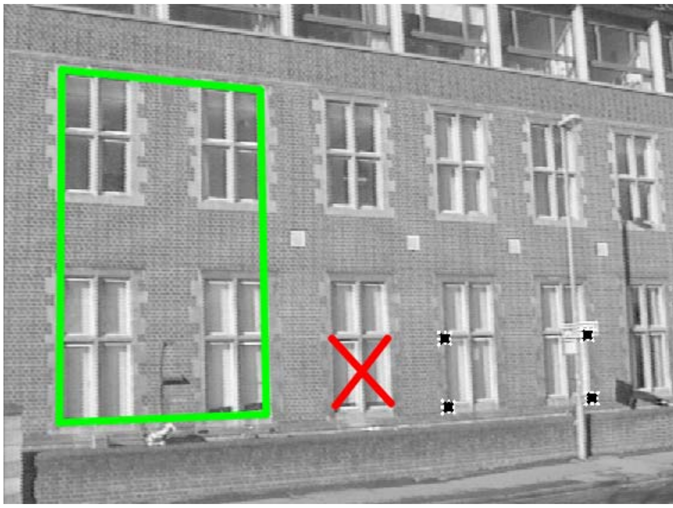
Figure 3: door01 using 2-step method

(a) Original with lines

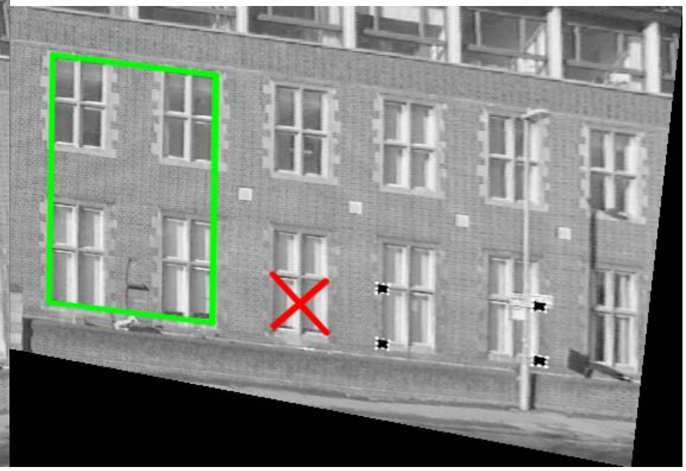
(b) Projective correction

(c) Affine correction

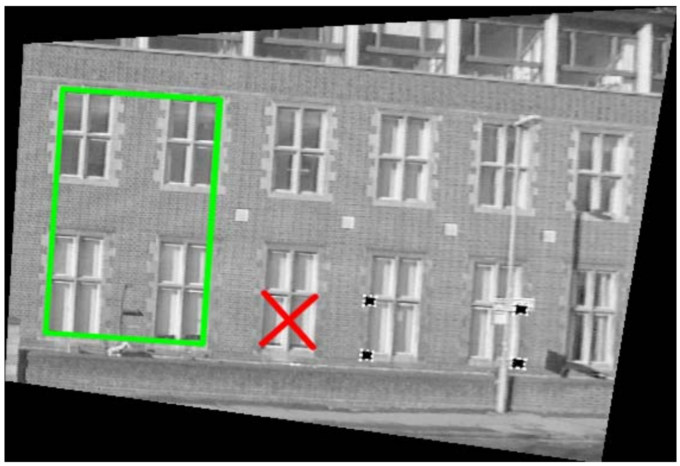
Figure 4: board01 using 2-step method



(a) Original with lines

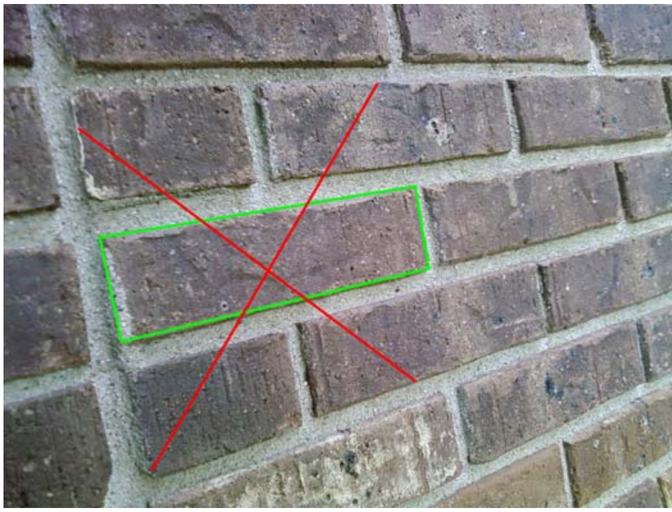


(b) Projective correction

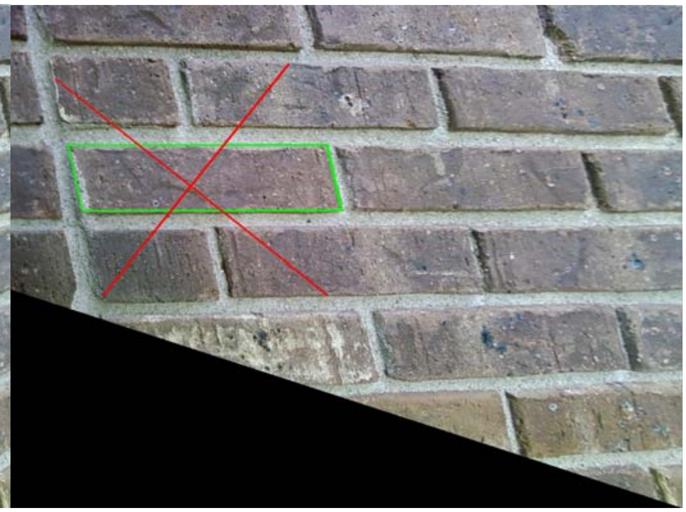


(c) Affine correction

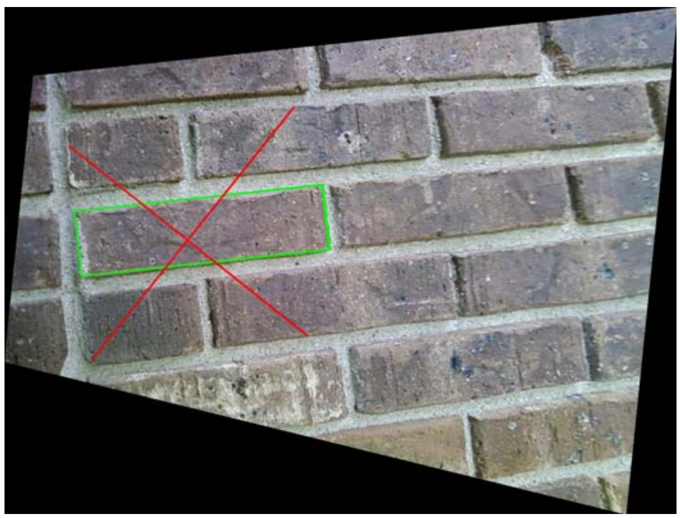
Figure 5: textbook example using 2-step method



(a) Original with lines

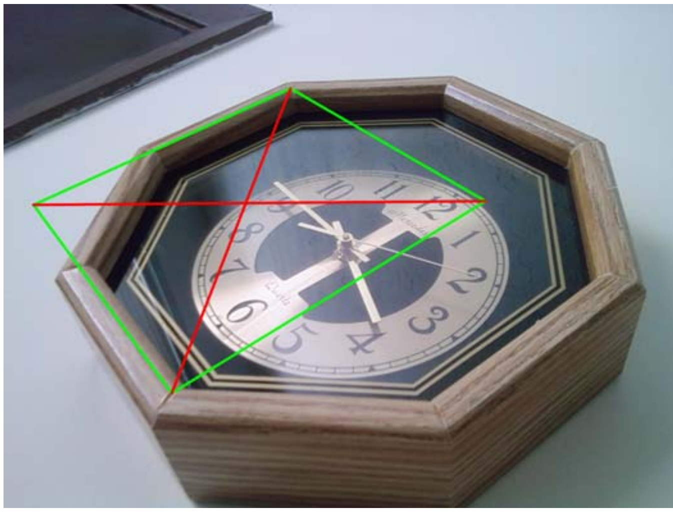


(b) Projective correction

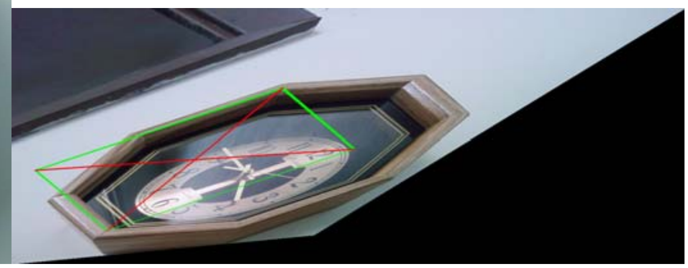


(c) Affine correction

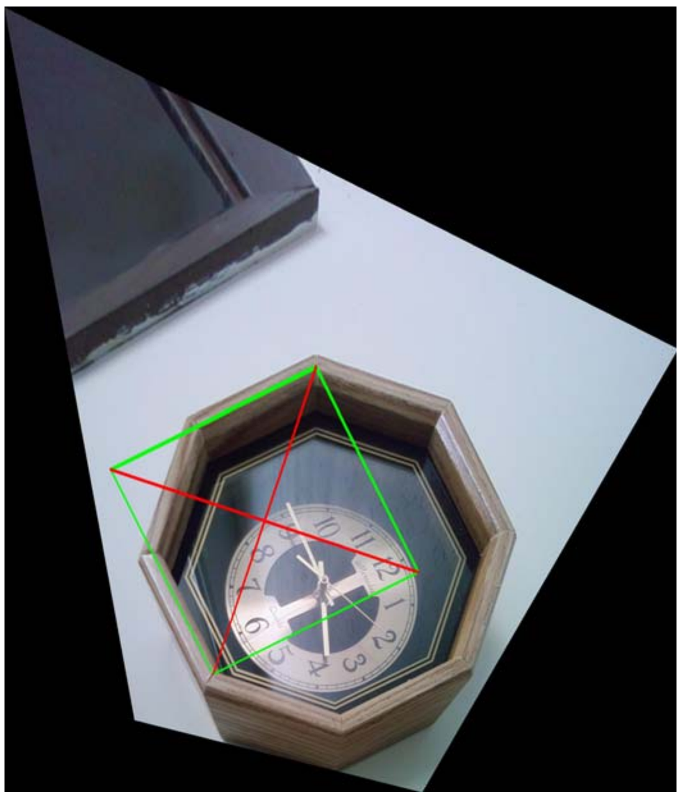
Figure 6: bricks using 2-step method



(a) Original with lines



(b) Projective correction



(c) Affine correction

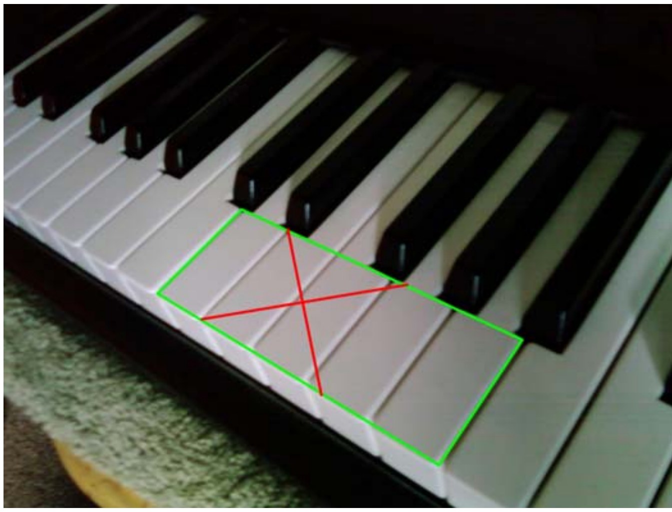
Figure 7: clock using 2-step method

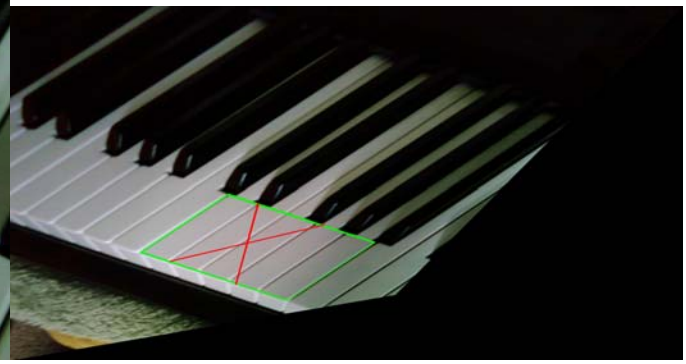
(a) Original with lines

(b) Projective correction

(c) Affine correction

Figure 8: mailboxes using 2-step method


(a) Original with lines


(b) Projective correction


(c) Affine correction

Figure 9: piano using 2-step method

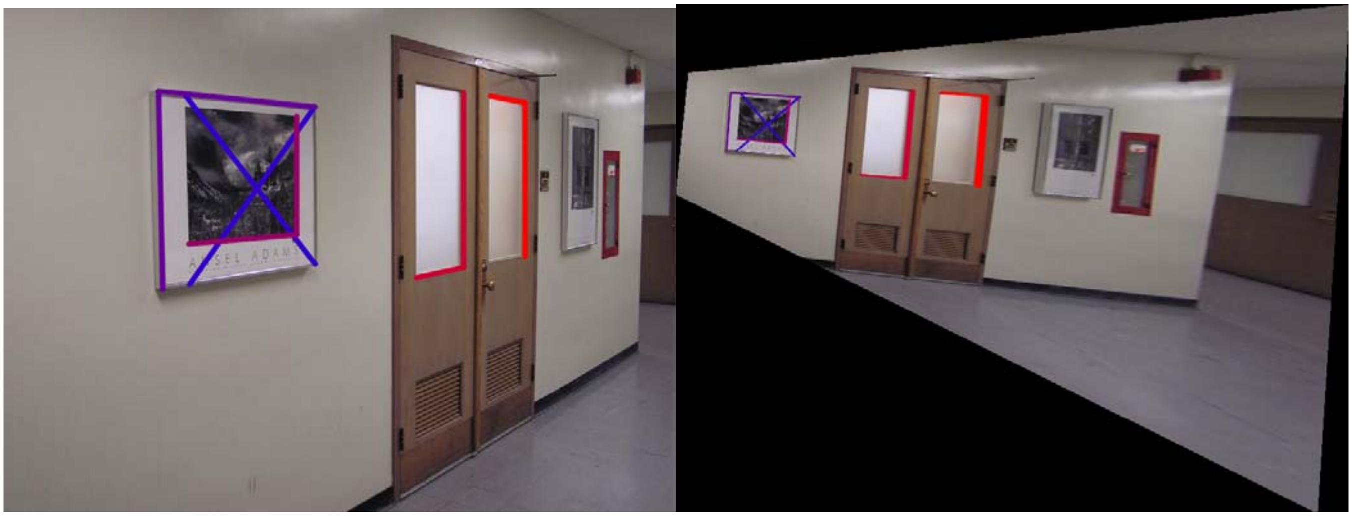
6 C^*_∞ Images

(a) Original with lines (b) Corrected image

Figure 10: adams01 using C^*_∞ method

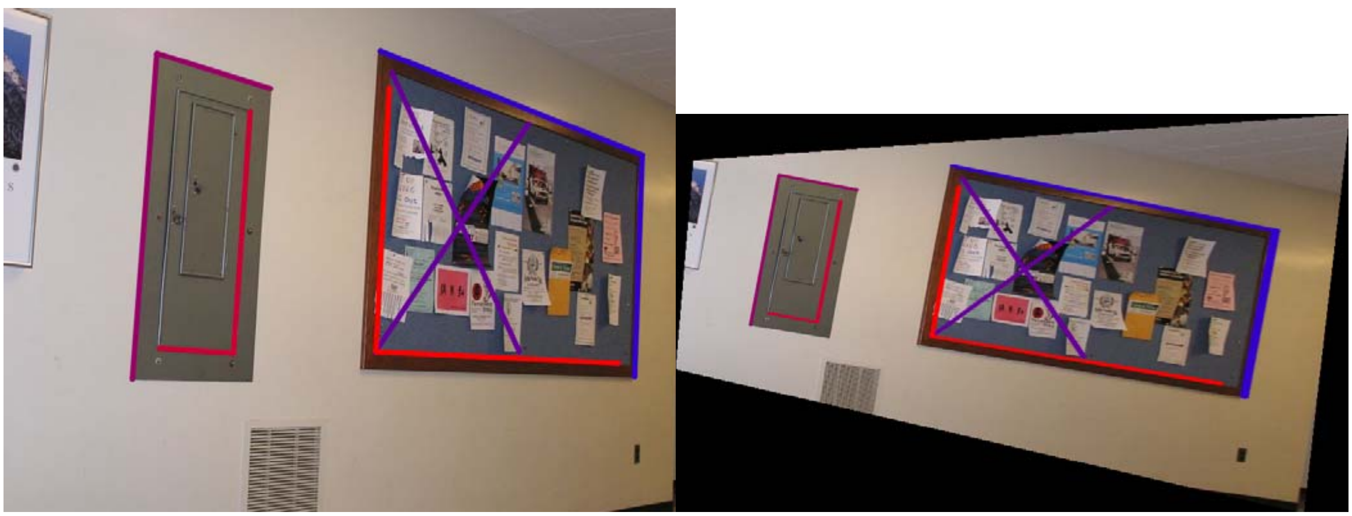
(a) Original with lines (b) Corrected image

Figure 11: adams02 using C^*_∞ method



(a) Original with lines (b) Corrected image

Figure 12: door01 using C^*_∞ method

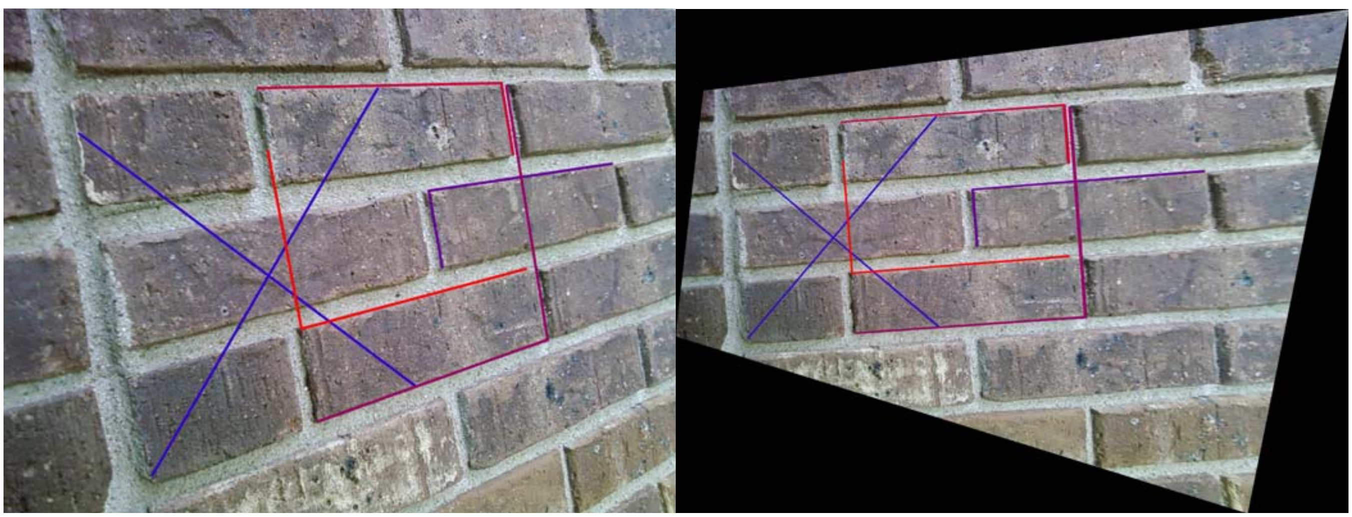


(a) Original with lines (b) Corrected image

Figure 13: board01 using C^*_∞ method

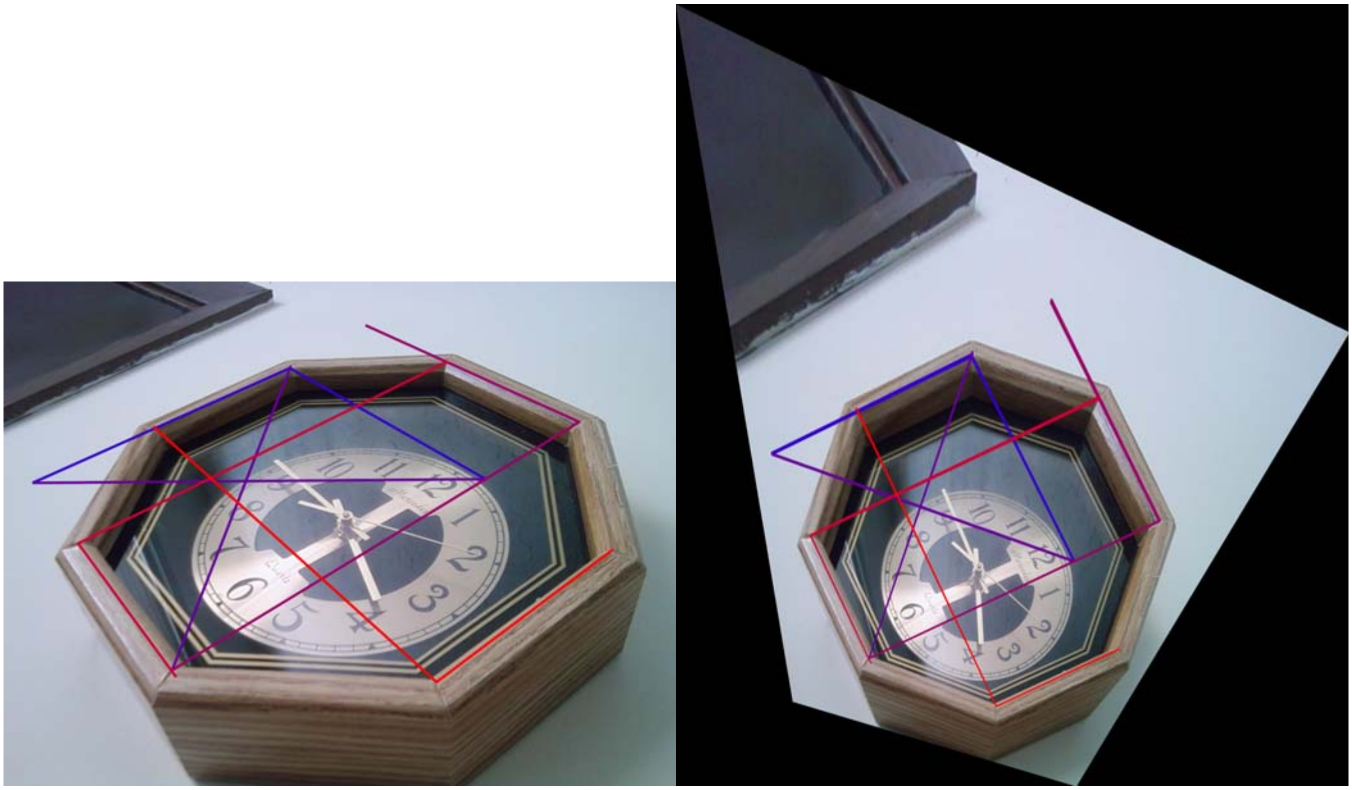
(a) Original with lines (b) Corrected image

Figure 14: textbook example using C_∞^* method



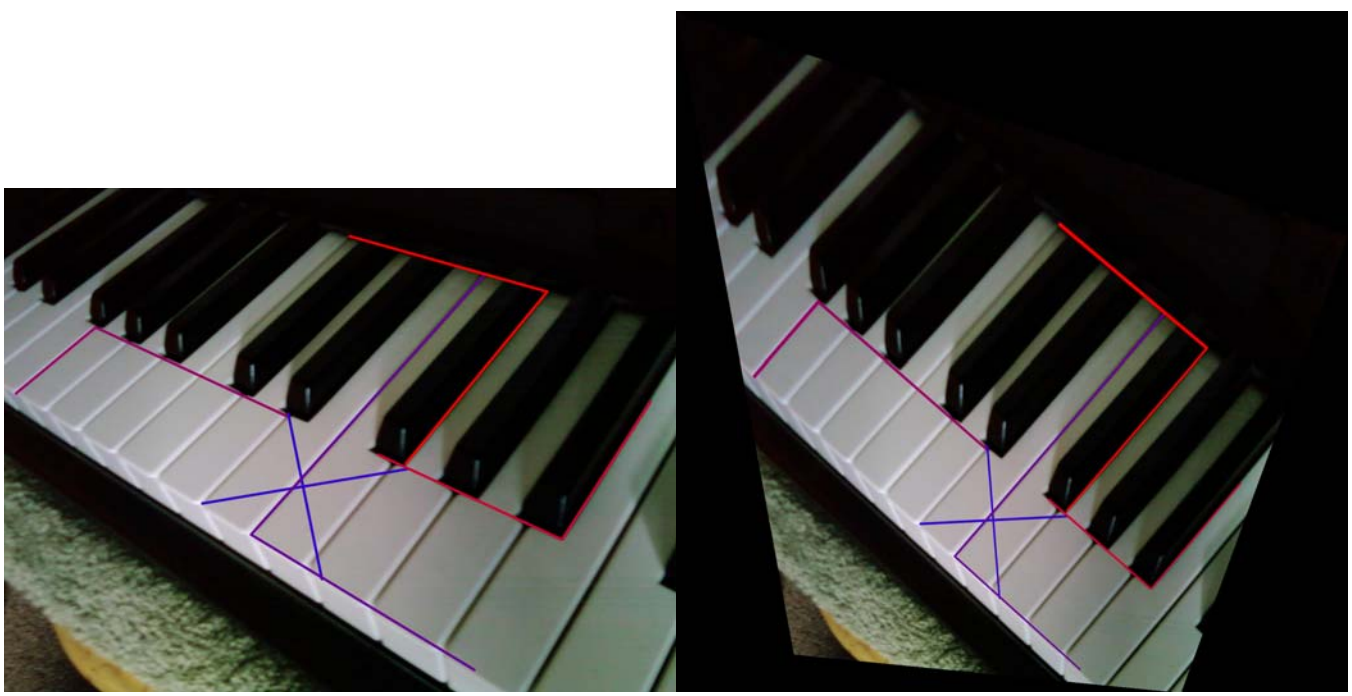
(a) Original with lines (b) Corrected image

Figure 15: bricks using C_∞^* method



(a) Original with lines (b) Corrected image

Figure 16: clock using C^*_∞ method



(a) Original with lines (b) Corrected image

Figure 17: piano using C^*_∞ method