ECE 661 Homework 7

Due: 11/06/2008 Thursday (before the class)

This homework is a straightforward extension of the work you did for homework 6.

This homework is an exercise in image mosaicing without full-blown camera calibration. Please mount your camera on a tripod and take N images by rotating the camera through different angles. N will preferably be odd, typically 9, so that one image can serve as a central image and the others as images on two sides of the central image. Make sure you have a sufficient view-angle overlap between the successive images so that you can compute the homography between them using the RANSAC-based approach you developed for Homework 6. Now chain the homographies and project all of the images into the sensor plane of the central image.

For extra credit, estimate rotation angle between the successive images from the eigenvalues of the homographies.

Notes.

- Clearly identify the steps you have taken to solve the problem with your own words.
- Your grade depends on the completeness and clarity of your work as well as the result.

ECE661 Computer Vision Homework 7

Hidekazu Iwaki

November 6, 2008

1 Mosaicing

Suppose we have the odd number n of images, and $c = (n+1)/2$th image is regarded as the center image. Let the relationship between a point $x^{(i)}$ on i-th image and the corresponding point $x^{(j)}$ on j-th image be ${}_jH_i$, that is, $x^{(j)} = {}_jH_i x^{(i)}$, and lett he relationship between a point x on mosaiced image and the corresponding point $x^{(c)}$ on center c-th image be H_c then the relationship between a point x on the mosaiced image and the correspoinding point $x^{(i)}$is

$$if\ c \leq i\ then\ x = H_i x^{(i)} = H_c \cdot {}_cH_{c+1} \cdot {}_{c+1}H_{c+2} \cdot ... \cdot {}_{i-1}H_i x^{(i)},$$

$$if\ c > i\ then\ x = H_i x^{(i)} = H_c \cdot {}_cH_{c-1} \cdot {}_{c-1}H_{c-2} \cdot ... \cdot {}_{i+1}H_i x^{(i)}.$$

Let original images' width and height are w and h. I used $4w$ as the width and $2h$ as the height for mosaiced image, and

$$H_c = \begin{pmatrix} 1 & 0 & 1.5w \\ 0 & 1 & 0.5h \\ 0 & 0 & 1 \end{pmatrix}.$$

Each ${}_{i+1}H_i\ (i = 1, ..., n)$ is calculated by same way of homework 4 (That is I used DLT for refinements of H). Note that ${}_iH_{i+1} =_{i+1} H_i^{-1}$.

2 Experimental Result

I will show two results of mosaicing.

2.1 Apartment

I will show the parameters I used and the results by Table.1.,Fig.1. In Fig.1, first raw is original images, second is correspondeces between image 1 and image 2, and third is mosaiced image by estimated homography . In second image in Fig.1 blue circles illustrate extracted Harris corners, all lines illustrate correspondence. Green lines illustrate lnliers, red lines illustrate outliers, and yellow lines illustrate four points initially chosen by RANSAC. Angles are calculated by MATLAB using command as follow:

$$angle(\ [1\ 0\ 0] * eig(H)\ ) * 180/pi$$

2.2 Forest

I will show the parameters I used and the results by Table.2.,Fig.2. In Fig.2, first raw is original images, second is correspondeces between image 1 and image 2, and third is mosaiced image by estimated homography . In second image in Fig.1 blue circles illustrate extracted Harris corners, all lines illustrate correspondence. Green lines illustrate lnliers, red lines illustrate outliers, and yellow lines illustrate four points initially chosen by RANSAC. And angles are calculated by MATLAB using command as follow:

$$angle(\ [1\ 0\ 0] * eig(H)\ ) * 180/pi$$

	1-2	2-3	3-4	4-5	5-6	6-7	7-8	8-9
$HarrisThreshold$	0.01							
$CorrespondingWindowSize$	21							
$SimilarityThreshold$ for SSD	0.3							
$SimilarityThreshold$ for NCC	0.7							
σ	3.0							
# of corners of each image	478 and 498	498 and 474	474and 440	440and 354	354and 368	368and 357	357and 346	346 and 309
# of correspondences	214	229	190	113	112	131	137	85
# of inliers	172	188	154	80	86	106	103	68
# of RANSAC samples (iterations)	9	8	9	16	11	9	12	9
Means of Projection Errors [pixel]	1.047	1.326	1.301	1.494	1.351	1.490	1.048	1.343
Angles between images[degree]	6.559	5.990	7.675	10.714	10.292	7.799	8.426	6.715

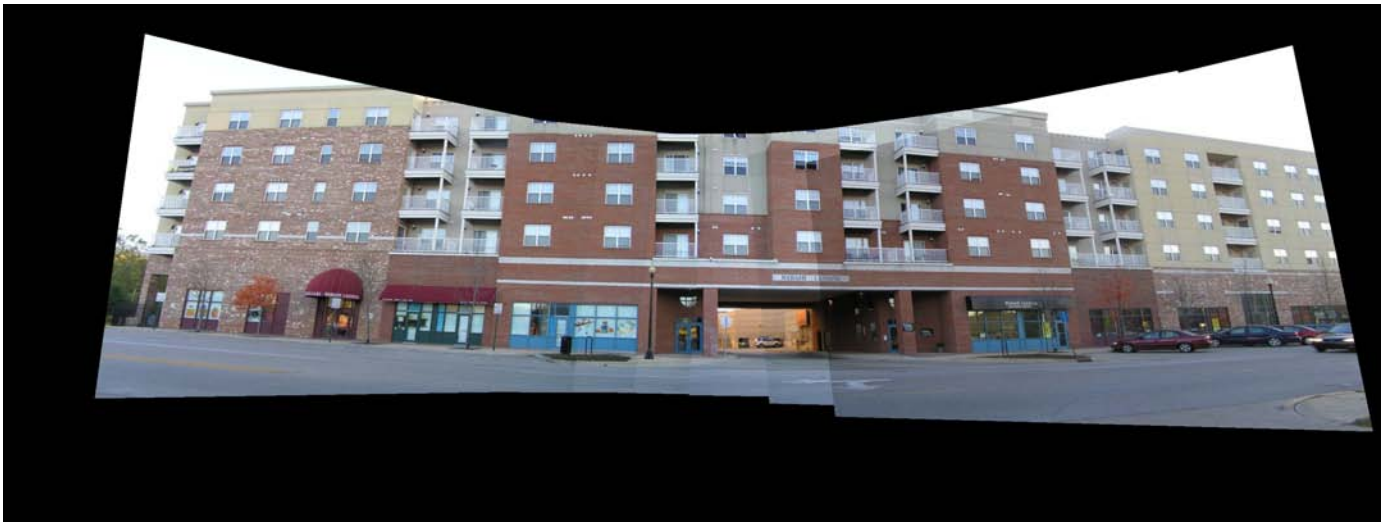
Table 1: Parameter and Experimental result

	1-2	2-3	3-4	4-5	5-6	6-7	7-8	8-9
$HarrisThreshold$	0.01							
$CorrespondingWindowSize$	21							
$SimilarityThreshold$ for SSD	0.3							
$SimilarityThreshold$ for NCC	0.7							
σ	3.0							
# of corners of each image	711 and 701	701and 692	692 and 699	699 and 686	686 and 685	685 and 703	703 and 646	646 and 649
# of correspondences	235	251	228	262	241	193	249	240
# of inliers	234	251	228	262	239	186	219	235
# of RANSAC samples (iterations)	2	1	1	1	2	3	6	2
Means of Projection Errors [pixel]	0.964	0.852	0.930	0.974	1.119	1.306	1.127	1.506
Angles between images [degree]	5.487	5.035	6.669	5.695	7.737	7.231	6.828	6.924

Table 2: Parameter and Experimental result

(a) Original Images

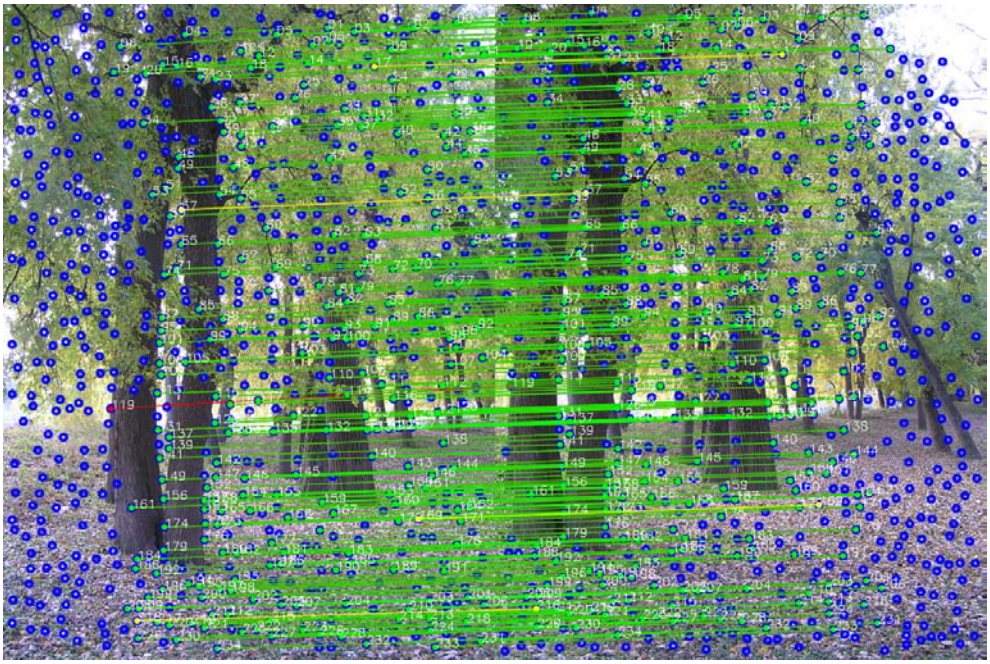
(b) A example of correspondences (between image1 and image2)



(c) Mosaiced Image

Figure 1: Apartment

(a) Original Images



(b) A example of correspondences (between image1 and image2)

(c) Mosaiced Image

Figure 2: Forest

3 C++ Source Code

I attach the source code.

```
#include <opencv/cv.h>
#include <opencv/highgui.h>
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <vector>
#include "CCvMat.hpp"

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

//Convert 2D image coordinates of CvPoint to Homogeneous Coordinates of CvMat
bool CvPnt2D32f2Hmg(CvPoint2D32f x, CvMat *tx) {
    if( tx->rows == 3 ){
        cvmSet(tx, 0, 0, x.x);
        cvmSet(tx, 1, 0, x.y);
        cvmSet(tx, 2, 0, 1);
        return true;
    }else{
        std::cerr << "Error in CvPnt2D32f2Hmg\n";
        return false;
    }
}

//Error Squared Projection Estimation
double SquaredProjectionError(std::pair<CvPoint2D32f, CvPoint2D32f> aPair,
        CvMat *aH) {
    CvMat *tx = cvCreateMat(3, 1, CV_32FC1);
    CvPnt2D32f2Hmg(aPair.first, tx);
    CCvMat tHx=CCvMat(aH)*CCvMat(tx);
    tHx=tHx*(1/cvmGet(tHx.m(),2,0));
    CvMat *txp = cvCreateMat(3, 1, CV_32FC1);
    CvPnt2D32f2Hmg(aPair.second, txp);
    double error = pow((tHx - CCvMat(txp)).n(), 2);
    cvReleaseMat(&tx);
    cvReleaseMat(&txp);
    return error;
}

// Solve A * h = 0 subject to |h|=1
bool GetPerspectiveTransform(int n, CvPoint2D32f *apX, CvPoint2D32f *apXp,
        CvMat *aH) {
    if (n >= 4) {
        int nParam = 9;
        CvMat *A = cvCreateMat(n * 2, nParam, 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->data.ptr + A->step * i * 2, tmpP, sizeof(float) * nParam
                    * 2);
        }
        CvMat *Ut = cvCreateMat(nParam, nParam, CV_32F);
        CvMat *Vt = cvCreateMat(nParam, nParam, CV_32F);
        CvMat *W = cvCreateMat(nParam, nParam, CV_32F);
        cvSVD((CCvMat(A).t() * CCvMat(A)).m(), W, Ut, Vt, CV_SVD_U_T
                |CV_SVD_V_T);//When A is M X N ( M < N ), then cvSVD did not work.
        cvReleaseMat(&A);
```

```
        CvMat *h = cvCreateMat(1, nParam, CV_32F);
        cvGetRow(Vt, h, Vt->rows - 1);
        memcpy(aH->data.ptr, h->data.ptr, sizeof(float) * nParam);
        cvReleaseMat(&h);
        cvReleaseMat(&W);
        cvReleaseMat(&Ut);
        cvReleaseMat(&Vt);
        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;
    double 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 scaleX = sqrt(2) / sqrt(meanX2 - meanX * meanX);
    double scaleY = sqrt(2) / sqrt(meanY2 - meanY * meanY);
    float tmp[] = { scaleX, 0, -meanX * scaleX, 0, scaleY, -meanY * scaleY, 0,
            0, 1 };
    memcpy(T->data.ptr, tmp, sizeof(float) * 9);
    for (uint i = 0; i < n; ++i) {
        CvMat *tx = cvCreateMat(3, 1, CV_32F);
        CvPnt2D32f2Hmg(x[i], tx);
        CvMat *Ttx = cvCreateMat(3, 1, CV_32F);
        cvmMul(T, tx, Ttx);
        xp[i].x = cvmGet(Ttx, 0, 0) / cvmGet(Ttx, 2, 0);
        xp[i].y = cvmGet(Ttx, 1, 0) / cvmGet(Ttx, 2, 0);
        cvReleaseMat(&Ttx);
        cvReleaseMat(&tx);
    }
    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()];
    CvMat *T = cvCreateMat(3, 3, CV_32F);
    for (uint i = 0; i < aPairs.size(); ++i)
        x[i] = aPairs[i].first;
    NormalizedPointSet(aPairs.size(), x, nx, T);
    delete[] x;

    //Normalize the coordinate of points on ImageB
    CvPoint2D32f *xp = new CvPoint2D32f[aPairs.size()];
    CvPoint2D32f *nxp = new CvPoint2D32f[aPairs.size()];
```

```
    CvMat *Td = cvCreateMat(3, 3, CV_32F);
    for (uint i = 0; i < aPairs.size(); ++i)
        xp[i] = aPairs[i].second;
    NormalizedPointSet(aPairs.size(), xp, nxp, Td);
    delete[] xp;

    //Calc Hd (Normalized H)
    CvMat *Hd = cvCreateMat(3, 3, CV_32F);
    if (GetPerspectiveTransform(aPairs.size(), nx, nxp, Hd)) {
        //nxp = Hd * nx
        delete[] nx;
        delete[] nxp;
        //Calc H = inv(Td) * Hd * T
        cvCopy((CCvMat(Td).i()*CCvMat(Hd)*CCvMat(T)).m(),aH);
        return true;
    } else {
        std::cerr << "Fail GetPerspectiveTransform in NormalizedDLT/n";
        delete[] nx;
        delete[] nxp;
        return false;
    }
}

//Bilinear interpolation
bool GetColorByBilinearInterpolation(IplImage *aImage, double aU, double aV,
        char aColor[3]) {
    int cn = aImage->nChannels;
    if ((aU >= 0) && (aU < aImage->width - 1) && (aV >= 0) && (aV
            < aImage->height - 1)) {
        int ui = (int) aU;
        int vi = (int) aV;
        double ud = aU - ui;
        double vd = aV - vi;
        int pos = ui * cn + vi * aImage->widthStep;
        for (int i = 0; i < cn; ++i) {
            double v[4];
            v[0] = (unsigned char) aImage->imageData[pos + i];
            v[1] = (unsigned char) aImage->imageData[pos + cn + i];
            v[2] = (unsigned char) aImage->imageData[pos + aImage->widthStep
                    + i];
            v[3] = (unsigned char) aImage->imageData[pos + cn
                    + aImage->widthStep + i];
            aColor[i] = (char) (v[0] * (1 - ud) * (1 - vd) + v[1] * (ud) * (1
                    - vd) + v[2] * (1 - ud) * (vd) + v[3] * (ud) * (vd)
                    + 0.499999);
        }
        return true;
    } else {
        for (int i = 0; i < cn; ++i) {
            aColor[i] = 0;
        }
        return false;
    }
}

//Transform apOrg by aH to apRes(Overlap Version)
bool WarpPerspective(IplImage *apOrg, IplImage *apRes, CvMat *aH) {
    CvMat *HInv = cvCreateMat(3, 3, CV_32F);
    cvInv(aH, HInv);
    int cn = apOrg->nChannels;
```

```
    int w=apRes->width;
    int h=apRes->height;
    for (int v = 0; v < h; ++v)
        for (int u = 0; u < w; ++u) {
            CvMat *X = cvCreateMat(3, 1, CV_32F);
            float tmp[] = { u, v, 1 };
            memcpy(X->data.ptr, tmp, sizeof(float) * 3);
            CvMat *Xp = cvCreateMat(3, 1, CV_32F);
            cvmMul(HInv, X, Xp);
            if (cvmGet(Xp, 2, 0) != 0) {
                double up, vp;//Restored Coordinates
                up = cvmGet(Xp, 0, 0) / cvmGet(Xp, 2, 0);
                vp = cvmGet(Xp, 1, 0) / cvmGet(Xp, 2, 0);
                char *color = new char[cn];
                if(GetColorByBilinearInterpolation(apOrg, up, vp, color)){
                    int pos = u * cn + v * apRes->widthStep;
                    for (int i = 0; i < cn; ++i)
                        apRes->imageData[pos + i] = color[i];
                }
                delete[] color;
            }
            cvReleaseMat(&X);
            cvReleaseMat(&Xp);
        }
    cvReleaseMat(&HInv);
    return true;
}

//Calculate NCC between pA and pB
// -1 <= NCC <= 1 ( if NCC=1 then pA and pB are exactly same)
double NCC(int aWinHalfSize, unsigned char *pA, unsigned char *pB) {
    int winWidth = 2 * aWinHalfSize + 1;
    int winSize = winWidth * winWidth;
    double avgA = 0, avgB = 0, avgA2 = 0, avgB2 = 0, avgAB = 0;
    for (int i = 0; i < winSize; ++i) {
        avgA += pA[i];
        avgB += pB[i];
        avgA2 += pA[i] * pA[i];
        avgB2 += pB[i] * pB[i];
        avgAB += pA[i] * pB[i];
    }
    avgA /= (double) winSize;
    avgB /= (double) winSize;
    avgA2 /= (double) winSize;
    avgB2 /= (double) winSize;
    avgAB /= (double) winSize;
    double varA = avgA2 - avgA * avgA;
    double varB = avgB2 - avgB * avgB;
    double covarAB = avgAB - avgA * avgB;
    return covarAB / sqrt(varA) / sqrt(varB);
}

//Calculate SSD between pA and pB
// 0 <= SSD <= 1 ( if SSD=0 then pA and pB are exactly same)
double SSD(int aWinHalfSize, unsigned char *pA, unsigned char *pB) {
    int winWidth = 2 * aWinHalfSize + 1;
    int winSize = winWidth * winWidth;
    double ssd = 0;
    for (int i = 0; i < winSize; ++i) {
        ssd += (pA[i] - pB[i]) * (pA[i] - pB[i]);
```

```
    }
    ssd = sqrt(ssd / (double) winSize) / 255;
    return ssd;
}

//Calculate SSD and NCC between at aPa in apImageA and at aPb in apImageB
//window size is 2*aWinHalfSize+1 X 2*aWinHalfSize+1
bool Sim(IplImage *apImageA, CvPoint2D32f aPa, IplImage *apImageB,
        CvPoint2D32f aPb, int aWinHalfSize, double *aSSD, double *aNCC) {
    int whs = aWinHalfSize;
    if ((aPa.x - whs) < 0 ||(aPa.x + whs) >= apImageA->width ||
        (aPa.y - whs) < 0 ||(aPa.y + whs) >= apImageA->height ||
        (aPb.x - whs) < 0 ||(aPb.x + whs) >= apImageB->width ||
        (aPb.y - whs) < 0 ||(aPb.y + whs) >= apImageB->height) {
        return false;
    }
    int winWidth = 2 * whs + 1;
    int winSize = winWidth * winWidth;
    unsigned char *a = new unsigned char[winSize];
    unsigned char *b = new unsigned char[winSize];
    int U0a = aPa.x - whs, V0a = aPa.y - whs;
    int U0b = aPb.x - whs, V0b = aPb.y - whs;

    //Extraction of templates
    for (int v = 0; v < winWidth; ++v) {
        int vStepA = (V0a + v) * apImageA->widthStep + U0a;
        int vStepB = (V0b + v) * apImageB->widthStep + U0b;
        for (int u = 0; u < winWidth; ++u) {
            a[v * winWidth + u] = (uchar) apImageA->imageData[vStepA + u];
            b[v * winWidth + u] = (uchar) apImageB->imageData[vStepB + u];
        }
    }
    *aSSD = SSD(whs, a, b);
    *aNCC = NCC(whs, a, b);
    delete[] a;
    delete[] b;
    return true;
}

//Find the corner in apImageB corresponding to the corner at aPa in apImageA
bool Corresponding(IplImage *apImageA, CvPoint2D32f aPa, IplImage *apImageB,
        int nPb, CvPoint2D32f *aPb, int *aIndexSSD, int *aIndexNCC,
        int aWinHalfSize, int aSearchDis, double aSSDTresh, double aNCCTresh) {
    double minssd = 10000000;
    *aIndexSSD = -1;
    double maxncc = -10000000;
    *aIndexNCC = -1;
    for (int i = 0; i < nPb; ++i) {
        double ssd, ncc;
        if (fabs(aPa.x - aPb[i].x) < aSearchDis && fabs(aPa.y - aPb[i].y)
                < aSearchDis)
            if (Sim(apImageA, aPa, apImageB, aPb[i], aWinHalfSize, &ssd, &ncc))
                if (ssd >= 0)
                    if (minssd > ssd) {
                        minssd = ssd;
                        *aIndexSSD = i;
                    }
                if (ncc >= -1)
                    if (maxncc < ncc) {
                        maxncc = ncc;
```

```
                    *aIndexNCC = i;
                }
    }
    if (minssd > aSSDTresh)     *aIndexSSD = -1;
    if (maxncc < aNCCTresh)     *aIndexNCC = -1;
    return true;
}

//Find correspondences
bool Correspondences(IplImage *apImageA, uint aNPa, CvPoint2D32f *aPa,
        IplImage *apImageB, uint aNPb, CvPoint2D32f *aPb, PointPairs *aPairs,
        int aWinHalfSize, int aSearchDis, double aSSDTresh, double aNCCTresh) {
    for (uint i = 0; i < aNPa; ++i) {
        int indexSSD, indexNCC;
        Corresponding(apImageA, aPa[i], apImageB, aNPb, aPb, &indexSSD,
                &indexNCC, aWinHalfSize, aSearchDis, aSearchDis, aNCCTresh);
        if ((indexSSD >= 0) && (indexSSD == indexNCC)) {
            //Back Matching
            int indexSSDb, indexNCCb;
            Corresponding(apImageB, aPb[indexSSD], apImageA, aNPa, aPa,
                    &indexSSDb, &indexNCCb, aWinHalfSize, aSearchDis,
                    aSearchDis, aNCCTresh);
            if ((indexSSDb >= 0) && (indexSSDb == indexNCCb) && ((int) i
                    == indexSSDb))
                aPairs->push_back(std::pair<CvPoint2D32f, CvPoint2D32f>(aPa[i],
                        aPb[indexSSD]));
        }
    }
    return true;
}

//Extract Harris Corner
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[w*h * 3];
    float *HarrisImage = new float[w*h];
    float *HarrisImageV = new float[w*h];
    int iws = apOrgImage->widthStep;
    int gws = w * 3;

    memset(GImage,0,sizeof(float)*w*h*3);
    memset(HarrisImage,0,sizeof(float)*w*h);
    memset(HarrisImageV,0,sizeof(float)*w*h);

    //Compute g(u,v)^T*g(u,v)
    for (int v = 1; v < h - 1; ++v) {
        unsigned char *img = (unsigned char*) apOrgImage->imageData;
        int vIStep = v * iws;
        int vGStep = v * gws;
        for (int u = 1; u < w - 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-1; ++v) {
    int vGStep = v * gws;
    for (int u = aWinHalfSize; u < w - aWinHalfSize-1; ++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-1; ++v) {
    int vHStep = v * w;
    for (int u = aMinDistance; u < w - aMinDistance-1; ++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;
}

//RANSAC
bool RANSAC(PointPairs aOrgPairs, PointPairs *aSelectedPairs, CvMat *aH,
        double aSigma = 1.0, PointPairs *aInitialPairs = 0, uint *aN = 0,
        double *errerSigma = 0) {
    int nPairsForEstimation = 4;
    CvMat *H = cvCreateMat(3, 3, CV_32FC1);
    float p = 0.99;
    float e = 0.5;
    float N = 1000;
    float t = sqrt(5.99) * aSigma;
    int sampleCount = 0;
    std::vector<uint> maxInliers;
    double sigmaMaxInliers = 0;
    int *pos = new int[nPairsForEstimation];
    while (N > sampleCount) {
        //Ramdam Sampling
        pos[0] = rand() % aOrgPairs.size();
        for (int i = 1; i < nPairsForEstimation; i++) {
            pos[i] = rand() % aOrgPairs.size();
            //Avoid choosing same index
            for (int j = 0; j < i; ++j)
                if (pos[j] == pos[i]){
                    i--;
                    break;
                }
        }
        PointPairs randam;
        for (int i = 0; i < nPairsForEstimation; ++i)
            randam.push_back(aOrgPairs[pos[i]]);
        double sigma = 0;//standard deviation of inliers
        if (NormalizedDLT(randam, H)) {//NormalizedDLT
            //Count inliers
            std::vector<uint> inliers;
            for (uint i = 0; i < aOrgPairs.size(); ++i) {
                double d2 = SquaredProjectionError(aOrgPairs[i], H);
                if (sqrt(d2) < t) {
                    inliers.push_back(i);
                    sigma += d2;
                }
            }
            sigma = sqrt(sigma / inliers.size());
            if ((maxInliers.size() < inliers.size()) ||(maxInliers.size()
                    == inliers.size()) && (sigmaMaxInliers > sigma)) {
                if (aInitialPairs != 0)     *aInitialPairs = randam;
                if (errerSigma != 0)        *errerSigma = sigmaMaxInliers;
                maxInliers = inliers;
                sigmaMaxInliers = sigma;
                cvCopy(H, aH);
                float outlierProb = 1 - ((float) maxInliers.size()
                        / (float) aOrgPairs.size());
                e = (e < outlierProb ?  e :  outlierProb);
```

```
                N = log(1 - p) / log(1 - pow((1 - e), nPairsForEstimation));
            }
        } else  std::cerr << "Bad samples\n";

        sampleCount += 1;
    }
    delete[] pos;
    aSelectedPairs->clear();
    for (uint i = 0; i < maxInliers.size(); ++i)
        aSelectedPairs->push_back(aOrgPairs[maxInliers[i]]);
    if (aN != 0) *aN = N;
    cvReleaseMat(&H);
    return true;
}

//Error Estimation
double MeanProjectionError(PointPairs aOrgPairs, CvMat *aH) {
    double error = 0;
    for (uint i = 0; i < aOrgPairs.size(); ++i)
        error += SquaredProjectionError(aOrgPairs[i], aH);
    return sqrt(error / aOrgPairs.size());
}

//Parameters
struct Params {
    double HarrisMaxMinRatio;
    int HarrisWin;
    int HariisMinDis;
    int CorrespoindingMaxDis;
    int CorrespoindingWin;
    double threshSSD;
    double threshNCC;
    double Sigma;
    Params(const char* aFileName) {
        read(aFileName);
    }
    bool read(const char* aFileName) {
        //Read parameter
        std::string tmp;
        std::ifstream ifs(aFileName);
        if (!ifs.fail()) {
            ifs >> tmp >> HarrisMaxMinRatio;
            ifs >> tmp >> HarrisWin;
            ifs >> tmp >> HariisMinDis;
            ifs >> tmp >> CorrespoindingMaxDis;
            ifs >> tmp >> CorrespoindingWin;
            ifs >> tmp >> threshSSD;
            ifs >> tmp >> threshNCC;
            ifs >> tmp >> Sigma;
            return true;
        } else {
            std::cerr << "Fail to open "<< aFileName << std::endl;
            return false;
        }
    }
};

bool Visualization(std::string fileName, IplImage *pOrgImageA,
        IplImage *pOrgImageB, uint corner_countA, CvPoint2D32f *cornersA,
        uint corner_countB, CvPoint2D32f *cornersB, PointPairs pairs,
```

```
    PointPairs selectedPairs, PointPairs initial, CvMat* H_RANSAC,
    uint Iteration, CvMat* H_NormalizedDLT) {
int wa = pOrgImageA->width;
int ha = pOrgImageA->height;
int sa = pOrgImageA->widthStep;
int wb = pOrgImageB->width;
int sb = pOrgImageB->widthStep;

//Preparation for Visualization
IplImage *pImage = cvCreateImage(cvSize(wa + wb, ha), IPL_DEPTH_8U, 3);
IplImage *pImage2 = cvCreateImage(cvSize(wa + wb, ha), IPL_DEPTH_8U, 3);
cvSet(pImage, cvScalar(0));
cvSet(pImage2, cvScalar(0));
for (int v = 0; v < ha; v++) {
    memcpy(&(pImage->imageData[v * pImage->widthStep]),
            &(pOrgImageA->imageData[v * sa]), wa * 3 );
    memcpy(&(pImage->imageData[v * pImage->widthStep + wa * 3]),
            &(pOrgImageB->imageData[v * sb]), wb * 3);
}
CvScalar ssdnccMacthed = CV_RGB(255, 0, 0);
CvScalar harrisCorner = CV_RGB(0, 0, 255);
CvScalar RANSACSelect = CV_RGB(0, 255, 0);
CvScalar RANSACInitial = CV_RGB(255, 255, 0);
CvScalar fontColor = CV_RGB(255, 255, 255);
CvFont font;
cvInitFont(&font, CV_FONT_HERSHEY_SCRIPT_SIMPLEX, 0.5, 0.5);

//Draw Detected Harris Corners
for (unsigned int i = 0; i < corner_countA; i++) {
    CvPoint pa = cvPointFrom32f(cornersA[i]);
    cvCircle(pImage, pa, 4, harrisCorner, 2, CV_AA);
}
for (unsigned int i = 0; i < corner_countB; i++) {
    CvPoint pb = cvPoint((int) cornersB[i].x + wa, (int) cornersB[i].y);
    cvCircle(pImage, pb, 4, harrisCorner, 2, CV_AA);
}

//Draw correspondences
for (unsigned int i = 0; i < pairs.size(); i++) {
    CvPoint pa = cvPointFrom32f(pairs[i].first);
    CvPoint pb = cvPoint((int) pairs[i].second.x + wa,
            (int) pairs[i].second.y);
    char c[3];
    sprintf(c, "%02d", i);
    cvPutText(pImage2, c, pa, &font, fontColor);
    cvPutText(pImage2, c, pb, &font, fontColor);
    cvCircle(pImage, pa, 3, ssdnccMacthed, -1, CV_AA);
    cvLine(pImage, pa, pb, ssdnccMacthed, 1, CV_AA);
    cvCircle(pImage, pb, 3, ssdnccMacthed, -1, CV_AA);
}
for (unsigned int i = 0; i < selectedPairs.size(); i++) {
    CvPoint pa = cvPointFrom32f(selectedPairs[i].first);
    CvPoint pb = cvPoint((int) selectedPairs[i].second.x + wa,
            (int) selectedPairs[i].second.y);
    cvCircle(pImage, pa, 3, RANSACSelect, -1, CV_AA);
    cvLine(pImage, pa, pb, RANSACSelect, 1, CV_AA);
    cvCircle(pImage, pb, 3, RANSACSelect, -1, CV_AA);
}
for (unsigned int i = 0; i < initial.size(); i++) {
    CvPoint pa = cvPointFrom32f(initial[i].first);
```

```
        CvPoint pb = cvPoint((int) initial[i].second.x + wa,
                (int) initial[i].second.y);
        cvCircle(pImage, pa, 3, RANSACInitial, -1, CV_AA);
        cvLine(pImage, pa, pb, RANSACInitial, 1, CV_AA);
        cvCircle(pImage, pb, 3, RANSACInitial, -1, CV_AA);
    }
    cvOr(pImage, pImage2, pImage);

    //Save Result
    cvSaveImage((fileName + "result"+ ".png").c_str(), pImage);
    std::ofstream ofs((fileName + "result.dat").c_str());
    ofs << "corner_countA= "<< corner_countA << std::endl;
    ofs << "corner_countB= "<< corner_countB << std::endl;
    ofs << "Matched Pairs= "<< pairs.size() << std::endl;
    ofs << "Inlier= "<< selectedPairs.size() << std::endl;
    (CCvMat(H_RANSAC) * (1 / cvmGet(H_RANSAC, 2, 2))).Write(&ofs, "H_RANSAC");
    ofs << "Error of RANSAC= "<< MeanProjectionError(selectedPairs, H_RANSAC)
            << std::endl;
    ofs << "Iteration of RANSAC= "<< Iteration << std::endl;
    (CCvMat(H_NormalizedDLT) * (1 / cvmGet(H_NormalizedDLT, 2, 2))).Write(&ofs,
            "H_NormalizedDLT");
    ofs << "Error of NormalizedDLT= "<< MeanProjectionError(selectedPairs,
            H_NormalizedDLT) << std::endl;
    cvWaitKey(30);
    cvReleaseImage(&pImage);
    cvReleaseImage(&pImage2);
    return true;
}

#define NIMAGE 9

int main(int argc, char **argv) {
    //  std::string fileName = "Apart/Apart";
    //  std::string fileName = "Forest1/Forest1";
    //  std::string fileName = "Forest2/Forest2";
    //  std::string fileName = "Forest3/Forest3";
        std::string fileName = "Forest4/Forest4";
    Params p((fileName + ".dat").c_str());

    //Get image
    IplImage *pOrgImg[NIMAGE];
    for (int i = 0; i < NIMAGE; ++i) {
        char tmp[2];
        sprintf(tmp, "%02d", i);
        std::cout << "Load "<< (fileName + tmp + ".png").c_str() << std::endl;
        pOrgImg[i] = cvLoadImage((fileName + tmp + ".png").c_str());
    }

    //Harris Corner Detection of Images
    unsigned int nCorner[NIMAGE];
    IplImage *pGry[NIMAGE];
    CvPoint2D32f *corners[NIMAGE];
    for (int i = 0; i < NIMAGE; ++i) {
        pGry[i] = cvCreateImage(cvGetSize(pOrgImg[i]), 8, 1);
        nCorner[i] = 10000;
        corners[i]
                = (CvPoint2D32f *) cvAlloc(nCorner[i] * sizeof(CvPoint2D32f));
        cvConvertImage(pOrgImg[i], pGry[i]);
        Harris(pGry[i], &(nCorner[i]), corners[i], p.HarrisWin, p.HariisMinDis,
                p.HarrisMaxMinRatio);
```

```
}

//Compute Transformation matrix H between each image pairs
PointPairs pairs[NIMAGE-1];
CvMat *H_RANSAC = cvCreateMat(3, 3, CV_32F);
CvMat *H_DLT[NIMAGE-1];//x(i+1) (pixel coordinate on image i+1) = H_DLT[i] * x(i) (pixel coordinate on image i)
for (int i = 0; i < NIMAGE - 1; ++i) {
    //Find correspondences
    Correspondences(pGry[i], nCorner[i], corners[i], pGry[i + 1], nCorner[i
            + 1], corners[i + 1], &pairs[i], p.CorrespoindingWin,
            p.CorrespoindingMaxDis, p.threshSSD, p.threshNCC);

    //RANSAC
    PointPairs selectedPairs, initial;
    uint Iteration;
    RANSAC(pairs[i], &selectedPairs, H_RANSAC, p.Sigma, &initial,
            &Iteration);

    //Normalized DLT
    H_DLT[i] = cvCreateMat(3, 3, CV_32F);
    NormalizedDLT(selectedPairs, H_DLT[i]);
    cvScale(H_DLT[i], H_DLT[i], 1 / cvmGet(H_DLT[i], 2, 2));
    char tmp[10];
    sprintf(tmp,"H_DLT[%d]",i);
    CCvMat(H_DLT[i]).disp(tmp);

    //Visualization
    char num[2];
    sprintf(num, "%02d", i);
    Visualization(fileName + num, pOrgImg[i], pOrgImg[i + 1], nCorner[i],
            corners[i], nCorner[i + 1], corners[i + 1], pairs[i],
            selectedPairs, initial, H_RANSAC, Iteration, H_DLT[i]);
}

/***************
 * Mosaicing
***************/
//Allocate memory for mosaiced image
int centerNum = NIMAGE/2;
int wc = pOrgImg[centerNum]->width;
int hc = pOrgImg[centerNum]->height;
int w = wc * 4;//Mosaiced Image width
int h = hc * 2;//Mosaiced Image height
IplImage *mosaic = cvCreateImage(cvSize(w, h), 8, 3);

//Allocate memory for transformation matrices
CvMat *H_MOC[NIMAGE];//xm (pixel coordinate on mosaiced image) = H_MOC[i] * x(i) (pixel coordinate on image i)
for (int i = 0; i < NIMAGE; ++i)
    H_MOC[i] = cvCreateMat(3, 3, CV_32F);

//H_MOC[centerNum] is only translation to (uOff,vOff)
int uOff = wc * 1.5;//u coordinate of the top-left corner of center image
int vOff = hc /2;    //v coordinate of the top-left corner of center image
cvSetIdentity(H_MOC[centerNum], cvScalar(1));
cvmSet(H_MOC[centerNum], 0, 2, uOff);
cvmSet(H_MOC[centerNum], 1, 2, vOff);

//Transform center image
WarpPerspective(pOrgImg[centerNum], mosaic, H_MOC[centerNum]);
```

```
    for (unsigned int i = centerNum+1; i < NIMAGE; i++) {
        //Transfomr right side images
        unsigned int right = i;
        cvCopy((CCvMat(H_MOC[right - 1])*CCvMat(H_DLT[right-1]).i()).m(), H_MOC[right]);
        WarpPerspective(pOrgImg[right], mosaic, H_MOC[right]);
        //Transfomr right side images
        unsigned int left = NIMAGE - i-1;
        cvCopy((CCvMat(H_MOC[left + 1]) * CCvMat(H_DLT[left]) ).m(), H_MOC[left]);
        WarpPerspective(pOrgImg[left], mosaic, H_MOC[left]);
    }

    //Save Result
    cvSaveImage((fileName + "MOSAIC.png").c_str(), mosaic);

    //Visualization
    cvNamedWindow("MOSAIC", -1);
    IplImage *mosaic2 = cvCreateImage(cvSize(w/2, h/2), 8, 3);
    cvResize(mosaic,mosaic2);
    cvShowImage("MOSAIC",mosaic2);
    cvWaitKey(-1);
    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;
}
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;
}
```

```
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();}
};

#endif /* CCVMAT_HPP_ */
```