Autonomous Modeling of a 3D Environment with Drones

Masterarbeit

zur Erlangung des akademischen Grades

Master of Science in Engineering (M.Sc.)

Eingereicht bei:

Fachhochschule Kufstein Tirol Bildungs GmbH

Data Science & Intelligent Analytics

Verfasser/in:

Julian Bialas, BSc

1910837917

Erstgutachter : Prof. (FH) PD Dr. Mario Döller

Zweitgutachter : Sebastian Danninger, MA

Abgabedatum:

06. July 2020

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Masterarbeit selbständig und ohne fremde Hilfe verfasst und in der Bearbeitung und Abfassung keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe. Die vorliegende Masterarbeit wurde noch nicht anderweitig für Prüfungszwecke vorgelegt.

Kufstein, 06. July 2020

Julian Bialas, BSc

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

SLAM Simulatanious Localization and Mapping

vSLAM visual SLAM

ROS Roboter Operating System

DSO Direct Sparse Odometry

DSM Direct Sparce Mapping

ORB Oriented FAST and Rotated BRIEF

RGB Red Green Blue

PTAM Parallel Tracking and Mapping

FAST Features from Accelerated and Segments Test

EuRoC European Robotics Challenge

CNN Convolutional Neural Network

TUM Technische Universität München

FH Kufstein Tirol
Data Science & Intelligent Analytics

Abstract of the thesis: **Autonomous Modeling of a 3D Environment with Drones**

Author: Julian Bialas, BSc
First reviewer: Prof. (FH) PD Dr. Mario Döller
Second reviewer: Sebastian Danninger, MA

Within the context of autonomous exploration and modeling of a 3D environment with drones, this work targets two distinct research questions. Most existing autonomous exploration frameworks divide the exploration task into three subproblems: localization, mapping, and path planning [1]. The location and mapping problem, which refers to the task of generating a local map of the environment while localizing the drone within it, is addressed by several already existing SLAM algorithms. The first examined research question to answer is what the best suited open-source visual SLAM algorithm for the exploration task is. Three algorithms were evaluated using predefined criteria addressing trajectory accuracy, point cloud accuracy and computation time. The results showed, that ORB SLAM outperforms other algorithms. The second research task was to develop a framework, that enables users to test and implement fully autonomous exploration systems within a virtual environment. The framework was developed in ROS and provides the possibility to navigate a drone within a simulation while the ORB SLAM algorithm is applied on the drone's camera output. With the help of suited transformations on the output of the ORB SLAM algorithm, ground truth data is then streamed within the framework and enables users to apply and test flight path planning algorithms in order to complete the autonomous exploration task.

06. July 2020

FH Kufstein Tirol
Data Science & Intelligent Analytics

Kurzfassung der Masterarbeit: **Autonomous Modeling of a 3D Environment with Drones**

Verfasser: Julian Bialas, BSc
Erstgutachter: Prof. (FH) PD Dr. Mario Döller
Zweitgutachter: Sebastian Danninger, MA

Im Kontext der automatisierten Erkundung und Modellierung einer 3D Umgebung, behandelt diese Arbeit zwei Forschungsfragen. Die meisten automatisierten Systeme unterteilen die automatisierte Erkundung in drei Teilprobleme: Lokalisierung, Kartierung, und Wegplanung [1]. Die Lokalisierungs- und Kartierungsaufgabe wird mit den bereits existierenden SLAM Algorithmen abgedeckt. In diesem Zusammenhang wird in der ersten Forschungsfrage untersucht, welcher open-source SLAM Algorithmus am besten geeignet ist, um eine automatisierte Umfelds Erkundung mit Drohnen durchzuführen. Hierbei werden die von den Algorithmen errechneten Flugkurven und Punktewolken sowie Rechenzeit untersucht. Die Ergebnisse haben gezeigt, dass ORB SLAM am besten geeignet ist für die oben beschriebene Aufgabe. Im zweiten Teil der Arbeit wurde analysiert, wie ein System implementiert werden kann, mithilfe dessen Nutzer ein automatisiertes Erkundungssystem in einer simulierten Umgebung entwickeln und testen können. In dem aus dieser Frage resultierende Framework können Nutzer eine Drohne, auf deren Kameradaten der ORB Algorithmus angewandt wird, innerhalb einer virtuellen Umgebung navigieren. Unter anderem der freie Zugriff auf die wahre Position der Drohne ermöglicht es Nutzern Wegplanungsalgorithmen auf die Drohne anzuwenden, um das automatisierte Erkundungssystem zu komplettieren.

06. July 2020

1. Introduction

Multiple applications exist for the autonomous exploration and mapping tasks using drones, such as search and rescue-, inspection- and surveillance operations [2].

The autonomous exploration task can be divided into three subproblems: localization, mapping, and path planning [1]. All three tasks should be performed simultaneously within an environment, which the drone has no information on a priori.

The localization task contains the estimation of the position of the drone within this environment and the mapping task refers to the incremental creation of a 3-dimensional map. There are several methods that combine these two tasks in a so-called simultaneous localization and mapping (SLAM) algorithm. The development of these SLAM algorithms is one of the most researched topics in the field of robotics [3].

> SLAM is used for many applications including mobile robotics, self-driving cars, unmanned aerial vehicles, or autonomous underwater vehicles [4].

When combining a SLAM algorithm with a path planning algorithm, an autonomous exploration system is created. The autonomous exploration using SLAM and a path planning algorithm is sometimes also referred to as active

SLAM [5]. This active SLAM process is displayed in figure 1. Details on how such a system can be initiated and terminated, can be found in section 3.3.

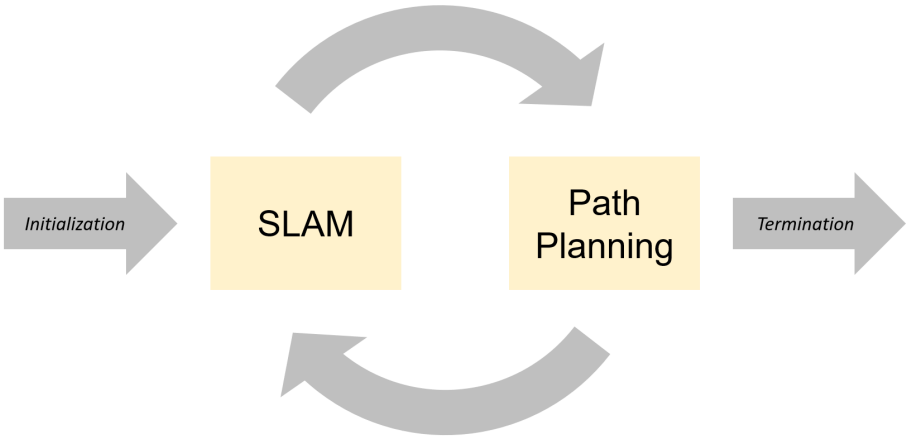


Figure 1: Automated exploration system, based on a SLAM algorithm and a path planning algorithm.

This work targets to answer two distinct research questions.

1. What is the most suitable open-source monocular visual SLAM algorithm for an exploration task?

   This work is limited to evaluate monocular visual SLAM (vSLAM) algorithms, meaning that the algorithm is only working with a single RGB (red, green, blue) camera as sensor. Therefore, since nowadays RGB cameras are either a standard on drones or can easily be upgraded, these drones are very affordable, making it highly available for a larger user group.

   Thus, in the first part of this work, three open-source monocular visual SLAM algorithms are evaluated. DSO (Direct Sparse Odometry) SLAM, DSM (Direct Sparce Mapping) SLAM and ORB (Oriented FAST and Rotated BRIEF) SLAM were investigated regarding the predefined criteria

of the accuracy of the resulting trajectory estimation, the point cloud accuracy and computational speed.

This was done by using the publicly available benchmark EuRoC dataset [6], containing video sequences filmed by a drone, the ground truth of the position of the drone and the point cloud of the environment. Thus, the three SLAM algorithms are applied on the video sequences, the resulting trajectories and map points are compared to the ground truth regarding the above-mentioned criteria.

2. What is a suitable framework to test and develop fully automated exploration systems within a simulated environment?

   In the second part of this work, a Roboter Operating System (ROS) framework that enables users to develop a fully automated exploration system within a virtual environment is proposed. This framework includes a process that provides a simulated Gazebo environment, making it possible to navigate a virtual drone within a simulated environment. The sensors and behavior of the drone are modeled realistically. Most importantly, the drone is equipped with a RGB camera, which allows the direct application of a vSLAM algorithm on the output. Furthermore, the most suitable algorithm, evaluated in the first part of the work is implemented in a subprocess of this framework.

   While the functionality and current state of the art of methods tackling the path planning task of the automated system are described and a suggestion on how it could be implemented into the framework is given in section 3.3.6, this work does not include an actual implementation of such an algorithm. Such implementations are left for further research.

   The suggested framework should rather function as an option for users to implement and test out new path planning algorithm, providing optimal prerequisites to do so.

   For example, the subprocess of the framework, in which the path planning

algorithm should run can be provided with all necessary data, such as sensor data of the drone and the estimated orientation, position and point cloud by the vSLAM algorithm. This stream data is preprocessed and standardized in real time, enabling users to directly use it for their purposes.

2. Evaluation of the vSLAM Algorithms

2.1 Related Work

2.1.1 Introduction and History of SLAM

SLAM algorithms have to solve a chicken-egg-problem, since in order for the robot to localize itself within a unknown environment, it needs a map. However, for building a map, the robot needs to localize itself within it [7]. Until today, a number of different solutions for the problem were developed.

Today, the problem can be formally defined in a probabilistic way. The goal is to compute

$$\mathbb{P}\left(m_{t+1}, x_{t+1} | z_{1:t+1}, u_{1:t}\right), \tag{2.1}$$

where m_{t+1} corresponds to the map (point cloud of the surroundings) at timepoint $t+1$, x_{t+1} to the camera orientation and position at timepoint $t+1$, $z_{1:t+1}$ to all observations made to this timepoint and $u_{1:t}$ all historic control input. In other words, a map and a series of positions should be computed that fit the observed environment and the control inputs in the most likely manner. However, most modern SLAM methods do not require the control input anymore but rather predict the next position with a motion model and validate it with

an optimization process. [8] [9]. This problem overview is displayed in figure 2.

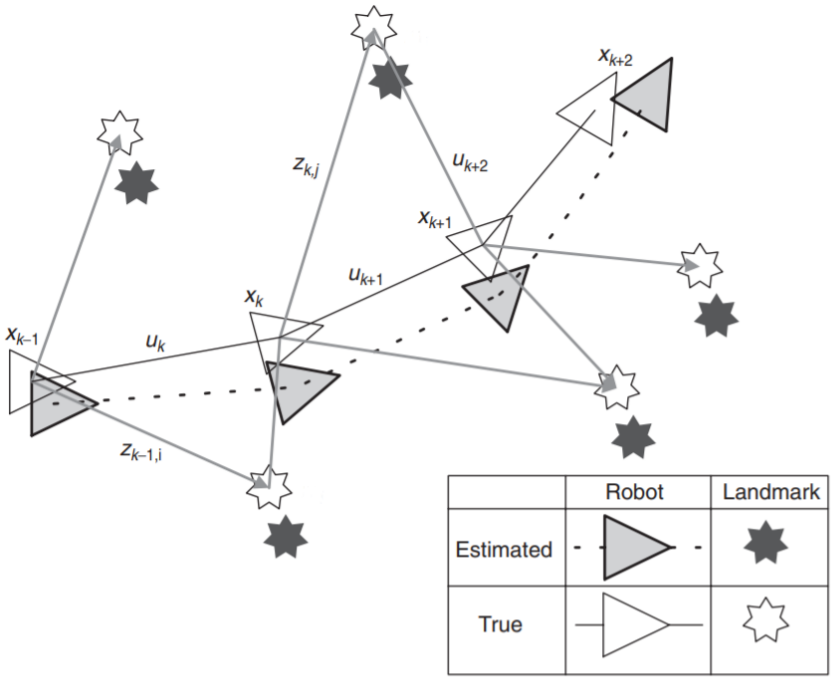


Figure 2: Overview of the SLAM concept. Source: [10]

With their work on the representation and estimation of spatial uncertainty [11] Smith et al. created the first relevant work in the field on SLAM in 1986. However, due to lacking computational resources, the engagement in this topic stayed mainly on a theoretical level.

> Work by Smith and Cheesman and Durrant-Whyte established a statistical basis for describing relationships between landmarks and manipulating geometric uncertainty. A key element of this work was to show that there must be a high degree of correlation between estimates of the location of different landmarks in a map [10].

Their work suggested for the first time that the estimation should be based on a large vector, containing all landmark observations, as shown in the formal definition of the problem in expression .

The first recognizable developed SLAM algorithms relied on optimization methods based on an extended Kalman filter, a non-linear state estimation approach that depends on a joint distribution of all measurements [12]. The first monocular vSLAM algorithm, developed in 2003, is based on the extended Kalman filter and is called MonoSLAM [13].

The computational cost of the MonoSLAM algorithm, and most other vSLAM algorithms grows in proportion to the number of observed features, making it hard to obtain real-time computation for medium or large environments [12]. Obviously, an algorithm runs in real-time, when the algorithm can process the images that are fed to it just as fast as they are produced. Ideally, it should be running recognizably faster than real-time on average, since processing time per image can fluctuate.

In 2007, PTAM (Parallel Tracking and Mapping) contributed with improvements by not running the tracking and mapping processes sequentially, as previous approaches had done, but rather by running theses processes in parallel on two distinctive threads [14].

In the subsequent years, more and more direct monocular visual vSLAM algorithm were developed. Unlike previous feature-based approaches, direct algorithms use the entire image as input in order to compute the term in expression 2.1.1. Therefore, the term is computed by optimizing the photometric error that results from comparing the intensities of each pixel after transformation [8].

> One of the main benefits of a direct formulation is that it does not require a point to be recognizable by itself, thereby allowing for a more finely grained geometry representation (pixelwise inverse depth). [8]

Other than direct SLAM algorithms, feature based methods, compute features

for each frame, that serve as input for further computation. These features usually consist of subsets of pixels that have remarkable intensities and arrangements, such as corners (landmarks). These methods evaluate the upper term by computing the geometric error, since the feature positions are geometric quantities. A main advantage of feature-based methods is the robustness regarding geometric distortions present in every-day-cameras [8].

While there are several evaluation studies [15] [16] [17] [9], [8], the comparison of the results is difficult.

> Whereas dozens of different techniques to tackle the SLAM problem have been presented, there is no gold standard for comparing the results of different SLAM algorithms. [18]

On the one hand, the difficulty lies in the dependency of the results on the choice of the alignment method that transforms the resulting output of the vSLAM algorithm to the reference frame of the ground truth. A different alignment function will generate a different result. This even led to a proposal of an evaluation method that is not dependent on the ground truth reference frame [18]. However, the proposed method has to be computed by performing manual steps and therefore never became the standard.

Also, the usage of different error matrices as well as different benchmark datasets and differences in computational power of the systems, on which the evaluation took place, can cause deviations in results. Finally, the accuracy of the point cloud, which is a crucial part for our proposed framework in the second part of this work to function correctly, has rarely been studied [17].

Therefore, in this chapter, DSO-, DSM- and ORB SLAM are compared using the performance indicators explained in section 2.2.

Of all existing vSLAM algorithms, this evaluation considers DSO-, DSM- and ORB SLAM for the evaluation of being a suitable candidate for the autonomous

exploration task. ORB SLAM was considered, because it showed the good results in different evaluations [15] [19]. DSO SLAM yields also good accuracy [19], and states to be able to track through scenes with very little texture, where feature based methods fail [8]. Finally, DSM SLAM states to be the most accurate direct SLAM method [20].

In the following section, an overview of how the algorithms work is given for each method. In order to understand this section, it is crucial to clarify basic definitions and vocabulary used in SLAM first.

2.1.2 Definitions

Keyframe

Most vSLAM Algorithms make usage of so called keyframes, as keyframe-based approaches have proven to be more accurate [21]. Keyframes are specifically selected images of the input sequences or video streams. In most cases, the keyframes store all of the existing map points as well as all of the computations and optimizations are made, based on the keyframes and data stored within them.

Covisibility Graph

Some algorithms create a covisibility graph in order to link keyframes and gain more information about the environment by having a different representation. In a covisibility graph, nodes are representing features and edges the covisibility between them. This can help the algorithms to perform optimizations and loop closing as explained in the following section.

Group of Rigid Transformations in 3D

SE(3) is the group of rigid transformations in 3D space [22]. Each Matrix $T \in \mathbb{R}^{4x4}$ with

$$T = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}$$

and $R \in \mathbb{R}^{3x3}$ being a rotation matrix and $t \in \mathbb{R}^3$ a translational vector, is an element of SE(3).

Sometimes, instead of representing the orientation by the rotation matrix R, the orientation is given by so called quaternions $q \in \mathbb{R}^4$. The transformation between these two standards is possible.

Point cloud

When we speak of a point cloud, we simply mean a formation of points in the three-imensional space, though having an x, y and z coordinate. Later in the evaluation of the algorithms, to save these point clouds, the PLY format has been chosen.

Camera Calibration and Distortion

For a SLAM algorithm to work with multiple camera types, such as fisheyes cameras, the camera input has to be standardizes. Therefore, different parameters must be provided. These parameters include

- The image size
- The focal lengths (x and y)

- The optical centers (x and y)
- The distortion parameters

The calculations of the resulting transformation in order to gain a standardized image, can be found in the OpenCV documentation.

2.1.3 ORB SLAM

ORB SLAM is a feature-based, state of the art SLAM method. The first version was published in 2015 [9]. Here, an overview of the functionality of ORB SLAM is provided by summarizing their published paper [9]. The algorithms run on three threats simultaneously. Each thread performs one of the following tasks: Tracking, Local Mapping and Loop Closing. An overview over the tasks can be found in figure 3. The explanation of these system components are described in the following subsections. A more detailed explanation can be found in the paper of Raul Mur-Artal et al [9].

Tracking

The tracking component determines the localization of the camera and decides when a new keyframe is being inserted. As shown in figure 3, the tracking is performed in four steps.

1. Feature Extracting

   Features are extracted using Oriented FAST and Rotated BRIEF [23]. This method starts by searching for FAST (Features from Accelerated and Segments Test). Thus, for each pixel x in the image, a circle of 16 pixels around that pixel are considered and checked wether at least eight of these 16 pixels demonstrate major brightness differences. If so, the pixel x is

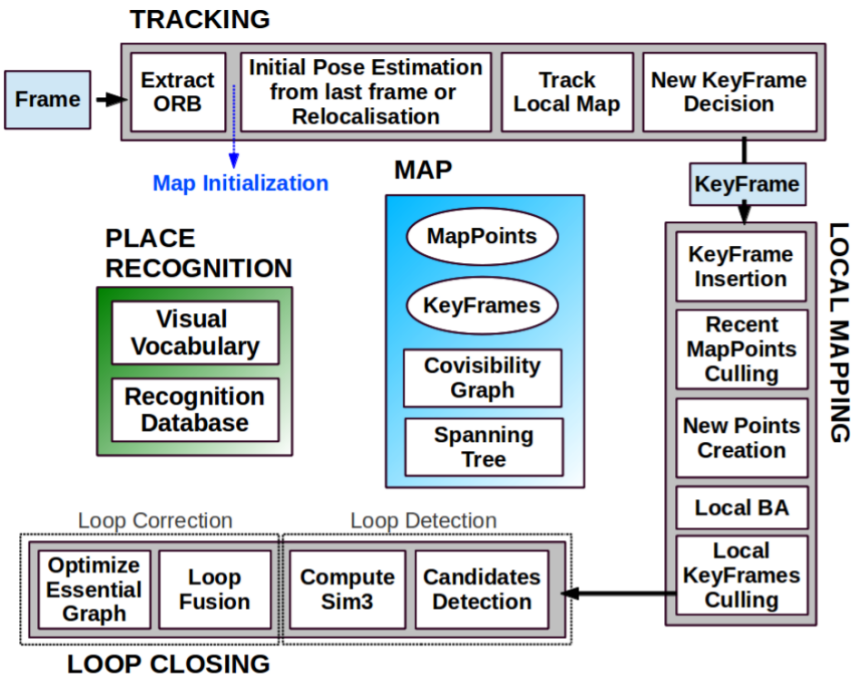


Figure 3: Overview of the system components extracted from [9]

considered as a keypoint, since it is likely to be an edge or corner. This is repeated multiple times after downsizing the image up to a scale of eight. To extract features evenly distributed over the image, it is divided into a grid, trying to extract five features per cell. Extracting features this way, makes the algorithm more stable to scale invariance. Next, the orientation of the extracted feature is calculated using an intensity centroid. Finally, the features are converted into a binary vectors (ORB descriptor) using a modified version, which is more robust to rotation, of BRIEF descriptors (Binary robust independent elementary feature). These ORB descriptors are then used for all feature matching tasks.

2. Initial Pose Estimation

   A constant velocity model is first run to predict to the camera pose. Then, the features of the last frame are searched. If no matches are found, a wider area around the last position is searched.

3. Track Local Map

When the camera pose is estimated, map point correspondences are searched in the local map, comprising keyframes that contain the observed map points and the keyframes from the covisibility graph. The pose is then corrected with all matched map points.

4. New Keyframe Decision

   To insert the current frame as a keyframe, the following conditions have to be met: more than 20 frames have to be passed from the last relocalization or keyframe insertion (when not idle), the current frame tracks at least 50 points or less than 90 percent of the points of the keyframe in the local map with the most shard mappoints.

Local Mapping

Whenever a new Keyframe K_i is inserted, the map is updated.

1. Keyframe Insertion

   The keyframe is inserted in the covisibility graph. Then, the spanning tree is updated using the keyframe with the most common points with K_i. Finally the keyframe is represented as a bag of words using the DBoW2 implementation. Therefore, the image is saved by the number of occurrences of features found in a predefined vocabulary of features. When the vocabulary is created with images general enough, it can be used for most environments.

2. Recent Map Points Culling

   A map point is removed from the map, when it is found in more than 25% in the frames where it is predicted to be visible. Also, it must be observed from more than two keyframes if more than one keyframe has passed from map point creation.

3. New Map Point Creation

   A map point is created by calculating the triangulation of the connected keyframes in the covisibility graph. For each map point, the 3D coordinate in the world coordinate system, its ORB descriptor, the viewing direction, the maximum and minimum distance at which the point can be observed is stored.

4. Local Bundle Adjustment

   The keyframe poses $T_i \in \mathrm{SE}(3)$ and map points $X_j \in \mathbb{R}^3$ are optimized by minimizing the reprojection error to the matched keypoints $x_{i,j} \in \mathbb{R}^2$. The error is computed by the following term:

$$e_{i,j} = x_{i,j} - \pi_i \left(T_i, X_j \right).$$

   i corresponds to the respective keyframe and j to the index of the map point. π_i is a projection function, calculating a transformation to project all keypoints on map points by minimizing a cost function, that can be found in [24].

   > In case of full BA (used in the map initialization) we optimize all points and keyframes, by the exception of the first keyframe which remain fixed as the origin. In local BA all points included in the local area are optimized, while a subset of keyframes is fixed. In pose optimization, or motion-only BA, all points are fixed and only the camera pose is optimized [9] .

   At this point, a local BA is performed.

5. Local Keyframe Culling

   In contrast to other SLAM algorithm, ORB SLAM deletes redundant keyframes, which decreases computational efforts, since computational complexity grows with the number of keyframes. All keyframes are

deleted, in which least 90 percent of the map points can be found in at least three other keyframes.

Loop Closing

The loop closing is computed based on the last inserted keyframe K_i.

1. Loop Candidates Detection

   First, the similarity of K_i to its neighbors in the covisibilty graph is computed by using the bag of words representation and a loop candidate K_l might be chosen.

2. Similarity Transformation

   In this step, the transformation is computed to map the map points from K_i on K_l. Since the scale can drift, it is computed in addition to the rotation matrix and translation using the method of horn.

3. Loop Fusion

   Here, duplicated map points are fused and the keyframe pose T_ω is corrected by the transformation calculated in the previous step. All map points of K_l are projected in K_i. All keyframes affected by the fusion will update the edges (shared map points) in the covisibilty graph.

4. Essential Graph Optimization

   Finally, the loop closing error is distributed over the essential graph.

Complement

Please note that this is the base model of ORB SLAM. A number of amendments, forks and merge requests exist, either made by the creators themselves, or by

external developers. Though it was slightly abbreviated for our purposes, in the evaluation part of this work, ORB SLAM3 is used.

The main feature that was added in this version and also affect the result, is the possibility to create multiple maps within one session. If the tracking is lost, simply a new map is created. When places are then revisited, the maps get merged and the tracking can therefore be resumed normally. This way, good performance results can be yielded, even with long periods of poor visual information [25].

In the second part of this work, ORB SLAM2 was used. This is simply due to the fact that compiling other versions on the platform described in the second chapter failed. However, for the monocular usage, to the author's best knowledge, no main differences are made to the above-described version.

2.1.4 DSO SLAM

Direct Sparse Odometry was developed in 2016 by the Technische Universität München. Here, the published paper "Direct Sparse Odometry" [8] by the creators is summarized.

Model Overview

The model optimizes the photometric error over a window of recent frames.

The camera is calibrated in two different ways. At first, a projection function is computed, considering the focal length and the optical centers, in order to map a pixel point in the image on a 3D map point (and the other way around). Secondly, a photometric camera calibration is applied. This calibration accounts for camera specific non-linear response functions that map scene irradiances on pixel intensities on the one hand and camera specific lens attenuation on

the other hand.

The model relies on minimizing the photometric error. This error over all frames is given by

$$E_{\text{photo}} := \sum_{i \in \mathcal{F}} \sum_{p \in \mathcal{P}_i} \sum_{j \in \text{obs}(p)} E_{pj}, \tag{2.2}$$

where $\mathcal{F}$ is the set of all frames, $\mathcal{P}_i$ the set of all pixel in frame i and $\text{obs}(p)$ the set of indices of frames, where the point p occurs. E_{pj} on the other hand is the difference in pixel intensities, calculated by the huber norm, after mapping the pixels on each other and applying an additional affine brightness transfer function. Also, E_{pj} does not only include comparing the point p, but also computes the pixel intensity difference of eight pixel neighbors of p, arranged in a spread pattern. This pattern is shown in figure 4.

> Our experiments have shown that 8 pixels, arranged in a slightly spread pattern [...] give a good trade-off between computations required for evaluation, robustness to motion blur, and providing sufficient information. [8]

Visual Odometry Front-End

The front end determine the sets $\mathcal{F}$, $\mathcal{P}_i$ and $\text{obs}(p)$. Also it initializes all parameters required to calculate the term 2.1.4. Finally it decides, when to remove points, outliers and keyframes. Frame and point management are closer described in the following section.

1. Frame Management

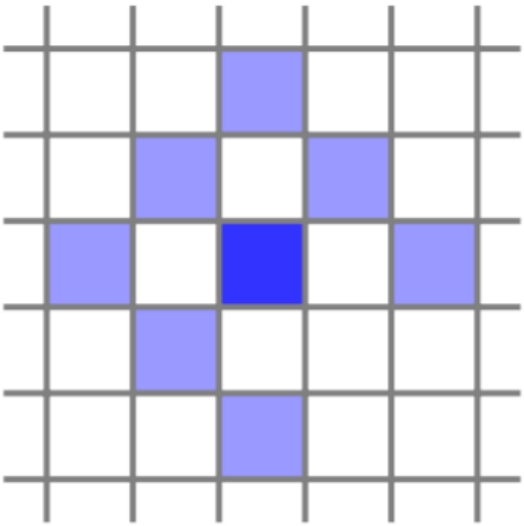


Figure 4: Neighbor pixels considered for the photometric error calculation. Source: [8]

If a new keyframe is created, all map points are mapped into it. In case the root mean squared error of the current frame is more than twice as high than the one before, direct image alignment is assumed to have failed and initialization is tried again. The algorithm tries to always work with seven active keyframes. All computations are made in reference to those keyframes.

For creating a new keyframe, 5-10 frames per second are considered for creation. A keyframe must meet all of the following requirements:

(a) The field of view changes.

(b) Camera translation causes occlusions.

(c) Camera exposure time changes significantly.

When deciding when to marginalize a keyframe, following roles are applied to the active keyframes I_1,...,I_7, with I_1 being the most recent:

(a) I_1 and I_2 are always kept.

(b) keyframes, whose number of points contained in I_1 is less then 5 percent are marginalized.

(c) If still too many keyframes are active, the ones having the largest distance to I_1, or the worst distributed in 3D space are removed.

2. Point Management

DSO always tries to keep 2000 active points in the map. The first step is select candidate points of the frame. To obtain equally distributed points, the image is split into blocks. In each block, the point with the largest gradient of pixel intensity is selected, if it is greater than a threshold, which is computed by adding seven to the overall median of the gradients. This is repeated twice while decreasing the threshold and doubling the block size, in order to also include points with weaker gradient. By gradient, simply the change of pixel intensities to all neighbors is meant.

Then, the point candidates are tracked in subsequent frames by minimizing E_{pj}. From the best match, the depth value is initialized.

When old points are marginalized, candidate points are activated in a way that points in the active map are as evenly distributed as possible. This is achieved by always selecting the point, which offers the largest distance to the next active point, after projecting it into the last active keyframe.

Since outlier only consume unnecessary resources, they are tried to be removed. For example, point for which E_{pj} surpasses a threshold are removed permanently.

2.1.5 DSM SLAM

Direct Sparce Mapping SLAM was released in April 2019 and works similar to DSO SLAM but claims to be more robust when revisiting areas. The algorithm can be separated into a tracking front-end and an optimization back-end that run on parallel threads. In this section, the respective paper of DSM SLAM [20] is summarized.

Model Overview

DSM SLAM uses the same model as DSO SLAM. The model is described in the previous section.

Front-End

The front-end, however, differs from DSO SLAM. While DSO SLAM cannot reuse points that have once being marginalized, DSM suggests a method to activate and inactivate keyframes and points to its needs.

1. Keyframe and map point selection

   When selecting keyframes and map points, two criteria play a role: the temporal and covisibility criteria. The temporal parts regards N_t keyframes as recent sliding window approach, just like DSO SLAM. With similar critirias for keyframe selection as DSO SLAM, whenever a new keyframe is inserted, another one is removed (from the temporal part).

   In order to regard re-observed points, DSM SLAM also considers N_c co-visibil keyframes to fill the latest keyframe I_0 with map points, favoring map points in depleted areas. This is achieved by the following steps:

   (a) Identify depleted areas

   All map points from the temporal part are mapped into the latest keyframe. For every pixel, the euclidean distance to the closest map point is computed. Obviously, large distances suggest depleted areas.

   (b) Co-visibil keyframe selection

   The keyframe within the old keyframes with the most map points in the above selected depleted area are chosen. Map points, where the viewing angle lies too far from the latest keyframe are removed

from the local map. This can be determined from the pose $T_i \in SE(3)$ that is saved for each keyframe I_i.

(c) Update distance map

The distance map, calculated in step 2, is updated with the new selected keyframe.

(d) Iterate

This process iterates until N_c keyframes are selected for the covisibility part.

The entire keyframe and map point selection is displayed in figure 5. In this case, N_t is equal to four and N_c is equal to three.

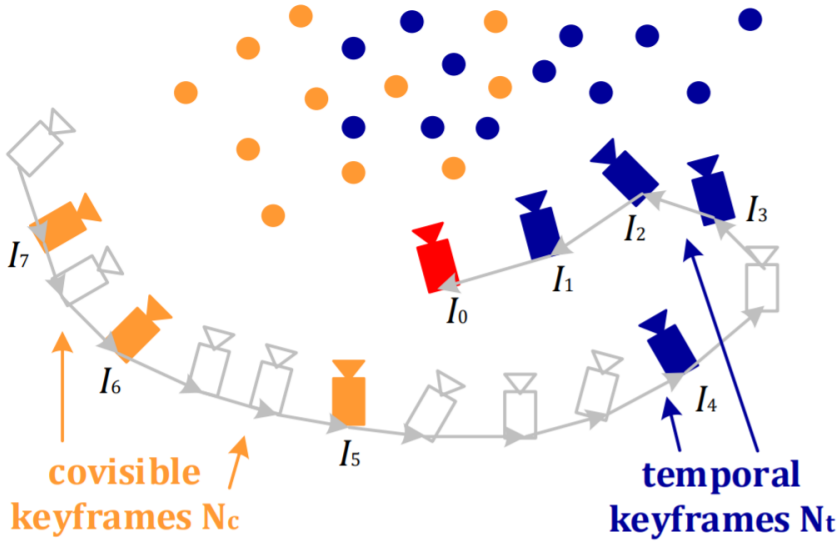


Figure 5: Keyframe and point selection of DSM SLAM. Source: [20]

2. Frame Tracking, New Keyframe decision, New Map Point Tracking

Frame tracking, new keyframe decision and new map point tracking work similar to DSO SLAM. The main idea is always to manipulate the local map by minimizing the photometric error displayed in equation 2.1.4.

However, unlike DSO SLAM, in each frame, the local map consisting

of the $N_t + N_c$ keyframes (referenced from the latest keyframe) and contained map points is projected into the frame. Obviously, this is the main difference compared to DSO SLAM, that keyframes and map points are not permanently culled from the map, and may be reactivated once the keyframe appears in the covisibility graph. Also, the management of outliers is similar to DSO-SLAM, trying to remove outliers as early as possible, in order to save computational resources.

2.2 Evaluation Methods

To evaluate the performance of DSO-, DSM- and ORB SLAM, the algorithms are fed with video sequences contained in the European Robotics Challenge (EuRoC) benchmark dataset. The algorithms are changed in a way, that they save the computed trajectory estimation to a .txt file and the resulting point cloud to a .PLY file.

Because an algorithm run on the same sequence can yield different results, each sequence is run three times for each algorithm. This way, the results are more reproducible.

2.2.1 Dataset

For the evaluation of the vSLAM Algorithms, the EuRoC dataset [6] was used. The dataset contains eleven video sequences, recorded with a micro aerial vehicle at 20 frames per second. The sequences have an image resolution of 752x480 pixels.

For each sequence, RGB images from two cameras exist. However, since the evaluation focuses on monocular SLAM methods, only the left camera was considered. Also the available inertial and camera pose data was not taken in consideration.

The first five sequences were recorded in the machine hall of the Eidgenössische Technische Hochschule Zürich, and the other six were recorded in a room, that was provided with additional obstacles. For the latter six sequences, the ground truth of the environment exists as a dense point cloud, as can be seen in figure 6.

Finally the true camera pose $\in$ SE(3) of the camera is known at a high frequency

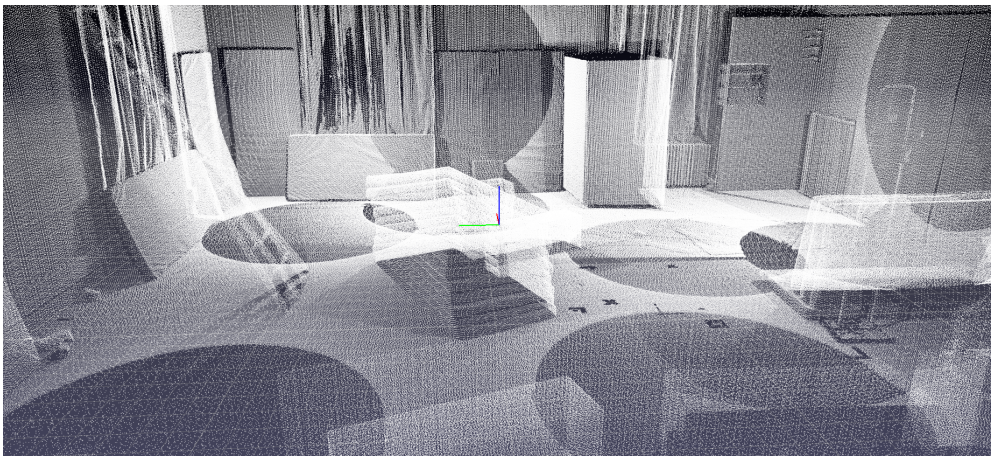


Figure 6: Point cloud ground truth of sequence V1_01_easy visualized with python package pptk

of over 200 points per second. This includes the camera position and the camera orientation.

An overview of the sequences is shown in table 1. For each sequence, the average camera velocity, rotation velocity and duration are provided.

2.2.2 Evaluation Criteria

Trajectory Comparison

1. Trajectory Alignment

   In order to compare the evaluated position of the camera at a given time with the ground truth of the position, the trajectories need to be aligned. This is because most SLAM Algorithms initialize the origin of their coordinate system with the camera position from the first frame. Whereas the ground truth of the trajectory uses a different origin and orientation. As a consequence, evaluated points $\{\widehat{x_i}\}_{i=0}^{N-1}$ can not be compared to the ground truth points $\{x_i\}_{i=0}^{N-1}$. Also, as described in the vSLAM Algorithms section, the minority of the existing vSLAM algorithms are recognizing the true

Table 1: Overview of the sequences included in the EuRoC Dataset

Sequence Name	Duration in s	Average Velocity in ms^{-1}	Angular Velocity in $rads^{-1}$	Point cloud available
MH_01_easy	182	0.44	0.22	No
MH_02_easy	150	0.49	0.21	No
MH_03_medium	132	0.99	0.29	No
MH_04_difficult	99	0.93	0.24	No
MH_05_difficult	111	0.88	0.21	No
V1_01_easy	144	0.41	0.28	Yes
V1_02_medium	83.5	0.91	0.56	Yes
V1_03_difficult	105	0.75	0.62	Yes
V2_01_easy	112	0.33	0.28	Yes
V2_02_medium	115	0.72	0.59	Yes
V2_03_difficult	115	0.75	0.66	Yes

scale of the coordinate system. For those two reasons, the target is to find $S = \{R, t, s\}$, while R being a rotation matrix, t a translation vector and s a scaling factor, such that

$$S = \underset{S'=\{R',t',s'\}}{\arg\min} \sum_{i=0}^{N-1} \left\| x_i - s' R' \widehat{x_i} - t' \right\|^2 \tag{2.3}$$

In other words, the evaluated points are rotated, translated and scaled in a way, that the sum squares error over the respective point distances is minimized. The upper expression is calculated by using the method of Umeyama [26].

Similar to a principal component analysis, Umeyama uses the singular value decomposition of the covariance matrix Σ of x and $\widehat{x}$. Thus, $\Sigma = UDV^T$ is yielded. Umeyama proves, that R, t and s can be calculated as followed:

$$\begin{aligned} R &= UWV^T \\ s &= \frac{1}{\sigma_p^2} \mathrm{tr}\,(DW) \\ t &= \mu_{\widehat{x}} - sR\mu_p \end{aligned}$$

with

$$W = \begin{cases} I, & \text{if } \det(U)\det(V) = 0 \\ \mathrm{diag}\,(1, 1, -1), & \text{otherwise} \end{cases}$$

σ_p being the standard deviation of x, μ the mean and tr the trace of a matrix.

2. Positional Error

The error between $\{\widehat{x_i}\}_{i=0}^{N-1}$ and $\{x_i\}_{i=0}^{N-1}$ is computed after aligning them with the upper method using the computed parameters $S = \{R, t, s\}$, yielding

$$\widehat{x'_i} = sR\widehat{x_i} - t$$

. Then the distances between the points are evaluated using the euclidean norm:

$$e_i = ||\widehat{\mathbf{x}'_\mathbf{i}} - \mathbf{x_i}||_2$$

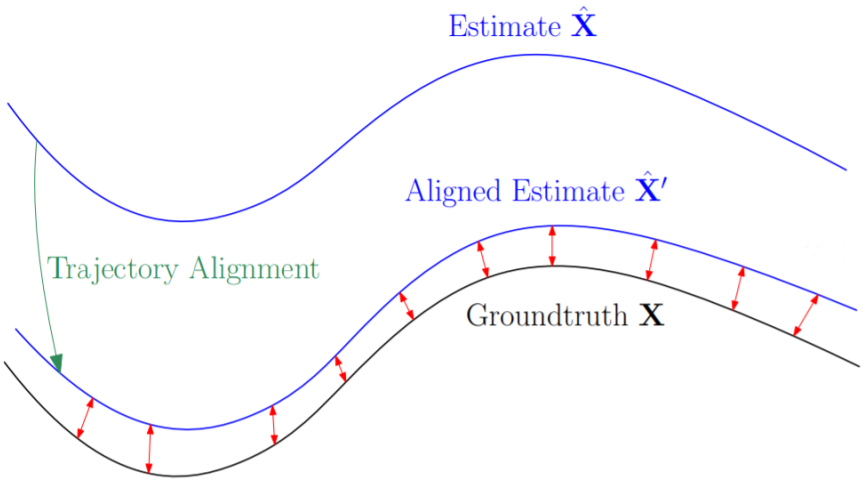


Figure 7: Trajectory error after alignment. Source: [27]

In figure 7 the computed errors are displayed.

These error terms are visualized over time and the overall mean is determined and again visualized with boxplots for each method, sequence and run. Additionally, the flight paths are plotted against each other after alignment, to gain visual information of the trajectories. This is done for all three axes.

Point cloud evaluation

The algorithms were manipulated in a way, that after evaluating each sequence, they write a .PLY file with all map points to the device. These map points are

then evaluated by the following methods. Obviously, this is only done for the latter six sequences, where a ground truth of the point cloud exists.

Additionally to the following methods, the point clouds are visually observed, trying to figure out, if the SLAM algorithms are also able to detect small obstacles, which is crucial for a successful autonomous navigation of the drone.

1. Positional Error

   Again, the map points are transformed using the method of Umeyama. However, it is crucial to note that the computed $S = \{R, t, s\}$ is not a result of aligning the point clouds, but rather the parameters for aligning the trajectories are used. This is done, to ensure, that trajectory and point cloud are transformed in the same way, and fit in the same reference frame.

   To compare the the transformed computed point cloud $P' = \left\{\widehat{p'}_i\right\}_{i=0}^{M_{\text{eval}}-1}$ to the ground truth point cloud $\{p_i\}_{i=0}^{M_{\text{gt}}-1}$, for a point in the evaluated point cloud, the distance to the closest point in the ground truth point cloud is calculated. This is assumed to be the point's ground truth position, since with several 100000-points in ground truth, this point in the ground truth point cloud should not be too far away from the orthogonal projection of the evaluated point.

   This situation is displayed in figure 8, where it gets clear, that the more points are available in the ground truth point cloud, the smaller is the distance in between them and therefore the smaller the error e becomes.

   Since calculating distances from several 100000 points to several 100000 points is computational very expensive, and in the current setup applying it an all sequences, algorithms and runs would require several days, only a subset of P' of 150 points per sequence and algorithms is taken into consideration. The indices for the subset I_{sub} are sampled from a even distribution of $\{i\}_{i=0}^{M_{\text{eval}}-1}$. Then, as mentioned, distances to the closest

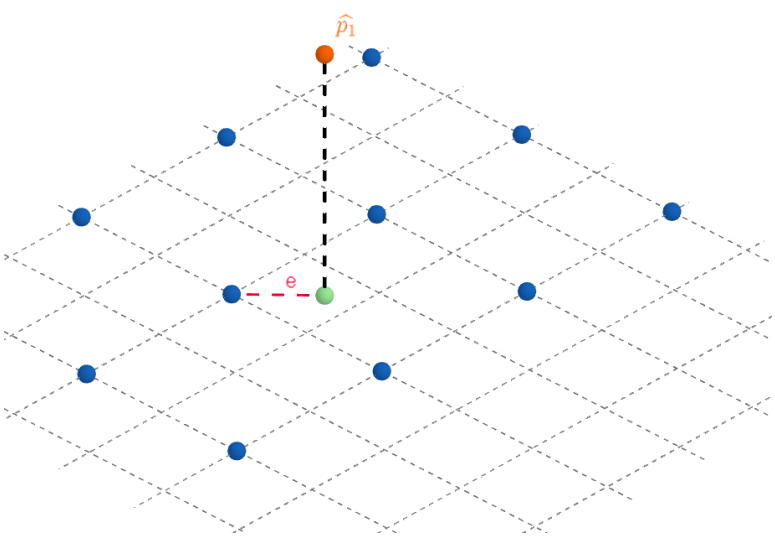


Figure 8: Orthogonal projection of a point in the evaluated point cloud ($\widehat{p_1}$) on the planar of the ground truth point cloud. The error e, determines how far the considered point for the ground truth lies from the actual point of the ground truth.

point in the ground truth point cloud is calculated for the sampled subset $P'_{\text{sub}} = \left\{\widehat{p'}_i\right\}_{i \in I_{\text{sub}}} \subset P'$.

The error term for $\widehat{p'_i} \in P'_{\text{sub}}$ is then given by

$$e_i = \arg\min_{j \in \{i\}_{i=0}^{M_{\text{gt}}-1}} \|\widehat{\mathbf{p}'_\mathbf{i}} - \mathbf{p_j}\|_2$$

.

These error terms are then plotted within a boxplot for each method over all sequences.

2. Density

As described in the second chapters, as a result of the functionality behind feature based methods, their evaluated point clouds are significantly less dense. To quantify the density, for each algorithm and sequence the absolute number of points generated by the algorithm is accessed.

Computation Time

Since the computational performance of an algorithm is crucial to perform in real time, the absolute time that is needed to process each sequence is measured for each algorithm. For each sequence the resulting speed is additionally evaluated in computed frames per second.

2.2.3 Setup and Environment

The entire evaluation is run on a virtual machine. The virtual machine is running through the program VirtualBox provided by Oracle. The host system is a Huawei MateBook D notebook with 16 GB of RAM and the basic model (10210U CPU 1.6 GHz) of an eight core i5. The operating system of the host machine is Windows 10 Home. The virtual machine is given 8 GB of Ram and 4 cores for the computations. The operating system of the virtual machine is Ubuntu 18.04. All further setup information can be extracted from the Github repository.

2.3 Results

The three SLAM algorithms were evaluated regarding the computed trajectories, the resulting point clouds and the computational cost.

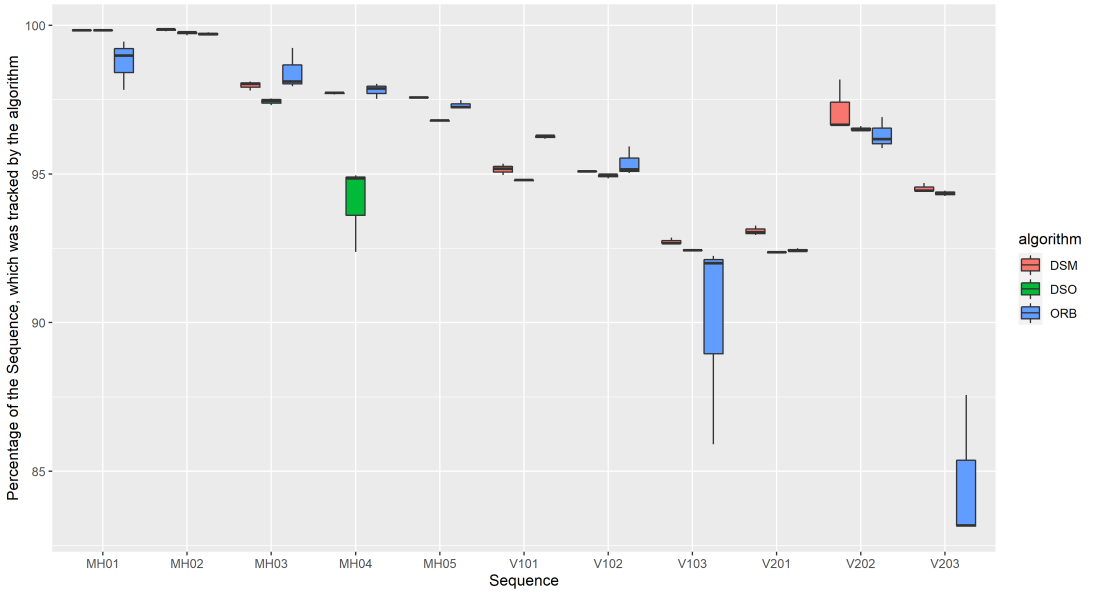


Figure 9: Boxplot of the percentages of the sequences, that were tracked by the algorithm for each run.

Each algorithm needs a certain time to start the tracking. For example, when the drone rests at the first ten seconds of the sequence, the SLAM algorithm will not be able to initialize, since it needs a transformation to recognize the position of a point or feature in space. The amount of required transformations is needed is algorithm-specific. Also, as discussed in section 2.1.3, ORB SLAM3 provides the possibility to create a new local map, once the tracking was lost and merge the new map with the old one, when the places from the old map are revisited within the new map. However, if the tracking is lost and hence a new map is created, but no places from the old map are revisited, these maps can not be merged. Therefore it is possible to yield multiple maps for a single sequence. In case this happens, only the map with the largest amount of keyframes is considered, and the resulting trajectory and point cloud is derived from this

respective map only. In figure 9 the percentages of the sequences where the tracking was successful are displayed. For most sequences, the percentage is similar between the algorithms and the runs. Small differences underlie the differences of the end time-point of the initialization process. However, for sequences V103 and V203, ORB SLAM created a second map, that could not be merged with the old one. Hence, for one run in sequence V103 and for all runs in sequence V203, a part of the tracking is missing. This needs to be taken into consideration when analyzing the results.

The fact, that in one run the tracking can be lost, while within the same algorithm and sequence the tracking is not lost in another run, suggests the conclusion that runs within the same algorithm and same sequence can yield different results. The confirmation of the conclusion is displayed in figures 10 and 11.

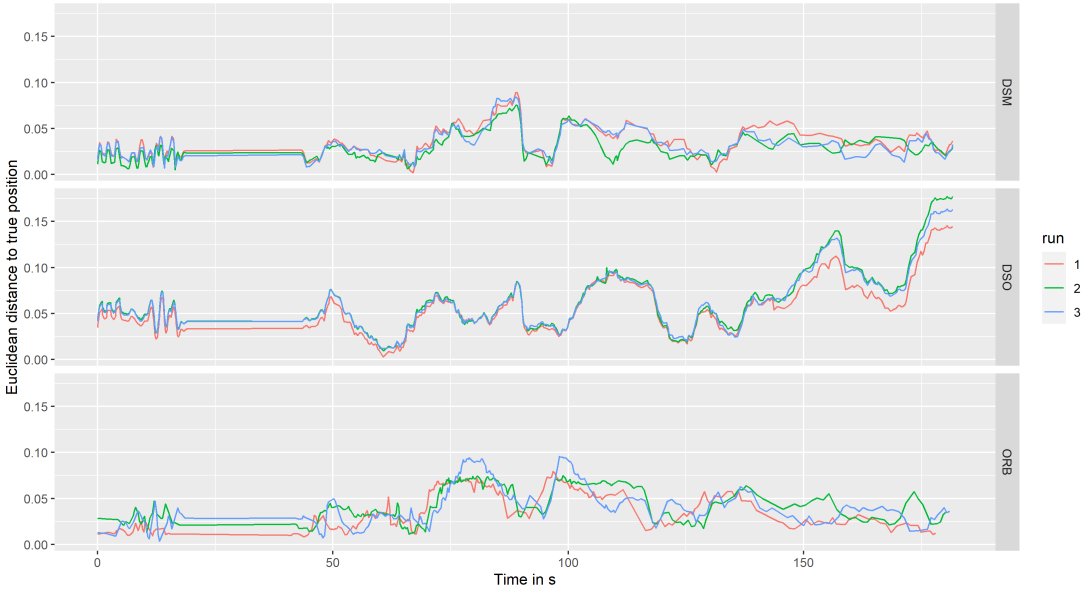


Figure 10: Distances to the ground truth position for the trajectory of all runs within an algorithm for sequence MH01

The first figure shows the results of all runs for the sequence MH01, which is an easy sequence to track, since the the velocity of the drone and the angular velocity is low, while the sequence provides a bright scene and good texture.

Differences between the runs for the sequence are not large, but they exist. For example the first run of ORB SLAM needs more time to initialize the tracking, which causes the tracking to start later and end earlier. These differences between runs may result from the fact, that the algorithms run in different threads, that perform simultaneously. This does not guarantee that the computations always occur in the same succession. Differences seem to be lower for DSO SLAM. This may be because DSO SLAM has no thread performing global optimization simultaneously.

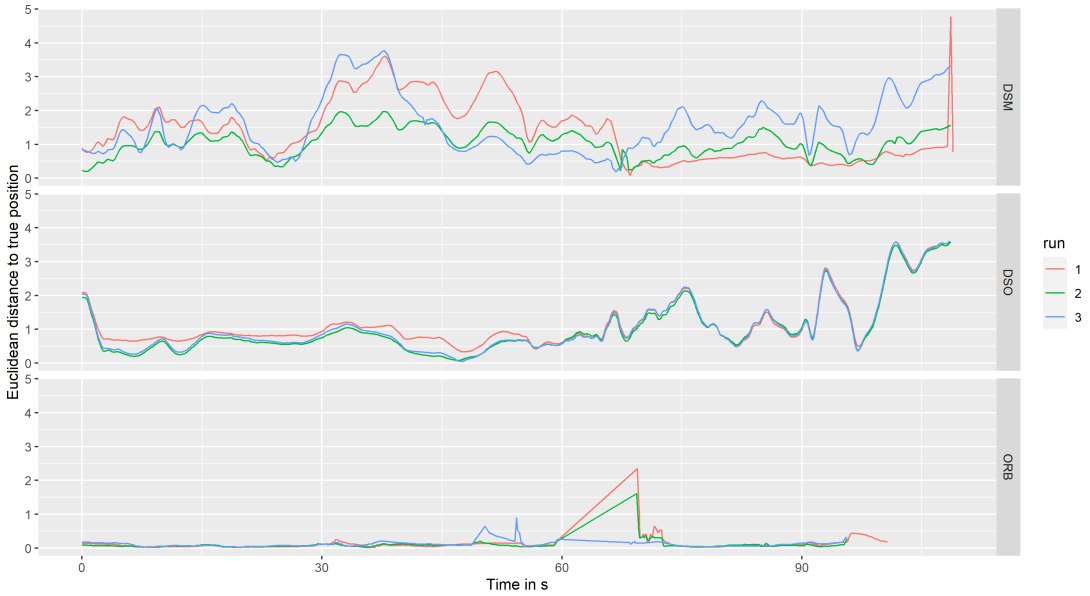


Figure 11: Distances to the ground truth position for the trajectory of all runs within an algorithm for sequence V203

For more difficult sequences, that might even cause some runs to loose tracking while others don't, as it can be seen in figure 11

2.3.1 Trajectory Evaluation

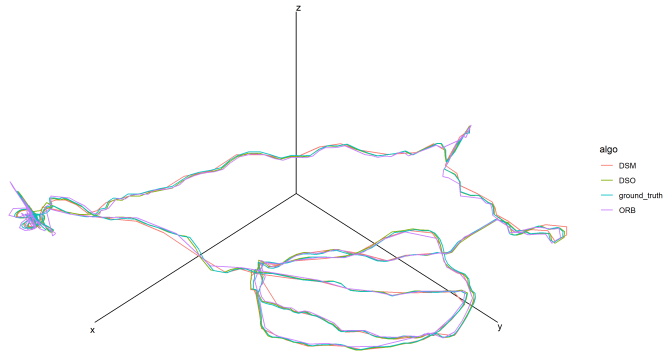
After alignment, as a first indicator for the accuracy of the computed trajectories, the trajectories were visually investigated by plotting the true position and the evaluated position into a coordinate system. This method for comparing

the trajectories is described in detail in section 1. Obviously, those trajectories, that align perfectly with the ground truth suggest that the position was correctly evaluated by the algorithm.

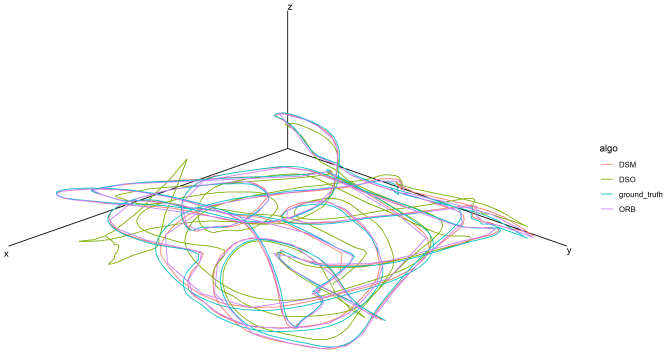
In figure 12 the computed trajectories are plotted against the ground truth for sequences MH01, V102 and V203 by inserting the trajectories into a 3D coordinate system. This is displayed for the first run only, in order to provide an overview. These sequences were selected, because they represent the results of the visual analysis well. In the first sequence, MH01, all algorithms showed excellent results. One reason for this could be, that in the first sequence, the camera only does very gentle movements, moving at an average speed of $0.41ms^{-1}$.

The second image shows the sequence V102. Here, ORB SLAM and DSM SLAM show good results, since in most parts, the trajectories are aligned with the ground truth trajectory. DSO SLAM on the other hand shows a significant difference in the flight path. When observing the trajectory closely, the assumption is raised, that the algorithm lost tracking and therefore computed significant wrong position data. This might have caused the calculated positions in the left of the plot, that have a large distance to the ground truth. The rest of the sequence might have been correctly estimated, however since the alignment is done minimizing the term 1, one significant mistake in the position estimation, might result in severely wrong results over the entire sequence after alignment. This is supported by watching the algorithm running and by the fact that in the plot the trajectory looks very similar to the ground truth trajectory, only transformed.

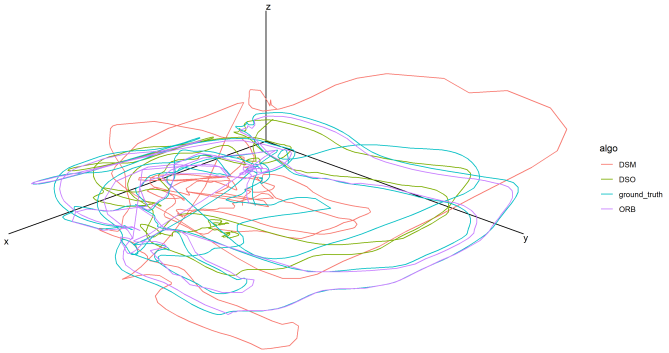
The last plot shows the results of the last sequence. Here, all algorithms had problems estimating a position that comes close to the ground truth position. ORB SLAM still showed acceptable performance. However, as described, ORB only tracks about 83 percent of the whole sequence.



(a) MH01



(b) V102



(c) V203

Figure 12: Ground truth flight path and evaluated flight path of each algorithm after alignment with the method of Umeyama in all axes in meters. Top the sequence MH01, middle the sequence V102 and bottum the sequence V203 are displayed.

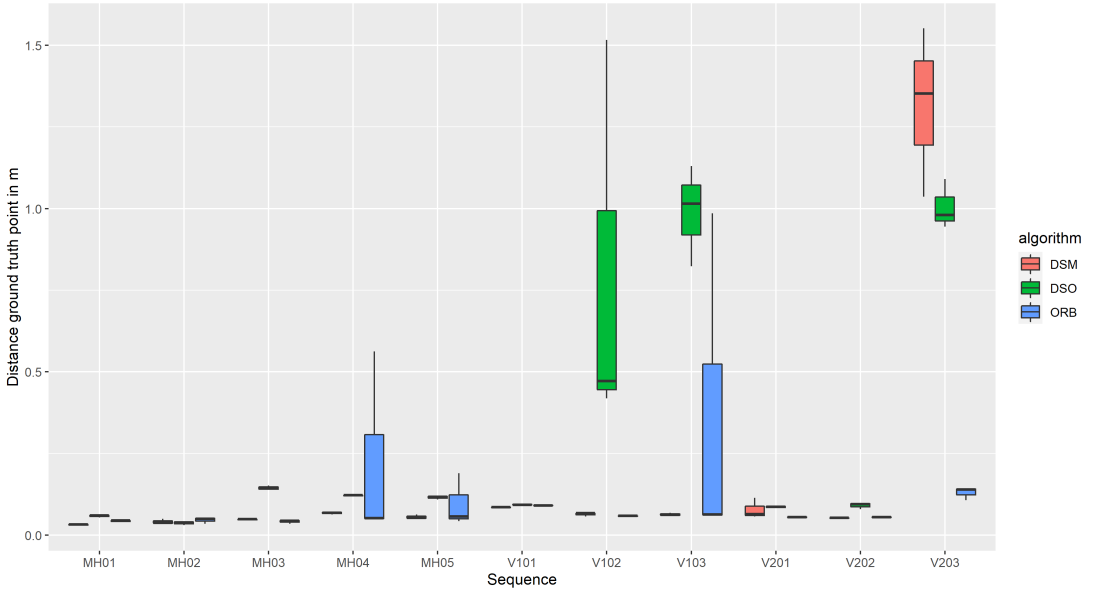


Figure 13: Boxplot of the mean positional errors of the runs for each sequence and algorithm

Furthermore, the euclidean distances between position of the keyframe and the true position of the latter are computed. For a keyframe, the entry of the ground truth data with the lowest distance in time to the time the keyframe was inserted is taken as reference point. This is justifiable, since the true position is sampled at a frequency of over 200 points per second.

Figure 13 shows the mean distances of all runs for all sequences and for all algorithms. As suspected, in the first five sequences in the machine hall all algorithm performed good. The availability of more features in the scenery and the slow motions of the camera might explain the yielding of these excellent results. Also, as it can be derived in figures 10 and 11, DSO SLAM tends to drift further apart from the ground truth position the longer the tracking is active. This might be the result of lacking the functionality to close loops and to optimize over the global map, as explained in section 2.1.4 .

To summarize, over all runs and sequences, including the sequences, where the tracking was lost, ORB SLAM had a positional difference to the ground

truth of 11 cm, DSO SLAM of 32cm and DSM SLAM of 17cm. When excluding sequence V103 and V203, ORB evaluated the position with an average error of 7cm, DSO SLAM with 17cm and DSM SLAM with 6cm.

2.3.2 Point Cloud Evaluation

For the evaluation of the computed point clouds, these point clouds were first visually observed, as described in section 2.2.2. Figure 14 shows the evaluated point clouds aligned with the ground truth point cloud for sequence V101. In this sequence the tracking of the trajectory was successfull for all three algorithms, thus, the errors of the resulting point clouds can not be a result of errors in alignment.

What becomes clear at the first glance is that ORB SLAM generates only few points, since only found keypoints are mapped in feature-based methods. To give these points better visibility, the point size was doubled in the ORB image. DSM- and DSO SLAM generate point clouds with significant higher density, where all structures of the room are immediately visible with good quality.

On the other side, the advantage of ORB-SLAM over the other two direct methods is the recognition of clear features in terms of structural differences in the scenes. Though DSO- and DSM SLAM also regard the differences in the pixel intensities, ORB SLAM detects the features on different scale levels and ensures that the regarded features are truly significant, as described in section 2.1.3. This also became clear when observing the point cloud. All significant features, and therefore important features for autonomous navigation, were successfully marked with a computed point. For example, this can be seen at the ladder in the third image of figure 14, where all the subsequent rungs contain at least one point.

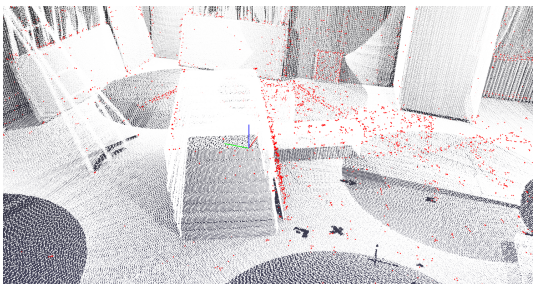
After elaboration on the point clouds, it also became clear, that the point clouds

of DSO, often generate point clouds, where multiple layers of points were falsely generated and all points had the same clear distance to the ground truth point cloud. This may be a result of the functionality of DSO slam, where marginalized keyframes are removed permanently. For revisited areas, the points are regenerated. This means that all errors made in the sequence accumulate and when an area is revisited, significant errors in the point cloud can occur. This can be seen when looking at the third image closely. When looking at the mattress in the middle of the room, the accumulated error expresses itself by points hovering in the air in a clear plane.

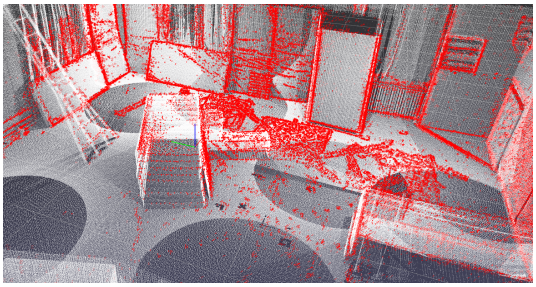
The density can also be expressed by numbers. The significant difference of the numbers of points can be seen in table 2. While ORB SLAM only generates close to 10000 points in the sequences, DSM SLAM generates more than 500000 points in most sequences and DSO SLAM more than 200000 in most sequences.

However, regarding the accuracy of the computed points by the algorithms, again ORB SLAM shows the best performance. In table 2, for all sequences, where a ground truth point cloud exists, the distance to the closest point in the ground truth point cloud is computed by randomly sampling 150 points per algorithm and sequence. The means of each run again averaged an shown in table 2.

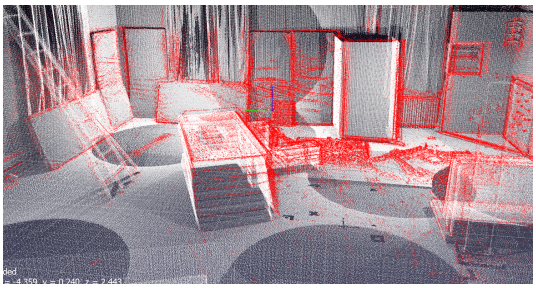
The result is also shown in figure 15, where the correlation between the trajectory accuracy and the point cloud accuracy become clearly visible. Averaged over all runs and sequences, a point generated by ORB SLAM lies 15cm from the next ground truth point, DSM SLAM 19cm and DSO SLAM 35cm. Unlike the trajectory error, the point cloud error of ORB SLAM is still the smallest when the Sequences V103 and V203 are not considered. Then ORB SLAM only causes an error of 4.8cm, DSM of 8.8cm and DSO of 21cm. This is because ORB SLAM does not generate as many outliers as DSO- and DSM SLAM, as it can be seen in figure 16.



(a) ORB



(b) DSM



(c) DSO

Figure 14: The ground truth of the point cloud from Sequence V101 (white points) and the evaluated points by each algorithm (red points). The points in Figure (a) are twice as large for better visibility (ORB-SLAM generates only few points).

Table 2: Absolute number of points and mean distance to the closest point in the ground truth point cloud in meter (written in parentheses) for each algorithm and sequence averaged over all runs.

Sequence Name	ORB	DSM	DSO
MH_01_easy	9959.7 (/)	683237.3 (/)	371389.7 (/)
MH_02_easy	9551 (/)	678000.0 (/)	342405.7 (/)
MH_03_medium	7652.7(/)	634178.7 (/)	372070.3 (/)
MH_04_difficult	10084.3 (/)	505642.7 (/)	211698.7 (/)
MH_05_difficult	10607.3 (/)	505677.3 (/)	234682.7 (/)
V1_01_easy	8088.3 (0.0418)	582530.7 (0.0593)	372805.7 (0.0734)
V1_02_medium	7927.3 (0.0486)	642992.0 (0.0672)	356028.7 (0.6028)
V1_03_difficult	9373.7 (0.3137)	786626.7 (0.0955)	445751.0 (0.6416)
V2_01_easy	10154.0 (0.0515)	589416.0 (0.0736)	244025.7 (0.0927)
V2_02_medium	9007.7 (0.0509)	785309.3 (0.1557)	494689.7 (0.0976)
V2_03_difficult	10985.7 (0.4350)	831517.3 (0.7208)	464663.3 (0.6189)

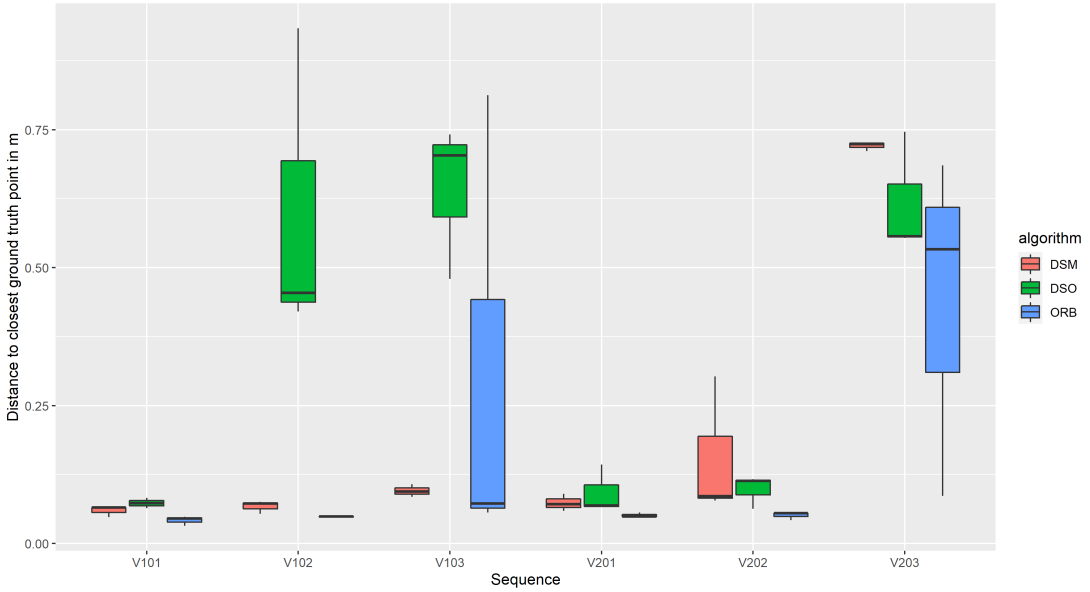


Figure 15: Boxplot of the means for each run of the euclidean distances between an evaluated point and the closest point of the ground truth point cloud. For computational feasibility, for each sequence, run and algorithm, 150 points for evaluation are sampled randomly

2.3.3 Computation Time

The computation time for each sequence and algorithm was measured, respectively the time the algorithm consumed to complete the computation of the sequence.

In order to evaluate, if an algorithm can run in realtime, the absolute times have to be broken down into the computed frames per second. The EuRoC dataset consists of sequences recorded at a frame frequency of 20 frames per second. This means the total frames per sequence amount to the duration of sequence in seconds times 20. For the computed frames per second, this value is then divided by the time the algorithm needed to process the sequence.

The results are displayed in table 3. ORB SLAM processes the frames at a frame per second rate of 17, DSM SLAM processes the frames at 4.17 frames

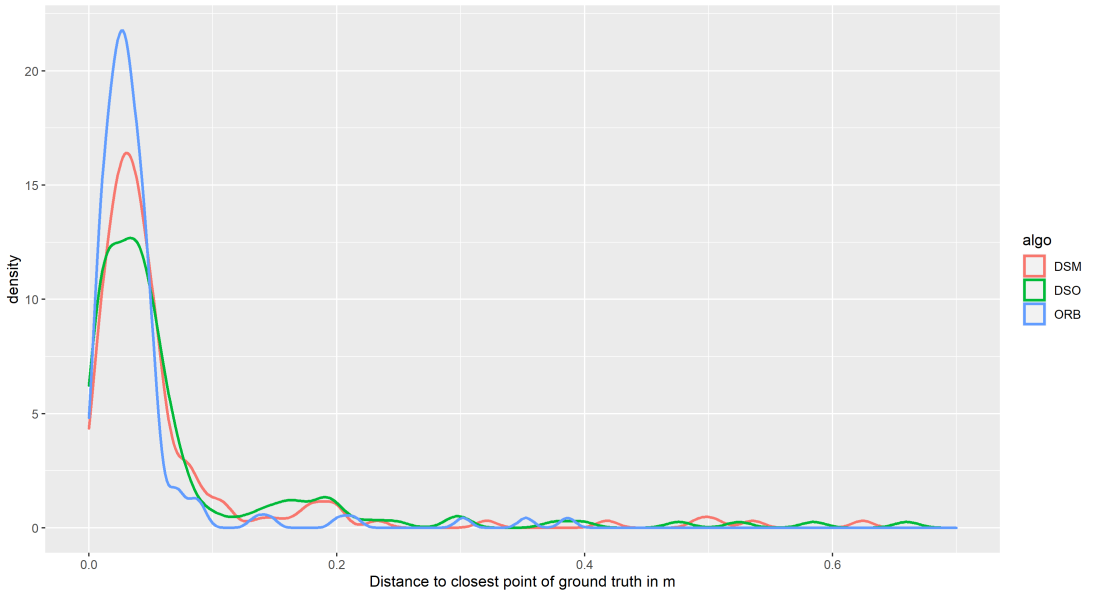


Figure 16: Distribution of the error distances of Sequence V101 and run 1.

per second and DSO SLAM at a rate of 5.22. Thus, none of the evaluated SLAM algorithms processes any of the evaluated sequences in realtime, but ORB SLAM is at least three times as fast as both other algorithms in processing the frames.

Table 3: Computation time (excluded time needed for initialization) of each sequence and algorithm, averaged over all runs. In parentheses the resulting computed frames per second are given.

Sequence Name	Computation Time in *s* ORB	Computation Time in *s* DSM	Computation Time in *s* DSO
MH_01_easy	215 (17.0)	688 (5.30)	798 (4.56)
MH_02_easy	174 (17.2)	538 (5.57)	732 (4.10)
MH_03_medium	157 (16.9)	656 (4.02)	403 (6.85)
MH_04_difficult	117 (16.9)	394 (5.02)	260 (7.69)
MH_05_difficult	132 (16.8)	385 (5.76)	372 (5.97)
V1_01_easy	168 (17.1)	612 (4.71)	628 (4.58)
V1_02_medium	101 (16.6)	678 (2.46)	345 (4.99)
V1_03_difficult	123 (17.1)	907 (2.32)	654 (3.22)
V2_01_easy	131 (17.1)	461 (4.87)	357 (6.28)
V2_02_medium	136 (17.0)	783 (2.95)	415 (5.55)
V2_03_difficult	133 (17.1)	796 (2.89)	640 (3.60)

2.4 Discussion

2.4.1 Conclusion of SLAM-Algorithm Evaluation

The evaluation showed, that ORB SLAM outperforms DSO- and DSM SLAM regarding trajectory accuracy, point cloud accuracy and computation time. Only when sequences are removed, where tracking was lost, DSM SLAM yielded a trajectory accuracy one cm better than ORB SLAM. The only major drawback, when using ORB SLAM in the framework introduced in the next chapter, is the low number of generated points. However, these points are more accurate and ORB SLAM reveals a lower computational time, which is also relevant for the later discussed path planning algorithm.

Computation is not yet in real time for ORB, however in order to lower the computation time, it is possible to decrease the frame frequency, but this will also increase the risk of losing the tracking, since the steps between the frames are greater and the constant velocity model cannot perform a sufficient accurate initial prediction of the new position. Also, it is possible to change the hyper-parameters of the algorithms to increase the computation speed. For example, instead of extracting 2000 features per image, only 1000 can be considered. Finally, the computation speed can be increased by decreasing the resolution of the input images or by upgrading the hardware with a better processor or a GPU.

2.4.2 Future Options

In the recent years, machine learning solutions for the SLAM problem where examined [28] [29]. Thus, in the works "D3VO: Deep Depth, Deep Pose and Deep Uncertainty for Monocular Visual Odometry" [29] and "CNN-SLAM:

Real-time dense monocular SLAM with learned depth prediction" [28] Yang et al. and tateno et al. introduce the usage of neuronal nets to either support the depth, pose and and uncertainty estimation or even do the prediction only based on the output of the Convolutional neuronal network. They claim to yield excellent results.

> We systematically evaluated the VO performance of D3VO on the two datasets. D3VO sets a new state-of-the-art on KITTI Odometry and also achieves state-of-the-art performance on the challenging EuRoC MAV, rivaling with leading mono-inertial and stereo-inertial methods while using only a single camera.

Unfortunately, these works are closed-source and thus can not be used for the proposed framework in the second chapter. However, machine learning approaches hold great opportunities for the future, as they have been outperforming traditional approaches in most other computer vision related areas.

> Deep learning has swept most areas of computer vision – not only high-level tasks like object classification, detection and segmentation [30] [31] [32], but also low-level ones such as optical flow estimation [33] [34] and interest point detection and description [35] [36]. [29]

3. Fully Automated Exploration System

3.1 Introduction

Just like the proposed system, most other automated exploration systems are based on the combination of a SLAM algorithm and a path planning algorithm [37] [38] [1] [39]. One example, where automated exploration with a drone is applied in real life is described in the work with the title "A Fully-Autonomous Aerial Robot for Search and Rescue Applications in Indoor Environments using Learning-Based Techniques" by Sampedro C. at al [37]. They propose a complex framework targeted for search and rescue operations.

One issue, that becomes clear and will not be treated in this work, is the weight of the hardware, that is necessary to perform all computations onboard. While the drone in Sampedro's work had to compute even more processes, such as running a convolutional neural network for object classification, the drone amounted to a total weight of 3.2 kg. Even if the drones for the purpose of the proposed framework will not be as heavy, processing the computations onboard of an ARdrone 2.0, as it is used in our simulation, is impossible. When taking the proposed framework to a real life scenario, as it is also discussed in this chapter, the drone would have to hold a wireless connection (WIFI in case of the ARdrone 2.0) to a server, that does all the computations. Or, a different drone model would have to be used, in order to carry the hardware, that is needed for computation.

Another framework suggested by Dowling et al. was build inter alia for purposes of autonomous UAV navigation in cities.

> Nevertheless, in confined areas such as cities, forest and buildings it is not possible to regulate UAV altitude and position because the GPS information is weak or not reliable. In addition, to achieve a complete autonomous flight-controlled system it is necessary to map the UAV's surrounding with high precision and accuracy in order to identify obstacle-free trajectories [40].

3.2 Related Work

As mentioned in the introduction of this chapter, the existing frameworks are mostly based on a SLAM algorithm and a path planning algorithm. While the existing SLAM algorithms are discussed in detail in the previous chapter, current methods on tackling the path planning task are discussed in this chapter in section 3.3.6.

The path planning task for the autonomous exploration is also related to visual servoing, which is a technique to control a robot, by directly feeding it visual information. However, for most of these techniques, the observation target has to be predefined and is also assumed to always be in the field of view.

> In most visual servoing methods, feedback signals for the servoing controller are estimated by means of multiple-view geometry assuming the target scene being always within the camera field of view (FOV) [41].

3.3 Automated Framework Proposal

In this section a framework to test and build an entire automated system is suggested. This framework includes a simulated environment, that enables the drone navigation within a simulation realistically. The environment is based on the Roboter Operating System (ROS).

The basic idea of our framework can be seen in figure 17. The main parts of the automated exploration system are the SLAM Algorithm itself and the flight path planning algorithm, that work simultaneously. The general concept is that each of the algorithms takes the output of the other as input.

In the following section the suggested ROS framework is proposed and described in detail.

ROS stands for Roboter Operating System and as the name suggests, it is a framework created to manage the software infrastructure within a robot. With the right drivers installed, it can access and use the robot's hardware and serves as a messenger system between robot components. ROS packages make it easy to reuse important functionalities.

Gazebo on the other hand is an open-source 3D dynamic simulator for robotics. It can accurately and efficiently simulate robots regarding their physics and behavior. While gazebo simulations of drones can be created from scratch, this work relies on a a gazebo simulation, that was introduced by the robotics department of the Technische Universität München (TUM). Details can be found in section 3.3.1.

The suggested ROS setup consists of different nodes. Each node runs a process where certain computations are performed. These computations are based on input data and in most cases, the nodes also create data to output to the system. Within ROS, data is shared and received over so called rostopics (in short top-

ics), which is a simple publish-subscribe-pattern for network communication. Each node can therefore publish the computed data over a certain topic or it can receive data from a certain topic by subscribing to it.The frequency for the data transmission to the system is also defined for each topic. The frequency for the data reception can also be manually defined for each subscription.

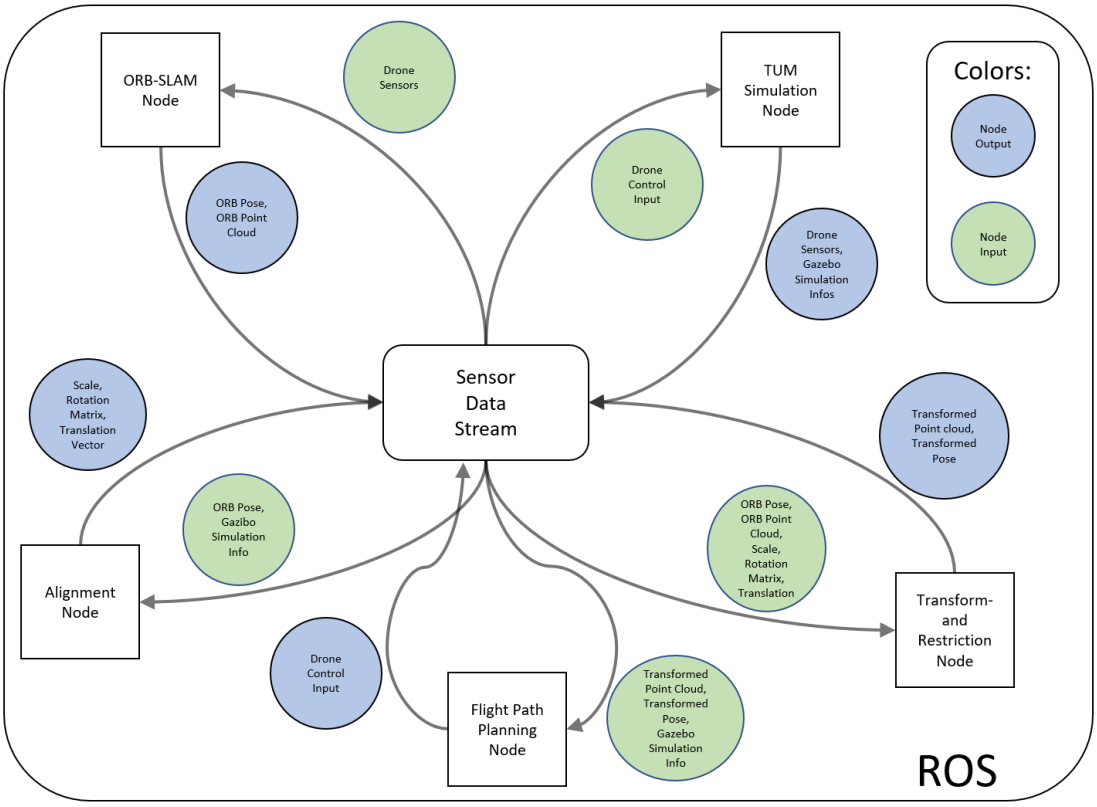


Figure 17: Overview over the suggested ROS framework for the simulated case

In figure 17 the suggested setup for an automated exploration framework within a virtual environment is displayed. The system consists of five nodes. While the functionality of the ORB SLAM node, the TUM simulation node and the fight path planning node may be derived by previous section, such as the upper part of this section and the introduction, the functionality of the other two nodes have not been introduced.

To take advantage of the fact, that running these algorithms in a simulated environment provides the system with information about the exact position of

the drone and the position of simulated environment, these information must be processed. This is done in the alignment node and the transformation and restriction node.

The alignment node computes the parameters, that are needed to transform the reference frame of the ORB SLAM algorithm to the reference frame of the gazebo simulation. The actual transformation of the point cloud and the pose computed by ORB SLAM are then performed in the transformation and restriction node. Additionally, this node restricts the searching space of the path planning algorithm to a finite area. With the help of these two nodes, the system is then able to:

- Identify the true scale of the point cloud
- Confirm the correctness of the computed point cloud
- Confirm the correctness of the computed trajectory
- Limit the searching space of the path planning algorithm
- Feed the path planning algorithm with collision information of the drone
- Identify undiscovered spaces

Obviously, when testing the system in a real life environment, the ground truth of the position of the environment and drone are not available. Nevertheless, in order to get the true scale of the resulting point cloud, a modified framework is introduced additionally. This framework suggests an initialization process, that allows for an estimation of the position of the drone. The framework for the real life case can be found in figure 18

The functionality of all nodes are described in detail in the next sections.

Since the nodes are only dependent on each other in a way that they communicate over standardized messages, they are independent of the programming

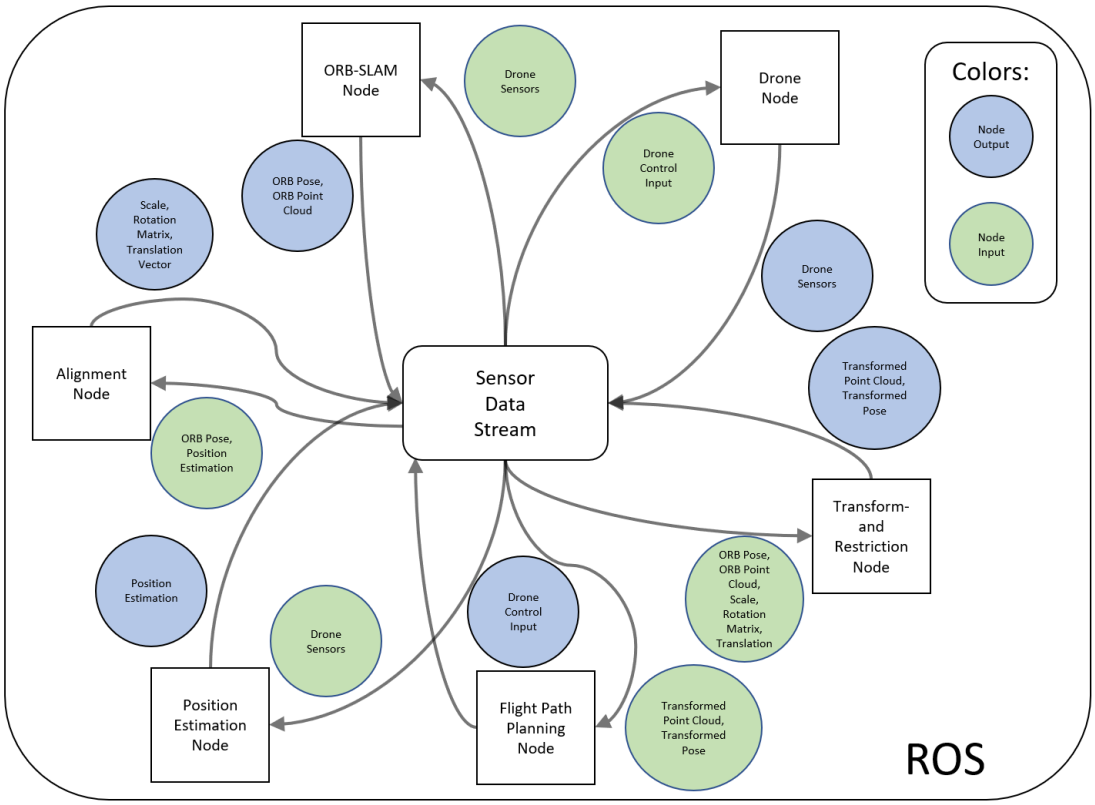


Figure 18: Overview over the suggested ROS framework for the real life case

language. This means that while the ORB SLAM node is implemented in C++, all nodes developed explicitly for this framework are implemented in python with rospy.

3.3.1 TUM Simulation Node

Thus, with the tum_simulation package you can navigate an AR.drone 1.0 and 2.0 in different worlds created within gazebo. This drone is equipped with a bottom camera and a front camera. Each camera logs their output to a topic. Additionally, message time stamps, the height sensor output, battery percentage, rotation velocity and acceleration are also logged to topics. While the drone can also be navigated using a PlayStation 3 controller, as displayed in figure 19, showing a section of the tum_simulation package content structure,

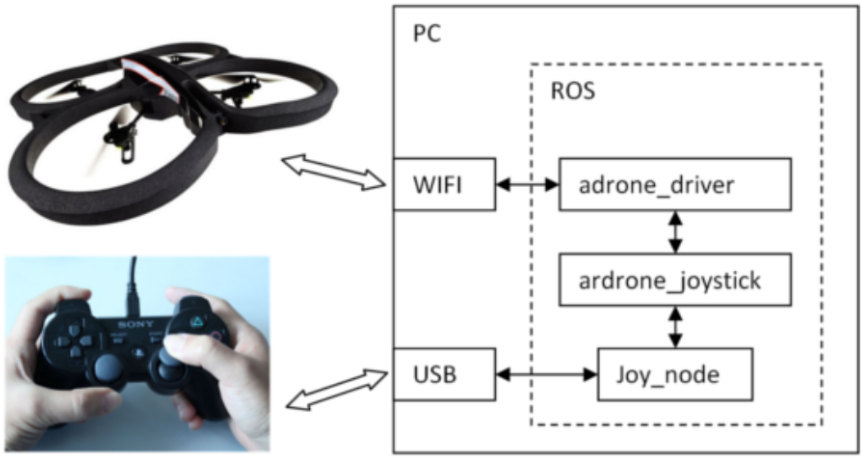


Figure 19: TUM simulator setup. Source: http://wiki.ros.org/tum_simulator

for an automated system, the drone should rather be addressed by publishing messages to the drone control topic (/cmd_vel) using the command line interface. For example by publishing messages of the class Twist, the drone can be navigated. With the command shown in listing 1, the drone is controlled to fly forward, as the linear translation vector points only in the x direction.

Listing 1: Drone navigation command

```
1        # navigate the drone forward
2    rostopic pub -r 10 /cmd_vel geometry_msgs/Twist  '{
       linear:  {x: 1.0, y: 0.0, z: 0.0}, angular: {x:
       0.0,y: 0.0,z: 0.0}}'
```

In the next sections, the input and output topics of the drone are discussed.

Input

1. /cmd_vel

   As mentioned, the node is subscribed to the /cmd_vel topic. Whenever a valid navigation message is received, the drone reacts accordingly, if it can.

Output

The drone has many sensors attached and more than thirty topics are logged to the system. However, here only the topics, that are of importance for the automation framework will be listed.

1. /ardrone/front/camera_info

   Over the /ardrone/front/camera_info topic, the node publishes messages of the class CameraInfo. These messages include information about image dimension, timestamp and information about the camera specific values, described in section 2.1.2. In the case of the ardrone 2.0, the front camera generates images with 640x360 pixels and the intrinsic camera matrix is given by:

$$K = \begin{pmatrix} 374.6706070969281 & 0.0 & 320.5 \\ 0.0 & 374.6706070969281 & 180.5 \\ 0 & 0 & 1 \end{pmatrix}$$

   From this matrix, the focal lenghts can be extracted.

2. /ardrone/front/image_raw

   The node publishes the actual output image data of the front camera in the /ardrone/front/image_raw topic. The metadata of the camera are provided in the topic /ardrone/front/camera_info described above.

3. /ardrone/navdata

   Messages of the specifically developed class Navdata are published to this topic by this node. These messages include information about battery percentage of the drone, state of the drone (e.g hovering, flying, init, landing...), pressure, temperature, wind, velocity and some more information. These messages are also timestamped.

4. /ardrone/takeoff

   As the name of the topic suggests, the drone takes off, when empty messages are published to the thread. How this looks exactly can be found in listing 2.1.2.

5. /gazebo/model_states

   Gazebo provides the possibility to access the current pose of each model existing in a respective gazebo world. For example the ardronetestworld, that is displayed in figure 21, consists of the drone itself, several houses, a barrier etc.

   Therefore, the /gazebo/model_states topic publishes the list of the pose, $x_i \in \mathrm{SE}(3)$ of each model. However, in the ardronetestworld, only the drone is dynamic and important for this topic, all other models are static and therefore the pose will not change.

6. /gazebo/collision

   It is possible to implement a contact sensor to a gazebo model. However, currently the implementation has not yet been successful, as described in section 3.4.1.

Real-life framework

For the real life framework, no gazebo simulation is running. Instead, a node processes the sensors of the drone within a node. The output topics of the drone are then limited to the drone sensors, as displayed in figure 18.

3.3.2 ORB SLAM Node

The ORB SLAM algorithm runs in a separate node. For the vocabulary file, needed for the bag of words approach, explained in section 1, the standard

vocabulary file provided by the authors are considered. For the virtual environment, it might be useful to provide a vocabulary file generated for this particular purpose, since in the simulation generated with the tum_simulation, edged might be of different shape, e.g particularly sharp.

The node publishes the pose $x_i \in SE(3)$ and map points computed by the ORB SLAM Algorithm. These features had to be implemented since the original ORB SLAM ROS implementation did not have an option for publishing data and projects incorporating this functionality.

Input

1. /ardrone/front/image_raw

   This topic is explained in the upper section 3.3.1.

Output

1. /orb/pose

   This topic publishes messages of the class PoseStamped. This class includes a header, where the frame_id and most importantly a timestamp can be provided. The pose is then given by x, y, and z position coordinates and the orientation is given with quaternions coordinates that are discussed in section 2.1.2. The topic is published at a frequency of 30 Hz.

2. /orb/map_points

   This topic publishes messages of the class PointCloud and also runs at a frequency of 30 Hz. The class consist of a vector of points of the class Point32, all having x, y, and z coordinates containing 32 bit data.

Calculation

For the calculation of the pose and map points, the section 2.1.3 can be referred to.

3.3.3 Alignment Node

As mentioned in 3.3.1 gazebo provides the true positions of all models present in the gazebo world. Most importantly, this includes the pose of the drone. In order to transform the point cloud that is computed by the ORB SLAM algorithm to the reference of the gazebo world, the estimated position by ORB SLAM and the true position are aligned, using the method of Umeyama.

Therefore, this node computes the variables that are needed for the transformation and outputs the resulting transformation matrix, translation vector and scale.

Input

1. /ardrone/true_position

   In order to align the trajectories, the true position of the drone is needed. While gazebo provides the respective data in the /gazebo/model_states topic, the model positions do not include a timestamp. Because a timestamp is needed for the alignment in order to only align the matching positions, another node was created to add a timestamp to the gazebo output positions. This node subscribes to the /gazebo/model_states topic and provides the data with a timestamp. To keep the time error that results in reading in the topic data as small as possible, the node runs at an quite high frequency of 100 Hz.

Because the node is very simple and only executes the task of stamping the true positions, it is not listed in this chapter.

2. /orb/pose

   To align the ground truth position and the estimated position of ORB SLAM, the /orb/pose topic published by the ORB SLAM node described in section 3.3.2 is subscribed to by the node.

Output

1. /scale

   The node publishes the scale computed with the method of Umeyama to the /scale topic.

2. /rotation_matrix

   The node publishes the rotation matrix as a flat numpy array computed with the method of Umeyama to the /rotation_matrix topic.

3. /translation

   The node publishes the scale as a flat numpy array computed with the method of Umeyama to the /translation topic.

Real-life framework

In case the framework is tested in real life, as the /ardrone/true_position topic will be unavailable, the node subscribes to the /drone_position_init topic published by the position estimation node instead.

Computations

If all necessary nodes are up and running, the data logged to /orb/pose at 15Hz and to /ardrone/true_position at 50 Hz is stored in two lists. This is done at a frequency of 10 Hz. Before aligning the points in the lists, it is waited until 50 points are logged to each list. If only one list has reached the length of 50, new elements are stacked on top, while the oldest are removed.

Then, the lists are culled in a way, that the minimum and maximum of the timestamp align. This is done in order to save unnecessary resources in the following computations. Since the lists now may be of different length because they origin from topics with different logging frequency, for the shorter list, for each element the element from the other list with the smallest time difference is matched. This ensures that the points estimated by ORB SLAM and from true position are measured at the same time.

Finally, the points are aligned by using the method of Umeyama, as described in section 1 and the scale, the rotation matrix and the translation vector are published to the topic.

These computational steps are processed in the main callback loop of the respective file for the node. The function of the main loop is shown in listing 2 in order to provide further clarification of the computation to the reader.

Listing 2: Main part of the scale estimation node

```
1 def update_trans_variables(self):
2   # return, if not enough points are available
3   if (len(self.est_pos_orb) < 50) or (len(self.true_pos)
      < 50):
4         return
5   else:
6     # get minimum and maximum time for each queue to
```

```
        figure out,
7       # how many points can be considered for alignment.
8       min_orb = np.min([pose_oi.header.stamp.to_sec() for
           pose_oi in self.est_pos_orb])
9           max_orb = np.max([pose_oi.header.stamp.to_sec()
               for pose_oi in self.est_pos_orb])
10
11      min_true = np.min([point_oi.header.stamp.to_sec() for
            point_oi in self.true_pos])
12          max_true = np.max([point_oi.header.stamp.to_sec()
                for point_oi in self.true_pos])
13
14      thresh_min = np.max([min_orb, min_true])
15      thresh_max = np.min([max_true, max_orb])
16
17      # cut off the queues
18      orb_oi = [pose_oi for pose_oi in self.est_pos_orb if
           pose_oi.header.stamp.to_sec() > thresh_min]
19      true_oi = [pose_oi for pose_oi in self.true_pos if
           pose_oi.header.stamp.to_sec() > thresh_min]
20
21      # for the shorter remaining queue, get the matching
           point
22      if len(orb_oi) <= len(true_oi):
23              orb_oi_final = orb_oi
24              true_oi_final = []
25          for pose_oi in orb_oi:
26              diffs_oi = [np.abs(pose_oi.header.stamp.
                   to_sec() - point_oi.header.stamp.to_sec())
                    for point_oi in true_oi]
```

```
27              true_oi_final.append(true_oi[diffs_oi.index(
                    min(diffs_oi))])
28
29          else:
30                  true_oi_final = true_oi
31                  orb_oi_final = []
32                  for pose_oi in true_oi:
33                          diffs_oi = [np.abs(pose_oi.header
                                .stamp.to_sec() - point_oi.
                                header.stamp.to_sec()) for
                                point_oi in orb_oi]
34                          true_oi_final.append(true_oi[
                                diffs_oi.index(min(diffs_oi))
                                ])
35
36          # now do the alignment and compute the scale
37          x_orb = [pose_oi.pose.position.x for pose_oi in
                orb_oi_final]
38          y_orb = [pose_oi.pose.position.y for pose_oi in
                orb_oi_final]
39          z_orb = [pose_oi.pose.position.z for pose_oi in
                orb_oi_final]
40
41          x_true = [pose_oi.pose.position.x for pose_oi in
                true_oi_final]
42          y_true = [pose_oi.pose.position.y for pose_oi in
                true_oi_final]
43          z_true = [pose_oi.pose.position.z for pose_oi in
                true_oi_final]
44
```

```
45          orb_points = np.column_stack((x_orb, y_orb, z_orb
               ))
46          true_points = np.column_stack((x_true, y_true,
               z_true))
47          s, R, t = align_umeyama(true_points, orb_points)
48          R = R.reshape([9,])
49
50          # finally publish the computed scale, matrix and
               vector
51          if s>0:
52                  self.scale_publisher.publish(Float64(s))
53          if sum(np.isnan(R)) == 0:
54                  self.rot_publisher.publish(R)
55          if sum(np.isnan(t)) == 0:
56                  self.trans_publisher.publish(t)
```

3.3.4 Position Estimation Node

This node is only relevant for the real life framework. There, since no ground truth is available for the position of the drone, the position estimation node approximates the position of the drone based on the velocity on the drone and the latest position. ORB SLAM, used in the visual monocular mode is, as mentioned, not able to extract the true scale of the environment. Estimating the true position enables us to also scale the computed point cloud by the ORB SLAM node to its true scale.

This method should be done in the initialization process by only doing translational movements with the drone, such as a takeoff and a short forward movement. No rotations should be performed with the drone since the drone's navigation data, such as the velocity, uses the body-frame as reference frame.

Thus, doing rotations would result in an incorrect estimation of the position.

While relying on the information about the velocity of the drone, the drone needs to have inertial sensors attached. Having these IMU sensors, as the ARdrone 2.0 in the simulation does, also ORB SLAM would benefit from accuracy, since an integration in the algorithm is implemented. However, if the drone does not have these sensors attached, this node can still be beneficial, since it is possible to manually publish messages to the /drone_position_init topic. The initialization process could then be applied by manually flying the drone one meter up, and one meter forward and logging these points to the system. Then the automation process could be started.

Following an overview over the input, output and computational functionality of the node is given.

Input

The node subscribes to the following topics:

1. /ardrone/navdata

   This topic is being published with the TUM simulation node described in section 3.3.1. This node only uses the velocity vectors $v_x \in \mathbb{R}, v_y \in \mathbb{R}, v_z \in \mathbb{R}$ given in the unit mms^{-1} included in the published navdata messages.

2. /drone_position_init

   The drone also subscribes to the /drone_position_init topic, published by itself. This topic includes messages of the class PointStamped. This class includes the x, y and z coordinate of the point itself, and a timestamp. The node also needs the information from this topic to read in the last position and update it based on the velocity by doing the computations explained in the following section. The x, y and z coordinates are transformed to

meter.

Output

The node publishes to the following topics:

1. /drone_position_init

   This topic is explained in the upper section. The updated points are published in this topic.

Computation

As mentioned, the computation is made based on the current velocity

$$v_i = \begin{pmatrix} v_{i,x} \\ v_{i,y} \\ v_{i,z} \end{pmatrix} \in \mathbb{R}^3$$

and the latest position point

$$x_{i-1} = \begin{pmatrix} x_{i-1,x} \\ x_{i-1,y} \\ x_{i-1,z} \end{pmatrix} \in \mathbb{R}^3$$

. Also, the time difference in seconds to the last point is extracted, which is easy, since all objects from the class PointStamped can be timestamped. The difference is given by $\Delta t_i = t_i - t_{i-1}$. The updated position is yielded by

$$x_i = x_{i-1} + \frac{\Delta t_i * v_i}{1000}.$$

Dividing by 1000 yields the unit meter.

This recursive methodology is shown in figure 20

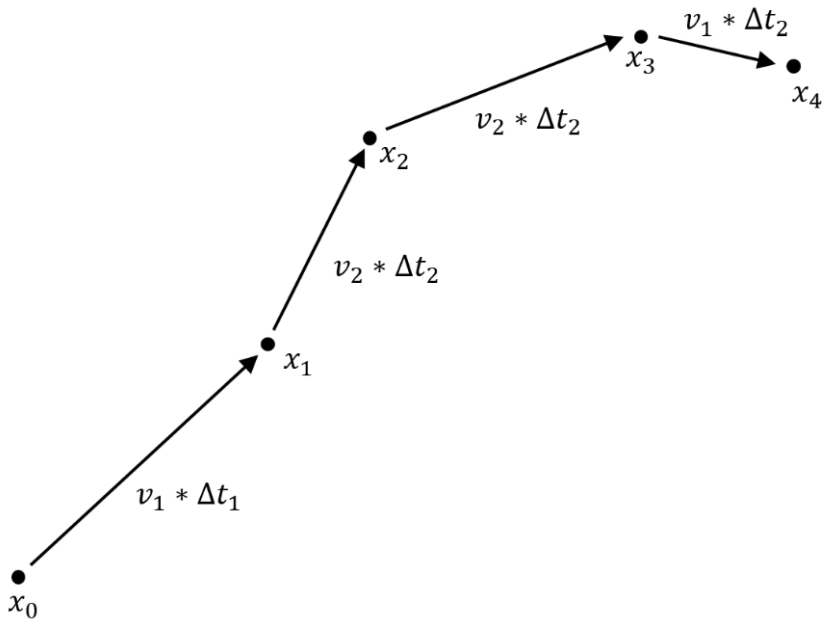


Figure 20: Calculation method for estimation of the position in the initialization process in order to find the true scale.

The implementation is easily deployed and the update function that is running in the main loop can be seen in listing 3.

Listing 3: Main part of the position estimation node

```
1         def update_position(self):
2 # get velocity in x, y and z direction
3 x_vel = self.navdata.vz
4 y_vel = self.navdata.vy
5 z_vel = self.navdata.vz
6
7 if x_vel is not None and y_vel is not None and z_vel is
     not None:
8         # get time difference
9         curr_time = rospy.Time.now()
10        time_diff = (curr_time - self.position.header.
```

```
            stamp).to_sec()
11
12          # create the new point
13          new_point = PointStamped()
14          new_point.header.stamp = curr_time
15          new_point.header.frame_id = "init"
16
17          # update positions
18          new_point.point.x = self.position.point.x +
               time_diff * x_vel / 1000
19          new_point.point.y = self.position.point.y +
               time_diff * y_vel / 1000
20          new_point.point.z = self.position.point.z +
               time_diff * z_vel / 1000
21
22          # publish
23          self.position_publisher.publish(new_point)
24          rospy.sleep(0.1)
```

3.3.5 Transformation and Restriction Node

This node processes the scale, rotation matrix and translation vector computed in the alignment node and transforms the ORB SLAM pose and the point cloud computed by ORB SLAM into the reference frame of the gazebo world. In addition, constrains are added to the searching space of the path finding algorithm by adding walls of points in the resulting point cloud of ORB SLAM.

These processes make sense for many reasons. On the one hand, transforming the ORB SLAM output in the gazebo world creates the possibility to compare the estimated position of the ORB SLAM algorithm to the ground truth position,

as it also has been done in the evaluation of the vSLAM algorithms. While it has not yet been managed to extract the ground truth point cloud of the gazebo world, as described in section 3.4.1, it is technically possible to also compare the ground truth point cloud to the generated one. On the other hand, it also enables users to track the proposed path by the path planning algorithm relative to its surrounding in the future.

The restriction of the searching space is useful because the aim of the path planning algorithm is to explore unseen areas in the searching space. Obviously, since the path planning algorithm relies exclusively on the point cloud, unseen areas are then defined as areas, where no points are available. The gazebo world, displayed in figure 21, exceeds in unlimited space and therefore will cause the path finding algorithm to navigate in infinite space. Also, the ground, sides and sky of the world have no texture, and will make it impossible to find features and therefore generate points for the ORB SLAM algorithm. Unfortunately, as described in the issue section 3.4.1, no other world is yet available. Thus, adding points to limit the searching space is crucial. For this reason most navigation algorithm are constructed for indoor navigation.

Input

1. /orb/map_points

   The ORB point cloud topic, described in section 3.3.2 is subscribed to by the node.

2. /orb/pose

   Since the pose is also transformed in the gazebo reference frame, the /orb/pose topic is also used as input.

Output

1. /pointcloud_transformed

   The node publishes data to the topic /pointcloud_transformed with messages of the class PointCloud. Therefore, the point cloud contains the transformed point cloud of ORB SLAM and the points that are inserted to limit the searching space of the flight path finding algorithm. The exact computations are shown in section below.

2. /pose_transformed

   Also, the orb pose is transformed and published in the /pose_transformed topic. The timestamp for the transformed pose is taken from the original pose calculated by ORB, since the alignment process relies on the timestamp, when the pose was computed.

Computation

First, all points p_i in the point cloud and the estimated positions are transformed with the rotation matrix R, scale s and transformation vector t received from the above described topics. The resulting points are therefore computed by:

$$p_i' = sRp_i + t$$

.

Following this, the restriction points are added to the point cloud. The ground plane of the gazebo world has a size of 100x100, but only the center is filled with objects (models). The middle of the plane lies in the exact origin of the gazebo world. The goal is to restrict the search space of the path planning algorithm to the hull of a cuboid with 15m height, 60m length and width. The cuboid is therefore given by:

$$\Omega = \{(x, y, z) \in \mathbb{R} : x \in [-30, 30] \wedge y \in [-30, 30] \wedge z \in [-30, 30]\}$$

.

Then, 10000 points for each side of the hull of Ω are added to the generated point cloud by ORB SLAM, after it has been transformed. This can simply been achieved in three consecutively for loops. This is shown in listing 4 for the upper and lower restrictions. These loops are run for the sides respectively.

Listing 4: Adding upper and lower restrictions to point cloud.

```
1
2  # bottom and top
3  for x_oi in np.linspace(-30, 30, 100):
4          for y_oi in np.linspace(-30, 30, 100):
5                  for z_oi in [0, 15]:
6                          p_out = Point32()
7                          p_out.x = x_oi
8                          p_out.y = y_oi
9                          p_out.z = z_oi
10                         pq.points.append(p_out)
```

Note that for the point cloud the coordinates are only stored in 32 bit to save resources, since the point cloud can contain ten thousands of data-points. In order not to overload the system, the transformation node runs on a frequency of only 5Hz, which results in smooth computation.

3.3.6 Flight Path Planning Node

The fight path planning node is in charge of autonomously navigating the drone. This should be done without colliding with any obstacles. On the other hand, since the goal is to explore the environment, the algorithm should always

thrive to visit new areas in order to generate new map points.

While the computation of this node is not yet implemented and is not part of this work, the desired input and output can be cleanly defined. Also, approaches currently used in for path planning approaches in active SLAM-application are described in the computation section.

Input

1. /pointcloud_transformed

   This topic is explained in the upper section.

2. /pose_transformed

   This topic is explained in the upper section.

3. /pose_transformed

   This topic is explained in the upper section.

4. /gazebo/collision

   For the simulated framework, the node also may subscribe to the /gazebo/collision topic. This enables users to penalize the algorithm in the training process, in case the obstacle avoidance did not succeed.

Output

1. /cmd_vel

   The node should publish navigation commands to /cmd_vel, as shown in example 1.

Computation approaches

While the research in the field of automated exploration algorithms is still in its early years [42] and will probably be mainly performed in simulated environment in the near future [38], some frameworks on tackling the task already exist [42], [38] [1] [43].

In this section, approaches, that the path flying node could be based on are explained.

> Typically, it consists of three stages [8]: (i) the identification of all possible locations to explore (ideally infinite), (ii) the computation of the utility or reward generated by the actions that would take the robot from its current position to each of those locations and (iii) the selection and execution of the optimal action [38].

A general framework for decision making processes within a certain environment was introduced with the Partially Observable Markov Decision Processes (POMDP) framework. The active SLAM problem can be formally defined within this framework. The POMDP framework relies on seven components [44], listed below.

- Set of States S
- Set of Actions A
- Set of conditional transition probabilities $T : \mathbb{P}(s'|s, a)$
- Reward function $R : A, S \rightarrow \mathbb{R}$
- Set of beliefs b
- Set of observations Z

- Set of conditional observation probabilities $O : \mathbb{P}(z|s)$

S defines all possible states, the drone can be in. In our case, a state is defined in the oriantation of the drone, the position of the drone and the collision sensor output. All possible states are contained in a combination of all possible poses $SE(3)$, while the respective translational vector has to be in the cuboid Ω defined in section 3.3.5, and all possible collision states, which is equal to $\{0, 1\}$. The current state can be extracted by the subscribed topics of this node.

All possible actions are all possible control commands, that can be published to the /cmd topic. This includes rotational and translational maneuvers. A possible command in order to make the drone fly foreward can be seen in listing 1.

The function T returns the probability, a desired state s' is accessed by taking an action a in the state s.

The reward function R returns a reward in the form of a real number for an action taken in a state. In our case, the reward function should be defined in a way that actions resulting in a great amount of new observed points should be rewarded greatly, while actions, that result in no new observed features or even a crash, should be given little, no or negative reward. Defining a good reward function is crucial in order for the algorithm to work properly. An example reward function for our case could look like this:

$$R(a, s) = \begin{cases} n & \text{for } n \text{ newly explored points by ORB SLAM, after action a in state s} \\ -15 & \text{if tracking is lost, after action a in state s} \\ -30 & \text{if the drone collided, after action a in state s} \end{cases}$$

The belief b defines the probability for every state, that the drone actually is in this state. Therefore all probabilities should add up to one. Not all algorithms require b [45].

The set of observations Z equals the current pose and and current map points. Finally, O returns the probability to make an observation z while being in the state s. The POMDP is looping the following steps: Take action based on belief state, take observations and update belief state. The goal of the POMDP framework to create a policy π that maps states into actions. This policy is optimal, when it maximizes the sum of expected reward in the future. There are several methods to find a policy, given the above components. In this paper, some of those methods are discussed.

1. Reinforcement learning

   Unlike most other machine learning algorithm, reinforcement learning does not require training data for learning behavior. The learning mechanism is only based on the reward, that is given for an action. Q-learning approaches do not need the definition of O and T. This is why Q-learning is defined as a model-free machine learning process [38].

   Q-learning is a method of reinforcement learning. Approaches on applying it for robot navigation exist [1][38][43]. Q-learning methods define a recursive function Q that is based on the accumulated reward for a series of actions. If a drone is in a state s, the action a' is performed, that maximizes Q [43].

$$a' = \pi(s) = \arg\max_{a} Q(s, a)$$

   At each iteration, the value of $Q(s, a)$ is updated with the following term:

$$\Delta Q(s, a) = \alpha(R(a, s) + \gamma \arg\max_{a'} Q(s', a') - Q(s, a)). \tag{3.1}$$

   γ is called the discount factor and is a value between 0 and 1 and defines, how much weight is given to the reward for steps that are further away from the current one. For example, if $\gamma = 0.8$, the third term of the accumulated sum is only weighted with $0.8^3 = 0.512$. Thus, the more

reliable the algorithm and the chosen reward function is, the higher γ can be chosen, and therefore the better the algorithm plans the navigation into the future.

The exact computational steps are shown in algorithm 1. First, the learning rate α and the discount function γ have to be predefined. One possibility to find the optimal values for α and γ is to perform a grid search. For the grid search and for the Q-learning algorithm also an mapping function has to designed, that maps the point cloud to a value that determines, how far the target exploration process has proceeded. This function could for example be examining the distribution of the point cloud within the searching space.

Also, a number of episodes to train the models has to be defined. This is equal to the number of runs that are performed until the agent, in our case the drone, gets to a terminal state. A terminal state is a state, where the agent is finished. In our case, that would be equal to the state, where the drone collided, or a termination constrain has been hit when the exploration task has been successfully finished. This termination constrain could for example be a reached threshold of the previous defined mapping function.

Then in the algorithm's while loop, in each episode, the algorithm iterates, by updating the Q-table, until a terminal state is reached. The Q-table is a matrix, that consists of all possible actions as columns and all possible states as rows. The Q-table is updated after every action by using the

update rule stated in equation 3.1.

Data: Parameters: $\alpha \in (0,1], \gamma \in (0,1]$, Initialize Q-table with arbitrary Q-values

for *episode* $\leftarrow$ 1 *to max episode* **do**

  Percieve s_t ;

  while s_t *not terminal* **do**

    select $a_t = \pi(s_t)$;

    Take a_t, Get $R(a_t, s_t)$, percieve s_{t+1};

    if s_{t+1} *is terminal* **then**

      $Q_t \leftarrow R(a_t, s_t)$;

    else

      $Q_t \leftarrow R(a_t, s_t) + \gamma \arg\max_{a'} Q(s_{t+1}, a')$

    end

    $Q(s_t, a_t) \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha Q_t$;

    $s_t \leftarrow s_{t+1}$;

  end

end

Algorithm 1: Q-learning algorithm. Source: [38]

In the recent years, successes were made in the appliance of deep reinforcement learning algorithms for robotic navigation and exploration [38] [1]. These approaches do not analytically compute the output of Q but rather a deep neural network is created in order to predict the output. Since for algorithm 1, in each step each action is calculated recursively, the computation requires a lot of computational resources. By approximating the function Q, this is avoided.

Also research has shown, that the trained models also yield good performance for environments, where they were not trained and have no a priori knowledge.

The results of [38] also shows, that the trajectory accuracy can be increased by autonomously navigating the drone.

> D3QN results are the most remarkable in the first environment, as it outperforms the reward that a human would obtain by manually controlling the robot (approx. 350). In the second environment both DDQN and D3QN show a good behavior. Despite DDQN have higher SR (and mean steps, thus), the higher mean reward obtained by D3QN proves the generation of more optimal trajectories: smoother movements and less spins. [38]

D3QN is also a path planning algorithm based on deep reinforcement learning proposed by Wen et al. in 2020 in their work "Path planning for active SLAM based on deep reinforcement learning under unknown environments" [1].

2. Other approaches

   Aside from the above-discussed reinforcement learning methods, other approaches to tackle the path planning task exist.

   In their work "Active SLAM and Exploration with Particle Filters Using Kullback-Leibler Divergence" [46], Carlone et al. suggested a framework to explore the three dimensional space, by feeding target locations to the drone. Therefore, the developers introduced so-called trajectory targets, which are points on the trajectory, to allow the drone to revisit places. This is done to increase accuracy and uncertainty. Furthermore, frontier targets are defined. These are points, that lie on the borders of the explored space and should be targeted in order to increase the map. When no more frontier targets exists in the map, the exploration task can considered to be finished.

   Back in 2006, Leung et al. published their work "Active SLAM using Model Predictive Control and Attractor based Exploration" [47]. They propose to navigate the robot by using model predictive control. The method minimizes a predefined cost function for N control actions into

the future. Then, the first action is performed. These steps iterate, while always shifting the control horizon into the future.

3.4 Evaluation of the Proposed Framework

Currently the framework is set up in an environment provided by theconstructsim.com. This platform is enabling ROS-developers to program in preconfigured ROS-environments. The environment comes with the possibility to open terminal consoles, a file management system and a gazebo simulator that automatically detects, when a gazebo simulation is running. Also, you have a graphical interface for other graphical applications, such as the viewer of ORB-SLAM. The current environments is set up with ROS kinetic and Ubuntu 16.04.6 LTS (Xenial). The tum_simulator, ORB SLAM and all of their dependencies were installed and compiled. The provided machine consists of 16 processing kernels and contains a total RAM space of 29 GB. The hard drive offers 92 GB of available space. However, theconstructsim.com limits each user to 8 hours daily on the platform.

The project is publicly available under the name tum simulator test.

Listing 5: Launching the simulated environment

```
1
2 # launch the gazebo simulation
3 roslaunch cvg_sim_gazebo ardrone_testworld.launch
4
5 # launch ORB-SLAM
6 ./run_orb_node.sh
7
8 # start timestamp node, to stamp the positions published
    by gazebo
```

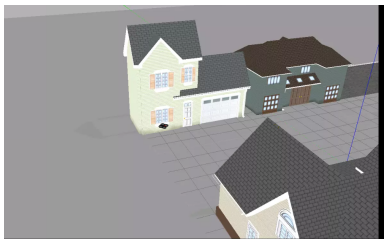
```
9  rosrun auto_explorer timestamp_adder.py
10
11 # publish the rotation matrix, scale, and translation
       vector (alignment node)
12 rosrun auto_explorer transformator_variable_updater.py
13
14 # start transformation and restriction node
15 rosrun auto_explorer position_updater.py
16
17 # takeoff with drone
18 rostopic pub -1 /ardrone/takeoff std_msgs/Empty
19
20 # Then start flight path planner algorithm
```

In listing 5 the commands for launching the gazebo simulation, the ORB SLAM node, all other nodes and the drone are displayed. After launching those applications, only the path planning algorithm node based on the resulting point cloud is missing. However, multiple solutions for such algorithms exist [48], applying it on the system is not part of this paper and will be done in further research.

In figure 21 the simulated environment with gazebo can be found in figure a. Here, the drone (in black) is flying in front of the building. The output of the front camera is shown in figure b. In figure c, the ORB SLAM algorithm was applied to the output of the front camera. Green dots represent the finding of a ORB-features.

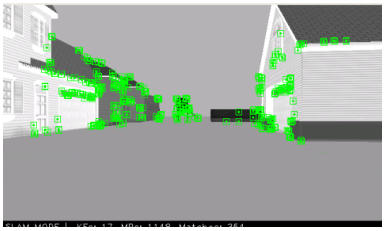
3.4.1 Known Issues

1. Only one world available



(a) gazebo simulation

(b) front camera simulation



(c) ORB applied on simulation

Figure 21: The drone in a gazebo simulation in a), the output of the front camera of the drone in b) and the ORB SLAM algorithm applied on the front camera output in with the detected ORB features marked green c).

For the tum_simulator it might be useful to continue the automation process in another simulated world. This is because the current world named ardrone_testworld does not contain any contours or relief on the ground, and sky, as shown in figure 21. Since the ORB SLAM algorithm is looking for features such as edges and changes in pixel intensities, it will not find any on the ground, which will result in no points available in this area for the resulting point cloud. While there are many worlds available in the tum_simulator package, since the package is originally not made for ROS kinetic, these worlds will not compile and running the command to start one of those worlds, as shown in listing 6 will result in the following error:

```
ERROR: cannot launch node of type [gazebo/spawn_model]:  gazebo.
```

So far, no solution has been detected, but one possible workaround would either be to build another world from scratch. Another possible solution would be to add flying constrains to the path planning algorithms, that

limits the environments on a predefined volume. This solution is implemented by the transformation and restriction node, described in section 3.3.5, which adds restriction points to the point cloud.

Listing 6: Launching different world

```

# launch different world named land_station1
roslaunch cvg_sim_gazebo land_station1.launch
```

2. No ground truth point cloud

   While the pose and position of the models in the gazebo world, such as the drone itself, the houses and other objects, are known, it was not yet possible to convert these objects into point clouds.

   This refuses the possibility to compare the evaluated points by the ORB algorithm in the ROS setup, to their true position. Users of the setup now have to solely rely on the results of evaluation described chapter one. However, the trajectories can still be compared.

3. Collision sensor not working

   Gazebo provides the possibility to attach collision sensors to models existing in the world. These sensors can simply be defined in the respective .XML file of the model. However, even after the implementation and no error message, the respective topic, where the data should be published, does not appear when calling the rostopic list command. So far, no solution could be found.

4. Summary

In the context of automated exploration and mapping of a three dimensional environment with drones, in this work we answered two distinct research questions.

In order to examine what the most suitable open-source monocular vSLAM algorithm to be used for an automated exploration task is, ORB-, DSO- and DSM SLAM were evaluated regarding predefined criteria. The results of the evaluation, that were yielded after applying these algorithm on the benchmark EuRoC dataset, exhibit, that with respect to the trajectory accuracy, the point cloud accuracy and the computation time, ORB SLAM outperforms the other vSLAM algorithms. Although ORB SLAM generates significantly less points compared to DSO- and DSM-SLAM for the resulting point cloud, ORB SLAM reveals a computation speed at least three times as high as both other methods. ORB SLAM also shows better results regarding tracking accuracy and point cloud accuracy.

Therefor, the proposed framework to answer the second research question, which targets to find a suitable framework to test and develop fully automated exploration systems within a simulated environment, relies on ORB SLAM to be a main component. The proposed framework is implemented in ROS and by using ORB SLAM, a gazebo simulation and further nodes, implemented specifically for this purpose, it is possible to develop and test flight path planning algorithms to complete the automated exploration and mapping task. In

the current setup, a drone can be navigated in a simulated environment while the ORB SLAM algorithm is applied on the drones front camera output. By transforming the output of the ORB SLAM algorithm to the reference frame of the gazebo simulation, users of the framework are enabled to further develop an automated exploration system by implementing the fight path planning algorithm within the framework.

Bibliography

[1] S. Wen et al. Path planning for active slam based on deep reinforcement learning under unknown environments. 2020.

[2] Stephan Weiss, Davide Scaramuzza, and Roland Siegwart. Monocular-slam-based navigation for autonomous micro helicopters in gps-denied environments. *J. Field Robotics*, 28:854–874, 11 2011.

[3] B. Hiebert-Treuer. An introduction to robot slam (simultaneous localization and mapping). 2015.

[4] C. Debeunne et al. A review of visual-lidar fusion based simultaneous localization and mapping. 2020.

[5] Cindy Leung, Shoudong Huang, and Gamini Dissanayake. Active slam using model predictive control and attractor based exploration. pages 5026 – 5031, 11 2006.

[6] M. Burri. The euroc micro aerial vehicle datasets. *The International Journal of Robotics Research*, 2016.

[7] W. Burgard et al. Introduction to mobile robotics.

[8] J. Engel et al. Direct sparse odometry. 2016.

[9] R. Mur-Artal. Orb-slam: a versatile and accurate monocular slam system. *IEEE TRANSACTIONS ON ROBOTICS*, 2015.

[10] H. DURRANT-WHYTE et al. Simultaneous localization and mapping: Part i. 2006.

[11] R. Smith et al. On the representation and estimation of spatial uncertainty. *The International Journal of Robotics Research*, 1986.

[12] T. Taketomi et al. Visual slam algorithms: a survey from 2010 to 2016. *IPSJ Transactions on Computer Vision andApplications*, 2017.

[13] Andrew J. Davison, Ian D. Reid, Nicholas D. Molton, and Olivier Stasse. Monoslam: Real-time single camera slam. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1052–1067, 2007.

[14] G. Klein et al. Parallel tracking and mapping for small ar workspaces. 2007.

[15] A. Huletski et al. Evaluation of modern visual slam methods. 2016.

[16] Oliver Roesler and Vignesh Padubidri Ravindranath. Evaluation of slam algorithms for highly dynamic environments. 2020.

[17] J. Wang et al. Mapping quality evaluation of monocular slam solutions for micro aerial vehicals. 2019.

[18] R. Kummerle et al. On measuring the accuracy of slam algorithms.

[19] Eldar Mingachev, Roman Lavrenov, Evgeni Magid, and Mikhail Svinin. *Comparative Analysis of Monocular SLAM Algorithms Using TUM and EuRoC Benchmarks*, pages 343–355. 09 2020.

[20] J. Zubizarreta et al. Direct sparse mapping. 2019.

[21] H. Strasdat. Visual slam: Why filter? *Image and Vision Computing*, 2000.

[22] E. Eade. Lie groups for computer vision. 2002.

[23] E. Rublee. Orb: an efficient alternative to sift or surf. 2012.

[24] B. Triggs. Bundle adjustment – a modern synthesis. *International Workshop on Vision Algorithms*, 2000.

[25] C. Campos et al. Orb-slam3: An accurate open-source library for visual, visual-inertial and multi-map slam. 2021.

[26] S. Umeyama. Least-squares estimation of transformation parameters between two point patterns. *IEEE Trans. Pattern Anal. Mach. Intell.*, 1991.

[27] Z. Zhang et al. A tutorial on quantitative trajectory evaluation for visual(-inertial) odometry. 2009.

[28] Keisuke Tateno, Federico Tombari, Iro Laina, and Nassir Navab. Cnn-slam: Real-time dense monocular slam with learned depth prediction. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6565–6574, 2017.

[29] N. Yang, L. von Stumberg, R. Wang, and D. Cremers. D3vo: Deep depth, deep pose and deep uncertainty for monocular visual odometry. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[30] Piotr Dollar Kaiming He, Georgia Gkioxari and Ross Gir-´ shick. Mask r-cnn. 2017.

[31] Ross Girshick Carsten Rother Alexander Kirillov, Kaiming He and Piotr Dollar. Panoptic. Panoptic segmentation. 2019.

[32] Ross Girshick Shaoqing Ren, Kaiming He and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. 2015.

[33] Eddy Ilg Philip Hausser Caner Hazirbas Vladimir Golkov Patrick van der Smagt Daniel Cremers Alexey Dosovitskiy, Philipp Fischer and Thomas Brox. Flownet: Learning optical flow with convolutional networks. 2015.

[34] Peiliang Li Tong Qin and Shaojie Shen. Vins-mono: A robust and versatile monocular visual-inertial state estimator. 2018.

[35] Tomasz Malisiewicz Daniel DeTone and Andrew Rabinovich. Superpoint: Self-supervised interest point detection and description. 2018.

[36] Tomas Pajdla Marc Pollefeys Josef Sivic Akihiko Torii Mihai Dusmanu, Ignacio Rocco and Torsten Sattler. A trainable cnn for joint detection and description of local features. 2019.

[37] C. Sampedro et al. A fully-autonomous aerial robot for search and rescue applications in indoor environments using learning-based techniques. 2019.

[38] J. Placed et al. Aa deep reinforcement learning approach for active slam. 2020.

[39] Guillaume Pepe, Massimo Satler, and Paolo Tripicchio. Autonomous exploration of indoor environments with a micro-aerial vehicle. 11 2015.

[40] L. Dowling et al. Accurate indoor mapping using an autonomous unmanned aerial vehicle (uav). 2018.

[41] Chenping Li, Xuebo Zhang, Haiming Gao, Runhua Wang, and Yongchun Fang. Bridging the gap between visual servoing and visual slam: A novel integrated interactive framework. *IEEE Transactions on Automation Science and Engineering*, pages 1–11, 2021.

[42] S. Chaves et al. Opportunistic sampling-based planning for active visual slam. 2013.

[43] E. Lopez et al. An active slam approach for autonomous navigation of nonholonomic vehicles. 2013.

[44] W.D. van den Hof. Robot search in unknown environments using pomdps. 2014.

[45] V. Thomas et al. Monte carlo information-oriented planning (revised version). 2020.

[46] L. Carlone et al. Active slam and exploration with particle filters using kullback-leibler divergence. 2014.

[47] Cindy Leung, Shoudong Huang, and Gamini Dissanayake. Active slam using model predictive control and attractor based exploration. pages 5026 – 5031, 11 2006.

[48] A. Gasparetto et al. Path planning and trajectory planning algorithms: a general overview. 2015.