Universidad del País Vasco

Euskal Herriko Unibertsitatea

Fraunhofer IPA

Visual SLAM using straight lines

Author: Oier Mees
Supervisor: Richard Bormann
Supervisor: Elena Lazkano

Computer Science Engineer

Acknowledgements

Nire familiari, nire euskarria izategatik eta bizitzan zoriontsu izateko baloreak erakusteagatik.

Elenari, aparteko tutorea izateagatik eta nire ikerkuntza gai bihurtu diren robotika eta adimen artifizialaren mundua erakusteagatik. Zu gabe, auskalo zertan ibiliko nintzatekeen.

To Richard Bormann, for giving me the opportunity of developing this thesis at Fraunhofer IPA and for having patience with me.

Contents

List of Figures

List of Tables

Abstract

The present thesis is focuses on the problem of Simultaneous Localisation and Mapping (SLAM) using only visual data (VSLAM). This means to concurrently estimate the position of a moving camera and to create a consistent map of the environment.

Since implementing a whole VSLAM system is out of the scope of a degree thesis, the main aim is to improve an existing visual SLAM system by complementing the commonly used point features with straight line primitives. This enables more accurate localization in environments with few feature points, like corridors.

As a foundation for the project, *ScaViSLAM*[1] by Strasdat *et al.* [46] is used, which is a state-of-the-art real-time visual SLAM framework. Since it currently only supports Stereo and RGB-D systems, implementing a Monocular approach will be researched as well as an integration of it as a *ROS*[2] package in order to deploy it on a mobile robot.

For the experimental results, the Care-O-bot service robot developed by Fraunhofer IPA will be used.

Key words: visual SLAM

[1]Source code available at `https://github.com/strasdat/ScaViSLAM`
[2]Robot Operating System, more information available at `www.ros.org`

Part I

Introduction

Table of Contents

Chapter 1

Motivation

Simultaneous Localization and Mapping [49], better known as SLAM, is a technique used by mobile robots and autonomous vehicles. It refers to the problem of concurrently building a map and localizing itself in an unknown environment. SLAM is an essential problem to be solved for autonomous mobile robots to explore unknown environments. Moreover, it is a key problem to be solved in order to spark a revolution in the field of robotics, whose impact on humanity will be comparable to the ones of the industrial revolution. We will see more and more autonomous robots perform everyday tasks to improve the quality of human life. While nowadays robots are usually designed for performing special tasks, some belief that general purpose service robots will be developed, causing an unpredictable impact. One of the obstacles on the road to that revolution is to solve SLAM efficiently with cheap hardware in real time and unlimited sized environments.

2D sensors like laser range finders and sonars [21] have been traditionally used for navigation algorithms. Due to the fact that these can only detect obstacles in its same plane, alternative sensors have been explored which provide richer 3D information, like 3D Flash LIDAR [32]. In recent times, there has been a growing interest in performing SLAM with cameras instead of lasers, because cameras have become much cheaper than lasers. This variant is called visual SLAM. Moreover, low-cost depth imaging cameras like the Microsoft Kinect [22] have made dense 3D point clouds available to everyone. The Kinect camera provides not only point clouds, but also color images. Point clouds are very useful to create a map of the environment by aligning consecutive point clouds, for instance via the Iterative Closest Point (ICP) and RANdom SAmple Consensus (RANSAC) algorithms for instance. Color images on the other hand, can be exploited to determine if the robot has already visited a place, which is known as loop closure problem.

On the other hand, monocular SLAM [19] has also been getting a lot of attention. The reason for this is that, monocular cameras would help driving down the manufacturing cost of robots even more than RGB-D systems. The downside of monocular SLAM is that it is more challenging. Similar to stereo-camera systems, a monocular camera needs to view the same scene from two different viewpoints in order to triangulate and estimate depth information.

Visual SLAM typically makes use of visual cues, such as feature point descriptors [16] in images, and tries to match them in consecutive frames. As lines provide richer information of the environment and are more robust to viewpoint changes and occlusions, they can be a good complement to feature points.

Because of the aforementioned reasons, the use of lines in monocular cameras and RGB-D cameras like the Kinect will be explored, in order to improve existing SLAM techniques.

Chapter 2

Preliminaries

Contents

2.1 SLAM

SLAM is commonly referred to as a chicken and egg problem. On the one hand the robot's pose has to be known to build a map. On the other hand, an accurate map is needed in order to perform localization. SLAM is challenging because an inaccurate estimation of the egomotion will have a negative impact on the map quality, which again will negatively influence the subsequent egomotion estimate and so on.

From a theoretical point of view, SLAM has been solved for small sized environments. SLAM systems have been used for promising prototypes in indoors and outdoors robots, even for aerial [13] and underwater [41] autonomous systems. Nevertheless, there are still some practical issues which need to be addressed in order to be used in commercial and industrial applications. Some of these issues are, the size of the maps when performing large scale SLAM and their representation, performing in dynamic environments with moving objects and multi-robot SLAM.

Because of the inherent noise of the sensor measurements, SLAM is best described with probabilistic distributions [49]. The concept of using probability distributions in this domain can be better understood with an example, such as the popular Markov Localization (ML) algorithm used for robot navigation. In figure 2.1 the classic Markov Localization example is illustrated, in a one dimensional corridor. At the beginning, the robots starts with a uniform probability distribution, as it considers itself equally likely to be at any point in space along the corridor. After detecting with its exteroceptive sensors that it is beside a door, it assigns higher probabilities to the positions/cells near the doors. After moving one meter forward, the new belief is smoother due to the inherent noise in robot motion, because the uncertainty has increased. Besides, the robot's motion is estimated using its proprioceptive sensors like wheel encoders. Next, it detects that it is besides a door again. When it sees the second door, the probability to be at the second door rises sharply because it is the only option according to its environment model, thus it diminishes its belief to be at the locations close to the other doors.

The Markov Localization and its extension to particle filters known as Monte Carlo Localization (MCL) [20], form the foundation of modern probabilistic localization. Virtually all localization and SLAM algorithms rely on the *Markov property* Eq. 2.1, which gives its name to the Markov Localization algorithm, to model their probability distributions. The *Markov property* assumes that the current state's probability distribution depends only on the previous state and that the future state depends only upon the present state and not on any sequence of past states. This only holds if the environment is *static* and doesn't change with time. Such an assumption is not realistic for typical human environments. If a moving object is erroneously associated as a landmark in the map, the localization will often fail and the map will deteriorate. The usual approach to cope with dynamic landmarks like humans is to classify objects as static or dynamic [53]. This way the static landmarks are used to build the map and the dynamic landmarks are tracked separately.

$$p(x_{t+1}|x_0...x_t) = p(x_{t+1}|x_t) \tag{2.1}$$

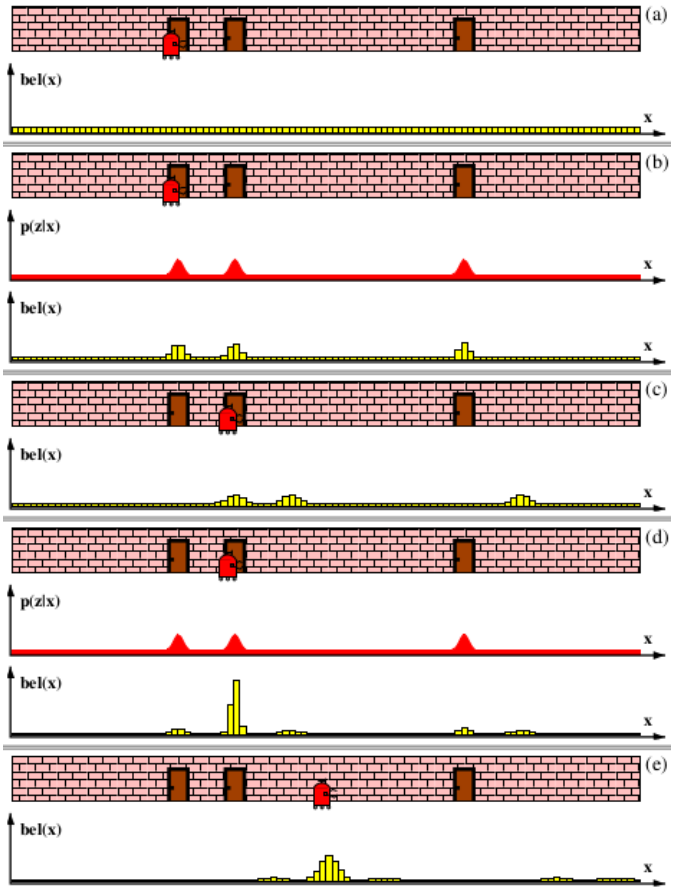


Figure 2.1: Markov Localization. From Thrun *et al.* [51]

2.1.1 Mathematical Definition of SLAM

Let us assume that the robot travels along a path given by $x_{1:T} = \{x_1, ..., x_T\}$ in an unknown environment. While moving, the robot acquires a sequence of odometry measurements $u_{1:T} = \{u_1, ..., u_T\}$ and observations $z_{1:T} = \{z_1, ..., z_T\}$. Two main forms of the problem are distinguished in the literature, *full* and *online* SLAM. Solving the *full* SLAM consists of estimating the posterior probability of the robot's path $x_{1:T}$ together with the map m given the initial location x_0, the measurements $u_{1:T}$ and the observations $z_{1:T}$:

$$p(x_{1:T}, m | z_{1:T}, u_{1:T}) \tag{2.2}$$

On the other hand, *online* SLAM tries to recover the current robot location instead of the whole path:

$$p(x_t, m | z_{1:T}, u_{1:T}) \tag{2.3}$$

Two more mathematical models are necessary to solve either SLAM problems. On the one hand, a model that relates measurements z_t to the map m and the robot location x_t: $z_t = h(x_t, m)$, and on the other hand, a model that relates the odometry u_t to the robot locations x_{t-1} and x_t: $x_t = f(x_{t-1}, u_t)$. A graphical model of the SLAM problem can be seen at figure 2.2.

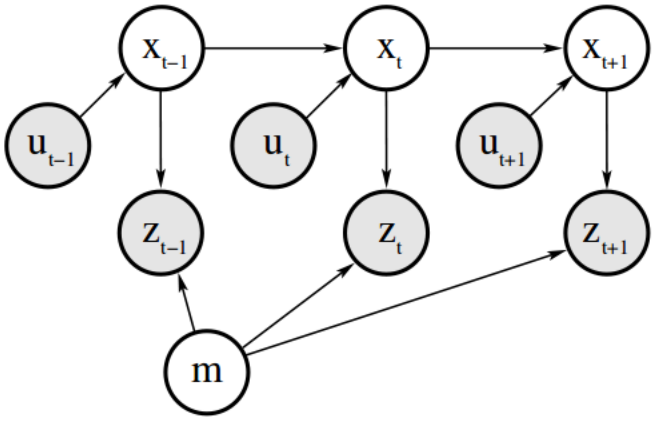
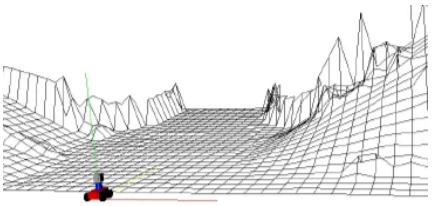


Figure 2.2: Graphical model of SLAM. Arcs indicate causal relationships, and shaded nodes are directly observable to the robot. From Thrun *et al.* [49]

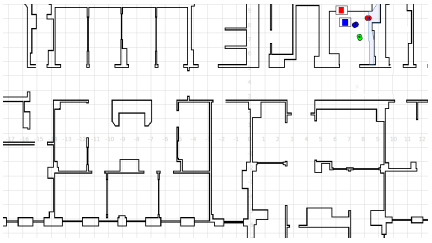
Note that on purely visual SLAM systems, such as the ScaViSLAM framework [46] this thesis is built upon, odometry information is not available. Instead, visual odometry techniques such as optical flow are used to estimate the camera's pose. Contrary to classical SLAM approaches where the robot's pose is

tracked in the two dimensional ground plane, in visual SLAM the pose of the camera is estimated in 3D.

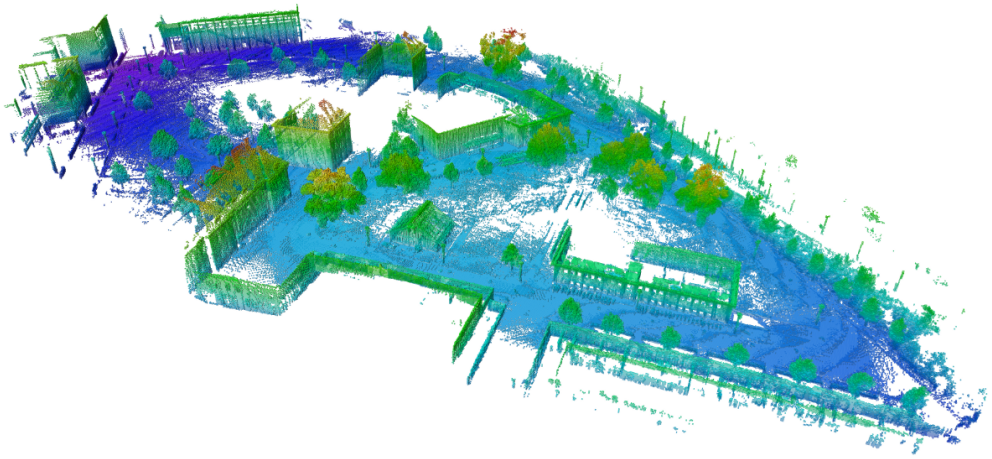
Regarding the poses, they usually are represented in the space of *SE(2)* or *SE(3)* rigid transformations, depending on the dimensions. The map can be represented in many different ways. Some of these representations, which can be seen at figure 2.3 , are point clouds, topological maps, occupancy grids in 2D or 3D like OctoMap and elevation maps.



(a) Extended Elevation Map by Pfaff *et al.* [40]



(b) 2D occupancy grid



(c) OctoMap by Hornung *et al.*[28]

Figure 2.3: Examples of different map representations.

Next, the three major paradigms for solving feature based SLAM will be discussed briefly: EKF SLAM, Particle Filter SLAM and GraphSLAM.

2.1.2 EKF SLAM

This is the earliest and the most studied approach, which relies on the extended Kalman Filter (EKF) to estimate the current robot's pose. EKF is an extension of the popular Kalman Filter for non-linear systems. As any EKF algorithm, EKF SLAM makes a Gaussian noise assumption for the robot motion and perception. As it can be seen in Eq. 2.4, all the past information is summarized in an extended state vector which includes the robot's pose and the position of the n landmarks/features: $\mu_t = (\vec{x}_t, \theta_t, m_1 ... m_n)$. Associated with the state vector is a covariance matrix Σ_t. This covariance matrix is quadratic on the size of the state vector and thus makes the algorithm computationally expensive for large areas.

$$p(x_t, m | z_{1:T}, u_{1:T}) \sim N(\mu_t, \Sigma_t) \tag{2.4}$$

Three covariances appear in the matrix, which can be seen at Eq. 2.5. The top left submatrix in yellow represents the uncertainty in the estimation of the robot's pose. The off-diagonal elements in green of the matrix represent the correlations in the estimates of the different variables like landmark's and the robot's pose. Finally, the submatrix in blue represents the map.

$$N(\mu_t, \Sigma_t) = \begin{pmatrix} x \\ y \\ \theta \\ l_1 \\ l_2 \\ \vdots \\ l_n \end{pmatrix}, \begin{pmatrix} \sigma_x^2 & \sigma_{xy} & \sigma_{x\theta} & \sigma_{xl_1} & \sigma_{xl_2} & \cdots & \sigma_{xl_n} \\ \sigma_{xy} & \sigma_y^2 & \sigma_{y\theta} & \sigma_{yl_1} & \sigma_{yl_2} & \cdots & \sigma_{yl_n} \\ \sigma_{x\theta} & \sigma_{y\theta} & \sigma_\theta^2 & \sigma_{\theta l_1} & \sigma_{\theta l_2} & \cdots & \sigma_{\theta l_n} \\ \sigma_{xl_1} & \sigma_{yl_1} & \sigma_{\theta l_1} & \sigma_{l_1}^2 & \sigma_{l_1 l_2} & \cdots & \sigma_{l_1 l_n} \\ \sigma_{xl_2} & \sigma_{yl_2} & \sigma_{\theta l_2} & \sigma_{l_1 l_2} & \sigma_{l_2}^2 & \cdots & \sigma_{l_2 l_n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \sigma_{xl_n} & \sigma_{yl_n} & \sigma_{\theta l_n} & \sigma_{l_1 l_n} & \sigma_{l_2 l_n} & \cdots & \sigma_{l_n}^2 \end{pmatrix} \tag{2.5}$$

At first, when the first measurements are taken, the covariance matrix is filled assuming that the features are uncorrelated, so the off-diagonal elements are zero. As the robot moves and takes more measurements, the features start becoming correlated. While moving, the uncertainty about the robot's egopose grows. Consequently, this gets reflected in the increasing uncertainty about the landmark's location over time.

It gets interesting when the robot sees an already mapped landmark and performs a loop closure. Thanks to this observation, the error of its pose is reduced as it can be seen in figure 2.4. Also, this observation helps reducing the estimate of other landmarks thanks to the correlation expressed in the covariance matrix. This means that any information gain about the robot's pose propagates through the map, improving the localization of the other landmarks. On

the other hand, it is obvious that false data associations can have disastrous effects on the map because of the correlation.

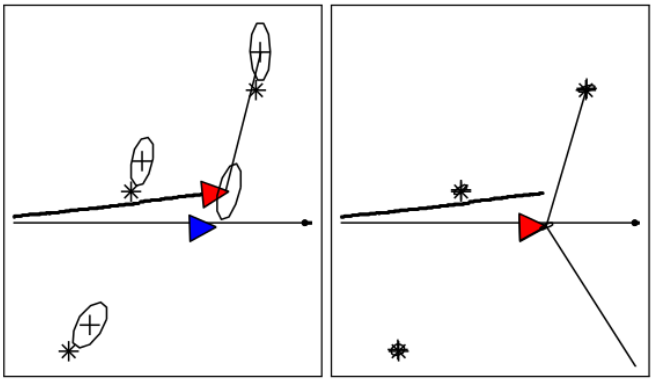


Figure 2.4: If the robot revisits an area and sees an already known landmark again, it can reduce the uncertainty about its location and also the uncertainty about other landmarks. At the end of the loop closure the error ellipses are much smaller. From Hertzberg *et al.* [27]

To summarise, the key limitation of EKF SLAM is the size of the covariance matrix, which grows quadratically with the number of landmarks/features, which it makes computationally expensive for large scale environments. The approach to solve this problem is to sparsify the matrix via approximations. Besides, EKF SLAM assumes that the data association problem is solved and is not robust to false data associations. Taking all this facts into account, it is not surprising that EKF SLAM has recently lost popularity.

2.1.3 Particle Filter SLAM

Particle Filter SLAM also addresses the problem of *online* SLAM. If EKF SLAM represented the uncertainties as a normal distribution, Particle Filters enable to represent any kind of distribution and to estimate non-linear, non-Gaussian processes. Particle Filters represent a distribution as a set of particles, which have been sampled randomly, from the state space and have an associated weight. In other words, a particle represents a concrete guess of the state, which includes the positions of the landmarks.

The algorithm of a particle filter works as follows:

1. Sample initial particles
2. Apply motion predictions to each particle
3. Make measurements
4. Compare each particle's prediction with the actual measurements and assign the normalized weights accordingly, giving particles with a good prediction higher weights.
5. Perform resampling/bootstrapping (draw with replacement from the last particle set) with unitary weights, in order to avoid the problem of degeneracy of the algorithm, which means that all but one weights are close to zero.

The key problem of Particle Filters in the context of SLAM is, that the number of needed particles grows *exponentially* with the dimension of the state space. At first this seems as a big disadvantage over EKF SLAM, since Gaussians scale between linearly and quadratically with the number of dimensions of the estimation problem. The trick to make this approach amenable to SLAM was first discovered by Murphy *et al.* [36] in 1999 and exploited by Montemerlo *et al.* [34] in the FastSLAM algorithm in 2002. FastSLAM solves the posterior using a factorization called Rao-Blackwellization. The factorization, shown in Eq. 2.6, exploits the dependency between the map and the poses of the robot. Given the robot's pose, mapping is not that difficult. The first term of the factorization represents the robot's pose posterior, which can be computed with the Monte Carlo Localization algorithm for instance. The second term represents the position of the landmarks. Then, if the robot's true path is known, the position of the landmarks are independent between them as can be seen in figure 2.5. Thus, the second term can be computed efficiently, with two dimensional EKFs for each landmark in the case of FastSLAM. This drastic reduction of the state space dimension makes Particle Filtering SLAM possible. As a consequence, FastSLAM can be implemented in $\mathcal{O}(N \log M)$.

$$p(x_{1:T}, l_{1:n}|z_{1:T}, u_{1:T}) = p(x_{1:T}|z_{1:T}, u_{1:T}) \cdot p(l_{1:n}|z_{1:T}, u_{1:T}) \tag{2.6a}$$
$$= p(x_{1:T}|z_{1:T}, u_{1:T}) \cdot p(l_{1:n}|x_{1:T}, z_{1:T}, u_{1:T}) \tag{2.6b}$$
$$= p(x_{1:T}|z_{1:T}, u_{1:T}) \cdot p(l_{1:n}|x_{1:T}, z_{1:T}) \tag{2.6c}$$
$$= p(x_{1:T}|z_{1:T}, u_{1:T}) \prod_{i=1}^{n} p(l_i|x_{1:T}, z_{1:T}) \tag{2.6d}$$
$$\tag{2.6e}$$

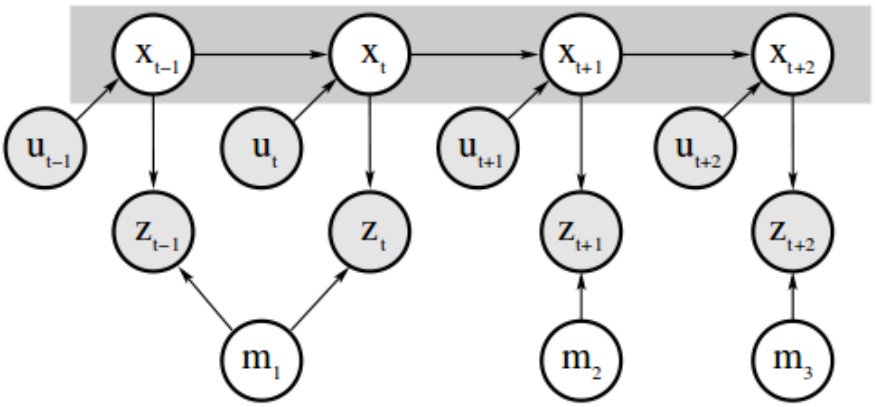


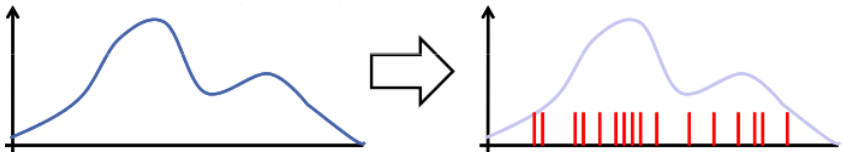
Figure 2.5: SLAM illustrated as a Bayes Network graph. Given the robot's path, the landmark variables are all disconnected (i.e. conditionally independent). From Thrun *et al.* [49]

FastSLAM makes data-associations on per particle basis, which allows the exploration of multiple hypothesis. This makes FastSLAM more robust to ambiguity in data association or even allows for delays until the uncertainty shrinks.

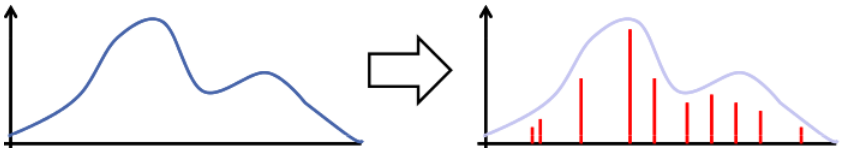
One of the sampling techniques that can be used for Particle Filter SLAM is importance sampling, which is illustrated in figure 2.6c. Importance sampling enables taking samples from another distribution and making inferences rather than from the distribution of interest. In case of FastSLAM 1.0, the proposal distribution is the motion model $x_t \sim p(x_{1:t}|x_{1:t-1}, u_{1:t})$. One problem of this approach arises when the uncertainty from the robot's motion is bigger than the measurement noise. In such cases, the sampled particles are spread very widely, but very few will fit with the measurements likelihood, resulting in a highly probable termination of the particles after the resampling phase.

In order to overcome this deficiency, an improved FastSLAM 2.0 algorithm was developed by Montemerlo *et al.*[35], which also considers measurements

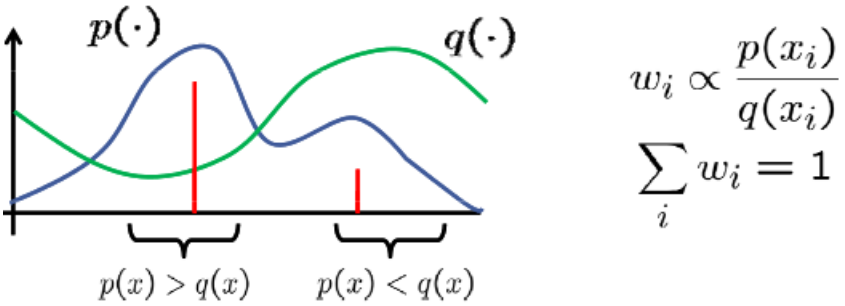
during sampling and consequently draws from $x_t \sim p(x_{1:t}|x_{1:t-1}, u_{1:t}, z_{1:t})$. This enables more accurate sampling and maps and the reduction on the number of particles needed.



(a) Random sampling. Source: Martin Lauer



(b) Random weighted sampling. Source: Martin Lauer



(c) Importance sampling: p is the target distribution and q the proposal sampled distribution. Source: Martin Lauer

Figure 2.6: Illustration of different sampling strategies.

To summarize, FastSLAM can represent multimodal beliefs due to its particle filter and scales very well with the number of landmarks (over 1 million) thanks to the Rao-Blackwell factorization. Also, it is robust to ambiguities in data-association.

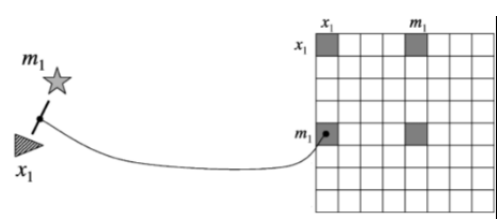
2.1.4 Graph based SLAM

Graph based SLAM techniques recover not only the current robot's pose, but all the path, solving the *full* SLAM problem by using nonlinear sparse optimization methods. The basic idea of GraphSLAM is that SLAM can be represented as a sparse graph of nodes and constraints between nodes. The nodes represent landmarks and robot locations. There are two types of edges: edges with odometry information between two robot poses and the arc between a pose and a landmark observation. Thus, we have motion arcs and measurement arcs. These nonlinear constraints should not be thought of as rigid constraints, but as soft contrainst that act like "springs" in a spring-mass system.

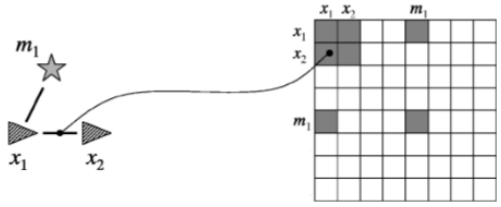
Consequently it is possible solve the *full* SLAM problem by relaxing these constraints and by computing the state of minimal energy of the network. In other words, after building the graph, one seeks for a node configuration which minimizes the constraint's error. This yields a maximum likelihood map and a corresponding set of robot poses. This means that graph based SLAM is usually divided in two tasks: a frontend which builds the graph, and a backend which tries to estimate the robot's pose given the edges by using graph optimization techniques. The frontend depends heavily on the sensor used, but the backend usually relies on some abstract data representation that is sensor agnostic. An example of this is the g^{2}o framework [31], which will be discussed later. Due to the fact that there are more observations than states, the backend has to solve an overdetermined system using standard least squares optimization techniques like Gauss-Newton or Levenberg-Marquardt.

In order to linearize the nonlinear constraints, a sparse information matrix was used in the GraphSLAM algorithm by Thrun *et al.* [50] which was presented in 2006. Using an information matrix has the advantage that for large scale maps, many of the off-diagonal components of the normalised information matrix are near zero. The GraphSLAM algorithm allows setting these values to zero, sparsifying the matrix and enabling more efficient information estimates and updates. Figure 2.7 illustrates how the information matrix is constructed and factorized.

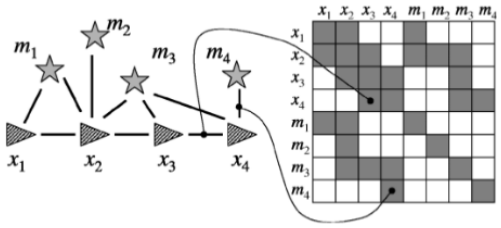
If the robot senses landmark m_1, a value, representing an arc, is added to the elements between x_1 and m_1, as shown in 2.7a. After the robot moves, an odometry reading u_2 leads to an arc between nodes x_1 and x_2 (2.7b). Consecutive application of these two basic steps leads to a graph of increasing size, as illustrated in 2.7c. Nevertheless this graph is sparse, in that each node is only connected to a small number of other nodes. The number of constraints in the graph is at worst linear in the time elapsed and in the number of nodes in the graph.



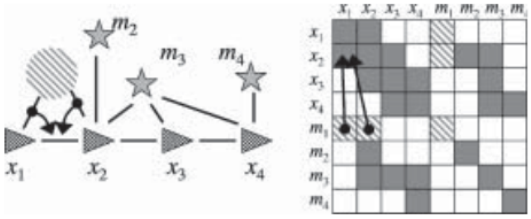
(a) Observation ls landmark m_1



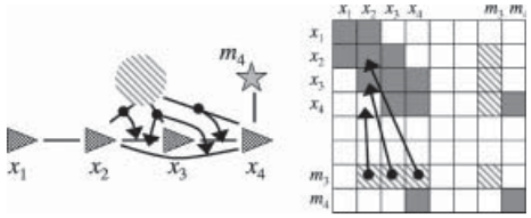
(b) Robot motion from x_1 to x_2



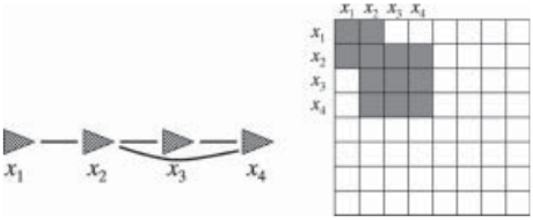
(c) Several steps later



(d) The removal of m_1 changes the link between x_1 and x_2



(e) The removal of m_3 introduces a new link between x_2 and x_4



(f) Final result after removing all map features

Figure 2.7: Illustration of the graph construction and factorization of the information matrix. Taken from Thrun *et al.* [50]

After constructing the information matrix, the matrix is factorized by eliminating the direct links between a pose and a landmark by introducing new constraints between the poses. At last, the robot's path can be recovered by inverting the information matrix. Finally, the landmark locations can be recovered one after another, using the original pose-to-landmark information. More details can be consulted in [50] and [49].

One of the advantages of graph based SLAM techniques over filtering methods is that it allows to revisit all data when building the map. This enables performing lazy data-association, i.e. to revisit old data-association in light of new information. Graph style techniques have usually a constant update-time and the required memory is linear with the number of landmarks. On the other hand, in EKF both factors scale quadratically with the number of features. As a disadvantage of GraphSLAM, if the robot's path is long, the optimization can be computationally expensive.

2.2 Visual SLAM

Visual SLAM is the research subfield of solving SLAM by just using visual information from different camera sensors, such as stereo-camera systems, monocular cameras or RGB-D cameras. Digital cameras are currently much more affordable than range/bearing sensors. Therefore, having a robust visual SLAM system would enable lowering the manufacturing cost of autonomous systems such as service robots.

Visual SLAM is closely related to the *photogrammetry* and *Structure-from-Motion* computer vision research fields. This research areas aim to infer the 3D structure from different photographs. The main technique used in these disciplines has been *bundle adjustment*, which is an iterative batch optimisation technique. First, given a set of corresponding points in images taken from different viewpoints, a 3D point cloud is created. Bundle adjustment boils down to minimizing the reprojection error of the three dimensional model and the associated points in the image. To achieve this, nonlinear least-squares algorithms like Levenberg–Marquardt have proven successful.

However, historically filtering methods have been favoured for real time visual SLAM applications due to the computational complexity of batch graph optimisation techniques. In 2007, Klein and Murray presented *PTAM* [30], a real time monocular SLAM system. They splitted tracking and mapping in two separate tasks. The frontend tracked the camera motion in real time and extracted a carefully chosen set of keyframes. The backend, which can operate at a lower frequency, concurrently creates a map by performing Bundle Adjustment over all the points in the map using some keyframes. One of the limitations of PTAM has been, that it doesn't scale well with bigger workspaces.

Visual SLAM techniques can be usually categorized as either filtering based ones or Bundle Adjustment based ones. Strasdat *et al.* [48] have recently shown that keyframe bundle adjustment outperforms filtering, since it gives the most accuracy per unit of computing time.

With the advent of powerful commodity GPGPU processors, *dense* methods for visual SLAM have been recently favoured in contrast to feature based approaches. Some of the advantages of *dense*, every pixel, methods like *DTAM* or *KinectFusion* by Newcombe *et al.* [38] [37], is that the created maps contain very rich and detailed 3D models of the environment and therefore can be used as input for other tasks like robotic grasping, in the case of service robots. Feature based maps on the other hand, have no environmental information between the tracked features. This leads to reliance on other sensors, like laser scanners, to find out if an obstacle exists between the features for instance.

2.2.1 ScaViSLAM

ScaViSLAM[1] by Strasdat *et al.* [47] is a state-of-the-art real-time visual SLAM framework based on keyframe Bundle Adjustment. Besides, it is the foundation of this thesis.

ScaViSLAM can be used with stereo-systems and RGB-D cameras. Despite a monocular camera being the most affordable of all visual sensors, it is also the most complicated to perform SLAM. This extra difficulty originates from the fact, that one has to estimate the depth of a feature point by triangulating from two different frames and perspectives.

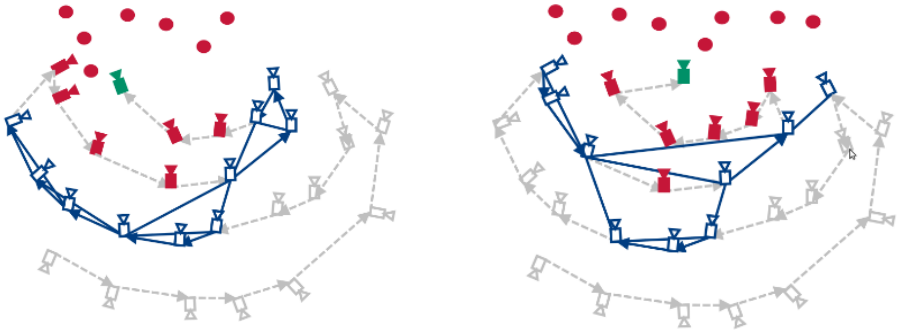
None of the SLAM solutions already described are appropriate for small and large scale environments. Opposite, the focus of *ScaViSLAM* is to present a unified visual SLAM solution for small and large scale environments, which until now have been tackled separately. To achieve this, a double window approach is used, which is illustrated in figure 2.8. The inner window is filled with point-pose constraints (as in bundle adjustment) and the outer window with pose-pose constraints (as in pose graph optimisation). The algorithm automatically builds a suitable connected graph of keyposes and constraints, dynamically selects inner and outer window membership and optimises both simultaneously in constant time. Optimisation if performed by using the g^{2}o framework [31] as the backend.

For loop closures, *ScaViSLAM* handles loopy local browsing by combining metric loop closures with top-down feature search in local neighbourhoods of the graph topology. Besides, large scale loop closures are handled with appearance-based place recognition, which is a standard *bag of visual words*[12] using SURF features.

Despite *ScaViSLAM* being a state-of-the-art visual SLAM framework, it crashes everytime the tracker gets lost or when the camera performs a sudden fast movement. Therefore, the aim of this thesis is to improve upon *ScaViSLAM* by also tracking lines, in order to cope with textureless environments.

Strasdat *et al.* claim to have implemented a modified version of *PTAM* as the frontend of *ScaViSLAM* for monocular systems. Apparently, distributing this implementation is not possible due to *PTAM*'s licence. There seems to be some confusion with this licence, because modified versions of PTAM that are freely available as open source exist. For example, ETHZ's version[2] for unmanned micro aerial vehicles [55].

[1] Source code available at `https://github.com/strasdat/ScaViSLAM`
[2] Source code available at `https://github.com/ethz-asl/ethzasl_ptam`



(a) Example graph (time step 1) (b) Example graph (time step 2)

Figure 2.8: Illustrations of the Double Window Optimization (DWO) framework. Keyframes and points in the inner window are shown in red, while keyframes in the outer window are shown in blue. The current reference keyframe is shown in green. From Strasdat *et al.* [46]

2.3 g²o Framework

g²o is a C++ framework for performing the optimization of nonlinear least squares problems that can be embedded as a graph or in a hyper-graph[3]. A hyper-graph is an extension of a graph where an edge can connect not only two, but multiple nodes.

2.3.1 Least Squares Optimization

A least squares minimization problem can be described by the following equations:

$$\mathbf{F}(\mathbf{x}) = \sum_{k \in \mathcal{C}} \underbrace{\mathbf{e}_k(\mathbf{x}_k, \mathbf{z}_k)^T \mathbf{\Omega}_k \mathbf{e}_k(\mathbf{x}_k, \mathbf{z}_k)}_{\mathbf{F}_k} \tag{2.7}$$

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmin}} \mathbf{F}(\mathbf{x}). \tag{2.8}$$

Here

- $\mathbf{x} = (\mathbf{x}_1^T, \ \dots \ , \mathbf{x}_n^T)^T$ is a vector of parameters, where each $\mathbf{x}_i$ represents a generic parameter block.

[3]This introduction has been taken literally from the official documentation, available at `https://github.com/RainerKuemmerle/g2o/blob/master/doc/g2o.pdf`

- $\mathbf{x}_k = (\mathbf{x}_{k_1}^T, \ \dots \ , \mathbf{x}_{k_q}^T)^T \subset (\mathbf{x}_1^T, \ \dots \ , \mathbf{x}_n^T)^T$ is the subset of the parameters involved in the k^{th} constraint.
- $\mathbf{z}_k$ and $\mathbf{\Omega}_k$ represent respectively the mean and the information matrix of a constraint relating the parameters in $\mathbf{x}_k$.
- $\mathbf{e}_k(\mathbf{x}_k\mathbf{z}_k)$ is a vector error function that measures how well the parameter blocks in $\mathbf{x}_k$ satisfy the constraint $\mathbf{z}_k$. It is $\mathbf{0}$ when $\mathbf{x}_k$ and $\mathbf{x}_j$ perfectly match the constraint. As an example, if one has a measurement function $\hat{\mathbf{z}}_k = \mathbf{h}_k(\mathbf{x}_k)$ that generates a synthetic measurement $\hat{\mathbf{z}}_k$ given an actual configuration of the nodes in $\mathbf{x}_k$. A straightforward error function would then be $\mathbf{e}(\mathbf{x}_k, \mathbf{z}_k) = \mathbf{h}_k(\mathbf{x}_k) - \mathbf{z}_k$.

If a good initial guess $\breve{\mathbf{x}}$ of the parameters is known, a numerical solution of Eq. 2.8 can be obtained by using the popular Gauss-Newton or Levenberg-Marquardt algorithms. The idea is to approximate the error function by its first order Taylor expansion around the current initial guess $\breve{\mathbf{x}}$

$$\begin{aligned} \mathbf{e}_k(\breve{\mathbf{x}}_k + \mathbf{\Delta x}_k) &= \mathbf{e}_k(\breve{\mathbf{x}} + \mathbf{\Delta x}) && (2.9) \\ &\simeq \mathbf{e}_k + \mathbf{J}_k \mathbf{\Delta x}. && (2.10) \end{aligned}$$

Here $\mathbf{J}_k$ is the Jacobian of $\mathbf{e}_k(\mathbf{x})$ computed in $\breve{\mathbf{x}}$ and $\mathbf{e}_k \overset{\text{def.}}{=} \mathbf{e}_k(\breve{\mathbf{x}})$. Substituting Eq. 2.10 in the error terms $\mathbf{F}_k$ of Eq. 2.7, we obtain

$$\begin{aligned} &\mathbf{F}_k(\breve{\mathbf{x}} + \mathbf{\Delta x}) && (2.11) \\ &= \mathbf{e}_k(\breve{\mathbf{x}} + \mathbf{\Delta x})^T \mathbf{\Omega}_k \mathbf{e}_k(\breve{\mathbf{x}} + \mathbf{\Delta x}) && (2.12) \\ &\simeq (\mathbf{e}_k + \mathbf{J}_k \mathbf{\Delta x})^T \mathbf{\Omega}_k (\mathbf{e}_k + \mathbf{J}_k \mathbf{\Delta x}) && (2.13) \\ &= \underbrace{\mathbf{e}_k^T \mathbf{\Omega}_k \mathbf{e}_k}_{\mathrm{c}_k} + 2 \underbrace{\mathbf{e}_k^T \mathbf{\Omega}_k \mathbf{J}_k}_{\mathbf{b}_k} \mathbf{\Delta x} + \mathbf{\Delta x}^T \underbrace{\mathbf{J}_k^T \mathbf{\Omega}_k \mathbf{J}_k}_{\mathbf{H}_k} \mathbf{\Delta x} && (2.14) \\ &= \mathrm{c}_k + 2\mathbf{b}_k \mathbf{\Delta x} + \mathbf{\Delta x}^T \mathbf{H}_k \mathbf{\Delta x} && (2.15) \end{aligned}$$

With this local approximation, the function $\mathbf{F}(\mathbf{x})$ given in Eq. 2.7 can be rewritten as

$$\begin{aligned} \mathbf{F}(\breve{\mathbf{x}} + \mathbf{\Delta x}) &= \sum_{k \in \mathcal{C}} \mathbf{F}_k(\breve{\mathbf{x}} + \mathbf{\Delta x}) && (2.16) \\ &\simeq \sum_{k \in \mathcal{C}} \mathrm{c}_k + 2\mathbf{b}_k \mathbf{\Delta x} + \mathbf{\Delta x}^T \mathbf{H}_k \mathbf{\Delta x} && (2.17) \\ &= \mathrm{c} + 2\mathbf{b}^T \mathbf{\Delta x} + \mathbf{\Delta x}^T \mathbf{H} \mathbf{\Delta x}. && (2.18) \end{aligned}$$

The quadratic form in Eq. 2.18 is obtained from Eq. 2.17 by setting $\mathrm{c} = \sum \mathrm{c}_k$, $\mathbf{b} = \sum \mathbf{b}_k$ and $\mathbf{H} = \sum \mathbf{H}_k$. It can be minimized in $\mathbf{\Delta x}$ by solving the linear system

$$\mathbf{H}\, \mathbf{\Delta x}^* = -\mathbf{b}. \qquad (2.19)$$

$\mathbf{H}$ is the information matrix of the system and is sparse by construction, having non-zeros only between blocks connected by a constraint. Its number of non-zero blocks is twice the number of constrains plus the number of nodes. This allows to solve Eq. 2.19 with efficient approaches like sparse Cholesky factorization or Preconditioned Conjugate Gradients (PCG). The linearized solution is then obtained by adding to the initial guess the computed increments

$$\mathbf{x}^* \quad = \quad \breve{\mathbf{x}} + \boldsymbol{\Delta}\mathbf{x}^*. \tag{2.20}$$

The popular Gauss-Newton algorithm iterates the linearization in Eq. 2.18, the solution in Eq. 2.19 and the update step in Eq. 2.20. In every iteration, the previous solution is used as linearization point and as initial guess.

The Levenberg-Marquardt (LM) algorithm is a nonlinear variant to Gauss-Newton that introduces a damping factor and backup actions to control the convergence. Instead of solving directly Eq. 2.19 LM solves a damped version of it

$$(\mathbf{H} + \lambda\mathbf{I})\,\boldsymbol{\Delta}\mathbf{x}^* \quad = \quad -\mathbf{b}. \tag{2.21}$$

Here λ is a damping factor: the larger the λ, the smaller are the $\boldsymbol{\Delta}\mathbf{x}$ terms. This is useful to control the step size in case of non-linear surfaces. The idea behind the LM algorithm is to dynamically control the damping factor. At each iteration the error of the new configuration is monitored. If the new error is lower than the previous one, lambda is decreased for the next iteration. Otherwise, the solution is reverted and lambda is increased.

The procedures described above are a general approach to multivariate function minimization. The general approach, however, assumes that the space of parameters $\mathbf{x}$ is Euclidean, which is not valid for several problems like SLAM or bundle adjustment. This may lead to sub-optimal solutions.

2.3.2 Least Squares on Manifolds

To deal with parameter blocks that span over a non-Euclidean spaces, it is common to apply error minimization on a manifold. A manifold is a mathematical space that is not necessarily Euclidean on a global scale, but can be seen as Euclidean on a local scale.

For example, in the context of SLAM problem, each parameter block $\mathbf{x}_i$ consists of a translation vector $\mathbf{t}_i$ and a rotational component α_i. The translation $\mathbf{t}_i$ clearly forms a Euclidean space. In contrast, the rotational components α_i span over the non-Euclidean 2D or 3D rotation group $SO(2)$ or $SO(3)$. To avoid singularities, these spaces are usually described in an over-parameterized way,

e.g., by rotation matrices or quaternions. Directly applying Eq. 2.20 to these over-parameterized representations breaks the constraints induced by the over-parameterization. The over-parameterization results in additional degrees of freedom and thus introduces errors in the solution. To overcome this problem, one can use a minimal representation for the rotation (like Euler angles in 3D). This, however, is then subject to singularities.

An alternative idea is to consider the underlying space as a manifold and to define an operator $\boxplus$ that maps a local variation $\boldsymbol{\Delta}\mathbf{x}$ in the Euclidean space to a variation on the manifold, $\boldsymbol{\Delta}\mathbf{x} \mapsto \mathbf{x} \boxplus \boldsymbol{\Delta}\mathbf{x}$. With this operator, a new error function can be defined as

$$\begin{aligned} \breve{\mathbf{e}}_k(\boldsymbol{\Delta}\tilde{\mathbf{x}}_k) &\stackrel{\text{def.}}{=} \mathbf{e}_k(\breve{\mathbf{x}}_k \boxplus \boldsymbol{\Delta}\tilde{\mathbf{x}}_k) && (2.22) \\ &= \mathbf{e}_k(\breve{\mathbf{x}} \boxplus \boldsymbol{\Delta}\tilde{\mathbf{x}}) \simeq \breve{\mathbf{e}}_k + \tilde{\mathbf{J}}_k \boldsymbol{\Delta}\tilde{\mathbf{x}}, && (2.23) \end{aligned}$$

where $\breve{\mathbf{x}}$ spans over the original over-parameterized space, for instance quaternions. The term $\boldsymbol{\Delta}\tilde{\mathbf{x}}$ is a small increment around the original position $\breve{\mathbf{x}}$ and is expressed in a minimal representation. A common choice for $SO(3)$ is to use the vector part of the unit quaternion.

In more detail, one can represent the increments $\boldsymbol{\Delta}\tilde{\mathbf{x}}$ as 6D vectors $\boldsymbol{\Delta}\tilde{\mathbf{x}}^T = (\boldsymbol{\Delta}\tilde{\mathbf{t}}^T\ \tilde{\mathbf{q}}^T)$, where $\boldsymbol{\Delta}\tilde{\mathbf{t}}$ denotes the translation and $\tilde{\mathbf{q}}^T = (\Delta q_x\, \Delta q_y\, \Delta q_z)^T$ is the vector part of the unit quaternion representing the 3D rotation. Conversely, $\breve{\mathbf{x}}^T = (\breve{\mathbf{t}}^T\, \breve{\mathbf{q}}^T)$ uses a quaternion $\breve{\mathbf{q}}$ to encode the rotational part. Thus, the operator $\boxplus$ can be expressed by first converting $\boldsymbol{\Delta}\tilde{\mathbf{q}}$ to a full quaternion $\boldsymbol{\Delta}\mathbf{q}$ and then applying the transformation $\boldsymbol{\Delta}\mathbf{x}^T = (\boldsymbol{\Delta}\mathbf{t}^T\, \boldsymbol{\Delta}\mathbf{q}^T)$ to $\breve{\mathbf{x}}$. In the equations describing the error minimization, these operations can nicely be encapsulated by the $\boxplus$ operator. The Jacobian $\tilde{\mathbf{J}}_k$ can be expressed by

$$\tilde{\mathbf{J}}_k = \left.\frac{\partial \mathbf{e}_k(\breve{\mathbf{x}} \boxplus \boldsymbol{\Delta}\tilde{\mathbf{x}})}{\partial \boldsymbol{\Delta}\tilde{\mathbf{x}}}\right|_{\boldsymbol{\Delta}\tilde{\mathbf{x}}=\mathbf{0}}. \quad (2.24)$$

Since in the previous equation $\breve{\mathbf{e}}$ depends only on $\boldsymbol{\Delta}\tilde{\mathbf{x}}_{k_i} \in \boldsymbol{\Delta}\tilde{\mathbf{x}}_k$ it is possible to further expand it as follows:

$$\begin{aligned} \tilde{\mathbf{J}}_k &= \left.\frac{\partial \mathbf{e}_k(\breve{\mathbf{x}} \boxplus \boldsymbol{\Delta}\tilde{\mathbf{x}})}{\partial \boldsymbol{\Delta}\tilde{\mathbf{x}}}\right|_{\boldsymbol{\Delta}\tilde{\mathbf{x}}=\mathbf{0}} && (2.25) \\ &= \begin{pmatrix} \mathbf{0} \cdots \mathbf{0}\ \tilde{\mathbf{J}}_{k_1}\ \cdots\ \tilde{\mathbf{J}}_{k_i}\ \cdots \mathbf{0}\ \cdots\ \tilde{\mathbf{J}}_{k_q} \mathbf{0} \cdots \mathbf{0} \end{pmatrix}. && (2.26) \end{aligned}$$

With a straightforward extension of notation, we set

$$\tilde{\mathbf{J}}_{k_i} = \left.\frac{\partial \mathbf{e}_k(\breve{\mathbf{x}} \boxplus \boldsymbol{\Delta}\tilde{\mathbf{x}})}{\partial \boldsymbol{\Delta}\tilde{\mathbf{x}}_{k_i}}\right|_{\boldsymbol{\Delta}\tilde{\mathbf{x}}=\mathbf{0}} \quad (2.27)$$

With a straightforward extension of the notation, it is possible to insert Eq. 2.23 in Eq. 2.13 and Eq. 2.16. This leads to the following increments:

$$\tilde{\mathbf{H}}\,\boldsymbol{\Delta}\tilde{\mathbf{x}}^{*} \;=\; -\tilde{\mathbf{b}}. \tag{2.28}$$

Since the increments $\boldsymbol{\Delta}\tilde{\mathbf{x}}^{*}$ are computed in the local Euclidean surroundings of the initial guess $\breve{\mathbf{x}}$, they need to be re-mapped into the original redundant space by the $\boxplus$ operator. Accordingly, the update rule of Eq. 2.20 becomes

$$\mathbf{x}^{*} \;=\; \breve{\mathbf{x}} \boxplus \boldsymbol{\Delta}\tilde{\mathbf{x}}^{*}. \tag{2.29}$$

In summary, formalizing the minimization problem on a manifold consists of first computing a set of increments in a local Euclidean approximation around the initial guess by Eq. 2.28, and second accumulating the increments in the global non-Euclidean space by Eq. 2.29.

2.4 Kinect Camera Sensor

At the end of 2010, Microsoft launched its structured light camera called Kinect, illustrated in figure 2.9, as a peripheral for the Xbox 360 console and has sold over 24 million devices as of February 2013. The Kinect camera has sparked a revolution in perception and robotics, because it offers accurate depth information with a low cost price, several orders of magnitude cheaper than range laser sensors.

The depth sensor produces a 640×480 depth map, with up to 1 cm accuracy, and the RGB camera produces a 8 bit VGA (640×480) video stream. Both sensors operate at 30Hz frame-rate. The ranging limit is between 1.2 and 3.5 m and the camera also features a multi-array microphone and a motorized pivot.

Figure 2.9: The Microsoft Kinect camera. Picture taken by Evan-Amos and distributed under public domain on Wikipedia.

The camera embodies an Infra-Red projector which emits a *LightCoding* pattern onto the environment. If a point in the pattern is detected in the camera

image, the depth of the corresponding pixel can be estimated using triangulation. An IR monochrome CMOS sensor then receives the projected pattern and processes it within an embedded PS1080 chip. More details about *Light-Coding* can be found on the patent by Javier Garcia *et al.*[24].

While the quality of the depth map is remarkable, there are usually holes, due to to materials that don't reflect IR or very fast camera motion for instance. Nevertheless, the Kinect has been given many applications since its introduction. Some of these applications include robotics mobile navigation [17], reconstruction of 3D environments [37] and visual SLAM [22].

2.5 Straight Line Representations

As already mentioned, the main goal of this thesis is to include lines into the visual SLAM problem, in order to enable more robust localization in textureless environments like corridors, where few feature points can be found. Such primitives are especially numerous in man-made structured or semi-structured environments. Feature points appearance changes significantly with the cameras motion. Eventhough invariant feature point descriptors like SIFT [33] can be used, they come at the expense of computational cost. Besides, feature points are prone to occlusions or can be blurred by fast camera motion. Lines have the advantage that they provide richer information of the environment and are more robust to viewpoint changes and partial occlusions. Therefore, they can be a good complement to feature points.

A line is defined by the join of two points or the intersection of two planes. Consequently, lines have 4 degrees of freedom in 3-space (i.e. 3D euclidean space). Lines are difficult to represent in 3-space since a natural representation for an object with 4 degrees of freedom would be a homogeneous 5-vector. The problem is, that a homogeneous 5-vector cannot easily be used in mathematical expressions together with the 4-vectors representing points and planes. To overcome this, a number of line representations have been proposed [26], from which Plücker coordinates have been chosen for this thesis. The reason for this is, that there is apparently no other closed form line representation as a parameter vector. Instead of using two points, intersection of planes etc. Plücker coordinates allows using directly linear algebra and matrices, similar to the case of using feature points.

On the one hand, overparametrized line representations can cause numerical instabilities and increase computational complexity. On the other hand, using a non minimal line representation might simplify the implementation. Next, an introduction to Plücker Coordinates will proceed, after all they have been

used intensively in this thesis.

2.5.1 Plücker Coordinates

A line is represented by a 4×4 skew-symmetric homogeneous matrix[4]. In particular, the line joining the two points A, B is represented by the matrix L with elements:

$$l_{ij} = A_i B_j - B_i A_j$$

or in vector notation:

$$\mathrm{L} = \mathrm{A}\mathrm{B}^T - \mathrm{B}\mathrm{A}^T \tag{2.30}$$

The matrix L is illustrated in 2.31:

$$L = \begin{bmatrix} 0 & l_{12} & l_{13} & l_{14} \\ -l_{12} & 0 & l_{23} & l_{24} \\ -l_{13} & -l_{23} & 0 & l_{34} \\ -l_{14} & -l_{24} & l_{34} & 0 \end{bmatrix} \tag{2.31}$$

The matrix L has following important properties:

- The representation has the required 4 degrees of freedom for a line. This accounted as follows: the skew-symmetric matrix L has 6 independent non- zero elements, but only their 5 ratios are significant, and furthermore because $det(L) = 0$ the elements satisfy a quadratic constraint. The net number of degrees of freedom is then 4.
- The matrix L is independent of the points A, B used to define it, since if a different point C on the line is used we obtain the same matrix L.

The Plücker line coordinates are the six non-zero elements of the 4×4 skew-symmetric Plücker matrix in 2.30, namely[5]

$$\mathcal{L} = \{l_{12}, l_{13}, l_{14}, l_{23}, l_{42}, l_{34}\} \tag{2.32}$$

This is a homogeneous 6-vector, and thus is an element of $\mathbb{P}^5$. It follows from evaluating $det(L) = 0$ that the coordinates satisfy equation Eq. 2.33

[4]This introduction to Plücker Coordinates is based on Hartley and Zissermans excellent *Multiple View Geometry* book [26]

[5]The element l_{42} is conventionally used instead of l_{24} as it eliminates negatives in many of the subsequent formulae

$$l_{12}l_{34} + l_{13}l_{42} + l_{14}l_{23} = 0. \tag{2.33}$$

A 6-vector L only corresponds to a line in 3-space if it satisfies the Plücker constraint Eq. 2.33.

A dual Plücker representation L^* is obtained for a line formed by the intersection of two planes P, Q:

$$\mathrm{L}^* = \mathrm{P}\mathrm{Q}^T - \mathrm{Q}\mathrm{P}^T \tag{2.34}$$

and has similar properties to L. The matrix L^* can be obtained directly from L by a simple rewrite rule:

$$l_{12} : l_{13} : l_{14} : l_{23} : l_{42} : l_{34} = l^*_{34} : l^*_{42} : l^*_{23} : l^*_{14} : l^*_{13} : l^*_{12} \tag{2.35}$$

Plücker coordinates are useful in algebraic derivations. For instance, they are often used to map 3D lines into images. This is because, if a line in 3-space is represented by Plücker coordinates then its image can be expressed as a linear map on these coordinates. The map between the Plücker line coordinates L and the image line coordinates l (a 3-vector) is represented by a single 3×6 matrix $\mathcal{P}$, called the line projection matrix. The homogeneous image line coordinates are then given by:

$$l = \mathcal{P}\mathcal{L} \tag{2.36}$$

The line projection matrix $\mathcal{P}$ is computed as follows:

$$\mathcal{P} = \begin{bmatrix} p^2 \wedge p^3 \\ p^3 \wedge p^1 \\ p^1 \wedge p^2 \end{bmatrix} \tag{2.37}$$

where P^{iT} are the rows of the point camera matrix P, and $\mathrm{P}i \wedge \mathrm{P}j$ are the Plücker line coordinates of the intersection of the planes $\mathrm{P}i \wedge \mathrm{P}j$, which can be computed using Eq. 2.34.

$$\mathrm{P} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} = \begin{bmatrix} \mathrm{P}^{1T} \\ \mathrm{P}^{2T} \\ \mathrm{P}^{3T} \end{bmatrix} \tag{2.38}$$

The camera projection matrix is given by:

$$\mathrm{P} = KR[I| - c] \tag{2.39}$$

where K is the camera calibration matrix, R is rotation matrix representing the orientation of the camera coordinate frame and c represents the coordinates of the camera center in the world coordinate frame. This leads to:

$$\mathrm{P} = KIT = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \left[\begin{array}{c|c} R_{3x3} & t_{3x1} \\ \hline 0_{1x3} & 1 \end{array}\right] \tag{2.40}$$

where f_x, f_y are the cameras focal length expressed in pixel distances and c_x, c_y is the principal point, usually the image center, in pixel coordinates. T is a homogeneous 4x4 camera transformation matrix.

A second and more efficient method exists to project a 3D line into the image. After calculating the projection matrix P with Eq. 2.40, instead of applying Eq. 2.37 we compute:

$$[l]_x = PLP^T \tag{2.41}$$

where $[l]_x$ is defined for a 3x3 skew-symmetric matrix as follows:

$$[l]_x = \begin{bmatrix} 0 & -l_3 & l_2 \\ l_3 & 0 & -l_1 \\ -l_2 & l_1 & 0 \end{bmatrix} \tag{2.42}$$

Both methods, Eq. 2.37 and Eq. 2.41 were implemented in Matlab and the results showed that the second method is faster to compute with only 147 operations versus the 177 operations needed for the first one.

The result of both methods is a vector with the homogeneous parameter values for a general 2 dimensional line equation form $Ax + By + C * 1 = 0$.

Part II

Project Management

Table of Contents

Chapter 3

Project Scope Statement

Contents

3.1 Description and goal

The goal of this project is to explore the use of lines in order to improve a visual SLAM system. Commonly feature points are used as cues for visual odometry and loop closures, but this approach has the disadvantage that the localization fails when not enough feature points are found in the environment.

As a foundation for the project, the *ScaViSLAM*[1] software by Strasdat *et al.* [46] is used. This is a state-of-the-art real-time visual SLAM framework based on *Bundle Adjustment*. *ScaViSLAM* currently only supports Stereo and RGB-D systems. Because monocular cameras are the most affordable camera setup, implementing a monocular approach will be researched. Besides, in order to deploy *ScaViSLAM* on a mobile robot, the integration of it as a *ROS*[2] package will also be explored.

[1]Source code available at `https://github.com/strasdat/ScaViSLAM`

[2]Robot Operating System, more information available at `www.ros.org`

Regarding the visual frontend, a line tracker will be implemented. There are several issues which will have to be researched:

- Adequate line representations
- Transformation and projection of the lines to the next frame
- Search for candidate lines that are near the projection
- Matching lines using a line descriptor

Afterwards, this information should be integrated in *ScaViSLAM*'s backend and its graph optimization process. In order to achieve this, its double window approach needs to be extended to incorporate line information. For example, some of the functionalities/methods that have to be rewritten, in order to take advantage of the line information, are:

- Definition of the inner and outer windows
- Deciding when to add a new keyframe
- Metric loop closures
- Computation of the topological neighborhood of a keyframe
- Defining new covisibility weight: number of feature points and lines that are visible from the topological neighborhood

To sum up, the broad goals of the thesis are:

- Porting *ScaViSLAM* to ROS
- Rewriting *ScaViSLAM* to support Monocular cameras
- Implement a line tracker in the visual frontend of *ScaViSLAM*
- Integrate the line tracker's information in *ScaViSLAM*'s backend

3.2 Scope

List of deliverables

- *Project Scope Statement (PSS)*
  - Description and goal: project definition.
  - Scope: identify which tasks are going to be performed and which are out of scope

a) List of deliverables: list of documents needed to finish a part of the project.

b) List of subtasks: list of all the subtasks included in the project

c) WDS diagram: a structured decomposition of the project's scope and processes.

- Planning: time estimation for each task

a) Gantt chart: graphical illustration of a projects schedule.

- Risks: identification of risks and description of prevention and contingency plans.

- Work Methodology: description of the management and planning of work and archive. Explanation of methodology for decision taking.

- *Thesis:* document that describes all the project.

List of subtasks

- **Education**
  - *T- Training*
    * T1 Literature research
    * T2 *ScaViSLAM* software
- **Tactical Processes**
  - *M- Management*
    * M1 Meetings
      * M11 Perform Meetings
    * M2 Archive Management
      * M21 Use Git to perform version control management and backup
  - *P- Planning*
    * P1 Do PSS
      * P11 Define description and goal
      * P12 Scope
        * P121 Do Work breakdown structure (WDS)
        * P122 Make list of subtasks

        * P123 Make list of deliverables
    * P13 Time planning
        * P131 Do Gantt chart
        * P132 Update Gantt chart
        * P133 Do time estimations
    * P14 Risk identification
        * P141 Make list
        * P142 Make contingency plan
    * P15 Work methodology
        * P151 Specify work methodology
  * P2 Plan new iteration

- **Operational Processes**
  - *D- Development*
    * D1 Do requirement gathering
    * D2 Do the analysis
    * D3 Do the design
    * D4 Do the implementation
    * D5 Perform tests
- *D- Documentation creation*
  * D1 Write Thesis
- *F- Finish*
  * F1 Prepare presentation
  * F2 Do presentation

WDS diagram

3.1 shows the work breakdown structure of the project.

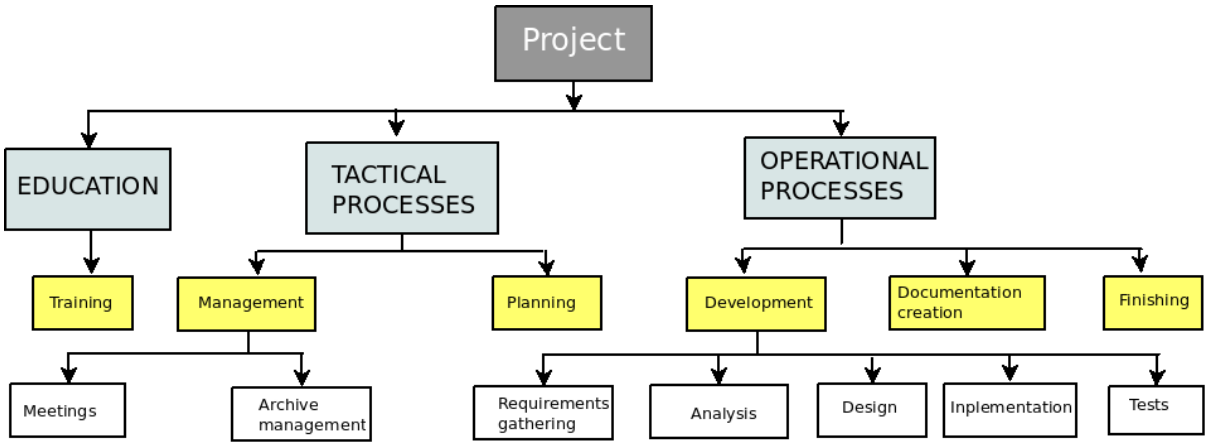


Figure 3.1: WDS diagram

3.3 Risks

When developing a project, there are some risk factors that can influence upon the accomplishment of deadlines or milestones, resulting in the failure of the planning and impacting on the project cost, schedule or performance. Some of these risk factors can be identified beforehand and contingency plans to reduce the impact can be elaborated. A list of identified risks and contingency plans are now presented. However, it is also necessary to take into account the unforeseen problems that could arise from the inexperience and lack of prior knowledge when developing this project.

List of Risks

–R1 *Technology related*

R11 Loss of data

Description The project can loose data caused by software, hardware failures or human mistakes. Examples include the computer falling on the floor and resulting in a broken hard drive, getting the computer stolen, power going out while working on a PC or errors in a data transfer. The potential impact and cost of this risk is very high, because it can force the repetition of already done work and therefore cause severe delays.

Prevention plan

i. Use *GIT*[3] to host all the project data in a external repository. Consequently we can use *GIT* as a DVCS as well as a

[3] *GIT* is a decentralized version control system(DVCS) created by Linus Torvals

backup system.

ii. Take good care of the hardware.

iii. Use Unix based operating systems, such as GNU/Linux and Mac OS X, because they are safer than Microsoft Windows. This results in a lower probability of getting the computer infected by viruses.

Contingency plan Recover data from the *GIT* repository.

R12 Camera sensor failure

Description The project relies on camera sensors like the Kinect, therefore a broken camera would result in the inability to continue work, causing big delays. In spite of the probability of breaking the camera being very low, the risk is ranked as medium since the impact would be very important.

Prevention plan Take good care of the cameras. Besides, it is possible with *ROS* to record camera data with the *rosbag* tool in order to play this data when there is no camera available.

Contingency plan Replace or buy a new camera as fast as possible. In the meantime use the with *rosbag* recorded data.

–R2 *Human related*

R21 Getting stuck because of lack of knowledge

Description Lack of prior knowledge can result in getting stuck when programming, leading to delays. Since *ScaViSLAM* is a very complex and huge software ($>$ 30000 lines of code) with scarce documentation, this is expected to happen often. Despite good prior knowledge of *ROS*, it can also happen to get stuck in *ROS* related issues. The author also has currently not much prior knowledge about projective geometry. Eventhough it can affect the quality of the product, this risk is ranked as of medium level.

Prevention plan *ROS* uses a community website[4] to ask and answers questions. An account should be created there. A copy of the book *Multiple View Geometry* book by Hartley and Zisserman [26] should be requested at the library for consulting projective geometry related issues.

[4] `http://answers.ros.org`

Contingency plan Ask the *ROS* community[4]when facing *ROS* related questions. For all other issues, ask one of the advisors or other experts.

R22 Health issues

Description For a long term project like this, it is possible to develop health issues. This can lead to problems meeting deadlines. This risk is ranked low, since the probability of suffering from a health issue for a prolonged period of time is very low.

Prevention plan Non existent, it is out of the projects scope.

Contingency plan After recovering, prioritize the project over other duties.

–R3 *Planning related*

R31 Don't meeting deadlines due to too big workload

Description Due to other duties or commitments it is possible not to meet a deadline. This can happen often, but the impact is low.

Prevention plan Try to do a good planning, estimate every week the number of hours dedicated to the project and try completing them.

Contingency plan After finishing with other duties and commitments, prioritize the project.

R32 Bad project planning

Description Due to the inexperience of the author, the planning could be wrong. The impact of this risk is variable, because it is difficult to estimate the resulting delays.

Prevention plan Expect replanning in the first planning.

Contingency plan Do a replanning of the tasks or the project. Try minimizing problems.

3.4 Work Methodology

Management and Planning

The student will develop the project at Fraunhofer IPA research centre in Stuttgart, Germany. This means that two advisors have been assigned to the

student. On the one hand, Elena Lazkano will be the advisor from the home university and on the other hand Richard Bormann will be the on-site advisor at Fraunhofer IPA.

As no other university courses or lectures have to be attended by the student, he will only focus on developing the project.

Regarding communication, emails and meetings will be used with Richard and emails and Skype with Elena.

Archive Management

The archive will be distributed in different computers. *GIT* will be used as a version control system and to provide backups in case of data loss. During development, Elena Lazkano will not have access to the code. The reason for this is that, in the repository where the code will be developed, private source code from Fraunhofer IPA exists. At the end of the project the code might be open sourced.

The advantage of *GIT* and generally of all distributed version control systems is that the whole archive is available locally. This means that you can experiment with local branches, without needing an internet connection, and merge the new feature easily in the main branch when the code is mature.

The archive has been divided in two repositories. One repository will host all the development code and the other repository will contain the thesis.

3.5 Planning

Before starting a project of this size, it is very important to do a good planning and to monitor it afterwards. First of all, the duration of each task has to be estimated and on the other side a development model for the project has to be chosen. Regarding this last one, an iterative and incremental life-cycle has been chosen. This way, each iteration will extend the project by improving an existent element or adding new functionalities.

On the other hand, the inexperience of the author regarding project planning has to be taken into account, resulting in difficult time estimates for the duration of the tasks. The estimations can be seen at table 3.1.

Time Estimations

TASK	ESTIMATION
Education/Training	
Literature research	40
ScaViSLAM software	40
Tactical processes	
Management	
Meetings	10
Archive management	3
Planning	
PSS	10
Plan new iteration	3
Operative processes	
Development	
Requirements gathering	5
Design	5
Analysis	5
Implementation	262
• Porting of *ScaViSLAM* to ROS	40
• Rewrite of *ScaViSLAM* to support monocular cameras	72
• Implementation of a line tracker in *ScaViSLAM*'s visual frontend	75
• Integration of the line tracker with *ScaViSLAM*'s backend	75
Tests	10
Documentation	
Thesis	50
Finishing *Presentation*	7
Total	450

Table 3.1: Time estimation table

Gantt Chart

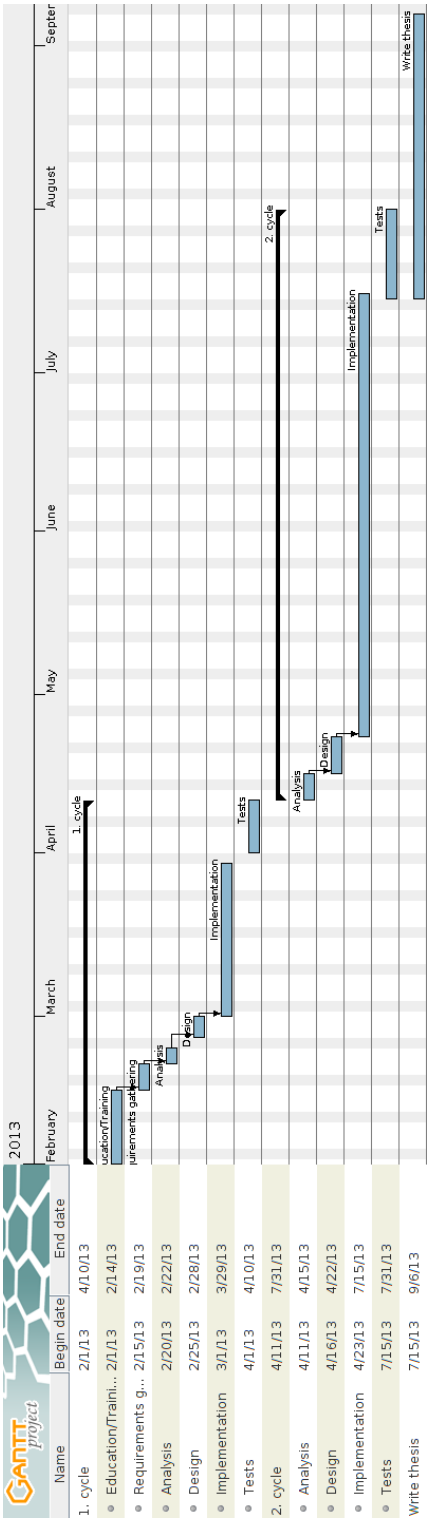


Figure 3.2: Estimated Gantt chart

Chapter 4

Estimations

Contents

4.1 Estimated time and real time

As mentioned before, due to the planning inexperience of the author and the size of the project it is difficult to make a good planning. As a consequence, meeting all the planned milestones and deadlines on time can be difficult. Knowing this, time for replanning has been foreseen from the beginning, in case that the first plan was inadequate.

In this section, the estimated duration of the tasks will be compared with the real duration. Besides, the reasons for the difference will be explained.

In the next figure, the estimated and real hours spent on the tasks of the project are shown.

TASK	**ESTIMATION**	**REAL**
Education/Training		
Literature research	40	40
ScaViSLAM software	40	60
Tactical processes		
Management		
Meetings	10	5
Archive management	3	2
Planning		
PSS	10	10
Plan new iteration	3	7
Operative processes		
Development		
Requirements gathering	5	2
Design	5	2
Analysis	5	2
Implementation	262	320
• Porting of *ScaViSLAM* to ROS	40	40
• Rewrite of *ScaViSLAM* to support monocular cameras	72	100
• Implementation of a line tracker in *ScaViSLAM*'s visual frontend	75	80
• Integration of the line tracker with *ScaViSLAM*'s backend	75	100
Tests	10	10
Documentation		
Thesis	50	50
Finishing *Presentation*	7	7
Total	450	523

Table 4.1: Comparison table between estimated and real duration of tasks

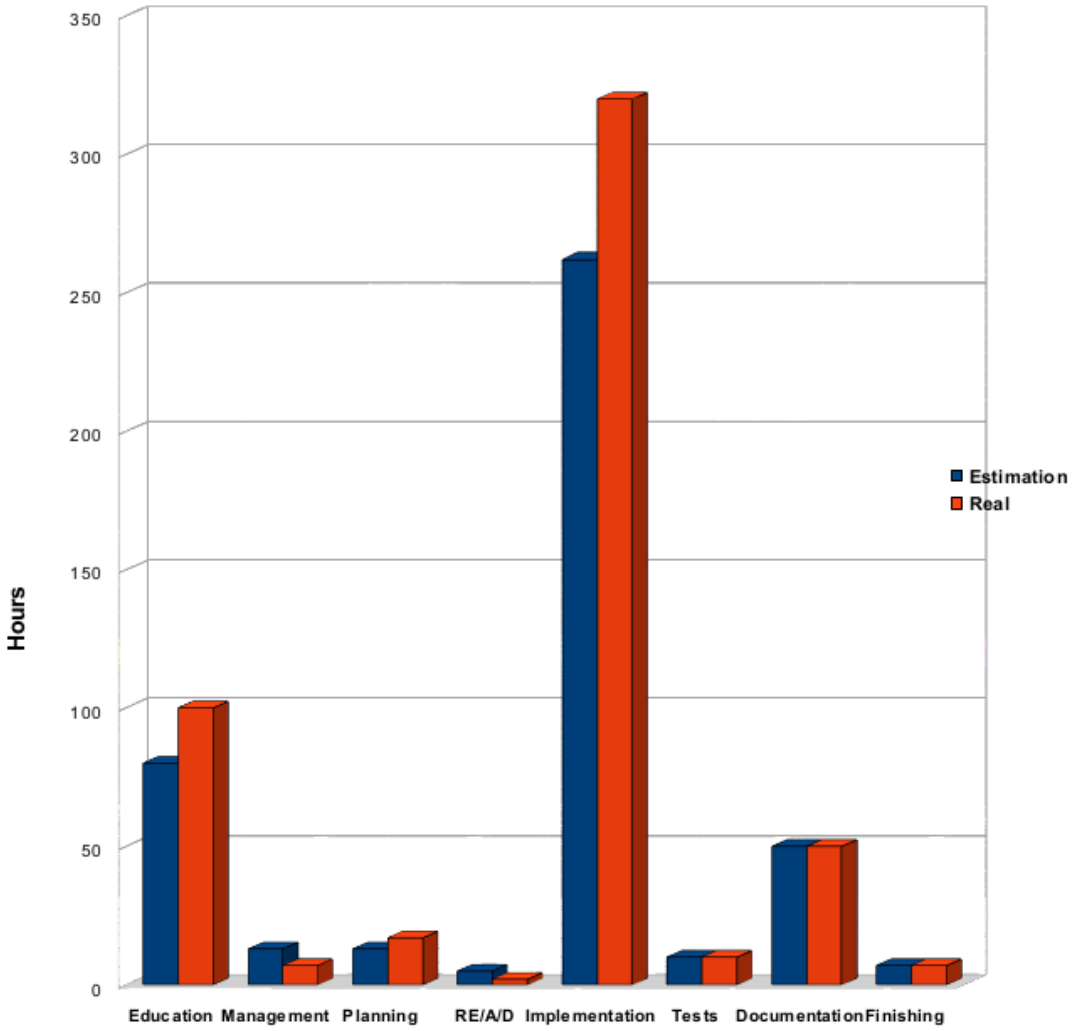


Figure 4.1: Plot with real and estimated task durations. RE stands for requirements gathering, A for Analysis and D for Design.

4.2 Assessment

In the previous chart and tables it seems that the only noticeable deviation of the original time estimation has been the time spent in development. Well, the biggest difference is not reflected in those tables, because it was the deadline for finishing the project. The initial planning was to develop until the end of July at Fraunhofer IPA and then spend August writing the thesis and doing the presentation in September. But at beginning of May, the university communicated that there had been a mistake and that the author had to present the thesis on July 18th. Apparently, only bachelor and master students can present their thesis in September and students from the old degree plan have to finish before August. This unexpected decision had a big impact on the project and forced to rush everything and prioritize the project by working more hours.

So the projects timeline, compared to the initially estimated Gantt chart3.2, has been as follows:

- February: Literature research was done and the porting of *ScaViSLAM* to *ROS* was started.
- March: The port to *ROS* was finalized. The rewrite to support monocular cameras was started.
- April: The realization came that the rewrite of *ScaViSLAM* to support monocular cameras was out of the scope of the project. Therefore, the development of the line tracker was started.
- May: The development of the line tracker was continued and the writing of the thesis was started.
- June: The development of the line tracker was finished. It was realized that there was no time for rewriting *ScaViSLAM* to add line information properly to its backend and that moreover, this was also probably out of the scope. As a contingency plan, it was tried to connect directly with the backend without rewriting *ScaViSLAM*. Besides, the writing of the thesis was continued.
- July: The connection between frontend and backend was finished. The evaluation was started but not finished on time. The thesis was delivered on July 11th and presented on July 18th.

Regardless of the change of deadlines, two of the initially four proposed goals were, in the authors opinion, out of the scope due to the complexity and size of *ScaViSLAM* with over 30000 lines of scarcely documented code. Also there was nobody to ask questions regarding the inner workings of *ScaViSLAM*, which

was the planned contingency plan for this situations, because the advisors or the researchers at Fraunhofer had no experience with the software and the author of *ScaViSLAM*, Hauke Strasdat, was no longer maintaining the project.

It is also worth mentioning that other two identified risks occurred. Concretely, the author was sick during a week in Mai and to cap it all, the Kinect camera broke in Juni, but was replaced by a new bought one, as stated in the contingency plan.

The author wants to thank Richard Bormann for rapidly deciding on buying a new Kinect camera. This was a difficult decision, because the department had already bought 5 new Asus Xtion Pro Live cameras, which unexpectedly didn't work with the OpenNI driver on GNU/Linux, because apparently Asus had made some changes, so that only old models would work with the driver at the time. That is why the alternative would have been to suggest the author to invest time in writing a driver or fixing the OpenNI driver for the Asus cameras. This would have had a critical impact on the project, since there would have been no development for a probably long period of time, when the author was rushing to finish the project because of the aforementioned change in deadlines. Thankfully this was avoided by buying a new Microsoft Kinect camera.

Part III

Development

Table of Contents

Chapter 5

Preliminary Work

Contents

5.1 Integrating ScaViSLAM into ROS

The first task involved porting *ScaViSLAM*[1], the visual SLAM software by Strasdat *et al.* [46] used as a foundation for this project, to the *ROS*[39] platform. The long term goal is to enable testing the implemented improvements in a real mobile robot, like the Care-O-Bot service robot developed by Fraunhofer IPA.

ScaViSLAM is written in C++ and uses CMake[2] to build the software. This makes it theoretically not too difficult to port it to *ROS*, since *ROS* packages also are usually written in C++ and make use of CMake.

However, there were two main obstacles which made the porting cumbersome. On the one hand, *ScaViSLAM* has a lot of external dependencies, which also had to be ported to *ROS* by writing wrappers. This means to *rosify* the external dependencies, i.e. creating *ROS* packages for them. On the other hand, *ScaViSLAM* made use of the *Point Cloud Library*[4] (PCL) OpenNI driver to operate the Kinect Camera, which had to be replaced with the *ROS*

[1] Source code available at `https://github.com/strasdat/ScaViSLAM`

[2] CMake is a popular cross-platform, open source build system. See `http://www.cmake.org` for more details.

OpenNI driver. The reason for this was, that the *PCL ROS* package ships without this driver, because the OpenNI driver is available as a separate *ROS* package.

Regarding the *ROS* release, the *Fuerte* version was used. Therefore, *rosbuild* was used instead of the new *catkin* build system introduced in the *Groovy ROS* version. In this first step, *ROS* wrappers were written, following this[3] tutorial, for Pangolin[5], VisionTools[8], Sophus[6] and g^{2}o [9]. The other dependencies could be solved by either using already existing *ROS* packages or installing them as system packages and using them via the *rosdep* tool. The first method was used to include OpenCV[11], a popular computer vision library. The last method on the other hand, was used for the Eigen library, Suitesparse, Boost and OpenGL. Eigen and Suitesparse, which are a linear algebra and sparse matrices computation library respectively, are heavily used.

5.2 Rewriting ScaViSLAM to support Monocular Cameras

After finishing the port of *ScaViSLAM* to *ROS* and modifying the code to replace the *PCL* OpenNI with the *ROS* OpenNI driver, a monocular rewrite of *ScaViSLAM* was explored. The motivation for this was, that monocular cameras are the most affordable of all camera setups. Therefore, a monocular visual SLAM system would help driving the manufacturing costs of mobile service robots even more down.

As mentioned before, Strasdat *et al.* claim to have implemented a modified version of *PTAM* as the frontend of *ScaViSLAM* for monocular systems. Apparently, distributing this implementation is not possible because of *PTAM*'s licence. There seems to be some confusion with this licence, because modified versions of PTAM that are freely available as open source exist. For example, ETHZ's version[4] for unmanned micro aerial vehicles [55].

As it was not possible to obtain the monocular version of *ScaViSLAM* from Strasdat *et al.*, the author duplicated efforts by trying to reimplement the same functionality. The approach followed was the same as described by Strasdat *et al.* First to try using *PTAM* as the visual frontend and then to rewrite the backend to make use of the *Sim(3)* group instead of *SE(3) lie groups*.

In order to integrate *PTAM*, two approaches were tested. First, integrating

[3] `http://www.ros.org/wiki/ROS/Tutorials/Wrapping%20External%20Libraries`

[4] Source code available at `https://github.com/ethz-asl/ethzasl_ptam`

it directly in the same *ROS* package and writing *ROS* wrappers for its dependencies (*libCVD*[2], *TooN*[7], *GVars*[10]). The second approach was to use ETHZ's modified *PTAM* version, which is already available as a *ROS* package.

Getting *PTAM* to run inside *ScaViSLAM* was achieved with both methods, but there was no time for rewriting *ScaViSLAM* to make use of *PTAM*. The sheer size of *ScaViSLAM*(>30000 lines of code) and its complexity, would suggest a much longer time frame than a month to implement such a big rewrite. After realizing that this task was delaying the start of the actual development part of the thesis and that it was not going to be finished on time, it was decided to stop with it. Besides, it was agreed upon to only continue with the monocular support if after finishing all other tasks there was still time left.

Chapter 6

Visual Frontend: Line Tracking Algorithm

Contents

6.1 Detection of Lines

The first step to write a Line tracker is to detect line segments in an image. For this purpose, the Probabilistic Hough Transform [29] is applied. In particular, the implementation provided by the popular OpenCV[11] library was used. But in order to use the Probabilistic Hough Transform and get good results, some preprocessing steps are necessary. The needed steps and their relationship are shown in figure 6.1.

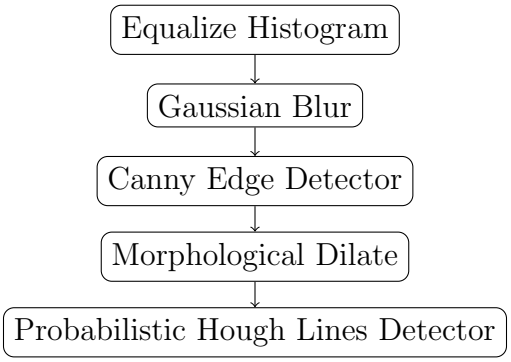
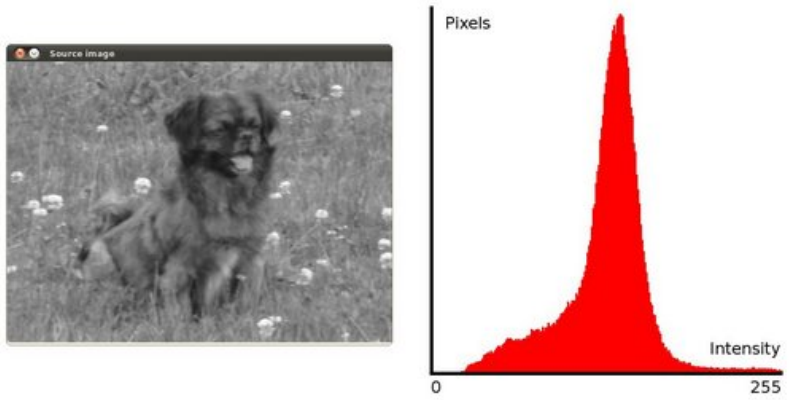


Figure 6.1: The preprocessing necessary for detecting lines in a image.

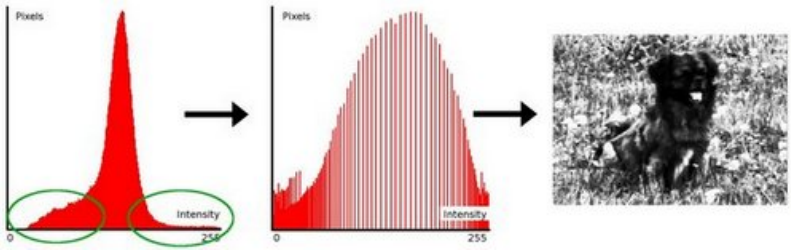
The individual preprocessing steps will now be briefly introduced.

6.1.1 Histogram Equalization

First of all, a *histogram equalization* is performed. A histogram is a graphical representation of the intensity distribution of an image. Thus, it quantifies the number of pixels for each intensity value of the image. Equalizing the histogram improves the contrast in an image, in order to stretch out the intensity range. Equalization implies mapping one distribution (the given histogram) to another distribution (a wider and more uniform distribution of intensity values), so that the intensity values are spreaded over the whole range. An example of the process is shown in Figure 6.2.



(a) An image and its histogram



(b) Histogram after equalization and the resulting image

Figure 6.2: Process of histogram equalization. Taken from the official OpenCV documentation.

6.1.2 Gaussian Blurring

Next, a Gaussian blurring or smoothing is performed in order to reduce noise. For that, a *filter* is applied, which can be visualized as a window/matrix sliding through the image. Gaussian filtering is done by convolving each point in the input array with a Gaussian kernel/function and then summing them all to

produce the output array. Applying a Gaussian blur has the effect of reducing the image's high-frequency components. Thus, a Gaussian blur is a low pass filter. The equation of a Gaussian function in two dimension can be seen in Eq. 6.1.

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{6.1}$$

A graphical illustration of a Gaussian blurring is shown in 6.3.

Figure 6.3: The effects of a small and a large Gaussian blur. Image by Lieu Song, distributed under public domain on Wikipedia.

Gaussian smoothing is commonly used as a preprocessing step before applying the Canny Edge Detector.

6.1.3 Canny Edge Detector

The popular Canny Edge Detector was developed by John F. Canny in 1986 [15]. Canny's aim was to discover the optimal edge detection algorithm. Therefore the algorithm aims to satisfy three main criteria:

- Low error rate: the algorithm should mark as many real edges in the image as possible.

- Minimal response: a given edge in the image should only be marked once, and where possible, image noise should not create false edges.
- Good localization: The distance between edge pixels detected and real edge pixels have to be minimized.

To satisfy these requirements Canny used the calculus of variations – a technique which finds the function which optimizes a given functional. The optimal function in Canny's detector is described by the sum of four exponential terms, but it can be approximated by the first derivative of a Gaussian. The main stages in the algorithm are:

- Noise reduction by using a Gaussian filter.
- Finding the intensity gradient of the image with the Sobel operator.
- Non-maximum suppression: removes pixels that are not considered to be part of an edge.
- Hysteresis thresholding to determine if a pixel is accepted as an edge.

In Figure 6.4 an example of the Canny Edge Detector is shown.

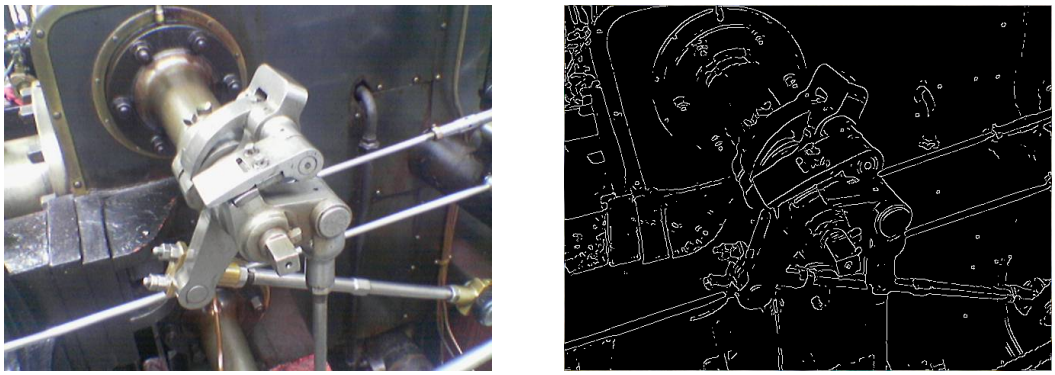


Figure 6.4: Application of the Canny Edge Detector. Images distributed under public domain on Wikipedia.

6.1.4 Morphological Dilation

The morphological dilation operator refers to convoluting an image with a structuring element, a rectangle in this case, for probing and expanding the shapes contained in the input image. The aim of this preprocessing step is to expand objects in order to fill small holes and connect disjoint objects. The operator works by computing the maximal pixel value overlapped by the

structuring element and replacing the image pixel in the center structuring element point position with that maximal value.

$$(A \oplus X)(x, y) = \max\{A(x + s, y + t) + X(s, t) | (s, t) \epsilon D_X\} \tag{6.2}$$

An example of the dilation operator can is shown in figure 6.5.

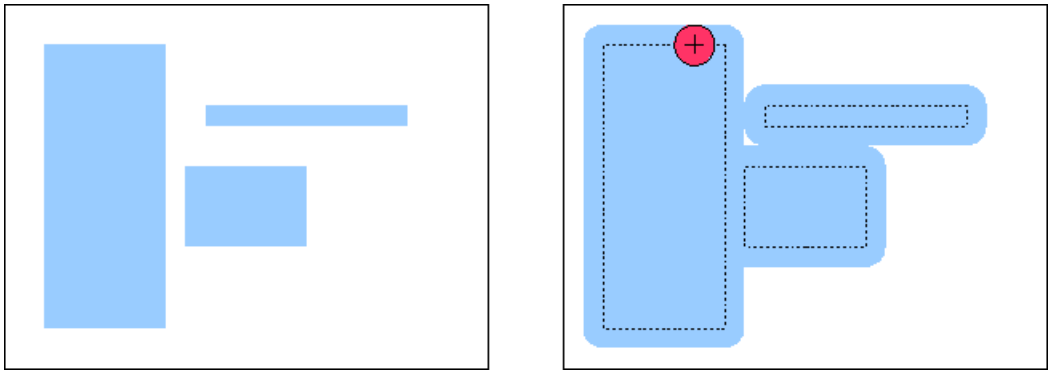


Figure 6.5: Applying dilation with a circle as a structuring element. Image by Martin Pfeiffer, distributed under public domain on Wikipedia.

6.1.5 Probabilistic Hough Transform

The Hough Transform is a method for estimating the parameters of a geometrical shape from its boundary points in images. The algorithm can be applied to estimate parameters of arbitray shapes. Due to imperfections in either the image data or the edge detector, there may be missing points as well as spatial deviations between the ideal line/circle/ellipse and the noisy edge points as they are obtained from the edge detector. For these reasons, it is often non-trivial to group the extracted edge features to an appropriate set of lines, circles or ellipses. As a consequence, the Hough Transform algorithm is applied as the final stage to detect lines in an image.

The algorithm will now be explained briefly for the case of lines, based on the official OpenCV documentation.

Straight lines are often described with the slope-intercept $y = mx + b$ formula. Vertical lines are problematic with this representation because the slope rises to unbound values in the parameter space. For example, for $x = 0$, m would have to be infinite. Therefore polar coordinates are preferred. Figure 6.6 shows the relations among the coordinates.

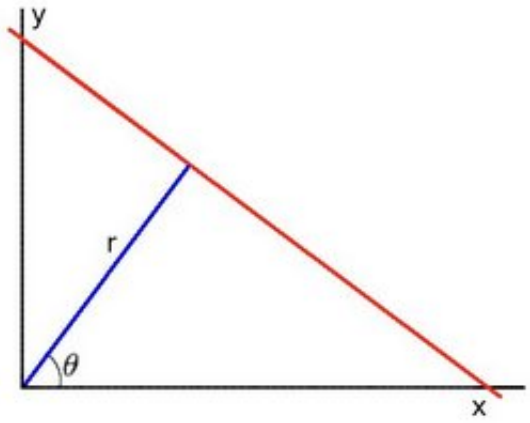


Figure 6.6: Polar coordinates: r represents the distance between the line and the origin, and θ is the angle of the vector from the origin to this closest point. Image distributed under public domain on Wikipedia.

With the polar coordinates, a straight line equation can be written as:

$$y = \left(-\frac{\cos\theta}{\sin\theta}\right) x + \left(\frac{r}{\sin\theta}\right) \tag{6.3}$$

And rearrange it as $r = x\cos\theta + y\sin\theta$. So for each point (x_0, y_0), it is possible to define the family of lines that goes through that point as:

$$r_\theta = x_0 \cdot \cos\theta + y_0 \cdot \sin\theta \tag{6.4}$$

Meaning that each pair (r_θ, θ) represents each line that passes by (x_0, y_0). If for a given point (x_0, y_0) we plot the family of lines that goes through it, we get a sinusoid. Besides, the points must satisfy $r > 0$ and $0 < \theta < 2\pi$.

If the curves corresponding to two points are superimposed, the location in the Hough space where they intersect corresponds to a line in the original image space that passes through both points. The more curves intersecting means that the line represented by that intersection have more points. An example is shown in figure 6.7.

The classic Hough Transform algorithm for lines keeps track of the intersection between curves of every point in the image. If the number of intersections is above some threshold, then it declares it as a line with the parameters (θ, r_θ) of the intersection point. On the other hand, the probabilistic variant of the Hough Transform algorithm returns the start and ending points of the line (x_0, y_0, x_1, y_1) and besides, it is more efficient.

In this thesis, only lines that are at least 80 pixel long and have no gaps between them are chosen. The reasoning behind it is, that long lines are more

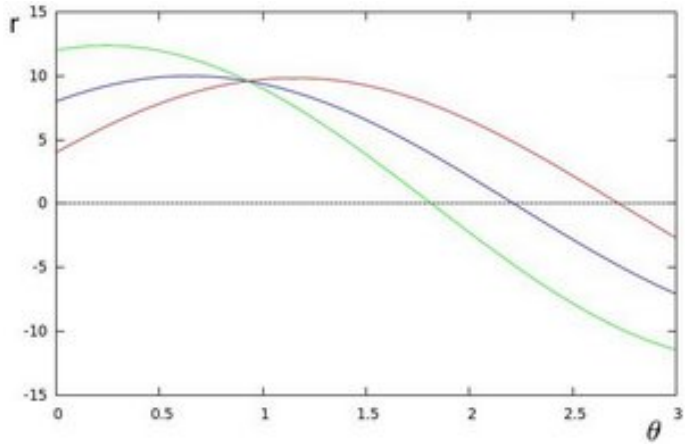


Figure 6.7: Intersection of different curves, corresponding to a line, in the Hough Parameter Space

robust against occlusions and are therefore more probable to be found in the next frames, after the camera has moved.

6.1.6 Evaluation and Future Work

It was found, unsurprisingly, that fitting straight lines to edges with the Canny edge detector and using the Hough transform are too slow for real-time operation. Therefore, implementing a more efficient approach like the one proposed by Davison *et al.* [45] or the Line Segment Detector by von Gioi *et al.* [25]remains open for future work. Besides, in the brief evaluation made of [25], no way was found to specify computing only lines that are at least 80 pixel long and have no gaps between them.

6.2 Plücker Parameters with Linear Regression

In section 2.5.1, a way of computing the Plücker parameters given two points were introduced. Due to the inherent noise, it is possible that the computed parameters aren't %100 accurate. Besides, the computation can fail, because the Kinect camera doesn't have depth information for the starting or ending points of the line segment. Therefore, a second way of computing the Plücker parameters was tested. By using multiple points on the line together with linear regression and Singular Value Decomposition (SVD), it is possible to achieve a higher accuracy for the computed Plücker parameters.

The SVD matrix factorization is given by:

$$A = U\Sigma V^T \tag{6.5}$$

where U is a m×m real unitary matrix, Σ is a diagonal matrix with nonnegative real numbers and V^T is a n×n real unitary matrix (the conjugate transpose of V).

We can use the SVD to solve a system of linear equations $Ax = b$. This is equivalent with minimizing the squared norm $||Ax - b||^2$, which is a linear least-squares optimization problem. In our case, $b = 0$, thus our minimization problem is $||Ax||$. Regarding the data matrix A, $m > n$ so we have an overdetermined system. As we will see later, for each pixel on a line we will have 4 rows in the data matrix. In general the system $Ax = 0$ has no exact solution, so we are interested in an approximation. Besides, we have to impose a constraint on x to avoid the trivial solution of $x = 0$. Therefore we want to find an x that minimizes $||Ax||$ subject to $||x|| = 1$.

So far, we have:

$$\begin{aligned} ||Ax|| &= \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{bmatrix} \rightarrow 0 \\ &= \begin{bmatrix} r_1 & r_2 & \dots & r_n \end{bmatrix} \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{bmatrix} = r_1^2 + r_2^2 + \ldots + r_n^2 \rightarrow 0 \\ \leftrightarrow & r^T r = (x^T A^T) Ax \end{aligned} \tag{6.6}$$

Therefore our minimization problem boils down to:

$$\begin{aligned} &||Ax|| \rightarrow 0 \\ &\min_x x^T A^T Ax \end{aligned} \tag{6.7}$$

Besides, U and V^T are orthogonal matrices. This is interesting because, an orthogonal matrix M has a norm-preserving property, i.e. for any vector v:

$$||Mv|| = ||v|| \tag{6.8}$$

Another property of orthogonal matrices, which we will make use of, is $QQ^T = Q^T Q = I$, where I is a identity matrix.

For this we derive:

$$\begin{aligned} x^T A^T A x &= x^T V \Sigma^T \underbrace{U^T U}_{=I} \Sigma V^T x \\ &= x^T V \Sigma^T \Sigma V^T x \\ &= y^T \Sigma^T \Sigma y, \qquad \text{where } y = V^T x \end{aligned} \tag{6.9}$$

As V^T is a rotation matrix and therefore and orthogonal matrix, by using the norm-preserving property Eq. 6.8 we have $|y| = 1$. Putting the equations together, we have:

$$y^T \Sigma^T \Sigma y \to 0 \leftrightarrow y = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \to Vy = x, \quad x = V \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \tag{6.10}$$

The diagonal n×n matrix $\Sigma^T \Sigma$ is formed by $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n$, which are the *singular* values of A. Note that they are the square roots of the eigenvalues of $A^T A$ and AA^T.

Now to the application to Plücker lines. A point x lies on the line only if $L^* x = 0$, where L^* is the dual Plücker matrix. Therefore we have:

$$\begin{aligned} &\begin{bmatrix} 0 & l_{34} & l_{42} & l_{23} \\ l_{34} & 0 & l_{14} & -l_{13} \\ -l_{42} & -l_{14} & 0 & l_{12} \\ -l_{23} & l_{13} & -l_{12} & 0 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \Rightarrow \\ &\begin{bmatrix} 0 & l_{34} & l_{42} & l_{23} \\ l_{34} & 0 & l_{14} & -l_{13} \\ -l_{42} & -l_{14} & 0 & l_{12} \\ -l_{23} & l_{13} & -l_{12} & 0 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} l_{34}Y + l_{42}Z + l_{23} \\ -l_{34}X + l_{14}Z - l_{13} \\ -l_{42}X - l_{14}Y + l_{12} \\ -l_{23}X + l_{13}Y - l_{12}Z \end{bmatrix} = \\ &= \begin{bmatrix} 0 & 0 & 0 & 1 & Z & Y \\ 0 & -1 & Z & 0 & 0 & -X \\ 1 & 0 & -Y & 0 & -X & 0 \\ -Z & Y & 0 & -X & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} l_{12} \\ l_{13} \\ l_{14} \\ l_{23} \\ l_{24} \\ l_{34} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \end{aligned} \tag{6.11}$$

After computing the SVD with *OpenCV*, the result lies in the sixth row of the V^T matrix.

6.2.1 Evaluation

After implementing this method, the approach showed to be too slow to compute. At first, *all* the points on the line were used for the SVD computation. Being the standard length of a line 100 pixel and obtaining 10-15 lines per frame, the computation of the Plücker parameters took 0.25 seconds per line. To put it in a nutshell, it took almost 4 seconds for each frame. In the following output, the computation of the Plücker coordinates for all lines chosen in one frame is shown. As a comparison, the Plücker parameters are computed with the two known approaches. On the one hand, the method of using only the start and endpoints of the line and on the other hand, taking all the points in the line. Besides, the time needed for each method is shown.

```
SVD performed, plücker Line: [-0.041354734, 0.94410014,
0.18098645, -0.23605314, 0.03955368, -0.13008896]
0.277316 s computing SVD
plucker param: 0.0415218 -0.942657 -0.177178 0.235664
 -0.0378645 0.14598
euclidean distance: 0.016
-------------------------------------------------------
SVD performed, plücker Line: [-0.039483715, 0.94527143,
0.18265331, -0.23557271, 0.040506002, -0.12002322]
0.749766 s computing SVD
-------------------------------------------------------
SVD performed, plücker Line: [-0.057846352, 0.93908143,
0.18991378, -0.24893782, 0.042822231, -0.12210207]
0.365418 s computing SVD
plucker param: 0.0578235 -0.938557 -0.190447 0.24886
 -0.0427687 0.125446
euclidean distance: 0.003
-------------------------------------------------------
SVD performed, plücker Line: [-0.055870593, 0.93761182,
 0.18792956, -0.24962136, 0.041955445, -0.13555101]
0.525342 s computing SVD
-------------------------------------------------------
SVD performed, plücker Line: [0.3318308, 0.92997813,
 0.14072883, -0.018807858, 0.025961243, -0.064781748]
0.136344 s computing SVD
-------------------------------------------------------
SVD performed, plücker Line: [0.32733569, 0.93144321,
0.14176606, 4.1757815e-08, 0.023832871, -0.067817144]
0.125375 s computing SVD
-------------------------------------------------------
```

```
SVD performed, plücker Line: [0.14009772, 0.97551262,
 0.10771098, -0.10750199, 0.022126563, -0.071418658]
0.098531 s computing SVD
-----------------------------------------------------
SVD performed, plücker Line: [0.10433333, 0.9711296,
 0.11351357, -0.15389577, 0.027991679, -0.093108244]
0.307274 s computing SVD
plucker param: -0.104225 -0.970539 -0.113551
0.153669 -0.0286435 0.0993076
euclidean distance: 0.006
-----------------------------------------------------
SVD performed, plücker Line: [0.10154942, 0.97180349,
 0.11528987, -0.1519475, 0.027461063, -0.09028852]
0.301903 s computing SVD
plucker param: -0.101486 -0.971165 -0.114242
 0.151869 -0.0281211 0.0981448
 euclidean distance: 0.007
-----------------------------------------------------
SVD performed, plücker Line: [-0.054181233, 0.93901604,
 0.19382325, -0.24748452, 0.044123441, -0.1206259]
0.188463 s computing SVD
-----------------------------------------------------
SVD performed, plücker Line: [-0.059585184, 0.94366312,
 0.18448827, -0.24960646, 0.043243423, -0.087978922]
0.15013 s computing SVD
plucker param: 0.0596027 -0.944087 -0.184155
 0.249676 -0.0434154 0.0837385
euclidean distance: 0.171
-----------------------------------------------------
SVD performed, plücker Line: [-0.43723068, 0.69431025,
 0.35207877, -0.38754061, 0.056524076, -0.22230734]
0.089843 s computing SVD
-----------------------------------------------------
SVD performed, plücker Line: [-0.041632656, 0.94932771,
0.17757563, -0.23752837, 0.040648367, -0.086245239]
0.121903 s computing SVD
plucker param: 0.0419276 -0.946112 -0.182988
 0.238659 -0.0415223 0.104633
euclidean distance: 0.019
3.99108 s for computeLines
```

Since processing one frame using only the two point Plücker method takes 0.07s, the SVD method would need to be 100 times faster to achieve the

same speed. On the other hand, the mean euclidean distance between both approaches was only of 0.037. This means, that the result achieved by the two point algorithm is already pretty accurate. The only downside is, that it is not always possible to compute, because the sensor may not have 3D information for one of the starting or ending points.

Reducing the number of pixels for the SVD method was also tried. For example, by considering only every fifth pixel, 0.514 seconds were needed for one frame, and when considering only every fifteenth pixel, 0.225 seconds. As this was still too much time, sticking with the two point algorithm was decided, but with some improvements.

As it was already mentioned in the preliminaries, one nice property of Plücker coordinates is, that the matrix L is independent of the points A, B used to define it. The reason for this is, that even if a different point C on the line is used we obtain the same matrix L. As the SVD is computationally too expensive, but the two point method often fails because of the aforementioned reasons, an approach that combines the best of both methods has been developed by exploiting Plücker's property.

The approach works as follows: if the Kinect doesn't return 3D data for the starting point, the coordinates that lie on the line with a distance of k px (in our case $k = 5$) are computed. For this, the normalized direction vector is used, whose computation is detailed in next section's Eq. 6.14. If sensing the 3D data for this coordinates still fails, the same procedure is repeated until a maximum of λk px distance, where $\lambda = 4$ tries (i.e. a distance of 20 px). For the ending point, the procedure is the same, with the only difference that the coordinates are decremented, i.e. $(x_b, y_b) = (x_b - r_x * 5, y_b - r_y * 5)$. This way, we still have an efficient method for computing Plücker coordinates and can effectively damp the effect of the sensor's noise.

6.3 Development of a new Line Descriptor

A simple line descriptor, that uses local neighbourhood information, has been developed with the objective of being efficient to compute and later on, to match two lines. The motivation for developing a new line descriptor was, that the line descriptors found in the literature are usually computationally too expensive to be used in a real time SLAM system. This is true for the popular mean–standard deviation line descriptor (MSLD) by Wang *et al.* [54], which uses a SIFT-like strategy. For example, Zang and Koch presented a line-based SLAM that uses EKF for camera pose estimation and a line-based recovery method [57]. In their approach MSLD is used for matching line segments between the current frame and stored key-frames in order to relocalize the system. As MSLD is quite computationally expensive, they used it only for recovering.

Other line descriptors, such as the recently presented ones by Fan *et al.* [23] and Zhang *et al.* [58](0.2-0.5s for each descriptor), were also discarded because of their computational cost. Therefore a new line descriptor has been developed to meet our requirements. The main idea of the descriptor is to compare intensity values in the neighbourhood of the line L. The images are expected to be captured with a high frequency, therefore the displacement of the lines between frames should not be too big. It follows, that we don't need a very sophisticated line descriptor. Moreover, as the line already has a clear direction, a rotation invariant descriptor like ORB [43] is not needed. The first approach for the new line descriptor will now be introduced.

As the probabilistic Hough Transform gives us starting and ending point of a line, the pixels on the line are computed using the classic Bresenham algorithm [14] introduced in 1965. The algorithm is very popular, because it only uses integer addition, subtraction and bit shifting, all of which are cheap operations for a computer, making the algorithm very efficient. The next steps are:

1. Compute parallel lines on top (line a) and on the bottom (line b) of the original line L with a distance of $k \cdot px$ (where $k = 5$) by using the normal of the line.
   (a) First compute the direction vector of line L
   $$\vec{r} = \vec{Q} - \vec{P} = \begin{pmatrix} x_q - x_p \\ y_q - y_p \end{pmatrix} \tag{6.12}$$
   (b) normalize it
   $$\hat{r} = \frac{\vec{r}}{\|\vec{r}\|} \tag{6.13}$$

(c) and compute the normal vector

$$\hat{n} = \begin{pmatrix} r_2 \\ -r_1 \end{pmatrix} \tag{6.14}$$

(d) now we can calculate the two parallel lines

$$(x_a, y_a) = (x_l + n_x \cdot k, y_l + n_y \cdot k) \tag{6.15}$$

$$(x_b, y_b) = (x_l - n_x \cdot k, y_l - n_y \cdot k) \tag{6.16}$$

2. Iterate through these lines by comparing the intensity values $a(w) > b(w)$ and setting a 1 or 0 in the descriptor accordingly.
3. If there are more cases where $b(w) > a(w)$, then change the direction of w, in order to identify a line even if it is rotated 180 degrees. Thus, the brighter side will be always on top.

This steps are illustrated in figure 6.8.

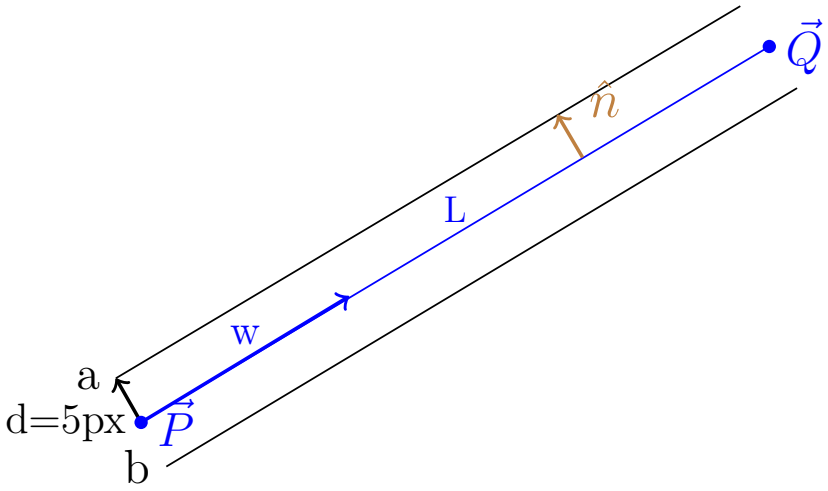


Figure 6.8: Illustration of the developed line descriptor.

The resulting line descriptor can be represented as:

$$[a(w=0) > b(w=0), a(w=1) > b(w=1) \ldots a(w=n) > b(w=n)] \tag{6.17}$$

6.3.1 Mean Intensity of Neighborhood

The first approach, suffers from the problem that the descriptor is not distinct enough to identify a line properly. The reason for this is, that it often contained

over %85 of one component (1 or 0), resulting this in false positives in the matching algorithm presented in the next section.

To fix this, the following improvements were made. Instead of only comparing the number of values where $a(w) > b(w)$, we take the mean intensity of the neighborhood around a(w) and compare it with the mean intensity of the neighborhood around b(w), as can be seen in figure 6.11. The neighborhood's size is variable, but it must be an odd numbered squared matrix in order for a(w) or b(w) to be in the center.

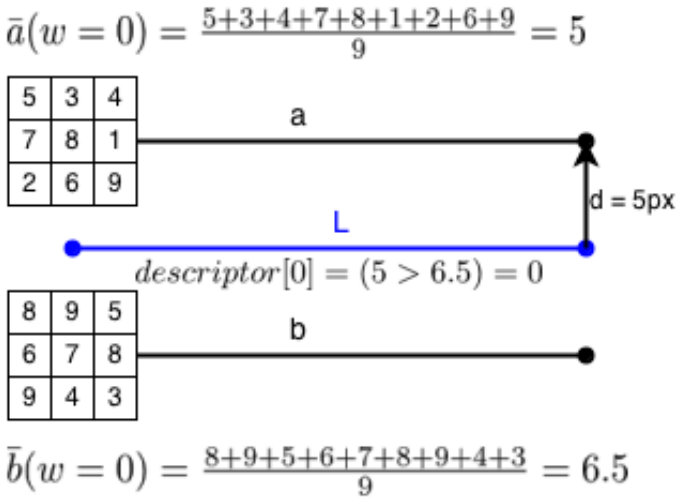


Figure 6.9: Mean intensity line descriptor using a 3x3 neighborhood.

The mean intensity line descriptor can be described as:

$$[\bar{a}(w=0) > \bar{b}(w=0), \bar{a}(w=1) > \bar{b}(w=1) \ldots \bar{a}(w=n) > \bar{b}(w=n)] \quad (6.18)$$

One drawback of this approach is that a section of the neighborhood may lie outside the image. In this case, we interpolate the values, using the last known ones.

To compute the mean intensity of the neighborhood, a summed area table, also known as integral image, is used. With this trick, the task of calculating the sum of pixels in some rectangle which is a subset of the original image can be done in constant time, i.e. $\mathcal{O}(1)$ complexity. This concept was first introduced to computer graphics in 1984 by Frank Crow [18]. In Computer Vision it was first used within the Viola–Jones object detection framework [52].

As the name suggests, the value at any point (x, y) in the summed area table is just the sum of all the pixels above and to the left of (x, y), inclusive.

Furthermore, the summed area table can be computed efficiently in a single pass over the image, using the fact that the value in the summed area table at (x, y) is just:

$$I(x,y) = i(x,y) + I(x-1,y) + I(x,y-1) - I(x-1,y-1) \tag{6.19}$$

Once the summed area table has been computed, the task of evaluating any rectangle can be accomplished in constant time with just four array references. Specifically, using the notation in the figure below, the value is just:

$$\sum_{\substack{A(x)<x'\leq C(x) \\ A(y)<y'\leq C(y)}} i(x',y') = I(C) + I(A) - I(B) - I(D) \tag{6.20}$$

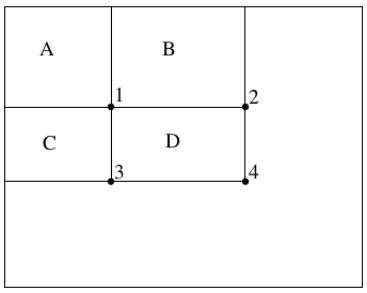


Figure 6.10: The sum of the pixels within rectangle D can be computed with four array references. The value of the integral image at location 1 is the sum of the pixels in rectangle A. The value at location 2 is $A + B$, at location 3 is $A + C$, and at location 4 is $A + B + C + D$. The sum within D can be computed as $4 + 1 - (2 + 3)$. Taken from [52].

Luckily, *OpenCV* comes with a predefined function to calculate an integral image.

This second approach works better than the first one, but the resulting descriptor is still not distinct enough. Therefore, a Sum of Squared Differences (SSD) version was implemented.

6.3.2 Sum of Squared Differences

Sum of Squared Differences (SSD) is a well known similarity measure, which is described as $\sum_{i,j \in W} (I_1(i,j) - I_2(x+i, y+j))^2$. Even though it is computationally more expensive than the previous approaches, it should also perform better as a line descriptor. To implement this, two fixed sized 5x3 neighborhood matrices are used, one above the line and the other below it. The

component a_{52} of the top matrix, is the pixel on top of the line pixel l_i and the component b_{12} of the down matrix is the pixel under l_i. The corresponding pixels of the neighborhood are calculated using the normal vector as seen in Eq. 6.14. The sum for each matrix is computed using integral images and the difference between them is saved in the descriptor.

The SSD line descriptor can be represented as:

$$\Big[\sum a(w=0) - \sum b(w=0), \sum a(w=1) - \sum b(w=1) \\ \ldots \sum a(w=n) - \sum b(w=n)\Big] \tag{6.21}$$

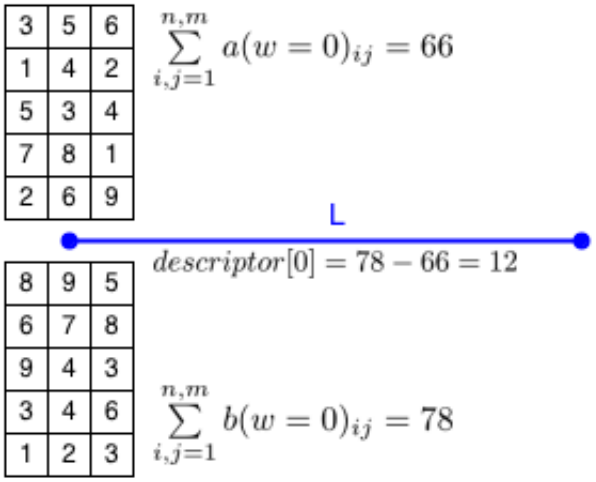


Figure 6.11: Sum of Squared Differences line descriptor using a 5x3 neighborhood.

Again, a section of the neighborhood may lie outside the image. In this case, we interpolate the values using the last known ones.

It appears that this line descriptor works better than the previous approaches, but a quantitative evaluation is necessary to confirm this. Besides, a slight modification must be done to the matching algorithm that follows.

If the evaluation shows that the line descriptor is still not good enough, some of the following ideas could be tested:

- Zero-mean Sum of Squared Differences (ZSSD)
  $\sum_{i,j \in W} ((I_1(i,j) - \overline{I_1}) - (I_2(x+i, y+j) - \overline{I_2}))^2$ this is computationally more expensive than SSD but gives correct results even if there is a constant offset between the pixel intensities.

- Transforming one of the image blocks to the other block's coordinate system, i.e. perform a perspective warp.
- Trying to emulate the FAST descriptor [42] for lines.

6.4 Matching of non-SSD Line Descriptors

Given two line descriptors l_1 and l_2, we seek to determine if they correspond to the same line in the image. For this, we first check that the size difference between both descriptors is not too big. Next, the minimal overlapping area between both descriptors has to be computed. Afterwards, the Hamming distance is calculated for all possible overlappings, which works as a sliding window. If the minimum error divided by the number of overlapping pixels is smaller than a threshold, a match is found. The algorithm can be understood much easier with the figure shown in 6.12, but is also presented in pseudocode form in Algorithm 1.

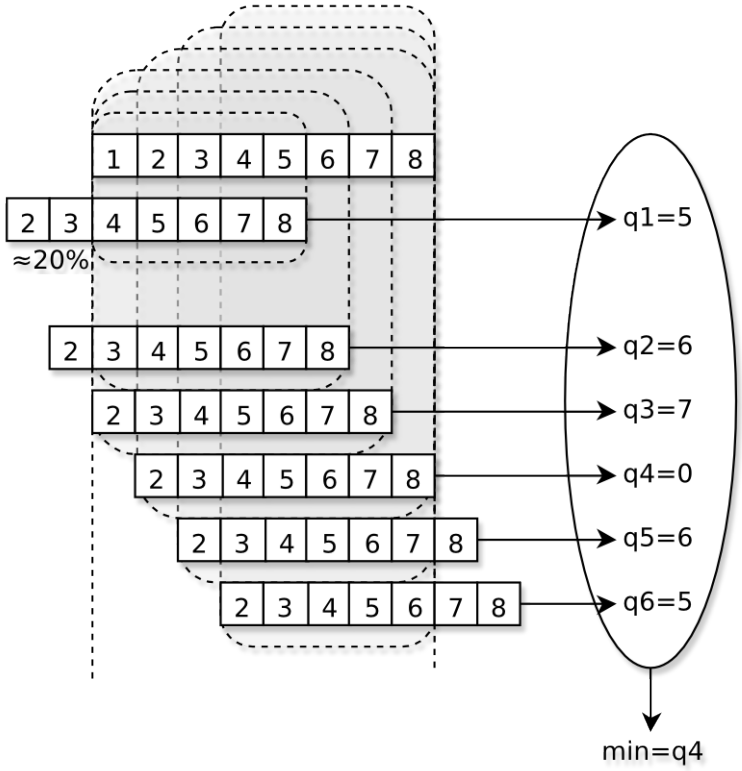


Figure 6.12: Sliding window line matching algorithm. For the sake of illustration, the arrays hold more numbers than just 0 and 1, which feature in the non-SSD versions of the line descriptor.

Algorithm 1: The line descriptor matching algorithm

Input: Line descriptors l_1, l_1
Output: True if there is a match, false if there is no match

```
if ((l1.size/l2.size) > 2.0) then
    return False
end
winSize ← ⌊(min(l1.size, l2.size) · 0.8)⌋
j ← (min(l1.size, l2.size) − winSize)
twentyPercent, jOld ← j; i, iNew, errorCount ← 0
maxBound ← (max(l1.size, l2.size) + twentyPercent)
initialize vector hammingErrors
while (i < winSize) & (winSize! = maxBound) & (twentyPercent! = −1)
do
    if l1[i]! = l2[j] then
        ++errorCount
    end
    if i == (winSize − 1) then
        if jOld! = 0 then
            j ← (jOld − 1); i ← 0; jOld ← j; ++winSize
        end
        else
            i ← (iNew + 1); ++iNew; j ← 0
            if winSize! = max(l1.size, l2.size) then
                ++winSize
            end
            else
                −−twentyPercent
            end
        end
        hammingErrors.push_back(errorCount)
        errorCount ← 0
    end
    else
        ++i; ++j
    end
end
minError ←findMinimumElement(hammingErrors)
normalizedMinError ← (minError/hammingErrors.size)
if normalizedMinError < 0.25 then
    return True
end
else
    return False
end
```

6.4.1 Matching of SSD Line Descriptors

The SSD line descriptor doesn't feature just 0s and 1s, but the difference of the neighborhoods sum. Therefore, a slight modification must be made to the aforementioned matching algorithm. Instead of incrementing the error if $l_1[i]! = l_2[j]$, we sum the squared error $(l_1[i] - l_2[j])^2$, completing the Sum of Squared Differences. The rest of the matching algorithm remains the same.

6.5 Guided Search for Matching

So far, a simple line descriptor and a matching algorithm have been presented. The next step is to maintain a set of tracked lines. For this, the coarse workflow is:

1. $frame = 1$: add all the found lines that are at least 80 pixels long and have no gaps to the tracked list. Each line has a counter with the value 3.
2. The robot/camera has moved.
3. $frame = 2$: try finding all the tracked lines in the current image. Decrement the counter of the lines that couldn't been found again.
4. $frame = 3, 4$: repeat steps from frame 2.
5. $frame = 5$: deleted lines from the tracked list if they haven't been found again three times in a row, i.e. their counter equals zero.
6. When adding a new keyframe, add new lines that weren't in the tracked list.

The counter is necessary, because the Hough Transform does not always detect the same amount of lines, even without camera motion. The reason for this can be different lighting conditions or perceived intensity values or camera noise. Moreover, the Canny Edge detector can fail to detect the same edges. To put in a nutshell, illumination changes, camera noise and failures in the preprocessing can lead to not detecting the same lines again, despite not moving the camera.

So, if in the first frame 15 lines are detected and in the second frame only 10, it is not suitable to discard 5 tracked lines directly because Hough failed to detect them. Besides, even if Hough detected the same line again, it is possible that the Kinect failed to return depth information for the line's points. In this case, the line is discarded when the computation of Plücker coordinates is not possible. By establishing the policy of only deleting lines if they weren't

matched in three consecutive frames, we avoid deleting lines that are still in the image.

Finally, the reason for adding only new lines when adding a new keyframe is onefold. ScaViSLAM computes the transformation of the current frame to the last keyframe. Therefore, if we would allow adding new lines in any frame, saving an anchorframe for each line would be necessary to transform and project all the lines to the same coordinate system, increasing the complexity.

The presented line descriptor and matching algorithms have been developed with computational efficiency in mind. In visual SLAM systems, the aim is to achieve real time performance. Therefore, it can not be afforded to spend too much time and resources just to track features.

The presented line descriptor and matching algorithm trade accuracy for computational efficiency. If one tries to match an existing line with all the lines found in the image, more than one positive match will probably be found. This happens because the presented line descriptor is not as robust or accurate in order to distinguish itself among all the lines found in the image. It was already mentioned, that as the images are expected to be captured with a high frequency, the displacement of the lines between frames should not be too big. Thus, a fancy line descriptor is not needed in this case.

Regarding the descriptor, an analogy can be made with SIFT feature points [33]. SIFT feature points are very robust and accurate, but are computationally very expensive, making their application in real time visual applications usually infeasible. To solve this, a *guided search* approach is used in this thesis. This means, that an estimation/projection over the tracked lines is made to reduce the number of candidate lines for matching. This can be seen in figure 6.13.

To determine the transformation between two consecutive frames, i.e. how the camera has moved, visual odometry is used. *ScaViSLAM* uses a dense tracking approach based on the Lucas-Kanade optical flow. Especially it implements the approach used by Newcombe *et al.* [38] by minimizing the following photometric energy:

$$x^2 = \sum_{u,v} \rho\left(\left(I_l^{[n]}(u,v) - I^{[n+1]}(\hat{z}_{mono}(T_{n+1} \cdot y_{u,v}))\right)^2\right) \tag{6.22}$$

with respect to the camera pose $T_{n+1} \in \mathbf{SE}(3)$. I_l and I_r are rectified images, $y_{u,v}$ the 3D points and ρ a robust kernel. "One way to interpret this least-squares method is the following: It estimates an optical flow field which is forced to be consistent with the dense point model as well as with a rigid body motion. The least-squares optimisation can be performed very efficiently

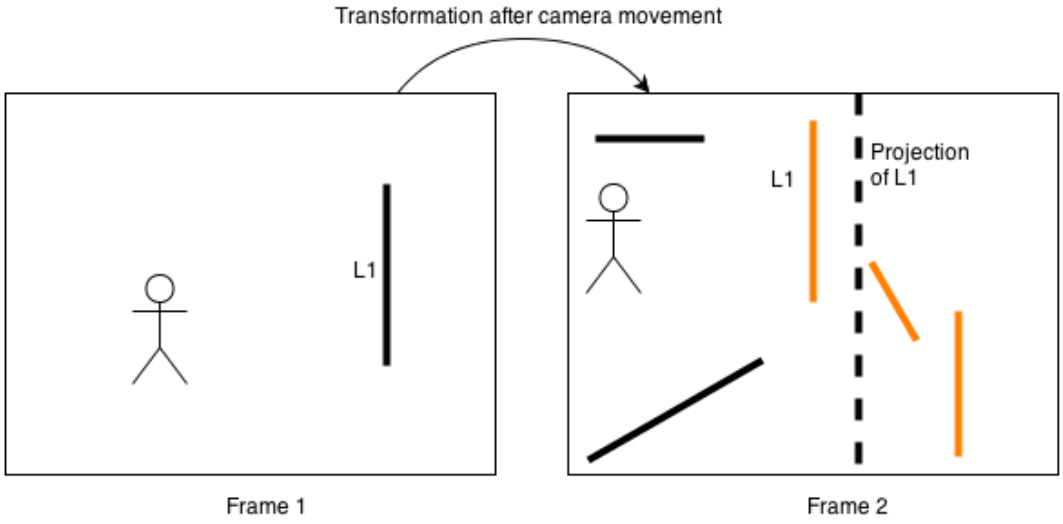


Figure 6.13: Guided Matching: After moving the camera, one seeks to find line 1. With visual odometry, the projection of the old line into the new frame can be calculated. Then, only lines that are *near* the projection (i.e. the orange lines) are considered for matching the descriptors.

on a modern GPU. Afterwards, the pose is refined using motion-only bundle adjustment on the sparse point cloud. This way we make sure that the pose remains consistent to the 3D points, which are estimated using double window optimisation"[1].

After computing the projection of the tracked line in the new frame, the euclidean distance between all the current lines to the projection is computed. This way, only lines that are near the projection are chosen to match their descriptors, as can be seen in figure 6.13. Note that, in projective space, a line $A * x + B * y + C = 0$ can be multiplied by a factor K and still represent the same line. Therefore, in order to compute the euclidean distance, both lines have to be normalized and the have same signs, in case of $k = -1$. For example, if after normalization we have $[-A, B, -C]$ and $[A, -B, C]$, one of the lines has to be multiplied by $k = -1$ before computing the euclidean distance.

The problem with using the euclidean distance to compare linear forms of lines is, that it is not possible to use a fixed threshold to determine which lines are near the projection. The reason for this is that steep lines cause the A, B components to rise by two orders of magnitude. Thus, with horizontal lines, a good threshold could be < 80, but with steep lines < 20000. Therefore, instead of using a fixed threshold, we always take the three lines with the smallest euclidean distance as candidate lines. Exploring the use of other

[1] Explanation taken from Strasdat's thesis [46]

similarity measures which don't suffer from this problem remains open. An interesting idea to overcome this problem would be to use polar coordinates, like the ones used by the Hough Transform.

Before detailing all the algorithm, explaining the transformation of Plücker lines into the new frame is needed. In section 2.5.1, the computation and backprojection of Plücker lines into the image were introduced, but assuming a non-moving camera. Now, before projecting the Plücker lines back into the image, the aforementioned transformation between two frames has to be applied. The formula for this is derived in Eq. 6.23 and the meaning of the used symbols is illustrated in figure 6.14.

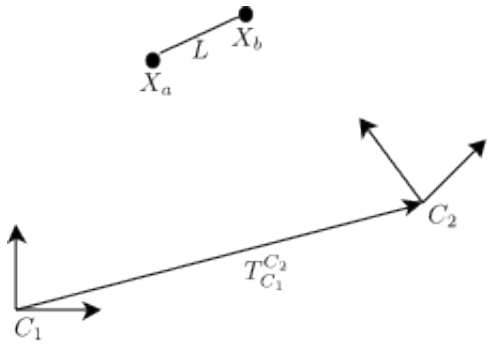


Figure 6.14: Transformation of Plücker lines between two different coordinate systems.

$$
\begin{aligned}
X^{C_1} &= T_{C_1}^{C_2} X^{C_2} \\
L^{C_1} &= X_a^{C_1} X_b^{C_1 T} - X_b^{C_1} X_a^{C_1 T} \\
L^{C_2} &= X_a^{C_2} X_b^{C_2 T} - X_b^{C_2} X_a^{C_2 T} \\
L^{C_1} &= T_{C_1}^{C_2} X_a^{C_2} X_b^{C_2 T} T_{C_1}^{C_2 T} - T_{C_1}^{C_2} X_b^{C_2} X_a^{C_2 T} T_{C_1}^{C_2 T} \\
L^{C_1} &\simeq T_{C_1}^{C_2} L^{C_2} T_{C_1}^{C_2 T} \\
&\text{Note: equality only if normalization is introduced.}
\end{aligned}
\tag{6.23}
$$

The algorithm to transform, project and match lines will now be introduced, in Algorithm 2. The computation related to Plücker coordinates can be found in section 2.5.1.

6.5.1 Algorithm

Algorithm 2: Update set of tracked lines every frame

Input: Set of tracked lines in last frame t_{n-1}, $T_{C_1}^{C_2}$ $4x4$ homogeneous transformation matrix, linesOnCurrentFrame
Output: Updated set of tracked lines in current frame t_n

```
projectionMatrixP ← KR[I| − c] using T_{C1}^{C2} for Eq. 2.40
foreach tracked line L do
    transformedPluckerMatrix L' ← T_{C1}^{C2} · L^{C2} · T_{C1}^{C2T}
    [l]_x = PL'P^T (see Eq. 2.41)
    lineEq = normalize([l]_x)
    if LineIsInsideFrame(lineEq, intersecPoint1, intersecPoint2) then
        nearestLines ←
        findNearestLinesOnCurrentFrame(lineEq, linesOnCurrentFrame)
        foreach nearestLine nL do
            matchTwoLineDescriptors(L.lineDescriptor, nL.descriptor)
        end
    end
end
UpdateTrackedLines
```

Some notes regarding Algorithm 2. After computing the tranformation and projection of the line, a check is performed to see if the line is still inside the current frame. Consequently, if it is not inside the frame anymore, i.e. has at least two intersection points with the frame's edges, we can discard the line and proceed with the next one. In order to calculate if the lines is inside the frame, we view the four edges of the frame also as lines. For this, since the coordinates of the edges are known, their $Ax + By + C * 1 = 0$ line equation forms are first computed using equation Eq. 6.24.

Given two points $\vec{P}$ and $\vec{Q}$, the line equation joining both points is defined as:

$$(P_y - Q_y)x + (Q_x - P_x)y + (P_x \cdot Q_y - Q_x \cdot P_y) = 0 \tag{6.24}$$

Then, the computation of the intersection between two lines $a_1 * x + b_1 * y + c_1 = 0$ and $a_2 * x + b_2 * y + c_2 = 0$, is easy. First, it is necessary to check if an intersection exists, by checking if they are parallel lines. There exists an intersection only if

$$a_1 \cdot b_2 \neq a_2 \cdot b_1 \tag{6.25}$$

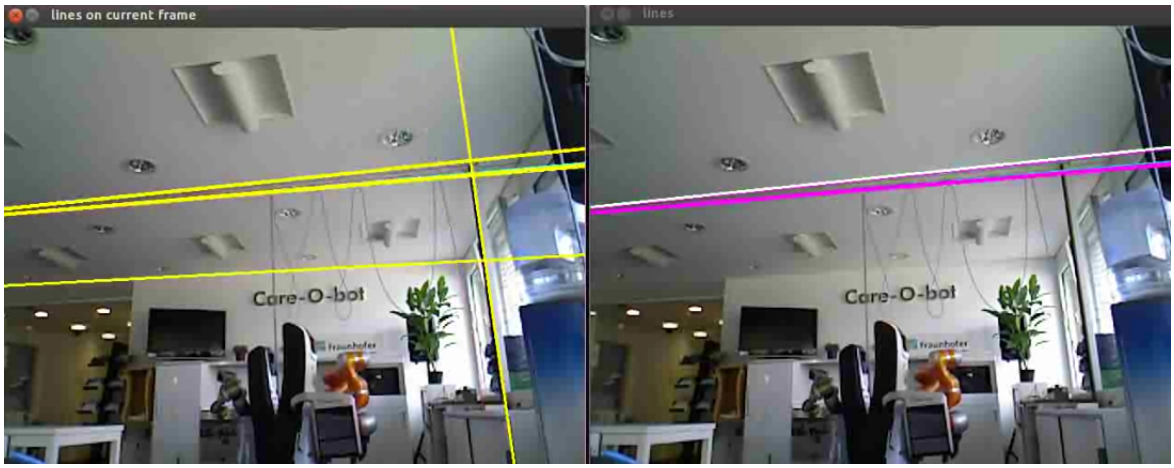
If an intersection exists, it can be computed using standard linear algebra (there are two equations and two variables). The result is given by Eq. 6.26:

$$\begin{aligned} x &= (b_2 \cdot c_1 - b_1 \cdot c_2)/(a_2 \cdot b_1 - a_1 \cdot b_2) \\ y &= (a_1 \cdot c_2 - a_2 \cdot c_1)/(a_2 \cdot b_1 - a_1 \cdot b_2) \end{aligned} \tag{6.26}$$

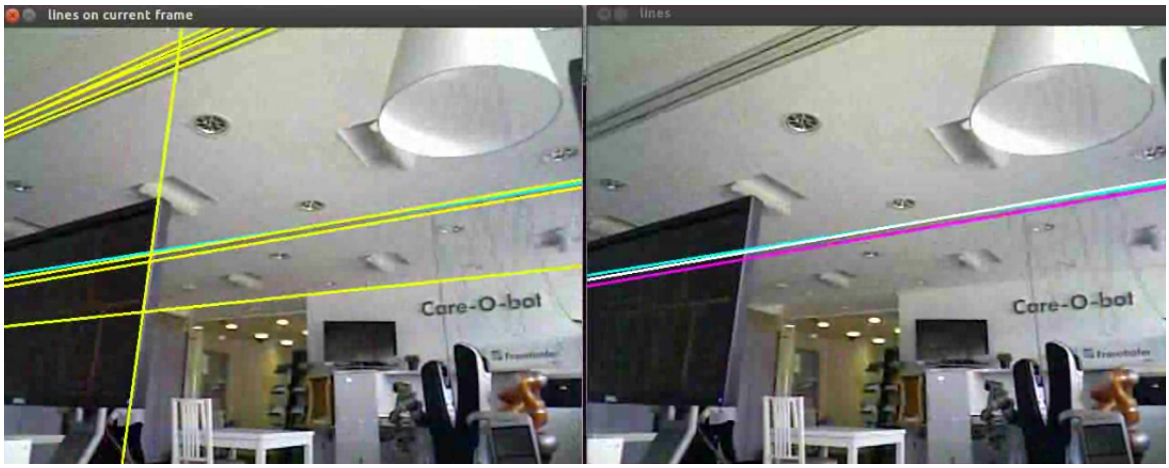
Now, the remaining bit is to check if the intersecting point $\vec{P}$ lies inside the frame. In the case of the Kinect camera, this means that $0 <= P_x <= 640$ and $0 <= P_y <= 480$. A line is considered to be inside of the frame, if it has two valid intersection points with two different edges of the frame.

6.5.2 Qualitative Evaluation

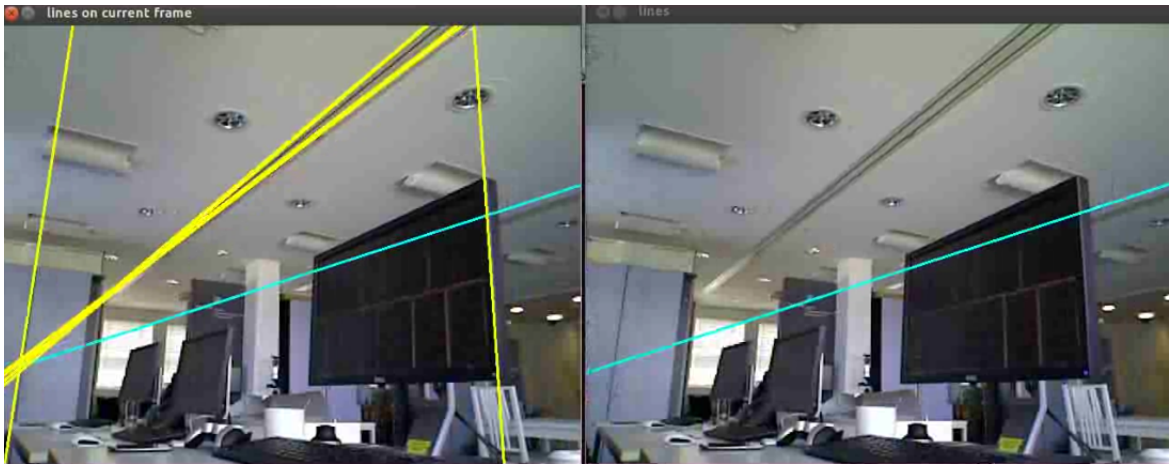
As no objective evaluation has been made of the developed line tracker, an example will be shown in figure 6.15 and 6.16, in which one line will be tried to match as the camera moves freely in 3D. Only one line is tracked for the sake of illustrating the guided search matching method, because when tracking 10-20 lines it is more challenging to visualize the matches.



(a) Projection and matched line without camera movement.

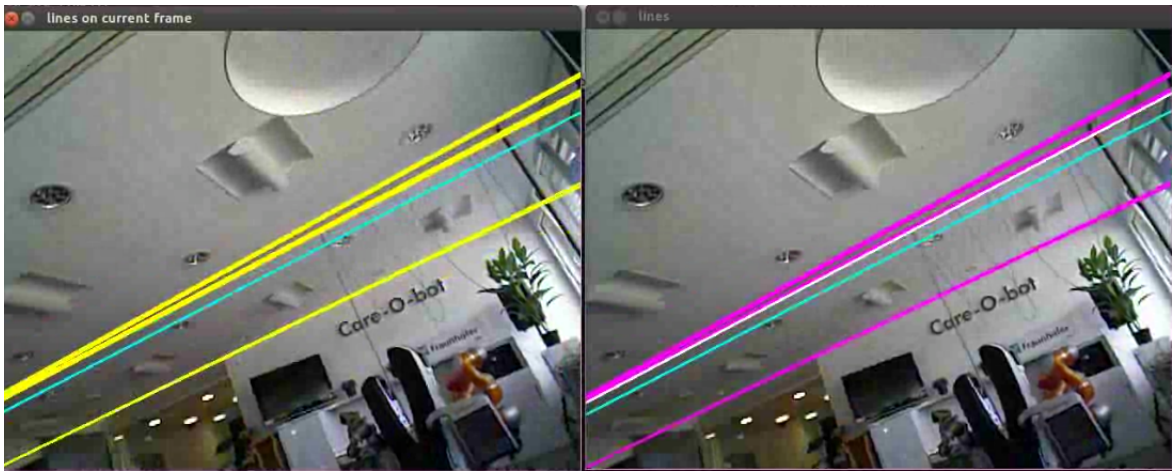


(b) After translating the camera to the left, the projection and the matched line are still correct.



(c) After translating the camera even more to the left, there is no match because that line wasn't detected with Hough, so there are no near candidates. Even if there were candidates, the descriptors would not match, since the initial line is now not visible anymore.

Figure 6.15: Example of tracking one line with camera translation. Yellow lines represent lines found on the current frame. The green line is the projection of the tracked line. Purple lines represent candidate lines that are near the projection. A white line illustrates the matched line.



(a) Camera has been rotated 45° and translated. The projection now has drifted a little bit because of the visual odometry, but the matched line is still correct.

(b) Camera has been rotated 90° and translated. Projection and matched line are correct.

Figure 6.16: Example of tracking one line with camera translation and rotation. The window on the left with the yellow lines show all the lines found on the current frame. The green line is the projection of the tracked line. Purple lines represent candidate lines that are near the projection. A white line illustrates the matched line.

Chapter 7

Optimization of lines in the Backend

Contents

7.1 Graph optimization with lines

Now that we have a line tracker, we want to use them to improve *ScaViSLAM*. This means, that the line's information should be integrated in *ScaViSLAM*'s backend and its graph optimization process. In order to achieve this, its double window approach needs to be extended to incorporate line information. For example, some of the functionalities/methods that have to be rewritten, in order to take advantage of the line information, are:

- Definition of the inner and outer windows
- Deciding when to add a new keyframe
- Metric loop closures
- Computation of the topological neighborhood of a keyframe
- Defining new covisibility weight: number of feature points and lines that are visible from the topological neighborhood

Due to project scheduling issues, it was not possible to rewrite *ScaViSLAM* completely to integrate the line's at all levels. Nevertheless, the lines were

incorporated to the optimization graph. At the beginning of the thesis GraphSLAM, ScaViSLAM and g^{2}o were introduced to illustrate how the SLAM problem can be represented as a graph and solved with optimization techniques. The general workflow of the implemented line integration is:

1. Save for each keyframe the currently tracked lines, i.e. the observations.
2. Maintain a *lineTable* data structure in the backend, where the optimized Plücker parameters for all tracked lines are saved. This data structure is independent from the frontend's *trackedLines* data structure. Therefore, even if a line is deleted from the frontend's *trackedLines*, it is not deleted from the backend's *lineTable*.
3. Compute in the backend an active line set, which is made up of all the unrepeated line's identifiers (*id*) that are associated with frames in the inner window. Even if the same line should appear in more than one frame, it's *id* will be appear only once in the active line set.
4. When *ScaViSLAM* decides to optimize, we search for lines of the active line set in all the frames included in the double window. If we find a match, we retrieve the last optimized Plücker coordinates from the *lineTable* and register the line as a vertex in the optimization graph if it hasn't been registered yet. If there are no last optimized values, for instance because it is a new line, we take the observation for initialization purposes. Next, we connect the Line vertex with the frame/pose vertex as an edge with the line's observation from that pose.
5. At the end of the optimization, we update *lineTable* with new optimized values. Besides, the optimized values are passed to the frontend to enable more accurate projection of the tracked lines.

ScaViSLAM still crashes when it sees insufficient feature points, because the line information was not integrated into its covisibility weight measure. Despite this problem, with the inclusion of the lines, the estimated pose of the camera should be more accurate.

7.2 Evaluation

For evaluation purposes, the optimization can be done by using only points, by using only lines or using both. A qualitative evaluation was done by running the software on the Care-O-Bot mobile robot and drawing the resulting poses in ScaViSLAM's GUI, as shown in Figure 7.1. Nevertheless, for an quantitative evaluation it is necessary to compare the estimated trajectory against the real one. This can be done in two different ways.

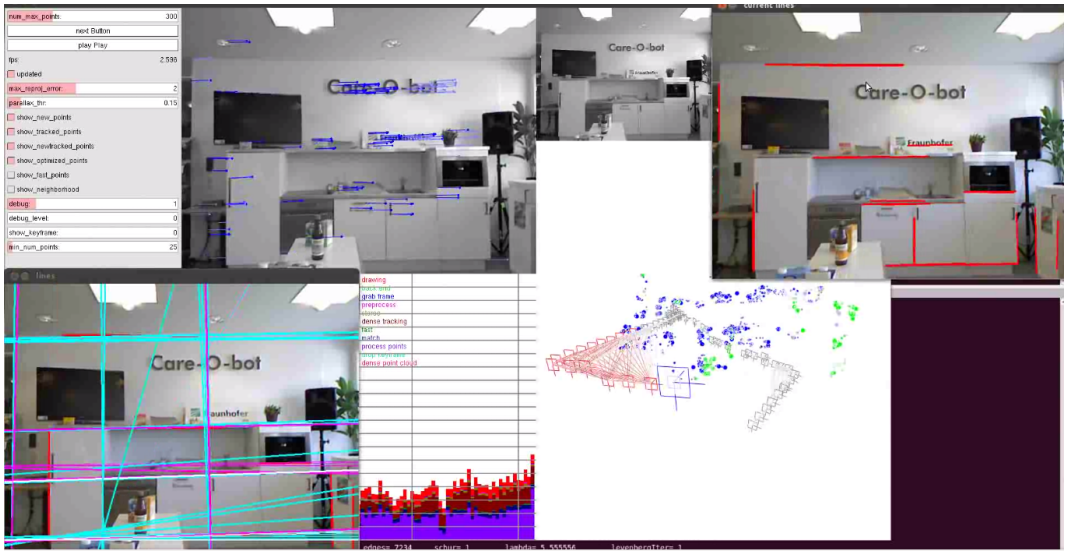


Figure 7.1: Image of ScaViSLAM running on the Care-O-Bot mobile robot. On the left down window, the line tracker is shown. In the middle, the estimated trajectory of the robot. On the top left window, the lines found on the current frame.

The first one is by using an already recorded data set, like the outdoors New College dataset[1] of Smith *et al.* [44]. This dataset was originally used to test ScaViSLAM. Thus, it would be interesting to run the same tests as Hauke Strasdat and compare the results. In order to achieve this, it is necessary to use the same error measure. As ScaViSLAM has no fixed global origin, comparing absolute poses is meaningless. Therefore, a relative error in terms of relative differences $\Delta T_{ij} := T_i T_j^{-1}$ between two absolute poses is used. Besides, the root mean square error is defined over the estimated and the true relative translations,

[1] `http://www.robots.ox.ac.uk/NewCollegeData/`

$$\sqrt{\frac{1}{|\mathcal{E}|}\sum_{E_{ij}\in\mathcal{E}}(\Delta t_{ij}^{[est]}-\Delta t_{ij}^{[true]})^2} \tag{7.1}$$

with Δt_{ij} being the translational component of ΔT_{ij}. ScaViSLAM defines the resulting poses in world coordinates. Therefore, in order to use the aforementioned error measure, we need to calculate the transformation between two consecutive poses, as can be seen in figure 7.2.

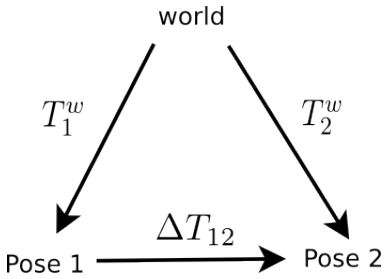


Figure 7.2: Image illustrating two poses in world coordinates and the transformation between them

This transformation can be computed as follows:

$$\begin{aligned} X_w &= T_1^w X_1 \\ X_w &= T_2^w X_2 \\ \Rightarrow T_2^w X_2 &= T_1^w X_1 \\ X_1 &= \underbrace{T_1^{-1w} T_2^w}_{\Delta T_{12}} X_2 \end{aligned} \tag{7.2}$$

The second method on the other hand, involves creating a new indoor dataset. In this case, by using the Care-O-bot mobile robot developed by Fraunhofer IPA, which is illustrated in figure 7.3. The advantage over the New College dataset is, that inside a building, i.e. in a man-made structured environment, it is easier to find lines, which leads to getting a better pose estimate by using line information. As the New College dataset takes place outdoors, not much lines can be found, which hampers testing the developed improvements which rely on straight line primitives.

Since our main goal was to improve localization in textureless areas where few feature points can be found, it makes sense to use such an environment for evaluation purposes. As the Care-O-Bot has a static map of the environment and is equipped with laser scanners, its localization should be accurate. In *ROS*, we can retrieve the poses through the *tf* topic.

Figure 7.3: Image of Care-O-bot mobile robot developed by Fraunhofer IPA

Sadly, it was not possible to perform this quantitative evaluation on time. A indoor dataset was recorded with the Care-O-Bot, and the developed software runs with both datasets, but it was not possible to implement the error measure on time. Nevertheless, the author will try to do this even after presenting the thesis.

Part IV

Future Work

Table of Contents

Chapter 8

Future Work

Contents

8.1 Future Work

Due to already mentioned project scheduling issues, it was not possible to finish everything on time. Therefore some issues that should or could be addressed in the future are:

- Finishing a quantitative evaluation of the resulting estimated poses by using only points information, using only lines or both. The New College outdoors dataset and a new created indoors dataset could be used.
- Explore the use polar coordinates to find nearby candidates of the projected line. This would probably help to overcome the limitations of the euclidean distance as a similarity measure with steep lines. Currently it doesn't allow the use of a fixed threshold and therefore we are forced to always choose the 3 lines with the smallest euclidean error as candidates.
- Do a quantitative evaluation of the line tracker, to see if the descriptor works as expected or new improvements are necessary. This could be done by using depth information around a line.
- Extend the use of line information to the remaining parts of ScaViSLAM. Currently, ScaViSLAM still crashes when it detects to few feature points because line information is not taken into account in the covisibility weight. Some areas where line information needs to be included are:

  - Definition of the inner and outer windows
  - Deciding when to add a new keyframe
  - Metric loop closures
  - Computation of the topological neighborhood of a keyframe
  - Defining new covisibility weight: number of feature points and lines that are visible from the topological neighborhood

- Finally, implementing a monocular frontend would be interesting. Possible approaches would be to integrate PTAM or to write an open source monocular tracker to overcome PTAM's restrictive license.

Bibliography

[1] *CMake*. http://www.cmake.org.

[2] *CVD*. http://www.edwardrosten.com/cvd/.

[3] *GIT dokumentazioa*. http://git-scm.com/documentation.

[4] *Point Cloud Library*. http://pointclouds.org.

[5] *Pangolin*. https://github.com/strasdat/Pangolin.

[6] *Sophus*. https://github.com/strasdat/Sophus.

[7] *TooN*. http://www.edwardrosten.com/cvd/toon.html.

[8] *VisionTools*. https://github.com/strasdat/VisionTools.

[9] *g2o*. https://github.com/strasdat/g2o.

[10] *GVars*. http://www.edwardrosten.com/cvd/gvars3.html.

[11] *OpenCV*. http://opencv.org.

[12] Adrien Angeli, David Filliat, Stéphane Doncieux, and J-A Meyer. Fast and incremental method for loop-closure detection using bags of visual words. *Robotics, IEEE Transactions on*, 24(5):1027–1037, 2008.

[13] Michael Blosch, Stephan Weiss, Davide Scaramuzza, and Roland Siegwart. Vision based mav navigation in unknown and unstructured environments. In *Robotics and automation (ICRA), 2010 IEEE international conference on*, pages 21–28. IEEE, 2010.

[14] Jack E Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems journal*, 4(1):25–30, 1965.

[15] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (6):679–698, 1986.

[16] Denis Chekhlov, Mark Pupilli, Walterio Mayol, and Andrew Calway. Robust real-time visual slam using scale prediction and exemplar based fea-

ture description. In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, pages 1–7. IEEE, 2007.

[17] Diogo Santos Ortiz Correa, Diego Fernando Sciotti, Marcos Gomes Prado, Daniel Oliva Sales, Denis Fernando Wolf, and Fernando Santos Osório. Mobile robots navigation in indoor environments using kinect sensor. In *Critical Embedded Systems (CBSEC), 2012 Second Brazilian Conference on*, pages 36–41. IEEE, 2012.

[18] Franklin C Crow. Summed-area tables for texture mapping. In *ACM SIGGRAPH Computer Graphics*, volume 18, pages 207–212. ACM, 1984.

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

[20] Frank Dellaert, Dieter Fox, Wolfram Burgard, and Sebastian Thrun. Monte carlo localization for mobile robots. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, volume 2, pages 1322–1328. IEEE, 1999.

[21] Albert Diosi and Lindsay Kleeman. Advanced sonar and laser range finder fusion for simultaneous localization and mapping. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 2, pages 1854–1859. IEEE, 2004.

[22] Nikolas Engelhard, Felix Endres, Jürgen Hess, Jürgen Sturm, and Wolfram Burgard. Real-time 3d visual slam with a hand-held rgb-d camera. In *Proc. of the RGB-D Workshop on 3D Perception in Robotics at the European Robotics Forum, Vasteras, Sweden*, volume 2011, 2011.

[23] Bin Fan, Fuchao Wu, and Zhanyi Hu. Robust line matching through line–point invariants. *Pattern Recognition*, 45(2):794–805, 2012.

[24] Javier Garcia, Zeev Zalevsky, et al. Range mapping using speckle decorrelation, October 7 2008. US Patent 7,433,024.

[25] Rafael Grompone von Gioi, Jérémie Jakubowicz, Jean-Michel Morel, and Gregory Randall. LSD: a Line Segment Detector. *Image Processing On Line*, 2012, 2012. doi: 10.5201/ipol.2012.gjmr-lsd.

[26] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, second edition, 2004.

[27] J. Hertzberg, K. Lingemann, and A. Nüchter. *Mobile Roboter: Eine Einführung aus Sicht der Informatik*. Springer-Verlag GmbH, 2012. ISBN 9783642017254.

[28] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots*, 2013. doi: 10.1007/s10514-012-9321-0. URL `http://octomap.github.com`. Software available at `http://octomap.github.com`.

[29] Nahum Kiryati, Yuval Eldar, and Alfred M Bruckstein. A probabilistic hough transform. *Pattern recognition*, 24(4):303–316, 1991.

[30] Georg Klein and David Murray. Parallel tracking and mapping for small ar workspaces. In *Mixed and Augmented Reality, 2007. ISMAR 2007. 6th IEEE and ACM International Symposium on*, pages 225–234. IEEE, 2007.

[31] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard. g2o: A general framework for graph optimization. In *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, Shanghai, China, May 2011.

[32] Jesse Levinson and Sebastian Thrun. Robust vehicle localization in urban environments using probabilistic maps. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 4372–4378. IEEE, 2010.

[33] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[34] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit. FastSLAM: A factored solution to the simultaneous localization and mapping problem. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, Edmonton, Canada, 2002. AAAI.

[35] Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. Fastslam 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. In *International Joint Conference on Artificial Intelligence*, volume 18, pages 1151–1156. LAWRENCE ERLBAUM ASSOCIATES LTD, 2003.

[36] Kevin Murphy. Bayesian map learning in dynamic environments. *Advances in Neural Information Processing Systems (NIPS)*, 12:1015–1021, 1999.

[37] Richard A Newcombe, Andrew J Davison, Shahram Izadi, Pushmeet Kohli, Otmar Hilliges, Jamie Shotton, David Molyneaux, Steve Hodges, David Kim, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *Mixed and Augmented Reality (ISMAR), 2011 10th IEEE International Symposium on*, pages 127–136. IEEE, 2011.

[38] Richard A Newcombe, Steven J Lovegrove, and Andrew J Davison. Dtam: Dense tracking and mapping in real-time. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2320–2327. IEEE, 2011.

[39] *Robot Operating System*. Open Source Robotics Foundation, www.ros.org.

[40] Patrick Pfaff, Rudolph Triebel, and Wolfram Burgard. An efficient extension to elevation maps for outdoor terrain mapping and loop closing. *The International Journal of Robotics Research*, 26(2):217–230, 2007.

[41] David Ribas, Pere Ridao, Juan Domingo Tardós, and José Neira. Underwater slam in man-made structured environments. *Journal of Field Robotics*, 25(11-12):898–921, 2008.

[42] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In *Computer Vision–ECCV 2006*, pages 430–443. Springer, 2006.

[43] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: an efficient alternative to sift or surf. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2564–2571. IEEE, 2011.

[44] Mike Smith, Ian Baldwin, Winston Churchill, Rohan Paul, and Paul Newman. The new college vision and laser data set. *The International Journal of Robotics Research*, 28(5):595–599, 2009.

[45] Paul Smith, Ian Reid, and Andrew Davison. Real-time monocular slam with straight lines. In *British Machine Vision Conference*, volume 1, pages 17–26, 2006.

[46] H. Strasdat. *Local Accuracy and Global Consistency for Efficient Visual SLAM*. PhD thesis, Imperial College, October 2012.

[47] H. Strasdat, A.J. Davison, J. M M Montiel, and K. Konolige. Double window optimisation for constant time visual slam. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2352–2359, 2011.

[48] H. Strasdat, J.M.M. Montiel, and A.J. Davison. Visual slam: Why filter? *Image and Vision Computing*, 30(2):65–77, 2 June 2012.

[49] Sebastian Thrun. Simultaneous localization and mapping. In *Robotics and cognitive approaches to spatial mapping*, pages 13–41. Springer, 2008.

[50] Sebastian Thrun and Michael Montemerlo. The graph slam algorithm with applications to large-scale mapping of urban structures. *The International Journal of Robotics Research*, 25(5-6):403–429, 2006.

[51] Sebastian Thrun, Wolfram Burgard, Dieter Fox, et al. *Probabilistic robotics*, volume 1. MIT press Cambridge, 2005.

[52] Paul Viola and Michael Jones. Robust real-time object detection. In *International Journal of Computer Vision*, 2001.

[53] Chieh-Chih Wang, Charles Thorpe, and Sebastian Thrun. Online simultaneous localization and mapping with detection and tracking of moving objects: Theory and results from a ground vehicle in crowded urban areas. In *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*, volume 1, pages 842–849. IEEE, 2003.

[54] Zhiheng Wang, Fuchao Wu, and Zhanyi Hu. Msld: A robust descriptor for line matching. *Pattern Recognition*, 42(5):941–953, 2009.

[55] Stephan Weiss, Markus W Achtelik, Simon Lynen, Margarita Chli, and Roland Siegwart. Real-time onboard visual-inertial state estimation and self-calibration of mavs in unknown environments. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 957–964. IEEE, 2012.

[56] Lilian Zhang and Reinhard Koch. Hand-held monocular slam based on line segments. In *Proceedings of the 2011 Irish Machine Vision and Image Processing Conference*, IMVIP '11, pages 7–14, Washington, DC, USA, 2011.

[57] Lilian Zhang and Reinhard Koch. Hand-held monocular slam based on line segments. In *Proceedings of the 2011 Irish Machine Vision and Image Processing Conference*, pages 7–14. IEEE Computer Society, 2011.

[58] Lilian Zhang and Reinhard Koch. Line matching using appearance similarities and geometric constraints. In *Pattern Recognition*, pages 236–245. Springer, 2012.