ECE661 Computer Vision Homework 8

Hidekazu Iwaki

November 20, 2008

1 Labeling Corner

I used the following steps to find and label corners:

1. **Canny Edge Detection:** I used cvCanny with $t1 = 500$, $t2 = 300$ for each original image. I will show some example of edge images in Fig.1b.
2. **Hough Transformation:** I used cvHoughLine2 with $t1 = 50$ for each edge image. I will show some example of detected lines as green lines in Fig.1c.
3. **Line Grouping:** I integrated similar detected lines in each image by using steps as the following:
   (a) I divied lines to two sets $\mathcal{H}$ and $\mathcal{V}$ which are almost perpendicular.
   (b) I measured the similarity between two lines in each set $\mathcal{H}$ or $\mathcal{V}$ using $d = \rho_r - \rho\cos(\theta - \theta_r)$. Where ρ and ρ_r are the distances of reference line and current line from origin, and θ and θ_r are the angles of their lines. I assumed a square have L [pixels] of size in each image. If $d < L/3$ then their two lines are regard as belonging to a same group.

   I will show some example of merged lines as red lines in Fig.1d.
4. **Label Corners as Intersection of lines:** I ordered merged lines in horizontal and vertical sets by using measurement $\rho_r - \rho\cos(\theta - \theta_r)$ in each image. Then intersections of their lines are labeled based on the order of lines.
5. **Refine the locations of corners to move to closest Harris corners:** To refine the coordinate of corners, I moved their corners to the closest Harris corners in each image. I will show some example of refined corners with labels as green crosses in Fig.2.

2 Camera Calibration

I used the following steps same as Zhang's papaer to calibrate Camera:

1. **Compute homograph matrices:** We obtain the relationship${}_wH_i$ $(i = 1, 2, ..., m)$ between the world coordinate defined on a pattern and all m images.
2. **Compute the intrinsic parameter matrix** K.
3. **Compute the extrinsic parameters** $\{{}_wR_i, {}_w\mathbf{t}_i\}_{i=1}^m$.
4. **Compute radial distortion parameter** k_1 **and** k_2**.**
5. **Refine all of above parameters** K**,** $\{{}_wR_i, {}_w\mathbf{t}_i\}_{i=1}^m$**,** k_1 **and** k_2 : I used Levenberg-Marquardt nonlinear least squares algorithms for the refinement.

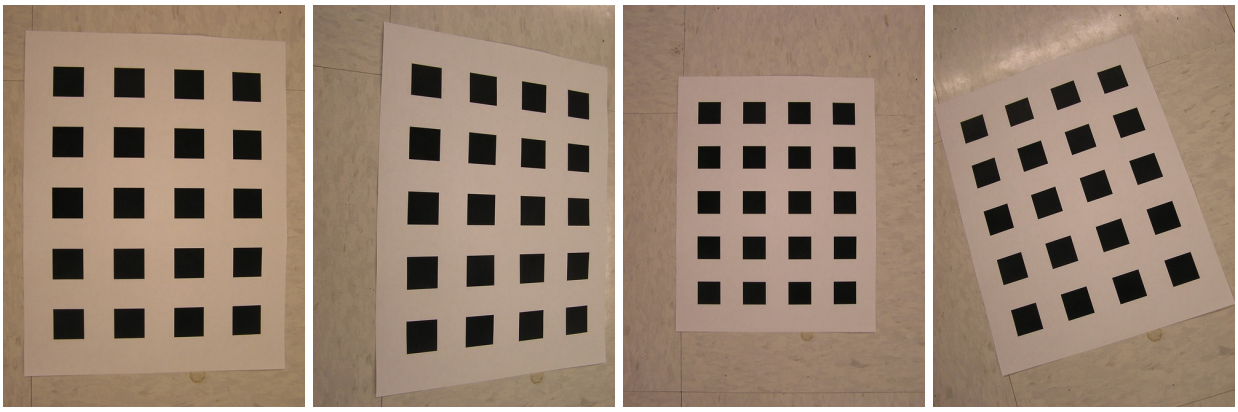
I will show reprojected points by using parameters before refinement on Fig.3 as yellow crosses, and after refinement on Fig.4 as red crosses.

3 Experimental Result.

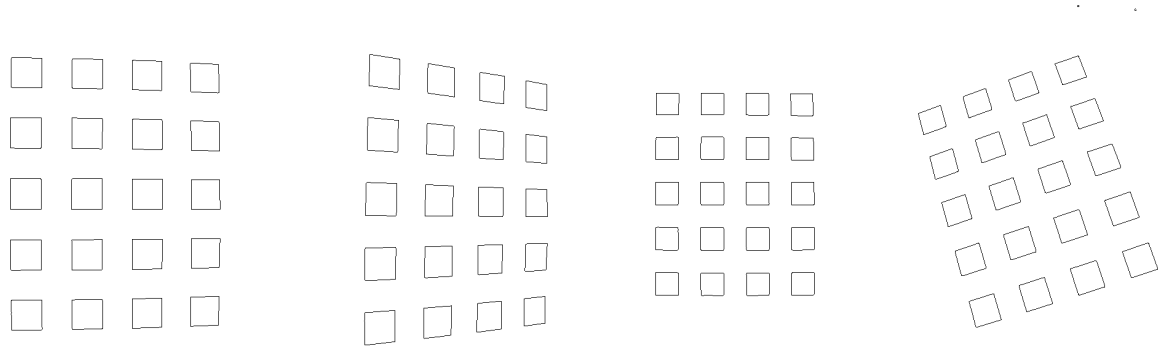
I calibrated cameras by using two image sets, A which is given by TA and B which is taken by my self as shown on Fig. 5 and Fig. 6. The number of images of A and B are 40 and 11. I will show results of the calibrations by using those two image sets on Table 1.

	Image set A	Image set B
# of Images	40	11
L	30	60
Initiall estimated K	$\begin{pmatrix} 755.490 & -1.563 & 314.533 \\ 0.000 & 766.834 & 260.540 \\ 0.000 & 0.000 & 1.000 \end{pmatrix}$	$\begin{pmatrix} 807.953 & 11.938 & 309.329 \\ 0.000 & 811.679 & 298.601 \\ 0.000 & 0.000 & 1.000 \end{pmatrix}$
Initiall estimated R_i (only for first 10 images)	$\begin{pmatrix} 0.048 & 0.029 & 0.011 \\ 0.045 & 0.266 & 0.010 \\ 0.064 & -0.125 & 0.164 \\ 0.081 & -0.347 & 0.066 \\ 0.064 & -0.154 & -0.093 \\ 0.063 & -0.141 & -0.008 \\ 0.038 & 0.371 & 0.114 \\ 0.211 & 0.479 & 0.064 \\ 0.078 & 0.607 & 0.038 \\ 0.032 & 0.662 & 0.211 \\ & \vdots & \end{pmatrix}$	$\begin{pmatrix} -0.019 & -0.514 & 0.037 \\ -0.015 & -0.251 & 0.014 \\ -0.014 & 0.203 & 0.010 \\ 0.229 & -0.020 & 0.050 \\ -0.190 & -0.024 & -0.008 \\ -0.034 & -0.017 & 0.006 \\ 0.002 & -0.374 & 0.006 \\ -0.030 & 0.250 & -0.016 \\ -0.314 & -0.001 & -0.031 \\ -0.001 & -0.106 & 0.011 \\ & \vdots & \end{pmatrix}$
Initiall estimated T_i (only for first 10 images)	$\begin{pmatrix} -7.638 & -11.638 & 55.569 \\ -7.605 & -11.540 & 54.407 \\ -4.945 & -12.453 & 58.246 \\ -5.981 & -11.242 & 55.404 \\ -7.127 & -11.290 & 52.916 \\ -5.708 & -10.718 & 59.208 \\ -6.535 & -14.147 & 61.980 \\ -4.378 & -12.598 & 55.057 \\ -4.725 & -12.669 & 59.783 \\ -4.158 & -14.588 & 57.528 \\ & \vdots & \end{pmatrix}$	$\begin{pmatrix} -7.040 & -9.825 & 38.841 \\ -11.121 & -10.576 & 41.526 \\ -12.669 & -8.711 & 46.863 \\ -11.259 & -9.492 & 43.262 \\ -12.262 & -10.743 & 47.789 \\ -12.878 & -9.767 & 56.275 \\ -10.058 & -9.291 & 53.376 \\ -13.443 & -8.310 & 59.412 \\ -13.175 & -10.305 & 64.766 \\ -11.582 & -10.222 & 41.521 \\ & \vdots & \end{pmatrix}$
Initiall estimated $k_1$and k_2	$0.308\ and\ -1.178$	$0.070\ and\ 0.083$
Reprojection Error of Initially estimated paramaters [pixels]	6.83637	5.39946
Refined K	$\begin{pmatrix} 730.743 & -1.563 & 324.863 \\ 0.000 & 732.498 & 242.538 \\ 0.000 & 0.000 & 1.000 \end{pmatrix}$	$\begin{pmatrix} 753.211 & 11.938 & 255.569 \\ 0.000 & 753.136 & 332.775 \\ 0.000 & 0.000 & 1.000 \end{pmatrix}$
RefinedR_i (only for first 10 images)	$\begin{pmatrix} 0.037 & 0.036 & 0.012 \\ 0.037 & 0.261 & 0.008 \\ 0.046 & -0.114 & 0.167 \\ 0.055 & -0.341 & 0.072 \\ 0.051 & -0.125 & -0.090 \\ 0.039 & -0.115 & -0.005 \\ 0.044 & 0.356 & 0.112 \\ 0.200 & 0.476 & 0.061 \\ 0.075 & 0.601 & 0.033 \\ 0.034 & 0.635 & 0.207 \\ & \vdots & \end{pmatrix}$	$\begin{pmatrix} -0.017 & -0.445 & 0.019 \\ -0.010 & -0.232 & 0.001 \\ -0.015 & 0.183 & 0.007 \\ 0.219 & -0.017 & 0.035 \\ -0.162 & -0.030 & -0.010 \\ -0.025 & -0.021 & -0.001 \\ 0.002 & -0.325 & -0.009 \\ -0.022 & 0.227 & -0.017 \\ -0.278 & -0.008 & -0.028 \\ 0.003 & -0.102 & 0.001 \\ & \vdots & \end{pmatrix}$
Refined T_i (only for first 10 images)	$\begin{pmatrix} -8.364 & -10.453 & 53.396 \\ -8.302 & -10.384 & 52.141 \\ -5.725 & -10.922 & 56.362 \\ -6.753 & -10.095 & 53.753 \\ -7.870 & -10.164 & 51.391 \\ -6.524 & -9.502 & 57.557 \\ -7.363 & -12.735 & 59.538 \\ -5.085 & -11.374 & 52.643 \\ -5.494 & -11.360 & 57.261 \\ -4.981 & -13.028 & 55.939 \\ & \vdots & \end{pmatrix}$	$\begin{pmatrix} -4.612 & -11.535 & 35.822 \\ -8.404 & -12.249 & 37.571 \\ -9.840 & -10.932 & 43.866 \\ -8.498 & -11.334 & 39.655 \\ -9.216 & -12.799 & 43.698 \\ -9.253 & -12.170 & 51.791 \\ -6.598 & -11.522 & 49.066 \\ -9.828 & -11.123 & 56.131 \\ -9.008 & -13.096 & 59.766 \\ -8.897 & -11.944 & 37.740 \\ & \vdots & \end{pmatrix}$
Refined $k_1$and k_2	$-0.243\ and\ 0.443$	$-0.228\ and\ 0.254$
Reprojection Error of I Refined paramaters [pixels]	1.02976	0.959057

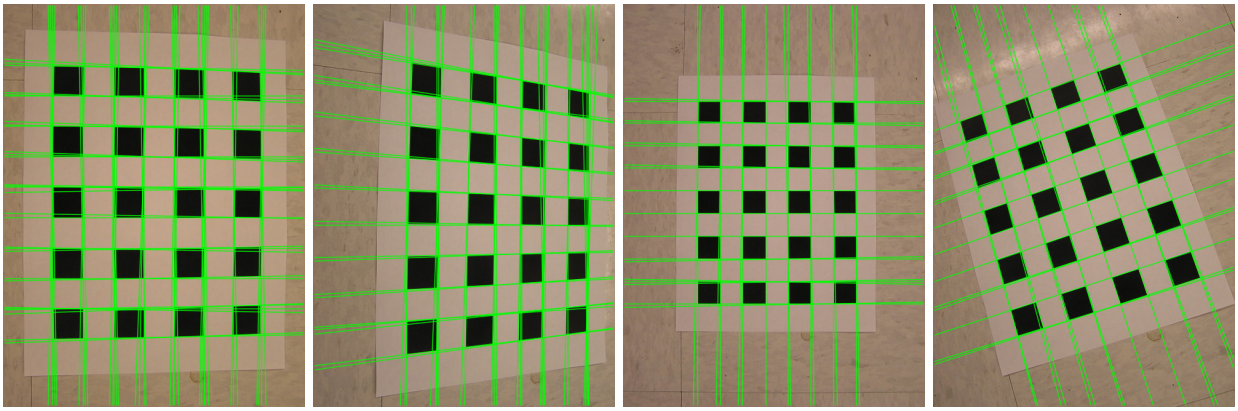
Table 1: Experimental result



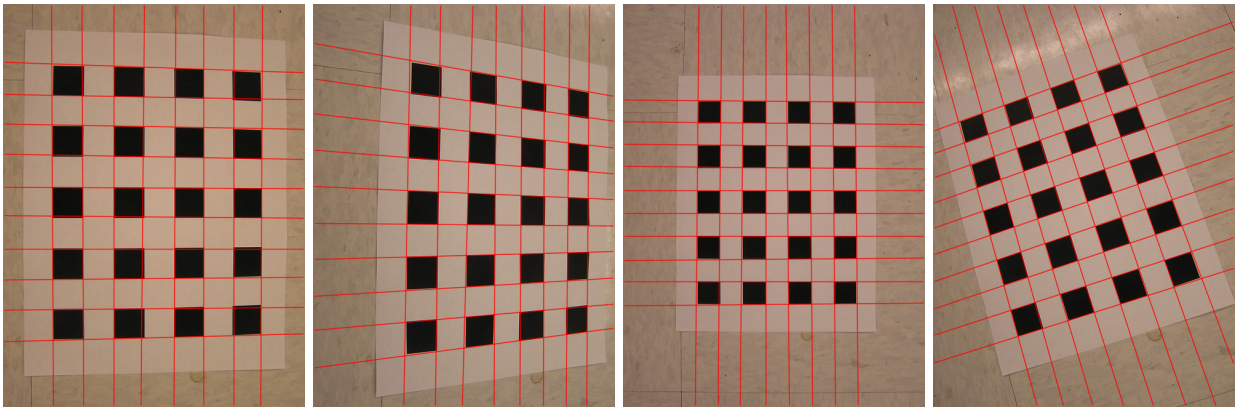
(a) Original Images



(b) Canny Edges



(c) Hough Lines



(d) Merged Lines

Figure 1: Line Extraction

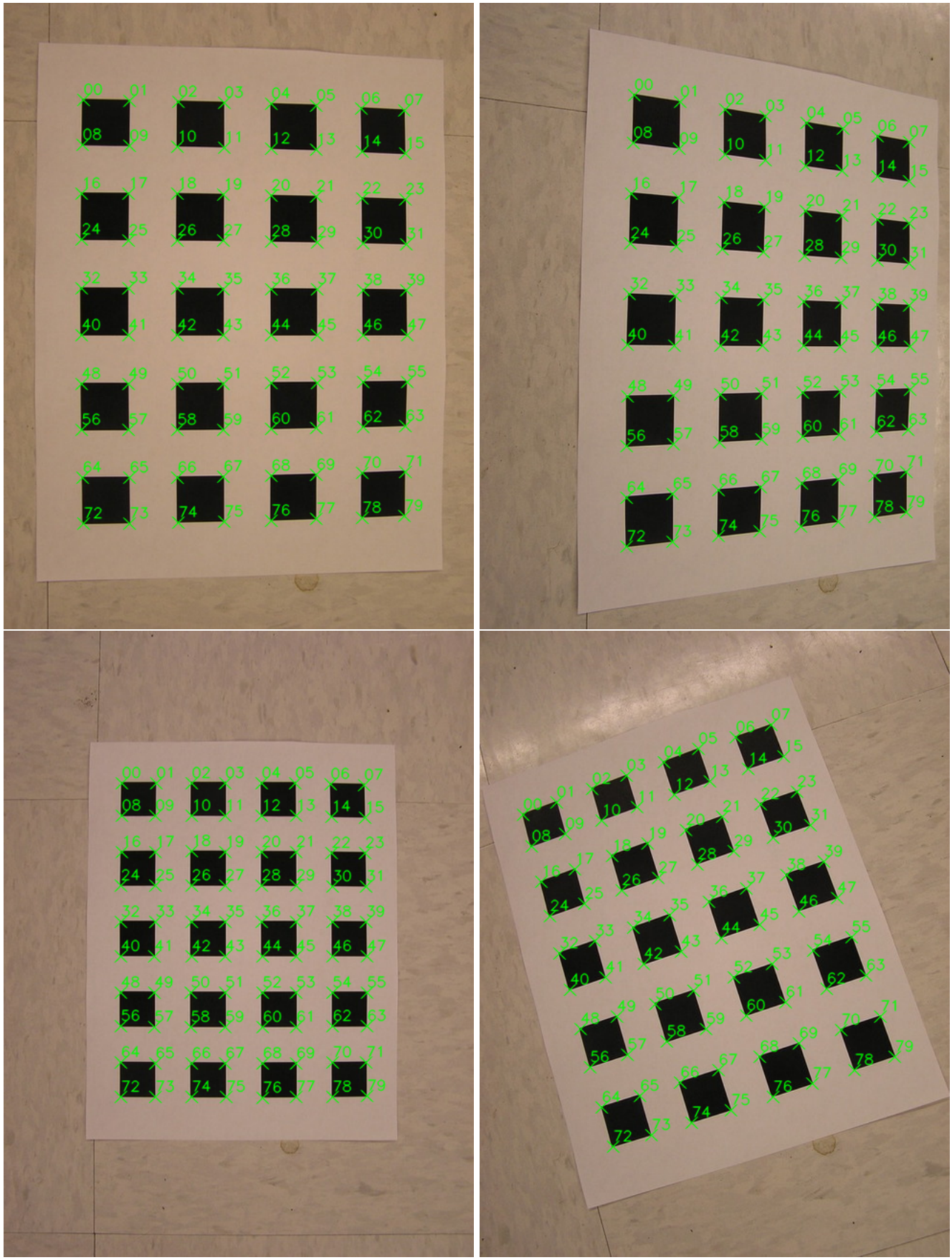


Figure 2: Labeled Corners

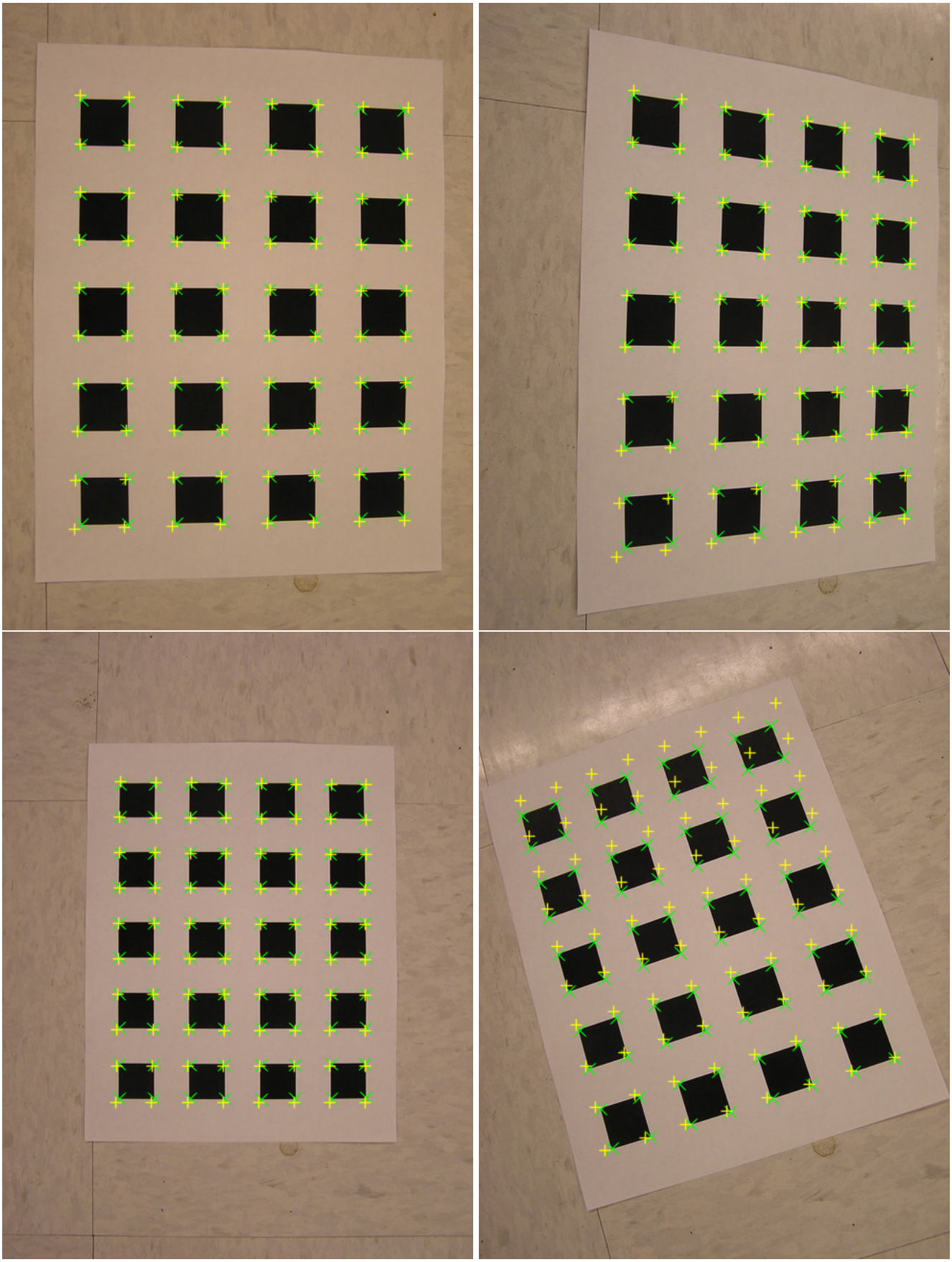


Figure 3: Reprojected points by initially estimated parameters

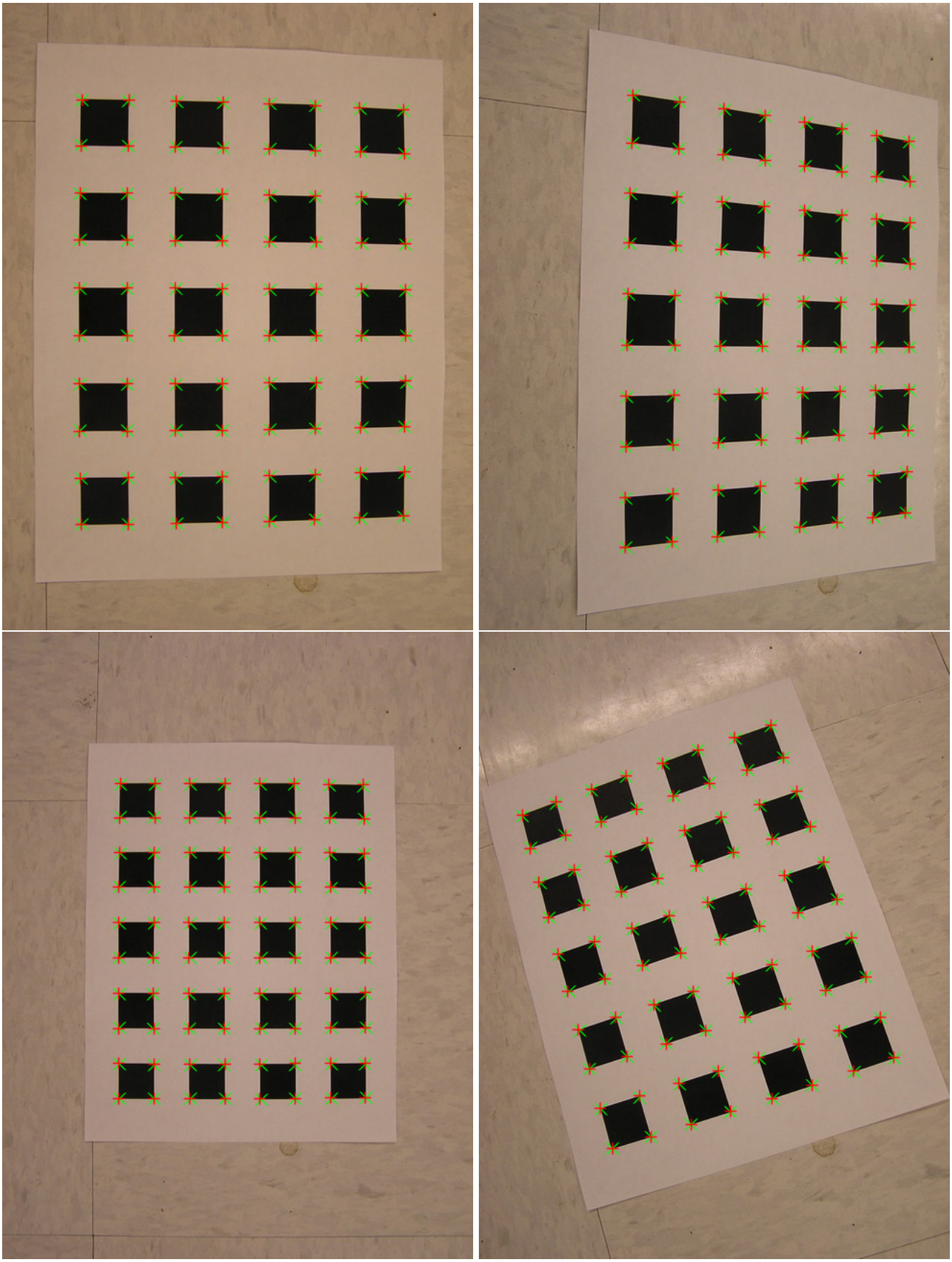


Figure 4: Reprojected points by refined parameters

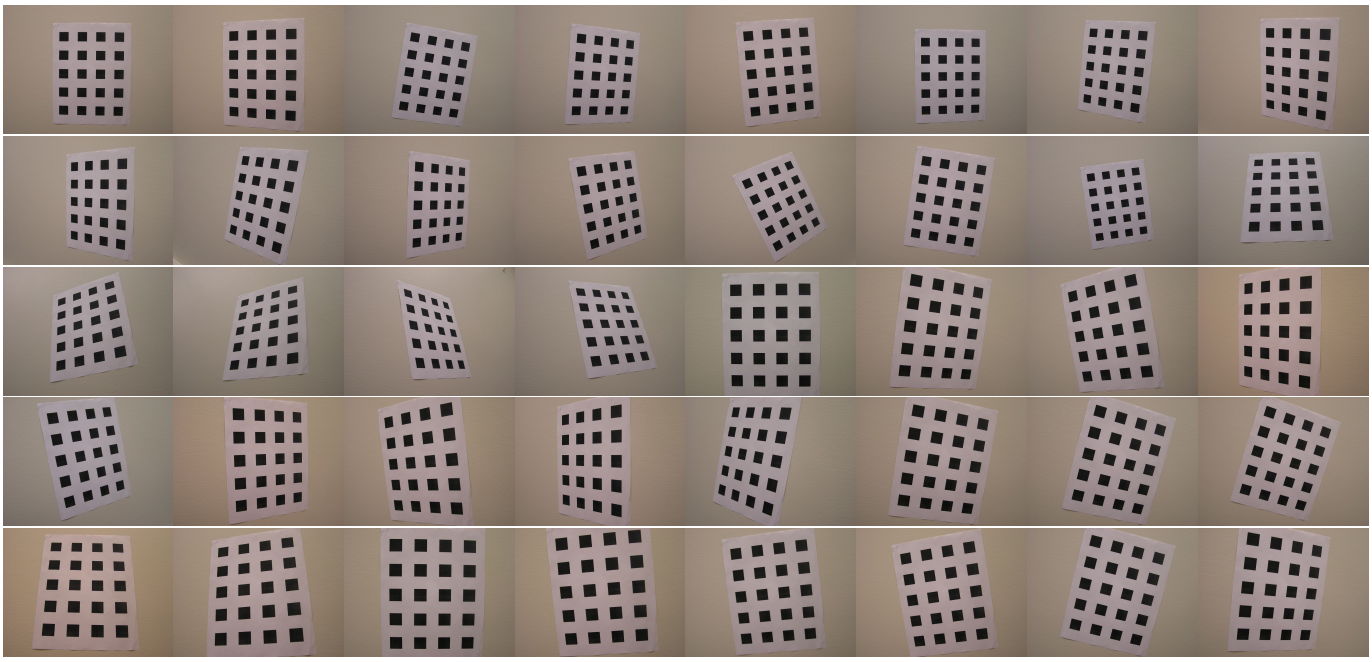


Figure 5: Image set A

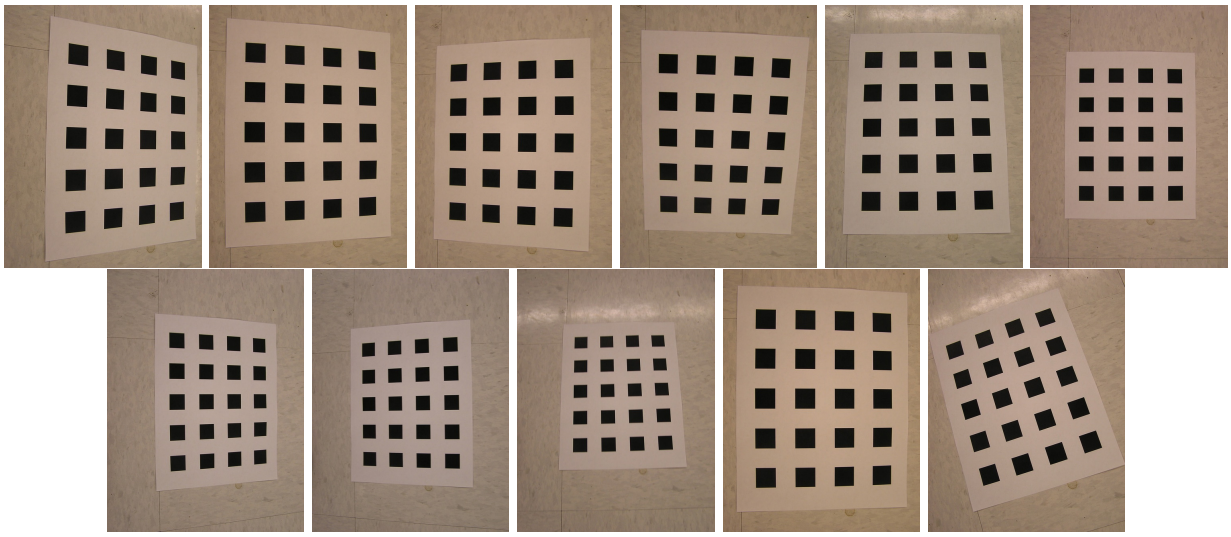


Figure 6: Image set B

4 C++ Source Code

I attach the source code.

```
#include <opencv/cv.h>
#include <opencv/highgui.h>
#include <iostream>
#include <vector>
#include <string>
#include <sstream>
#include "CCvMat.hpp"
#include "lm.h"

using namespace std;

typedef std::vector<std::pair<CvPoint2D32f, CvPoint2D32f> > PointPairs;
typedef pair<float,float> Line;

//Convert p to K, R, T and D
bool P2KRTD(double *p, int m, CvMat *aK, CvMat *aR, CvMat *aT, CvMat *aDistort) {
    cvSet(aK, cvScalar(0));
    float k[] = { p[0], p[1], p[2], 0, p[3], p[4], 0, 0, 1 };
    memcpy(aK->data.ptr, k, sizeof(float) * 9);
    float d[] = { p[5], p[6], 0, 0 };
    memcpy(aDistort->data.ptr, d, sizeof(float) * 4);
    for (int i = 0; i < aR->rows; ++i) {
        int tmp = 7 + i * 6;
        cvmSet(aR, i, 0, p[tmp + 0]);
        cvmSet(aR, i, 1, p[tmp + 1]);
        cvmSet(aR, i, 2, p[tmp + 2]);
        cvmSet(aT, i, 0, p[tmp + 3]);
        cvmSet(aT, i, 1, p[tmp + 4]);
        cvmSet(aT, i, 2, p[tmp + 5]);
    }
    return true;
}

//Convert K, R, T and D to p
bool KRTD2P(double *p, CvMat *aK, CvMat *aR, CvMat *aT, CvMat *aDistort) {
    p[0] = cvmGet(aK, 0, 0);//alpha
    p[1] = cvmGet(aK, 0, 1);//gamma
    p[2] = cvmGet(aK, 0, 2);//u0
    p[3] = cvmGet(aK, 1, 1);//beta
    p[4] = cvmGet(aK, 1, 2);//v0
    p[5] = cvmGet(aDistort, 0, 0);//k1
    p[6] = cvmGet(aDistort, 1, 0);//k2
    for (int i = 0; i < aR->rows; ++i) {
        int tmp = 7 + i * 6;
        p[tmp + 0] = cvmGet(aR, i, 0);
        p[tmp + 1] = cvmGet(aR, i, 1);
        p[tmp + 2] = cvmGet(aR, i, 2);
        p[tmp + 3] = cvmGet(aT, i, 0);
        p[tmp + 4] = cvmGet(aT, i, 1);
        p[tmp + 5] = cvmGet(aT, i, 2);
    }
    return true;
}

//Reprojection
static void Repr(double *p, double *x, int m, int n, void *adata) {
    int Np = 80;
```

```
    int NImage = n / Np / 2;
    CCvMat K(3, 3, CV_32F);
    CCvMat D(4, 1, CV_32F);
    CCvMat R(NImage, 3, CV_32F);
    CCvMat T(NImage, 3, CV_32F);
    P2KRTD(p, m, K.m(), R.m(), T.m(), D.m());

    CvMat * objPnts = (CvMat *) adata;
    int np = 0;
    for (int nImg = 0; nImg < NImage; ++nImg) {
        float rv[] = { cvmGet(R.m(), nImg, 0), cvmGet(R.m(), nImg, 1), cvmGet(
                R.m(), nImg, 2) };
        CCvMat rotation_vector(3, 1, CV_32F,(char *) rv);
        float tv[] = { cvmGet(T.m(), nImg, 0), cvmGet(T.m(), nImg, 1), cvmGet(
                T.m(), nImg, 2) };
        CCvMat translation_vector(3, 1, CV_32F,(char *) tv);
        CCvMat obj_point(Np, 3, CV_32F);
        for (int i = 0; i < Np; ++i) {
            cvmSet(obj_point.m(), i, 0, cvmGet(objPnts, np, 0));
            cvmSet(obj_point.m(), i, 1, cvmGet(objPnts, np, 1));
            cvmSet(obj_point.m(), i, 2, cvmGet(objPnts, np, 2));
            np++;
        }

        CCvMat proj_img_point(Np, 2, CV_32F);
        cvProjectPoints2(obj_point.m(), rotation_vector.m(),
                translation_vector.m(), K.m(), D.m(), proj_img_point.m());

        for (int i = 0; i < Np; ++i) {
            int j = (nImg * Np + i) * 2;
            x[j] = cvmGet(proj_img_point.m(), i, 0);
            x[j + 1] = cvmGet(proj_img_point.m(), i, 1);
        }
    }
}

string dtos(double d,char*format=0){
    char tmp[256];
    if(format==0) sprintf(tmp,"%5.2f",d);
    else sprintf(tmp,format,d);
    return string(tmp);
}
//LM report
string LMReport(double *info){
    string s;
    s += "\n";
    s += "_/_/_/_/_/_/_/_/_/_/ LMReport _/_/_/_/_/_/_/_/_/_/_/_/_/_/_/\n";
    s += "\n";
    s += "\t||e||_2 of initial p   = "+ dtos(info[0]) + "\n";
    s += "\t||e||_2 of estimated p = "+ dtos(info[1]) + "\n";
    s += "\t||J^T e||_inf          = "+ dtos(info[2]) + "\n";
    s += "\t||Dp||_2               = "+ dtos(info[3]) + "\n";
    s += "\tmu_max[J^T J]_ii ]     = "+ dtos(info[4]) + "\n";
    s += "\t# iterations           = "+ dtos(info[5]) + "\n";
    s += "\treason for terminating = ";
    switch ((int)info[6]) {
        case 1:  s += "stopped by small gradient J^T e\n"; break;
        case 2:  s += "stopped by small Dp\n"; break;
        case 3:  s += "stopped by itmax\n";  break;
        case 4:  s += "singular matrix.  Restart from current p with increased mu\n"; break;
```

```
        case 5:  s += "no further error reduction is possible.  Restart with increased mu\n";break;
        case 6:  s += "stopped by small ||e||_2\n"; break;
        case 7:  s += "stopped by invalid (i.e.  NaN or Inf) \"func\" values.  This is a user error\n"; break;
        default:  break;
    }
    s += "\t# function evaluations = "+ dtos(info[6]) + "\n";
    s += "\t# Jacobian evaluations = "+ dtos(info[7]) + "\n";
    s += "\n";
    s += "_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/\n\n";

    return s;
}

//Camera Calibration using LM
bool CameraCalibrationLM(CvMat *objPnts, CvMat *imgPnts, CvMat *pntCnts, CvMat *aInitK,
        CvMat *aInitR, CvMat *aInitT, CvMat *aInitDistort, CvMat *aK,
        CvMat *aR, CvMat *aT, CvMat *aDistort, string *report) {

    //Measurement
    int n = 2 * imgPnts->rows;
    double *x = new double[n];
    for (int i = 0; i < imgPnts->rows; ++i) {
        int j = i << 1; // i*2
        x[j] = cvmGet(imgPnts, i, 0);
        x[j + 1] = cvmGet(imgPnts, i, 1);
    }

    //Parameter
    int m = 5 + 2 + (3 + 3) * aR->rows;
    double *p = new double[m];

    //Convert K, R, T and D to p
    KRTD2P(p, aInitK, aInitR, aInitT, aInitDistort);

    //LM
    double opts[LM_OPTS_SZ]={LM_INIT_MU,1E-12,1E-12,1E-15,LM_DIFF_DELTA};
    double info[LM_INFO_SZ];
    dlevmar_dif(Repr, p, x, m, n, 10000, opts, info, NULL, NULL,
            objPnts);
    *report=LMReport(info);

    //Convert p to K, R, T and D
    P2KRTD(p, m, aK, aR, aT, aDistort);

    delete[] p;
    delete[] x;
    return true;
}

// Solve A * h = 0
bool GetPerspectiveTransform(int n, CvPoint2D32f *apX, CvPoint2D32f *apXp,
        CvMat *aH) {
    if (n >= 4) {
        int np = 9;
        CCvMat A(n * 2, np, CV_32F);
        for (int i = 0; i < n; ++i) {
            float tmpP[] = { apX[i].x, apX[i].y, 1, 0, 0, 0, -apXp[i].x
                    * apX[i].x, -apXp[i].x * apX[i].y, -apXp[i].x, 0, 0, 0,
                    -apXp[i].x, -apX[i].y, -1, apXp[i].y * apX[i].x, apXp[i].y
                            * apX[i].y, apXp[i].y };
```

```
            memcpy(A.m()->data.ptr + A.m()->step * i * 2, tmpP, sizeof(float) * np
                    * 2);
        }
        CCvMat w(np, np, CV_32F),u(np, np, CV_32F),v(np, np, CV_32F);
        (A.t()*A).SVD(&w,&u,&v);
        for (int i = 0; i < np; ++i)
            cvmSet(aH, i / 3, i % 3, cvmGet(v.m(), i, np-1));
        return true;
    } else {
        std::cerr << "Too few points in GetPerspectiveTransform/n";
        return false;
    }
}

//Normalize the coordinates of point set
bool NormalizedPointSet(uint n, CvPoint2D32f *x, CvPoint2D32f *xp, CvMat *T) {
    double meanX = 0, meanX2 = 0, meanY = 0, meanY2 = 0;
    for (uint i = 0; i < n; ++i) {
        meanX += x[i].x;
        meanY += x[i].y;
        meanX2 += x[i].x * x[i].x;
        meanY2 += x[i].y * x[i].y;
    }
    meanX /= (double) n;
    meanY /= (double) n;
    meanX2 /= (double) n;
    meanY2 /= (double) n;
    double sx = sqrt(2) / sqrt(meanX2 - meanX * meanX);
    double sy = sqrt(2) / sqrt(meanY2 - meanY * meanY);
    float tmp[] = { sx, 0, -meanX * sx, 0, sy, -meanY * sy, 0, 0, 1 };
    memcpy(T->data.ptr, tmp, sizeof(float) * 9);
    for (uint i = 0; i < n; ++i) {
        float txf[]={x[i].x,x[i].y,1};
        CCvMat tx(3, 1, CV_32F, (char*)txf);
        CCvMat Ttx = CCvMat(T)*tx;
        xp[i].x = cvmGet(Ttx.m(), 0, 0) / cvmGet(Ttx.m(), 2, 0);
        xp[i].y = cvmGet(Ttx.m(), 1, 0) / cvmGet(Ttx.m(), 2, 0);
    }
    return true;
}

//Normalized DLT
bool NormalizedDLT(PointPairs aPairs, CvMat *aH) {
    //Normalize the coordinate of points on ImageA
    CvPoint2D32f *x = new CvPoint2D32f[aPairs.size()];
    CvPoint2D32f *nx = new CvPoint2D32f[aPairs.size()];
    CCvMat T(3, 3, CV_32F);
    for (uint i = 0; i < aPairs.size(); ++i)
        x[i] = aPairs[i].first;
    NormalizedPointSet(aPairs.size(), x, nx, T.m());
    delete[] x;

    //Normalize the coordinate of points on ImageB
    CvPoint2D32f *xp = new CvPoint2D32f[aPairs.size()];
    CvPoint2D32f *nxp = new CvPoint2D32f[aPairs.size()];
    CCvMat Td(3, 3, CV_32F);
    for (uint i = 0; i < aPairs.size(); ++i)
        xp[i] = aPairs[i].second;
    NormalizedPointSet(aPairs.size(), xp, nxp, Td.m());
    delete[] xp;
```

```
    //Calc Hd (Normalized H)
    CCvMat Hd(3, 3, CV_32F);
    if (GetPerspectiveTransform(aPairs.size(), nx, nxp, Hd.m())) {
        //nxp = Hd * nx
        delete[] nx;
        delete[] nxp;

        //Calc H = inv(Td) * Hd * T
        cvCopy((Td.i()*Hd*T).m(),aH);
        return true;
    } else {
        std::cerr << "Fail GetPerspectiveTransform in NormalizedDLT/n";
        return false;
    }
}

//Harris Corner Detector
bool Harris(IplImage *apOrgImage, unsigned int *aNPoint, CvPoint2D32f *apP,
        int aWinHalfSize, int aMinDistance, double aRatioMaxMin) {
    int winWidth = 2 * aWinHalfSize + 1;
    int winSize = winWidth * winWidth;
    int w = apOrgImage->width;
    int h = apOrgImage->height;
    float *GImage = new float[apOrgImage->imageSize * 3];
    float *HarrisImage = new float[apOrgImage->imageSize];
    float *HarrisImageV = new float[apOrgImage->imageSize];
    int iws = apOrgImage->widthStep;
    int gws = w * 3;

    //Compute g(u,v)^T*g(u,v)
    for (int v = 1; v < apOrgImage->height - 1; ++v) {
        unsigned char *img = (unsigned char*) apOrgImage->imageData;
        int vIStep = v * iws;
        int vGStep = v * gws;
        for (int u = 1; u < apOrgImage->width - 1; ++u) {
            int intPos = vIStep + u;
            double sobleU = img[intPos - iws - 1] + 2 * img[intPos - 1]
                    + img[intPos + iws - 1] - img[intPos - iws + 1] - 2
                    * img[intPos + 1] - img[intPos + iws + 1];
            double sobleV = img[intPos - iws - 1] + 2 * img[intPos - iws]
                    + img[intPos - iws + 1] - img[intPos + iws - 1] - 2
                    * img[intPos + iws] - img[intPos + iws + 1];
            int tmp = vGStep + u * 3;
            GImage[tmp + 0] = sobleU * sobleU;
            GImage[tmp + 1] = sobleV * sobleU;
            GImage[tmp + 2] = sobleV * sobleV;
        }
    }

    //Compute G(u,v) and H(u,v) then find H_max
    double max = -1000000000;
    for (int v = aWinHalfSize; v < h - aWinHalfSize; ++v) {
        int vGStep = v * gws;
        for (int u = aWinHalfSize; u < w - aWinHalfSize; ++u) {
            int intPos = vGStep + u * 3;
            double g11 = 0, g12 = 0, g22 = 0;
            for (int vv = -aWinHalfSize; vv < aWinHalfSize; ++vv) {
                for (int uu = -aWinHalfSize; uu < aWinHalfSize; ++uu) {
                    int tmp = intPos + vv * gws + uu * 3;
```

```
                    g11 += GImage[tmp + 0];
                    g12 += GImage[tmp + 1];
                    g22 += GImage[tmp + 2];
                }
            }
            g11 /= (double) winSize;
            g12 /= (double) winSize;
            g22 /= (double) winSize;
            HarrisImage[v * w + u] = (g11 * g22 - g12 * g12) + 0.04 * (g11
                    + g22) * (g11 + g22);
            if (max < HarrisImage[v * w + u])
                max = HarrisImage[v * w + u];
        }
    }
    delete[] GImage;

    //Non-maximum suppression
    for (int v = aMinDistance; v < h - aMinDistance; ++v) {
        int vHStep = v * w;
        for (int u = aMinDistance; u < w - aMinDistance; ++u) {
            int intPos = vHStep + u;
            float intVal = HarrisImage[intPos];
            HarrisImageV[vHStep + u] = intVal;
            for (int vv = -aMinDistance; vv < aMinDistance; ++vv)
                for (int uu = -aMinDistance; uu < aMinDistance; ++uu)
                    if (intVal < HarrisImage[intPos + vv * w + uu]) {
                        HarrisImageV[vHStep + u] = 0;
                        break;
                    }
        }
    }

    //Extract corners
    unsigned int count = 0;
    for (int v = 0; v < h; ++v) {
        int vHStep = v * w;
        for (int u = 0; u < w; ++u)
            if ((HarrisImageV[vHStep + u] != 0) && (HarrisImageV[vHStep + u]
                    > max * aRatioMaxMin))
                if (count < *aNPoint) {
                    apP[count].x = u;
                    apP[count].y = v;
                    count++;
                }
    }
    *aNPoint = count;
    delete[] HarrisImage;
    delete[] HarrisImageV;

    return true;
}

//Intersection
CvPoint2D32f Intersection(pair<float, float> l1, pair<float, float> l2) {
    float a[] = {cos(l1.second), sin(l1.second), -l1.first};
    CCvMat am(3, 1, CV_32F,(char*) a);
    float b[] = {cos(l2.second), sin(l2.second), -l2.first};
    CCvMat bm(3, 1, CV_32F,(char*) b);
    CCvMat cm = am.CrossProductWith(bm);
    return cvPoint2D32f(cvmGet(cm.m(), 0, 0) / cvmGet(cm.m(), 2, 0),
```

```
                        cvmGet(cm.m(), 1, 0) / cvmGet(cm.m(), 2, 0));
}

//Line Grouping
bool LineGrouping(vector<Line > *lines, CvSize corner_size,
        vector<pair<float, float> > *mrgLines, float meanDist) {
    vector<int> nLines[2];
    pair<float, float> firstLine = (*lines)[0];
    int n;
    //Grouping
    for (unsigned int i = 0; i < lines->size(); i++) {
        pair<float, float> line = (*lines)[i];
        //Divide to two sets which are almost perpendicular each other
        if (fabs(sin(line.second - firstLine.second)) < sin(CV_PI/6)) n = 0;
        else n = 1;
        pair<int, float> minD=pair<int, float>(0,10000);
        //Find nearest group
        for (unsigned int i = 0; i < mrgLines[n].size(); ++i) {
            float gRho = mrgLines[n][i].first / (float) nLines[n][i];
            float gTheta = mrgLines[n][i].second / (float) nLines[n][i];
            float Rho = line.first,Theta = line.second;
            float d = fabs( gRho - Rho * cos(Theta - gTheta) );
            if (d < minD.second)    minD = pair<int, float>(i,d);
        }
        if (minD.second < meanDist/3) {//If near group exists, then adding
            mrgLines[n][minD.first].first += line.first;
            mrgLines[n][minD.first].second += line.second;
            nLines[n][minD.first]++;
        } else {// else create another group
            mrgLines[n].push_back(line);
            nLines[n].push_back(1);
        }
    }
    //Calculate means
    for (int n = 0; n < 2; n++)
        for (unsigned int i = 0; i < mrgLines[n].size(); i++) {
            mrgLines[n][i].first /= (float) nLines[n][i];
            mrgLines[n][i].second /= (float) nLines[n][i];
        }

    //Decide horizontal and vertical line sets
    if (mrgLines[0].size() > mrgLines[1].size()) {
        vector<pair<float, float> > tmp = mrgLines[1];
        mrgLines[1] = mrgLines[0];
        mrgLines[0] = tmp;
    }

    //Sort lines in each of horizontal and vertical sets
    for (int n = 0; n < 2; n++) {
        float rho0 = mrgLines[n][0].first;
        float theta0 = mrgLines[n][0].second;
        for (unsigned int i = 0; i < mrgLines[n].size(); i++) {
            float rho = mrgLines[n][i].first;
            float theta = mrgLines[n][i].second;
            //project center point to perpendicular line
            mrgLines[n][i].first = rho * cos(theta - theta0) - rho0;
        }
        sort(mrgLines[n].begin(), mrgLines[n].end());
        //restore rho
        for (unsigned int i = 0; i < mrgLines[n].size(); i++) {
```

```
            float rho = mrgLines[n][i].first;
            float theta = mrgLines[n][i].second;
            mrgLines[n][i].first = (rho + rho0) / cos(theta - theta0);
        }
    }
    if ((mrgLines[0].size() != (unsigned int) corner_size.width)
            ||(mrgLines[1].size() != (unsigned int) corner_size.height)) {
        cerr << "Fail to detect lines\n";
        cerr << "nLinesX="<< mrgLines[0].size() << "/"<< corner_size.width
                << endl;
        cerr << "nLinesY="<< mrgLines[1].size() << "/"<< corner_size.height
                << endl;
        return false;
    }
    return true;
}

//Drawing
bool DrawLines(IplImage *src, vector<pair<float, float> > lines, CvScalar color) {
    for (unsigned int j = 0; j < lines.size(); ++j) {
        float rho = lines[j].first;
        float theta = lines[j].second;
        CvPoint pt1, pt2;
        double a = cos(theta), b = sin(theta);
        double x0 = a * rho, y0 = b * rho;
        pt1.x = cvRound(x0 + 1000 * (-b));
        pt1.y = cvRound(y0 + 1000 * (a));
        pt2.x = cvRound(x0 - 1000 * (-b));
        pt2.y = cvRound(y0 - 1000 * (a));
        cvLine(src, pt1, pt2, color, 1, CV_AA, 0);
    }
    return true;
}

//Draw Lines
bool DrawLines(IplImage *src, CvSeq * lines,CvScalar color) {
    vector<Line> vLines;
    for (int i = 0; i < lines->total; i++) {
        float* line = (float*) cvGetSeqElem(lines, i);
        vLines.push_back(Line(line[0],line[1]));
    }
    DrawLines(src, vLines,color);
    return true;
}

//Draw Corners
bool DrawCorners(IplImage *src, CvPoint2D32f* pnts, int n, CvFont *font=0) {
    int cl = 6;
    for (int i = 0; i < n; i++) {
        CvPoint2D32f x = pnts[i];
        char tmp[10];
        sprintf(tmp, "%02d", i);
        if(font!=0)cvPutText(src, tmp, cvPoint(x.x, x.y-cl), font, CV_RGB(0,255,0));
        cvLine(src, cvPoint(x.x - cl, x.y - cl), cvPoint(x.x + cl, x.y + cl),
                CV_RGB(0,255,0),1, CV_AA);
        cvLine(src, cvPoint(x.x + cl, x.y - cl), cvPoint(x.x - cl, x.y + cl),
                CV_RGB(0,255,0),1, CV_AA);
    }
    return true;
}
```

```
//Draw Reproject Corners
bool DrawReprojCorners(IplImage *src, int nImg, CvMat *objPnts, int Np,
        CvMat *aK, CvMat *aR, CvMat *aT, CvMat *aDistort, CvScalar color) {
    int np = Np * nImg;
    float rvf[] = { cvmGet(aR, nImg, 0), cvmGet(aR, nImg, 1), cvmGet(aR, nImg, 2) };
    CCvMat r(3, 1, CV_32F,(char *) rvf);
    float tvf[] = { cvmGet(aT, nImg, 0), cvmGet(aT, nImg, 1), cvmGet(aT, nImg, 2) };
    CCvMat t(3, 1, CV_32F,(char *) tvf);
    CCvMat obj(Np, 3, CV_32F);
    for (int i = 0; i < Np; ++i) {
        cvmSet(obj.m(), i, 0, cvmGet(objPnts, np, 0));
        cvmSet(obj.m(), i, 1, cvmGet(objPnts, np, 1));
        cvmSet(obj.m(), i, 2, cvmGet(objPnts, np, 2));
        np++;
    }
    CCvMat prj(Np, 2, CV_32F);
    cvProjectPoints2(obj.m(), r.m(),t.m(), aK, aDistort, prj.m());
    int cl = 6;
    for (int i = 0; i < Np; i++) {
        CvPoint2D32f x = cvPoint2D32f(cvmGet(prj.m(), i, 0), cvmGet(prj.m(), i, 1));
        cvLine(src, cvPoint(x.x - cl, x.y), cvPoint(x.x + cl, x.y ), color, 1,  CV_AA);
        cvLine(src, cvPoint(x.x, x.y - cl), cvPoint(x.x, x.y + cl), color, 1,   CV_AA);
    }
    return true;
}

//Find Corners
bool FindCorners(IplImage *src, CvSize corner_size, float meanDist, CvPoint2D32f *pnts,
        int *n, IplImage *lineImage1 = 0, IplImage *lineImage2 = 0,
        IplImage *edgeImage = 0) {
    if (!src)
        return -1;
    //Canny
    IplImage* srcGray = cvCreateImage(cvGetSize(src), 8, 1);
    cvCvtColor(src, srcGray, CV_BGR2GRAY);
    IplImage* dst = cvCreateImage(cvGetSize(src), 8, 1);
    cvCanny(srcGray, dst, 500, 300, 3);
    if (edgeImage != 0) {
        cvCopy(dst, edgeImage);
        cvNot(edgeImage,edgeImage);
    }

    //Hough Transform
    CvMemStorage* strg = cvCreateMemStorage(0);
    CvSeq* slines = 0;
    slines = cvHoughLines2(dst, strg, CV_HOUGH_STANDARD, 1, CV_PI/180, 50, 0, 0);
    vector<Line > lines;
    for (int i = 0; i < slines->total; i++) {
        float* line = (float*) cvGetSeqElem(slines, i);
        if (line[0] < 0){//if Theta > 180 then Rho < 0 !!
            lines.push_back(Line(-line[0],line[1]-CV_PI));
        }else
            lines.push_back(Line(line[0],line[1]));
    }
    cvReleaseMemStorage(&strg);

    //Marge lines
    vector<pair<float, float> > mrgLines[2];
    bool flag = true;
```

```
    if (LineGrouping(&lines, corner_size, mrgLines, meanDist)) {
        unsigned int nHarris = 1000;
        CvPoint2D32f *harris = new CvPoint2D32f[nHarris];
        Harris(srcGray, &nHarris, harris, 1, 15, 0.001);
        for (unsigned int i = 0; i < mrgLines[1].size(); i++) {
            for (unsigned int j = 0; j < mrgLines[0].size(); j++) {
                int index=j + i * mrgLines[0].size();
                //Compute Intersection
                pnts[index] = Intersection(mrgLines[0][j], mrgLines[1][i]);

                //Find the nearest Harris corner
                pair<CvPoint2D32f,float> minD=pair<CvPoint2D32f,float>(cvPoint2D32f(0,0),1000);
                for (unsigned int h = 0; h < nHarris; h++) {
                    CvPoint2D32f x1 = pnts[index];
                    CvPoint2D32f x2 = harris[h];
                    float d = sqrt((x1.x - x2.x) * (x1.x - x2.x)
                                + (x1.y - x2.y) * (x1.y - x2.y));
                    if (d < minD.second) {
                        minD.first  = x2 ;
                        minD.second = d;
                    }
                }

                //Move to the nearest Harris corner
                if (minD.second < 10)
                    pnts[index] = minD.first;
            }
        }
        //# of corners
        *n = mrgLines[0].size() * mrgLines[1].size();
        delete[] harris;

        if (lineImage1 != NULL) DrawLines(lineImage1, lines, CV_RGB(0,255,0));
        if (lineImage2 != NULL){
            DrawLines(lineImage2, mrgLines[0], CV_RGB(255,0,0));
            DrawLines(lineImage2, mrgLines[1], CV_RGB(255,0,0));
        }
        flag = true;
    } else {
        flag = false;
    }

    cvReleaseImage(&srcGray);
    cvReleaseImage(&dst);

    return flag;
}

//Estimete Radial Distortion
bool EstimateRadialDistortion(CvMat *objPnts, CvMat *imgPnts, CvMat *pntCnts,
        CvMat **H, CvMat *K, CvMat *aDistort) {
    int NImage = pntCnts->rows;
    int n = 0;
    for (int nImg = 0; nImg < NImage; ++nImg)
        n += pntCnts->data.i[nImg];
    CCvMat D(2 * n, 2, CV_64F);
    CCvMat d(2 * n, 1, CV_64F);

    double u0 = cvmGet(K, 0, 2);
    double v0 = cvmGet(K, 1, 2);
```

```
    double alpha = cvmGet(K, 0, 0);
    double beta = cvmGet(K, 1, 1);
    double gamma = cvmGet(K, 0, 1);
    int np = 0;
    CCvMat Kinv = CCvMat(K).i();
    for (int nImg = 0; nImg < NImage; ++nImg) {
        CCvMat RT = Kinv * CCvMat(H[nImg]);
        for (int i = 0; i < pntCnts->data.i[nImg]; ++i) {
            float
                    tmpM[] = { cvmGet(objPnts, np, 0), cvmGet(objPnts, np, 1),
                            1 };
            CCvMat M(3, 1, CV_32F,(char *) tmpM);
            CCvMat Mc = RT * M;
            double x = cvmGet(Mc.m(), 0, 0) / cvmGet(Mc.m(), 2, 0);
            double y = cvmGet(Mc.m(), 1, 0) / cvmGet(Mc.m(), 2, 0);
            double r2 = x * x + y * y;
            double r4 = r2 * r2;
            double u = cvmGet(imgPnts, np, 0);
            double v = cvmGet(imgPnts, np, 1);
            double uh = u0 + alpha * x + gamma * y;
            double vh = v0 + beta * y;
            cvmSet(D.m(), np * 2, 0, (u - u0) * r2);
            cvmSet(D.m(), np * 2, 1, (u - u0) * r4);
            cvmSet(D.m(), np * 2 + 1, 0, (v - v0) * r2);
            cvmSet(D.m(), np * 2 + 1, 1, (v - v0) * r4);
            cvmSet(d.m(), np * 2, 0, uh - u);
            cvmSet(d.m(), np * 2 + 1, 0, vh - v);
            np++;
        }
    }
    CCvMat k = (D.t() * D).i() * D.t() * d;
    cvmSet(aDistort, 0, 0, cvmGet(k.m(), 0, 0));
    cvmSet(aDistort, 1, 0, cvmGet(k.m(), 1, 0));
    cvmSet(aDistort, 2, 0, 0);
    cvmSet(aDistort, 3, 0, 0);
    return true;
}

//Calibrate Camera
bool CameraCalibration(CvMat *objPnts, CvMat *imgPnts, CvMat *pntCnts,
        CvSize imgSize, CvMat *aK, CvMat *aR, CvMat *aT, CvMat *aDistort) {
    int NImage = pntCnts->rows;

    //Compute H
    CvMat **H = new CvMat *[NImage];
    int np = 0;
    for (int nImg = 0; nImg < NImage; ++nImg) {
        H[nImg] = cvCreateMat(3, 3, CV_32F);
        PointPairs pairs;
        for (int i = 0; i < pntCnts->data.i[nImg]; ++i) {
            std::pair<CvPoint2D32f, CvPoint2D32f> p;
            p.first.x = cvmGet(objPnts, np, 0);
            p.first.y = cvmGet(objPnts, np, 1);
            p.second.x = cvmGet(imgPnts, np, 0);
            p.second.y = cvmGet(imgPnts, np, 1);
            pairs.push_back(p);
            np++;
        }
        NormalizedDLT(pairs, H[nImg]);
        cvConvertScale(H[nImg], H[nImg], 1 / cvmGet(H[nImg], 2, 2));
```

```
}

//Compute b
CCvMat V(2 * NImage, 6, CV_64F);
for (int nImg = 0; nImg < NImage; ++nImg) {
    double h11 = cvmGet(H[nImg], 0, 0);
    double h12 = cvmGet(H[nImg], 1, 0);
    double h13 = cvmGet(H[nImg], 2, 0);
    double h21 = cvmGet(H[nImg], 0, 1);
    double h22 = cvmGet(H[nImg], 1, 1);
    double h23 = cvmGet(H[nImg], 2, 1);
    double v12[6] = { h11 * h21, h11 * h22 + h12 * h21, h12 * h22, h13
            * h21 + h11 * h23, h13 * h22 + h12 * h23, h13 * h23 };
    double v11[6] = { h11 * h11, h11 * h12 + h12 * h11, h12 * h12, h13
            * h11 + h11 * h13, h13 * h12 + h12 * h13, h13 * h13 };
    double v22[6] = { h21 * h21, h21 * h22 + h22 * h21, h22 * h22, h23
            * h21 + h21 * h23, h23 * h22 + h22 * h23, h23 * h23 };
    for (int i = 0; i < 6; ++i) {
        cvmSet(V.m(), 2 * nImg + 0, i, v12[i]);
        cvmSet(V.m(), 2 * nImg + 1, i, v11[i] - v22[i]);
    }
}
CCvMat w(6, 6, CV_64F), u(6, 6, CV_64F), v(6, 6, CV_64F);
(V.t() * V).SVD(&w, &u, &v);
double b11 = cvmGet(v.m(), 0, 5);
double b12 = cvmGet(v.m(), 1, 5);
double b22 = cvmGet(v.m(), 2, 5);
double b13 = cvmGet(v.m(), 3, 5);
double b23 = cvmGet(v.m(), 4, 5);
double b33 = cvmGet(v.m(), 5, 5);

//Comupte Initial K
double v0 = (b12 * b13 - b11 * b23) / (b11 * b22 - b12 * b12);
double lamda = b33 - (b13 * b13 + v0 * (b12 * b13 - b11 * b23)) / b11;
double alpha = sqrt(lamda / b11);
double beta = sqrt(lamda * b11 / (b11 * b22 - b12 * b12));
double gamma = -b12 * alpha * alpha * beta / lamda;
double u0 = gamma * v0 / beta - b13 * alpha * alpha / lamda;
cvSet(aK, cvScalar(0));
cvmSet(aK, 0, 0, alpha);
cvmSet(aK, 0, 1, gamma);
cvmSet(aK, 0, 2, u0);
cvmSet(aK, 1, 1, beta);
cvmSet(aK, 1, 2, v0);
cvmSet(aK, 2, 2, 1);

//Estimation of R and T
CCvMat Kinv = CCvMat(aK).i();
for (int nImg = 0; nImg < NImage; ++nImg) {
    CCvMat h1(3, 1, CV_64F), h2(3, 1, CV_64F), h3(3, 1, CV_64F);
    cvGetCol(H[nImg], h1.m(), 0);
    cvGetCol(H[nImg], h2.m(), 1);
    cvGetCol(H[nImg], h3.m(), 2);
    double a = 1 / (Kinv * h1).n();
    CCvMat r1 = Kinv * h1 * a;
    CCvMat r2 = Kinv * h2 * a;
    CCvMat r3 = r1.CrossProductWith(r2);
    float q[9];
    memcpy(q, r1.m()->data.ptr, sizeof(float) * 3);
    memcpy(q + 3, r2.m()->data.ptr, sizeof(float) * 3);
```

```
        memcpy(q + 6, r3.m()->data.ptr, sizeof(float) * 3);
        CCvMat Q(3, 3, CV_32F,(char *) q);
        Q = Q.t();
        CCvMat w(3, 3, CV_32F), u(3, 3, CV_32F), v(3, 3, CV_32F);
        Q.SVD(&w, &u, &v);
        CCvMat R = u * v.t();
        CCvMat r(3, 1, CV_32F);
        cvRodrigues2(R.m(), r.m());
        memcpy(aR->data.ptr + aR->step * nImg, r.m()->data.ptr, sizeof(float) * 3);
        CCvMat t = Kinv * h3 * a;
        memcpy(aT->data.ptr + aT->step * nImg, t.m()->data.ptr, sizeof(float)   * 3);
    }

    EstimateRadialDistortion(objPnts, imgPnts, pntCnts, H, aK, aDistort);

    for (int nImg = 0; nImg < NImage; ++nImg)
        cvReleaseMat(&H[nImg]);
    delete[] H;
    return true;
}

//Porjection Error
double ProjError(CvMat *objPnts, CvMat *imgPnts, CvMat *pntCnts,
        CvSize imgSize, CvMat *aK, CvMat *aR, CvMat *aT, CvMat *aDistort) {
    int NImage = pntCnts->rows;
    int np = 0;
    double error = 0;
    for (int nImg = 0; nImg < NImage; ++nImg) {
        int Np = pntCnts->data.i[nImg];
        float rv[] = { cvmGet(aR, nImg, 0), cvmGet(aR, nImg, 1), cvmGet(aR,
                nImg, 2) };
        CCvMat rotation_vector(3, 1, CV_32F,(char *) rv);
        float tv[] = { cvmGet(aT, nImg, 0), cvmGet(aT, nImg, 1), cvmGet(aT,
                nImg, 2) };
        CCvMat translation_vector(3, 1, CV_32F,(char *) tv);
        CCvMat obj_point(Np, 3, CV_32F);
        CCvMat img_point(Np, 2, CV_32F);
        for (int i = 0; i < pntCnts->data.i[nImg]; ++i) {
            cvmSet(obj_point.m(), i, 0, cvmGet(objPnts, np, 0));
            cvmSet(obj_point.m(), i, 1, cvmGet(objPnts, np, 1));
            cvmSet(obj_point.m(), i, 2, cvmGet(objPnts, np, 2));
            cvmSet(img_point.m(), i, 0, cvmGet(imgPnts, np, 0));
            cvmSet(img_point.m(), i, 1, cvmGet(imgPnts, np, 1));
            np++;
        }
        CCvMat proj_img_point(Np, 2, CV_32F);
        cvProjectPoints2(obj_point.m(), rotation_vector.m(),
                translation_vector.m(), aK, aDistort, proj_img_point.m());

        CCvMat e = (img_point - proj_img_point).t() * (img_point
                - proj_img_point);
        error += e.tr();
    }
    return sqrt(error / (double) np);
}

int main(int argc, char **argv) {
    CvFont font;
    cvInitFont(&font, CV_FONT_HERSHEY_SIMPLEX, 0.5, 0.5, 0, 1, CV_AA);
    cvNamedWindow("Corners", CV_WINDOW_AUTOSIZE);
```

```
    int row = 10; //rows of corners
    int col = 8; //cols of corners
//  int m = 40; //# of image
    int m=11;    //# of image
    CvSize corner_size = cvSize(col, row);//corner matrix size

    //
    //Corner Extraction and Labeling
    //
    CvPoint2D32f *allCorners = new CvPoint2D32f[row * col * m];//all corners
    int N = 0;//total # of corners
    CCvMat pntCnts(m, 1, CV_32S); //# of deteceted corners in each image
    CvSize imgSize;
    for (int i = 0; i < m; ++i) {
        //Open Image
        char filename[256];
//      if (i < 20) sprintf(filename, "Test/P10100%02ds", i + 1);
//      else              sprintf(filename, "Test/P10100%02ds", i + 27);
                sprintf(filename, "Calib/P10100%02d", i+26);
        cerr << "Load "<< filename << endl;
        IplImage *img = cvLoadImage((string(filename) + ".jpg").c_str());
        if (img == NULL) {
            cout << "Can't Load "<< filename << endl;
            std::exit(0);
        }

        //Find Corners
        int find_num;
        IplImage *lineImage1 = cvCloneImage(img);
        IplImage *lineImage2 = cvCloneImage(img);
        IplImage *edgeImage = cvCreateImage(cvGetSize(img), 8, 1);
        if (FindCorners(img, corner_size, 60, allCorners + N, &find_num, lineImage1,
                lineImage2,edgeImage)) {
            pntCnts.m()->data.i[i] = find_num;
            N += find_num;
        }
        cvSaveImage((string(filename) + "L1.png").c_str(), lineImage1);
        cvSaveImage((string(filename) + "L2.png").c_str(), lineImage2);
        cvSaveImage((string(filename) + "E.png").c_str(), edgeImage);
        imgSize = cvGetSize(img);
        cvReleaseImage(&img);
        cvReleaseImage(&lineImage1);
        cvReleaseImage(&lineImage2);
        cvReleaseImage(&edgeImage);
    }

    //
    //Camera Calibration
    //
    if (N == m * col * row) {
        //Set object coordinates
        float sqSize = 1.0 * 2.54;
        CCvMat objPnts(N, 3, CV_32F);
        for (int k = 0, cnt = 0; k < m; ++k)
            for (int j = 0; j < row; ++j)
                for (int i = 0; i < col; ++i) {
                    cvmSet(objPnts.m(), cnt, 0, sqSize * i);
                    cvmSet(objPnts.m(), cnt, 1, sqSize * j);
                    cvmSet(objPnts.m(), cnt, 2, 0);
```

```
            cnt++;
        }
    //Set image coordinates
    CCvMat imgPnts(N, 2, CV_32F);
    for (int i = 0; i < N; ++i) {
        cvmSet(imgPnts.m(), i, 0, allCorners[i].x);
        cvmSet(imgPnts.m(), i, 1, allCorners[i].y);
    }

    //Initial Calibration
    CCvMat K(3, 3, CV_32F), distort(4, 1, CV_32F), R(m, 3, CV_32F), T(m, 3, CV_32F);
    CameraCalibration(objPnts.m(), imgPnts.m(), pntCnts.m(), imgSize,
            K.m(), R.m(), T.m(), distort.m());
    double projError = ProjError(objPnts.m(), imgPnts.m(), pntCnts.m(),
            imgSize, K.m(), R.m(), T.m(), distort.m());

    //Refinement of Calibration
    CCvMat rfK(3, 3, CV_32F), rfDistort(4, 1, CV_32F);
    CCvMat rfR(m, 3, CV_32F),rfT(m, 3, CV_32F);
    string lmReport;
    CameraCalibrationLM(objPnts.m(), imgPnts.m(), pntCnts.m(), K.m(), R.m(), T.m(),
            distort.m(), rfK.m(), rfR.m(), rfT.m(), rfDistort.m(), &lmReport);
    double projError2 = ProjError(objPnts.m(), imgPnts.m(), pntCnts.m(), imgSize,
            rfK.m(), rfR.m(), rfT.m(), rfDistort.m());

    //
    //for report
    //
    N = 0;
    for (int i = 0; i < m; ++i) {
        //Open Image
        char filename[256];
//          if (i < 20) sprintf(filename, "Test/P10100%02ds", i + 1);
//          else                sprintf(filename, "Test/P10100%02ds", i + 27);
                    sprintf(filename, "Calib/P10100%02d", i+26);
        cerr << filename << endl;
        IplImage *img = cvLoadImage((string(filename) + ".jpg").c_str());
        if (img == NULL) {
            cout << "Can't Load "<< filename << endl;
            std::exit(0);
        }

        //Find Corners
        IplImage *img1 = cvCloneImage(img);
        DrawCorners(img1, allCorners + N, pntCnts.m()->data.i[i], &font);
        IplImage *img2 = cvCloneImage(img);
        cvSaveImage((string(filename) + "C.png").c_str(), img1);
        DrawCorners(img2, allCorners + N, pntCnts.m()->data.i[i]);
        DrawReprojCorners(img2, i, objPnts.m(), pntCnts.m()->data.i[i],
                K.m(), R.m(), T.m(), distort.m(), CV_RGB(255,255,0));
        IplImage *img3 = cvCloneImage(img);
        cvSaveImage((string(filename) + "C1.png").c_str(), img2);
        DrawCorners(img3, allCorners + N, pntCnts.m()->data.i[i]);
        DrawReprojCorners(img3, i, objPnts.m(), pntCnts.m()->data.i[i],
                rfK.m(), rfR.m(), rfT.m(), rfDistort.m(), CV_RGB(255,0,0));
        cvSaveImage((string(filename) + "C2.png").c_str(), img3);
        imgSize = cvGetSize(img);
        cvReleaseImage(&img);
        cvReleaseImage(&img1);
        cvReleaseImage(&img2);
```

```
            cvReleaseImage(&img3);
            N += pntCnts.m()->data.i[i];
        }
        //Open data file
        ofstream ofs("Calib/data.txt");
        ofs << "\n_/_/_/_/_/_/_/  Initial Estimation _/_/_/_/_/_/_/_/\n\n";
        ofs << "K=\n"<< K.tex() << "distort=\n"<< distort.tex();
        ofs << "R=\n"<< R.tex() << "T=\n"<< T.tex();
        ofs << "projError of initial="<< projError << endl;
        ofs << "\n_/_/_/_/_/_/_/_/_/ LM Estimation _/_/_/_/_/_/_/_/_/\n\n";
        ofs << "K=\n"<< rfK.tex() << "distort=\n"<< rfDistort.tex();
        ofs << "R=\n"<< rfR.tex() << "T=\n"<< rfT.tex();
        ofs << "projError after LM  ="<< projError2 << endl;
        ofs << lmReport;
    }

    cvDestroyWindow("Corners");
    delete[] allCorners;

    return 0;

}

/*
 * CCvMat.hpp
 *
 *  Created on: Oct. 20, 2008
 *      Author: hide
 */

#ifndef CCVMAT_HPP_
#define CCVMAT_HPP_
#include "opencv/cv.h"
#include <fstream>

class CCvMat{
    CvMat* mat;
public:
    template <class T> bool set(T *data, int size, int begin=0){
        if((uint)mat->step==sizeof(T)*mat->cols){
            if(begin+size <= mat->rows*mat->cols){
                memcpy(mat->data.ptr+begin*sizeof(T),data,size*sizeof(T));
                return true;
            }else{
                std::cerr << "memory access violation in CvMat::set\n";
                return false;
            }
        }else{
            std::cerr << "data type is wrong in CvMat::set\n";
            return false;
        }
    }
    template <class T> bool get(T *data, int size, int begin=0){
        if((uint)mat->step==sizeof(T)*mat->cols){
            if(begin+size <= mat->rows*mat->cols){
                memcpy(data,mat->data.ptr+begin*sizeof(T),size*sizeof(T));
                return true;
            }else{
                std::cerr << "memory access violation in CvMat::get\n";
                return false;
```

```
        }
    }else{
        std::cerr << "data type is wrong in CvMat::get\n";
        return false;
    }
}
CCvMat rowVec(int row){
    if(mat->rows>=row){
        char *d=new char[mat->step];
        memcpy(d,mat->data.ptr+row*mat->step,mat->step);
        CCvMat tmp(mat->cols,1,mat->type,d);
        delete [] d;
        return tmp;
    }else{
        std::cerr << "row = "<< row << " must be less than mat->rows = "<< mat->rows << "/n";
        return CCvMat(mat->cols,1,mat->type);//OpenCV has a fatal bug to deal single row vector.  See http://www.nabble.com/CvM
step-problem-td18313298.html
    }
}
CCvMat colVec(int col){
    if(mat->cols>=col){
        uint uSize=mat->step/mat->cols;
        char *d=new char[uSize*mat->rows];
        for (int i = 0; i < mat->rows; ++i) {
            memcpy(d+uSize*i,mat->data.ptr+(i*mat->step+uSize*col),uSize);
        }
        CCvMat tmp(mat->rows,1,mat->type,d);
        delete [] d;
        return tmp;
    }else{
        std::cerr << "col = "<< col << " must be less than mat->cols = "<< mat->cols << "/n";
        return CCvMat(mat->rows,1,mat->type);
    }
}
bool set(CvMat* aM){
    cvReleaseMat(&mat);
    mat=cvCloneMat( aM );
    return true;
}
CvMat* m(){
    return mat;
}
CCvMat(int r, int c,int t){
    if(r!=1){
        mat=cvCreateMat(r,c,t);
    }else{
        std::cerr << "Fail to initialize CCvMat for r=1.  \n";
        std::cerr << "OpenCV has a fatal bug to deal single row vector.\n";
        std::cerr << "See http://www.nabble.com/CvMat-step-problem-td18313298.html\n";
    }
}
CCvMat(int r, int c,int t, char* data){
    if(r!=1){
        mat=cvCreateMat(r,c,t);
        memcpy(mat->data.ptr,data,r*mat->step);
    }else{
        std::cerr << "Fail to initialize CCvMat for r=1.  \n";
        std::cerr << "OpenCV has a fatal bug to deal single row vector.\n";
        std::cerr << "See http://www.nabble.com/CvMat-step-problem-td18313298.html\n";
    }
```

```
}
CCvMat(CvMat *m){
    mat=cvCloneMat( m );
}
~CCvMat(){
    cvReleaseMat(&mat);
}
CCvMat(const CCvMat& obj){
    mat=cvCloneMat( obj.mat );
}
CCvMat operator=(const CCvMat& obj){
    cvReleaseMat(&mat);
    mat=cvCloneMat( obj.mat );
    return *this;
}
CCvMat operator+(const CCvMat& obj){
    CCvMat tmp=*this;
    cvmAdd(mat,obj.mat,tmp.mat);
    return tmp;
}
CCvMat operator-(const CCvMat& obj){
    CCvMat tmp=*this;
    cvmSub(this->mat,obj.mat,tmp.mat);
    return tmp;
}
CCvMat operator*(const CCvMat& obj){
    CCvMat tmp(mat->rows,obj.mat->cols,mat->type);
    cvmMul(mat,obj.mat,tmp.mat);
    return tmp;
}
CCvMat operator*(const double d){
    CCvMat tmp=*this;
    cvConvertScale(mat, tmp.mat, d);
    return tmp;
}
double operator()(int r, int c){
    return cvmGet(mat,r,c);
}
CCvMat operator/(const double d){
    CCvMat tmp=*this;
    cvConvertScale(mat, tmp.mat, 1/d);
    return tmp;
}

CCvMat Invert(){
    CCvMat tmp=*this;
    cvInvert(mat,tmp.mat);
    return tmp;
}
CCvMat PseudoInverse(){
    CCvMat tmp=t();
    cvPseudoInverse(mat,tmp.mat);
    return tmp;
}
CCvMat Transpose(){
    CCvMat tmp(mat->cols,mat->rows,mat->type);
    cvTranspose(mat,tmp.mat);
    return tmp;
}
double Norm(){
```

```
    return cvNorm(mat);
}
double Trace(){return cvTrace(mat).val[0];}
double Determinant(){return cvDet(mat);}
CCvMat CrossProductWith(const CCvMat& obj){
    CCvMat tmp=*this;
    cvCrossProduct(mat,obj.mat,tmp.mat);
    return tmp;
}
double DotProductWith(const CCvMat& obj){
    return cvDotProduct(obj.mat,mat);
}
bool Display(const char* aName){
    std::cout << std::endl;
    std::cout << aName << " =\n";
    for(int j=0; j < mat->rows; j++){
        for(int i=0; i < mat->cols; i++){
            std::cout << cvmGet(mat,j,i) << " ";
        }
        std::cout << std::endl;
    }
    return true;
}
bool Write(std::ofstream *ofs,const char* aName){
    if(!ofs->fail()){
        *ofs << std::endl;
        *ofs << aName << " =\n";
        for(int j=0; j < mat->rows; j++){
            for(int i=0; i < mat->cols; i++){
                *ofs << cvmGet(mat,j,i) << " ";
            }
            *ofs << std::endl;
        }
        return true;
    }else{
        return false;
    }
}
bool SVD(CCvMat *w,CCvMat *u=0,CCvMat *v=0){
    if(u!=0){
        if(v!=0){
            CvMat *W=cvCloneMat(w->m());
            CvMat *U=cvCloneMat(u->m());
            CvMat *V=cvCloneMat(v->m());
            cvSVD(mat, W, U, V, CV_SVD_U_T|CV_SVD_V_T);
            w->set(W);
            u->set(U);
            *u=u->t();
            v->set(V);
            *v=v->t();
            cvReleaseMat(&W);
            cvReleaseMat(&U);
            cvReleaseMat(&V);
        }else{
            CvMat *W=cvCloneMat(w->m());
            CvMat *U=cvCloneMat(u->m());
            cvSVD(mat, W, U);
            w->set(W);
            u->set(U);
            cvReleaseMat(&W);
```

```
            cvReleaseMat(&U);
        }
    }else{
        CvMat *W=cvCloneMat(w->m());
        cvSVD(mat, W);
        w->set(W);
        cvReleaseMat(&W);
    }
    return true;
}
CCvMat i(){return Invert();}
CCvMat t(){return Transpose();}
CCvMat pi(){return PseudoInverse();}
void disp(const char* aName){Display(aName);}
double tr(){return Trace();}
double det(){return Determinant();}
double n(){return Norm();}
uchar* p(){return mat->data.ptr;}
std::string tex(const char *name=0){
    std::string s;
    if(name!=0) s = std::string(name) + std::string("=\n");
    s += "\\left(\\begin{array}{ccc}\n";
    for(int j=0; j < mat->rows; j++){
        for(int i=0; i < mat->cols; i++){
            char tmp[255];
            sprintf(tmp,"%8.4f",cvmGet(mat,j,i));
            s += std::string(tmp);
            if(j< mat->rows-1) s+= "&";
        }
        if(j< mat->rows-1)s += "\\\\";
        s+="\n";
    }
    s += "\\end{array}\\right)\n";
    return s;
}
};

#endif /* CCVMAT_HPP_ */
```