POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

# Autocalibration of monocular cameras for autonomous driving scenarios


**Supervisor**
prof. Massimo Violante

**Candidate**
Luigi Ferrettino

**Internship Tutor**
Luxoft (Objective Software Italia SRL)
ing. Stefano Moccia


December 2020

*To those who I love,*
*they helped me during*
*this journey*

# Abstract

Object identification, detection and distance calculation are crucial topics for the autonomous driving world. Different technologies are used in the automotive field to reach the expected results. LIDAR and radar technologies are widely used for distance measurement, but they are quite expensive and rather not able to cover all the use cases (e.g. road lanes or traffic signs identification). For this reason, those sensors are often used in conjunction with algorithms where also camera data is integrated. A possible alternative to LIDAR and radars is the usage of stereo cameras. The stereo camera is a sensing technology using two cameras to capture images. Since stereo cameras acquire images with multiple cameras, they estimate the distance between the camera itself and the object surface by exploiting points triangulation, which is not possible with a monocular camera system. In this way, stereo vision has a broad range of applications. This work aims at proposing a method to self-calibrate four monocular cameras in order to retrieve with an acceptable approximation the positions of the cameras in the real-world coordinates. This will be an important step in order to exploit future achievements with the growing availability of stereo cameras.

# Acknowledgements

A special thanks goes to Dr. Marco Bottero, responsible for the organization of the Italian Branch of the Luxoft Group in terms of business development, sales, human resources, operational structure and results. He helped me during my stay, even in the particular circumstances generated from the smart working situation. Furthermore, a general thanks goes to Dr. Mattia Raffero and all the members of the MINERVA team group during my extracurricular stage, which were always available to help me when needed, even in an indirectly manner.

# Contents

**List of Tables** 8

**List of Figures** 9

**1 Introduction** 11

**2 Background** 13
- 2.1 Pinhole camera model . . . 13
  - 2.1.1 Camera matrices . . . 14
  - 2.1.2 Camera calibration . . . 16
- 2.2 Perspective-n-Point . . . 17
  - 2.2.1 RANSAC . . . 18
- 2.3 Epipolar geometry . . . 20
  - 2.3.1 Epipolar constraint . . . 21
  - 2.3.2 Fundamental matrix . . . 23
- 2.4 Multi-view geometry . . . 24
  - 2.4.1 Structure from motion (SfM) . . . 26

**3 Research** 29
- 3.1 Calibration methods . . . 29
  - 3.1.1 Tsai's calibration technique . . . 30
- 3.2 Object detection . . . 31
  - 3.2.1 Fast/Faster R-CNN . . . 31
  - 3.2.2 YOLO . . . 32
  - 3.2.3 3D object detection . . . 33
- 3.3 Features extraction and matching . . . 35
  - 3.3.1 Extraction . . . 35
  - 3.3.2 Matching . . . 39
- 3.4 Perspective-n-Point . . . 40
  - 3.4.1 Gao's P3P solver . . . 41
  - 3.4.2 Lambda twist . . . 41
  - 3.4.3 PnP-Net . . . 41

3.5 Structure from motion . . . . . 42

**4 Design and development** 44
4.1 Introduction . . . . . 44
4.2 Design . . . . . 44
4.2.1 Custom pipeline . . . . . 44
4.2.2 OpenCV's pipeline . . . . . 47
4.3 Development . . . . . 49
4.3.1 Frameworks and requirements . . . . . 49
4.3.2 Intrinsic calibration . . . . . 50
4.3.3 Custom pipeline . . . . . 50
4.3.4 OpenCV's pipeline . . . . . 56

**5 Results** 61
5.1 Introduction . . . . . 61
5.1.1 Initial calibration . . . . . 61
5.1.2 Datasets . . . . . 64
5.2 Custom pipeline . . . . . 64
5.3 OpenCV's pipeline . . . . . 69

**6 Conclusions** 73
6.1 Future works . . . . . 73

**A Build and install OpenCV** 75
A.1 Dependencies . . . . . 75
A.1.1 Ceres solver . . . . . 75
A.1.2 CUDA . . . . . 76
A.1.3 VTK . . . . . 76
A.2 Build and install . . . . . 76

# List of Tables

2.1 Conditions needed for metric upgrade based on the number of views. 27
5.1 Performance comparison from $n$ number of views between different methods for features extraction and matching . . . . . 66
5.2 Performance comparison from $n$ number of views between different methods for features extraction and matching on raw data . . . . . 68

# List of Figures


1.1 Use case scenario . . . 11
2.1 Pinhole camera model. Source: Fei-Fei Li - Stanford Vision Lab (2011) 13
2.2 From retina plane to pixels plane . . . 15
2.3 Projection of a point from the world reference system. Source: Fei-Fei Li - Stanford Vision Lab (2011) . . . 16
2.4 Camera calibration scenario. Source: Fei-Fei Li - Stanford Vision Lab (2011) . . . 17
2.5 PnP problem scenario. Source: OpenCV Docs . . . 18
2.6 Triangulation scenario. Source: Fei-Fei Li - Stanford Vision Lab (2011) 20
2.7 Epipolar geometry. Source: Wikipedia.org . . . 21
2.8 Stereo vision dimensions. Source: Fei-Fei Li - Stanford Vision Lab (2011) . . . 22
2.9 Disparity calculation from two views. . . . 25
2.10 The structure for motion problem. Source: Fei-Fei Li - Stanford Stereo Lab (2011) . . . 26
3.1 Faster R-CNN architecture, Source: Girshick et al. ICCV2015 . . . 32
3.2 YOLO steps. Source: Redmon et al. CVPR2016 . . . 33
3.3 Stereo R-CNN architecture. Source: Li et al. . . . 34
3.4 Frustum PointNets architecture. Source: Qi et al. . . . 35
3.5 DoF for each octaves in Gaussian Pyramid. Source: OpenCV Docs 36
3.6 SIFT local extrema. Source: OpenCV Docs . . . 37
3.7 SuperGlue architecture. Source: Sarlin et al. 2020 . . . 40
3.8 PnP-Net architecture. Source: Sheffer et al. ECCV2020 . . . 42
3.9 Representation of the sliding-window with bundle adjustment method 43
4.1 Custom pipeline's flow . . . 46
4.2 OpenCV pipeline's flow . . . 48
4.3 mAP and average loss at 1000/6000 epochs . . . 52
5.1 Chess board calibration process (first) . . . 62
5.2 Chess board calibration process (second) . . . 63
5.3 Samples from the Multi-View Car Dataset . . . 64
5.4 Raw sample from the Android camera . . . 65
5.5 *panda_2.jpg*, *panda_4.jpg* and *panda_6.jpg* . . . 66

5.6 *panda_8.jpg* and *panda_10.jpg* . . . . . 66
5.7 Common filtered matches on *panda_2.jpg* with *panda_4.jpg* and *panda_6.jpg* by exploiting SuperPoint+SuperGlue . . . . . 67
5.8 Pose estimation on *panda_2* with $n = 3$ . . . . . 67
5.9 Common filtered matches on on the first camera from the $n = 3$ raw images . . . . . 68
5.10 Pose estimation on the first and second view on raw data . . . . . 69
5.11 Cameras pose estimation with $n = 8$ on the Multi-View Car Dataset 69
5.12 Cameras pose estimation with $n = 3$ on raw data taken from the Android camera . . . . . 70
5.13 Cameras pose estimation with $n = 3$ on the Multi-View Car Dataset 71
5.14 Images used for a $n = 8$ views reconstruction . . . . . 72

# Chapter 1

# Introduction

The research topic aims at developing a solution to self calibrate four monocular cameras in an outdoor area where in future autonomous cars will be driven. By following the generic steps:

1. **Automatically identify from the scene some fixed objects**
2. **Calculate the relative positions of the cameras**

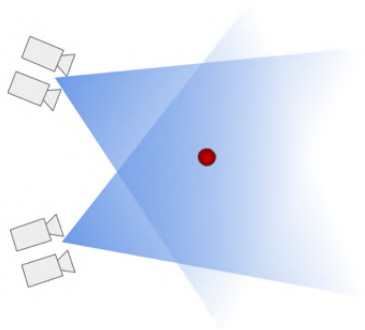


Figure 1.1. Use case scenario

The method will output the final positions of the four cameras. To achieve the abovementioned steps, a research activity on state of the art computer vision algorithms needs to be conducted with a specific focus on techniques to identify from the scene the pose of fixed reference points and use those points to calculate the positions of the four cameras. Expected result would be to develop a fully-automatic calibration program, suitable for obtaining extrinsics parameters (relative transform

matrices) between multiple monocular cameras. Extrinsics parameters are required for fusing the 3D data of multiple sensors, since the data must be processed in the same geometrical frame. This is often needed in applications where, for instance, the cameras are used for an highway, a parking area or a production facility. The program will employ multiple computer vision algorithms, for detecting feature in the images, match them and evaluate their alignment. The use case scenario (Fig. 1.1) is the ending point of the script, with all the four camera positions retrieved.

In order to better understand and then solve the problem in its entirety, the *divide et impera* approach is well suited. The functional decomposition in sub-problems would be:

- Identify from the scene a fixed object.
- Choose the reference points.
- Retrieve from the reference points the extrinsic parameters.
- Output the camera positions.

# Chapter 2

# Background

In order to explore and better understand the proposed methods in the literature, some basic knowledge of computer vision is fundamental, starting from the camera model and its subsequent implications.

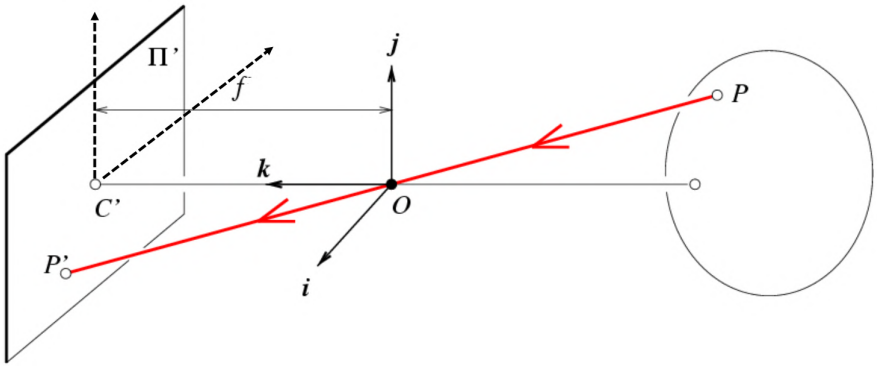



Figure 2.1. Pinhole camera model. Source: Fei-Fei Li - Stanford Vision Lab (2011)

## 2.1 Pinhole camera model

It is the simplest, yet one of the most effective, camera models. It can be easily built by having a little aperture (the *pinhole)* in a box so that the light of the scene passes through it and project an inverted scene in the opposite side of the box. From a geometrical point of view (Fig. 2.1), the model introduces a non-linear transformation between the point P in world-coordinates and the point P' on the camera film:

$$P = \begin{bmatrix} x \\ y \\ z \end{bmatrix} -> P' = \begin{bmatrix} x' \\ y' \end{bmatrix} \text{ from a simple geometrical derivation } \begin{cases} x' = f' \frac{x}{z} \\ y' = f' \frac{y}{z} \end{cases}$$

Since that the aforementioned transformation is not linear (because of the division), the homogeneous coordinates are used. In particular, 2.1 are used to transform *to* homogeneous coordinates, while 2.2 are used to transform *from* homogeneous coordinates.

$$(x, y) => \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, (x, y, z) => \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.1}$$

$$\begin{bmatrix} x \\ y \\ \omega \end{bmatrix} => (x/\omega, y/\omega), \begin{bmatrix} x \\ y \\ z \\ \omega \end{bmatrix} => (x/\omega, y/\omega, z/\omega) \tag{2.2}$$

The homogeneous coordinates have the advantage that the coordinates of points (even points at infinity) can be represented using finite coordinates. They allow affine transformations and, in general, projective transformations to be easily represented by a matrix. The *perspective projection transformation* is used in the homogeneous coordinates to retrieve the image point starting from the scene point by a matrix multiplication, as showed in 2.3.

$$P' = \begin{bmatrix} fx \\ fy \\ z \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ f & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}, \quad P' = MP \tag{2.3}$$

### 2.1.1 Camera matrices

The next step is to obtain the image as pixels from the projected image film by a using bottom-left coordinate system (Fig. 2.2). This transformation must take into account two important parameters:

- Off-set addition. The center C' of the film could not be aligned with the center $C = [c_x, c_y]$ of the pixel image. The geometrical derivation must then take this into account by adding the two C' coordinates:

  $(x, y, z) -> (f\frac{x}{z} + c_x, f\frac{y}{z} + c_y)$

- Metric to pixel conversion. Since that each pixel could capture an area and could not be necessarily be non-square, two scale parameters are added: $k$ and

$l$. They depends on the physical properties of the captured pixel and they are implicitly used with $\alpha = fk$ and $\beta = fl$ (focal length in terms of pixels):

$(x, y, z) -> (\alpha\frac{x}{z} + c_x, \beta\frac{y}{z} + c_y)$

- Skew parameter. This parameter is used to represent how the angle between the x and y axes is distorted in the pixel representation. It is a simple parameter $s$ that multiplies y in the x coordinate transformation:

  $(x, y, z) -> (\alpha\frac{x}{z} + c_x + s\frac{y}{z}, \beta\frac{y}{z} + c_y)$

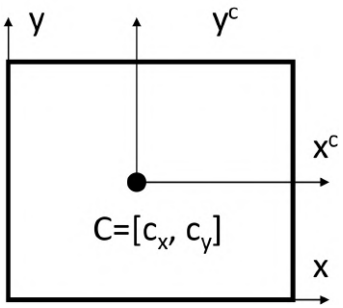



Figure 2.2. From retina plane to pixels plane

The M matrix form of the discovered translation is extrapolated in 2.4.

$$P' = \begin{bmatrix} \alpha\ x + sy + c_x z \\ \beta\ y + c_y z \\ z \end{bmatrix} = \begin{bmatrix} \alpha & s & c_x & 0 \\ 0 & \beta & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.4}$$

From the transformation 2.4, the *intrinsic camera matrix* is retrieved as:

$$P' = \begin{bmatrix} \alpha & s & c_x & 0 \\ 0 & \beta & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = K[I\ \ 0]P => K = \begin{bmatrix} \alpha & s & c_x \\ 0 & \beta & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.5}$$

The intrinsic matrix has 5DoF and it is essential in order to project a pixel point into the image film plane point. This mapping is defined within the camera reference system: what if a point is represented in world reference system? A possible scenario is shown in Fig. 2.3. In particular, in 4D homogeneous mapping, some external parameters are needed: the translation matrix T and the rotation matrix R. In order to retrieve the P' pixel point now, two transformations are needed:

- Translation and rotation from world coordinates to camera local coordinates (2.6).
- Projection from local camera coordinates to pixel coordinates (2.7).

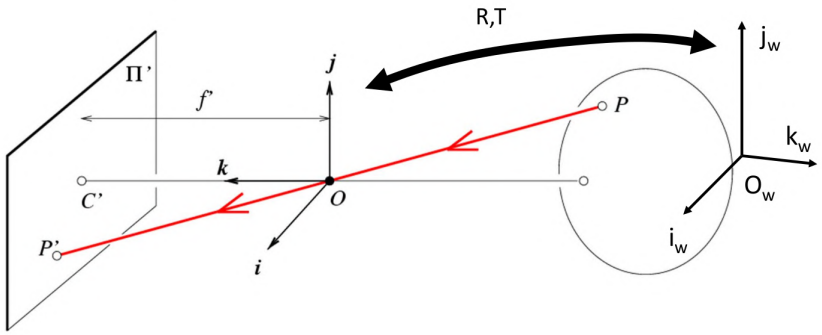



Figure 2.3. Projection of a point from the world reference system. Source: Fei-Fei Li - Stanford Vision Lab (2011)

$$P = [R\ T]P_w \tag{2.6}$$

$$P' = K[I\ 0]P \tag{2.7}$$

By using 2.6 into 2.7, the direct transformation will be:

$$P' = K[R\ T]P \tag{2.8}$$

Where the matrix $[R\ T]$ represents the *extrinsic camera matrix* and the matrix $M = K[R\ T]$ defines *projective cameras*.

### 2.1.2 Camera calibration

The goal of camera calibration is to estimate intrinsic and extrinsic parameters from one or (better) multiple images. As shown in Fig. 2.4, starting from n correspondences:

- $P_1 ... P_n$ of known position in $[O_w, i_w, j_w, k_w]$.
- $p_1 ... p_n$ of known position in the image system.

The two matrices can be retrieved. Since the M matrix has 11 unknown, 11 equations and about 6 points are required in order to retrieve a result.

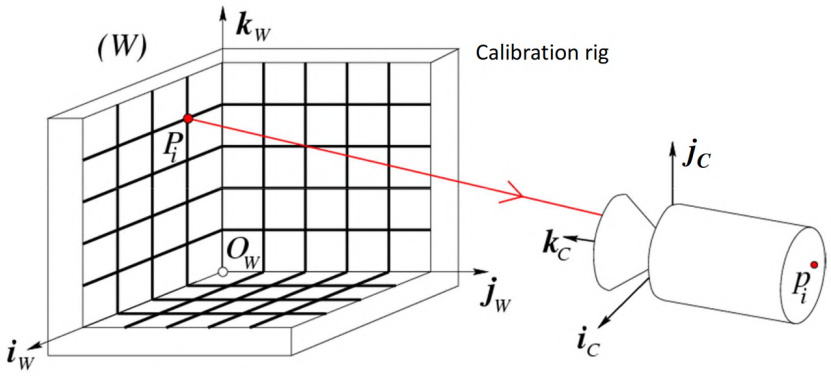



Figure 2.4. Camera calibration scenario. Source: Fei-Fei Li - Stanford Vision Lab (2011)

There are many methods to estimate M, the most immediate one is a simple *Singular Value Decomposition* (SVD). Knowing that the general problem is $p_i -> MP_i$:

$$p_i -> \begin{bmatrix} u_i \\ v_i \end{bmatrix} = \begin{bmatrix} \frac{\boldsymbol{m_1} \cdot P_i}{\boldsymbol{m_3} \cdot P_i} \\ \frac{\boldsymbol{m_2} \cdot P_i}{\boldsymbol{m_3} \cdot P_i} \end{bmatrix}, M = \begin{bmatrix} \boldsymbol{m_1} \\ \boldsymbol{m_2} \\ \boldsymbol{m_3} \end{bmatrix} \tag{2.9}$$

By dividing the coordinates, 2.9 would be rewritten as:

$$u_i = \frac{\boldsymbol{m_1} P_i}{\boldsymbol{m_3} P_i} -> u_i(\boldsymbol{m_3} P_i) = \boldsymbol{m_1} P_i -> u_i(\boldsymbol{m_3} P_i) - \boldsymbol{m_1} P_i = 0 \tag{2.10}$$

$$v_i = \frac{\boldsymbol{m_2} P_i}{\boldsymbol{m_3} P_i} -> u_i(\boldsymbol{m_3} P_i) = \boldsymbol{m_2} P_i -> v_i(\boldsymbol{m_3} P_i) - \boldsymbol{m_2} P_i = 0 \tag{2.11}$$

And finally, the equation system can be written as $\boldsymbol{P}m = 0$, where $\boldsymbol{P}$ is known and m is not. The SVD has to be performed on $\boldsymbol{P} -> U_{2nx12} D_{12x12} V^T_{12x12}$ in order to retrieve a solution.

## 2.2 Perspective-n-Point

It is the problem of estimating the pose of a calibrated camera given a set of 3D points in world coordinates and the correspondent projections in 2D points of the image (Fig. 2.5).

The pose of the camera has 6DoF and it is defined by the rotation (roll, pitch, yaw) and the translation of the camera with respect to the world. The $n$ parameter represents the number of correspondences used, with the particular case of $n$=3 that is called P3P, representing the minimal form. An important aspect to underline is

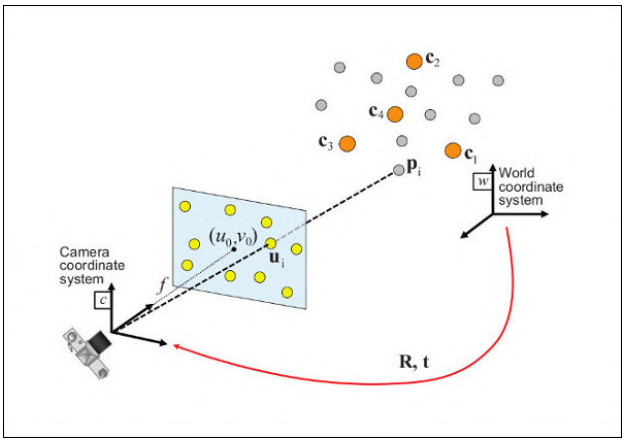



Figure 2.5. PnP problem scenario. Source: OpenCV Docs

that the estimation is up to a scalar multiplier, since the subject dimensions may be under-scaled or up-scaled during the caption.

The *PnP* problem could be seen as a simplification to the general camera calibration problem, by assuming known calibration parameters (i.e. the K matrix), since they depends only on the camera physical properties. During the years, many methods have been developed for finding a reliable and rapid solution, but an important common aspect is the use of the *RANdom SAmple Consensus* (RANSAC) as an outlier detection method [1].

### 2.2.1 RANSAC

This algorithm is categorized as "non-deterministic", since it does produce a reasonable result only with a certain probability, that increases as iterations increase. It works by random sampling the observed data exploiting a voting scheme to find outliers in a dataset with both outliers and inliers. By iterating, the bigger consensus set is saved, giving as stopping conditions only the number of iterations (defined as input). In the listing below (2.1) is possible to distinguish two main actions:

1. In the first step, a sample subset containing minimal data items is randomly selected from the input dataset. A fitting model and the corresponding model parameters are computed using only the elements of this sample subset. The cardinality of the sample subset is the smallest sufficient to determine the model parameters.

2. In the second step, the algorithm checks which elements of the entire dataset are consistent with the model instantiated by the estimated model parameters obtained from the first step. A data element will be considered as an outlier if it does not fit the fitting model instantiated by the set of estimated model

parameters within some error threshold that defines the maximum deviation attributable to the effect of noise.

The use of RANSAC in many computer vision problems is very useful in order to remove the outliers (i.e. wrong point correspondences 3D-2D) increasing robustness and precision. In particular, the Perspective-n-Point algorithm would be too much sensitive to outliers without it.

```
Given:
    data − A set of observations.
    model − A model to explain observed data points.
    n − Minimum number of data points required to estimate model parameters.
    k − Maximum number of iterations allowed in the algorithm.
    t − Threshold value to determine data points that are fit well by model.
    d − Number of close data points required to assert that a model fits well to data.

Return:
    bestFit − model parameters which best fit the data (or null if not found)

iterations = 0
bestFit = null
bestErr = something really large

while iterations < k do
    maybeInliers := n randomly selected values from data
    maybeModel := model parameters fitted to maybeInliers
    alsoInliers := empty set
    for every point in data not in maybeInliers do
        if point fits maybeModel with an error smaller than t
            add point to alsoInliers
    end for
    if the number of elements in alsoInliers is > d then
        // This implies that we may have found a good model
        // now test how good it is.
        betterModel := model parameters fitted to all points in maybeInliers and alsoInliers
        thisErr := a measure of how well betterModel fits these points
        if thisErr < bestErr then
            bestFit := betterModel
            bestErr := thisErr
        end if
    end if
    increment iterations
end while

return bestFit
```

Listing 2.1. RANSAC Algorithm

## 2.3 Epipolar geometry

The problem of recovering structure from a single view lies in the fact that there is intrinsic ambiguity of the mapping between 3D and 2D. That is why the triangulation from at least two views of the same scene is required (Fig. 2.6). By using triangulation it is possible to recover a really good approximation of the point distance from the stereo camera system center.

The optimal triangulation problem can be described as follows:

$$\min_{X} \quad d^2(x_1, P_1X) + d^2(x_2, P_2X) \tag{2.12}$$

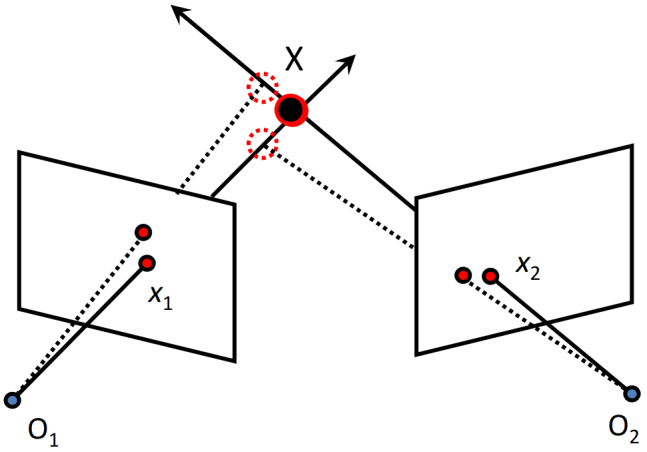



Figure 2.6. Triangulation scenario. Source: Fei-Fei Li - Stanford Vision Lab (2011)

The geometry described by a stereo vision system (Fig. 2.7) is called *epipolar geometry*. Before digging into the topic, some useful keywords are presented:

- **Epipole**: Since that the centers of the cameras lenses are distinct, each center projects onto a distinct point into the other camera's image plane. These two image points, denoted by $e_L$ and $e_R$, are called epipoles or epipolar points. Both epipoles $e_L$ and $e_R$ in their respective image planes and both optical centers OL and OR lie on a single 3D line.

- **Epipolar line**: The line $O_L - X$ is seen by the left camera as a point because it is directly in line with that camera's lens optical center. However, the right camera sees this line as a line in its image plane. That line $(e_R - x_R)$ in the right camera is called *epipolar line*. Symmetrically, the line $O_R - X$ seen by the right camera as a point is seen as epipolar line $e_L - x_L$ by the left camera. An epipolar line is a function of the position of point X in the 3D

space (i.e. as X varies, a set of epipolar lines is generated in both images). Since the 3D line $O_L - X$ passes through the optical center of the lens $O_L$, the corresponding epipolar line in the right image must pass through the epipole $e_R$ (and correspondingly for epipolar lines in the left image). All epipolar lines in one image contain the epipolar point of that image. In fact, any line which contains the epipolar point is an epipolar line since it can be derived from some 3D point X.

- **Epipolar plane**: As an alternative visualization, the points X, $O_L$ and $O_R$ can be considered. They form a plane called *epipolar plane*. The epipolar plane intersects each camera's image plane where it forms lines (i.e the epipolar lines). All epipolar planes and epipolar lines intersect the epipole regardless of where X is located.

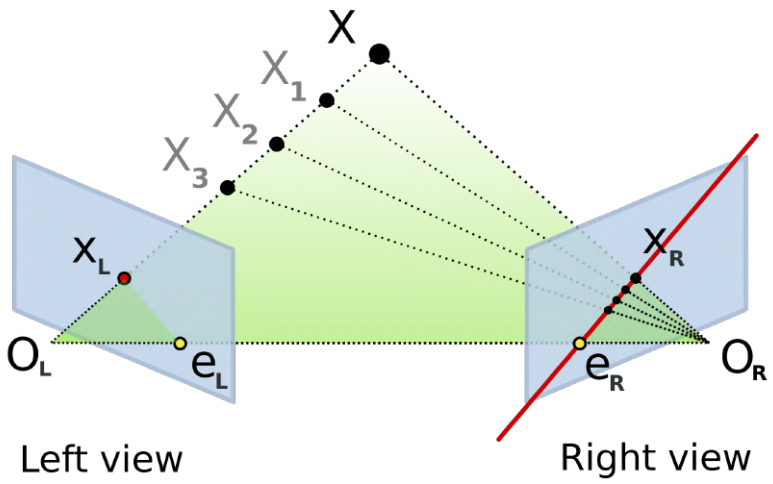



Figure 2.7. Epipolar geometry. Source: Wikipedia.org

### 2.3.1 Epipolar constraint

Starting from two views of the same scene and knowing the camera matrices and positions, how it is possible to find the correspondent right point, given the left point? The *epipolar constraint* assures us that potential matches from $x_L$ have to lie on the corresponding epipolar line $x_R - e_R$, while potential matches from $x_R$ have to lie on the corresponding epipolar line $x_L - e_L$ (Fig. 2.7). This means that,

thanks to the epipolar constraint:

$$p- > MP = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} <=> p- > M'P = \begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} \tag{2.13}$$

And consequently, by using the projection matrices (with known K matrices):

$$M = K[I\ \ 0], M' = K[R\ \ T] - > M = [I\ \ 0], M' = [R\ \ T] \tag{2.14}$$

In order to project the point from left to right, a translation and rotation must be applied: $T \times (Rp')$. So:

$$p^T \cdot [T \times (Rp')] = 0 - > p^T \cdot [[T_x] \cdot Rp'] = 0 \tag{2.15}$$

Where $[T_x]$ is the skew matrix of $T$ and $[T_x] \cdot R$ represents the *essential matrix* $E$. In Fig. 2.8 some of the most important properties are explained:

- $Ep_2$ is the epipolar line associated with $p_2$ ($l_1 = Ep_2$)
- $E^T p_1$ is the epipolar line associated with $p_1$ ($l_2 = E^T p_1$ )
- $E$ is singular (rank two)
- $Ee_2 = 0$ and $E^T e_1 = 0$
- $E$ is 3x3 matrix; 5 DoF

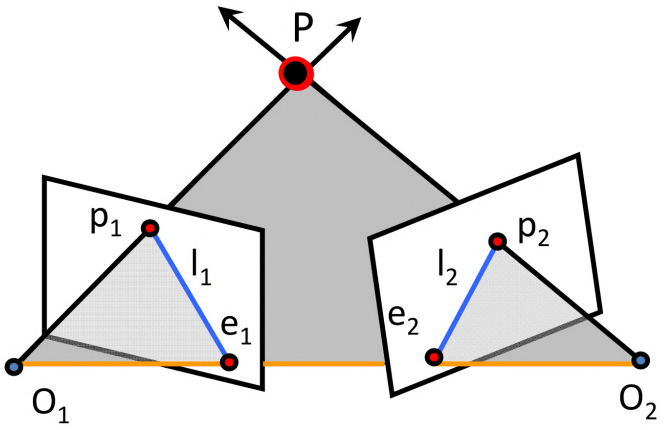



Figure 2.8. Stereo vision dimensions. Source: Fei-Fei Li - Stanford Vision Lab (2011)

**E estimation**

Given the absolute importance of this matrix, an estimation algorithm was proposed by Longuet-Higgins in 1981. The *eight-point algorithm* [2] is a method widely used for retrieving the E matrix given at least eight correspondences between the two stereo pairs of images. The basic eight-point algorithm consists of three principal steps:

1. First, it formulates a homogeneous linear equation, where the solution is directly related to $E$.
2. Then, it solves the equation, taking into account that it may not have an exact solution.
3. Finally, the internal constraints of the resulting matrix are managed.

The first step is described in Longuet-Higgins' paper, while the second and third steps are standard approaches in estimation theory.
The constraint defined by the essential matrix $E$ is 2.14, so

$$p^T E p' = 0 \tag{2.16}$$

For corresponding image points represented in normalized image coordinates. The problem which the algorithm solves is to determine $E$ for a set of matching image points. In practice, the image coordinates of the image points are affected by noise and the solution may also be over-determined (which means that it may not be possible to find $E$ which satisfies the above constraint exactly for all points. This issue is addressed in the second step of the algorithm.

### 2.3.2 Fundamental matrix

What happens if K matrices are unknown? The problem would involve not only the point $p$, but $p- > K^{-1}p$. Following this idea and substituting it into 2.15, the new equation will be:

$$(K^{-1}p)^T \cdot [T \times (RK'^{-1}p')] = 0- > p^T K^{-T} \cdot [T_x] \cdot RK'^{-1}p' = 0 \tag{2.17}$$

It is clear that a new relation between $p$ and $p'$ is found: the *fundamental matrix* $F = K^{-T} \cdot [T_x] \cdot RK'^{-1}$.

$$p^T F p' = 0 \tag{2.18}$$

The equation 2.18 has many important consequences and implications. By looking at the Fig. 2.8, some properties can be described:

- $Fp_2$ is the epipolar line associated with $p_2$ ($l_1 = Fp_2$)
- $F^T p_1$ is the epipolar line associated with $p_1$ ($l_2 = F^T p_1$ )
- $F$ is singular (rank two)
- $Fe_2 = 0$ and $F^T e_1 = 0$
- $F$ is 3x3 matrix; 7 DoF

$F$ captures information about the epipolar geometry of two views and camera parameters together. It gives constraints on how the scene changes under viewpoint transformation (without reconstructing the scene). It is a powerful tool in:

- 3D reconstruction
- Multi-view object/scene matching

**F estimation**

An estimation algorithm was proposed by Hartley, by modifying the eight-point algorithm in *normalized eight-point algorithm* [3]. The adopted procedure is near the same as for the $E$ estimation. His analysis of the problem showed that the it is caused by the poor distribution of the homogeneous image coordinates in their space. In order to address this issue, Hartley added another step: *normalization*. He proposed that the coordinate system of each of the two images should be transformed, independently, into a new coordinate system. According to the following principle:

- The origin of the new coordinate system should be centered (have its origin) at the center of gravity of the image points (a translation of the original origin to the new one).
- After the translation, the coordinates are uniformly scaled so that the mean distance from the origin to a point equals $\sqrt{2}$.

## 2.4 Multi-view geometry

The next and final step is to extend the epipolar geometry to multi-view camera system. In order to understand and categorize this project work and explore the multiple-view geometry problem, a necessary step is defined by the resolution of the triangulation problem in stereo vision. The estimation of the distance between the stereo system and an object surface is divided into two sub-problems:

- Solve the correspondence problem.

- Use corresponding observations to triangulate.

The depth estimation can be obtained by analyzing the disparity (inversely proportional to depth). By using Fig. 2.9 as reference, the disparity is:

$$p - p' = \frac{B \cdot f}{z} \tag{2.19}$$

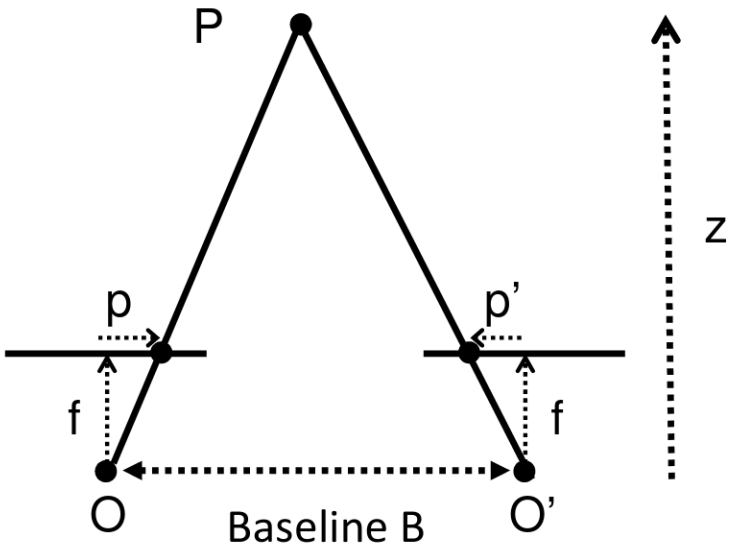



Figure 2.9. Disparity calculation from two views.

For what concerns the correspondence problem, the solution is a little bit more elusive. One of the most used families of methods are the *correlation methods*. These methods use a fixed sliding window in both the images, trying to maximize the dot product between the $w$ vector (vectorization of the pixels in the sliding windows of the first image) and the $w'$ vector (vectorization of the pixels in the sliding windows of the second image). These methods suffer of occlusions, fore shortening effects and other problems, but during the years they have been improved with some non-local constraints addition:

- **Uniqueness**: For any point in one image, there should be at most one matching point in the other image.
- **Ordering**: Corresponding points should be in the same order in both views.
- **Smoothness**: Disparity is typically a smooth function of x (except in occluding boundaries).

Finally, after solving the triangulation problem, the very objective of this first chapter can be categorized in a particular multi-view geometry problem: *structure from motion*.

### 2.4.1 Structure from motion (SfM)

The structure from motion problem (Fig. 2.10) can be formalized as follows. From the $m \times n$ correspondences $\boldsymbol{p_{ij}}$, estimate:

- $m$ projection matrices $\boldsymbol{M_i}$ (motion).
- $n$ 3D points $\boldsymbol{P_j}$ (structure).

SfM can be solved up to a N-degree of freedom ambiguity. In the general case (nothing is known) the ambiguity is expressed by an arbitrary affine or projective transformation. When the cameras are calibrated though, the ambiguity is reduced to a scaled factor (similarity). The scale (for calibrated cameras) will be the one and only ambiguity. That is why, when the reconstruction is up to scale, it is called *metric*.

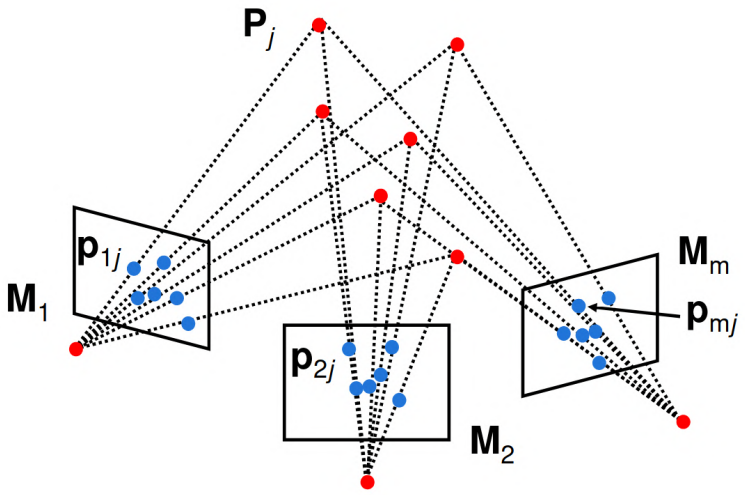



Figure 2.10. The structure for motion problem. Source: Fei-Fei Li - Stanford Stereo Lab (2011)

In order to recover camera and geometry up to an ambiguity, the first thing to do is to use the camera affine approximation (if possible), since it simplifies a lot the problem. Then, the *factorization method* comes to help:

1. Given: $m$ images and $n$ features $\boldsymbol{p}_{ij}$
2. For each image $i$, center the feature coordinates

3. Construct a $2m \times n$ measurement matrix $\boldsymbol{D}$:
   - Column $j$ contains the projection of point $j$ in all views
   - Row $i$ contains one coordinate of the projections of all the $n$ points in image $i$
4. Factorize $\boldsymbol{D}$:
   - Compute SVD: $\boldsymbol{D} = \boldsymbol{U}\boldsymbol{W}\boldsymbol{V}^T$
   - Create $\boldsymbol{U}_3$ by taking the first 3 columns of $\boldsymbol{U}$
   - Create $\boldsymbol{V}_3$ by taking the first 3 columns of $\boldsymbol{V}$
   - Create $\boldsymbol{W}_3$ by taking the upper-left $3 \times 3$ block of $\boldsymbol{W}$
5. Create the motion and shape matrices:
   - $\boldsymbol{M} = \boldsymbol{M} = \boldsymbol{U}_3$ and $\boldsymbol{S} = \boldsymbol{W}_3\boldsymbol{V}_3^T$ (or $\boldsymbol{U}_3\boldsymbol{W}_3^{\frac{1}{2}}$ and $\boldsymbol{S} = \boldsymbol{W}_3^{\frac{1}{2}}\boldsymbol{V}_3^T$)

| Conditions | N. Views |
|---|---|
| Constant internal parameters | 3 |
| Aspect ratio and skew known<br>Focal length and offset vary | 4* |
| Aspect ratio and skew known<br>Focal length and offset vary | 5* |
| Skew =0, all other parameters vary | 8* |

Table 2.1. Conditions needed for metric upgrade based on the number of views.

Once that both camera and geometry are recovered, an metric upgrade can be performed in order to have only scale ambiguity and remove perspective or affine ambiguity. Some prior knowledge on cameras or scene can be used to add constraints and remove ambiguities. In table 2.4.1 are shown some of the possible conditions (prior knowledge) with the respective number of views.

### Bundle adjustment

It is used as non-linear method for refining structure and motion. Since it can be applied before or after the metric upgrade, it is flexible. It refines the structure and motion by minimizing the re-projection error:

$$E(M, P) = \sum_{i=1}^{m} \sum_{j=1}^{n} D(p_{ij}, M_i P_j)^2 \tag{2.20}$$

On one hand, it can be very useful, since it handles large number of views while handling missing data. On the other hand, it is a large minimization problem (parameters grow with number of views) and requires a good initial condition.

# Chapter 3

# Research

This chapter is devoted to the research on the state-of-the-art methods and approaches that can be found in literature. In particular, for each problem encountered in this project, some deep learning or computer vision solutions are presented. The research has been developed in order to solve sequentially camera calibration's problems, object detection and pose estimation, features detection and structure from motion. Once that each positive and negative aspect of each method has been analyzed, a final solution is proposed (that better suits the needs and the available tools in order to implement it).

## 3.1 Calibration methods

The first problem encountered faces the initial calibration of the camera's intrinsic parameters (see 2.4). This step must be performed before using any cameras, since this matrix is fundamental in order to recover the local coordinates of the pinhole camera model from the pixel ones. A common behaviour is to save these parameters in a persistent storage.

The previously used approach for camera calibration (linear regression and SVD) directly estimates 11 unknowns in the M matrix using known 3D points $(X_i, Y_i, Z_i)$ and measured feature positions $(u_i, v_i)$ (see 2.10 and 2.11). As advantages it is true that all the specifics of the camera are summarized in one matrix and that it can predict where any world point will map to in the image. Unfortunately, since it mixes up internal and external parameters, it can not tell about any specific parameter. This approach is pose-specific.

A better approach is given by a probabilistic view of least square: the *non-linear least square methods*. They are the most used methods for intrinsic matrix camera calibration. In particular, they refer to the *General Calibration Problem*:

$$X_{measurement} = f(P_{parameter}) \tag{3.1}$$

In this scenario, the most used optimization algorithms are:

- Newton method.
- Levenberg-Marquardt Algorithm (LMA)[1]. Is used in many software applications for solving generic curve-fitting problems. However, as with many fitting algorithms, the LMA finds only a local minimum, which is not necessarily the global minimum. The LMA interpolates between the Gauss–Newton algorithm (GNA) and the method of gradient descent.

Both of them are iterative algorithms and start from an initial solution. The estimated solution may be function of the initial solution and the initial solution is crucial for the speed of the algorithm. Since a good initial solution is required, a possible algorithm for the camera calibration could be summarized as follows:

1. Solve linear part of the system to find approximated solution.
2. Use this solution as initial condition for the full system.
3. Solve full system (including distortion) using Newton or LMA.

Where the typical assumptions for computing initial condition are:

- Zero-skew, square pixel.
- $u_0, v_0$, known center of the image.
- No distortion.

### 3.1.1 Tsai's calibration technique

The technique is implemented with a two-stage algorithm that first estimates the pose and then it computes the focal length, distortion coefficients and the z-axis translation by the optimization of the re-projection error [4].

#### First stage

The goal of the first stage is the estimation of $m_1$ and $m_2$:

$$p_i = \begin{bmatrix} u_i \\ v_i \end{bmatrix} = \frac{1}{\lambda} \begin{bmatrix} \frac{m_1 P_i}{m_3 P_i} \\ \frac{m_2 P_i}{m_3 P_i} \end{bmatrix} -> \frac{u_i}{v_i} = \frac{m_1 P_i}{m_2 P_i} \tag{3.2}$$

[1] Donald Marquardt, *An Algorithm for Least-Squares Estimation of Nonlinear Parameters*, in SIAM Journal on Applied Mathematics, vol. 11, n. 2, 1963, pp. 431–441, DOI:https://dx.doi.org/10.1137%2F0111030.

The equation 3.2 can be rewritten as an equations system:

$$\begin{cases} v_1(m_1 P_1) - u_1(m_2 P_1) = 0 \\ v_i(m_1 P_i) - u_i(m_2 P_i) = 0 \\ \vdots \\ v_n(m_1 P_n) - u_n(m_2 P_n) = 0 \end{cases} \tag{3.3}$$

Then, the estimation of $m_1$ and $m_2$ can be performed by solving this equation system.

#### Second stage

In the second stage the Tsai's algorithm computes the focal length, distortion coefficients and the z-axis translation. This is done by estimating $m_3$, given $\lambda$, $m_1$ and $m_2$ ($m_3$ is a non linear function of $\lambda$, $m_1$ and $m_2$, see 3.2).

As it was already mentioned, LMA is of particular use in this kind of problems.

## 3.2 Object detection

The next sub-problem encountered is the object detection problem. This kind of problems had in the last decade a big step. In the era of machine learning and deep learning, there are many deep neural networks and solutions to solve object detection, even directly in 3D (by performing a pose estimation too). During the research process many deep learning architectures have been studied and analyzed, until only some of them were chosen to possibly be integrated in the project solution.

### 3.2.1 Fast/Faster R-CNN

R-CNNs are the state-of-the-art object detection architectures using regions proposal. In particular, the *Fast* and *Faster* revisions (Fig. 3.1) give really good results [5]. In order to performs the detections, the network follows these steps:

1. Feed the image into a ConvNet.
2. Run selective search on the feature map for region proposals.
3. Pool the proposals into FC layers.
4. Use Softmax to classify and regression to draw the bounding boxes.

Even with recent revisions (Fast/Faster), R-CNNs are really precise, but heavily slow in training and testing time.

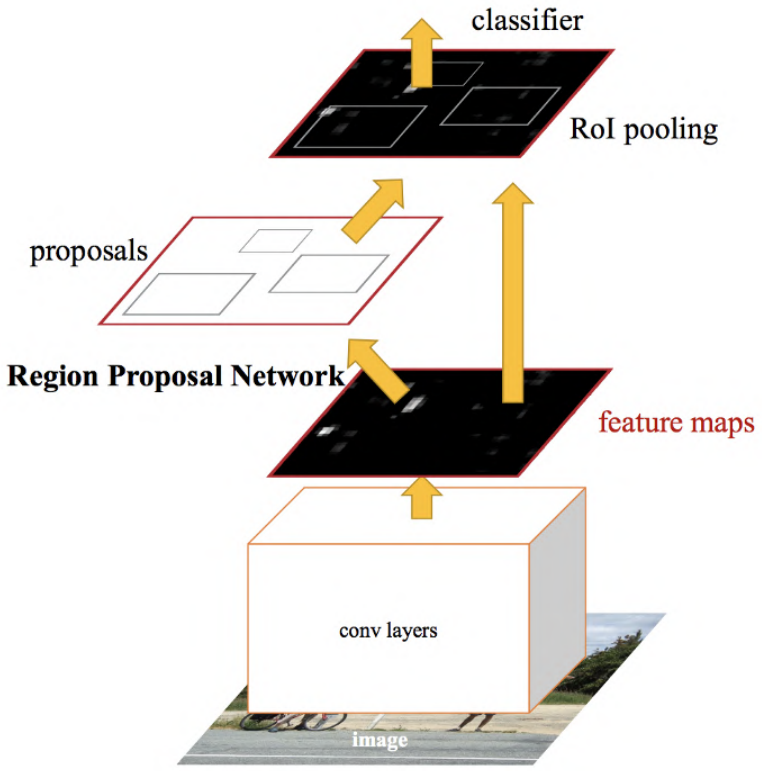



Figure 3.1. Faster R-CNN architecture, Source: Girshick et al. ICCV2015

### 3.2.2 YOLO

*You Only Look Once* (YOLO) and all its *v4* [6] are the state-of-the-art object detection architectures that do not use regions proposal. In order to retrieve the bounding boxes of the detected objects, the steps are as follows (Fig. 3.2):

1. SxS grid on input.
2. For each grid cell predicts B bounding boxes, C class probability and confidence.
3. Encode as S x S x (5 * B + C) tensor.
4. Final predictions.

YOLO has a good precision, it is not really powerful with groups of objects or far distance detection, but it is fast (real-time 30fps). Since the objective is to recognize fixed objects in a relatively closed open-space (i.e. a parking area) in order to filter out some features points of an image that do not belong to the object,

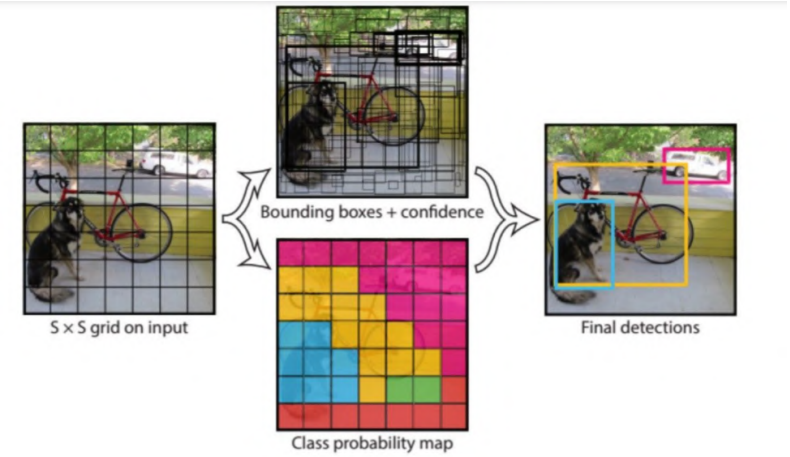



Figure 3.2. YOLO steps. Source: Redmon et al. CVPR2016

YOLO seems more suited: our interest is focused on close objects, not overlapped between them and medium to big sized. Furthermore, the fact that YOLO doesn't recognize small or distanced objects is a "feature" for our problem, since it can behave like a filter that discards spurious matches.

### 3.2.3 3D object detection

The 3D object detection uses the latest state-of-the-art research papers and tries to detect the 3D object with its pose (in order to be recognized from different points of view). This kind of networks were initially analyzed because of the potential given by the stereo cameras. After some problems during the project, it was decided to use simple monocular cameras. Those methods are not applicable on a monocular camera, but they will be presented anyway in order to describe the entire conducted process of research.

#### Stereo R-CNN

This first approach (Li, et al.) is an extension of R-CNN for Stereo Cameras[2]. The network architecture (Fig. 3.3) is composed by two similar pipelines. Both of them use ResNet-101 as CNN backbone, but the first one is used for keypoint prediction while the second one for the stereo regression. The entire Stereo R-CNN outputs stereo boxes, key points, dimensions, and the viewpoint angle, followed by the 3D box estimation and the dense 3D box alignment module.

[2]Peiliang Li, Xiaozhi Chen, and Shaojie Shen. *Stereo R-CNN based 3D Object Detection for Autonomous Driving*. 2020. URL: https://arxiv.org/abs/1902.09738

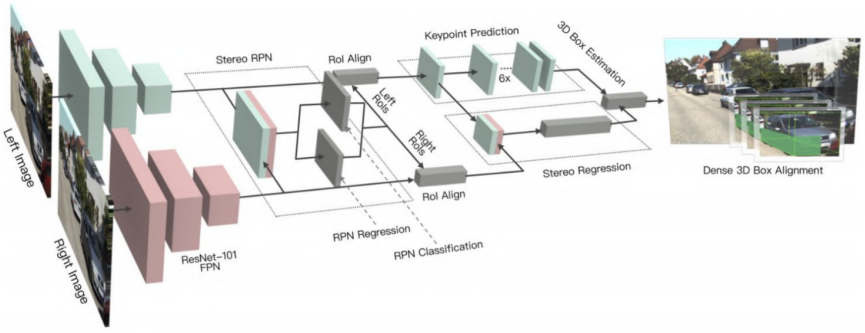



Figure 3.3. Stereo R-CNN architecture. Source: Li et al.

### Frustum PointNets

This second network is a hybrid approach: since a stereo cameras can behave (with some approximation) like a LiDAR, giving the point cloud and a starting RGB-D image, there are in the literature many 3D object detection algorithms that uses Deep Learning Networks on point clouds to retrieve the region and the 3D representation of the objects (e.g. Qi, et al.)[3].

The network architecture (Fig. 3.4) woks as follows:

- First, it leverages a 2D CNN object detector to propose 2D regions and classify their content.
- 2D regions are then lifted to 3D and thus become frustum proposals.
- Given a point cloud in a frustum ($n \times c$ with $n$ points and $c$ channels of $XYZ$, intensity etc. for each point), the object instance is segmented by binary classification of each point.
- Based on the segmented object point cloud ($m \times c$), a light-weight regression PointNet (T-Net) tries to align points by translation such that their centroid is close to amodal box center.
- At last, the box estimation net estimates the amodal 3D bounding box for the object.

[3]Charles R. Qi, , Wei Liu, Chenxia Wu, Hao Su, and Leonidas J. Guibas. *Frustum PointNets for 3D Object Detection from RGB-D Data*. 2018. URL: https://arxiv.org/abs/1711.08488

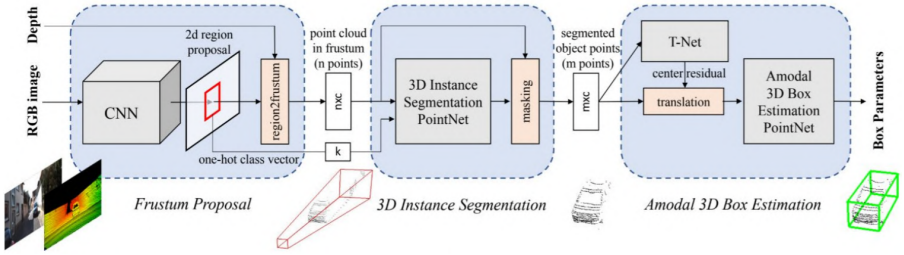



Figure 3.4. Frustum PointNets architecture. Source: Qi et al.

## 3.3 Features extraction and matching

Once that the bounding boxes of the reference object have been found by exploiting one of the previous object detection methods, it is now necessary to apply some features detection algorithm on the photos. This is done in order to retrieve from the n monocular cameras as much features points as possible, filter out the ones that are not in the bounding box and try to match those features among the different photos so that at the end of this step we will have the 3D points of the object and the corresponding pixel coordinates for each photos (i.e. cameras).

In the literature, a lot of features detection and matching algorithms are available. The research process produced as candidates the most efficient and used ones for features extraction (SIFT, SURF) and features matching (Brute-force, FLANN) with the most innovative and performing SuperPoint (extraction) and SuperGlue (matching).

### 3.3.1 Extraction

The features extraction process takes as input one or multiple images and returns as output the keypoints (as list of pixel coordinates) that represent the features of the image. Based on the specific algorithm used, these keypoints could be more or less robust or more or less difficult to retrieve (by means of the computational complexity).

#### Scale-invariant feature transform (SIFT)

The SIFT algorithm is widely used nowadays. A peculiar property of this algorithm is the *scale-invariant* factor, that allows to retrieve the same points even if the scale of the image was changed [7]. This algorithm is composed by mainly four steps:

- Scale-space extrema detection
- Keypoint localization

- Orientation assignment
- Keypoint descriptor

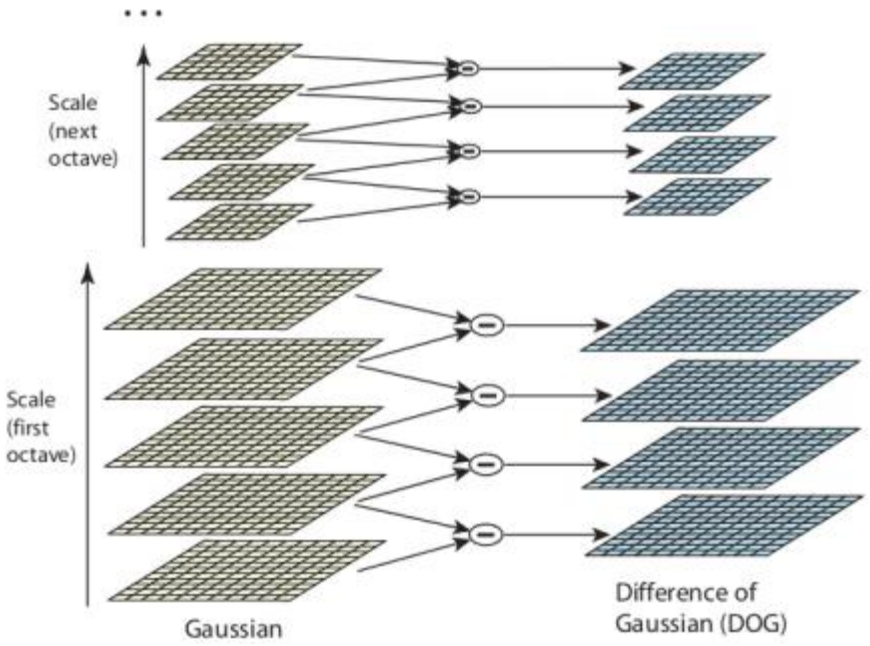



Figure 3.5. DoF for each octaves in Gaussian Pyramid. Source: OpenCV Docs

**Scale-space extrema detection** From the image above, it is obvious that we can't use the same window to detect keypoints with different scale. It is OK with small corner, but to detect larger corners we need larger windows. For this, scale-space filtering is used. In it, Laplacian of Gaussian is found for the image with various $\sigma$ values. LoG acts as a blob detector which detects blobs in various sizes due to change in $\sigma$. In short, $\sigma$ acts as a scaling parameter. For example, in the above image, Gaussian kernel with low $\sigma$ gives high value for small corner while Guassian kernel with high $\sigma$ fits well for larger corner. So, we can find the local maxima across the scale and space which gives us a list of $(x, y, \sigma)$ values (which means there is a potential keypoint at $(x, y)$ at $\sigma$ scale.

But this LoG is a little costly, so SIFT algorithm uses Difference of Gaussians which is an approximation of LoG. Difference of Gaussian is obtained as the difference of Gaussian blurring of an image with two different $\sigma$, let it be $\sigma$ and $k\sigma$. This process is done for different octaves of the image in Gaussian Pyramid (Fig. 3.5).

Once this DoG are found, images are searched for local extrema over scale and space. For example, one pixel in an image is compared with its 8 neighbours as

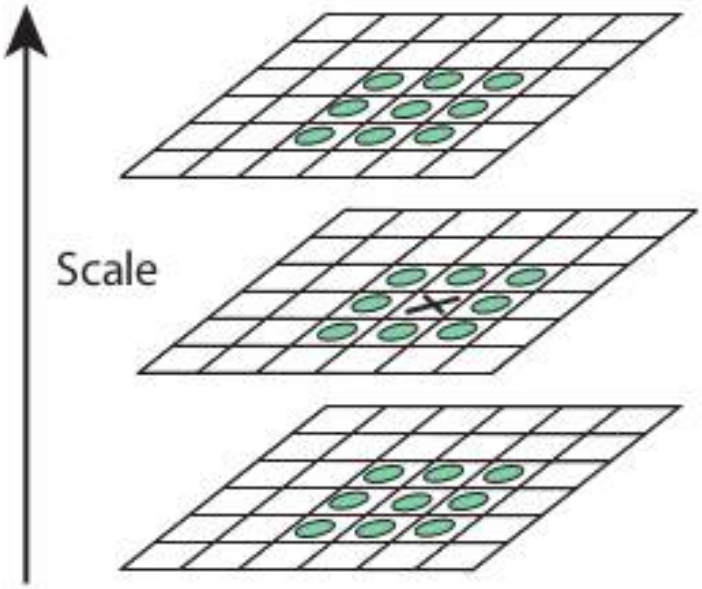



Figure 3.6. SIFT local extrema. Source: OpenCV Docs

well as 9 pixels in next scale and 9 pixels in previous scales. If it is a local extrema, it is a potential keypoint. It basically means that keypoint is best represented in that scale (Fig. 3.6).

Regarding different parameters, the paper gives some empirical data which can be summarized as:

- `number of octaves = 4`
- `number of scale levels = 5`
- `initial` $\sigma = 1.6$,
- $k = \sqrt{2}$

**Keypoint localization** Once that potential keypoints locations are found, they have to be refined to get more accurate results. It is used the Taylor series expansion of scale space to get more accurate location of extrema, and if the intensity at this extrema is less than a threshold value (0.03 as per the paper), it is rejected.

DoG has higher response for edges, so edges also need to be removed. For this, a concept similar to Harris corner detector is used: a $2 \times 2$ Hessian matrix ($H$) to compute the principal curvature. It is known from Harris corner detector that for edges, one eigen value is larger than the other. So here it is used a simple function: if this ratio is greater than a threshold, that keypoint is discarded. With this step

any low-contrast keypoints and edge keypoints are removed and what remains are strong interest points.

**Orientation assignment** An orientation is assigned to each keypoint to achieve invariance to image rotation. A neighborhood is taken around the keypoint location (depending on the scale), and the gradient magnitude and direction is calculated in that region. An orientation histogram with 36 bins covering 360 degrees is created (it is weighted by gradient magnitude and gaussian-weighted circular window with $\sigma = 1.5 \times scale$). The highest peak in the histogram is taken and any peak above 80% of it is also considered to calculate the orientation. It creates keypoints with same location and scale, but different directions. This is done in order to contribute to the stability of matching.

**Keypoint descriptor** The keypoint descriptor is created. A $16 \times 16$ neighbourhood around the keypoint is taken. It is divided into 16 sub-blocks of $4 \times 4$ size. For each sub-block, 8 bin orientation histogram is created. So, a total of 128 bin values are available. It is represented as a vector to form keypoint descriptor. In addition to this, several measures are taken to achieve robustness against illumination changes, rotation and so on.

Finally, during the keypoint matching, keypoints between two images are matched by identifying their nearest neighbours. In some cases, the second closest-match may be very near to the first (due to noise or some other reasons). In that case, ratio of closest-distance to second-closest distance is taken. If it is greater than 0.8, they are rejected. This step eliminates around 90% of false matches while discards only 5% correct matches (as per the paper).

### Speed-up robust features (SURF)

As the name suggests, it is a speeded-up version of SIFT. SIFT uses Lowe approximated Laplacian of Gaussian with Difference of Gaussian for finding scale-space. SURF, goes a little further and approximates LoG with Box Filter. One big advantage of this approximation is that convolution with box filter can be easily calculated with the help of integral images (even in parallel for different scales). Also, the SURF algorithm relies on determinant of Hessian matrix for both scale and location [8].

In short, SURF adds a lot of features to improve the speed in every step. Analysis shows it is 3 times faster than SIFT while performance is comparable to SIFT. SURF is good at handling images with blurring and rotation, but not good at handling viewpoint change and illumination change.

### 3.3.2 Matching

The features matching step is done after the features extraction one. In particular, when the features are extracted in two images, the matching algorithms try to associate each feature of the first image to each feature of the second one. The output of the matching is a list of matches that contains the pixel coordinates of the first keypoint, the pixel coordinates of the corresponding keypoint in the second image and the confidence of the match.

This step is crucial, since it allows to retrieve the same real-world points from different angles in two different photos.

#### Brute-force

The most simple and accurate matcher is the brute-force matcher. It takes the descriptor of one feature in the first set and it matches it with all other features in the second set using some distance calculation (e.g. L2/L1 norm, Hamming distance). The closest one is returned.

Unfortunately, for big set of features, this algorithm is really slow, since it its computational complexity is really demanding.

#### FLANN-based

In order to solve the computational complexities of the brute-force for large-scale datasets, this is a new method that uses approximation when retrieving the distance measurement between features. In fact, FLANN stands for *Fast Library for Approximate Nearest Neighbors* [9].

A good matching routine would uses the Lowe's ratio test too. Lowe proposed to use a distance ratio test to try to eliminate false matches. The distance ratio between the two nearest matches of a considered keypoint is computed and it is a good match when this value is below a threshold. Indeed, this ratio allows helping to discriminate between ambiguous matches (distance ratio between the two nearest neighbors is close to one) and well discriminated matches.

#### SuperGlue

SuperGlue can be seen as deep learning applied on features matching. It is a neural network that matches two sets of local features by jointly finding correspondences and rejecting non-matchable points. Assignments are estimated by solving a differentiable optimal transport problem (whose costs are predicted by a graph neural network). As the researchers states [10]:

*"We introduce a flexible context aggregation mechanism based on attention, enabling SuperGlue to reason about the underlying 3D scene and feature assignments*

*jointly. Compared to traditional, hand-designed heuristics, our technique learns priors over geometric transformations and regularities of the 3D world through end-to-end training from image pairs. SuperGlue outperforms other learned approaches and achieves state-of-the-art results on the task of pose estimation in challenging real-world indoor and outdoor environments. The proposed method performs matching in real-time on a modern GPU and can be readily integrated into modern SfM or SLAM systems."*

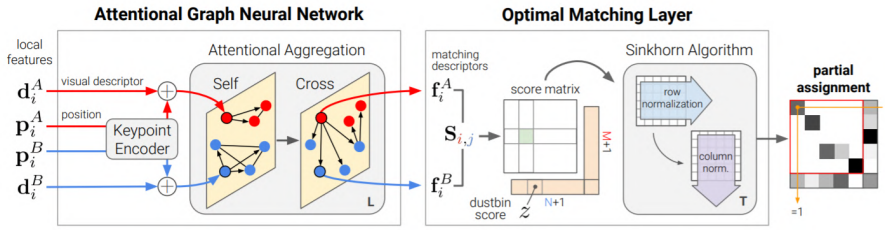



Figure 3.7. SuperGlue architecture. Source: Sarlin et al. 2020

The architecture (Fig. 3.7) is composed by:

- Attentional graph neural network. It maps keypoint positions with their descriptors into a vector in order to create a more powerful representations of them.
- Optimal matching layer. The representations $\boldsymbol{f}$ are passed to this layer that creates a matrix with each scores and tries to find the optimal partial assignment by exploiting the Sinkhorn-Knopp algorithm[4]

Basically, SuperGlue is a *matcher* that can use any local features detector (like SIFT), but it works really good with the SuperPoint detector: a fully-convolutional neural network architecture for interest point detection and description, trained using a self-supervised domain adaptation framework called Homographic Adaptation [11].

## 3.4 Perspective-n-Point

The already analyzed Perspective-n-Point can be seen as a sub-problem of the more general calibration problem. Given an image, the pixel features coordinates and the world coordinates in 3D, PnP tries to estimate the pose of the camera with

[4]From Wikipedia: "A simple iterative method to approach the double stochastic matrix by alternately re-scale all rows and all columns of A to sum to 1."

respect to the world, giving as output the rotation and translation. Fischler and Bolles [1] summarized the problem as follows:

*"Given the relative spatial locations of n control points, and given the angle to every pair of control points from an additional point called the Center of Perspective (CP), find the lengths of the line segments joining CP to each of the control points."*

### 3.4.1 Gao's P3P solver

The Gao's P3P solver is one of the most implemented and tested methods for the PnP problem [12]. The algebraic approach of this method uses Wu-Ritt's zero decomposition algorithm to give a complete triangular decomposition for the P3P equation system. This decomposition provides the first complete analytical solution to the P3P problem.

Then, a complete solution classification for the P3P equation system is also proposed (i.e. give explicit criteria for the P3P problem to have one, two, three, and four solutions). By combining the analytical solutions with the criteria, the *Complete Analytical Solution with the assistance of Solution Classification* (CASSC) algorithm is built, which may be used to find complete and robust numerical solutions to the P3P problem.

The P3P problem is a particular case of the more generic PnP one. More in details, the P3P solves the problem by using only 3 points (plus one for retrieving the correct solution). Obviously, the P3P algorithms can be used with more than three points in order to achieve better results.

### 3.4.2 Lambda twist

In 2018, Persson and Nordberg [13] proposed a new method for the P3P problem: lambda twist. This method does not find all roots to a quartic and discard geometrically invalid and duplicate solutions in a post-processing step, but it exploits the underlying elliptic equations (which can be solved by a fast and numerically accurate diagonalization). This diagonalization requires a single real root of a cubic, which is then used to find the solutions. Unlike the direct quartic solvers, this method never computes geometrically invalid or duplicate solutions.

Since it is a novel approach, the implementation and the integration in major libraries (e.g. OpenCV) is not extensively tested.

### 3.4.3 PnP-Net

Sheffer et al. in 2020 [14] proposed an application of deep neural networks on the PnP problem. The PnP-Net exploits an initial phase implemented by a neural network that estimates the coarse pose, followed by a second stage that uses the coarse

pose and solves the leas-squares problem of non-linear parameters by exploiting the already discussed Levenberg–Marquardt algorithm.

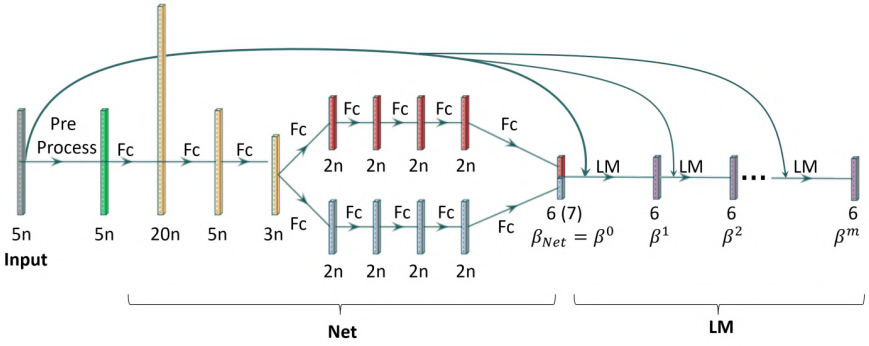



Figure 3.8. PnP-Net architecture. Source: Sheffer et al. ECCV2020

The architecture (Fig. 3.8) could be improved by adding a detection and matching stages before the PnP-Net itself and training the pipeline end-to-end. Moreover, another improvement could be the training of another network that will output the likelihoods of each correspondence being wrong. These outputs could be used in the LM stage as informative weights and improve the overall precision and robustness.

Unfortunately, since this approach is relatively recent, at this time there is no implementation proposed by the authors.

## 3.5 Structure from motion

The more generic problem of *Structure from Motion* (SfM) refers to the entire pipeline needed in order to recover the correspondences between images and the reconstruction of the 3D objects. The general flow is described by the following steps:

1. Features extraction on multiple images (see 3.3.1).
2. Features matching between multiple images (see 3.3.2).
3. RANSAC filter for outliers (see 2.2.1).
4. Place the world-coordinates as the first image local coordinates (first camera).
4. For each new image:
   - Retrieve the point cloud produced by the previous images.
   - Retrieve the pose of the new image with P3P using the point clouds (see 3.4).

5. Optimize the re-projection error:
   - Sliding-window with bundle adjustment (Fig. 3.9, see 2.4.1).

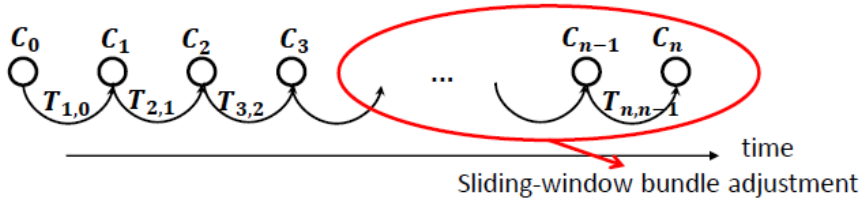



Figure 3.9. Representation of the sliding-window with bundle adjustment method

Each step has its own set of dedicated methods. The SfM problem requires complex theories and algorithms to support it, and it needs to be improved in accuracy and speed, so there are not many mature commercial applications.

# Chapter 4

# Design and development

## 4.1 Introduction

As it was already mentioned before, in order to build a program capable of implementing an autocalibration technique, it is necessary to follow a fixed pre-defined design and development steps.

The design of the solution (or solutions) aims at structuring the logical flow of the general pipeline with the singular implementation of the chosen algorithm in order to solve a particular step. During the design, the best algorithms and methods are chosen from the literature (see 3), based on their *pros&cons* with respect to the contextualization of the method itself in our particular situation. This is done so that the core of the project work (i.e. the development process) can be focused only on the code and not on how it works or why or when.

The development of the solution aims at building the designed logical flow with the chosen algorithms and methods in order to retrieve some experimental results on the proposed solutions.

## 4.2 Design

Two solutions are proposed: a custom pipeline and an OpenCV pipeline. The first one uses all the latest and state-of-the-art computer vision algorithms found in the literature with deep learning's object detection techniques. The second one implements the standard techniques defined in the *sfm* library from OpenCV.

### 4.2.1 Custom pipeline

The custom pipeline (Fig. 4.1) is:

1. Features extraction on multiple images.

2. Features matching between multiple images.
3. Filters for outliers.
4. Place the world-coordinates as the first image local coordinates (first camera).
5. For each image:
   - Retrieve the point cloud produced by the previous images.
   - Retrieve the pose of the image with P3P using the point clouds.
6. Optimize the re-projection error:
   - Sliding-window with bundle adjustment.

By looking at the generic proposed pipeline, the first encountered problem to solve is *features extraction on multiple images*. As demonstrated in DeTone et al. [14], the SuperPoint network is best suited for the task, since it is far more precise than the standard methods (i.e. ORB, Harris, SIFT) and has good performances thanks to the CUDA GPU acceleration. It is obvious that, with this choice for the first step, the choice for the *features matching between multiple images* is automatically defined by SuperGlue. The SuperPoint+SuperGlue end-to-end solution for features extraction and matching will be combined with more *classic* approaches like SURF+FLANN. This is done in order to study the contextualization of the problem and validate the two methods to choose the best one.

The next step is *filtering*. In particular, two level of filters are applied:

- *YOLOv4*
- *RANSAC*

It is clear that YOLOv4 is properly defined as a neural network for object detection tasks, but here is used to retrieve the bounding boxes around the car tires (i.e. tires object detection) and filter out all the points that are not fixed on them. This is done because the external area of the tires may be simplified in the 3D space and seen as a *coplanar* object[1]. This is an important aspect for future steps in order to convert the coordinates between different systems. In addition, the RANSAC algorithm is used as second level for general outliers removal.

The results of the two-staged filter would ideally be pixel points of a coplanar object without spurious matches. In order to convert the pixel coordinates in real world coordinates, the pipeline needs the intrinsic parameters of the camera (previously estimated and saved persistently). By exploiting the intrinsic matrix,

[1] All the points lie on the same plane

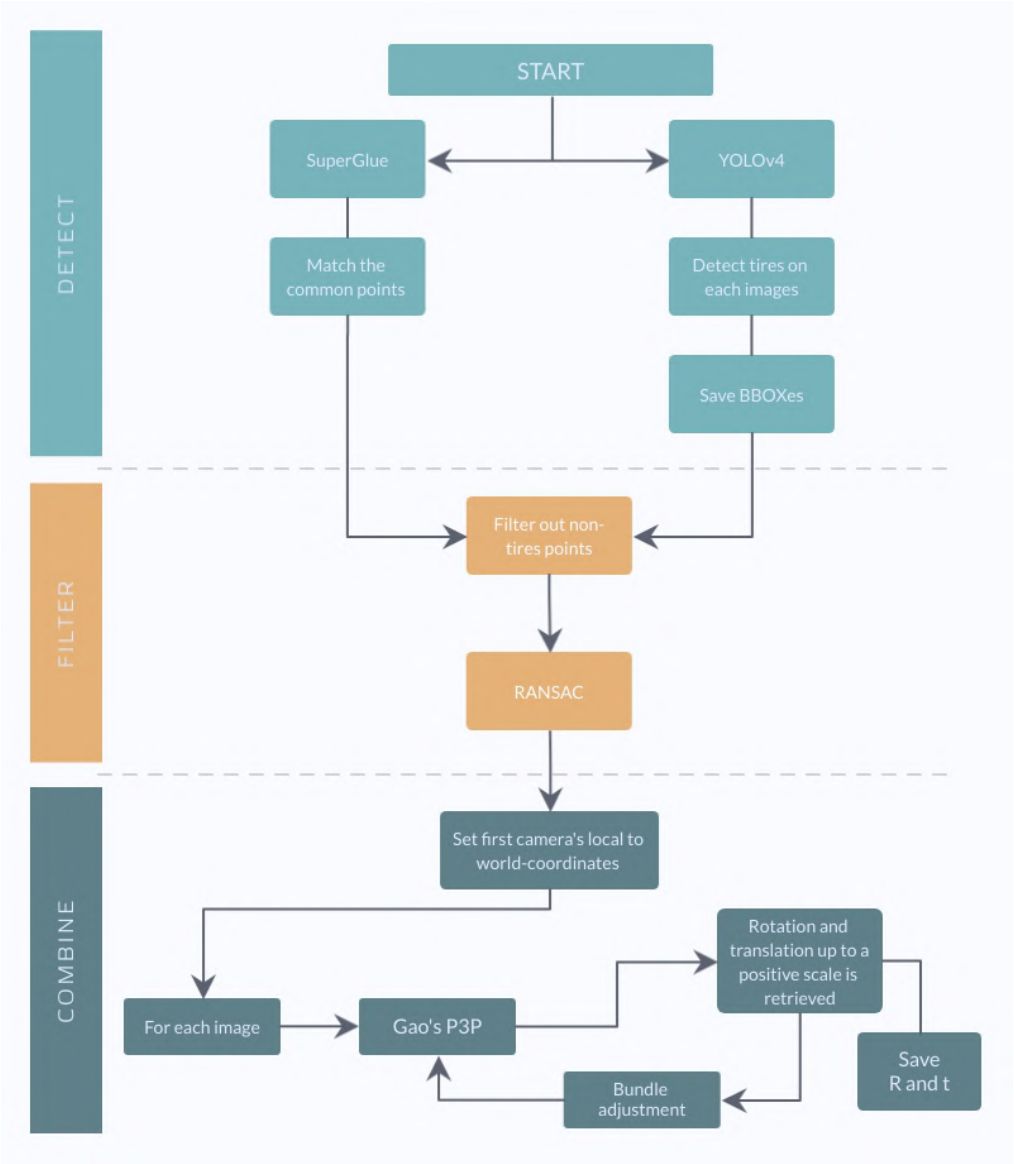



Figure 4.1. Custom pipeline's flow

the pixel coordinates are converted into local coordinates $(X, Y)$. Since that all the points lie on the same area (i.e. plane), the $Z$ coordinate (the distance between the camera and the point in local camera coordinates) can be simplified with an arbitrary and equal value (e.g. $Z = 1$), giving the complete 3D local coordinates $(X, Y, Z)$. Finally, the first camera's local coordinates system is placed as the world coordinates system.

Since that all the pixel correspondences are known in each image (thus, the 3D world coordinates points too), the point cloud is used to force the pose estimation

from the other $n-1$ images with respect to the first image by exploiting the Gao's P3P method.

Finally, for each image, the extrinsic parameters matrix (rotation and translation up to a scale factor with respect to the first image) is estimated, thus the position of each camera (i.e. image).

The final optimization is devoted to the Levenberg-Marquardt algorithm running as a sliding window in each pair of images.

This custom method uses the first image as reference image and confront it with each one of the other $n-1$ images in order to find the common matches and build the point cloud.

### 4.2.2 OpenCV's pipeline

A more *classic* approach can be exploited with OpenCV (Fig. ). In particular, the pipeline would be:

1. Estimates a precise initial reconstruction using the matches of two views:
   - Select common matches of the two images
   - Robustly estimate the fundamental matrix
   - Estimate the essential matrix from the fundamental matrix
   - Extract the relative motion from the essential matrix
   - If the first image has no camera, create the camera and initialize the pose to be the world coordinate frame
   - Estimate the absolute pose of the second camera from the first pose and the estimated motion.
   - Create and add the cameras to the reconstruction
   - Reconstruct only the inliers matches (point triangulation)
   - Perform a metric bundle adjustment
2. Estimates the pose of the keyframes using the already reconstructed points. For every keyframes (starting the `first_keyframe_index th`):
   - The keyframe is localized (by resection)
   - If the resection has not failed, the inliers tracks are reconstructed by point triangulation
   - If new points are created, a global bundle adjustment is performed

In order to reconstruct an initial solution for the first two images, the epipolar geometry is used. In particular:

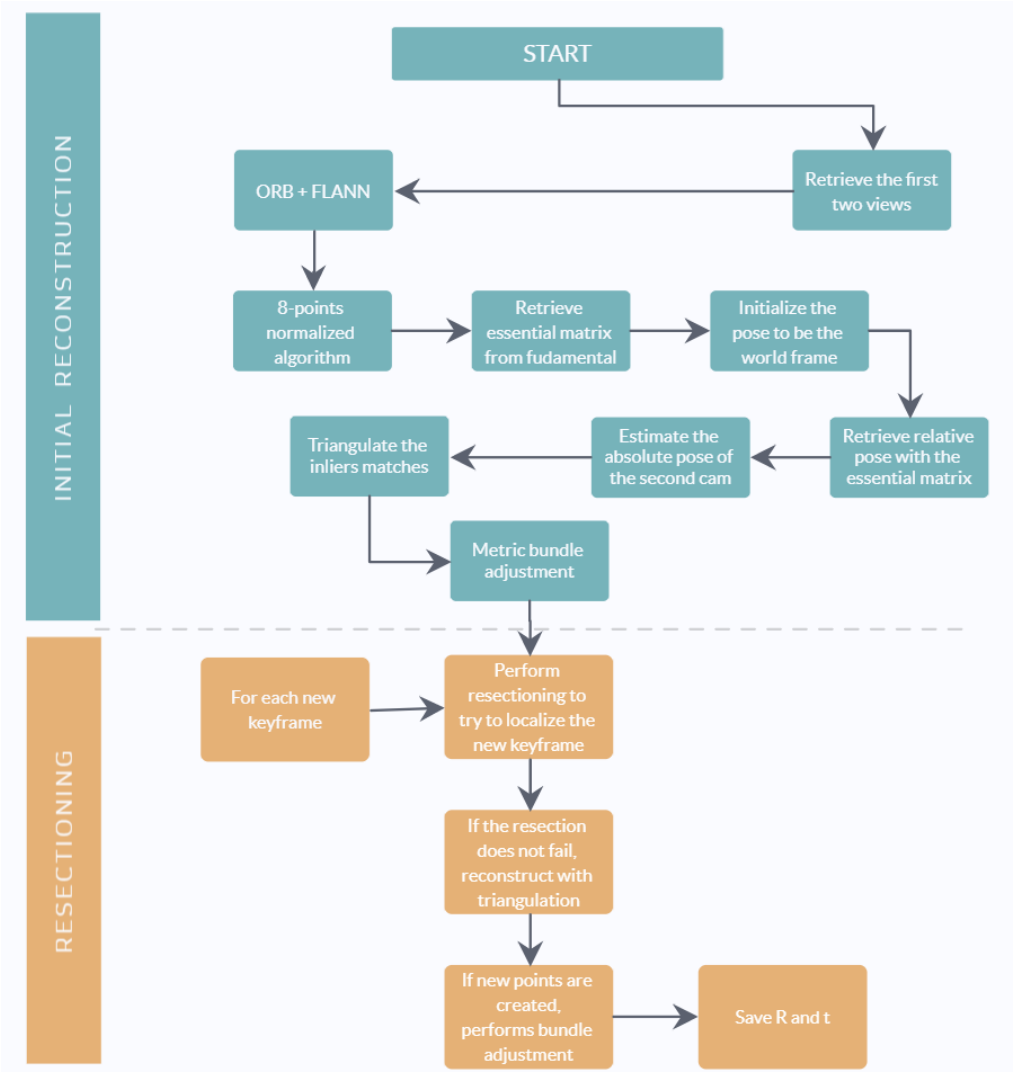



Figure 4.2. OpenCV pipeline's flow

- ORB+FLANN are used to get the matches.
- The fundamental matrix is then estimated by using the normalized eight-point algorithm.
- The essential matrix is retrieved from the fundamental matrix using the intrinsic parameters matrix of the camera.
- The relative motion (rotation and translation up to a positive scale) with respect to the first camera is retrieved from the essential matrix [Longuet-Higgins].

- The pose of the first camera is set as world coordinates and the absolute pose of the second camera is recovered:

$$R_2 = R_1 \times dR \tag{4.1}$$

$$t_2 = R_1 \times dt + t_1 \tag{4.2}$$

  Where $dt$ and $dR$ are the relative pose (rotation and translation) of the second camera with respect to the first camera retrieved in the previous step.

- Triangulate the inliers matches and perform a bundle adjustment

This first solution would be the final one only if the number of views are equal to two. If the views are greater than two, the pipeline estimates the pose using the already reconstructed points:

- The pose is simply tried to be retrieved by resection and then intersection.
- A final metric bundle adjustment is performed to reduce the re-projection error.

With respect to the custom pipeline this one is incremental (i.e. it analyzes images by images and one at the time), while the custom one uses the first image as reference and matches it with all the other $n - 1$ ones.

## 4.3 Development

The development section is the true core of the project work. Here is explained how the designed flows are implemented and what are the choices that have been made at this level to reach the final objective.

In particular, first they have been analyzed libraries and frameworks used and then a exhaustive analysis of the two implemented pipelines is proposed. The code is available on GitHub[2].

### 4.3.1 Frameworks and requirements

In order to develop the two flows, some prerequisites are required. The used operating system is *Ubuntu 20.04*.

An important aspect (in particular for the custom pipeline) is the availability of a CUDA capable device (NVIDIA discrete card), because of the faster inferences

[2] https://github.com/nopesir/thesis-project

on YOLOv4 and SuperGlue. CUDA Toolkit 11.1[3] has to be installed and fully functional (see A). Furthermore:

- OpenCV with OpenCV non-free enabled and CUDA enabled must be compiled from scratch and installed (see A).
- Other libraries (a `requirements.txt` file is available for the `pip3` Python environment).

### 4.3.2 Intrinsic calibration

The first essential step is the development of a script used to calibrate the intrinsic parameters of the camera (or the cameras). The `chess-board_calibration.py` script uses the built-in OpenCV libraries and methods to perform a classic chessboard calibration.

The important input data needed for calibration of the camera is the set of 3D real world points and the corresponding 2D coordinates of these points in the image. 2D image points can be easily found from the image (i.e. the locations where two black squares touch each other in chess boards).

For the 3D correspondent points, since the chessboard is a coplanar object (i.e. $Z = 0$), only $(X, Y)$ values are needed. The simplest way is to pass the points as (0,0), (1,0), (2,0), ... which denotes the location of points. In order to find the pattern in the chess board, it can be used the function `cv.findChessboardCorners()` with the kind of pattern of the grid (e.g. $8 \times 8$).

Once that a pattern is obtained, the corners are stored in a list and the corners are found (increasing their accuracy using `cv.cornerSubPix()`). The pattern is also drawn using `cv.drawChessboardCorners()`. The final step is the calibration:

```
ret, mtx, dist, _, _ = cv.calibrateCamera(objpoints, imgpoints,
    gray.shape[::-1], None, None)
```

Then, if `ret` is `True` (i.e. the matrices are valid), `mtx` (i.e. the camera matrix) and `dist` (i.e. the distortion coefficients) are saved in a `.npz` file persistently, ready to be used in the two pipelines.

For a rapid test in order to check the results, the photo is *undistorted* and shown. Furthermore, the script `chess-board_test.py` can be run in order to draw a cube for each image that is placed exactly on the chessboard.

### 4.3.3 Custom pipeline

Since the custom pipeline is built by exploiting the latest methods found in literature, its algorithms and components must be optimized accordingly. In particular,

[3] https://developer.nvidia.com/cuda-toolkit

YOLOv4 has to be firstly trained in order to detect the tires. The output of this training will be a `.weights` file that represent the trained network, ready to be used to test and recognize the tires. Moreover, the SuperGlue network must be configured and implemented in the pipeline too, alongside with the lambda twist P3P and the global main program flow.

### YOLOv4

**Training** The platform used for the training stage is the *Google Colaboratory* (Colab) platform[4] on the Darknet framework [15]. As training dataset, a subset of the Open Images V4 dataset is used[5] [16], that contains only photos and annotations of the car's tires. In order to download 15k images from Open Images, `OIDv4_ToolKit` is used [17].

Once that the dataset is ready and uploaded to Google Drive, an `.ipynb` is created in order to download the Darknet source code, build with CUDA acceleration and prepare the dataset from the uploaded `.zip`. More in details, `darknet.zip` contains:

- The Darknet source code.
- The configuration of the network (i.e. epochs, learning rate, layers definition).
- The train dataset ($\approx 15k$ images from Open Images).
- The validation dataset for mAP ($\approx 1.5k$ images from Open Images, about 10% the train set).

The code copies the entire `darknet.zip` and extracts it. Then, some changes are done into the `Makefile` in order to enable CUDA and finally the source is ready to be built:

```
%cp /content/drive/My\ Drive/Tesi\ Luigi/darknet.zip /content/
!unzip darknet.zip
%cd darknet/
!sed -i 's/OPENCV=0/OPENCV=1/' Makefile
!sed -i 's/GPU=0/GPU=1/' Makefile
!sed -i 's/CUDNN=0/CUDNN=1/' Makefile
!sed -i 's/CUDNN_HALF=0/CUDNN_HALF=1/' Makefile

!make -j4
```

[4] A cloud SaaS that enables Python execution in the browser without any configuration, free GPU access and simple sharing.

[5] https://storage.googleapis.com/openimages/web/index.html

The configuration of the training parameters is left as the default YOLOv4 paper. Then, the training phase starts. In particular, they have been used as starting weights the already pre-trained convolutional layers (`yolov4.conv.137`) in order to speed up the whole process:

```
%cd /content/darknet
!chmod +x darknet
!./darknet detector train obj.data yolov4-thesis.cfg \
   /content/drive/My\ Drive/yolov4.conv.137 -dont_show -map
```

This step will require about 8-10 hours of training, during which the *mean average precision* (mAP) was already past the 85% after $\frac{1}{6}$ of the process (Fig. 4.3). The final output of the training is the `yolov4-thesis_best.weights` file, safely stored on Google Drive and ready to be used for the tires detection.

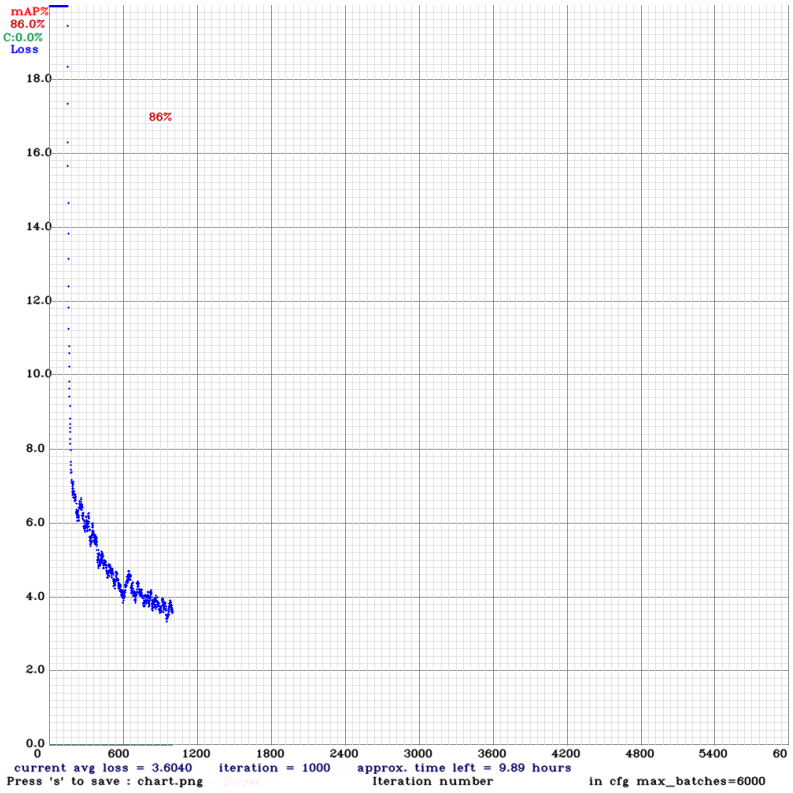



Figure 4.3. mAP and average loss at 1000/6000 epochs

Since that the weights file was too big for the remote repository, an automates script `weights-download.sh` is created in order to automatically download and save the required weights into the required folder. The script uses `curl`, that can be installed by simply typing on the bash /terminal's command line (Ubuntu 20.04):

```
$ sudo apt install curl
```

**Integration** In order to integrate YOLOv4 into the pipeline, the first step is the creation of a Darknet's `shared object` (.so) library starting from the source code. Then, in order to adapt the compiled code to be used from Python, a `ctypes` Python wrapper `wrapper.py` is created. This wrapper contains all the function redefinitions, types redefinitions and object redefinitions used in the main code to retrieve the bounding boxes of a specific tire in an image. For example, the `load_network()` function is used to load the weights into the CUDA cores and wait for a detection to start:

```
def load_network(config_file, data_file, weights, batch_size=1):
    """
    load model description and weights from config files
    args:
        config_file (str): path to .cfg model file
        data_file (str): path to .data model file
        weights (str): path to weights
    returns:
        network: trained model
        class_names
        class_colors
    """
    network = load_net_custom(
        config_file.encode("ascii"),
        weights.encode("ascii"), 0, batch_size)
    metadata = load_meta(data_file.encode("ascii"))
    class_names = [metadata.names[i].decode("ascii") for
      i in range(metadata.classes)]
    colors = class_colors(class_names)
    return network, class_names, colors
```

By using the developed wrapper, the YOLOv4 network can retrieve the two bounding boxes of the two lateral tires of the car that can be used to filter out the matched points that do not lay on them. This particular task is fundamental for the next reconstruction, since it guarantees that all the matches are on a coplanar object.

### SuperGlue

The SuperPoint+SuperGlue end-to-end network is used to extract and match the image pairs. In particular, the implementation uses the already trained weights of both the GNNs (i.e. graph neural networks) to perform the inferences and save persistently the matches as `.npz` files. Unfortunately, the researchers did not publish the code for the training process, so no fine-tuning technique has been conducted to optimize the final solution.

The original implementation of SuperGlue is then used by exploiting the `match_pairs.py` script and passing the configuration (indoor/outdoor weights,

maximum number of keypoints, minimum threshold, and so on). This configuration is stored alongside all the other configuration variables as a globally accessible `config.py` script. Here, the path variables are stored too, alongside the YOLOv4 configuration (e.g. weights file).

While YOLOv4 is retrieving the bounding boxes from an image pair, SuperGlue retrieves and saves the matches. With respect to YOLOv4 and its Darknet Python wrapper, SuperGlue is already written in Python using *Pytorch* as deep learning library[6], thus no wrapper is needed. Due to this, the script is called by a `subprocess.call()`. Since that the output pairs are stored in files, a custom function deserializes the matches and the keypoints and outputs the common matches between multiple pairs. Each pair file is formatted as follows:

```
>>> npz.files
['keypoints0', 'keypoints1', 'matches', 'match_confidence']
>>> npz['keypoints0'].shape
(382, 2)
>>> npz['keypoints1'].shape
(391, 2)
>>> npz['matches'].shape
(382,)
>>> np.sum(npz['matches']>-1)
115
>>> npz['match_confidence'].shape
(382,)
```

For each keypoint in `keypoints0`, the `matches` array indicates the index of the matching keypoint in `keypoints1`, or -1 if the keypoint is unmatched.

In order to compare the SuperGlue with a more classical approach, a SURF + FLANN features matching function is implemented too, that does the same thing as the GNNs, but in a very different way (i.e. without exploiting neural networks theory). In order to distinguish the two implemented matchers, two different functions are developed in the `utils.py` script:

- `run_surf(images, network)`, that accepts the loaded YOLOv4 network and the images and returns the common matches.

- `run_superglue(pairs_folder, network, images)`, that accepts the loaded network, the images and the folder containing the pairs info and returns the matches.

[6] https://pytorch.org/docs/stable/

### Main

The main program, represented by the `run.py` script, implements the already explained custom pipeline (see Fig. 4.1). In particular, it imports all the configuration from the `config.py` file and uses the implemented methods from the `utils.py` script. Furthermore, it loads the defined images in `images.txt` and `images-sg.txt`. The first text file contains the relative path of the images to analyze, one for each row:

```
images/1.jpg
images/2.jpg
images/3.jpg
```

While the second text file contains the same data in another format (this is done in order to adapt the features to SuperGlue):

```
1.jpg 2.jpg 0
1.jpg 3.jpg 0
```

Where the relative path is not used, only the names of the images organized as pair for each row with an ending 0.

The generic algorithm retrieves the images as pairs by taking the first image as the image given by the principal camera. For example, for $n$ images, the `images.txt` file would be:

```
images/1.jpg
images/2.jpg
images/3.jpg
...
images/n.jpg
```

And the `images-sg.txt`:

```
1.jpg 2.jpg 0
1.jpg 3.jpg 0
1.jpg 4.jpg 0
...
1.jpg n.jpg 0
```

Once that the two tires are recognized with YOLOv4 and SuperGlue has retrieved the matches, the matches are filtered based on the bounding boxes and then by using the RANSAC algorithm provided by the OpenCV library. Once that the main pipeline has the common matches, it creates the 3D correspondences by positioning the world coordinates as the first camera's local coordinates and solving the P3P problem of each camera and between each pair.

The final outcome of the algorithm is the position of the $n$ cameras, retrieved from the rotation and translation with respect to the real world coordinates (the first camera). Starting from $R$ (rotation matrix) and $t$ (translation vector), the point $C$ of the camera can be extracted as follows:

```
R = cv.Rodrigues(rvecs)[0]
C = -R.T.dot(t)
```

Where `rvecs` is the rotation vector given as output from the PnP function, that must be converted to a rotation matrix $R$. In particular, by using a math notation, the point is retrieved as follows:

$$C = -R^T \cdot t \tag{4.3}$$

Obviously, the point does not take into account a positive scale factor $\delta$ (impossible to obtain from a metric reconstruction without stereo cameras). The complete solution would be:

$$C = (-R^T \cdot t)\delta \tag{4.4}$$

### 4.3.4 OpenCV's pipeline

The second implemented pipeline exploits the standard algorithms and methods in order to perform the reconstruction and obtain the relative positions of the cameras. As shown in Fig. 4.2, the general flow operates in one or more stages, depending on the number of cameras (images) given as input to the pipe.

The minim number of views is obviously two. With two views, the first stage performs an initial reconstruction (i.e. initial cloud point solution). Afterwards, starting from the third one, the reconstruction uses only the second stage of the pipeline by trying to perform resectioning in order to localize the new keypoints on the new image. The code is separated from the custom pipeline and it is written in C++ exploiting the OpenCV library [18].

#### Initial reconstruction

The initial reconstruction is implemented in OpenCV through the `libmv` library[7]. In particular, the function that reconstruct the structure from two views is:

```
bool InitialReconstructionTwoViews(const Matches &matches,
                                   Matches::ImageID image1,
                                   Matches::ImageID image2,
                                   const Mat3 &K1,
                                   const Mat3 &K2,
                                   const Vec2u &image_size1,
                                   const Vec2u &image_size2,
                                   Reconstruction *recons);
```

[7]Library for Multiview Reconstruction, https://developer.blender.org/tag/libmv/

As features detection algorithm, ORB is used [19], while FLANN is the choice for features matching. This is done before calling the aforementioned function, where the matches are passed as input params.

At first, some default parameters are set (e.g. epipolar threshold and outliers probability) and the matches are converted into a matrix representation in order to better find the correct matches. Then, obviously, if the common points are less then 7, the method cannot proceed because of the requirements of the subsequent algorithm used for the Fundamental matrix $F$ estimation: the normalized eight-point algorithm. This method takes the previously saved params and the matches returning the estimated matrix. In order to retrieve only the inliers (since the interest is on the estimation of the relative motion), a filter is applied to the output of the estimation:

```
FundamentalFromCorrespondences7PointRobust(x0,x1,
                                           epipolar_threshold,
                                           &F, &feature_inliers,
                                           outliers_probability);
 // Only inliers are selected in order to estimate the relative
  motion
 Mat2X v0(2, feature_inliers.size());
 Mat2X v1(2, feature_inliers.size());
 size_t index_inlier = 0;
 for (size_t c = 0; c < feature_inliers.size(); ++c) {
   index_inlier = feature_inliers[c];
   v0.col(c) = x0.col(index_inlier);
   v1.col(c) = x1.col(index_inlier);
 }
```

From the Fundamental matrix, the essential matrix is recovered by using the two cameras intrinsic parameters represented by $K1$ and $K2$ (see [20], section 9.6):

$$E = K_2^T \times F \times K_1 \tag{4.5}$$

Finally, the motion from the Essential matrix and the correspondences is recovered. More in details, the rotation and translation are extracted by following a single value decomposition (SVD) in the `MotionFromEssential` function (see [20], result 9.19). Since that multiple solutions are possible, another function `MotionFromEssentialChooseSolution` chooses one of the four possible motion solutions from an essential matrix. It decides the right solution by checking that the triangulation of a match $x_1 - -x_2$ lies in front of the cameras ([20], section 9.6.3, Geometrical interpretation of the 4 solutions). The whole process is implemented in the function:

```
bool MotionFromEssentialAndCorrespondence(const Mat3 &E,
                                          const Mat3 &K1,
                                          const Vec2 &x1,
                                          const Mat3 &K2,
                                          const Vec2 &x2,
```

```
                                                Mat3 *R,
                                                Vec3 *t) {
  std::vector<Mat3> Rs;
  std::vector<Vec3> ts;
  MotionFromEssential(E, &Rs, &ts);
  int solution = MotionFromEssentialChooseSolution(Rs, ts, K1, x1,
   K2, x2);
  if (solution >= 0) {
    *R = Rs[solution];
    *t = ts[solution];
    return true;
  } else {
    return false;
  }
}
```

That returns true if the saved $R$ and $t$ are valid, based on the previously explained functions.

Once that the initial motion estimation is done, the algorithm tries to perform the initial point intersection by triangulation in order to reconstruct the points by using the following function:

```
uint PointStructureTriangulationCalibrated(const Matches &matches,
                CameraID image_id,
                size_t minimum_num_views,
                Reconstruction *reconstruction,
                vector<StructureID> *new_structures_ids = NULL);
```

That Reconstructs unreconstructed point tracks observed in the image `image_id` using theirs observations (matches) when the intrinsic parameters are known.

To be reconstructed, the tracks need to be viewed in more than `minimum_num_views` images. The method consists of the following steps:

1. Selects the tracks that haven't been already reconstructed.

2. Reconstructs the tracks into structures.

3. Remove outliers (points behind one camera or at infinity).

4. Creates and add them in reconstruction.

Finally, the number of structures reconstructed is returned and the list of triangulated points is modified accordingly (`*reconstruction`). If new points are added, the last step of the initial reconstruction is a metric bundle adjustment. The initial reconstruction is finished.

### Resectioning

If the total keyframes (i.e. images) in input are $n > 2$, after the initial reconstruction, the resectioning is performed $n-2$ times (see Fig. 4.2) by exploiting the following function:

```
bool IncrementalReconstructionKeyframes(const Matches &matches,
                        const vector<Matches::ImageID> &kframes,
                        const int first_keyframe_index,
                        const Mat3 &K,
                        const Vec2u &image_size,
                        Reconstruction *reconstruction,
                        int *keyframe_stopped_index)
```

That implements an euclidean resection algorithm with the previously reconstructed structures and new structures are estimated (by points triangulation). A bundle adjustment is performed on all the reconstruction each time a keyframe is localized.

In a final step, non-keyframes are localized using the resection method implemented in the following function:

```
bool ReconstructionNonKeyframes(const Matches &matches,
                                const Mat3 &K,
                                const Vec2u &image_size,
                                std::list<Reconstruction *> *
  reconstructions);
```

With an important specification on the input that the images must be ordered (i.e. like video frames). This last step estimates the pose of non already localized frames using the already reconstructed points by resection. It performs also a bundle adjustment when $X = 10$ new cameras are localized. The method automatically detect the reconstruction the frame may belongs.

### SfM

As already specified, all the previous methods were originally implemented in `libmv`. In order to use the library in C++ through OpenCV, the `sfm`'s OpenCV pipeline is implemented in the `reconstruct` method. Moreover, the third party library `viz` from OpenCV is used in order to output a simple 3D mapping of the reconstructed points alongside the cameras with their positions and orientations.

The code is listed in `sfm.cpp`. This main code takes as inputs the text file with all the absolute paths of the images to be reconstructed (ordered), the focal length $f$ and the image principal point $x$ coordinates and $y$ coordinates ($c_x$ and $c_y$). The method only works for projective cameras, since no affine models have been created.

After loading the specified images by looking at the file, the code starts the pipeline of Fig.4.2 by calling the `reconstruct` method:

```
reconstruct(images_paths, Rs_est, ts_est, K, points3d_estimated,
  is_projective);
```

That returns the cameras and the 3D points. The `viz` library is the configured and prepared in order to simulate the 3D scene:

```
viz::Viz3d window("Coordinate Frame");
window.setWindowSize(Size(600, 600));
window.setWindowPosition(Point(150, 150));
window.setBackgroundColor(); // black by default
```

While the `path` (the affine 3D transformation that represents a camera) is built using the translation and rotation vectors retrieved from the reconstruction:

```
cout << "Recovering cameras ... ";
vector<Affine3d> path;
for (size_t i = 0; i < Rs_est.size(); ++i)
   path.push_back(Affine3d(Rs_est[i], ts_est[i]));
cout << "[DONE]" << endl;
```

The positions of the estimated cameras are printed, while the clound points and the paths are passed to the viz library:

```
if (point_cloud_est.size() > 0)
{
  cout << "Rendering points   ... ";
  viz::WCloud cloud_widget(point_cloud_est, viz::Color::green());
  window.showWidget("point_cloud", cloud_widget);
  cout << "[DONE]" << endl;
}
else
{
  cout << "Cannot render points: Empty pointcloud" << endl;
}
if (path.size() > 0)
{
  cout << "Rendering Cameras  ... ";
  window.showWidget("cameras_frames_and_lines",
           viz::WTrajectory(path, viz::WTrajectory::BOTH,
                   0.1, viz::Color::green()));
  window.showWidget("cameras_frustums",
           viz::WTrajectoryFrustums(path, K, 0.1, viz::Color::
  yellow()));
  window.setViewerPose(path[0]);
  cout << "[DONE]" << endl;
}
else
{
  cout << "Cannot render the cameras: Empty path" << endl;
}
```

Finally, a window frame appears, centered on the first camera position. This window can be explored by the user in order to see the structure of the scene with the cloud point and the cameras correctly positioned and aligned.

# Chapter 5

# Results

## 5.1 Introduction

After the design and the implementation of the two solutions, a testing phase is required (with validation) in order to contextualize the results, confront them and evaluate their performances.

The test scenario has been developed according to the following specifications:

- The $n$ cameras from $n$ different positions are simulated with one camera (an Android smartphone) and $n$ photos from different positions.
- Each camera must "see" the same two lateral tires of the car (same side).
- All the pipelines estimate the positions with an unknown positive scaling factor $\delta$.
- The used camera must be previously calibrated.
- It is preferable that only one car is placed in the scenario. If more cars are visible, only the closest one is taken into account.

Before looking and comparing the results, some important words about the calibration process and the used datasets are required.

### 5.1.1 Initial calibration

In order to use the camera in the depicted scenario, an initial calibration for the intrinsic parameters (see 2.1.1) must be performed.

The calibration is performed by exploiting the *chessboard* pattern, a well-known and detectable object that can be used (with the help of OpenCV) to obtain an estimation of the $K$ matrix (i.e. the intrinsic matrix).

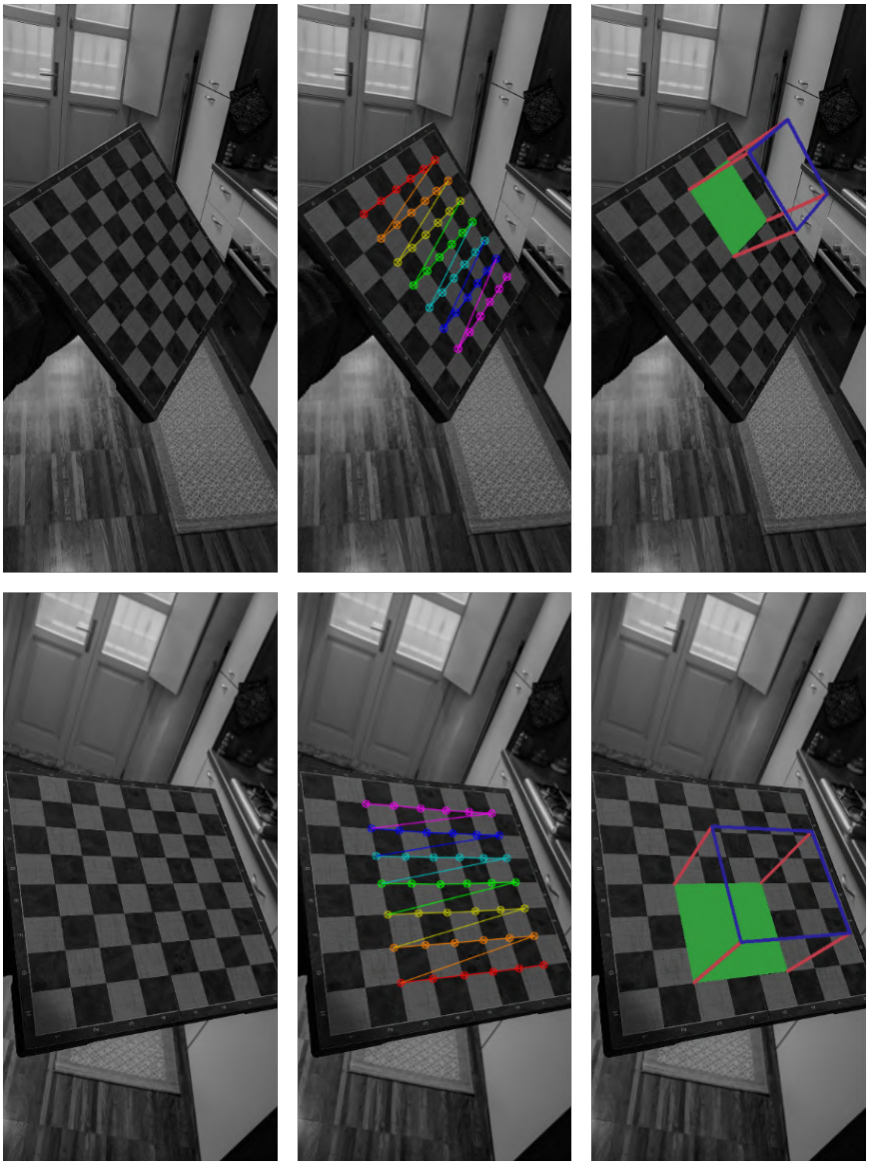


Figure 5.1. Chess board calibration process (first)

In order to achieve an initial calibration, a $8 \times 8$ chessboard is used. The Python scripts implemented for this purpose are `chess-board_calibration.py` and `chess-board_test.py`. As shown in the figures (Fig. 5.1 and Fig. 5.2), the initial photo (first on the left) is the input of the calibration process. The second column of images represents the output of the first script, that matches the chessboard corners ($6 \times 7$) with the 3D points $(0, 0, 0)$, $(1, 0, 0)$, $(1, 1, 0)$, $(1, 2, 0)$

$\ldots(2,0,0)$ $\ldots(5,6,0)$. The calibration script tries to perform a sub-pixel refinement too in order to iterate and find the sub-pixel accurate location of corners or radial saddle points as described in [21]. The calibration is then run by using standard algorithm and the output of the script is `camera.npz` with all the estimated parameters stored.

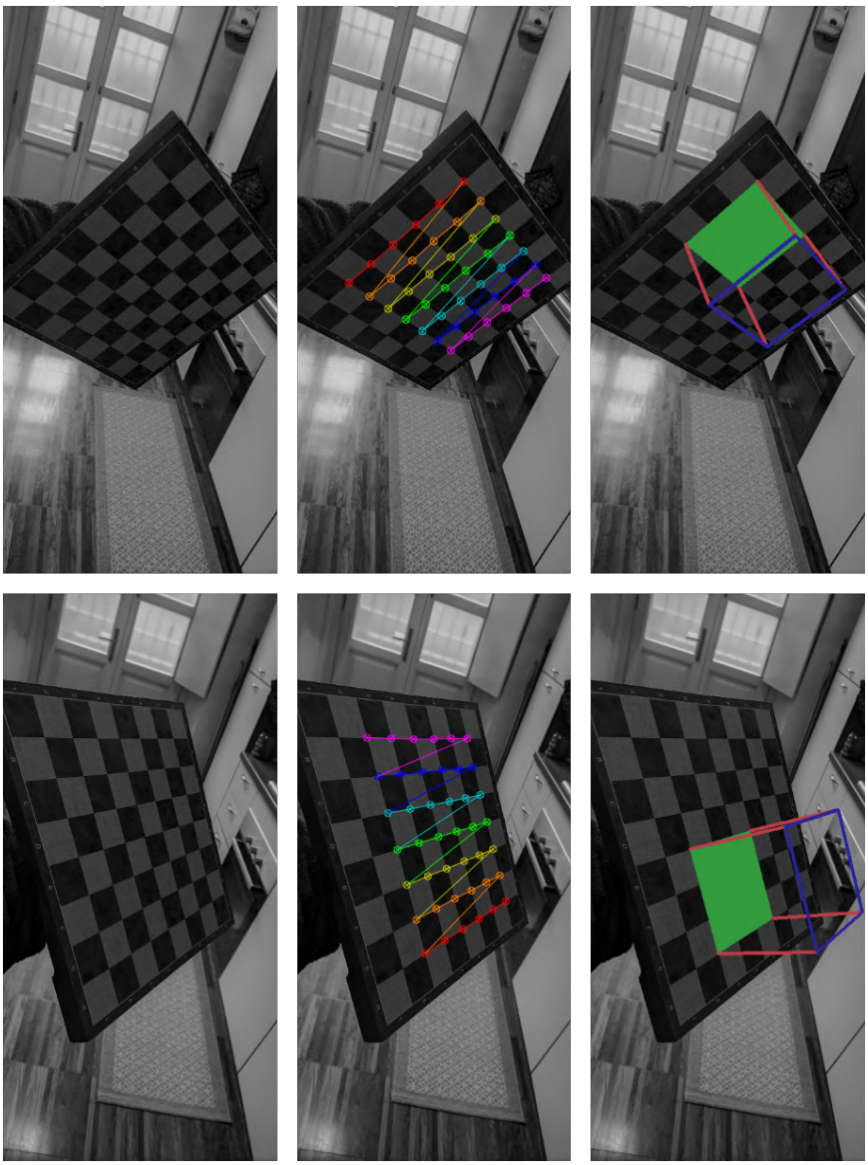


Figure 5.2. Chess board calibration process (second)

This file will be the input of the `chess-board_test.py` script, that uses the estimated matrices in order to draw a cube centered in $(0,0,0)$ of dimension 3 (i.e.

three chessboard squares). The output of the second script would be the third images to the right (Fig. 5.1, Fig. 5.2). The test script is a validation that is done in order to see if the calibration process was performed good enough. From the images, it is clear that it is not precise to the pixel, but really acceptable. Clearly, with much more images (100-1000) and different poses, the precision should grow.

Once that the parameters file is saved persistently, the matrix can be used in both the two pipelines.

### 5.1.2 Datasets

During the validation and the testing phase (apart from the Open Images subset for YOLOv4), only one dataset has been used: the Multi-View Car Dataset [22]. In particular, this dataset is used as initial test scenario for both the pipelines. It contains 20 sequences of cars as they rotate by 360 degrees (one image every 3-4 degrees) at it is best suited to retrieve some initial results before the final test on raw and new data. In Fig. 5.3 some images samples are shown.

Figure 5.3. Samples from the Multi-View Car Dataset

The camera parameters are fixed, so that no calibration is needed in order to use the dataset. Moreover, since that not every photos is strictly in compliance with the above-mentioned specifications (i.e. they do not capture the two side tires at the same time), the dataset is reduced to a subset.

In addition, some raw images (Fig. 5.4) were taken with the same Android camera from different views in order to explore the performances and precision of the two pipelines in a real-world application.

## 5.2 Custom pipeline

The custom pipeline has been tested by exploiting the available methods for features extraction/matching in order to retrieve the best combination so far (with and without GPU CUDA acceleration). In particular, the analyzed methods are:

Figure 5.4. Raw sample from the Android camera

- *SURF* for features extraction and *BF* (brute force) for features matching.
- *SIFT* for features extraction and *FLANN* for features matching.
- *SuperPoint* for features extraction and *SuperGlue* for features matching.

The important parameters to take into account are surely the general number of common matches (retrieved among $n$ images) and the number of common filtered matches after running YOLOv4 and RANSAC (a method can give as result many common matches, but poor matching on the tires of the car). Those parameters can estimate some experimental results in order to choose which features extraction/matching method is best suited for the real-world task.

In Tab. 5.1, the three possible algorithms are compared for different number of images (i.e. $n$ from 2 to 5 views). The comparison is performed by using the subset of Multi-View Car Dataset exploiting the images (Fig. 5.5 and Fig. 5.6):

- *panda_2.jpg* and *panda_4.jpg* for $n = 2$
- *panda_2.jpg*, *panda_4.jpg* and *panda_6.jpg* for $n = 3$
- *panda_2.jpg*, *panda_4.jpg*, *panda_6.jpg* and *panda_8.jpg* for $n = 4$
- *panda_2.jpg*, *panda_4.jpg*, *panda_6.jpg*, *panda_8.jpg* and *panda_10.jpg* for $n = 5$

Figure 5.5. *panda_2.jpg*, *panda_4.jpg* and *panda_6.jpg*

Figure 5.6. *panda_8.jpg* and *panda_10.jpg*

| | common matches | common filtered matches | $n$ |
|---|---|---|---|
| SIFT+FLANN | 268 | 21 | 2 |
| | 146 | 11 | 3 |
| | **96** | 6 | 4 |
| | **96** | 2 | 5 |
| SURF+BF | **463** | 20 | 2 |
| | **185** | 6 | 3 |
| | 80 | 3 | 4 |
| | 80 | 3 | 5 |
| SuperPoint + SuperGlue | 241 | **22** | 2 |
| | 140 | **15** | 3 |
| | 82 | **13** | 4 |
| | 78 | **12** | 5 |

Table 5.1. Performance comparison from $n$ number of views between different methods for features extraction and matching

From the retrieved results, it is clear that the classic methods have much more common matches on the entire image (SIFT+FLANN wins for 4 and 5 views). On the other hand, SuperPoint+SuperGlue has much more robust and filtered matches (on the two tires) as shown in Fig. 5.7.

So far, the SuperPoint+SuperGlue method is the more suited one. In order to retrieve the positions of the $n$ cameras, the entire pipeline is run, giving as output the positions and image itself with the pose estimation represented by a cube. The

Figure 5.7. Common filtered matches on *panda_2.jpg* with *panda_4.jpg* and *panda_6.jpg* by exploiting SuperPoint+SuperGlue

pipeline has some problems when the number of views is greater than 3 (sometimes only the first image pose is correctly estimated), since that the number of common matches drastically decreases.

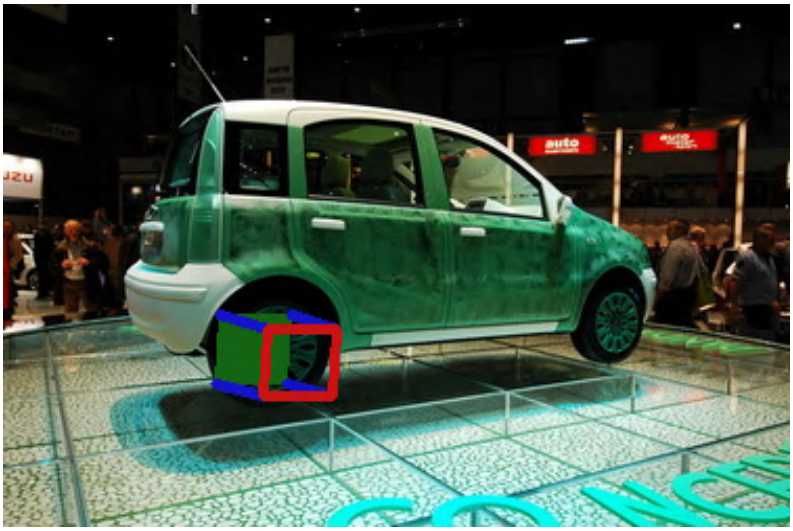


Figure 5.8. Pose estimation on *panda_2* with $n = 3$

As the number of matches decreases, the PnP method is less accurate. In the Fig. 5.8, the first view is correctly estimated, since that the drawn cube has the correct pose. It is true that the accuracy is not as good as expected. The cause

can be found on the poor matches and especially on the unknown dimensions (i.e. the distances of each point from the camera). Unfortunately, the used camera is a monocular one and it does not have the possibility to estimate it (like it is done with a stereo camera).

Obviously, in order to extensively test the pipeline, some other data has been fed to it. In particular, another test scenario is represented by some more raw photos taken directly from the Android camera (previously calibrated). The images (Fig. 5.4) have been tested with the same features matching/extraction algorithm as the previous dataset (Tab. 5.2)

| | common matches | common filtered matches | $n$ |
|---|---|---|---|
| SIFT+FLANN | 268 | 16 | 2 |
| | 134 | 7 | 3 |
| SURF+BF | **463** | **67** | 2 |
| | **194** | **29** | 3 |
| SuperPoint + SuperGlue | 241 | 26 | 2 |
| | 154 | 17 | 3 |

Table 5.2. Performance comparison from $n$ number of views between different methods for features extraction and matching on raw data

In this new scenario, the combination of SURF+BF gives the best results (Fig. 5.9), while the pose estimation from each cameras is still a little bit failing due to lacks of matches and dimensionality. Unfortunately, the estimation is not all failed. On the first image (i.e. the first view) the pose is correctly retrieved, but on the second one it fails (Fig. 5.10).

Figure 5.9. Common filtered matches on on the first camera from the $n = 3$ raw images

Figure 5.10. Pose estimation on the first and second view on raw data

## 5.3 OpenCV's pipeline

The OpenCV pipeline (Fig. 4.2) exploits epipolar geometry to obtain the camera positions from multiple views. As the only features extraction and matching algorithm, *ORB+BF* is used.

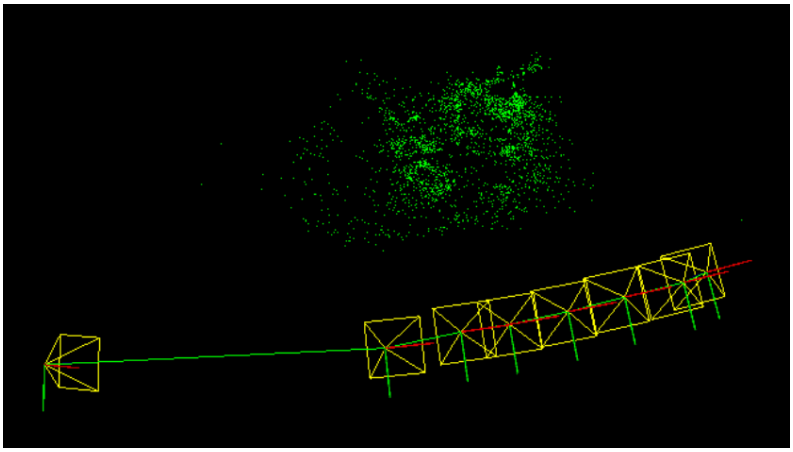


Figure 5.11. Cameras pose estimation with $n = 8$ on the Multi-View Car Dataset

Test have been run on the two datasets (i.e. the Multi-View Car Dataset and raw images). In particular, on the first one for $n = 2$ to $n = 8$ with really good estimations. For instance, taking as views `panda_2.jpg`, `panda_4.jpg` and `panda_6.jpg`, the estimation gives as output the 3D viz in Fig. 5.13.

Moreover, the OpenCV pipeline has gone even further with the number of views (i.e. cameras): eight cameras estimation. As shown in Fig. 5.11, the eight cameras represented by the eight images in Fig. 5.14 have been correctly placed and estimated.

Until now, the OpenCV pipeline performs better than the custom pipeline. A final test would be on the raw Android camera images that, for comparison, will

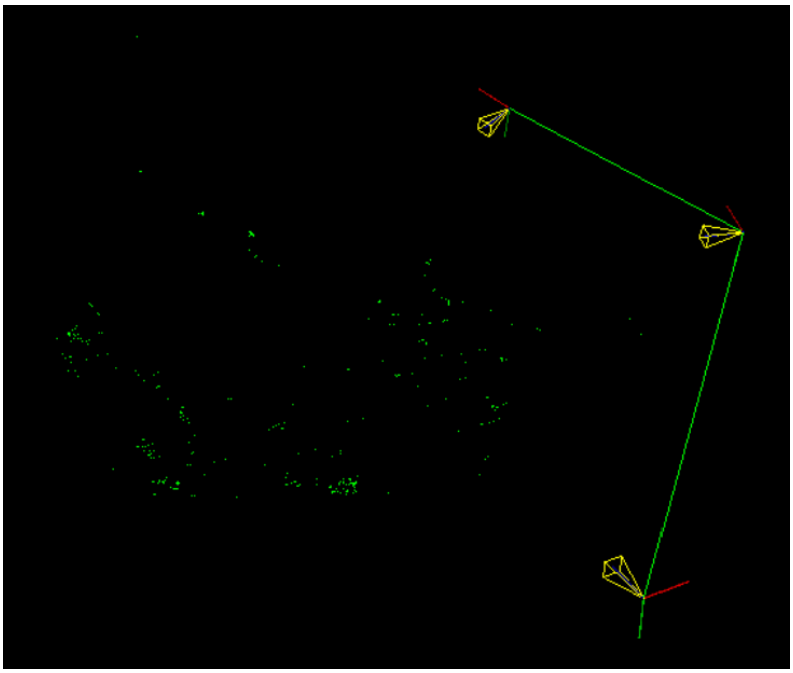
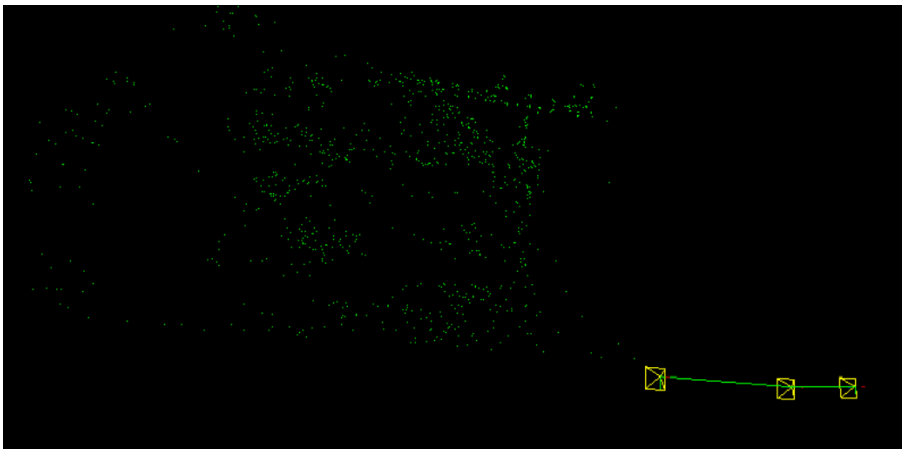


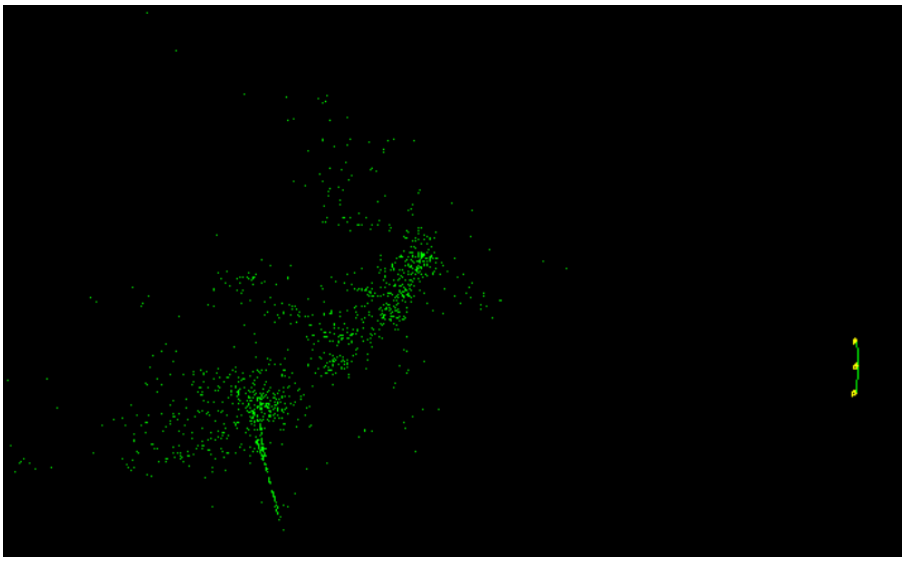
Figure 5.12. Cameras pose estimation with $n = 3$ on raw data taken from the Android camera

be the same used with the custom pipeline's tests. By exploiting $n = 3$ views represented by the first three images in Fig. 5.9, the pipeline correctly estimated the three camera's positions (Fig. 5.12).

Under those circumstances and in this particular scenario, the OpenCV pipeline seems to perform better. The truth is that it would perform better in any other scenarios, with some exceptions caused by the specific favorable custom pipeline features. The reliability of the epipolar geometry would be difficult to reproduce with deep learning techniques, but it surely can be improved.



(a)



(b)

Figure 5.13. Cameras pose estimation with $n = 3$ on the Multi-View Car Dataset

(a) panda_2

(b) panda_4

(c) panda_6

(d) panda_8

(e) panda_10

(f) panda_12

(g) panda_14

(h) panda_16

Figure 5.14. Images used for a $n = 8$ views reconstruction

# Chapter 6

# Conclusions

The two implemented pipelines represents two different approaches for the auto-calibration. It is clear that the OpenCV pipeline is more robust, since it exploits the epipolar geometry concepts. On the other hand, the custom pipeline tries to elude the epipoles by implementing a deep learning object detection filter and the coplanar property. Since that the custom pipeline relies on DL models, it can be too specific (i.e. the models performs well only in specific conditions).

The standard approach used in the OpenCV pipeline demonstrates that it is not true that "newer is better", because this experimental results proves the opposite. During the testing phase, some other minor checks have been made on other data, showing literally different behaviours for the custom pipeline, strongly dependent on the particular model of the car or on light conditions, hue, contrast and so on.

Moreover, the pipelines are particularly limited because of the monocular cameras, that does not provide depth information and automatically distance measurements. The analyzed scenarios is really restricted in terms of specifications and limitations, but that is not the objective of this thesis work. With this work, two possible approximate solutions are proposed for multiple camera pose estimation. Unfortunately, the performances are not the state-of-the-art, but the original goal relies on analyzing this particular kind of problems and reach a solution.

## 6.1 Future works

During the experimental results evaluation it has been proved that the epipolar geometry could lead to more accuracy when analyzing structure from motion scenarios. An interesting future work could be based on more state-of-the-art deep learning techniques that exploits the epipoles and the theory behind it. Moreover, some more advanced 3D object pose estimation models could be used with an adequate hardware like stereo cameras or LiDAR technologies (see 3.2.3).

There is a wide range of possibilities for future development and this work can

be seen as an exploring step in order to lay the foundations for them. Furthermore, with the help of hybrid approaches to the Perspective-n-point problem [14], the path should lead to a stable and reliable unified solution soon.

# Appendix A

# Build and install OpenCV

This appendix is devoted to the custom build and installation of *OpenCV v4.5.0* with all the necessary tools and dependencies on Ubuntu 20.04.

## A.1 Dependencies

Before compiling and installing, `cmake` must be installed on the machine.
From the terminal:

```
$ sudo apt-get update
$ sudo apt-get install cmake cmake-qt-gui
```

### A.1.1 Ceres solver

Ceres solver [23] is a large scale non-linear optimization library that can be used in some of the implemented algorithms in OpenCV. Before compiling it, some dependencies are required:
From the terminal:

```
$ sudo apt-get update
$ sudo apt-get install libeigen3-dev libgflags-dev \
    libgoogle-glog-dev libatlas-base-dev \
    libsuitesparse-dev
```

Then, the code must be downloaded and extracted:

```
$ cd $HOME
$ wget https://github.com/ceres-solver/ceres-solver/
   archive/1.14.0.tar.gz
$ tar xf ceres-solver-1.14.0.tar.gz
```

Finally, a folder for the build is created and the code is built and installed:

```
$ mkdir ceres-build
$ cd ceres-build && cmake ../ceres-solver-1.14.0 .
$ make -j4
$ make -j2 test && sudo make install
```

Now ceres-solver is finally installed and ready.

### A.1.2 CUDA

Another important dependency is the NVIDIA CUDA Sdk [24] that enables the GPU for general purpose applications in order to use the acceleration provided by NVIDIA's multiple cores.
From the terminal:

```
$ wget https://developer.download.nvidia.com/compute/cuda/repos/
   ubuntu2004/x86_64/cuda-ubuntu2004.pin
$ sudo mv cuda-ubuntu2004.pin /etc/apt/preferences.d/cuda-
   repository-pin-600
$ sudo apt-key adv --fetch-keys https://developer.download.nvidia.
   com/compute/cuda/repos/ubuntu2004/x86_64/7fa2af80.pub
$ sudo add-apt-repository \
   "deb https://developer.download.nvidia.com/compute/cuda/repos/
   ubuntu2004/x86_64/ /"
$ sudo apt-get update
$ sudo apt-get -y install cuda
```

The process is a little bit long and it requires some time. When it is finished, the CUDA Toolkit is finally available.

### A.1.3 VTK

The VTK library is required for the compilation and use of `viz` OpenCV's library, necessary in order to visualize the reconstructed cameras as outputs.
From the terminal:

```
$ sudo apt install libvtk7-dev libvtk7-qt-dev
```

## A.2 Build and install

OpenCV is widely available already compiled and ready-to-use. Unfortunately, some third-party libraries are required (available in OpenCV's `contrib` modules collection). In order to build OpenCV with those external libraries, CUDA and Ceres Solver are required.
First, the source code of both must be downloaded and extracted:

```
$ cd $HOME
$ wget https://github.com/opencv/opencv/archive/4.5.0.tar.gz
$ wget https://github.com/opencv/opencv_contrib/archive/4.5.0.tar.
   gz
$ tar xf 4.5.0.tar
$ tar xf 4.5.0.tar.1
```

In order to configure OpenCV, the `WITH_CUDA` and `OPENCV_ENABLE_NON_FREE` flags must be set. Moreover, the `OPENCV_EXTRA_MODULES_PATH` has to be set with the path of the `modules` folder in the `opencv_contrib` folder.
Finally, the build folder is created and OpenCV is build and installed:

```
$ cd $HOME/opencv-4.5.0
$ mkdir build && cd build
$ cmake ../ .
$ make -j4
$ make -j2 test && sudo make install
```

To check that all went well, from the terminal:

```
$ opencv_version
4.5.0
```

# Bibliography


[1] Martin A. Fischler and Robert C. Bolles. «Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography». In: *Readings in Computer Vision* (1987), 726–740. DOI: 10.1016/b978-0-08-051581-6.50070-2.

[2] H.C. Longuet-Higgins. «A computer algorithm for reconstructing a scene from two projections». In: *Readings in Computer Vision*. Ed. by Martin A. Fischler and Oscar Firschein. San Francisco (CA): Morgan Kaufmann, 1987, pp. 61–62. ISBN: 978-0-08-051581-6. DOI: https://doi.org/10.1016/B978-0-08-051581-6.50012-X. URL: http://www.sciencedirect.com/science/article/pii/B978008051581650012X.

[3] R. I. Hartley. «In defense of the eight-point algorithm». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 19.6 (1997), pp. 580–593. DOI: 10.1109/34.601246.

[4] R. Tsai. «A versatile camera calibration technique for high-accuracy 3D machine vision metrology using off-the-shelf TV cameras and lenses». In: *IEEE J. Robotics Autom.* 3 (1987), pp. 323–344.

[5] Shaoqing Ren et al. «Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks». In: *CoRR* abs/1506.01497 (2015). arXiv: 1506.01497. URL: http://arxiv.org/abs/1506.01497.

[6] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. *YOLOv4: Optimal Speed and Accuracy of Object Detection*. 2020. arXiv: 2004.10934 [cs.CV].

[7] D. G. Lowe. «Object recognition from local scale-invariant features». In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. Vol. 2. 1999, 1150–1157 vol.2. DOI: 10.1109/ICCV.1999.790410.

[8] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. «SURF: Speeded Up Robust Features». In: *Computer Vision – ECCV 2006*. Ed. by Aleš Leonardis, Horst Bischof, and Axel Pinz. Springer Berlin Heidelberg, 2006. ISBN: 978-3-540-33833-8.

[9] Marius Muja and David Lowe. «Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration.» In: vol. 1. Jan. 2009, pp. 331–340.

[10] Paul-Edouard Sarlin et al. «SuperGlue: Learning Feature Matching with Graph Neural Networks». In: *CVPR*. 2020. URL: https://arxiv.org/abs/1911.11763.

[11] Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. *SuperPoint: Self-Supervised Interest Point Detection and Description*. 2018. arXiv: 1712.07629 [cs.CV].

[12] Xiao-Shan Gao et al. «Complete solution classification for the perspective-three-point problem». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 25.8 (2003), pp. 930–943. DOI: 10.1109/TPAMI.2003.1217599.

[13] Mikael Persson and Klas Nordberg. «Lambda Twist: An Accurate Fast Robust Perspective Three Point (P3P) Solver». In: *Proceedings of the European Conference on Computer Vision (ECCV)*. Sept. 2018.

[14] Roy Sheffer and Ami Wiesel. *PnP-Net: A hybrid Perspective-n-Point Network*. 2020. arXiv: 2003.04626 [cs.CV].

[15] Joseph Redmon. *Darknet: Open Source Neural Networks in C*. http://pjreddie.com/darknet/. 2013–2016.

[16] Alina Kuznetsova et al. «The Open Images Dataset V4: Unified image classification, object detection, and visual relationship detection at scale». In: *arXiv:1811.00982* (2018).

[17] Angelo Vittorio. *Toolkit to download and visualize single or multiple classes from the huge Open Images v4 dataset*. https://github.com/EscVM/OIDv4_ToolKit. 2018.

[18] G. Bradski. «The OpenCV Library». In: *Dr. Dobb's Journal of Software Tools* (2000).

[19] E. Rublee et al. «ORB: An efficient alternative to SIFT or SURF». In: *2011 International Conference on Computer Vision*. 2011, pp. 2564–2571. DOI: 10.1109/ICCV.2011.6126544.

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

[21] W. Förstner et al. "A fast operator for detection and precise location of distinct points, corners and centres of circular features". In: *ISPRS Intercommission Workshop*, June 1987.

[22] Mustafa Özuysal, Vincent Lepetit, and Pascal Fua. «Pose Estimation for Category Specific Multiview Object Localization». In: *Cvpr: 2009 Ieee Conference On Computer Vision And Pattern Recognition, Vols 1-4*. IEEE Conference on Computer Vision and Pattern Recognition (2009), pp. 778–785. DOI: 10.1109/CVPR.2009.5206633.

[23] Sameer Agarwal, Keir Mierle, and Others. *Ceres Solver*. http://ceres-solver.org.

[24] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. *CUDA, release: 11.1.X*. 2020. URL: https://developer.nvidia.com/cuda-toolkit.