Homography estimation

Introduction

Table of Contents

The estimation of an homography from coplanar points can be easily and precisely achieved using a Direct Linear Transform algorithm [4] [7] based on the resolution of a linear system.

Source code

The following source code that uses OpenCV is also available in **homography-dlt-opencv.cpp** file. It allows to estimate the homography between matched coplanar points. At least 4 points are required.

```
#include <iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/calib3d/calib3d.hpp>

cv::Mat homography_dlt(const std::vector< cv::Point2d
        > &x1, const std::vector< cv::Point2d > &x2)
{
  int npoints = (int)x1.size();
  cv::Mat A(2*npoints, 9, CV_64F, cv::Scalar(0));

  // We need here to compute the SVD on a (n*2)*9
       matrix (where n is
  // the number of points). if n == 4, the matrix has
       more columns
  // than rows. The solution is to add an extra line
       with zeros
  if (npoints == 4)
    A.resize(2*npoints+1, cv::Scalar(0));
```

```
// Since the third line of matrix A is a linear combination of the first and second lines
// (A is rank 2) we don't need to implement this third line
for(int i = 0; i < npoints; i++) {              // Update matrix A using eq. 23
  A.at<double>(2*i,3) = -x1[i].x;               // -xi_1
  A.at<double>(2*i,4) = -x1[i].y;               // -yi_1
  A.at<double>(2*i,5) = -1;                     // -1
  A.at<double>(2*i,6) =  x2[i].y * x1[i].x;     // yi_2 * xi_1
  A.at<double>(2*i,7) =  x2[i].y * x1[i].y;     // yi_2 * yi_1
  A.at<double>(2*i,8) =  x2[i].y;               // yi_2

  A.at<double>(2*i+1,0) =  x1[i].x;             // xi_1
  A.at<double>(2*i+1,1) =  x1[i].y;             // yi_1
  A.at<double>(2*i+1,2) =  1;                   // 1
  A.at<double>(2*i+1,6) = -x2[i].x * x1[i].x;   // -xi_2 * xi_1
  A.at<double>(2*i+1,7) = -x2[i].x * x1[i].y;   // -xi_2 * yi_1
  A.at<double>(2*i+1,8) = -x2[i].x;             // -xi_2
}

// Add an extra line with zero.
if (npoints == 4) {
  for (int i=0; i < 9; i ++) {
    A.at<double>(2*npoints,i) = 0;
  }
}

cv::Mat w, u, vt;
cv::SVD::compute(A, w, u, vt);

double smallestSv = w.at<double>(0, 0);
unsigned int indexSmallestSv = 0 ;
for (int i = 1; i < w.rows; i++) {
  if ((w.at<double>(i, 0) < smallestSv) ) {
    smallestSv = w.at<double>(i, 0);
    indexSmallestSv = i;
  }
}
```

```
  cv::Mat h = vt.row(indexSmallestSv);

  if (h.at<double>(0, 8) < 0) // tz < 0
    h *=-1;

  cv::Mat _2H1(3, 3, CV_64F);
  for (int i = 0 ; i < 3 ; i++)
    for (int j = 0 ; j < 3 ; j++)
      _2H1.at<double>(i,j) = h.at<double>(0, 3*i+j);

  return _2H1;
}

int main()
{
  std::vector< cv::Point2d > x1; // Points projected
       in the image plane linked to camera 1
  std::vector< cv::Point2d > x2; // Points projected
       in the image plane linked to camera 2

  std::vector< cv::Point3d > wX; // 3D points in the
       world plane

  // Ground truth pose used to generate the data
  cv::Mat c1tw_truth = (cv::Mat_<double>(3,1) << -0.1,
       0.1, 1.2); // Translation vector
  cv::Mat c1rw_truth = (cv::Mat_<double>(3,1) <<
       CV_PI/180*(5), CV_PI/180*(0), CV_PI/180*(45));
       // Rotation vector
  cv::Mat c1Rw_truth(3,3,cv::DataType<double>::type);
       // Rotation matrix
  cv::Rodrigues(c1rw_truth, c1Rw_truth);

  cv::Mat c2tc1 = (cv::Mat_<double>(3,1) << 0.01,
       0.01, 0.2); // Translation vector
  cv::Mat c2rc1 = (cv::Mat_<double>(3,1) << CV_PI/180*
       (0), CV_PI/180*(3), CV_PI/180*(5)); // Rotation
       vector
  cv::Mat c2Rc1(3,3,cv::DataType<double>::type); //
       Rotation matrix
  cv::Rodrigues(c2rc1, c2Rc1);

  // Input data: 3D coordinates of at least 4 coplanar
       points
  double L = 0.2;
  wX.push_back( cv::Point3d(  -L, -L, 0) ); // wX_0 (-
       L, -L, 0)^T
  wX.push_back( cv::Point3d( 2*L, -L, 0) ); // wX_1 (
       L, -L, 0)^T
  wX.push_back( cv::Point3d(   L,  L, 0) ); // wX_2 (
       L,  L, 0)^T
```

```
  wX.push_back( cv::Point3d(  -L,  L, 0) ); // wX_3 (-
      L,  L, 0)^T

  // Input data: 2D coordinates of the points on the
      image plane
  for(int i = 0; i < wX.size(); i++) {
    // Compute 3D points coordinates in the camera
      frame 1
    cv::Mat c1X = c1Rw_truth*cv::Mat(wX[i]) +
      c1tw_truth; // Update c1X, c1Y, c1Z
    // Compute 2D points coordinates in image plane
      from perspective projection
    x1.push_back( cv::Point2d( c1X.at<double>(0,
      0)/c1X.at<double>(2, 0),     // x1 = c1X/c1Z
                                c1X.at<double>(1,
      0)/c1X.at<double>(2, 0) ) ); // y1 = c1Y/c1Z

    // Compute 3D points coordinates in the camera
      frame 2
    cv::Mat c2X = c2Rc1*cv::Mat(c1X) + c2tc1; //
      Update c2X, c2Y, c2Z
    // Compute 2D points coordinates in image plane
      from perspective projection
    x2.push_back( cv::Point2d( c2X.at<double>(0,
      0)/c2X.at<double>(2, 0),     // x2 = c2X/c2Z
                                c2X.at<double>(1,
      0)/c2X.at<double>(2, 0) ) ); // y2 = c2Y/c2Z
  }

  cv::Mat _2H1 = homography_dlt(x1, x2);

  std::cout << "2H1 (computed with DLT):\n" << _2H1 <<
      std::endl;

  return 0;
}
```

Source code explained

First of all we inlude OpenCV headers that are requested to manipulate vectors and matrices.

```
#include <iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/calib3d/calib3d.hpp>
```

Then we introduce the function that does the homography estimation.

```
cv::Mat homography_dlt(const std::vector< cv::Point2d
      > &x1, const std::vector< cv::Point2d > &x2)
```

From a vector of planar points $\mathbf{x_1} = (x_1, y_1, 1)^T$ in image I_1 and a vector of matched points $\mathbf{x_2} = (x_2, y_2, 1)^T$ in image I_2 it allows to estimate the homography $^2\mathbf{H}_1$:

$$\mathbf{x_2} = {}^2\mathbf{H}_1\mathbf{x_1}$$

The implementation of the Direct Linear Transform algorithm to estimate $^2\mathbf{H}_1$ is done next. First, for each point we update the values of matrix A using equation (23). Then we solve the system $\mathbf{Ah} = 0$ using a Singular Value Decomposition of $\mathbf{A}$. Finaly, we determine the smallest eigen value that allows to identify the eigen vector that corresponds to the solution $\mathbf{h}$.

```
  int npoints = (int)x1.size();
  cv::Mat A(2*npoints, 9, CV_64F, cv::Scalar(0));

  // We need here to compute the SVD on a (n*2)*9
       matrix (where n is
  // the number of points). if n == 4, the matrix has
       more columns
  // than rows. The solution is to add an extra line
       with zeros
  if (npoints == 4)
    A.resize(2*npoints+1, cv::Scalar(0));

  // Since the third line of matrix A is a linear
       combination of the first and second lines
  // (A is rank 2) we don't need to implement this
       third line
  for(int i = 0; i < npoints; i++) {                 //
       Update matrix A using eq. 23
    A.at<double>(2*i,3) = -x1[i].x;                  // -
       xi_1
    A.at<double>(2*i,4) = -x1[i].y;                  // -
       yi_1
    A.at<double>(2*i,5) = -1;                        //
       -1
    A.at<double>(2*i,6) =  x2[i].y * x1[i].x;        //
       yi_2 * xi_1
    A.at<double>(2*i,7) =  x2[i].y * x1[i].y;        //
       yi_2 * yi_1
```

```
    A.at<double>(2*i,8) =  x2[i].y;                 // yi_2

    A.at<double>(2*i+1,0) =  x1[i].x;               // xi_1
    A.at<double>(2*i+1,1) =  x1[i].y;               // yi_1
    A.at<double>(2*i+1,2) =  1;                     // 1
    A.at<double>(2*i+1,6) = -x2[i].x * x1[i].x;     // -xi_2 * xi_1
    A.at<double>(2*i+1,7) = -x2[i].x * x1[i].y;     // -xi_2 * yi_1
    A.at<double>(2*i+1,8) = -x2[i].x;               // -xi_2
  }

  // Add an extra line with zero.
  if (npoints == 4) {
    for (int i=0; i < 9; i ++) {
      A.at<double>(2*npoints,i) = 0;
    }
  }

  cv::Mat w, u, vt;
  cv::SVD::compute(A, w, u, vt);

  double smallestSv = w.at<double>(0, 0);
  unsigned int indexSmallestSv = 0 ;
  for (int i = 1; i < w.rows; i++) {
    if ((w.at<double>(i, 0) < smallestSv) ) {
      smallestSv = w.at<double>(i, 0);
      indexSmallestSv = i;
    }
  }

  cv::Mat h = vt.row(indexSmallestSv);

  if (h.at<double>(0, 8) < 0) // tz < 0
    h *=-1;
```

From now the resulting eigen vector $\mathbf{h}$ that corresponds to the minimal eigen value of matrix $\mathbf{A}$ is used to update the homography ${}^{2}\mathbf{H}_{1}$.

```
  cv::Mat _2H1(3, 3, CV_64F);
  for (int i = 0 ; i < 3 ; i++)
    for (int j = 0 ; j < 3 ; j++)
      _2H1.at<double>(i,j) = h.at<double>(0, 3*i+j);
```

Finally we define the main function in which we will initialize the input data before calling the previous function.

```
int main()
```

First in the main we create the data structures that will contain the 3D points coordinates *wX* in the world frame, their coordinates in the camera frame 1 *c1X* and 2 *c2X* and their coordinates in the image plane *x1* and *x2* obtained after perspective projection. Note here that at least 4 coplanar points are requested to estimate the 8 parameters of the homography.

```
  std::vector< cv::Point2d > x1; // Points projected
       in the image plane linked to camera 1
  std::vector< cv::Point2d > x2; // Points projected
       in the image plane linked to camera 2

  std::vector< cv::Point3d > wX; // 3D points in the
       world plane
```

For our simulation we then initialize the input data from a ground truth pose that corresponds to the pose of the camera in frame 1 with respect to the object frame; in *c1tw_truth* for the translation vector and in *c1Rw_truth* for the rotation matrix. For each point, we compute their coordinates in the camera frame 1 *c1X* = (c1X, c1Y, c1Z). These values are then used to compute their coordinates in the image plane *x1* = (x1, y1) using perspective projection.

Thanks to the ground truth transformation between camera frame 2 and camera frame 1 set in *c2tc1* for the translation vector and in *c2Rc1* for the rotation matrix, we compute also the coordinates of the points in camera frame 2 *c2X* = (c2X, c2Y, c2Z) and their corresponding coordinates *x2* = (x2, y2) in the image plane.

```
  // Ground truth pose used to generate the data
  cv::Mat c1tw_truth = (cv::Mat_<double>(3,1) << -0.1,
       0.1, 1.2); // Translation vector
```

```
cv::Mat c1rw_truth = (cv::Mat_<double>(3,1) <<
     CV_PI/180*(5), CV_PI/180*(0), CV_PI/180*(45));
     // Rotation vector
cv::Mat c1Rw_truth(3,3,cv::DataType<double>::type);
     // Rotation matrix
cv::Rodrigues(c1rw_truth, c1Rw_truth);

cv::Mat c2tc1 = (cv::Mat_<double>(3,1) << 0.01,
     0.01, 0.2); // Translation vector
cv::Mat c2rc1 = (cv::Mat_<double>(3,1) << CV_PI/180*
     (0), CV_PI/180*(3), CV_PI/180*(5)); // Rotation
     vector
cv::Mat c2Rc1(3,3,cv::DataType<double>::type); //
     Rotation matrix
cv::Rodrigues(c2rc1, c2Rc1);

// Input data: 3D coordinates of at least 4 coplanar
     points
double L = 0.2;
wX.push_back( cv::Point3d(  -L, -L, 0) ); // wX_0 (-
     L, -L, 0)^T
wX.push_back( cv::Point3d( 2*L, -L, 0) ); // wX_1 (
     L, -L, 0)^T
wX.push_back( cv::Point3d(   L,  L, 0) ); // wX_2 (
     L,  L, 0)^T
wX.push_back( cv::Point3d(  -L,  L, 0) ); // wX_3 (-
     L,  L, 0)^T

// Input data: 2D coordinates of the points on the
     image plane
for(int i = 0; i < wX.size(); i++) {
  // Compute 3D points coordinates in the camera
     frame 1
  cv::Mat c1X = c1Rw_truth*cv::Mat(wX[i]) +
     c1tw_truth; // Update c1X, c1Y, c1Z
  // Compute 2D points coordinates in image plane
     from perspective projection
  x1.push_back( cv::Point2d( c1X.at<double>(0,
     0)/c1X.at<double>(2, 0),     // x1 = c1X/c1Z
                             c1X.at<double>(1,
     0)/c1X.at<double>(2, 0) ) ); // y1 = c1Y/c1Z

  // Compute 3D points coordinates in the camera
     frame 2
  cv::Mat c2X = c2Rc1*cv::Mat(c1X) + c2tc1; //
     Update c2X, c2Y, c2Z
  // Compute 2D points coordinates in image plane
     from perspective projection
  x2.push_back( cv::Point2d( c2X.at<double>(0,
     0)/c2X.at<double>(2, 0),     // x2 = c2X/c2Z
                             c2X.at<double>(1,
     0)/c2X.at<double>(2, 0) ) ); // y2 = c2Y/c2Z
```

```
}
```

From here we have initialized $\mathbf{x_1} = (x1, y1, 1)^T$ and $\mathbf{x_2} = (x2, y2, 1)^T$. We are now ready to call the function that does the homography estimation.

```
cv::Mat _2H1 = homography_dlt(x1, x2);
```

Resulting homography estimation

If you run the previous code, it we produce the following result that shows that the estimated pose is equal to the ground truth one used to generate the input data:

```
2H1 (computed with DLT):
[0.5425233873981674, -0.04785624324415742,
       0.03308292557420141;
 0.0476448024215215, 0.5427592708789931,
       0.005830349194436123;
 -0.02550335176952741, -0.005978041062955012,
       0.6361649706821216]
```