
openMVG Documentation

Release 1.2

Pierre MOULON & al.

Oct 08, 2017

Contents

1	openMVG libraries	1
2	openMVG samples	29
3	openMVG softwares & tools	33
4	patented	61
5	dependencies	63
6	third_party	65
7	FAQ	67
8	Bibliography	69
9	Why another library	71
10	openMVG overview	73
11	Acknowledgements	75
12	License	77
13	Dependencies	79
14	Indices and tables	81
	Bibliography	83

openMVG provide a collection of tiny libraries that allow to solve computer vision problems and build complete SfM pipeline.

image

Image Container

OpenMVG `Image<T>` class is a generic image container based on an Eigen aligned row-major matrix of template pixel type `T`. Images can store grayscale, RGB, RGBA or custom data.

`Image<T>` provide basic pixel read and write operation.

See examples from `openMVG/images/image_test.cpp`:

```
// A 8-bit gray image:
Image<unsigned char> grayscale_image_8bit;

// A 32-bit gray image:
Image<double> grayscale_image_32bit;

// Multichannel image: (use pre-defined pixel type)

// A 8-bit RGB image:
Image<RGBColor> rgb_image_8bit;
Image<Rgb<unsigned char> > rgb_image2_8bit;

// 8-bit RGBA image
Image<RGBAColor> rgba_image_8bit;
Image<Rgba<unsigned char> > rgba_image2_8bit;

// 32 bit RGB image:
Image<Rgb<double> > rgb_image_32bit;
```

Image I/O

Loading and writing of 8 bits (gray and color) images are supported in the following formats:

- ppm/pgm,
- jpeg,
- png,
- tiff.

See examples from openMVG/images/image_IO_test.cpp:

```
// Read a grayscale image (if conversion need, it is done on the fly)
Image<unsigned char> gray_image;
bool bRet = ReadImage("Foo.imgExtension", &gray_image);

// Read a color image
Image<RGBColor> rgb_image_gray;
bool bRet = ReadImage("Foo.imgExtension", &rgb_image);
```

Drawing operations

The following operations are available:

- lines,
- circles,
- ellipses.

See examples from openMVG/images/image_drawing_test.cpp:

```
Image<unsigned char> image(10,10);
image.fill(0);

// Pixel access is done as matrix (row, line)
int row = 2;
int column = 4;
image(row, column) = 127;

// Horizontal scanline
DrawLine( 0, 5, w-1, 5, 255, &image);

// Circle of radius 3 and center (5,5)
const int radius = 3;
const int x = 5, y = 5;
DrawCircle(x, y, radius, (unsigned char)255, &image);

// Ellipse of center (5,5) and (3,0)
const int radius1 = 3, radius2 = 1, angle = 0;
const int x = 5, y = 5;

DrawEllipse(x, y, radius1, radius2, (unsigned char)255, &image, (double)angle);

// Example with a RGB image
Image<RGBColor> imageRGB(10,10);
DrawCircle(x, y, radius, RGBColor(255,0,0), &imageRGB);
```

numeric

This module provides math and linear algebra utils that relies on *[Eigen]* library. Eigen is a C++ template library for linear algebra.

Basic idea is to provide to openMVG :

- a high level memory container for matrices and vectors,
- an easy matrices and vectors manipulation,
- a collection of numeric solvers and related algorithms.

Vector, Matrix containers

OpenMVG redefines some Eigen basis type (points, vectors, matrices) for code consistency and clarity:

- `Vec2` a single 2d point stored as a column matrix (x,y),
- `Vec3` a single 3d point stored as a column matrix (x,y,z),
- `Vec2f`, `Vec3f` float version.
- `Vec` a vector of value (double precision),
- `Vecf` a vector of floating point value,
- `Mat` the generic matrix container,
- `Mat2X` a collection of 2d points stored by column,
- `Mat3X` a collection of 3d points stored as column.

Note: Default memory alignment is column major.

```
// Create a set of 2D points store as column
Mat2X A(2, 5);
A << 1, 2, 3, 4, 5,
     6, 7, 8, 9, 10;
A.col(); // return a column vector : (1,6)^T
A.row(); // return a row vector : (1,2,3,4,5)
```

Linear algebra

- SVD/QR/LU decomposition.

To know more

Please visit: http://eigen.tuxfamily.org/dox/group__QuickRefPage.html

features

This module provides generic container for features and associated descriptors.

Features

Provide basic structure and IO to store Point based features.

Classes to store point characteristics:

- **PointFeature**
 - Store the position of a feature (x,y).
- **SIOPointFeature**
 - Store the position, orientation and scale of a feature (x,y,s,o).

Descriptors

Provide basic structure and IO for descriptor data.

- **template <typename T, std::size_t N> class Descriptor.**
 - Store N value(s) of type T as contiguous memory.

```
// SIFT like descriptor
using siftDescriptorData = Descriptor<float, 128>;

// SURF like descriptor
using surfDescriptorData = Descriptor<float, 64>;

// Binary descriptor (128 bits)
using binaryDescriptor_bitset = Descriptor<std::bitset<128>, 1> binaryDescriptor_
->bitset;
// or using unsigned chars
using binaryDescriptor_uchar = Descriptor<unsigned char, 128/sizeof(unsigned char)>;
```

KeypointSet

Store a collection of features and their associated descriptors: `template<typename FeaturesT, typename DescriptorsT> class KeypointSet`. Basic IO is provided.

```
// Define SIFT Keypoints:

// Define the SIFT descriptor [128 floating point value]
using DescriptorT = Descriptor<float, 128>;

// Use SIFT compatible features (scale, orientation and position)
using FeatureT = SIOPointFeature;

// Describe what a collection of local feature is for a given image:
using FeatsT = std::vector<FeatureT>;
using DescsT = std::vector<DescriptorT>;
```



```
// Link features and their descriptors as a collection:
using KeypointSetT = KeypointSet<FeatsT, DescsT>;
```

cameras

This module provides different camera models.

Pinhole camera model

A camera could be approximated by a projective model, often called pinhole projection. The simplest representation of a camera is a light sensible surface (sensor): an image plane, a lens (projective projection) at a given position and orientation in space.

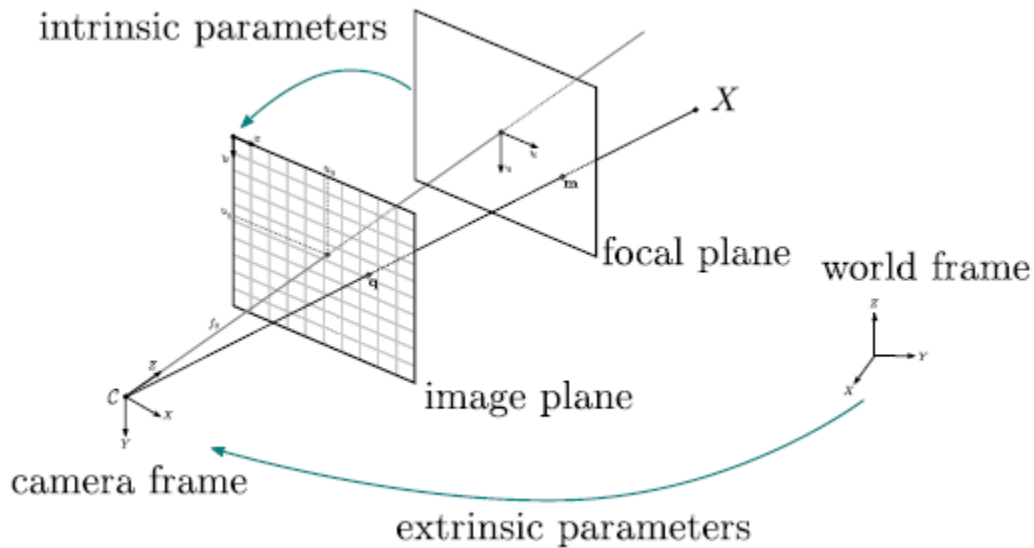


Fig. 1.1: The pinhole camera model. An oriented central projective camera.

The pinhole camera geometry models the projective camera with two sub-parametrizations, intrinsic and extrinsic parameters. Intrinsic parameters model the optic component (without distortion) and extrinsic model the camera position and orientation in space. This projection of the camera is described as:

$$P_{3 \times 4} = K[R|t] = \begin{bmatrix} f * k_u & 0 & c_u \\ 0 & f * k_v & c_v \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R_{3 \times 3} & t_x \\ t_y \\ t_z \end{bmatrix}$$

- Intrinsic parameters $[f; cu; cv]$:
 - ku, kv : scale factor relating pixels to distance (often equal to 1),
 - f : the focal distance (distance between focal and image plane),
 - cu, cv : the principal point, which would be ideally in the centre of the image.
- Extrinsic parameters $[R|t] = [R] - RC$:
 - R : the rotation of the camera to the world frame,

- t : the translation of the camera. t is not the position of the camera. It is the position of the origin of the world coordinate system expressed in coordinates of the camera-centred coordinate system. The position, C , of the camera expressed in world coordinates is $C = -R^{-1}t = -R^T t$ (since R is a rotation matrix).

A 3D point is projected in a image with the following formula (homogeneous coordinates):

$$x_i = PX_i = K[R|t]X_i$$
$$\begin{bmatrix} u_i \\ v_i \\ w_i \end{bmatrix} = \begin{bmatrix} f * k_u & & c_u \\ & f * k_v & c_v \\ & & 1 \end{bmatrix} \begin{bmatrix} R_{3 \times 3} & \begin{matrix} t_x \\ t_y \\ t_z \end{matrix} \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \\ Z_i \\ W_i \end{bmatrix}$$

OpenMVG Pinhole camera models

- Pinhole intrinsic
 - Pinhole_Intrinsic : public IntrinsicBase
 - * classic pinhole camera (Focal + principal point and image size).
 - Pinhole_Intrinsic_Radial_K1 : public Pinhole_Intrinsic
 - * classic pinhole camera (Focal + principal point and image size) + radial distortion defined by one factor.
 - * can add and remove distortion
 - Pinhole_Intrinsic_Radial_K3 : public Pinhole_Intrinsic
 - * classic pinhole camera (Focal + principal point and image size) + radial distortion by three factors.
 - * can add and remove distortion
 - Pinhole_Intrinsic_Brown_T2 : public Pinhole_Intrinsic
 - * classic pinhole camera (Focal + principal point and image size) + radial distortion by three factors + tangential distortion by two factors.
 - * can add and remove distortion
 - Pinhole_Intrinsic_Fisheye : public Pinhole_Intrinsic
 - * classic pinhole camera (Focal + principal point and image size) + fish-eye distortion by four factors.
 - * can add and remove distortion
- Simple pinhole camera models (intrinsic + extrinsic(pose))

```
// Setup a simple pinhole camera at origin
// Pinhole camera P = K[R|t], t = -RC
Mat3 K;
K << 1000, 0, 500,
    0, 1000, 500,
    0, 0, 1;
PinholeCamera cam(K, Mat3::Identity(), Vec3::Zero());
```

multiview

The multiview module consists of:

- a collection of solvers for 2 to n-view geometry constraints that arise in multiple view geometry,
- a generic framework “Kernel” that can embed these solvers for robust estimation.

First accessible solvers are listed and explained and the “Kernel” concept is documented.

2-view solvers (2d-2d correspondences)

openMVG provides solver for the following geometric estimation:

- affine,
- homographic,
- fundamental,
 - 7 to n pt,
 - 8 to n pt (Direct Linear Transform) [HZ].
- essential,
 - 8 to n pt (Direct Linear Transform) [HZ],
 - 5pt + intrinsic [Stewenius], [Nister].

N-View geometry estimation

- Triangulation
 - 2 to n view (Direct Linear Transform),
 - 2 to n view (Iterated least square).
- Rotation averaging
 - L2 (sparse) [Martinec],
 - L1 (sparse) [Chatterjee].
- Translation averaging
 - L2 Chordal [Kyle2014],
 - SoftL1 ‘approximation of the LInf method of [GlobalACsfM]’.

Homography matrix:

The homography matrix maps the relation between two projections of a plane: *Figure*.

H is a (3 x 3) matrix that links coordinates in left and right images with the following relation.

$$x'_i = Hx_i \quad (1.1)$$

OpenMVG implementation follows the DLT (Direct Linear Transform) explained in [HZ] book: H can be estimated from 4 to n corresponding points.

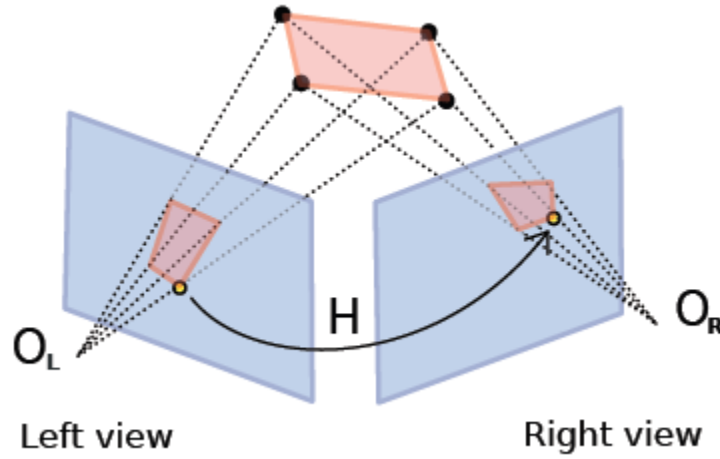


Fig. 1.2: The homography matrix and the point to point constraint.

Fundamental matrix:

The fundamental matrix is a relation between two images viewing the same scene where those point's projections are visible in the two images. Given a point correspondence between two views (x_i, x'_i) :

We obtain the following relation:

$$x_i'^T F x_i = 0$$

F is the (3×3) Fundamental matrix, it puts in relation a point x to a line where belong the projection of the 3D X point. $l'_i = Fx_i$ designs the epipolar line on which the point x'_i could be. The relation $x_i'^T F x_i = 0$ exists for all corresponding point belonging to a stereo pair.

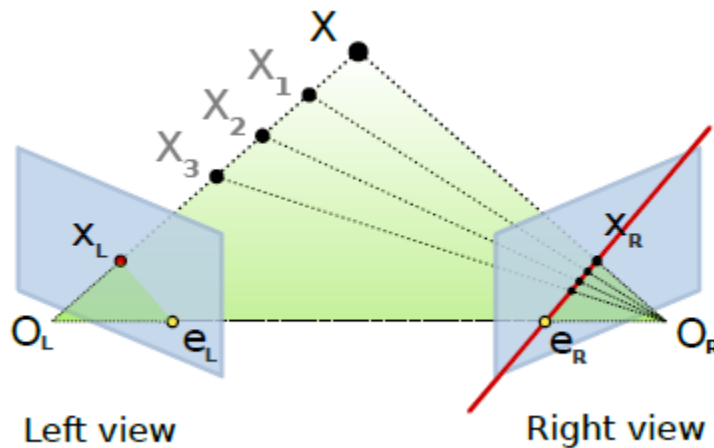


Fig. 1.3: The fundamental matrix and the point to line constraint.

The fundamental matrix is sometime called bifocal-tensor, it is a 3×3 matrix of rank 2 with 7 degree of freedom. 8 ou

7 correspondences are sufficient to compute the F matrix. Implementation follows the DLT (Direct Linear Transform) explained in [HZ] book.

Relative pose estimation (Essential matrix)

Adding intrinsic parameters to the fundamental matrix gives a metric “object” that provides the following relation $E = K'^T F K$, this is the Essential relation explained by Longuet-Higgins in 1981 [Longuet]. This essential matrix links the relative position of the camera to the fundamental matrix relation.

$$E = R[t]x = K'^T F K$$

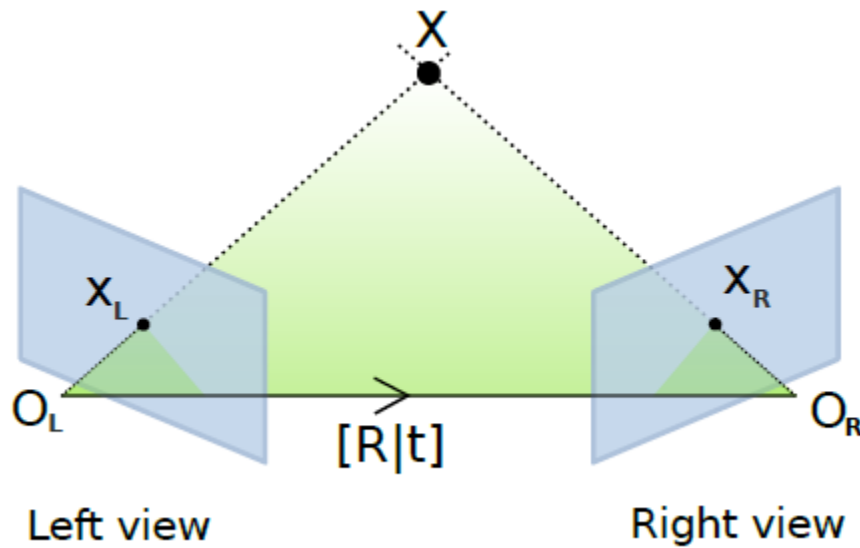


Fig. 1.4: The essential matrix geometric relation.

Absolute pose estimation/Camera resection (Pose matrix)

Given a list of 3D-2D point correspondences it is possible to compute a camera pose estimation. It consists in estimating the camera parameters of the right camera that minimizes the residual error of the 3D points re-projections, it's an optimization problem that trying to solve P parameters in order to minimize

$$\min \sum_{i=1}^n x_i - P(X_i).$$

openMVG provides 3 different solvers for this problem:

- 6pt Direct Linear Transform [HZ],
- 3pt with intrinsic EPnP [Ke],
- 3pt with intrinsic P3P [Kneip].

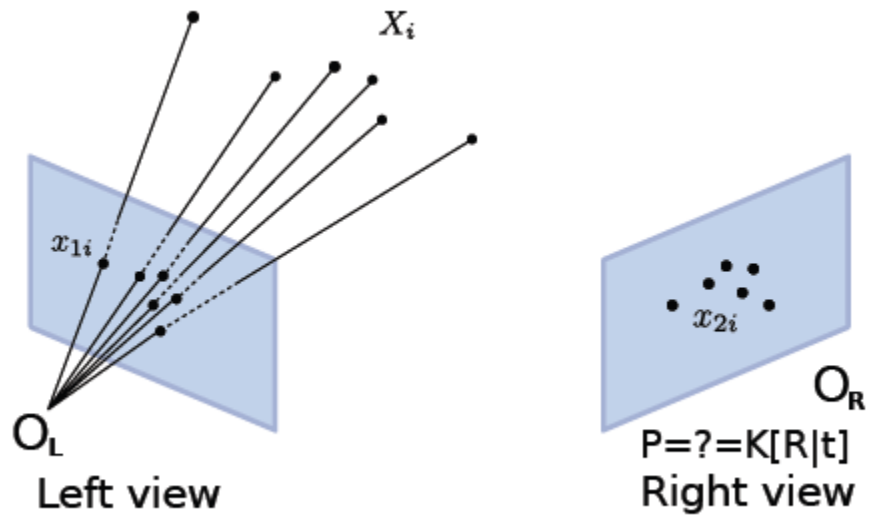


Fig. 1.5: Resection/Pose estimation from 3D-2D correspondences.

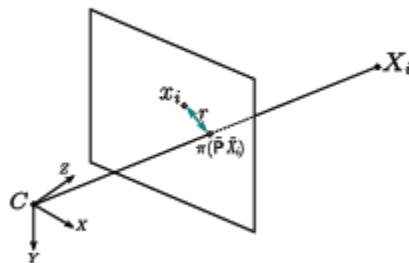


Fig. 1.6: Residual error.

Kernel concept

In order to use the solver in a generic robust estimation framework, we use them in conjunction with the Kernel class that allow to link:

- data points,
 - the set of correspondences that are used for a robust estimation problem.
- a model solver/estimator,
- a metric to measure data fitting to a putative model.

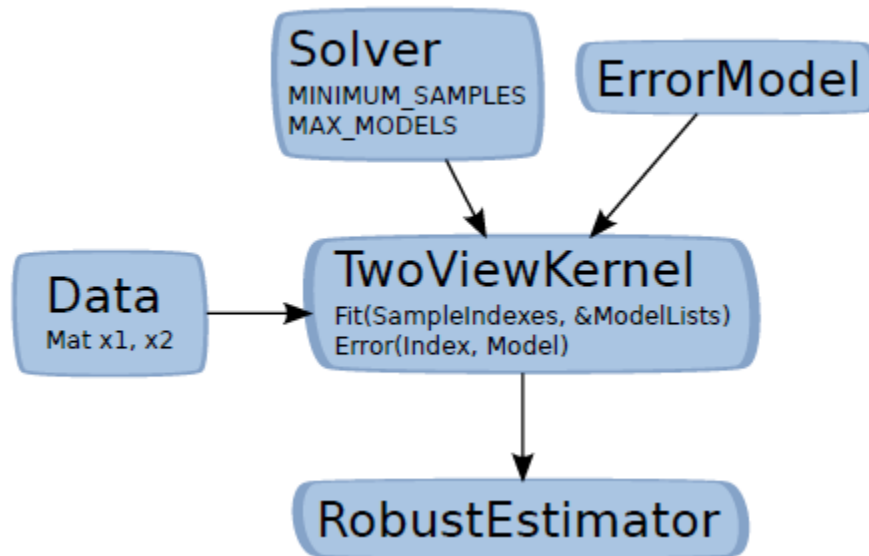


Fig. 1.7: The Kernel concept (the two view case).

Solver: MINIMUM_SAMPLES: The minimal number of point required for the model estimation, MAX_MODELS: The number of models that the minimal solver could return, A Solve function that estimates a model from MINIMUM_SAMPLES to n vector data.

ErrorModel: An metric function that return the error of a sample data to the provided model.

Kernel: Embed data (putative), the model estimator and the error model. This kernel is core brick used in the open-MVG robust estimator framework.

linear programming

Linear programming [LP] is a technique for the optimization of a linear objective function, subject to linear equality and linear inequality constraints such as:

$$\begin{array}{ll}
 \text{maximize} & \mathbf{c}^T \mathbf{x} \\
 \text{subject to} & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\
 \text{and} & \mathbf{x} \geq \mathbf{0}
 \end{array} \tag{1.2}$$

where \mathbf{x} represents the vector of variables (to be determined), \mathbf{c} and \mathbf{b} are vectors of (known) coefficients, \mathbf{A} is a (known) matrix of coefficients.

openMVG linear programming tools

openMVG provides tools to:

- configure Linear programs (LP container),
- solve Linear Programs (convex or quasi convex ones).

openMVG linear program container

openMVG provides a generic container for LP (Linear Programming problems) that can be dense or sparse.

```
// Dense LP
LP_Constraints
// Sparse LP
LP_Constraints_Sparse
```

It allows to embed:

- objective function c and the problem type (minimization or maximization),
- constraints (coefficients A , Sign, objective value b),
- bounds over x parameters (\leq , $=$, \geq).

openMVG linear program solvers

openMVG provide access to different solvers (not exhaustive):

- OSI_CLP (COIN-OR) project,
- MOSEK commercial, free in a research context.

Those solver have been chosen due to the stability of their results and ability to handle large problems without numerical stability (LPSolve and GLPK have been discarded after extensive experiments).

I refer the reader to `openMVG/src/openMVG/linearProgramming/linear_programming_test.cpp` to know more.

openMVG linear programming module usage

The linear programming module of openMVG can be used for:

- solve classical linear problem (optimization),
- test the feasibility of linear problem,
- optimize upper bound of feasible problem (quasi-convex linear programs).

classical linear problem solving (optimization)

Here an example of usage of the framework:

```
// Setup the LP (fill A,b,c and the constraint over x)
LP_Constraints cstraint;
BuildLinearProblem(cstraint);

// Solve the LP with the solver of your choice
std::vector<double> vec_solution(2);
#ifdef OPENMVG_HAVE_MOSEK
    MOSEK_SolveWrapper solver(2);
```



```

#else
    OSI_CLP_SolverWrapper solver(2);
#endif
// Send constraint to the LP solver
solver.setup(cstraint);

// If LP have a solution
if (solver.solve())
    // Get back estimated parameters
    solver.getSolution(vec_solution);

```

Linear programming, feasible problem

openMVG can be use also to test only the feasibility of a given LP

$$\begin{array}{ll}
 \text{find} & \mathbf{x} \\
 \text{subject to} & A\mathbf{x} \leq \mathbf{b} \\
 \text{and} & \mathbf{x} \in \mathcal{D}
 \end{array} \quad (1.5)$$

Linear programming, quasi convex optimization

openMVG used a lot of L infinity minimisation formulation. Often the posed problems are quasi-convex and dependent of an external parameter that we are looking for (i.e the maximal re-projection error for which the set of constraint is still feasible).

Optimization of this upper bound parameter can be done by iterating over all the possible value or by using a bisection that reduce the search range at each iteration.

Require: gammaLow, gammUp (Low and upper bound of the parameter to optimize)
Require: the LP problem (cstraintBuilder)
Ensure: the optimal gamma value, or **return** infeasibility of the constraints set.

```

BisectionLP(
    LP_Solver & solver,
    ConstraintBuilder & cstraintBuilder,
    double gammaUp = 1.0, // Upper bound
    double gammaLow = 0.0, // lower bound
    double eps = 1e-8, // precision that stop dichotomy
    const int maxIteration = 20) // max number of iteration
{
    ConstraintType constraint;
    do
    {
        ++k; // One more iteration

        double gamma = (gammaLow + gammaUp) / 2.0;

        //-- Setup constraint and solver
        cstraintBuilder.Build(gamma, constraint);
        solver.setup( constraint );

        //-- Solving
        bool bFeasible = solver.solve();

        //-- According feasibility update the corresponding bound
        //--> Feasible, update the upper bound
        //--> Not feasible, update the lower bound
        (bFeasible) ? gammaUp = gamma; : gammaLow = gamma;
    } while (gammaUp - gammaLow > eps);
}

```

```

} while (k < maxIteration && gammaUp - gammaLow > eps);
}

```

Multiple View Geometry solvers based on L-Infinity minimization

openMVG provides Linear programming based solvers for various problem in computer vision by minimizing at the same time the maximal error over a series of cost function and some model parameters. It uses a L-Infinity minimization method.

openMVG implements problems introduced by [\[LinfNorm\]](#) and generalized by [\[LinfNormGeneric\]](#) to solve multiple view geometry problem.

Rather than considering quadratic constraints that require SOCP (Second Orde Cone Programming) we consider their LP (linear program) equivalent. It makes usage of residual error expressed with absolute error ($|a| < b$). Inequalities are transformed in two linear inequalities $a < b$ and $-b < -a$ to be used in the LP framework. Using LP rather than SCOP allow to have better solving time and easier constraint to express (see. [\[Arnak\]](#) for more explanation).

OpenMVG propose solvers for the following problems:

- N-view triangulation [\[LinfNorm\]](#),
- Resection or pose matrix estimation [\[LinfNorm\]](#),
- Estimation of translations and structure from known rotations,
 - two formulation are implemented,
 - * the simple one [\[LinfNorm\]](#),
 - * the robust based on slack variables [\[OlssonDuality\]](#).
- Translation averaging: - Registration of relative translations to compute global translations [\[GlobalACSfM\]](#).

robust_estimation

Performing model estimation is not an easy task, data are always corrupted by noise and “false/outlier” data so robust estimation is required to find the “best” model along the possible ones.

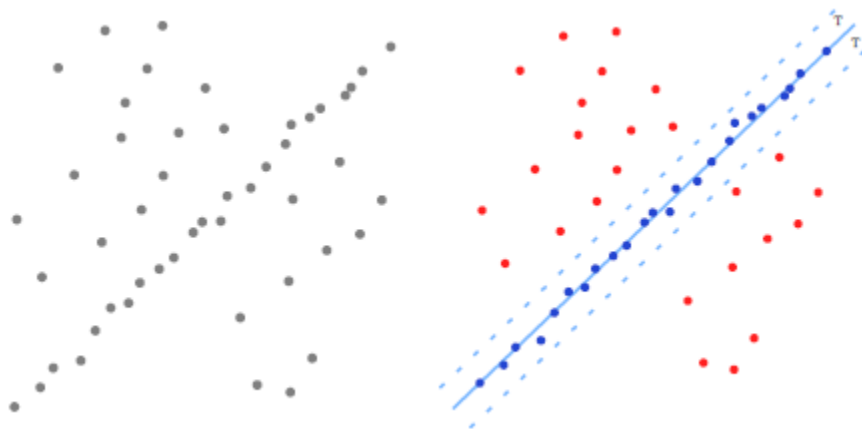


Fig. 1.8: Robust estimation: Looking for a line in corrupted data.

openMVG provides many methods to estimate one of the best possible model in corrupted data:

- Max-Consensus,
- Ransac,
- LMeds,
- AC-Ransac (A Contrario Ransac).

Max-Consensus

The idea of Max-Consensus is to use a random picked subset of data to find a model and test if this model is good or not the whole dataset. At the end you keep the model that best fits your cost function. Best fit defined as the number of data correspondences to the model under your specified threshold T .

Algorithm 1 Max-Consensus

```

Require: correspondences
Require: model solver, residual error computation
Require:  $T$  threshold for inlier/outlier discrimination
Require: maxIter the number of performed model estimation
Ensure: inlier list
Ensure: best estimated model Mbest
    for  $i = 0 \dots \text{maxIter}$  do
        Pick NSample random samples
        Evaluate the model  $M_i$  for the random samples
        Compute residuals for the estimated model
        if  $\text{Cardinal}(\text{residual} < T) > \text{previousInlierCount}$  then
            previousInlierCount =  $\text{Cardinal}(\text{residual} < T)$ 
            Mbest =  $M_i$ 
        end if
    end for

```

Here an example of how find a best fit line:

```

Mat2X xy ( 2 , 5);
// Defines some data points
xy << 1, 2, 3, 4, 5, // x
     3, 5, 7, 9, 11; // y

// The base model estimator and associated error metric
LineKernel kernel ( xy );

// Call the Max-Consensus routine
std::vector<uint32_t> vec_inliers;
Vec2 model = MaxConsensus ( kernel , ScorerEvaluator<LineKernel >(0.3) , &-vec_
    ↪inliers );

```

Ransac

Ransac [[RANSAC](#)] is an evolution of Max-Consensus with a-priori information about the noise and corrupted data amount of the data. That information allows to reduce the number of iterations in order to be sure to have made sufficient random sampling steps in order to find the model for the given data confidence. The number of remaining steps is so iteratively updated given the inlier/outlier ratio of the current found model.

Here an example of how find a best fit line:

```

Mat2X xy ( 2 , 5);
// Defines some data points
xy << 1, 2, 3, 4, 5, // x
      3, 5, 7, 9, 11; // y

// The base model estimator and associated error metric
LineKernel kernel ( xy );

// Call the Ransac routine
std::vector<uint32_t> vec_inliers;
Vec2 model = Ransac ( kernel, ScorerEvaluator<LineKernel >(0.3) , &vec_inliers );

```

AC-Ransac A Contrario Ransac

RANSAC requires the choice of a threshold T , which must be balanced:

- Too small: Too few inliers, leading to model imprecision,
- Too large: Models are contaminated by outliers (false data).

AC-Ransac [\[ACRANSAC\]](#) uses the a contrario methodology in order to find a model that best fits the data with a confidence threshold T that adapts automatically to noise. It so finds a model and its associated noise. If there is too much noise, the a contrario method returns that no model was found.

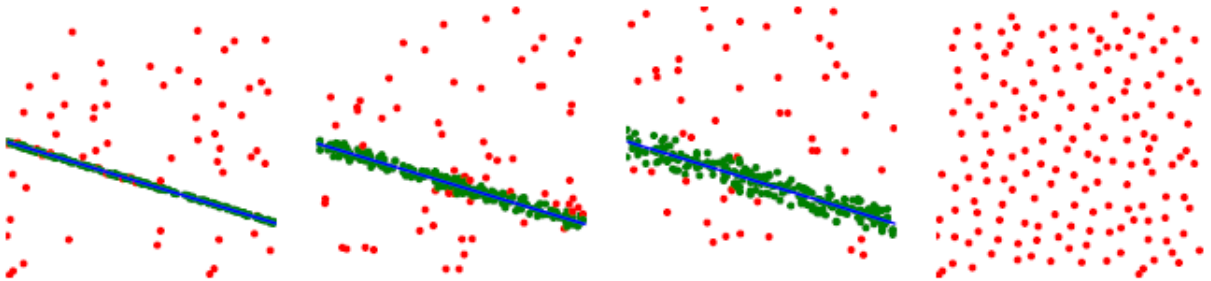


Fig. 1.9: A contrario robust estimation, noise adaptivity.

Here an example of how to find a best fit line, by using the a contrario robust estimation framework: It is a bit more complex, we use a class in order to perform the a contrario required task.

```

Mat2X xy ( 2 , 5);
// Defines some data points
xy << 1, 2, 3, 4, 5, // x
      3, 5, 7, 9, 11; // y

// The acontrario adapted base model estimator and associated error metric
const size_t img_width = 12;
ACRANSACOneViewKernel<LineSolver, pointToLineError, Vec2> lineKernel(xy, -img_width,
↪img_width);

// Call the AC-Ransac routine
std::vector<uint32_t> vec_inliers;
Vec2 line;
std::pair<double, double> res = ACRANSAC(lineKernel, vec_inliers, 300, &line);
double dPrecision = res.first;
double dNfa = res.second;

```

matching

Nearest neighbor search

We provide a generic interface to perform:

- **Nearest neighbor search (NNS)**
- **K-Nearest Neighbor (K-NN)**

Three implementations are available:

- a Brute force,
- an Approximate Nearest Neighbor [\[FLANN\]](#),
- a Cascade hashing Nearest Neighbor [\[CASCADEHASHING\]](#).

This module works for data of any dimensionality, it could be use to match:

- 2 or 3 vector long features (points),
- 128, 64, vector long features (like global/local feature descriptors).

Using the **Nearest neighbor search (NNS)** let you find pairs of elements $((i, j))$ from sets A and B that are the closest for a given metric d :

$$\{(P_A^i, P_B^j) : j = \underset{k}{\operatorname{argmin}} \quad d(\operatorname{desc}(P_A^i), \operatorname{desc}(P_B^k))\}$$

Using the **K-NN** will return you tuple of elements: $(i; (j, k))$ if 2 nearest values have been asked for the Inth query.

Example of usage:

```
// Setup the matcher
ArrayMatcherBruteForce<float> matcher;
// The reference array
float array[] = {0, 1, 2, 3, 4};
// Setup the reference array of the matcher
matcher.Build(array, 5, 1);

//--
// Looking for the nearest neighbor:
//--
// Perform a query to look which point is closest to 1.8
float query[] = {1.8f};
int nIndex = -1;
float fDistance = -1.0f;
matcher.SearchNeighbour(query, &nIndex, &fDistance);

// nIndex == 2 ; // index of the found nearest neighbor
// fDistance == 0.4; // squared distance

//--
// Looking for the K=2 nearest neighbor
//--
IndMatches vec_nIndices;
vector<float> vec_fDistance;
const int K = 2;
matcher.SearchNeighbours(query, 1, &vec_nIndices, &vec_fDistance, K);

// vec_nIndices = {IndMatch(0,2), IndMatche(0,1)};
```

Metric customization

- L2 (used by default):

$$d(x, y) := \|x - y\|_2$$

- i.e L1 for binary descriptor (Hamming distance),

$$d(x, y) = \sum (x \oplus y)$$

- user customized distance (L1, ...).

Image domain, correspondences filtering (KVLD)

When used with descriptors found putative matches can be filtered thanks to different filters:

- Symmetric distance (Left-Right check).
 - Keep only mutal correspondences: a match is kept if it is the same in the A->B and B->A order.

$$\{(P_A^i, P_B^j) : j = \operatorname{argmin}_k d(\operatorname{desc}(P_A^i), \operatorname{desc}(P_B^k)), i = \min_k d(\operatorname{desc}(P_A^k), \operatorname{desc}(P_B^j))\}$$

- “Nearest Neighbor Distance Ratio” distance check can be performed to remove repetitive elements.
 - As many nearest points have been asked we can measure the similarity between the N-nearest neighbor. If the ratio of distance is inferior to a threshold δ the match is kept else it is rejected (since the risk of confusion is higher). It allows to avoid ambiguous correspondences. δ is often chosen between 0.6 and 0.8.

$$\{(P_A^i, P_B^j) : j = \operatorname{argmin}_k d(\operatorname{desc}(P_A^i), \operatorname{desc}(P_B^k)) < \delta \min_{k \neq j} d(\operatorname{desc}(P_A^i), \operatorname{desc}(P_B^k))\}$$

- K-VLD Filter (K-Virtual Line Descriptor) [\[KVLD12\]](#)
 - A virtual segment (a segment/line between two points) is kept if at least one of it's supporting point is linked to K virtual segments. It produces a coherent photometric graph of the features from the set of points A and B. Below: Top (SIFT putative matches found by NNS), Bottom: K-VLD coherent matches.

tracks

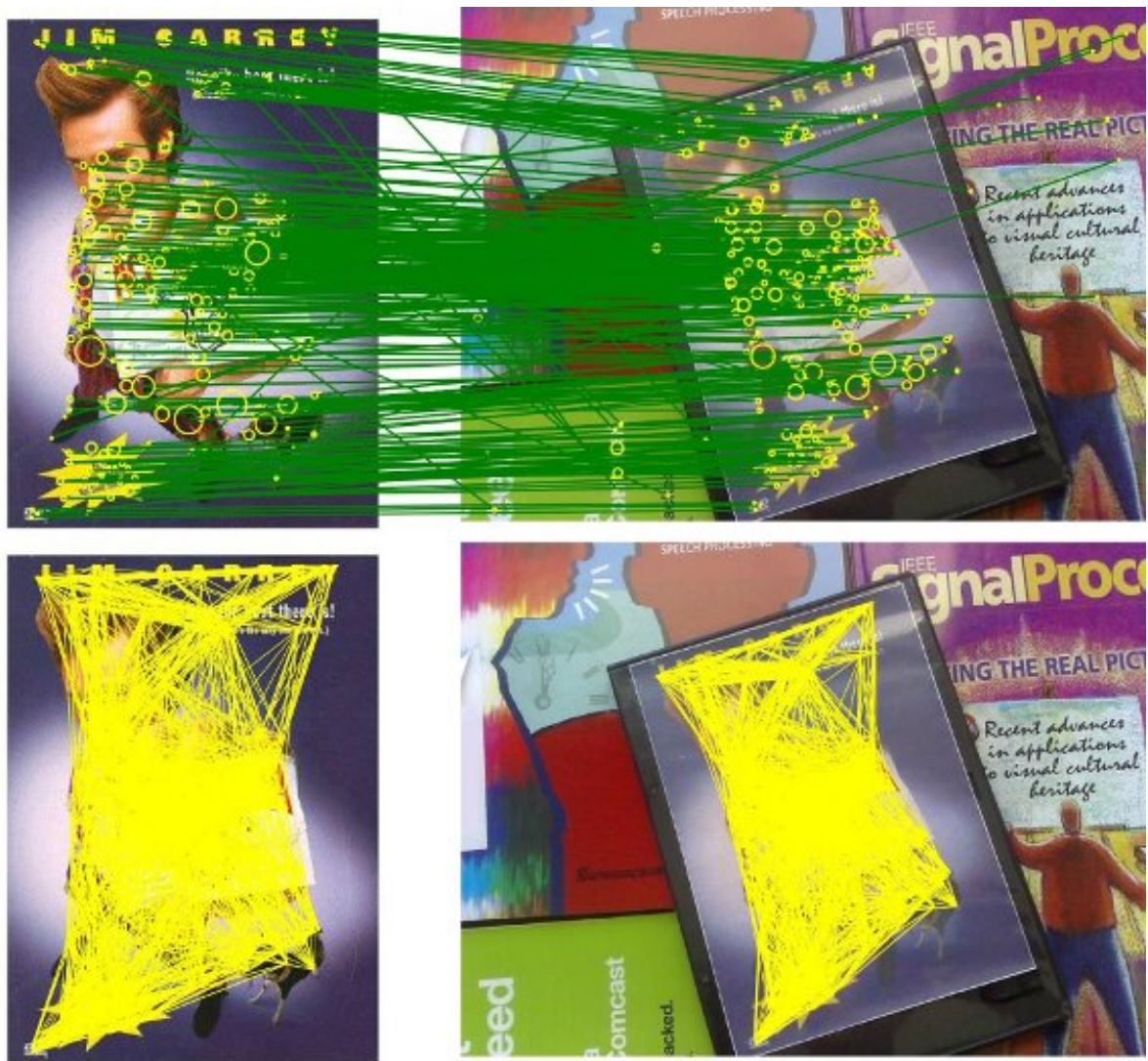
The problem of feature points tracking is to follow the position of a characteristic point in a set of images. These multi-view correspondences are called tracks. Track identification in a set of images (ordered, or not) is an important task in computer vision. It allows solving geometry-related problems like video stabilization, tracking, match-moving, image-stitching, structure from motion and odometry.

un/ordered feature tracking

Considering n pairwise feature correspondences as input we want sets of corresponding matching features across multiple images, as illustrated in the following figures with video frames.

The openMVG library provides an efficient solution to solve the union of all the pairwise correspondences. It is the implementation of the CVMP12 paper “Unordered feature tracking made fast and easy” [\[TracksCVMP12\]](#).

Some comments about the data structure:



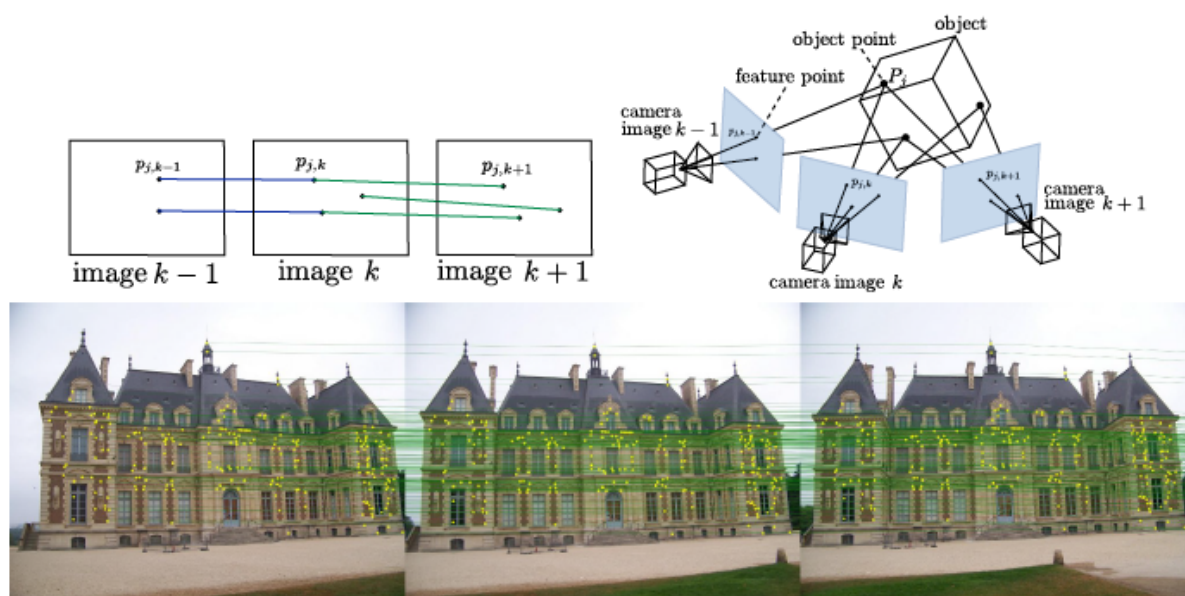


Fig. 1.10: From features to tracks.

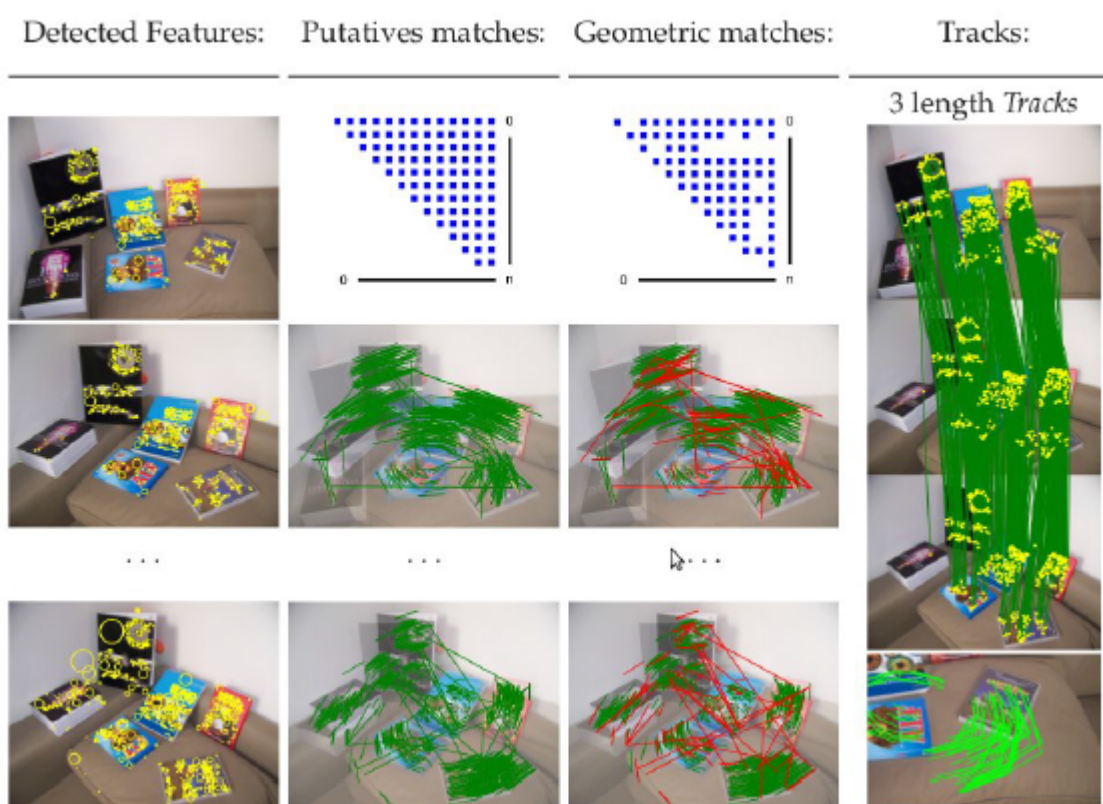


Fig. 1.11: Feature based tracking.


```

using namespace openMVG::matching;

// pairwise matches container:
PairWiseMatches map_Matches;

// Fil the pairwise correspondences or load a series of pairwise matches from a file
PairedIndMatchImport("matches.f.txt", map_Matches);

//-----
// Compute tracks from pairwise matches
//-----
TracksBuilder tracksBuilder;
tracks::STLMapTracks map_tracks; // The track container
tracksBuilder.Build(map_Matches); // Build: Efficient fusion of correspondences
tracksBuilder.Filter();           // Filter: Remove track that have conflict
tracksBuilder.ExportToSTL(map_tracks); // Build tracks with STL compliant type

// In order to visit all the tracks, follow this code:
for (tracks::STLMapTracks::const_iterator iterT = map_tracks.begin();
     iterT != map_tracks.end(); ++ iterT)
{
    const IndexT trackId = iterT->first;
    const tracks::submapTrack & track = iterT->second;
    for ( tracks::submapTrack::const_iterator iterTrack = track.begin();
         iterTrack != track.end(); ++iterTrack)
    {
        const IndexT imageId = iterTrack->first;
        const IndexT featId = iterTrack->second;

        // Get the feature point
    }
}

```

geometry

Pose

Pose3 defines the 3D Pose as a 3D transform: $[RIC] \ t = -RC$

```

// Define two poses and combine them
Pose3 pose1(RotationAroundX(0.02), Vec3(0,0,-2));
Pose3 pose2(RotationAroundX(0.06), Vec3(-1,0,-2));
Pose3 combinedPose = pose1 * pose2;

// Apply the pose to a 3DPoint (World to local coordinates):
const Vec3 pt = combinedPose(Vec3(2.6453,3.32,6.3));

```

Frustum & Frustum intersection

Define a camera Frustum from Pose3 and intrinsic parameters as:

- an infinite Frustum (4 Half Spaces) (a pyramid);
- a truncated Frustum (6 Half Spaces) (a truncated pyramid).

This structure is used for testing frustum intersection (see if two camera can share some visual content).

```
// Build two truncated frustum and test their intersection
Frustum frustum1(somedata, minDepth, maxDepth);
Frustum frustum2(somedata, minDepth, maxDepth);
bool bIntersect = frustum1.intersect(frustum2);

// Build two infinite frustum and test their intersection
Frustum frustum1(somedata);
Frustum frustum2(somedata);
bool bIntersect = frustum1.intersect(frustum2);
```

7DoF registration between point set

Find the rigid registration between point set using a scale, rotation and translation model.

```
// Simulate two point set, apply a known transformation and estimate it back:
const int nbPoints = 10;
const Mat x1 = Mat::Random(3,nbPoints);
Mat x2 = x1;

const double scale = 2.0;
const Mat3 rot = (Eigen::AngleAxis<double>(.2, Vec3::UnitX())
  * Eigen::AngleAxis<double>(.3, Vec3::UnitY())
  * Eigen::AngleAxis<double>(.6, Vec3::UnitZ())).toRotationMatrix();
const Vec3 t(0.5,-0.3,.38);

for (int i=0; i < nbPoints; ++i)
{
  const Vec3 pt = x1.col(i);
  x2.col(i) = (scale * rot * pt + t);
}

// Compute the Similarity transform
double Sc;
Mat3 Rc;
Vec3 tc;
FindRTS(x1, x2, &Sc, &tc, &Rc);
// Optional non linear refinement of the found parameters
Refine_RTS(x1,x2,&Sc,&tc,&Rc);

std::cout << "\n"
  << "Scale " << Sc << "\n"
  << "Rot \n" << Rc << "\n"
  << "t " << tc.transpose();

std::cout << "\nGT\n"
  << "Scale " << scale << "\n"
  << "Rot \n" << rot << "\n"
  << "t " << t.transpose();
```

geodesy

This module provides conversion tool to manipulate data in some geographic coordinate system.

Datum conversion

- **lla_to_ecef, ecef_to_lla**
 - Conversion between the WGS84 GPS Latitude/Longitude/Altitude datum and the ECEF coordinate system.
- **lla_to_utm**
 - Conversion from the WGS84 GPS Latitude/Longitude/Altitude datum to the UTM coordinate system.

Structure from Motion core

sfm

sfm is the module related to Structure from Motion. It handles storage of SfM related data and method to solve SfM problems (camera pose estimation, structure triangulation, bundle_adjustment).

A generic SfM data container

SfM_Data class contains all the data used to describe input of a SfM problem:

- a collection of **View**
 - the used images
- a collection of **camera extrinsic**
 - the camera poses
- a collection of **camera intrinsic**
 - the camera internal projection parameters
- a **structure**
 - the collection of landmark (3D points associated with 2d view observations)

```
struct SfM_Data
{
    /// Considered views
    Views views;

    /// Considered poses (indexed by view.id_pose)
    Poses poses;

    /// Considered camera intrinsics (indexed by view.id_cam)
    Intrinsics intrinsics;

    /// Structure (3D points with their 2D observations)
    Landmarks structure;

    // ...
}
```

View concept

The view store information related to an image:

- image filename
- id_view (must be unique)
- id_pose
- id_intrinsic
- image size

Note that thanks to the usage of ids we can defined shared poses & shared intrinsics.

View type is **abstract** and provide a way to add new custom View type: i.e. GeoLocatedView (add GPS position, ...)

Camera Poses concept

The camera pose store a 3D pose that define a camera rotation and position (camera rotation and center).

Camera Intrinsic concept

Define the parameter of a camera. It can be shared or not. Intrinsics parameter are **abstract** and provide a way to easily add new custom camera type.

Structure/Landmarks concept

It defines the structure:

- 3D point with 2D view features observations.

SfM_Data cleaning

Generic interface are defined to remove outlier observations:

- use a given residual pixel error to discard outlier,
- use a minimal angle along the track bearing vectors.

Triangulation

Once the SfM_Data is filled with some landmark observations and poses we can compute their 3D location.

Two method are proposed:

- A blind method:
 - Triangulate tracks using all observations,
 - Inlier/Outlier classification is done with a cheirality test,
- A robust method:
 - Triangulate tracks using a RANSAC scheme,
 - Check cheirality and a pixel residual error.

Non linear refinement, Bundle Adjustment

OpenMVG provides a generic bundle_adjustment framework to refine or keep as constant the following parameters:

- internal orientation parameters (intrinsics: camera projection model),
- external orientation parameters (extrinsics: camera poses),
- structure (3D points).

```
SfM_Data sfm_data;
// initialize the data
// ...

const double dResidual_before = RMSE(sfm_data);

// Bundle adjustment over all the parameters:
std::shared_ptr<Bundle_Adjustment> ba_object = std::make_shared<Bundle_Adjustment_
↳Ceres>();
ba_object->Adjust(sfm_data);

const double dResidual_after = RMSE(sfm_data);
```

Bundle Adjustment (ajustement de faisceaux), is a non linear optimization problem. It looks to minimizing the residual error of a series of user cost functions (the reprojection errors of the structure X_j to the images measures x_j^i). According:

- X_j the Jnth 3D point of the structure of the scene,
- x_j^i the observation of the projection of the 3D point X_j in the image i ,
- P_i the projection matrix of the image i

From a user provided initial guess the vector of parameters: $\{X_j, P_i\}_{i,j}$: camera parameters $\{P_i\}_i$ and the scene structure $\{X_j\}_j$ are refined in order to minimizes the residual reprojection cost:

$$\underset{\{P_i\}_i, \{X_j\}_j}{\text{minimize}} \left\| \sum_{j=0}^m \sum_{i=0}^n x_j^i - P_i X_j \right\|_2$$

OpenMVG proposes options in order to tell if a parameter group must be kept as constant or refined during the minimization.

SfM Pipelines

OpenMVG provides ready to use and customizable pipelines for:

- solving sequential/incremental SfM,
- solving global SfM,
- computing a Structure from known camera poses.

Sequential SfM

The [\[ACSfM\]](#) SfM is based on the implementation used for the paper “Adaptive Structure from Motion with a contrario model estimation” published at ACCV 2012.

The incremental pipeline is a growing reconstruction process. It starts from an initial two-view reconstruction (the seed) that is iteratively extended by adding new views and 3D points, using pose estimation and triangulation. Due

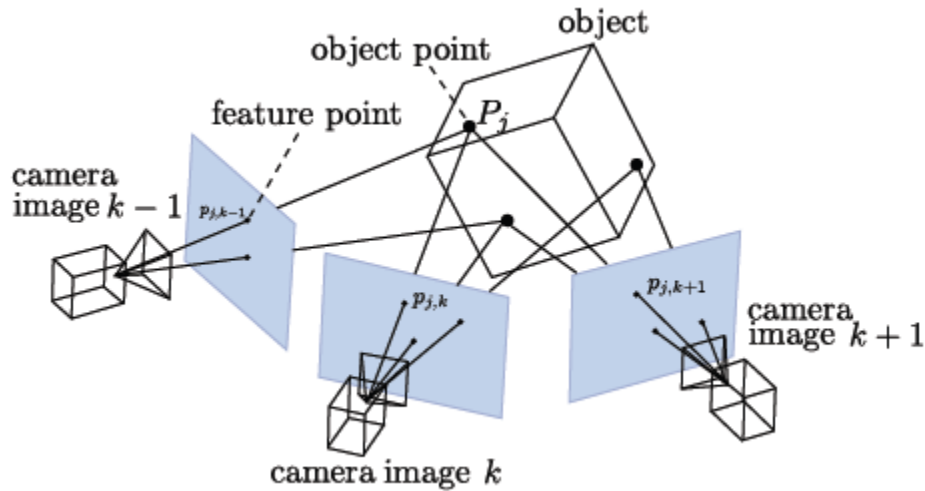


Fig. 1.12: Figure: From point observation and intrinsic camera parameters, the 3D **structure** of the scene is computed **from** the estimated **motion** of the camera.

to the incremental nature of the process, successive steps of non-linear refinement, like Bundle Adjustment (BA), are performed to minimize the accumulated error (drift).

Incremental Structure from Motion

```

Require: internal camera calibration (possibly from EXIF data)
Require: pairwise geometry consistent point correspondences
Ensure: 3D point cloud
Ensure: camera poses
compute correspondence tracks  $t$ 
compute connectivity graph  $G$  (1 node per view, 1 edge when enough matches)
pick an edge  $e$  in  $G$  with sufficient baseline
* robustly estimate essential matrix from images of  $e$ 
triangulate validated tracks, which provides an initial reconstruction
contract edge  $e$ 
while  $G$  contains an edge do
  pick edge  $e$  in  $G$  that maximizes  $\text{union}(\text{track}(e), \text{3D points})$ 
  * robustly estimate pose (external orientation/resection)
  triangulate new tracks
  contract edge  $e$ 
  perform bundle adjustment
end while

```

Steps marked by a * are robust estimation performed using the a-contrario robust estimation framework.

Global SfM

[*GlobalACSfM*] is based on the paper “Global Fusion of Relative Motions for Robust, Accurate and Scalable Structure from Motion.” published at ICCV 2013.

Multi-view structure from motion (SfM) estimates the position and orientation of pictures in a common 3D coordinate frame. When views are treated incrementally, this external calibration can be subject to drift, contrary to global methods that distribute residual errors evenly. Here the method propose a new global calibration approach based on the fusion of relative motions between image pairs.

```

Require: internal camera calibration (possibly from EXIF data)
Require: pairwise geometry consistent point correspondences
Ensure: 3D point cloud
Ensure: camera poses

compute relative pairwise rotations
detect and remove false relative pairwise rotations
  - using composition error of triplet of relative rotations
compute the global rotation
  - using a least square and approximated rotations
compute relative translations
  - using triplet of views for stability and colinear motion support
compute the global translation
  - integration of the relative translation directions using a  $l-\infty$  method
final structure and motion
  - link tracks validated per triplets and compute global structure by triangulation
  - refine estimated parameters in a 3 step Bundle Adjustment
  - refine structure and translations
  - refine structure and camera parameters (rotations, translations)
  - refine if asked intrinsics parameters

```

Structure computation from known camera poses

This class allows to compute valid 3D triangulation from 2D matches and known camera poses.

```

Require: internal and external camera calibration
Require: features and corresponding descriptor per image view
Ensure: 3D point cloud

initialize putatives matches pair from
  - a provided pair file
  - or automatic pair computed from camera frustum intersection
for each pair
  - find valid epipolar correspondences
for triplets of view
  - filter 3-view correspondences that leads to invalid triangulation
merge 3-view validated correspondences
  - robustly triangulate them
save the scene with the update structure

```

Todo (complete documentation):

- **system** - Provide access to accurate timer function.
- **graph** - Manipulation of graphs (connected CC, triplet listing...)
- **exif** - Collection of tools to extract and use image EXIF data.
- **matching_image_collection** - Functions to perform local matching and geometric filtering of putative correspondences in image collections.
- **stl** - Enhancement of some STL functionality.

openMVG samples

openMVG focus on a strong implementation checking of the provided features. To do so it provides unit test (that assert code results and helps user to see how the code must be used) but it provides also illustrated samples of the major features.

The samples can be seen as showcase and tutorials:

imageData

- some pictures for each of the following examples.

features_siftPutativeMatches

Show how:

- extract SIFT features and descriptors,
- match features descriptors,
- display the computed matches.

features_affine_demo

Show how:

- use the MSER/TBMR region detector
- display regions fitted ellipses

features_image_matching

Show how:

- use the Image_describer interface to extract features & descriptors
- match the detected regions
- display detected features & corresponding matches

features_kvld_filter

Show how:

- filter putative matches with the K-VLD filter [\[KVLD12\]](#).

features_repeatability

Show how to use Oxford's "Affine Covariant Regions Datasets" image datasets in order to compute feature position and or descriptor matching repeatability measures.

multiview_robust_homography

Show how:

- estimate a robust homography between features matches.

multiview_robust_homography_guided

Show how:

- estimate a robust homography between features matches,
- extend the putative matches with a H guided filter,
- warp the query image over the reference image.

multiview_robust_fundamental

Show how:

- estimate a robust fundamental matrix between features matches.

multiview_robust_fundamental_guided

Show how:

- estimate a robust fundamental matrix between features matches,

- extend the putative matches with a F guided filter.

multiview_robust_essential

Show how:

- estimate a robust essential matrix between features matches,
- compute the 3D structure by triangulation of the corresponding inliers.

multiview_robust_essential_ba

Show how:

- refine with bundle_adjustment the Structure and Motion of a scene
- for different camera model:
 - Refine $[X], [f, R | t]$ (individual cameras),
 - Refine $[X], [R | t]$, shared $[f]$,
 - Refine $[X], [R | t]$, shared brown disto models.

multiview_robust_essential_spherical

Show how:

- estimate a robust essential matrix between two spherical panorama
- triangulate remaning inliers.

exif_Parsing

Show how:

- parse JPEG EXIF metadata

exif_sensorWidthDatabase

Show how:

- use the camera sensor width database

cameras_undisto_Brown

Show how:

- undistord a picture according known Brown radial parameters.

openMVG_sample_pano_converter

Show how extract many rectilinear images from a spherical panorama.

Don't hesitate to help to extend the list.

SfM: Structure from Motion

Structure from Motion computes an external camera pose per image (the motion) and a 3D point cloud (the structure) from:

- images,
- some intrinsic camera parameters,
- corresponding geometric valid features accross images.

openMVG SfM tools

- **2 Structure from Motion (SfM) pipeline:**
 - an Incremental Structure from Motion chain [*ACSfM*] (ACCV 2012),
 - a Global Structure from Motion chain [*GlobalACSfM*] (ICCV 2013),
- **1 Structure from known Motion (SfM) pipeline:**
 - Structure computation from known camera poses and features.
- **tools to visualize:**
 - features,
 - photometric/geometric matches correspondences,
 - features tracks.
- **tools to export to existing Multiple View Stereovision (MVS) pipeline:**
 - [*PMVS*], CPMVS.

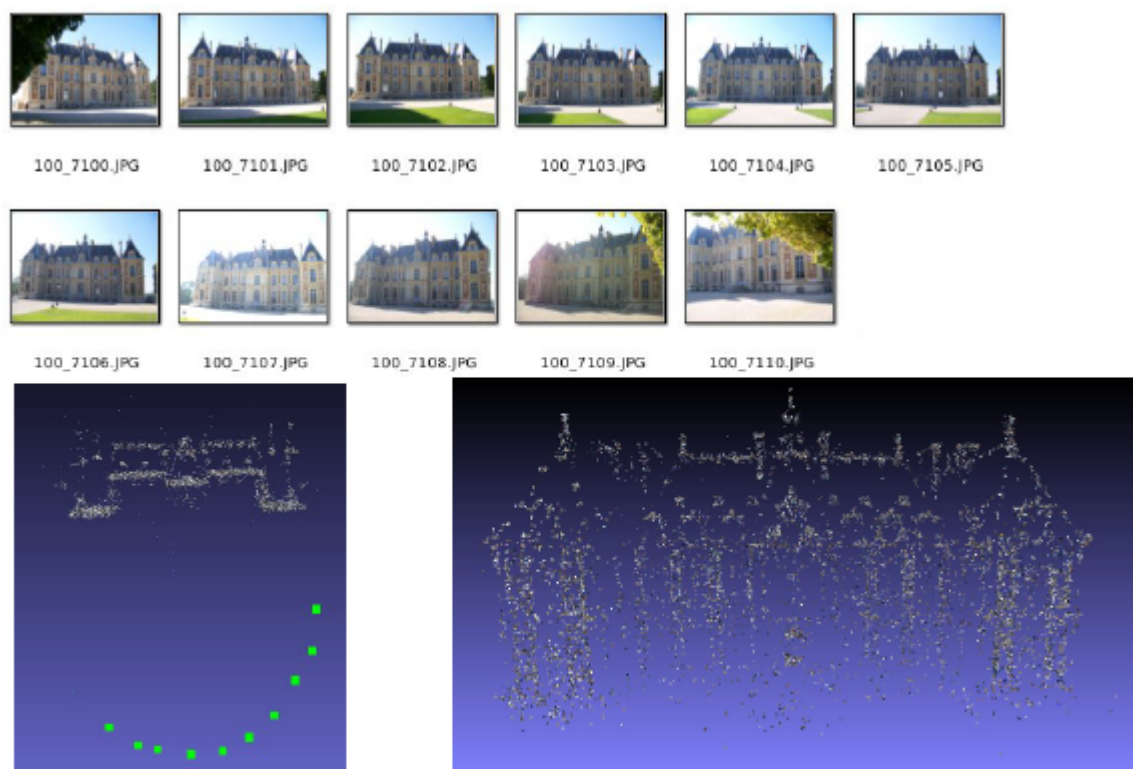


Fig. 3.1: Figure : Input images, estimated camera location and structure.

OpenMVG SfM pipelines

OpenMVG SfM pipelines run as a 4 step process:

1. Image listing

openMVG_main_SfMInit_ImageListing

The first task to process an image dataset in openMVG pipelines consist in creating a **sfm_data.json** file that describe the used image dataset.

This structured file lists for each images an object called a View. This view store image information and lists:

- the image name,
- the image size,
- the internal camera calibration information (intrinsic parameters) (if any).

Each **View** is associated to a camera **Pose** index and an **Intrinsic** camera parameter group (if any). Group means that camera intrinsic parameters can be shared between some Views (that leads to more stable parameters estimation).

```
$ openMVG_main_SfMInit_ImageListing -i [] -d [] -o []
```

Arguments description:

Required parameters:

- **[-il-imageDirectory]**
- **[-dl-sensorWidthDatabase]** openMVG/src/openMVG/exif/sensor_width_database/sensor_width_camera_database.txt
- **[-ol-outputDirectory]**

Optional parameters:

- **[-fl-focal]** (value in pixels)
- **[-kl-intrinsics]** Kmatrix: “f;0;ppx;0;f;ppy;0;0;1”
- **[-cl-camera_model]** Camera model type:
 - 1: Pinhole
 - 2: Pinhole radial 1
 - 3: Pinhole radial 3 (default)
- **[-gl-group_camera_model]**
 - 0-> each view have it’s own camera intrinsic parameters
 - 1-> (default) view can share some camera intrinsic parameters

```
// Example
$ openMVG_main_SfMInit_ImageListing -d /home/user/Dev/openMVG/src/openMVG/exif/sensor_
↪width_database/sensor_width_camera_database.txt -i /home/user/Dataset/ImageDataset_
↪SceauxCastle/images -o /home/user/Dataset/ImageDataset_SceauxCastle/matches

If you have installed OpenMVG on your machine your could do:
$ openMVG_main_SfMInit_ImageListing -d /usr/share/openMVG/sensor_width_camera_
↪database.txt -i /home/user/Dataset/ImageDataset_SceauxCastle/images -o /home/user/
↪Dataset/ImageDataset_SceauxCastle/matches
```

It will produce you a sfm_data.json file that is used by openMVG as a scene description.

Once your have computed your dataset description you can compute the image features:

openMVG_main_ComputeFeatures

Compute image description for a given sfm_data.json file. For each view it compute the image description (local regions) and store them on disk

```
$ openMVG_main_ComputeFeatures -i [..\matches\sfm_data.json] -o [...\matches]
```

Arguments description:

Required parameters:

- **[-il-input_file]**
 - a SfM_Data file
- **[-ol-outdir path]**
 - path were image description will be stored

Optional parameters:

- **[-fl-force: Force to recompute data]**
 - 0: (default) reload previously computed data (useful when you have kill the process and want to continue to compute)
 - 1: useful when you change have changed a command line parameter, force recomputing and re-saving.
- **[-ml-describerMethod]**
 - Used method to describe an image:
 - * SIFT: (default),
 - * AKAZE_FLOAT: AKAZE with floating point descriptors,
 - * AKAZE_MLDB: AKAZE with binary descriptors.
- **[-ul-upright]**
 - Use Upright feature or not
 - * 0: (default, rotation invariance)
 - * 1: extract upright feature (orientation angle = 0°)
- **[-pl-describerPreset]**
 - Used to control the Image_describer configuration:
 - * NORMAL,
 - * HIGH,
 - * ULTRA: !!Can be time consuming!!

Use mask to filter keypoints/regions

Sometime you may want to compute features/regions only on some parts of your images. It could include the following cases:

- You know that in your acquisition some areas may disturb the SfM pipeline (even if OpenMVG is known to be robust to a fair amount of outliers) and lower the quality of the subsequent reconstruction. For example, in some close range configurations, you may prefer to move the object itself instead of moving the camera. Masks can also help you to deal with hemispherical fish-eyes, by masking useless zone of the sensor.
- You want to speed up the computation by reducing the number features/regions and thus the number of tie points.

For this kind of needs you can use a mask. A mask is simply a binary image having the same size (width and height) than the target image. The black areas on a mask denote the “bad parts”, *i.e.* the areas to be masked and for which descriptors are not computed. A point is kept if the mask value at the point position is different than 0. In `openMVG_main_ComputeFeatures`, the association of a mask and an image is implicit. It uses the following conventions:

- It tries to load a global `mask.png` file from where the images are stored.
- It tries to load an individual `<image_name>_mask.png` from directory where the current image is stored.

The individual mask **always** takes precedence over the global one.

Once `openMVG_main_ComputeFeatures` is done you can compute the Matches between the computed description.

openMVG_main_ComputeMatches

This binary compute images that have a visual overlap. Using image descriptions computed by **openMVG_main_ComputeFeatures**, we establish the corresponding putative photometric matches and filter the resulting correspondences using some robust geometric filters.

```
$ openMVG_main_ComputeMatches -i [..\matches\sfm_data.json] -o [...\matches]
```

Arguments description:

Required parameters:

- **[-i-input_file]**
 - a SfM_Data file
- **[-o-out_dir path]**
 - path where putative and geometric matches will be stored

Optional parameters:

- **[-f-force: Force to recompute data]**
 - 0: (default) reload previously computed data (useful when you have kill the process and want to continue to compute)
 - 1: useful when you change have changed a command line parameter, force recomputing and re-saving.
- **[-r-ratio]**
 - (Nearest Neighbor distance ratio, default value is set to 0.8). Using 0.6 is more restrictive => provides less false positive.
- **[-g-geometric_model]**
 - type of model used for robust estimation from the photometric putative matches
 - * f: Fundamental matrix filtering

- * e: Essential matrix filtering
- * h: Homography matrix filtering
- **[-nl-nearest_matching_method]**
 - AUTO: auto choice from regions type,
 - For Scalar based descriptor you can use:
 - * BRUTEFORCEL2: BruteForce L2 matching for Scalar based regions descriptor,
 - * ANNL2: Approximate Nearest Neighbor L2 matching for Scalar based regions descriptor,
 - * CASCADEHASHINGL2: L2 Cascade Hashing matching,
 - * **FASTCASCADEHASHINGL2: (default).** L2 Cascade Hashing with precomputed hashed regions, (faster than CASCADEHASHINGL2 but use more memory).
 - For Binary based descriptor you must use:
 - * BRUTEFORCEHAMMING: BruteForce Hamming matching for binary based regions descriptor,
- **[-vl-video_mode_matching]**
 - (sequence matching with an overlap of X images)
 - * X: with match 0 with (1->X), ...]
 - * 2: will match 0 with (1,2), 1 with (2,3), ...
 - * 3: will match 0 with (1,2,3), 1 with (2,3,4), ...]
- **[-ll-pair_list]**
 - file that explicitly list the View pair that must be compared

Once matches have been computed you can, at your choice, you can display detected, matches as SVG files:

- **Detected keypoints:** openMVG_main_exportKeypoints
- **Putative, Geometric matches:** openMVG_main_exportMatches
- **Tracks:** openMVG_main_exportTracks

Or start the 3D reconstruction:

openMVG_main_IncrementalSfM

The [\[ACSfM\]](#) SfM is an evolution of the implementation used for the paper “Adaptive Structure from Motion with a contrario model estimation” published at ACCV 2012.

The incremental pipeline is a growing reconstruction process. It starts from an initial two-view reconstruction (the seed) that is iteratively extended by adding new views and 3D points, using pose estimation and triangulation. Due to the incremental nature of the process, successive steps of non-linear refinement, like Bundle Adjustment (BA) and Levenberg-Marquardt steps, are performed to minimize the accumulated error (drift).

Algorithm of the Incremental/Sequential Structure from Motion

```
Require: internal camera calibration (possibly from EXIF data)
Require: pairwise geometry consistent point correspondences
Ensure: 3D point cloud
Ensure: camera poses
compute correspondence tracks t
compute connectivity graph G (1 node per view, 1 edge when enough matches)
```

```

pick an edge e in G with sufficient baseline
* robustly estimate essential matrix from images of e
triangulate validated tracks, which provides an initial reconstruction
contract edge e
while G contains an edge do
  pick edge e in G that maximizes union(track(e), 3D points)
  * robustly estimate pose (external orientation/resection)
  triangulate new tracks
  contract edge e
  perform bundle adjustment
end while

```

Information and usage

The chain is designed to run on a sfm_data.json file and some pre-computed matches.

```

$ openMVG_main_IncrementalSfM -i Dataset/matches/sfm_data.json -m Dataset/
↪matches/ -o Dataset/out_Incremental_Reconstruction/

```

openMVG_main_IncrementalSfM displays to you some initial pairs that share an important number of common point. Please select two image index that are convergent and the 3D reconstruction will start. The initial pair must be choosen with numerous correspondences while keeping a wide enough baseline.

Arguments description:

Required parameters:

- [-i|-input_file]
 - a SfM_Data file
- [-m|-matchdir]
 - path were geometric matches were stored
- [-o|-outdir]
 - path where the output data will be stored

Optional parameters:

- [-a|-initialPairA NAME]
 - the filename image to use (i.e. 100_7001.JPG)
- [-b|-initialPairB NAME]
 - the filename image to use (i.e. 100_7002.JPG)
- [-c|-camera_model]
 - The camera model type that will be used for views with unknown intrinsic:
 - * 1: Pinhole
 - * 2: Pinhole radial 1
 - * 3: Pinhole radial 3 (default)
 - * 4: Pinhole radial 3 + tangential 2
 - * 5: Pinhole fisheye

- **[-fl-refineIntrinsics]** User can control exactly which parameter will be considered as constant/variable and combine them by using the 'l' operator.
 - ADJUST_ALL -> refine all existing parameters (default)
 - NONE -> intrinsic parameters are held as constant
 - ADJUST_FOCAL_LENGTH -> refine only the focal length
 - ADJUST_PRINCIPAL_POINT -> refine only the principal point position
 - ADJUST_DISTORTION -> refine only the distortion coefficient(s) (if any)
 - NOTE Options can be combined thanks to 'l':
 - * ADJUST_FOCAL_LENGTH|ADJUST_PRINCIPAL_POINT -> refine the focal length & the principal point position
 - * ADJUST_FOCAL_LENGTH|ADJUST_DISTORTION -> refine the focal length & the distortion coefficient(s) (if any)
 - * ADJUST_PRINCIPAL_POINT|ADJUST_DISTORTION -> refine the principal point position & the distortion coefficient(s) (if any)

openMVG_main_GlobalSfM

[GlobalACSfM] is based on the paper “Global Fusion of Relative Motions for Robust, Accurate and Scalable Structure from Motion.” published at ICCV 2013.

Multi-view structure from motion (SfM) estimates the position and orientation of pictures in a common 3D coordinate frame. When views are treated incrementally, this external calibration can be subject to drift, contrary to global methods that distribute residual errors evenly. Here the method propose a new global calibration approach based on the fusion of relative motions between image pairs.

Algorithm of the Global Structure from Motion

```
Require: internal camera calibration (possibly from EXIF data)
Require: pairwise geometry consistent point correspondences
Ensure: 3D point cloud
Ensure: camera poses

compute relative pairwise rotations
detect and remove false relative pairwise rotations
  - using composition error of triplet of relative rotations
compute the global rotation
  - using a dense least square and approximated rotations
compute relative translations
  - using triplet of views for stability and colinear motion support
compute the global translation
  - integration of the relative translation directions using a  $l-\infty$  method.
final structure and motion
  - link tracks validated per triplets and compute global structure by triangulation,
  - refine estimated parameter in a 2 step Bundle Adjustment
    - refine structure and translations
    - refine structure and camera parameters (rotations, translations).
```

Information and usage

The chain is designed to run on a `sfm_data.json` file and some pre-computed matches. The chain will only consider images with known approximate focal length. Image with invalid intrinsic id will be ignored.

```
$ openMVG_main_GlobalSfM -i Dataset/matches/sfm_data.json -m Dataset/matches/
↪ -o Dataset/out_Global_Reconstruction/
```

Arguments description:

Required parameters:

- **[-i|--input_file]**
 - a SfM_Data file
- **[-m|--matchdir]**
 - path where geometric matches were stored
- **[-o|--outdir]**
 - path where the output data will be stored

Optional parameters:

- **[-r|--rotationAveraging]**
 - 1: L1 rotation averaging *_[Chatterjee]*
 - 2: (default) L2 rotation averaging *_[Martinec]*
- **[-t|--translationAveraging]**
 - 1: (default) L1 translation averaging *_[GlobalACSfM]*
 - 2: L2 translation averaging *_[Kyle2014]*
 - 3: (default) SoftL1 minimization *_[GlobalACSfM]*
- **[-f|--refineIntrinsics]** User can control exactly which parameter will be considered as constant/variable and combine them by using the 'l' operator.
 - ADJUST_ALL -> refine all existing parameters (default)
 - NONE -> intrinsic parameters are held as constant
 - ADJUST_FOCAL_LENGTH -> refine only the focal length
 - ADJUST_PRINCIPAL_POINT -> refine only the principal point position
 - ADJUST_DISTORTION -> refine only the distortion coefficient(s) (if any)
 - NOTE Options can be combined thanks to 'l':
 - * ADJUST_FOCAL_LENGTHlADJUST_PRINCIPAL_POINT -> refine the focal length & the principal point position
 - * ADJUST_FOCAL_LENGTHlADJUST_DISTORTION -> refine the focal length & the distortion coefficient(s) (if any)
 - * ADJUST_PRINCIPAL_POINTlADJUST_DISTORTION -> refine the principal point position & the distortion coefficient(s) (if any)

_[GlobalACSfM] default settings are “-r 2 -t 3”.

Tips

As a dense image network is required to perform global SfM it is required to detect more **Regions points** per image to ensure a high probability of matching.

Please use:

- “-p HIGH” option on openMVG_main_ComputeFeatures
- “-r .8” on openMVG_main_ComputeMatches.

Export detected regions as SVG files:

- **Detected keypoints:** openMVG_main_exportKeypoints

Automatic pixel focal computation from JPEG EXIF metadata

Intrinsic image analysis is made from JPEG EXIF metadata and a database of camera sensor width.

Note: If you have image with no metadata you can specify the known pixel focal length value directly or let the SfM process find it automatically (at least two images that share common keypoints and with valid intrinsic group must be defined)

The focal in pixel is computed as follow (if the EXIF camera model and maker is found in the provided sensor width database)

$$\text{focal}_{pix} = \frac{\max(w_{pix}, h_{pix}) * \text{focal}_{mm}}{\text{ccd}w_{mm}}$$

- focal_{pix} the EXIF focal length (pixels),
- focal_{mm} the EXIF focal length (mm),
- w_{pix}, h_{pix} the image of width and height (pixels),
- $\text{ccd}w_{mm}$ the known sensor width size (mm)

From lists.txt to sfm_data.json

Old openMVG version (<0.8) use a lists.txt file to describer image parameters.

Example of a lists.txt file where focal is known in advance

```
0000.png;3072;2048;2760;0;1536;0;2760;1024;0;0;1
0001.png;3072;2048;2760;0;1536;0;2760;1024;0;0;1
0002.png;3072;2048;2760;0;1536;0;2760;1024;0;0;1
...
```

Example of a lists.txt file where focal is computed from exif data

```
100_7100.JPG;2832;2128;2881.25;EASTMAN KODAK COMPANY;KODAK Z612 ZOOM DIGITAL CAMERA
100_7101.JPG;2832;2128;2881.25;EASTMAN KODAK COMPANY;KODAK Z612 ZOOM DIGITAL CAMERA
100_7102.JPG;2832;2128;2881.25;EASTMAN KODAK COMPANY;KODAK Z612 ZOOM DIGITAL CAMERA
...
```

You can convert this file to a valid sfm_data.json file by using the **openMVG_main_ConvertList** application.

2. Image description computation

3. Corresponding images and correspondences computation

4. SfM solving (2 methods)

5. Optional further processing

openMVG_main_ComputeSfM_DataColor

Compute the color of the Structure of a sfm_data scene.

Use a very simple approach:

```
a. list the track id with no color
b. list the most viewed view id
c. color the track that see the view
d. go to a. until uncolored track are remaining
```

Information and usage

The application is designed to run on a sfm_data.json file The sfm_data file should contains:

- valid view with some defined intrinsics and camera poses,
- (optional existing structure).

```
$ openMVG_main_ComputeSfM_DataColor -i Dataset/out_Reconstruction/sfm_data.json -
  ↳o Dataset/out_Reconstruction/sfm_data_color.ply
```

Arguments description:

Required parameters:

- [-i]

 - a SfM_Data file
- [-o]

 - output scene with updated landmarks color

openMVG_main_ComputeStructureFromKnownPoses

This application compute corresponding features and robustly triangulate them according the geometry of the known camera intrinsics & poses.

Algorithm of the application

```
Require: internal + external camera calibration
Require: image description regions (features + descriptors)
Ensure: 3D point cloud
compute image visibility
  list all the pair that share common visual content
    - camera frustum based
    - or structure visibility (SfM tracks) based
list triplets of view from pairs
```

```
for each triplets compute 3 view tracks
  if tracks triangulable add correspondences to p
link 3 views validated matches (p) as tracks
robustly triangulate them
```

Information and usage

The chain is designed to run on a sfm_data.json file and some pre-computed matches. The sfm_data file should contains: - valid view with some defined intrinsics and camera poses, - (optional existing structure).

```
$ openMVG_main_ComputeStructureFromKnownPoses -i Dataset/out_Reconstruction/
→sfm_data.json -o Dataset/out_Reconstruction/robustFitting.json
```

Arguments description:

Required parameters:

- [-il-input_file]
 - a SfM_Data file with valid intrinsics and poses and optional structure
- [-ml-matchdir]
 - path where image descriptions were stored
- [-ol-outdir]
 - path where the updated scene data will be stored

Optional parameters:

- [-fl-match_file]
 - path to a matches file (pairs of the match files will be listed and used)
- [-pl-pair_file]
 - path to a pairs file (only those pairs will be considered to compute the structure) The pair file is a list of view indexes, one pair on each line
- [-bl-bundle_adjustment]
 - perform a bundle adjustment on the scene (OFF by default)
- [-rl-residual_threshold]
 - maximal pixels reprojection error that will be considered for triangulations (4.0 by default)

openMVG_main_ExportUndistortedImages

This application export undistorted images from known camera parameter intrinsic.

Algorithm of the application

```
Require: internal + camera calibration
Require: images
Ensure: undistorted images
for each view
  if view has valid intrinsic
    undistort and save the undistorted view
```


Information and usage

The chain is designed to run on a `sfm_data.json` file. The `sfm_data` file should contains: - valid view with some defined intrinsics,

```
$ openMVG_main_ExportUndistortedImages -i Dataset/out_Reconstruction/sfm_
↳data.json -o Dataset/out_Reconstruction/undistortedImages
```

Arguments description:

Required parameters:

- **[-i|-input_file]**
 - a SfM_Data file with valid intrinsics and poses and optional structure
- **[-o|-outdir]**
 - path where the undistorted images will be stored
- **[-r|-exportOnlyReconstructed]**
 - Export only the images that have valid intrinsic and pose data (Can be 0(default) or 1)
- **[-n|-numThreads]**
 - number of thread(s)

5. Optional further processing (3rd party)

MVS: Multiple View Stereovision

Once camera position and orientation have been computed by OpenMVG, Multiple View Stereo-vision algorithms could be used to compute a dense representation of the scene by generating point clouds or mesh surfaces.

Since OpenMVG does not have itself a MVS module it proposes exporter to various existing solutions:

Export to CPMVS

OpenMVG exports [*CPMVS*] ready to use project (images, projection matrices and ini configuration file).

```
$ openMVG_main_openMVG2CPMVS -i Dataset/outReconstruction/sfm_data.json -o Dataset/
↳outReconstruction
```

MVE (Multi-View Environment)

MVE can import a converted openMVG SfM scene and use it to create dense depth map and complete dense 3D models.

```
# Convert the openMVG SfM scene to the MVE format
$ openMVG_main_openMVG2MVE -i Dataset/outReconstruction/sfm_data.json -o Dataset/
↳outReconstruction

#--
# shell script example
#--
```

```
directory=Dataset/outReconstruction/MVE
resolution=2

# MVE
dmrecon -s$resolution $directory
scene2pset -ddepth-L$resolution -iundist-L$resolution -n -s -c $directory $directory/
↳OUTPUT.ply
fssrecon $directory/OUTPUT.ply $directory/OUTPUT_MESH.ply
meshclean $directory/OUTPUT_MESH.ply $directory/OUTPUT_MESH_CLEAN.ply

Call any tool without arguments to see the help.
```

You will need to compile MVE tools and [FSSR](#).

Export to MVS Texturing

If you don't want to use the full MVE pipeline but only [MVS Texturing \[Waechter2014\]](#) to project a set of oriented images on a mesh, one solution is to use the `openMVG_main_openMVG2MVSTEXTURING` binary. This binary converts your `SfM_Data` file into one format used by MVS Texturing. In addition, you may need to undistort your images with `openMVG_main_ExportUndistortedImages` as it's not handled by the `openMVG_main_openMVG2MVSTEXTURING` tool.

OpenMVS Open Multiple View Stereovision

[OpenMVS](#) allows to compute dense points cloud, surface and textured surfaces of OpenMVG scenes.

To use OpenMVG scene in OpenMVS you can:

- use the `openMVG_main_openMVG2openMVS` exporter interface
- or use the `OpenMVS InterfaceOpenMVG SceneImporter`

Importation/Exportation

```
# Export the OpenMVG scene to the OpenMVS data format
$ openMVG_main_openMVG2openMVS -i PATH/sfm_data.(json/xml/bin) -d OUTPUT_PATH -o_
↳OUTPUT_PATH/Scene

# Or you can Import the OpenMVG scene to the OpenMVS data format using the OpenMVS_
↳binary
$ InterfaceOpenMVG -i PATH/sfm_data.json -o scene.mvs
```

Dense point-cloud reconstruction for obtaining a complete and accurate as possible point-cloud

```
# Compute dense depth map per view and merge the depth map into a consistent point_
↳cloud
$ DensifyPointCloud scene.mvs
```

Mesh reconstruction for estimating a mesh surface that explains the best the input point-cloud

```
# The initial point cloud be:
# - the calibration one (scene.mvs),
$ ReconstructMesh scene.mvs
# - or the dense one (scene_dense.mvs)
$ ReconstructMesh scene_dense.mvs
```

Mesh texturing for computing a texture to color the mesh

```
# Compute the texture
$ TextureMesh scene_dense_mesh.mvs
```

Export to PMVS/CMVS

OpenMVG exports [\[PMVS\]](#) ready to use project (images, projection matrices and pmvs_options.txt files).

Once a 3D calibration have been computed you can convert the SfM_Ouput files to a PMVS project.

```
$ openMVG_main_openMVG2PMVS -i Dataset/outReconstruction/sfm_data.json -o Dataset/
→outReconstruction
$ pmvs Dataset/outReconstruction/PMVS/ pmvs_options.txt
```

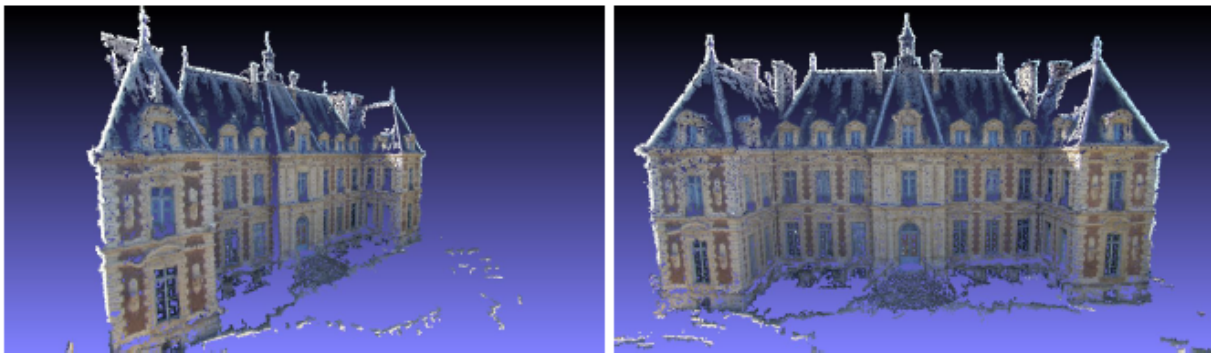


Fig. 3.2: Figure: Multiple View Stereo-vision point cloud densification on the estimated scene using [\[PMVS\]](#).

In order to use CMVS for large scene openMVG2PMVS exports also the scene in the Bundler output format.

```
$ openMVG_main_openMVG2PMVS -i Dataset/outReconstruction/sfm_data.json -o Dataset/
→outReconstruction
$ cmvs Dataset/outReconstruction/PMVS/ [MaxImageCountByCluster=100]
$ cmvs Dataset/outReconstruction/PMVS/ 30
$ genOption Dataset/outReconstruction/PMVS/
$ sh Dataset/outReconstruction/PMVS/pmvs.sh
```

You can either run by hand all the process or use pre-defined python scripts (that are using some default options).

OpenMVG SfM pipelines demo

A complete ready to use tutorial demo is exported in your build directory. It clones an image dataset and run the SfM pipelines on it:

- openMVG_Build/software/SfM/tutorial_demo.py

In order to use easily the Sequential or the Global pipeline, ready to used script are exported in your build directory:

- openMVG_Build/software/SfM/SfM_SequentialPipeline.py
- openMVG_Build/software/SfM/SfM_GlobalPipeline.py

To use them simply run:

```
$ cd openMVG_Build/software/SfM/
$ python SfM_SequentialPipeline.py [full path image directory] [resulting directory]
$ python SfM_SequentialPipeline.py ~/home/user/data/ImageDataset_SceauxCastle/images ~
↪ /home/user/data/ImageDataset_SceauxCastle/Castle_Incremental_Reconstruction

$ python SfM_GlobalPipeline.py [full path image directory] [resulting directory]
```

More details about openMVG tools

To know more about each tool visit the following link and read the doc below:

SfM_data format

SfM_Data is a data container. It contains:

- **Views** - images
- **Intrinsics** - intrinsics camera parameters
- **Poses** - extrinsic camera parameters
- **Landmarks** - 3D points and their 2D Observations

The View and Intrinsic concept are abstract and so:

- any internal camera model can be used
- any metadata can be stored related to a view.

Dynamic loading of stored object is performed thanks to the cereal serialization library (this library allow polymorphism serialization).

PS: We strongly advise to use a 3 directories based data organisation structure

- **images**
 - your image sequence.
- **matches**
 - directory used to store image information, images features, descriptors and matches information.
- **outReconstruction**
 - directory used to store the SfM result and process log.

Ground Control Points registration

Structure from Motion find camera poses & a structure to an unknown scale of the real metric measure. In order to restore the correct scale and orientation to known 3D coordinates, OpenMVG provide a GUI to perform Ground Control Point registration.

- Given some 3D known ground control points and their 2D observations in some images, we can compute the rigid transformation to move the SfM scene to the known 3D coordinates. The rigid transformation (7Dof) is computed in least square between the triangulation of the observations and the known 3D positions).

- For your ground control points you can use specific pattern or object for which you know the measures (post-it, ruler, calibration target...).

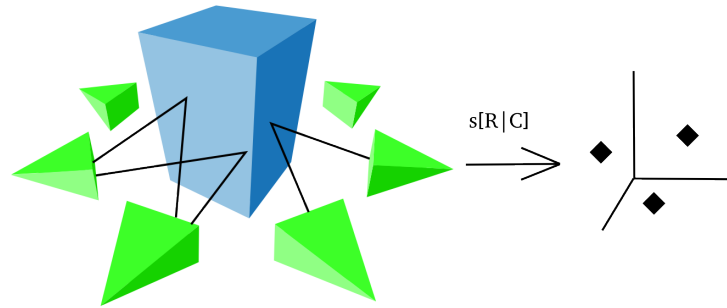
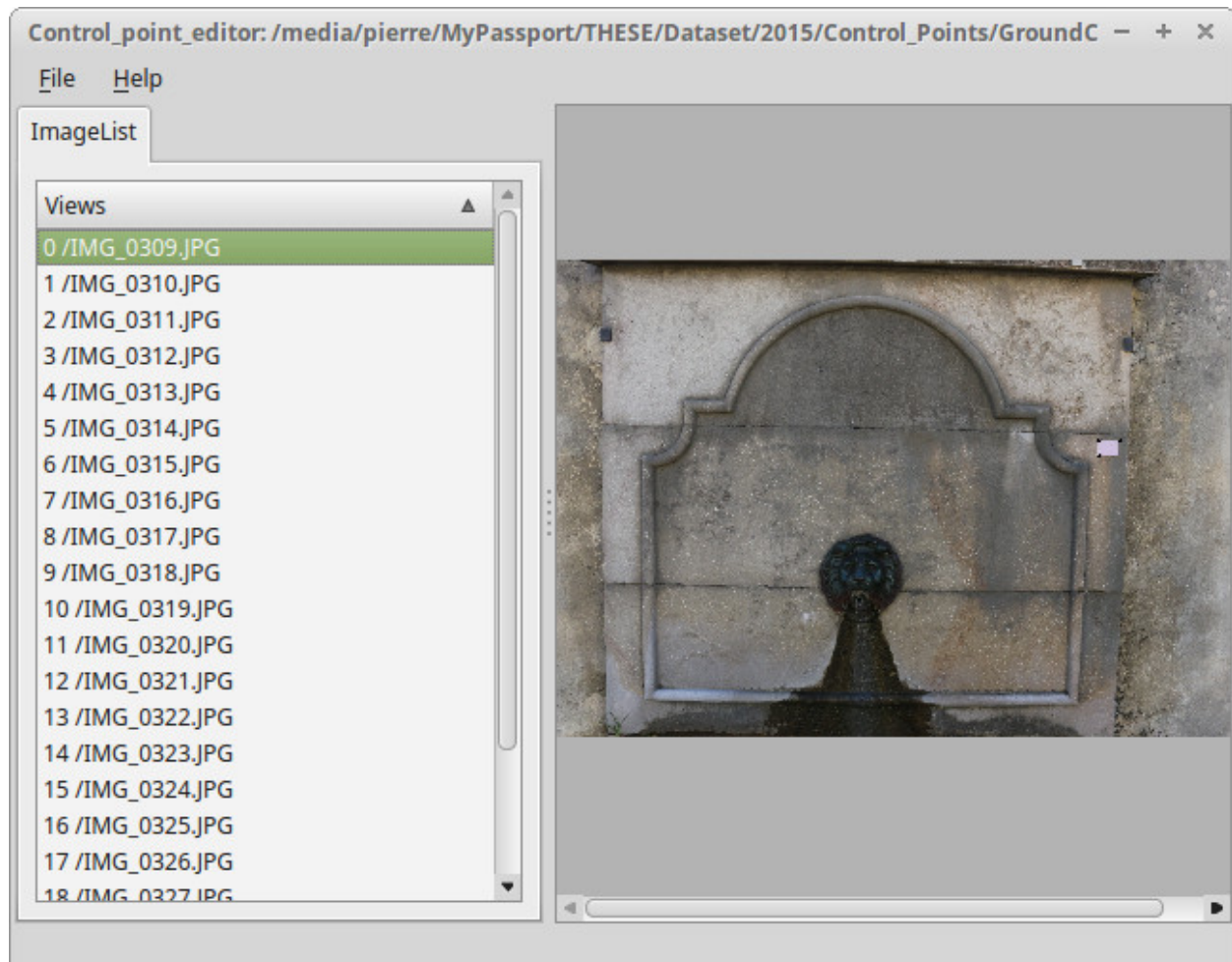


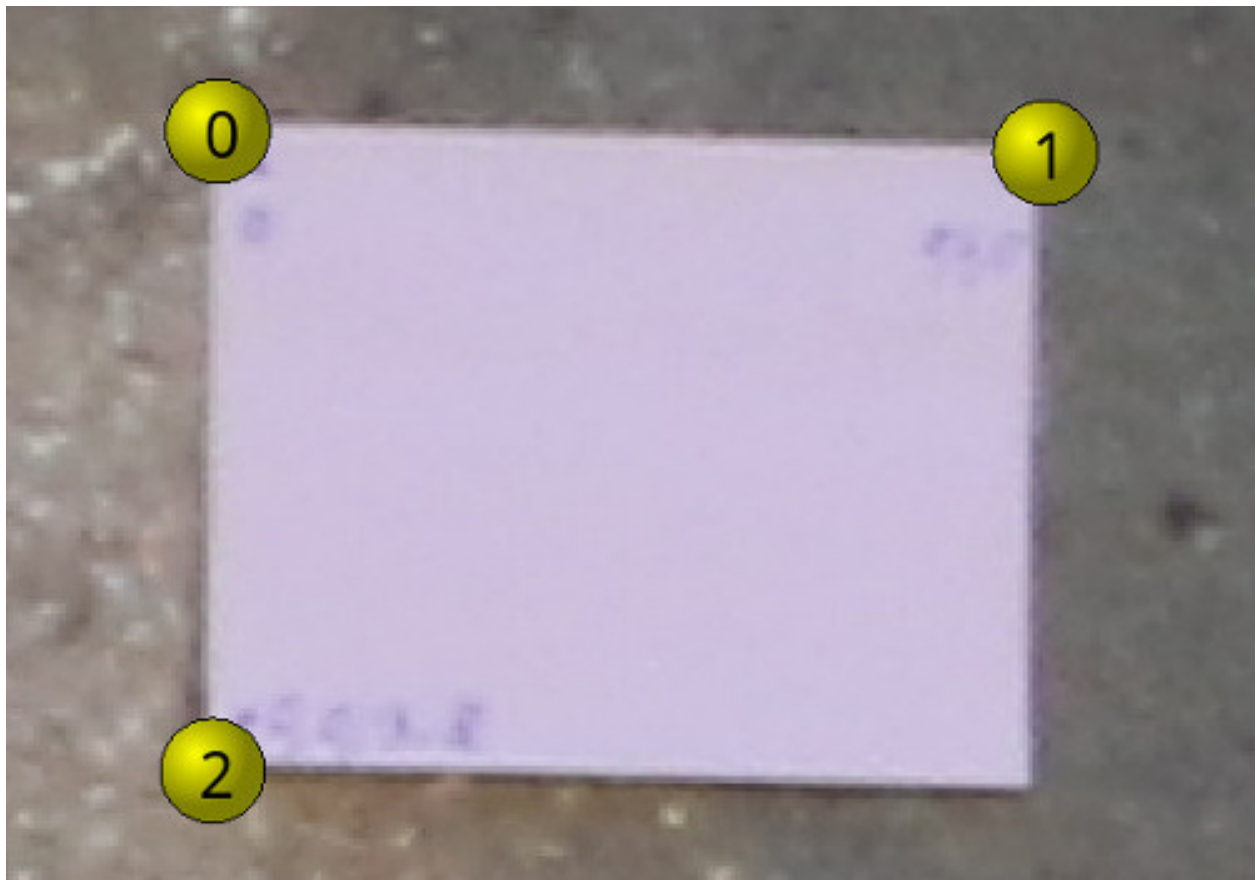
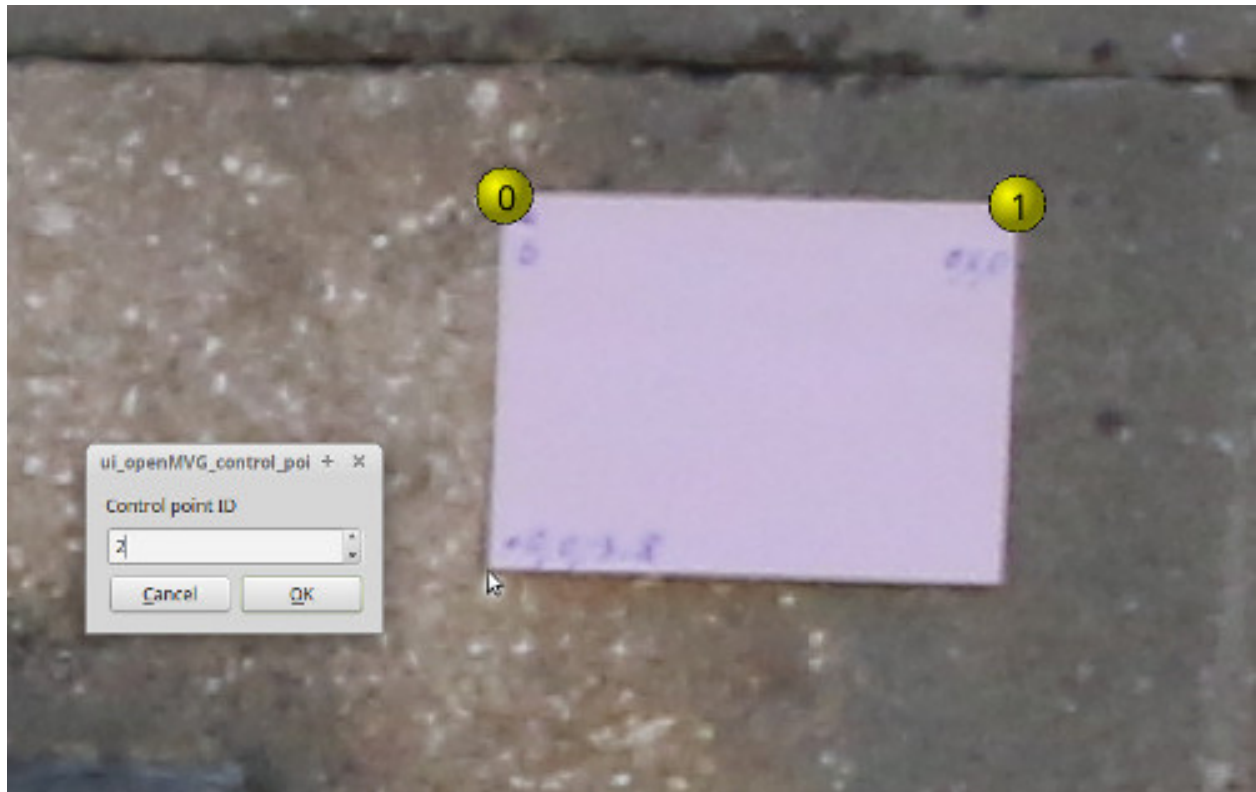
Fig. 3.3: Figure : Ground Control Point registration principle illustration.

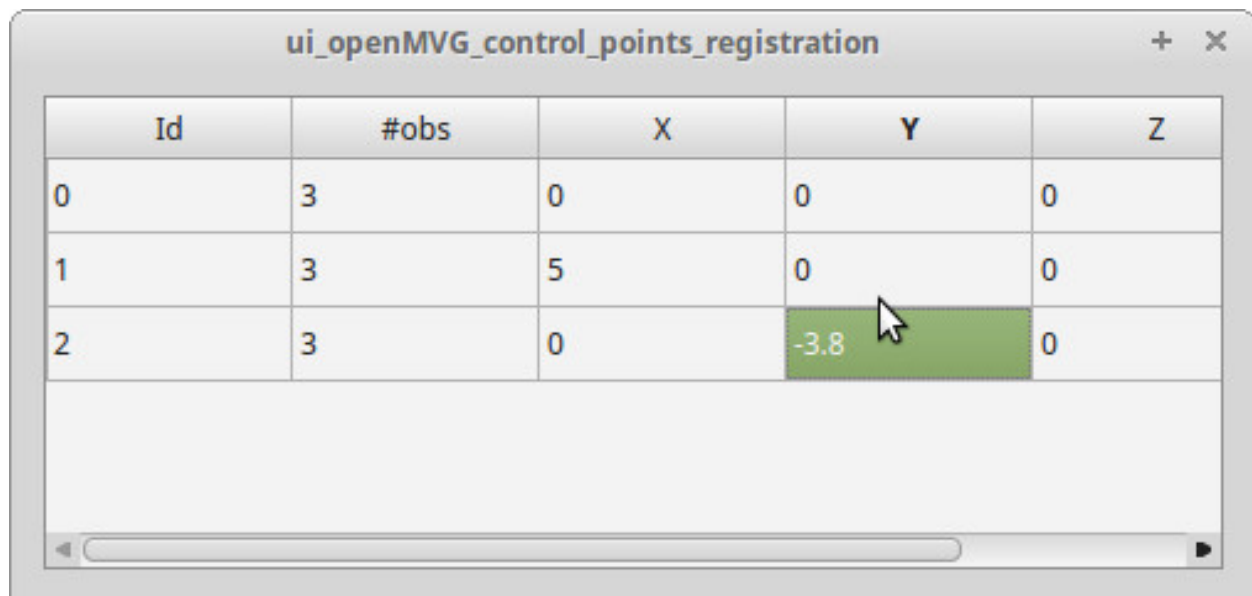
Short tutorial

In this sample we scale the scene from the known dimensions of a paper target (see the post-it on the images below).

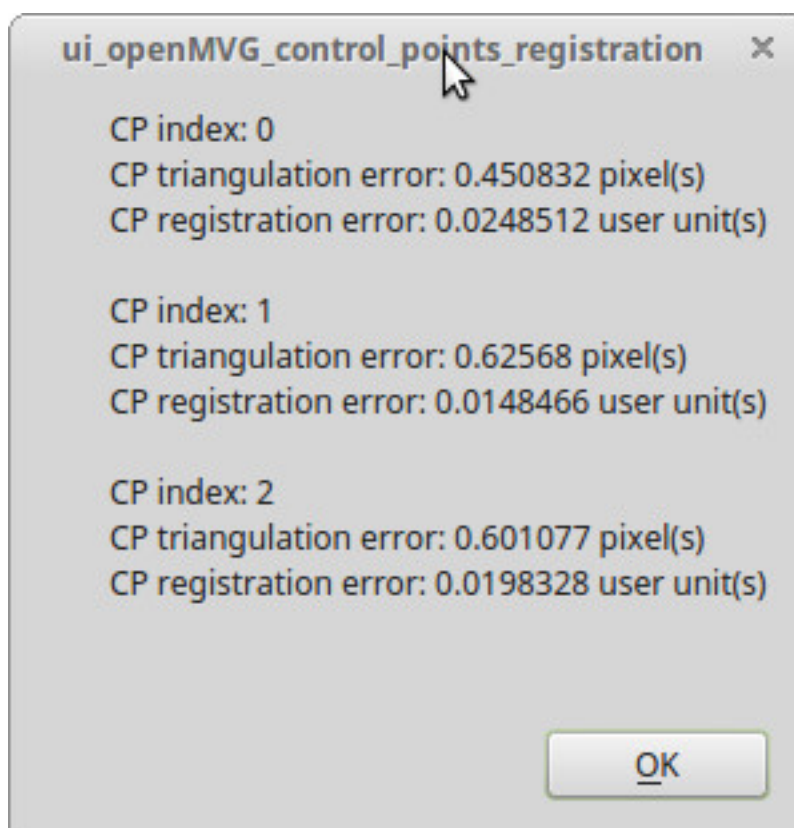
- 1. Open your scene in the GUI
 - File/Open Project (CTRL+O) and open an sfm_data file.
 - double click on an image on the list to display it on the right
 - zoom level can be changed with CTRL+WHEEL
- 2. Set your GCP observations positions within the images
 - choose your image,
 - click inside it (at the right position or not, you can move it later) & select your GCP unique ID,
 - then the Control point observation appears
 - to move your GCP click to it, and keep left mouse button clicked while moving it.
- 3. Provide the 3D GCP known X,Y,Z positions
 - File/Control Point Edition (CTRL+E)
 - double click in the X,Y,Z cells and edit the 3D coordinates
 - close the window to save them
- 4. Perform the registration
 - File/Control Point Registration (CTRL+R)
 - If at least 3D control points exist and iff their observations are triangulable, the registration will be performed and a report appears.
- 5. Save your scene
 - File/Save Project (CTRL+S) and save your sfm_data file.
 - Post-process or use directly the point cloud to perform measures.
 - i.e here once MVS have been computed we can measure the eyes distance => ~5 cm







Id	#obs	X	Y	Z
0	3	0	0	0
1	3	5	0	0
2	3	0	-3.8	0





Localization

This module provide tools to localize images into an existing Structure from Motion scene.

openMVG_main_SfM_Localization

```
$ openMVG_main_SfM_Localization -i [] -m [] -o [] -q []
```

Arguments description:

Required parameters:

- **[-il-input_file]** The input SfM_Data scene (must contains a structure and camera poses, eg. the sfm_data.bin generated by SfM)
- **[-ml-match_dir]** path to the matches that corresponds to the provided SfM_Data scene
- **[-ol-out_dir]** path to save the found camera position
- **[-ul-match_out_dir]** path to the directory where new matches will be stored. If empty matches are stored in match_dir directory.
- **[-ql-query_image_dir]** path to an image OR to the directory containing the images that must be localized (the directory can also contain the images from the initial reconstruction)

Optional parameters:

- **[-rl-residual_error]** upper bound of the residual error tolerance
- **[-sl-single_intrinsics]** (switch) when switched on, the program will check if the input sfm_data contains a single intrinsics and, if so, take this value as intrinsics for the query images. (OFF by default)
- **[-el-export_structure]** (switch) when switched on, the program will also export structure to output sfm_data while OFF will only export VIEWS, INTRINSICS and EXTRINSICS. (OFF by default)
- **[-nl-numThreads]** number of thread(s)

```
// Example
$ openMVG_main_SfM_Localization -i /home/user/Dataset/ImageDataset_SceauxCastle/
↪reconstruction/sfm_data.bin -m /home/user/Dataset/ImageDataset_SceauxCastle/matches_
↪-o ./ -q /home/user/Dataset/ImageDataset_SceauxCastle/images/100_7100.JPG
```

Geodesy: Usage of GPS prior

Contents

- *Geodesy: Usage of GPS prior*
 - *Introduction*
 - *Usage of pose prior*
 - *For SfM to XYZ (GPS) registration*
 - *Use-case: command line used for a flat UAV survey*

Introduction

In order to compute scene in the user coordinates system, the user can use some Geodesy data (i.e GPS or XYZ prior). Those informations can be used in two ways in OpenMVG:

To speed up the pairwise feature matching step:

- limit the number of pair by matching only neighbor images.

For SfM:

- As rigid prior,
- As non rigid prior.

Usage of pose prior

To speed up the pairwise feature matching step

```
$ openMVG_main_ListMatchingPairs -G -n 5 -i Dataset/matching/sfm_data.bin -o_
↪Dataset/matching/pair_list.txt
```

Required parameters:

- **[-il-input_file]** The input SfM_Data file (must have pose prior see “Scene initialization” below)
- **[-ol-output_file]** The pair list filename (will be send to main_ComputeMatches to limit the number of pairs)

Enable the neighboring view limitation

- **[-G|-gps_mode]** Enable the computation of GPS neighborhood
- **[-nl|-neighbor_count]** The number of maximum neighbor per poses (list the N closest poses GPS corresponding XYZ positions.)

For SfM to XYZ (GPS) registration

Rigid Registration [SfM]

Once you have performed SfM on your scene, you can adjust your scene coordinate system to fit your GPS datum. A 7DoF similarity can be computed between the SfM camera center position & the image GPS datum. The found similarity is applied to Camera position & landmarks.

Note: GPS conversion to XYZ Image position is computed in ECEF by using valid WGS84 GPS latitude, longitude & altitude EXIF data.

```
$ openMVG_main_geodesy_registration_to_gps_position -i Dataset/out_
↪Reconstruction/sfm_data.bin -o Dataset/out_Reconstruction/sfm_data_
↪adjusted.bin
```

Arguments description:

Required parameters:

- **[-il-input_file]**
 - a SfM_Data file with valid intrinsics and poses and optional structure

- [-o|--output_file]
 - filename in which the registered SfM_Data scene must be saved

Optional parameters:

- [-m|--method] method that will be used to compute the GPS registration
 - 0: Registration is done by using a robust estimation (Minimization of the Median fitting error).
 - 1(default): Registration is done using all corresponding points.

Non-Rigid Registration (adaptive registration) [SfM]

Pose prior (GPS/Odometry) can be use as a non rigid prior during OpenMVG SfM Bundle Adjustment phase. To do so, XYZ pose prior must be set in the initial SfM_Data file and the usage of prior must be set in the SfM binaries.

Here the command line change that you must notice in order to use the GPS prior data.

- First you must initialize the scene with XYZ view's pose prior.
- Then you must tell to the SfM pipeline to use the pose prior data.

Scene initialization

```
$ openMVG_main_SfMInit_ImageListing -P [other args]
```

Optional parameters:

- [-P]
 - Setup a XYZ position prior for each view that have valid EXIF GPS data.
- [-w]
 - Allow to customize the pose prior weight in each XYZ dimention i.e. default is equivalent to “1.0;1.0;1.0”

Force pose prior usage

```
$ openMVG_main_IncrementalSfM -P [other args]
$ openMVG_main_GlobalSfM -P [other args]
```

Optional parameters:

- [-P]
 - Enable the usage of view's motion priors data.

Use-case: command line used for a flat UAV survey

Launched on the dataset Swiss Quarry [Geotagged multispectral images](#).

```
// Initialize the scene
// Pose prior for each view will be set thanks to the GPS data
openMVG_main_SfMInit_ImageListing \
-P \
-d sensor_width_camera_database.txt \
```

```

-i /media/pierre/SenseFly/Quarry/geotagged-images/ \
-o /media/pierre/SenseFly/Quarry/quary_output/

thread_count=8
openMVG_main_ComputeFeatures \
-i /media/pierre/SenseFly/Quarry/quary_output/sfm_data.json \
-o /media/pierre/SenseFly/Quarry/quary_output/matches \
-n $thread_count

// Limit the number of pairs that will be used for matching
// -> limit to pose neighborhood thanks to the pose center prior location
openMVG_main_ListMatchingPairs \
-G \
-n 8 \
-i /media/pierre/SenseFly/Quarry/quary_output/sfm_data.json \
-o /media/pierre/SenseFly/Quarry/quary_output/matches/pair_list.txt

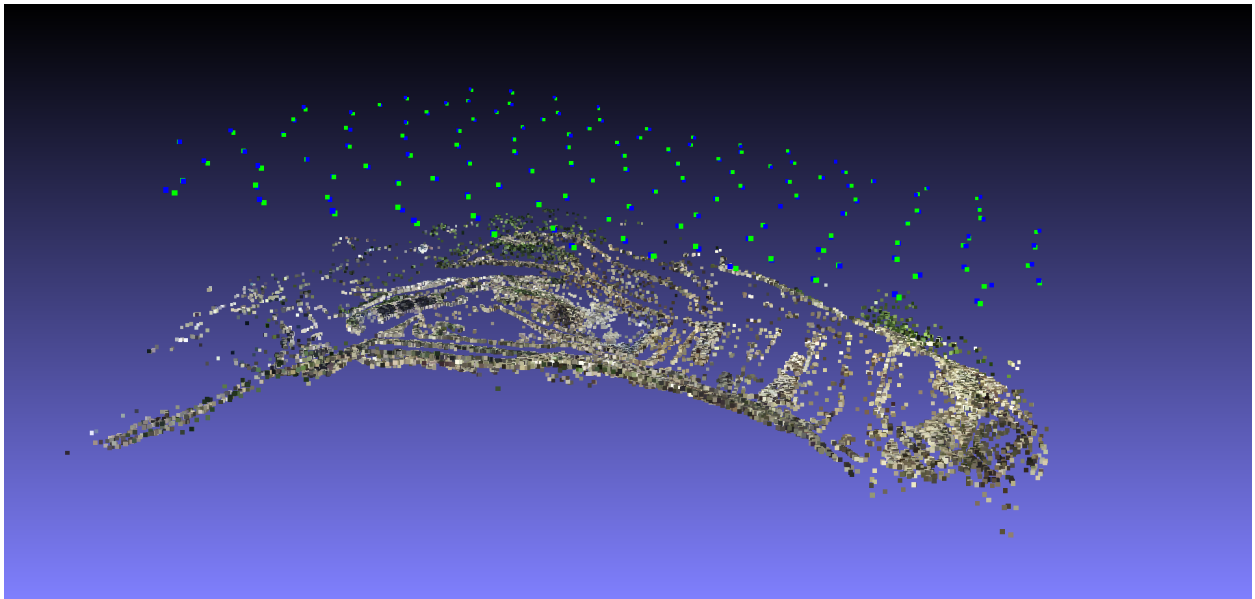
openMVG_main_ComputeMatches \
-i /media/pierre/SenseFly/Quarry/quary_output/sfm_data.json \
-l /media/pierre/SenseFly/Quarry/quary_output/matches/pair_list.txt \
-g e \
-o /media/pierre/SenseFly/Quarry/quary_output/matches/

openMVG_main_GlobalSfM \
-P \
-i /media/pierre/SenseFly/Quarry/quary_output/sfm_data.json \
-m /media/pierre/SenseFly/Quarry/quary_output/matches/ \
-o /media/pierre/SenseFly/Quarry/quary_output/global_reconstruction

```

Here the final results:

In green the SFM poses, and in blue the GPS Exif ECEF poses. We clearly see that the two coordinates system are close.



Color Harmonization

In a multiple-view image acquisition process, color consistency is not ensured. This is an important problem for image fusion tasks: object texturing or mosaics blending for example. In automatic mode, the camera adapts its settings – shutter-speed and aperture– to the captured image content. Therefore the color of objects changes over an image sequence.



In order to restore the color consistency, a transformation model between reference and observed colors have to be estimated. It introduces two main problems:

- the data selection (common pixels between images),
- the estimation of a reliable color transformation between those pixels.

This module propose an interface to solve this problem:

[Moulon13] propose a global multi-view color consistency solution that in a first step selects robustly the common color information between images and in a second step estimates the color transformations that set all pictures in a common color reference, which involves a global minimization. It uses a compact histogram representation of the shared color information per image pairs and solve the color consistency by using a linear program with a gain and offset model for each image.

A reference have to be choosen in order to set the color reference.

Here the obtained results on the image sequence after having choosen a “white” or “blue” image as reference:



The openMVG licence is MPL2 but some code used to provide interesting fonctionnalities have different licences or are subject to patent.

Such code takes place in the patented directory.

SIFT

In order to compute corresponding points between images pairs, openMVG uses natively SIFT keypoints and their associated descriptors. The used code is a subset of the [\[VLFEAT\]](#) library.

You can replace this keypoints and descriptors provided by any version of your choice to use openMVG in a non-research context. Suggestions for features points:

- CORNERS: HARRIS, FAST,
- REGIONS: MSER,
- BLOBS: AKAZE.

Descriptors:

- BINARY: M-LDB (see AKAZE paper), BRIEF, Nested shape descriptor,
- FLOATING POINT: DAISY, LIOP descriptors.

OpenMVG library uses the git submodule idea to make its repository lighter.

Here are the libraries used through the submodule concept:

GLFW

USAGE

openMVG uses GLFW library in order to provide an easy way to create OpenGL windows context.

Description

“*[GLFW]* is an Open Source, multi-platform library for creating windows with OpenGL contexts and managing input and events. It is easy to integrate into existing applications and does not lay claim to the main loop.

GLFW is written in C and has native support for Windows, OS X and many Unix-like systems using the X Window System, such as Linux and FreeBSD.

GLFW is licensed under the zlib/libpng license.”

OPENEXIF

USAGE

openMVG uses OpenEXIF library in order to read JPEG EXIF metadata.

Description

“*[OpenExif]* is an object-oriented library for accessing Exif formatted JPEG image files. The toolkits allow for creating, reading, and modifying the metadata in the Exif file. It also provides mean of getting and setting the main image and the thumbnail image. “

OSI CLP

USAGE

openMVG uses the *[OSI]* and the *[CLP]* solver in order to solve linear programs *[LP]*.

CLP has been choosen because it is known to support problems of up to 1.5 million constraints *[CLP FAQ]*.

[LPSOLVE] has been tested but tests shown that it is less reliable (sometimes, there is no convergence to a solution).

Description

[OSI] Osi (Open Solver Interface) provides an abstract base class to a generic linear programming (LP) solver, along with derived classes for specific solvers. Many applications may be able to use the Osi to insulate themselves from a specific LP solver. (CLP and MOSEK backend are supported in openMVG).

[CLP] Clp (Coin-or linear programming) is an open-source linear programming solver written in C++.

openMVG use some third party libraries. It is not necessary to write again something that exists and already well designed. The libraries are included in the openMVG in order to ease compilation and usage.

ceres-solver

[Ceres] Solver is a portable C++ library that allows for modeling and solving large complicated nonlinear least squares problems.

The cxsparse backend is included in openMVG.

cmdLine

A tiny library to handle C++ command line with named parameters.

CppUnitLite

[CppUnitLite] is a tiny library to handle UNIT test.

Eigen

[Eigen] is a high-level C++ library of template headers for linear algebra, matrix and vector operations, numerical solvers.

Eigen is used for:

- dense matrix and array manipulations,
- sparse matrix container.

flann

[FLANN] is a library for performing fast approximate nearest neighbor searches in high dimensional spaces.

histogram

A tiny class to compute data distribution of data in a provided data range.

htmlDoc

A class to ease HTML data export

lemon

[LEMON] stands for Library for Efficient Modeling and Optimization in Networks. It is a C++ template library providing efficient implementations of common data structures and algorithms with focus on combinatorial optimization tasks connected mainly with graphs and networks.

progress

A tiny class to handle progress status of an application

stlAddition

A collection of tools to extend the STL library.

stlplus3

[STLplus] is a collection of reusable C++ components for developers already familiar with the STL. Only the `file_system` component is used: It provides access to a set of operating-system independent file and folder handling functions.

Image Input/Output

- jpeg
- png
- zlib

CHAPTER 7

FAQ

To fill

CHAPTER 8

Bibliography

OpenMVG (Multiple View Geometry) is a library for computer-vision scientists and targeted for the Multiple View Geometry community.

It is designed to provide an easy access to:

- Accurate Multiple View Geometry problem solvers,
- Tiny libraries to perform tasks from feature detection/matching to Structure from Motion,
- Complete Structure from Motion pipelines.

CHAPTER 9

Why another library

The openMVG credo is “**Keep it simple, keep it maintainable**”.. It means provide a readable code that is easy to use and modify by the community.

All the **features and modules are unit-tested**. This test driven development ensures:

- more consistent repeatability (assert code works as it should in time),
- that the code is used in a real context to show how it should be used.

OpenMVG is cut in various modules:

- **Libraries**, core modules,
 - comes with unit tests that assert algorithms results and show how use the code.
- **Samples**,
 - show how to use the library to build high_level algorithms.
- **Softwares**,
 - ready to use tools to perform toolchain processing:
 - features matching in un-ordered photo collection,
 - SfM: tools and Structure from Motion pipelines,
 - color harmonization of photo collection.

Cite Us

If you use openMVG for a publication, please cite it as:

```
@misc{openMVG,  
  author = "Pierre Moulon and Pascal Monasse and Renaud Marlet and Others",  
  title = "OpenMVG",  
  howpublished = "\url{https://github.com/openMVG/openMVG}",  
}
```


CHAPTER 11

Acknowledgements

openMVG authors would like to thank:

- libmv authors for providing an inspiring base to design the openMVG library,
- Mikros Image, LIGM-Imagine laboratory and Foxel SA for support,
- Mikros Image, LIGM-Imagine laboratory authorization to make this library an open-source project.

CHAPTER 12

License

openMVG library is release under the MPL2 (Mozilla Public License 2.0). It integrates some sub-part under the MIT (Massachusetts Institute of Technology) and the BSD (Berkeley Software Distribution) license. Please refer to the `copyright.md` and `license` files contained in the source for complete license description.

CHAPTER 13

Dependencies

OpenMVG comes as a standalone distribution, you don't need to install libraries to make it compile and run. On Linux openMVG will use the local png, zlib and jpeg libraries if they are available.

CHAPTER 14

Indices and tables

- `genindex`
- `modindex`
- `search`

Bibliography

- [GLFW] <http://www.glfw.org/>
- [OpenExif] <http://openexif.sourceforge.net/>
- [OSI] <https://projects.coin-or.org/Osi>
- [CLP] <https://projects.coin-or.org/Clp/wiki>
- [CLP_FAQ] <https://projects.coin-or.org/Clp/wiki/FAQ>
- [LPSOLVE] <http://lpsolve.sourceforge.net/5.5/>
- [Ke] **An Efficient Algebraic Solution to the Perspective-Three-Point Problem.** Ke, T.; Roumeliotis, S. CVPR 2017
- [Kneip] **A Novel Parametrization of the P3P-Problem for a Direct Computation of Absolute Camera Position and Orientation.** Kneip, L.; Scaramuzza, D. ; Siegwart, R. CVPR 2011
- [HZ] **Multiple view geometry in computer vision.** Hartley, Richard, and Andrew Zisserman. Vol. 2. Cambridge, 2000.
- [Stewenius] **Recent Developments on Direct Relative Orientation.** H. Stewenius, C. Engels and D. Nister. ISPRS 2006
- [Nister] **An Efficient Solution to the Five-Point Relative Pose.** D. Nister PAMI 2004
- [Longuet] **A computer algorithm for reconstructing a scene from two projections.** Longuet-Higgins, H. C. Readings in Computer Vision: Issues, Problems, Principles, and Paradigms, MA Fischler and O. Firschein, eds (1987): 61-62.
- [Chatterjee] **Efficient and Robust Large-Scale Rotation Averaging.** Avishek Chatterjee and Venu Madhav Govindu, ICCV 2013.
- [Kyle2014] **Robust Global Translations with 1DSfM.** Kyle Wilson and Noah Snavely, ECCV 2014.
- [Martinec] **Robust Multiview Reconstruction.** Daniel Martinec, 2008.
- [ACSfM] **Adaptive structure from motion with a contrario model estimation.** Pierre Moulon, Pascal Monasse, and Renaud Marlet. In ACCV, 2012.
- [GlobalACSfM] **Global Fusion of Relative Motions for Robust, Accurate and Scalable Structure from Motion.** Pierre Moulon, Pascal Monasse and Renaud Marlet. In ICCV, 2013.
- [TracksCVMP12] **Unordered feature tracking made fast and easy** Pierre Moulon and Pascal Monasse, CVMP 2012.

- [KVLD12] **Virtual Line Descriptor and Semi-Local Graph Matching Method for Reliable Feature Correspondence.** Zhe LIU and Renaud MARLET, BMVC 2012.
- [Moulon13] **Global Multiple-View Color Consistency.** Pierre Moulon, Bruno Dussutour and Pascal Monasse, CVMP 2013.
- [PMVS] **Accurate, dense, and robust multi-view stereopsis.** Yasutaka Furukawa and Jean Ponce. IEEE Trans. on Pattern Analysis and Machine Intelligence, 32(8):1362-1376, 2010.
- [CMPMVS] **Multi-View Reconstruction Preserving Weakly-Supported Surfaces.** M. Jancosek, T. Pajdla, CVPR 2011.
- [Wachter2014] **Let There Be Color! Large-Scale Texturing of 3D Reconstructions.** M. Wachter, N. Moehrle, and M. Goesele, ECCV 2014.
- [FLANN] **Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration.** Muja, Marius, and David G. Lowe. VISAPP (1). 2009.
- [Linfinity] **L infinity Minimization in Geometric Reconstruction Problems.** Richard I. Hartley, Frederik Schaffalitzky. CVPR 2004.
- [LinfinityGeneric] **Multiple-View Geometry under the L infinity Norm; Multiple View Geometry and the L infinity-norm.** Fredrik Kahl, Richard Hartley, 2008; and Fredrik Kahl, ICCV 2005.
- [Arnak] **Robust estimation for an inverse problem arising in multiview geometry.** Arnak S. Dalalyan, Renaud Keriven. In J. Math. Imaging Vision, 2012.
- [RANSAC] **Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography.** Fischler, Martin A., and Robert C. Bolles. Communications of the ACM 24.6 (1981): 381-395.
- [ACRANSAC] **Automatic homographic registration of a pair of images, with a contrario elimination of outliers.** Moisan, Lionel, Pierre Moulon, and Pascal Monasse. Image Processing On Line 10 (2012)
- [VLFeat] **VLFeat: An Open and Portable Library of Computer Vision Algorithms** A. Vedaldi and B. Fulkerson, 2008. <http://www.vlfeat.org/>, <http://www.vlfeat.org/overview/sift.html>.
- [OlssonDuality] **Outlier Removal Using Duality.** Carl Olsson, Anders Eriksson and Richard Hartley, Richard. CVPR 2010.
- [CASCADEHASHING] **Fast and Accurate Image Matching with Cascade Hashing for 3D Reconstruction** Jian Cheng, Cong Leng, Jiayang Wu, Hainan Cui, Hanqing Lu. CVPR 2014.
- [Ceres] **Ceres Solver** Sameer Agarwal and Keir Mierle and Others, <http://ceres-solver.org>
- [CppUnitLite] **CppUnitLite** <http://c2.com/cgi/wiki?CppUnitLite>
- [Eigen] **Eigen** http://eigen.tuxfamily.org/index.php?title=Main_Page
- [LEMON] **Lemon** <http://lemon.cs.elte.hu/trac/lemon>
- [STLplus] **STLplus** <http://stlplus.sourceforge.net/>
- [LP] **Linear Programming** http://en.wikipedia.org/wiki/Linear_programming