Berner
Fachhochschule

ORB Slam Point Cloud generation on Apalis iMX8

Stefan Eichenberger

Januar 2019

Marcus Hudritsch
TSM CPVR Lab, BFH

Abstract

For navigating a robot around obstacles we require a map of its environment. Because robots can be small and independent of power plugs, they have limited processing power. This work analyzes how suitable ORB SLAM is for an embedded processor and what improvements are required to make it work on such a platform.

Executive Summary

Simultaneous Location and Mapping (SLAM) is an important topic in computer vision today. The possibilities for this technology are infinite. We can use SLAM for navigation, object recognition and augmented reality. Often this algorithms need a high performance graphics card or a high performance processor. For industrial and robotics purposes we have constraints in space, temperature and power consumption. This doesn't allow the use of high performance CPUs. This work analyzes the ORB SLAM algorithm on how it performs on a modern embedded ARM CPU. The long-term goal is to have a system available that generates a map and to use this map for further navigation.

Contents

Chapter 1

Introduction

To find the position of a robot we can use different sensors. Navigation systems use IMUs, magneto meters, radio signals and for outdoor navigation GPS. Humans however, do a lot of navigation by eye. Embedded systems today have enough processing power to make complex calculations. Therefore, a possibility to estimate the position of the robot can also be computer vision. One field in computer vision is SLAM where we try to create a map of the environment and to estimate the pose of the camera based on camera images. In this project, we analyze the performance of ORB SLAM [2] running on an iMX8 QuadMax as embedded system. We use a stereo camera to create a point cloud and use this point cloud to calculate the changes in pose and position. As we will see, by using a stereo camera we can reconstruct a point cloud with a known scale. This allows us to create a 3D map with real world scale of the environment around the camera.

1.1 Apalis iMX8 QuadMax

Toradex supports this project with embedded hardware. Toradex is a system on module provider focusing on embedded ARM devices for industrial applications. They are interested to see some applications and demos running on the Apalis iMX8 QuadMax (figure 1.1). Therefore, we try to port the SLAM application onto this platform. One area where NXP, the manufacturer of the processor, tries to focus the iMX8 is computer vision in industry and automation. Therefore, this CPU should be a good fit for this kind of application. NXP launches different iMX8 variants. Publicly available today is only the iMX8M which is performance wise placed between the iMX8 and the iMX8X. The Apalis iMX8 QuadMax used in this project is the processor with highest performance. They will release it in 2019. This work uses an alpha silicon of the processor which is provided by Toradex.

Figure 1.1: Apalis iMX8 from Toradex

1.1.1 Features

The iMX8 QuadMax has the following specification:

- Two Cortex A72 high performance processors
- Four Cortex A53 low power processors
- Two Cotex M4 realtime MCUs
- Two Vivante GC7000 GPUs each with 8 cores with each capable to process vectors of length 4
- Industrial temperature range from -40°C to 85°C
- 16GB eMMC Flash
- 4GB LPDDR4 RAM

The iMX8 is focusing on industrial applications. Therefore, it doesn't use state-of-the-art production processes. They produce the iMX8 in 28nm structure while Smartphone processors today use 7nm technology. However, bigger structures make the processor less prone to high temperatures. The two Cortex A72, the four Cortex A53 and the Vivante GPU have the performance level of a today's mid-end Smartphone. The two Cortex M4 processors could be used to do Realtime processing. However, we don't use them in this work.

1.2 Goals

We can describe the goal of this project as follows:

- Porting ORB SLAM to the Apalis iMX8 QuadMax
- Analyzing different possibilities to speed up the ORB SLAM algorithm
- Add an extension that creates a dense point cloud which could later be used to extract detailed maps
- Allow SLAM in realtime with min. 20fps to track fast movements
- Evaluate the results and start discussion about further work

The focus of this work is to figure out how we can use or implement a SLAM algorithm that later could reconstruct the world around the camera in 3D. This 3D world or 3D point cloud could in a later step, that is not part of this work, used to plan paths around obstacles and to estimate the position of a camera in a known environment.

1.3 Time Plan

The project was planned to end in early January. The concept phase started well. Unfortunately, standard ORB SLAM didn't perform well for our purposes. First optimizations only improved the performance a little. However, we implemented a dense point cloud generator based on this implementation to figure out what challenges we have to fight there. The time plan of this project is shown in figure 1.2.

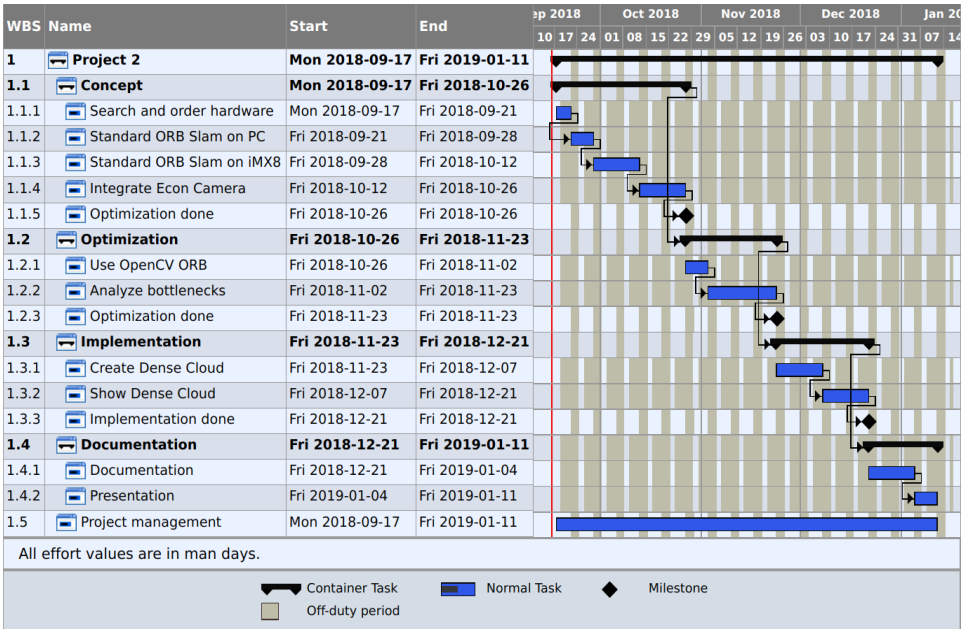


Figure 1.2: Time Plan

Chapter 2

SLAM Evaluation

In this section we analyze different documented SLAMs. We can split them in two groups, indirect and direct SLAM (figure 2.1). Indirect methods analyze an image and try to extract features. This feature points are matched directly to further images. Based on these matches we can estimate the pose of the camera. Direct methods operate on intensity variances. The goal is to find a pose that minimizes the intensity difference between two images. Because minimizing the intensity difference over the whole image is computational expensive most methods operated only on edges or corners of the image. We call such methods direct semi dense or direct sparse. In the next sections we will have a discussion about these different approaches.

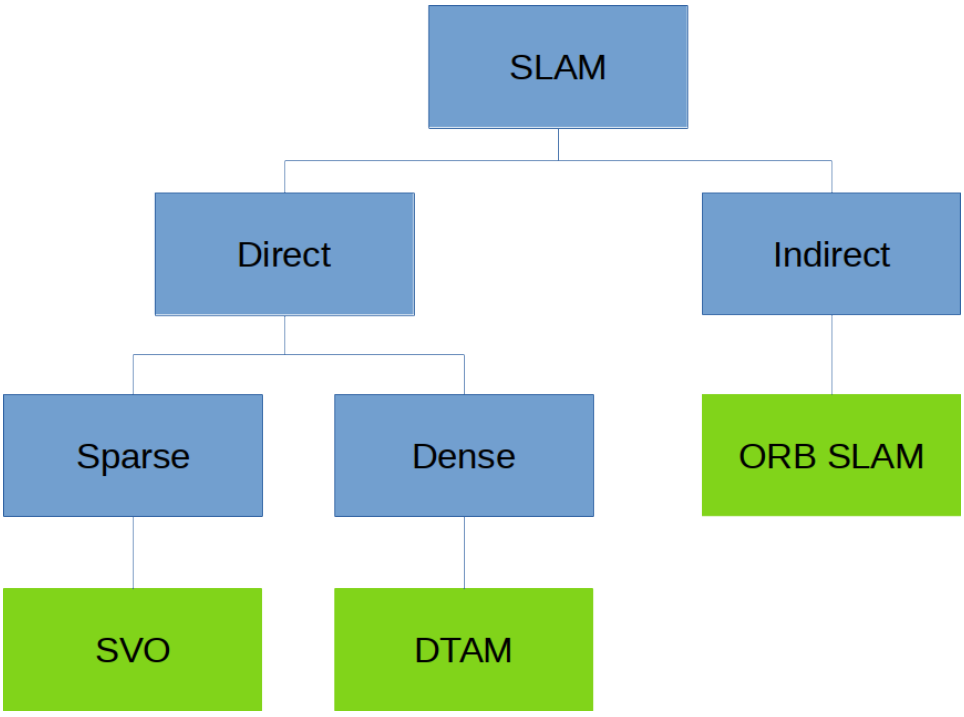


Figure 2.1: SLAM Modes

2.1 Indirect method

A well documented indirect method is ORB SLAM [2]. ORB SLAM extracts ORB keypoints which are then matched with a known 3D point cloud. Based on these matches it can do PnP with RANSAC [19] to estimate the current pose and position. Indirect methods can estimate the pose through PnP which isn't computationally expensive. On the other side computing descriptors costs CPU time. We will describe ORB SLAM in more detail in chapter 5.

2.2 Direct method

In this section we will analyze two direct approaches a dense and a sparse approach. Dense methods use all pixels on the image for tracking while sparse methods only use a subset. Figure 2.2 shows a comparison of dense sparse and semi-dense approaches. We don't dig into semi-dense approaches they work similar to sparse methods but with more keypoints.

(a) Sparse (b) Semi-Dense (c) Dense

Figure 2.2: Comparison of sparse, dense and semi-dense approaches [5]

2.2.1 Dense

There are not that many dense SLAMs. One is DTAM [4] which uses the intensity values of the whole image to estimating the pose. The idea is to minimize the intensity difference (energy) between two images by optimizing the camera pose. The energy defined at one pixel position is defined as shown in equation 2.1. To estimate the pose and position it tries to find a projection matrix that minimizes E_t as shown in equation 2.2.

$$\begin{aligned} E_{xy} &= I_1(x,y) - I_2(x',y') \\ \begin{pmatrix} x' * s' \\ y' * s' \\ s' \end{pmatrix} &= P * XYZ \end{aligned} \tag{2.1}$$

Where:

E_{xy} : Energy at a specific position
I_1 : Previous image
I_2 : Current image
x, y : Point position in previous image
x', y' : Point position in current image
s' : Scaling value
P : Projection matrix
XYZ : Point position in 3D cloud

$$E_t = \min(\sum_{x=0}^{C} \sum_{y=0}^{R} E_{xy}) \tag{2.2}$$

Where:

E_t : Total energy
E_{xy} : Energy at pixel position x/y in image
C : Image columns
R : Image rows

Direct dense method deliver a dense point cloud which we can use directly to create maps and 3D objects. However, they are computational expensive. Therefore, they are not appropriate for embedded devices.

2.2.2 Sparse

Sparse direct SLAMs like SVO [5] don't optimize the Energy over the whole image but the energy at sparse points instead. One possibility of a sparse point is e.g. a corner point found by FAST [6]. Similar to sparse methods there are semi dense methods which try to minimize the energy on several points laying on edges. Canny Edge detection or DoG can for example find such edges. The advantage of sparse and semi dense methods is that they are less computationally expensive than dense methods.

Sparse direct methods don't create dense clouds immediately however they can work on weaker keypoints than indirect methods. Therefore, they can create denser clouds than e.g. ORB SLAM. They can also be computational less expensive than indirect methods because they don't have to calculate expensive features [5].

2.3 Stereo and Monocular SLAM

In this project, the focus is on stereo SLAM. A lot of today's paper focus on monocular SLAM. The reasons are that monocular cameras are cheaper and monocular cameras are available in Smartphones. Monocular SLAM has two problems we don't have with stereo SLAM. First an initialization process has to estimate the movement between two frames. We can only create a 3D point cloud based on two images with known position, in the monocular case the position of two images is random. Therefore, the algorithm needs to guess the pose and transformation over a few images until it knows the poses. It does that by making assumptions on the world or by trying to optimizing the point positions over several images. However, even when successfully initialized the second problem persists. Monocular SLAM is not able to guess the scale of the point cloud. The intuition for this issue is that we can't distinguish between the movement of the camera on a small model close to the scene or the movement of the camera on the original model further away from the scene. With stereo SLAM we don't have this issue. For each camera pose we get two images with a known distance between the two camera sensors (baseline). Therefore, we can calculate the depth of each pixel in real world distance. From this we can generate a point cloud assuming that the initial camera pose is at origin. We therefore have an instantaneous initialization and can calculate the real world position in a known unit e.g. meters. Tracking between two camera poses is however the same on monocular and stereo SLAM. However we can again insert new keypoints immediately on stereo cameras without the need of using a second camera pose.

2.4 Decision

In this project, we port ORB SLAM onto iMX8 and see how it performs. We use ORB SLAM because of the following reasons:

- Open source implementation available
- Powerful framework for displaying point cloud and images
- Good documentation
- Indirect methods seem promising because PnP is fast
- The CPVR lab already has experience with ORB on Smartphones

Based on ORB SLAM we try to analyze how we can use the iMX8 and whether ORB SLAM fits well for this platform.

Chapter 3

Camera Evaluation

To do stereo computer vision, we need a stereo camera. There are a few stereo cameras available on the market. It is also possible to build a stereo camera with two monocular cameras. However, we decided against that because it would require a mechanism to synchronize the images from both streams. We analyze three stereo cameras available on the market.

3.1 ZED

The ZED camera [9] is a stereo camera from Stereo Labs. Here the specification of this camera:

- Full HD color images with 30fps
- USB3.0
- UVC compliant
- Linux SDK
- Baseline 120mm
- IMU sensor with 6DoF
- Price: 449$

To run their SDK a Dual Core CPU with 2.3 GHz and CUDA $>$ 3.0 is required. However, it's also possible to use the camera without their SDK.

3.2 ECON Tara

The Tara camera [10] from Econ is another stereo camera. In comparison to the ZED camera it delivers a reduced resolution and only gray scale images.

- 752x480 gray scale images with 60fps
- USB3.0
- Global Shutter Camera
- UVC compliant

- Baseline 60mm
- IMU sensor with 6DoF
- Open Source SDK
- Price: 149$

It has a small OpenCV based SDK with open source software. In comparison to the ZED SDK it is less powerful.

3.3 Intel RealSense D435

The Intel RealSense camera [11] is a infrared stereo camera. In comparison to RGB stereo cameras it uses infrared images for stereo vision. It seems that this sensors are less prone to reflections. A third camera delivers RGB images in full HD.

- 1280x720 with 90fps (IR), 1920x1080 with 30fps (RGB)
- USB3.0
- Global Shutter Camera
- UVC compilant (kernel patch required)
- Baseline 50mm
- Open Source SDK (librealsense)
- Price: 179$

The Intel SDK seems quite powerful and is open source as well. However, the indepth details of the camera are not publicly available.

3.4 Decision

We decided to use the Econ Tara because it has an outstanding price ratio and because it is a conventional stereo camera. However, the Intel RealSense camera is also a great fit because it delivers depth images directly. This reduces the CPU load on the iMX8. If we require Full HD, we should use the ZED camera. Another advantage of the ZED camera is the bigger baseline of 120mm. This allows depth estimation with higher range. We decided against this camera mainly because of the high price and because of the closed source SDK. We can't benefit from Full HD sensors because embedded systems only have limited CPU resources.

Chapter 4

Camera Calibration

Camera calibration is necessary to find a model that expresses the properties of a camera. Properties of a camera are:

- Focal length of the camera lense
- Principal point on the camera sensor, where the z axis of the cameras coordinate system goes through.
- Distortion of the camera lense
- Size of a pixel

One example where we need to know the camera model is if we project virtual objects into a real world image. Figure 4.1 shows an image of a checkerboard where we project a virtual cube onto. One camera model in this image took distortion into account while the other one ignored it.

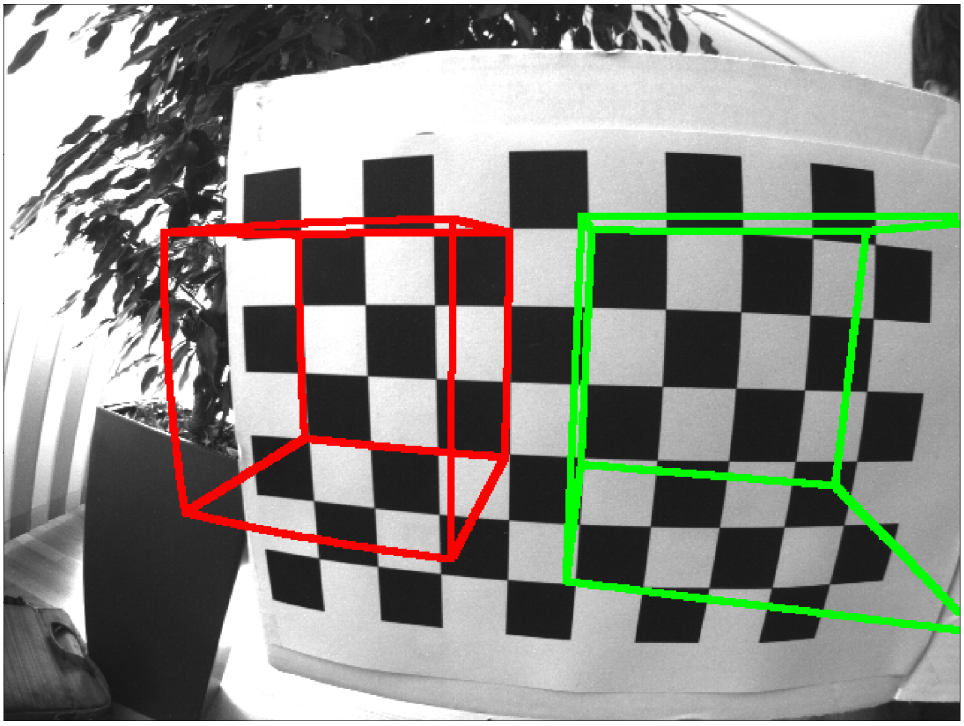


Figure 4.1: Applied camera model with (red) and without(green) distortion

We also need to know the camera model when doing SLAM. With known camera model we can rectify (see 4.4) the image so we can assume the camera as a "perfect'' pinhole camera model with known image center and known focal width. Additional to the above parameters we need to align the images of the left and right camera in stereo vision. The cameras aren't perfectly aligned in the vertical direction. However, we need to have them aligned because this makes it possible to search for matches only on the epipolar line [7]. Further the cameras are slightly rotated which means we must inverse rotate the image to have a perfect alignment. So additional to the properties of a monocular camera we need to know the following parameters:

- Y offset of the right camera to the left camera in pixels
- Rotation matrix of the right camera with left as reference
- The distance between left and right camera center (baseline)

4.1 Camera Model

The camera model expresses how any point in the three-dimensional space is projected onto a two-dimensional image. As a first approximation it assumes that all rays are going through one point. This is called the pinhole

camera model (figure 4.2a). Given this assumption we can describe the projection of a 3D point onto a 2D image as shown in equation 4.1. We calculate the pixel location x, y on the image by normalizing with s as show in equation 4.2 [18].

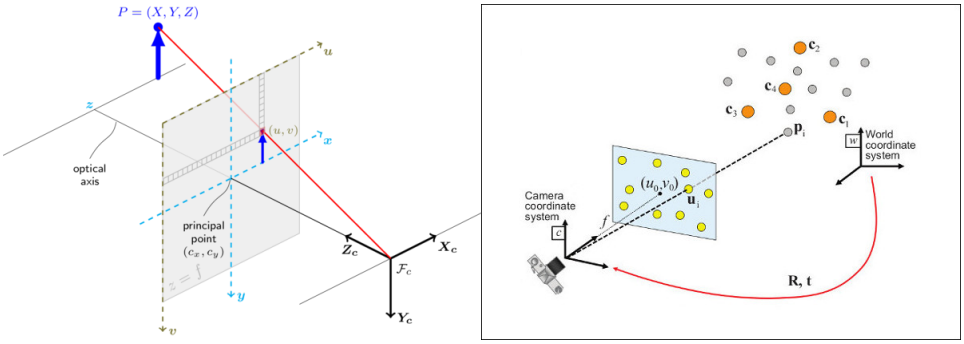
$$\begin{pmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \begin{pmatrix} u \\ v \\ s \end{pmatrix} \tag{4.1}$$

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} u/s \\ v/s \end{pmatrix} \tag{4.2}$$

Where:

$$\begin{aligned} X, Y, Z &: \text{point in the 3D world} \\ u, v, s &: \text{point in 2D image not normalize} \\ x, y &: \text{point in 2D image normalized with s} \\ f_x, f_y &: \text{focal length of the camera} \\ c_x, c_y &: \text{principal point} \\ t_x, t_y, t_z &: \text{location of the camera} \\ r_{ij} &: \text{part of the rotation matrix} \end{aligned}$$

We can describe the intuition as follows. A Point (X,Y,Z) is projected onto an image sensor (u/s,v/s) by the multiplication of the intrinsic times the extrinsic matrix. The extrinsic matrix describes where the pinhole of the camera is located in the three dimensional space. The intrinsic camera matrix describes how the camera is constructed. For example, in figure 4.2b we translate and rotate a point p_i with the extrinsic matrix so we can describe its coordinates with the pinhole as origin. Further we transform the point with the intrinsic camera matrix onto the image sensor.



(a) Pinhole model (b) 3D point to 2D point

Figure 4.2: Image projection

The goal of camera calibration is to find the intrinsic camera matrix. When doing SLAM we try to find the extrinsic matrix. When doing PnP 5.1.5 we normally get the projection matrix. With known intrinsic matrix we can calculate the position and pose (extrinsic matrix) of the camera by multiplying the projection matrix with the inverse of the intrinsic matrix as shown in equation 4.3.

$$E = I^{-1} * P \tag{4.3}$$

E : Extrinsic Matrix
I : Intrinsic Matrix
P : Projection Matrix

A good reference for camera calibration is the OpenCV documentation [12]. We won't describe this process in more detail.

4.2 Stereo Camera Model

Additional to the monocular parameters we need to find the parameter of the stereo camera. An image taken by a stereo camera normally is misaligned as shown in the top row of figure 4.3. The goal is to find the rotation matrix as well as the Y offset to get a perfectly aligned image pair as shown in the bottom row of figure 4.3.

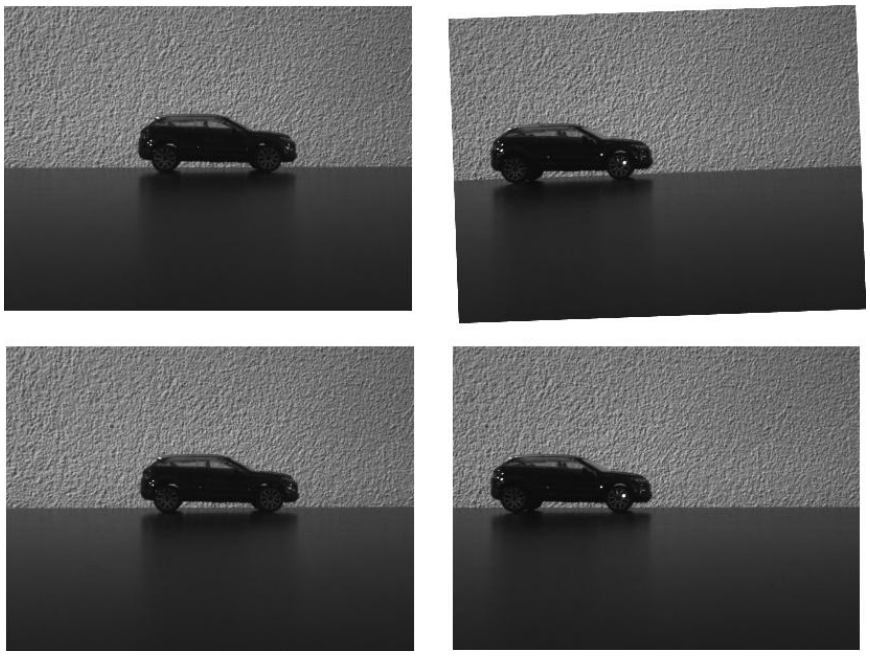
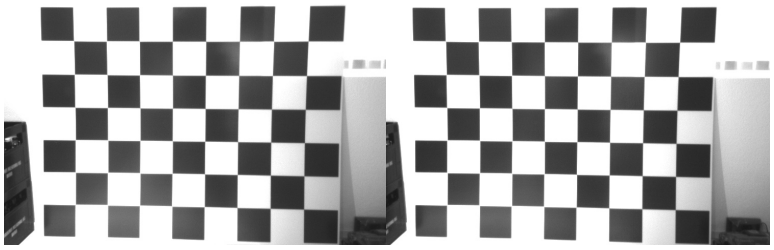


Figure 4.3: Stereo Calibration, top row misaligned image, bottom row aligned images



(a) left image used for calibration (b) right image used for calibration

(c) left image after rectification (d) right image after rectification

Figure 4.4: Top: Two images used to calibrate the camera, Bottom: New images after rectify

Figure 4.4 shows a "real" example of a calibrated stereo camera. In comparison to example 4.3 this calibration also corrects distortion and equalizes fx and fy for both cameras. Only rectified images as shown in figure 4.4 are used for SLAM in the following chapters. The images are rectified right after capturing from camera.

Another parameter stereo calibration can calculate is the baseline between both cameras. The baseline is the distance of the centers of the left and the right camera. For that we have to specify the length of one checkerboard field. With known size we can calculate the baseline. However, the calibration process will output focal length in pixels times baseline in meters as shown in equation 4.4. As we will see in equation 5.1 this is the value needed for calculating the depth from disparity.

$$b_f = \frac{b * f}{p_s} \tag{4.4}$$

b_f : Output of calibration process
b : basline in m
f : focal lenght in m
p_s : Pixel size in m

4.3 Checkerboard Size

For camera calibration it is important to use a big checkerboard. The bigger the board the better the calibration result gets. The reason is hidden in equation 4.1. When we try to find the projection matrix we will always have a small error. This error comes from restrictions regarding resolution and misplacement while detecting corners. However, if the checkerboard is far away the error will be divided through the distance because we are not interested in scale while doing camera calibration. This means the farer away the checkerboard is the smaller will be the influence of our error.
For calibrating the camera we printed a checkerboard onto an advertisement board as offered by [13].
To see how well the calibration process performs we have to analyze the reprojection error that can be received by stereoCalibExtended of OpenCV. The error should be below 0.5 pixels, the smaller the better.
For calibrating the Econ Tara, a checkerboard of size $1500x1000mm$ was used. The Checkerboard fields have a size of $166x166mm$ which leads to $8x6$ inner corners detectable by the calibration script described in the next section.

4.4 Econ Tara calibration

For calibrating the Econ Tara camera we can use the econ_calibration_images.py script. We find this script in the project repo under test [29].

```
python3 econ_calibration_images.py <calib folder> --size <field size>
```

<field size> is the height and width of one checkerboard field. The images in the folder <calib folder> are gray scale and have the form <name><counter>_left and <name><counter>_right. We can use the script extract_channels.py to create the images:

```
python3 extract_channels.py <camera> <hidraw> <calib folder>
```

This script expects the path to the Econ Tara camera, the path to the hidraw device (to set auto-exposure) and the output folder. Pressing "w" in the image preview will save a new image. Pressing "q" will quit the program.

Chapter 5

ORB SLAM

In this chapter we discuss more deeply how ORB SLAM works and how we port it to the iMX8 processor.

5.1 How it works

ORB SLAM tries to find corners with FAST corner detection [6]. This gives us points which should lay on the intersection between two edges. We call this points keypoints. At this points we calculate the ORB descriptors [1] which describe the keypoints in a unique way. Based on this keypoints ORB SLAM will estimate the pose and position of the camera in various steps shown in figure 5.1.

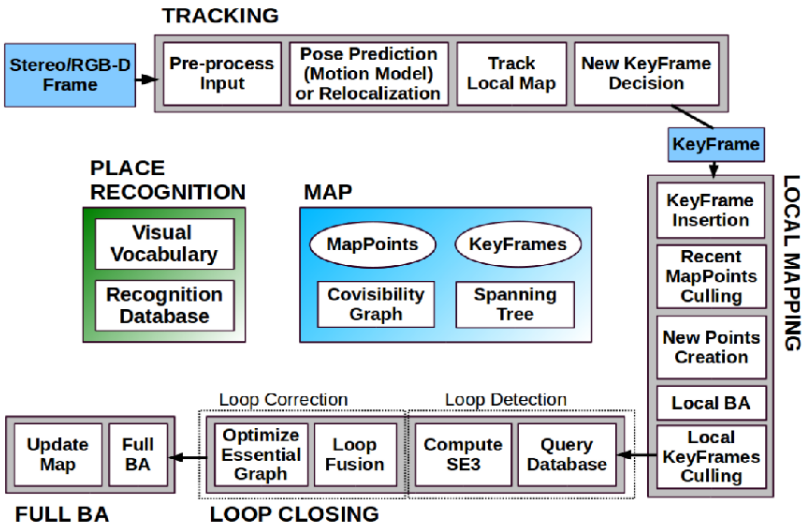


Figure 5.1: ORB SLAM 2 [3]

ORB SLAM uses multiple threads to improve the performance of the algorithm, this threads and it's tasks are described in the following sections.

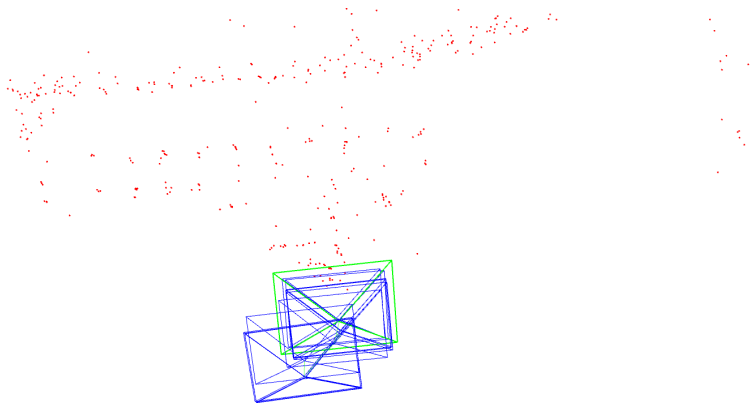


Figure 5.2: ORB SLAM 2 keyframes (blue), keypoints (red) current camera view (green)

In the following section we will use the terms keypoints, keyframes, pose and position. In figure 5.2 we see what this terms mean. The pose and position is where the image was taken and in which direction the camera was looking. Pose and position is equivalent to the extrinsic camera matrix.

5.1.1 Initialization

Before starting with tracking the system needs an initialization step. In this step it generates the first point cloud with camera at origin. For stereo SLAM this step is simple:

1. Calculate ORB keypoints on the left and right image
2. Match the ORB descriptors of both images
3. Calculate the disparity (difference) in pixel between the left and right image of each point
4. Calculate depth with equation 5.1
5. Calculate x and y position with equation 5.2

$$\begin{aligned} d &= \frac{f * b}{p_s * \Delta} \\ \Delta &= x_{left} - x_{right} \end{aligned} \tag{5.1}$$

d : depth
b : baseline in m
f : Focal length in m
p_s : Pixel size in m
Δ : Disparity
x_{left} : X Position of keypoint on left image
x_{right} : X Position of keypoint on right image

$$X = \frac{d * x}{f_x}$$
$$Y = \frac{d * y}{f_y} \tag{5.2}$$

X : X position with camera as origin
Y : Y position with camera as origin
d : Depth=Z position with camera as origin
x, y : x,y position in pixels in the image
f_x, f_y : Focal length

This gives us an initial point cloud with camera as origin. We use this point cloud to estimate further poses as described in the following section. We will also have a look at the differences between stereo and monocular initialization in section 5.2.

5.1.2 Tracking

Tracking is the most important task where the algorithm estimates the pose and position of the current camera view. It performs the following steps:

1. Find ORB keypoints on both images

With known motion model from previous tracking:

2. Search for matching ORB descriptors based on motion model
3. If the movement goes to the right we take the left image for the search if the movement goes to the left we take the right image
4. Backproject points from 3D cloud with projection matrix found with motion model
5. Optimize pose and position with Levenberg Marquardt [8] to minimize the backprojection error

6. Calculate the motion model by using the pose and position of the previous and current frame

With unknown motion model:

2. Search best matching keyframe with DBoW2 [16]
3. Match ORB descriptors with best fitting keyframe
4. Find the current pose as described in section 5.1.5
5. Calculate the motion model by using the pose and position of the previous and current frame

At the end we check if the current frame is a potential keyframe. We do this by checking how many frames were captured since the last keyframe, if a lot of outliers were detected and if more than 35% of points are new with regards to the reference keyframe. If it fulfills everything the new frame is a possible keyframe.

5.1.3 Local Mapping

Local mapping tries to map the newly found keypoints into the world map. It only does local mapping for possible keyframes found in the previous step.

1. Transform new keypoints to the global coordinate system based on the camera pose
2. Do bundle adjustment [20]. It is likely that points we already know do not 100% match the position where we see the points. Bundle adjustment fuses the position of points found more than once.
3. Check if more than 10% of the points in the current frame compared to the points in all keyframes are new. If not we drop the keyframe.

5.1.4 Loop Closing

Loop closing tries to detect loops and will then do a bundle adjustment to increase the precision of the point cloud.

1. Match latest keyframe with all keyframes stored in the database with DBoW2 [16]
2. Check if the match closes a loop
3. Optimize corresponding points by doing bundle adjustment over all keyframe poses and point positions

Loop Closing increases the precision of the point cloud because it can remove drift. However, it only takes effect if a loop is detected as shown in figure 5.3.

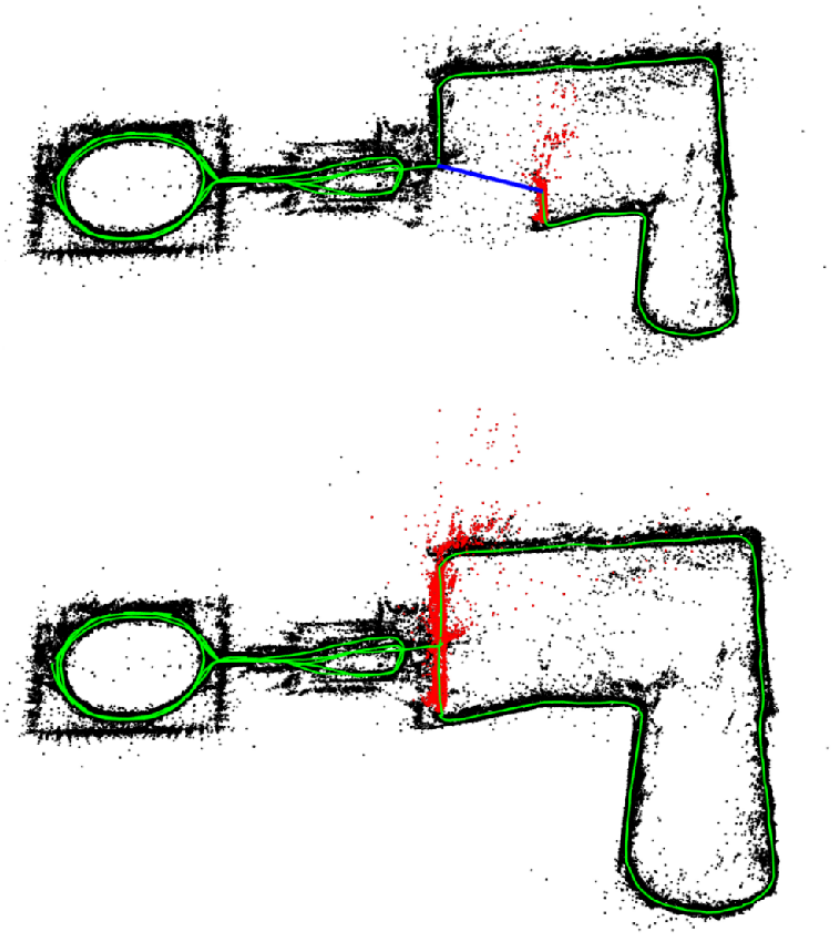


Figure 5.3: A map before (top) and after (bottom) loop closing. The blue line shows the error before bundle adjustment.

5.1.5 Linear Camera Pose Estimation

Linear Camera Pose Estimation (also Linear Perspective-n-Point) is the step where we find a projection matrix based on a 3D/2D point mapping. We have to know at least 6 points to do Perspective-n-Point (PnP) with the linear model. By doing PnP we want to find the projection matrix p_{ij} in 5.3.

$$\begin{pmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{pmatrix} * \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \begin{pmatrix} u \\ v \\ s \end{pmatrix} \tag{5.3}$$

Where:

p_{ij} : Unknown camera projection values of projection matrix P
X, Y, Z : 3D Point
u, v, s : 2D Point

We can rewrite equation 5.3 as shown in equation 5.4. We don't know u and v directly. We only have the pixel position which is $x = u/s, y = v/s, 1$. There is no way to estimate s. We know that s equals to the third row of equation 5.4. If we multiply x and y by the third row, the equation will still hold (equation 5.5). As a side effect we can remove the third row because the left and right side are equal which always holds. We finally have two equation per image point. Another trick we use is to set p_{34} to 1. This makes the Z position of the camera the norm for the other unknowns. Because of this change the trivial solution where every unknown is zero disappears. We can solve the final equation 5.6 for all unknowns p_{ij} with linear least square.

$$\begin{pmatrix} X & Y & Z & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & X & Y & Z & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & Y & Z & 1 \end{pmatrix} \begin{pmatrix} p_{11} \\ p_{12} \\ p_{13} \\ p_{14} \\ p_{21} \\ p_{22} \\ p_{22} \\ p_{23} \\ p_{31} \\ p_{32} \\ p_{33} \\ p_{34} \end{pmatrix} = \begin{pmatrix} u \\ v \\ s \end{pmatrix} \tag{5.4}$$

$$\begin{pmatrix} u \\ v \\ s \end{pmatrix} = \begin{pmatrix} x * (X * p_{11} + Y * p_{12} + Z * p_{13} + p_{14}) \\ y * (X * p_{21} + Y * p_{22} + Z * p_{23} + p_{24}) \\ 1 * (X * p_{31} + Y * p_{32} + Z * p_{33} + p_{34}) \end{pmatrix} \tag{5.5}$$

$$\begin{pmatrix} X & Y & Z & 1 & 0 & 0 & 0 & 0 & -xX & -xY & -xZ \\ 0 & 0 & 0 & 0 & X & Y & Z & 1 & -yX & -yY & -yZ \end{pmatrix} \begin{pmatrix} p'_{11} \\ p'_{12} \\ p'_{13} \\ p'_{14} \\ p'_{21} \\ p'_{22} \\ p'_{23} \\ p'_{24} \\ p'_{31} \\ p'_{32} \\ p'_{33} \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \quad (5.6)$$

Where:

$$\begin{aligned} p_{ij} &: \text{unknown camera projection values} \\ p'_{ij} &: \text{same as c but normalized with } p_{34} \\ X, Y, Z &: \text{3D Point} \\ u, v, s &: \text{2D Point with scale factor s} \\ x, y &: \text{2D Point normalize to s=1} \end{aligned}$$

We miss one last step. In equation 5.5 u,v,s and p_{ij} are unknown. Therefore, setting all unknowns to 0 would be a valid solution. If we set p_{34} to 1 we can get rid of this trivial solution where all unknowns are zero. However, we now have to find the right value for p_{34}. If we calculate the intrinsic times the extrinsic matrix as shown in equation 4.1 we end up in equation 5.7. We focus on the third row. We know that the norm of a column of a rotation matrix must be one. The reason for this is that a rotation matrix will never change the length of a vector. Therefore, $h = \sqrt{r_{31}^2 + r_{32}^2 + r_{33}^2}$ must be one [21]. Because we normalized t_z to one, this assumption does not hold for our solution. The camera distance t_z will therefore be $1 * 1/h$. To correct the projection matrix we can multiply the matrix with $1/h$ (equation 5.8).

$$P = \begin{pmatrix} f_x r_{11} + p_x r_{31} & f_x r_{12} + p_x r_{32} & f_x r_{13} + p_x r_{33} & f_x t_x + p_x t_z \\ f_y r_{21} + p_y r_{31} & f_y r_{22} + p_y r_{32} & f_y r_{23} + p_y r_{33} & f_y t_y + p_y t_z \\ r_{31} & r_{32} & r_{33} & t_z \end{pmatrix} \quad (5.7)$$

$$P = 1/h \begin{pmatrix} p'_{11} & p'_{12} & p'_{13} & p'_{14} \\ p'_{21} & p'_{22} & p'_{23} & p'_{24} \\ p'_{31} & p'_{32} & p'_{33} & 1 \end{pmatrix} \quad (5.8)$$

Where:

$$\begin{aligned} P &: \text{projection matrix} \\ p'_{ij} &: \text{projection matrix found with } t_z = 1 \\ f_x, f_y &: \text{focal length} \\ c_x, c_y &: \text{principal point} \\ r_{ij} &: \text{Parameters of rotation matrix} \\ t_{x,y,z} &: \text{location of the camera} \\ h &: \text{scaling inverse } tz = 1/\sqrt{r'^2_{31} + r'^2_{32} + r'^2_{33}} \end{aligned}$$

Now that we know all unknowns we can calculate the extrinsic matrix that describes the pose. We need to multiply the inverse intrinsic matrix times the projection matrix as shown in equation 4.3. If we prefere angles instead of the rotation matrix we can extract them as shown in equation 5.10. Equation 5.9 shows the rotation matrix derived from angles. To get back angles we use the fact that $arctan(sin(x)/cos(x)) = x$.

$$R = \begin{pmatrix} c(\theta_y)c(\theta_z) & -c(\theta_y)s(\theta_z) & s(\theta_y) \\ c(\theta_x)s(\theta_z)s(\theta_y)c(\theta_z) & c(\theta_x)c(\theta_z) - s(\theta_x)s(\theta_y)s(\theta_z) & -s(\theta_x)c(\theta_y) \\ s(\theta_x)s(\theta_z) - c(\theta_y)c(\theta_z) & c(\theta_x)s(\theta_y)s(\theta_z) + s(\theta_x)c(\theta_z) & c(\theta_x)c(\theta_y) \end{pmatrix} \tag{5.9}$$

Where:

$$\begin{aligned} \theta_n &: \text{Rotation in n direction} \\ s &: \text{sin} \\ c &: \text{cos} \end{aligned}$$

From that we get:

$$\begin{aligned} \theta_y &= arcsin(sin(\theta_y)) = arcsin(r_{13}) \\ \theta_x &= arctan(\frac{-sin(\theta_x)cos(\theta_y)}{cos(\theta_x)cos(\theta_y)}) = arctan(\frac{r_{23}}{r_{33}}) \\ \theta_z &= arctan(\frac{-cos(\theta_y)sin(\theta_z)}{cos(\theta_y)cos(\theta_z)}) = arctan(\frac{r_{12}}{r_{11}}) \end{aligned} \tag{5.10}$$

Where:

$$\begin{aligned} \theta_n &: \text{Rotation in n direction} \\ r_{nn} &: \text{Rotation matrix term see equation 4.1} \end{aligned}$$

Because we normally have more than 6 points available for PnP we use RANSAC [19] to find a good solution and to eliminate outliers.

5.2 Comparison Mono to Stereo

In this work we only talk about stereo SLAM. But what are the differences to a monocular SLAM system?
A monocular camera can't see depth. This means that from one single image we can't estimate the keypoints in the 3D world. Therefore, we need an initialization process that will try to estimate the depth over several images. There are different strategies for solving the initialization problem. ORB SLAM calculates two models in parallel one when most points are laying on a plane and one model when most points are randomly distributed. In one model we search the essential matrix (fundamental matrix with 5 DoF) in the other we search the homography matrix [2]. If the initialization succeeds one of this matrices will converge to a small reprojection error while the other wont. The camera pose and position is calculated by decomposing the converged matrix.
Other approaches use an extended Kalman filter (EKF) and try to optimize the camera pose over the first n frames [5]. The EKF starts with points at random depth values and converges after several frames to the "real" depth values. For a monocular camera it is not possible to find all 6 DoF because the fundamental and homography matrix only have 5 DoF. They therefore set the 6 parameter to 1 which means we don't get a real "scale" value. All measurements with monocular cameras will therefore have an unknown scale.

5.3 Port to iMX8

The iMX8 BSP is built with Yocto. NXP and Toradex provide a BSP layer which contains a kernel and all necessary proprietary libraries for OpenGL. How we build the BSP and the SDK is out of scope of this project. We find more details about the BSP under the following references:

- Yocto project [14]
- Toradex BSP [15]

From the Yocto build we can also get an SDK which contains an ARM64 toolchain that can be used to compile additional programs and libraries. This toolchain was used to build ORB SLAM 2 for iMX8. For the following section we assume that the toolchain is installed under /opt/fsl-imx-x11/4.9.51-mx8-beta.

5.3.1 Pangolin

Pangolin [24] is required by ORB SLAM for showing the sparse map and the camera pose. A simple git checkout of the pangolin sources is enough to

receive the sources. Before building the project with CMake the following patch should be applied:

```
diff --git a/CMakeLists.txt b/CMakeLists.txt
index 32a2d78..683ef38 100644
--- a/CMakeLists.txt
+++ b/CMakeLists.txt
@@ -15,6 +15,9 @@ SET(CPACK_DEBIAN_PACKAGE_MAINTAINER "Steven Lovegrove")
 SET(CPACK_PACKAGE_VERSION_MAJOR ${PANGOLIN_VERSION_MAJOR})
 SET(CPACK_PACKAGE_VERSION_MINOR ${PANGOLIN_VERSION_MINOR})
 SET(CPACK_PACKAGE_VERSION_PATCH "0")
+SET(CMAKE_INCLUDE_SYSTEM_FLAG_CXX "-I")
+SET(CMAKE_INCLUDE_SYSTEM_FLAG_C "-I")
+
 include(CPack)

 option( BUILD_EXAMPLES "Build Examples" ON )
```

After that the following commands will compile and install Pangolin:

```
. /opt/fsl-imx-x11/4.9.51-mx8-beta/environment-setup-aarch64-poky-linux
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=<INSTALL_DIR> ..
make -j
make install
```

This will build the Pangolin library for ARM64. INSTALL_DIR should point to a directory which is used as local "root filesystem". This root filesystem should be copied via scp to the iMX8.

5.3.2 OpenCV

We modified ORB SLAM 2 for this project by adding different ORB implementations and by adding Densification. To make this version work we need the modified OpenCV version from [27]. The branch se-3.4.3 contains some modification which allows to reuse the image pyramid generated by ORB. Also the CMake flags are changed to allow compilation on iMX8. Additionally we need to compile OpenCV Contribute [28] version tag 3.4.3. To build OpenCV plus OpenCV Contribute the following commands are required:

```
. /opt/fsl-imx-x11/4.9.51-mx8-beta/environment-setup-aarch64-poky-linux
. /opt/fsl-imx-x11/4.9.51-mx8-beta/sysroots/x86_64-pokysdk-linux/\
environment-setup.d/cmake.sh

mkdir build
cd build

export PKG_CONFIG_LIBDIR="/opt/fsl-imx-x11/4.9.51-mx8-beta/\
sysroots/aarch64-poky-linux/usr/lib/pkgconfig/"
cmake -DWITH_ITT=NO -DWITH_LIBV4L=YES -DWITH_TBB=NO\
 -DOPENCV_EXTRA_MODULES_PATH=<path to OpenCV contrib>\
 -DCMAKE_BUILD_TYPE=Release ..
```

We need to copy all libraries found in the lib directory to the iMX8.

5.3.3 ORB SLAM 2

The sources of this project are based on ORB SLAM 2 [25]. Instead of using the sources of the original implementation, the modified sources [26] under branch work_se should be used. They contain some fixes for newer compiler versions as well as additional sources for the Econ Tara camera. After fetching the sources with git the following commands will build the binary:

```
. /opt/fsl-imx-x11/4.9.51-mx8-beta/environment-setup-aarch64-poky-linux
mkdir build
cd build
cmake -DPangolin_DIR=<INSTALL_DIR/usr/lib/cmake/Pangolin>\
 -DCMAKE_INSTALL_PREFIX=<INSTALL_DIR>\
 -DOpenCV_DIR:PATH=<OPENCV_BUILD_DIR> ..
make -j
make install
```

This will build ORB SLAM for ARM64. INSTALL_DIR is the same directory as it was for Pangolin. After that we can copy the whole INSTALL_DIR to the iMX8 root directory and start econ_stereo to start the SLAM system. The OPENCV_BUILD_DIR should point to the build directory of OpenCV in section 5.3.2.

5.3.4 ORB SLAM 2 usage

For the Econ Tara a special stereo_econ application is available. This application can be called as follows:

```
./stereo_econ path_to_vocabulary path_to_settings path_to_video
```

The first argument path_to_vocabulary points to the ORB dictionary for DBoW2, path_to_settings should point to the Econ.yaml and path_to_video points to the Econ stereo camera device.
In comparison to the orignal ORB SLAM 2 implementation some additional parameters are added:

stereosgbm.cn	CN parameter fo sgbm
stereosgbm.preFilterCap	Pre filter capacity for sgbm
stereosgbm.windowSize	Window size for sgbm
stereosgbm.blockSize	Block size for sgbm
stereosgbm.minDisparity	Minimum disparity for sgbm
stereosgbm.speckleRange	Speckle Range for sgbm
stereosgbm.disp12MaxDiff	Max diff for sgbm
stereosgbm.uniquenessRatio	Uniqueness ratio for sgbm
stereosgbm.speckleWindowSize	Speckle window size for sgbm
stereosgbm.numberOfDisparities	Total number of disparities for sgbm
cpu.system	CPU assigned to system thread
cpu.stereoleft	CPU assigned to ORB thread for left image
cpu.stereoright	CPU assigned to ORB thread for right image
cpu.loopclosing	CPU assigned to loop closing thread
cpu.localmapping	CPU assigned to local mapping thread
cpu.densify	CPU assigned to densify thread
cpu.viewer	CPU assigned to viewer
Densify.enabled	0 if densify is disabled 1 if enabled
ORBextractor.orbextractor	Orb extractor: 0->Original, 1->OpenCV, 2->OpenCV OCL
Densify.topmargin	top margin for depth image
Densify.bottommargin	bottom margin for depth image
Densify.leftmargin	left margin for depth image
Densify.rightmargin	right margin for depth image

The SGBM features are hard to tune the values in Econ.yaml were taken from the stereo_match.cpp example of OpenCV [27]. The ORBextractor.orbextractor can be used to switch between different ORB implementations in section 7.1. Densify.*margin is used to add margins to depth images. This is necessary because points close to the boarder have higher uncertainty.

Chapter 6

Densification

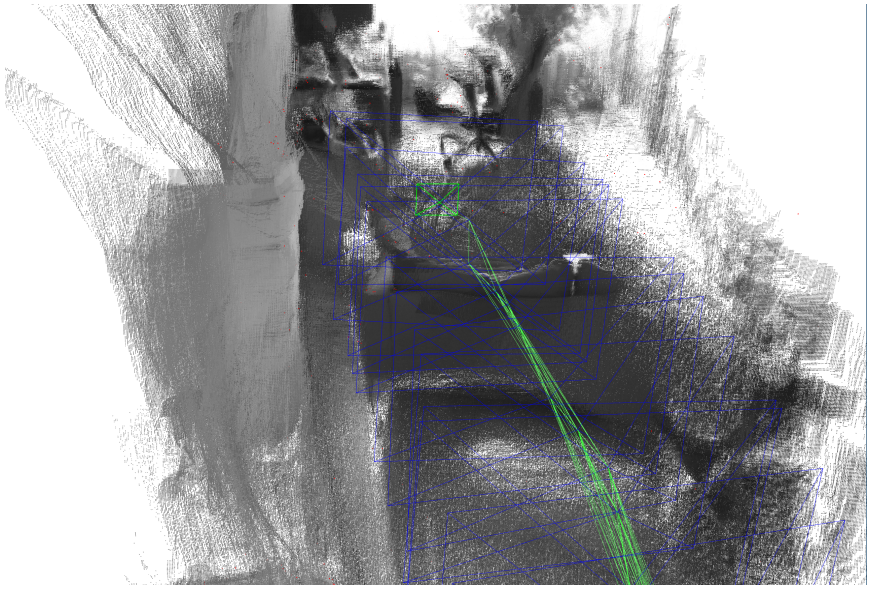


Figure 6.1: A dense cloud generated with SGBM and ORB SLAM

ORB SLAM only delivers a sparse map. This sparse map can't be used for creating maps or to reconstruct 3D objects. The focus of ORB SLAM is to find a stable pose and position of the camera. In this section we discuss how we can improve ORB SLAM by densifing the point cloud as shown in figure 6.1. We also will show a first implementation that is part of the modified ORB SLAM sources (see section 5.3.3).
In the first section we see how the stereo images can be used to create a depth map. The next section describes how to use the depth map to generate a point cloud. The two last sections discuss two possibilities on how a depth map of several poses could be fused together. However, as we will see the current approach doesn't work as expected because the point cloud size increases too fast. Therefore, we don't have CPU time to fuse the point clouds effectively.

6.1 Depth Map

A depth map is an image that contains depth values instead of color intensities. They are generated from stereo cameras or from ToF cameras. We will only talk about how depth images are made from stereo cameras. Image 6.2c shows a depth image made from two stereo images.

(a) left image (b) right image (c) depth image

Figure 6.2: Depth image from stereo

The algorithm for crating depth images matches the intensities of the left and right images. Because we vertically aligned the camera, the algorithm starts matching the right image at pixel position of the left image and searches from there to the right as shown in figure 6.3. The position with the least intensity difference between left and right is where we calculate the distance. We call the distance between the left and right image disparity. An object far away has less disparity than a near object. A point at infinity will have a disparity of zero. As we see in figure 6.2 the depth image is noisy in areas where we have low or no texture. To reduce the influence of noise, the algorithm not only matches the intensity at one pixel, it uses a block of several pixels instead (e.g. 9x9).

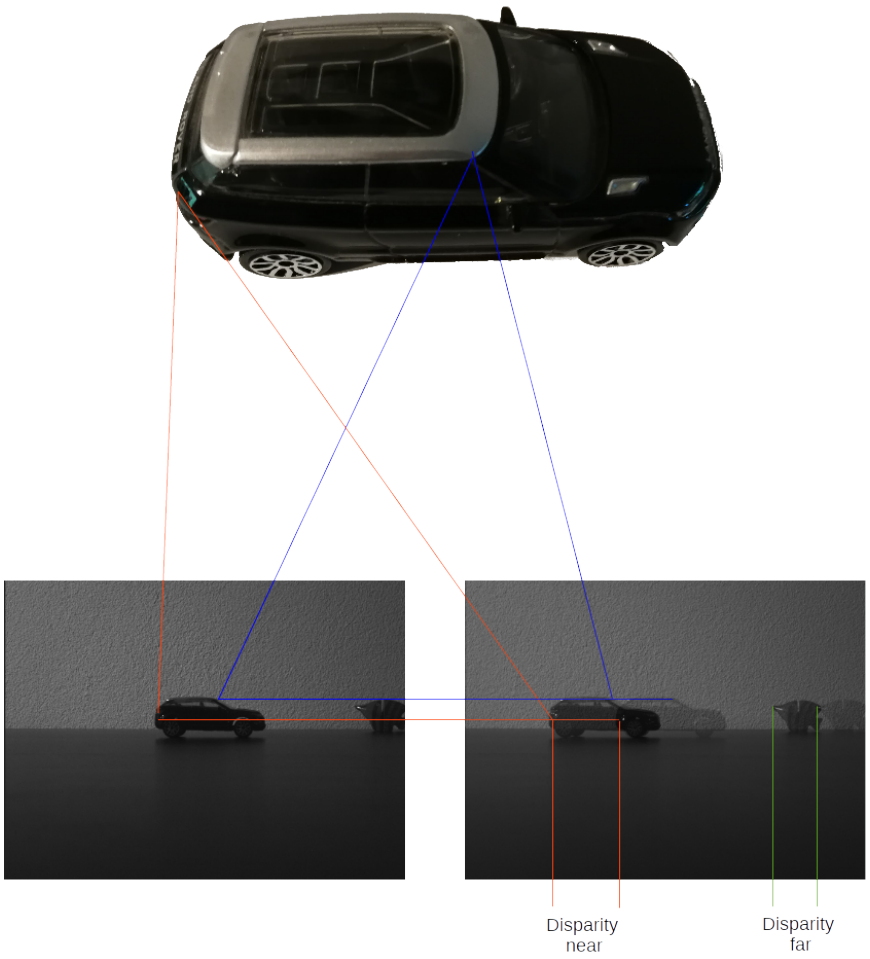


Figure 6.3: Disparity with epipolar (horizontal) line, left: left image, right: right image with transparent left image

As described we calculate the disparity by doing block matching. However, this algorithm is still prone to noise. Therefore, we use a modified version of this algorithm called Semi Global Block Matching (SGBM) [17]. Compared to normal block matching this algorithm penalties big disparity differences between neighbouring pixels. We also calculate the disparity for the left and right image to get values for pixels only visible in one image. Further we generate a confidence map if pixels have different disparities when we do left and right matching. We use this confidence map to filter vague disparities and replace them with values of their neighbourhood.

6.2 Point Cloud from Depth

From the depth map generated by the previous section we calculate a dense point cloud based on the current pose. We transform the depth information into the three dimensional space. An example of such a point cloud is shown in figure 6.4. The point cloud is generated from the depth image by multiplying each pixel with the current camera pose.

$$\begin{pmatrix} X' \\ Y' \\ Z' \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{pmatrix} * \begin{pmatrix} X \\ Y \\ d \\ 1 \end{pmatrix} \tag{6.1}$$

X', Y', Z' : World position of the pixel
r_{ij} : Rotation of extrinsic camera matrix
$t_{x,y,z}$: Translation of extrinsic camera matrix
X, Y : X,Y with current view as origin, see 5.2
d : Depth or Z value with current view as orign

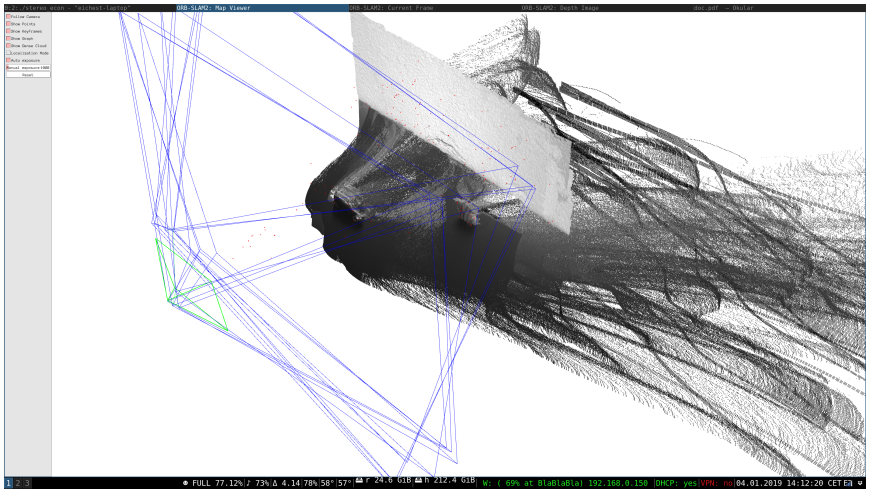


Figure 6.4: Dense point cloud generated from different keyframes

To reduce the CPU load, we add new dense points to the cloud only for keyframes. What we see in figure 6.4 is that because of the uncertainty of the depth map we have tons of outliers. Further we don't fuse points at the same location therefore we generate a lot of points that have the same position within a statistical uncertainty.

6.3 Point Cloud fusion

There are different papers that describe the problem of point cloud fusion. If we have two views that partially show the same region we have to merge the overlapping part. This section describes two methods to fuse point clouds. REMODE is a probabilistic approach while volumetric 3D mapping is a more direct approach.

6.3.1 REMODE

REMODE [22] uses a probabilistic approach to fuse data from different views. Only points with high enough probability are drawn on the 3D map. This generates a semi-dense point cloud which has a high probability of being true. However, it requires a lot of frames seeing the same scene for getting high accuracy, this is exactly what we try to avoid because we already have problems to perform well with our current approach. Therefore, this algorithm isn't suitable for us.

6.3.2 Volumetric 3D Mapping

Volumetric 3D Mapping [23] is designed to run on a CPU. It assigns a box to each 3D pixel and puts this box into an octree. If we want to add a new pixel, the algorithm checks whether it fits in an existing box. Unfortunately, a proof of concept implementation showed that even with this approach the CPU load explodes when using one point for each image pixel.

Chapter 7

Discussion

In this chapter we discuss the status of the work. We analyse the current results and see if we can improve ORB SLAM to perform at 20fps.

7.1 Results

The current implementation is heavily based on the original ORB SLAM 2 implementation as described in section 5.3. Because we used normal ORB SLAM the results from the original paper are still valid. However, what we are interested in is the realtime frame rate and therefore the time used per frame for tracking. The ORB SLAM 2 implementation for standard datasets calculates this time and prints the mean and median time after each run. For comparison we used the KITTI04 dataset with 271 images. The resolution of each image is 1226x370 where the ECON Tara has a resolution of 752x480. The KITTI dataset is a standard benchmark for SLAM therefore it was taken. In comparison to Euroc another dataset it has smaller sequences and finishes therefore faster.

We compare the ORB SLAM ORB implementation with the ORB implementation from OpenCV. To be fair we reduce the feature count when using OpenCV from 1000 to 800 because OpenCV ORB finds more features than the ORB SLAM implementation. The bigger the point cloud gets the slower the algorithm becomes, therefore tracking more features wouldn't be a fair comparison.

As we see from table 7.1 we achieve best timings on the iMX8 when using the OpenCV ORB SLAM implementation. We could extend this version that it behaves the same as the ORB SLAM implementation and would still performs better. However, we are still at a low frame rate of around 6 frames per seconds. When we compare the results of the iMX8 to a i5-7Y54 Intel processor, we see similar results with around 7fps. Unfortunately, none of these result is near to 20fps. We consider 20fps as realtime because our eye would recognize that as fluent. 6-7fps work well for slow movements but the system can't handle fast movements because of the bad performance. The biggest disappointment is that the OpenCL implementation of ORB in OpenCV is worse than the CPU implementation. We therefore can't

CPU	Impl.	feat.	feat. det.	med. tt	mean tt
iMX8	ORB	1000	548	0.18307	0.185202
iMX8	ORB, Lin. sched.	1000	548	0.390867	0.387753
iMX8	OpenCV	1000	757	0.190571	0.210072
iMX8	OpenCV	800	610	**0.155441**	**0.167176**
iMX8	OpenCV, Lin. sched.	800	610	0.358689	0.362126
iMX8	OpenCV OpenCL	800	610	0.50622	0.771859
i5-7Y54	ORB	1000	546	0.18884	0.184628
i5-7Y54	OpenCV	1000	756	0.154901	0.152786
i5-7Y54	OpenCV	800	610	0.135715	0.132304

Table 7.1: Comparison of ORB SLAM tracking time in KITTI04 using ORB or OpenCV implementation

increase the performance by using the GPU instead of the CPU.
What we also see in table 7.1 is a speciality of the iMX8. The Linux scheduler isn't aware that this processor has two different processor types. Therefore, it can't move a task which requires a fast CPU to a Cortex A72 processor. We therefore added the possibility to the ORB SLAM configuration file to bind a task to a specific processor. This allows us to use the two Cortex A72 for tracking and other high performance tasks. We see that when relying on Linux scheduler the tracking time nearly doubles. This isn't a problem on the Intel processor, because all processors are of the same type.

7.2 Optimizations

The current frame rate on the iMX8 as well as on a laptop CPU is too low. This makes it hard to get a robust tracking and mapping.

7.2.1 Different ORB Implementation

To increase the performance we can use a different ORB implementation. By using the OpenCV ORB implementation we see that higher frame rates are possible. However, this improves the performance only by around 20%. To get to 20fps we would have to improve the performance by factor 4.

7.2.2 Using OpenCL

OpenCV ORB supports finding FAST corners and calculating ORB descriptor with OpenCL. However, a first test showed that using OpenCL is even slower than using the CPU. So this approach doesn't look promising for now.

7.2.3 Reduce Resolution

Using lower resolution images would improve the performance of the algorithm. However, this is not something we want to do because it also decreases the stability. With a resolution of 752x480 we already limit the resolution to a realistic level for embedded devices.

7.2.4 Using a different SLAM algorithm

Because we don't see too many options to improve ORB SLAM a different SLAM algorithm could be used. We will discuss that in chapter 8.

7.3 Problems

In this section we try to describe some problems of ORB SLAM.

7.3.1 Complexity

ORB SLAM has grown to a complex algorithm. It's therefore hard to trace the algorithm in detail and to find bottlenecks. The use of multi threading increases the complexity even more.

7.3.2 Tracking

For our use case we don't need a full SLAM algorithm and tracking would be sufficient. Because we do densification we won't necessarily benefit of bundle adjustment, because this would invalidate the dense cloud. When doing tracking based on a motion model we don't benefit from ORB descriptors.

7.3.3 Old dependencies

ORB SLAM depends on old libraries this generates a lot of warnings while compiling. To use it in a serious project, we would have to eliminate this warnings.

7.3.4 Slow compilation

Modifying ORB SLAM is a pain. Recompiling ORB SLAM takes a lot of time because it uses libeigen which heavily uses templates. Maybe this could be improved by fixing include directives.

7.3.5 Workflow

For doing proof of concepts C++ is a heavy and unflexible language. A mix between scriptable (e.g. Python or Matlab) and compilable (e.g. C++) language could improve the workflow.

Chapter 8

Direct Approach

Because a lot of new papers describe sparse direct approaches as faster than indirect methods we analyze one of them. This could be the basis for further work specially when using it in combination with data from an IMU.

8.1 SVO

SVO [5] stands for sparse visual odometry. It is a sparse direct approach to the SLAM problem. The first monocular implementation was released as open source. The improved SVO2 source code is not publicly available. However, the paper describes the algorithm in detail so it should be possible to do a similar implementation. We can describe the main parts of the algorithm as follows.
Initialization (Stereo):

1. Search corner points with FAST
2. Split the image into areas of fixed size (e.g. 32x32)
3. Take the FAST corner with maximum response in each area if there is any
4. (optional) If there is no FAST corner take the point with highest gradient on an edge (see 8.1)
5. With a stereo camera we can do a block matching for each of this 100 points to find the depth
6. Add current frame as keyframe

Tracking:

1. For each point we try to minimize the photometric error (intensity difference)
2. We use a motion model received from previous tracking as initial pose and position to optimize it with Levenberg-Marquardt

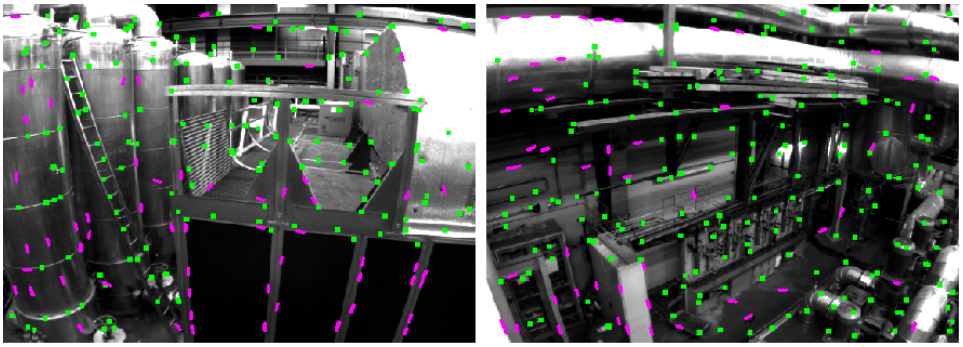
   (optional) The motion model can be improved with an IMU

3. (optional) For each point we calculate the backprojection into the image

4. (optional) For each point we search the minimal intensity difference within a small window around the backprojection

5. (optional) We calculate the difference of the position from backprojection and intensity search

6. (optional) We adjust the 3D points with bundle adjustment to minimize the mismatch between position of the backprojection and position found by intensity search.



(a) (b)

Figure 8.1: Example of sparse points used by SVO in Euroc dataset [5]. The green points are fast corners the magenta points are edgelets used in areas with no corners

This algorithm can't do loop closing as ORB SLAM does. However, it would be possible to add such improvement if necessary (e.g. by using ORB descriptors only on keyframes). Due to it's simplicity they report 13.27ms to process one 640x480 frame in EUROC Machine Hall 1 on a Nvidia Jetson TX1. The CPU of this processor is comparable to the one on the iMX8. This would end up in a frame rate up to 75fps which is above 60fps, the maximum of the Econ camera.

8.2 Densification

The SVO paper describes the possibility to do densification based on sparse points. Another problem however is that dense clouds consume a lot of memory, which is not something you have granted on embedded systems. Because SVO uses corners as key points an approach could be to generate a mesh

between all keypoints. To get a better feeling of the environment, keyframes can serve as textures to render a 3D map. To increase the resolution we need to get closer to the object, with a closer view we will detect more keypoints in the same area which makes the cloud denser. With this approach we don't need a point for each pixel. This reduces the CPU and memory consumption and gives room to do filter operations (like plane detection) on the 3D cloud.

8.3 Outlook

SVO SLAM is especially interesting because of its simplicity and because it is possible to remove a lot of optional features. By removing this optional features the tracking results become worse. If we combine tracking with data from IMU, we can hopefully compensate the loss of precision. Together with a stereo camera we also don't need the complicated initialization process described in the SVO paper because we can directly calculate the depth of each point with the first frame.

Chapter 9

Conclusion

The current results for ORB SLAM show it is not convenient for embedded devices. It is hard to optimize the algorithm because of its complexity. Even with multi-threading and multiprocessing in place we can't speed up the algorithm to over 10 frames per second. Therefore, we propose to analyze a direct sparse approach in further work for doing SLAM instead of using an indirect sparse approach. The disadvantage of being computationally more expensive in point matching is compensated by the advantage of not having the need to extract expensive features. We showed that we can reduce SVO to a simple implementation when not using all optional features. We therefore propose to use SVO in further work for doing SLAM and to generate a mesh from keypoints found by SVO. This mesh can be used as map for navigating a robot around obstacles.

List of Figures

Bibliography

[1] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski
ORB: an efficient alternative to SIFT or SURF
IEEE International Conference on Computer Vision (ICCV), Barcelona, Spain, November 2011, pp. 2564–2571.

[2] Raul Mur-Artal, J. M. M. Montiel, Juan D. Tardos
ORB-SLAM: a Versatile and Accurate Monocular SLAM System
arXiv:1502.00956v2

[3] Raul Mur-Artal, J. M. M. Montiel, Juan D. Tardos
ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras
arXiv:1610.06475v2

[4] Richard A. Newcombe, Steven J. Lovegrove and Andrew J. Davis
DTAM: Dense Tracking and Mapping in Real-Time
doi:10.1109/ICCV.2011.6126513

[5] Christian Forster et al
SVO: Semi-Direct Visual Odometry for Monocular and Multi-Camera Systems
doi:10.1109/TRO.2016.2623335

[6] E. Rosten, R. Porter, and T. Drummond
Faster and better: A machine learning approach to corner detection
doi:10.1109/TPAMI.2008.275

[7] Richard Hartley
Multiple View Geometry in Computer Vision
ISBN-10: 0-511-18618-5, p240-249

[8] Wikipedia
Levenberger-Marquardt algorithm
https://en.wikipedia.org/wiki/Levenberg%E2%80%93Marquardt_algorithm

[9] Stereo Labs
ZED Stereo Camera
https://www.stereolabs.com/zed/

[10] Econ
Tara Stereo Camera
https://www.e-consystems.com/3D-USB-Stereokamera-de.asp

[11] Intel
Intel RealSense
https://realsense.intel.com/depth-camera/#D415_D435

[12] OpenCV
Camera Calibration
https://docs.opencv.org/3.4.3/dc/dbb/tutorial_py_calibration.html

[13] mydisplay.ch
Advertisement board manufacturer
https://www.mydisplays.ch/

[14] Yocto Project
Yocto Project Mega Manual
https://www.yoctoproject.org/docs/latest/mega-manual/mega-manual.html

[15] Toradex AG
Build Apalis iMX8 OpenEmbedded Project Bring-up Image
https://developer.toradex.com/software/linux/linux-software/build-apalis-imx8-yoctoopenembedded-bring-up-image

[16] D. Gálvez-López and J. D. Tardós
Bags of binary words for fast place recognition in image sequences
IEEE Trans. Robot., vol. 28, no. 5, pp. 1188–1197, 2012.

[17] Heiko Hirschmüller
Stereo Processing by Semi-Global Matching and Mutual Information
IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE

[18] Peter Corke
Robotics, Vision and Control
Springer, ISBN 978-3-319-54413-7, chapter 11+12, page 319+

[19] wikipedia
RANSAC
https://de.wikipedia.org/wiki/RANSAC-Algorithmus

[20] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon
Bundle adjustment a modern synthesis
in Vision algorithms: theory and practice, 2000, pp. 298–372.

[21] Ying Wu,
Image Formation and Camera Calibration
http://users.eecs.northwestern.edu/ yingwu/teaching/EECS432/Notes/camera.pdf

[22] Pizzoli, Matia and Forster, Christian and Scaramuzza, Davide
REMODE: Probabilistic, Monocular Dense Reconstruction in Real Time
IEEE International Conference on Robotics and Automation (ICRA), 2014

[23] Frank Steinbrücker, Jürgen Sturm, Daniel Cremers
Volumetric 3D mapping in real-time on a CPU
doi:10.1109/ICRA.2014.6907127

[24] Steven Love Grove
Pangolin Project
https://github.com/stevenlovegrove/Pangolin.git

[25] Raul Mur-Artal, J. M. M. Montiel, Juan D. Tardos
ORB-SLAM2 Implemenation
https://github.com/raulmur/ORB_SLAM2

[26] Raul Mur-Artal, J. M. M. Montiel, Juan D. Tardos
ORB-SLAM2 Implemenation modified version
https://github.com/eichenberger/ORB_SLAM2.git

[27] OpenCV
OpenCV computer vision framework (modified version)
https://github.com/eichenberger/opencv

[28] OpenCV
OpenCV computer vision framework extensions
https://github.com/opencv/opencv_contrib

[29] Stefan Eichenberger
Project 2: ORB Slam Point Cloud generation on Apalis iMX8
https://github.com/eichenberger/pa2