Hamburg University of Technology
Institute of Mechanics and Ocean Engineering

Prof. Dr.-Ing. Seifried

and

University of Western Australia
Robotics and Automation Lab

Prof. Dr. Bräunl

Implementation and Evaluation of SLAM on the BlueROV2 Underwater Robot

Master thesis

Chris Kahlefendt

October 31, 2017

TUHH
Technische Universität Hamburg-Harburg

Author's Declaration

I solemnly declare that I have written this thesis independently, and that i have not made use of any aid other than those acknowledged in this thesis. Neither this research paper, nor any other similar work, has been previously submitted to any examination board.

Hamburg, October 31, 2017 Chris Kahlefendt

Contents

List of Figures

List of Tables

List of Acronyms

ARM Articulated Robot Motion

AUV Autonomous Underwater Vehicle

BA Bundle Adjustment

BRIEF Binary Robust Independent Elementary Features

C-KLAM Constrained Keyframe-based Localization and Mapping

CNN Convolutional Neural Network

COP Closed-form Online Pose-chain

CPU Central Processing Unit

DLT Direct Linear Transformation

DT Deferred Triangulation

DTAM Dense Tracking and Mapping

DoF Degrees of Freedom

DP Distributed Particle

DPPTAM Dense Piecewise Planar Tracking and Mapping

DSO Dense Sparse Odometry

DVL Doppler Velocity Log

DVO Dense Visual Odometry

EIF Extended Information Filter

EKF Extended Kalman Filter

FAB-MAP Fast Appearance-based Mapping

FAST Features from Accelerated Segment Test

FPS Frames Per Second

GPS Global Positioning System

GPU Graphics Processing Unit

IMU Inertial Measurement Unit

JSON JavaScript Object Notation

LAN Local Area Network

LIDAR Light Detection and Ranging

LSD Large Scale Direct

MR Multi Robot

NID Normalized Information Distance

OKVIS Open Keyframe-based Visual Inertial SLAM

ORB Oriented Fast and Rotated Brief

PEM Photometric Error Minimization

PTAM Parallel Tracking and Mapping

RANSAC Random Sample Consensus

RD Robust Dynamic

REBVO Realtime Edge-based Visual Odometry

REMODE Regularized Monocular Depth Estimation

RFM Relative Feature Measurements

RGB-D Red, Green, Blue and Depth

RK Robust Keyframe-based

ROV Remotely Operated Vehicle

ROS Robot Operating System

ROVIO Robust Visual Inertial Odometry

SCP Secure Copy Protocol

SEIF Sparse Extended Information Filter

SIFT Scale Invariant Feature Transform

SLAM Simultaneous Localization and Mapping

SPLAM Simultaneous Planning, Locating and Mapping

SURF Speeded-Up Robust Features

SVD Singular Value Decomposition

SVO Semidirect Visual Odometry

TCP Transmission Control Protocol

UAV Unmanned Aerial Vehicle

UDP User Datagram Protocol

UI User Interface

UKF Unscented Kalman Filter

UML Unified Modeling Language

UWA University of Western Australia

VIN Visual Inertial Navigation

VO Visual Odometry

WLAN Wireless Local Area Network

YAML YAML Ain't Markup Language

1 Introduction

This chapter explains the motivational background for this thesis and gives an overview over how it is structured.

1.1 Motivation

The underwater environment offers a multitude of tasks that could be performed by autonomous robots. They could for example monitor reefs, examine offshore oil rigs, explore sub sea caves or collect data for environmental studies. The appeal for using autonomous robots for these tasks is that they can generally reduce the costs and the risks involved when performing them manually. For a robot to be able to work autonomously it needs be able to localize itself, sometimes in surroundings which have never been mapped before. Since Global Positioning System (GPS) is not available underwater, localization needs to be performed in a different way. This could for example be solved by so-called underwater GPS based on acoustic localization. The problem with these techniques is that they are expensive and only work in the local area where they have been installed. A location-independent and low-cost solution to this problem could be the usage of SLAM.

SLAM has been one of most researched topics in robotics over the last 10 to 20 years. Any robot that is going to be used in different, non-preprogrammed environments needs the capability to move around and localize in this environment. Preferably without causing any collisions. Even though this is considered to be a solved problem for many scenarios it is still considered an unsolved problem underwater.

While on land most SLAM algorithms are based on cameras or Light Detection and Ranging (LIDAR) as their main perceptional sensor, most underwater SLAM algorithms are build around sonar. Sonar is a sensor that can work in the absence of light which is definitely a plus underwater but it generally is also a lot more expensive than a camera. An interesting question in light of this is: how well would a camera-based SLAM approach work underwater in situations where there is no real need for sonar?

For answering questions like this the Robotics and Automation Lab at the University of Western Australia (UWA) owns a so-called BlueROV2 underwater robot. This robot is a low-cost Remotely Operated Vehicle (ROV) equipped with a single camera and an Inertial Measurement Unit (IMU). Part of the research of making this robot work autonomously underwater is to implement a SLAM solution. That is where this thesis puts its focus.

The goal of this thesis is to find, implement and evaluate a SLAM algorithm that works in the underwater environment with the limited perceptional possibilities the BlueROV2's sensors provide.

1.2 Thesis Structure

The thesis is structured into seven chapters. The first one being this chapter which serves as an introduction.

Chapter 2 covers the basics of SLAM. It explains it both in a general form and with the help of example implementations. Furthermore it tries to give an overview over SLAM in the underwater scenario and explains which algorithm was chosen for this thesis.

In chapter 3 the chosen algorithm ORB SLAM is explained. By separating the algorithm into its modules and giving an in-depth explanation of each one this chapter serves as a complement to the explanations given by the ORB SLAM authors ([1], [2], [3], [4]). In addition to written explanation a diagram visualizing the information flow in ORB SLAM's code is supplied.

Chapter 4 summarises the information regarding the used robot: Blue Robotic's BlueROV2. It presents the robot's specifications and all modifications made.

ORB SLAM's evaluation is given in chapter 5. This chapter describes how the performed experiments were executed and how the necessary data was collected. It then proceeds to describe the most important experiments and gives an evaluation for each one. This is followed by a summary of the results with regard to ORB SLAM in the underwater scenario and also ORB SLAM in general.

Using the results described in chapter 5, chapter 6 gives suggestions on how to improve on the discovered problems. It also explains all enhancements that were made on ORB SLAM during this thesis.

The last chapter, chapter 7, concludes the thesis by summarizing the results and presenting what future works should focus on.

2 Simultaneous Localization and Mapping (SLAM)

This chapter focuses on trying to give an overview over what SLAM is and how it works. To achieve this SLAM is described both in a general form and with the help of example implementations. In addition it also outlines the challenges faced in the underwater environment, how other works have tried to cope with them and the decision process of picking an algorithm for this thesis.

2.1 Basics

One of the major problems in mobile robotics is localization. Localization describes the estimation of the changes in a robot's pose while this robot is moving. The pose includes the position and the orientation of the robot. This means that localization is really a problem of estimating coordinate transformations between a world frame and the robot's local coordinate frame [5]. The difficulty of solving this problem lies in the fact that poses cannot be measured directly. Instead the pose has to be derived from sensor data, which commonly includes noise. In order to track the pose of a robot the sensor data has to be accumulated which in turn means that every error in the sensor data will be accumulated as well. In practice this can quickly lead to a large deviation of the estimate from the real pose. So when trying to maximise the precision of localization what really has to be done is to minimize the uncertainty in the estimation.

Over the years several different methods have been developed that try to solve the problem of localization and solving this problem is considered one of the major achievements in the field of robotics over the recent years [6]. One of the main steps in finding a solution for this problem was the invention of SLAM.

SLAM provides a robot with the possibility to be placed in an unknown environment where it will incrementally build a map of its surroundings and localize itself in it. As the name implies it not only solves the problem of localization but also provides a map while doing so which is another major use-case for mobile robots. A minimum requirement for using SLAM is that the robot has to have at least one sensor which is able to collect data about its environment. This can be a camera, a laser scanner, sonar or any other sensor capable of sensing the robot's surroundings. Internal sensors which measure the robot's movement (accelerometers, gyroscopes, magnetometers, wheel encoders, etc.) can be used as an additional source of information. In general SLAM consists of several different steps which are repeated within every operational cycle [7].

1. **Robot Motion**

   By moving, the uncertainty in the robot's pose increases depending on the noise and errors in the data used for estimating the robot's movement.

2. **Odometry Reading and Robot Location Prediction**

   By collecting the data from the robot's internal sensors and/or external control

inputs the robot's new position and orientation are estimated by the so called *motion model*.

3. **Landmark Observation and Data Association**

   Using its visual sensors the robot detects features in its environment. These features are commonly called *landmarks*. In which way these landmarks are extracted depends on the implementation and the used sensors. The extracted landmarks are then matched to landmarks which were detected in earlier iterations in a process known as data association. The part which is then used to estimate the position of the observed landmarks in the context of the overall map and the robot's pose estimate is what is called the *observation model*.

4. **Location Correction**

   Based on the motion and the observation model the estimations are fused and the overall estimates for robot pose and landmark positions are updated.

5. **Adding New Landmarks**

   Landmarks which had not been detected before and are not part of the map yet will be added to the map. These can then be used for data association in the next iteration.

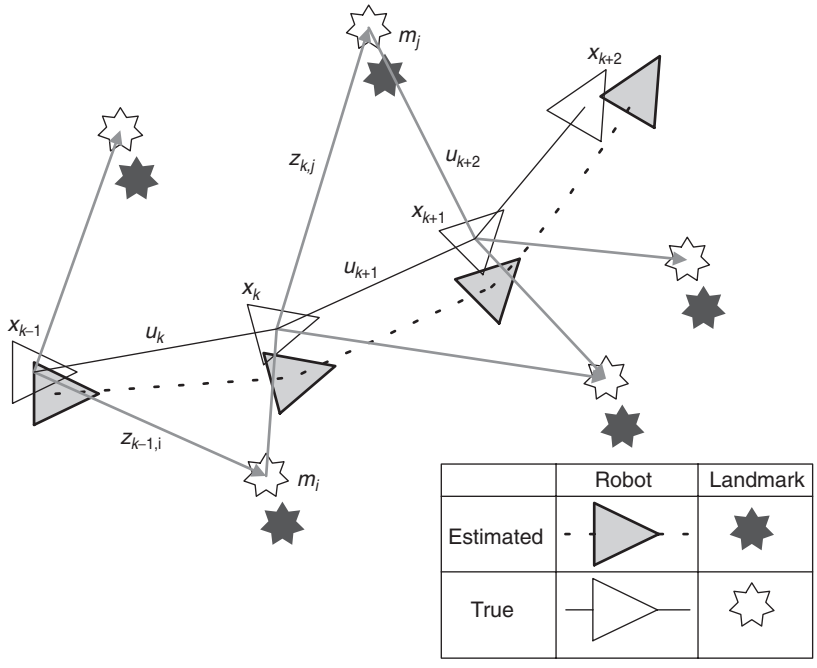


Figure 2.1: The essential SLAM problem. The robot (white triangle) can only estimate its pose (grey triangles) and its surrounding by observing landmarks (stars) and how the observations change with each iteration. Figure source: [6]

SLAM algorithms are mostly separated into a front-end and a back-end [8]. Using this separation the front-end resembles the part of the program which interprets the sensor data by e.g. extracting landmarks and executing data association. The back-end on the other hand describes the part of the algorithm which is responsible for estimating the robot's pose and the map (i.e. the landmark locations). Figure 2.2 sketches this relationship for a typical SLAM system.

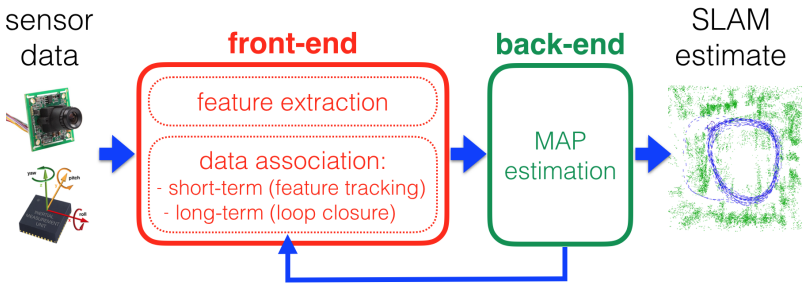


Figure 2.2: Front-end and back-end in a typical SLAM system. The back-end can provide feedback to the front-end for loop closure (see 2.1.2) detection. Figure source: [8]

2.1.1 Mathematical Formulation

As mentioned before the solution of the localization problem very much lies in the minimization of the uncertainty which is introduced when the robot moves. The probabilistic formulation of the SLAM problem is described by equation 2.1 as presented in [6].

$$P(x_k, m | Z_{0:k}, U_{0:k}, x_0) \tag{2.1}$$

P	Probability distribution
x_k	Pose of the robot
$m = \{m_1, m_2, ..., m_n\}$	Set of all landmark positions (assumed to be time invariant)
$Z_{0:k} = \{z_1, z_2, ..., z_k\}$	Set of all landmark observations
$U_{0:k} = \{u_1, u_2, ..., u_k\}$	History of control inputs (odometry readings from sensors)
x_0	Starting pose

P describes the probability distribution of the joint posterior density of landmark position and vehicle pose at time k, given all past landmark observations, control inputs and the initial pose of the robot. So the current robot pose x_k and the locations of all landmarks $\{m_1, m_2, ..., m_n\}$ can not be directly computed. Instead SLAM computes where the robot and the landmarks most likely currently are.

Looking at 2.1 it is obvious that the pose and map estimation depends mostly on two things. The first being the odometry readings collected via the robot's internal sensors which provide the control inputs $U_{0:k}$. Using the motion model in 2.1 the readings are used to get a first estimate of the robot's position.

$$P(x_k|x_{k-1}, u_k) \tag{2.2}$$

x_{k-1} : Last robot pose
u_k : Current control input (odometry reading)

The second influence on the estimations are the observations, in other words the data provided by the visual sensor (camera, laser or equivalent). These observations are dependant on the current robot's pose and the overall map, as can be seen in the observation model in 2.3.

$$P(z_k|x_k, m) \tag{2.3}$$

Based on the two models 2.2 and 2.3 the SLAM problem 2.1 is then solved in two separate steps [6]. First the *Time-update* is computed. Based on the motion model and the estimation of the previous iteration the probability is updated as shown in 2.4.

$$P(x_k, m|Z_{0:k-1}, U_{0:k}, x_0) = \int P(x_k|x_{k-1}, u_k) \cdot P(x_{k-1}, m|Z_{0:k-1}, U_{0:k-1}, x_0)dx \tag{2.4}$$

As a second step the new landmark observation z_k is also taken into account by using the observation model and calculating the *Measurement-update* as in 2.5.

$$P(x_k, m|Z_{0:k}, U_{0:k}, x_0) = \frac{P(z_k|x_k, m)P(x_k, m|Z_{0:k-1}, U_{0:k}, x_0)}{P(z_k|Z_{0:k-1}, U_{0:k})} \tag{2.5}$$

2.1.2 Loop Closure

What sets SLAM apart from other localization approaches like Visual Odometry (VO) is the creation of a map. This map is not only created for the purpose of mapping an environment. It also provides a possibility for refining the robot localization in a way dead reckoning[1] approaches cannot. By revisiting an already mapped location and recognizing that the robot has been there already the algorithm can correct the drift that accumulates over time. This process is generally described as *loop closure*. A graphical representation of this can be viewed in figure 2.3.

[1] Describes the process of localization by accumulation of sensor data as in common odometry.

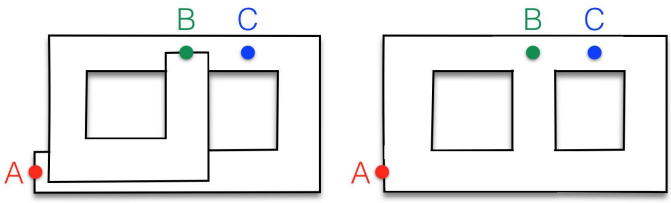


Figure 2.3: **Left:** Shows a map created by odometry. The map does not incorporate crossings or T-sections as revisited locations are not recognized. In this case B and C are far apart points in a long corridor even though they are close in reality. **Right:** SLAM creates a map using *loop closures* which introduces "shortcuts" in the map and resembles the actual environment. Figure source: [8]

2.2 Example Implementations

Since SLAM is a very actively researched field a lot of different implementations have been developed over the recent years. The following section will try to give an idea of how actual SLAM implementations work and what differences exist. It will do this by first giving a deeper explanation of the three implementations which are considered to be the "main SLAM formulations of the classical age" [8]:

- Extended Kalman Filter (EKF)-SLAM,
- FastSLAM and
- Graph SLAM

In order to show how SLAM has developed since, the more recent

- Dense Tracking and Mapping (DTAM),
- RatSLAM,
- Large Scale Direct (LSD) SLAM and
- Oriented Fast and Rotated Brief (ORB) SLAM

algorithms are explained as well. Additionally an overview over all algorithms which were found during the investigation is given in section 2.3.

2.2.1 EKF-SLAM

EKF-SLAM is one of the first SLAM approaches created. The EKF in its name is what characterizes its main feature: the usage of an EKF as a back-end. Since this also means that EKF-SLAM does not impose a front-end it can be seen as a family of different SLAM implementations with varying front-ends rather than a fixed implementation.

In EKF-SLAM the motion model is modelled as:

$$P(x_k|x_{k-1}, u_k) \iff x_k = f(x_{k-1}, u_k) + w_k \tag{2.6}$$

and the observation model as:

$$P(z_k|x_k, m) \iff z_k = h(x_k, m) + v_k \tag{2.7}$$

where w_k and v_k resemble zero-mean, white Gaussian noise, f models the robot's kinematics and h describes the geometry of the observation [6]. The added noise values are used to deal with the uncertainty in the measurements.

Implementing an EKF back-end comes with a couple of problems as pointed out by [5] and [6]. First, the computational costs rise quadratically with the amount of landmarks kept in the map [9]. This leads to problems when wanting to create feature rich or large maps. This can however be reduced to a linear relationship as explained in [10]. Second, due to the assumption that w_k and v_k are zero-mean, white and Gaussian the linearization used in the EKF can cause intolerable errors in the estimation. Third, the EKF cannot incorporate negative observations[2] in its estimations due to its Gaussian nature. Consequently it does not process all available information.

Even though EKF-SLAM has these limitations EKFs are a widely used back-end and can be regarded as a well researched topic that can achieve very good results when used under the right conditions.

2.2.2 FastSLAM

The idea of FastSLAM was first introduced by Montemerlo et al. [11] in 2002. This section will explain what is known as FastSLAM2.0, since this implementation has been proven to be superior [12]. What makes FastSLAM possible is a structural property of the SLAM problem: "correlations in the uncertainty among different map features [landmark position estimates] arise only through robot pose uncertainty" [13]. Thanks to this, the problem can be split into two subproblems. First: the robot localization which is done with the help of particle filters[3] in FastSLAM. Second: estimating the landmark locations. In order to be able to split the SLAM problem into these two parts the assumption has to be made that the exact robot pose history is known.

Doing this the SLAM problem 2.1 can be rewritten as:

$$P(X_{0:k}, m|Z_{0:k}, U_{0:k}, x_0) = P(m|X_{0:k}, Z_{0:k})P(X_{0:k}|Z_{0:k}, U_{0:k}, x_0) \tag{2.8}$$

$X_{0:k} = \{x_0, x_1, ..., x_k\}$: Robot pose history

[2] The absence of a landmark that is expected to be there.

[3] A probabilistic approach to solving the localization problem. Each *particle* represents a hypothesis for the current robot location. Over time, given enough data, particles in wrong locations are removed while those that are most likely to be correct remain. An explanation is given in [14].

Note that this is now the probability distribution on the pose history $X_{0:k}$ rather than the single pose x_k. In FastSLAM each particle computes an estimate of the robot's pose based on its individual pose and measurement history. These estimates are then given weights which decay over time. In a resampling step old erroneous particles are replaced with new ones when necessary. At which time to resample best is an open problem [6].

As can be seen in equation 2.8 the landmark estimation $P(m|X_{0:k}, Z_{0:k})$ depends on the pose history $X_{0:k}$ which is bound to a single particle. Therefore each particle requires its own landmark location estimation. Another feature of FastSLAM is that estimating the landmark's positions can be done for each landmark individually. This is generally done using one EKF per landmark. While this sounds like it would increase the computational costs as compared to EKF-SLAM, it does in fact potentially decrease them. Since the matrices used by the EKFs stay much smaller the computation cost can be reduced to $O(Nlog(K))$ with K being the number of landmarks and N the number of particles used. A detailed overview over the differences between FastSLAM and EKF-SLAM can be found in the work of Calonder [15].

Just like EKF-SLAM, FastSLAM only proposes a back-end and can be used with different front-ends. An interesting factor about FastSLAM is that, due to the nature of the particle filter, each particle can implement its own front-end.

2.2.3 Graph SLAM

Graph SLAM is a variant of SLAM which does not run and update while the robot is moving and collecting data. It is what is considered to be an *offline* SLAM approach which solves the so-called *full* SLAM problem [16]. For the *full* SLAM problem the full pose history $X_{0:k}$ and the map m are calculated from the full set of measurements $Z_{0:k}$ and control inputs $U_{0:k}$. The advantage of this approach is that instead of processing and then discarding the measurement data like in filtering approaches (e.g. EKF-SLAM and FastSLAM), the data is recorded and then used at the time of map building. This generally achieves higher map accuracy than the filtering techniques [17].

In Graph SLAM the SLAM problem is considered in its graph-based formulation where each node represents a robot pose or a landmark observation and the edges symbolize sensor measurements that constrain the measured poses and landmark observations. An example for a resulting graph is presented in figure 2.4.

The constraints represented by the edges of the graph are generally of nonlinear nature. Graph SLAM resolves these by linearizing them and applying standard optimization techniques. This way it is possible to find the configuration of robot poses that best fits the underlying constraints. A mathematical derivation and detailed explanations about the used algorithms can be found in [17].

Summarizing Graph SLAM can be regarded as split into two parts: constructing a graph from data readings and optimizing this graph to find the most likely configuration of robot poses given the introduced constraints. Generally the first part is handled by the front-end which needs to be defined in such a way that it forms a graph from its observations. The second part is handled by the back-end which performs the graph optimization. The actual

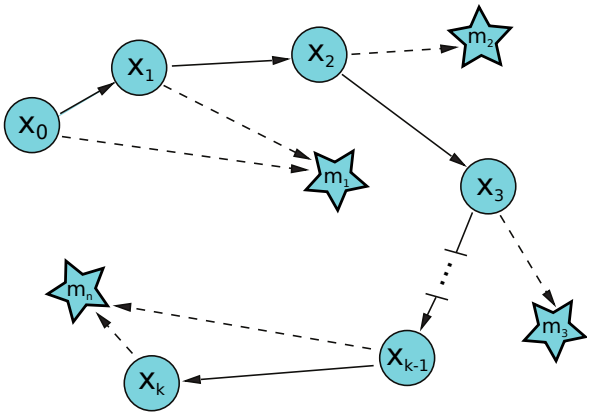
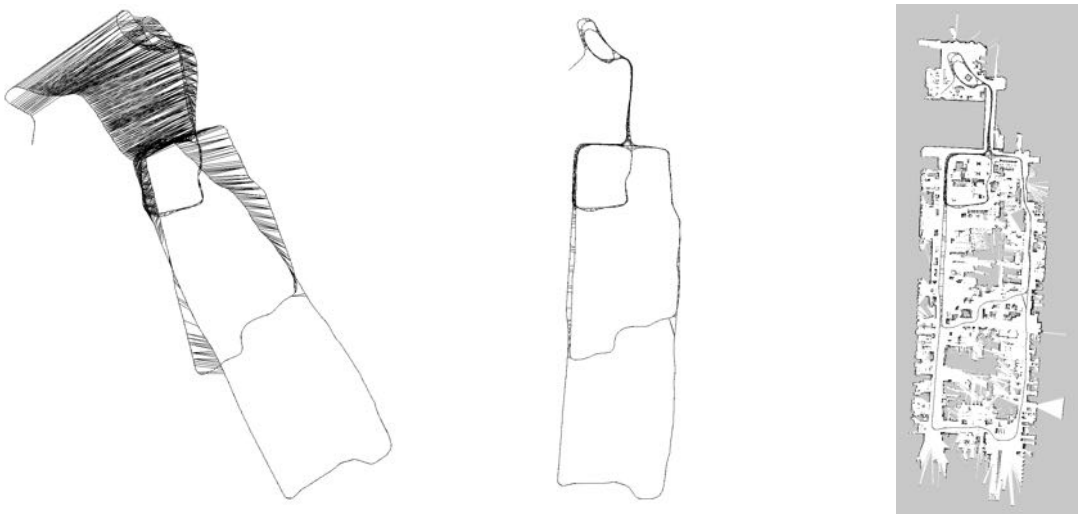


Figure 2.4: Graph based representation of the SLAM problem with circles symbolizing robot poses and stars symbolizing landmarks. Both solid and dashed edges represent constraints. Solid edges only connect consecutive robot poses, whereas dashed edges connect robot poses with landmarks that were measured from the corresponding pose.

map is then rendered based on the obtained robot poses. An example of the process can be seen in figure 2.5



(a) (b) (c)

Figure 2.5: An example of the Graph SLAM process. (a) shows the raw graph. (b) represents the graph after optimization. In (c) the map is rendered from the calculated robot poses. Figure source: [18]

2.2.4 DTAM

DTAM is a monocular SLAM approach which is capable of creating dense 3D models from a single moving camera [19]. It does this by using a depth map which is created from multiple frames with different viewpoints of the same scenery. One such created model is shown in figure 2.6.

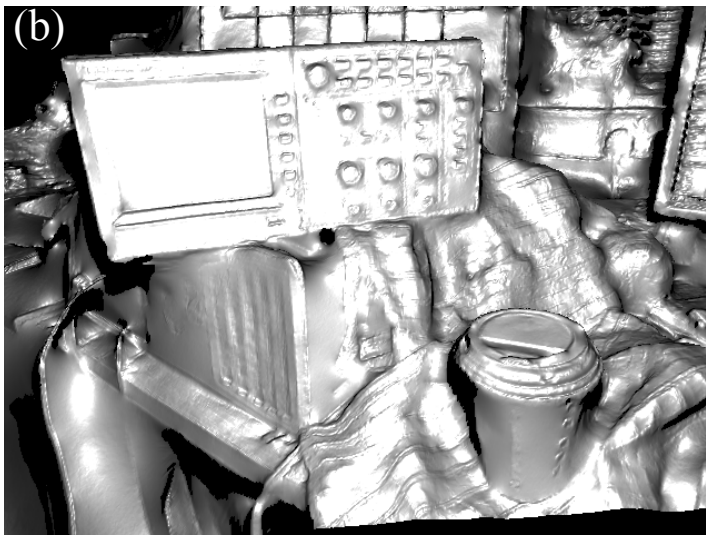


Figure 2.6: An example of an environment modelled with DTAM. Figure source: [19]

The camera movement is then tracked by aligning the whole image with the textured 3D model. According to Newcombe et al.[19] tracking works in real time and supersedes other approaches like Parallel Tracking and Mapping (PTAM) [20], especially when the camera is moved quickly and looses focus.

As this approach works with large images on a per pixel basis it is very computationally expensive and heavily relies on Graphics Processing Unit (GPU)-based parallel computation. Newcombe et al. also mention that the algorithm has problems with tracking in dynamic lighting conditions.

2.2.5 RatSLAM

RatSLAM is a biologically inspired approach to solving the SLAM problem. Taking inspiration of computational models of a rodents hippocampus[4] Milford et al. [21] created the possibility to track a robot's movement using a continuous attractor network structure explained in [22]. The system consists of three main parts:

1. Pose cells
2. Local view cells
3. Experience map

How these three modules interconnect is visualized in figure 2.7.

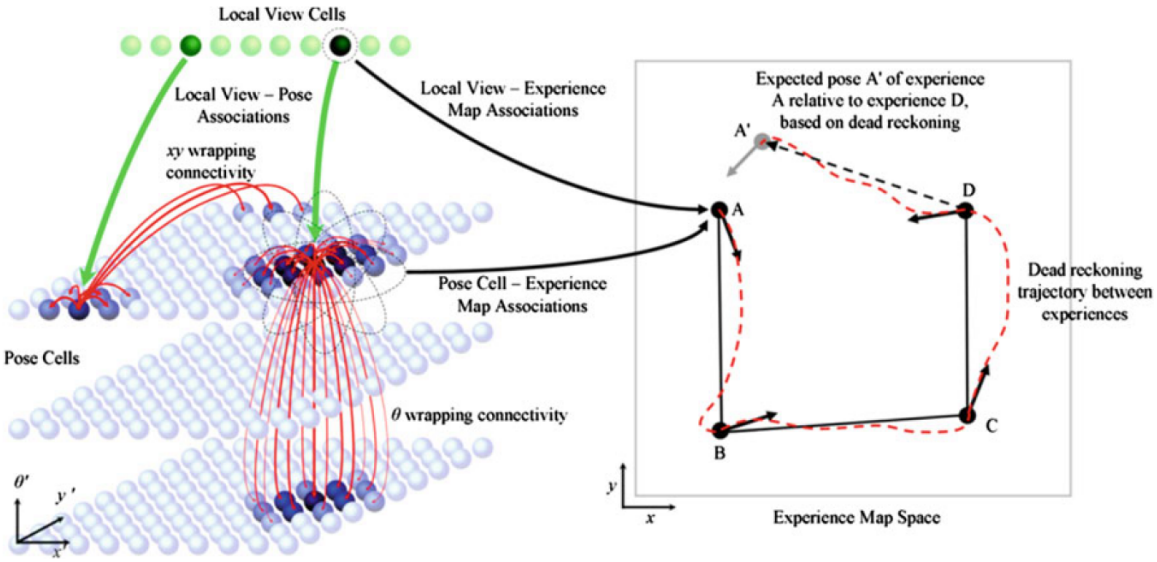


Figure 2.7: The three main RatSLAM modules. *Pose cells* are visualized by the layers of blue spheres. The green spheres on the top represent the *local view cells*. The *experience map* is shown on the right side. Figure source: [23]

Pose cells are structured in a three-dimensional prism in which the three dimensions represent the three degrees of freedom of a planar robot: x, y and θ. These pose cells form a network through excitatory connections that wrap across all boundaries of the network. By exciting these cells, so-called *activity packets* are created which resemble the network's estimate of the robot's current pose. When the robot moves, the odometry readings are used to excite cells in such a way that the activity packets follow that motion. In figure 2.7 these activity packets are represented by the pose cells colored in a darker blue. Figure 2.7 also shows an interesting feature of RatSLAM: it can keep track of multiple

[4] A specific area of the brain.

pose hypotheses at the same time. Which hypothesis is regarded as the best estimate is based on its excitation level. By also using inhibition[5] the system makes sure that activity packets which are not being excited anymore decay over time.

The local view cells are able to memorize distinct visual scenes in the robot's surroundings. When a new scene is detected it is saved in the local view cell. This local view cell is then linked to the pose cell in the center of the currently dominant activity packet. When the same scene is detected again the local view cell excites the pose cell through this saved connection. Through this process RatSLAM incorporates its own way of loop closing.

Due to the fact that the pose cells form a finite network but the connections wrap around the boundaries it can theoretically represent an infinitely large area. This does however come with a few problems: when wrapping around the edges it can happen that the same pose cells represent multiple physical places. This is also possible the other way around such that multiple cells represent the same physical place. The experience map represents a graphical map which keeps track of a unique pose estimate based on pose cell and local view cell information. An in-depth explanation of this part is given in [24].

Ball et al. [23] were able to show that the algorithm can produce detailed maps of different scale and under varying circumstances. Milford et al. however also state that the system "deals rather inefficiently with cyclic changes such as day-night time cycles" and "major changes to the topology or geometry of the environment" [25].

2.2.6 ORB SLAM

ORB SLAM is a visual SLAM approach which derives its name from the ORB feature descriptor. The ORB feature descriptor was developed by Rublee et al. [26] as "an efficient alternative to Scale Invariant Feature Transform (SIFT) and Speeded-Up Robust Features (SURF)". According to Mur-Artal et al. these features provide "good invariance to changes in viewpoint and illumination" [2] and are cheap to compute. As is obvious from the use of ORB features, ORB SLAM uses a feature-based front-end. The back-end works on a keyframe-based[6] graph-optimization procedure.

Mur-artal et al. utilize different techniques of other approaches like PTAM's parellelization of mapping and localization with the difference that ORB SLAM uses three parallel threads instead of two as in PTAM. An overview over the algorithm's structure and information flow is given in figure 2.8.

The first step that has to be taken for ORB SLAM to run is to initialize a map. This task is handled automatically and is described in [2]. A map consists of *map points* and *keyframes*. Each map point represents a 3D point in world coordinates and stores information about all the keyframes from which it was observed. Keyframes store information about the camera pose, the camera configuration parameters and the extracted ORB features for

[5] The decrease of excitation of a neuron over time.

[6] Instead of using every single camera image, keyframe-based approaches only use such images which minimize informational redundancy. By using such "key" frames the computational costs can be greatly reduced.

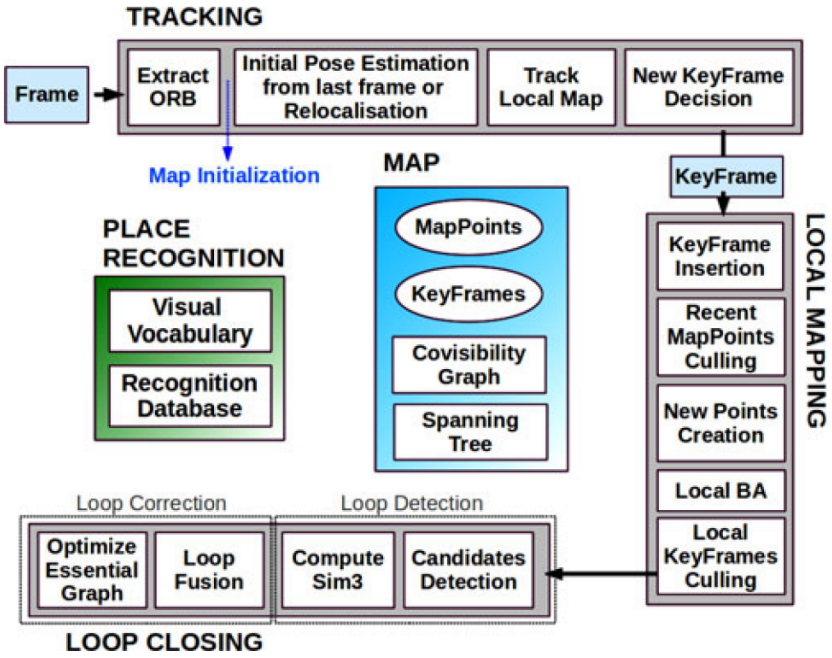


Figure 2.8: Overview of the ORB SLAM process. The white boxes inside a larger grey box are tasks which share a thread. Figure source: [2]

this frame. Keyframes and map points are used to build a *covisibility graph*. In this graph each keyframe is a node and the nodes are connected through an edge if they share at least 15 map point observations. Additionally a *spanning tree* is created that, starting with the initial keyframe, connects only those keyframes which share the most map point observations. The last graph introduced is the *essential graph*. This graph is a reduced version of the covisibility graph which holds the same nodes but only those edges which have a high covisibility and loop closing edges. This allows for a faster optimization of the global map without much accuracy loss. An overview over all graphs used is given in figure 2.9.

Figure 2.8 also shows the place recognition module which is used for relocalization after tracking loss and loop closures. This module saves representations of seen locations in a database for later matching.

As can be seen in figure 2.8, ORB SLAM works with three parallel threads. The tracking thread is responsible for tracking the movement of the camera. In a first step this thread extracts the ORB features. Then these features are matched with those of the previous frame. If this works, a constant velocity model is used to estimate the new camera pose. If it fails, a relocalization based on the place recognition database is initialized. Once the initial estimate has been retrieved, the local map is searched for correspondences between image features and map points. The local map consists of keyframes K_1 which share map points with the current frame and those which neighbor K_1 keyframes in the covisibility

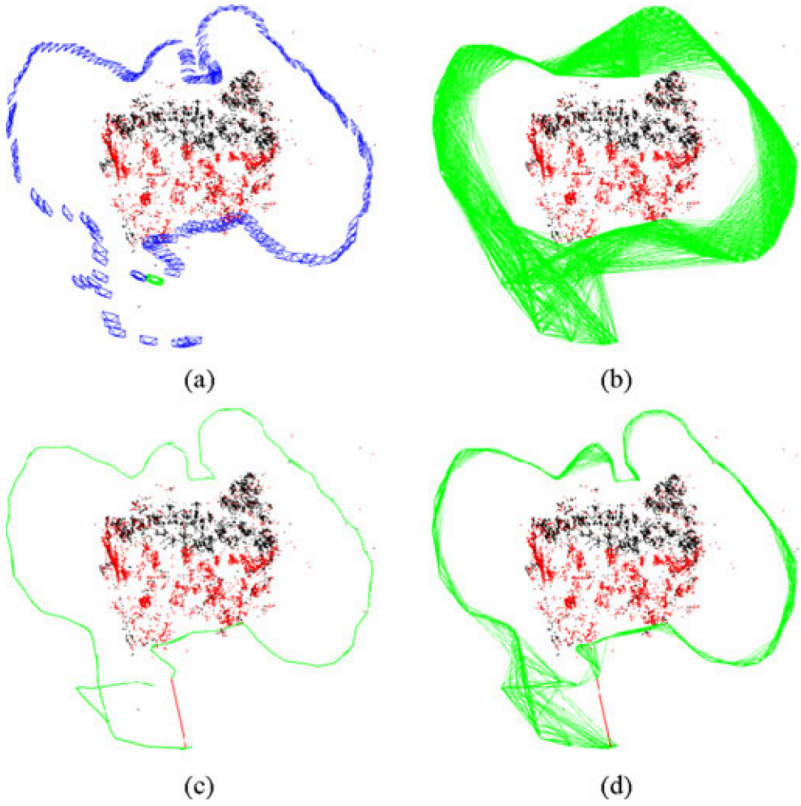


Figure 2.9: An overview over ORB SLAM's graphs. (a) shows keyframes in blue, current camera pose in green and map points in black and red (red are local map points). (b) shows the covisibility graph, (c) the spanning tree and (d) the essential graph. Figure source: [2]

graph. Doing this the estimated pose can be optimized with minimal computational cost [2]. The algorithm then decides whether the current frame is used as a new keyframe.

The local mapping thread keeps the local map up to date by inserting new keyframes into the covisibility graph and spanning tree. It also tracks whether new map points are being tracked in following keyframes. If not these are removed. By applying a local Bundle Adjustment (BA)[7] the current keyframe, all those keyframes connected to it and all map points seen by those connected keyframes are optimized. In order to keep the computation costs low redundant keyframes are gradually removed from the map.

Once local mapping is done with a keyframe it is passed on to the loop closing thread. Using this keyframe a search for loop closing candidates is started in the place recognition database. These candidates are then evaluated and if the loop is accepted it is incorporated in the covisibility graph. The last step is the optimization of the essential graph.

An interesting extension to ORB SLAM has been proposed in [3]. In this implementation

[7] An explanation for this technique can be found at 3.7.

IMU data is used as an additional information source which was shown to improve the accuracy of the overall process.

2.2.7 LSD SLAM

Much like the in section 2.2.4 described DTAM, LSD SLAM uses a dense front-end which works directly on pixel intensities. It changes the DTAM approach in such a way that the process is able to run in real-time on a Central Processing Unit (CPU) and incorporates loop closures.

By using a dense approach Engel et al. [27] attempt to overcome the problem of only being able to use visual information which complies with the used feature type as in feature-based approaches. By working directly on pixel intensities the algorithm is able to use all information in an image. An issue related to this is generally higher computational costs due to the large amount of pixels. In order to get around this, LSD SLAM uses a semi-dense formulation. Semi-dense methods only calculate the depth for those image regions which have maximum information gain for motion estimation. This is the case for image regions which have a non-negligible gradient. Figure 2.10 shows the difference between a dense and a semi-dense depth map.

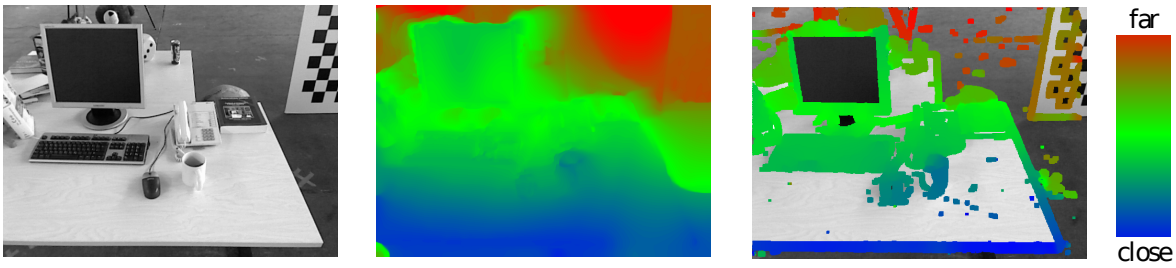


Figure 2.10: Difference between dense and semi-dense depth maps. Left: the original image, Middle: a dense depth map of the image on the left, Right: a semi-dense depth map of the image on the left. Figure source: [28]

The overall structure of the LSD SLAM algorithm is given in figure 2.11.

As can be seen the algorithm consists of three modules: tracking, depth map estimation and map optimization.

In LSD SLAM the map is created as a pose graph consisting of keyframes. Each keyframe contains a camera image, an inverse depth map[8] and the inverse depth map's variance. The edges in the map represent the transformation between keyframes using 3D similarity

[8] Depth estimation with a monocular camera can only be done through observing the same points from different viewpoints. This computation is done using the parallax (angle between viewpoint axes). For far away points this parallax is very low which makes computing the depth hard. Using the inverse depth notation improves the conditions of the depth estimation. More on this can be found in [29].

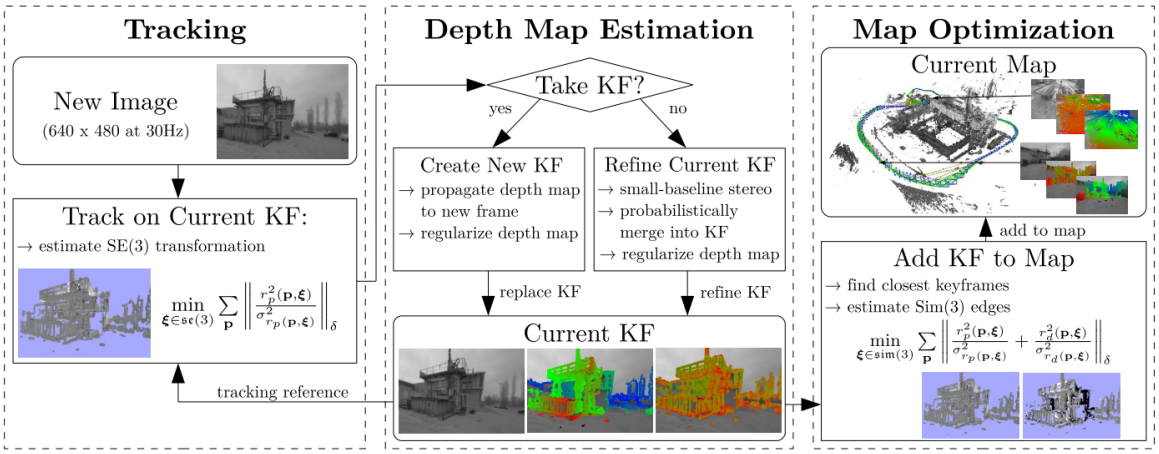


Figure 2.11: Overview of the LSD SLAM algorithm structure. Figure source: [27]

transforms[9]. This way of building a map is capable of taking scale drift into account. Scale drift is a problem that comes with using monocular cameras. These are unable to measure metric scale. Maps created by a monocular SLAM algorithm are generally relative to an unknown scale factor. Graphically speaking this means the algorithm does not know whether it maps a real room or a perfect miniature model of it. Through inaccuracies during map creation the scale can drift and through this introduce errors in the estimation.

In order to start tracking the algorithm first needs to be initialized. In LSD SLAM this is done through a random depth map. By then moving the camera the algorithm will converge to a correct depth configuration.

The tracking part then takes a new image and tracks motion by Photometric Error Minimization (PEM)[10] with respect to the current keyframe. The pose which minimizes this is the one corresponding to the new viewpoint.

The current keyframe is replaced by a new one as soon as the camera moves too far away from the existing map. The new keyframe's depth map is initialized by projecting known depths of the last keyframe into it. It then replaces the old keyframe and is used for future tracking. Such frames which do not become keyframes are utilized to increase the accuracy of the current keyframe's depth map. An in-depth explanation of this process is given in [28].

When a new keyframe is created it is incorporated in the map by aligning its edges with that of the previous keyframe. At this point the algorithm also checks for possible loop closures through comparison of other close-by keyframes and the usage of OpenFABMAP [32]. The created map is continuously optimized in the background by means of pose graph optimization using the g2o framework [33].

[9] Rigid body movements can be expressed as similarity transforms by using Lie-Algebra. More on this can be found in [30] and [31].

[10] Given two images try to overlap them in such a way that the accumulated pixel intensity differences are minimal. This can be done by translating, stretching, scaling and rotating the image.

2.3 Overview

As is obvious from the previous section, SLAM implementations vary a lot. As stated in 2.1 a complete SLAM approach consists of a front-end and a back-end. There are two kinds of front and back-ends:

Front-ends:

1. So called sparse, indirect or feature-based front-ends try to extract significant, easily recognizable points in images making use of feature descriptors as ORB, SIFT or SURF. Due to only working on specific image points, sparse methods are not as computationally expensive as dense methods and mostly run on a CPUs.
2. Direct or dense front-ends perform data association on the complete image by working on all pixels and using PEM. Since dense front-ends work on every pixel they are usually computationally expensive and often rely on GPUs for running in real time. One of their advantages is that they produce dense maps which are easily recognizable by humans as shown in figure 2.6. The problem of being computationally heavy can be somewhat reduced by using a semi-dense approach as done in LSD SLAM.

Back-ends:

1. Filtering back-ends like EKF SLAM or FastSLAM take the information provided by the front-end and feed it into a filter (e.g. an EKF) which then tries to make sense of it given the current state of its pose and map estimation. In these approaches the result only depends on the current filtered state and its current input.
2. Non-filtering back-ends are generally based on a graph representation of the observations produced by the front-end. These approaches solve the *full* SLAM problem by optimizing over the complete pose history. In most cases this is solved via BA techniques.

Due to the vast amount of different algorithms it is hard to get an overview over them all. There are several papers that try to survey the existing algorithms and examine which of these are considered state-of-the-art. However, since SLAM is such a rapidly developing research field as soon as these papers are published, new algorithms have already been released. Three of the most recent papers trying to give such an overview are:

1. Younes et al. [34]: A paper released in July 2016. Contains a large table on past and present filtering and non-filtering approaches. It also describes different non-filtering techniques and compares them with respect to data association, initialization, pose estimation, map generation and more.

2. Yousif et al. [35]: This paper published in November 2015 gives an overview about the basics of both visual odometry and visual SLAM.

3. **Cadena et al. [8]:** Summarizes the current state of SLAM and proposes different directions of where future research in SLAM could lead. This paper was published in April 2016.

Even though SLAM is such a widely researched topic it was not possible to find a list of existing approaches. This made it difficult to get an overview of which algorithms exist and what approaches have already been tried. Since there are so many different SLAM implementations out there, it is not feasibly to explain all of them in detail in the scope of this thesis. In order to spare future researchers the work of having to collect different SLAM approaches this thesis contains a list of all SLAM implementations that were found during the performed search. The list includes the algorithm's name, related references and, if available, a link to its code. It can be found at A.1.

2.4 SLAM in the Underwater Scenario

Over the years visual navigation algorithms like SLAM and visual odometry have been applied in a multitude of scenarios including: in the air (Unmanned Aerial Vehicles (UAVs)), on the road (autonomous cars), inside buildings or even on Mars [36]. Another interesting environment for such algorithms is in the water. There has been a large research interest in SLAM in the underwater scenario but due to the difficult conditions of the environment it is still considered "an unsolved problem in robotics" [37]. The following sections outline existing works which investigated SLAM in the underwater scenario and the challenges that are commonly encountered. Due to reasons explained in 2.5 the section focuses on non-filtering monocular approaches.

2.4.1 Existing Approaches

There are several works which try to give an outline of the current state of research on this topic. Paull et al. [38] for example explain different sensors used for communication and localization and also present a short review on multiple SLAM techniques used in the underwater environment. Two larger overviews over previous works in this field are given in [37] and [39]. Both papers contain a list of previous implementations. As can be seen there, most of these approaches focus on filtering techniques and use sonar as their main sensor.

Using a non-filtering SLAM approach with a monocular camera setup in a underwater environment has only been tested in few works so far. Li et al. [40] compare different open-source SLAM and visual odometry algorithms on eight different datasets. These datasets include data recorded in both underwater and land environments using different means of recording. Two datasets were recorded with a wheeled robot, both in and outside. Another two were recorded with a drone, again both in and outside. The third pair was recorded with an AUV once on a coral reef and once inside a wreck. The last two sets where recorded using handheld cameras. Once a camera was mounted on a drifter and moved by the waves alone and once it was moved manually. The qualitative results of the

survey are given in table 1. Quantitative results on some of the datasets can be found in [40].

The colors in the table indicate the performance of the given algorithm:

- Green: accurate results
- Yellow: localization running for the whole experiment, large deviation of estimated and real location
- Orange: camera pose was only tracked in some parts of the complete trajectory
- Red: localization did not work at all

Table 1: Qualitative performance of different open source packages in different environments. Datasets: wheeled robot outdoors (W/Out), wheeled robot indoors (W/In), drone outdoors (D/Out), drone indoors (D/In), AUV on coral reef (A/Out), AUV inside wreck (A/In), camera drifting (C/Dr), camera manual movement (C/Man). Table source: [40].

Package	W/Out	W/In	D/Out	D/In	A/Out	A/In	C/Dr	C/Man
MonoSLAM								
libVISO								
PTAM								
ORB SLAM								
SVO								
LSD SLAM								
RatSLAM								
COLMAP								
g^2o								
Ceres								

The best results on the underwater datasets (right four columns) are achieved by PTAM and ORB SLAM. COLMAP, g^2o and Ceres are packages used for graph optimization and use ORB SLAM's pose graph as input so they cannot be considered a standalone SLAM implementation. Li et al. also state that "ORB SLAM is the package that provides the best results in terms of accuracy" [40].

Another paper working on monocular underwater SLAM was published by Concha et al.[41]. They propose a semi-dense approach similar to LSD SLAM which incorporates the ability to differentiate between image areas with bad sight and such with good visual information. They do not use any form of optimization on their pose-graph and can therefore only map small areas. No information regarding the accuracy of the algorithm is provided.

Unfortunately, the works by Li et al. [40] and Concha et al. [41] were the only available sources of information that could be found on the topic of non-filtering monocular underwater SLAM at the time of writing this thesis. Any other works that were investigated

either used a filtering approach or a different sensor setup. Fortunately some of these works provide information which is also interesting with regard to the non-filtering monocular approach.

In [42] Negre et al. explore a new way of detecting loop closures. By forming and recognizing feature clusters they are able to show superior results in comparison to ORB SLAM's loop closing approach. Since their implementation relies on a stereo camera setup it is not directly applicable to the BlueROV2. Published code for their loop closure approach can be found at `https://github.com/srv/libhaloc`.

Chaves et al. [43] use an active SLAM approach which tries to increase the localization accuracy by actively searching for trajectories which maximize loop closures. They use a saliency prediction which helps create paths the robot can use for finding good loop closing opportunities.

Yet another approach has been put forward by Silveira et al.. In [44] they explain their DolphinSLAM approach which is derived from RatSLAM. Based on insights from [45] their front-end uses SURF for feature detection. They are able to show that the algorithm is capable of localizing an AUV in different environments. This is also not a purely camera driven SLAM variant since they incorporate a Doppler Velocity Log (DVL) and sonar into their framework.

2.4.2 Challenges

As mentioned before the underwater scenario bears many challenges which are not present on land. These includes:

Monotony: In many locations the sea floor consists mainly of sand and does not offer many recognizable structures which could be used for extracting landmarks. This may cause some SLAM algorithms to fail.

Turbidity: Seawater generally has a lot of different particles in it that can cause trouble when using cameras. Feature-based approaches might falsely detect particles as features which can have strong effects on the algorithm's accuracy.

Dynamics: Water tends to be highly dynamic so even if the robot is not actively trying to move, it is still subject to currents and water movement. Many SLAM algorithms rely on a kinematic model estimate the current pose which needs to take the water dynamics into account.

Loss of colours: Due to the effect that water absorbs different wavelengths of light at different depths the visual appearance of the environment quickly becomes monotonous. Color gradients can be a valuable information source which might not be available underwater.

Lighting: Not only does water absorb light, any movement of the water surface will also cause the light to scatter. This leads to dynamic shapes on the ground which

make it difficult for an algorithm to detect static landmarks. An example of this effect can be seen in figure 2.12.

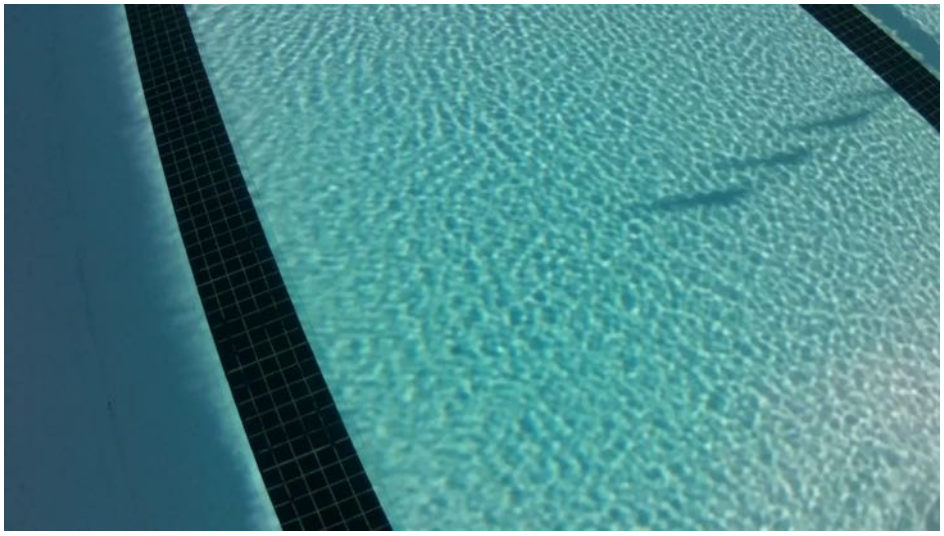


Figure 2.12: Ripples of light on the floor of the UWA pool.

Sensors: On land, devices can make use of GPS for localization. Underwater GPS does not work. Instead robots have to either rely on internal sensors like cameras, sonar and IMUs or on acoustic positioning systems which are expensive and limited to the region they are installed in.

Communication: Signals used by common communication measures like Wireless Local Area Network (WLAN) or radio do not travel well in water. A common way to communicate is to use acoustic transmission. The problem with these techniques is that they can have high latency and a low data transfer rate.

Processing: Due to size, cooling and cost constraints Autonomous Underwater Vehicles (AUVs) are mostly equipped with very little processing power and often rely on hardware like Raspberry Pis.

2.5 Algorithm Choice

An interesting observation that was made in the paper by Younes et al. [34] is that starting from about 2010 most new SLAM implementations were non-filtering approaches. So over the last seven years research focus shifted from filtering to non-filtering techniques. As a matter of fact the question whether a non-filtering approach is generally superior to filtering was posed by Strasdat et al. in the same year [46]. In this paper they compare an EKF algorithm with using BA for optimization and come to the conclusion that “filter-based SLAM frameworks might be beneficial if a small processing budget is available, but that

BA optimisation is superior elsewhere". Based on the general direction SLAM approaches have taken and the results of [46] it was decided to also focus on non-filtering approaches for this thesis.

The choice of applicable algorithms is further reduced by the given conditions. Which sensors can be used is limited to the sensors available on the BlueROV2: a camera and the Pixhawk's integrated IMUs. Therefore the only SLAM algorithms that are feasible are ones which do not rely on any other sensors.

These two constraints: focusing on non-filtering approaches and having only a camera and an IMU as sensors shorten the list of SLAM algorithms at A.1 significantly. Those that remain are listed at A.2. This list also contains some more information about the algorithms listed there. It describes which front-end / back-end is used and includes some general notes on how the algorithm works and what separates it from other approaches.

Due to the fact that visual SLAM is commonly a very computationally expensive task, processing power is another constraint that has to be taken into account. Which hardware can be used relies on the way the ROV is used, especially on whether a tether-connection (Local Area Network (LAN)) can be used for communication or not. If the robot can stay connected to a computer on land or a boat the computations can be run on that computer. If however the tether would become a hazard, when mapping caves or reefs for example, it would be preferable to be able to run all processing on the ROV itself. In that case the robot would have to operate completely autonomous while it is underwater. Since that was not possible at the time of writing this thesis the ROV was connected via tether during all experiments. This also means that a normal laptop can be used for processing. The algorithm choice should keep in mind that the robot is supposed to work autonomously at some point. Considering that the BlueROV2's processing power is constrained to that of a Raspberry Pi 3, modern visual SLAM algorithms would most likely not be able to run directly on the ROV's hardware without considerable time invested into runtime optimization. Algorithms which definitly are too computationally heavy are those which rely on a GPU for processing. For that reason only implementations which are able to run on a CPU are going to be considered.

Another important factor is that the ROV should be able to create and record a map of its environment. This also rules out all VO implementations listed in A.2.

Also, since it would not be possible to rewrite a complete SLAM algorithm in the time frame of this thesis the software has to be open source.

Based on these five criteria:

1. non-filtering
2. works with only camera, potentially integrates IMU
3. runs on a CPU
4. creates and records a map
5. is open source

only the following algorithms remain a possible choice:

- Dense Piecewise Planar Tracking and Mapping (DPPTAM)
- Deferred Triangulation (DT) SLAM
- LSD SLAM
- ORB SLAM
- PTAM
- Robust Keyframe-based (RK) SLAM

As can be seen in table 1, ORB SLAM was able to show that its results on underwater data were superior to LSD SLAM [40]. Even though PTAM actually performed better in the underwater sequences evaluated in 1, results from other evaluations like the one in [2] or the results shown on the KITTI datasets[11] suggest that ORB SLAM is more accurate than both PTAM and LSD SLAM. DPPTAM is an approach that heavily relies on planar surfaces which are not common in the underwater environment. Even though RK SLAM released an executable for their method, there is no actual code available. DT SLAM actually has some very interesting ideas like being able to track 2D features and creating several sub maps which can be fused. However it was not possible to find any evaluation regarding this method. ORB SLAM provides three well written papers ([2], [3], [4]), well documented code and has been thorougly evaluated. It also comes with extra features like a localization mode that allows to record a map and then use it for localization without further extending it. In [4] it is shown that IMU integration is possible and that it can be used to further improve on ORB SLAM's accuracy. Furthermore there are also implementations of it for both Android and iOS phones that would allow to add a phone to the BlueROV2 for additional processing power. So for all the reasons stated the final choice for a SLAM method fell on ORB SLAM.

[11] `http://www.cvlibs.net/datasets/kitti/eval_odometry.php`

3 ORB SLAM

The following chapter explains the ORB SLAM algorithm. While the basics of it were already presented in 2.2.6 this section focuses on a more in-depth explanation of the algorithm's modules and the techniques that are used within them.

ORB SLAM mainly consists of seven separate modules:

1. Feature extraction
2. Data association
3. Initialization
4. Tracking
5. Relocalization
6. Local mapping
7. Loop closing

In order to make it easier to understand how these modules work together and to get a visual overview over ORB SLAM's code, a Unified Modeling Language (UML)-like diagram was created. It can be viewed at A.4. A legend explaining the diagram is given at A.3.

3.1 Feature extraction

The figure at A.5 shows the part of A.4 which is responsible for extracting the image features. As is visible there the first thing ORB SLAM does is it creates an image pyramid. When creating an image pyramid the original image is blurred and subsampled and the resulting image is added as a layer on top of the original image. By doing this a given number of times it results in a pyramid like construct with the original image at the bottom and layers of increasingly smaller images on top of it. Figure 3.1 shows such a pyramid with four layers and a scale factor of 2.

Doing this allows to run a feature extraction on different scale levels. Due to the nature of image features like Features from Accelerated Segment Test (FAST), a feature extraction might miss out on significant features when only looking at one level of scale. Using an image pyramid is a way of making sure that this does not happen. This is an approach that was also taken by Klein et al. [20] in their PTAM algorithm. Instead of using only four levels with a scale factor of 2 like in PTAM, Mur-Artal et al. chose to use eight levels with a scale factor of 1.2 [1].

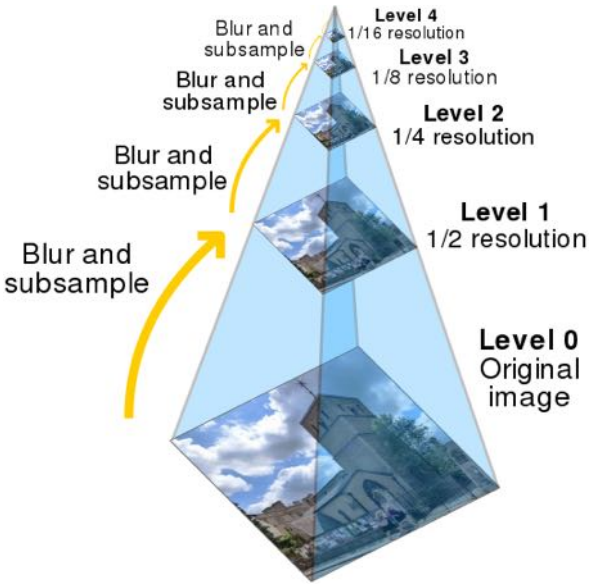


Figure 3.1: An image pyramid with four levels and a scale factor of 2. Figure source: [47]

As a second step ORB SLAM then computes FAST corners on every pyramid level. FAST is a corner detection which was first published by Rosten and Drummond in 2006 [48]. ORB SLAM does not use the machine learned approach described in [48] but uses the OpenCV implementation which is shown in figure 3.2.

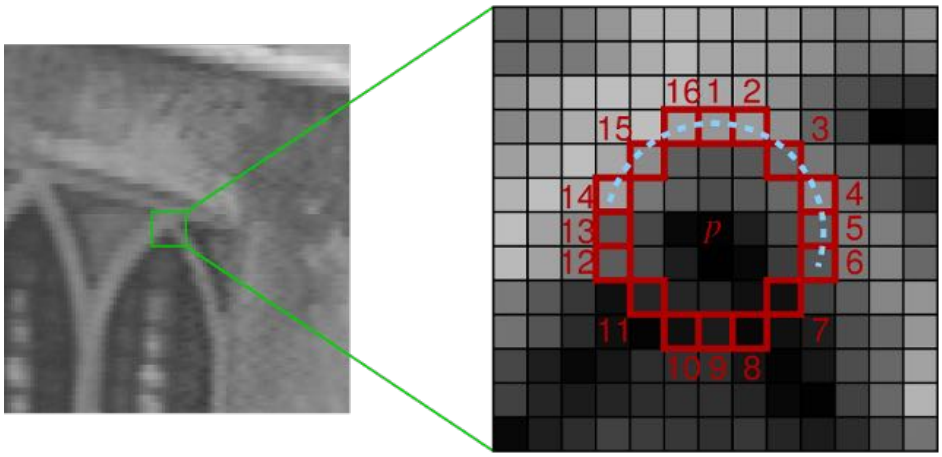


Figure 3.2: Principle of FAST corner detection. For each pixel p in an image a circle with a radius of three pixels around it is considered. A corner is detected as such when at least nine contiguous pixels in this circle are either brighter or darker than p plus a threshold t. By summing up the absolute intensity difference between p and the 16 pixels around it and then keeping only the one with the highest value, duplicate detection of the same corners can be avoided. Figure source: [48]

The advantage of FAST is, that in comparison to other corner detectors like e.g. Harris it is, as its name suggests, a lot faster. However, it is not very robust to noise. A comparison of different methods and their speed is given in [48].

ORB SLAM divides every image in the image pyramid into cells of 30 by 30 pixels and tries to compute FAST corners for each of them. If it does not succeed (cannot detect a single corner) with the initial threshold $t_{init} = 20$ it tries again with a lower threshold $t_{min} = 7$. This way the algorithm makes sure to treat different image regions differently. It then proceeds to distribute the computed corners over each image in the image pyramid. This is based on a maximum desired amount of features n_f which can be configured. To achieve an equal distribution the algorithm will, given it found any corners, divide the image into four equally large cells and then divide these cells again into four equally large subcells. This is repeated until all remaining cells either contain exactly one corner or the amount of cells n_c is equal to n_f. If the latter is the case it will only keep the corner with the highest score for each cell that contains more than one. The rest is simply discarded.

After the detection of FAST corners the next step is to calculate the Oriented Fast and Rotated Brief (ORB) feature descriptor. As its name indicates it is an extension and combination of FAST and Binary Robust Independent Elementary Features (BRIEF). The idea for this descriptor was proposed by Rublee et al. in 2011 [26]. In its original version FAST did not contain any orientation component. ORB adds this by using so-called *intensity centroids* [49]. When defining the moments of an image patch as:

$$m_{pq} = \sum_{x,y} x^p y^q I(x, y) \tag{3.1}$$

m_{pq} Image moment
$I(x, y)$ Intensity at position (x, y)

these moments can be used to compute a *centroid* C:

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right) \tag{3.2}$$

The orientation of the corner can then be calculated by:

$$\theta = atan2(m_{01}, m_{10}) \tag{3.3}$$

As mentioned before ORB also improves on the BRIEF descriptor. The BRIEF descriptor was designed by Calonder et al. in 2010 [50]. It is binary string descriptor which allows for comparison using the Hamming distance[12]. This allows for much faster feature comparison than using the L_2 norm as done in other approaches.

[12] The Hamming distance between two strings which are equally long is equal to the amount of corresponding symbols in which they differ. For binary strings this equals the number of ones in the result of a XOR operation on the given strings.

The BRIEF descriptor is defined as a vector of n binary tests:

$$f_n(p) = \sum_{1 \leq i \leq n} 2^{i-1} \tau(p; x_i, y_i) \tag{3.4}$$

f_n feature descriptor
p an image patch of size S x S pixels

with the binary test $\tau(p; x_i, y_i)$:

$$\tau(p; x_i, y_i) = \begin{cases} 1 & if\ p(x) < p(y) \\ 0 & otherwise \end{cases} \tag{3.5}$$

$p(x)$ intensity of a pixel in a smoothed version of p at $x = (u, v)^T$

Figure 3.3 shows an example of how the test points are chosen.

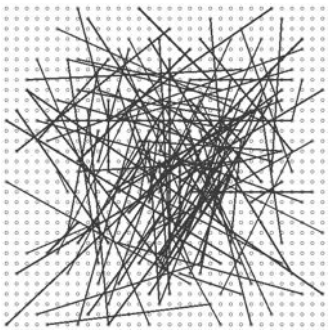


Figure 3.3: A pattern resulting from the way the test points (x_i, y_i) are chosen for the BRIEF feature descriptor. Figure source: [50].

This results in four things which have to be chosen when using BRIEF descriptors:

1. the length of the binary vector n,
2. the patch size S,
3. the spatial arrangement of the chosen test points (x_i, y_i) and
4. the way of smoothing the image patches.

For ORB Rublee et al. chose $n = 256$ and $S = 31$. They also use an integral image approach for smoothing the image patch.

Even though Calonder et al. were able to show that BRIEF achieves better performance than SIFT or SURF, BRIEF suffers from not being able to handle in plane rotation.

For this reason ORB introduces a rotated BRIEF variant (rBRIEF). rBRIEF uses a optimized pattern for picking the test points which was learned from a large set of points by maximizing variance and minimizing correlation. The pattern is shown in figure 3.4.

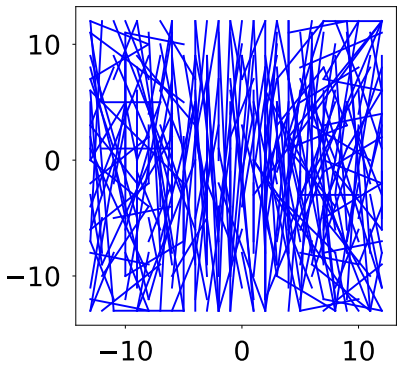


Figure 3.4: A plot of the rBRIEF pattern used by ORB and ORB SLAM.

An exact explanation of how this pattern was created can be found in [50]. This pattern is then rotated to match the orientation which was previously recovered from the FAST corner which allows for ORB to be resistant to rotational changes.

After extracting all features ORB SLAM proceeds to undistort them using the camera calibration parameters the user needs to provide. So ORB SLAM does not actually ever undistort the whole image but only the features after detecting them.

Figure 3.5 shows an exemplary result of ORB SLAM's feature extraction.

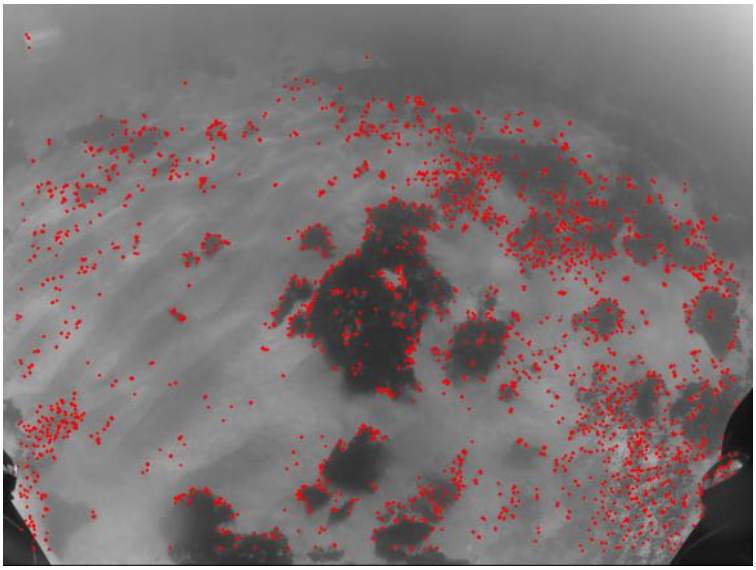


Figure 3.5: ORB SLAM's feature extraction run on an image captured with the BlueROV2. The red dots resemble found features using $n_f = 4000$.

3.2 Data Association

Unlike the other modules feature matching does not happen in one certain place in the code. Different ways of finding correspondence between features in separate frames exist. For initialization for example the 2D features detected in the current frame have to be associated with the 2D features detected in the initial frame. Later when there is a map, 3D map points have to be associated with 2D features in the current frame. In general ORB SLAM uses three different ways of achieving this:

1. search by projection,
2. search by bag of words and
3. search by similarity transformation.

Since which ,approach is used depends on the individual module, how data association is performed is explained in the section of that module.

3.3 Initialization

Initialization is a large module. An overview over it is given in A.6 and A.7. As is visible in A.6 initialization is only started if there were at least 100 features extracted from the current and the last frame. When the initialization is started the first frame is used as an initial frame against which the following frames will be compared.

In order to find an association between the 2D features seen in the initial frame and the ones seen in the current one, every feature is compared with regard to two properties: its descriptor distance and its orientation. First for each feature in the initial frame all features in a small area (200 x 200 pixels) around this feature in the current frame are collected. In order to avoid having to confirm for every feature whether it lies in the desired area the images are divided into a cell grid with 64 columns and 48 rows. So the features which are recovered from the area around the feature in the initial frame are not actually recovered from an area within 200 x 200 pixels but rather from all cells which lie inside that reach. Then for each feature within that range the descriptor distance to the initial feature is calculated. Two features are considered to be matching with regard to distance when the descriptor distance is less than 50 and the shortest distance found is shorter than 0.9 times the second shortest distance.

If the distance check was ok the orientation of every match is checked. For this all distance matches' orientations are sorted into bins of 12 degrees size. If a match's orientation should not be within the three most common bins it is not considered a match anymore.

Initialization only proceeds if at least 100 matches were found amongst all detected features. If so, the algorithm computes a homography H and a fundamental matrix F in parallel. A homography relates points seen from different viewpoints as shown in figure 3.6. It fulfills the following equation:

$$x_i' = Hx_i \tag{3.6}$$

$x_i \in \mathbb{R}^3$ Image point in first coordinate frame (in homogeneous coordinates)
$x_i' \in \mathbb{R}^3$ Image point in second coordinate frame (in homogeneous coordinates)

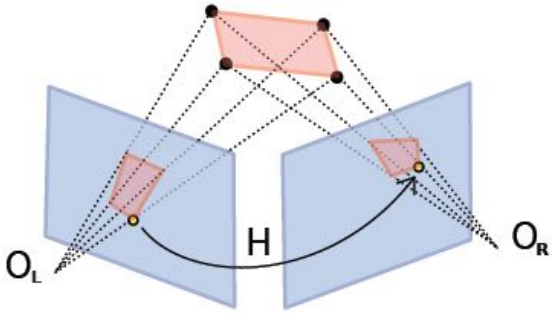


Figure 3.6: A homography relates image points belonging to the projection of commonly seen points in a plane. The homography matrix H directly maps from an image point belonging to the projection in coordinate system O_L to the corresponding image point in coordinate system O_R, if the intrinsic camera parameters are the same. Figure source: [51]

A fundamental matrix also relates corresponding points in stereo images with each other. It does however not rely on the scene being planar. As shown in figure 3.7 the fundamental matrix F describes the epipolar geometry of a scene. More on this topic can be found in [52]. A fundamental matrix F needs to fulfill:

$$x_i'^T F x_i = 0 \tag{3.7}$$

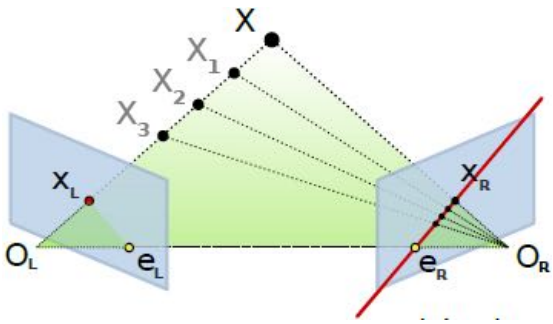


Figure 3.7: The fundamental matrix F constrains where the projections of a commonly seen point has to lie. Given a projection x_L in one image the corresponding point in the other image is constrained to a line Fx_i. Figure source: [51]

Both matrices are estimated in a Random Sample Consensus (RANSAC) algorithm as described in [52] and [2]. By then projecting multiple matches from one frame into the

other using the computed transformations, ORB SLAM calculates a score based on the reprojection error. Whether to use the homography or the fundamental matrix is decided based on a heuristic defined as:

$$R_H = \frac{S_H}{S_H + S_F} \tag{3.8}$$

R_H Decision heuristic
S_H Homography score
S_F Fundamental matrix score

If $R_H > 0.40$ the homography is chosen for pose recovery, otherwise the fundamental matrix. In the case of the homography eight motion hypotheses are recovered from a process explained in [53]. In the fundamental matrix case four hypotheses are reconstructed. How this is done is explained in [52].

In both cases all hypotheses are checked by triangulating all previously determined feature matches and checking whether they lie in front of both cameras, the parallax is big enough and reprojection error is small enough. The triangulation is done using a Direct Linear Transformation (DLT) algorithm as described in [52]. This algorithm needs the two corresponding homogeneous 2D points $x, x' \in \mathbb{R}^3$ and the two camera matrices $P, P' \in \mathbb{R}^{3x4}$. Where:

$$x = PX \tag{3.9}$$
$$x' = P'X \tag{3.10}$$

hold, with $X \in \mathbb{R}^4$ being the corresponding 3D point in homogeneous world coordinates. X can then be determined via:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix},\ x' = \begin{bmatrix} x'_1 \\ x'_2 \\ 1 \end{bmatrix},\ P = \begin{bmatrix} P_1^T \\ P_2^T \\ P_3^T \end{bmatrix},\ P' = \begin{bmatrix} P_1'^T \\ P_2'^T \\ P_3'^T \end{bmatrix},\ A = \begin{bmatrix} x_1 P_3^T - P_1^T \\ x_2 P_3^T - P_2^T \\ x'_1 P_3'^T - P_1'^T \\ x'_2 P_3'^T - P_2'^T \end{bmatrix}$$

$$[U, S, V] = \text{SVD}(A)$$

$$\text{with } U, S, V \in \mathbb{R}^{4x4},\ V = \begin{bmatrix} V_1^T \\ V_2^T \\ V_3^T \\ V_4^T \end{bmatrix} \text{ and } V_i^T = [v_{i1}, v_{i2}, v_{i3}, v_{i4}]$$

$$X = \frac{1}{v_{44}} V_4 \tag{3.11}$$

Note that the last step in 3.11 has to be performed because the matching points have been normalized before. The minimum parallax is set to one degree and at least 90% of all matches need to be correctly triangulated with a hard cap set to a minimum of 50

points triangulated. There are also a few more constraints depending on whether the homography or the fundamental matrix hypotheses are being used.

In case of using the homography the hypothesis with the second best result (second most matches which are considered to be inliers after checking reprojection error) has to have less than 0.75 times the amount of inliers the best hypothesis has. In the fundamental matrix case no two other hypotheses are allowed to have more than 0.7 times the amount of inliers the best hypothesis has. Using these constraints the initialization module makes sure that the initialization is only accepted when there is a hypothesis which is clearly better than the others and avoids initializing badly.

Once the initialization is accepted an initial map is created using the initial and the current frame as the first two keyframes and all the triangulated matches as initial map points. At this point a first global BA is run on the initial map. If there should not be at least 100 map points in the map after this the initialization will again be discarded and the process repeated.

3.4 Tracking

When the initialization was successful the tracking module is going to be used to estimate the movement of the camera and map the surroundings. This module's code overview is given in A.8 and A.9. As can be seen in A.8 tracking is done in one of two ways depending on whether a motion model exists or not. ORB SLAM uses a constant motion model for its calculations. When tracking was successful in the last frame it will simply assume that the camera will again move the same way for this frame, this allows the algorithm to use a simpler approach for tracking.

Tracking with a motion model:

When using a constant motion model the algorithm can run data association by assuming that the features which were mapped to a map point in the previous image can be found within a certain area in the new image which is determined based on this motion. How large that area is depends on which level of the scale pyramid the feature was extracted from. For a feature found in the base scale it will check an area of 14 by 14 pixels. Should this not bear more than 20 matches with the previous frame it will try again with a window of 28 by 28 pixels. Every feature of the current frame which is accepted during this search will also be associated with the map point the matched feature from the last frame was associated with. If even after the second search less than 15 matches were made the tracking will be retried without using the motion model.

Tracking without a motion model:

Should tracking with a motion model fail or no motion model exist ORB SLAM will take a different approach to tracking. Here instead of projecting features into the current frame, the current frame's features are transformed into a bag of words vector and then compared with features seen in the current keyframe.

For this ORB SLAM uses the DBoW2 library. DBoW2 was developed by Gálves-López and Tardós in 2012. Their approach is described in [54]. In general the bag of (visual) words technique is a way of converting an image in such a way that it is represented by a sparse numerical vector. Using these vectors DBoW2 creates a hierarchical database as depicted in figure 3.8.

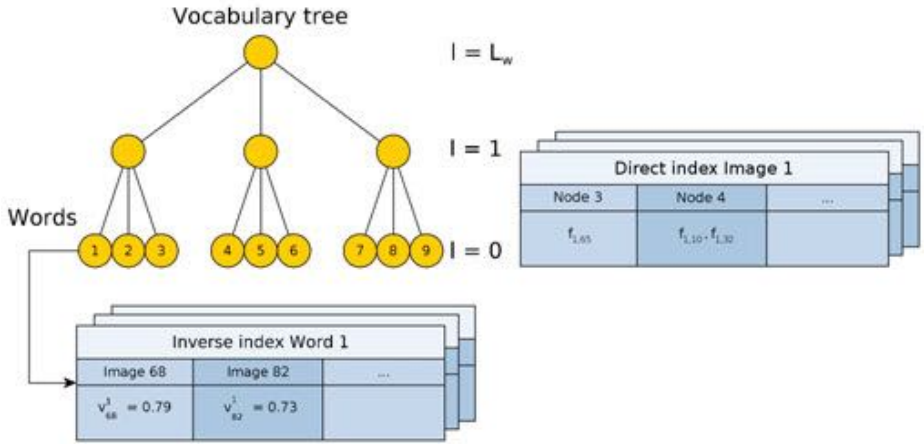


Figure 3.8: A sketch of the database structure formed by DBoW2. The nodes in the tree represent clusters of feature vectors. The root node at level $l = L_w$ holds all stored feature descriptors. To create a new level in the tree the feature descriptors are clustered into k_w clusters using k-median clustering based on their Hamming distance. This is repeated until the whole tree is created. The leaves then represent the W visual words. In DBoW for every word a so called inverse index is saved which is a list of pairs $< I_t, v_t^i >$. I_t being an image in which that word was seen and v_t^i the weight for that word in that image. Furthermore the database also contains a direct index which keeps track of all feature descriptors in a given image. Figure source: [54]

The database described in 3.8 is created in an offline step in advance to the actual algorithm using it. If the training set is sufficiently big the resulting vocabulary can also be used in different scenarios than the training images came from.

When now trying to find images similar to a given query image the feature descriptors extracted from the query image are compared with the tree nodes. For each feature descriptor the tree is traversed from the root to the leaves and a corresponding word is found. The corresponding word is the one for which the Hamming distance between the feature descriptor and the word's centroid is minimal. With this process it is possible to determine which words are present in the query image and how often each word occurs. By then simply creating a bag of words vector $\nu_t \in \mathbb{R}^W$ which holds how often each word was seen in the image, the query image can be compared to other images by using the L_1-score $s(\nu_1, \nu_2)$:

$$s(\nu_1, \nu_2) = 1 - \frac{1}{2} \left| \frac{\nu_1}{|\nu_1|} - \frac{\nu_2}{|\nu_2|} \right| \tag{3.12}$$

For ORB SLAM the database was created with a dataset of roughly 10000 images of both outdoor and indoor scenarios. The database was built with $L_w = 6$ and $k_w = 10$ which results in a total of approximatively one million words [1].

The tracking module can make use of this database by comparing the current frame's features only with those features in the current keyframe that belong to the same node in the database tree at a certain level. For these it then tries to find the feature that has the lowest Hamming distance below a given threshold. Again, when the search results in less than 15 matches tracking will be aborted. If this happens relocalization will be triggered.

Both the tracking approach with motion model and the one without will, given that they were able to find 15 or more matches, use these to optimize the previously assumed pose. This optimization is done using the g^{2}o library [33]. Since after the optimization some matches might not be considered matches anymore, the algorithm will do another check on the amount of matches remaining once the optimization is completed.

If the tracking was successful up onto this point the second part of tracking will be triggered. This part is shown in A.9. In order to keep the processing times minimal ORB SLAM keeps a local map which only contains the set of keyframes K_1 which share map points with the current frame and a set of keyframes K_2 which are direct neighbors of those in K_1 in the covisibility graph. Now that the algorithm has an estimate of the current camera pose as well as some map points tracked in the current frame it will use its local map to further refine the pose estimation. This is done in five steps as described in [2].

For all map points in both K_1 and K_2:

1. Compute the projection into the current frame.
2. Compute the angle between current viewing ray ν and the map point's mean viewing direction n. Discard if $\nu \cdot n < cos(60°)$.
3. Calculate the distance d from map point to camera center. Discard if it is out of the scale invariance region of the map point $d \notin [d_{min}, d_{max}]$.
4. Compute the scale in the frame by d/d_{min}.
5. Compare the representative descriptor D[13] of the map point with the unmatched features in the current frame, at the predicted computed scale and close to its projection in the frame. Associate the map point with the best match.

Once this process was done for all map points in the local map the pose will again be optimized using the newly created matches. If after the optimization less than 30 matches

[13] The associated ORB descriptor whose Hamming distance is minimal with respect to all other associated descriptors from keyframes in which it is observed [2].

between map points and the current image's features remain, tracking will be aborted and relocalization started. If the algorithm has just recently (within the last five frames) relocalized itself the minimum matches necessary are raised to 50.

When tracking succeeds the last decision the tracking module has to make is whether the current frame should become a keyframe. For this to happen four conditions have to be met:

1. The last relocalization must have happened more than 20 frames ago.
2. Local mapping is idling or more than 20 frames have passed since last insertion of a keyframe.
3. The current frame tracks at least 50 points.
4. The current frame tracks less than 90% of the points in the last keyframe.

This point also marks the end of the tracking cycle. Once this part is done the next frame will be loaded and processed.

3.5 Relocalization

When tracking fails completely ORB SLAM falls back to relocalization mode. When in this mode, the algorithm will try to find a keyframe which views the same scene that is visible in the current frame. This also means that no further mapping or localization is done until the algorithm has found a view it already knows. ORB SLAM uses the bag of word approach explained in 3.4 for relocalization. The first step that is taken is to convert the image into the bag of words representation. Then the database is queried to find all keyframes which are similar to the current frame. To find these candidates the following steps are taken:

1. Find all keyframes which share at least one word with the current frame.
2. Find the keyframe that shares the most words w_{max} with the current frame. Discard all keyframes which share less than $0.8 \cdot w_{max}$ with the current frame.
3. Compute the similarity score as in equation 3.12 for all remaining candidates.
4. Find the keyframe with the best score s_{max}. A keyframe's score is calculated by taking its own score and adding to it the score of the 10 keyframes with best covisibility (given they are a part of the candidates as well). Discard all keyframes which have a score lower than $0.75 \cdot s_{max}$.

Should this not yield any candidates the process is stopped and will be retried with the next frame. If there are candidates the algorithm will do a search by using the bag of words approach as done in the case of tracking without a motion model (3.4). Any candidates

that achieve less than 15 matches are directly discarded. For all remaining candidates the algorithm will try to estimate a pose from the matches made. This is done via the EPnP algorithm described in [55]. The EPnP algorithm allows for estimating the pose of a calibrated camera by using n 3D to 2D point correspondences. Using the correspondences made through the previous matching, ORB SLAM runs five iterations with $n = 4$ randomly sampled points and tries to recover a pose from these. A pose is considered to be found when at least 10 points of the previously determined matches demonstrate a small enough reprojection error. If so the algorithm will use the recovered pose and optimize it using all point correspondences. Should there be less than 50 matches after this the algorithm will try to refine the pose up to two times by first finding more matches through projecting map points tracked by the candidate keyframe into the current frame and then optimizing the pose again. The exact way this is done is illustrated in A.10. A candidate keyframe is only accepted for relocalization when at the end of all checks at least 50 matches between the candidate's tracked map points and the current frame's features were found.

3.6 Local Mapping

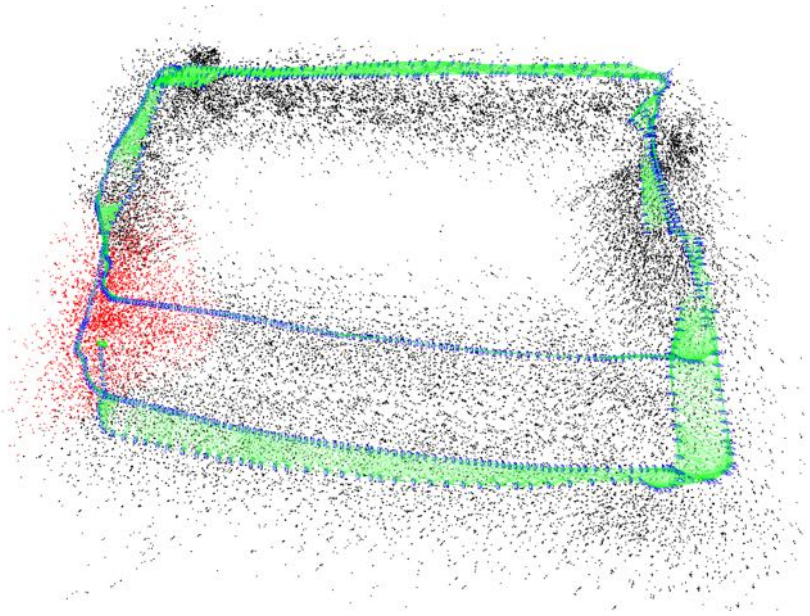


Figure 3.9: A map created and visualized by ORB SLAM. The blue shapes resemble estimated camera poses for keyframes. Green lines connect those keyframes which are connected in the covisibility graph. Black and red dots visualize map points. Map points drawn in red belong to the current local map. The local map is defined by the sets of keyframes K_1 and K_2. K_1 contains all keyframes which share map points with the current frame. K_2 consists of all keyframes which are direct neighbours of a keyframe in K_1 within the covisibility graph.

As described previously ORB SLAM keeps a local map of its current surrounding. The

local mapping module is the part of the algorithm which is responsible for keeping this map up to date by inserting and removing both keyframes and map points and constantly optimizing it. In order to be able to do this permanently this module runs in its own thread.

The general structure of this module is depicted in A.11. The module is run in a loop where it checks every 3 ms whether there are any new keyframes which have to be integrated in the local map. Once a new keyframe arrives the following steps are taken:

1. Integrate the new keyframe into the local map by updating all necessary connections and objects.
2. Check whether map points in the local map can be deleted. A map point will be deleted in two cases:
   (a) Its ratio between how often it should be visible from a frame and how often it was actually matched is lower than 0.25.
   (b) It has been matched less than three times and more than one keyframe has been added since it was first included in the local map.

   If a map point is not deleted within the first two keyframes it will stay in the map and will not be deleted anymore. Note that ORB SLAM does not actually delete map points from memory. It simply marks them as unusable.
3. New map points are created by triangulating feature matches between the new keyframe and the 20 keyframes which share the most map points with the current keyframe. Only feature matches which are not yet related to a certain map point are triangulated.
4. If there are no new keyframes waiting for insertion, search for duplicate map points and fuse them.
5. If there are still no new keyframes for insertion, perform a local BA and afterwards remove such keyframes which share at least 90% of their map points with at least three other keyframes.

Bundle Adjustment (BA) is a technique which is widely used in graph based SLAM algorithms. As described in equation 3.9 the projection from a 3D point X to its corresponding 2D point x in an image can be achieved through the camera matrix P. However in a realistic scenario cameras are subjected to noise so this relation may not always be fulfilled correctly. When now faced with a scenario where a set of cameras with respective camera matrices P^i sees a set of 3D points X_j, as shown in figure 3.10, BA allows to optimize the solution, assuming Gaussian noise.

The goal of BA is to estimate the camera matrices $\hat{P}^i$ and 3D points $\hat{X}_j$ whose corresponding projections $\hat{x}^i_j$ correctly fulfill: $\hat{x}^i_j = \hat{P}^i \hat{X}_j$, while also minimizing the reprojection error between the reprojected point and the measured x^i_j for every frame it is viewed in [52].

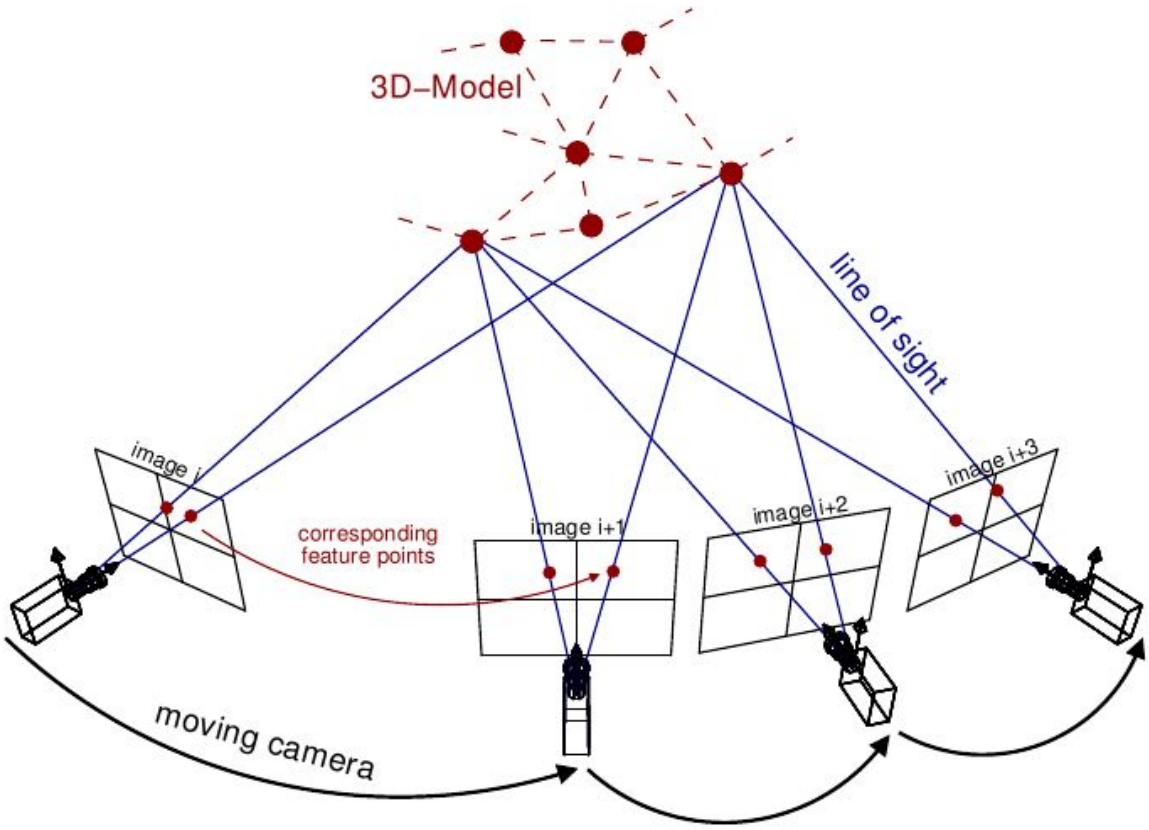


Figure 3.10: Illustration of the situation BA is used in. In theory the lines of sight corresponding with an observation of a 3D point from multiple points of view should cross in that point in 3D space. Given the noise cameras are subject to this is likely not to be the case in a real scenario. Figure source: [56]

This means BA minimizes the reprojection error function:

$$\min_{\hat{P}^i, \hat{X}_j} \sum_{ij} d(\hat{P}^i \hat{X}_j, x^i_j)^2 \tag{3.13}$$

where $d(x, y)$ stands for the image distance between homogeneous points x and y.

Due to the large number of unknowns this can quickly become an unmanageable task. Every camera matrix $\hat{P}^1$ comes with 11 degrees of freedom and every 3D point $\hat{X}_j$ with 3. So for a scenario with m views and n 3D points the algorithm needs to find a solution for $3n + 11m$ parameters.

To account for this problem ORB SLAM uses the g^{2}o library for its graph optimization needs. g^{2}o is able to deal efficiently with large amounts of parameters by restructuring the optimization problem based on its sparse properties. More information on the topic can be found in [33].

3.7 Loop Closing

Much like the local mapping module the loop closing module also runs in its own thread. This is due to the fact that it needs to observe incoming data permanently in order to be able to detect loops. In ORB SLAM loop closing is done in three steps:

1. Detecting a loop (diagram at A.12)
2. Confirming the loop (diagram at A.13)
3. Refining the loop (diagram at A.14)

Loop Detection

In analogy to local mapping ORB SLAM only looks for loops when a new keyframe is available. As is evident in A.11 the loop detection will be notified of a new keyframe as soon as local mapping is done processing it. Once loop closing receives the new keyframe it will start to look for other keyframes connected to it via the covisibility graph. For each of these it will compute a score according to equation 3.12. Given there are at least 10 keyframes in the map and the last loop closure was at least 10 keyframes ago it will pose a query to the keyframe database asking for all keyframes which share at least one word with the current keyframe and have a higher score than the minimum of those just calculated. The keyframes returned by the keyframe database are then used as loop closing *candidates*.

For each of these candidates a *candidate group* is created which contains all keyframes which are connected to this candidate. ORB SLAM then checks these groups for *consistency*. A group is considered to be consistent when it overlaps with a candidate group which was created for the last keyframe that was passed to the loop detection module. Overlapping means that two candidate groups share at least one common keyframe. Once a candidate group has achieved a consistency count of more than two, it is passed on to the second part of the loop closing module. A visualization of the procedure is given in figure 3.11.

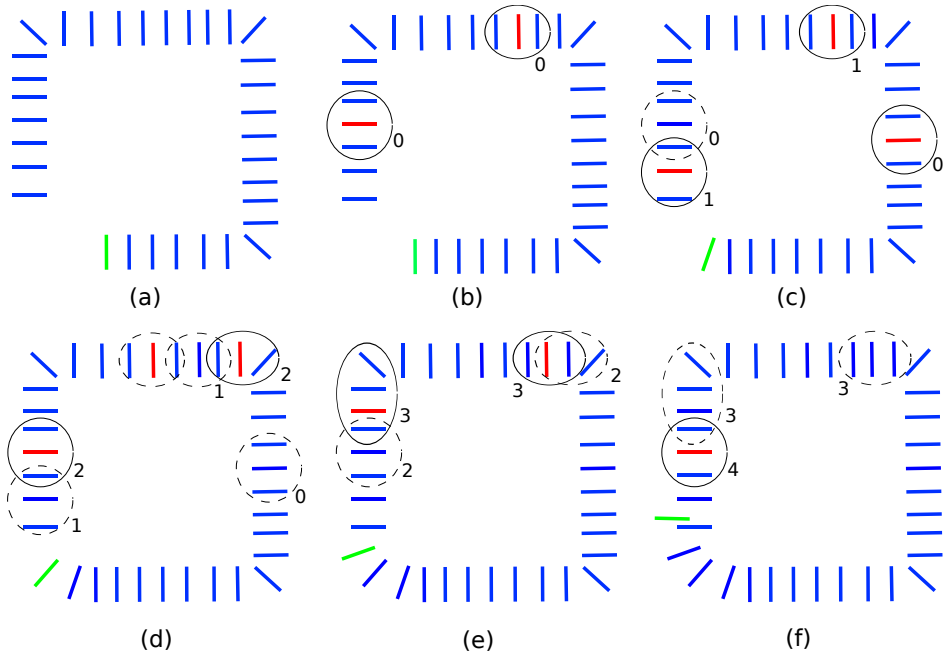


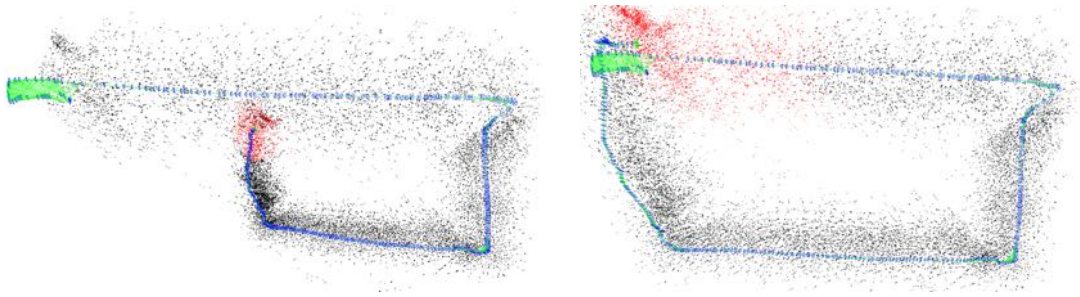
Figure 3.11: All six figures a-f sketch a possible setup of keyframes viewed from above. Blue lines resemble keyframes which are part of the map, the green keyframe is the current keyframe and red keyframes illustrate loop closing candidates. Closed circles represent candidate groups which are going to be saved for the next iteration, the dashed ones are deleted after the current iteration. (a) depicts the initial situation. No keyframes are considered as candidates. (b) shows the creation of two candidate groups and initialization of their consistency counter. In (c) the new group on the left overlaps with the old one and therefore increases its consistency counter. So does the one on top where the candidate keyframe stayed the same. On the right a new candidate group is created. (d) shows that when two new groups overlap the old one (top) only the one tested first goes on to increase its counter. The group on the right is deleted because there was no new group overlapping it. Sketch (e) shows how two groups reach a consistency count of three. At this point they will be used for loop confirmation. (f) illustrates that groups are not deleted once they reach a count of three. If they keep finding overlaps they will be used for confirmation with the next iteration again.

Loop Confirmation

Once a candidate group has accumulated enough consistency its last associated keyframe will be used to evaluate whether it can actually be considered as a loop. In order to check whether the current keyframe and a candidate keyframe correspond, ORB SLAM searches for matches between them in a similar way to what was explained in 3.4 for the case without a motion model. Only that now the algorithm does not look for matches between 2D and 3D points but can match the map points seen in the current keyframe directly to the map points seen in the candidate keyframe by using their saved ORB descriptors.

When successful (at least 20 matches found) ORB SLAM now holds matches between pairs

of map points which correspond to the same 3D point in the real environment. However, internally, due to accumulated drift, these map points will most likely not be very close to each other even though they represent the same real point. Figure 3.12 illustrates this situation.



(a) (b)

Figure 3.12: Result of mapping with ORB SLAM while driving the BlueROV2 in a rectangular path. (a) shows the state of the map created by ORB SLAM just before detecting a loop closure. (b) shows the same map a few frames later than (a), just after a loop has been detected and closed. This figure demonstrates the accumulation of error due to e.g. scale-drift and how loop closing can account for it.

In localization the solution can generally drift with respect to six degrees of freedom. Three rotational degrees of freedom and three translational. For monocular localization however a seventh degree of freedom becomes important as well: scale. Similar to humans that loose their sense of depth when trying to see with only one eye, a SLAM algorithm using a single camera cannot know about the scale of its environment because it doesn't have a reference. Simply put a monocular SLAM algorithm has no idea whether it is looking at life size room or a perfect miniature representation of it. This is why it can only estimate its surrounding up to a factor of scale between internally kept *units* and real distances in meters. Unfortunately this also means that this factor is another degree of freedom which can drift over time, as can be seen in figure 3.12.

For the sake of finding and correcting the drift accumulated over time, ORB SLAM estimates a similarity transformation[14] between the current keyframe and the loop closing candidate keyframe. Such a similarity transformation S can be described as:

$$S = \begin{bmatrix} sR & t \\ 0 & 1 \end{bmatrix} \tag{3.14}$$

$S \in Sim(3)$	Similarity transformation
$s \in \mathbb{R}$	Scale factor
$R \in SO(3)$	Rotation matrix
$t \in \mathbb{R}^3$	Translation vector

[14] A matrix which allows conversion between representations of the same mathematical construct (e.g. a transformation matrix) from one base to another.

$Sim(3)$ and $SO(3)$ are so called *Lie groups*, $SO(3)$ represent the group of rotations in 3D space and $Sim(3)$ the group of similarity transformations in 3D space. Every *Lie group* has a related *Lie algebra* which is a space in which the representation is minimal. What this means is easily understood through the following example:

Most of the time rotations are expressed through rotational matrices which are over-parametrized (need nine parameters to describe three degrees of freedom) or as euler angles which suffer from gimble lock or singularities. In $SO(3)$'s corresponding *Lie algebra* $so(3)$ every rotation can be expressed as a linear combination of three so called generators[31]:

$$G_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}, \; G_2 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix}, \; G_3 = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{3.15}$$

such that any rotation can be expressed as:

$$\omega = [\omega_1, \omega_2, \omega_3]^T \in \mathbb{R}^3 \tag{3.16}$$

$$\omega_1 G_1 + \omega_2 G_2 + \omega_3 G_3 \in so(3) \tag{3.17}$$

The same is true for similarity transformations which are then simply a combination of seven generators. An in-depth explanation of the topic is given in [31].

ORB SLAM computes this similarity transformation using the method of Horn described in [57]. The transformation is checked by using it to see how large the error is when projecting the corresponding map points. Should this check produce more than 20 matches the transformation is accepted.

The algorithm then goes on to doing another guided search for more 3D point to 3D point correspondences with the help of the computed similarity transformation and use the found matches to further optimize the transformation. Should at least 20 matches remain after optimization the candidate is accepted for one last test. At this point ORB SLAM performs another search for more matching map points by also projecting map points found in the loop closure candidate's connected keyframes. If after this at least 40 matches were found overall the candidate is accepted and the loop will be closed and refined.

Loop Refinement

At this point the loop has already been accepted. The last step that is done by ORB SLAM is to refine the loop. In order to implement the loop the local mapping module has to be interrupted so the map cannot change while it is being optimized. Then the connections in the covisibility graph are updated such that the current keyframe is connected to those which are at the location around the candidate keyframe. ORB SLAM then proceeds to collect all keyframes which are connected to the current one and uses the computed similarity transformation to correct the pose of all connected keyframes and their map points. This effectively moves all keyframes and map points closely connected with the

current keyframe to where they are going to be after the loop closure. Afterwards the connections in the covisibility graph are updated.

Even though the pose of keyframes and map points around the current keyframe have now been corrected, the problem of correcting all other keyframes in the loop and removing duplicate map points still needs to be solved. For removing duplicate map points the algorithm projects all map points seen by connected keyframes into its image coordinates and tries to match them with features. If it finds a match for a feature that is already linked to a different map point the map point will be considered a duplicate and removed. Otherwise it will be used as a new observation. These matches are the first step of linking the two loop closure sides. Using these the local connections in the covisibility graph are again updated which now also connects keyframes from both sides.

The next step is to propagate the correction over the essential graph (a subgraph of the covisiblity graph, see 2.2.6). This way the optimization can focus on the most important edges of the pose-graph and will be much quicker than if it would include all edges kept in the covisibility graph. The optimization is run over Sim(3) constraints as detailed in [2].

As a last step a global bundle adjustment is started in a separate thread which will optimize over the all keyframes and map points while at the same time local mapping is rescheduled to be run again. According to [3] this is a very costly operation which also comes with the problem that the optimized output of the BA has to be merged with the current map once it is done. Since the map keeps on changing while the global BA is running the algorithm has to update all those frames which were not yet part of the BA. This is solved by updating those keyframes which were not updated according to the transformation of their parent keyframe in the spanning tree. Not updated map points will be updated according to the transformation of their reference keyframe.

4 BlueROV2

The robot used for this thesis is the BlueROV2. A ROV developed and sold by Blue Robotics. The following section will give an overview about the robot's specifications and why it is an interesting robot for underwater research.

As can be seen in figure 4.1 the BlueROV2 is a relatively small robot of about 0.5 m length, 0.3 m width and 0.2 m height [58].

(a) (b)

Figure 4.1: Blue Robotics' BlueROV2. (a) front view, (b) back view. Figure source: [59]

All electronics are safely installed in two watertight enclosures rated up to 100 m depth. The top enclosure holds the logical components, the bottom one the battery. It is equipped with six Blue Robotics T200 thrusters set up in a configuration as shown in figure 4.2.

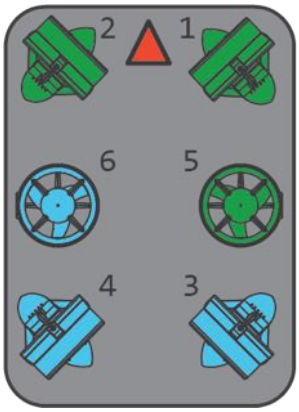


Figure 4.2: Top-view of the BlueROV2's vectored thruster setup. Thrusters 1-4 are used for rotational and forward-backward motion. Thrusters 5 and 6 for moving up and down. The coloring indicates that thrusters 1, 2 and 5 hold the same shape of rotors as do 3, 4 and 6. The red triangle points to the front of the robot. Figure source: [59]

Its thruster setup allows it to move freely within four degrees of freedom. Roll and pitch are not controllable.

For communication purposes the BlueROV2 can be bought with a tether of up to 300 m, the one available at UWA has a 100 m tether. This allows for easy LAN communication even when the robot is in deep water. It also has two LED lights that can be dimmed during use.

Since the robot used at UWA is equipped with the advanced electronics package, it comes with a Raspberry Pi 3 that takes care of transmitting data from the ROV to a connected computer. Furthermore, this package provides a Raspberry Pi camera with wide angle lens. The camera looks straight ahead and can be rotated by about 45° up and down. This is the only sensor the robot has to perceive landmarks in its environment. Other sensor's include a pressure, depth and temperature sensor and an IMU including gyroscope, accelerometer and magnetometer. The IMU is part of the used controller, a Pixhawk Autopilot (px4), which has not only the one but two IMUs integrated. The Pixhawk (figure 4.3) collects all sensor data except for camera images, bundles it and outputs it on its serial port.

Figure 4.3: A 3DR pixhawk as used on the BlueROV2. Pixhawk is an open-hardware project and can be used for control of various devices like AUVs, Rovers or ROVs. Figure source: [60]

This serial port is read by the Raspberry Pi and then sent to a connected client via User Datagram Protocol (UDP). In parallel to the sensor data the Raspberry Pi also sends a camera video stream via UDP. More on the way data is transferred is explained in section 5.1.

For ease of use the ROV provides three different modes:

Manual Mode: The standard mode in which no stabilization is performed.

Stabilize Mode: In this mode the ROV stabilizes roll and holds its heading (yaw), as long as the user is not trying to turn. Depth control has to be done by the user.

Depth Hold Mode: In this mode the robot behaves as in stabilize mode but also keeps its current depth constant. The user can still control to go up and down but otherwise the ROV stays at the same depth.

What makes the BlueROV2 so interesting as a research tool is its relatively low price of about 3000-4000 US $ and its portability. Even with a hundred meters tether it is still possible for a single person to carry the robot and the cable at the same time. Furthermore, thanks to the Pixhawk and Raspberry Pi, it is easily extendible in terms of both hard- and software.

4.1 Modifications

In order to add more information to the recorded datasets another IMU and a GPS module were added to the ROV. As additional IMU a Xsens MTi module was used. For GPS a Parallax GPS module was chosen.

Xsens MTi

The Xsens MTi is an IMU developed and marketed by the Dutch company *Xsens*. It features a 3-Degrees of Freedom (DoF) gyroscope, a 3-DoF accelerometer and a 3-DoF magnetometer. What sets the Xsens IMU apart from standard IMUs is that it runs an on board sensor fusion which promises high accuracy measurements. The sensor works at an update frequency of 100 Hz. It was mounted so the x-axis points towards the front, the y-axis points down and the z-axis to the left of the ROV. For calibration purposes a dataset was recorded in which the ROV is placed on all six sides. Documentation for the IMU can be found at [61].

Figure 4.4: Xsens MTi IMU. Figure source: [61]

Parallax GPS module

The Parallax GPS module is a standard GPS module. It was added to the ROV in the hope that it might be able to read the ROV's position while it is driving at the surface. The conducted experiments quickly showed that the measurements are very inconsistent and completely drop out as soon as the robot is barely beneath the water's surface. For this reason the GPS readings were not used in the evaluation. They are however still part of the recorded data.

5 Evaluation

The following chapter describes the evaluation of ORB SLAM on the BlueROV2. As a first step section 5.1 outlines how the data flow was modified for data collection. This is followed by the description and evaluation of the performed experiments in 5.2. At last 5.3 summarizes the collected results.

5.1 Data Acquisition

There are only two kinds of data the BlueROV2 provides:

1. Camera images and
2. IMU/barometer data.

The way the data is transferred from the BlueROV2 in its original state is illustrated in figure 5.1.

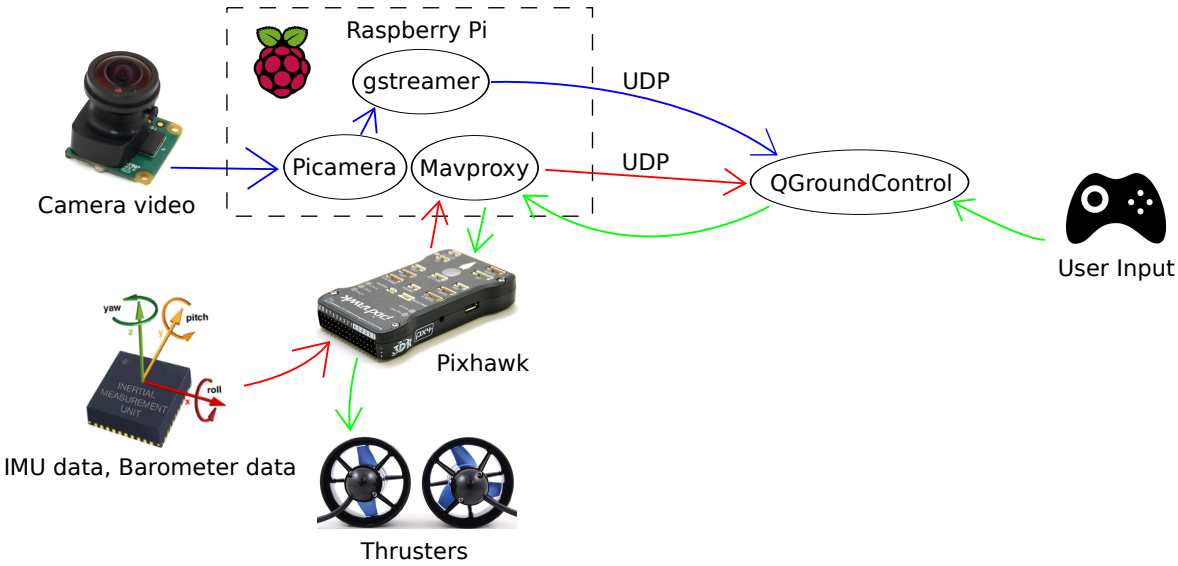


Figure 5.1: A sketch of the original data flow of the BlueROV2. Blue arrows represents camera images, red arrows are IMU/barometer data and green arrows symbolize user input.

Camera images are fetched from the camera by the *raspivid* process running on the Raspberry Pi. These images are fed into *gstreamer* which sends them to the network via UDP. IMU and barometer are both connected to the Pixhawk which uses it for internal calculations and converts it to MAVlink protocol[15]. A process called *mavproxy* then reads

[15] Micro Aerial Vehicle protocol is a communications protocol used mainly for drones. For the BlueROV2 all system information is transmitted using this protocol. An example of how this protocol looks can be viewed in A.15.

these MAVlink messages from the Pixhawk's serial port and sends them via UDP to the LAN.

All UDP streams are received by a ground control station program (e.g. QGroundControl or Mission planner) which shows the video stream to the user and feeds its interface elements with the received MAVlink messages to show information like depth and current heading.

Unfortunately there was no way to easily record both the video and the MAVlink messages. In addition the video sometimes showed significant lags and stutter which was not acceptable for data recording.

To make sure that data was always sent synchronized and to have a configurable and transparent way of sending, receiving and recording the data flow was changed as depicted in figure 5.2.

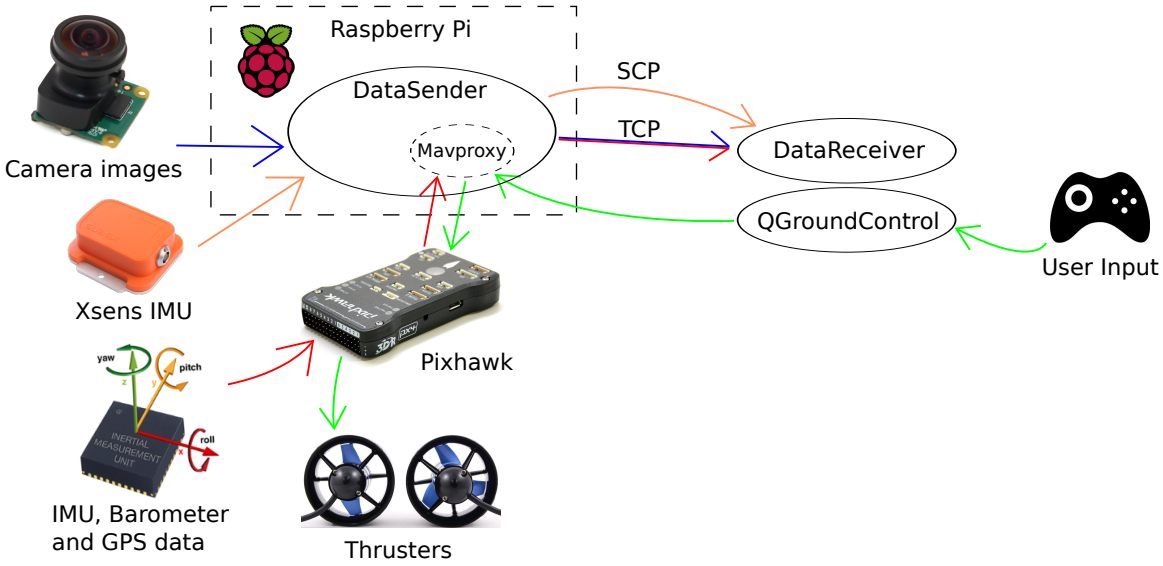


Figure 5.2: A sketch of the modified data flow.

Note that the camera now outputs single images instead of a video. This allows for a better synchronization between camera and MAVlink messages. The data of the newly connected GPS module is integrated into the MAVlink messages that are output by the Pixhawk. Synchronization happens in the *DataSender* process. This is a python program which launches an individual thread for:

1. polling images from the camera,
2. a modified *mavproxy* subprocess which collects MAVlink messages from and passes user input to the Pixhawk and
3. recording the Xsens IMU output to a file stored on the Raspberry Pi.

The program will always wait for a new image to arrive from the camera. While it is waiting it keeps collecting all incoming MAVlink messages. Once an image arrives it forms a data package containing:

- the size of the new image,
- the new image,
- all MAVlink messages currently stored and
- a timestamp

This data package is then sent via Transmission Control Protocol (TCP) to the connected computer. There it is received by the *DataReceiver* process which displays the image to the user and writes all data to a previously specified folder on the hard drive. The content of the received data packages is written to JavaScript Object Notation (JSON) files. An excerpt of such a file is given in A.15.

Due to the Xsens IMU having a high update frequency of 100 Hz its data is written to a separate file on the Raspberry Pi. This is done to avoid slowing down the TCP connection. Instead the file is copied via Secure Copy Protocol (SCP) at the very end of the recoding. Unfortunately this means that the Xsens IMU data is not synchronized with every frame like the MAVlink data. Instead the data contains a timestamp of the recording that can be used for later synchronization.

5.2 Experiments

Evaluating ORB SLAM in the underwater scenario proofed to suffer from a major issue: It is very hard to create ground truth.

On land ground truth can be easily derived from GPS. While recording the experiments it quickly became clear that the GPS module added to the ROV was not able to collect any usable data. Once the ROV moved only a little under water's surface the readings immediately stopped. There are technical solutions to this problem like acoustic location systems but these are very expensive and were not available for this thesis. In order to be able to provide ground-truth for at least some of the recorded datasets physical references were used. These include e.g. driving on the edges of a pool with known dimensions or driving along a stretch of jetty which could be easily measured. But even if there was a physical reference, waves, surge and inaccurate robot controls made it difficult to follow them precisely.

In light of this issue the evaluation performed here cannot be based on calculating an error between ground truth and estimated trajectory. Instead it needs to be evaluated with regard to other aspects. Available options include:

1. **Frames needed for initialization:** Counting the frames necessary for achieving successful initialization could be used as an indicator for how the surroundings influence the performance of the algorithm. The problem with this criterion is that it does not take into account whether the robot moved at all during the time and therefore whether initialization was possible at all.

2. **Frames successfully tracked:** Although the amount of frames successfully tracked does not contain any information about the quality of the tracking it is a condition for providing any results at all. Without any tracking ORB SLAM will not produce any result. When defined as a percentage of the overall frame count in a dataset it can be a good indicator for the difficulties encountered within the dataset.

3. **Times tracking is lost:** Since a loss of tracking generally happens due to outer influences ORB SLAM cannot handle well, this criterion can add information not available when simply counting the amount of images tracked. For example in a case where tracking is lost a lot of times but the algorithm can relocalize immediately this criterion could help to visualize this situation.

4. **Amount of loops found:** Even though in most cases loop closures help significantly improving the overall outcome of the localization, in some cases where they are falsely detected they can be very destructive. This is also visible in some of the presented experiments. Since this is not representable by counting the amount of loops this is a poor evaluation criterion.

5. **Qualitative validation:** In spite of not being able to measure the accuracy of an estimation in numbers, a simple qualitative assessment of how well a trajectory was estimated can add a lot value to an evaluation as already shown in 2.4.1.

For this thesis it was decided to use a combination of the percentage of frames tracked, the times tracking is lost and a qualitative validation for evaluation. The qualitative validation splits results into three categories as follows:

Very good: Both the estimated trajectory and the map closely represent the robot's movement and the actual environment without any noticeable deviation.

OK: Estimated trajectory and map still very much represent the robot's movement and environment but they might have minor notable inaccuracies.

Poor: This will be chosen for major unresolved inaccuracies in either map and/or trajectory and in cases of falsely detected loops or false relocalization.

Due to heavy use of RANSAC and multi-threading ORB SLAM is a highly non-deterministic algorithm. To account for this, every experiment was run and evaluated ten times. The final result is represented in the following way:

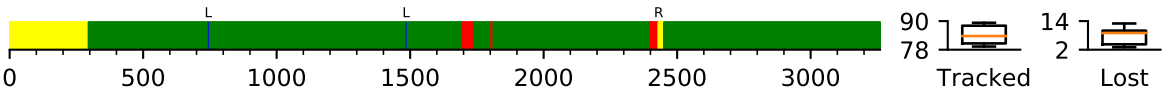


Figure 5.3: The bar on the left represents ORB SLAM's states plotted over the frames of an experiment. For this plot the most representable of all ten tests is picked. Yellow areas signify initialization, green stands for successful tracking and red signals relocalization. Additionally, found loops are marked by a blue bar and an L. Manually triggered resets are marked by an R. The box plots on the right visualize the distributions of tracked frames in percent and times tracking was lost per thousand images over all ten tests.

For some of the datasets the robot moved on a trajectory which did not revisit the same places. In these cases a severe problem with ORB SLAM is that when tracking fails the algorithm will forever try to relocalize. In oder to see how the algorithm performs on the rest of the data, the algorithm is rerun for cases where there is a single relocalization spanning over more than 50% of the frames. Only this time a manual reset is triggered after not relocalizing for more than 30 frames.

Another problem that was encountered was that the robot always saw its own parts in the images created during recording, as can be seen in figure 6.1 (a). This caused ORB SLAM to extract and track features in those image areas and lead to frequent tracking failure. In order to be able to properly evaluate ORB SLAM the solution described in 6.1.1 was used for all experiments.

The main goal while recording datasets was to cover as many different scenarios as possible so that the ORB SLAM algorithm could be tested with varying conditions. For that reason the recorded scenarios include:

- Man-made structures like pool, jetties, pipes and boats

- Natural environments like reefs, river beds or sea floor
- Night and day settings
- Varying depths
- Sunshine and cloudy weather

In total 46 datasets were recorded in nine different locations. Every dataset follows a given directory structure. An example of this structure looks like this:

```
River_at_UWA
├── 2017-09-06
│   └── Sideways_along_jetty
│       ├── data_2017-09-06 17:03:07.347200.json
│       ├── Image000001.jpg
│       ├── ...
│       ├── Image004248.jpg
│       ├── rgb.txt
│       └── XsensLog_1504520730640
├── description.txt
└── ORB_parameters.yaml
```

The root directory describes the location. On the second level datasets are grouped by their recording date. This way it is made sure that datasets with similar environmental conditions are grouped together. Every dataset consists of a JSON file (example given in A.15), all recorded images, a *rgb.txt* and the Xsens IMU's log. *rgb.txt* is a file containing timestamp-image pairs which is the way ORB SLAM expects its input. Furthermore there is a *description.txt* which aims to give a short description on all included datasets. The last file is the *ORB_parameters.yaml*. This file contains all parameters needed for running ORB SLAM on a particular dataset.

If not explicitly explained all ORB SLAM parameters were left at their default values. An example file can be viewed at A.16. *ORB_parameters.yaml* also contains the camera calibration parameters used. For this reason if camera parameters are specific to a certain date or even dataset the files may be found at a deeper level in the tree. Camera calibration was performed using a camera calibration package [62] provided by the Robot Operating System (ROS).

The following sections will describe and evaluate the results of the experiments by location. For reasons of simplicity not every single dataset is described but rather those which provide the most information. For a quick and short overview of the experiments' results please refer to the summary (5.3) at the end of this chapter.

5.2.1 Laboratory

Table 2: Laboratory experiment details

#	Name	Data	Length	Ground-truth
1	Around_table	3260 images	~3 min 23 s	-

(a) (b)

Figure 5.4: (a) Sample image. (b) ORB SLAM result viewed from above.

Location:
UWA robotics laboratory

Description:
This dataset was created for testing how well ORB SLAM performs on the ROV's hardware. For recording the robot was placed on an office chair and then rolled around a table which stands in the middle of the room. The dataset also includes a couple of very quick motions which were supposed to cause ORB SLAM's tracking to fail. Furthermore there are two situations where a moving person is in the frames.

Evaluation:

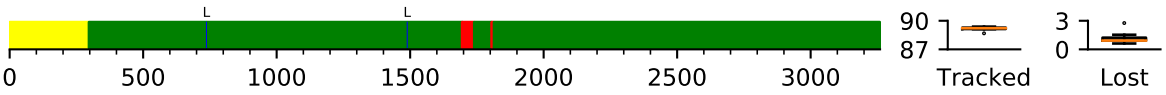


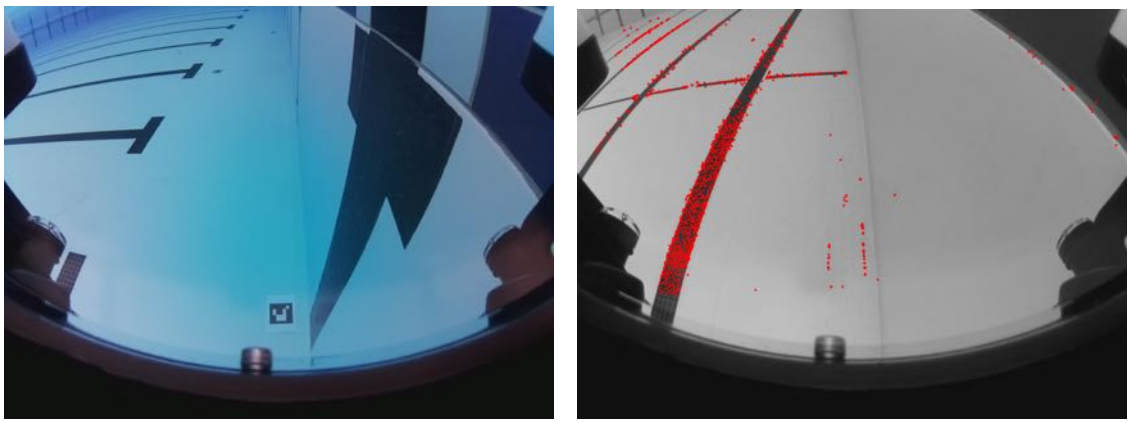
Figure 5.5: Laboratory experiment results. Quality: very good.

As figure 5.5 shows ORB SLAM is able to track the camera movement almost all of the time. Initialization seems to take a while but in fact for the first few hundred frames in this dataset there simply is no movement. As soon as the camera moves ORB SLAM initializes and starts tracking. There are two short episodes where tracking is always lost. These are the mentioned quick motions which were supposed to provoke just that. Other times when tracking is lost stem from problems with the recording where a few pictures were not transmitted. The estimated trajectory shown in figure 5.4 (b) resembles the actual taken path very closely. Furthermore ORB SLAM was able to detect two loops in every test and was not affected by the person moving through the image. As can be seen on the right of figure 5.5 the results are very stable over all ten plots with just a single outlier that had trouble with tracking after a relocalization. These results clearly show that ORB SLAM can perform very well on the ROV's hardware. At least in a scenario which is not underwater.

5.2.2 Pool

Table 3: Pool experiment details

#	Name	Data	Length	Ground-truth
2	No_markers	15374 images, IMU: Pixhawk	∼16 min 01 s	Yes
3	Markers	20166 images, IMU: Pixhawk	∼21 min 31 s	Yes



(a) (b)

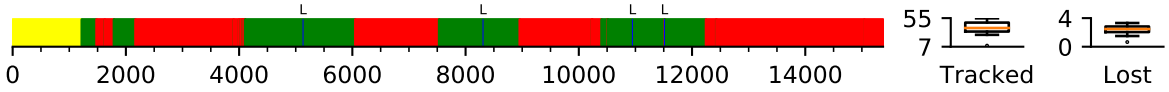
Figure 5.6: Pool dataset: (a) sample image. (b) example of poor feature distribution.

Location:
UWA pool

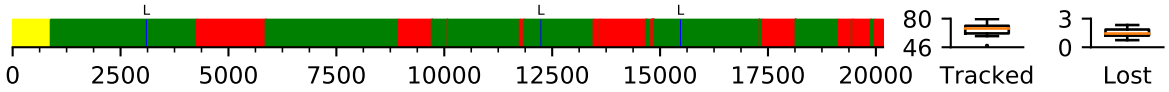
Description:

This experiment was performed at the UWA pool. The goal of it was to use the edges of the pool as ground truth. The pool has a rectangular shape with dimensions 33.5 m by 25 m. The 33.5 m is the length of the first edge the ROV is driving along. Inside the pool there are black lines made of tiles on both wall and floor. Since it was suspected that ORB SLAM might have trouble working in this very symmetric and feature scarce environment one experiment was done with 20 unique printed markers placed on the pool's floor. One of these markers can be seen in figure 5.6 (a). In order to give the algorithm enough chances for loop closures the ROV was driven along the pool's walls in both datasets whilst always staying on the surface.

Evaluation:



(a) Ex. 2: No_markers. Quality: poor

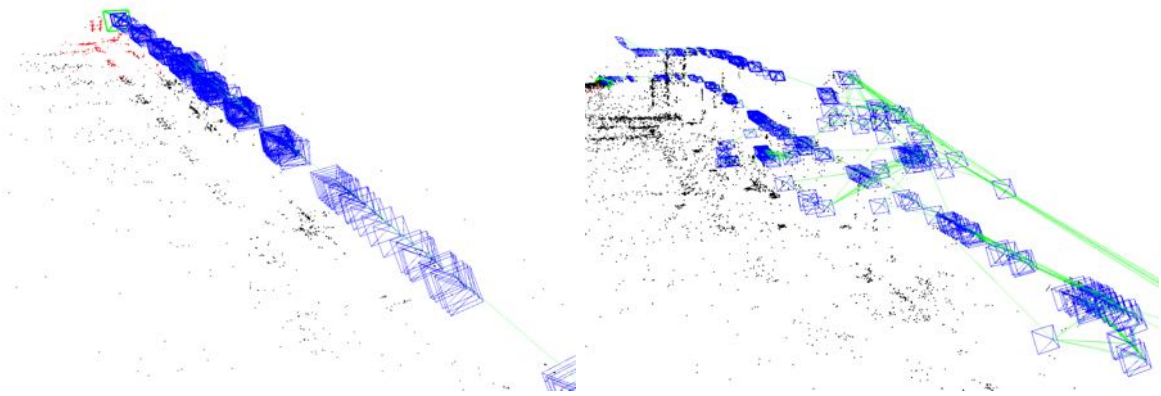


(b) Ex. 3: Markers. Quality: poor

Figure 5.7: Pool experiment results.

The pool proved to be a very difficult environment for ORB SLAM to work in. Figure 5.7 (a) illustrates several of the problems encountered. The periodical tracking loss and regain shows that while the robot is driving along the wall where it is moving orthogonal to the black lines on the pool floor, as shown in figure 5.6 (a), it can mostly maintain tracking. While driving along the other wall where the ROV follows the black line on the floor, as in 5.6 (b), tracking is lost very quickly. Tracking is most likely to be lost because of the poor feature distribution shown in 5.6 (b). Since there are lots of features on the black line which barely differ and do not change much over timeORB SLAM fails to properly associate the seen features between frames. The results for dataset 3 proved to be just as unusable as those of dataset 2. The markers help to reduce the times tracking is lost but the overall results are still poor.

The resulting trajectory for the dataset without markers is shown in figure 5.8 (a). It is obvious that ORB SLAM's estimated trajectory is only a single line even though it was tracking on two sides of the pool. What happens is that due to the symmetry of the pool ORB SLAM relocalizes within the map created along the first wall even though it is on the exact opposite side of the pool. The problem with the repetitive environment is further demonstrated by the fact that the first loop is found before a full path around the pool was driven. Due to relocalization and loops detected ORB SLAM simply keeps jumping back and forth within the map shown in 5.8 (a).



(a) (b)

Figure 5.8: ORB SLAM results for pool experiment. (a) without, (b) with markers.

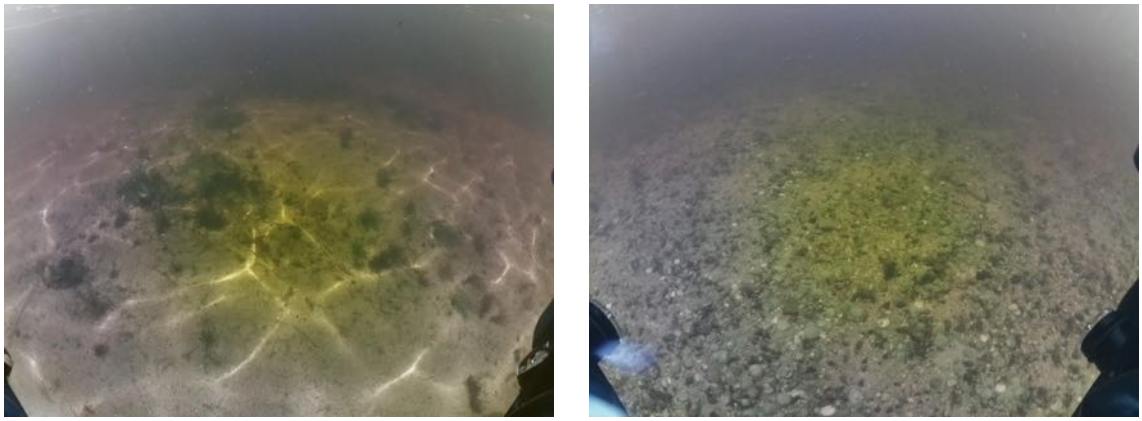
5.2.3 Point Walter

Table 4: Details for experiments at Point Walter.

#	Date	Name	Data	Length	Ground-truth
4	08.09.17	Along_jetty	3986 images IMU: Xsens, Pixhawk GPS	∼4 min 24 s	Yes
5	08.09.17	Circle	2377 images IMU: Xsens, Pixhawk GPS	∼2 min 32 s	No
6	08.09.17	Rectangle	2513 images IMU: Xsens, Pixhawk GPS	∼2 min 42 s	Partially
7	14.09.17	Circle	2570 images IMU: Xsens, Pixhawk	∼2 min 42 s	No
8	14.09.17	Figure_eight	3012 images IMU: Xsens, Pixhawk	∼3 min 21 s	No
9	14.09.17	Rectangle	3821 images IMU: Xsens, Pixhawk	∼4 min 12 s	Partially

Location:
In Swan River at Point Walter, Bicton, Perth

Description:
For this experiment data was recorded on two different days. The first three datasets were taken on a sunny day whereas the other three were recorded in cloudy weather. Aim of



(a) Sunshine (b) Cloudy weather

Figure 5.9: Point walter experiment: sample images.

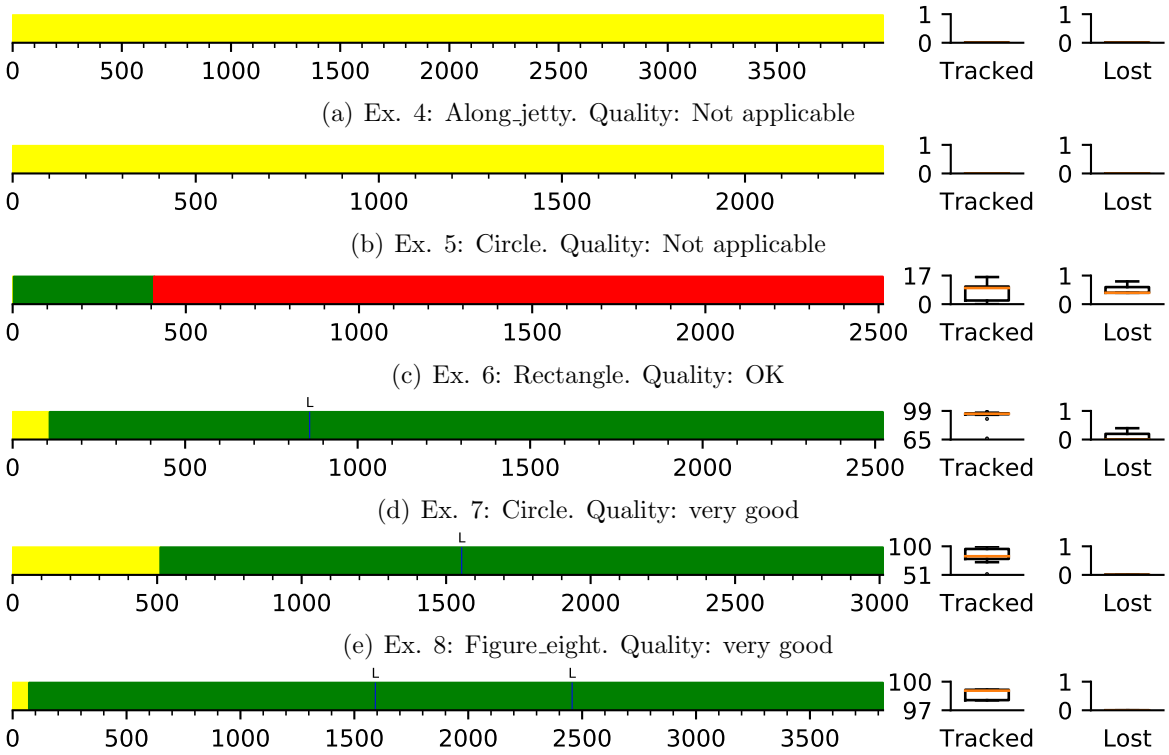
this experiment was to see the difference lighting makes on ORB SLAM's results. In both cases the ROV was driven in different figures like a circle or an eight which makes it easier to determine if ORB SLAM is estimating the correct trajectory. Three of these datasets include ground truth. For dataset 4 the ROV was driven back and forth a 19.5 m long stretch of jetty. In datasets 6 and 9 the first side of the driven rectangle was measured. For dataset 6 it was 11.2 m long and for dataset 9 it was 6 m long.

The surrounding at Point Walter is a sandy sea floor with a few rocks and some algae. In some images the foundations of the jetty from which the ROV was launched are visible as well. The difference between the datasets taken on the two different days can be seen in figure 5.9. It is obvious that (a) was taken on a sunny day as there is a lot of light ripples on the sea floor. In (b) on the other hand these ripples are not present.

Evaluation:

Looking at the plots in figure 5.10 it is evident that ORB SLAM struggles when faced with the lighting conditions on a sunny day. Due to the ripples visible in figure 5.9 (a) ORB SLAM cannot initialize and has no chance of tracking the robot's movement. In fact for dataset 3 it was only able to initialize because at the beginning of the recording there is a cloud in front of the sun, changing the lighting conditions for a short time. As soon as the cloud is gone and the ripples are seen again ORB SLAM looses tracking and can neither relocalize nor reinitialize when the algorithm is reset.

In contrast, tracking worked very well for datasets 3-6. As presented in figure 5.11 ORB SLAM was able to recover the driven trajectory accurately. The fact that the shapes are not perfect does not stem from ORB SLAM not tracking the driven trajectory correctly but rather from not being able to drive the same trajectory flawlessly multiple times. An interesting observation that can be made from the results on these datasets is that ORB SLAM detected very little loop closures. This does not mean that the algorithm fails at detecting possible loop closures but rather that it is able to recognize already mapped places, even in an environment as monotonous as the one at Point Walter.



(a) Ex. 4: Along_jetty. Quality: Not applicable

(b) Ex. 5: Circle. Quality: Not applicable

(c) Ex. 6: Rectangle. Quality: OK

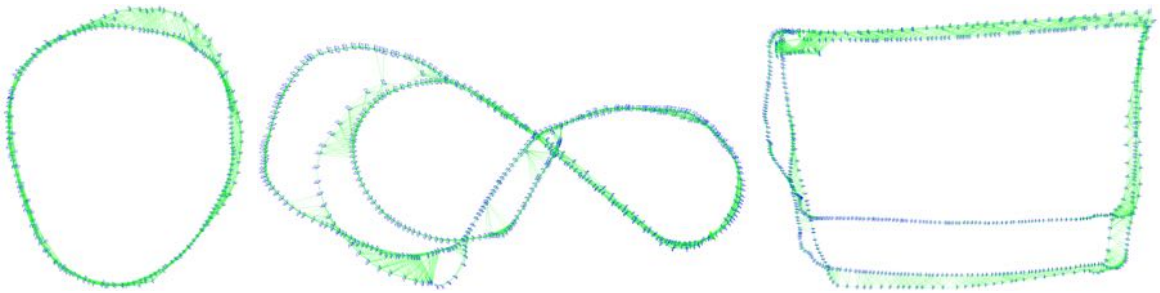
(d) Ex. 7: Circle. Quality: very good

(e) Ex. 8: Figure_eight. Quality: very good

(f) Ex. 9: Rectangle. Quality: very good

Figure 5.10: Point Walter experiment results.

(a) Ex. 6: Rectangle



(b) Ex. 7: Circle

(c) Ex. 8: Figure_eight

(d) Ex. 9: Rectangle

Figure 5.11: ORB SLAM trajectory estimates for Point Walter experiments.

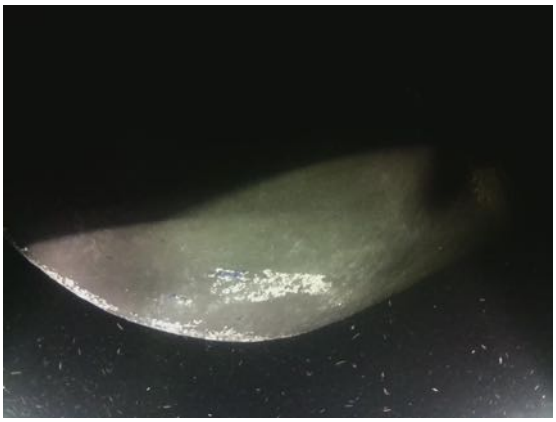
5.2.4 Fremantle Marina

Table 5: Fremantle marina experiment details

#	Date	Name	Data	Length	Depth	GT
10	10.09.17	Exploring _slope	7333 images IMU: Xsens, Pixhawk GPS	∼8 min 04 s	0-3 m	No
11	14.09.17	Along_rocks_ and_objects	11876 images IMU: Xsens, Pixhawk	∼13 min 24 s	0-2 m	No
12	14.09.17	Hull	8255 images IMU: Xsens, Pixhawk	∼13 min 24 s	0-0.6 m	No

(a) Dataset 10 at ground level

(b) Objects visible in dataset 11



(c) Boat hull at night, dataset 12

Figure 5.12: Fremantle Marina experiment sample images.

Location:
Marina in Fremantle, Perth

Description:
The area where these datasets were recorded in is part of the Fremantle Marina. It is right at the sea but surrounded by stone walls so there are barely any waves. The point where the robot was put in the water is characterized by a steep slope made of loose stones which leads down to a flat sandy ground with a few plants and some man made objects like pipes and a chair.

So far all described experiments had been performed at very shallow areas with the ROV at the water's surface or just beneath it. The goal of experiment 10 was to test how ORB SLAM would work when varying the depth and moving to deeper areas. The slope at this spot allowed to follow it to a depth of about 3 m. For this experiment the ROV was moved around this area in a wide circle as shown in figure 5.13 while moving up and down the slope.

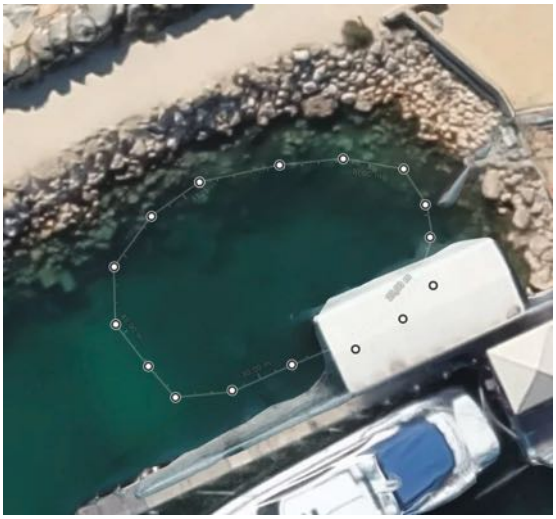
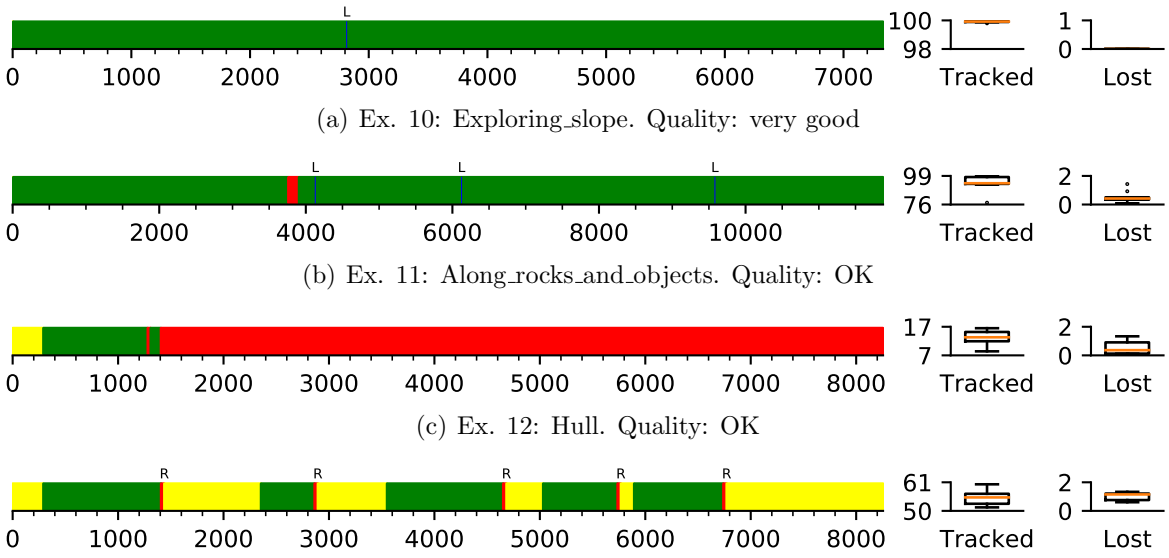


Figure 5.13: Aerial view of the area dataset 10 was recorded in. Source: Google Maps

The second and third dataset were recorded at nighttime. Originally the idea for the second dataset was to record it in the exact same spot as the first one to see and compare how the algorithm performs at night using the robot's lights. Since that spot had a lot of oil and dirt floating around in the water it was decided to move a few meters further and create a dataset in a bit of a different area. In this area there are a lot of objects like a shopping cart, a tyre and two large pipes. The ROV was driven back and forth along the slope so it could observe the objects from different viewing angles.

For the third dataset the ROV was driven along the side of the hull of a boat anchored in the marina. This was done to see whether ORB SLAM is able to deal with the ROV's camera facing upwards and no other visible surroundings but the submerged part of a boat as shown in figure 5.13 (c).

Evaluation:



(a) Ex. 10: Exploring_slope. Quality: very good

(b) Ex. 11: Along_rocks_and_objects. Quality: OK

(c) Ex. 12: Hull. Quality: OK

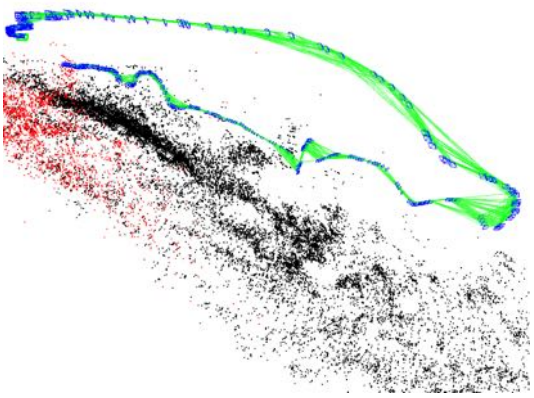
(d) Ex. 12: Hull with resetting. Quality: OK

Figure 5.14: Fremantle Marina experiment results.

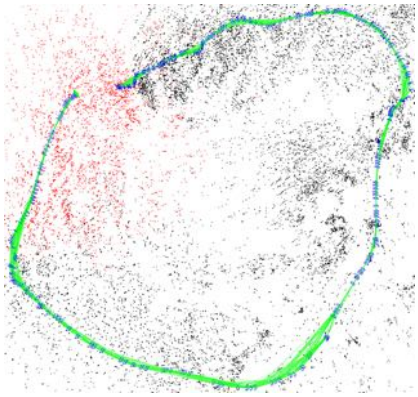
As is visible in figure 5.14 (a) tracking works perfectly for the first data set. ORB SLAM is able to track every single frame and never gets lost in any of the 10 test runs. Figure 5.15 (a) shows that the algorithm successfully tracked the ROV moving up and down the slope. (b) shows the estimated trajectory from above a few frames before the algorithm detects a loop. As can be seen there it is able to track the ROV's movement back to the starting point without much error and it is able to do this over a long path with varying depth.

Experiment 2 demonstrates that ORB SLAM is also able to track the robot's movement at night. But even though it was possible to track the ROV for almost all frames there is a major flaw within its result. After the ROV has reached the point furthest away from the start and turns around ORB SLAM does not reuse the map already created but rather creates a complete second mapping of the same environment. This effect is visible in figure 5.15 (c) where there are two representations of the pipe shown in figure 5.12 (b) in the created map. As soon as the first loop is closed the drift is correct as shown in 5.8 (d). ORB SLAM does however not remove the duplicate points but uses one part of the map for going in one direction and the other for the other direction.

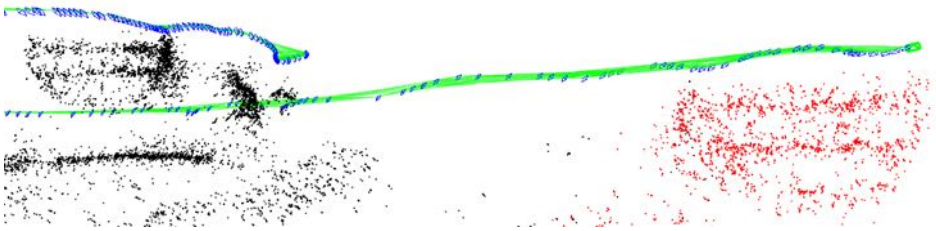
With regard to the third experiment ORB SLAM looses tracking very quickly. This happens because the lights on the BlueROV2 cannot be turned up far enough which leads to only small parts of the hull being visible as shown in figure 5.12 (c). Because ORB SLAM is not able to relocalize for the remainder of the experiment, it was repeated with automatic resetting after 30 frames of failed relocalization. As is visible by the difference between figure 5.14 (c) and (d) it misses out on a lot of tracking opportunities by permanently trying to relocalize. Another problem that was apparent during this experiment is that when the boat moves ORB SLAM will mirror this movement onto the ROV.



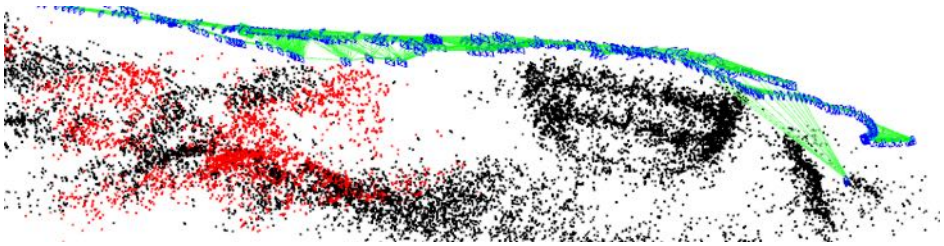
(a) Ex. 10: Trajectory from side



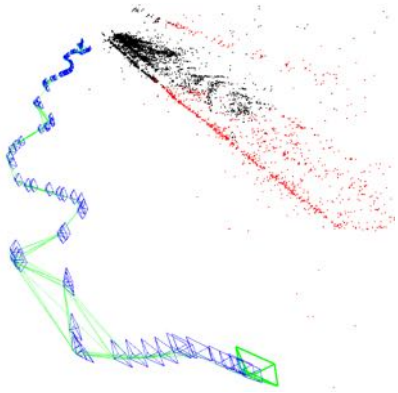
(b) Ex. 10: Trajectory from top, a few frames before loop closure



(c) Ex. 11: Twice mapped pipe



(d) Ex. 11: Merged pipes after loop



(e) Ex. 12: Part of mapped hull

Figure 5.15: ORB SLAM results for Fremantle Marina experiments.

5.2.5 Omeo Wreck

Table 6: Omeo wreck experiment details

#	Date	Name	Data	Length	Depth	GT
13	10.09.17	Along_wreck	9156 images IMU: Xsens, Pixhawk GPS	∼9 min 46 s	0-2 m	No

(a) Structured area (b) Unstructured area

Figure 5.16: Omeo Wreck experiment sample images.

Location:

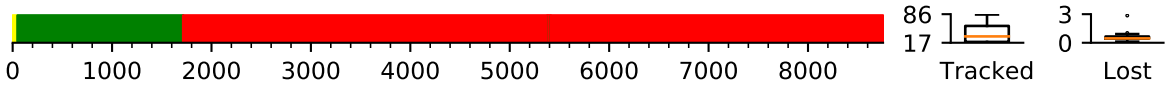
Omeo Wreck, Coogee Beach, Perth

Description:

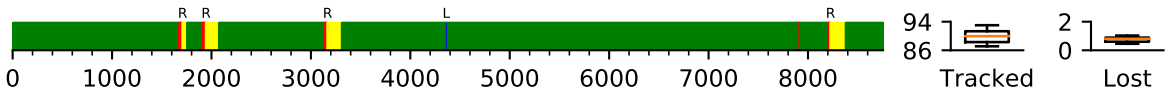
The Omeo was a trading ship that sunk in 1905 close to the shores of Perth [63]. Since its wreck lies only about 20 meters form shore it is very easy to access. For this experiment the ROV was driven up and down the length of the wreck. In order to avoid getting the tether stuck on some part of the wreck this was only done on the side facing the shore. The footage contains a mix of structured and unstructured surroundings as the wreck lies on sandy ground (see figure 5.16 (a) and (b)).

Evaluation:

As is visible in figure 5.17 (a) ORB SLAM struggles to consistently track the robot's movements. This is mostly due to the described mix of unstructured and structured areas present in the dataset. While the ROV moves around structured areas tracking works well and the estimated trajectory closely follows the robot's motion. When moving over

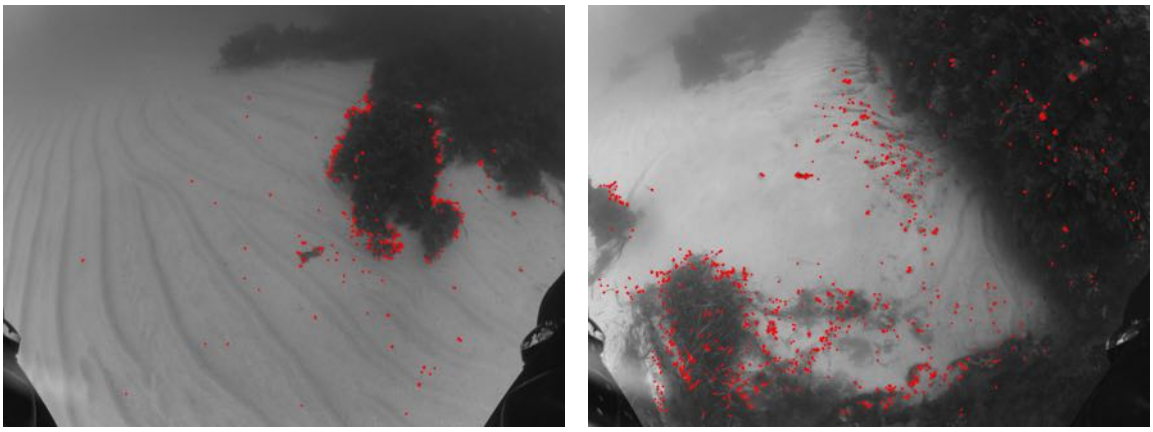


(a) Ex. 13: Along_wreck. Quality: OK



(b) Ex. 13: Along_Wreck with resetting. Quality: OK

Figure 5.17: Omeo wreck experiment results.



(a) (b)

Figure 5.18: Images showing problematic feature detection on areas consisting mainly of sand. While ORB SLAM can find a lot of features in structured areas it is barely able to find any on sandy ground.

unstructured areas tracking quickly fails because it cannot find enough features. This problem is demonstrated in figure 5.18.

Even though the data offers a lot of opportunities for loop closing and relocalization ORB SLAM is not able to pick up on most of them. For some this most likely happens because the change in viewing angle is too large but in other cases the algorithm does not pick up on the possibility although the viewing angle is similar. This circumstance leads to tracking mostly being lost early on with only a few test being able to track over a longer time.

When resetting the algorithm after 30 frames instead of waiting for relocalization the overall tracked frames can be increased a lot as can be seen in figure 5.17 (b). This also allows for the detection of loop closures which highly improves the estimated trajectory. The result is shown in figure 5.19.

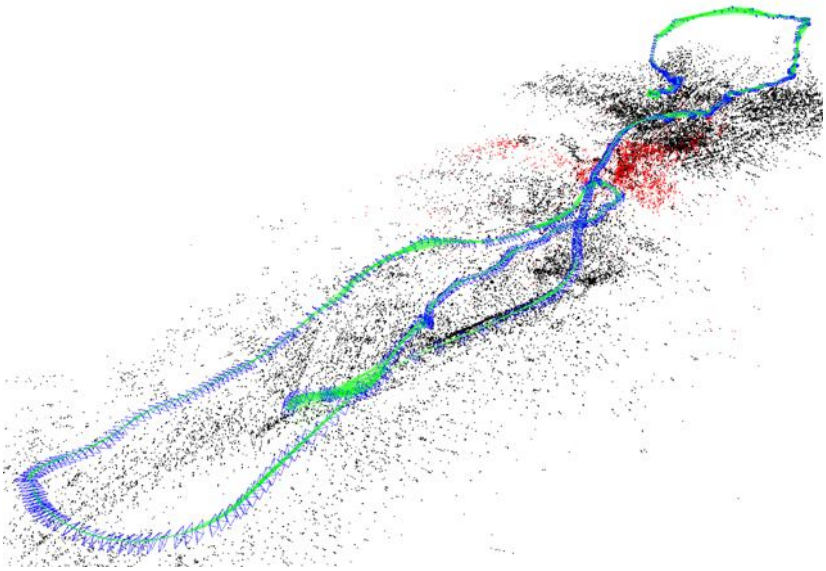


Figure 5.19: ORB SLAM results for Omeo Wreck experiment.

5.2.6 Boat

Table 7: Boat experiment details

#	Date	Name	Data	Length	Depth	GT
14	15.09.17	Reef	9806 images IMU: Xsens, Pixhawk	∼10 min 49 s	1 - 5.5 m	No
15	15.09.17	Poles	5380 images IMU: Xsens, Pixhawk	∼6 min 49 s	0 - 5.5 m	No

(a) Ex. 14: Reef

(b) Ex. 15: Poles

Figure 5.20: Boat experiment sample images.

Location:
Different locations in the sea west of Fremantle.

Description:
For the final experiment the robot was launched from a boat directly into the sea. The goal of this experiment was to test how ORB SLAM would react to all the conditions that come with working in the open sea. These datasets include heavy movements due to strong surge, lots of marine life, long algae which moves with the water and, since it was recorded on a sunny day, very dynamic lighting.

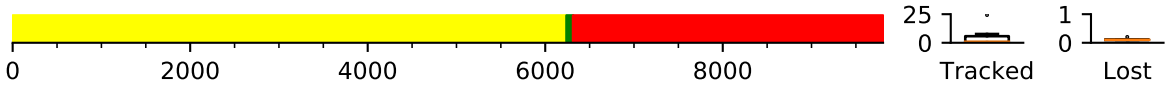
Dataset 14 incorporates many of these problematic conditions. In this dataset the ROV starts at a depth of about 1 m and slowly moves down to 5.5 meters while moving along large rocks covered in algae and coral. Since the robot starts close to the surface there are rays of lights moving through the water for the first half of the dataset. Furthermore there was a lot of wild live at this spot with schools of fish circling the robot as can be seen in figure 5.20 (a). Since there was a strong surge the algae on the rocks a lot.

In dataset 15 the robot starts at the ground at about 5.5 meters in an area with three wooden poles covered in coral which stretch from the bottom of the sea to above surface level. Here the ROV moves along a piece of wood which lies on the sea floor to where the three poles are and then moves along the poles standing upright. After this the ROV follows a pole all the way from the floor to the surface and back down before it leaves the scene along the piece of wood where it began. This area also contains quite a bit of movement since there was heavy surge, fish and moving algae.

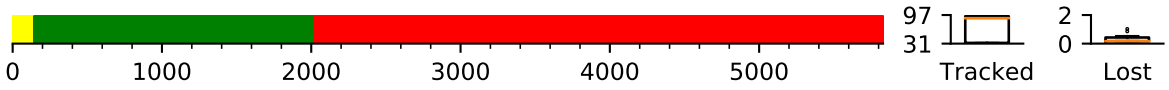
Evaluation:
Figure 5.21 (a) demonstrates that ORB SLAM cannot handle the highly dynamic environment the dataset was recorded in. Most tests completely failed to track even a single frame. What is interesting about this dataset is that those tests where initialization worked it always did so when the ROV reached about 3-4 m depth where the problems caused by the lighting are not as strong anymore. For those short stretches where tracking worked the estimated trajectory actually followed the ROV's movement closely.

The observation that the problematic effects of sunlight weaken with depth is further reinforced by dataset 15. In this dataset the robot starts at a depth of about 5.5 m and shows no trouble with initializing, even though the data was recorded on the same day with only a short boat ride between them. Overall the results produced on this dataset are much better in comparison to the reef dataset. As can be seen in figure 5.21 (b) more than 50% of the tests are able to achieve almost 97 % successfully tracked frames. There is however also a significant amount of tests where tracking is lost due to a quick movement around frame 2000. As was also the case for datasets 12 and 13 the overall tracked frames and therefore the overall gathered information can be increased by resetting the algorithm when it tries to relocalize for too long. Due to its three-dimensionality this is a dataset for which it is very easy to follow whether ORB SLAM's estimation is accurate or not. The

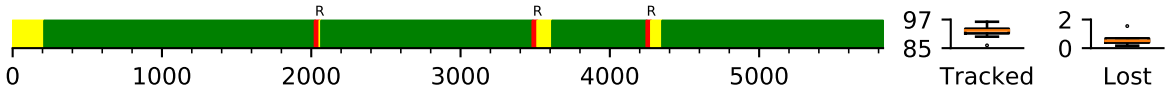
results are presented in figure 5.22. As can be seen there the algorithm is able to create an accurate estimation of the three vertical poles and how the robot moved around them.



(a) Ex. 14: Reef. Quality: OK

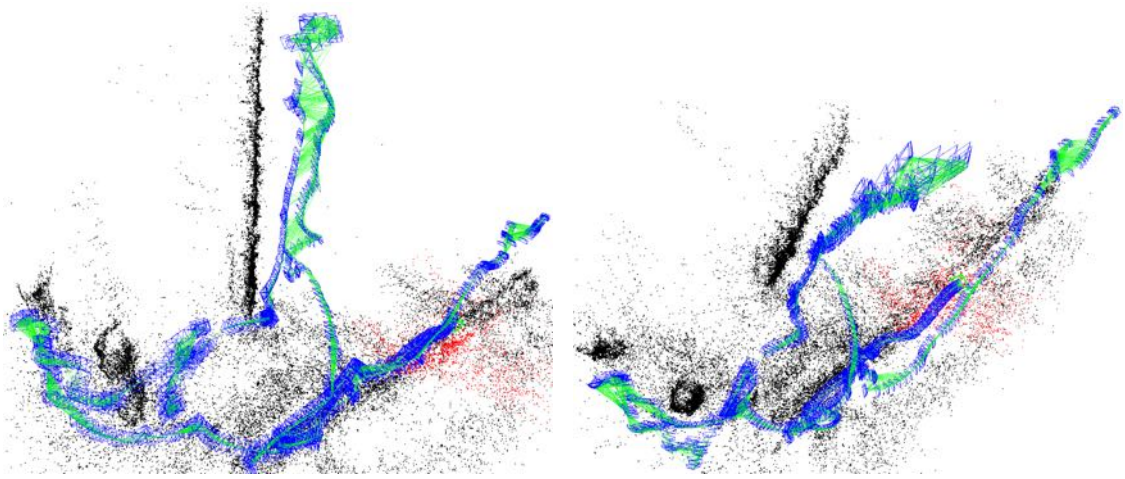


(b) Ex. 15: Poles. Quality: very good



(c) Ex. 15: Poles with resetting. Quality: very good

Figure 5.21: Boat experiment results.



(a) Trajectory from side

(b) Trajectory from above

Figure 5.22: ORB SLAM results for Boat experiment.

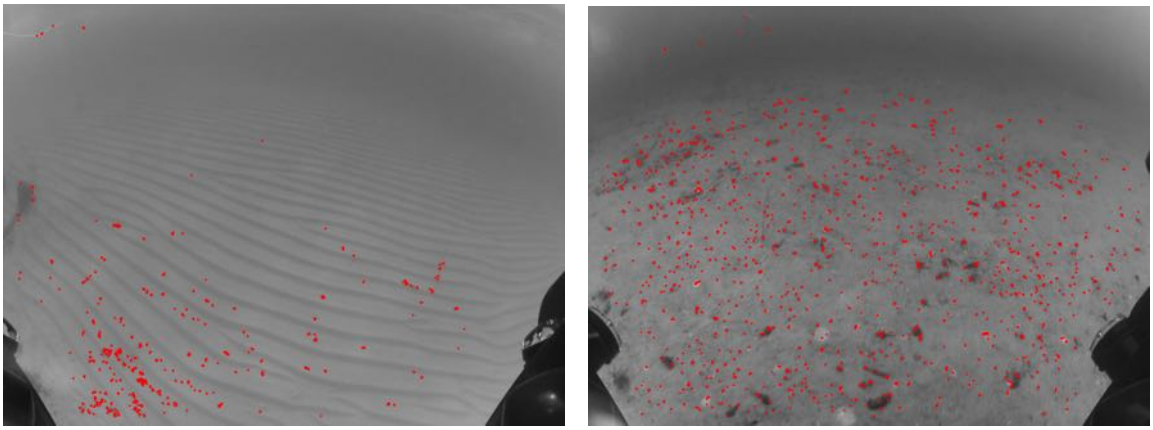
5.3 Results

This section serves as an overview over the observations that can be made from the experiments explained in the previous section. 5.3.1 puts a focus on all the insights which are related to working in the underwater environment. 5.3.2 summarizes all observations with regard to ORB SLAM.

5.3.1 Observations on underwater scenario

For a more structured overview the collected results are presented with respect to the underwater challenges presented in 2.4.2.

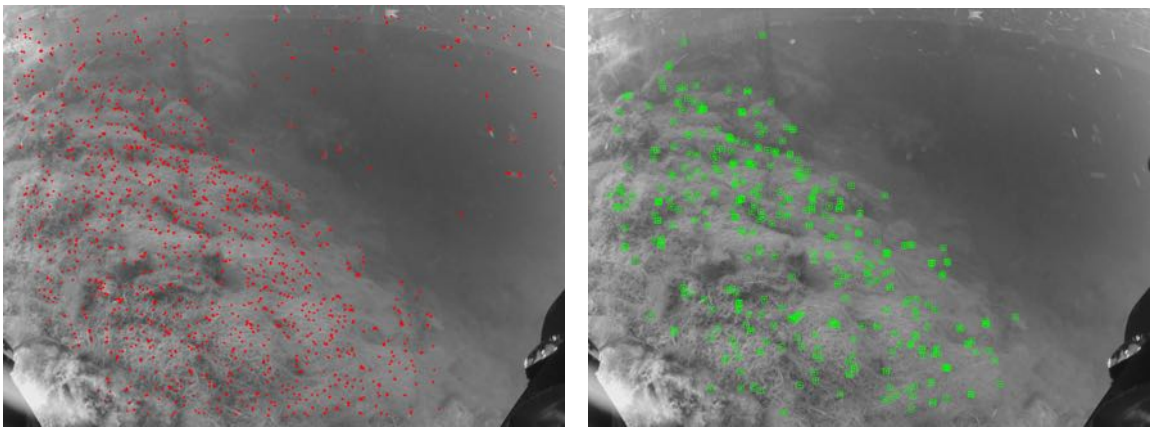
Monotony:
The problem of having areas with minimal structural change is observable multiple times within the executed experiments. Especially the datasets 4-9 and 13 contain a lot of monotonous areas. What is interesting about these datasets is that the results on the two different locations vary a lot. The experiments show that monotonous surroundings can be an issue for ORB SLAM's tracking, but it only really becomes a problem when the areas are both monotonous and low in texture. This becomes obvious when comparing the results at Omeo Wreck (5.2.5) with those at Point Walter 5.2.3. While tracking gets lost often due to low texture areas in the Omeo Wreck experiment, the experiment at Point Walter proved to be have one of the best overall results. The reason for this is depicted in figure 5.23. As is evident ORB SLAM is not able to extract enough features in low texture areas (a) as there are little corners. As soon as there are a few spots of different color on the ground as in (b) the algorithm is able to extract more features which helps with tracking.



(a) Low texture area in Omeo Wreck experiment (b) High texture area in Point Walter experiment

Figure 5.23: A comparison of extracted features in monotonous areas with low (a) and high texture (b). The red dots resemble extracted FAST corners.

Turbidity:
Having particles flowing in the water is something that can be observed in every single dataset collected. In practice ORB SLAM demonstrated that this is mostly not a problem. Figure 5.24 demonstrates this.



(a) Features extracted (b) Matches found

Figure 5.24: A frame showing algae covered rocks taken from dataset 10. (a) shows the extracted features in red. (b) illustrates matches found between the last frame's and the current frame's features in green. As can be seen in the upper right corner of (a) ORB SLAM does extract features from particles floating in the water. (b) however shows that none of these features was considered a match.

Even though ORB SLAM does extract features from particles, in most cases these are not matched. This happens mostly due to the fact that features extracted from particles move differently between frames than the features on the ground do. In spite of proving that good results can be achieved in the presence of turbidity it was not possible to rule out that they have an influence on the outcome.

Dynamics:
The dynamical nature of the underwater environment is most noticeable in those datasets recorded in the open sea (5.2.6). There are fish swimming through the images, algae moving with the surge and strong currents and waves moving the robot in unexpected ways. The impact of these dynamic changes on ORB SLAM is most apparent during initialization. As shown in figure 5.25 the algorithm tends to produce false feature matches during initialization when there is a lot of motion in the scene. This often leads to a repeatedly failing initialization.

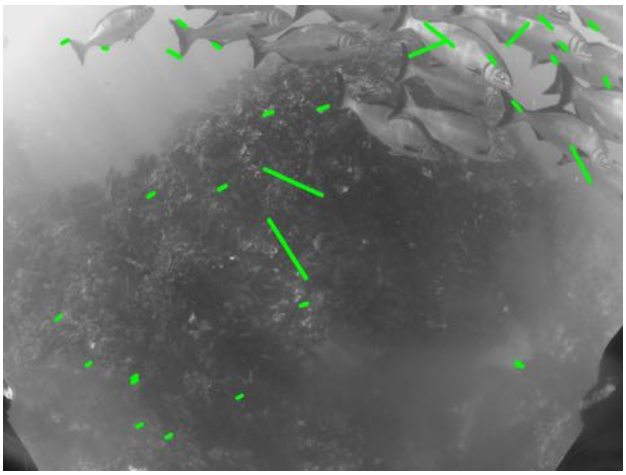


Figure 5.25: Initialization in a highly dynamic environment. Green lines connect feature matches between the initial and the current frame. In normal cases the green lines produced align with the motion of the robot. As in dynamic environments the features extracted move independent from the how the robot moves ORB SLAM creates wrong matches and often fails to initialize.

Nevertheless once the algorithm is able to initialize, the dynamic movements do not seem to have an effect on the tracking result anymore. Much like it is the case for turbidity the movement causes the features to move too much which results in ORB SLAM not tracking them. Figure 5.26 demonstrates this effect.

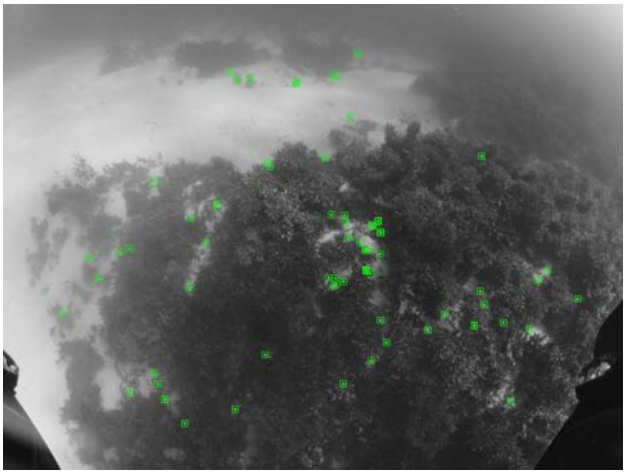
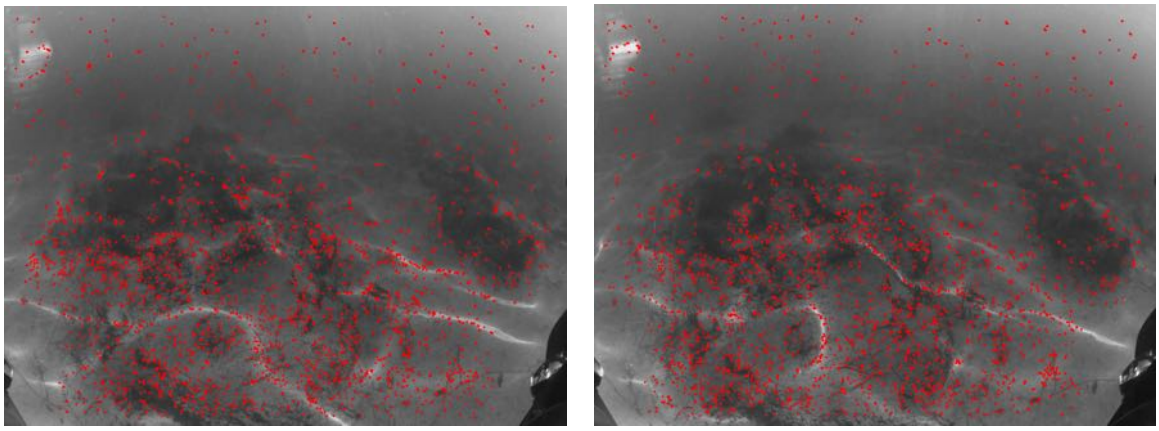


Figure 5.26: Tracking in a scene with moving algae. In the presence of moving objects ORB SLAM is not able to match features extracted within these objects. Instead it only tracks points which are in areas with little motion as can be seen in this image where most tracked features lie between the moving algae.

But even if there is no direct observable influence on the quality of tracking it can easily cause tracking to fail due to not being able to find enough matches.

Loss of colors:
Since ORB SLAM only works on gray scale images the loss of color has no effect on its results.

Lighting:
Lighting is definitely the environmental influence with the biggest impact on the results of ORB SLAM. This is especially visible when looking at the results of the Point Walter experiments. The only difference between datasets 4-6 and datasets 7-9 is that they were recorded on two different days with different lighting conditions. While ORB SLAM cannot even initialize in datasets 4-6 the results on datasets 7-9 proved to be some of the best overall. The problems on datasets with a lot of sunlight stem from the light ripples on the ground shown in 2.12 and 5.9 (a). As can be seen in figure 5.27 these ripples will cause ORB SLAM to detect many features along them.



(a) (b)

Figure 5.27: Feature extraction on two consecutive images of dataset 7. The ripples on the floor move a lot even in the short time between two images (less than 100 ms) which makes the scene very dynamic and keeps ORB SLAM from matching the features.

Furthermore the lighting also causes reflections in the ROV's casing as in the upper left corner of the images above. It also makes particles in the water more reflective and leads to even more features extracted on these. All these effects combined make the scene so highly dynamic that initialization constantly fails.

The negative effects of sunlight decrease when moving to deeper depth. Datasets 14 and 15 show that once the ROV moves below 3-4 m the ripples are less noticeable and initialization is more likely to be successful.

5.3.2 Observations on ORB SLAM

This section focuses on problems the experiments uncovered which are not related to the underwater scenario but rather to ORB SLAM itself. While the algorithm proved to be able to produce very good results when given the right conditions, there are also situations in which it runs into problems or behaves strangely. These include:

Permanent relocalization: A problem that occurs in several experiments is that tracking is lost at a certain point in the dataset and from thereon the only thing ORB SLAM does is trying to relocalize. In some cases the ROV is never moved back to the area where tracking was lost. ORB SLAM's behaviour in these cases is problematic because it leads to ignoring all the information present in the rest of the dataset. This problem is most visible in the results on datasets 12, 13 and 15. By resetting the algorithm once relocalizing failed for more than 30 frames it was possible to drastically increase the overall amount of successfully tracked frames. The problem with resetting is that it deletes all previous estimations. A possible solution to this could be the creation of sub maps proposed in 6.2.2.

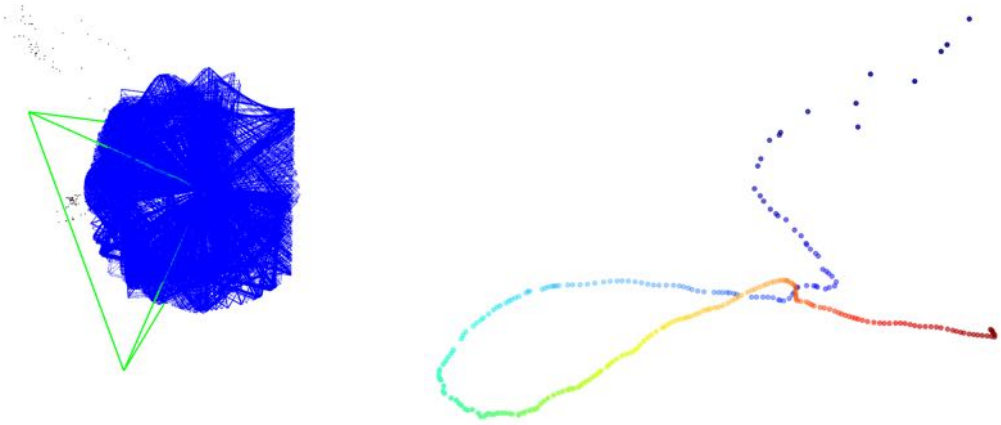
Problems with symmetry: Experiments 2 and 3 clearly demonstrate that ORB SLAM cannot handle areas which are highly repetitive or symmetric. In these environments it tends to create false relocalizations and form non-existing loops which in turn leads to a misrepresentation of the surroundings and the robot's movements.

Sudden scale-drift: A problem that can be observed in some of the datasets is that sometimes ORB SLAM suddenly starts to project all estimated map points right in front or even into the current camera position. This results in no more observable movement within ORB SLAM's visualization. Since tracking is not lost it keeps creating a map and key frames which creates a cluster as shown in figure 5.28 (a). Due to the fact that when plotting the estimated trajectory it looks like an actual path (figure 5.28 (b)), the assumption is made that this effect is caused by a quick and large scale-drift.

It mostly occurs when the ROV is at the water's surface where it is being moved by the waves, for example in the beginning of dataset 13.

Duplicate map: In some cases when the robot is moved along the same scene multiple times with a large change in viewing angle it happens that ORB SLAM creates duplicate maps for the same environment. On the next pass it then uses the one corresponding to the current viewing angle. This problem is most apparent in dataset 11.

Motion mirroring: Dataset 12 showed that when ORB SLAM only localizes relative to a movable object and there is nothing else visible in the scene, the algorithm mirrors the objects movements onto the ROV.



(a) (b)

Figure 5.28: (a): ORB SLAM's visualization of a map created while experiencing scale-drift. The green shape resembles the current camera pose, blue the estimated keyframe poses (the trajectory). All estimated keyframe poses seem to be bundled in a cluster. (b) 3D plot of the same estimated keyframe trajectory. The coloring indicates the temporal progression with blue being the start of the trajectory and red its end. This 3D plot shows that the trajectory was tracked, only the scale in ORB SLAM's visualization drifted so quickly that everything seems to be clustered within one point.

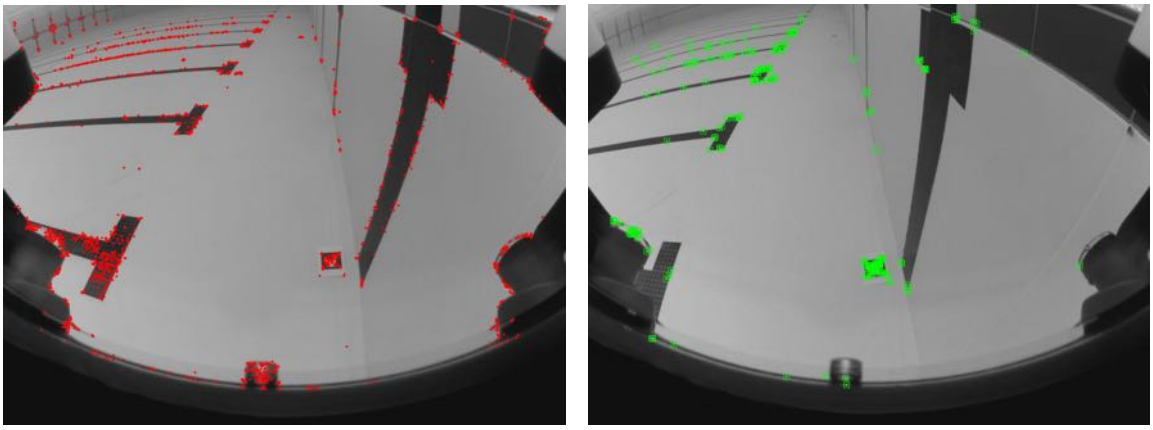
6 Improvements

While working with ORB SLAM and the BlueROV2 a lot of both bigger and smaller problems were discovered. Some of these were directly fixed or improved on while others could not be implemented within the time frame of this thesis. The following sections explain which enhancements have been realized and give suggestions on which parts of the algorithm could still be improved.

6.1 Implemented Improvements

6.1.1 False feature detection

Due to the way the BlueROV2 is constructed its own components are almost always visible in images recorded by the camera. This is especially true when the camera is facing down, which is what proved most effective for the use with ORB SLAM. One of the first problems discovered while experimenting was that ORB SLAM extracts a lot of features from the edges of the ROV's casing and components as can be seen in figure 6.1.



(a) (b)

Figure 6.1: Detection of features in the ROV's own components. (a) shows features found and (b) shows tracked feature matches.

These features, once tracked, quickly lead to large inaccuracies in the estimated trajectories and sometimes to complete tracking failure. In order to solve this problem both a hardware and a software solution were combined.

Hardware:

As can be seen in figure 4.1 (a) there is a dome shaped casing on the front side of the ROV. This is where the camera is located. The major reason for the casing being visible within the recorded images is that the camera sits relatively far back within that dome. By simply placing the camera further towards the front it was possible to greatly reduce

the image areas covered by the ROV's components. The difference introduced is obvious when comparing figures 5.6 and 5.9.

For moving the camera further to the front the BlueROV2's camera mount was remodeled and 3D printed. Figure 6.2 shows the old and new camera mount side by side. The displacement introduced by this can be seen in figure 6.3.

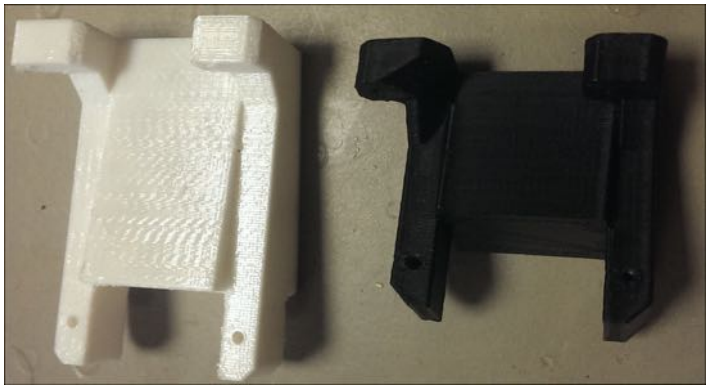
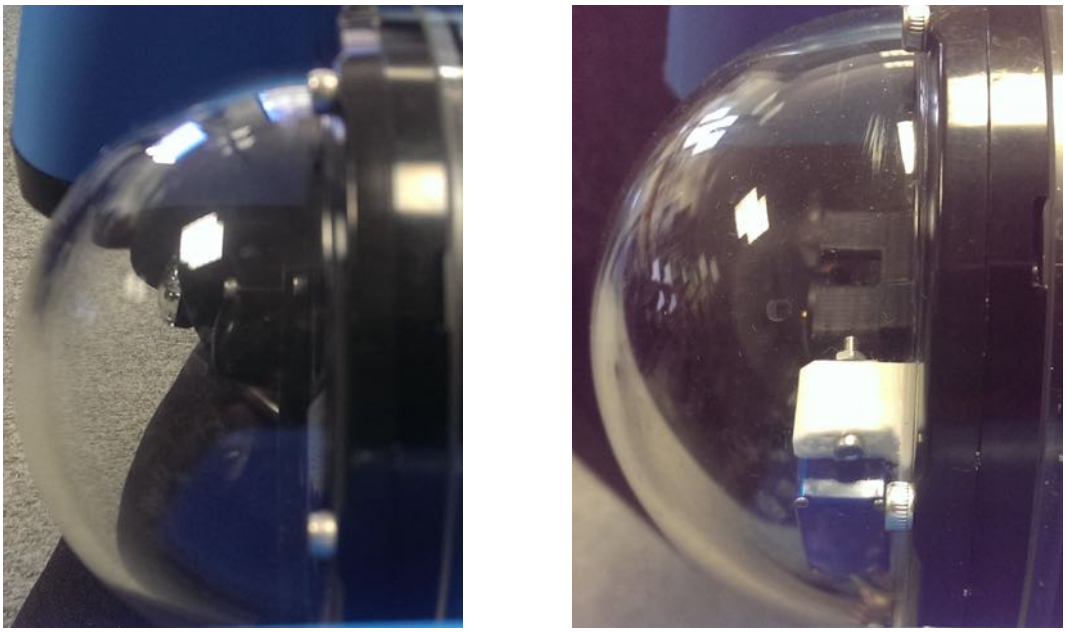


Figure 6.2: The ROV's camera mounts side by side. The black mount is the one originally delivered with the BlueROV2. The white mount was designed to be 2 cm longer.



(a) Before (b) After

Figure 6.3: Before and after comparison of the mounted camera's position within the BlueROV2's casing. On the left the camera is not visible because it sits so far back inside the dome that it is hidden by the black plastic. After the changes the camera sits further to the front of the dome.

Software

Since improving the ROV's camera mount did not completely remove its components from the image, the remaining image areas had to be excluded from feature extraction. For this purpose a python script was created which allows to select rectangular regions for exclusion from feature detection. Within this script rectangles can be drawn by clicking and dragging the mouse. Should something have gone wrong the selections can be cleared by pressing *space*. Once the selection is as desired the regions can be saved by pressing *enter*. On saving the rectangles are exported to the YAML file loaded by ORB SLAM for its parameters and then used by the feature extraction to know which regions to exclude. A demonstration of the script in action is given in figure 6.4. The difference this can make is demonstrated in figure 6.5.

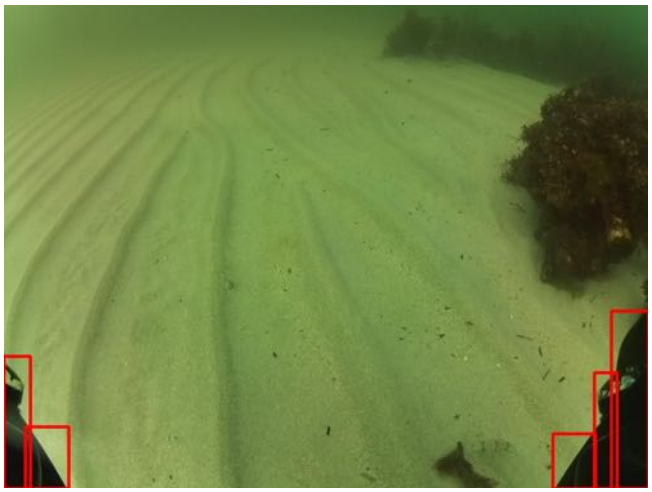
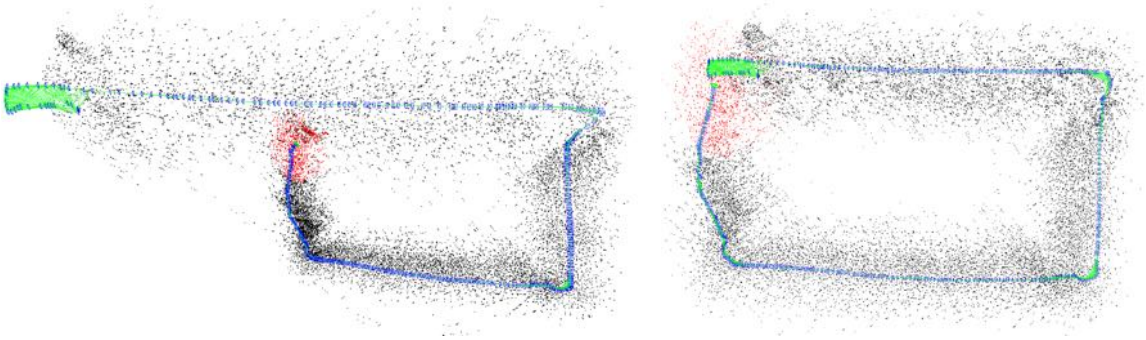


Figure 6.4: Example of excluding regions from feature extraction using the provided python script. The red rectangles shown in the bottom left and right corners are drawn with the mouse and then exported to ORB SLAM.



(a) (b)

Figure 6.5: Tracking result on dataset 9 with (b) and without (a) excluding the ROV's parts from feature extraction. When the parts are not excluded more drift is accumulated.

6.1.2 Logging

A main problem of ORB SLAM is that it is very hard to know what is currently happening within the algorithm. For example one never really knows why initialization is failing or why tracking suddenly stops. To make this information available while the program is running a logging module based on the c++ *boost* libraries was implemented.

This module allows to log in two ways and is easily extendible. By simply writing one of:

```
LOG(INFO) << "Logged to the console.";
LOG(STATUS) << "Logged to orb_slam_status.log.";
DLOG(INFO) << "Logged to the console when build with DEBUG flag.";
DLOG(STATUS) << "Logged to orb_slam_status.log when build with DEBUG flag.";
```

a message will be logged either to the console or to a file. Using *DLOG(INFO)* or *LOG(INFO)* allows to log information to the console. Both *DLOG(STATUS)* and *LOG(STATUS)* will log to a file called *orb_slam_status.log*. This was also used to log the status of ORB SLAM during the performed experiments and the basis for the plots shown in 5.2.

Using this module logging messages were integrated into all modules described in 3 which can be easily switched on and off using the buttons provided in the user interface. An example for the logging messages produced during loop closing is shown in figure 6.6.

```
[INFO]<Loop Closing> - &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&& LOOP CLOSING
[INFO]<Loop Closing> - Found 1 candidate keyframes for loop closing.
[INFO]<Loop Closing> - Checking consistency. Current number of consistent candidate groups: 1
[INFO]<Loop Closing> - New number of consistent candidate groups: 1
[INFO]<Loop Closing> - Consistency count for group: 0 = 3
[INFO]<Loop Closing> - Have enough consistency, going on to confirmation.
[INFO]<Loop Closing> - Using consistent candidates to compute Sim3.
[INFO]<Loop Closing> - Candidate 0 has 49 matches with current keyframe. -> Using it!
[INFO]<Loop Closing> - Found Sim(3) for candidate 0 trying to optimize it.
[INFO]<Loop Closing> - 76 matches after optimization.
[INFO]<Loop Closing> - Thats enough, candidate accepted for last test!
[INFO]<Loop Closing> - Doing one last matching with map points found in keyframes connected to candidate.
[INFO]<Loop Closing> - Found 140 more matches. Total is now: 216
[INFO]<Loop Closing> - Loop accepted!
Loop detected!
[INFO]<Loop Closing> - Implementing and refining loop.
Local Mapping STOP
[INFO]<Loop Closing> - Stopped local mapping.
[INFO]<Loop Closing> - Moving keyframes and map points local to current keyframe to their new position.
[INFO]<Loop Closing> - Removing duplicate map points.
[INFO]<Loop Closing> - Connecting both sides of the loop
[INFO]<Loop Closing> - Propagating loop through essential graph.
[INFO]<Loop Closing> - Starting a global bundle adjustment in a new thread
```

Figure 6.6: A logging example for the loop closing module.

In order for the logging not to have an impact on the performance of ORB SLAM all implemented logging was introduced using the *DLOG(INFO)* variant which will only make logging available when ORB SLAM is compiled with the *DEBUG* flag.

6.1.3 Parameters

While analyzing ORB SLAM's code an observation that could be made within every module was that there are a lot of hard coded parameters. A good example for this is the initialization module. For initialization to start there have to be at least 100 features extracted from the current image. To continue there have to be at least 100 matches between the features of the current frame and the features of the last frame. When choosing the homography for initialization (which is a decision based on a hard coded threshold) for a motion hypothesis to be accepted there need to be at least 50 matches which were triangulated correctly (again a decision based on a hard coded threshold) and the hypothesis with the second most matches triangulated must not have more than 75 % of the triangulated matches the one with most matches has.

While some of these parameters are certainly useful others seem like they were arbitrarily chosen and just stayed that way because they worked well enough. A point were this is very obvious is that after a motion hypothesis is accepted for initialization with 50 triangulated matches the initialization will actually be discarded again if there are not at least 100 points in the initial map, which cannot be possible if there were only 50 points triangulated.

As a first step to make these hard coded parameters more visible every occurrence in the code was marked with a comment:

```
if (nmatches < 100) //param
...
```

Doing this resulted in finding and marking over 150 occasions of hard coded parameters. In order to be able to give these parameters a proper evaluation in a future work they should be easily configurable and changeable during runtime both from within the code and from the User Interface (UI).

For this purpose a parameter class was introduced into ORB SLAM's code. This class allows to easily create parameters and their corresponding UI elements. There are four different UI elements that can be linked to a parameter's instance. Creating them works as follows:

```
Parameter<bool> button("Do something once", false, false, ParameterGroup::MAIN, []{});
Parameter<bool> toggle("Switch on/off", false, true, ParameterGroup::MAIN, []{});
Parameter<float> slider("Parameter1", 10, 0, 100, ParameterGroup::INITIALIZATION, []{});
Parameter<int> textBox("Parameter2", 10, ParameterGroup::INITIALIZATION, []{});
```

As the snippet shows, every parameter belongs to a parameter group. This defines where the corresponding UI element will be created. There is a pane associated to each of ORB SLAM's modules. When creating a parameter with parameter group *INITIALIZATION* its corresponding UI element is placed in the pane for initialization parameters.

6.1.4 Convenience Improvements

In addition to the already mentioned improvements a few usability enhancements were integrated into ORB SLAM. They include:

1. **Pause function:** Since ORB SLAM generally processes tens of frames per second it can sometimes become hard to follow what is currently going on. For the purpose of being able to slowly step through the process a pausing function was implemented which allows to stop the algorithm at any time. Once stopped it is then possible to step through the frames one by one and see how ORB SLAM works on every single image.

2. **Fast-forward function:** In contrast to the pause function there are also situations were one just wants to take a quick look at a certain situation inside a dataset. With datasets which are often more than 10000 images long waiting for one single situation can quickly become tiring. For this reason a fast forwarding function was implemented where a number of frames can be entered that are then going to be processed as fast as the CPU allows. After the entered frames have passed ORB SLAM returns to normal speed.

3. **Trigger relocalization:** This function was implemented for easier evaluation of how well ORB SLAM is able to relocalize within some datasets. Pressing the *Trigger relocalization* button which was added to ORB SLAM's main menu allows to simulate a tracking loss at any time.

4. **Plotting trajectories:** Even though ORB SLAM saves its estimated trajectory after it is done, there is no means of visualizing it again once the process is done. During this thesis a tool was created which can plot the driven trajectory in both 2D and 3D.

6.2 Suggestions

Even though it was possible to show that, given the right conditions, ORB SLAM can work very well underwater, there are still a lot of problems which could be improved. Both in the context of underwater robotics and ORB SLAM in general. Since it was not possible to implement and test them all this section serves as a collection of ideas to further improve on the collected results.

6.2.1 Underwater scenario

This section outlines ideas that could be tried in order to improve on the problems that were encountered while using ORB SLAM in the underwater scenario. Obviously most of the problems encountered could be solved by using additional sensors like sonar. Since this thesis focuses on using the sensors provided by the BlueROV2 (camera and IMU) only improvements possible with regard to this sensor setting are considered.

Monotony

As was shown in 5.3.1 monotony only becomes a problem when it is combined with low-texture areas as is the case for areas consisting mainly of sand. When faced with these situations ORB SLAM consistently looses tracking or fails to produce any results.

A very interesting idea for overcoming this would be to mix feature-based methods with dense approaches. Even when there are little point features visible as e.g. in figure 5.18 (a) dense approaches should still be able to to track movement based on visible gradients.

Dynamics:

A simple solution to overcome the problem of tracking loss due to quick motions would be to increase the overall frames per second the datasets are recorded with. Since this would result in less movement per frame it should also allow for more robust tracking.

Another possibility would be to use a motion model that can predict the robot's movements based on its control inputs. This would however only be able to take movements into account which stem from the robot's thrusters. Motion induced by surge or waves would still remain a problem.

Lighting:

As was shown in the performed experiments lighting is clearly the biggest issue for camera based underwater SLAM. While the scattering of light definitely amplifies the problem in the underwater environment it is also a problem that is encountered when dealing with SLAM on land. Recent approaches like [64] or [65] were able to show that using illumination invariant image transformations as proposed in [66] could greatly improve on the problem introduced by varying lighting conditions.

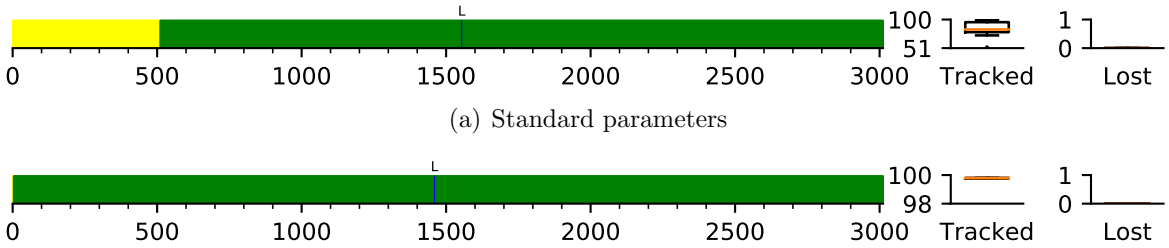
6.2.2 ORB SLAM

This section lists improvements that are not specific to the underwater setting but could rather improve ORB SLAM's performance in any given environment.

Parameter evaluation

As already mentioned in 6.1.3 ORB SLAM includes many hard coded parameters which for the most part seem to have been chosen arbitrarily. It would be very interesting to see how much of an effect optimizing these parameters could have on the outcome of the algorithm. Using the tools provided by this thesis, outcomes could be easily logged and tested.

To show how much tweaking some parameters can accomplish the fifth dataset presented in 5.2.3 was rerun after lowering the hard coded thresholds on the initialization. Before initialization sometimes took very long so that the amount of tracked frames fluctuated between 50 % and almost a 100 %. By simply lowering the minimum feature matches between current and initial frame and the minimum map points in the initial map from 100 to 50 and then testing it 10 times this fluctuation can be completely removed without any noticeable accuracy loss. The results can be viewed in 6.7.



(a) Standard parameters

(b) Tweaked parameters

Figure 6.7: Results on dataset 8 with standard parameters (a) and after changing parameters (b). While the estimated trajectory and map did not noticeably change it was possible to completely remove initialization inconsistency.

For reasons of comparability optimizing parameters should be done on data with ground truth instead of on the data provided by this thesis.

Sub maps

As explained in 5.3.2 ORB SLAM sometimes ignores a lot of information due to constantly trying to relocalize once tracking is lost. This problem could be solved by keeping track of sub maps. Whenever tracking is lost two threads could be started. One thread tries to relocalize within already known maps and another thread starts the normal initialization and tracking process. When initialization was successful a new map is created which can be merged with existing maps once the relocalization thread finds a match within an older map. For correction of drift between maps the same approach as in the loop closing module could be used.

Runtime Optimization

Even though ORB SLAM is already able to run on a CPU in real-time there is still much room for performance optimization within its code. There is for example a reimplementation [67] of the DBoW2 library which claims to be up to 80 times faster than the one used by ORB SLAM.

IMU Integration

As already proposed by the ORB SLAM authors in [4] introducing the usage of IMU measurements into the algorithm can overcome some of the problems linked to monocular SLAM. Unfortunately the related code has never been publicly released so it would have to be implemented manually. The method described is able to successfully determine scale and get rid of scale drift while simultaneously estimating gyroscope and accelerometer biases. Furthermore it also allows to extract direction of gravity which allows to create maps aligned with the actual environment.

What is not even mentioned in [4] is that it could also be used for sanity checks for relocalization and loop closing. As was visible in the pool experiments in 5.2.2 ORB SLAM has problems in highly symmetric environments and tends to create wrong relocalizations and loops. By incorporating IMU measurements it could be made sure that the algorithm does not relocalize on the wrong side of the pool. Furthermore this could also be used to negate the effect seen in experiment 12 in 5.2.4 where ORB SLAM maps the movement of the object being tracked onto the ROV.

Since IMU data has been recorded as a part of the provided datasets, the datasets could be reused for evaluating this feature.

Tracking 2D Features

As long as ORB SLAM tries to initialize, it does not retain any of the information extracted from the frames. Taking inspiration from DT SLAM the extracted 2D features could also be tracked and then incrementally triangulated. By doing this DT SLAM is able to track pure rotations and does not need an explicit initialization as ORB SLAM does. For more information on how DT SLAM realizes this see [68].

7 Conclusion and Outlook

This thesis was created with the goal of finding and evaluating a SLAM algorithm applicable in the underwater environment using only the limited capabilities of the BlueROV2. For the purpose of finding a suitable algorithm an extensive literature review was performed and summarized in the lists at A.1 and A.2. To the best of the author's knowledge these lists present the most complete overview over SLAM algorithms available at the time of writing this thesis.

As a result of this research the ORB SLAM algorithm was picked for testing on the BlueROV2. The thesis provides an in-depth explanation of ORB SLAM explaining the used techniques and the overall structure of the algorithm. The description of the program's internal workings is supported by a detailed diagram visualizing the flow of information in its code. In addition usability functions like logging and easily manipulatable parameters were introduced. All of this allows for an easier introduction into the topic and will make development simpler for anyone working on the topic in the future.

For testing the algorithm over 40 different datasets were recorded in varying environments. The evaluation performed on these showed that ORB SLAM can work very well given the right conditions. It does however also show that the algorithm struggles with some of the characteristics of the underwater environment. Especially the highly dynamic lighting and surroundings with lots of moving objects like fish and algae caused ORB SLAM to fail repeatedly.

It was not possible to completely exhaust the potential of the collected data within the time frame of this thesis. For example the recoded IMU data has not been used at all yet. This data could be utilized for testing IMU integration which should be able to significantly improve on discovered problems like the ones encountered in symmetric environments as described in 5.3. The datasets could also be used for testing other enhancements like the introduction of sub maps or tracking of 2D features. But these are not the only direction future works could be headed. Section 6.2 provides a wide range of other suggestions for improving ORB SLAM in general and with regard to the underwater scenario.

One thing this thesis was not able to provide was data with sufficient ground truth. This is definitely a large problem and should be a major focus for future works since so far it was only possible to evaluate the algorithm's accuracy in a qualitative way. While the data provided can be utilized to show that enhancements work and improve the result, it is not possible to perform any comparison with other algorithms as long as there is no ground-truth.

In summary this thesis was able to show that SLAM can work well underwater, even when only using a camera. Nonetheless it also demonstrates that there is still a lot of room for improvement. While the explanations and visualisations provided by this thesis allow for a quicker introduction into the topic, the tools, data and insights created and collected can be used as the foundation for the steps that need to be taken to further increase ORB SLAM's performance underwater.

References

[1] Raúl Mur-Artal and Juan D. Tardós. "Fast relocalisation and loop closing in keyframe-based SLAM". In: *Proc. - IEEE Int. Conf. Robot. Autom.* (2014), pp. 846–853. ISSN: 10504729. DOI: 10.1109/ICRA.2014.6906953.

[2] Raul Mur-Artal, J. M M Montiel and Juan D. Tardos. "ORB-SLAM: A Versatile and Accurate Monocular SLAM System". In: *IEEE Trans. Robot.* 31.5 (2015), pp. 1147–1163. ISSN: 15523098. DOI: 10.1109/TRO.2015.2463671. arXiv: 1502.00956.

[3] Raul Mur-Artal and Juan D. Tardos. "ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras". In: (2016). ISSN: 15523098. DOI: 10.1109/TRO.2012.2197158. arXiv: 1610.06475.

[4] Raul Mur-Artal and Juan D. Tardos. "Visual-Inertial Monocular SLAM with Map Reuse". In: *IEEE Robot. Autom. Lett.* (2017). ISSN: 2377-3766. DOI: 10.1109/LRA.2017.2653359. arXiv: 1610.05949.

[5] Sebastian Thrun. *Probabilistic robotics*. Vol. 45. 3. 2002. DOI: 10.1145/504729.504754. URL: http://portal.acm.org/citation.cfm?doid=504729.504754.

[6] Tim Bailey and Hugh Durrant-Whyte. "Simultaneous localization and mapping (SLAM): Part I". In: *IEEE Robot. Autom. Mag.* 13.3 (2006), pp. 108–117. ISSN: 10709932. DOI: 10.1109/MRA.2006.1638022.

[7] Megan R Naminski. *An Analysis of Simultaneous Localization and Mapping (SLAM) Algorithms*. 2013. URL: http://goo.gl/nSzyLV.

[8] Cesar Cadena et al. "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age". In: *IEEE Trans. Robot.* 32.6 (2016), pp. 1309–1332. ISSN: 15523098. DOI: 10.1109/TRO.2016.2624754. arXiv: arXiv:1606.05830v2.

[9] A. H. A. Rahman S. B. Samsuri, H. Zamzuri, M. A. A. Rahman, S. A. Mazlan. "Computational Cost Analysis Of Extended Kalman Filter In Simultaneous Localization & Mapping (EKF-SLAM) Problem For Autonomous Vehicle". In: *ARPN J. Eng. Appl. Sci.* 10.17 (2015), pp. 7764–7768. URL: https://goo.gl/QvvmJL.

[10] L.M. Paz, J.D. Tardos and J. Neira. "Divide and Conquer: EKF SLAM in O(n)". In: *IEEE Trans. Robot.* 24.5 (2008), pp. 1107–1120. ISSN: 1552-3098. DOI: 10.1109/TRO.2008.2004639.

[11] Michael Montemerlo et al. "FastSLAM : A Factored Solution to the Simultaneous Localization and Mapping Problem". In: (2002). URL: https://goo.gl/gcKxC3.

[12] Michael Montemerlo et al. "Fast SLAM 2.0 : an improved particle filtering algorithm for simultaneous localization and mapping that provably converges". In: *Int. Jt. Conf. Artif. Intell. IJCAI* (2003), pp. 1151–1156. URL: http://robots.stanford.edu/papers/Montemerlo03a.pdf.

[13] Sebastian Thrun et al. "Fastslam: An efficient solution to the simultaneous localization and mapping problem with unknown data association". In: *J. Mach. Learn. Res.* (2004). URL: http://robots.stanford.edu/papers/Thrun03g.pdf.

[14] Sebastian Thrun. "Particle Filters in Robotics (Invited Talk)". In: (2002). arXiv: 1301.0607. URL: http://arxiv.org/abs/1301.0607.

[15] Michael Calonder. *EKF SLAM vs. FastSLAM - A Comparison*. URL: https://goo.gl/QzCTtZ.

[16] Giorgio Grisetti et al. "A tutorial on graph-based SLAM". In: *IEEE Intell. Transp. Syst. Mag.* (2010), pp. 31–43. ISSN: 1939-1390. DOI: 10.1109/MITS.2010.939925.

[17] Sebastian Thrun and Michael Montemerlo. "The GraphSLAM Algorithm with Applications to Large-Scale Mapping of Urban Structures". In: *Int. J. Rob. Res.* 25 (2006), pp. 403–429. DOI: 10.1177/0278364906065387.

[18] Pieter Abbeel. *GraphSLAM (Slides)*. URL: https://goo.gl/Uyq8qD (visited on 20/08/2017).

[19] Richard A. Newcombe, Steven J. Lovegrove and Andrew J. Davison. "DTAM: Dense tracking and mapping in real-time". In: *Proc. IEEE Int. Conf. Comput. Vis.* (2011), pp. 2320–2327. ISSN: 1550-5499. DOI: 10.1109/ICCV.2011.6126513.

[20] Georg Klein and David Murray. "Parallel tracking and mapping for small AR workspaces". In: *2007 6th IEEE ACM Int. Symp. Mix. Augment. Reality, ISMAR* (2007). DOI: 10.1109/ISMAR.2007.4538852.

[21] Michael Milford, Gordon Wyeth and David Prasser. "RatSLAM: a hippocampal model for simultaneous localization and mapping". In: *Robot. Autom. ...* May 2004 (2004), pp. 403–408. ISSN: 1050-4729. DOI: 10.1109/ROBOT.2004.1307183.

[22] Alexei Samsonovich and Bruce L. McNaughton. "Path integration and cognitive mapping in a continuous attractor neural network model." In: *J. Neurosci.* 17.15 (1997), pp. 5900–5920. ISSN: 0270-6474. URL: https://www.ncbi.nlm.nih.gov/pubmed/9221787.

[23] David Ball et al. "OpenRatSLAM: An open source brain-based SLAM system". In: *Auton. Robots* 34.3 (2013), pp. 149–176. ISSN: 09295593. DOI: 10.1007/s10514-012-9317-9.

[24] Michael John Milford. *Robot Navigation from Nature*. Vol. 41. 2008, p. 196. ISBN: 9783540775195. DOI: 10.1007/978-3-540-77520-1.

[25] Michael Milford. *RatSLAM and Grid Cells (Slides)*. URL: https://goo.gl/akZsHj (visited on 17/11/2017).

[26] Ethan Rublee et al. "ORB: An efficient alternative to SIFT or SURF". In: *Proc. IEEE Int. Conf. Comput. Vis.* (2011), pp. 2564–2571. ISSN: 1550-5499. DOI: 10.1109/ICCV.2011.6126544.

[27] Jakob Engel, Thomas Schoeps and Daniel Cremers. "LSD-SLAM: Large-Scale Direct Monocular SLAM". In: (2014), pp. 834–849. ISSN: 16113349. DOI: 10.1007/978-3-319-10605-2_54.

[28] Jakob Engel, Jurgen Sturm and Daniel Cremers. "Semi-dense visual odometry for a monocular camera". In: *Proc. IEEE Int. Conf. Comput. Vis.* (2013), pp. 1449–1456. ISSN: 1550-5499. DOI: 10.1109/ICCV.2013.183.

[29] Javier Civera, Andrew J Davison and J M Martinez Montiel. "Inverse Depth Parameterization for Monocular SLAM". In: *IEEE Trans. Robot.* 24.5 (2008), pp. 932–945. ISSN: 15523098. DOI: 10.1109/TRO.2008.2003276.

[30] H. Strasdat, J. M. M. Montiel and A. Davison. "Scale Drift-Aware Large Scale Monocular SLAM". In: *Robot. Sci. Syst. VI* (2010). ISSN: 2330765X. DOI: 10.15607/RSS.2010.VI.010. URL: http://www.roboticsproceedings.org/rss06/p10.pdf.

[31] Ethan Eade. *Lie Groups for 2D and 3D Transformations*. 2013. URL: http://ethaneade.com/lie.pdf.

[32] Arren Glover et al. "OpenFABMAP: An Open Source Toolbox for Appearance-based Loop Closure Detection". In: *Int. Conf. Robot. Autom.* May. Saint Paul, MN, 2012. DOI: 10.1109/ICRA.2012.6224843. URL: https://eprints.qut.edu.au/50317/1/glover{_}ICRA2012{_}final.pdf.

[33] Rainer Kümmerle et al. "G2o: A general framework for graph optimization". In: *Proc. - IEEE Int. Conf. Robot. Autom.* June (2011), pp. 3607–3613. ISSN: 10504729. DOI: 10.1109/ICRA.2011.5979949.

[34] Georges Younes, Daniel Asmar and Elie Shammas. "A survey on non-filter-based monocular Visual SLAM systems". 2016. URL: http://arxiv.org/abs/1607.00470.

[35] Khalid Yousif, Alireza Bab-Hadiashar and Reza Hoseinnezhad. "An Overview to Visual Odometry and Visual SLAM: Applications to Mobile Robotics". In: *Intell. Ind. Syst.* 1.4 (2015), pp. 289–311. ISSN: 2363-6912. DOI: 10.1007/s40903-015-0032-7. URL: http://link.springer.com/10.1007/s40903-015-0032-7.

[36] Mark Maimone, Yang Cheng and Larry Matthies. "Visual Odometry on the Mars Exploration Rovers". In: *2005 IEEE Int. Conf. Syst. Man Cybern.* 1 (2007), pp. 903–910. ISSN: 1070-9932. DOI: 10.1002/rob.20184. URL: http://ieeexplore.ieee.org/document/1571261/.

[37] Felipe Guth et al. "Underwater SLAM: Challenges, state of the art, algorithms and a new biologically-inspired approach". In: *5th IEEE RAS/EMBS Int. Conf. Biomed. Robot. Biomechatronics* (2014), pp. 981–986. ISSN: 2155-1774. DOI: 10.1109/BIOROB.2014.6913908. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6913908.

[38] Liam Paull et al. "AUV navigation and localization: A review". In: *IEEE J. Ocean. Eng.* 39.1 (2014), pp. 131–149. ISSN: 03649059. DOI: 10.1109/JOE.2013.2278891.

[39] Franco Hidalgo and Thomas Bräunl. "Underwater Robot SLAM : Instrumentation and Frameworks". In: *6th Int. Conf. Autom. Robot. Appl.* (2015). URL: http://ieeexplore.ieee.org/abstract/document/7081165/.

[40] A. Quattrini Li et al. "Experimental Comparison of open source Vision based State Estimation Algorithms". In: *Int. Symp. Exp. Robot.* January. 2016, pp. 775–786. DOI: 10.1007/978-3-319-50115-4_67.

[41] Alejo Concha et al. "Real-time localization and dense mapping in underwater environments from a monocular sequence". In: *MTS/IEEE Ocean. 2015 - Genova Discov. Sustain. Ocean Energy a New World* 1 (2015). DOI: 10.1109/OCEANS-Genova.2015.7271476.

[42] Pep Lluis Negre, Francisco Bonin-Font and Gabriel Oliver. "Cluster-based loop closing detection for underwater slam in feature-poor regions". In: *Proc. - IEEE Int. Conf. Robot. Autom.* 2016-June (2016), pp. 2589–2595. ISSN: 10504729. DOI: 10.1109/ICRA.2016.7487416.

[43] Stephen M. Chaves et al. "Opportunistic sampling-based active visual SLAM for underwater inspection". In: *Auton. Robots* 40.7 (2016), pp. 1245–1265. ISSN: 15737527. DOI: 10.1007/s10514-016-9597-6.

[44] Luan Silveira et al. "An open-source bio-inspired solution to underwater SLAM". In: *IFAC-PapersOnLine* 28.2 (2015), pp. 212–217. ISSN: 24058963. DOI: 10.1016/j.ifacol.2015.06.035.

[45] Rafael Garcia and Nuno Gracias. "Detection of Interest Points in Turbid Underwater Images". In: *OCEANS* (2011). DOI: 10.1109/Oceans-Spain.2011.6003605.

[46] Hauke Strasdat, J. M M Montiel and Andrew J. Davison. "Real-time monocular SLAM: Why filter?" In: *Proc. - IEEE Int. Conf. Robot. Autom.* (2010), pp. 2657–2664. ISSN: 10504729. DOI: 10.1109/ROBOT.2010.5509636.

[47] *Wikipedia: Pyramid(image processing)*. URL: https://goo.gl/dN7RdV (visited on 26/09/2017).

[48] Edward Rosten and Tom Drummond. "Machine learning for high-speed corner detection". In: *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)* 3951 LNCS (2006), pp. 430–443. ISSN: 16113349. DOI: 10.1007/11744023_34.

[49] Paul L Rosin. "Measuring Corner Properties". In: *Comput. Vis. Image Underst.* 73.2 (1999), pp. 291–307. ISSN: 1077-3142. DOI: 10.1006/cviu.1998.0719.

[50] Michael Calonder et al. "BRIEF: Binary robust independent elementary features". In: *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)* 6314 LNCS.PART 4 (2010), pp. 778–792. ISSN: 03029743. DOI: 10.1007/978-3-642-15561-1_56.

[51] OpenMVG Authors. *Homography matrix*. URL: http://openmvg.readthedocs.io/en/latest/openMVG/multiview/multiview/ (visited on 29/09/2017).

[52] Andrew Zisserman Richard Hartley. *Multiple View Geometry*. 2003. ISBN: 9780521540513. URL: https://goo.gl/2WmqiC.

[53] O.D. Faugeras and F. Lustman. "Motion and Structure From Motion in a Piecewise Planar Environment". In: *Int. J. Pattern Recognit. Artif. Intell.* 02.03 (1988), pp. 485–508. ISSN: 0218-0014. DOI: 10.1142/S0218001488000285. URL: http://www.worldscientific.com/doi/abs/10.1142/S0218001488000285.

[54] Dorian Galvez-Lopez and Juan D. Tardos. "Bags of Binary Words for Fast Place Recognition in Image Sequences". In: *IEEE Conf. Comput. Vis. Pattern Recognit.* 28.5 (2012), pp. 1188–1197. ISSN: 1552-3098. DOI: 10.1109/TRO.2012.2197158. URL: http://doriangalvez.com/papers/GalvezTRO12.pdf.

[55] Vincent Lepetit, Francesc Moreno-Noguer and Pascal Fua. "EPnP: An accurate O(n) solution to the PnP problem". In: *Int. J. Comput. Vis.* 81.2 (2009), pp. 155–166. ISSN: 09205691. DOI: 10.1007/s11263-008-0152-6.

[56] Chris Sweeney. *Structure from Motion (SfM)*. URL: http://www.theia-sfm.org/sfm.html (visited on 06/10/2017).

[57] Berthold K. P. Horn. "Closed-form solution of absolute orientation using unit quaternions". In: *J. Opt. Soc. Am. A* 4.4 (1987), p. 629. ISSN: 1084-7529. DOI: 10.1364/JOSAA.4.000629. URL: https://www.osapublishing.org/abstract.cfm?URI=josaa-4-4-629.

[58] BlueRobotics. *BlueROV 2 Datasheet*. URL: http://bluerobotics.com/downloads/bluerov2.pdf (visited on 10/10/2017).

[59] BlueRobotics. *BlueROV2 Website*. URL: http://www.bluerobotics.com/store/rov/bluerov2/ (visited on 10/10/2017).

[60] PIXHAWK Project Developers. *Pixhawk Autopilot*. URL: https://pixhawk.org/modules/pixhawk (visited on 10/10/2017).

[61] Xsens. *Xsens MTi documentation*. URL: https://www.xsens.com/products/mti/ (visited on 27/10/2017).

[62] James Bowman and Patrick Mihelich. *ROS Camera Calibrration*. URL: http://wiki.ros.org/camera{_}calibration.

[63] Western Australian Museum. *Shipwreck Databases*. URL: http://www.museum.wa.gov.au/maritime-archaeology-db/wrecks/omeo (visited on 19/10/2017).

[64] Seonwook Park, Thomas Schöps and Marc Pollefeys. "Illumination Change Robustness in Direct Visual SLAM". In: *ICRA(International Conf. Robot. Autom.* Singapore, 2017. DOI: 10.1109/ICRA.2017.7989525. URL: https://cvg.ethz.ch/research/illumination-change-robust-dslam/park2017icra.pdf.

[65] Colin McManus et al. "Shady dealings: Robust, long-term visual localisation using illumination invariance". In: *Proc. - IEEE Int. Conf. Robot. Autom.* (2014), pp. 901–906. ISSN: 10504729. DOI: 10.1109/ICRA.2014.6906961.

[66] Sivalogeswaran Ratnasingam and T. Martin McGinnity. "Chromaticity space for illuminant invariant recognition". In: *IEEE Trans. Image Process.* 21.8 (2012), pp. 3612–3623. ISSN: 10577149. DOI: 10.1109/TIP.2012.2193135.

[67] Rafael Muñoz-Salinas. *Fast Bag of Words (FBoW)*. 2017. URL: https://github.com/rmsalinas/fbow (visited on 10/11/2017).

[68] Herrera C. Daniel et al. "DT-SLAM: Deferred triangulation for robust SLAM". In: *Proc. - 2014 Int. Conf. 3D Vision, 3DV 2014* (2014), pp. 609–616. DOI: 10.1109/3DV.2014.49.

[69] John Wang and Edwin Olson. "AprilTag 2: Efficient and robust fiducial detection". In: *IEEE Int. Conf. Intell. Robot. Syst.* 2016-Novem (2016), pp. 4193–4198. ISSN: 21530866. DOI: 10.1109/IROS.2016.7759617.

[70] Edwin Olson. "AprilTag: A robust and flexible visual fiducial system". In: *Proc. - IEEE Int. Conf. Robot. Autom.* (2011), pp. 3400–3407. DOI: 10.1109/ICRA.2011.5979561.

[71] Matthew Klingensmith, Siddartha S. Sirinivasa and Michael Kaess. "Articulated Robot Motion for Simultaneous Localization and Mapping (ARM-SLAM)". In: *IEEE Robot. Autom. Lett.* 1.2 (2016), pp. 1156–1163. ISSN: 23773766. DOI: 10.1109/LRA.2016.2518242.

[72] Jan Steckel and Herbert Peremans. "Spatial sampling strategy for a 3D sonar sensor supporting BatSLAM". In: *IEEE Int. Conf. Intell. Robot. Syst.* 2015-Decem (2015), pp. 723–728. ISSN: 21530866. DOI: 10.1109/IROS.2015.7353452.

[73] Jan Steckel and Herbert Peremans. "BatSLAM: Simultaneous Localization and Mapping Using Biomimetic Sonar". In: *PLoS One* 8.1 (2013). ISSN: 19326203. DOI: 10.1371/journal.pone.0054076.

[74] Angela Dai et al. "BundleFusion: Real-time Globally Consistent 3D Reconstruction using On-the-fly Surface Re-integration". In: 36.3 (2017). arXiv: 1604.01093.

[75] Katrin Pirker, M Ruther and Horst Bischof. "CD SLAM - continuous localization and mapping in a dynamic world". In: *IEEE/RSJ Int. Conf. Intell. Robot. Syst. IROS* (2011), pp. 3990–3997. DOI: 10.1109/IROS.2011.6094588.

[76] Katrin Pirker, Matthias Rüther and Horst Bischof. "Histogram of Oriented Cameras - A New Descriptor for Visual SLAM in Dynamic Environments". In: *Br. Mach. Vis. Conf.* (2010), pp. 76.1–76.12. DOI: 10.5244/C.24.76.

[77] Esha D. Nerurkar, Kejian J. Wu and Stergios I. Roumeliotis. "C-KLAM: Constrained keyframe-based localization and mapping". In: *Proc. - IEEE Int. Conf. Robot. Autom.* (2014), pp. 3638–3643. ISSN: 10504729. DOI: 10.1109/ICRA.2014.6907385.

[78] Keisuke Tateno et al. "CNN-SLAM: Real-time dense monocular SLAM with learned depth prediction". In: (2017). arXiv: 1704.03489. URL: http://arxiv.org/abs/1704.03489.

[79] Gijs Dubbelman and Brett Browning. "COP-SLAM: Closed-Form Online Pose-Chain Optimization for Visual SLAM". In: *IEEE Trans. Robot.* 31.5 (2015), pp. 1194–1213. ISSN: 15523098. DOI: 10.1109/TRO.2015.2473455.

[80] Gijs Dubbelman and Brett Browning. "Closed-form Online Pose-chain SLAM". In: *IEEE Int. Conf. Robot. Autom.* 2013. DOI: 10.1109/ICRA.2013.6631319. URL: https://goo.gl/CQ7kZ8.

[81] Gijs Dubbelman, Isaac Esteban and Klamer Schutte. "Efficient trajectory bending with applications to loop closure". In: *IEEE/RSJ 2010 Int. Conf. Intell. Robot. Syst. IROS 2010 - Conf. Proc.* (2010), pp. 4836–4842. ISSN: 2153-0858. DOI: 10.1109/IROS.2010.5652656.

[82] Danping Zou and Ping Tan. "CoSLAM : Collaborative Visual SLAM in Dynamic Environments". In: *IEEE Trans. Pattern Anal. Mach. Intell.* (2013). DOI: 10.1109/TPAMI.2012.104.

[83] Guilherme B. Zaffari et al. "Exploring the DolphinSLAM's parameters". In: (2016). DOI: 10.1109/OCEANSAP.2016.7485531.

[84] Austin Eliazar and Ronald Parr. "DP-SLAM 2.0". In: *IEEE Int. Conf. Robot. Autom.* 2 (2004), pp. 1314–1320. ISSN: 1050-4729. DOI: 10.1109/ROBOT.2004.1308006.

[85] Austin Eliazar and Ronald Parr. *DP-SLAM: Fast, robust simultaneous localization and mapping without predetermined landmarks.* URL: https://goo.gl/aCQE5K (visited on 04/11/2017).

[86] Alejo Concha and Javier Civera. "DPPTAM: Dense piecewise planar tracking and mapping from a monocular sequence". In: *IEEE Int. Conf. Intell. Robot. Syst.* 2015-Decem (2015), pp. 5686–5693. ISSN: 21530866. DOI: 10.1109/IROS.2015.7354184.

[87] Jakob Engel, Vladlen Koltun and Daniel Cremers. "Direct Sparse Odometry". In: (2016). ISSN: 0162-8828. DOI: 10.1109/TPAMI.2017.2658577. arXiv: 1607.02565. URL: http://arxiv.org/abs/1607.02565.

[88] Christian Kerl, Jurgen Sturm and Daniel Cremers. "Dense visual SLAM for RGB-D cameras". In: *IEEE Int. Conf. Intell. Robot. Syst.* (2013), pp. 2100–2106. ISSN: 21530858. DOI: 10.1109/IROS.2013.6696650.

[89] Bo He et al. "A novel combined SLAM based on RBPF-SLAM and EIF-SLAM for mobile system sensing in a large scale environment". In: *Sensors* 11.11 (2011), pp. 10197–10219. ISSN: 14248220. DOI: 10.3390/s111110197.

[90] F. Auat Cheein et al. "Optimized EIF-SLAM algorithm for precision agriculture mapping based on stems detection". In: *Comput. Electron. Agric.* 78.2 (2011), pp. 195–207. ISSN: 01681699. DOI: 10.1016/j.compag.2011.07.007. URL: http://dx.doi.org/10.1016/j.compag.2011.07.007.

[91] Weizhen Zhou, Jaime Valls Miro and Gamini Dissanayake. "Information-efficient 3-D visual SLAM for unstructured domains". In: *IEEE Trans. Robot.* 24.5 (2008), pp. 1078–1087. ISSN: 15523098. DOI: 10.1109/TRO.2008.2004834.

[92] Joan Sola. *Simulataneous localization and mapping with the extended Kalman filter.* 2014. URL: https://goo.gl/fuod9F (visited on 04/11/2017).

[93] Zeyneb Kurt-Yavuz and Sirma Yavuz. "A comparison of EKF, UKF, FastSLAM2.0, and UKF-based FastSLAM algorithms". In: *INES 2012 - IEEE 16th Int. Conf. Intell. Eng. Syst. Proc.* (2012), pp. 37–43. DOI: 10.1109/INES.2012.6249866.

[94] Tim Bailey et al. "Consistency of the EKF-SLAM algorithm". In: *IEEE Int. Conf. Intell. Robot. Syst.* 1 (2006), pp. 3562–3568. ISSN: 10504729. DOI: 10.1109/IROS.2006.281644.

[95] Søren Riisgaard and Morten Rufus Blas. *SLAM for Dummies.* URL: https://goo.gl/9szuuA (visited on 04/11/2017).

[96] Thomas Whelan et al. "ElasticFusion: Dense SLAM Without A Pose Graph". In: *Robot. Sci. Syst. XI* (2015). ISSN: 2330765X. DOI: 10.15607/RSS.2015.XI.001. URL: http://www.roboticsproceedings.org/rss11/p01.pdf.

[97] Arren J. Glover et al. "FAB-MAP + RatSLAM: Appearance-based SLAM for multiple times of day". In: *Proc. - IEEE Int. Conf. Robot. Autom.* (2010), pp. 3507–3512. ISSN: 10504729. DOI: 10.1109/ROBOT.2010.5509547.

[98] Rohan Paul and Paul Newman. "FAB-MAP 3D: Topological mapping with spatial and visual appearance". In: *Proc. - IEEE Int. Conf. Robot. Autom.* (2010), pp. 2649–2656. ISSN: 10504729. DOI: 10.1109/ROBOT.2010.5509587.

[99] Mark Cummins and Paul Newman. "Highly Scalable Appearance-Only SLAM - FAB-MAP 2.0". In: *Rss* (2009), pp. 1–8. ISSN: 10504729. DOI: 10.1109/ROBOT.2008.4543473.

[100] M. Cummins and P. Newman. "FAB-MAP: Probabilistic Localization and Mapping in the Space of Appearance". In: *Int. J. Rob. Res.* 27.6 (2008), pp. 647–665. ISSN: 0278-3649. DOI: 10.1177/0278364908090961. URL: http://ijr.sagepub.com/cgi/doi/10.1177/0278364908090961.

[101] Mohamed Abouzahir et al. "FastSLAM 2.0 running on a low-cost embedded architecture". In: *2014 13th Int. Conf. Control Autom. Robot. Vision, ICARCV 2014*. December. 2014, pp. 1421–1426. ISBN: 9781479951994. DOI: 10.1109/ICARCV.2014.7064524.

[102] Kurt Konolige and Motilal Agrawal. "FrameSLAM : from Bundle Adjustment to Realtime Visual Mappping". In: *IEEE Trans. Robot.* 24.5 (2008), pp. 1–11. ISSN: 15523098. DOI: 10.1109/TRO.2008.2004832.

[103] Katrin Pirker et al. "GPSlam: Marrying Sparse Geometric and Dense Probabilistic Visual Mapping". In: *Procedings Br. Mach. Vis. Conf. 2011* (2011), pp. 115.1–115.12. DOI: 10.5244/C.25.115. URL: http://www.bmva.org/bmvc/2011/proceedings/paper115/index.html.

[104] Xinyan Yan, Vadim Indelman and Byron Boots. "Incremental sparse GP regression for continuous-time trajectory estimation and mapping". In: *Rob. Auton. Syst.* 87 (2017), pp. 120–132. ISSN: 09218890. DOI: 10.1016/j.robot.2016.10.004. arXiv: 1504.02696.

[105] Jing Dong, Byron Boots and Frank Dellaert. "Sparse Gaussian Processes for Continuous-Time Trajectory Estimation on Matrix Lie Groups". In: (2017). arXiv: 1705.06020. URL: http://arxiv.org/abs/1705.06020.

[106] Edwin Olson, John Leonard and Seth Teller. "Fast Iterative Alignment of Pose Graphs with Poor Initial Estimates". In: *ICRA(International Conf. Robot. Autom.* May. 2006, pp. 2262–2269. DOI: 10.1109/ROBOT.2006.1642040.

[107] Stefan Kohlbrecher et al. "A flexible and scalable SLAM system with full 3D motion estimation". In: *9th IEEE Int. Symp. Safety, Secur. Rescue Robot. SSRR 2011* (2011), pp. 155–160. ISSN: 2374-3247. DOI: 10.1109/SSRR.2011.6106777.

[108] Michele Pirovano. "Kinfu - an open source implementation of Kinect Fusion + case study: implementing a 3D scanner with PCL". 2012. URL: https://homes.di.unimi.it/borghese/Teaching/IntelligentSystems/Documents/PirovanoMichele-VisualReconstructionReport.pdf.

[109] S Izadi et al. "KinectFusion: real-time 3D reconstruction and interaction using a moving depth camera". In: *Proc. 24th Annu. ACM User Interface Softw. Technol. Symp. - UIST '11* (2011), pp. 559–568. DOI: 10.1145/2047196.2047270.

[110] Richard A. Newcombe et al. "KinectFusion: Real-time dense surface mapping and tracking". In: *2011 10th IEEE Int. Symp. Mix. Augment. Reality, ISMAR 2011* (2011), pp. 127–136. DOI: 10.1109/ISMAR.2011.6092378.

[111] Thomas Whelan et al. "Robust real-time visual odometry for dense RGB-D mapping". In: *Proc. - IEEE Int. Conf. Robot. Autom.* i (2013), pp. 5724–5731. DOI: 10.1109/ICRA.2013.6631400.

[112] Thomas Whelan et al. "Deformation-based loop closure for large scale dense RGB-D SLAM". In: *2013 IEEE/RSJ Int. Conf. Intell. Robot. Syst.* (2013), pp. 548–555. ISSN: 2153-0858. DOI: 10.1109/IROS.2013.6696405. URL: http://ieeexplore.ieee.org/document/6696405/.

[113] Thomas Whelan, Michael Kaess and Maurice Fallon. *Kintinuous: Spatially extended kinectfusion*. 2012. URL: https://goo.gl/G6NVLH (visited on 04/11/2017).

[114] Ludovico Russo et al. "A ROS Implementation of the Mono-Slam Algorithm". In: *CS IT-CSCP*. 2014, pp. 339–351. DOI: 10.5121/csit.2014.4131. URL: http://www.airccj.org/CSCP/vol4/csit41831.pdf.

[115] Andrew Davison et al. "MonoSLAM: real-time single camera SLAM." In: *Pattern Anal. Mach. Intell. (PAMI), IEEE Trans.* 29.6 (2007), pp. 1052–67. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2007.1049. URL: http://www.ncbi.nlm.nih.gov/pubmed/17431302.

[116] Siddharth Choudhary et al. *Multi Robot Object-based SLAM*. 2016. URL: https://goo.gl/c6XC1d (visited on 04/11/2017).

[117] Simoes Martins Joao Alexandre. "MRSLAM - Multi-Robot Simultaneous Localization and Mapping". Dissertation. Universidade de Coimbra, 2013, p. 67. URL: https://goo.gl/2yUzEa.

[118] Xun S. Zhou and Stergios I. Roumeliotis. "Multi-robot SLAM with unknown initial correspondence: The robot rendezvous case". In: *IEEE Int. Conf. Intell. Robot. Syst.* (2006), pp. 1785–1792. DOI: 10.1109/IROS.2006.282219.

[119] A. Howard. "Multi-robot Simultaneous Localization and Mapping using Particle Filters". In: *Int. J. Rob. Res.* 25.12 (2006), pp. 1243–1256. ISSN: 0278-3649. DOI: 10.1177/0278364906072250. URL: http://ijr.sagepub.com/cgi/doi/10.1177/0278364906072250.

[120] Yufeng Liu and Sebastian Thrun. "Gaussian Multi-Robot SLAM". In: *Adv. Neural Inf. Process. Syst.* (2003). URL: https://goo.gl/oiUJPb.

[121] G Pascoe et al. "NID-SLAM: Robust Monocular SLAM using Normalised Information Distance". In: *Comput. Vis. Pattern Recognit.* (2017). URL: https://goo.gl/ASUFAw.

[122] Stefan Leutenegger et al. "Keyframe-based visual-inertial odometry using nonlinear optimization". In: *Int. J. Rob. Res.* 34.3 (2015), pp. 314–334. ISSN: 0278-3649. DOI: 10.1177/0278364914554813. URL: http://journals.sagepub.com/doi/10.1177/0278364914554813.

[123] Stefan Leutenegger. "Unmanned solar airplanes - Design and Algorithms for Efficient and RobustAutonomous Operation". Dissertation. ETH Zurich, 2014. URL: https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/90524/eth-46751-02.pdf.

[124] S Leutenegger et al. "Keyframe Based Visual Inertial SLAM Using Nonlinear Optimization". In: *Proc. Robot. Sci. Syst.* (2013), p. 0. ISSN: 0278-3649. DOI: 10.1177/0278364914554813.

[125] Shichao Yang et al. "Pop-up SLAM: Semantic monocular plane SLAM for low-texture environments". In: *IEEE Int. Conf. Intell. Robot. Syst.* 2016-Novem (2016), pp. 1222–1229. ISSN: 21530866. DOI: 10.1109/IROS.2016.7759204. arXiv: 1703.07334.

[126] William Maddern et al. "Augmenting RatSLAM using FAB-MAP-based visual data association". In: *Proc. Australas. Conf. Robot. Autom.* October (2009). URL: http://www.araa.asn.au/acra/acra2009/papers/pap122s1.pdf.

[127] Michael Milford, Gordon Wyeth and David Prasser. "RatSLAM on the edge: Revealing a coherent representation from an overloaded rat brain". In: *IEEE Int. Conf. Intell. Robot. Syst.* (2006), pp. 4060–4065. DOI: 10.1109/IROS.2006.281869.

[128] Michael J. Milford, David Prasser and Gordon F. Wyeth. "Experience Mapping : Producing Spatially Continuous Environment Representations using RatSLAM". In: *Proc. Australas. Conf. Robot. Autom. 2005* (2005), pp. 1–10. URL: http://eprints.qut.edu.au/32840/.

[129] Wei Tan. "Robust Monocular SLAM in Dynamic Environments". In: *IEEE Int. Symp. Mix. Augment. Real. 2013*. 2013. DOI: 10.1109/ISMAR.2013.6671781.

[130] Juan Jose Tarrio and Sol Pedre. "Realtime edge-based visual odometry for a monocular camera". In: *Proc. IEEE Int. Conf. Comput. Vis.* 11-18-Dece (2016), pp. 702–710. ISSN: 15505499. DOI: 10.1109/ICCV.2015.87.

[131] Matia Pizzoli, Christian Forster and Davide Scaramuzza. "REMODE: Probabilistic, monocular dense reconstruction in real time". In: *Proc. - IEEE Int. Conf. Robot. Autom.* (2014), pp. 2609–2616. ISSN: 10504729. DOI: 10.1109/ICRA.2014.6907233.

[132] Saurav Agarwal, Vikram Shree and Suman Chakravorty. "RFM-SLAM: Exploiting Relative Feature Measurements to Separate Orientation and Position Estimation in SLAM". In: (2016). arXiv: 1609.05235. URL: http://arxiv.org/abs/1609.05235.

[133] Felix Endres et al. “3D Mapping with an RGB-D Camera”. In: *IEEE Trans. Robot.* Vol. 30. 1. 2012, pp. 1–11. DOI: 10.1109/TRO.2013.2279412.

[134] F Endres et al. “An evaluation of the RGB-D SLAM system”. In: *IEEE Int. Conf. Robot. Autom.* Vol. 3. c. 2012, pp. 1691–1696. ISBN: 9781467314046. DOI: 10.1109/ICRA.2012.6225199.

[135] Haomin Liu, Guofeng Zhang and Hujun Bao. “Robust Keyframe-based Monocular SLAM for Augmented Reality”. In: *IEEE Int. Symp. Mix. Augment. Real. Robust.* 2016. ISBN: 9781509036417. DOI: 10.1109/ISMAR.2016.24.

[136] Michael Bloesch et al. “Robust visual inertial odometry using a direct EKF-based approach”. In: *IEEE Int. Conf. Intell. Robot. Syst.* 2015-Decem (2015), pp. 298–304. ISSN: 21530866. DOI: 10.1109/IROS.2015.7353389.

[137] Christopher Mei et al. “RSLAM: A system for large-scale mapping in constant-time using stereo”. In: *Int. J. Comput. Vis.* 94.2 (2011), pp. 198–214. ISSN: 09205691. DOI: 10.1007/s11263-010-0361-7.

[138] Hauke Strasdat et al. “Double Window Optimisation for Constant Time Visual SLAM”. In: *Int. Conf. Comput. Vis.* 2011, pp. 2352–2359. URL: http://ieeexplore.ieee.org/document/6126517.

[139] A. Torres-Gonzalez, J. R. Martinez-De Dios and A. Ollero. “Efficient robot-sensor network distributed SEIF range-only SLAM”. In: *Proc. - IEEE Int. Conf. Robot. Autom.* (2014), pp. 1319–1326. ISSN: 10504729. DOI: 10.1109/ICRA.2014.6907023.

[140] M. R. Walter, R. M. Eustice and J. J. Leonard. “Exactly Sparse Extended Information Filters for Feature-based SLAM”. In: *Int. J. Rob. Res.* 26.4 (2007), pp. 335–359. ISSN: 0278-3649. DOI: 10.1177/0278364906075026. URL: http://ijr.sagepub.com/cgi/doi/10.1177/0278364906075026.

[141] Dongdong Bai et al. “CNN Feature boosted SeqSLAM for Real-Time Loop Closure Detection”. In: (2017). arXiv: 1704.05016. URL: http://arxiv.org/abs/1704.05016.

[142] N Sünderhauf, Peer Neubert and Peter Protzel. “Are we there yet? Challenging SeqSLAM on a 3000 km journey across all four seasons”. In: *Int. Conf. Robot. Autom.* (2013), pp. 1–3. URL: http://www.tu-chemnitz.de/etit/proaut/rsrc/openseqslam.pdf.

[143] Michael J. Milford and Gordon F. Wyeth. “SeqSLAM: Visual route-based navigation for sunny summer days and stormy winter nights”. In: *Proc. - IEEE Int. Conf. Robot. Autom.* (2012), pp. 1643–1649. ISSN: 10504729. DOI: 10.1109/ICRA.2012.6224623.

[144] Sayem Mohammad Siam and Hong Zhang. “Fast-SeqSLAM: A Fast Appearance Based Place Recognition Algorithm”. In: *2017 IEEE Int. Conf. Robot. Autom.* (2017), pp. 5702–5708. ISSN: 10504729. DOI: 10.1109/ICRA.2017.7989671. URL: http://ieeexplore.ieee.org/document/7989671/.

[145] Renato F Salas-moreno et al. “SLAM ++ : Simultaneous Localisation and Mapping at the Level of Objects”. 2013. URL: https://goo.gl/kyEqRj.

[146] Nicola Fioraio and Luigi Di Stefano. "SlamDunk: Affordable Real-Time RGB-D SLAM". In: *Comput. Vis. - ECCV*. Zurich: Springer International Publishing, 2014, pp. 401–414. URL: https://doi.org/10.1007/978-3-319-16178-5{_}28.

[147] Christian Forster et al. "SVO: Semidirect Visual Odometry for Monocular and Multicamera Systems". In: *IEEE Trans. Robot.* 33.2 (2017), pp. 249–265. ISSN: 15523098. DOI: 10.1109/TRO.2016.2623335. arXiv: 1204.3968.

[148] Christian Forster, Matia Pizzoli and Davide Scaramuzza. "SVO: Fast semi-direct monocular visual odometry". In: *Proc. - IEEE Int. Conf. Robot. Autom.* May. 2014, pp. 15–22. DOI: 10.1109/ICRA.2014.6906584.

[149] Meng Wu and Jian Yao. "Adaptive UKF-SLAM based on magnetic gradient inversion method for underwater navigation". In: *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)* 9245 (2015), pp. 237–247. ISSN: 16113349. DOI: 10.1007/978-3-319-22876-1_21.

[150] Hongjian Wang et al. "An adaptive UKF based SLAM method for unmanned underwater vehicle". In: *Math. Probl. Eng.* 2013 (2013). ISSN: 1024123X. DOI: 10.1155/2013/605981.

[151] Guoquan P. Huang, Anastasios I. Mourikis and Stergios I. Roumeliotis. "On the complexity and consistency of UKF-based SLAM". In: *Proc. - IEEE Int. Conf. Robot. Autom.* (2009), pp. 4401–4408. ISSN: 10504729. DOI: 10.1109/ROBOT.2009.5152793.

[152] Niklas Karlsson et al. "The vSLAM algorithm for robust localization and mapping". In: *Proc. - IEEE Int. Conf. Robot. Autom.* 2005 (2005), pp. 24–29. ISSN: 10504729. DOI: 10.1109/ROBOT.2005.1570091.

[153] Pedro F Felzenszwalb and Daniel P Huttenlocher. "Efficient Graph-Based Image Segmentation". In: *Int. J. Comput. Vis.* 59.2 (2004), pp. 167–181. DOI: 10.1023/B:VISI.0000022288.19776.77.

A Appendix

A.1 List of found SLAM/VO implementations

Table 8: List of SLAM / VO algorithms

Name	References	Code
AprilSLAM	[69] (2016), [70] (2011)	github.com/ProjectArtemis/aprilslam
ARM SLAM	[71] (2016)	-
BatSLAM	[72] (2015), [73] (2013)	-
BundleFusion	[74] (2011)	github.com/niessner/BundleFusion
CD SLAM	[75] (2011), [76] (2010)	-
C-KLAM	[77] (2014)	-
CNN SLAM	[78] (2017)	-
COP SLAM	[79] (2015), [80] (2013), [81] (2010)	-
CoSLAM	[82] (2013)	github.com/danping/CoSLAM
DolphinSLAM	[83] (2016), [44] (2015)	github.com/dolphin-slam
DP SLAM	[84] (2004), [85] (2003)	users.cs.duke.edu/p̃arr/dpslam
DPPTAM	[86] (2015)	github.com/alejocb/dpptam
DSO	[87] (2016)	github.com/JakobEngel/dso
DT SLAM	[68] (2014)	github.com/plumonito/dtslam
DTAM	[19] (2011)	github.com/anuranbaka/OpenDTAM
DVO	[88] (2013)	github.com/tum-vision/dvo_slam
EIF SLAM	[89] (2011), [90] (2011), [91] (2008)	-
EKF SLAM	[9] (2015), [92] (2014), [93] (2012) [10] (2008), [94] (2006), [6] (2006) [95] (2004), [5] (2002)	-
ElasticFusion	[96] (2015)	github.com/mp3guy/ElasticFusion
FAB-MAP	[32] (2012), [97] (2010), [98] (2010) [99] (2009), [100] (2008)	github.com/arrenglover/openfabmap
FastSLAM	[101] (2014) [7] (2013), [93] (2012), [13] (2004), [12] (2003), [11] (2002)	github.com/bushuhui/fastslam
FrameSLAM	[102] (2008)	-
GPSLAM	[103] (2011)	-
GP-SLAM	[104] (2017), [105] (2017)	github.com/gtrll/gpslam
Graph SLAM	[16] (2010), [106] (2006), [17] (2006)	-
Hector SLAM	[107] (2011)	github.com/tu-darmstadt-ros-pkg/hector_slam
KinectFusion	[108] (2012), [109] (2011), [110] (2011)	github.com/PointCloudLibrary/pcl
Kintinious	[111] (2013), [112] (2013), [113] (2012)	github.com/mp3guy/Kintinuous
LSD SLAM	[27] (2014), [28] (2013)	github.com/tum-vision/lsd_slam
MonoSLAM	[114] (2014), [115] (2007)	github.com/rrg-polito/mono-slam
MR SLAM	[116] (2016), [117] (2013), [118] (2006), [119] (2006), [120] (2003)	-

NID SLAM	[121] (2017)	-
OKVIS	[122] (2015), [123] (2014), [124] (2013)	github.com/ethz-asl/okvis_ros
ORB SLAM	[4] (2017), [3] (2016), [2] (2015)	github.com/raulmur/ORB_SLAM2
Pop-up SLAM	[125] (2016)	github.com/shichaoy/pop_up_image
PTAM	[20] (2007)	github.com/Oxford-PTAM/PTAM-GPL
RatSLAM	[23] (2013), [126] (2009), [24] (2008), [127] (2006), [128] (2005), [21] (2004)	github.com/davidmball/ratslam
RD SLAM	[129] (2013)	-
REBVO	[130] (2016)	github.com/JuanTarrio/rebvo
REMODE	[131] (2014)	github.com/uzh-rpg/rpg_open_remode
RFM SLAM	[132] (2016)	github.com/sauravag/edpl-rfmslam
RGB-D SLAM	[133] (2012) [134] (2012)	github.com/felixendres/rgbdslam_v2
RKSLAM	[135] (2016)	zjucvg.net/rkslam/rkslam.html
ROVIO	[136] (2014)	github.com/ethz-asl/rovio
RSLAM	[137] (2011)	-
ScaViSLAM	[138] (2011)	github.com/strasdat/ScaViSLAM
SEIF SLAM	[139] (2014), [140] (2007)	-
SeqSLAM	[141] (2017), [142] (2013), [143] (2012) [144] (2017)	github.com/subokita/OpenSeqSLAM github.com/siam1251/Fast-SeqSLAM
SLAM++	[145] (2013)	-
SlamDunk	[146] (2015)	github.com/m4nh/skimap_ros
SVO	[147] (2017), [148] (2014)	github.com/uzh-rpg/rpg_svo
UKF SLAM	[149] (2015), [150] (2014), [151] (2009)	-
vSLAM	[152] (2005)	wiki.ros.org/vslam

A.2 Overview of non-filtering, monocular SLAM/VO implementations

Table 9: Non-filtering, monocular graph-based SLAM/VO approaches

Name	Characteristics	
CD-SLAM	*Front-end:*	Histogram of Oriented Cameras (HoC) descriptor [76]
	Back-end:	Keyframe-based BA, graph optimization
	Loop closing:	Yes, using FAB-MAP
	Code:	-
	Refs(year):	[75](2011)
	Notes:	Focus on highly dynamic environments, Keep map proportional to explored space not time, Usage of HoC descriptor in feature based front end
C-KLAM	*Front-end:*	SIFT features
	Back-end:	Custom to C-KLAM, based on BA
	Loop closing:	Possible but was not achieved in test
	Code:	-
	Refs(year):	[77](2014)
	Notes:	Focus on making use of data between keyframes, Incorporates IMU data in graph optimization
CNN SLAM	*Front-end:*	Dense approach based on LSD SLAM
	Back-end:	$Sim(3)$ optimization and BA
	Loop closing:	Not mentioned in paper
	Code:	-
	Refs(year):	[78](2017)
	Notes:	Uses trained CNN to predict depth maps, Can be used to correct scale drift, Relies on use of GPU, Implements semantic labeling (i.e. distinguish walls and floor)
COP SLAM	*Front-end:*	Any front-end that creates pose-graphs
	Back-end:	Optimizing pose-chains using trajectory bending[81]
	Loop closing:	Is a back-end $\rightarrow$ no detection, only optimization
	Code:	-
	Refs(year):	[80](2013), [79](2015)
	Notes:	Optimizes sparse pose-graph (pose-chain), 50 to 200 times faster than G^2O, Only optimizes the robot pose, not the map, Has an extension which can account for scale-drift

Dolphin-SLAM	*Front-end:*	Sensor dependant, Sonar: HU moments, Image: SURF
	Back-end:	RatSLAM back-end (not really graph-based)
	Loop closing:	Yes, using FAB-MAP
	Code:	`https://github.com/dolphin-slam/dolphin_slam`
	Refs(year):	[44](2015), [83](2016)
	Notes:	Focus on SLAM in underwater scenario, Multiple sensors: sonar, camera, IMU and DVL, Based on RatSLAM
DP-PTAM	*Front-end:*	Piecewise planar dense
	Back-end:	Semi-dense map with estimation of planar surfaces, map optimization not mentioned
	Loop closing:	Not mentioned in paper
	Code:	`https://github.com/alejocb/dpptam`
	Refs(year):	[86](2015)
	Notes:	Reconstruction of dense maps using only CPU, Reduced cost due to planar surface estimation via superpixels [153] with the assumption that low color-gradient areas are mostly planar
DSO	*Front-end:*	Sparse and direct
	Back-end:	None (odometry only)
	Loop closing:	No
	Code:	`https://github.com/JakobEngel/dso`
	Refs(year):	[86](2015)
	Notes:	Optimizes camera intrinsics and extrinsics, Works well in low textured areas, Distributes sampled pixels such that, when available, high gradients are used, otherwise takes weak gradients
DTAM	*Front-end:*	Dense method
	Back-end:	Creates dense map, no graph optimization
	Loop closing:	No
	Code:	`https://github.com/anuranbaka/OpenDTAM`
	Refs(year):	[19](2011)
	Notes:	Relies on GPU for computation, Creates feature-rich, textured, dense map, Robust against quick movement and camera defocus

DT SLAM	*Front-end:*	FAST, separated 2D and 3D feature matching
	Back-end:	BA on pose and features of all sub maps
	Loop closing:	Yes
	Code:	`https://github.com/plumonito/dtslam`
	Refs(year):	[68](2014)
	Notes:	Holds off 3D feature triangulation until enough parallax is observed (Deferred triangulation), Can incorporate purely rotational movement frames, Creates sub-maps which can be merged later to avoid scale inconsistencies
FAB-MAP	*Front-end:*	Appearance-based bag of words approach with SURF
	Back-end:	Place recognition in appearance-space, no metric map
	Loop closing:	Yes
	Code:	`https://github.com/arrenglover/openfabmap`
	Refs(year):	[100](2008), [99](2009), [32](2012)
	Notes:	Needs training on an environment similar to the one it is going to be used in, Place recognition via visual bag of words database, No metric map creation, mostly used as a tool for loop closure detection in other approaches like LSD SLAM
LSD SLAM	*Front-end:*	Semi-dense
	Back-end:	Creates semi-dense map, pose-graph optimization (g2o)
	Loop closing:	Yes, small via $sim(3)$ and large via FAB-MAP
	Code:	`https://github.com/tum-vision/lsd_slam`
	Refs(year):	[28](2013), [27](2014)
	Notes:	Semi-dense, keyframe-based, runs on CPU, Makes use of Lie-algebra for tracking and optimization, Scale-drift aware
NID SLAM	*Front-end:*	Semi-dense, NID metric
	Back-end:	Creates semi-dense map, pose-graph optimization (g2o)
	Loop closing:	Yes, using FAB-MAP
	Code:	-
	Refs(year):	[121](2017)
	Notes:	Focus on lighting, weather and structural changes, Relies on GPU for computation, Scale-drift aware (based on LSD SLAM)

ORB SLAM	*Front-end:*	Indirect using ORB feature descriptor
	Back-end:	BA on sub-graphs, uses local maps
	Loop closing:	Yes, using a place recognition database
	Code:	`https://github.com/raulmur/ORB_SLAM`
	Refs(year):	[2](2015)
	Notes:	Uses three parallel threads for tracking, local map creation and loop closing, Uses a bag of words approach for place recognition, Focus on real-time operation, runs on CPU, Focus on long-term localization, not detailed maps
ORB SLAM2!	*Front-end:*	Indirect using ORB feature descriptor
	Back-end:	BA on sub-graphs, uses local maps
	Loop closing:	Yes, using a place recognition database
	Code:	`https://github.com/raulmur/ORB_SLAM2`
	Refs(year):	[3](2016)
	Notes:	Extension of ORB SLAM for RGB-D and stereo camera setups
Visual Inertial ORB SLAM	*Front-end:*	Indirect using ORB feature descriptor
	Back-end:	BA on sub-graphs, uses local maps
	Loop closing:	Yes, using a place recognition database
	Code:	-
	Refs(year):	[4](2017)
	Notes:	Modular extension of ORB SLAM for incorporating IMU readings, Takes gyroscope and accelerometer bias into account, Derives scale from IMU measurements
PTAM	*Front-end:*	Indirect, using FAST corner detector
	Back-end:	Local and global BA
	Loop closing:	No
	Code:	`https://github.com/Oxford-PTAM/PTAM-GPL`
	Refs(year):	[20](2007)
	Notes:	First to parallelize tracking and mapping, Uses keyframes to create map, Optimizes map in background when exploring already known areas

Rat-SLAM	*Front-end:*	Not specified
	Back-end:	Neural network based on the hippocampus of rodents
	Loop closing:	Yes, via local view cells
	Code:	`https://github.com/davidmball/ratslam`
	Refs(year):	[21](2004), [128](2005), [127](2006), [24](2008), [23](2013)
	Notes:	Back-end that can make use of different input data both visual and internal (IMU), Composed of pose cells, local view cells, and experience map
RD SLAM	*Front-end:*	Indirect, using SIFT
	Back-end:	BA based on PTAM
	Loop closing:	No
	Code:	`http://www.zjucvg.net/rdslam/rdslam.html`
	Refs(year):	[129](2013)
	Notes:	Based on PTAM, Focus on dynamic environments with changes in structure and illumination, According to author still fails frequently, Relies on GPU due to SIFT features
REBVO	*Front-end:*	Between direct and indirect, using edges as features
	Back-end:	None (odometry only)
	Loop closing:	No
	Code:	`https://github.com/JuanTarrio/rebvo`
	Refs(year):	[130](2016)
	Notes:	Tracks edges instead of features or pixels, Focus on running on embedded devices, Has extension for IMU integration, Depth estimated via EKF
RE-MODE	*Front-end:*	Direct, dense method, bayesian estimation for depth
	Back-end:	Creates dense map, no graph optimization
	Loop closing:	No
	Code:	`https://github.com/uzh-rpg/rpg_open_remode`
	Refs(year):	[131](2014)
	Notes:	Depth estimation via a bayesian scheme, Can smooth camera measurement noise, Relies on GPU for processing

RFM SLAM	*Front-end:*	Indirect, but not specified which features
	Back-end:	Splits orientation and position estimation for 2D case
	Loop closing:	Yes
	Code:	-
	Refs(year):	[132](2016)
	Notes:	Separates orientation and position estimation, 2D case can be extended to 3D, Focus on computationally cheaper pose optimization
RK SLAM	*Front-end:*	Indirect, FAST, uses homographies for tracking
	Back-end:	Local and global optimization using BA
	Loop closing:	Yes
	Code:	`http://www.zjucvg.net/rkslam/rkslam.html`
	Refs(year):	[135](2016)
	Notes:	Focus on fast motion and rotation, Extracts and matches 3D planes in keyframes, Has extension to incorporate IMU data
Seq SLAM	*Front-end:*	Looks for minimum in difference of image sequences
	Back-end:	No map creation or localization, only place recognition
	Loop closing:	Yes
	Code:	`https://github.com/subokita/OpenSeqSLAM`
	Refs(year):	[143](2012)
	Notes:	Focus on extreme changes in environment, Like FAB-MAP no real SLAM approach which localizes a robot and builds a map, rather used for loop closure
SVO	*Front-end:*	Semi-direct, dense pixel batches and FAST
	Back-end:	BA for pose, keeps a small map of fixed size
	Loop closing:	No
	Code:	`https://github.com/uzh-rpg/rpg_svo`
	Refs(year):	[148](2014), [147](2017)
	Notes:	Focus on runtime, runs on embedded devices, Comes with two settings: *fast* and *accurate*, Fixed number of keyframes in map, Needs high FPS $\sim$ 60 FPS

A.3 ORB SLAM code diagram explanation

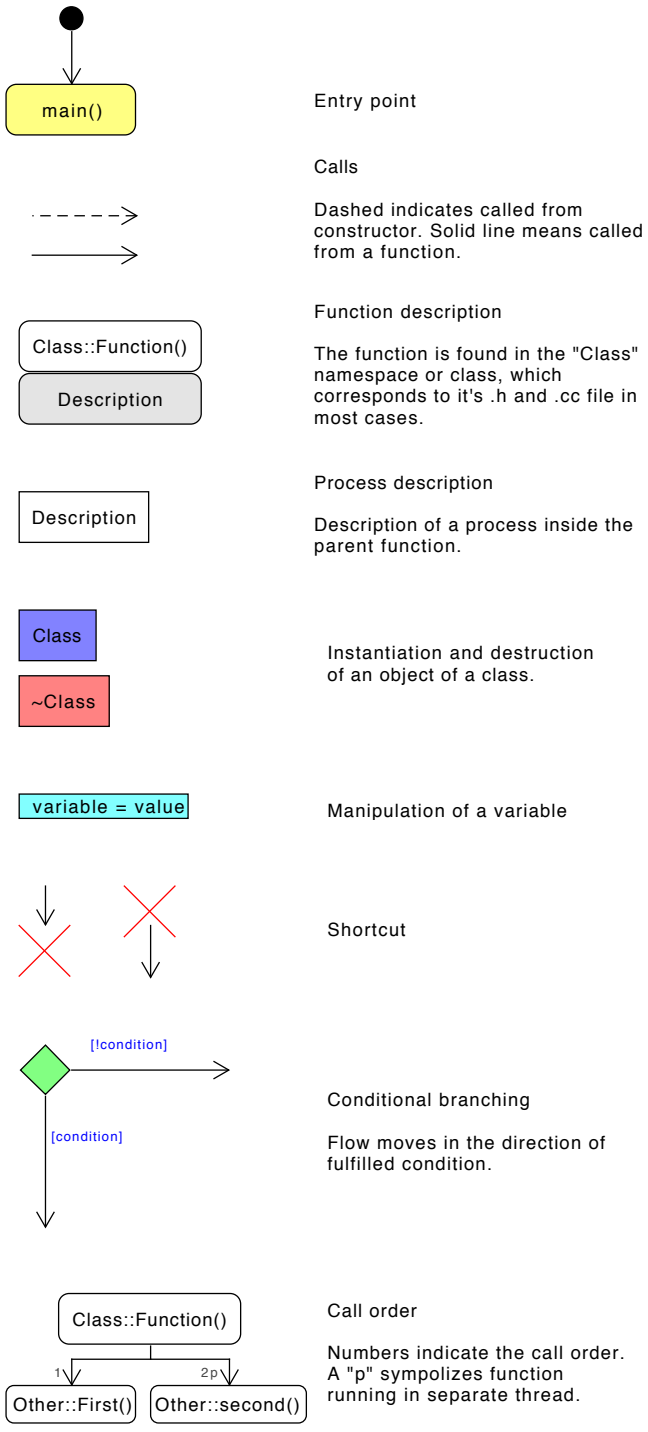


Figure A.1: Legend for diagrams on the following pages.

A.4 ORB SLAM code diagram

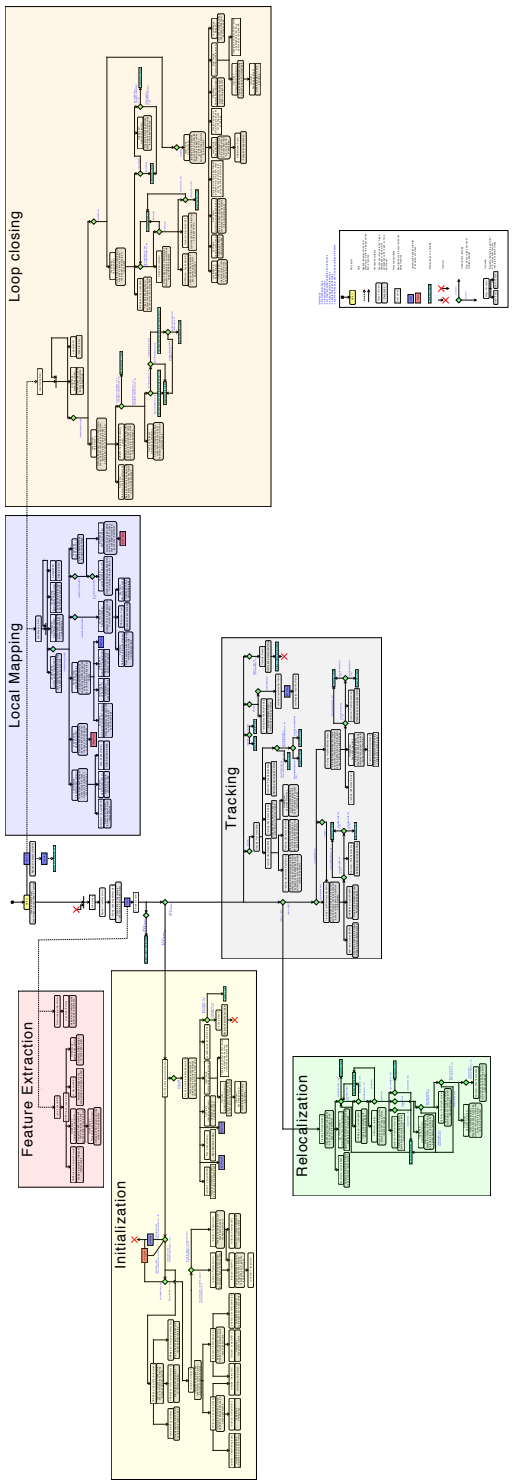


Figure A.2: ORB SLAM code structure diagram. The colored blocks resemble the individual code modules.

For original size visit: `https://raw.githubusercontent.com/kafendt/ORB_SLAM2_Accessible/master/docs/ORB_Code_activity.pdf`

A.5 ORB SLAM feature extraction diagram

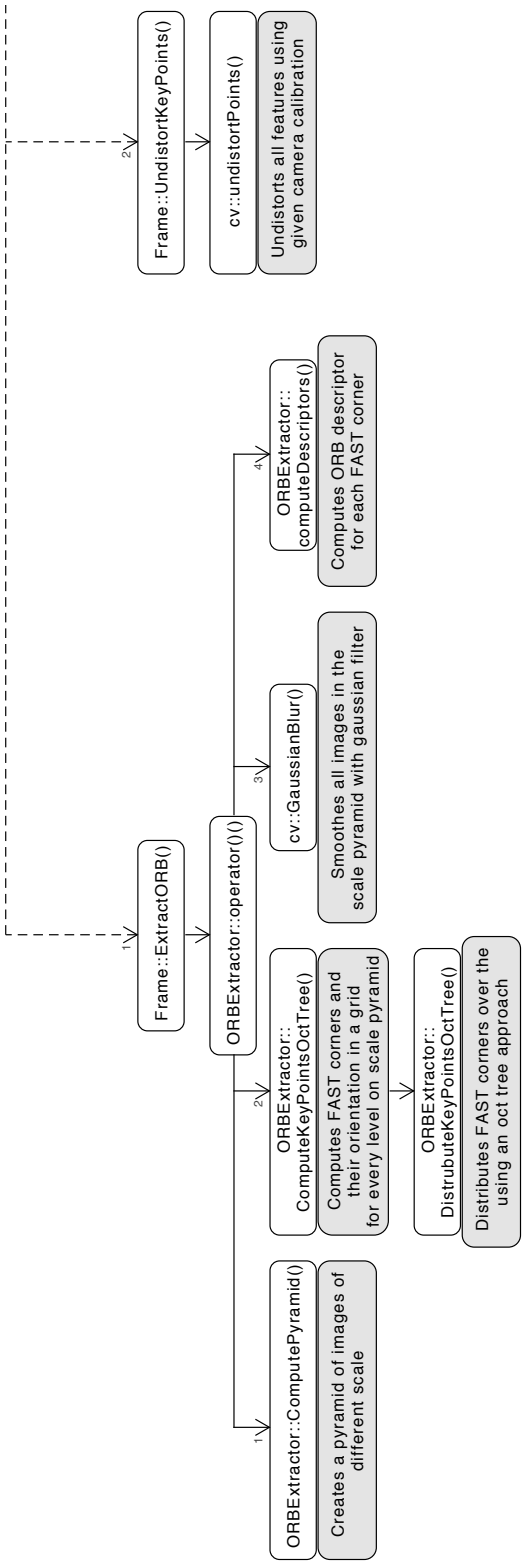


Figure A.3: ORB SLAM feature extraction module diagram

A.6 ORB SLAM initialization diagram (part 1)

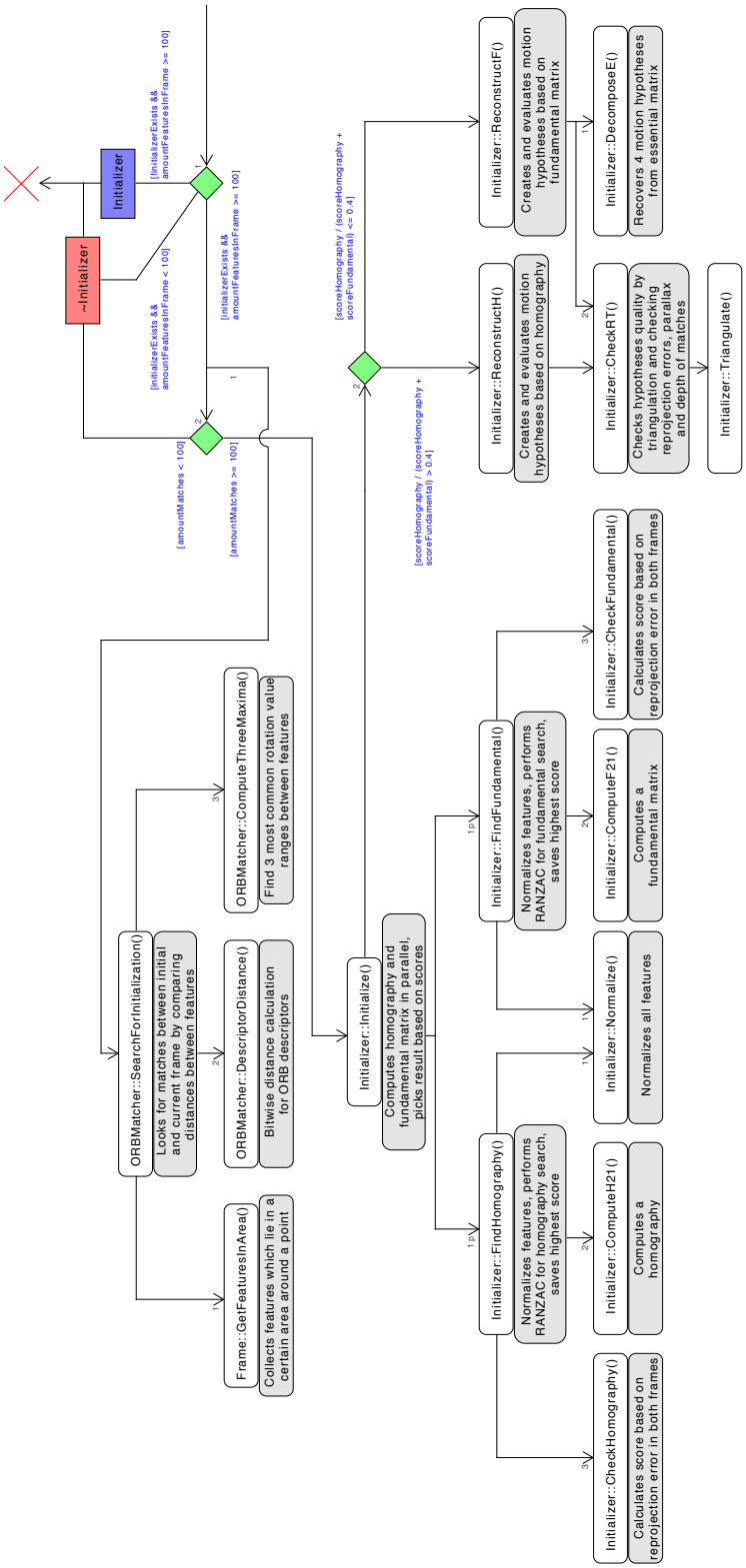


Figure A.4: ORB SLAM initialization module diagram (part 1)

A.7 ORB SLAM initialization diagram (part 2)

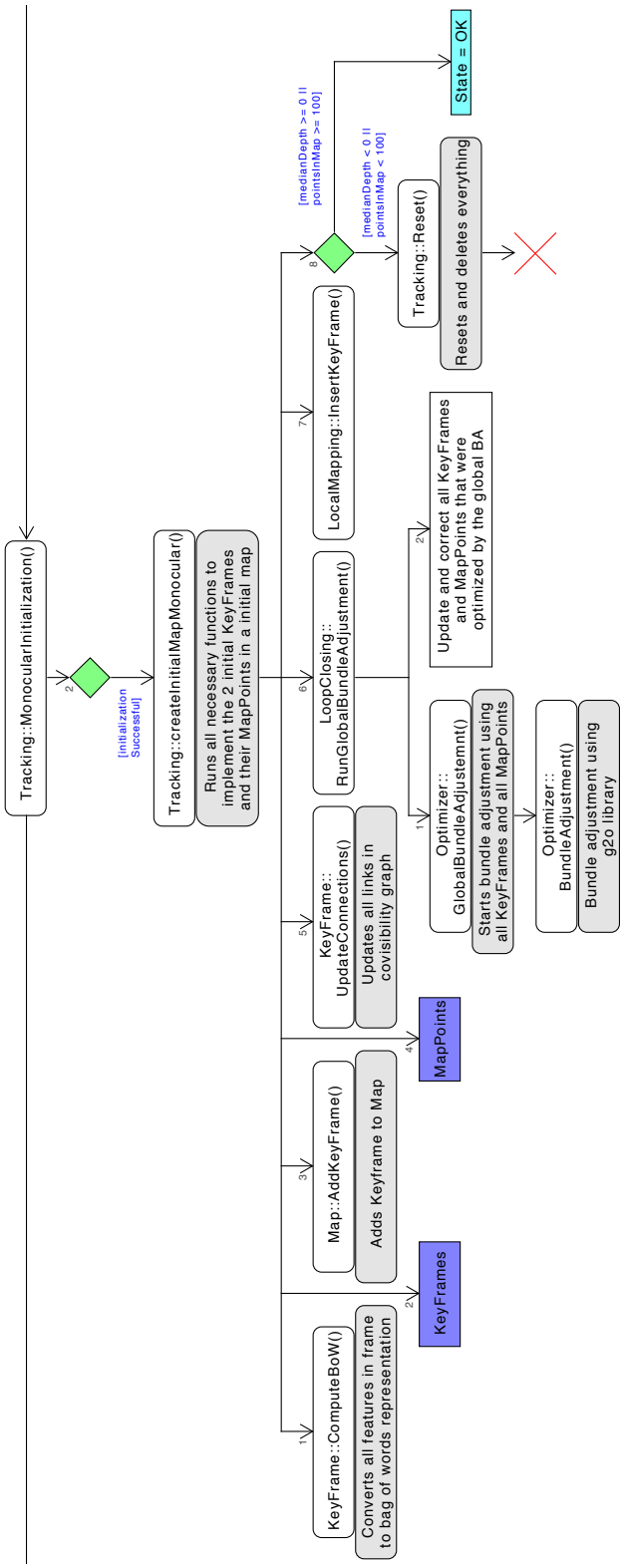


Figure A.5: ORB SLAM initialization module diagram (part 2)

A.8 ORB SLAM tracking diagram (part 1)

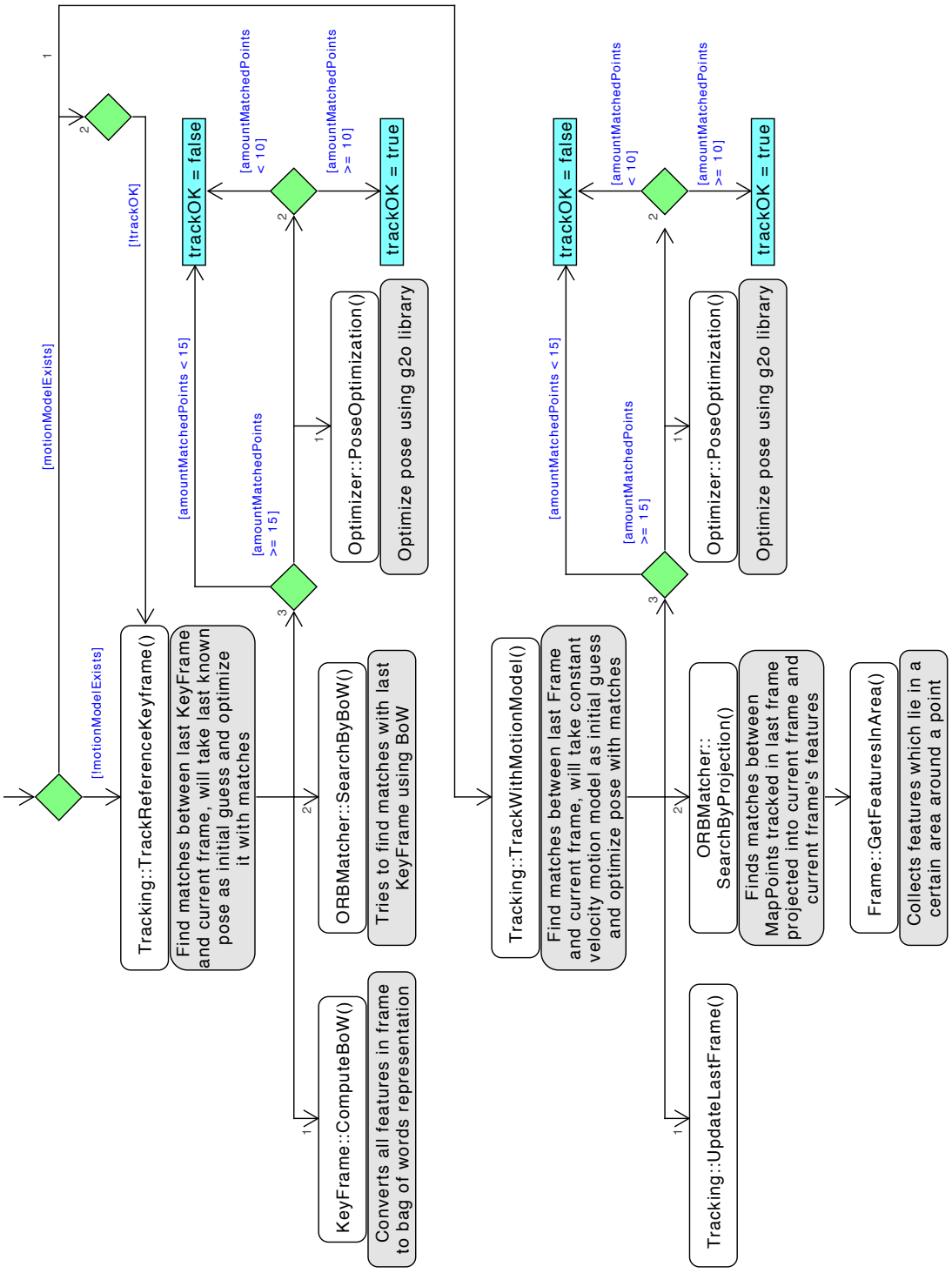


Figure A.6: ORB SLAM tracking module diagram (part 1)

A.9 ORB SLAM tracking diagram (part 2)

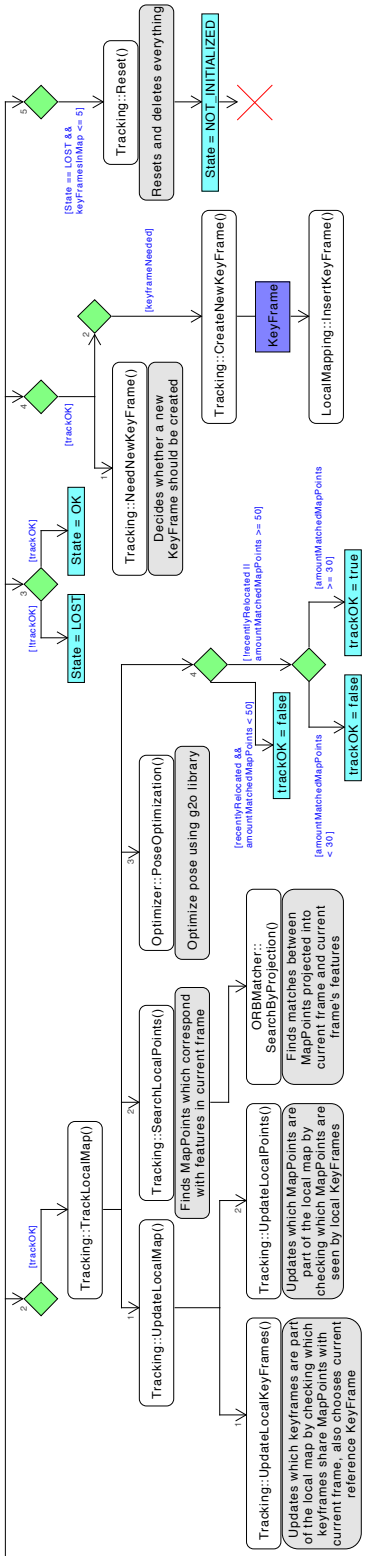


Figure A.7: ORB SLAM tracking module diagram (part 2)

A.10 ORB SLAM relocalization diagram

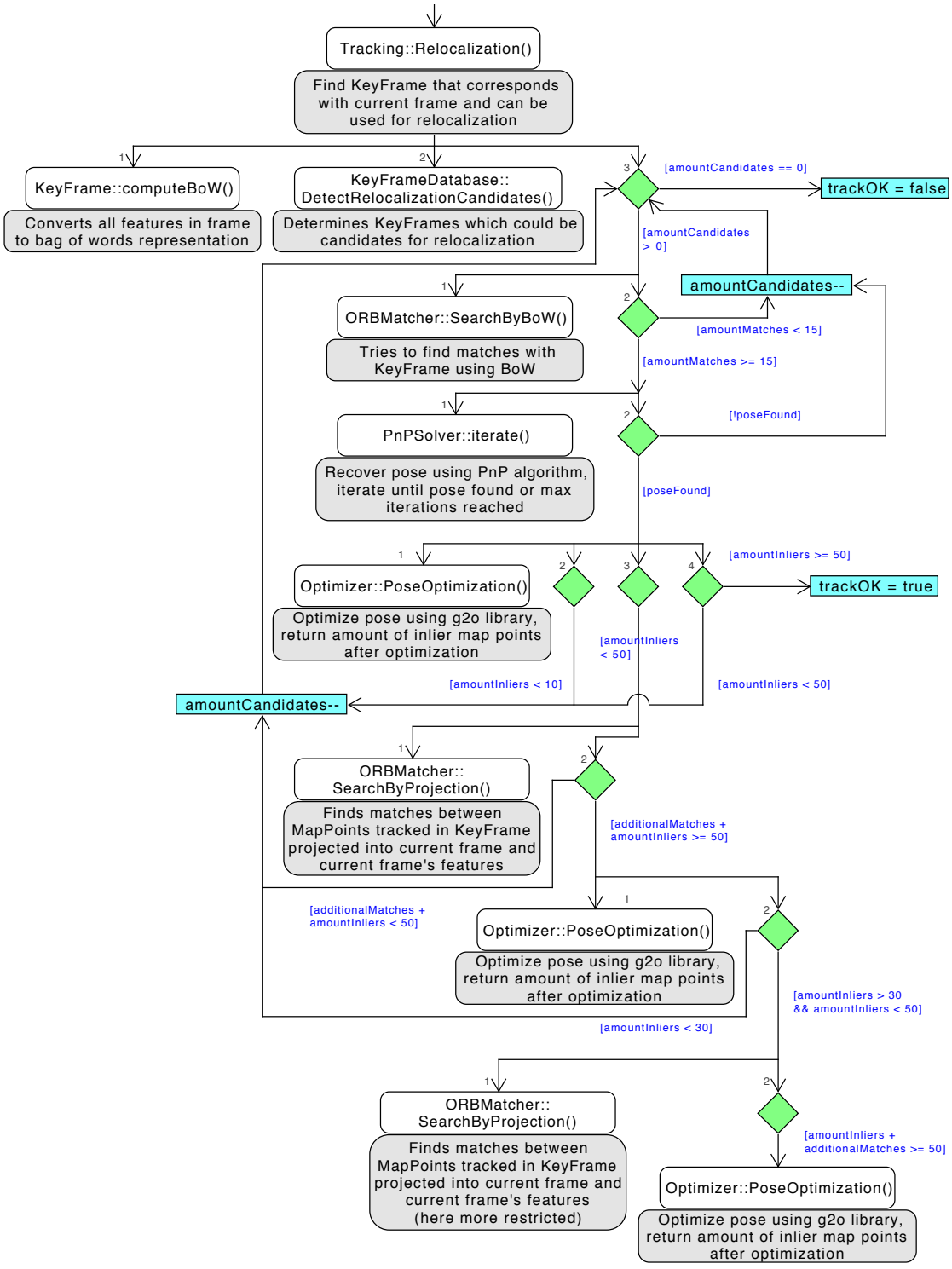


Figure A.8: ORB SLAM relocalization module diagram

A.11 ORB SLAM local mapping diagram

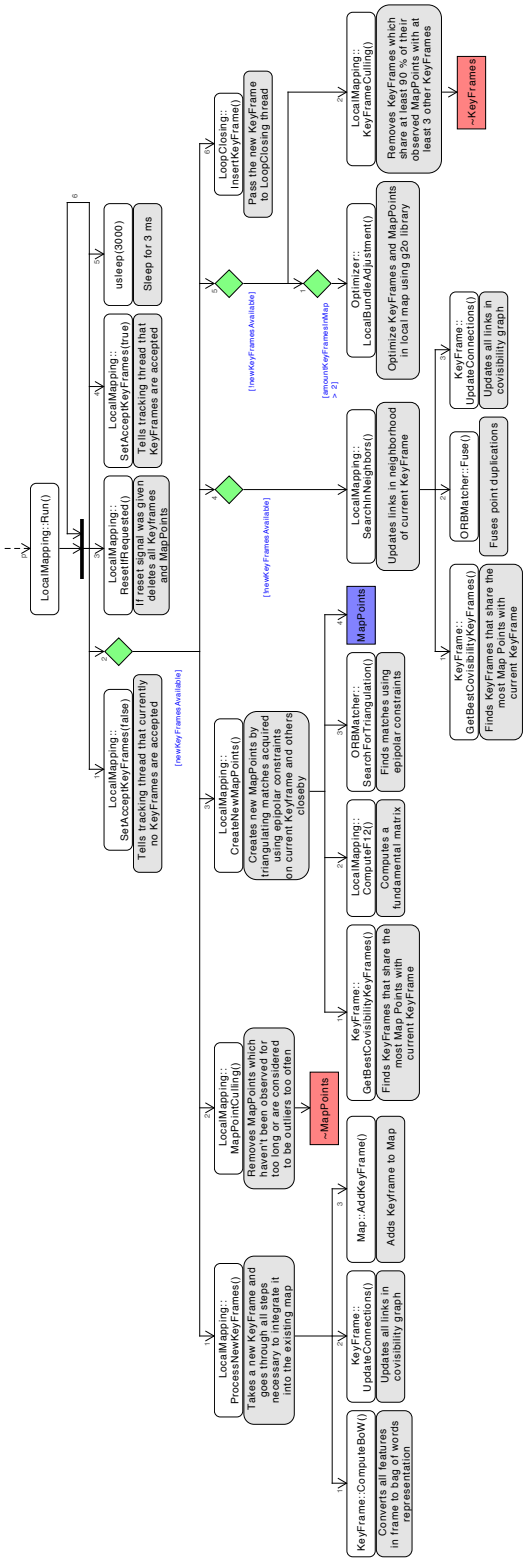


Figure A.9: ORB SLAM local mapping module diagram

A.12 ORB SLAM loop closing diagram (part 1)

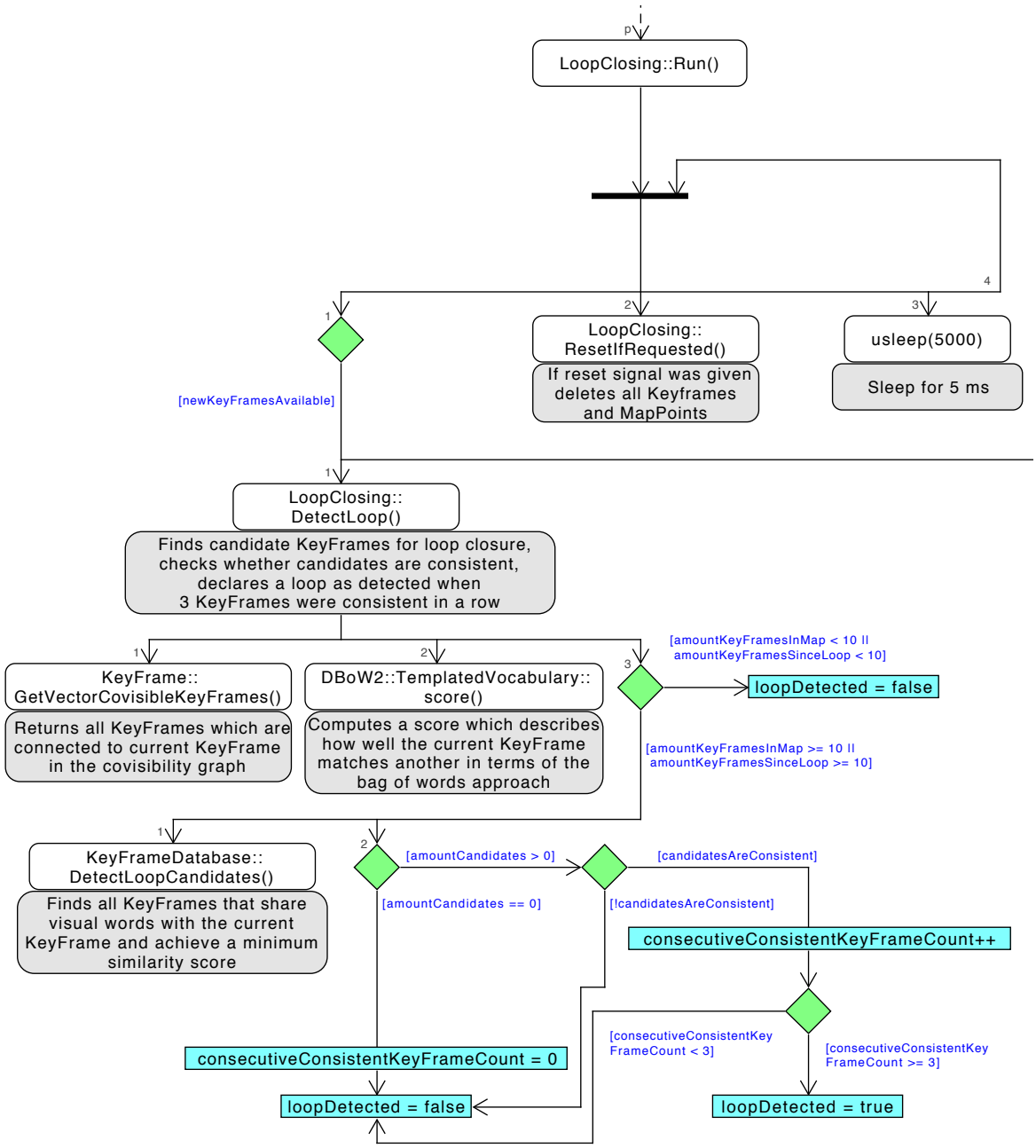


Figure A.10: ORB SLAM loop closing module diagram (part 1)

A.13 ORB SLAM loop closing diagram (part 2)

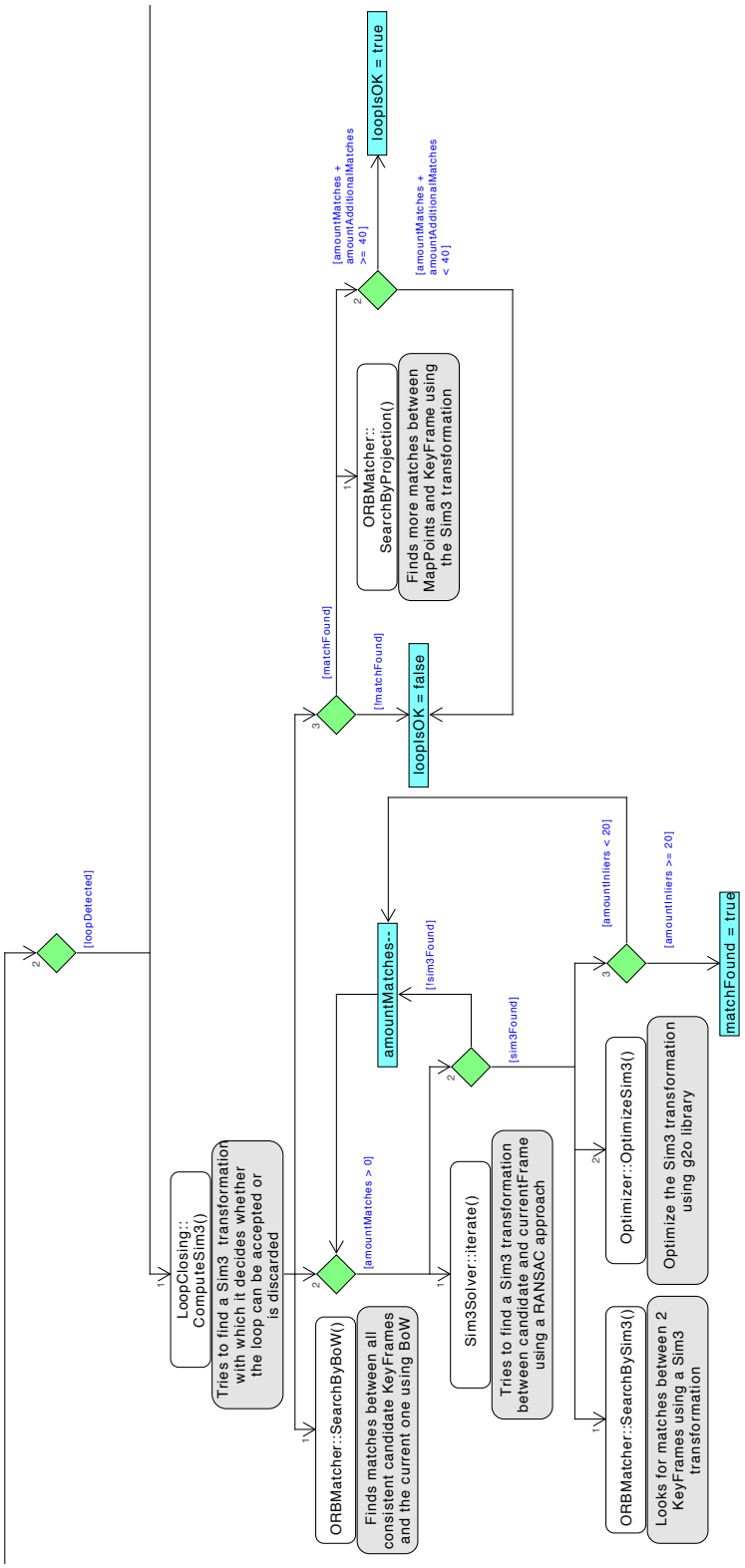


Figure A.11: ORB SLAM loop closing module diagram (part 2)

A.14 ORB SLAM loop closing diagram (part 3)

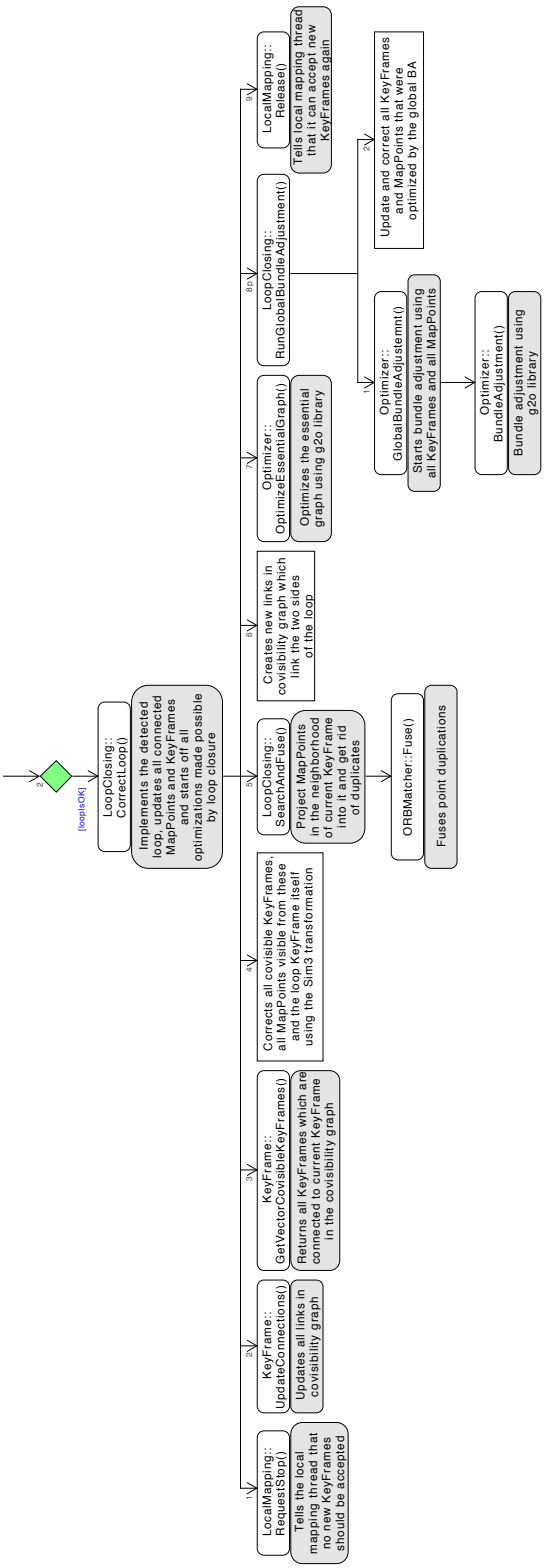


Figure A.12: ORB SLAM loop closing module diagram (part 3)

A.15 Excerpt of a JSON file created during recording.


```
...
{
    "timestamp": 1504782803806,
    "image": "Image000033.jpg",
    "imu_data": [
        "RAW_IMU {time_usec :3387046337, xacc :142, yacc : -22, zacc : -996, xgyro : 5, ygyro : 27, zgyro : 2, xmag : -78, ymag :-230, zmag :-549}",
        "SCALED_IMU2 {time_boot_ms :3387046, xacc :123, yacc :-32, zacc : -999, xgyro : 4, ygyro : 0, zgyro : -70, xmag :0, ymag : 0, zmag : 0}",
        "SCALED_PRESSURE {time_boot_ms :3387046, press_abs :1095.18615723, press_diff : 79.0070266724, temperature :5727}",
        "SCALED_PRESSURE2 {time_boot_ms :3387046, press_abs :1277.70275879, press_diff :264.452728271, temperature :1905}"
    ]
},
{
    "timestamp": 1504782803779,
    "image": "Image000034.jpg",
    "imu_data": [
        "SERVO_OUTPUT_RAW {time_usec :3387086678, port :0, servo1_raw :1525, servo2_raw :1474, servo3_raw :1525, servo4_raw : 1474, servo5_raw : 1509, servo6_raw : 1499, servo7_raw : 1100, servo8_raw : 1800, servo9_raw : 0, servo10_raw : 0, servo11_raw : 0, servo12_raw : 0, servo13_raw : 0, servo14_raw : 0, servo15_raw : 0, servo16_raw : 0}",
        "RC_CHANNELS_RAW {time_boot_ms :3387086, port :0, chan1_raw :1500, chan2_raw :1500, chan3_raw :1500, chan4_raw :1500, chan5_raw :1420, chan6_raw : 1500, chan7_raw : 1500, chan8_raw : 1800, rssi : 0}",
        "ATTITUDE {time_boot_ms :3387086, roll :0.0108106117696, pitch : 0.0513137206435, yaw : 2.21572089195, rollspeed : 0.00796211417764, pitchspeed : 0.0290557350963, yawspeed :0.00267279311083}",
    ]
},
...
```

A.16 Example of a ORB SLAM parameter file

```
%YAML:1.0

Camera.fx: 511.213444
Camera.fy: 510.574592
Camera.cx: 507.456071
Camera.cy: 277.696519

Camera.k1: −0.286686
Camera.k2: 0.065394
Camera.p1: 0.004120
Camera.p2: −0.000145
Camera.k3: 0.0

Camera.fps: 15

Camera.RGB: 1

ORBextractor.nFeatures: 2000

ORBextractor.scaleFactor: 1.2

ORBextractor.nLevels: 8

ORBextractor.iniThFAST: 20
ORBextractor.minThFAST: 7

Viewer.KeyFrameSize: 0.05
Viewer.KeyFrameLineWidth: 1
Viewer.GraphLineWidth: 0.9
Viewer.PointSize:2
Viewer.CameraSize: 0.08
Viewer.CameraLineWidth: 3
Viewer.ViewpointX: 0
Viewer.ViewpointY: −0.7
Viewer.ViewpointZ: −1.8
Viewer.ViewpointF: 500

ORBextractor.ExcludedRegions: [[863, 518, 1024, 768], [790, 702, 861, 768], [224, 738, 790,
    768], [68, 729, 222, 768],
[0, 472, 83, 768], [80, 663, 207, 730], [0, 134, 56, 260], [0, 16, 29, 133], [995, 2, 1024,
    352], [934, 100, 1002, 331],
[0, 255, 22, 273]]
```