



Escuela  
Politécnica  
Superior

# Map Slammer



Robotics Engineering Degree

## End-of-Degree thesis

Author:

José Miguel Torres Cámara

Mentor/s:

Miguel Ángel Cazorla Quevedo

Francisco Gómez Donoso

July 2019



Universitat d'Alacant  
Universidad de Alicante



# Map Slammer

---

Densifying Scattered KSLAM 3D Maps with Estimated Depth

**Author**

José Miguel Torres Cámara

**Mentor/s**

Miguel Ángel Cazorla Quevedo

*Ciencias de la Computación e Inteligencia Artificial (CCIA)*

Francisco Gómez Donoso

*Ciencias de la Computación e Inteligencia Artificial (CCIA)*



Robotics Engineering Degree



Escuela  
Politécnica  
Superior



Universitat d'Alacant  
Universidad de Alicante

ALICANTE, July 2019



## Preamble

There are a range of small-size robots that cannot afford to mount a three-dimensional sensor due to energy, size or power limitations. However, the best localisation and mapping algorithms and object recognition methods rely on a three-dimensional representation of the environment to provide enhanced capabilities.

Thus, in this work, a method to create a dense three-dimensional representation of the environment by fusing the output of a keyframe-based SLAM (KSLAM) algorithm with predicted point clouds is proposed. It will be demonstrated with quantitative and qualitative results the advantages of this method, focusing in three different measures: localisation accuracy, densification capabilities and accuracy of the resultant three-dimensional map.

This work has been supported by the Spanish Government TIN2016-76515R Grant, supported with European Regional Development Fund (ERDF) funds. I have been supported by a Spanish Government grant for cooperating in research tasks ID 998142. Part of this work has been submitted to the ROBOT'19 conference in O Porto, Portugal (Torres-Camara et al., 2019).



# Acknowledgements

I would like to start by thanking the whole *RoViT* research group for the support and help I have received from them during all the project. They were always there to propose alternatives and to help me with the problems I came across. They have also shown me one of the bests work environments I have seen and have guided me during my introduction to the research world.

On the other hand, I arrived here beside all my classmates and friends, learning from each others and growing together, generating links that I hope will last forever. Each one of us has collaborated to generate the environment we all enjoyed and took profit from.

Finally, this work, and my whole stay in the University could have not been possible without the constant support of my family and the sacrifices they have done in order to provide me with the best possible environment I could have ever had. They, specially my parents, have always been there, giving me pieces of advice and points of view on each complex situation or difficulty I have ever had. This has helped me to become the person I am and to build the path I am proudly following.



*To my family, friends and classmates.  
I am who I am because all of you.*



*Sometimes you want to give up.  
But if you stick with it,  
you are going to be rewarded.*

Jimi Hendrix.



# Contents

<b>1. Introduction</b>	<b>1</b>
<b>2. State of the Art</b>	<b>3</b>
2.1. KSLAM methods . . . . .	3
2.2. Depth estimation . . . . .	4
2.3. Scattered maps densification . . . . .	5
<b>3. Objectives</b>	<b>7</b>
<b>4. Methodology</b>	<b>9</b>
4.1. Computer vision . . . . .	9
4.1.1. 2D vision and image registration . . . . .	12
4.1.2. 3D vision and pointclouds . . . . .	14
4.2. SLAM . . . . .	17
4.3. Deep Learning . . . . .	19
4.3.1. Convolutional Neural Networks . . . . .	20
4.3.2. DeMoN . . . . .	22
<b>5. Approach</b>	<b>27</b>
5.1. Integration . . . . .	29
5.2. KSLAM for computing the 3D support points . . . . .	30
5.3. Keypoints reprojection . . . . .	31
5.4. Depth estimation for generating the pointclouds . . . . .	33
5.5. Transformation obtaining and applying . . . . .	35
5.6. Refinement . . . . .	36
<b>6. Experimentation, Results and Discussion</b>	<b>37</b>
6.1. Datasets description . . . . .	37
6.2. Localisation accuracy benchmark . . . . .	37
6.3. Densification capabilities test . . . . .	38
6.4. Accuracy of the returned 3D map . . . . .	40
<b>7. Conclusions and Future Work</b>	<b>43</b>
<b>References</b>	<b>45</b>
<b>Acronyms and abbreviations list</b>	<b>49</b>
<b>A. Annexed I. ORB-SLAM2 usage tutorial</b>	<b>51</b>

<b>B. Annexed II. Summary</b>	<b>55</b>
B.1. Motivation and concept . . . . .	55
B.2. Approach . . . . .	55
B.3. Results . . . . .	55
<b>C. Annexed III. Resumen en Español</b>	<b>57</b>
C.1. Motivación y concepto . . . . .	57
C.2. Desarrollo . . . . .	57
C.3. Resultados . . . . .	57

# List of Figures

2.1. Sample CNN-SLAM output . . . . .	6
2.2. Sample semi-dense reconstruction . . . . .	6
4.1. 2D (u, v) projection of a 3D (X, Y, Z) point using the intrinsic parameters. . . . .	10
4.2. Diagram of a pinhole camera. . . . .	10
4.3. Representation of the epipolar lines in a stereoscopic setup. . . . .	11
4.4. Effects of radial distortion. . . . .	11
4.5. Representation of the traditional pipeline in image registration. . . . .	12
4.6. Example of image registration. . . . .	12
4.7. Example of feature matching. . . . .	14
4.8. A pointcloud before (left) and after (right) applying a Voxel Grid filter. . . . .	17
4.9. A pointcloud before (top) and after (bottom) being filtered by SOR. . . . .	18
4.10. Representation of the uncertainty increase of the pose (shaded grey) and perceptions (red) of a mobile robot. . . . .	19
4.11. Graphical representation of a neuron. . . . .	20
4.12. Graphical representation of a fully connected neural network. . . . .	21
4.13. Convolution computation. . . . .	22
4.14. Convolutional Neural Network representation. . . . .	22
4.15. Convolutional Neural Network representation. . . . .	23
4.16. Convolutional Neural Network representation. . . . .	23
4.17. Qualitative examples of the performance of the iterative net (left) and the refinement one (right), in comparison with the groundtruth (GT). . . . .	24
4.18. Qualitative comparison between the DeMoN result and the ones archived by other methods (top) and the groundtruth (GT) using various datasets (right). . . . .	25
5.1. Pipeline of the proposal. This pipeline is looped to finally build a full 3D map of the environment. . . . .	28
5.2. Diagram representing the information flow originated by the bashscript. . . . .	31
5.3. Extracted keypoints (green) vs reprojected ones (red). . . . .	32
5.4. Diagram representing the execution flow of the reprojection step. . . . .	34
5.5. Diagram representing the execution flow of the depth estimation step. . . . .	35
5.6. Diagram representing the execution flow of the transformation step. . . . .	36
6.1. Ground truth (black) vs ORB-SLAM2 (O-S2) (blue left) and the obtained (blue right) trajectories. The distances are shown in red. The representations (from left to right) correspond to fr1/xyz, fr1/desk and fr2/desk. . . . .	39
6.2. O-S2 scattered output (left) vs my dense one (right). This results correspond to the fr2/xyz sequence. . . . .	40
6.3. The method's output with Red-Green-Blue (RGB) information (from fr2/xyz). . . . .	40

7.1. Output of the method using an own prepared challenging sequence, seen from the side (left) and with perspective (right). . . . .	43
A.1. Monocular (left) and RGB-D (right) cameras in Pepper. . . . .	51

# List of Tables

5.1.	Internal parameters and distortion coefficients for the used groups of datasets.	33
5.2.	Percentage of used keyframes and RANSAC threshold. . . . .	36
6.1.	Absolute average trajectory error and average relative pose error achieved by O-S2 and this proposal. . . . .	38
6.2.	Averaged count of points across all the frames of each sequence of the data set for O-S2 and this method. . . . .	38
6.3.	Mean distance between the nearest neighbour of each point of the produced 3D representations to the ground truth representation. . . . .	41



# Listings

A.1. ROS master configuration examples . . . . .	52
A.2. boot_config.json file extract . . . . .	52
A.3. Pepper bringup command . . . . .	53
A.4. Camera calibration node launching command . . . . .	53
A.5. Monocular ORB-SLAM2 launching command . . . . .	53



# 1. Introduction

One of the key features of any robot is the ability to localise itself in unknown environments as well as to build the corresponding map. This problem is known as Simultaneous Localisation And Mapping (SLAM) and is one of the main state of the art methodologies to provide a mobile robot with navigation capabilities without prior knowledge. This is not an easy problem because, in order to create a good map, an accurate localisation is needed. In the same way, to perform accurate localisation tasks, a precise map is needed.

There are several approaches to solve the SLAM problem, but the most accurate methods to perform this rely on three-dimensional data inputs. Usually, this kind of data is provided either by a LIght Detection And Ranging (LIDAR) sensor, by a stereoscopic camera or by an RGB-Depth (RGB-D) camera. These sensors are commonly mounted on mid-size robots and larger ones, like the Turtlebot and the Pepper robots, some bigger Autonomous Ground Vehicle (AGV) or even self-driving cars. However, small-size robots do not provide them. There are many reasons that preclude three-dimensional sensors in robots with reduced dimensions. For instance, size and weight of these sensors could interfere in the movement of the robot or they could drain their batteries soon enough. Nonetheless, this kind of robots are usually equipped with a regular RGB camera that does not show the mentioned drawbacks and is less expensive.

Some recent approaches, under the KSLAM methodology, rely in 2D landmark detection to perform localisation and tracking. However, the KSLAM methods based in RGB cameras, whilst performing accurately, do not provide a dense map of the environment so they can not benefit from the navigation methods that takes advantage of dense 3D data, since they lack spatial information to ensure an obstacle-free path.

The small-sized robots are intended to be deployed in locations hard to reach for greater robots and are mainly used for inspection. For instance, they are commonly used for evaluating the state of the gas and water pipelines (Raziq Asyraf Md Zin et al., 2012; Afiqah Binti Haji Yahya et al., 2014), or even the electric power lines (Deng et al., 2014). They are also used in rescue tasks (Linder et al., 2010) due to their ability to get into narrow paths under collapsed buildings. The main drawback of these robots is that their high-level control and planning are usually based in an expert teleoperating them.

In this work, the proposal is a method able to generate dense maps using only monocular RGB cameras, making possible to generate them without the need of three-dimensional sensors and, thus, with small-size robots. This will make it possible for them to take advantage of dense maps in previously unknown environments (such as motion planning and proper obstacle avoidance). Furthermore, it would also be possible to generate human-interpretable maps of zones with difficult access or in which there is some kind of hazard in rescue or exploration tasks.

The proposal uses the scattered 3D points provided by a state-of-the-art KSLAM method as support to correctly place 3D pointclouds of the environment generated by depthmaps

estimated with a Deep Learning-based algorithm. The designed pipeline only takes pairs of color images as input and returns a dense three-dimensional map. This approach can be implemented in any robot equipped with a color camera to enable localisation and dense three-dimensional map generation of the environment, which are far more easy to understand for human agents and, as mentioned before, it provides advantages regarding path planning and object recognition.

The rest of the document is organised as follows. First, relevant works are reviewed in Chapter 2, where the recent advances of some related fields of study are also narrated. Next, in Chapter 3, the main purposes of this work are exposed, both from a general personal point of view regarding which abilities I wanted to improve and an specific technical one, commenting the particular target results and the steps to be followed. Along Chapter 4, I expose the fields, technologies, methods and implementations used in this work. Then, the approach is thoroughly explained in Chapter 5. Chapter 6 describes the experiments that have been carried out to validate the method and the corresponding discussion of the results. Finally, the conclusions of this work and future research directions are drawn in Chapter 7.

## 2. State of the Art

In this Chapter, the main contributions to solve the SLAM problem and depth prediction from color images are reviewed. Also, some recent approaches regarding map densification are mentioned.

### 2.1. KSLAM methods

The solutions based on keyframes are becoming the most common and efficient approaches for building a monocular visual SLAM system at the expense of filter-based methods (Younes et al., 2017). In Strasdat et al. (2010), it was demonstrated that keyframe-based family of methods outperforms filter-based one.

The main idea of the keyframe-based methods is splitting the camera tracking and the mapping in parallel tasks, as originally presented in Parallel Tracking And Mapping (PTAM) (Klein & Murray, 2007). It performed well in real time for small environments, and was used for augmented reality applications. For this work, I will focus in landmark-based systems. These kind of systems takes advantage of keypoint detection and matching alongside subsequent frames in order to estimate the camera motion (and position).

At the beginning, these methods use a visual initialisation module to establish an initial 3D map and the camera pose. When the system captures a new frame, the data association module guesses a new pose using the information taken from previous frames, establishing associations with the three-dimensional map. Then, an error vector between the matches is calculated and minimised using the pose optimisation module. If the minimisation fails, they usually take advantage of different techniques to recover from this error. In the case of normal frames, the pipeline ends with this step. If the frame was selected as key frame, the system looks for landmarks, triangulates their positions and expands the resulting map. In parallel, a task of map maintenance is running to detect loop closures and minimise their errors.

In order to infer the new pose of the camera, the association module locates 2D features on the image and establish a search window around their locations to find the correspondence on the previous images. Each feature is associated with a descriptor to measure the similarity between them. Several descriptors have been used, such as low level ones, as *Sum of Squares Differences (SSD)*, *Zero-Mean SSD* and *Normalised Cross Correlation (NCC)* (Hisham et al., 2015), or high level ones, as *SIFT* (Lowe, 2004), *ORB* (Rublee et al., 2011) or *SURF* (Bay et al., 2006). This step requires fast matching structures to ensure real time performance.

For the visual initialisation, the most common approaches do not use a known position of the camera with respect to a plane, and use the methods proposed in Longuet-Higgins (1981) to remove the depth from the problem, employing the essential and homography matrices. The side effect of this fact is that the reconstructed 3D scenes are scaled by a unknown factor. The depth at which each keypoint is located is initialised randomly with values of

large variance and it is updated in a looping process until this variance converges in successive frames.

*ORB-SLAM (O-S)* (Mur-Artal et al., 2015) is an state-of-the-art monocular keyframe-based SLAM technique that uses ORB keypoints and descriptors to perform the association step. It extracts corners using 8 pyramid levels over the entire image and dividing them into cells to calculate the descriptor. Then, this method discretises the descriptor into a bag of words (using a given vocabulary) to speed up the feature matching. The viewpoint could be an issue for the description, so *O-S* chooses the descriptors from the keyframe with the slight viewpoint difference with the current frame.

To estimate the camera pose, *O-S* considers that the camera moves with a constant speed, and detects abrupt motions if the number of matched features goes under a threshold. If this kind of motion is detected, the map points are projected onto the current frame and matched with the current descriptors. It is important to remark that it defines a local map with the features of the key frames near the current frame, so it allows to carry out real-time processing. As an output, it generates a set of 3D camera poses. For this work, it was modified in order to also get the 3D scattered map and the 2D keypoints alongside the frame in which they appear, as will be explained in Section 5.2.

## 2.2. Depth estimation

Regarding depth estimation, in recent years, many deep learning approaches have appeared to estimate depth from monocular images, using an end-to-end architecture. Additionally, some of them perform motion estimation too, so they can be suitable to solve the pose estimation problem.

In 2014, David Eigen published one of the state-of-the-art methods in this field (Eigen et al., 2014). First, it uses a network that predict the depth from the monocular frame in a rough-scale. Later, this prediction is refined passing local regions to another network specialised in fine details.

In 2016, Iro Laina presented another interesting research in this area (Laina et al., 2016). It presents a single fully convolutional residual network that carries out the depth prediction in a more efficient manner, combining convolutions with the upsampling of feature maps. Furthermore, their approach did not rely in any post-processing techniques, worked in real time and contains fewer parameters (while archiving better results) than the state of the art in its time. This work archived a real-time working model that was able to estimate a depth map in only 55 ms of a single image. This result can be decreased by computing images in batches. As an example, in the paper it is mentioned that, when using a batch of 16 images, the required time to process each one is only 14 ms.

However, single-image methods tend to have problems to estimate the depth when dealing with unseen types of images. A detailed review on the current state of the art is done in Bhoi (2019), in which five papers that try to solve this problem (monocular depth estimation) are reviewed. In it, it is shown a clear tendency to the usage of a deep Convolutional Neural Network (CNN) as the network architecture, trained using supervised techniques. In general lines, this problem started using multi-scale features (in order to learn the structure of the data) and scale-invariant loss functions. Then, the usage of Conditional Random Field (CRF) was introduced to fuse the multi-scale features. Later on, encoder-decoders started to be used.

---

Finally, these approaches started to leave behind the monocular base in pro of start using two images as an input, which allowed, among other things, disparity measurements.

In 2017, Depth and Motion Network (DeMoN) (Ummenhofer et al., 2017) proposed a method to profit the stereopsis. It features a CNN to estimate the depth and the camera motion from a subsequent pair of images, similar to the *Structure from Motion (SfM)* technique, but with a Machine Learning-based system. This proposal calculates the dense correspondences, the depth and the camera motion between two frames using a single network. This approach has demonstrated that outperforms the reconstruction of *SfM* with two frames, so the dense representation could be suitable to generate 3D maps in combination with SLAM. Nonetheless, this approach renders the predictions in an arbitrary scale. The scale is not even consistent between two subsequent predictions.

### 2.3. Scattered maps densification

Given the recent advances regarding monocular depth estimation, some approaches trying to apply it to several fields have appeared. One of them is Tateno et al. (2017). They fused it with monocular visual SLAM using the sparse map produced by it as an initial guess for a deep CNN that performed the depth estimation. However, this work is focused in solving the scale ambiguity in monocular SLAM and in improving the robustness of this methods when exposed to environments with low textures. At the same time, by using the output of a monocular Visual SLAM (VSLAM) algorithm, they also archive to increase the robustness of their depth estimation method when working with new data. A sample output of this method is shown in Figure 2.1.

Furthermore, some approaches that are not Deep Learning-based has also been proposed. This is the case of Mur-Artal & Tardos (2015). In their job, they try to demonstrate the potential of feature-based SLAM. They did this by developing a method that provided with more robust and accurate reconstruction with comparable or even better densification capabilities of other state-of-the-art methods of its time. This work makes use of a combination of epipolar geometry and statistics to archive semi-dense reconstructions of the environment, as the one shown in Figure 2.2.



**Figure 2.1:** Sample CNN-SLAM output.

**Source:** Tateno et al. (2017)



**Figure 2.2:** Sample semi-dense reconstruction.

**Source:** Mur-Artal & Tardos (2015)

---

### 3. Objectives

The main objective of this work is the development of a method to build dense 3D maps using only a sequence of monocular RGB frames fusing a KSLAM method and Deep Learning (DL) techniques. Furthermore, I got involved in this project in order to:

- Increase my knowledge and understanding about SLAM methods and researching the state of the art of this technology, as well as the open problems.
- Go in depth in actual state of the art KSLAM methods (O-S and O-S2). This includes gathering experience not only in their theoretical bases but also in their versions and installation and launching processes.
- Learn how to apply the KSLAM method using a real mobile robot (Pepper), in order to see the challenges it suppose and its disadvantages and imperfections in comparison to the usage of datasets.
- Analyse some popular data-sets focused in testing VSLAM methods.
- Delve into Deep Learning and CNNs, with a practical perspective with the purpose of being able to use it as a tool and slightly modify other people work to complement mine.
- Design an innovative way to generate dense maps using small robots and simple hardware.
- Increase my programming and technology integration abilities.
- Gather more experience regarding GNU/Linux-based developing environments.
- Become familiar with the research work-flow and the scientific method.
- Get better in documenting my work and reporting it both in a written and oral manner.

The specific objective of the project is to make use of the points of the sparse map provided by O-S2 in order to, using DL techniques to estimate depthmaps using monocular images, build dense and accurate three-dimensional maps without neither the need of 3D cameras or the traditional pipeline for registering pointclouds.



## 4. Methodology

### 4.1. Computer vision

As analysed by Jirbandey (2018), computer vision has been a fast-growing field since a long time ago. It started to get enough relevance to be considered an independent field of study back in the 1960s. Its aim was always to mimic the human capacity to extract information from our environment by observing it using an automated process.

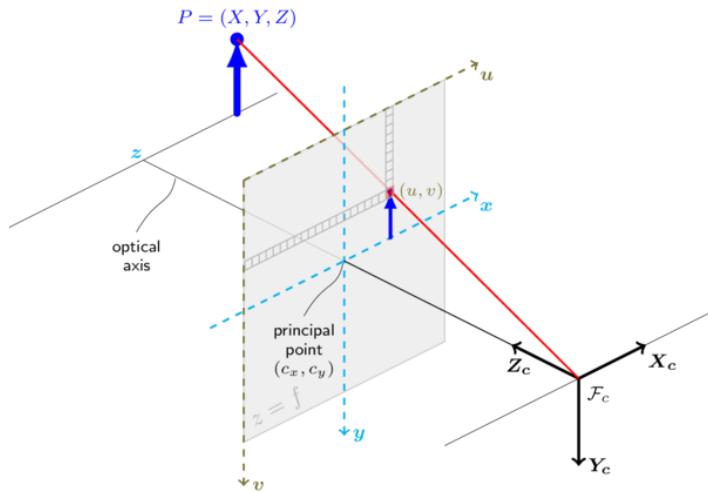
The traditional approach involved different processes that used neighbourhoods, intensity, escalations and sliding windows alongside other common resources to get information about the images, as keypoints and descriptors, to find correspondences between different frames or borders and corners to distinguish one objects from others.

As the computer vision field advanced, new techniques were researched and developed in order to get not only 2D but also 3D information as well. To do so, the researchers based their work in the projection process that transformed the 3D environment in the 2D representations commonly known as *photographs*. This projection is often approximated by using the the intrinsic parameters of the camera that is being used and the pinhole camera model. If a lens has to be taken in account, the distortion coefficients also have to be used.

The intrinsic camera parameters establish how the 3D to 2D projection is done. They are the focal length ( $f$  measured in pixels in  $f_x = F/p_x$  and  $f_y = F/p_y$ ), the optical centre principal point ( $c_x$  and  $c_y$ , in pixels) and the skew ( $s$ , which will be 0 always that the image axes are perpendicular, that is the most common case). The parameter meaning can be more clearly appreciated in the Figure 4.1. All these values are often represented as the matrix  $K$  (Equation 4.1). This information is more detailed in OpenCV-Documentation (n.d.).

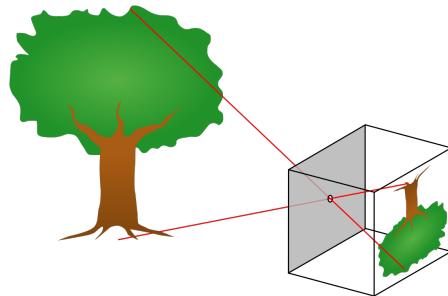
$$K = \begin{pmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \quad (4.1)$$

The pinhole camera model is the mathematical relationship between the 3D points of the environment and the 2D ones resulting from the projection using a camera obscura-like device. As mentioned in Wikipedia-contributors (2019a), the first existing references about this optical phenomenon are found in Chinese Mozi writings (circa 500 BC) and the Aristotelian Problems (circa 300 BC – 600 AD). The demonstrations of this effect consist in a light-proof box with a reduced size hole on it. This will make the light reflected by the objects outside to be projected upside-down inside the *camera*, as represented in the Figure 4.2. The projection is done using the Equation 4.2, based in the intrinsic parameters of the used camera.



**Figure 4.1:** 2D ( $u, v$ ) projection of a 3D ( $X, Y, Z$ ) point using the intrinsic parameters.

**Source:** OpenCV docs - Camera calibration and 3D reconstruction



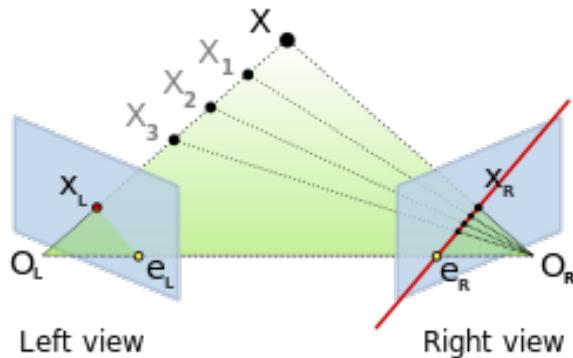
**Figure 4.2:** Diagram of a pinhole camera.

**Source:** Wikipedia - Pinhole camera model, DrBob (original); Pbrosks13 (redraw)

$$P_{2D} = K \begin{pmatrix} I & 0 \end{pmatrix} P_{3D} = \begin{pmatrix} f_x & s & c_x & 0 \\ 0 & f_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_{3D} \\ y_{3D} \\ z_{3D} \\ 1 \end{pmatrix} \quad (4.2)$$

When using the pinhole camera model and knowing the intrinsic parameters, it is possible to *revert* the previously explained process to generate the epipolar lines of every pixel in an image. These lines are the ones connecting the centre of the image with the real corresponding 3D point. The epipolar lines of a stereoscopic setup can be seen in Figure 4.3 ( $X_L - X$  and  $X_R - X$ ).

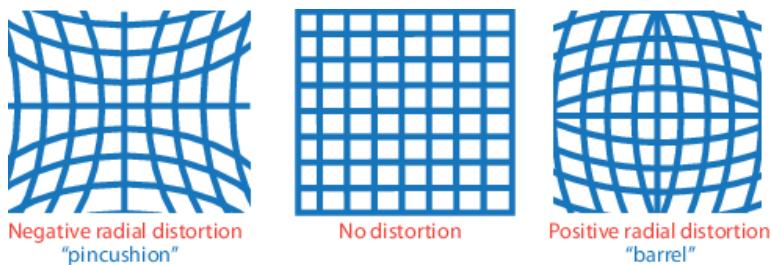
Another relevant fact is that the vast majority of commercial cameras usually carry integrated lenses, which provides them with some advantages such as better performance on higher or shorter ranges or lighter or darker environments. This makes the pinhole model become imperfect. That is the reason why, as previously mentioned, distortion coefficients



**Figure 4.3:** Representation of the epipolar lines in a stereoscopic setup.

**Source:** Wikipedia - Epipolar Geometry, Arne Nordmann

are used. They can be classified in radial ( $k_1, k_2, k_3, k_4, k_5$ , and  $k_6$ ) and tangential ( $p_1$  and  $p_2$ ). The mentioned examples are the one considered in Open Computer Vision library (OpenCV), but it is possible to use others of even higher order to get more precision. In this work, the only ones considered are the radial ones of 1st to 3rd order and the tangential ones until the 2nd order (both inclusive). Not considering this distortion coefficients leads to some deformation in the images. The effect of not considering the radial ones can be appreciated in Figure 4.4. Again, this is more detailed in OpenCV-Documentation (n.d.).



**Figure 4.4:** Effects of radial distortion.

**Source:** Mathworks - Camera calibration

Another concept that is worth to mention are the extrinsic parameters of a pair of cameras. This, alongside the intrinsic ones and the pinhole camera model, are the basis of stereoscopic vision and makes it possible to compute depthmaps and pointclouds out of two bi-dimensional images of the same environment captured using a calibrated pair of cameras. To do so, it is necessary not only to be able to know the intrinsic parameters of each camera but also the relationship between the two devices. This will make it possible to, knowing the two epipolar lines for each pixel existing in both images, compute their intersection to find the exact 3D position of a point represented by the mentioned pixel (known as *match*), as exposed in Wikipedia-contributors (2019b).

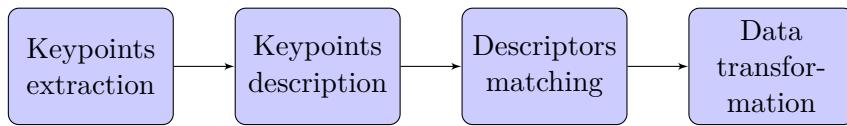
The extrinsic parameters are just a transformation matrix, with its 3x3 rotation matrix

$(R)$  and the rotation vector  $(T)$ , as can be appreciated in Equation 4.3.

$$T = \begin{pmatrix} R & T \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (4.3)$$

#### 4.1.1. 2D vision and image registration

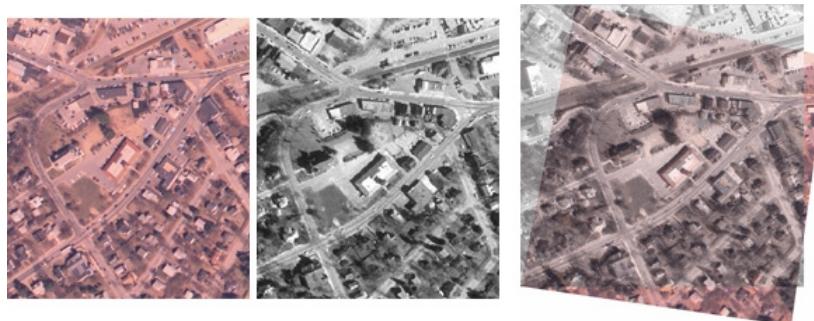
As mentioned before, a big part of the computer vision field has always been the image registration. Traditionally it has been done following a clearly defined pipeline represented in the Figure 4.5.



**Figure 4.5:** Representation of the traditional pipeline in image registration.

**Source:** Own preparation

This problem consists in transforming sets of data in order to make all of them able to be overlapped with one of the images. This makes it possible to simultaneously visualise information contained in different images in case all that information could not be captured in a single one. This can be due to the angle of vision of the camera, the requirement of a certain configuration (ISO, exposure, etc.), sensor or distance to the scene. An example of this can be seen in Figure 4.6.



**Figure 4.6:** Example of image registration.

**Source:** Mathworks - Image registration

The extraction step is about detecting which pixels of the image are specially relevant. It is usually done using some features considered relevant, such as the contrast among their neighbourhood, for example. In OpenCV-Documentation (2015), several keypoint extractors are mentioned, such as Harris, SIFT, SURF, FAST or ORB.

As listed in OpenCV-Documentation (2014b), SIFT, SURF, BRIEF, BRISK, ORB or FREAK are examples of descriptors. These algorithms are designed to build descriptions

out of a keypoints list (if they are local) or from an entire image (if they are global). This features are commonly assumed to be almost unique of the data they were extracted from, so it can be used to recognise it in another sampling in which the same data is captured. Depending on the descriptor, it can be robust to escalation, rotation, light variance or other changes, so it is necessary to know this kind of differences between the descriptors to choose the more appropriate one for the target problem.

It is worth to mention ORB (OpenCV-Documentation, 2014d), given that it is the extractor and descriptor used by O-S2 to find *landmarks/features* in the environment. The complete name for this method is *Oriented FAST and Rotated BRIEF* and it has its origin in the OpenCV labs, developed as an alternative to SIFT and SURF not only because of their computational cost but because they are patented. As its name implies, it is a combination of the FAST (OpenCV-Documentation, 2014c) keypoint extractor (Features from Accelerated Segment Test) and the BRIEF (OpenCV-Documentation, 2014a) descriptor (Binary Robust Independent Elementary Features), but with many modifications in order to increase its performance. After applying the FAST descriptor, the Harris corner measure is used in order to select N *top-keypoints*. Also, escalation and rotation robustness are introduced using pyramid and the corners orientation (connecting them with the centroids of the patches, which are extracted using the intensity). Finally, BRIEF is used to describe the extracted features. In order to fix its poor performance when rotation is involved, the descriptors are combined with the orientation of the keypoints.

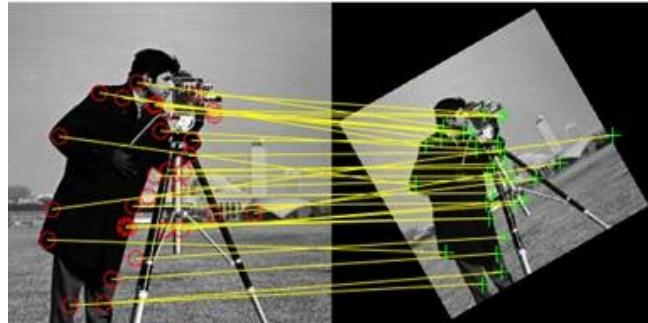
Last, the *matching* consist in comparing ones descriptions to others in order to determine which ones correspond to the same data (pixel or image). This is commonly done by computing distances, as explained in the Section *Basics of Brute-Force Matcher* in OpenCV-Documentation (2013). Some of these distances are:

- **L1 Norm:** Also known as the Manhattan Distance, this norm is the sum of the magnitude of the vector to be evaluated. In the case of comparing [0, 0] with [3, 4], the L1 Norm will be  $\|d\|_1 = |x| + |y| = 3 + 4 = 7$ .
- **L2 Norm** or Euclidean Norm is, by far, the most used and represents the shorter distance between two points. For example, applying it again to [0, 0] and [3, 4], the result will be  $\|d\|_2 = \sqrt{|x|^2 + |y|^2} = \sqrt{3^2 + 4^2} = \sqrt{9 + 16} = \sqrt{25} = 5$ .
- The **Hamming Norm** is the amount of changes that need to be done in a *word* of data in order to transform it into the one it is being compared with. For example, the Hamming Norm for "111" and "010" will be 2 and, for "sake" and "take", 1. It is commonly used alongside binary string descriptors.

An example of the matching step can be seen in Figure 4.7.

Finally, the data transformation consists in computing which transformation will make the extracted, described and matched features be better aligned. This is done by computing the best geometrical transformation that best archive the mentioned alignment.

Nonetheless, nowadays this pipeline is almost abandoned except for simple processes, low resource platforms and some other specific situations. The reason why this happened is the introduction of Machine Learning (ML), DL and, more specifically, CNNs in the Computer Vision field. This techniques clearly outperforms the previous state-of-the-art methods, given their capacity of self-learn the solving process that best suits a given problem. On the other



**Figure 4.7:** Example of feature matching.

**Source:** Mathworks - Image registration

side, the usage of this technology has some disadvantages, as their mathematical complexity, the resources needed to train them or the huge amount of data that is usually needed to train a model to take full advantage of its potential.

#### 4.1.2. 3D vision and pointclouds

The Computer Vision field has enjoyed a lot of huge advances regarding the solving of bi-dimensional problems that has led us to a situation in which most of them can be considered solved thanks to the refined techniques developed by researchers all around the world and to the computational power available.

Regarding this last point, when dealing with three-dimensional data, a way bigger amount of data has to be computed. As minimum, it is needed to consider the point's position in the 3D space (otherwise, it would be a 2D problem), multiplying the information by 2. Furthermore, even some algorithms can be applied to three-dimensional data, many others can not. This is due to the unstructured nature of the pointclouds. Depending on the viewpoint, two points that might seem to be next to each other, can actually be at tenths or hundreds of meters away. This last one is the biggest issue when solving 3D problems.

The most common methods to acquire 3D data are based in different technologies, such as:

- **RGB-D:** They combine *traditional* RGB information acquired from a normal 2D camera with depth information. This extra information can be extracted in many ways. Some of them can be Time of Flight (ToF) or the deformation of a projected (usually infrared) pattern.
- **Stereo:** This technology is based in the concepts explained in the beginning of Section 4.1. Summarising, it uses stereoscopic vision, which is based in applying epipolar geometry to a calibrated pair of cameras. This makes it possible to, given two images, get as many matches as possible and compute their correspondent epipolar lines for each viewpoint. Having two lines and knowing the position of one of their points (the  $[0, 0, 0]$  for one and the equivalent of transforming this one using the extrinsic parameters), it is possible to calculate the intersection of the two points. Doing this for every match, will provide with the distance they are from each camera. This two distances can be

used to triangulate the 3D positions (and color information) of each match, building the pointcloud.

- **SfM:** It tries to replicate stereoscopic vision using only one RGB camera, but moving it. This technology applies the same methods than the last listed option but the *extrinsic parameters* here will be an estimated transformation (rotation and translation) obtained by looking for matches and registering the pairs of images, making use of the concepts explained in Subsection 4.1.1.

Furthermore, there are other options that are not as extended as these ones, which are a current research field. This are the methods that allows getting depthmaps without the need of neither two cameras (calibrated or not) or using the traditional pipeline. This is archived using DL and is explained more in-depth in Section 2.2 of Chapter 2. This approaches have a particular relevance in this work, given than one of the key parts of it is DeMoN, which will be explained in the Subsection 4.3.2 of this Chapter.

Since 3D data involve positions defined by vectors in a three-dimensional space, some mathematical theory can be applied in order to archive really useful modifications of the data. The most used (and one of the simplest) process is the homogeneous transform. In linear algebra, these matrices are used to represent linear transformations being rotations, translations and escalations the most useful. They are represented as shown in Equation 4.4.

$$T = \begin{pmatrix} R & T \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (4.4)$$

In case an escalation factor is necessary, it is done like shown in Equation 4.5, which assume no rotation (identity matrix) or translation (zero matrix). In the first matrix a different escalation factor is considered for each axis ( $s_x$ ,  $s_y$  and  $s_z$ ). A uniform escalation can be done by using the same factor in the previously mentioned components or by using the notation on the right.

$$S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1/S \end{pmatrix} \quad (4.5)$$

The direct application of this transformations is the process known as *pointclouds registration*. This consist in use several sets of data to build a bigger one using some or the whole of the information in the mentioned pointclouds. By transforming each pointcloud in order to locate it in its proper position and orientation taking one of the others as a reference, it is possible to build this *meta-pointcloud*, which will contain information impossible to get in only one capture. This situation can be caused by many things, such as scene occlusions or the view angle or range of the camera.

Even the computation of the transformed clouds given a transformation matrix is simple ( $p' = T \cdot p$ ), the complex part of the registration process resides in the obtaining of the matrix. This is one of the keysteps of this work and consist in applying algorithms that

usually estimate the needed transformation of a set of points to minimise the distance to another set. This process requires a lot of computational power, so applying it to the full pointclouds is usually non-viable (since they usually contain 200k to millions of points). Given this limitation, the solution is to apply a similar process of the one explained in Subsection 4.1.1. That is to say, extract keypoints and describing them in order to find matches. Finally, the estimation will be done using only this sub-set of already matched points. It is worth to mention that, when working with 3D data, it is possible to use both extract and describe points the 3D position information of the points and their environments, so it is common to see the usage of surface normals or curvature, for example. However, techniques as searching matches in a neighbourhood instead of the whole set have more complexity and require more computational resources.

One of the most used tools when performing registration is Singular Value Decomposition (SVD). As explained in MIT-OpenCourseWare (2016), the Singular Value Decomposition consists in factorise a rectangular matrix  $A$  of dimensions  $m \times n$  into three factors:  $U$  (orthogonal),  $\Sigma$  (diagonal) and  $V^T$  (orthogonal), as shown in Equation 4.6. On the right side of the equation, an example where  $m = n = 2$  is shown.

$$A_{m \times n} = U_{m \times m} \Sigma_{m \times n} V_{n \times n}^T \Rightarrow \begin{pmatrix} u_{1x} & u_{2x} \\ u_{1y} & u_{2y} \end{pmatrix} \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix} \begin{pmatrix} v_{1x} & v_{1y} \\ v_{2x} & v_{2y} \end{pmatrix} \quad (4.6)$$

Regarding the diagonal matrix of the SVD factors,  $\Sigma$ , the values on its diagonal are called singular values. Furthermore, the system bases on  $U$  and  $V^T$  are named singular vectors.

This factorisation makes it able to compute estimations of rigid transformations between two sets of points with known correspondences. This is also solvable using methods based in Levenberg Marquardt, but the ones based in SVD, make it possible to estimate the transformation in only one step. However, the Levenberg Marquardt based methods, since are iterative processes, allow to adjust the process in each iteration while it is being executed, which is necessary in some cases, such as Iterative Closest Point (ICP) (dave\_mm0, 2012).

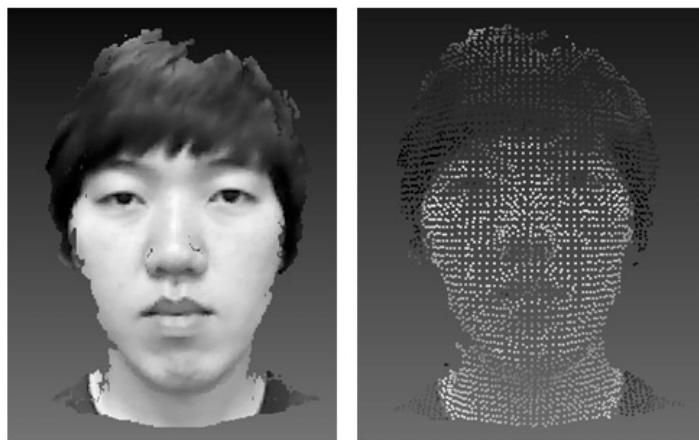
Another relevant aspect of SVD is that, since one of its factors is a diagonal matrix, it can estimate escalation. In fact, even for this project only a uniform escalation is needed, it could provide the method with different escalations for each axis, being one of them each singular value.

The other two factors, the orthogonal matrices, are used as rotations. This way, SVD can represent a rotation, an escalation and another rotation by its *raw* factors. Last, the translation is archived by computing the vector between the two centres of gravity of the clouds, making it possible to transform one cloud in another if the differences include rotation, translation and escalation, which is the case of this work as will be explained later on. Furthermore, in many cases the escalation is not even necessary, given it is only an issue when the information has not been extracted using a 3D sensor (this kind of sensors take real measurements of the world, while the methods based on one 2D image are based on projections).

Another widely-used method is Random Sample Consensus (RANSAC). It is an iterative algorithm based on statistics that, applied to the registration problem, allows the computation of the transformations using only some of the point pairs (matches) that are available. This makes possible to compute several possible transformation using only the strictly necessary points (or a few more) to compute a transformation in the space that is being used (2D, 3D,

etc.). In the case of this work, each frame provided over 250 points to work with to compute the transformations. Taking in account that some of these could be errors introduced by both O-S2 or DeMoN, computing the transformation with RANSAC introduces a huge reduction in the probability of using one of these errors to compute the transformations given that, sampling points randomly, it is way more probable to take a good one than a *bad* one.

Given the huge amount of data involved in 3D problems, the filters are a keypart of a considerable number of the proposed approaches to solve them. One of the most commonly used spatial filters is the Voxel Grid. This consists in dividing the 3D space into small cells (a *grid of voxels*), deleting every point inside each voxel and creating one new point in the centroid of each removed sub-set, generating it with the mean RGB values in case the pointcloud includes color information. The result of applying an exaggerated one can be seen in Figure 4.8.



**Figure 4.8:** A pointcloud before (left) and after (right) applying a Voxel Grid filter.

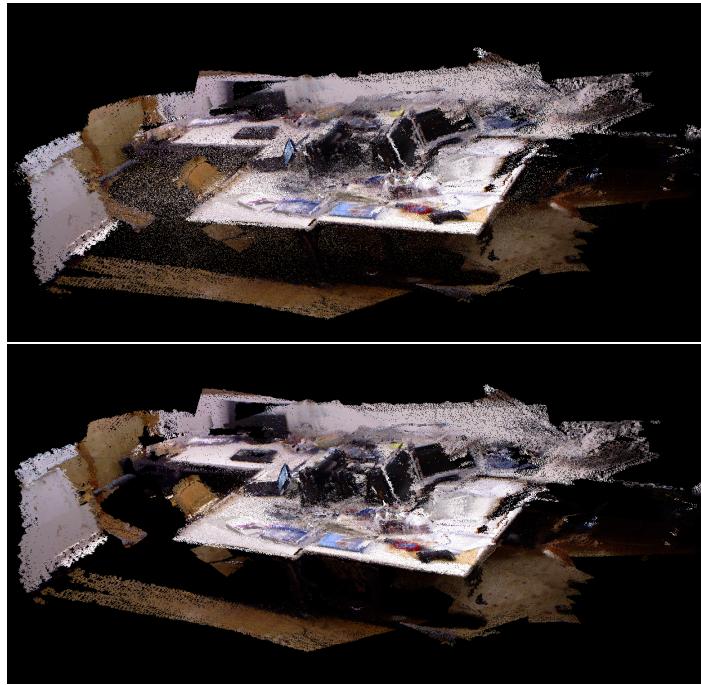
**Source:** ResearchGate - Hakil Kim

Another widely used filter is Statistical Outlier Removal (SOR). As its name implies, its function is to remove outliers using statistics. This method works by computing the mean distance and its variance from each point to its neighbours. The points whose mean distance and/or variance are too different from the mean ones of the whole set of points are considered outliers and, as a consequence, removed from the pointcloud. A more detailed explanation is in the PCL tutorials and an example of the effect this filter has in one of the groundtruths I used for this work is the one shown in Figure 4.9.

## 4.2. SLAM

SLAM is one of the most common techniques in mobile robotics. It allows to both map a place and localise a robot in it at the same time. It is considered pretty much solved in a relatively big amount of applications but there are situations in which the actual state of this technology is not enough, such as highly dynamic scenarios or when high velocities are demanded.

The SLAM algorithms try to, at the same time, build a map and localise the used sensor



**Figure 4.9:** A pointcloud before (top) and after (bottom) being filtered by SOR.

**Source:** Own preparation

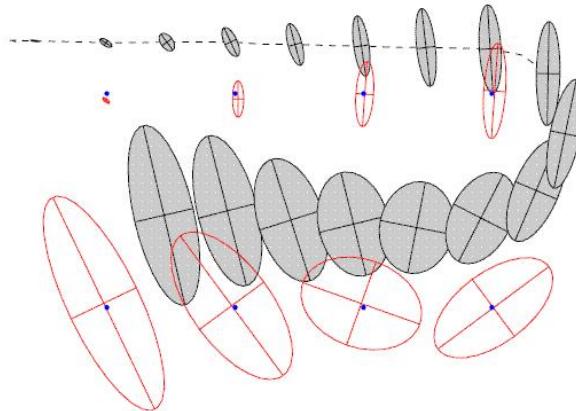
in it. This is a complex problem since in order to archive a proper localisation, a good map is needed and, in order to build a good map, the robot must be perfectly located. To solve this problem, SLAM algorithms make use of statistics to keep a mathematical representation of the uncertainty carried during the whole execution and the relationships between the involved factors (the position of the robot and the *things* perceived), allowing to reduce this ambiguities when the uncertainty of any previously stored data is reduced.

It is to say, as the robot moves, the uncertainty regarding its pose will increase, since the odometry information is not perfect, and so will do the one regarding the perceived environment (measures, beacons pose or whatever approach is being used), as can be appreciated in Figure 4.10.

However, the key *trick* of SLAM is the loop closing. This makes it able to reduce the variance of every pose and perception when a perception seen in the past is seen again. This is possible because every perception is related to the one before it and to the pose when it was perceived. This allows the SLAM methods to discard non-feasible past hypotheses of the last perception given a new one (seen before) and back-propagate this new information towards every past pose and perception.

This techniques can be divided in two big groups depending on what are they based into:

- **Extended Kalman Filter (EKF) based:** As its name points, this type use the Extended Kalman Filters to track the position and update the so far built map. Its implementation is really heavy and requires a lot of computational resources, so it is commonly only used in small spaces. It's main field of application is in the Virtual and Artificial Reality because it offers a precise localisation and in the worst case, the



**Figure 4.10:** Representation of the uncertainty increase of the pose (shaded grey) and perceptions (red) of a mobile robot.

**Source:** ResearchGate - Max Pfingsthorn

algorithm will be applied in just one room while in the common ones, just an object such as a table. An optimisation of this algorithm is the UKF SLAM (Unscented Kalman Filter SLAM).

- **Particle filters based:** This category tries to minimise the computational cost of the other one. It does it by using many *particles*, which are hypothesis of the current situation (robot's pose and built map). Each one will assume it is right, allowing us to make some significant simplifications in the probabilistic part of the algorithm. This SLAM branch has some remarkable modifications such as the rao-blackwellized SLAM (also known as FAST SLAM).

Obviously, SLAM can be performed using many sensors, but the more popular are distance and shape information provided by LIDAR sensors and cameras (RGB, RGB-D or Stereo). The SLAM methods using this last kind of devices are known as VSLAM algorithms. As explained in VisionOnline (2018), this systems work by tracking a set of (key)points extracted from a sequence of frames to triangulate their 3D position (when using only one 2D camera). Even, as said before, it is possible to use RGB-D or Stereo cameras, I am going to focus on the use of only one 2D RGB camera, keeping in mind that it is the base of this work.

The VSLAM algorithms using only one RGB camera are classified as Monocular VSLAM. This methods are usually based in keyframes, so they are known as KSLAM. Depending on the approach, a frame will be considered or not a keyframe taking in account some indicators, such as the number of keypoints, the time since another frame was considered a keyframe, etc. This is more deeply exposed in Section 2.1, in Chapter 2, where some specific works are referenced.

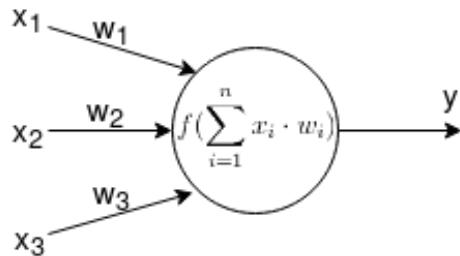
## 4.3. Deep Learning

*Deep Learning* is a field which borders are not clearly defined. However, it is possible to understand it as a subset of the algorithms and techniques that compose *Machine Learning*

that, at the same time, is a part of the giant known as *Artificial Intelligence*.

While Artificial Intelligence (AI) is a concept that refers to provide machines with something similar to what is considered *intelligence*, ML is a more defined field. It is focused in more particular concepts, such as algorithms and methods that makes the computers able to learn something after being provided with some data. The key difference is that ML is not as broad as AI and focuses in implementations, not in concepts. Last, the DL is a subset of ML, commonly understood as the part of ML which involves Neural Networks (NN) or, depending on who you ask, *deep* NNs. Here, a new discussion can be opened: the debate about what *deep* means. For some people it implies two or more layers, while for others is just an abstract concept identified with an unspecified level of complexity.

The concept (Artificial) Neural Network (NN) has been mentioned as a key part of the last explanation. These are computational models inspired in the vastly unknown behaviour of the brains, which contain (Natural) NN. These models are nets of computational units named *simple perceptrons* or *neurons*. This unit is a really simple concept, a *thing* with several inputs ( $x_1, x_2, \dots, x_i$ ) and one output ( $y$ ). If the sum of the inputs multiplied by certain weights ( $w_1, w_2, \dots, w_i$ ) pass a certain threshold, the output is *activated*, being 1 instead of 0. A clarifying illustration is shown in Figure 4.11, in which a simple perceptron with 3 inputs can be seen.



**Figure 4.11:** Graphical representation of a neuron.

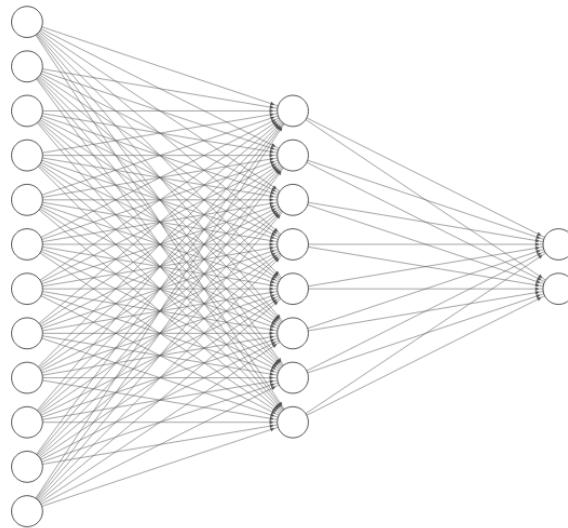
**Source:** Own preparation

Just as its name implies, a NN is just a network of neurons. The most simple structures are known as *fully connected* and their structure is a certain number of *layers*, with a certain number of perceptrons in which the output of every perceptron of the layer  $n$  is sended as an input to each perceptron of the layer  $n + 1$ . The layers of neurons in which the outputs are not the output of the network are named *hidden layers*. A fully connected NN with an input layer of 12 neurons, a hidden layer with 8 and an output one with 2 is shown in Figure 4.12.

### 4.3.1. Convolutional Neural Networks

Given the huge research power that has been being invested in the DL field, a lot of new types of networks with many different ways of combining them are being invented and applied in the solving of many problems. In Tch (2017), 27 different types of NNs are graphically represented and briefly explained.

Between all of these options, the ones having an special relevance for this work are the Convolutional Neural Networks. This type of networks apply their operation as convolutions.



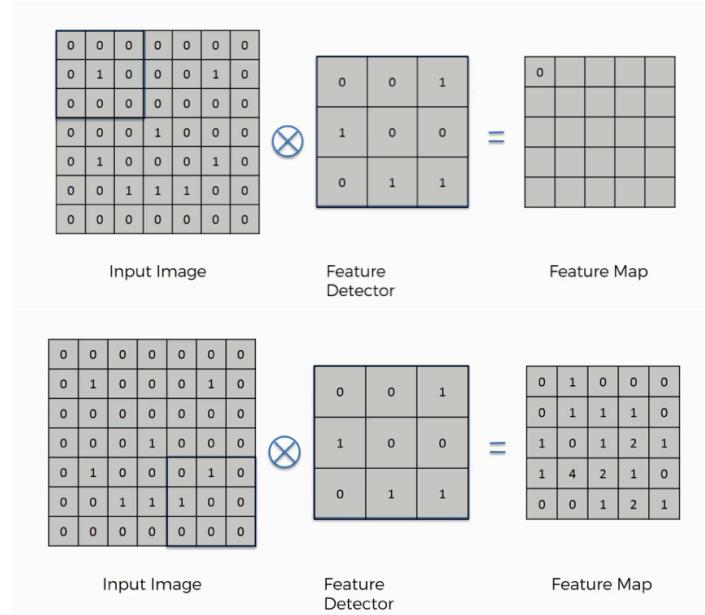
**Figure 4.12:** Graphical representation of a fully connected neural network.

**Source:** Own preparation

A *convolution* is, from a practical point of view in the computer vision field, as explained in SuperDataScience-Team (2018), a feature extractor. In order to apply a convolution to a given matrix, you have to *locate* the *kernel* over the top left corner of the target matrix and multiply and sum all the matching elements. The result will be placed on the top right corner of the output matrix. Next, the kernel is moved one element to the right and apply the same operation, getting the next element of the output matrix. This process is repeated until the arrival to the end of the target matrix, moment in which the next step will be to go to the beginning of the next line and repeat the process until the end of the target matrix is reached. An illustration of this can be seen on Figure 4.13.

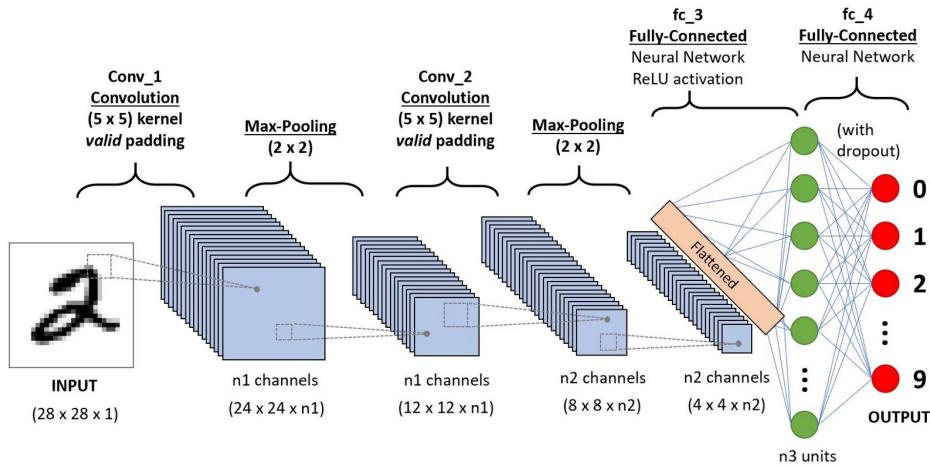
A CNN will use convolutions as the function that transform the input in order to get a result. Convolutions are the basis of digital image processing, since they can effectively provide binary images with the borders, corners and many other more complex features of images. They are also really efficient and highly susceptible of being implemented making use of the huge parallelism allowed by the cores of a GPU, keeping in mind that what is needed to be done are a lot of independent easy operations (so they do not require high clock speeds).

Given the huge potential convolutions have for Computer Vision, the CNNs take advantage of it letting a computer learn which transformations are the best to transform an image in order to learn to extract certain information. This is archived by applying a sequence of convolutions that, after a good training, are able to distinguish cats from dogs or detect where the driving lane is in a road. A graphical representation of a CNN is shown in Figure 4.14.



**Figure 4.13:** Convolution computation.

Source: SuperDataScience - Convolution operation



**Figure 4.14:** Convolutional Neural Network representation.

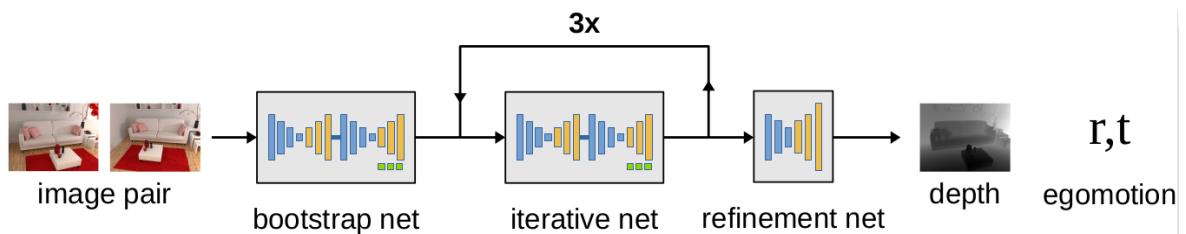
Source: Towards Data Science - A Comprehensive Guide to Convolutional Neural Networks

### 4.3.2. DeMoN

The DEpth and MOTion Network is a work developed in a collaboration between the University of Freiburg and the Daimler AG R&D department. In this work, they "formulate structure from motion as a learning problem". As said in the abstract of their paper, they trained a CNN to get an estimated depthmap and camera motion (rotation and translation).

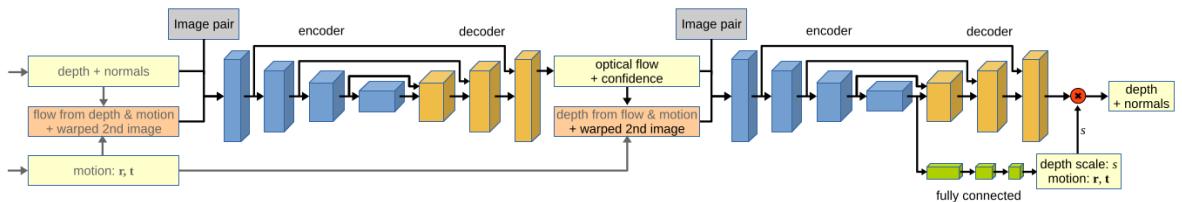
As an input, this network receives a pair of unconstrained images that are supposed to be two relatively close points of view of the same object/scene. Basically, the network is a sequence of encoder-decoder networks that estimate and refine the mentioned data. The general architecture can be seen in Figure 4.15. They use three nets:

- **Bootstrap net:** Receives the pair of images and performs an initial depth and motion estimation using an optical-flow estimation net and a confidence measure on its result (provided by a encoder-decoder network), which is forwarded to another encoder-decoder net, which estimates the depth, surface normals and camera motion. Furthermore, it also predicts a scale factor using a fully connected network. Its detailed architecture can be seen in Figure 4.16.
- **Iterative net:** This is the core of the network. It has been trained to refine the depth, normals and motion provided by the last network. Its architecture is identical to the bootstrap one but with additional inputs. A qualitative example of its performance over its iterations can be seen in Figure 4.17.
- **Refinement net:** This final net not only keeps refining the results but also upscale the depthmaps. While the previous network worked in  $64 \times 48$  resolution, this one does it at  $256 \times 192$  (the full size of the inputs). Its effect on the final result is exposed in Figure 4.17.



**Figure 4.15:** Convolutional Neural Network representation.

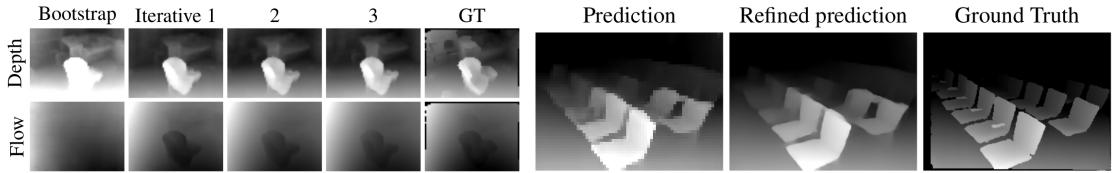
**Source:** Ummenhofer et al. (2017)



**Figure 4.16:** Convolutional Neural Network representation.

**Source:** Ummenhofer et al. (2017)

Regarding the training procedure, several loss functions have been used, since the outputs of the net have a wide variety of characteristics, being the most relevant the different amount of dimensions, going from the high-dimensional depthmaps to the low-dimensional camera



**Figure 4.17:** Qualitative examples of the performance of the iterative net (left) and the refinement one (right), in comparison with the groundtruth (GT).

**Source:** Ummenhofer et al. (2017)

motion. Even some other loss functions has been used with the normals, the optical flow ( $L_2$  norm) and the quality of this flow ( $L_1$  norm), I am only going to mention the ones regarding the depthmaps and the motion, since they are the main outputs of the network.

Being  $\xi$  the inverse depth and  $\hat{\xi}$  the groundtruth, the depthmap loss is computed using the  $L_1$  norm, which is already explained with an example in Subsection 4.1.1 of this Chapter. The per-pixel  $(i, j)$  loss function is Equation 4.7. Note that the predicted scale ( $s$ ) is applied to the depth prediction.

$$L_{depth} = \sum_{i,j} |s\xi(i, j) - \hat{\xi}(i, j)| \quad (4.7)$$

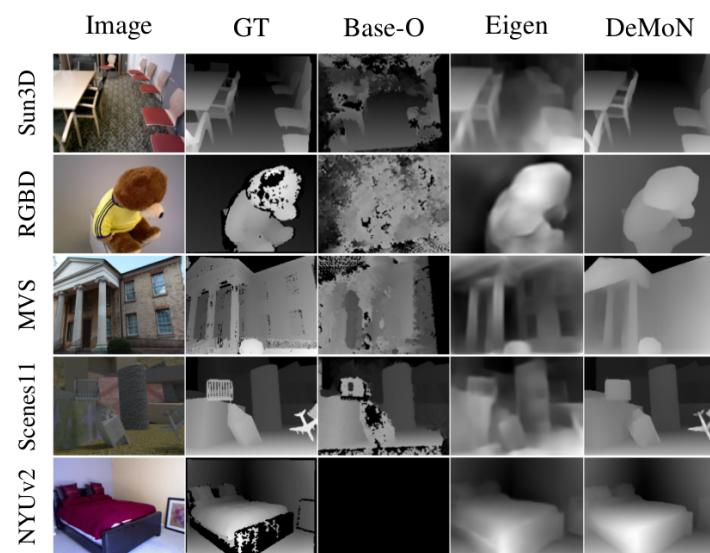
With regard to the motion loss, this output is parametrised using three parameters for rotation ( $r$ ) and other 3 for translation ( $t$ ), being  $\hat{r}$  and  $\hat{t}$  the groundtruths. It is worth to mention that the translation groundtruth was normalised to  $\|\hat{t}\|_2 = 1$  and that the magnitude of  $\hat{r}$  encodes the angle rotation. Keeping that in mind, the loss function is the Equation 4.8.

$$L_{rotation} = \|r - \hat{r}\|_2 L_{translation} = \|t - \hat{t}\|_2 \quad (4.8)$$

Finally, another loss function is used in order to penalise relative errors between the pixels of a neighbourhood based on a gradient. However, this one is not going to be explained since is only a way to improve the results and to archive more uniform outputs.

In order to illustrate in a qualitative way the results of DeMoN, a comparison with the results archived by other similar methods with various datasets is shown in Figure 4.18.

Even without looking at the correspondent quantitative results shown in the paper, it is clearly seen the outperforming sharpness and surface uniformity in the DeMoN results relative to the other state-of-the-art methods.



**Figure 4.18:** Qualitative comparison between the DeMoN result and the ones archived by other methods (top) and the groundtruth (GT) using various datasets (right).

**Source:** Ummenhofer et al. (2017)



## 5. Approach

The source files of this work is publicly available in my GitHub profile, in the repository *mapSlammer*<sup>1</sup>.

This approach is focused in generating dense 3D maps using a KSLAM method and estimated depthmaps. The proposed pipeline takes as an input a sequence of color images (internally processed by pairs) and returns the dense point cloud that represent the 3D map of the environment in which the sequence was filmed. This method is intended to provide small-size robots with three-dimensional dense mapping capabilities in order to make them able to take advantage of the amount of information contained in dense maps (in contrast to the sparse ones). However, it could be deployed in any robot that features a regular color camera to solve spatial, weigh, cost or simplicity issues.

To do so, a moving robot capturing color images is assumed. As the robot is moving, there exists a transformation between two consecutive frames. This transformation explains the movement of the robot between these two frames. Both frames are forwarded to O-S2, which follows a KSLAM approach. This method was modified in order to return the sparse 3D point cloud of the landmarks it use to localise the camera, along with the estimated camera motion (that is the only original output).

A depth estimator, which is based in the DeMoN (Ummenhofer et al., 2016) approach, is run as the next step. This Deep Learning-based method takes as input a pair of images as well and returns the estimated depth map of the scene with an arbitrary scale. This depth map is then projected into a dense 3D point cloud that represent the environment depicted in the most recent image coordinate frame. This is done by using the intrinsic camera parameters of the camera which the dataset used to train DeMoN was recorded with.

Then, the 3D landmarks returned by the KSLAM method are re-projected into the most recent image of both so a set of 2D points are generated. These 2D points are looked up into the point cloud generated before using the very same coordinates. This is straightforward as there is a 3D point for each 2D point, since the pointcloud is generated using a 2D depthmap (from which the data is taken using the mentioned coordinates). At this point, the available data is a set of scattered 3D landmark points computed by the KSLAM and the corresponding 3D points obtained from deep learning-based estimation. Since they were stored in the same order, it is possible to know which point from one cloud correspond to which one of the other, removing the need of running the traditional 3D matching pipeline. These correspondences are used to compute a transformation (translation, rotation and scale) between both subsets of points so the estimated dense point cloud is aligned with the scattered 3D landmarks.

It is worth noting that both sets of points yield different, arbitrary scales. It should also be highlighted that as the dense points clouds are estimated, they yield some error.

To compute the correct transformation, the chosen option was to use a RANSAC (A. Fischler & C. Bolles, 1981) approach. This algorithm makes it possible to find the best transfor-

---

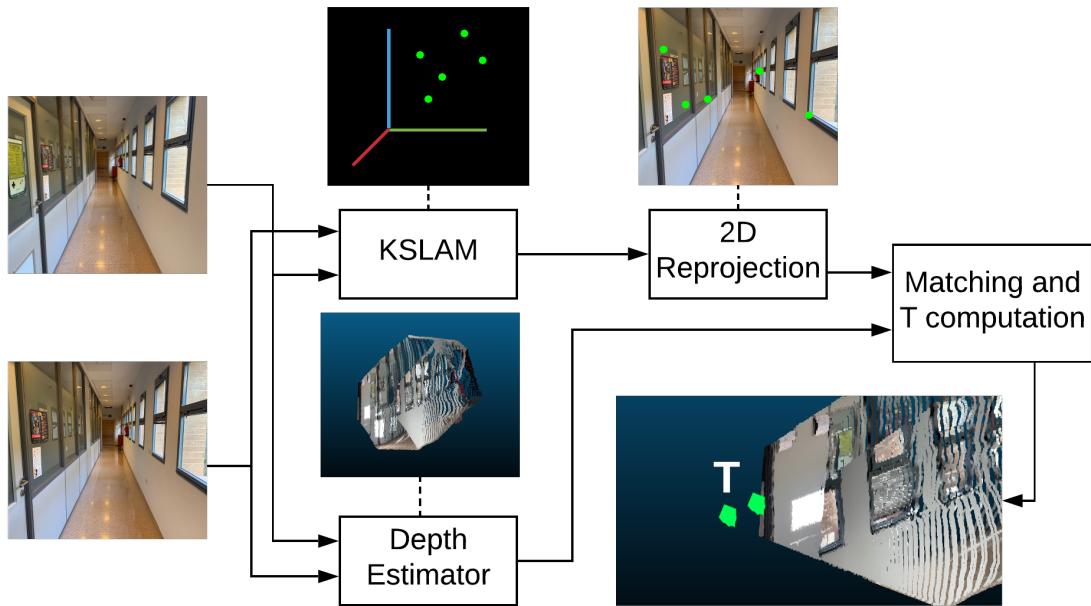
<sup>1</sup><https://github.com/jmtc7/mapSlammer>

mation despite having some error in the corresponding points. To deal with the scale issue, the transformation is computed using SVD with scale component (Eggert et al., 1997).

The RANSAC process is as follows. First, 10 random correspondences are chosen and used to compute the transformation (rotation, translation and scale) using SVD. The resulting transformation is then applied. Finally, the inliers are counted. A correspondence is considered an inlier when the corresponding points are under a distance threshold. This process runs in a loop for 300 times. Finally, the best transformation is returned, which will be the one that achieved the greatest number of inliers. This step also helps to filter erroneous transformations. If the best transformation does not explain at least the 25% of the points, this key frame is discarded.

I implemented this because, since the points are estimated, keeping in mind the possible existence of errors was a must. If big enough errors in the sparse subset of the estimated cloud appear, the SVD process will return a not good enough transformation, so it should be detected in some way in order to discriminate this transformation (or even the entire cloud if none of the RANSAC iterations find a proper transformation).

The returned transformation matches the relative motion of the camera between two key frames. So the proposed algorithm returns the camera localisation and the corresponding dense 3D map. This process is applied in a loop, for each pair of frames of the sequence, to finally reconstruct the environment in which the camera is moving. A diagram of the pipeline is depicted in Figure 5.1.



**Figure 5.1:** Pipeline of the proposal. This pipeline is looped to finally build a full 3D map of the environment.

Source: Torres-Camara et al. (2019)

---

Next, the different pieces that compose this pipeline are explained more in-depth.

## 5.1. Integration

The two key parts of this work were O-S2 and DeMoN, introducing the problem of working with two different programming languages (C++ and Python, respectively). Even I had available several options to process the information in order to implement this project (OpenCV can be used both in C++ and in Python and there are alternatives to the Point Cloud Library (PCL) for Python), I needed to use the previously mentioned source codes. Given this situation, I saw myself in the need of integrating them and, in order to solve this problem, I made use of bashscripting.

This approach for the integration allowed me to develop independently and in a more easy way each sub-part of the project, in whatever language I wanted to as far as it could be run using the command line and, being in a GNU/Linux environment, that opened my way to everything. Furthermore, by using a bashscript, it was extremely easy to work with directories and files in order to delete old things to avoid failures. It also made it easier to manage the verbosity of the result using log files. Nonetheless, it allowed me to develop an interactive script that made it easier to run tests with different datasets or camera settings and working with intermediate outputs to test the changes in some part of the project without running every step, saving a lot of time while debugging or implementing new features or modifications.

This bashscript works in a clearly defined environment in which not only logs and outputs were stored, but also as much datasets and camera settings as I wanted to were available to the script and, of course, the source codes and executable files. The environment was organised using the next directories:

- **logs:** Where the log files were stored, including a junk file where the (considered) useless outputs were written. This last feature allowed me to not be bothered with the named outputs while keeping them available in case I needed to debug something.
- **output\_files:** This folder contains one sub-directory for each step of the process (the KSLAM method, 3D to 2D reprojection, depth estimation and transformation and registration). The output of the refining is stored at the root of the environment to make it as available as possible to the end user (and to me to locate it quicker when developing). It is worth to mention that all the outputs names are parametrised and configurable from the bash script, so they can be changed from the bashscript without causing problems in any file. The output files of each part were:
  - **ORB-SLAM2:** 3 files, one containing the trajectory followed by the camera (KeyFrameTrajectory.txt), another with the extracted keypoints of each keyframe (KeyPoints.txt) and the last one, with the 3D points that compose the resulting map (MapPoints.txt).
  - **Reprojection:** One file for each keyframe (not frame), which name starts by "reprojectedKeyPoints\_ ", is followed by the timestamp from when the keyframe was captured and ends by ".txt". Each one of these files contain a list of the coordinates of the pixels used to generate a 3D point of the map. Each line corresponds to one keypoint and contain the X and Y coordinates.

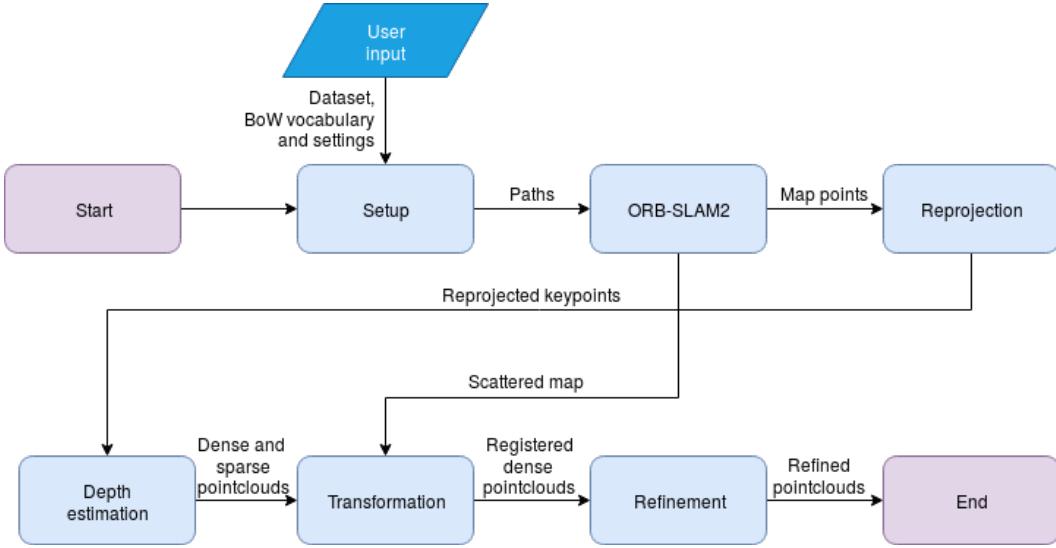
- **Depth estimation:** Two pointclouds for each keyframe. One, which name starts with "demonPoints\_FULL\_ ", is followed by the timestamp in which the keyframe was captured and ends with the extension, ".txt". This contains the full pointcloud extracted from the DeMoN estimation of the depthmap. The other file follows the same name structure but starting only with "demonPoints\_" and its content is a subset of the estimated points. This subset is made out of the points that correspond to the coordinates obtained by the previous step (reprojection). Each line correspond to one point and the numbers are the X, Y and Z coordinates of it. The "FULL" files also follow this information with the colour of the point in RGB format.
- **Transformation:** Its output files are ".pcd" files containing the pointclouds obtained by transforming the ones on the last step in order to fit them in the O-S2 map. It is worth to mention that some were rejected, as will be explained more in-depth in Section 5.5, so this folder will contain less pointcloud than the last one. Again, both the full and subset clouds are saved, with names starting by "pointcloud\_" and "pointcloud\_SPARSE\_ ", respectively, and followed by the timestamp and the file extension (" .pcd").
- **Refinement:** Finally, once every cloud has been properly transformed or rejected and the sequence is over, the final map is refined, giving as the final output the "fine\_sparse\_map.pcd" and "fine\_dense\_map.pcd". The first one is just the accumulation of every sparse sub-map of the last step, while the second one is the accumulation of the full sub-maps after being applied a filtration process, specified in Section 5.6.
- **programs:** As its name implies, it contains the programs (source and executable files) of each part of the project. Obviously it is sub-divided as the output\_files one, containing one folder for the files corresponding to each part.
- **sample\_sourceDataset:** A folder containing every testing dataset I used. In particular, they were fr1/xyz, fr1/desk, fr1/desk2, fr2/xyz and fr2/desk from the TUM dataset and someones done by myself, as one recorded with a Kinect RGB-D camera in a corridor near RoViT's laboratory.

Obviously, each steps needed information from its predecessor and, some of them, from the user. A diagram showing both the steps and the flow of the information can be seen in Figure 5.2. Apart of this information, it is always sent if the step has or not to be executed (which is asked to the user), the output directory where the results of the step has to be saved, the name format of it and the same two things regarding the correspondent log file.

## 5.2. KSLAM for computing the 3D support points

This part of the project consist in processing monocular frames and, using O-S2, generating an scattered 3D map of the environment where the sequence was/is being recorded. Originally, the files in the O-S2 repository (raulmur, 2017), only provided a file with the trajectory followed by the camera. This file, named *KeyFrameTrajectory.txt*, is composed by several

---



**Figure 5.2:** Diagram representing the information flow originated by the bashscript.

**Source:** Own preparation

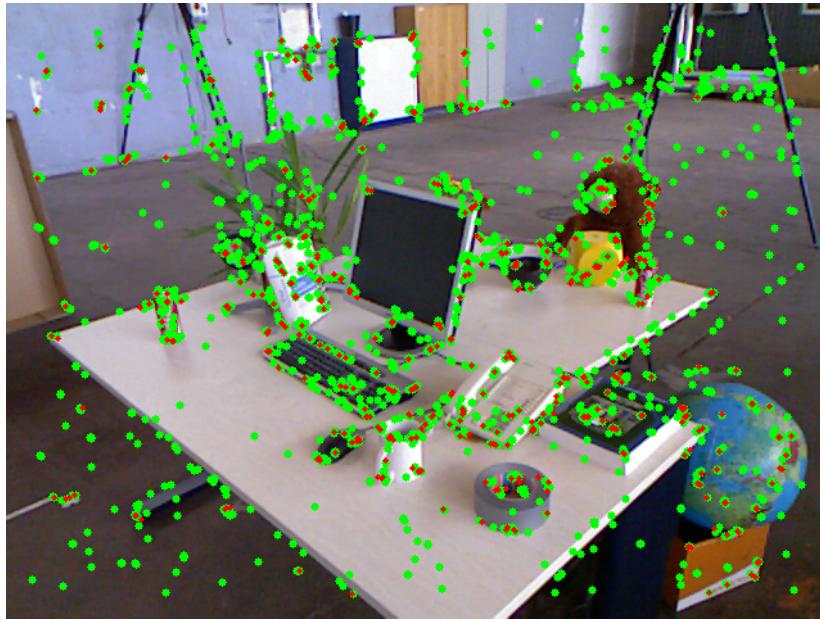
lines, being each one of them the pose of one of the keyframes conforming the trajectory. Each line has 8 columns: The timestamp corresponding to the keyframe, the X, Y and Z coordinates of the pose and a quaternion ( $q_1$ ,  $q_2$ ,  $q_3$  and  $q_4$ ) representing its orientation. Both position and orientation are relative to the world origin (which is equal to the first pose), as established by ComputerVisionGroup-TUM (n.d.).

However, in this project, the map points were needed, so the example source code was modified in order to get them. This modification was based in the code from Paulins (2016). This modification gets all the keyframes of the map using the  *GetAllKeyFrames()* function and iterates over all of them (variable *vpKFs*), gets all the generated points of each keyframe (*pKF*) using the *GetMapPoints()* function. Then, iterating over the points, the coordinates (*coords*) of each one of them (*point*) are extracted using the *GetWorldPos()* function. Finally, in the output file for the 3D map points (*MapPoints.txt*), the timestamp (*pKF->mTimeStamp*), the point ID (*point->mdId*) and the coordinates (*coords.at<float>(0-1-2, 0)*) are written.

Another extension was made in order to get the keypoints extracted from each keyframe. This was entirely done by myself and consists in getting the keypoints (*mvKeys*) of the keyframe, iterate over them and write in the output file (*KeyPoints.txt*) one line for each keypoint. In the lines, the timestamp and the X and Y coordinates are written (*keypoint.pt.x-y*).

### 5.3. Keypoints reprojection

During this step, the points of the 3D maps are processed frame by frame, it is to say, the sets of points generated using each frame are reprojected apart of the sets of points extracted from other frames. This step is necessary since not every extracted keypoint is used to generate a map point. That happens because, in order to do so, it must appear in at least two consecutive frames so, if during the matching process (using the Bag of Words (BoW)



**Figure 5.3:** Extracted keypoints (green) vs reprojected ones (red).

**Source:** Own preparation

descriptions derived from the ORB ones) fails for some keypoints, they will not be used to generate map points. This is clearly appreciated in Figure 5.3, where every extracted keypoint in a frame is shown in green and the ones reprojected from the map are in red. Every red point is inside a green one (since its 3D correspondence was generated from it) but not every green point has a red one in it (since not everyone could be matched with another one on a consecutive frame).

The reprojection is possible in case the intrinsic parameters of the used camera are known. Since these parameters were available for every used dataset and I calibrated the Kinect used to generate the proprietary dataset, it was not a problem. In a real case, the team using this method will know the camera they are using and, in case not to have the parameters, they could easily calibrate it.

The reprojection, having the intrinsic parameters is straightforward since it consists in recreating the working flow of a camera, explained in Section 4.1 from Chapter 4 and illustrated in Figure 4.1. In essence, a 3D point can be converted in a 2D one using the intrinsic parameters by applying the transformation shown in Equation 4.2 of the mentioned Section. As the used cameras carry integrated lenses, the distortion coefficients are necessary in order to minimise the error in the projection. For this work, the used ones were the 1st, 2nd and 3rd order radial distortion coefficients and the 1st and 2nd tangential ones. All these parameters for the used datasets are shown in Table 5.1, which were used with a precision of 6 decimals. Even this process is easy to implement, I did it (specially because of the distortion coefficients and because the code clarity) using the OpenCV function that already implements this, named *projectPoints()* (OpenCV-Documentation, n.d.).

The trickiest part of this step was to apply the accumulative transformations that the motion of the camera introduced. This is due to the map points (in the *MapPoints.txt* file),

	Proprietary corridor	Freiburg 1 group	Freiburg 2 group
$f_x$	589.322232	517.306408	520.908620
$f_y$	589.849429	516.469215	521.007327
$c_x$	321.140897	318.643040	325.141442
$c_y$	235.563195	255.313989	249.701764
$k_1$	0.108984	0.262383	0.231222
$k_2$	-0.239831	-0.953104	-0.784899
$k_3$	-0.001984	1.163314	-0.003257
$p_1$	0	-0.005358	-0.000105
$p_2$	0	0.002628	0.917205

**Table 5.1:** Internal parameters and distortion coefficients for the used groups of datasets.

which poses are referenced to the global coordinate system, which is the one of the first camera pose. That is why, in order to project to the moving camera plane, the 3D map points coordinates had to be transformed to the coordinate system of the keyframe which points are being reprojected.

With the purpose of performing this transformations, I used the ones in the *KeyFrame-Trajectory.txt* file, that are the ones of the camera and, since the points were generated using projections done over the planes of the moving camera, also correspond to the points. An easier way to see why these transformations are valid is thinking about the movement relativity. It does not matter whether if a camera moves in an environment or the environment moves around the camera, as long as they carry the exact same movement (but inverted), the perceived images will match.

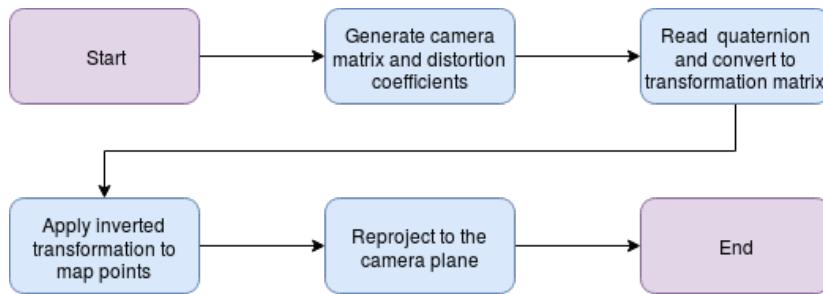
Keeping that in mind, the transformations I use are the ones of the *KeyFrameTrajectory.txt* file but inverted. To do so, I transform the quaternions into transformation matrices, inverting them and applying them to every point extracted from the frame that is being processed. The transformation from quaternion to transformation matrix is performed using an adapted version of the code from Baker (2017), in which the principles of this conversion are explained. The main point to keep in mind is that the used method assumes that the quaternion is normalised, which is the case of the used ones. Next, the inverse of a transformation matrix has the structure shown in Equation 5.1, where  $\vec{p}$  is the translation vector and  $\vec{x}$ ,  $\vec{y}$  and  $\vec{z}$  the orthogonal vectors representing the rotation.

$$T = \begin{pmatrix} x_x & y_x & z_x & p_x \\ x_y & y_y & z_y & p_y \\ x_z & y_z & z_z & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \Rightarrow T^{-1} = \begin{pmatrix} x_x & x_y & x_z & -\vec{p} \cdot \vec{x} \\ y_x & y_y & y_z & -\vec{p} \cdot \vec{y} \\ z_x & z_y & z_z & -\vec{p} \cdot \vec{z} \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (5.1)$$

A flow chart of the sub-steps followed in this part of the project is shown in Figure 5.4.

## 5.4. Depth estimation for generating the pointclouds

As it has been mentioned several times along this document, the depth estimation is done using a DL based method. In particular, I have used DeMoN, which is a deep CNN trained



**Figure 5.4:** Diagram representing the execution flow of the reprojection step.

**Source:** Own preparation

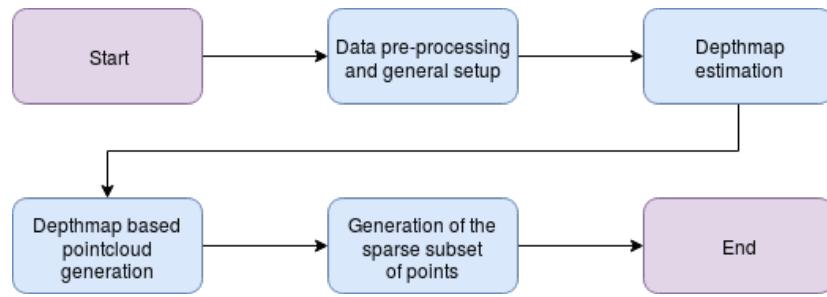
to, using two unconstrained frames of a scene, estimate a depthmap correspondent to one of them and the translation and rotation that the camera has done. In other words, it solves the SfM problem with a ML approach. It also provides some other intermediate outputs such as the surface normals, the optical flow and its quality and a scale factor. As explained in the Subsecion 4.3.2 of the Chapter 4, this is done using three CNNs based on several encoder-decoder networks in order to get an initial prediction and refine and upscale it.

DeMoN has been chosen among other alternatives because of its great performance in comparison to them. Furthermore, it was tested before integrating it in the project with images taken using different cameras in order to check its robustness when it is not working with data from the camera it was trained with.

Obviously, the main sub-part of this step is the estimation of the depthmap using the network, being this the first thing done (after some required data pre-processing and setup, such a re-escalation of the images to fit the input, a check on the channel order, the initialisation of the network, etc.). After doing it, since the objective is to generate a 3D map, the next thing to do is to convert the obtained depthmap to a pointcloud. This is done by generating the epipolar lines using the base-frame and setting where in this infinite line the point is located by using the estimated depth. It is important to highlight that to perform this pointcloud generation, the intrinsic parameters that must be utilised are the ones from the Sun3D camera, not the ones of the camera used to record the sequence that is being processed. This is due to the fact that DeMoN was trained using images from a Sun3D camera, so in order to generate the epipolar lines from DeMoN's output, the calculations must take into account its intrinsics. The pointcloud generation computation is done by the function `compute_point_cloud_from_depthmap()`, from the `depthmotionnet.vis` library.

The last thing done during this step is the generation of a subset of points of the estimated cloud. This subset will be conformed by the estimated 3D points corresponding the ones composing the O-S2 scattered map. I extract only these points by reading the files generated by the last step (Section 5.3), in which the 2D keypoints used to generate the 3D map points (obtained by reprojecting the map points) are listed. By reading the keypoints coordinates and generating another pointcloud only with the estimated depths of these pixels (and their epipolar lines), I end up building an estimation of the scattered map (but acquired using DeMoN, not O-S2).

As a summary, a flow chart regarding the execution of this step is shown in Figure 5.5.



**Figure 5.5:** Diagram representing the execution flow of the depth estimation step.

**Source:** Own preparation

## 5.5. Transformation obtaining and applying

The aim of this part is to escalate, locate and orientate the estimated pointclouds in their corresponding pose, assuming the O-S2 scattered map as the ideal condition.

In order to archive this without the time and resource consuming traditional registration pipeline (which also lacks robustness) and keeping in mind that, in order to register two clouds, correspondences are needed, the chosen approach has been to use the two scattered maps obtained until now (the O-S2 and the subset of the DeMoN ones). As the points of the demon one have been generated following the same order in which the keypoints of the reprojection output file are read and their order is the same of the points in the O-S2 map (since they were obtained reprojecting them), these sets of points are already matched if the pointclouds are read in order.

Making sure this assumption is satisfied, applying SVD (method explained in Subsection 4.1.2 of Chapter 4) to compute the needed transformation to match the sets is straightforward. This way, the errors introduced by false matches are removed, leaving as the unique errors the ones due to the usage of points badly located. These errors are not inconsiderable keeping in mind that the subset extracted from the DeMoN prediction, even its accuracy, is just an estimation, so some outliers are susceptible to appear.

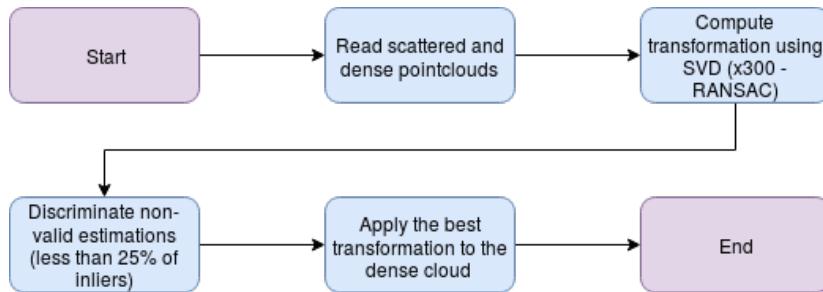
Obviously, the method that comes up to anyone's mind when trying to compute something with (unknown) outliers in the data, is RANSAC (again, explained in Subsection 4.1.2 of Chapter 4). For this particular case, I implemented it in a way that uses 10 random matches of the sets and 300 iterations. Furthermore, if none of the computed transformations (using SVD) could make the 25% of the set to be inliers, the actual keyframe was rejected and its estimated pointcloud will not be transformed or be used for the final reconstruction.

It is worth to mention that the threshold used to determine which matches were or not inliers was individually adjusted for each one of the testing datasets in order to use over the 20% of keyframes to generate *valid* pointclouds. Thanks to this approach, the mean of used keyframes to generate valid pointclouds was 18.85%, with a variance of 0.00032. Even a different threshold was chosen for each dataset, the thresholds were really close between them, being the mean one 0.0123, with a variance of 0.00002. This data can be better appreciated in Table 5.2.

As a summary, a flow chart regarding the execution of this step is shown in Figure 5.6.

	Used keyframes (%)	Threshold
freiburg1_xyz	15.75	0.0125
freiburg1_desk	16.22	0.01
freiburg1_desk2	18.92	0.02
freiburg2_xyz	21.21	0.009
freiburg2_desk	19.14	0.01
Mean	18.85	0.0123
Varinace	0.0003153	0.0000202

**Table 5.2:** Percentage of used keyframes and RANSAC threshold.



**Figure 5.6:** Diagram representing the execution flow of the transformation step.

**Source:** Own preparation

## 5.6. Refinement

Once having pointclouds of several view points of a scene, the unique thing left to do is to read every pointcloud and put all of them together. It is necessary to take into consideration the fact of redundancy. It is almost impossible to get pointclouds with no redundancy in the data they contain. If a robot is moving around a table while capturing images, it is highly probable that in two or more pictures the same content is shown, such as a part of the mentioned table, some of the objects over it, the ground, etc. This is why the applying of a spatial filter could reduce the size of the map without implying a relevant data loss.

These step consist in putting together every transformed cloud and applying over the result Voxel Grid and SOR, both explained in Subsection 4.1.2 from Chapter 4. These two filters are the ones I chose in order to improve the final results of my work in order to archive by using them are reducing the size of the final map, increase its sharpness and decrease the amount of noise (outliers).

# 6. Experimentation, Results and Discussion

In this Chapter, the experimentation methodology and details about the dataset are explained. The different benchmark metrics along the corresponding results of the experiments are also given.

## 6.1. Datasets description

The chosen RGB-D SLAM dataset and benchmark is the proposed in Sturm et al. (2012), as it is one of the main state of the art data sets to test SLAM methods. This data set contains the color and depth images of a Microsoft Kinect sensor along the ground-truth trajectory of the sensor. The data was recorded at full frame rate (30 Hz) and sensor resolution (640x480). The ground-truth trajectory was obtained from a high-accuracy motion-capture system with eight high-speed tracking cameras (100 Hz). The dataset is composed of several sequences but the evaluation of the approach was done using the following ones: freiburg1\_xyz, freiburg1\_desk, freiburg1\_desk2, freiburg2\_xyz and freiburg2\_desk. These sequences feature a range on different linear and angular velocities which will challenge the benchmarked algorithms.

In addition, several sequences of different indoor environments for qualitative evaluation were recorded. One of them was in one corridor of the building where the RoViT laboratory is located. This sequence was the selected one because the challenges it introduces, such as high contrasts, low variation in the structure and a lot of reflections and lightning flashes due to the glasses in the corridor and the windows. The results for this sequence can be seen in Chapter 6.

## 6.2. Localisation accuracy benchmark

The authors of the dataset also provide some metrics to measure the accuracy of the benchmarked methods. These metrics are thoroughly explained in their work. First, the Absolute Trajectory Error (ATE) measures the difference between points of the true and the estimated trajectory, highlighting if the final global trajectory is accurate or not. Then, Relative Pose Error (RPE) measures the error in the relative motion between pairs of time stamps and so states the accuracy of local trajectory over a fixed time interval. The implementation of both metrics are freely available by the authors of the data set and are used as provided.

The results achieved with the proposal are stated in Table 6.1. The average ATE and RPE of O-S2 are 0.011588 and 0.014281, with a variance of 0.00006 and 0.000114, respectively. Given this statistic data it is possible to conclude that the error that the KSLAM commits is over 1 cm yielding a really low variance across the testing data sets. Thus, I consider it a robust method that can provide with a trustful estimation of the camera pose.

Then, the average ATE and RPE of this proposal are 0.089858 and 0.150763, with a variance of 0.001297 and 0.012235. In spite of this data showing a bit more dependence on

	ATE (O-S2)	RPE (O-S2)	ATE (Map Slammer)	RPE (Map Slammer)
freiburg1_xyz	0.008126	0.016362	0.032302	0.032302
freiburg1_desk	0.015965	0.018959	0.104802	0.219784
freiburg1_desk2	0.022276	0.028443	0.119201	0.290972
freiburg2_xyz	0.002442	0.001433	0.077768	0.048959
freiburg2_desk	0.009132	0.006209	0.115219	0.161799
Average	0.011588	0.014281	0.089858	0.150763
Variance	0.000059	0.000114	0.001297	0.012235

**Table 6.1:** Absolute average trajectory error and average relative pose error achieved by O-S2 and this proposal.

the data set, this relationship is small enough to trust the obtained averages, which points out that translations of over 10 cm are needed to fit the point clouds.

It is also worth to mention that it was not possible to find good enough transformations for every keyframe used by O-S2, leading to some mismatches in the associations between the obtained trajectory and the ground truth, as can be appreciated in Figure 6.1.

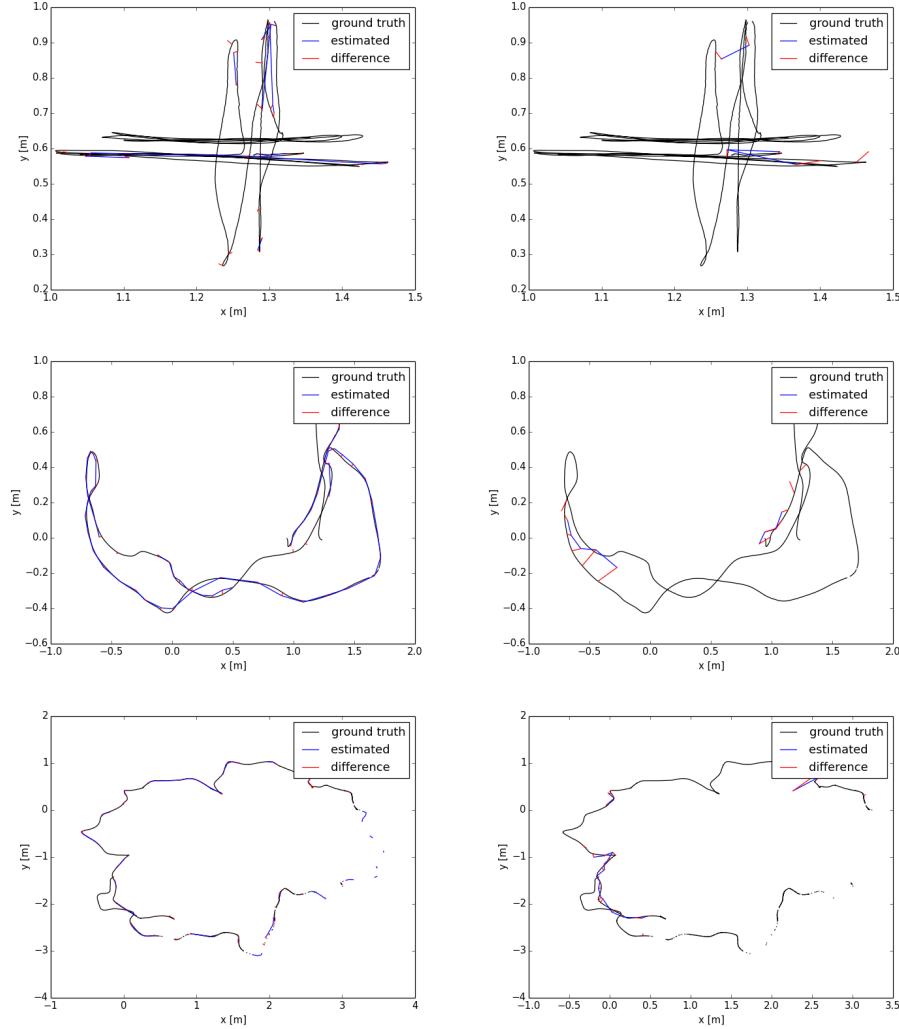
### 6.3. Densification capabilities test

One of the main contributions of this work is that it returns a dense 3D reconstruction of the environment. To measure its densification capabilities, the count of points averaged across all the frames of each sequence of the data set was computed. The results are reported in Table 6.2.

	O-S2	Map Slammer
freiburg1_xyz	315,156250	49152
freiburg1_desk	265,337838	49152
freiburg1_desk2	202,824324	49152
freiburg2_xyz	284,484848	49152
freiburg2_desk	270,320988	49152
Average	250.537375	49152
Variance	3102.91105	0

**Table 6.2:** Averaged count of points across all the frames of each sequence of the data set for O-S2 and this method.

As it can be seen in 6.2, the O-S2 approach yielded few 3D points. This is expectable as these points are the result of matching 2D key points (ORB) across a minimum of two frames over time. Regarding my method, it always extracts 49152 3D points from each frame. That is due to the performed depth estimation, in which the frames are re-sized to 256x192 and, since predicted values are gotten for each pixel, I end up having always the mentioned amount of 49152 points for each frame. This known amount of data provides not only with



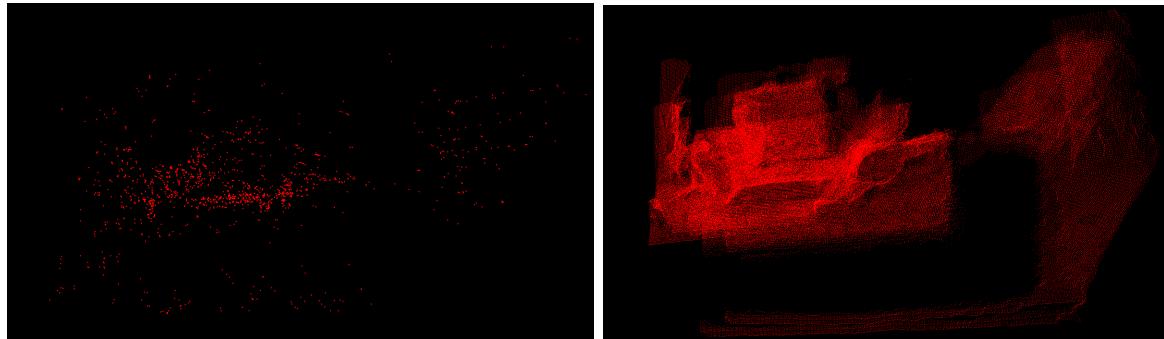
**Figure 6.1:** Ground truth (black) vs O-S2 (blue left) and the obtained (blue right) trajectories. The distances are shown in red. The representations (from left to right) correspond to fr1/xyz, fr1/desk and fr2/desk.

**Source:** Own preparation

more certainty about the outputs but with almost 200 times more 3D data that O-S2 did (the average of O-S2 provided points is 250,537375). This can be clearly appreciated in the Figure 6.2.

This fact also justifies the designed pipeline, in which the only points used are the ones provided by the KSLAM algorithm, which allows registering the estimated point clouds with much less computational requirements compared with the utilisation of a traditional registering pipeline which is based on the extraction and the description of the keypoints, and matching between them.

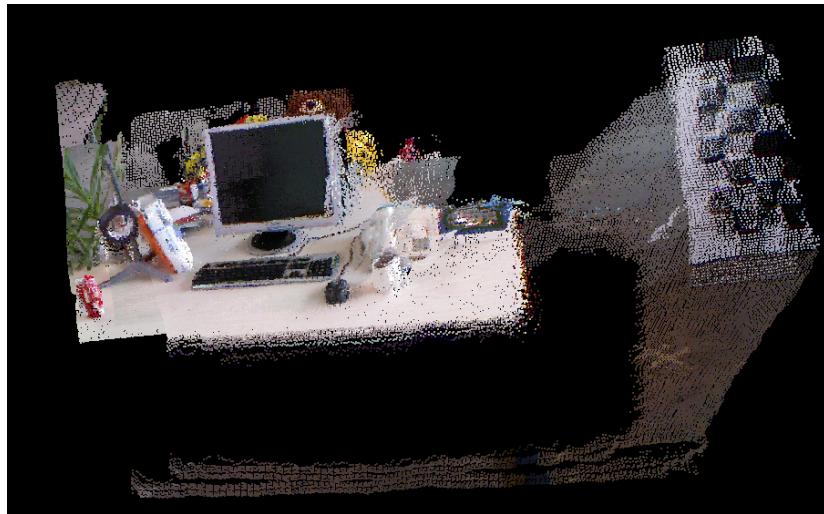
Furthermore, as the depths and, therefore, the 3D points were extracted using estimations of each pixel, it is possible to assign to each of this points its corresponding color information,



**Figure 6.2:** O-S2 scattered output (left) vs my dense one (right). This results correspond to the fr2/xyz sequence.

**Source:** Own preparation

achieving the result shown on the Figure 6.3.



**Figure 6.3:** The method's output with RGB information (from fr2/xyz).

**Source:** Own preparation

## 6.4. Accuracy of the returned 3D map

The previous metric measured the density of points. However, the goal of this approach is not only to produce a dense representation of the environment but also to do it in as accurately as possible. To measure the precision, the quality indicator is to compute the mean distance of the nearest neighbour between the full 3D representation achieved by this approach and by O-S2 (the sparse map) to the ground truth. It could have been done using the estimated point clouds, but this would have given worst results (taking into consideration the error of the predictions), so I chose to use only the points created by O-S2 to be as critical as possible

with my method. The results are shown in Table 6.3.

	ORB-SLAM2	Map Slammer
freiburg1_xyz	0.086565	0.0310728
freiburg1_desk	0.040083	0.0464012
freiburg1_desk2	0.086565	0.0626974
freiburg2_xyz	0.066125	0.0557921
freiburg2_desk	0.004647	0.0035932
Average	0.065162	0.039911
Variance	0.000474	0.000553

**Table 6.3:** Mean distance between the nearest neighbour of each point of the produced 3D representations to the ground truth representation.

As the results show, the distances to the nearest neighbour of the output of my method is between 3 and 6 cm. Being more precise, the average distance is less than 4 cm (0.039911), with a variance of 0.000553. These results show a great precision level, keeping in mind that the points are generated from an estimated depthmap.

Regarding the O-S2 related measurements, higher distances can be appreciated (between 4 and 8 cm), being 0.065162 the average one, with a reduced variance (0.000474). This demonstrates that this method not only provides almost 200 times more data than the used KSLAM method (as demonstrated in Section 6.3), but this data is also more precise.

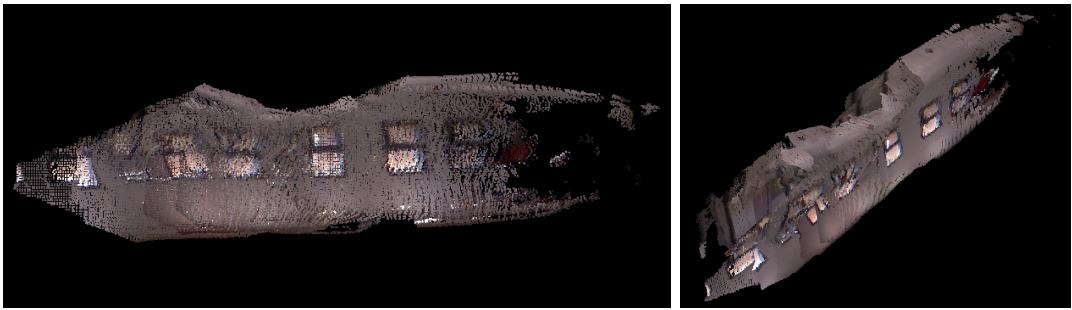


## 7. Conclusions and Future Work

In this work, a method which fuses a visual KSLAM algorithm with predicted point clouds to generate a dense three-dimensional map of the environment is proposed. The proposal also outputs the camera pose within the map. This method is monocular based, namely, it only requires a color camera to be executed.

The method achieves a localisation error of about 10 cm, which is accurate enough to be deployed in an actual robot. In addition, my approach provides about 50k points per frame, which is way more dense than the original ORB-SLAM2 visual SLAM algorithm that provides about 202 – 284 points per frame. Finally, the three-dimensional map produced by this method yields an error of about 4 cm compared with the ground truth. A video of the proposal and its result can be seen at <sup>1</sup>.

In addition, the approach was tested in different, challenging scenarios for qualitative evaluation. The result can be appreciated in Figure 7.1.



**Figure 7.1:** Output of the method using an own prepared challenging sequence, seen from the side (left) and with perspective (right).

Nonetheless, my proposal shows some limitations. First, neither the KSLAM algorithm nor the predicted point clouds yield a real world scale. Furthermore, the scale is not even consistent between two consecutive predicted point clouds. However, the KSLAM does provide a constant yet arbitrary scale. Thus, this approach provides the very same scale so it does not represent the environment in real world measure units. In addition, the system relies on the KSLAM localisation capabilities. If it fails to provide an approximately correct camera pose, my method will fail, since the KSLAM would locate the points in a wrong place, forcing the method to register the dense point cloud there.

The mentioned limitations are the main focus for the future work. The plan is to tune the SLAM localisation so it can be robust against eventual localisation disturbances. In addi-

---

<sup>1</sup><https://youtu.be/b74P3ykYE34>

tion, another improvements could be introduced by using other SLAM or depth estimation methods or even tuning the actual ones (for example, using another vocabularies for O-S2 or performing Transfer Learning with DeMoN and using the sparse maps as an additional input). Finally, including pixel-level semantic information will enable a hybrid traditional and semantic localisation. This can be done by relying in deep learning based pixel-wise classification algorithms.

## References

- Afiqah Binti Haji Yahya, N., Ashrafi, N., & Humod, A. (2014, 01). Development and adaptability of in-pipe inspection robots. *IOSR Journal of Mechanical and Civil Engineering*, 11, 01-08. doi: 10.9790/1684-11470108
- A. Fischler, M., & C. Bolles, R. (1981, 06). Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24, 381-395. doi: 10.1145/358669.358692
- Baker, M. J. (2017). *Maths - conversion quaternion to matrix*. <http://www.euclideanspace.com/mathematics/geometry/rotations/conversions/quaternionToMatrix/index.htm>.
- Bay, H., Tuytelaars, T., & Van Gool, L. (2006). Surf: Speeded up robust features. In *European conference on computer vision* (pp. 404–417).
- Bhoi, A. (2019). Monocular depth estimation: A survey. *CoRR*, abs/1901.09402. Retrieved from <http://arxiv.org/abs/1901.09402>
- ComputerVisionGroup-TUM. (n.d.). *File formats*. [https://vision.in.tum.de/data/datasets/rgbd-dataset/file\\_formats](https://vision.in.tum.de/data/datasets/rgbd-dataset/file_formats).
- dave\_mmm0. (2012). *Svd-based estimation vs levenberg marquardt-based*. <http://wwwpcl-users.org/SVD-based-estimation-vs-Levenberg-Marquardt-based-td4020253.html>.
- Deng, C., Wang, S., Huang, Z., Tan, Z., & Liu, J. (2014). Unmanned aerial vehicles for power line inspection: A cooperative way in platforms and communications. *JCM*, 9, 687-692.
- Eggert, D., Lorusso, A., & Fisher, R. (1997, Mar 01). Estimating 3-d rigid body transformations: a comparison of four major algorithms. *Machine Vision and Applications*, 9(5), 272–290. Retrieved from <https://doi.org/10.1007/s001380050048> doi: 10.1007/s001380050048
- Eigen, D., Puhrsch, C., & Fergus, R. (2014). Depth map prediction from a single image using a multi-scale deep network. In *Advances in neural information processing systems* (pp. 2366–2374).
- Hisham, M., Yaakob, S. N., Raof, R. A., Nazren, A. A., & Embedded, N. W. (2015). Template matching using sum of squared difference and normalized cross correlation. In *2015 ieee student conference on research and development (scored)* (pp. 100–104).
- Jirbandey, A. (2018). *A brief history of computer vision and ai image recognition*. <https://www.pulsarplatform.com/blog/2018/brief-history-computer-vision-vertical-ai-image-recognition/>.

- Klein, G., & Murray, D. (2007). Parallel tracking and mapping for small ar workspaces. In *Proceedings of the 2007 6th ieee and acm international symposium on mixed and augmented reality* (pp. 1–10).
- Laina, I., Rupprecht, C., Belagiannis, V., Tombari, F., & Navab, N. (2016). Deeper depth prediction with fully convolutional residual networks. In *2016 fourth international conference on 3d vision (3dv)* (pp. 239–248).
- Linder, T., Tretyakov, V., Blumenthal, S., Molitor, P., Holz, D., Murphy, R., ... Surmann, H. (2010, July). Rescue robots at the collapse of the municipal archive of cologne city: A field report. In *2010 ieee safety security and rescue robotics* (p. 1-6). doi: 10.1109/SSRR.2010.5981550
- Longuet-Higgins, H. C. (1981). A computer algorithm for reconstructing a scene from two projections. *Nature*, 293(5828), 133.
- Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2), 91–110.
- MIT-OpenCourseWare. (2016). *Singular value decomposition (the svd)*. <https://www.youtube.com/watch?v=mBcLRGuAFUk>.
- Mur-Artal, R., Montiel, J. M. M., & Tardos, J. D. (2015). Orb-slam: a versatile and accurate monocular slam system. *IEEE transactions on robotics*, 31(5), 1147–1163.
- Mur-Artal, R., & Tardos, J. D. (2015). Probabilistic semi-dense mapping from highly accurate feature-based monocular slam.
- OpenCV-Documentation. (n.d.). *Camera calibration and 3d reconstruction*. [https://docs.opencv.org/2.4/modules/calib3d/doc/camera\\_calibration\\_and\\_3d\\_reconstruction.html](https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html).
- OpenCV-Documentation. (2013). *Feature matching*. [https://docs.opencv.org/3.0-beta/doc/py\\_tutorials/py\\_feature2d/py\\_matcher/py\\_matcher.html](https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_matcher/py_matcher.html).
- OpenCV-Documentation. (2014a). *Brief (binary robust independent elementary features)*. [https://opencv-python-tutroals.readthedocs.io/en/latest/py\\_tutorials/py\\_feature2d/py\\_brief/py\\_brief.html](https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_brief/py_brief.html).
- OpenCV-Documentation. (2014b). *Common interfaces of descriptor extractors*. [https://docs.opencv.org/2.4.10/modules/features2d/doc/common\\_interfaces\\_of\\_descriptor\\_extractors.html](https://docs.opencv.org/2.4.10/modules/features2d/doc/common_interfaces_of_descriptor_extractors.html).
- OpenCV-Documentation. (2014c). *Fast algorithm for corner detection*. [https://docs.opencv.org/3.0-beta/doc/py\\_tutorials/py\\_feature2d/py\\_fast/py\\_fast.html](https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_fast/py_fast.html).
- OpenCV-Documentation. (2014d). *Orb (oriented fast and rotated brief)*. [https://docs.opencv.org/3.0-beta/doc/py\\_tutorials/py\\_feature2d/py\\_orb/py\\_orb.html](https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_orb/py_orb.html).
- OpenCV-Documentation. (2015). *Feature detection and description*. [https://docs.opencv.org/3.1.0/db/d27/tutorial\\_py\\_table\\_of\\_contents\\_feature2d.html](https://docs.opencv.org/3.1.0/db/d27/tutorial_py_table_of_contents_feature2d.html).

- Paulins. (2016). *Open drone map project*. [https://github.com/paulinus/OpenDroneMap/blob/orb\\_slam2/modules/odm\\_slam/src/OdmSlam.cpp](https://github.com/paulinus/OpenDroneMap/blob/orb_slam2/modules/odm_slam/src/OdmSlam.cpp).
- raulmur. (2017). *Orb-slam2 repository*. [https://github.com/raulmur/ORB\\_SLAM2](https://github.com/raulmur/ORB_SLAM2).
- Raziq Asyraf Md Zin, M., MD Saad, J., Anuar, A., Talip Zulkarnain, A., & Sahari, K. (2012, 12). Development of a low cost small sized in-pipe robot. *Procedia Engineering*, 41, 1469–1475. doi: 10.1016/j.proeng.2012.07.337
- ROS-Wiki. (2015). *Ros pepper Bringup package*. [http://wiki.ros.org/pepper\\_bringup](http://wiki.ros.org/pepper_bringup).
- ROS-Wiki. (2017). *Ros camera Calibration package*. [http://wiki.ros.org/camera\\_calibration](http://wiki.ros.org/camera_calibration).
- ROS-Wiki. (2018). *Ros master*. <http://wiki.ros.org/Master>.
- Rublee, E., Rabaud, V., Konolige, K., & Bradski, G. R. (2011). Orb: An efficient alternative to sift or surf. In *Iccv* (Vol. 11, p. 2).
- Strasdat, H., Montiel, J., & Davison, A. J. (2010). Real-time monocular slam: Why filter? In *2010 ieee international conference on robotics and automation* (pp. 2657–2664).
- Sturm, J., Engelhard, N., Endres, F., Burgard, W., & Cremers, D. (2012, Oct.). A benchmark for the evaluation of rgb-d slam systems. In *Proc. of the international conference on intelligent robot systems (iros)*.
- SuperDataScience-Team. (2018). *Convolutional neural networks (cnn): Step 1- convolution operation*. <https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-1-convolution-operation>.
- Tateno, K., Tombari, F., Laina, I., & Navab, N. (2017). CNN-SLAM: real-time dense monocular SLAM with learned depth prediction. *CoRR, abs/1704.03489*. Retrieved from <http://arxiv.org/abs/1704.03489>
- Tch, A. (2017). *The mostly complete chart of neural networks, explained*. <https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464>.
- Torres-Camara, J. M. (2019). *Lanzamiento de orb-slam con pepper*. [http://www.rovit.ua.es/wiki/index.php/Lanzamiento\\_de\\_ORB-SLAM\\_con\\_Pepper](http://www.rovit.ua.es/wiki/index.php/Lanzamiento_de_ORB-SLAM_con_Pepper).
- Torres-Camara, J. M., Escalona, F., Gomez-Donoso, F., & Cazorla, M. (2019). *Map slammer: Densifying scattered kslam 3d maps with estimated depth*.
- Ummenhofer, B., Zhou, H., Uhrig, J., Mayer, N., Ilg, E., Dosovitskiy, A., & Brox, T. (2016). Demon: Depth and motion network for learning monocular stereo. *CoRR, abs/1612.02401*. Retrieved from <http://arxiv.org/abs/1612.02401>
- Ummenhofer, B., Zhou, H., Uhrig, J., Mayer, N., Ilg, E., Dosovitskiy, A., & Brox, T. (2017). Demon: Depth and motion network for learning monocular stereo. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 5038–5047).

- VisionOnline. (2018). *What is visual slam technology and what is it used for?* <https://www.visiononline.org/blog-article.cfm/What-is-Visual-SLAM-Technology-and-What-is-it-Used-For/99>.
- Wikipedia-contributors. (2019a). *Camera obscura* — wikipedia, the free encyclopedia. [https://es.wikipedia.org/w/index.php?title=Camera\\_Obscura&oldid=113071546](https://es.wikipedia.org/w/index.php?title=Camera_Obscura&oldid=113071546).
- Wikipedia-contributors. (2019b). *Epipolar geometry* — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/w/index.php?title=Epipolar\\_geometry&oldid=902879743](https://en.wikipedia.org/w/index.php?title=Epipolar_geometry&oldid=902879743).
- Younes, G., Asmar, D., Shammas, E., & Zelek, J. (2017). Keyframe-based monocular slam: design, survey, and future directions. *Robotics and Autonomous Systems*, 98, 67–88.

# Acronyms and abbreviations list

<b>AGV</b>	Autonomous Ground Vehicle.
<b>AI</b>	Artificial Intelligence.
<b>ATE</b>	Absolute Trajectory Error.
<b>BoW</b>	Bag of Words.
<b>CNN</b>	Convolutional Neural Network.
<b>CRF</b>	Conditional Random Field.
<b>DeMoN</b>	Depth and Motion Network.
<b>DL</b>	Deep Learning.
<b>EKF</b>	Extended Kalman Filter.
<b>ERDF</b>	European Regional Development Fund.
<b>ICP</b>	Iterative Closest Point.
<b>IMU</b>	Inertial Movement Unit.
<b>KSLAM</b>	keyframe-based SLAM.
<b>LIDAR</b>	LIght Detection And Ranging.
<b>ML</b>	Machine Learning.
<b>NCC</b>	Normalised Cross Correlation.
<b>NN</b>	Neural Network.
<b>O-S</b>	ORB-SLAM.
<b>O-S2</b>	ORB-SLAM2.
<b>OpenCV</b>	Open Computer Vision library.
<b>PCL</b>	Point Cloud Library.
<b>PTAM</b>	Parallel Tracking And Mapping.
<b>RANSAC</b>	Random Sample Consensus.
<b>RGB</b>	Red-Green-Blue.
<b>RGB-D</b>	RGB-Depth.
<b>ROS</b>	Robotics Operating System.
<b>RoViT</b>	Robotics and Three-dimensional Vision.
<b>RPE</b>	Relative Pose Error.
<b>SfM</b>	Structure from Motion.
<b>SLAM</b>	Simultaneous Localisation And Mapping.
<b>SOR</b>	Statistical Outlier Removal.
<b>SSD</b>	Sum of Squares Differences.
<b>SVD</b>	Singular Value Decomposition.
<b>ToF</b>	Time of Flight.

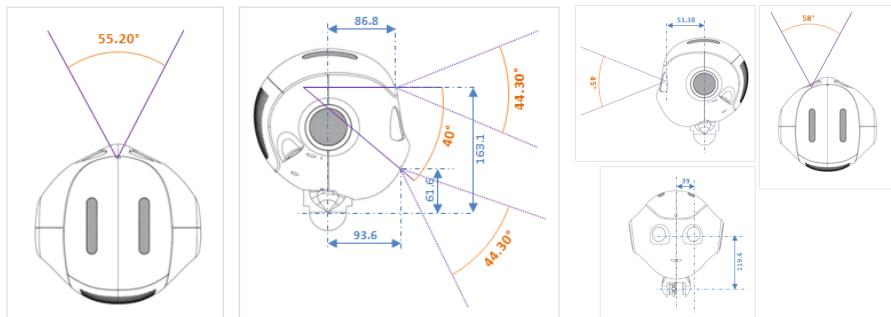
**VSLAM** Visual SLAM.

## A. Annexed I. ORB-SLAM2 usage tutorial

O-S2 is the evolution of O-S, being both projects of the University of Zaragoza. It implements a KSLAM method available to work with monocular, RGB-D and stereo cameras. Since it is the version used in this project, I will focus in the monocular one. As explained in Section 2.1 from Chapter 2, it uses ORB to extract and describe features to discretise the obtained descriptors using a bag of words in order to increase the performance even more, since it is aimed to provide a real-time KSLAM algorithm.

Regarding the implementation, it can be ran both using Robotics Operating System (ROS) or with a C++ program. Examples of both of them are provided in their public repository (raulmur, 2017). Now, even the main content of this project was developed using the C++ implementation and offline datasets, I will describe how the ROS version can be launched using a real robot.

The used robot for the process was the Pepper, a humanoid mobile platform provided with two monocular cameras (in the front and in the mouth) and with one RGB-D 3D sensor (in the eyes), as shown in Figure A.1.



**Figure A.1:** Monocular (left) and RGB-D (right) cameras in Pepper.

Sources: Aldebaran documentation - 2D cameras and 3D camera

When executing O-S2 using real-time data, it is possible to use two possible workflows: compute everything in the robot (being limited by its reduced computational resources) or use the robot only to provide the data and carry the computation in a remote server. These both cases are detailed in Torres-Camara (2019), where I explain how to do it step by step. However, since the less limiting approach (and the more *complex*) is the one using the robot only as a data source, I will focus on this one.

The steps to follow are:

- **Configure the ROS environment:** Define the IP and port to look for the ROS master.

- **Configure and launch Pepper:** Configure the camera (resolution, framerate, etc.) and launch the robot.
- **Execute O-S2:** Select the vocabulary for the BoW and the settings file (which contain the intrinsic parameters and information about the camera configuration and settings for the algorithm).

Regarding the ROS configuration, it is necessary to configure where to look for the ROS master (ROS-Wiki, 2018). This is done by configuring the environment variable of the GNU/Linux ecosystem named *ROS\_MASATER\_URI*. It should contain the IP address and the port where the ROS master will be being executed. By default, the port used by ROS is 11311 and the IP will be the one of our server. Some examples of how to configure this using the command line would be the ones in Listing A.1.

Listing A.1: ROS master configuration examples

```
1 export ROS_MASTER_URI=http://localhost:11311
2 export ROS_MASTER_URI=http://172.18.33.122:11311
```

Next, the Pepper configuration has to be done. O-S2 uses images with VGA resolution (640x480), while Pepper, by default, provides them with qVGA (320x240), even its hardware is able to provide the required resolution. That is why the configuration file in which this settings are specified should be edited. In case the Pepper robot would have a *normal* installation of ROS, this file will be in the *naoqi\_driver* package. However, the Pepper in the Robotics and Three-dimensional Vision (RoViT) laboratory uses a wrapper to integrate the robot in ROS. In this particular installation, the package is located in */home/nao/.ros-root/ros1\_inst/share/naoqi\_driver/*.

Regardless of the case, once in the package directory, the next step is to open the json file *boot\_config.json*, located in the subdirectory *share*. It will have an structure similar to the one in Listing A.2.

Listing A.2: boot\_config.json file extract

```
1 "_comment": "QQVGA = 0, QVGA = 1, VGA = 2",
2 "converters":
3 {
4     "front_camera":
5     {
6         "enabled" : true,
7         "resolution" : 1,
8         "fps" : 10,
9         "recorder_fps" : 15
10    }, [...]
```

The referred extract is obviously related to the front camera, but there are configuration options for the bottom and the depth ones, the Inertial Movement Unit (IMU), the odometry, the tactile sensors, etc.

As it has been mentioned, the wanted configuration is VGA resolution. Furthermore, the more framerate available, the better. This was one of the problems found with the usage of a remote server. The maximum rates I reached was 8 fps using Wi-Fi and 10 using a wired connection. However, when computing everything from inside the robot, the rate could easily

---

raise to 20 and 30, probably because the implementation of the communications, given that I tried to use a private net to avoid the security of the one of the University.

The next step is to launch the Pepper robot, what is done using the *pepper\_bringup* package (ROS-Wiki, 2015). This brings up the robot in the ROS ecosystem and is a process that requires the robot IP and the ROS master one. It is done by executing the command shown in Listing A.3 (assuming the robot's IP is *172.18.33.122* and the ROS master's one is *172.18.33.87*).

Listing A.3: Pepper bringup command

```
1 roslaunch pepper_bringup pepper_full.launch nao_ip:="172.18.33.122" roscore_ip←
  ↪ :="172.18.33.87"
```

Finally, the only thing left is to launch O-S2. This step requires to know the mode (Mono, RGBD or Stereo), the path to the BoW vocabulary and the path to the camera and algorithm settings. The researchers from the University of Zaragoza provides settings for the cameras used in some popular datasets (TUM, KITTI and EuRoC). However, in order to use the Pepper cameras, I needed to calibrate it to get an estimation of its intrinsic parameters. I did so by using the *camera\_calibration* package in ROS (ROS-Wiki, 2017). The command to run the node to calibrate the front RGB camera of the Pepper is exposed in Listing A.4 (assuming a chessboard calibration pattern with 6x4 corners separated by 0.03175 meters) and the command to execute the KSLAM is the one shown in Listing A.5.

Listing A.4: Camera calibration node launching command

```
1 rosrun camera_calibration cameracalibrator.py --size 6x4 --square 0.03175 ←
  ↪ image:=/pepper_robot/naoqi_driver/camera/front/image_raw camera:=/←
  ↪ pepper_robot/naoqi_driver/camera/front --no-service-check
```

Listing A.5: Monocular ORB-SLAM2 launching command

```
1 rosrun ORB_SLAM2 Mono PATH_TO_VOCABULARY PATH_TO_SETTINGS_FILE
```



## B. Annexed II. Summary

### B.1. Motivation and concept

There are a range of small-size robots that cannot afford to mount a three-dimensional sensor due to energy, size or power limitations. However, the best localisation and mapping algorithms and object recognition methods rely on a three-dimensional representation of the environment to provide enhanced capabilities.

Keeping in mind this situation, in this work a method to generate three-dimensional representations of the environment is proposed. This is done by fusing the output of a keyframe-based visual SLAM (KSLAM) with depthmaps estimations obtained by a Deep-Learning based method. Furthermore, it will be demonstrated with quantitative and qualitative results the advantages of this method, focusing in three different measures: localisation accuracy, densification capabilities and accuracy of the resultant three-dimensional map.

### B.2. Approach

The method starts generating the 3D points with ORB-SLAM2, to continue reprojecting them to the image plane from which they were extracted using the intrinsic camera parameters, the pinhole camera model and the distortion coefficients. Next, the Depth and Motion Network (DeMoN) is used to generate depthmaps that are used to generate dense pointclouds estimated using pair of frames and, making use of the ORB-SLAM2 points reprojections, the sub-set of DeMoN points corresponding to the scattered map obtained with the KSLAM algorithm is obtained.

The next step is to use the two sparse maps (the ORB-SLAM2 output and the sub-set of the DeMoN estimation) to apply Singular Value Decomposition (SVD), what makes it possible to adjust the position, orientation and scale of the dense cloud to place it in its correspondent pose in the map. SVD is applied several times as part of a RANdom SAmples Consensus (RANSAC) based adjustment, what makes it possible to discriminate the pointclouds that could not be properly located (those in which the inliers are less than the 25% of the points).

Once the pointclouds are "registered", a voxel grid and a Statistical Outlier Removal (SOR) are applied in order to archive sharper and lighter results with less noise and redundancy.

### B.3. Results

All this process ends up obtaining almost 200 times more data of each frame than ORB-SLAM2 did. The dense maps archived a reduction of the error of the KSLAM method of an approximately 33% (sacrificing temporal and computational cost).

A video of the proposal and its result can be seen at <sup>1</sup>. The source files of this work is publicly available in my GitHub profile, in the repository *mapSlammer*<sup>2</sup>.

---

<sup>1</sup><https://youtu.be/b74P3ykYE34>

<sup>2</sup><https://github.com/jmtc7/mapSlammer>

---

## C. Annexed III. Resumen en Español

### C.1. Motivación y concepto

Hay muchos robots de tamaño reducido en los que no es viable incorporar sensores tridimensionales debido a las limitaciones relativas al consumo eléctrico, el tamaño, la masa, etc. Sin embargo, la mayoría de los mejores algoritmos de localización y mapeado y métodos de reconocimiento de objetos se basan en representaciones tridimensionales del entorno.

Teniendo en cuenta esta situación, en este trabajo se propone un método para generar representaciones 3D del entorno combinando la salida de un algoritmo de *keyframe-based visual SLAM* (KSLAM) con estimaciones de *depthmaps* basadas en *Deep Learning*. También se demuestra cualitativa y cuantitativamente el resultado satisfactorio y las ventajas de este método. Esto se realiza en base tres medidas: La precisión en la localización, la capacidad de densificación y la precisión del mapa tridimensional resultante.

### C.2. Desarrollo

El método comienza generando los puntos 3D con ORB-SLAM2, para continuar reproyectándolos al plano imagen del que fueron extraídos usando los parámetros intrínsecos de la cámara. Después, se hace uso de DeMoN (Depth and Motion Network) para generar mapas de profundidad a partir de los que se obtienen nubes de puntos densas estimadas utilizando los *frames* a pares y, utilizando las reproyecciones de los puntos de ORB-SLAM2, el subconjunto de puntos de DeMoN que le corresponde al mapa disperso generado al principio.

Se usan los mapas dispersos de ORB-SLAM2 y DeMoN para aplicar *Singular Value Decomposition* (SVD), lo que permite ajustar la posición, orientación y escala de la nube densa y colocarla en su lugar correspondiente del mapa. SVD se aplica reiteradas veces como parte de un ajuste por *Random Sample Consensus* (RANSAC), lo que permite discriminar las nubes de puntos que no logren ajustarse suficientemente bien (aquellas que no logren que un 25% de sus puntos sean *inliers*).

Una vez "registradas" todas las nubes de puntos, se aplica un *voxel grid* y un *Statistical Outlier Removal* (SOR) para conseguir resultados más definidos, ligeros y con menos ruido y redundancia.

### C.3. Resultados

Todo este proceso lleva a obtener casi 200 veces más datos de cada *frame* de los que extrae ORB-SLAM2. Los mapas densos logran una reducción del error respecto a los del método de KSLAM de aproximadamente un 33% (sacrificando coste temporal y computacional).

Un video del resultado de la propuesta puede verse en <sup>1</sup>. Todo el código está disponible públicamente en <sup>2</sup>.

---

<sup>1</sup><https://youtu.be/b74P3ykYE34>

<sup>2</sup><https://github.com/jmtc7/mapSlammer>

---