## Contents

1 Homography Library Hierarchical Index ........................................... 1
   1.1 Homography Library Class Hierarchy ........................................ 1

2 Homography Library Class Index .................................................. 3
   2.1 Homography Library Class List ............................................... 3

3 Homography Library File Index .................................................... 5
   3.1 Homography Library File List ................................................ 5

4 Homography Library Class Documentation .................................... 7
   4.1 _motion_params Struct Reference .......................................... 7
   4.2 ProjectiveHomography Class Reference .................................. 9

5 Homography Library File Documentation ....................................... 15
   5.1 Homography.hpp File Reference ............................................. 15
Chapter 1

Homography Library Hierarchical Index

1.1 Homography Library Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

- _motion_params .......................................................... 7
- ProjectiveHomography ................................................. 9
Chapter 2

Homography Library Class Index

2.1 Homography Library Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

- _motion_params (Motion parameters that define a homography matrix) ........................................... 7
- ProjectiveHomography (Homography computation functions) .................................................. 9
Chapter 3

Homography Library File Index

3.1 Homography Library File List

Here is a list of all files with brief descriptions:

Homography.hpp ................................................................. 15
Chapter 4

Homography Library Class Documentation

4.1 _motion_params Struct Reference

Motion parameters that define a homography matrix.
#include <Homography.hpp>

Public Attributes

- Matrix<3, 3> rotation
- Vector<3> scaledTranslation
- Vector<3> normal

4.1.1 Detailed Description

Motion parameters that define a homography matrix.

4.1.2 Member Data Documentation

4.1.2.1 Matrix<3,3> _motion_params::rotation

Rotation between camera frames.

4.1.2.2 Vector<3> _motion_params::scaledTranslation

Translation between camera frames, divided by the distance from plane of image features to the optical center of the first camera.

4.1.2.3 Vector<3> _motion_params::normal

Unit normal to the plane of image features defined in first camera frame.

The documentation for this struct was generated from the following file:
• Homography.hpp
4.2 ProjectiveHomography Class Reference

Homography computation functions.
#include <Homography.hpp>

Public Member Functions

- ProjectiveHomography (int nFeatures, homography_method_t method)
- ~ProjectiveHomography ()
- int compute (MatrixBase<> &p2, MatrixBase<> &p1, Matrix<3, 3> &Hpn, VectorBase<> &sc, Matrix<3, 3> &dev, int maxItr=100)

4.2.1 Detailed Description

Homography computation functions.

Homography relates pixel coordinates of coplanar feature points when recorded by a camera from two different poses. If the 3D homogeneous coordinates of a feature point on a plane are q1 = [x1/z1, y1/z1, 1]' and q2 = [x2/z2, y2/z2, 1]' relative to the camera frame at position 1 (I1) and 2 (I2), respectively, we get the following euclidean relationship:

\[ q_2 = \left(\frac{z_1}{z_2}\right) \cdot H_e \cdot q_1, \]

where \( \left(\frac{z_1}{z_2}\right) \) is called the ‘depth ratio’, and ‘He’ is the ‘Euclidean homography’ defined as:

\[ H_e = (R + \left(\frac{t}{d}\right) \cdot transpose(n)). \]

Here ‘R’ is the 3x3 rotation matrix that defines the transformation I1 -> I2, ‘t’ is the translation between I1 and I2 defined in frame I2, ‘d’ and ‘n’ are the distance, and the unit normal to the plane of feature points, respectively, when camera is at I1. Defined in terms of pixel coordinates p1 (= C * q1) and p2 (= C * q2), for a calibrated camera with ‘C’ as the internal calibration matrix, we get the following relationship:

\[ p_2 = \left(\frac{z_1}{z_2}\right) \cdot H_p \cdot p_1, \]

where Hp is called ‘projective homography’. The projective and Euclidean homographies are hence related as follows:

\[ H_p = C \cdot H_e \cdot inverse(C). \]

Pay attention to these definitions when you use functions provided in this class. This class primarily provides methods to compute ~projective~ homography from image coordinates of planar feature points, and methods to decompose ~Euclidean~ homography into motion parameters R, t/d and n. Either the least squares technique (fast), or the optimal algorithm (slow) developed by Kanatani et al. can be utilized to compute the homography matrix, if the features are coplanar. If the feature points are non-coplanar, an implementation of a method developed by Ezio Malis is provided that allows computation of homography of a ‘virtual’ plane. The algorithm given in Faugeras’ book and Shastry’s book are implemented for decomposition of the homography matrix.

The implementation of optimal homography computation algorithm is not my work. It is a minor adaption of the original implementation by Naoya Ohta and Shimizu Yoshiyuki (1999/2/25) of Computer Science Dept., Gunma University.

References:


Example Program:

```
// Homography.t.cpp - Example program for homography and decomposition
// Project : Computer Vision Utilities (cvutils)
// Author : Vilas Kumar Chitrakaran (cvilas@ces.clemson.edu)

#include "Homography.hpp"
#include "QMath/Transform.hpp"
#include <iostream>
using namespace std;

// main
int main()
{
    Matrix<3, 8> p2; // pixel coordinates from second frame
    Matrix<3, 8> p1; // pixel coordinates from first frame
    motion_params_t motion[2]; // recovered motion parameters
    Matrix<3, 3> Hn, Gn; // Euclidean and projective Homography
    Matrix<3, 3> A; // Camera intrinsic calibration matrix
    Vector<8> scale; // scale factor

    // initialize for virtual parallax method
    ProjectiveHomography homography(8, e_vp);
    A = 2400, 0, 360, 0, 2400, 240, 0, 0, 1;
    p1 = 310, 316, 577, 573, 359, 365, 550, 547,
         45, 288, 267, 58, 44, 287, 271, 53,
         1, 1, 1, 1, 1, 1, 1, 1;
    p2 = 315.011, 334.141, 590.244, 575.653, 363.81, 382.601, 564.053, 549.991,
         59.5576, 302.665, 264.716, 58.2838, 56.1194, 298.789, 271.933, 54.8274,
         1, 1, 1, 1, 1, 1, 1, 1;
    Matrix<3,3> tmp;
    if(homography.compute(p2, p1, Gn, scale, tmp) == -1) {
        cerr << "Homography determination failed." << endl;
    
Generated on Sat Aug 26 14:52:11 2006 for Homography Library by Doxygen
return -1;
}

// get Euclidean Homography
Hn = inverse(A) * Gn * A;

// decompose homography and recover motion parameters
if(decomposeHomography(Hn, motion[0], motion[1]) == -1) {
    cerr << "Homography Decomposition failed." << endl;
    return -1;
}

cout << "--------- Recovered motion ---------" << endl;
for(int i = 0; i < 2; ++i) {
    cout << "--------- Solution " << i << " ---------" << endl;
    cout << "Rotation: " << endl << motion[i].rotation << endl;
    cout << "Scaled Translation: " << transpose(motion[i].scaledTranslation) << endl;
    cout << "normal: " << transpose(motion[i].normal) << endl;
}
return 0;

//==============================================================================
// decomposeHomography.t.cpp - Example program for homography decomposition
// Project : Computer Vision Utilities (cvutils)
// Author : Vilas Kumar Chitrakaran (cvilas@ces.clemson.edu)
//==============================================================================
#include "Homography.hpp"
#include "QMath/Transform.hpp"
#include <iostream>
using namespace std;
Matrix<3,3> computeRotationMatrix(double r, double p, double y);
// compute rotation matrix from euler angles

//========================================================================
// main - demonstrates homography decomposition
//========================================================================
int main()
{
    Matrix<3,3> H, R;
    Vector<3> t, n;
    double d;
    motion_params_t motion[2];

    // initialize a homography matrix
    R = computeRotationMatrix(M_PI/6.0, 0.0, -M_PI/3.0); // rotate 30°, 0°, -60°
    t = -0.1, 0.0, 0.0; // translate
    d = 1; // distance to feature plane
    n = 1.0, 0.0, 0.0; // normal to feature plane
    H = R + (1.0/d) * t * transpose(n); // Euclidean homography
    H = (1.0/H(3,3)) * H;

    // decompose homography - two solutions
    decomposeHomography(H, motion[0], motion[1]);

cout << "--------- Actual motion parameters ---------" << endl;
cout << "Rotation: " << endl << R << endl;
cout << "Scaled Translation: " << (1.0/d) * transpose(t) << endl;
cout << "normal: " << transpose(n) << endl;
    
    for(int i = 0; i < 2; ++i) {
        cout << "--------- SHASTRY'S SOLUTION " << i+1 << " ---------" << endl;
    }
    
    return 0;
}
cout << "Rotation: " << endl << motion[i].rotation << endl;
cout << "Scaled Translation: " << transpose(motion[i].scaledTranslation) << endl;
cout << "Normal: " << transpose(motion[i].normal) << endl;
}

double tmp1, tmp2;
tmp1 = dotProduct(motion[0].normal,n);
tmp2 = dotProduct(motion[1].normal,n);

if(tmp1 > tmp2)
  cout << "** Correct solution: 1 **" << endl << endl;
else
  cout << "** Correct solution: 2 **" << endl << endl;

// decompose homography - normal known
decomposeHomography(H, n, motion[0]);
cout << "======== FAUGERAS' SOLUTION with known normal =========" << endl;
cout << "Rotation: " << endl << motion[0].rotation << endl;
cout << "Scaled Translation: " << transpose(motion[0].scaledTranslation) << endl;
cout << "Normal: " << transpose(motion[0].normal) << endl;
return 0;
}

//========================================================================
// computeRotationMatrix
//========================================================================
Matrix<3,3> computeRotationMatrix(double r, double p, double y)
{
  Matrix<3,3> R;
  Transform t;
  t = rpyRotation(r,p,y);
  t.getSubMatrix(1,1,R);
  return R;
}

4.2.2 Constructor & Destructor Documentation

4.2.2.1 ProjectiveHomography::ProjectiveHomography (int nFeatures,
                                                       homography_method_t method)

Default constructor. Allocates memory required for computations.

Parameters:

nFeatures The number of feature points used in the computation of Homography.

method Specify method for computing homography (least squares method, kanatani algorithm, or Malis' virtual parallax method).

4.2.2.2 ProjectiveHomography::~ProjectiveHomography ()

Default destructor frees allocated resources.
4.2.3 Member Function Documentation

4.2.3.1 int ProjectiveHomography::compute (MatrixBase<> & p2, MatrixBase<> & p1, Matrix<3, 3> & Hpn, VectorBase<> & sc, Matrix<3, 3> & dev, int maxItr = 100)

Compute the normalized projective homography from homogeneous image coordinates such that $p_2 = sc \times Hpn \times p_1$.

Parameters:
- $p_2, p_1$ Sets of homogeneous image coordinates arranged as columns of a matrix. IMPORTANT NOTE: If using the virtual parallax method, it will be assumed that the first three columns of this matrix define the virtual plane (input).
- $Hpn$ The estimated projective homography, normalized by the element at (3,3), i.e., $Hpn = [1.0/Hp(3,3)] \times Hp$. (output)
- $sc$ The scale factor for every feature point (output). IMPORTANT NOTE: If using the virtual parallax algorithm, only the first three elements of this vector (corresponding to features that define the virtual plane) will have valid values. This is because the scale factor for all other features cannot be accurately determined without using information from decomposition of the homography.
- $dev$ Deviation matrix, valid only when using the optimal algorithm (output).
- $maxItr$ Maximum number of iteration allowed for convergence, valid only when using the optimal algorithm (input).

Returns:
- number of iterations if using the optimal algorithm, 1 if using the least squares method, and -1 on error.

The documentation for this class was generated from the following file:

- Homography.hpp
Chapter 5

Homography Library File Documentation

5.1 Homography.hpp File Reference

#include "QMath/Vector.hpp"
#include "QMath/RowVector.hpp"
#include "QMath/GSLCompat.hpp"
#include <gsl/gsl_linalg.h>
#include <gsl/gsl_combination.h>
#include <gsl/gsl_cblas.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_eigen.h>
#include <math.h>

Classes

- struct _motion_params
  
  Motion parameters that define a homography matrix.

- class ProjectiveHomography
  
  Homography computation functions.

Typedefs

- typedef _motion_params motion_params_t
- typedef enum _homography_method homography_method_t
Enumerations

- `enum _homography_method { e_ls = 1, e_kk, e_vp }`

  Method used to compute homography matrix.

Functions

- `int decomposeHomography (const Matrix< 3, 3 > &Hen, motion_params_t &motion0, motion_params_t &motion1)`
- `int decomposeHomography (const Matrix< 3, 3 > &Hen, const Vector< 3 > &n, motion_params_t &motion)`

5.1.1 Typedef Documentation

5.1.1.1 `typedef struct _motion_params motion_params_t`

5.1.1.2 `typedef enum _homography_method homography_method_t`

5.1.2 Enumeration Type Documentation

5.1.2.1 `enum _homography_method`

  Method used to compute homography matrix.

  Enumerator:
  - `e_ls` least squares method (for coplanar features)
  - `e_kk` Kanatani optimal algorithm (for coplanar features).
  - `e_vp` Virtual parallax method (Malis) (for non-coplanar features).

5.1.3 Function Documentation

5.1.3.1 `int decomposeHomography (const Matrix< 3, 3 > &Hen, motion_params_t &motion0, motion_params_t &motion1)`

Decompose a Euclidean homography to obtain the motion parameters. Two physically possible solutions are returned. If the unit normal vector is known (see class description), the solution to decomposition is unique, in which case use the subsequent method with the same name to compute the motion parameters more accurately.

IMPORTANT NOTE: This method does not produce valid solutions when there is no motion, or when the motion between camera frames is a pure rotation.

Parameters:

  - `Hen` The normalized Euclidean homography `[Hen = inverse(C) * Hpn * C, where C is the camera intrinsic calibration matrix]` (input).
  - `motion0`
  - `motion1` The two physically possible solutions (output).

Returns:

  0 on success, -1 on error.
5.1.3.2 int decomposeHomography (const Matrix<3, 3> & Hen, const Vector<3> & n, motion_params_t & motion)

If the unit normal vector is known, this method can be used to decompose Euclidean homography more accurately than the previous method with the same name.

**Parameters:**
- **Hen** The normalized Euclidean homography $\text{Hen} = \text{inverse}(C) \ast \text{Hpn} \ast C$, where $C$ is the camera intrinsic calibration matrix (input).
- **n** The known unit normal vector (see class description) (input).
- **motion** The motion parameters from homography decomposition (output)

**Returns:**
- 0 on success, -1 on error.
Index

~ProjectiveHomography
  ProjectiveHomography, 12
  Homography.hpp, 16
  _motion_params, 7
  normal, 7
  rotation, 7
  scaledTranslation, 7
compute
  ProjectiveHomography, 13
decomposeHomography
  Homography.hpp, 16
e_kk
  Homography.hpp, 16
e_ls
  Homography.hpp, 16
e_vp
  Homography.hpp, 16
Homography.hpp, 15
  _homography_method, 16
  decomposeHomography, 16
  e_kk, 16
  e_ls, 16
  e_vp, 16
  homography_method_t, 16
  motion_params_t, 16
homography_method_t
  Homography.hpp, 16
motion_params_t
  Homography.hpp, 16
normal
  _motion_params, 7
ProjectiveHomography, 9
  ProjectiveHomography, 12
ProjectiveHomography
  ~ProjectiveHomography, 12
  compute, 13
  ProjectiveHomography, 12
rotation
  _motion_params, 7
  scaledTranslation
  _motion_params, 7