Homography Correction

– Ajay Charan
UG201211002

OBJECTIVE

The objective of this homework is to eliminate the projective distortion in an image of a planar scene. Using the two-step method. First the projective distortion is removed using two sets of parallel lines. Once the projective distortion is removed, only the affine distortion remains. The affine distortion is removed using two sets of orthogonal lines.

SOLUTION

2-Step Method

In the 2-step method, we first remove the projective distortion from the image by using two sets of parallel lines. In all the example images, there exist several rectangular shapes. Thus we compute two sets of parallel lines from the points of a single rectangle. In the physical world, parallel lines intersect at the line at infinity, l_∞. In the image space, however, parallel lines intersect at finite points due to the projective deformation. The line joining the points of intersection of two sets of parallel lines is called the vanishing line, $l_v = (l_1, l_2, l_3)$. The image is corrected for projective distortion via the homography that sends the vanishing line l_v back to l_∞.

Once the projective distortion has been removed, we must then remove the affine distortion. To this end, we note that the angle θ between two lines l and m in the undistorted image can be expressed in terms of the projected lines l' and m'

We then invoke orthogonality of the lines selected. The orthogonality condition thus reduces to

$$(l'_1 m'_1, l'_1 m'_2 + l'_2 m'_1, l'_2 m'_2)s = 0,$$

where $s = (s_{11}, s_{12}, s_{22})^\top$. The matrix S is symmetric and homogeneous. Thus it has two degrees of freedom and two orthogonal line pairs can be used to compute the null vector s. After determining S, an SVD is used to determine K and subsequently H_A. Finally, the homography used to eliminate both projective and affine distortion is given by $H = H_P H_A$.

RESULTS

Fig. 1. Original

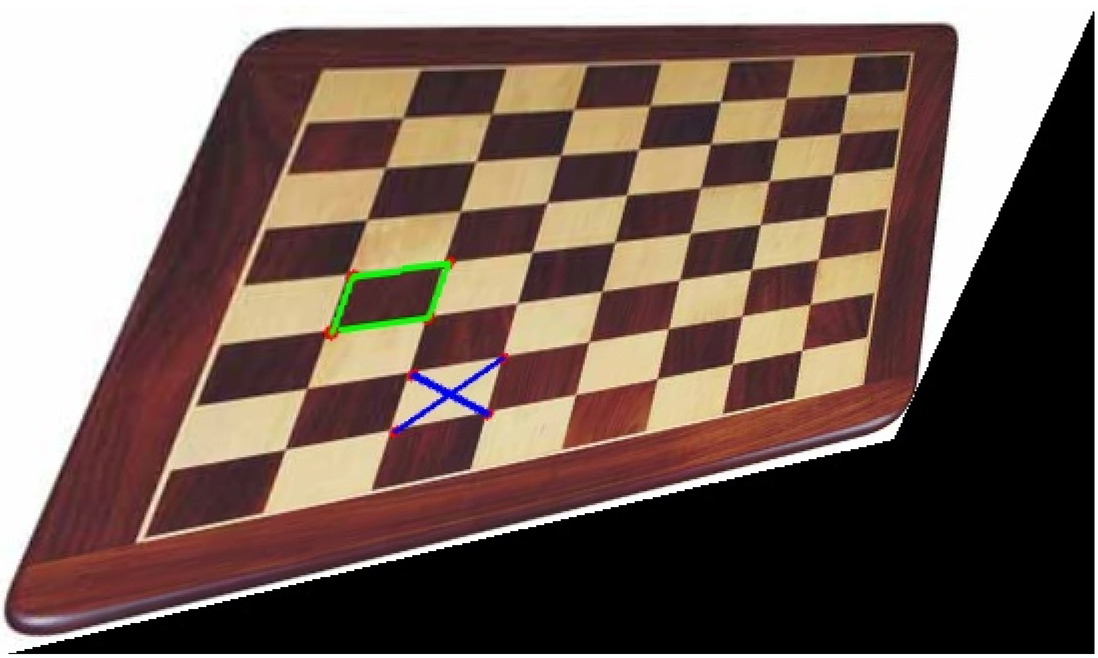

Fig. 2. Projective correction

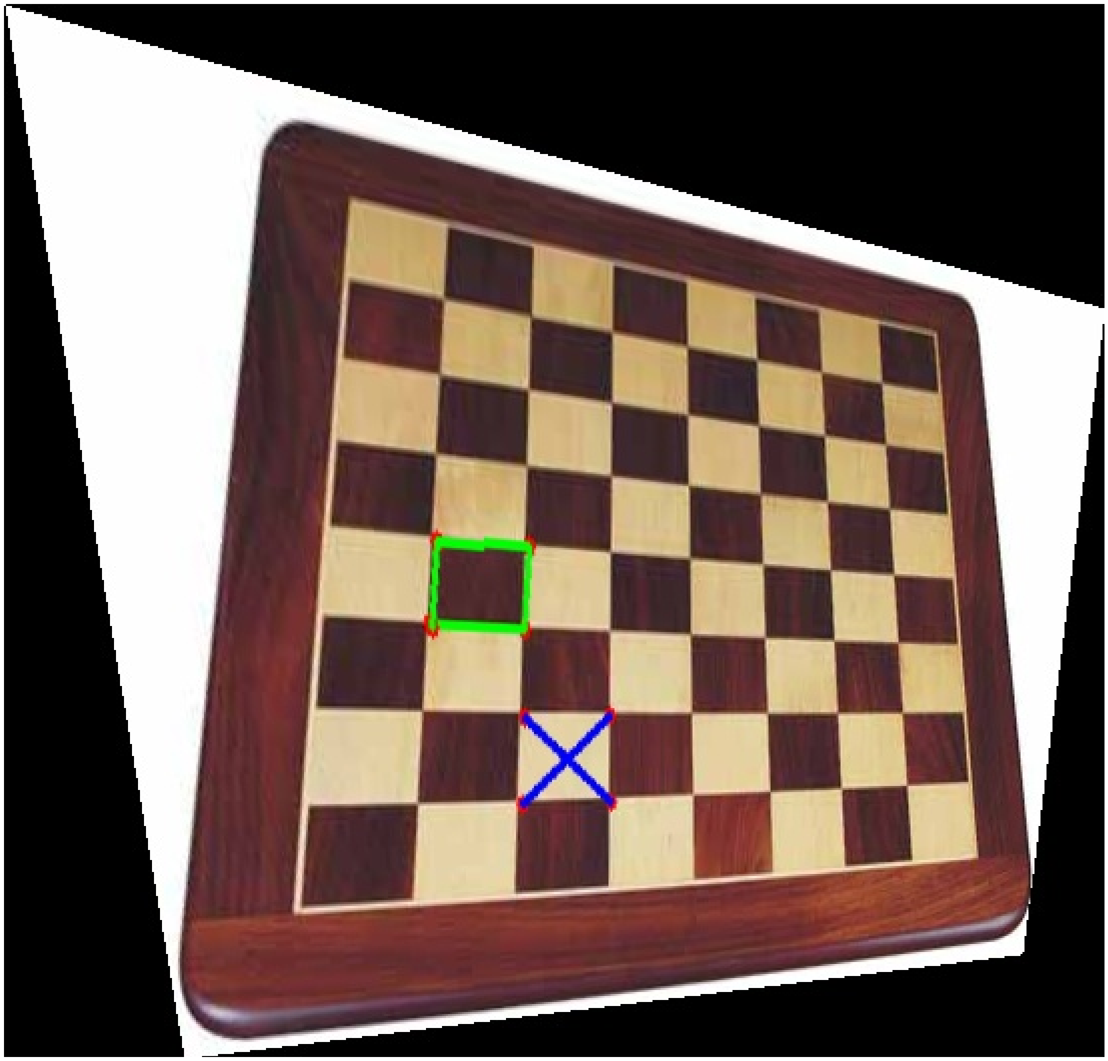

Fig. 3. Affine correction

Fig. 4. Original

Fig. 5. Projective correction

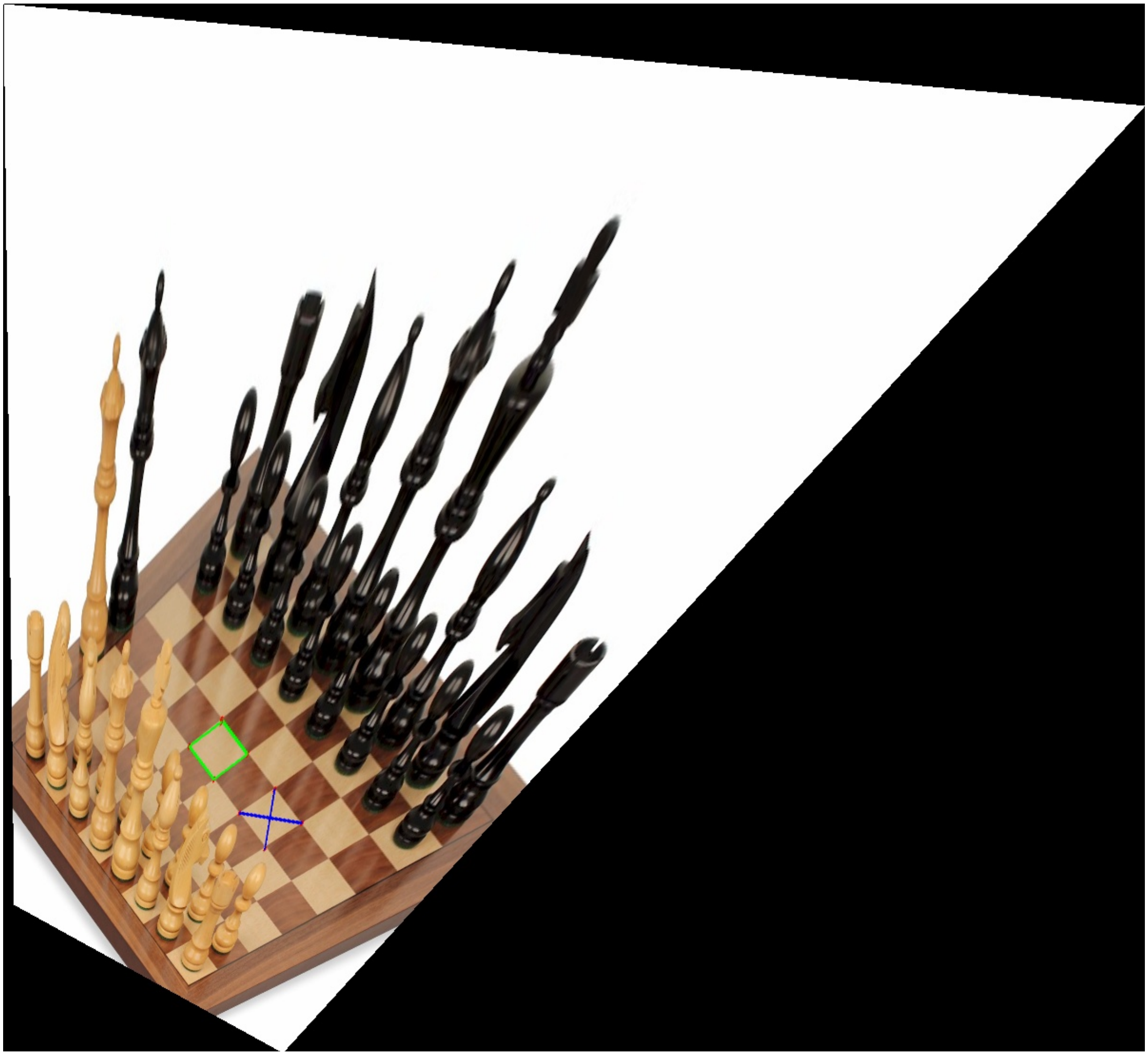

Fig. 6. Affine correction

Fig. 7. Original

Fig. 8. Projective correction

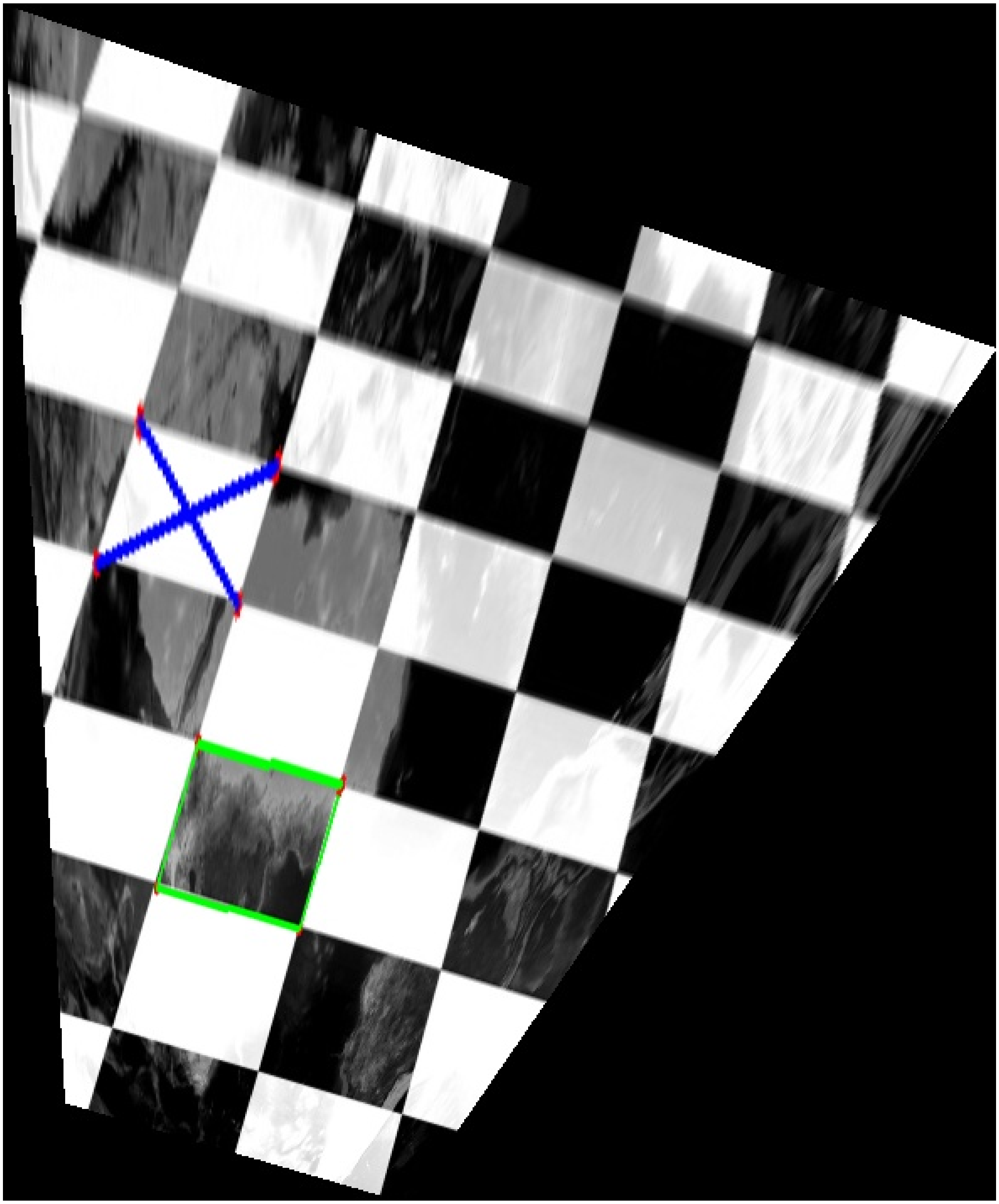

Fig. 9. Affine correction

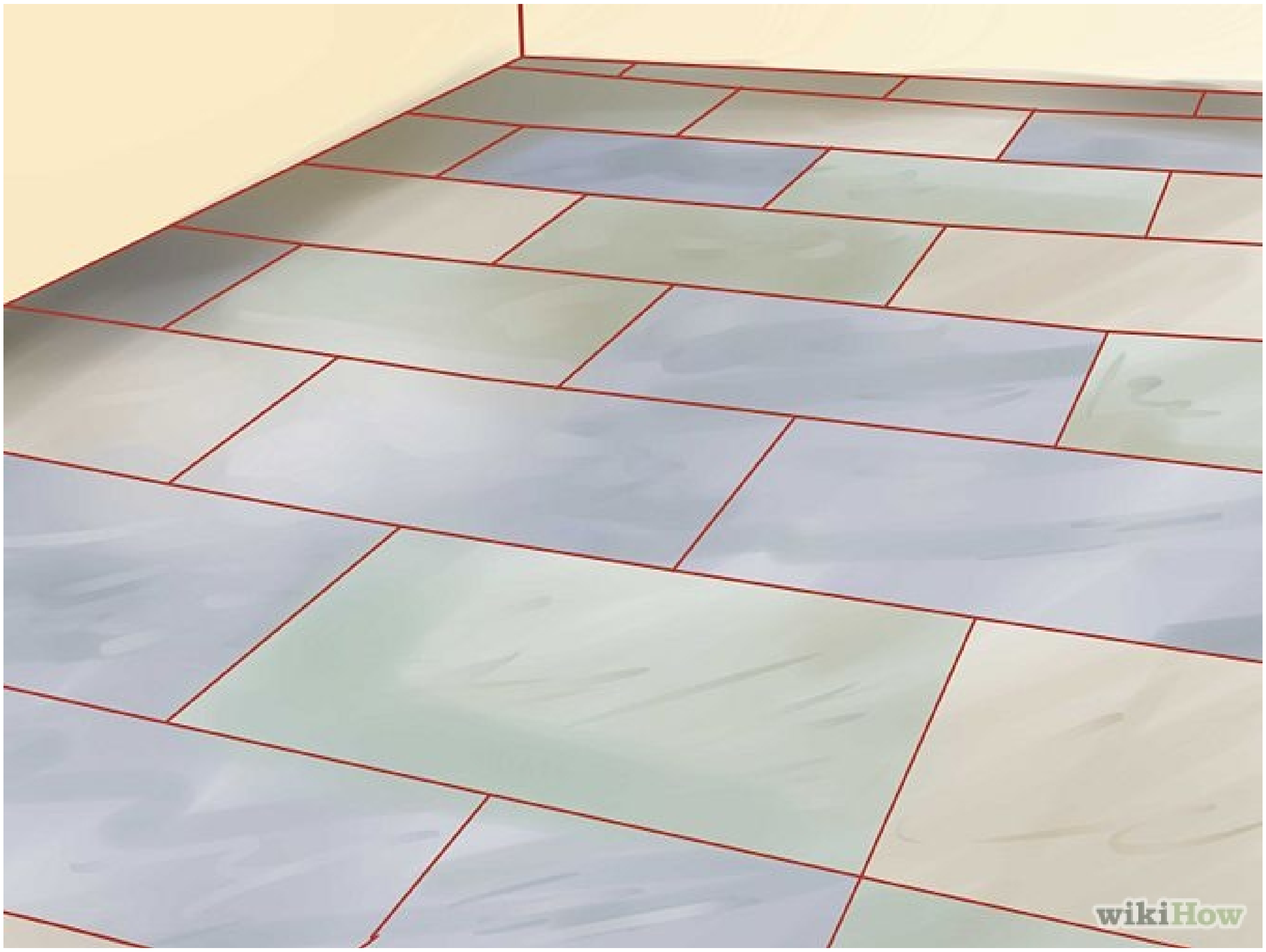

Fig. 10. Original

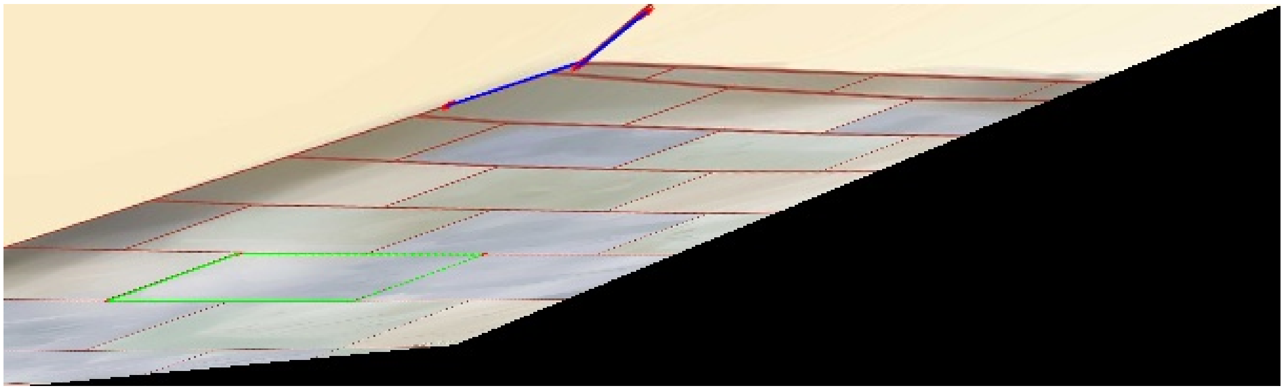

Fig. 11.   Projective correction

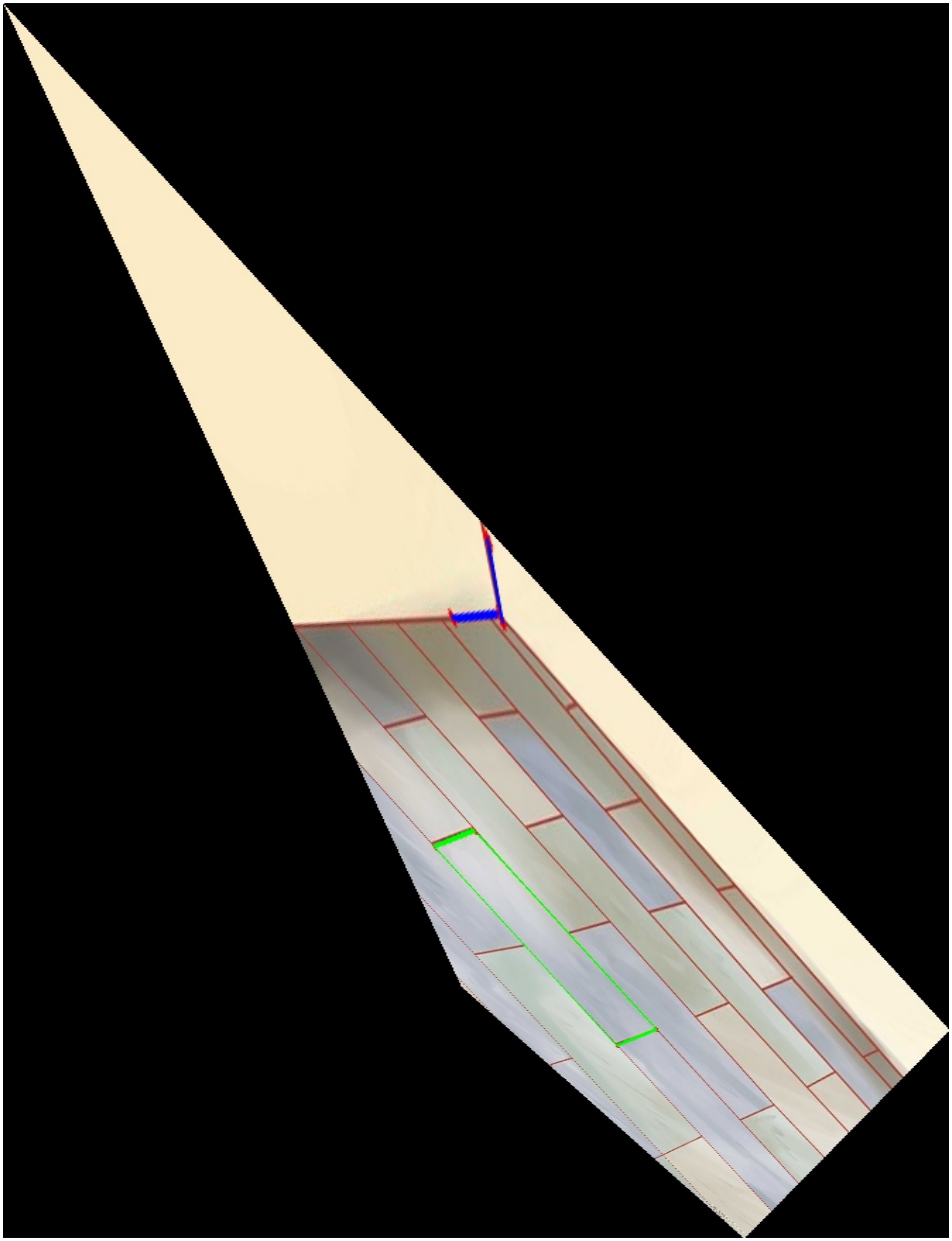

Fig. 12. Affine correction

Fig. 13. Original

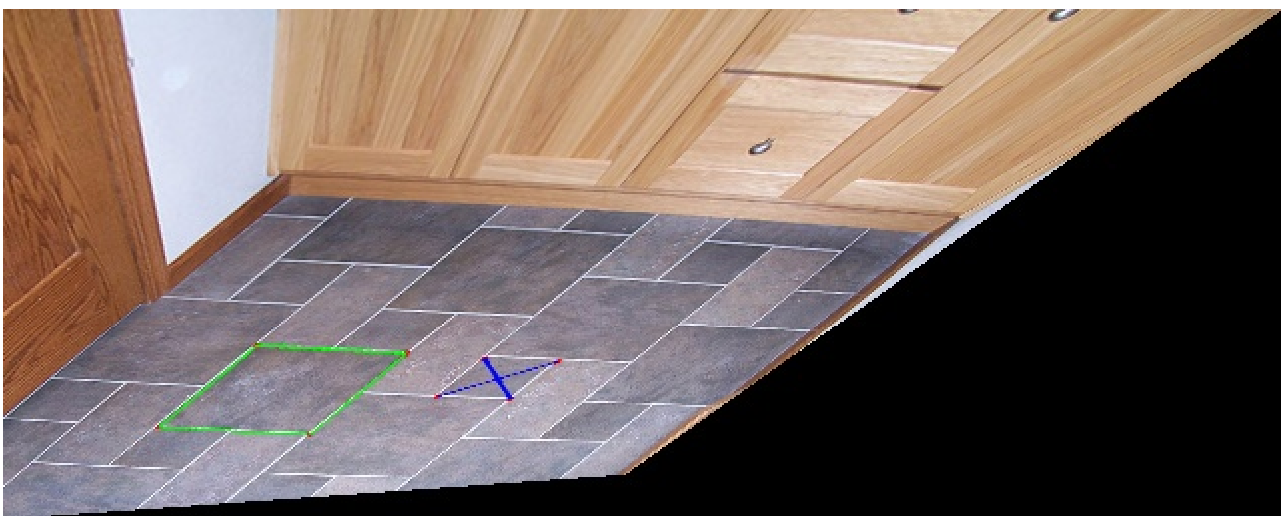

Fig. 14. Projective correction

Fig. 15. Affine correction

Fig. 16.   Original

Fig. 17. Projective correction

Fig. 18. Affine correction

C++ Code for 2-Step Method

```
/* A program to correct affine and projective distortions in given image
 *
 * Author: Ajay Charan <UG201211002 AT iitj.ac.in>
 * Requires OpenCV
 * Usage: APrect <input image>
 * Select 4  points from a rectangle in the world plane to form two sets
of parallel lines
 * Select 4 points forming two orthogonal lines in the world plane
 */

#include "opencv/cv.hpp"
#include "opencv2/highgui.hpp"
#include <iostream>
#include <string>

using namespace cv;
using namespace std;


const char* winDisp = "Select the points";

Mat imgIn;
vector<Point3d> pointsProjDist;
vector<Point3d> pointsAffDist;
int idxPoints = 0;

// Setup mouse callbacks to register points when user clicks
void on_mouse(int click, int x, int y, int /*flags*/, void*)
{
	if(imgIn.empty())
		{return;}

	if(click == CV_EVENT_LBUTTONUP)
	{
		idxPoints++;
		cout<< "Point: {"<<x<<","<<y<<"]"<<endl;
		// Draw circle on selected point location
	circle(imgIn, Point(x,y), 3, CV_RGB(255,0,0), -1);

		if(idxPoints<6)
		{
			pointsProjDist.push_back(Point3d(x,y,1));
			if(idxPoints>1)
			{
				Point3d p1 = pointsProjDist[idxPoints -2];
				Point3d p2 = pointsProjDist[idxPoints -1];
				// Draw line between points
				line(imgIn, Point2d(p1.x, p1.y),
```

```
Point2d(p2.x,p2.y), CV_RGB(0,0,255), 2);
                }
            }
            else if(idxPoints <10)
            {
                pointsAffDist.push_back(Point3d(x,y,1));
                if(idxPoints % 2 == 1)
                {
                    Point3d p1 = pointsAffDist[idxPoints -7];
                    Point3d p2 = pointsAffDist[idxPoints -6];
                    // Draw lines joining points
                    line(imgIn, Point2d(p1.x, p1.y), Point2d(p2.x,
p2.y), CV_RGB(0,0,255), 2);
                }
            }
            imshow(winDisp, imgIn);
        }
}

// Correct input image given a homography
Mat correct_distortion(Mat imgIn, Mat H, string filename, string add)
{

        int idxRow, idxCol;
        // Create vector with boundaries in image plane
        vector<Point3d> coords_img;
        coords_img.push_back(Point3d(0, 0, 1));
        coords_img.push_back(Point3d(0, imgIn.rows-1, 1));
        coords_img.push_back(Point3d(imgIn.cols-1, 0, 1));
        coords_img.push_back(Point3d(imgIn.cols-1, imgIn.rows-1, 1));

        cout<<"coords_img = "<< coords_img<<endl;
        // Compute corresponding coordinates in world plane
        Mat M_coords_img = Mat(coords_img, true);
        M_coords_img = M_coords_img.reshape(1,4);
        cout << "M_coords_img t = " << M_coords_img.t() << endl;


        Mat M_coords_wld = H.inv() * M_coords_img.t();
        cout << "M_coords_world = " << M_coords_wld << endl;

        M_coords_wld.row(0) = M_coords_wld.row(0).mul(1 /
M_coords_wld.row(2));
        M_coords_wld.row(1)= M_coords_wld.row(1).mul(1 /
M_coords_wld.row(2));

        cout << "M_coords_world normalized = " << M_coords_wld << endl;

// Get minimum and maximum coordinates
        double xmin, xmax, ymin, ymax;
        minMaxLoc(M_coords_wld.row(0), &xmin, &xmax, 0, 0);
        minMaxLoc(M_coords_wld.row(1), &ymin, &ymax, 0, 0);
        cout << "xmin = " << xmin << endl;
        cout << "xmax = " << xmax << endl;
```

```
      cout << "ymin = " << ymin << endl;
      cout << "ymax = " << ymax << endl;

// Determine size of corrected image, keeping image width same
      double scale = imgIn.cols / (xmax - xmin);
      double height_out = (int) ((ymax - ymin) * scale);

      cout << "scale = " << scale << endl;
      cout << "height_out = " << height_out << endl;

// Create corrected image (output image)
      Mat image_out(height_out, imgIn.cols, CV_8UC3);

// Find values for corrected image by taking set of coordinates from
world plane to image plane and interpolating
// Create temporary world and image coordinates
      Mat coords_wld_temp(3, 1, CV_64FC1);
      Mat coords_img_temp(3, 1, CV_64FC1);

      coords_wld_temp.at<double> (2, 0) = 1; // Set third component to 1
      double step=1/ scale;
      for (idxCol = 0; idxCol < image_out.cols; idxCol++)
      {
      // Set x coordinate
            coords_wld_temp.at<double> (0, 0) = (double) idxCol * step+
xmin;
            for (idxRow = 0; idxRow < image_out.rows; idxRow++)
            {
                  double x, y, dx, dy;
                  // Set y coordinate
                  coords_wld_temp.at<double> (1, 0) = (double) idxRow *
step+ ymin;
            coords_img_temp=H*coords_wld_temp;
                  // Normalize wrt third coordinate
                  x = coords_img_temp.at<double> (0, 0)
/coords_img_temp.at<double> (2, 0);
                  y = coords_img_temp.at<double> (1, 0) /
coords_img_temp.at<double> (2, 0);
                  if (x < 0 || x > imgIn.cols - 1 || y < 0 || y >
imgIn.rows - 1)
                  {
                        continue;
                  }
                  // Take decimal part of x and y coordinates
                  dx = x - (int) x;
                  dy = y - (int) y;
                  Vec3b i00 = imgIn.at<Vec3b> (int(y), int(x));
                  if (dx != 0.0 || dy != 0.0)
                  {
                        Vec3b i10 = imgIn.at<Vec3b> (int(y), int(x + 1));
                        Vec3b i01 = imgIn.at<Vec3b> (int(y + 1), int(x));
                        Vec3b i11 = imgIn.at<Vec3b> (int(y + 1), int(x +
1));
                        image_out.at<Vec3b> (idxRow, idxCol) = i00 *(1 -
```

```
dx)*(1 - dy) + i10 * dx * (1 - dy) + i01 * (1 - dx) * dy+ i11 * dx * dy;
                }
                else
                {
                     image_out.at<Vec3b> (idxRow, idxCol) = i00;
                }
           }
     }

     cout << "Saving image..." << endl;
     // Save corrected image
     int lastindex = filename.find_last_of(".");
     string rawname = filename.substr(0, lastindex);
     imwrite(rawname.append("_").append(add).append(".jpg"), image_out);
     return image_out;
}

Point3d normPoint3d(Point3d p)
{
     p.x = p.x / p.z;
     p.y = p.y / p.z;
     p.z = 1;
     return p;
}

Point3d mat2point3d(Mat p)
{
     Point3d p3d = Point3d(p.at<double> (0), p.at<double> (1),
p.at<double> (2));
     return p3d;
}



Mat homography_projective()
{
// Compute vanishing line
     Point3d l1 = pointsProjDist[0].cross(pointsProjDist[1]);
     Point3d l2 = pointsProjDist[3].cross(pointsProjDist[2]);
     Point3d l3 = pointsProjDist[1].cross(pointsProjDist[2]);
     Point3d l4 = pointsProjDist[4].cross(pointsProjDist[3]);
     Point3d P = l1.cross(l2);
     Point3d Q = l3.cross(l4);
     Point3d VL = P.cross(Q);
// Normalize vanishing line
     VL = normPoint3d(VL);

     cout << "Vanishing Line: " << VL << endl;
// Build homography matrix to correct projective distortion
     Mat Hp = Mat::eye(3, 3, CV_64FC1);
     Hp.at<double> (2, 0) = VL.x;
     Hp.at<double> (2, 1) = VL.y;
     Hp.at<double> (2, 2) = VL.z;
     cout << "Hp = " << Hp << endl;
```

```
    return Hp;
}

Mat homography_affine(Mat Hp)
{

// Compute two sets of orthogonal lines
    Point3d l = pointsAffDist[0].cross(pointsAffDist[1]);
    Point3d m = pointsAffDist[2].cross(pointsAffDist[3]);
    Point3d n = pointsProjDist[0].cross(pointsProjDist[1]);
    Point3d o = pointsProjDist[1].cross(pointsProjDist[2]);
// Correct projective distortion of original lines
    Mat lmat = Hp.inv().t() * Mat(l, true);
    Mat mmat = Hp.inv().t() * Mat(m, true);
    Mat nmat = Hp.inv().t() * Mat(n, true);
    Mat omat = Hp.inv().t() * Mat(o, true);
// Convert Mat back to Point3d
    l = mat2point3d(lmat);
    m = mat2point3d(mmat);
    n = mat2point3d(nmat);
    o = mat2point3d(omat);
    cout << "l = " << l << endl;
    cout << "m = " << m << endl;
    cout << "n = " << n << endl;
    cout << "o = " << o << endl;

    double mldata[2][2] = { { l.x * m.x, l.x * m.y + l.y * m.x }, { n.x
* o.x,n.x * o.y + n.y * o.x } };
    double bdata[2][1]={{ -l.y * m.y }, { -n.y * o.y } };
    Mat ML = Mat(2, 2, CV_64FC1, mldata);
    Mat b = Mat(2, 1, CV_64FC1, bdata);

    cout << "ML = " << ML << endl;
    cout << "b = " << b << endl;


    Mat s = ML.inv() * b;
    cout << "s = " << s << endl;
// Build S matrix
    double Sdata[2][2]={{ s.at<double> (0), s.at<double> (1) }, { s.at<
    double> (1), 1 } };
    Mat S = Mat(2, 2, CV_64FC1, Sdata);
    cout << "S = " << S << endl;

// Compute SVD of S
    Mat U, D2, D, Ut;
    SVD::compute(S, D2, U, Ut, 0);
    cout << "U = " << U << endl;
    cout << "Ut = " << Ut << endl;
    cout << "D2 = " << D2 << endl;
// Find D from D^2
    pow(D2, 0.5, D);
    D = Mat::diag(D);
    cout << "D = " << D << endl;
```

```
// Build A
      Mat A = U * D * U.inv();
      cout << "A = " << A << endl;
// Build homography matrix to correct affine distortion
      Mat Ha;
      double Acdata[2][1] = { { 0 }, { 0 } };
      Mat Ac = Mat(2, 1, CV_64FC1, Acdata);
      hconcat(A, Ac, Ha);

      double Ardata[1][3] = { { 0, 0, 1 } };
      Mat Ar = Mat(1, 3, CV_64FC1, Ardata);
      vconcat(Ha, Ar, Ha);
      cout << "Ha = " << Ha << endl;
      return Ha;
}

// Main function
int main(int argc, char** argv)
{
      cout << "Use:" << endl << " left click to add new points;"<< endl
<< "Press key 'r'  to run the program" << endl << endl;

// Load input image
      if (argc < 2)
      {
            cout << " Usage: APrect image_path" << endl;
            return -1;
      }
      else
      {
            imgIn = imread(argv[1], CV_LOAD_IMAGE_COLOR);
            if (!imgIn.data) // Check for invalid input
            {
                  cout << "Could not open or find the image" << endl;
                  return -1;
            }
      }
      // Show image in external window
      namedWindow(winDisp, CV_WINDOW_AUTOSIZE);
      imshow(winDisp, imgIn);
      // Set mouse callback (to get points in image from user clicks)
      cvSetMouseCallback(winDisp, on_mouse);
      cout << "1) Select 4 points from a rectangle in the world plane to
form"<< "two sets of parallel lines" << endl<< "2) Select 4 points
forming two orthogonal lines in the world plane"<< endl;
      for (;;)
      {
            // Wait for user to press key
            uchar key = (uchar) waitKey();
            // Exit if escape key pressed
            if (key == 27)
                  break;

            // Run program if ’r’ key pressed
```

```
            if (key == 'r')
            {
            // Compute homography for removing projective distortion
                  Mat Hp = homography_projective();
            // Correct projective distortion
                  correct_distortion(imgIn, Hp.inv(), argv[1], "proj");
            // Compute homography for removing affine distorion
                  Mat Ha = homography_affine(Hp);
            // Correct affine distortion
                  correct_distortion(imgIn, Hp.inv() * Ha, argv[1], 
"affine");
            cout<<"aal is well"<<endl;
                  return 0;
            }
      }
}
```