openMVG: Library reference

Pierre MOULON

February 8, 2013

Contents

Chapter 1

Introduction

OpenMVG (Multiple View Geometry) is a library for computer-vision scientists and especially targeted to the Multiple View Geometry community. It is designed to provide an easy access to the classical problem solvers in Multiple View Geometry and solve them accurately..

Why another library

The openMVG credo is: "Keep it simple, keep it maintainable". OpenMVG targets 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 that the code works as it should and enables more consistent repeatability. Furthermore, it makes it easier for the user to understand and learn the given features.

Acknowledgements

openMVG authors would like to thank libmv authors for providing an inspiring base to design the openMVG library. Authors also would like to thank Mikros Image and LIGM-Imagine laboratory for support and authorization to make this library as an open-source project.

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 license file contained in the source for complete license description.

Dependencies

OpenMVG come as a standalone distribution, you don't need to install libraries to make it compiles and run. Exception for the Linux library is made for the png, zlib and jpeg library in order to use the one of the system.

Chapter 2

The openMVG library

The openMVG library is cut in various modules:

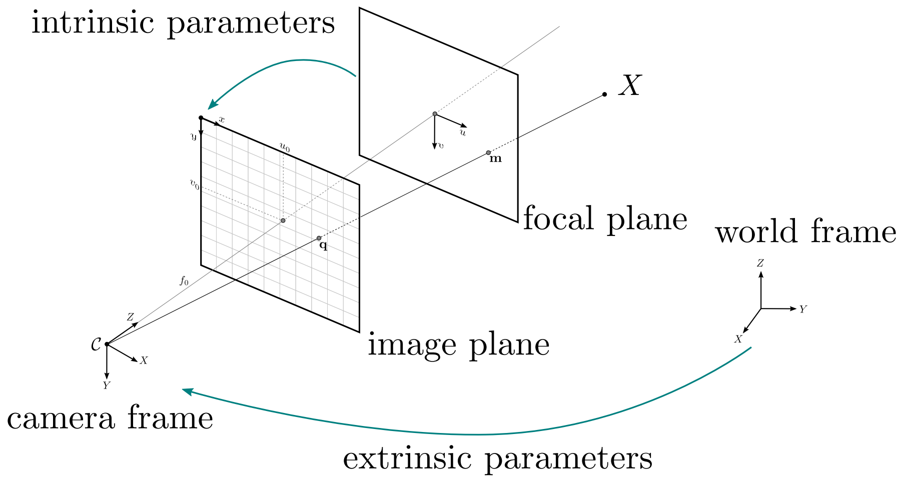
- image,
- multiview,
- robust_estimation,
- features,
- matching,
- numeric,
- tracks.

The main contribution of the openMVG library is the multiview and robust_estimation libraries that allow to handle easily Multiple View Geometry related task. The other libraries provide a common framework for all the samples that the user is free to use if desired.

Chapter 3

Pinhole camera geometry

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) and by a position in space.



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:

$$P_{3\times4} = \begin{bmatrix} f*k_u & & c_u \\ & f*k_v & c_v \\ & & 1 \end{bmatrix} \begin{bmatrix} & & & t_x \\ & R_{3\times3} & & t_y \\ & & & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.1}$$

- Intrinsic parameters $[f, c_u, c_v]$:

  k_u, k_v : scale factor relating pixels to distance (often equal to 1 or *height/width* sensor ratio),

  f : the focal distance (distance between focal and image plane),

  c_u, c_v : 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):

$$x_i = PX_i = K[R|t]X_i$$

$$\begin{bmatrix} u_i \\ v_i \\ w_i \end{bmatrix} = \begin{bmatrix} f * k_u & & c_u \\ & f * k_v & c_v \\ & & 1 \end{bmatrix} \begin{bmatrix} & & & t_x \\ & R_{3\times 3} & & t_y \\ & & & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \\ Z_i \\ W_i \end{bmatrix} \tag{3.2}$$

Chapter 4

openMVG 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 that can embed these solvers for robust estimation. It uses a concept called "kernel", explained later in this section.

4.1 Two-view geometric estimation:

openMVG provides solver for the following problems:

- affine,
- homographic,
- fundamental,
  - 7 to n pt,
  - 8 to n pt.
- essential,
  - 8 to n pt,
  - 5pt + intrinsic.

4.1.1 Homography matrix:

The homography matrix map the relation between two projection of a plane. It is a (3×3) matrix that links coordinates in left and right images with the following relation 4.1. Implementation follows the DLT (Direct Linear Transform) explain in [5] book.

$$x'_i = Hx_i \tag{4.1}$$

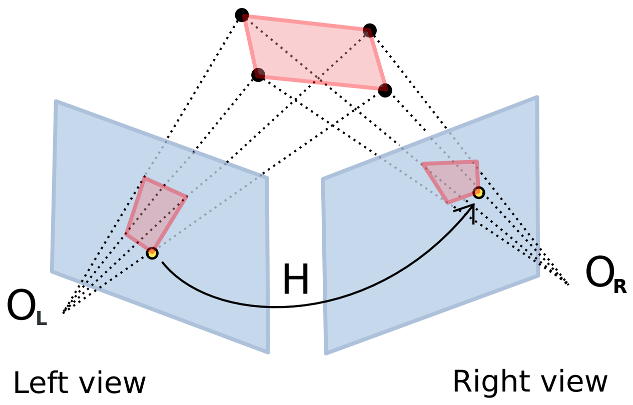


Figure 4.1: The homography matrix and the point to point constraint.

4.1.2 Fundamental matrix:

The fundamental matrix is a relation between two image viewing the same scene where points projection are visible in the two images. Given a point correspondence between two view: $x_i = (u_i, v_i, 1)^T$ and $x'_i = (u'_i, v'_i, 1)^T$, we obtain the following relation:

$$x_i'^T F x_i = 0 \tag{4.2}$$

F is the (3×3) Fundamental matrix, it put in relation a point x to a line where belong the projection of the 3D X point. $l'_i = Fx_i$ design 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.

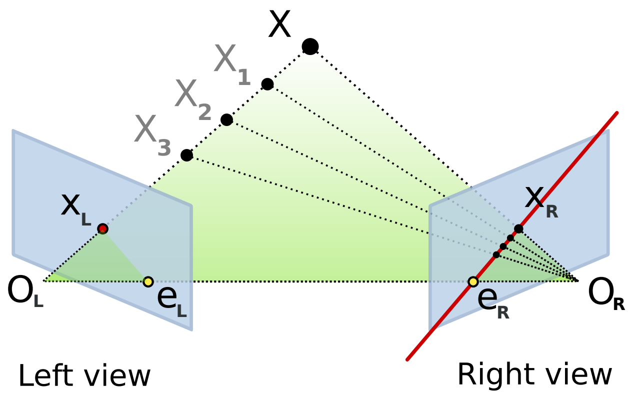


Figure 4.2: The fundamental matrix and the point to line constraint.

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 computes the F matrix. Implementation follows the DLT (Direct Linear Transform) explain in [5] book.

4.1.3 Essential matrix

Adding intrinsic parameter to the fundamental matrix gives a metric "object" that provide the following relation $E = K'^T FK$, this is the Essential relation explained by Longuet-Higgins in 1981 [8]. This essential matrix links the relative position of the camera to the fundamental matrix relation.

$$E = R[t]\times = K'^T FK \tag{4.3}$$

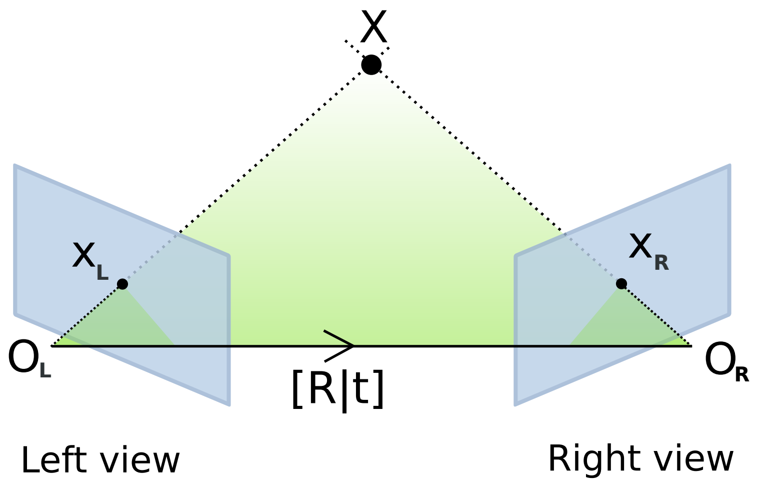


Figure 4.3: Essential matrix.

4.1.4 Kernel concept

A kernel is an association of data points, a model solver and an error estimator, i.e: 4.4.

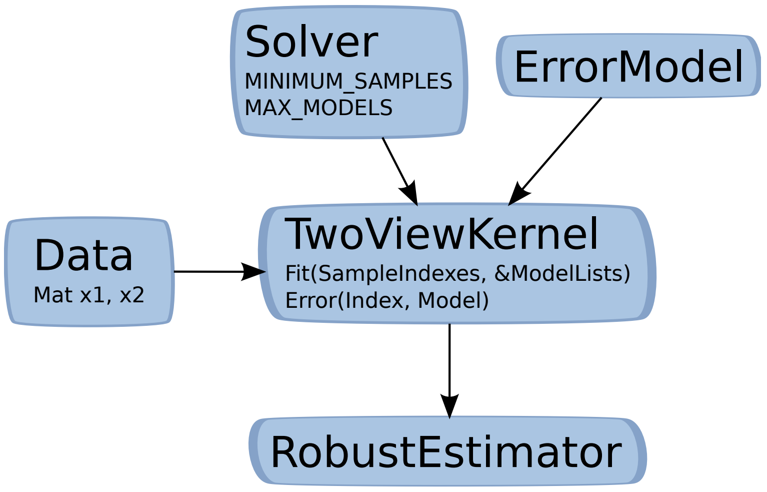


Figure 4.4: The Kernel concept (the two view case).

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 Error 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 openMVG robust estimator framework.

Chapter 5

3D-2D geometric constraint:

openMVG provides solver for the following problems:

- pose estimation/ camera resection,

  6pt [5],

  4pt with intrinsic EPnP [7],

  3pt with intrinsic P3P [6].

5.1 Camera resection/pose estimation:

Given a serie of 3D-2D image plane correspondences it's possible to compute a camera pose estimation (i.e. Fig. 5.1). It consists in estimating the camera parameter of the right camera that minimize the residual error of the 3D points re-projections (see Fig. 5.2), it's an optimization problem that trying to solve P parameter in order to minimize $min \sum_{i=1}^{n} x_i - P(X_i)$.

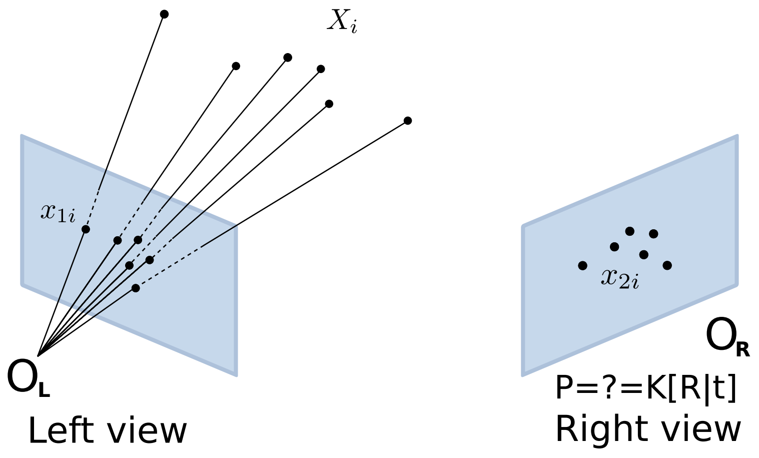


Figure 5.1: Resection/Pose estimation from 3D-2D correspondences.

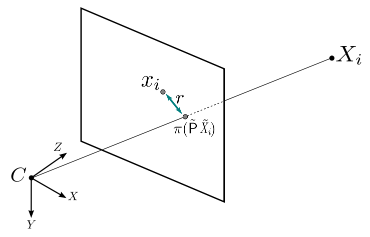
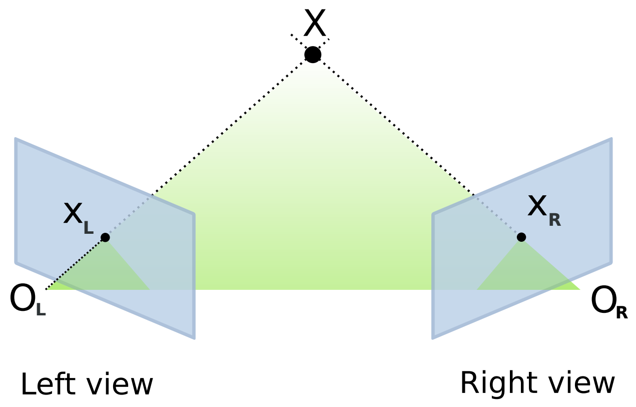


Figure 5.2: Residual error.

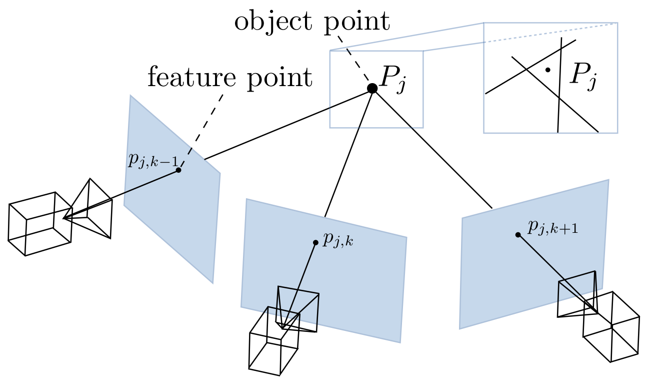
Chapter 6

Triangulation:

- 2 views,



- N-views



Chapter 7

openMVG 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.

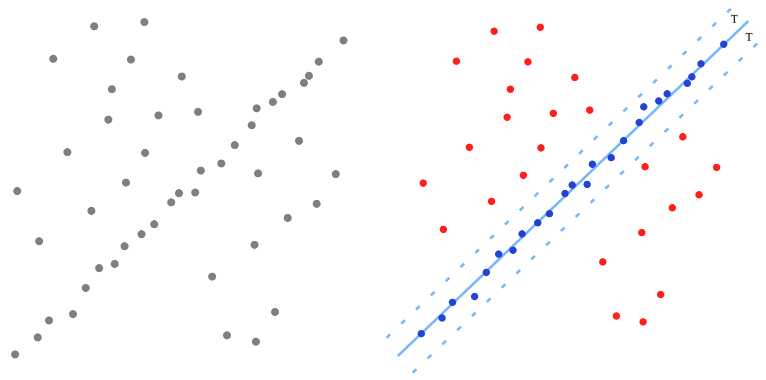


Figure 7.1: Robust estimation: Looking for a line in corrupted data.

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

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

7.1 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 fit 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 $M_{best}$
  for $i = 0 \to maxIter$ do
    Pick $N_{Sample}$ random samples
    Evaluate the model $M_i$ for the random samples
    Compute residuals for the estimated model
    if $Cardinal(residual < T) > previousInlierCount$ then
      $previousInlierCount = Cardinal(residual < T)$
      $M_{best} = M_i$
    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<size_t> vec_inliers;
Vec2 model = MaxConsensus(kernel, ScorerEvaluator<LineKernel>(0.3), &↩
   vec_inliers);
```

7.2 Ransac

Ransac [2] is an evolution of Max-Consensus with a-priori information about the noise and corrupted data amount of the data. Those informations allow to reduce the number of iteration 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<size_t> vec_inliers;
Vec2 model = Ransac(kernel, ScorerEvaluator<LineKernel>(0.3), &vec_inliers);
```

7.3 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 [10] 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 find a model and it's associated noise. If there is too much noise, the *a contrario* methods return that no model was found.

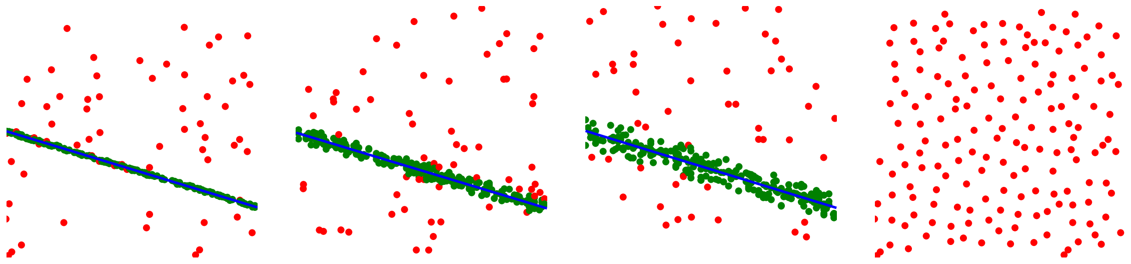


Figure 7.2: *a contrario* robust estimation, noise adaptivity.

Here an example of how find a best fit line, by using the *a contrario* robust estimation framework: It's 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 a contrario adapted base model estimator and associated error metric
const size_t image_width = 12;
ACRANSACOneViewKernel<LineSolver, pointToLineError, Vec2> lineKernel(xy, ↩
   image_width, image_width);

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

Chapter 8

openMVG image:

This module provides generic algorithms for image related tasks:

Container A generic image container:

Template pixel type: gray, RGB, RGBA,

User specified bit depth, uchar, float, double.

Image I/O Loading, writing:

ppm/pgm, jpeg, png

Drawing operations

lines,

circles,

ellipses.

Chapter 9

openMVG matching:

A generic interface to perform K-Nearest Neighbor search:

- Brute force,
- Approximate Nearest Neighbor (FLANN [11]).

This module works for any size of data, it could be use to match 128, 64, vector long descriptors or to find closest 3D points. The used metric is customizable and enable matching under L2 or a user customized distance (L1, ...). The Nearest Neighbor retrieval allows to retrieve the N nearest points and so to perform easily the "Nearest Neighbor Distance Ratio" distance check to remove repetitive elements.

Chapter 10

openMVG 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.

The "track" computation problem. Considering n pairwise feature correspondences as input we want sets of corresponding matching features across multiple images, as illustrated in Figures 10.1, 10.2 with video frames.

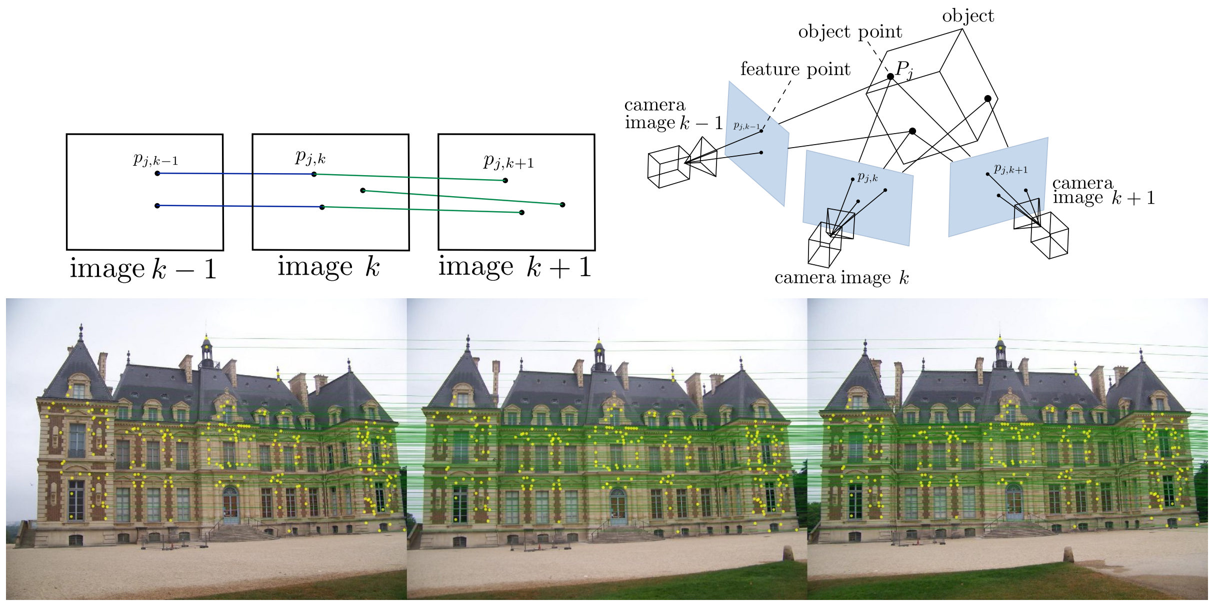


Figure 10.1: From features to tracks

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" [9].

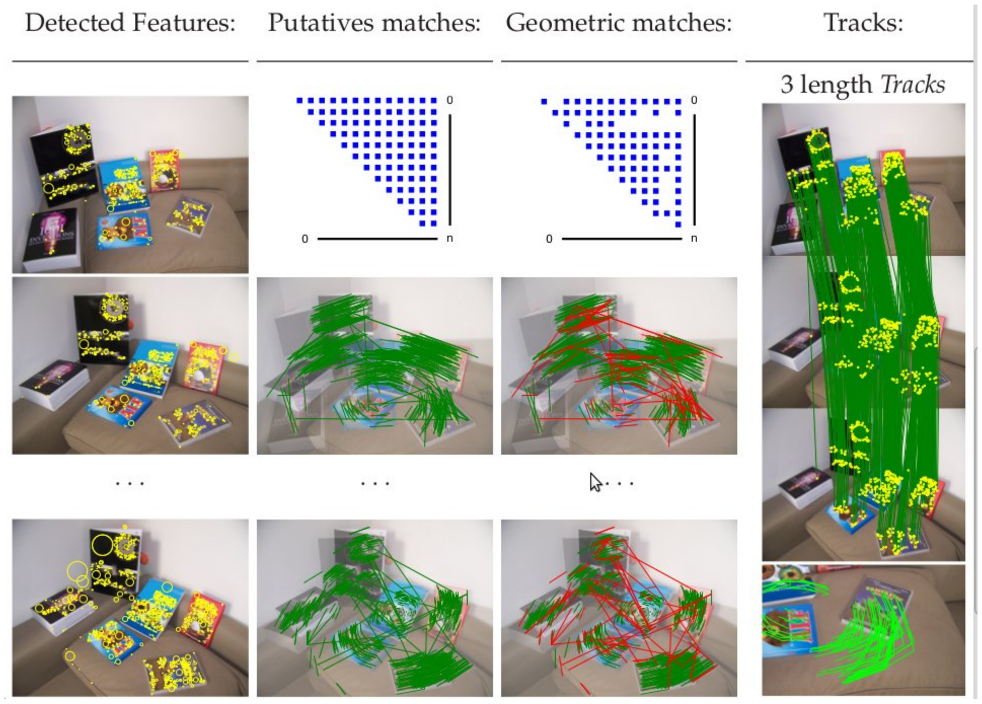


Figure 10.2: Feature based tracking.

Some comments about the data structure:

```
// Data structure to store a track: collection of {ImageId,FeatureId}
//   The corresponding image points with their imageId and FeatureId.
typedef std::map<size_t,size_t> submapTrack;
// A track is a collection of {trackId, submapTrack}
typedef std::map< size_t, submapTrack > STLMAPTracks;

STLMAPTracks map_tracks;

// 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 size_t trackId = iterT->first;
 const tracks::submapTrack & track = iterT->second;
 for( tracks::submapTrack::const_iterator iterTrack = track.begin();
   iterTrack != track.end(); ++iterTrack)
 {
   size_t imageId = iterTrack->first;
   size_t featId = iterTrack->second;

   // Get the feature point
 }
}
```

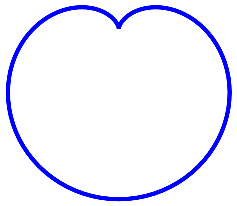
Chapter 11

Vector drawing:

openMVG considers that visualizing data is important. OpenMVG provides a class that help to perform vector graphics (SVG) drawing in order to have the best possible visualization of his algorithm output. Vector graphics allows keeping details when you zoom what is not done when you use raster graphics. (SVG files are supported by web navigator and the Inkscape software).

```
// Draw a cardiod curve with the svg polyline
// http://en.wikipedia.org/wiki/Cardioid
  { // Pre-compute (x,y) curve points
    size_t nbPoints = 120;
    std::vector<float> vec_x(nbPoints, 0.f), vec_y(nbPoints, 0.f);
    double S = 20.;
    for (size_t i = 0; i < nbPoints; ++i) {
      const double theta = i * 2 * M_PI / nbPoints;
      //-- Cardioid equation
      vec_x[i] = (3*S + S*(2.*sin(theta)-(sin(2.*theta))));
      vec_y[i] = (2*S - S*(2.*cos(theta)-(cos(2.*theta))));
    }
    // Create a svg surface and add the cardiod polyline
    svgDrawer svgSurface(6*S, 6*S); //Create a svg object
    svgSurface.drawPolyline(
      vec_x.begin(), vec_x.end(),
      vec_y.begin(), vec_y.end(),
      svgStyle().stroke("blue", 2));

    //Export the SVG stream to a file
    std::string sFileName = "ThirdExample.svg";
    std::ofstream svgFile(sFileName.c_str());
    svgFile << svgSurface.closeSvgFile().str();
    svgFile.close();
  }
```



Here the result exported vector graphic:

Chapter 12

3D point cloud handler:

- PLY point cloud export (PLY is known as the Polygon File Format or the Stanford Triangle Format).

Different interfaces have been written in order to export:

- Point cloud,
- Coloured point cloud,
- Camera location.

Chapter 13

Structure from Motion:

Structure from Motion (Fig. 13.2) computes an external camera pose per image (the motion) and a 3D point cloud (the structure) representing the pictured scene (Fig. 13.1). Inputs are images and internal camera calibration information (intrinsic parameters). Feature points are detected in each image (e.g., SIFT) and matched between image pairs. There are two main approaches to correlate detected features and solve the SfM problem: the *incremental* pipeline and the *global* method.

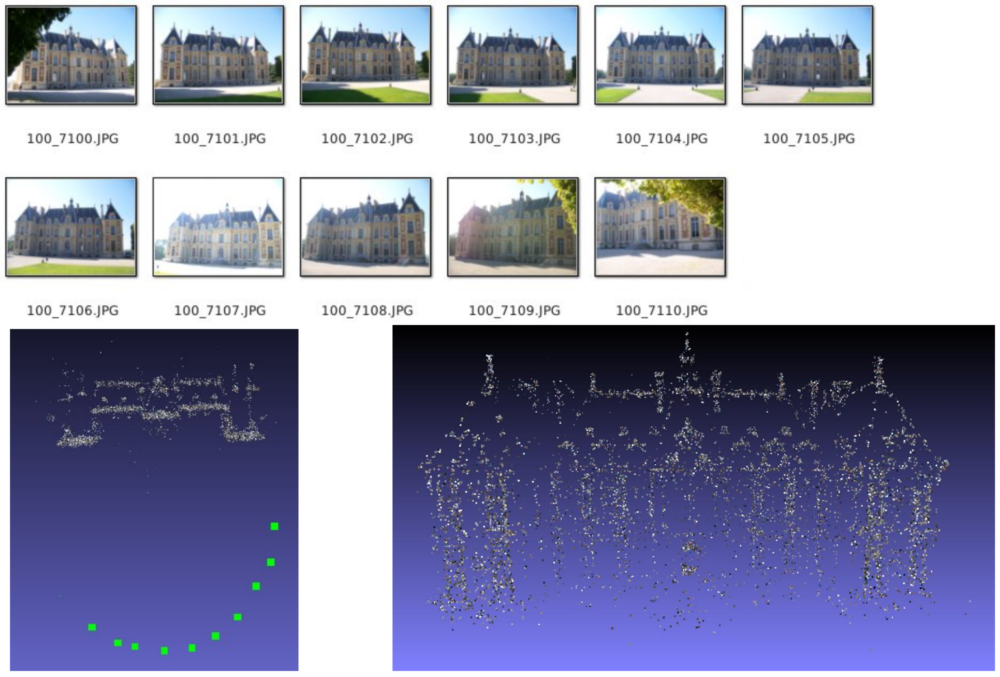


Figure 13.1: Input images, estimated camera location and structure.

openMVG proposes a customizable implementation of an Incremental Structure from Motion chain, it is the implementation use for the paper "Adaptive Structure from Motion with *a contrario* model estimation" [10] published at ACCV 2012.

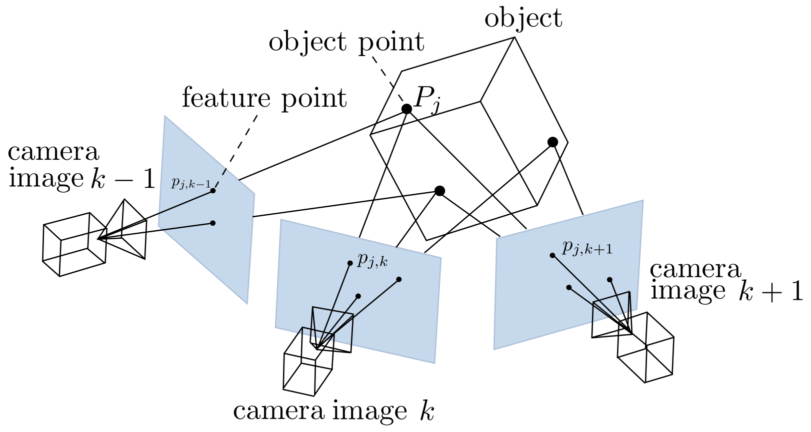


Figure 13.2: Structure from Motion.

The incremental pipeline is a growing reconstruction process (i.e algorithm 3). 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). In this implementation we use the ceres-solver, a generic non-linear least squares problems solver [1], in order to perform Bundle Adjustment steps.

The general feature correspondence and SfM processes are described in algorithms 2 and 3. The first algorithm outputs pairwise correspondences that are consistent with the estimated fundamental matrix. The initial pair must be chosen with numerous correspondences while keeping a wide enough baseline. The second algorithm takes these correspondences as input and yields a 3D point cloud as well as the camera poses. Steps marked with a star ($*$) are estimated within the *a contrario* framework. This allows critical thresholds to be automatically adapted to the input images (and remove the choice of an empiric T threshold value).

Algorithm 2 Computation of geometry-consistent pairwise correspondences

```
Require: image set
Ensure: pairwise point correspondences that are geometrically consistent
  Compute putative matches:
    detect features in each image and build their descriptor
    match descriptors (brute force or approximate nearest neighbor)
  Filter geometric-consistent matches:
*   estimate fundamental matrix $F$
```

Algorithm 3 Incremental Structure from Motion

```
Require: internal camera calibration (matrix $K$, 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 $t \cap e$, which provides an initial reconstruction
  contract edge $e$
  while $G$ contains an edge do
    pick edge $e$ in $G$ that maximizes track($e$) $\cap$ {3D points}
*   robustly estimate pose (external orientation/resection)
    triangulate new tracks
    contract edge $e$
    perform bundle adjustment
  end while
```

Once camera position and orientation have been computed, Multiple View Stereo-vision algorithms to compute a dense scene representation could be used. OpenMVG exports a PMVS [3, 4] ready to use data. Figure 13.3 shows an example on the scene of the figure 13.1.

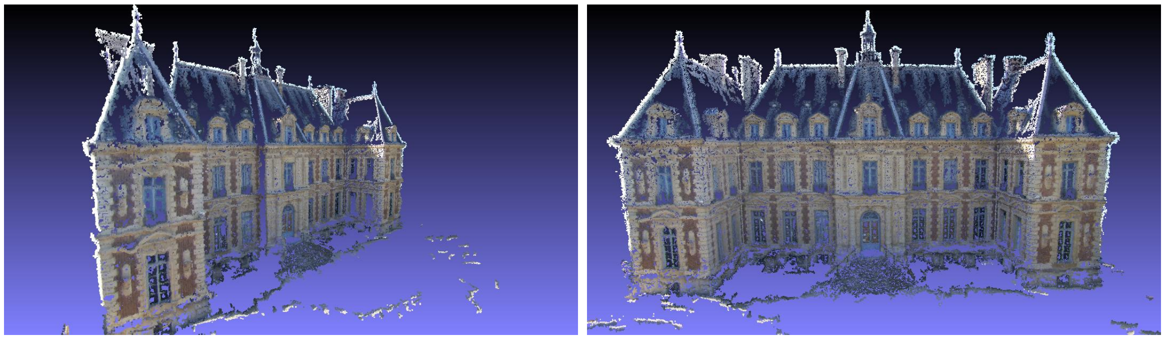


Figure 13.3: Multiple View Stereo-vision densification on the estimated scene using PMVS [3, 4].

Tools to draw detected Keypoints, putative matches, geometric matches and tracks are provided along the SfM binary directory.

13.1 Structure from Motion chain usage

The *a contrario* Structure from Motion chain take as input a sequence of **distortion corrected images** and the intrinsic associated calibration matrix K. This calibration matrix is stored as a "K.txt" file as a raw ascii 3×3 matrix in the same directory as the pictures.
Using a 3 directories based data organisation structure is suggested:

images

- your image sequence,
- K.txt

matches (the points and matches information will be saved here)

outReconstruction (directory where result and log of the 3D reconstruction will be exported)

Point matching:

The first step consists in computing relative image matches (i.e algorithm 2): You have to use the *openMVG_main_computeMatches* software in the software/SfM openMVG module.

```
$ openMVG_main_computeMatches −i /home/pierre/Pictures/Dataset/images −e *.↩
    JPG −o /home/pierre/Pictures/Dataset/matches
```

Arguments are the following:

-i|–imadir the path where image are stored.

-e|–ext image extension i.e "*.jpg" or "*.png". Case sensitive.

-o|–outdir path where features, descriptors, putative and geometric matches will be exported.

-r|–distratio optional argument (Nearest Neighbor distance ratio, default value is set to 0.6).

-s|–octminus1 optional argument (Use the octave -1 option of SIFT or not, default value is set to false: 0).

Once matches have been computed you can, at your choice, display detected points, matches or start the 3D reconstruction.

Point, matching visualization:

Three softwares are available to display:

Detected keypoints *openMVG_main_exportKeypoints*

Putative, Geometric matches *openMVG_main_exportMatches*

Tracks *openMVG_main_exportTracks*

SfM, 3D structure and camera calibration:

The main binary in order to run the SfM process is *openMVG_ main_ IncrementalSfM*, it use previous computed data and is implemented as explained in algorithm 3.

```
$ openMVG_main_IncrementalSfM −i /home/pierre/Pictures/Dataset/images/ −m /↩
    home/pierre/Pictures/Dataset/matches/ −o /home/pierre/Pictures/Dataset/↩
    outReconstruction/
```

openMVG_ main_ IncrementalSfM displays to you some initial pairs that share an important number of common point. Please select two image index and the 3D reconstruction will start.

Bibliography

[1] Sameer Agarwal and Keir Mierle. *Ceres Solver: Tutorial & Reference*. Google Inc.

[2] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM (CACM)*, 24(6):381–395, 1981.

[3] Yasutaka Furukawa, Brian Curless, Steven M. Seitz, and Richard Szeliski. Towards internet-scale multi-view stereo. In *CVPR*, 2010.

[4] Yasutaka Furukawa and Jean Ponce. Accurate, dense, and robust multi-view stereopsis. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 32(8):1362–1376, 2010.

[5] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[6] Laurent Kneip, Davide Scaramuzza, and Roland Siegwart. A novel parametrization of the perspective-three-point problem for a direct computation of absolute camera position and orientation. In *CVPR*, pages 2969–2976, 2011.

[7] Vincent Lepetit, Francesc Moreno-Noguer, and Pascal Fua. EPnP: an accurate $O(n)$ solution to the PnP problem. *International Journal of Computer Vision (IJCV)*, 81(2):155–166, 2009.

[8] Longuet. A computer algorithm for reconstructing a scene from two projections. *Nature*, 293:133–135, September 1981.

[9] Pierre Moulon and Pascal Monasse. Unordered feature tracking made fast and easy. In *CVMP*, 2012.

[10] Pierre Moulon, Pascal Monasse, and Renaud Marlet. Adaptive structure from motion with a contrario model estimation. In *ACCV*, 2012.

[11] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Application VISSAPP'09)*, pages 331–340. INSTICC Press, 2009.