Autonomous modeling of an 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 : Robert Kathrein, MSc

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

Sperrvermerk

Ich habe die Sperrung meiner Masterarbeit beantragt, welche von der Studiengangsleitung genehmigt wurde.

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

JS JavaScript

FH Kufstein Tirol
Data Science & Intelligent Analytics

Abstract of the thesis: **Autonomous modeling of an 3D environment with drones**

Author: Julian Bialas, BSc
First reviewer: Prof. (FH) PD Dr. Mario Döller
Second reviewer: Robert Kathrein, MSc

06. July 2020

FH Kufstein Tirol

Data Science & Intelligent Analytics

Kurzfassung der Masterarbeit: **Autonomous modeling of an 3D environment with drones**

Verfasser: Julian Bialas, BSc

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

Zweitgutachter: Robert Kathrein, MSc

06. July 2020

1. Introduction

Multiple applications exist for the autonomous exploration and mapping tasks for drones; such as search and rescue-, inspection- and surveillance operations [20].

The autonomous exploration task can can be devided into three subproblems: localization, mapping and path planning [21]. All three tasks should be performed simulataniously within an anvorinment, 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 incrmentional creation of a 3-demensional map. Multiple methods exist, that combine these two tasks in a so called simulatanious localization and mapping (SLAM) algorithm. The development of these SLAM algorithms is one of the most researched topics in the field of robotics [8].

> SLAM is used for many applications including mobile robotics, self-driving cars, unmanned aerial vehicles, or autonomous underwater vehicles [3].

When combining a SLAM algorithm with with a path planning algorithm, an autonomous exploration system is created. The autonomous exploration using SLAM and a path planning algorithm is sometimes also refered to as active SLAM [9]. This active SLAM process is displayed in figure 14. Details

on how such an system can be initiated and terminated, can be found in section 3.2.

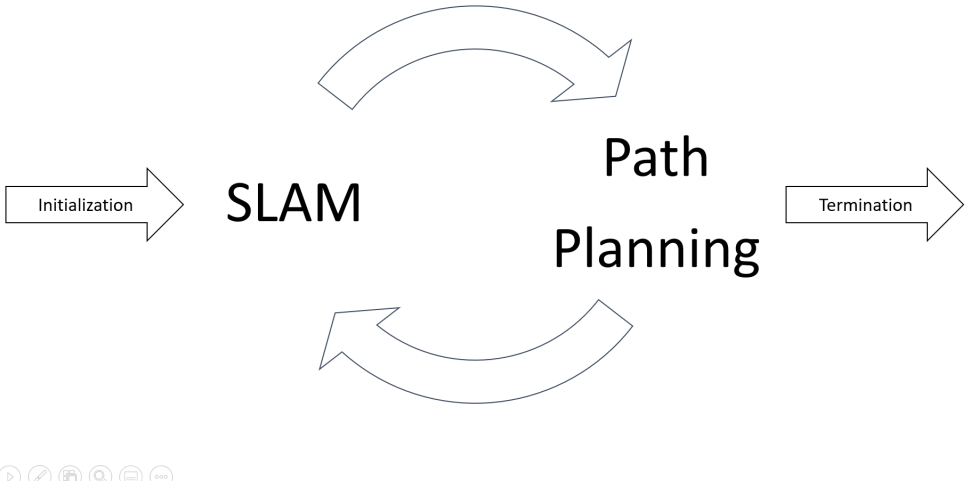


Figure 1: Automated exploration system, based on a SLAM algorithm and a path planning algorithm.

This work targets to answer two destinct research questions.

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

   This paper is limited to evaluate monocular 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, for this first part of this work, three open source monocular visual SLAM algirithms 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 pointcloud accuracy and computational speed.

This was done by using the publicly available benchmark EuRoC dataset [1], containing video sequences filmed by a drone, the groundtruth of the position of the drone and the pointcloud of the environment. Thus, the three SLAM algorithms are applied on the video sequences, the resulting trajectories and map points are compared to the groundtruth regarding the above mentioned criteria.

2. How can an framework be set up, where fully automated exploration systems can be tested and developed within a simulated environment?

   In the second part of this work, a Roboter Operating System (ROS) framework, that enables users to develop an fully automated exploration system within a virtual environment is suggested. 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 bahavior of the drone are modeled realisticly. Most importantly, the drone is equipped with a RGB camera, making it possible to directy apply 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 are given in section 3.2.8, this work does't include an actual implementation of such an algorithm. Such implementations are left for further research.

   The suggest 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 be running can be provided with all necessery data, such as sensor data of the drone and the estimated orientation

position and pointcloud 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 and Selection of vSLAM Algorithm

2.1 Related Work

2.1.1 Introduction and History of SLAM

SLAM is one of the most emerging research topics in robotics [8]. It is applied in various applications, such as in augmented reality to estimate the camera pose, autonomous navigation and computer vision-based online 3D modeling [16] [6]. The problem can be defined in a probabilistic way. The goal is to compute

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

Where, as can be found in figure 2, m_{t+1} is the map (pointcloud of the surroundings) at timepoint $t+1$, x_{t+1} the camera pose and position at timepoint $t+1$, $z_{1:t+1}$ all observations made to this timepoint and $u_{1:t}$ all historic control input. However, most modern SLAM methods don't require the control input anymore.

With their work on the representation and estimation of spatial uncertainty [14] Smith et al created the first relevant work in the field on SLAM in 1986.

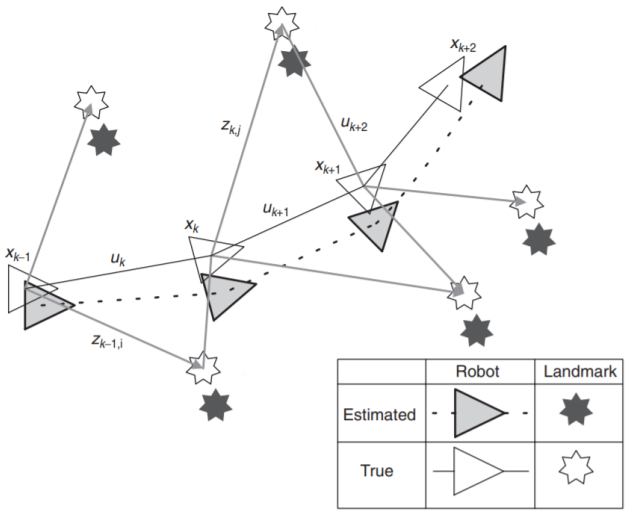


Figure 2: Overview of the SLAM concept. Source: [4]

However, due to lacking computational recources, the engagement in this topic stayed aminly on a theoretical level.

> The result of this conversation was a recognition that consistent probabilistic mapping was a fundamental problem in robotics with major conceptual and computational issues that needed to be addressed. [4]

This is because already then it was recognized, that camera pose and landmark positions had to be updated when moving further or optimizing the map and is a huge computational effort as the map grows.

As time progressed, so did the computational resources and the algorithms became more advanced. Huge evolvements were achieved after 2010 [16], mainly because of the increased use of augmented reality application, that may rely on real time vSLAM algorithms. The developed vSLAM methods can be devidet into two different classes: direct and feature based methods.

Direct algorithms use the entire image as input in order to compute the term in quotation **??**. Therefore, the term is computed by optimizing the photometric

error. This is the error, that results from comparing the intensities of each pixel after transformation (optimization) [6].

> 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). [6]

Feature based methods on the other hand, compute features for each frame, that serve as imput for further computation. These features usually are subsets of pixels, that have remarkable intensities and arrangement, such as corners (landmarks). These methods evaluate the upper term by computing the geometric error, since the feature positions are geomtric quantities. A main advantage of feature based methods is the rubostness over geometric distortions present in every-day-cameras [6].

Of all existing vSLAM algorithms, this work considers DSO, DSM and ORB SLAM for the evaluation of beeing a suited candide for flying a drone autonomously. This desicion is based on other research results in this area.

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

2.1.2 Definitions

Keyframe

Most vSLAM Algorithms make usage of so called keyframes, as these keyframe-based approaches have proven to be more accurate [15]. Keyframes are specific selected images of the input sequences or video streams. In most cases, the

keyframes store all of the existing map points and all of the computations and optimizations are made, based on the keyframes and data stored within them.

Covisibility Graph

updating the edges resulting from the shared map points with other keyframes.

Group of Rigid Transformations in 3D

SE(3) is the group of rigid transformations in 3D space [5]. Each Matrix $T \in \mathbb{R}^{4x4}$ with

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

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

Pointcloud

When we speak of pointcloud, we simply mean a formation of points in the threedimensional space, though having an x y and z coordinate. Later in the evaluation of the algorithms, to save these pointclouds, the PLY format has been chosen.

Camera Calibration

2.1.3 ORB-SLAM

ORB SLAM is a feature based, state of the art slam method. The first version was published in 2015 [11]. Here, an overview of the functionality of ORB SLAM is provided. The Algorithms runs on three threats simultanously. Each thread performes one of the following tasks: Tracking, Local Mapping and Loop Closing. An overview over the tasks can be found in figure 3. The explaination of these system components are described in the following subsections. A more detailed explaination can be found in the paper of Raul Mur-Artal et al [11].

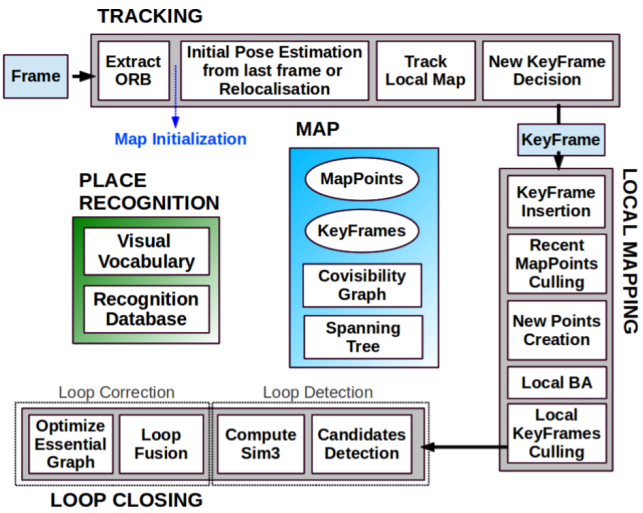


Figure 3: Overview of the system components extracted from [11]

Tracking

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

1. Feature Extracting

Features are extracted using Oriented FAST and Rotated BRIEF [13]. This method starts by searching for FAST (Features from Accelerated and Segments Test). Herefor, for each pixel x in the image, a circle of 16 pixels around that pixels is considered and checked if at least eight of these 16 pixels have major brightness differences. If so, the pixel x is considered as a keypoint, since it is likely to be an edge or corner. This is repeated again and again after downsizing the image up to a scale of eight. To extract features evenly distributed over the image, it is devided 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 a 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).

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, containing keyframes that contain the observed map points and the keyframes from the covisibility graph. The pose is then corrected with all matched mappoints.

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 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. Therefor, the image is saved by the number of occurances of features found in a 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 mappoint is removed from the map, when it is found in more than 25/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 maximim 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 minizing 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 is the respective Keyframe and j the index of the map Point. π_i is a projection function, calculation a transformation to project all keypoints on mappoints by minimizing a cost function, that can be found in [18].

> 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.

At this point, a local BA is performed.

5. Local Keyframe Culling

   With difference to other SLAM algorithm, ORB slam delets redundant keyframes, which decreases computational efforts, since computational complexity grows with the number of keyframes. All keyframes are deleted, where at least 90 percest 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 neighbours 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 computet, to map the map points from K_i on K_l. Since scale can drift, also the scale is computet in addition to the

rotation matrix and translation using the mothod 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 edjes (shared map points) in the covisibilty graph.

4. Essential Graph Optimization

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

2.1.4 DSO-SLAM

Direct Sparse Odometry was developed in 2016 by the Technische Universität München.

Model Overview

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

The camera is calibrated in two different ways. First a projection function is computed, considering the focal length and the pricipal point, 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 attenuations on the other hand.

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

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

,

where $\mathcal{F}$ is the set of all frames, $\mathcal{P}_i$ the set of all pixel in frame i and 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 include comparing the point p, but also computes the pixel intensitie difference of eight pixel neighbors of p, arranged in a spread pattern.

Visual Odometry Front-End

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

1. Frame Managemant

   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 beeing the most recent:

(a) I_1 and I_2 are always kept

(b) keyframes, whos amount 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 distribuited in 3D space are removed.

2. Point Managemant

DSO always tries to keep 2000 active points in the map. The first step is select candidates 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 threshhold, which is computed by adding seven to the overall median of the gradients. This is repeated twice while decreasing the threshhold and doubling the block size, in order to also include points with weaker gradient.

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 distribuited 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 ressources, 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 seperated into a tracking front-end and an optimization back-end that run on parallel threads.

Model Overview

DSM uses the same model as DSO. The model is described in section 2.1.4.

Front-End

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

1. keyframe and mappoint selection

   When selecting keyframes and mappoints, 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. With similar critirias for keyframe selection as DSO, whenever a new keyframe is inserted, anotherone is removed (from the temporal part).

   In order to regard reobserved points, DSM also considers N_c covisibil keyframes to fill the latest keyframe I_0 with map points, favouring map points in depleted areas. This is achived 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 eucleadian distance to the closest

map point is computed. Obvously, large distances suggest depleted areas.

(b) covisibil Keyframe Selection

Select the keyframe within the old keyframes with the most map points in the above selected depleted area. 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 $T_i \in \mathrm{SE}(3)$ that is saved for each keyframe I_i.

(c) Update distance map

Update the distance map, calculated in step 2, with the new selected keyframe.

(d) iterate

Iterate until N_c keyframes are selected for the covisibility part.

The entire keyframe and mappoint selection is displayd in figure 4. In this case, N_t is equal to four and N_c is equal to three.

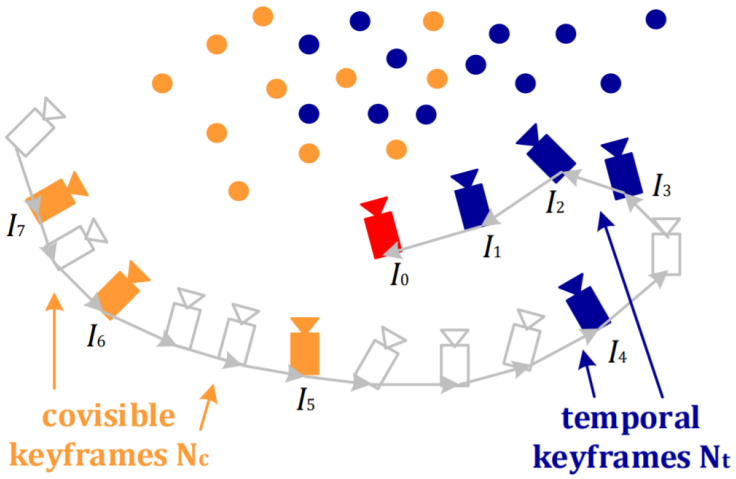


Figure 4: Keyframe and Point Selection of DSM. Source: [23]

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

Frame Tracking, New Keyframe desicion 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 **??**.

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 also brings the 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 covisibilty 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

2.2.1 Dataset

For the evaluation of the vSLAM Algorithms, the EuRoC dataset [1] was used. The dataset contains eleven video sequences, recorded with a micro aerial vehicle at 20 frames per second. The sequences have a resultion 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 at ETH Zürich, and the other six were recorded in a room, that was provided with additional obsticals. For the latter six sequences, the groundtruth of the environment exists as a dense pointcloud, as can be seen in figure 5.

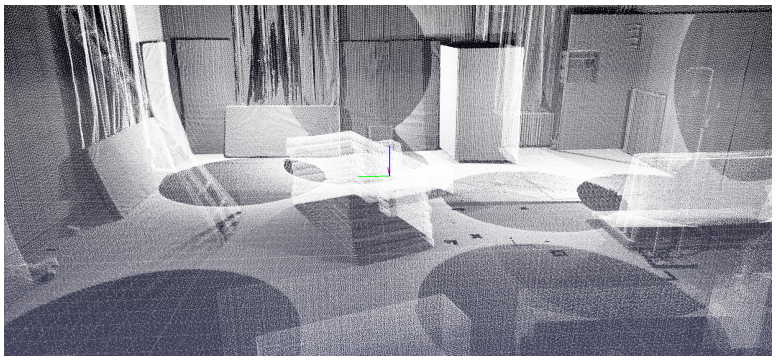


Figure 5: Pointcloud ground truth of sequence V1_01_easy visualized with python package pptk

Finally the true position of the camera is known at a high frequency of over 200 points per second. An overview of the sequences is shown in table 1.

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

Sequence Name	Duration in s	Average Velocity in ms^{-1}	Pointcloud available
MH_01_easy	182	0.44	No
MH_02_easy	150	0.49	No
MH_03_medium	132	0.99	No
MH_04_difficult	99	0.93	No
MH_05_difficult	111	0.88	No
V1_01_easy	144	0.41	Yes
V1_02_medium	83.5	0.91	Yes
V1_03_difficult	105	0.75	Yes
V2_01_easy	112	0.33	Yes
V2_02_medium	115	0.72	Yes
V2_03_difficult	115	0.75	Yes

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 innitilize 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. 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 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

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

.

In other words, the evaluated points are rotated, translated and scaled in a way, that the sum squared error over the point distances is minimized. The upper expresssion is calculated by using the method of Umeyama [19].

Similar to principal component analysis, Umeyama uses the singular value decomposition of the covarianve 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:

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

with

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

σ_p beeing 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 computet after aligning them with the upper method using the computet parameters $S = \{R, t, s\}$, yielding

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

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

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

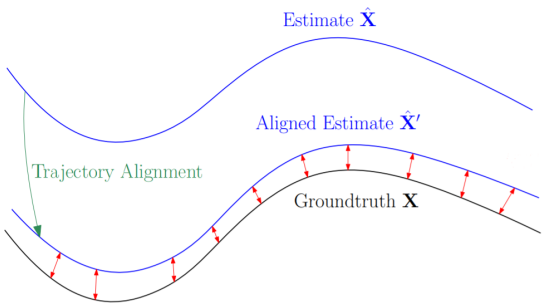


Figure 6: Trajectory error after alignment. Source: [22]

In figure 6 the computet errors are displayed.

These error terms are visualized over time and the overall mean is determined and again visualized with boxplots for each method. 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.

Pointcloud 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 latter six sequences, where a groundtruth of the pointcloud 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 successfull navigation.

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 world reference.

   To compare the the transformed computet 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 evalueated point cloud, the distance to the closest point in the ground truth point cloud is calculated. This is assumed to be the points ground truth position, since with several 100000-points in groundtruth, this point in the ground truth point cloud should not be too far away from the orthogonal projection of the evaluated point.

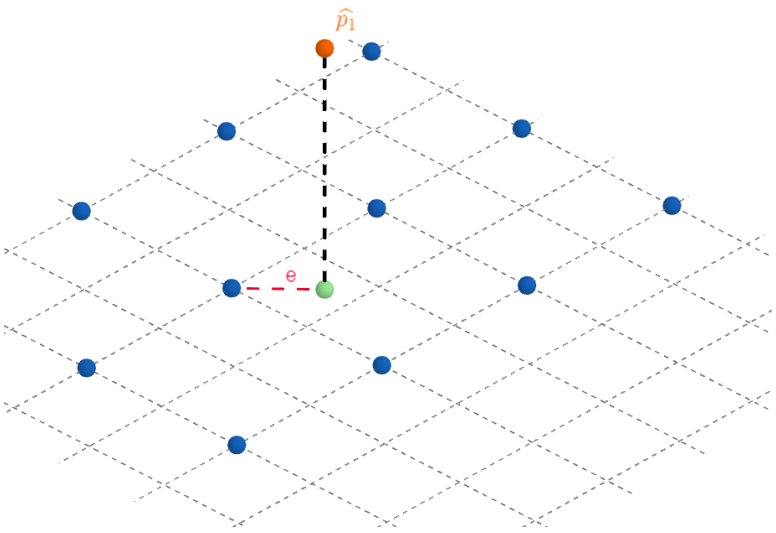


Figure 7: Orthogonal projection of a point in the evaluated pointcloud ($\widehat{p_1}$) on the planar of the groundtruth point cloud. The error e, determines how far the considerd point for the ground truth lies from the actual point of the ground truth.

This situation is displayed in fiture 7, 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 and algorithms would require more than a day, only a subset of P' of 1000 points per sequence and algorithms is taken into consideration. The indices for the subset I_{sub} are sampled from a even distrubution of $i_{i=0}^{M_{\text{eval}}-1}$. Then, as mentioned, distances to the closest point in the ground truth point cloud is calculated for the sampled subset 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

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

.

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

2. Density

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

Computation Time

Since the computational performance of an algorithm is crutial to perform in real time, the absolute time that is needed to process each sequence is measured for each algorithm. The time required for initialization is substracted, since it is not decisive for the assessment, if the algorithm can be run in real time. For

each sequence the resulting speed is additionally evaluated in computet frames per second.

2.2.3 Setup and Environment

Evaluation

The entire evaluation is run on a virtual machine. The host system is a lenovo yoga with eight GB of RAM and the basic model (8250U CPU @1.6 GHz 1.80GHz) of an eight core i5. The operating system of the host machine is Windows 10 Home. The virtual machine is given 5 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.

Flight Path Planning

2.3 Results

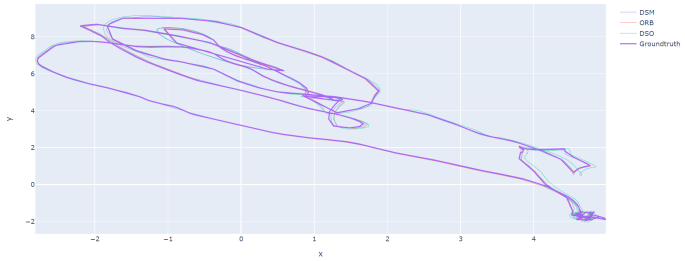
The three SLAM algorithms were evaluated regarding the computed trajectories, the resulting pointclouds and the computational complexity.

2.3.1 Trajectory Evaluation

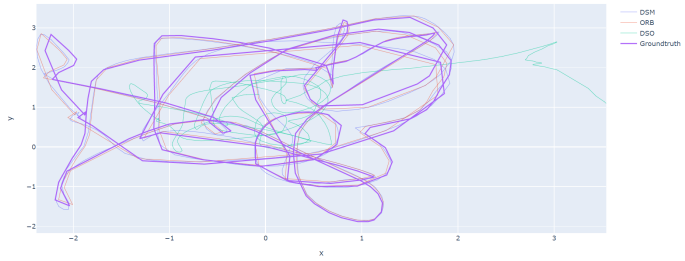
In order to evaluate the quality of the trajectory computed by the algorithms, the trajectory firstly had to be transformed into the world reference of the ground truth data. This is because the algorithms usually initialize the origin with the first (key)frame and the groundtruth data doesn't. Furthermore, monocular visual slam algorithms are generally not capable to extract the true scale. The alignment was performed using the method of Umeyama, described in the **??** section. After alignment, as a first indicator for the accuracy of the computed trajectories, the trajectories were visually observed by plotting the true position and the evaluated position into a coordiante system. The x, y and z axis were observed seperatly. This method for comparing the trajectories is described in detail in section **??**. Obviously, trajectories, that align perfectly with the groundtruth suggest that the position was correctly evaluated by the algorithm.

In figure 8 the computed trajectories are plotted against the groundtruth for sequences MH01, V102 and V203 considering only the x and y coordinate. These sequences were selected, because they represent the results of the visual analalysis 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 gentel movements, moving at an average speed of $0.41 ms^{-1}$.

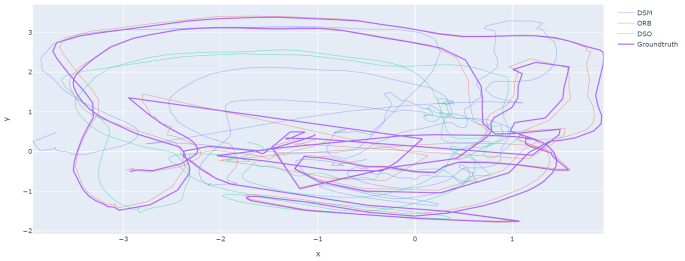
The second image shows the sequence V102. Here, ORB and DSM show good results, since in most parts, the trajoctories are aligned perfectly. DSO on the



(a) MH01



(b) V102



(c) V203

Figure 8: Ground truth flight path and evaluated flight path of each algorithm after alignment with the method of Umeyama in the x and y axis in meters. Left the sequence MH01, middle the sequence V102 and right the sequence V203 is displyed.

other hand shows a significant difference in the flight path. When observing the trajectory closely, the assumtion is raised, that the algorithm lost tracking and therefore computed significant wrong postion data. This might have caused the calculated positions in the right of the plot, that have a large distance to the groundtruth. The rest of the sequence might have been correctly estimated, however since the alignment is done minimzing **??**, one significant mistake in the position estimation, might result in horrific 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 groundtruth, only with a significantly smaller scale.

The last plot shows the results of the last sequence. Here, all algorithms had problems estimating a position that comes close to the groundtruth position. ORB SLAM still showed acceptable performance.

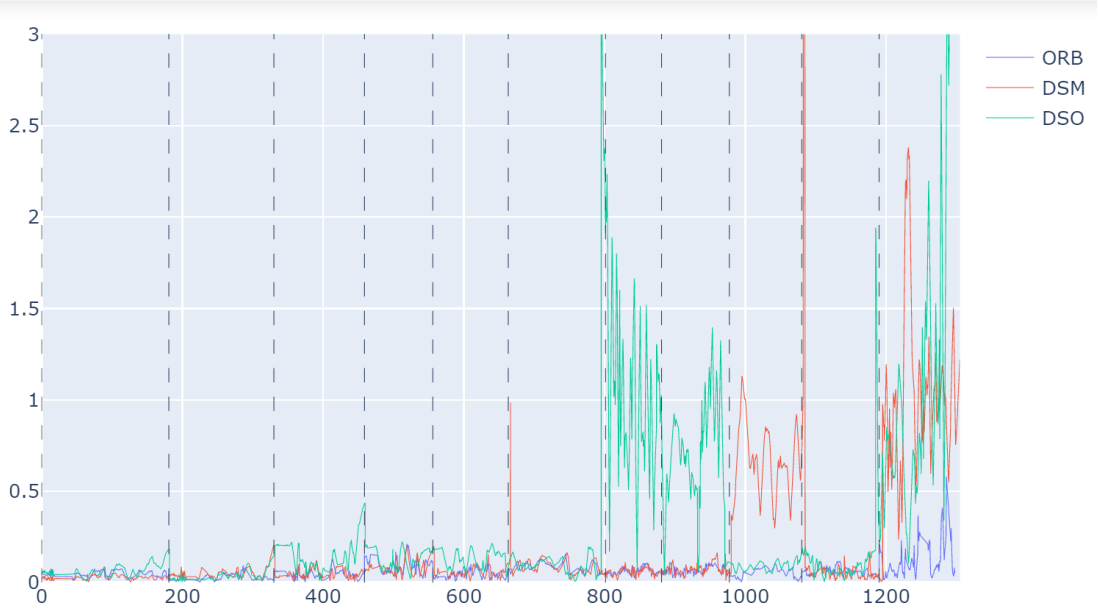


Figure 9: The positional error over time in meters. The vertical lines indicate the beginning of a new sequence

Furthermore, the eucleadian distances between position of the keyframe and the true postion of the latter are computed. For a keyframe, the entry of the groundtruth 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 9 shows these distances over time 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 results. Also, it can be noticed, that DSO SLAM drifts further apart from the ground truth position with the continuety of the sequence in the sequences MH01, MH03 and V202. This might be the result of lacking the functionality to close loops and to optimize over the global map, as explained in section **??**.

In figure 10 a boxplot of all computed distances over all sequences is displayed. This plot summarieses the results of the trajectory analysis. The ORB algorithm, yielding a median positional error of 5.4 cm, performes slightly better than the DSO algorithm. DSO SLAM shows inconsistent results with a median positional error of more than 10cm and a large amount of errors greater than half a meter.

2.3.2 Pointcloud Evaluation

For the evaluation of the computed point clouds, these point clouds were first visually observed, as described in section **??**. Figure 11 shows the evaluated point clouds aligned with the ground truth point cloud for sequence V101. This is a sequence, where 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 first glance is, that as mentioned, ORB generates only few points, since only found keypoints are mapped in feature based methods. To give these points better visibilty, the point size was doubled in the ORB

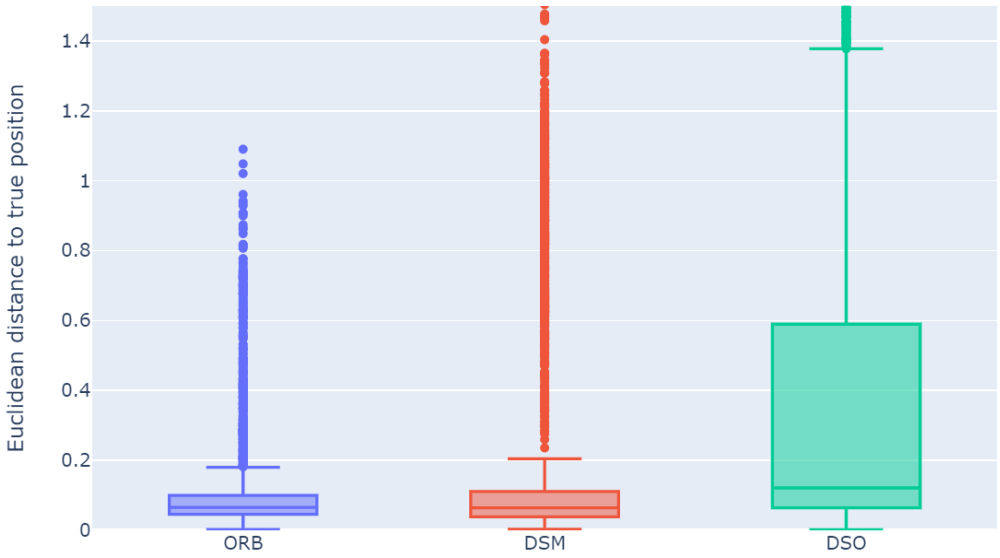


Figure 10: Boxplot of all euclideans distances between the ground truth position of the keyframe and the evaluated position after alignment with the method of Umeyama. Outliers greater than 1.5 are not displayed for clearity reasons.

image. DSM and DSO slam generate point clouds with significant higher density, where all structures of the room are clearly visible at first glance.

However, the advantace of ORB-SLAM over the other two direct methods is the recognizing of clear features in terms of strucural differences in the scenes. Though, DSO and DSM also regard the differences in pixel intensities ORB, as described in section **??**, detects the features on different scale levels and ensures, that the regarded features are in fact significant. 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 leiter?? in image three of figure 11, where all sprossen contain at least one point.

After closely observing the point clouds, it became clear, that the point clouds of DSO, often times generate point clouds, where multiple layers of points were falsly generated, where all points had the same clear distance to the ground truth point cloud. This may be a result of working of DSO slam, where

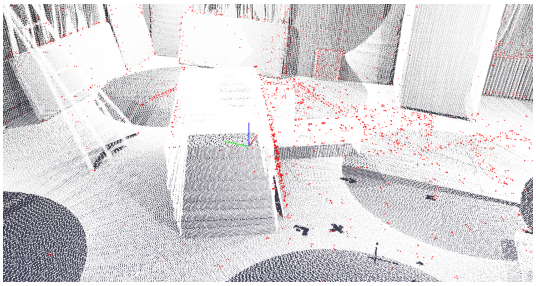
Table 2: Absolute Number of points and distance to closest point in the ground truth pointcloud in meter for each algorithm and sequence.

Sequence Name	ORB	DSM	DSO
MH_01_easy	8958 (/)	675720 (/)	361633 (/)
MH_02_easy	8692 (/)	700920 (/)	343804 (/)
MH_03_medium	7445 (/)	614264 (/)	371752 (/)
MH_04_difficult	7943 (/)	495752 (/)	208445 (/)
MH_05_difficult	8373 (/)	517712 (/)	232415 (/)
V1_01_easy	7075 (0.049)	6108440 (0.066)	374257 (0.066)
V1_02_medium	6517 (0.042)	648440 (0.187)	366513 (1.458)
V1_03_difficult	7983 (0.052)	775080 (0.092)	448212 (0.459)
V2_01_easy	5688 (0.049)	584552 (0.58)	247905 (0.086)
V2_02_medium	8129 (0.074)	733992 (0.078)	490608 (0.104)
V2_03_difficult	7359 (0.17)	921312 (0.645)	465396 (0.677)

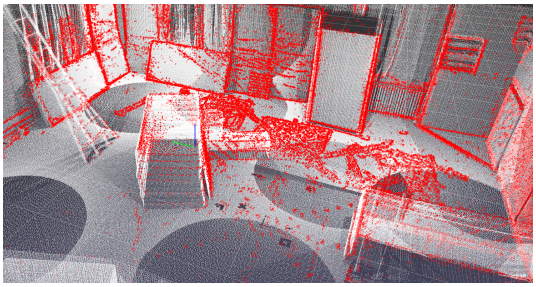
keyframes, that are marginalized, are removed permanently. When revisiting areas, the points are again generated. This means that all errors made in the sequence accumulate and when an area is revisited, significant errors in the point cloud can be made. This can be seen when looking at the third image closely. When looking at the matraccess 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 more than 200000 in most sequences.

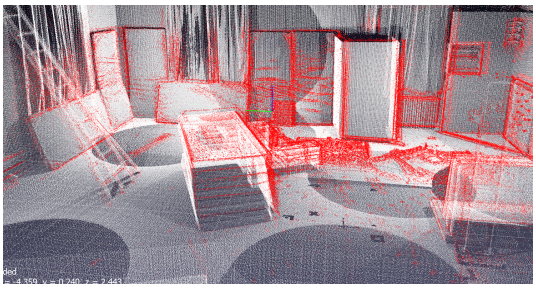
However, regarding the accuracy of the computed points by the algorithms, again ORB-SLAM shows the best performance. In table 2, for all sequences, where a groundtruth pointcloud exists, the distance to the closest point in the groundtruth point cloud is computed by randomly sampling 500 points



(a) ORB



(b) DSM



(c) DSO

Figure 11: The groundtruth of the Pointcloud from Sequence V101 (white points) and the evaluated points by each algorithm (red points). The points in Figure (a) are wtice as large for better visability (ORB-SLAM generates only few points).

per algorithm and sequence. In all except the last, ORB-SLAM yields a mean distance of less than 10cm while DSM and DSO SLAM also yield results greater than 50 cm for certain sequences.

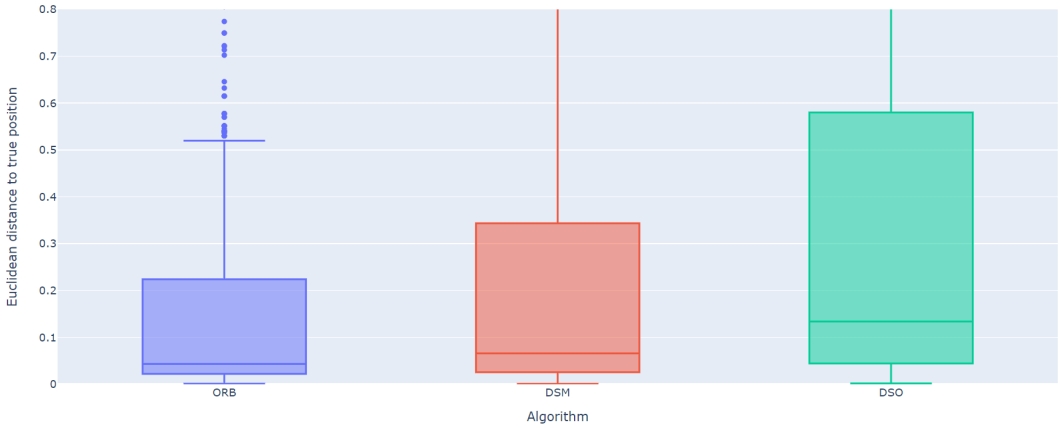


Figure 12: Boxplot of the eucleadian distances between an evaluated point and the closest point of the ground truth point cloud. For computational feasibility, for each sequence and algorithm, 500 points for evaluation are sampled randomly

2.3.3 Calculation Time

For each sequence and algorithms, the computational time was taken, that the algorithm took, to complete the computation of the sequence. The time needed for initialization was not considered for the absolut time.

In order to evaluate, if an algorithm can run in realtime, the absolut times have to be broken down into the computed frames per second. The euroc dataset consists of sequences recorded at a frame frequence of 20 frames per second. This means the total frames per sequence are amount to duration_of_sequence x 20. For the computed frames per second, this value is then devided by the time the algorithm needed to process the sequence.

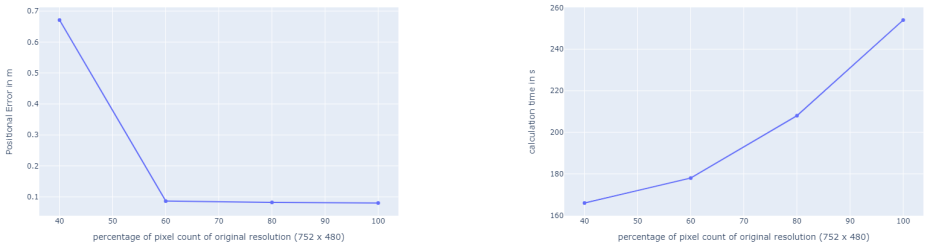
It is possible to decrease the frame frequence, but this will also increase the risk

Table 3: Computation Time (excluded time needed for initialization) of each Sequence and Algorithm. In parenteses 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	257 (14,2)	1098 (3,3)	749 (4,9)
MH_02_easy	209 (14,4)	984 (3,1)	690 (4,4)
MH_03_medium	198 (13,3)	1369 (1,9)	707 (3,7)
MH_04_difficult	165 (12)	896 (2,2)	504 (3,9)
MH_05_difficult	193 (11,5)	825 (2,7)	633 (3,5)
V1_01_easy	253 (11,4)	1383 (2,1)	905 (3,2)
V1_02_medium	150 (11,1)	1550 (1,1)	820 (2,0)
V1_03_difficult	186 (11,3)	2262 (0,9)	1134 (1,9)
V2_01_easy	187 (12)	1045 (2,1)	612 (3,7)
V2_02_medium	162 (14,2)	1675 (1,4)	1522 (1,5)
V2_03_difficult	143 (16,1)	1600 (1,4)	793 (2,9)

of loosing the tracking, since the steps between the frames are greater and the ??foreward motion model?? cannot pridict the new position as well für einen gewissen algo.

The results are displayed in table 3. ORB-SLAM processes the frames at a frame per second rate ranging from 11,1 (V102) to 16,1 (V203), DSM SLAM processes the frames at a range from 0,9 (V103) to 3,3 (MH01) frames per second and DSO slam at a rate in range of 1,5 (V202) to 4,9 (MH01). Thus, none of the evaluated slam algorithms processes any of the evaluated sequences in realtime.



(a) trajectory error

(b) computation time

Figure 13: Influence of downsizing of the images on the trajectory error (a) and the computation time (b) for the sequence V101.

2.4 Discussion

2.4.1 Conclusion of SLAM-Algorithm Evaluation

3. Full automated exploration system

3.1 Existing systems

3.2 suggested ROS framework

In this section a framework to test and build an entire automated system is suggested. This framework includes a simulated envorinment, that realisicly makes it possible, to navigate a drone within an simulated environment. The environment is based on the Roboter Operating System (ROS) and is completely simulated.

The basic idea of the framework can be seen in figure 14. The main parts of the automated exploration system are the SLAM Algorithm and the flight path planning algorithm, that work simultaniously. The general concept is that each of the algorithms takes the output of the other as input.

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

ROS stands for Roboter Operating System and as the name suggests, is a framework for a software infrastracture within a robot. With the right drivers installed, it can access and use the robots hardware and serves as a messenger system between robot components. Ros packages make it easy to reuse im-

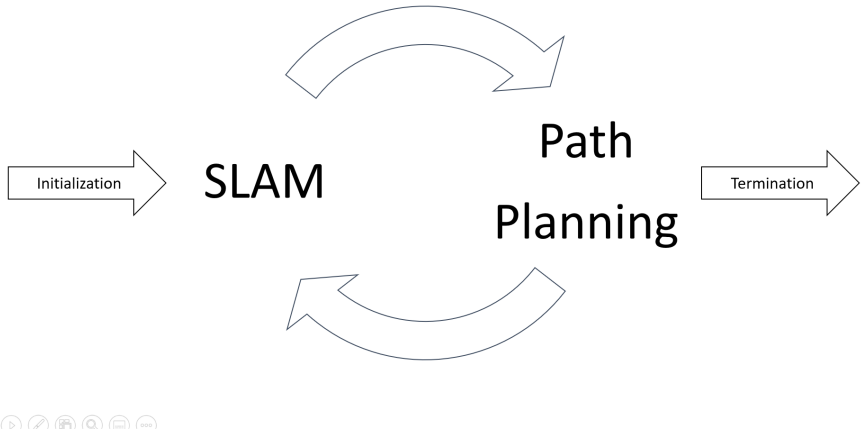


Figure 14: Automated SLAM system

portant functionalities. Gazebo on the other hand is a 3D dynamic simulator for robotics. It can accurately and efficiently simulate robots regarding their physics.

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, also output data. Within ROS, data is shared and recieved over so called rostopics (in short topics). Each node can therefore publish the computed data over a certain topic or it can recieve data from a certain topic by subscribing it. The frequency, the data is streamed to the system is also defined for each topic. The frequency the data is recieved can also be meanually defined for each subsription.

In figure 15 the suggested setup is displayed. The system consists of six nodes. The ORB-SLAM-Algorithm, together with the Path planning algorithm, form the nodes with the most complex computations. To complete the system and to provide a setup, where path planning algorithms can easily be tested, the other four nodes play a crucial part in the setup. 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 communi-

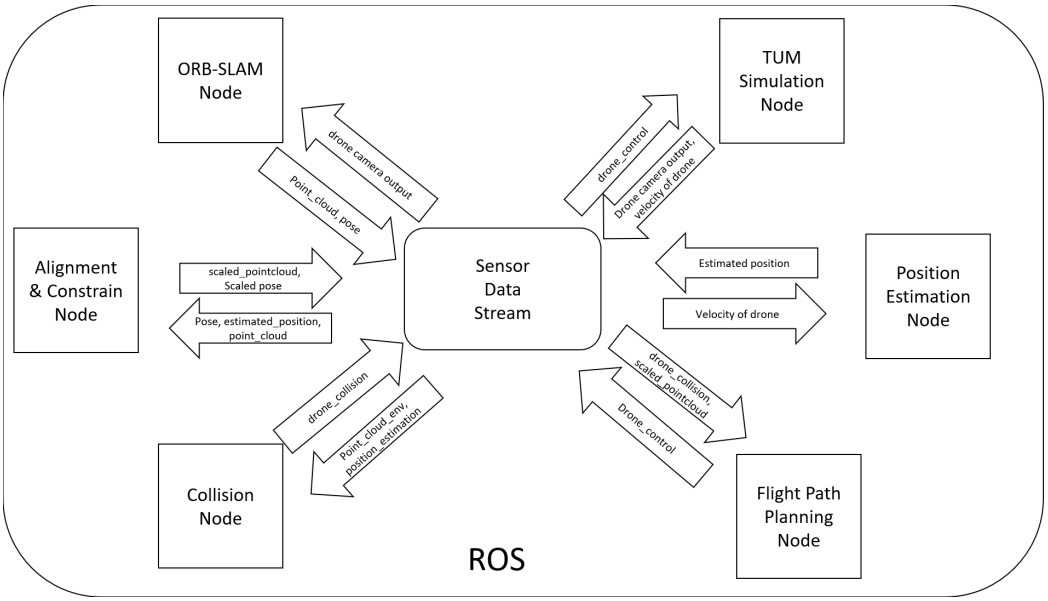


Figure 15: Overview over the suggested ROS framework

cate over standardized messenges, they are independent of the programming language. This means that while the ORB-SLAM node is implemented in C++, the position estimation node and the scale recognition node are implemented in python with rospy.

3.2.1 TUM Simulation Node

Thus, with the tum_simulation package you can navigate an AR.drone 1.0 and 2.0 in different worlds created with a gazebo node. This drone is eqipped with a bottom camera and a frot camera. The cameras each log their output to a ros topic. Additionally, message time stamps, the height sensor output, battery percentage, rotation velocity and accelaration are also logged to rostopics. While the drone can also be navigated using a playstation 3 controller, as shown in figure 16, showing a section of the tum_simulation package content structure, for an automated system, the drone should rather be addressed using the command line interface. For example the command shown in sourcecode listing 1 will make the drone fly foreward.

Listing 1: drone navigation command

```
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}}'
```

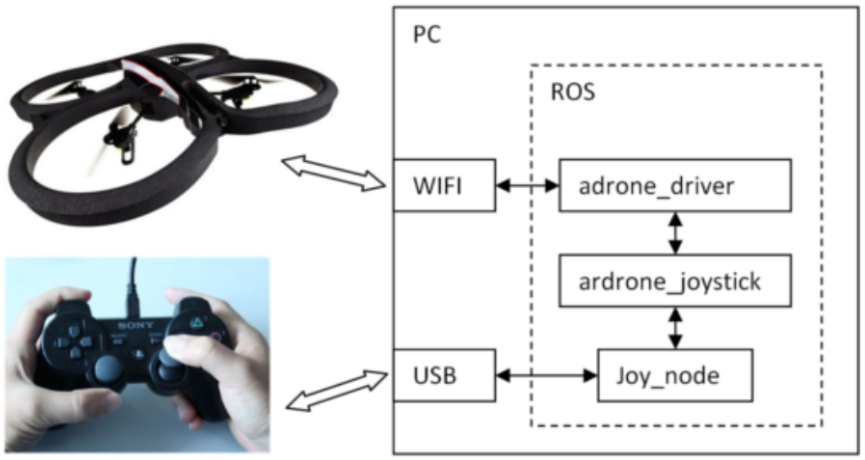


Figure 16: tum simulator setup. Source: *http : //wiki.ros.org/tum*$_s$*imulator*

Input

1. /cmd_vel

   This topic is explained in the upper section **??**.

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 listed.

1. /ardrone/front/camera_info

   Over the /ardrone/front/camera_info topic, the node publishes messenges of the class CameraInfo. These messenges include information about

image dimension, timestamp and imformation about the camera specific values, described in section **??**. In the case of the ardrone 2.0, the front camera generates images with 640x360 pixels and the instrinct camera matrix is given by:

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

2. /ardrone/front/image_raw

   The node publishes the output images of the front camera. The meta-data of the camera are provided in the topic /ardrone/front/camera_info described above.

3. /ardrone/navdata

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

4. /ardrone/takeoff

   As the name of the topic suggests, the drone takes off, when empty messenges are published to the thred. How this looks exactly can be found in listing **??**.

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 **??**, 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.

6. /ardrone/collision

   This topic is explained in the upper section **??**.

3.2.2 ORB-SLAM Node

The ORB-SLAM algorithm runs in a seperate node. For the vocabulary file, needed for the bag of words approach, explained in section **??**, the standard vocabulary file proveded by the authers are considered. For the virtual environment, it might be useful to provide a vocabulary file generated for this particular purpuse, since in the simulation generated with the tum_simulation, edjed might be of different shape, e.g paricularly sharp.

The node publishes the pose $x_i \in \mathrm{SE}(3)$ and map_points. Because the original ORB ROS implementation didn't have an option for publishing data and projects, that implemented this functionality, these features had to be implemented.

Input

1. /ardrone/front/image_raw

   This topic is explained in the upper section **??**.

Output

1. /orb/pose

   This topic publishes messenges 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 oriantation is given with quaternions coordinates. The topic is published at a frequncy of 30 Hz.

2. /orb/map_points

   This topic publishes messanges 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 **??** can be refered to.

3.2.3 Alignment node

One benefit of implementing the automated system in a virtual environment is that the true position of the drone and the environment is known at all times. As mentioned in sectionasdfasdf 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 pointcloud that is computetd by the ORB-SLAM algorithm to the reference of the gazebo world, the estimated position by ORB-SLAM and the true position are again aligned with the method of umeyama.

This node computes this 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 this 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 privides 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 only executes the task of stamping the true positions, it is not listed in this chapter.

2. /orb/pose

   To align the groundtruth position and the estimated position of ORB, the /orb/pose topic published by the ORB node described in section **??** 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.

3.2.4 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 at two lists. This is done at a frequency of 10 Hz. Before aligning the points in the list, it is waited until 50 points are logged to each list. If only one list has reached the lengt 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 uneccessary resources in the following. transformation step. 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 timedifference is matched. This assures that the points estimated by orb and from the voloity are measured at the same time.

Finally, the points are aligned by using the method of umeyame, as described in section **??** 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 respoctive 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         # since no scale can be computed
4         if (len(self.est_pos_orb) < 50) or (len(self.
             true_pos) < 50):
5               print("not enough data available, waiting
                   ...")
```

```
6                 return
7
8           # return the scale if it already has been
               calculated
9           else:
10                # get minimum and maximum time for each
                     queue to figure out,
11                # how many points can be considered for
                     alignment. This is only done once!
12                min_orb = np.min([pose_oi.header.stamp.
                     to_sec() for pose_oi in self.
                     est_pos_orb])
13                max_orb = np.max([pose_oi.header.stamp.
                     to_sec() for pose_oi in self.
                     est_pos_orb])
14
15                min_true = np.min([point_oi.header.stamp.
                     to_sec() for point_oi in self.true_pos
                     ])
16                max_true = np.max([point_oi.header.stamp.
                     to_sec() for point_oi in self.true_pos
                     ])
17
18                thresh_min = np.max([min_orb, min_true])
19                thresh_max = np.min([max_true, max_orb])
20
21                # cut off the queues
22                orb_oi = [pose_oi for pose_oi in self.
                     est_pos_orb if pose_oi.header.stamp.
                     to_sec() > thresh_min]
```

```
23                  true_oi = [pose_oi for pose_oi in self.
                        true_pos if pose_oi.header.stamp.
                        to_sec() > thresh_min]
24
25                  # for the shorter remaining queue, get
                        the matching point
26                  if len(orb_oi) <= len(true_oi):
27                          orb_oi_final = orb_oi
28                          true_oi_final = []
29                          for pose_oi in orb_oi:
30                                  diffs_oi = [np.abs(
                                        pose_oi.header.stamp.
                                        to_sec() - point_oi.
                                        header.stamp.to_sec())
                                         for point_oi in
                                        true_oi]
31                                  true_oi_final.append(
                                        true_oi[diffs_oi.index
                                        (min(diffs_oi))])
32
33                  else:
34                          true_oi_final = true_oi
35                          orb_oi_final = []
36                          for pose_oi in true_oi:
37                                  diffs_oi = [np.abs(
                                        pose_oi.header.stamp.
                                        to_sec() - point_oi.
                                        header.stamp.to_sec())
                                         for point_oi in
                                        orb_oi]
```

```
38                                  true_oi_final.append(
                                        true_oi[diffs_oi.index
                                        (min(diffs_oi))])
39
40                  # now do the alignment and compute the
                        scale
41                  x_orb = [pose_oi.pose.position.x for
                        pose_oi in orb_oi_final]
42                  y_orb = [pose_oi.pose.position.y for
                        pose_oi in orb_oi_final]
43                  z_orb = [pose_oi.pose.position.z for
                        pose_oi in orb_oi_final]
44
45                  x_true = [pose_oi.pose.position.x for
                        pose_oi in true_oi_final]
46                  y_true = [pose_oi.pose.position.y for
                        pose_oi in true_oi_final]
47                  z_true = [pose_oi.pose.position.z for
                        pose_oi in true_oi_final]
48
49                  orb_points = np.column_stack((x_orb,
                        y_orb, z_orb))
50                  true_points = np.column_stack((x_true,
                        y_true, z_true))
51                  s, R, t = align_umeyama(true_points,
                        orb_points)
52                  R = R.reshape([9,])
53
54                  # finally publish the computed scale,
                        matrix and vector
```

```
55                  if s>0:
56                          self.scale_publisher.publish(
                               Float64(s))
57                  if sum(np.isnan(R)) == 0:
58                          self.rot_publisher.publish(R)
59                  if sum(np.isnan(t)) == 0:
60                          self.trans_publisher.publish(t)
```

3.2.5 Scale Recognition Node

This node estimates the true scale of the pointcloud. This is beneficial to later use the pointcloud for real life purposes. This is done by aligning the estimated position based on the velocity of the drone in the initialization process and the estimated position of the drone by the ORB-SLAM Algorithm. If the scale has been calculated, the node logs the constant scale to the topic. Note, that the scale is still just an estimation, since the scale may drift while the ORB-SLAM Algorithm continues running. This is called scale drift.

Input

1. /drone_position_init

   This topic is explained in the upper section **??**.

2. /orb/pose

   This topic is explained in the upper section **??**.

Output

1. /scale_estimation

The node publishes a topic called /scale_estimation, which is a float64 representing the calculated scale. If the scale is not yet computed, nothing is published.

computation

If all nodes are up and running, the data logged to /orb/pose at 15Hz and to /scale_estimation at 10 Hz is stored at two lists. This is done at a frequency of 10 Hz. Before aligning the points in the list, it is waited until the 25 points are logged to each list. If only one list has reached the lengt of 25, new elements are stacked on top, while the oldest are removed.

Then, the lists are culled in a way, that the timestamps of the oldest and newest messages are aligned. Even though the scale is only computed once, this saves unneccessary resource. 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 timedifference is matched. This assures that the points estimated by orb and from the voloity are measured at the same time.

Finally, the points are aligned by using the method of umeyame, as described in section **??** and the scale is published to the topic.

These computational steps are processed in the main callback loop of the respoctive file for the node. The function of the main loop is shown in listing.

3.2.6 Position Estimation Node

The position estimation node appriximates the position of the drone based on the valocity 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 foreward movement. No rotations should be performed with the drone since the drones navigation data, such as the veloxity, uses the bodyframe 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 in the simulation does, also ORB SLAM would benifit from accuracy, since it the integration in the algorithm is implemented. However, if drone does not have these sensors attached, this node can still be of benefit, since it is possible to manually publish messanges to the topic. The initialization process could then be applied by manually flying the drone one meter up, and one meter foreward and logging these points to the system. Then the automazation process could be started.

Folling an overview over the input, output and computational functionality of the node.

Input

The node subscribes to the following topics:

1. /ardrone/navdata

   This topic is beeing published with the tum simulation node described in item **??**. 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 messenges.

2. /drone_position_init

The drone also subscribes to the /drone_position_init topic, published by itself. This topic includes messenges 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 timedifference in seconds to the last point is extracted, which is easy, since all onject from the class PointStamped can be timestamped. The difference is given by $\Delta t_i = t_i - t_{i-1}$. The updated position is the yielded by

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

.

Dividing by 1000 yields the unit meter.

This recursive methodoly is shown in figure 17

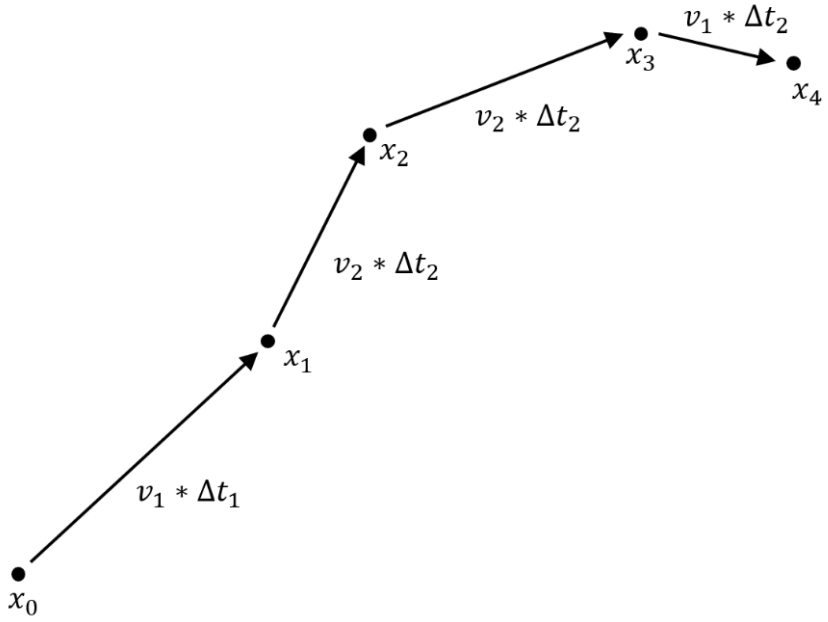


Figure 17: Calculation method for estimation the position in the initialization porcess in order to find the true scale.

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)
```

3.2.7 Transformation and Constrain Node

This node processes the scale, rotation matrix and translation vector computed in the alignment node and transformes the orb pose and the pointcloud com-

puted 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.

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 groundtruth position as it also has been done in the evaluation of the vSLAM algorithms. While it is was not yet managed to extract the groundtruth pointcloud of the gazebo world, as described in issue **??**, it is technically possible, to also compare the groundtruth pointcloud to the generated one. On the other hand, it also enables users to track the path planned by the path planning algorithm relative to its surrounding in the future.

The restriction of the searching space is useful, because the aim the path planning algorithm is to explore unseen areas in in the searching space. Obviosly, since the path planning algorithm relys exclusivly on the pointcloud, unseen areas are then defined as areas, where no points are available. The gazebo world, displayed in figure **??**, 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 issue **??**, no other world is yet available. Thus, adding points to limit the searching space is crucial. This is the reason why most navigation algorithm are constructed for indoor navigation **??**.

Input

1. /orb/map_points

   The ORB point cloud topic, described in section **??** subscribed to by the node.

2. /orb/pose

   Since also the pose is 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 messenges of the class PointCloud. Therefore, the pointcloud contains the transformed pointcloud 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 transfomed 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 recieved from the above described topics. The resulting points are therefore computed by:

$$p_i' = sRp_i + t$$

.

Then, the restriction points are added to the pointcloud. 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 axact 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:

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

.

Then, 10000 points for each side of the hull of Ω are added to the generated point cloud by ORB, after it was 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 pointcloud.

```
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 resouces, since the pointclouds can contain ten thousands of datapoints. In order not to overload the system, the transformation node runs on a frequency of only 5Hz, which results in smooth computation.

3.2.8 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 obsticals. 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, three algorithms, currently used in for path planning approaches in active SLAM-application are described in the computation section.

Input

1. /global_map_points

   This topic is explained in the upper section **??**.

2. /global_map_points

   This topic is explained in the upper section **??**.

3. /colliding

Output

1. /cmd_vel This topic is explained in the upper section **??**.

Computation approaches

While the research in the field of automated exploration algorithms is still in its early ears [2] and will probaply be mainly performed in simulated environment

in the near future [12], some frameworks on tackeling the task already exist [2], [12] [21] [10]. 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 [12].

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 with this framework. The POMDP framework relies on seven components [**?**], 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 **??**, 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 cammands, that can be published to the /cmd topic. This includes rotational and translational manovers. A possible command in order to make the drone fly forewar?? can be seen in listing **??**.

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 greately, 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:

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

The belief b defines the probabilty for every state, that the drone actually is in this state. Therefore all probabilites should add up to one. Not all algorithms require b [17].

The set of observations Z equals the current pose and and current map points. Finally, O returns the probabilty to make an observation z while being in the state s. The POMDP is looping the follwing 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 exprected reward in the future. There are several methods to find a policy, given the above components. In this paper, three of those methods are discussed.

1. Monte Carlo

2. Reinforcement learning

   Unlike most other machine learning algorithm, reinforcement learning does not require training data for learning behavior. The learning mechanism is soly based on the reward, that is given for an action. Q-learning approaches don't need the definition of O and T. This is why Q-learning is defined as a model-free machine learning process [12].

   Q-learning is a method of reinforcement learning. Approaches on applying it for robot navigation exist [21][12][10]. Q-learning approaches don't need the definition of O and T. Instead, Q-learning defines 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 [10].

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

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

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

   .

   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 cummulated sum is only weihted with $0.8^3 = 0.512$. Thus, the more relyable the algorithm and the chosen reward funktion 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. The algorithm

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: [12]

In the recent years, successes were made in the applyance of deep reinforcement learning algorithms for robotic navigation and exploration [12] [21]. These approaches don't 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 recursivly, the computation requires a lot of computational resouces. 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 knowlege.

The results of [12] also shows, that the trajectory accuracy can be increased be autonomously navigating the drone.

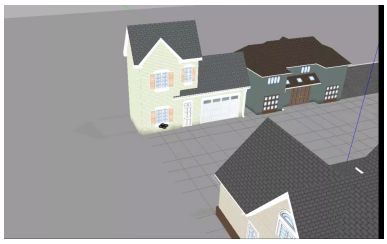
> D3QN results are the most remarkable in the first environment, as it outperforms the reward that a human would obtain by manually controlling the robot (approx. 350). In the second environment both DDQN and D3QN show a good behavior. Despite DDQN have higher SR (and mean steps, thus), the higher mean reward obtained by D3QN proves the generation of more optimal trajectories: smoother movements and less spins. [12]

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 [21].

3.3 Current setup

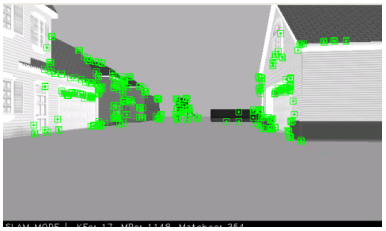
In figure [**?**] the simulated environment with gazebo can be found in figure a. Here, the drone (in black) is flying in front of the building. The output of the front camera is shown in figure b. In figure c, the ORB-SLAM-Algorithm was applied to the output of the front camera. Green dots represent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

Currently the framework is set up in an environment provided by theconstruct-sim.com. This platform is enabling ROS-developers to program in preconfigured ROS-environments. The environment comes with the possibility to open terminal consols, a file management system, 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.



(a) gazebo simulation

(b) front camera simulation



(c) ORB applied on simulation

Figure 18: 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).

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 are already installed. 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 rosrun ORB_SLAM2 Mono ${PATH_TO_VOCABULARY} ${
```

```
    PATH_TO_SETTINGS_FILE}
7
8  # start the scale estimation node
9  rosrun auto_explorer scale_updater.py
10
11 # start the position estimation node
12 rosrun auto_explorer position_updater.py
13
14 # takeoff with drone
15 rostopic pub -1 /ardrone/takeoff std_msgs/Empty
16
17 # Then start flight path planner algorithm
```

In listing 5 the commands for launching the gazebo simulation, ORB-SLAM and the drone are displayed. After launching thouse applications, only the path planning algorithm based on the resulting point cloud is missing. However, multiple solutions for such algorithms exist [7], appying it on the system is not part of this paper and will be done in further research.

3.3.1 Known Issues

1. Only one world available

   For the tum_simulator it might be useful to continue the atomation process in another simulated world. This is because the current world named ardrone_testworld doesn't contain any contours or relief on the ground, and sky, as shown in figure **??**. Since the ORB-SLAM algorithm is looking for features such as edjes and changes in pixel intensites, it will not find any on the ground, which will result in no points available in this are 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 follwing error:

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

So far, no solution has been detected, but one possible workaround would either 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 bla bla node, described in section **??**, which restriction points to the pointcloud.

Listing 6: launching different world

```
1
2 # launch different world named land_station1
3 roslaunch cvg_sim_gazebo land_station1.launch
```

2. No ground truth pointcloud

   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 pointclouds.

   This refuses the possibilty 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 in section **??**.

3. No bumper

4. Summary

Bibliography

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

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

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

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

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

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

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

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

[9] C. Leung, S. Huang, and G. Dissanayake. Active slam using model predictive control and attractor based exploration. pages 5026 – 5031, 11 2006.

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

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

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

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

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

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

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

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

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

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

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

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

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

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