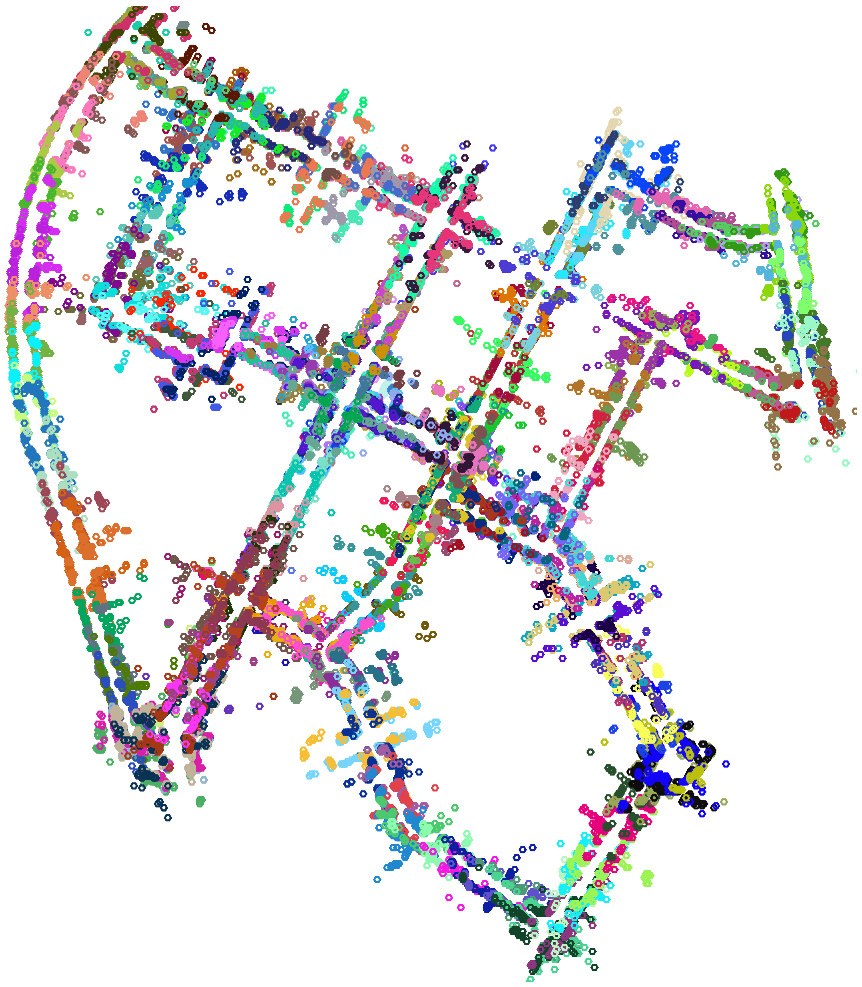
CHALMERS
UNIVERSITY OF TECHNOLOGY



Simultaneous Localization and Mapping

for Vehicle Localization using LIDAR Sensors

Master's thesis in Master Programme Systems Control and Mechatronics

Robin Lindholm
Carl-Johan Pålsson

Department of Signals and Systems
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2015

Master's thesis EX026/2015

Simultaneous Localization and Mapping

for Vehicle Localization using LIDAR Sensors

Robin Lindholm
Carl-Johan Pålsson

Department of Signals and System
Chalmers University of Technology
Gothenburg, Sweden 2015

Simultaneous Localization and Mapping
for Vehicle Localization using LIDAR sensors
ROBIN LINDHOLM
CARL-JOHAN PÅLSSON

Supervisor: Anders Karlsson, Volvo Group
Examiner: Lennart Svensson, Signals and Systems

Master's Thesis EX026/2015
Department of Signals and Systems
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Map of keypoints from the Kitti Vision Benchmark Suite Dataset.

Typeset in LaTeX
Gothenburg, Sweden 2015

LIDAR SLAM Simultaneous Localization and Mapping
for Vehicle Localization using LIDAR Sensors
ROBIN LINDHOLM
CARL-JOHAN PÅLSSON

Department of Signals and Systems
Chalmers University of Technology

Abstract

The vehicle industry is developing increasingly advanced driver assistance systems, making progress towards a fully self-driving vehicle. Potential benefits include increased safety, lower fuel consumption and higher productivity. To have an effective control system, a reliable position estimate and a map of the environment are crucial.

A dilemma in robotics is that in order to construct a map, an accurate robot position estimation is necessary, but to know the robot's position a map is required. Simultaneous Localization and Mapping (SLAM) is the problem of constructing a map of an unknown environment while simultaneously performing localization within the map.

This thesis describes an implementation of the GraphSLAM algorithm using LIDAR sensors for perception of the environment. The algorithm includes both a robust front-end for graph construction and a robust back-end for graph optimization. A feature-based registration algorithm is implemented to detect loop closures.

The algorithm was tested on public datasets and a proprietary dataset with good results. Different types of maps were generated including compact maps designed for vehicle localization. A separate algorithm based on the particle filter was implemented for localization within the generated maps. The algorithm was able to determine vehicle position with sub-meter level accuracy. Several sources of errors were identified and suggestions for future improvements were proposed.

Keywords: SLAM, GraphSLAM, Localization, LIDAR, Features, Loop Closure, Robust.

Acknowledgements

We would like to express our sincerest gratitude to our supervisors Anders Karlsson and Erik Stenborg. Their support and unlimited patience and enthusiasm helped fuel us during all the challenges we encountered. We would also like to express our indebtedness to our examiner Lennart Svensson whose endless curiosity and attention to details helped us understand the theory behind the problem.

Samuel Scheidegger and Mathias Ernst, who worked on visual SLAM, have our deepest gratitude for all the collaboration between our groups during the entire project. Many great ideas and solutions came to fruition from all the discussions, as both groups worked towards solutions for very similar problems.

On top of that all we also want to thank everyone involved in the study groups including Nicklas Gustafsson, Maryam Fatemi, Lars Hammarstrand and Malin Lundgren.

Maybe the biggest thanks goes to Volvo Group for taking the initiative to, and financing the project and we appreciate the support they showed during the project. Without them there wouldn't have been any project all.

Volvo Cars also deserves credit for granting us access to data from their Autonomous Drive Dataset and for providing support during the project.

Robin Lindholm & Carl-Johan Pålsson, Gothenburg, June 2015

Contents

List of Figures

List of Tables

List of Algorithms

1
Introduction

The vehicle industry is adding more and more advanced driver assistance systems making progress towards a fully self-driving vehicle. The potential benefits of autonomous vehicles are large and numerous. Each year about 1.2 million people are killed in road traffic accidents, of which 90% are caused by human error [1]. Advanced control algorithms can reduce fuel consumption by 4-10% by accelerating and decelerating more smoothly than a human driver and significantly decrease the cost of congestion [2]. By having a computer in control of the vehicle, the driver is able to do other productive or enjoyable activities such as working, reading, watching a movie or even sleeping during the trip. To have an effective control system, a reliable position estimate and a map of the environment are crucial.

A dilemma in robotics is that in order to construct a map, accurate robot position estimation is necessary, but to know the robot's position a map is required. Similar as many other catch-22 problems one solution is to solve both problems at the same time. Simultaneous Localization and Mapping (SLAM) is the process of updating a map of an unknown environment while keeping track of the position of the vehicle. SLAM has been applied in self-driving cars, unmanned aerial and underwater vehicles, planetary rovers, domestic robots and even within human bodies. [3]

One environment sensing technology, commonly used for obstacle detection, is Light Detection and Ranging (LIDAR), which estimates the distance to an object from the time delay between a light pulse and the detection of the Reflection. By moving the ranging sensor during the scan a point cloud representing the environment is generated. These point clouds can be used for place recognition and the motion estimation needed to solve the SLAM problem.

1.1 Objective

The aim of this project is to examine the available algorithms for solving the SLAM problem and find which ones are suitable for LIDAR sensor data. Challenges for a full scale commercial implementation are investigated by implementing a complete solution. It is the goal that the results will be useful for Volvo Group, Volvo Cars and other parties in deciding which algorithms to develop further.

1.2 Scope

This project will examining available methods and focus on implementing a complete SLAM algorithm. Only LIDAR based sensors will be examined, other vision sensors such as mono- and stereo cameras will not be examined. A real time solution for embedded controllers is not attempted and while execution is analyzed and tried to be minimized, the project is not limited to a real time solution.

1.3 Work Division

Most of the work in this report was done in collaboration of both the authors of this thesis and thus responsibility was mostly equally shared. For example the Graph-SLAM main algorithm, the Front-End, the cartography and the localization was done by both authors working side by side. Carl-Johan had the major responsibility for the Back-End and the pre-processing while Robin had the major responsibility for the feature based registration and the robust loop closure. Carl-Johan was responsible for the C++ implementation of GICP while Robin were responsible for the two Matlab implementations of ICP. The rest of the work(literature search, report writing, testing, evaluation etc) was not divided, but performed in collaboration between the two authors.

1.4 Outline

In chapter 1 the problem is introduced and the objective of the thesis is defined. Chapter 2 gives an outline of how the project was performed. The following two chapters gives first the fundamental background theory and later more specific theory for the SLAM problem. This division is a bit vague, but the intent is that by moving out some of the more general theory outside the SLAM theory it will make the SLAM theory more readable. Chapter 5 introduces the datasets and the sensors used in the datasets. In chapter 6 the implementation is presented, with elaborations about the choices made in the implementation. Chapter 7 presents some relevant results from the implementation. The chapter is complemented by chapter 8 where the results are evaluated and analyzed and limitations are investigated. In chapter 9 future work for further improvement and some interesting challenges found during the project is presented.

2
Methods

In this chapter we present a brief description of the method used in the project.

2.1 Literature Study

To acquire relevant knowledge about the problem a literature study was done. S.Thrun [4] presents an extensive overview of the SLAM problem, which was used frequently during the project. The lectures from the course Robot Mapping at Freiburg University were watched and the exercise material was performed which included implementing EKF-SLAM, FastSLAM and GraphSLAM [5]. As challenges were found during the project several different academic articles were read, some of these are cited in the thesis.

2.2 Study Groups

The project was performed in collaboration with another master thesis group working SLAM using cameras, one professor, two industrial PhDs acting as supervisors, three additional PhDs with knowledge about the subject and one engineer from Autoliv AB. The group had meetings weekly in the first half of the project where topics were chosen to be discussed, relevant scientific papers were read and presentations about the topics were done by one of the master thesis groups. The topics for the presentations held by this group include

- GraphSLAM
- Registration
- Condition and Marginalization in GraphSLAM
- Robust Front-End
- Feature-Based Registration for Point Clouds
- Robust Back-End

2.3 Implementation

To simplify the development of the complete implementation of a SLAM algorithm for LIDAR sensors, we opted to use datasets with high quality sensors. Public datasets from Ford Campus Dataset, Kitti Vision Benchmark Dataset and proprietary datasets from Volvo Cars were selected.

We developed a pre-processing MATLAB script that converts the data from the datasets, to an easy to access MATLAB format. In this script the different timestamps from the different sensors were converted to Unix time format and features were extracted from LIDAR scans. This batch pre-processing turned out to be useful, so the main algorithm did not have to redo the processing while minor changes were made to the main algorithm. Especially since the data heavy LIDAR sensors were used.

A complete implementation of a SLAM algorithm was developed which loops through the preprocessed data and constructs various different types of maps. Maps that were produced include large point clouds for the entire observed environment during the drive, map of a specified region of the drive, maps of features and maps of features along a predefined path. One point cloud of the entire environment could be useful for visualization purposes, for path planning or for generating different kinds of maps. The map of features along a predefined path was tried for localization of a car based on new LIDAR data and noisy odometry measurements.

3 Background Theory

In this chapter, the underlying background theory for this thesis is presented. Most, if not all, of these topics are very broad and cannot be fully explored within this work.

3.1 Recursive Bayesian Filtering

Bayes filter is the most general algorithm for calculating posterior distributions [4]. It estimates the posterior distribution $bel(x_t)$ over the state x_t recursively based on measurements z_t and control inputs u_t at time t. Bayes filter assumes the true state to be an unobserved Markov process and the measurements to be observed states from a hidden Markov model. Both of these are based on the Markov property; if it is possible to make predictions for the future based solely on the present state of the system equally well as if the system's entire history was known, the future and past are independent.

Algorithm 1 Bayes Filter

```
Require: $bel(x_{t-1}),\ u_t,\ z_t$
1: for all $x_t$ do
2:     $\overline{bel}(x_t) = \int p(x_t|u_t, z_t)\ bel(x_{t-1})\ dx_{t-1}$
3:     $bel(x_t) = \eta\ p(z_t|x_t)\ \overline{bel}(x_t)$
4: end for
5: return $bel(x_t)$
```

Bayes filter, presented in Algorithm 1, operates in basically two steps, prediction and update. In the prediction step, line 2, the previous posterior distribution is extrapolated to the time of the measurement according to the control input. In the update step in line 3, the measurement is taken into account to improve the estimate.

3.1.1 Kalman Filter

The Kalman filter is probably the most recognized technique for implementing Bayes filters [4]. It works in the same manner as the Bayesian filter but with the assumption of Gaussian distributions. The Kalman filter represents posterior distributions by the moment parametrization. For each time t, the posterior distribution is represented by the mean μ_t and covariance Σ_t. Three properties are required for the posterior to be Gaussian:

1. The state transition probability $p(x_t|u_t, z_t)$ must be linear with added Gaussian noise:
$$x_t = A_t x_{t-1} + B_t u_t + \varepsilon_t \tag{3.1}$$
where x_t is the state vector at time t and u_t is the control vector evaluated at time t. A_t and B_t are the linear state transition and control input models respectively. The random noise ε represents the uncertainty introduced by the state transition, as a distribution with zero mean and covariance Q_t. The mean of the posterior is given by $A_t x_{t-1} + B_t u_t$ and the covariance by Q_t.
$$\begin{aligned} &p(x_t|u_t, z_t) = \\ &\quad \det(2\pi Q_t)^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(x_t - A_t x_{t-1} - B_t u_t)^\top Q_t (x_t - A_t x_{t-1} - B_t u_t)\right) \end{aligned} \tag{3.2}$$

2. The measurement probability $p(z_t|x_t)$ must also be linear with added Gaussian noise:
$$z_t = C_t x_t + \delta_t \tag{3.3}$$
where z_t is the measurement vector and C_t the linear measurement model. δ represents the measurement noise as a distribution with zeros mean and covariance R_t. Thus the measurement probability is given by
$$p(z_t|x_t) = \det(2\pi R_t)^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(z_t - C_t x_t)^\top R_t (z_t - C_t x_t)\right) \tag{3.4}$$

3. The initial distribution $bel(x_0)$ must be normally distributed with mean μ_0 and covariance Σ_0
$$bel(x_0) = p(x_0) = \det(2\pi\Sigma_0)^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(x_0 - \mu_0)^\top \Sigma_0 (x_0 - \mu_0)\right) \tag{3.5}$$

The Kalman filter, presented in Algorithm 2, operates in a similar fashion as the Bayes filter algorithm with prediction and update steps. In lines 1 and 2, the predicted distribution $\overline{bel}_t$ represented by $\overline{\mu}_t$ and $\overline{\Sigma}_t$ is calculated up to the time of the measurement by incorporating the control input u_t but not the measurement itself.

Algorithm 2 Kalman Filter

```
Require: $\mu_{t-1},\ \Sigma_{t-1},\ u_t,\ z_t$
1: $\overline{\mu}_t = A_t \mu_{t-1} + B_t u_t$
2: $\overline{\Sigma}_t = A_t \Sigma_{t-1} A_t^\top + R_t$

3: $K_t = \overline{\Sigma}_t C_t^\top (C_t \overline{\Sigma}_t C_t^\top + Q_t)^{-1}$
4: $\mu_t = \overline{\mu}_t + K_t(z_t - C_t \overline{\mu}_t)$
5: $\Sigma_t = (I - K_t C_t)\overline{\Sigma}_t$
6: return $\mu_t,\ \Sigma_t$
```

In lines 3 through 5, the measurement is incorporated to estimate the posterior distribution $bel(x_t)$. The variable K_t is the Kalman gain, which specifies to what degree the new measurement should be incorporated into the new state estimate. In line 4, the mean is adjusted in proportion to the Kalman gain and the innovation. The innovation is the difference between the measurement z_t and the expected measurement $C_t\overline{\mu}_t$. Finally, the covariance Σ_t is adjusted due to the information from the measurement.

3.1.2 Particle Filter

The Particle filter is a nonparametric implementation of the Bayes filter, where the posterior distribution $bel(x_t)$ is estimated by a finite number of parameters [4]. By representing a distribution by a set of random state samples drawn from the distribution, a broader space of distributions can be represented than by, for example, Gaussians. This representation has the advantage that it's able to model nonlinear transformations. The samples are referred to as particles, hence the name particle filter, and are denoted by

$$\mathcal{X}_t = \left\{x_t^{[1]}, x_t^{[2]}, \ldots, x_t^{[M]}\right\} \tag{3.6}$$

where M is the number of particles. Generally the filter will perform better with more particles but with more particles computational complexity grows. Every particle $x_t^{[m]}$ is a hypothesis of the true state at time t. Ideally, a state hypothesis x_t should have a likelihood proportional to the posterior $bel(x_t)$ to be included in the set $\mathcal{X}_t$.

$$x_t^{[m]} \sim p(x_t|z_{1:t}, u_{1:t}) \tag{3.7}$$

It follows that the denser a region is populated with particles the more likely it is for the true state to be within this region.

Algorithm 3 Particle Filter

```
Require: $\mathcal{X}_{t-1},\ u_t,\ z_t$
 1: $\overline{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$
 2: for $m = 1$ to $M$ do
 3:     Sample $x_t^{[m]} \sim p(x_t|u_t, x_{t-1}^{[m]})$
 4:     $\omega_t^{[m]} = p(z_t|x_t^{[m]})$
 5:     $\overline{\mathcal{X}}_t = \overline{\mathcal{X}}_t + \langle x_t^{[m]}, \omega_t^{[m]} \rangle$
 6: end for
 7: for $m = 1$ to $M$ do
 8:     Draw $i$ with probability $\propto \omega_t^{[i]}$
 9:     Add $x_t^{[i]}$ to $\mathcal{X}_t$
10: end for
11: return $\mathcal{X}_t$
```

The Particle filter does have some similarities with the Kalman filter, namely the sampling in line 3. This is more or less a prediction step where the control input is integrated to extrapolate the state to the time of the measurement. In the update step, the measurement is used to calculate the importance factor, or weight, for each particle. This weight is the probability of the measurement z_t given the particle $x_t^{[m]}$, as shown in line 4.

In line 7 through 10, in Algorithm 3, the real strength of the Particle filter occurs, which is the resampling step. Resampling is when a new set of particles $\mathcal{X}_t$ is drawn from the temporary set $\overline{\mathcal{X}}_t$, where the probability of drawing a specific particle is defined by its importance factor. This changes how the particles are distributed. The resampling step refocuses the particle set to regions with high posterior probability as the particles with low importance weights tend to not be included. Inference can be done by choosing the weighted mean of the distribution, by taking the most likely particle or by making a histogram of the particles.

3.1.3 Information Canonical Parametrization

Gaussian distributions are usually represented by moments, a mean vector μ and a covariance matrix Σ [4]. A duality of this representation is the information canonical parametrization where a Gaussian distribution is represented by an information vector ξ and an information matrix Ω. These representations are connected through an inversion of the covariance

$$\begin{aligned} \Omega &= \Sigma^{-1} \\ \xi &= \Sigma^{-1}\mu \end{aligned} \tag{3.8}$$

This relation holds in the other direction as well but with the information matrix inverted. Thus the parametrization of the distribution in the two different domains are inversions of each other

$$p(\alpha,\beta) = \underbrace{\mathcal{N}\left(\begin{bmatrix}\alpha\\ \beta\end{bmatrix}; \begin{bmatrix}\mu_\alpha\\ \mu_\beta\end{bmatrix}, \begin{bmatrix}\Sigma_{\alpha\alpha} & \Sigma_{\alpha\beta}\\ \Sigma_{\beta\alpha} & \Sigma_{\beta\beta}\end{bmatrix}\right)}_{\text{Covariance domain}} = \underbrace{\mathcal{N}^{-1}\left(\begin{bmatrix}\alpha\\ \beta\end{bmatrix}; \begin{bmatrix}\xi\alpha\\ \xi_\beta\end{bmatrix}, \begin{bmatrix}\Omega_{\alpha\alpha} & \Omega_{\alpha\beta}\\ \Omega_{\beta\alpha} & \Omega_{\beta\beta}\end{bmatrix}\right)}_{\text{Information domain}} \tag{3.9}$$

There are benefits and drawbacks with using either domain. What is cheap in the covariance domain is expensive in the information domain and vice versa. Usually, a matrix inversion is considered an expensive operation and avoided wherever possible.

Table 3.1: Marginalization and conditioning in the Covariance and Information domain

	Covariance domain	Information domain
Marginalization $p(\alpha) = \int p(\alpha,\beta)$	$\mu = \mu_\alpha$ $\Sigma = \Sigma_{\alpha\alpha}$	$\xi = \xi_\alpha - \Omega_{\alpha\beta}\Omega_{\beta\beta}^{-1}\xi_\beta$ $\Omega = \Omega_{\alpha\alpha} - \Omega_{\alpha\beta}\Omega_{\beta\beta}^{-1}\Omega_{\beta\alpha}$
Conditioning $p(\alpha\|\beta) = \frac{p(\alpha,\beta)}{p(\beta)}$	$\mu' = \mu_\alpha + \Sigma_{\alpha\beta}\Sigma_{\beta\beta}^{-1}(\beta - \mu_\beta)$ $\Sigma' = \Sigma_{\alpha\alpha} - \Sigma_{\alpha\alpha}\Sigma_{\beta\beta}^{-1}\Sigma_{\beta\alpha}$	$\xi' = \xi_\alpha - \Omega_{\alpha\beta}\beta$ $\Omega' = \Omega_{\alpha\alpha}$

As seen in Table 3.1, the covariance domain offers cheap marginalization and expensive conditioning while the information domain has the opposite pricing. For some applications different one of these representations will greatly reduce computational complexity.

3.2 Projective Geometry

Projective geometry is commonly used in computer graphics, 3D computer vision and robotics and has some advantages compared to the standard Euclidean geometry. For instance, points at infinity can be represented by finite coordinates, transformations can be easily represented as matrices and formulas are often simpler and more symmetric.

Homogeneous coordinates are used in projective geometry just as Cartesian coordinates are used in Euclidean geometry. By definition, a coordinate $\mathbf{x}$ is said to be homogeneous if $\mathbf{x}$ and $\lambda\mathbf{x}$ represent the same object for $\lambda \neq 0$.

The transition between homogeneous and Cartesian coordinates is quite straightforward. A Cartesian coordinate has an equivalent homogeneous coordinate, which for the 2D case is

$$\mathbf{x} = \underbrace{\begin{bmatrix} x \\ y \end{bmatrix}}_{Cartesian} \Leftrightarrow \underbrace{\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} wx \\ wy \\ w \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}}_{Homogeneous} \tag{3.10}$$

Rigid body transformations in projective geometry can be represented by an invertible linear mapping on the form

$$M = \begin{bmatrix} R & \mathbf{t} \\ \mathbf{0}^\top & 1 \end{bmatrix} \tag{3.11}$$

where R is a rotation matrix and $\mathbf{t}$ a translation vector.

A projective transform is a linear mapping

$$\mathbf{x}_j = M_{ij}\mathbf{x}_i. \tag{3.12}$$

where the transform M_{ij} map homogeneous coordinate $\mathbf{x}_j$ from $\mathbf{x}_i$. These transforms can be chained to map over multiple coordinates as

$$\mathbf{x}_2 = M_{12}M_{01}\mathbf{x}_0 \tag{3.13}$$

where the coordinate $\mathbf{x}_2$ has been mapped over $\mathbf{x}_1$ from $\mathbf{x}_0$.

The inverse of a transformation represent a mapping in the opposite direction. Therefore, an analytical expression of said inverse can be very beneficial. This expression can be derived by solving

$$I = M^{-1}M = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} R & \mathbf{t} \\ \mathbf{0}^\top & 1 \end{bmatrix} \tag{3.14}$$

for A, B, C and D. By applying the orthogonality property of the rotation matrix, the following result is acquired:

$$M^{-1} = \begin{bmatrix} R^\top & -R^\top\mathbf{t} \\ \mathbf{0}^\top & 1 \end{bmatrix}. \tag{3.15}$$

A transform can be represented by a vector containing both a coordinate $\mathbf{x}$ and an orientation θ, i.e. a pose. The symbol $\bowtie$ is introduced to denote the change in representation:

$$\begin{aligned} \mathbf{p}^{\bowtie} &= M \\ M^{\bowtie} &= \mathbf{p} \end{aligned} \tag{3.16}$$

where

$$\mathbf{p} = \begin{bmatrix} \mathbf{x} \\ \theta \end{bmatrix} \tag{3.17}$$

and $\bowtie$ signify the mapping

$$\begin{bmatrix} R(\theta) & \mathbf{t}(\mathbf{x}) \\ \mathbf{0}^\top & 1 \end{bmatrix} \xleftrightarrow{\bowtie} \begin{bmatrix} \mathbf{x} \\ \theta \end{bmatrix} \tag{3.18}$$

where R and T are the rotation matrix and translation vector respectively. It is worth noting that poses can be described from several different frames of reference, e.g. M_{ij} and M_{oj} describe the same pose p_j.

3.3 Registration

Registration is the process of calculating the transformation which aligns two sets of data into one coordinate frame for comparison and integration [6]. Both image and point set registration algorithms are widely used in many fields to capture and analyze real world data. For robots and vehicles, registration is often used to estimate the ego-motion of said robot or vehicle. This is done by calculating the transformation which best aligns the data from two different samples while subject to sensor noise.

There exist several methods to estimate the transformations between two 3D objects, but this thesis has mainly focused on iterative closest point and feature-based registration. Both methods estimates the rigid body transformation between two sets of corresponding points. A brief overview of how one can estimate the rigid body transformation between two corresponding sets of points and both the algorithms algorithms are presented here.

3.3.1 Estimating Rigid Body Transformations

Finding the rigid body transformation between two sets of corresponding points is the backbone of registration and is known as the absolute orientation problem. The problem can be stated as the following. Assume two corresponding point sets $\{a_i\}$ and $\{b_i\}$, $i = 1 \dots N$ are related by

$$a_i = Rb_i + T + \mathbf{v}_i \tag{3.19}$$

where R is a rotation matrix, $\mathbf{t}$ a translation vector and $\mathbf{v}_i$ is a noise vector. The goal is to find the optimal rotational matrix R^* and translational vector $\mathbf{t}^*$ which minimizes the least-squares criterion

$$\Sigma^2 = \sum_{i=1}^{N} \|a_i - Rb_i - \mathbf{t}\|^2. \tag{3.20}$$

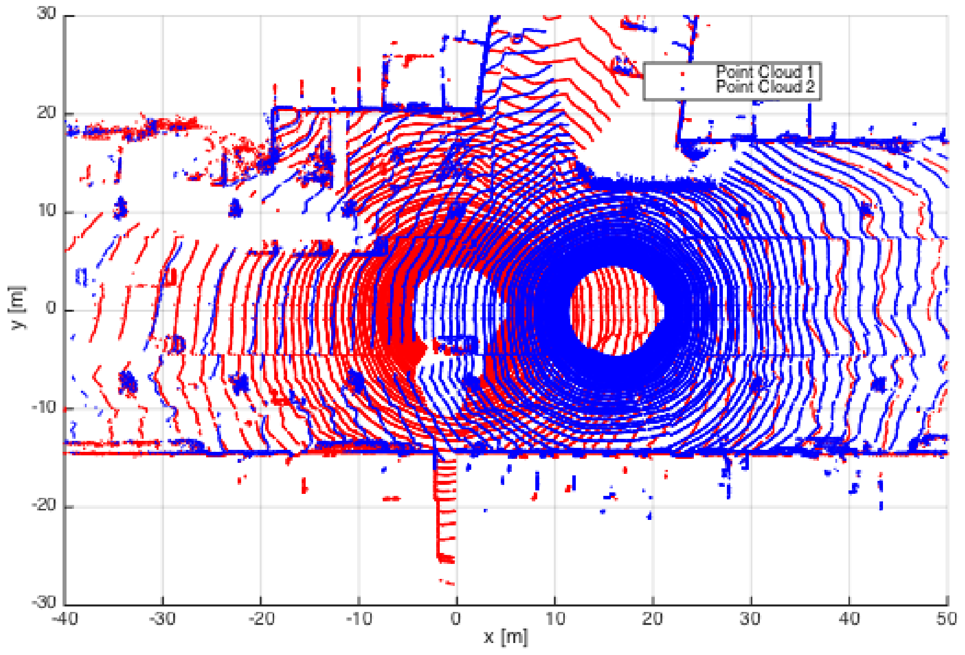


Figure 3.1: Registration of two point clouds viewed from above. The red point cloud is taken 0.3seconds after the blue point cloud and has been registered to the blue point cloud using GICP. As can be seen from the picture the walls and cars line up well while the ground scans around the vehicle do not have corresponding points, which is reasonable since the vehicle has moved. Scans were from the Volvo Cars Dataset.

The problem is well studied and many algorithms are available [7]. Singular Value Decomposition (SVD) is a fast and high performing algorithm often used. Other approaches include the calculation of the eigensystem while exploiting the properties of the rotation matrix or using quaternions to represent rotations.

Algorithm 4 Rigid transform estimation using SVD

```
Require: $\{a_i\}$, $\{b_i\}$
1: $\bar{a} = \frac{1}{N}\sum_{i=1}^{N} a_i, \qquad a_{ci} = a_i - \bar{a}$
2: $\bar{b} = \frac{1}{N}\sum_{i=1}^{N} b_i, \qquad b_{ci} = b_i - \bar{b}$
3: $H = \sum_{i=1}^{N} b_{ci} a_{ci}^{\top}$
4: $[U, \Lambda, V] = \text{SVD}(H)$
5: $R^* = VU^{\top}$
6: $\mathbf{t}^* = \bar{a}_{ci} - R^* \, \bar{b}_{ci}$
```

The outline of using SVD to estimate rigid body transformations is presented in Algorithm 4. It assumes that both point sets have the same centroid, i.e. geometric center. These centroids are calculated in line 1 and 2. This lead to 3.20 being minimized when the trace of R^*H is maximized. The rotation matrix which provides this maximum is calculated in line 5. The translation is the difference between the point sets after the rotation.

3.3.2 Iterative Closest Point

Iterative Closest point(ICP) is an algorithm, which minimizes the difference between two point clouds by iteratively finding correspondences between the two sets of points [8]. In the algorithm, one cloud, the *Target*, is fixed while the other cloud, the *Source*, is transformed. In each iteration the closest neighbor of each point in the source is found by using a search algorithm and the rigid body transformation between the target points and their closest neighbor can be estimated in the same way as in algorithm 4. The entire target point cloud is then transformed using the rigid body transformation estimation and a new closest neighbor search is performed. This process is iterated until convergence, thus the name Iterative Closest Point.

Algorithm 5 Iterative Closest Point

```
Require:
     Point clouds: $A = \{a_i\}, B = \{b_i\}$
     Initial transformation $M_0$
 1: $M = M_0$
 2: while not converged do
 3:     for $i = 1$ to $N$ do
 4:         $c_i =$ FindClosestPointInA$(M \cdot b_i)$
 5:         if $\|c_i - M \cdot b_i\| \leq d_{max}$ then
 6:             $w_i = 1$
 7:         else
 8:             $w_i = 0$
 9:         end if
10:     end for
11:     $M = \underset{M}{\operatorname{argmin}} \{\sum_i w_i \|M \cdot b_i - c_i\|^2\}$
12: end while
13: return $M$
```

In Algorithm 5, the outline of point-to-point ICP is presented. The target A and source B is of course necessary but the initial transformation M_0 is more or less optional. If no initial transform is known, it is set to identity, which correspond to no transformation at all. In line 4, the set of target points which are closest to the source set is found. For some applications it might be beneficial to down-sample the point clouds before registration. This reduces the necessary computational power at the expense of accuracy.

To account for the fact that some points will not have any correspondence, a threshold d_{max} is used to define the weights in line 5 though 9. This threshold represents a trade-off between accuracy and convergence in most implementations of ICP. In line 11 a similar problem to equation 3.20 is expressed which a similar problem as solved in algorithm 4. This sequence of operations it repeated until converged or a fixed number of iterations has been reached.

Another popular implementation of ICP is the point-to-plane variant. It improves the performance by taking advantage of surface normals. Instead of minimizing the expression in line 11, the point-to-plane algorithm minimizes the error along the surface normals with

$$M = \underset{T}{\operatorname{argmin}} \left\{ \sum_i w_i \|\eta_i (M \cdot b_i - c_i)\| \right\} \tag{3.21}$$

where η_i is the surface normal at m_i. Point-to-plane is generally more accurate and robust than the standard point-to-point ICP but requires significantly more computational power to calculate the surface normals [8, 9]. A further extension of this is the Generalized ICP (GICP) which improves on both the point-to-point and point-to-plane frameworks. This is partially done by taking into account the surface information from both clouds contrary to only one cloud. A derivation of this implementation can be found in [8].

3.3.3 Feature-Matching

Instead of taking whole point clouds into consideration, a few *features* can be used to register said clouds [10]. This greatly reduces the computational complexity of the problem as well as the amount of information to store. A feature is generally defined as an "interesting" part of an image or a pattern which differs from its immediate surrounding. Since the point clouds often do not have complete point-point correspondence, which can happen when the point clouds only partially overlap, a lot of effort has been made into feature selection and description extraction to improve correspondence selection.

Two sets of features from two different scans are matched and a set of matching feature pairs is selected. For each matching pair, a homogeneous transformation can be calculated through the least-squares formulation in 3.20. Usually, there are some incorrect matches and resulting transformations which have to be removed with an outlier detection algorithm e.g. RANSAC.

A feature is often generated in basically two steps, detection and description. In the detection step, the image is searched for points of interest or keypoints. Corners, intersections of two or more edges, are generally stable between images and thus often used as interest points. Popular algorithms for corner detection in images include the Harris, and Shi-Tomasi corner detection algorithms [11]. While there are much fewer algorithms for 3D feature detection there has been a number of publications proposing such algorithms in the last years. Commonly used algorithms for 3D point clouds include NARF and 3D-SIFT and for 2D LIDAR point clouds the algorithm FLIRT, which handle occlusion phenomena from ranging data and take advantage of the scale invariance [12, 13].

In the description step, the areas around the keypoints are described. A good keypoint descriptor should capture most of the important and distinctive information in the region of a keypoint so that the keypoint can be recognized if encountered in another image. Another important aspect is the computational cost for comparing descriptions. Commonly used descriptors for images include SIFT, SURF and BRISK [14]. For 2D LIDAR point clouds, FLIRT is a powerful descriptor [12], and

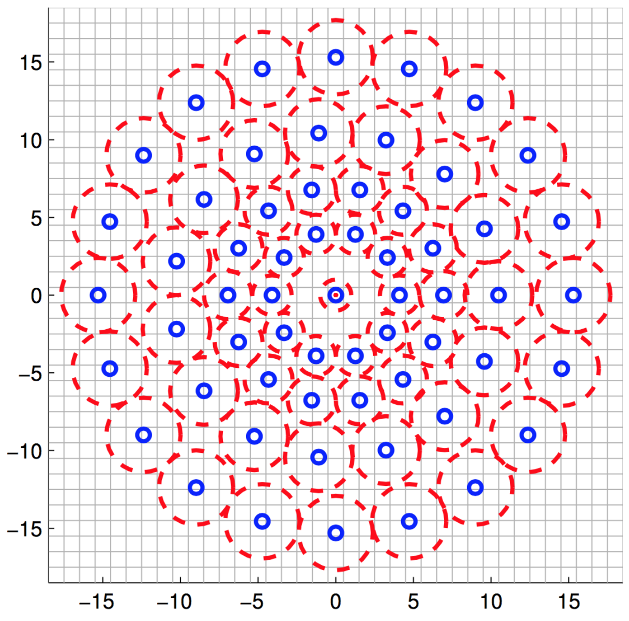


Figure 3.2: Brisk sampling pattern [14]. The blue circles represent the sampling locations and the red dashed circles the corresponding standard deviation of the Gaussian kernel used to smooth the intensity values at the sampling location.

for 3D point clouds there are many popular descriptors such as PFH, FPFH and SHOT [13].

As the Binary Robust Invariant Scalable Keypoints (BRISK) descriptor was used in this thesis, a brief presentation of it is provided here. The BRISK descriptor is composed as a binary string of simple brightness comparison tests around the keypoint in a fixed pattern, Figure 3.2. By identifying the characteristic direction of each keypoint, rotation invariance is achieved which is key for general robustness. Since Brisk has a native binary representation it has an order of magnitude faster keypoint extraction and approximately three times faster feature-matching compared to other popular methods such as SIFT and SURF [14].

3.4 Gauss-Newton Method

The method of least-squares is an approach to the approximate solution of overdetermined systems [15]. These problems are usually formulated to minimize the objective function

$$f(x) = \frac{1}{2}\|r(x)\|_2^2 = \frac{1}{2}\sum_{j=1}^{m} r_j^2(x) = \frac{1}{2}\sum_{j=1}^{m}\left(\Phi(x;t_j) - y_j\right)^2 \tag{3.22}$$

where r_j is the residual, i.e. the difference between the predictions $\phi(x;t_j)$ and the observed value y_j at each t_j. One general algorithm to solve these problems is through the iterative Newton's method

$$\begin{array}{l} \nabla^2 f(x_k)\ \Delta x_k = -\nabla f(x_k) \\ x_{k+1} = x_k + \Delta x_k \end{array} \quad , \quad k = 0, 1, \ldots \tag{3.23}$$

which converge towards a root of f. Finding the inverse of the Hessian can be very expensive, especially in higher dimensions. In such cases it is better to solve for the search direction Δx_k through some linear solver. By collecting all residuals into a vector, the derivatives of f can be expressed as

$$\nabla f(x) = \sum_{j=1}^{m} r_j(x) \nabla r_j(x) = J(x)^\top r(x) \tag{3.24}$$

$$\begin{aligned}\nabla^2 f(x) &= \sum_{j=1}^{m} \nabla r_j(x) \nabla r_j(x)^\top + \sum_{j=1}^{m} r_j(x) \nabla^2 r_j(x) \\ &= J(x)^\top J(x) + \sum_{j=1}^{m} r_j(x) \nabla^2 r_j(x).\end{aligned} \tag{3.25}$$

The Gauss-Newton method is a modified Newton's method where 3.23 is simplified to

$$\begin{matrix} J_k^\top J_k \, \Delta x_k = -J_k^\top r_k \\ x_{k+1} = x_k + \Delta x_k \end{matrix} \quad , \quad k = 0, 1, \ldots \tag{3.26}$$

by neglecting the residual Hessians $\nabla^2 r_j(x)$ in 3.25. This lowers the computational time considerably since only one derivative evaluation is necessary per iteration. It is even possible to circumvent the matrix multiplication in 3.26 by reforming the equation into the linear least-squares problem

$$\min_{\Delta x} \frac{1}{2} \| J_k \, \Delta x + r_k \| \tag{3.27}$$

which can be solved with linear solvers such as SVD or Cholesky factorization.

3.5 A* Algorithm

The A* algorithm is a widely used search algorithm in computer science for finding paths and traversing graphs[16]. It has been proven to find the shortest path in optimal time under certain well-defined conditions [17]. A* improves the time performance of Dijkstra's algorithm by applying a heuristic. Like all informed search algorithms, A* follows the route that is the most likely to lead towards the goal. What separates A* from other best-first search algorithms is the inclusion of the distance traveled as a heuristic. This heuristic must be admissible i.e. not overestimate the distance to the goal. Pseudo code for A* is shown in Algorithm 6.

The A* algorithm starts with the addition of the initial node *start* to the open set of nodes Q. The closed set R and the *path* is initialized to an empty set. In each step, the node with the lowest score is moved from the open set to the closed set while all its neighboring nodes are evaluated. If a neighbor is not already in the open set or is closer to the goal than the current node it is added to the open set. This is repeated until the goal node is reached or the open set is empty. If the goal is reached, the sequence of nodes *path* from *start* to *goal* is returned. Otherwise, false is returned when the open set becomes empty .

Algorithm 6 A*

```
Require: $start, goal$
 1: $Q = start$
 2: $R,\ path = \emptyset$
 3: while $Q \neq \emptyset$ do
 4:     $current = \min(\text{score}(Q))$
 5:     if $current = goal$ then
 6:         return $path = \text{reconstruct\_path}(came\_from)$
 7:     end if
 8:     $Q = Q \setminus current$
 9:     $R = R \cup current$
10:     for each $neighbor$ of $current$ do
11:         if $neighbor \in R$ then
12:             continue
13:         end if
14:         $tentative\_score = \text{score}(current) + \text{distance}(current, neighbor)$
15:         if $neighbor \notin Q$ or $tentative\_score < \text{score}(neighbor)$ then
16:             $came\_from(neighbor) = current$
17:             if $neighbor \notin Q$ then
18:                 $Q = Q \cup neighbor$
19:             end if
20:         end if
21:     end for
22: end while
23: return false
```

4
Simultaneous Localization and Mapping Theory

Simultaneous Localization and Mapping (SLAM) is the process of updating a map of an unknown environment while keeping track of the position of the vehicle. In this chapter, theory for the SLAM problem is presented. The three main categories of algorithms are presented in a chronological order of when they developed.

There are two different cases of the SLAM problem, the *online* and the *full* SLAM problem. In the online SLAM problem, the posteriori distribution of the current pose $\mathbf{p_t}$ and map m is estimated

$$p(\mathbf{p}_t, m \mid z_{1:t}, u_{1:t}) \tag{4.1}$$

where $z_{1:t}$ and $u_{1:t}$ are the measurements and control inputs up to time t respectively. The full SLAM problem estimates the posterior of the entire trajectory [3]

$$p(\mathbf{p}_{0:t}, m \mid z_{1:t}, u_{1:t}). \tag{4.2}$$

4.1 EKFSLAM

Extended Kalman Filter SLAM (EKFSLAM) was the first SLAM algorithm developed and is still one of the most influential algorithms [4, 5]. It applies the EKF framework to the online SLAM problem by using maximum likelihood data association to find a solution to

$$p(\mathbf{p}_t, m \mid z_{1:t}, u_{1:t}) = \eta \underbrace{p(z_t \mid \mathbf{p}_t, m)}_{\text{Measurement model}} \int \underbrace{p(\mathbf{p_t} \mid \mathbf{p}_{t-1}, u_t)}_{\text{Motion model}} \underbrace{p(\mathbf{p}_{t-1} \mid z_{1:t-1}, u_{1:t-1})}_{\text{Prior distribution}} d\mathbf{p}_{t-1}, \tag{4.3}$$

which is equation 4.1 expanded with the use of Markov assumptions and Bayes' rule. The algorithm assumes Gaussian noise for both motion and perception. If the uncertainty of the posterior is large, the linearization can introduce large errors making the algorithm fail.

Maps in EKFSLAM are feature based and comprised of point landmarks. The point landmarks are stored in the state vector and thus estimated along with the pose. Due to computational limitations, the number of points landmarks are usually kept small. The algorithm work best when these is little ambiguity between landmarks, e.g. when using artificial landmarks as beacons. This can be feasible for some environments like factories while long roads are not as viable.

The EKFSLAM algorithm does not scale very well with large maps. The complexity of the algorithm is $O(n^2)$ where n is the number of landmarks. With this complexity the algorithm is computationally intractable for large scale mapping.

4.2 FastSLAM

The FastSLAM algorithm is based on the particle filter approach to the SLAM problem [4, 18]. Since particle filters suffer from the curse of dimensionality, using particles to estimate all landmarks would be infeasible. The FastSLAM algorithm circumvents this curse by factorizing the full SLAM posterior 4.2 into

$$p(\mathbf{p}_{0:t}, m \mid z_{1:t}, u_{1:t}) = p(\mathbf{p}_{0:t} \mid z_{1:t}, u_{1:t}) \prod_{i=1}^{M} p(m_i \mid \mathbf{p}_{0:t}, z_{1:t}) \tag{4.4}$$

which is a product of the robot path posterior and the landmark posterior conditioned on the robot path estimate. The structure of the posterior allows for a version of particle filters known as Rao-Blackwellized Particle Filter. In these, the particles represent the posterior over some variables while other probability density functions represents the other variables.

In FastSLAM, each particle represents a hypothesis of a robot's path and the position of the landmarks. Since the landmarks are conditionally independent of each other given the robot path, the landmarks can be estimated separately. FastSLAM estimates each of these sub problems through a low dimensional EKF. By using a particle filter for the motion, no linearization is necessary in this step and nonparametric distributions can be used while the EKF still assumes a Gaussian distribution for the landmarks positions.

The complexity of FastSLAM is $O(M \cdot N)$ where M is the number of particles and N is the number of landmarks. By exploiting the fact that particles share common landmarks, the computational complexity can be reduced to $O(M \log N)$ if implemented well.

FastSLAM2.0 improves on the FastSLAM re-sampling step by taking the measurement into account when re-sampling the particles. This makes the algorithm perform equally well with less particles.

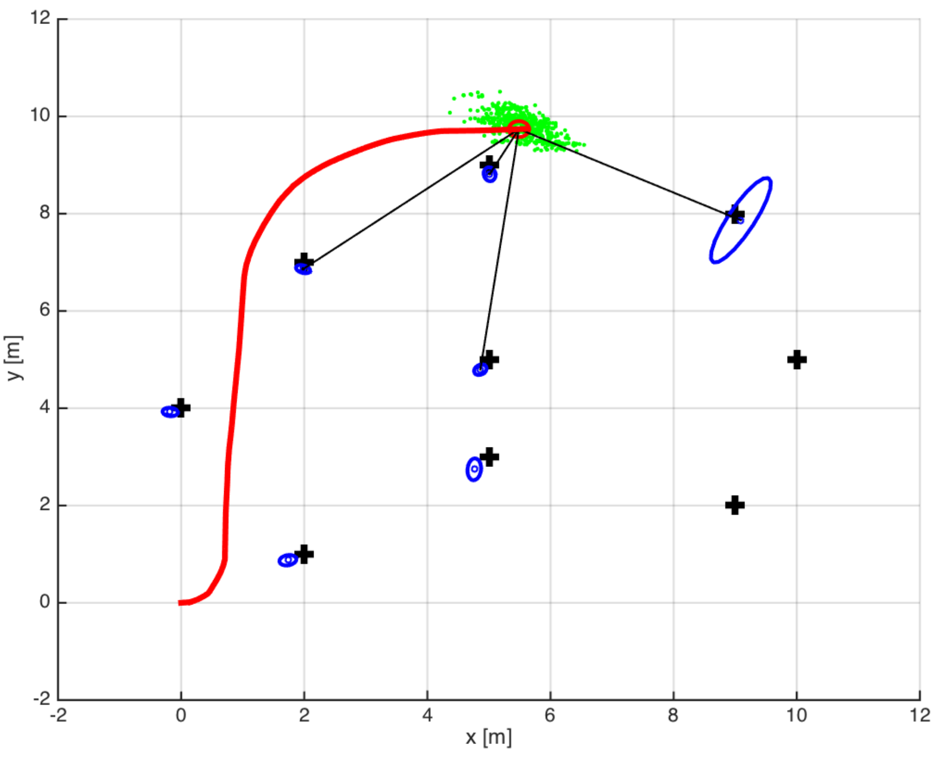


Figure 4.1: FastSLAM algorithm in action. The green dots are the particles. Each particle has its own hypothesis of the robots pose. The red line represents the trajectory of the particle with the highest weighting. The blue ellipsoids are the 1σ-region of the landmarks depicted with a black cross.

4.3 GraphSLAM

The basic idea behind the GraphSLAM algorithm is to model the posterior as a graph where poses and landmarks are represented by nodes, and motions and measurements as edges between the nodes [19, 4, 5, 20]. Edges can be viewed as soft constraints between the nodes. As the motion and measurements are subject to noise the constraints will be contradictory. The goal of the algorithm is to find the poses and map which maximizes the full SLAM posterior 4.2, i.e.

$$[\mathbf{p}^*, m^*] = \underset{\mathbf{p},m}{\operatorname{argmax}}\; p(\mathbf{p}_{0:t}, m \mid z_{1:t}, u_{1:t}). \tag{4.5}$$

To find the maximum, the posterior is expanded to the recursive form

$$p(\mathbf{p}_{0:t}, m \mid z_{1:t}, u_{1:t}) = \\ \eta\; p(z_t \mid \mathbf{p}_t, m)\; p(\mathbf{p}_t \mid \mathbf{p}_{t-1}, u_t)\; p(\mathbf{p}_{0:t-1}, m \mid z_{1:t-1}, u_{1:t-1}) \tag{4.6}$$

which, when considering all times t, leads to the closed form

$$\begin{aligned} p(\mathbf{p}_{0:t}, m \mid z_{1:t}, u_{1:t}) &= \eta\, p(\mathbf{p}_0, m_0) \prod_t p(\mathbf{p}_t \mid \mathbf{p}_{t-1}, u_t)\, p(z_t \mid \mathbf{p}_t, m) \\ &= \eta\, p(\mathbf{p}_0, m_0) \prod_t \left[p(\mathbf{p}_t \mid \mathbf{p}_{t-1}, u_t) \prod_i p(z_t^i \mid \mathbf{p}_t, m) \right] \end{aligned} \tag{4.7}$$

where the term $p(\mathbf{p}_0.m_0)$ is the prior over both the initial pose $\mathbf{p}_0$ and the map m. This prior can be factorized into the independent $p(\mathbf{p}_0)$ and $p(m_0)$. Since there is no prior knowledge about the map, the corresponding prior is placed in the normalizer η.

The robot's motion and measurements are assumed to be normally distributed, and the prior can be expressed as a Gaussian distribution. This correspond to the expressions

$$p(\mathbf{p}_t \mid \mathbf{p}_{t-1}, u_t) = \eta \exp\left\{ -\frac{1}{2} \big(\mathbf{p}_t - g(\mathbf{p}_{t-1}, u_t)\big)^\top Q_t^{-1} \big(\mathbf{p}_t - g(\mathbf{p}_{t-1}, u_t)\big) \right\} \tag{4.8}$$

$$p(z_t^i \mid \mathbf{p}_t, m) = \eta \exp\left\{ -\frac{1}{2} \big(z_t^i - h(\mathbf{p}_{t-1}, m_t)\big)^\top R_t^{-1} \big(z_t^i - h(\mathbf{p}_{t-1}, m_t)\big) \right\} \tag{4.9}$$

$$p(\mathbf{p}_0) = \eta \exp\left\{ -\frac{1}{2} \mathbf{p}_0^\top \Omega_0 \mathbf{p}_0 \right\} \tag{4.10}$$

where g is the motion model and h the measurement model. These equations constitute the constraints or errors whose distributions should be maximized. Since only the exponents are relevant in this regard, the negative logarithm of the posterior

$$\begin{aligned} &- \log p(\mathbf{p}_{0:t}, m \mid z_{1:t}, u_{1:t}) \\ &\qquad = const. + \log p(\mathbf{p}_0) + \sum_t \left[\log p(\mathbf{p}_t \mid \mathbf{p}_{t-1}, u_t) + \sum_i \log p(z_t^i \mid \mathbf{p}_t, m) \right] \end{aligned} \tag{4.11}$$

is examined. The summation of all the quadratic terms from 4.8-4.10 can be collected into one term. This lead to the following

$$\begin{aligned} [\mathbf{p}^*, m^*] &= \operatorname*{argmax}_{\mathbf{p}, m}\ p(\mathbf{p}_{0:t}, m \mid z_{1:t}, u_{1:t}) \\ &= \operatorname*{argmin}_{\mathbf{p}, m}\ \sum_k \mathbf{e}_k^\top \Omega_k \mathbf{e}_k \end{aligned} \tag{4.12}$$

where $\mathbf{e}_k$ are the errors from constraint k and Ω_k the information of said constraint.

The algorithm can be simplified to work without landmarks m. Pose-GraphSLAM utilizes only poses to make nodes and edges are only made between pose-nodes. This lead to the chain of poses representing the map instead of landmarks. Since no landmarks are used, *loop closure*, the act of recognizing previously visited areas, has to be managed with the landmark information stored within the nodes. Compared to using landmarks in the state vector, this leads to a smaller and sparser graph, but features are not optimized relative to the pose of the scan. By comparing multiple correspondences s at the same time, the probability of incorrect loop closures is greatly reduced.

The GraphSLAM algorithm is generally split in two separate problems, the *Front-End* which constructs the graph from the measurements and the *Back-End* which solves the optimization of the state vector $\mathbf{p}$.

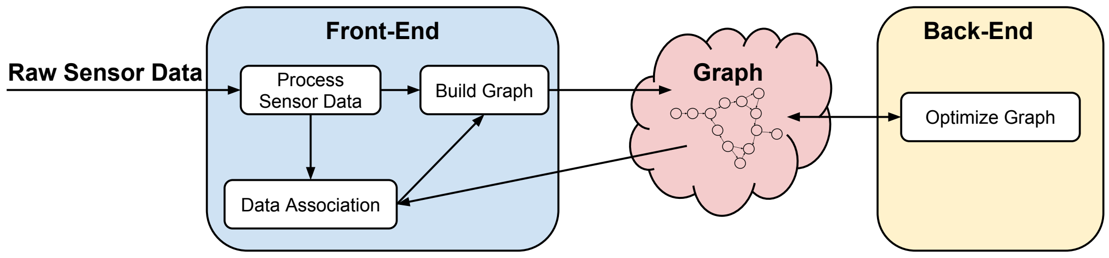


Figure 4.2: Division of the GraphSLAM algorithm into Front-End and Back-End. The Front-End reads the sensors data and constructs the graph that the Back-End optimizes.

4.3.1 Front-End

The Front-End builds the graph of nodes and edges from sensor measurements. While the robot is moving temporally, edges between poses are created in a chain between each pose and its previous and succeeding pose. New landmarks are added as a node with an edge to the pose from which it was first observed. When the robot sees a previously observed landmark, a new edge is formed between the landmark and the pose node currently observing the landmark.

In the pose-GraphSLAM algorithm, since no landmarks are stored, loop closures are formed as constraints between the current pose and a previously observed pose. Loop closure constraints can be formed when a match has been observed between two scans from two different poses without direct temporal connection. Since there is bigger problem of ambiguity for constraints between poses far away from each other temporally, they are often treated differently.

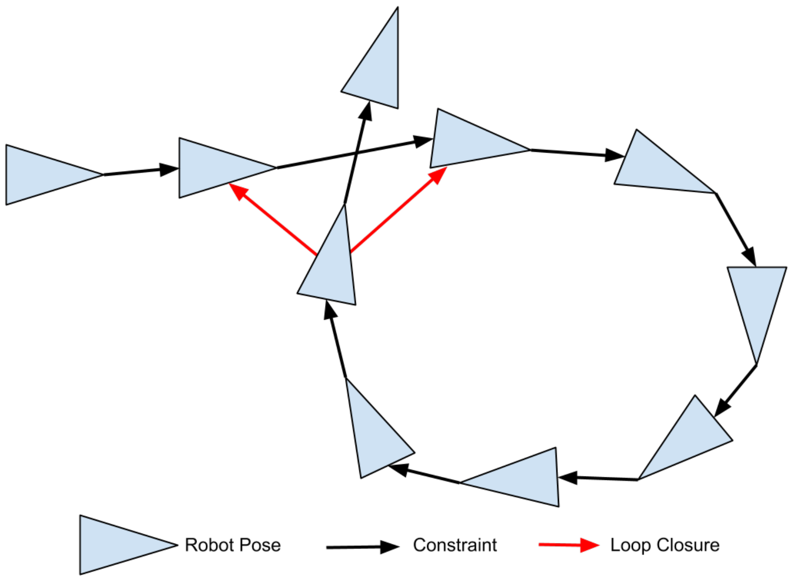


Figure 4.3: Illustration of a loop closure. When the robot returns to an area it has previously visited and is able to recognize where it is in the environment relative to its previous pose.

The constraints from the temporal chain of the robot's movements can either be from odometry sensors, scan- or feature matching, or a combination of multiple sources. Dead reckoning will generally produce low quality maps, as sensors subject to noise will drift over time [5]. Loop closures are therefore desirable to improve the estimation of both the poses and the map. However, doing a correct loop closure can be hard for many reasons. Objects may move over time and similar looking environments may be hard to distinguish. For example, indoor environments often contain walls and corners that appear similar to other walls and corners. It is crucial to avoid incorrect loop closures since they will greatly affect the optimization of the graph.

E.Olson [21] proposed to solve the global ambiguity problem by clustering loop closure hypotheses in local groups, Figure 4.4, and the local ambiguity problem by finding the most pairwise consistent set of hypotheses.

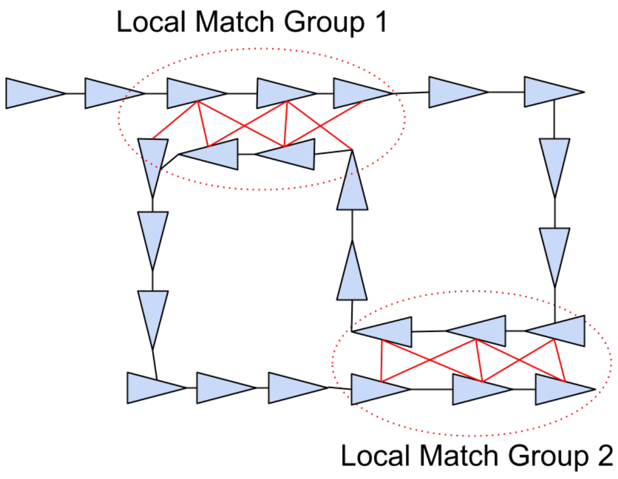


Figure 4.4: Drawing illustrating two local groups of loop closures. The triangles are poses, the black line are odometry constraints and the red lines are loop closure hypotheses.

The main idea of Olson's algorithm is to form a pairwise consistency matrix A. Each element A_{ij} is the probability that the hypotheses h_i and h_j agree. By finding the shortest path between the hypotheses, a loop can be constructed as seen in Figure 4.5 and the probability of hypothesis agreement can be estimated with the information stored in the edges. The shortest path can be found with a graph search algorithm such as A*.

For all combinations of hypotheses, an indicator vector $\mathbf{v}$ is formed

$$\mathbf{v}_i = \begin{cases} 1 & \text{if } h_i \text{ is true} \\ 0 & \text{if } h_i \text{ is false} \end{cases} \tag{4.13}$$

where $\mathbf{v}_i$ tells whether hypothesis h_i is true or not. The goal is to find the indicator vector with the highest average pairwise consistency λ through the relation

$$\lambda = \frac{\mathbf{v}^\top A \mathbf{v}}{\mathbf{v}^\top \mathbf{v}}. \tag{4.14}$$

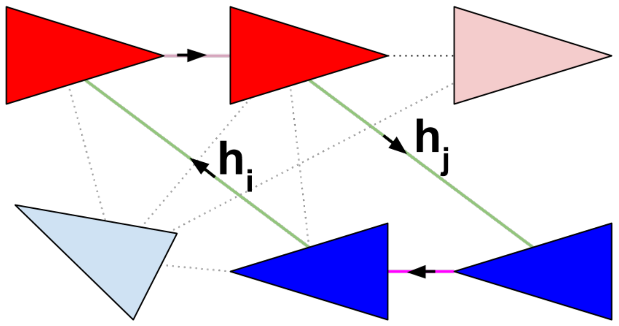


Figure 4.5: Drawing illustrating a set of hypotheses. A pair of hypotheses h_i and h_j are said to be consistent if the transformations around the loop returns to the origin.

where A consists of the elements

$$A_{ij} = e^{T^{\bowtie}\Omega_T (T^{\bowtie})^\top} \tag{4.15}$$

with the transformation T in pose form and information Ω_T around the loop of hypotheses. The $\bowtie$ symbol signify the change from transformation to pose representation as described in Section 3.2.

This is a NP-hard problem but can be treated as if continuous. By differentiating 4.14 and setting it equal to zero, the problem becomes equivalent to

$$A\mathbf{v} = \lambda\mathbf{v} \tag{4.16}$$

which is the characteristic equation. The dominant eigenvector $\mathbf{v}_1$ maximizes 4.14. If the fraction between the two largest eigenvalues is large, e.g. $\lambda_1/\lambda_2 > 2$, $\mathbf{v}_1$ is regarded as locally unambiguous. If no sufficiently dominant eigenvalue is found, more hypotheses should be collected until one eigenvalue is dominant. After discretizing $\mathbf{v} = \mathbf{v}_1$, the hypotheses marked as true are accepted as loop closures.

4.3.2 Back-End

The Back-End optimizes the poses and landmarks based on the information stored in the graph. This is done by finding the minimum of the least-squares objective function 4.12 by using a solver, typically Gauss-Newton. The error is the difference between the poses $\mathbf{p}$ and landmarks m in the state vector and the, from the measurements, observed poses $\hat{\mathbf{p}}$ and landmarks $\hat{m}$ through some sensors. However, since poses contain orientation, the nonlinearity makes it incorrect to simply subtract them. Instead, the transformation representation of poses comes in handy as the error $\mathbf{e}_{ij}$ can be defined through the transformation E_{ij} from the observed pose $\hat{\mathbf{p}}_j$ to the expected pose $\mathbf{p}_j$ within the same coordinate frame, as visualized in Figure 4.6. For the pose-GraphSLAM algorithm, the error can be defined as

$$\mathbf{e}_{ij} = \left(\hat{M}_{ij}^{-1} M_{ij}\right)^{\bowtie} = \left(\hat{M}_{ij}^{-1}(M_{oi}^{-1} M_{oj})\right)^{\bowtie} \tag{4.17}$$

where $\bowtie$ denote the change between pose and transformation as described in Section 3.2. Further expansion of the error yields

$$\mathbf{e}_{ij} = \begin{bmatrix} R_e & \mathbf{t}_e \\ \mathbf{0}^\top & 1 \end{bmatrix}^{\bowtie} = \begin{bmatrix} \hat{R}_{ij}^\top R_{oi}^\top R_{oj} & \hat{R}_{ij}^\top \left(R_{oi}^\top (\mathbf{t}_{oj} - \mathbf{t}_{oi}) - \hat{\mathbf{t}}_{ij} \right) \\ \mathbf{0}^\top & 1 \end{bmatrix}^{\bowtie} \tag{4.18}$$

when applying 3.11 and 3.15 to 4.17.

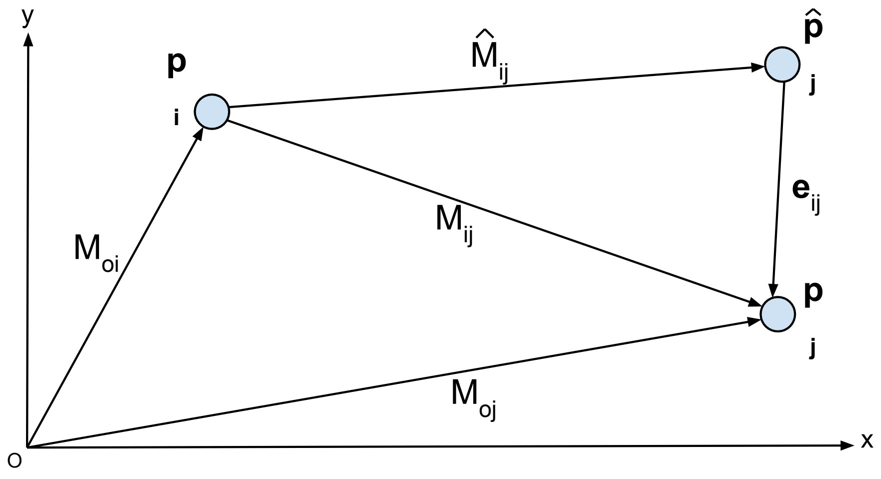


Figure 4.6: Illustration of the GraphSLAM error function. The arrows denote the direction of the transformations. M denotes transformations and $\mathbf{e}_{ij}$ the error in vector form.

The optimization of the graph is very sensitive to outliers, e.g. incorrect loop closures. These can contort the graph beyond recognition. It is therefore important to remove outliers before the optimization (Front-End) or compensate for them during the optimization (Back-End). For increased robustness, outliers should be handled in both the Front-End and Back-End.

Dynamic Covariance Scaling (DCS) scales the information of the edges based on the objective function [22]. This reduces the impact of large errors which can appear from bad constraints. The optimization may converge slower as all large errors are affected including those from good constraints. DCS computes a scaling factor as

$$s_k = \min\left(1, \frac{2\Phi}{\Phi + \mathcal{X}_k^2}\right) \tag{4.19}$$

where $\mathcal{X}_k^2 = \mathbf{e}_l^\top \Omega_k \mathbf{e_k}$ from 4.12 and Φ the upper bound for all robust constraints. Φ is effectively a tuning parameter. The scaling factor is one as long as the error is within bounds and decreases when outside as in Figure 4.7. The scaling factor is applied to the objective function according to

$$[\mathbf{p}^*, m^*] = \underset{\mathbf{p}, m}{\operatorname{argmin}} \sum_k \mathbf{e}_k^\top s^2 \Omega_k \mathbf{e}_k. \tag{4.20}$$

By scaling the information in the edges, the constraints are relaxed and make the problem less stiff.

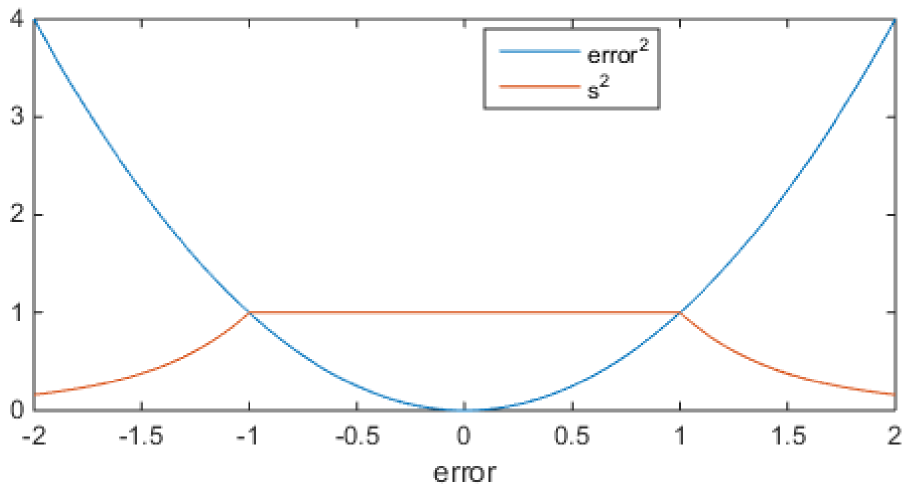


Figure 4.7: Dynamic Covariance Scaling of an error from a constraint. The scaling factor diminishes as the error increases beyond a specified limit.

Gauss-Newton is a commonly used solver for nonlinear least-squares problems an can be applied. To find the Jacobian of the error, the nonlinear error function 4.18 can be approximated through a first order Taylor series expansion around the state vector $\mathbf{p}$ as

$$\mathbf{e}_{ij}(\mathbf{p}) \approx \mathbf{e}_{ij}(\mathbf{p}) + J_{ij}\Delta\mathbf{p} \tag{4.21}$$

with the Jacobian J_{ij}. Since the error is only relying on the two poses $\mathbf{p_i}$ and $\mathbf{p}_j$, the Jacobian can be simplified to

$$J_{ij} = \frac{\partial \mathbf{e}_{ij}(\mathbf{p})}{\partial \mathbf{p}} = \left[\mathbf{0} \;\cdots\; \underbrace{\frac{\partial \mathbf{e}_{ij}(\mathbf{p}_i)}{\partial \mathbf{p}_i}}_{A_{ij}} \;\cdots\; \underbrace{\frac{\partial \mathbf{e}_{ij}(\mathbf{p}_j)}{\partial \mathbf{p}_j}}_{B_{ij}} \;\cdots\; \mathbf{0}\right] \tag{4.22}$$

and the poses evaluated individually. The linear system in 3.26 is then constructed as

$$H\Delta\mathbf{p} = -\mathbf{b} \tag{4.23}$$

$$\mathbf{p} = \mathbf{p} + \Delta\mathbf{p} \tag{4.24}$$

where the coefficient vector b and matrix H are computed according to

$$\mathbf{b}^\top = \sum_{ij} \mathbf{b}_{ij}^\top = \sum_{ij} \mathbf{e}_{ij}^\top \Omega_{ij} J_{ij} \tag{4.25}$$

$$H = \sum_{ij} H_{ij} = \sum_{ij} J_{ij}^\top \Omega_{ij} J_{ij} \tag{4.26}$$

where H is sparse due to the structure of J_{ij}. The system is then solved for $\Delta\mathbf{p}$ with a linear solver such as SVD which determines the step length and direction.

5
Datasets

In this chapter, an overview of the used datasets is presented. None of these datasets were gathered specifically for this thesis and often for slightly different purposes than evaluating LIDAR based SLAM. Nevertheless, they were useful for the project and by using different parts of the datasets different aspects of the problem could be analyzed. Besides the datasets presented here the Ford Campus Dataset [23] was also used to test loop closures. However as the Kitti Vision Benchmark Suite dataset contained a longer drive with several loop closures the datasets and results from the dataset are not presented here. The sensors that were used are also described in this chapter.

5.1 KITTI Vision Benchmark Suite

The KITTI Vision Benchmark Suite is a public computer vision and robotic algorithm benchmarking dataset, which was done in collaboration between Karlsruhe Institute of Technology and Toyota Technological Institute at Chicago. While the main intention for including the LIDAR data was to provide accurate ground truth for camera vision algorithms, the LIDAR data proved useful for the algorithms implemented in this thesis. The datasets that were used contained medium density residential areas in Karlsruhe with pedestrians and moving vehicles. [24]

5.2 Volvo Dataset

Since this thesis was done in collaboration with Volvo Cars, access to two proprietary datasets from Volvo Cars was granted during the thesis. These datsets were collected as a part of Volvo Drive Me project, which intends to put 100 self-driving cars on public roads in Gothenburg Sweden starting 2017 [25]. The first of these datasets include short drives high-density residential areas within Gothenburg. The other set include high way driving in a full circle around Gothenburg including a longer tunnel.

5.3 Sensors

The KITTI and Volvo Cars datasets were gathered with vehicles equipped with the inertial navigation system OXTS RT 3003 and the LIDAR Velodyne HDL-64E. Both these are high quality sensors with high a commercial prices which might make them infeasible for large-scale commercial production. A brief presentation of these sensors are given below.

5.3.1 Velodyne HDL-64E

The Velodyne HDL-64E is a high end LIDAR sensor designed for obstacle detection and navigation of autonomous vehicles [26]. It operates by measuring distance with 64 layers of lasers, which by spinning around provides a complete 360° horizontal field of view. The horizontal field of view is 26.8°. Horizontal angles are measured to a 0.09° angular resolution and the distance has an accuracy of less than 2cm. The system is capable of delivering 1.3 million points per second at a frame rate between 5 and 20Hz. The detection range depends on reflectivity of the surface. Pavement can generally be detected at 50m range and cars at around 120m distance.

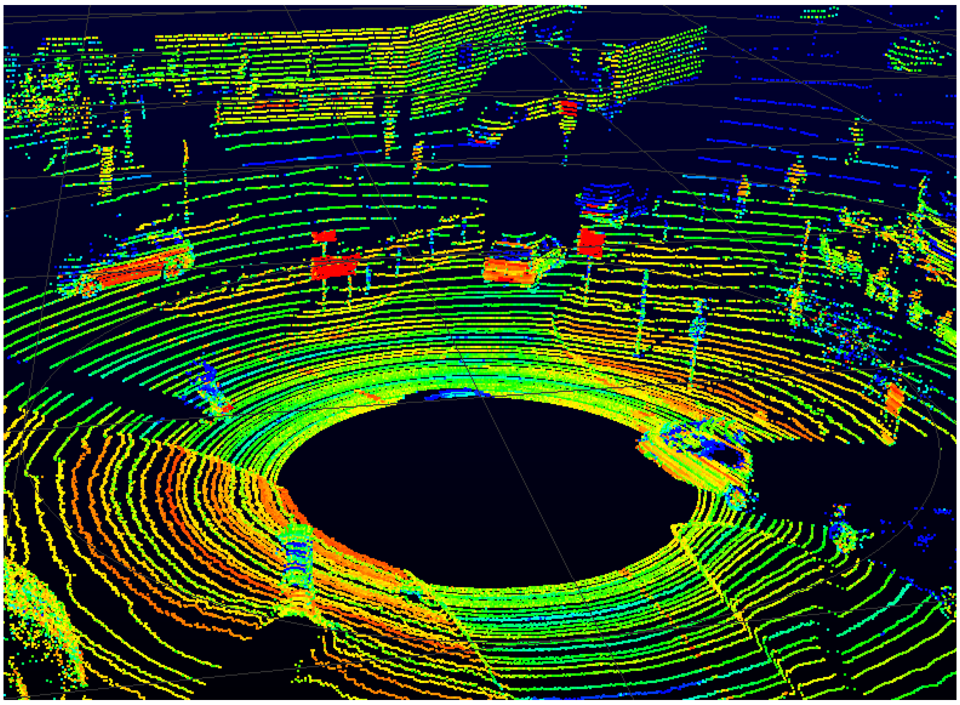


Figure 5.1: An example of a Velodyne HDL-64E point cloud. Colors indicate the reflectivity of each point. A point is blue if very little is reflected back, or red if almost all is reflected back. The point cloud is from the Volvo Cars Dataset and was displayed using the open source software VeloView.

In the datasets used in this thesis, the spin rate was set to 10Hz for both the KITTI Vision Benchmark Suite and Volvo Cars Dataset. This correspond to each point cloud consisting of around 130 000 points. Each point has an xyz-coordinate relative to the LIDAR and a reflectivity value related to the infrared reflectance of the surface of the point.

5.3.2 OXTS RT 3003

The OXTS RT3003 is an advanced precision Inertial and GPS Navigation system for measuring motion, position and orientation [27]. By combining Inertial Navigation System(INS) with GPS receivers it improves performance over GPS-only readings in urban and tree-covered environments. It has a Velocity accuracy (RMS) of 0.05km/h, roll/pitch accuracy(1σ) of 0.03° and heading accuracy(1σ) of 0.1°. GPS position RMS using Real Time Kinematic (RTK) with L1/L2 is 0.02m.

6 Implementation

To investigate which challenges exist in using LIDAR sensors for SLAM for vehicle localization, a complete solution was implemented. The goal of the implementation was to:

1. Find out where the vehicle has been
2. Create a map from the data
3. Localize a vehicle within the map using different data

Three different registration algorithms were implemented for finding the ego motion and for recognizing previous visited places. The available SLAM methods were compared and one was selected to be implemented. In addition a basic localization algorithm was implemented. Here motivations behind the choice of algorithms are presented with explanations of how they were implemented.

6.1 Choice of SLAM Algorithm

After having done a literature study, three main categories of SLAM algorithms were identified, the EKFSLAM, FastSLAM and GraphSLAM algorithm. These algorithms were compared to see which algorithm was most suitable to be used for the implementation. A comparison of the algorithms can be seen in Table 6.1.

A short note on the choice of parallelizability as a criterion in Table 6.1. Graphics Processor Units (GPUs) have gotten very powerful in recent years and GPUs designed for sensor fusion in vehicles have started to become popular. An example of this is the NVIDIA Tegra Drive PX, which is a GPU designed for use in vehicles [28]. Since GPUs main advantage is in massive parallel computations, it could be of interest to see which algorithms could gain more from having access to powerful parallel computational power. EKFSLAM's main computational bottleneck is the matrix inversion, which is hard to parallelize. FastSLAM has a re-sampling step, which is hard to parallelize but might not be needed to be performed very often. The main bottleneck of GraphSLAM is the error linearization, which is parallelizable.

Since the vehicles will be driving in large environments over long time it was decided to use GraphSLAM, since the algorithm scales well and is robust over long drives. Further, only pose nodes are used to reduce the size of the graph along with the optimization complexity.

Table 6.1: Comparison of the three main categories of SLAM algorithms. In the complexity row, n is the number of landmarks, k the number of particles and e is the number of edges. [5]

	EKFSLAM	FastSLAM	GraphSLAM
Complexity	$O(n^2)$	$O(k * logn)$	$O(e)$
Distribution	Gaussian	Any	Gaussian+ outlier rejection
Linearization	Once	Not needed	Re-linearize
Flexibility	0	+	++
Large scale	-	+	++
Parallelizability	-	+	++
Pros	Easy to implement well known	Can use negative information	Scales well robust
Cons	Can not handle large maps	Hard to recover, need many particles to be robust	Harder to implement

6.2 Registration

Three algorithms for registration were implemented, ICP, GICP and a feature-based registration algorithm. A description of the algorithms is presented here.

6.2.1 ICP Algorithm

An ICP algorithm has been implemented by Jakob Wilm [29] for Matlab. This algorithm was augmented to include a prior from e.g. odometry. Another extension was the option to lock degrees of freedom (DoF) in the transformation by excluding them from the optimization. With this extension, the algorithm can align point clouds longitudinal while ignoring the lateral DoF. The lateral DoF is instead supplied through a prior. This could be useful in for example tunnels where the walls are symmetric and lateral alignment is hard. An example of this can be seen in Figure 6.1.

6.2.2 GICP Algorithm

A GICP algorithm was implemented as a program with the use of the open source Point Cloud Library (PCL) [30] in C++. The program was called from Matlab through the operating system console. It required point clouds to be converted to the Point Cloud Data (PCD) format. The PCD filenames of the point clouds to align was supplied to the program and the resulting transformation matrix was returned. The option to supply a prior transformation was also implemented.

A down-sampling program was also implemented to reduce the size of the PCD files. Smaller files decrease the time to load said files. The downs-sampling program makes use of the voxel grid filter in PCL and requires a PCD file to operate on. The filter approximates all points within a voxel with its centroid. This preserves

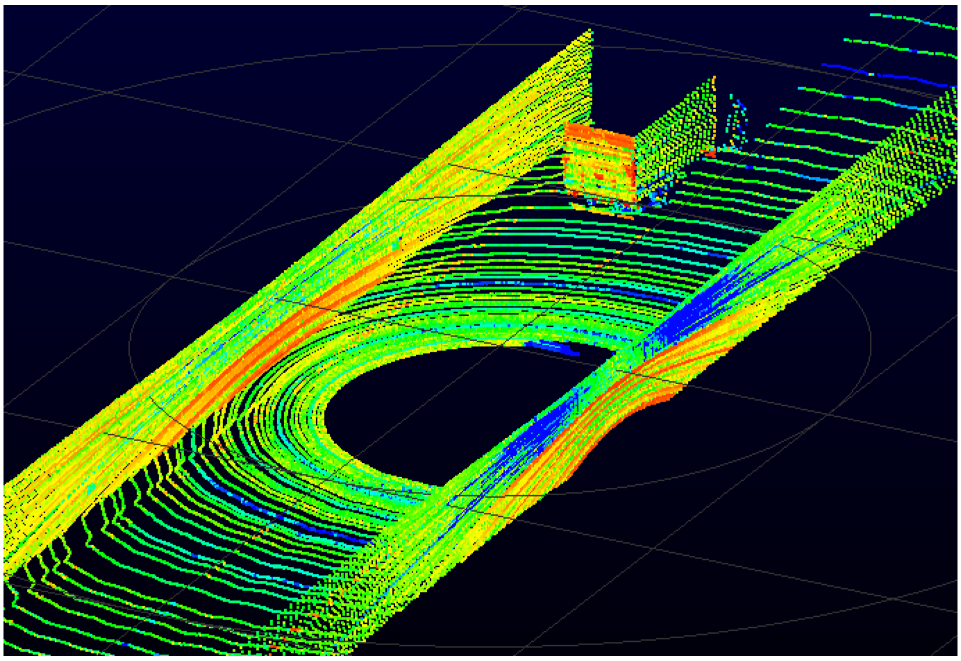


Figure 6.1: An example of a feature-poor environment where the feature matching algorithm struggles. ICP will have a hard time estimating the correct lateral transformation. The scan was from taken from the Volvo Cars Dataset from the Lundby tunnel in Gothenburg.

the shape of the point cloud more accurately than using the center of the voxel to represent the points.

6.2.3 Feature-Based Registration

Implementations of 3D based feature registration using the algorithms NARF, SHOT and FPFH were tried on the data using code from tutorials in PCL. Though the implemented methods seemed to work well for some smaller point clouds, the implemented methods where unable to perform well on the data in the datasets. As time was limited it was decided to not investigate these algorithms further.

Instead a 2D based feature extraction algorithm was implemented based on the work of Y.Li and E.Olsen [31]. Slight modifications were made in the rendering of the image where only the height was considered rather than the difference in height. Another difference was that the BRISK feature detector was used instead of the Kanade-Tomasi method. It was found that the algorithm performed better with these choices but, as this was not the main focus of this paper, no extensive work was done to analyze potential improvements with the implementation of the original choices.

In Algorithm 7 presents a short summary of the implemented feature extraction algorithm. The points in the point cloud are scaled to a pixel grid along the xy-plane, lines 3-4. The z-coordinates are moved in relation to the ground and assigned to the corresponding grid tile. As shown in line 5, only the highest point is saved in each grid tile. This produces a projected height map of the scan. The height then

Algorithm 7 Feature Extraction

```
Require: Point cloud: $PC = \{x_i, y_i, z_i\}$
 1: for all points $i$ do
 2:     $i = \text{floor}(\text{rescale}(x)) + 1$
 3:     $j = \text{floor}(\text{rescale}(y)) + 1$
 4:     $A_{ij} = \max(A_{ij}, z - \text{sensorHeight}())$
 5: end for
 6: $img = \text{matrix2grayscale}(A)$
 7: $img = \text{gaussianKernel}(img)$
 8: $keypoints = \text{KeypointExtractor}(img)$
 9: $descriptions = \text{DescriptionExtractor}(img, keypoints)$
10: return $keypoints, descriptions$
```

converted to a gray scale image, line 6, and smoothed with a Gaussian kernel, i.e. a Gaussian function. In line 8-9, the keypoints and descriptors are extracted through an algorithm like BRISK.

To select correct correspondences the transformations are filtered, using the outlier detection algorithm RANSAC, and grouping the correspondence pairs into inliers and outliers. The inliers are assumed to be the correct matches between features and the outliers are ignored. If the number of inliers is deemed to be too small, the matching is classified as a failure otherwise are the inlier keypoints used to calculate the least-square optimal rigid transformation matrix M as in Section 3.3.1. The number of inliers is used to scale the information of the edges created through the feature matching. An illustration of the process is shown in Figure 6.2.

A function to reduce the number of features was implemented. The function removes the unstable features and keeps the stable features. Stable features are defined as the features that can be observed in temporally consecutive scans. This results in a reduction in size and a sped comparison between sets of features when generating loop closure hypotheses. Matched pairs are then compared and outlier pairs removed. Only features that had at least one inliers with neighboring scans are saved, the other features are deemed to be unstable and are removed. An option for how many temporal steps in each direction to be compared is available, by default it was set to two to allow all scans to overlap.

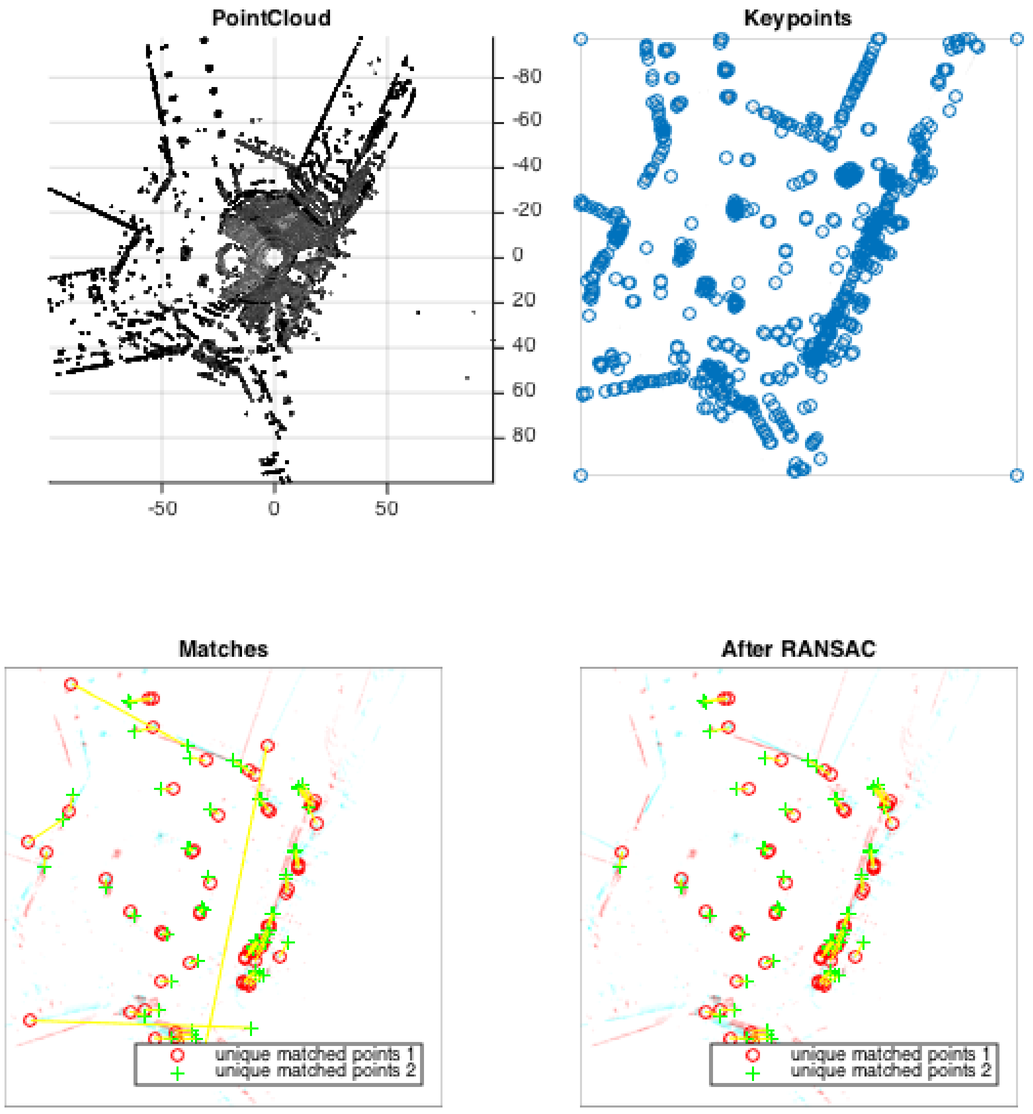


Figure 6.2: FeatureThe feature-based registration process is illustrated in four steps. In the top left plot shows the original point cloud. The top right shows the extracted keypoints. In the down left plot, matching keypoint pairs are show. As can be seen there are some outliers from incorrect matches but most matches are correct. In the down right plot the outliers have been removed using RANSAC. As can be seen the vehicle has rotated between the scans. The scans were taken from the Volvo Cars Dataset driving through a roundabout in a high-density residential area.

6.3 GraphSLAM Algorithm

A GraphSLAM algorithm was implemented for the data in the datasets. Since the datasets were collected in advance, the algorithm was not made to run in a real time application.

An overview of the algorithm is presented in Algorithm 8. It requires a set of data to loop over. The graph g is initialized before the loop with no nodes nor edges. In line 3, the Front-End builds the graph and finds loop closure hypotheses. A robust loop closure algorithm was implemented in line 4, to sort out the bad hypotheses from the graph. The graph is then optimized in line 5 by the Back-End. The functions after Front-End does not have to run in every cycle of the loop and can depend on flags raised by the Front-End or in fixed intervals.

Algorithm 8 GraphSLAM

```
Require: $data = \{data_i\}$
1: initialize graph $g$
2: for all $data_i$ do
3:     $g$ = Front-End($g, data_i$)
4:     $g$ = RobustLoopClosure($g$)
5:     $g$ = Back-End($g$)
6: end for
7: return $g$
```

6.3.1 Front-End

The implemented Front-End processes the data of each time step separately, and adds nodes and edges to the graph based on the data in the current loop cycle. The algorithm is split into two parts, creating nodes and edges for moving forward and loop closure detection. An overview of the Front-End algorithm is presented in 9.

The first part of the algorithm is responsible for creating the nodes and the edges between consecutive nodes. To avoid unnecessary large graphs, it first performs a check in line 1, to see whether the vehicle is moving or not. In the case of the vehicle standing still, the graph is returned without any alterations. If the vehicle is moving, a new node is created and added to the graph. A GPS edge is created between the origin and the current node $newNode$ based on the data in lines 4-5. Another edge between the current node $newNode$ and the previous node is created with a transformation M and added to the graph in lines 6-8. The transformation M is from registration methods such as ICP or feature-matching but can also be from odometry.

The second part of the algorithm finds and adds loop closures to the graph. In line 9 it first finds nodes in the graph close to the current node with two conditions. The nodes must to be within a specified distance from the current GPS signal and have a driven distance from the current node above a specified threshold. The specified driven distance should be larger than the specified GPS distance to avoid loop closures between temporally close nodes. This returns a set of nodes to search for

Algorithm 9 Front-End

```
Require: $g, data_i$
 1: if isMoving($data_i$) = $true$ then
 2:     $newNode$ = createNode($data_i$)
 3:     $g$ = add2graph($g, newNode$)
 4:     $edge\_GPS$ = createEdge($data_i, newNode$)
 5:     $g$ = add2graph($g, edge\_GPS$)

 6:     $M$ = registration($newNode$, previousNode($g$))
 7:     $edge$ = createEdge($M, newNode$, previousNode($g$))
 8:     $g$ = add2graph($g, edge$)

 9:     $search\_nodes$ = findLCNodes($g, newNode$)
10:     for all $node \in search\_nodes$ do
11:         $success, M$ = featureMatching($newNode, node$)
12:         if $success = true$ then
13:             $LC\_edge$ = createEdge($M, newNode, node$)
14:             $g$ = add2graph($g, LC\_edge$)
15:         end if
16:     end for
17: end if
18: return $g$
```

loop closures. To reduce the potentially large set, the set is sorted by GPS distance and a subset selected for loop closure detection.

The loop closure detection is performed in lines 10-16. For each node in this subset of nodes, a transformation is estimated from the current node. The transformation is then verified through a few conditions to avoid ill-formed loop closures. If the feature-matching fails no loop closure is generated. If the feature-matching algorithm was successful, a loop closure edge is created and added to the graph.

A robust loop closure algorithm was implemented based on Olson's proposal described in Section 4.3.1. The algorithm attempts to remove ill-formed loop closures from the graph, by constructing a pairwise consistency matrix A from a set of loop closures. The loop closures are treated as hypotheses to be tested and the search algorithm A* was used to find the shortest set of links between the pairs of hypotheses. These links was then used to build the consistency matrix. The hypotheses that fail the test are removed from the graph.

A slight modification was done to equation 4.15 in Olson's proposal. A coefficient of -0.5 was multiplied with the exponent to better reflect a Gaussian probability density function

$$A_{ij} = e^{-\frac{1}{2}\mathbf{T}^{\bowtie}\Omega(T^{\bowtie})^{\top}} \tag{6.1}$$

where the $\bowtie$ signify the change from transformation to pose representation as described in Section 3.2.

Through email correspondence, E.Olson agreed that this better mimics the probability density function but states that in the end this is more of an engineering problem and that experimentation generally is more useful.

The information Ω_T was set to identity, as this greatly simplified the problem and the implemented method for calculating the information in edges was untested. From the consistency matrix, the dominant eigenvector v is calculated and discretized as described in 4.3.1. The discretization of implemented through the following expression:

$$\mathbf{v}_i = \begin{cases} 1 & \text{if } \mathbf{v}_i \leq 0.9 \\ 0 & \text{if } \mathbf{v}_i < 0.9 \end{cases} \tag{6.2}$$

6.3.2 Back-End

The implemented Back-End finds the state vector that minimizes the errors from the constraints as explained in Section 4.3.2 with the Gauss-Newton method. The types of edges that should be optimized can be selected though an optional argument. For example whether the edges from the GPS should be accounted for in the optimization or not.

Algorithm 10 Back-End

```
Require: $g$
 1: while not Converged do
 2:     Initialize $H, \mathbf{b}$
 3:     for all $edges$ do
 4:         $[\mathbf{e}, A, B] = \text{linearize}(edge)$
 5:         $s = \text{DCS}(\mathbf{e}, \text{getInformation}(edge))$
 6:         $[H_{ij}, \mathbf{b}_{ij}] = [H_{ij}, \mathbf{b}_{ij}] + \text{buildLinearSystem}(\mathbf{e}, A, B, s)$
 7:     end for
 8:     $\Delta\mathbf{p} = \text{solveLinear}(H, -b)$
 9:     $\mathbf{p} = \mathbf{p} + \Delta\mathbf{p}$
10: end while
11: return $g$
```

An overview of the Back-End algorithm is presented in Algorithm 10. The algorithm loops over all edges and builds a linear system that can be solved for the step direction. The coefficient matrix H and vector $\mathbf{b}$ for the entire graph are initialized inside the loop, as they have to be recalculated for each iteration of the optimization. The linearization is performed in line 4 where the error $\mathbf{e}$ and the Jacobians A and B in 4.22 are calculated. Scaling factors are calculated according to 4.19 in line 5. The components of the coefficient matrix and vector are calculated in line 6 and summed together over the loop. Once the for-loop is complete, the linear system is solved for the step direction and the state vector is updated in lines 8 and 9. This sequence of operations is then repeated until a convergence criterion is reached or a maximum number of iterations is reached.

6.3.3 Pre-processing

To speed up the algorithm and simplify the code, a data pre-processing step was implemented. The script is essentially the time expensive steps of the algorithm performed separate of the simulation to save time when running the algorithm. The script performs the following tasks:

- Convert timestamps to the same time format, in this case, UNIX.
- Rotate point clouds so that the driving direction is along the x-axis.
- Convert point clouds to binary files for smaller file size and shorter loading times.
- Down-sample point clouds and store in PCD format for the GICP registration algorithm.
- Interpolate inertial and GPS measurements to LIDAR sampling frequency.
- Convert GPS reading to xyz coordinates in a local reference frame.
- Extract keypoints and descriptors from the point clouds.
- Remove unstable features.

6.3.4 Cartography

A couple of different methods for producing maps were implemented. These operate based on an optimized graph and the information stored in the nodes to generate maps for different purposes. A brief description of each method and their usage is presented here.

The first method *Graph2PointCloud* generates one large point cloud from all the scans in a drive. Each scan is transformed from the origin to the state of the node it was captured in. An option to only use every other scan was implemented to reduce the amount of points and make it manageable. This type of map could be useful for many purposes such as creating a topological map, detect road surfaces or visualization purposes. Since the map contain a lot of information, it requires plenty of data storage and computations based on it might take time.

The second method *Graph2PointCloudXY* is a variation of *Graph2PointCloud*. Instead of using all point clouds, only point clouds with points within a specified area are used. These clouds are selected by finding the nodes within a desired radius of the area's centroid. These clouds are then aligned and reduced through a pass through filter of the area. This map could be of interest for generating bitmaps of the ground though infra-red reflectivity, as proposed for localization by J.Levison [32] or if only a certain area of the map i of interest.

The last method *Graph2LocalizationMap* reduces a pose-graph into a map of landmarks along a path. A landmark contains a pose and features from the scan. Thus, localization is performed by registration between the current pose and the landmark in the map. The landmarks are selected based on their distance from the path. A fixed radius between the landmarks can be specified as an optional, which would further reduce the data size of the map. This map could be used for localization purposes in autonomous navigation.

6.3.5 Localization

An algorithm was implemented for localization within the map consisting of reduced pose-nodes. The localization was done by using a particle filter, as described in section 3.1.2, where each particle is a hypothesis of the vehicle pose. In the motion update, each particle is updated with odometry measurements or with feature-matching between the current pose and the previous pose, with added Gaussian noise. For the measurement update, a feature match is attempted between the features from the current scan and the features stored in the closest map node according to the GPS measurement of the current scan.

7
Results

In this chapter results, from using the implemented algorithms on the datasets, is be presented. The evaluated algorithms were for registration, for SLAM and for localization. Whenever possible the results will be quantified with numbers but when no ground truth exists to compare the results the results are illustrated with figures.

7.1 Registration

Three different registration algorithms were implemented. The methods were

1. ICP with Prior in Matlab
2. GICP using the Point Cloud Library (PCL)
3. Feature-based matching algorithm

LIDAR data from a drive in high-density residential area from the Volvo Cars Dataset was selected to be used with GPS, measurements as ground truth. The OXTS RT3003 sensors [27] has a velocity and a rotational velocity RMS error of

$$\begin{aligned}\sigma_{v,GPS} &= 0.05km/h \approx 0.014m/s\\ \sigma_{\omega,GPS} &= 0.02^\circ/s\end{aligned}$$

.

As the LIDAR had a scan rate of $10Hz = 0.1s$ which, the ground truth has a translational and a rotational RMS error of

$$\begin{aligned}\sigma_{tr,GPS} &= 0.014m/s * 0.1s = 0.0014m\\ \sigma_{rot,GPS} &= 0.02^\circ/s * 0.1s = 0.002^\circ\end{aligned}$$

Each algorithm tries to estimate the rigid body transformation between scans which consists of a translation and a rotation. The estimation is compared to the ground truth from the GPS.

7.1.1 ICP

The implemented ICP algorithm was evaluated. Since the algorithm was slow and prone to converge to local minima, a prior was used. The prior was provided by the feature-based registration algorithm since it is fast and work differently from ICP. In the test, 15 iterations was used which along with a decent prior is sufficient for convergence if possible. Results for ICP registration are shown in Figure 7.1 with the errors visualized in Figure 7.2. The mean and standard deviation are shown in Table 7.1.

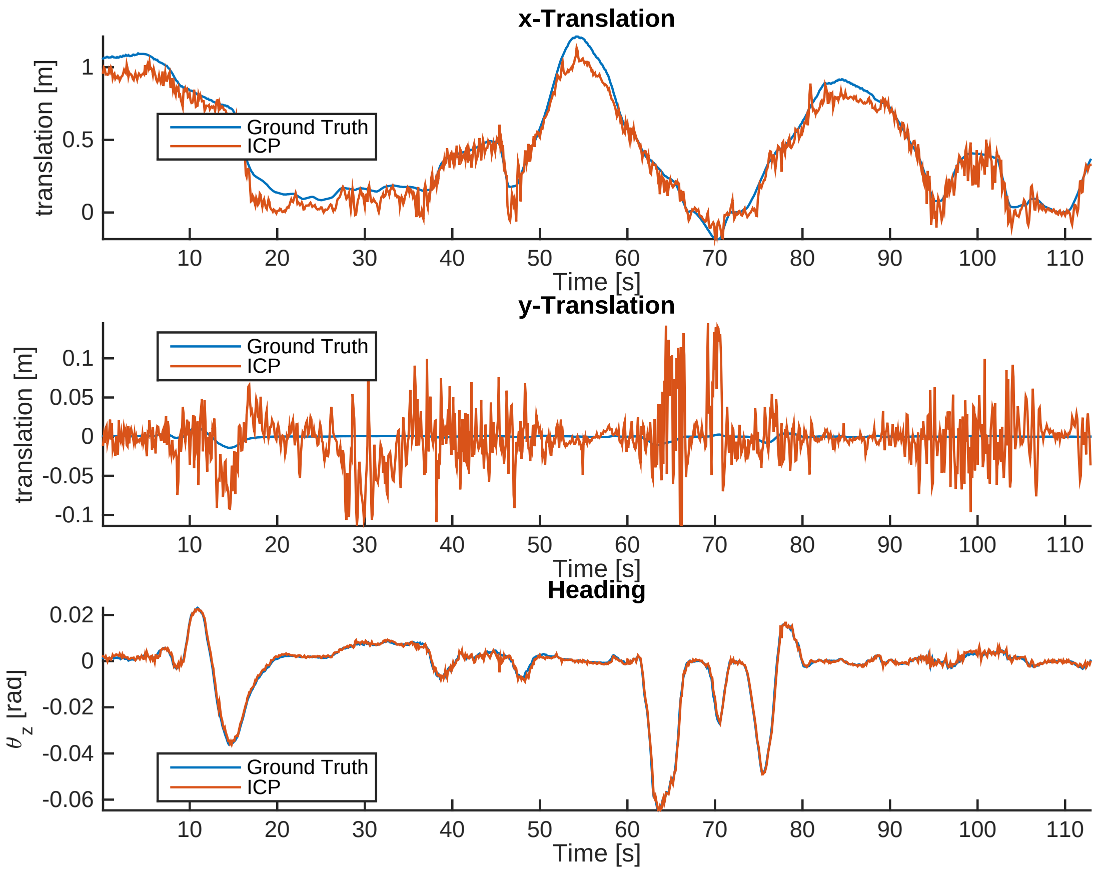


Figure 7.1: ICP registration. In the top plot shows the translation in lateral direct. A value of 1m here implies that the vehicle is moving 1m in the 0.1s time between the scans and thus has a momentary velocity of 10m/s. The middle plot similarly shows the longitudinal translation. The bottom plot shows the rotation around the z-axis, which is how the heading of the vehicle is changing in radians. A value of 0.02rad here implies that the vehicle is translating 0.02rad in 0.1s or has an angular velocity of 0.2rad/s.

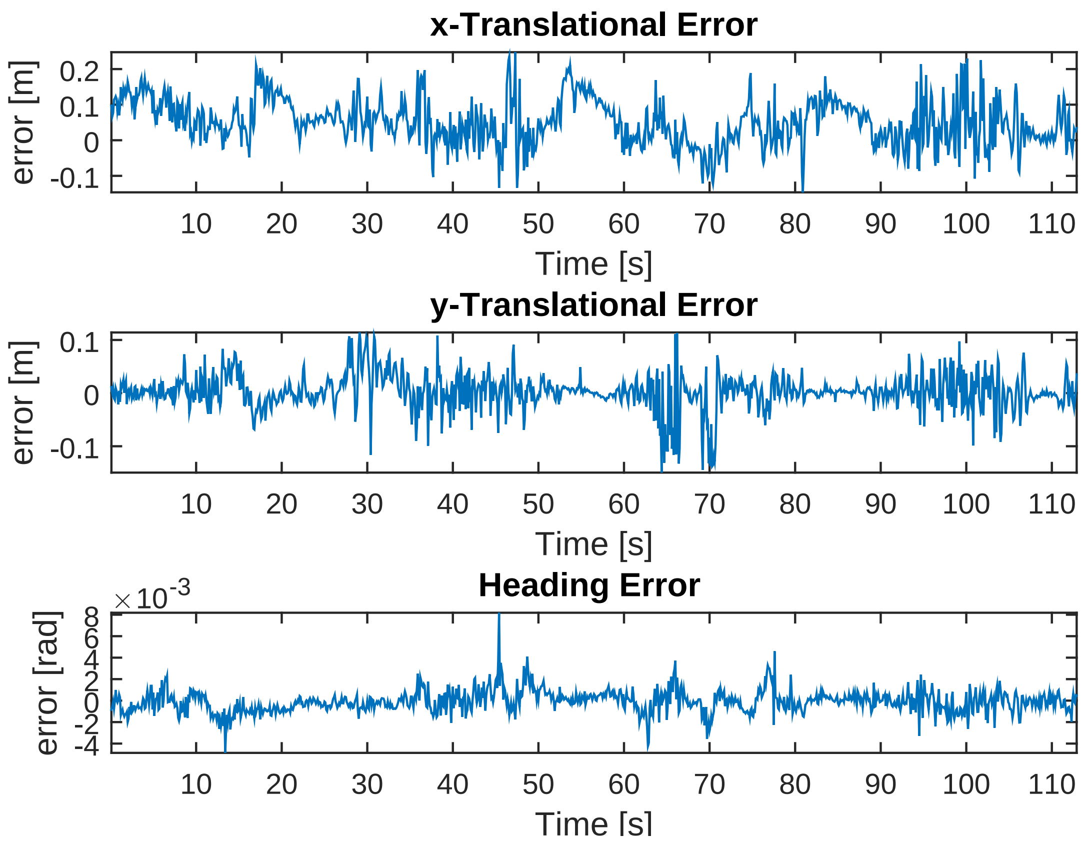


Figure 7.2: Error of ICP registration. The first plot shows the error of the estimation in lateral direction from the feature-based registration algorithm. The second plot shows the longitudinal translation estimation error. The third plot shows the error in rotation around the z-axis. The forth plot shows the pitch measured by the IMU.

Table 7.1: Mean and standard deviation of the error for the ICP algorithm

ICP-error	μ	σ
x [m]	0.051	0.064
y [m]	0.0024	0.034
heading [°]	-0.0049	0.059

7.1.2 GICP

A similar analysis was done on the implemented GICP algorithm. The algorithm was run with down-sampled point clouds. Results for ICP registration are shown in Figure 7.3 with the errors visualized in Figure 7.4.

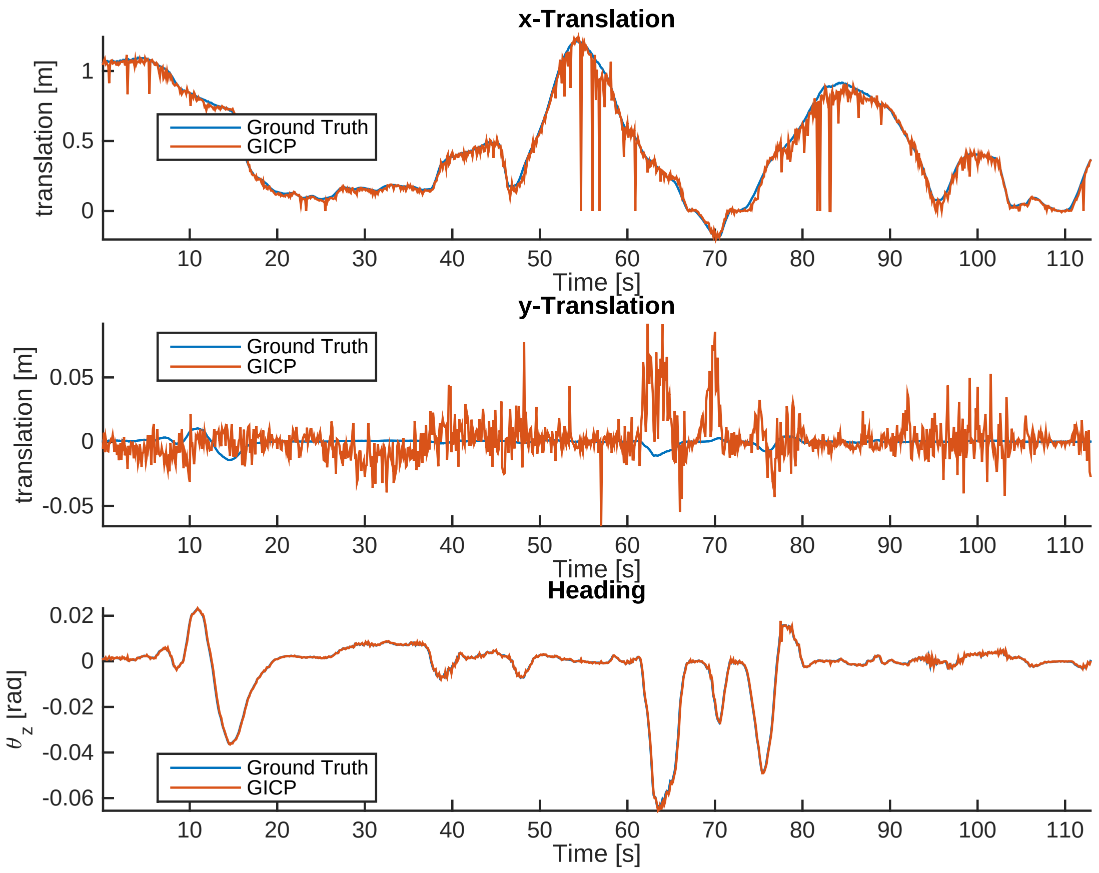


Figure 7.3: GICP registration. In the top plot the translation in lateral direct is shown. A value of 1m here implies that the vehicle is moving 1m in the 0.1s time between the scans and thus has a momentary velocity of 10m/s. The middle plot similarly shows the longitudinal translation. The bottom plot shows the rotation around the z-axis, which is how the heading of the vehicle is changing in radians. A value of 0.02rad here implies that the vehicle is translating 0.02rad in 0.1s or has an angular velocity of 0.2rad/s.

The error of the GICP registration can be seen in Figure 7.2. As can be seen in the figure, GICP sometimes fail to correctly estimate the longitudinal translation. This can occur when objects, which are detected by many points, are moving between scans, e.g. large trucks. The mean and standard deviation of the error for GICP can be seen in Table 7.2.

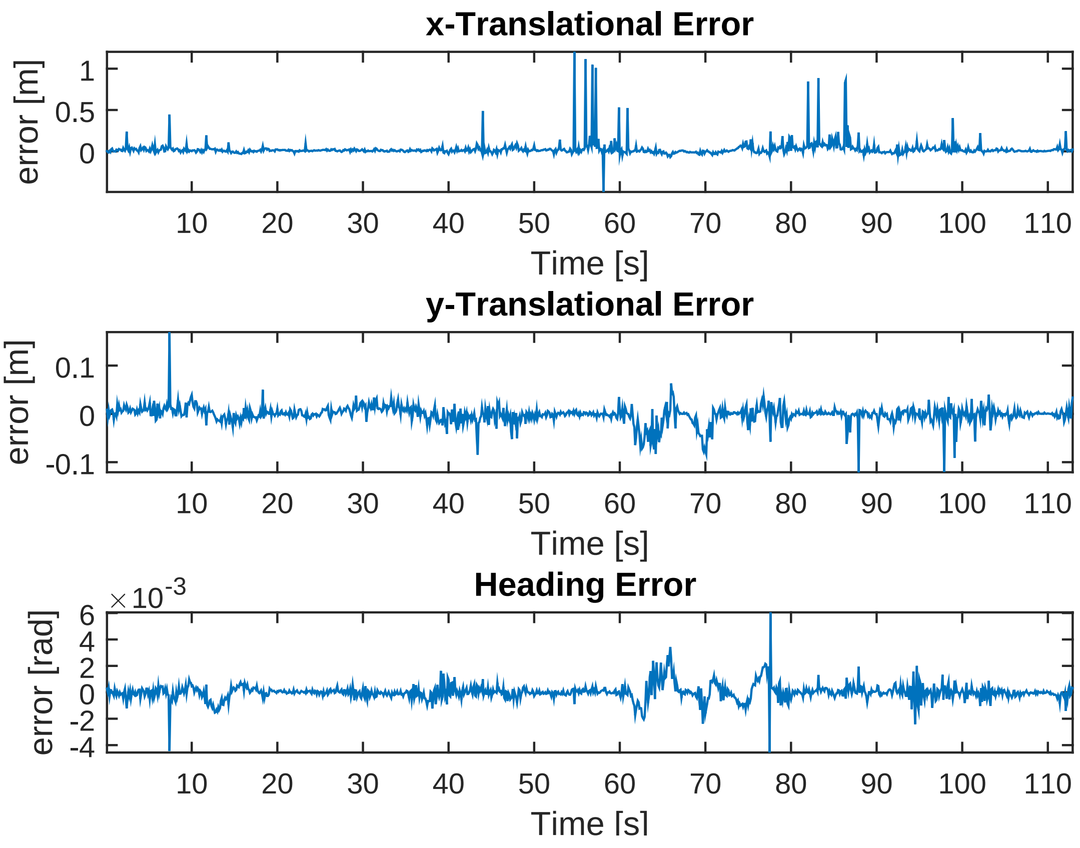


Figure 7.4: Error of GICP registration. The first plot shows the error of the estimation in lateral direction for the GICP registration algorithm. The second plot shows the longitudinal translation estimation error. The third plot shows the error in rotation around the z-axis. The forth plot shows the pitch measured by the IMU.

Table 7.2: Mean and standard deviation of the error for the GICP algorithm

GICP-error	μ	σ
x [m]	0.023	0.086
y [m]	-0.0015	0.017
heading [°]	$2.7 \cdot 10^{-6}$	0.034

7.1.3 Feature-Based Registration

Results for the implemented feature-based registration algorithm can be seen in Figure 7.5. As can be seen in the figure, the algorithm is stable and gives a good approximation of the rigid body transformation.

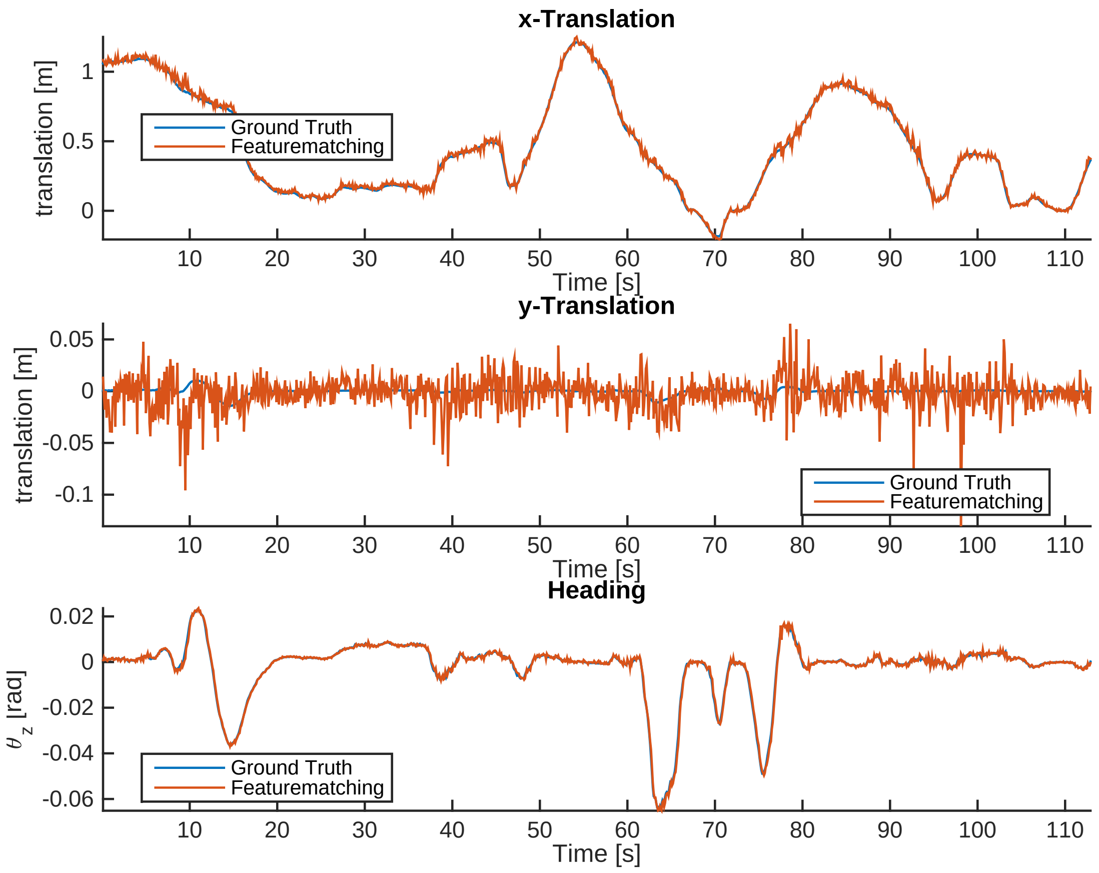


Figure 7.5: feature-based registration. The top plot shows the translation in lateral direct. A value of 1m here implies that the vehicle is moving 1m in the 0.1s time between the scans and thus has a momentary velocity of 10m/s. The middle plot similarly shows the longitudinal translation. The bottom plot shows the rotation around the z-axis, which is how the heading of the vehicle is changing in radians. A value of 0.02rad here implies that the vehicle is translating 0.02rad in 0.1s or has an angular velocity of 0.2rad/s.

To see which factors influence the error, the number of inliers and the pitch of the vehicle was analyzed and compared with the error. As can been seen in Figure 7.6 the translational error is larger when the scans are taken from different pitches like driving over speed bumps. The covariance and standard deviation of the error from the feature-based registration can be seen in Table 7.3

Table 7.3: Mean and standard deviation of the error for the feature-based registration algorithm

feature matching-error	μ	σ
x [m]	-0.0084	0.018
y [m]	0.0023	0.016
heading [°]	$6.0 \cdot 10^{-6}$	0.049

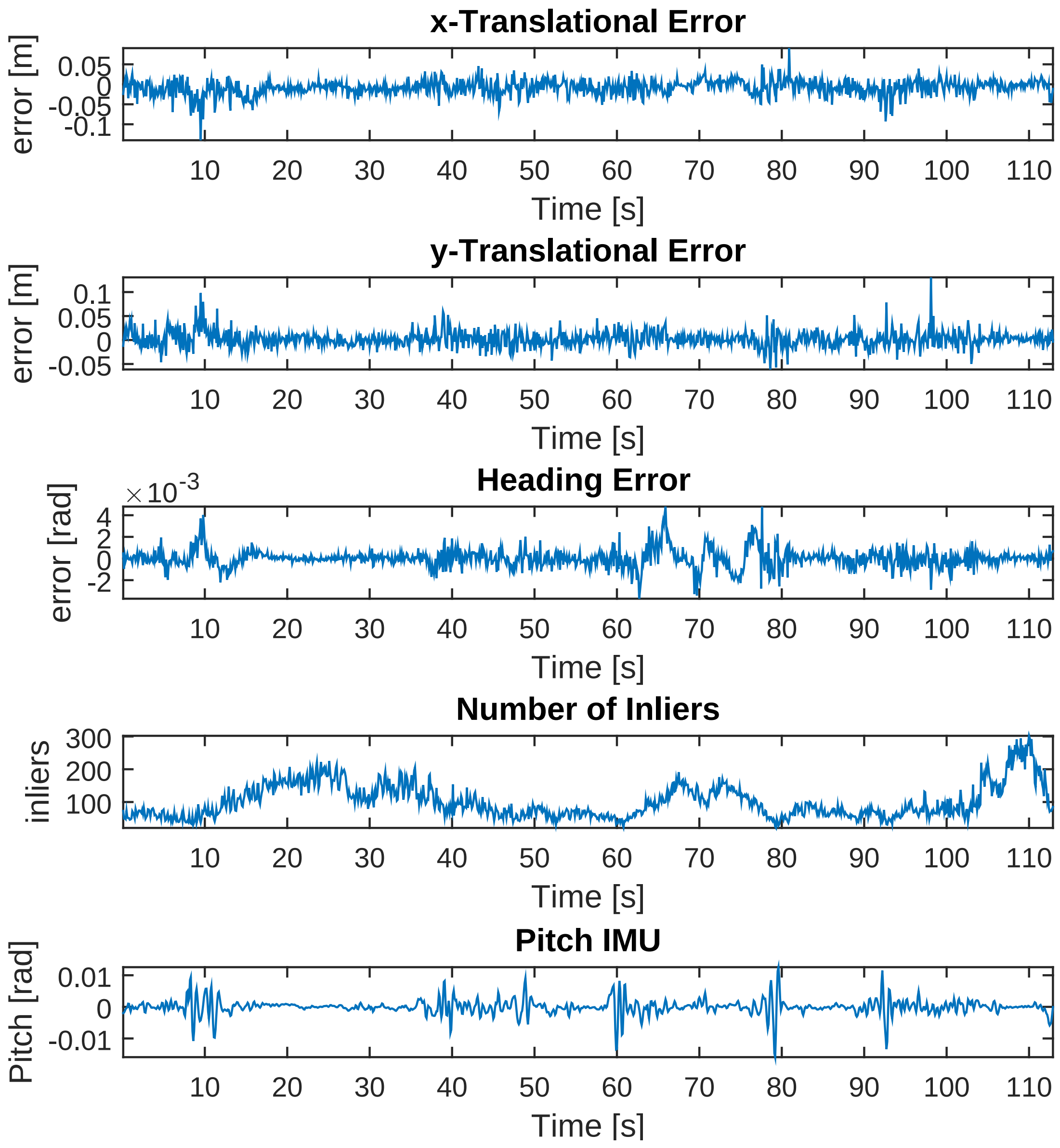


Figure 7.6: Error of feature-based registration. The first plot shows the error of the estimation in lateral direction from the feature-based registration algorithm. The second plot shows the longitudinal translation estimation error. The third plot shows the error in rotation around the z-axis. To see how if the error correlates with something, the number of inliers from the feature-based registration is shown in the forth plot and the pitch from the IMU is shown in the fifth plot. As can be seen the error is generally larger when the pitch is changing.

7.1.4 Execution Time

Execution time was compared between the algorithms. While the algorithms are not optimized and further improvement is possible, a comparison should still give a rough estimate about their relative performance. The Feature-matching was run in Matlab on the features, ICP was run in Matlab on the full point clouds and GICP in C++ on the down-sampled point clouds. Results can be seen in Table 7.4. The implemented feature-based registration algorithm is order of magnitudes faster than the ICP-based algorithms. While the creation of the keypoints requires some computations it only has to be performed once for each scan.

Table 7.4: Mean run time for the registration algorithms

Algorithm	Mean run time (s)
ICP	6.0
GICP	1.3
Feature-matching	0.024

7.2 SLAM

The implemented SLAM algorithm was tested on the Kitti Vision Benchmark Suite Dataset. The dataset contained longer drives with several areas of loop closures and had high quality LIDAR data synchronized with high quality GPS measurements. Thus it was used to test ability to detect loop closures, to test optimization and to generate different kinds of maps. From the dataset the drive *Residential 2011_10_03_drive_0027 Drive* from the Kitti dataset was selected. The drive include a 7min 35s drive in a medium density residential area with several areas of potential loop closures. GPS coordinates of the drive can be seen in Figure 7.7.

Since the implemented feature-based registration algorithm was found to produce best results and was much faster than ICP and GICP, it was chosen to be used for loop closure. For pose-pose constraints between successive nodes, the OXTS RT3003 would give more accurate measurements. However, in some tests the feature-based registration algorithm was used to get a more visually observable drift.

The feature-based registration algorithm was able to find many loop closures, illustrated in Figure 7.8a. By optimizing the loop closures, using a robust Back-End, the drift in the dead reckoning is almost completely compensated for, as can be seen in Figure 7.8b. The graph looks almost identical to the GPS measurements in Figure 7.7.

The Gauss-Newton optimization algorithm was able to scale well since sparsity of the linearized information matrix H was maintained. An illustration of the sparsity can be seen in Figure 7.9.

While the loop closure hypotheses generated from the dataset seemed correct, robust loop closure was still used to optimize the loop closure constraints with highest average pairwise consistency. As the algorithms main goal is to remove incorrect loop closures, some loop closure with slightly lower average pairwise consistency due

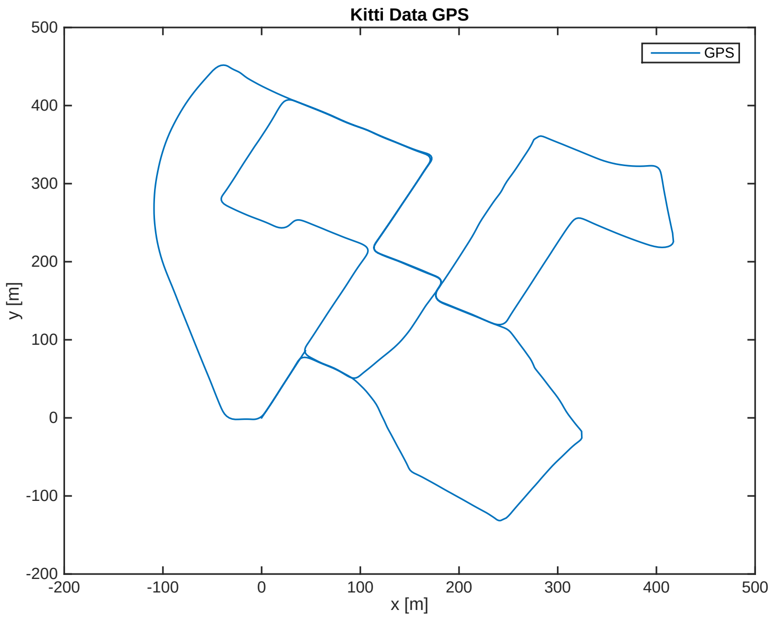
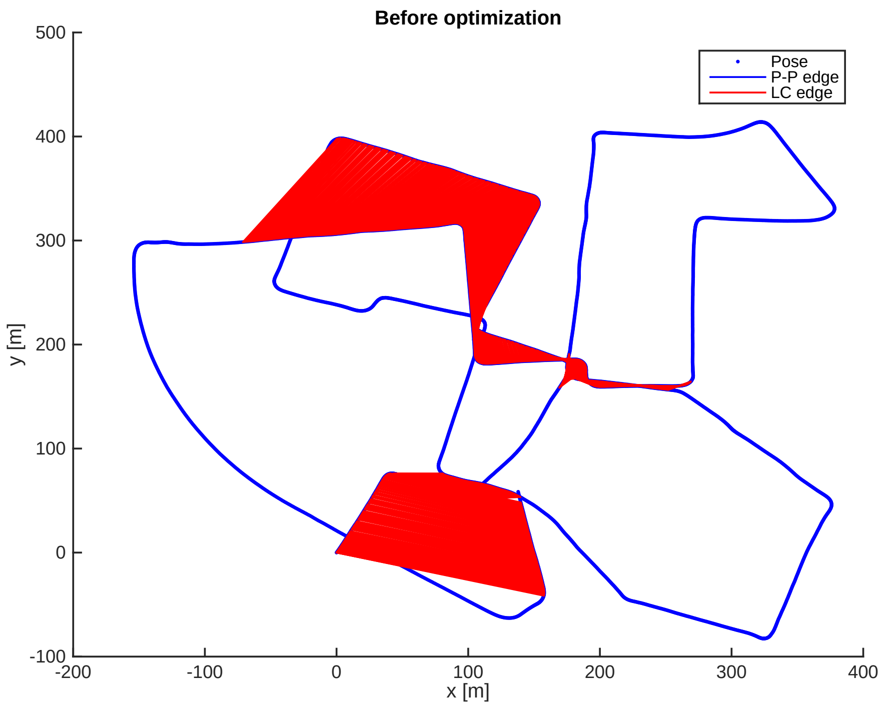


Figure 7.7: Kitti Dataset GPS. All the GPS measurements plotted to illustrate how the vehicle was moving during the drive. As can be seen from the figure several areas of potential loop closure exist.

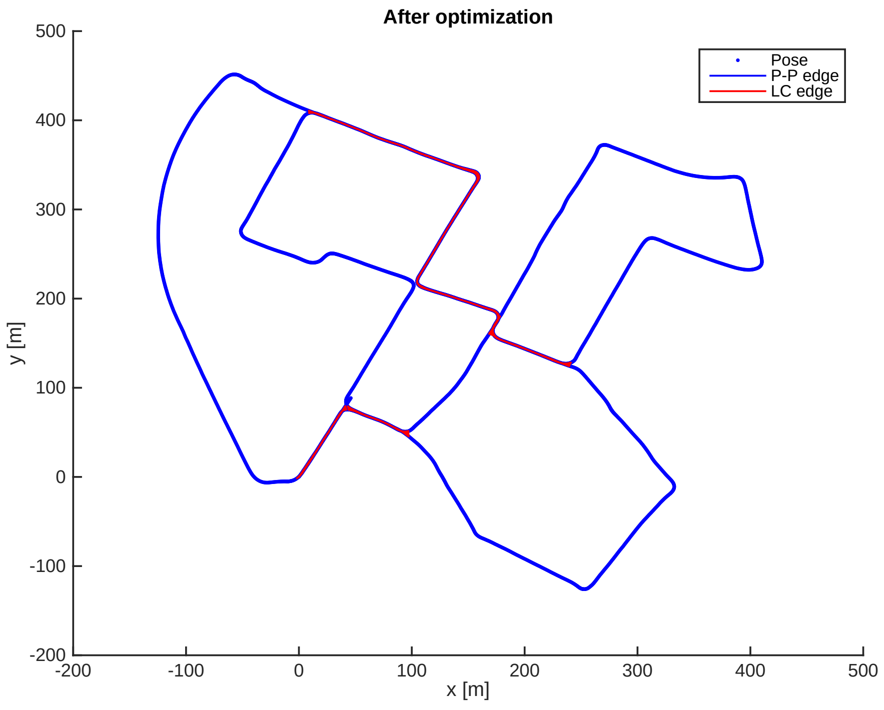
to registration errors were also rejected. The main computational bottleneck was found to be the A* algorithm for finding the shortest path between the hypotheses. It is likely that this algorithm could have been optimized, but as this was not the focus of the project it was not prioritized.

A final graph using all sensor data, with robust loop closure and optimized with a robust Back-End is shown in Figure 7.10. Since the GPS measurements were from high quality sensors, they had a lot of information and thus formed strong constraints on the graph in the optimization. As can be seen the resulting graph is very similar to the ground truth shown in Figure 7.7.

Once a well-optimized graph is available it can be used for many different purposes. To illustrate the observed environment during a drive, a single point cloud of many scans was created. This was done by registering several scans into the same reference frame based on the estimated robot pose at the time of the scan. An example of this is shown in Figure 7.11. This can be used to get a topological view of the area. Filtered point clouds were used to reduce the size of the map and not all scans were used. If all data was used, the resulting point cloud would have almost 600 million points for the drive. As the project was done on personal computers this was would have been intractable.



(a) Before optimization



(b) After optimization

Figure 7.8: Loop Closure Kitti Dataset. In the top figure the dead reckoning using feature-based registration algorithm is shown in blue with the found loop closures in red. In the bottom plot the same graph is shown after optimization of pose-pose- and loop closure constraints.

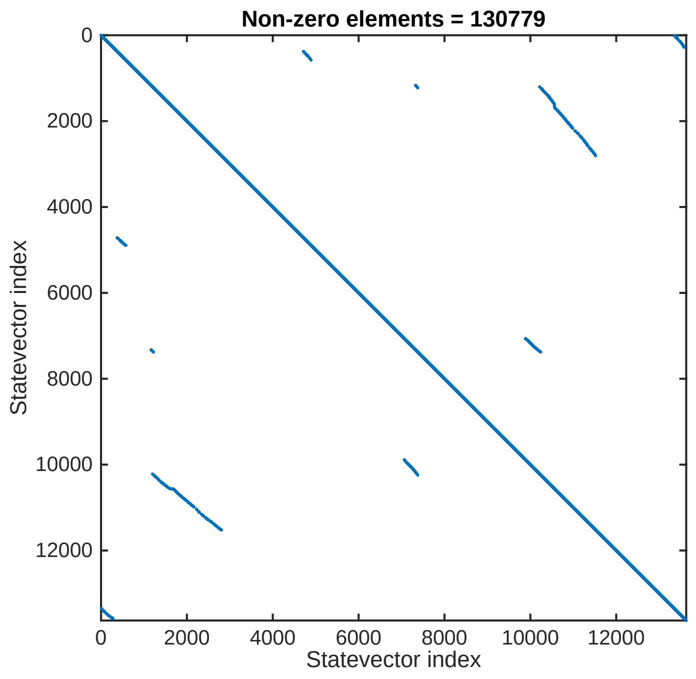


Figure 7.9: Sparsity of the linearized information matrix H. Non-zero elements in H are filled in blue. For each element i in the state vector, which has an edge to another element j in the state vector $\mathbf{p}$, H_{ij} and H_{ji} will be nonzero. As can be seen the matrix is sparse and most elements are along the diagonal with the loop closures can be seen outside the diagonal.

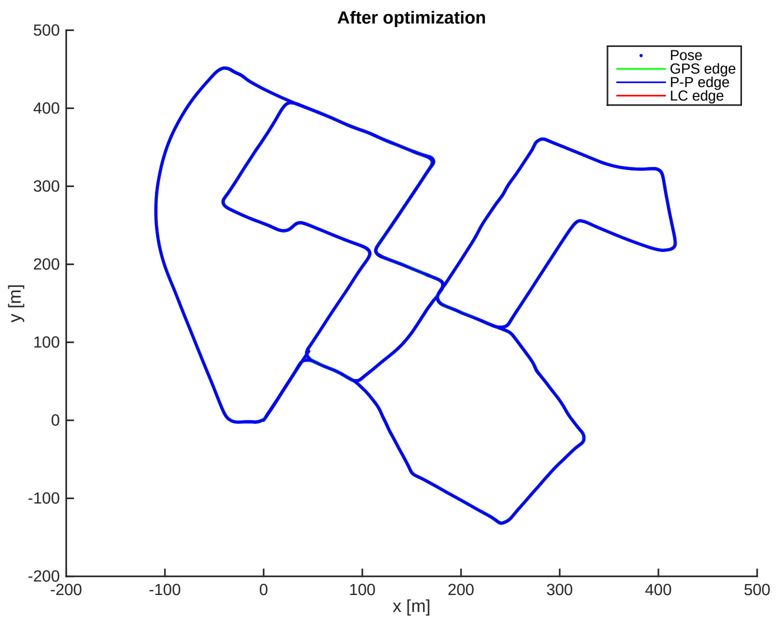


Figure 7.10: Kitti Dataset using all sensor data. The figure shows the graph using the OXTS sensor for consecutive pose-pose constraints and feature-based registration loop closure constraints. The graph has been optimized using robust loop closure and robust Back-End.

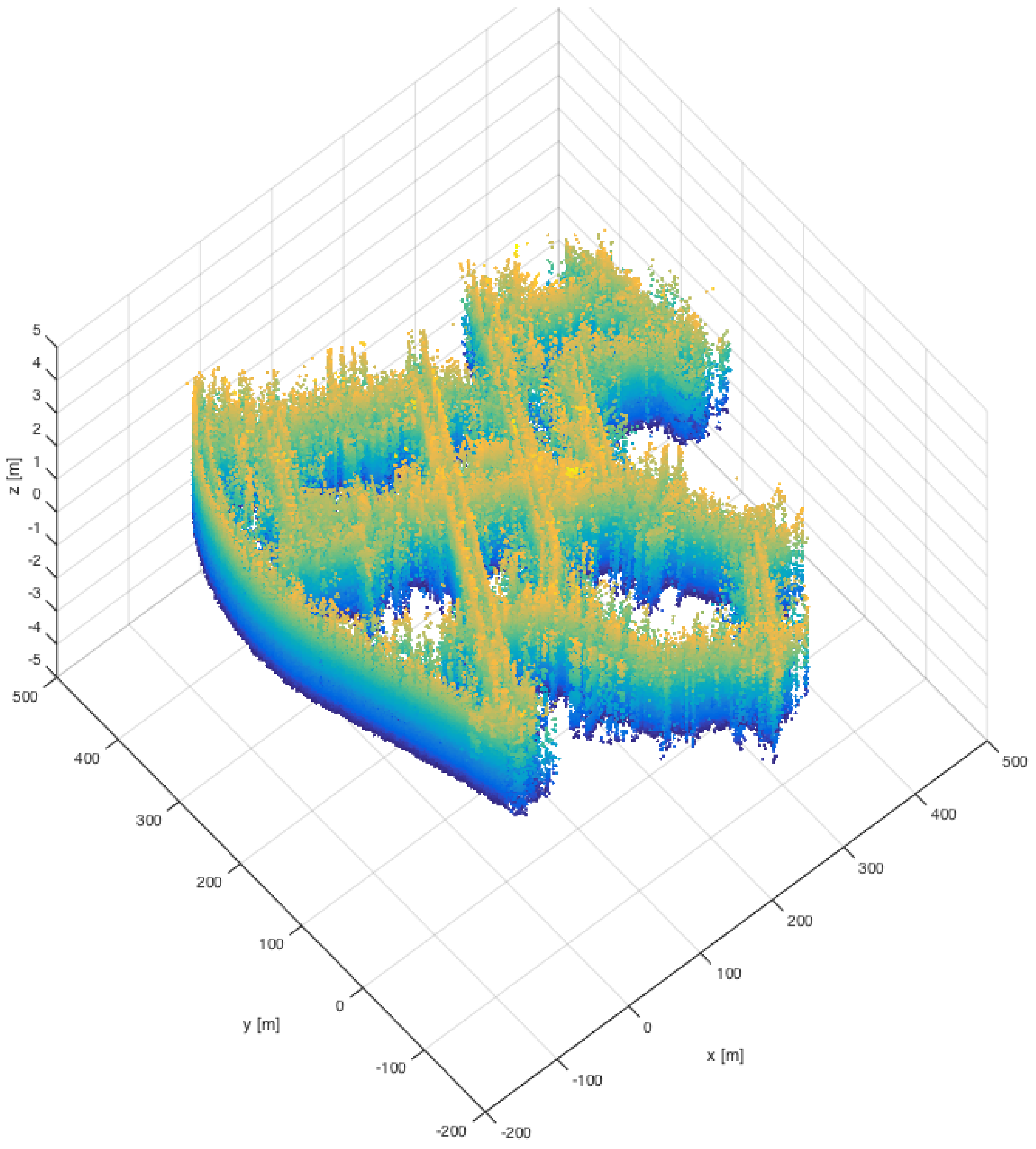


Figure 7.11: Merged point clouds from Kitti Dataset. Each dot indicate one point from the down-sampled point clouds registered into the same reference frame. Blue color indicates that the points are located near the ground and yellow color that they located above the vehicle. The z-axis has a different scale than the x- and y-axes to get a 3D perspective.

7.3 Localization

As the project was done in collaboration with Volvo Group and Volvo Cars, for whom localization might be of interest, a rudimentary localization algorithm was implemented as described in section 6.3.5. Two drives from the Volvo Cars Dataset were selected, one to generate a map and another to localize in the generated map.

The map was generated from a graph from the GraphSLAM algorithm of the first drive, with landmarks spaced approximately 20m from each other along the requested trajectory. The landmarks consist of features from scans and the estimated poses of the scans. An illustration of the map is shown in Figure 7.12

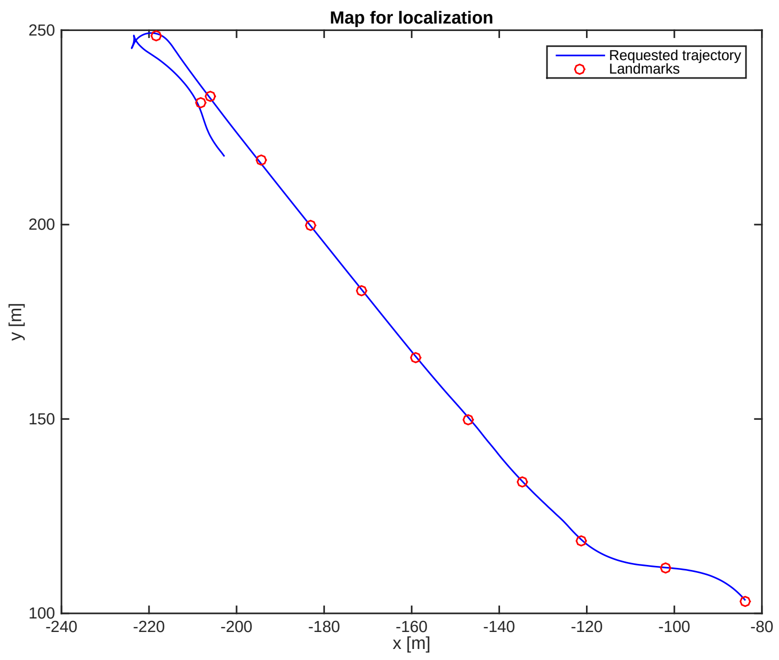


Figure 7.12: Map for localization. The map consists of landmarks placed out along the requested trajectory.

The average data size of a landmark in this high-density residential area is $12kB$. Thus the average data size per kilometer is

$$datasize/km \approx 12kB/20m \cdot 1000m/km = 0.6MB/km.$$

Localization in the map was performed using a particle filter. Motion update is performed by using noisy odometry data, and measurement update is performed using LIDAR data and the map. An illustration of how the particles looked can be seen in Figure 7.13.

The pose estimation of the particle filter was compared to GPS measurement as ground truth and the results is shown in Figure 7.14.

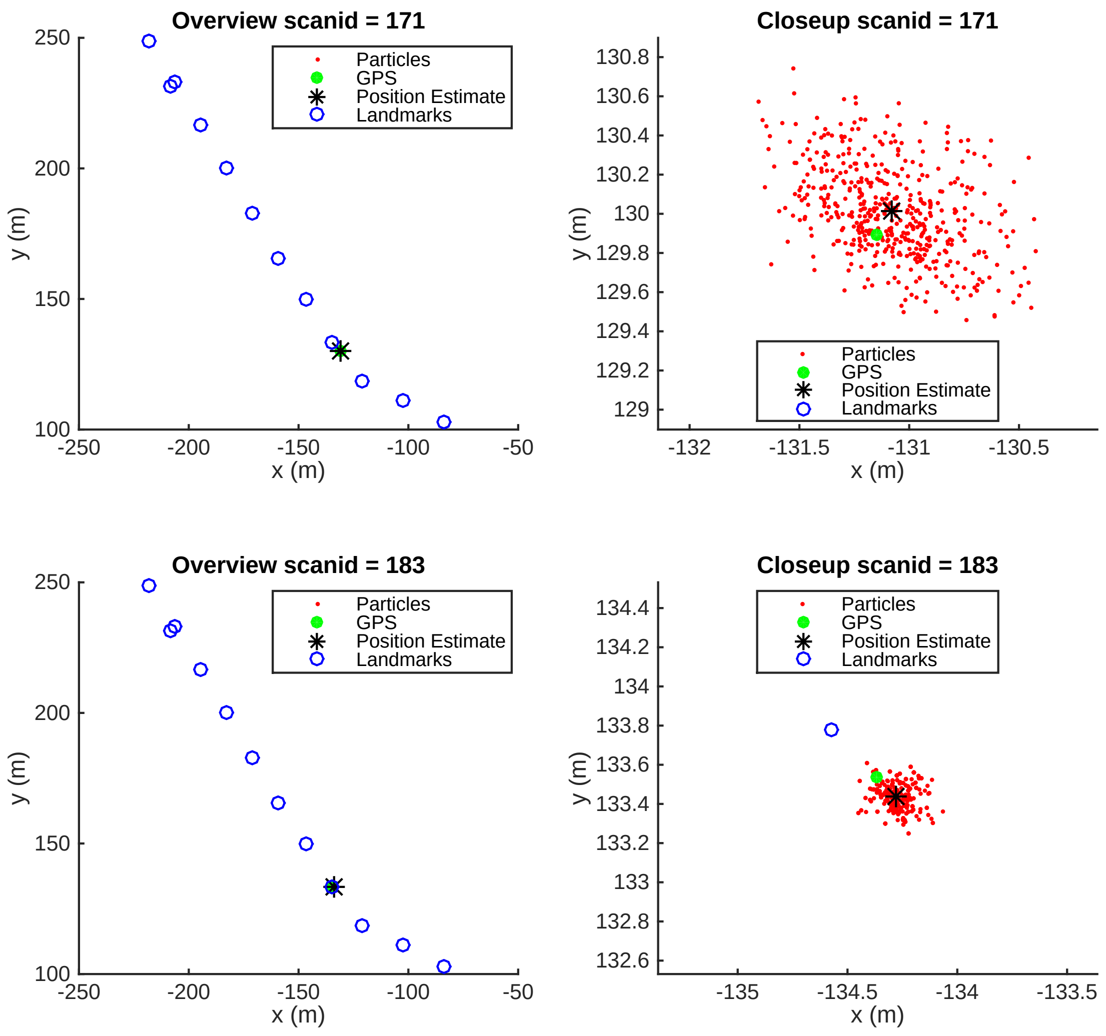


Figure 7.13: Results for localization using a particle filter. In the top plots, time step t = 171 is shown, zoomed out to the left and zoomed in on the right. Since no landmark has been observed for a few scans the uncertainty is large and the particles are spread out. In the bottom plots scan at time step t=183, a landmark has been observed and the uncertainty has shrunk.

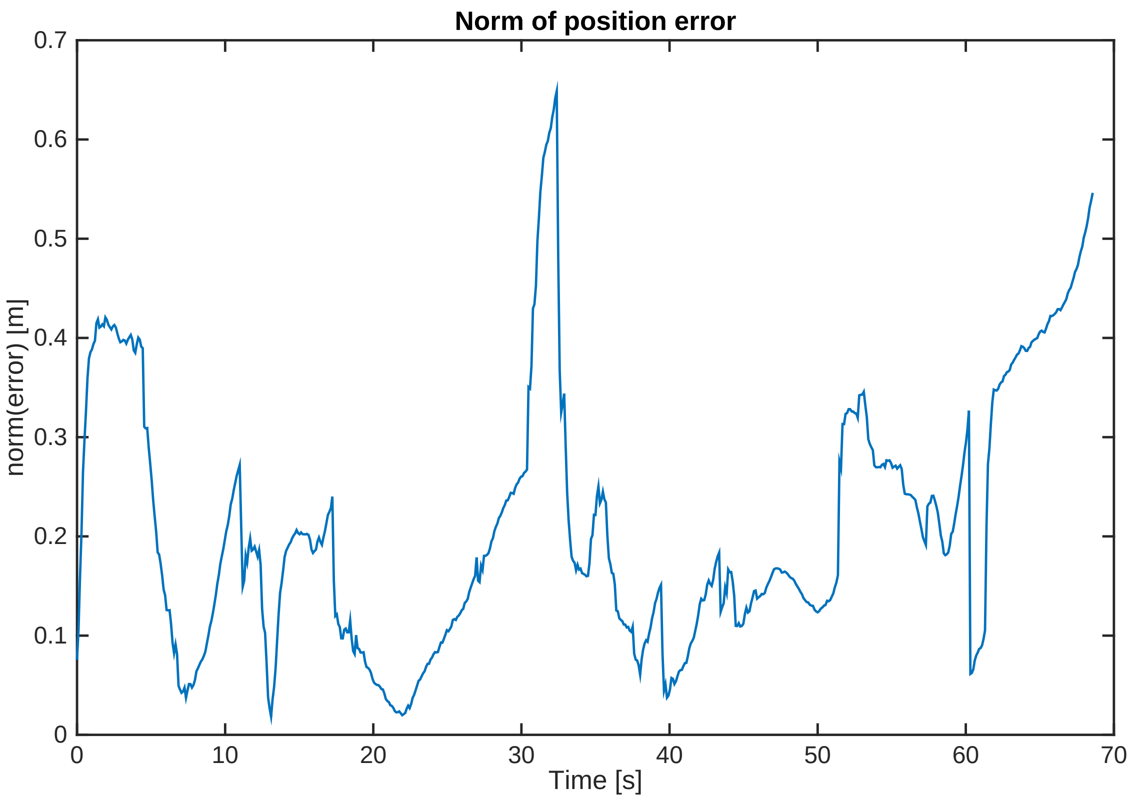


Figure 7.14: Position error of the localization algorithm. The norm of the x and y position error of the localization. The error stays below 0.7m all the time.

8
Evaluation

In this thesis we have presented a full solution for Simultaneous Localization And Mapping(SLAM) with LIDAR sensors. The solution can either work on only LIDAR data or with a combination of LIDAR, GPS and odometry data.

A feature-matching algorithm was implemented for LIDAR data projected onto a 2D plane, which was found to be able to be able to perform registration better than ICP and GICP for the datasets. The feature-matching algorithm was able to find loop closures hypothesis in the all areas revisited in the datasets, but was struggling with estimating the ego-motion in featureless environments such as tunnels.

Loop closures are gathered and the ones with highest average pairwise consistency are selected using a robust Front-End based on Olson's Proposal. A robust back-end using Dynamic Covariance Scaling was implemented for graph optimization. The loop closures had a great impact on the resulting map and the optimization was able to remove most of the drift in the dead reckoning. From the resulting graphs from the GraphSLAM algorithm, different types of maps were created including maps for localization. A rudimentary localization algorithm based on a particle filter was implemented and was found to be able to localize the vehicle in the map within meter level precision. While the localization error was not negligible, there were many potential error sources identified.

8.1 Registration

Three different registration algorithms were implemented. It was found that the feature-matching algorithm was the fastest and performed the best. The algorithm is based on camera vision algorithms but manages to produce good results on LIDAR data projected onto images. There are similar algorithms more suitable for 2D LIDAR data, which can handle occlusion phenomena from ranging data and take more advantage of the scale invariance. An example of this is the FLIRT feature detector and feature descriptor [12]. However since FLIRT is not included in the Computer Vision System Toolbox, it was decided to instead use the included Shi-Tomasi feature detector and the BRISK feature descriptor. It is likely that another combination of feature detector and descriptor would have performed better but would have required more time to implement. As the time for the thesis was limited it was decided to not implement these.

The way the feature-matching algorithm was implemented it could only do registration in the two dimensional plane. As the datasets used were mostly on flat surfaces this was not an issue but might be an issue for other datasets. When the vehicle changes pitch between scans it leads to worse performance as the 2D projections will look different. A solution to this is to use a sensor that measures the pitch, the costs of these sensors are inconsequential to the cost of a LIDAR[31]. By transforming the point cloud based on the pitch measurement the quality of the 2D transformation estimation can likely be improved.

Another way is to use a feature-matching algorithm which extracts features in 3D. A few of these algorithms were tried but implementations were unsuccessful for the data in the datasets, and the time performance was very low. As the time for the project was limited it was decided to not investigate these algorithms further.

The feature-matching algorithm performed well in feature rich environments such as the high density residential areas in the Kitti Vision Benchmark Dataset and the Volvo Cars Dataset. For areas with fewer features such as The Lundby Tunnel it was unable to perform registration. A camera would likely have an easier time detecting features in these environments. J.Levinsson proposed to use bitmaps of the ground infrared reflectivity for localization in these environments [32]. For the SLAM problem one solution is to rely on IMU and odometry measurement inside the tunnel and GPS measurements before and after the tunnel. The LIDAR data can still be used to generate the map used for localization, for example by using the infrared reflectivity of the ground. It should be noted that an advantage of our solution, compared to ground reflectivity based algorithms, should be higher robustness to weather phenomena such as snow.

8.2 SLAM Algorithm

A SLAM algorithm based on the GraphSLAM method was implemented. While other algorithms are still popular, most state of the art SLAM algorithms today are based on the GraphSLAM framework. The algorithm was implemented in MATLAB.

To be able to work with different datasets, a pre-processing algorithm was set up to convert the datasets into a standardized MATLAB format suitable for the GraphSLAM algorithm. As the projects goal was not to do a real time performance implementation, the pre-processing was not optimized for performance and likely could be sped up an order of magnitude and likely be run in real time.

Once the pre-processing was done the implemented GraphSLAM Front-End algorithm could run in real time even without optimization of the code. It is not clear if the SLAM algorithm will be run in real time or offline after the data has been gathered. For large scale mapping of roads it seems likely that an offline solution, where a server collects data from many vehicles after they have gathered the data, will be used to solve the SLAM problem. But for smaller local maps, a real time solution might be of interest.

The implemented Back-End was implemented for both 2D data and 3D data. Since the analytical Jacobian of the error function is complex in the 3D case, a numerical solution was implemented. However the implementation of this function was very slow. As the datasets was mostly in flat environments it was decided to only use two-dimensional maps for the project. Other implementations are available, for example *g2o* written in C++ [33]. The implemented solution in MATLAB was not optimized for performance and is likely orders of magnitude slower at optimizing the graph for 3D.

A robust Front-End was implemented based on Olson's Proposal. As the projected unfolded it was found to not improve performance much, as the number of incorrect loop closures were non-existent the way the loop closure hypothesis were generated. By using the GPS the global ambiguity problem was solved and by comparing hundred of features to hundred of features and removing the outliers and having requirements on the number of inliers, the local ambiguity problem was solved. Still the algorithm was able found to remove some correct loop closure hypotheses which had slightly less consistent measurements than the other hypotheses.

The SLAM algorithm was based on the Pose-GraphSLAM idea where no landmarks are stored in the state vector. This reduces the computational complexity of the algorithms as the features are not optimized and also reduces incorrect loop closures. However the location of the features will have higher uncertainty and using them for localization will produce worse results. For an implementation like this, a robust front-end would likely have a much greater impact.

8.3 Localization

To try out localization a rudimentary localization algorithm was implemented based on a particle filter. This was only intended as a demonstration that SLAM algorithms can be used to generate maps for localization. A drive was selected to generate a map another drive, in the same area, was selected to test localization. The drives were short and lacked good areas for loop closure lowering the quality of the map. While the algorithm was rudimentary, it was still able to localize the vehicle within a radius of 0.7m in a dataset not designed for SLAM or localization, using only LIDAR data and noisy odometry. A few different potential sources of error were identified:

- Ground truth was a GPS sensor subject to measurement noise
- The map was based on a single drive going in the same direction most of the time, a map from a drive revisiting the places many times would likely perform better
- The feature-based registration algorithm used can likely be improved to give better estimates over longer distances between scans
- Covariance matrices for the particle filter can likely be better tuned

The map was fairly compact at around 0.6MB/km, a rate that could even be possible to download over the cellular network.

8.4 Choice of Datasets

As the thesis involved doing solving many small sub-problems it was decided to start with ideal data. This would reduce the likelihood of getting stuck and simplify verification of different aspects of the project. Once a solution for high quality sensors is complete it is easier to try and adapt this solution to work with cheaper and more robust sensors of lower quality. Three suitable datasets were identified, the Ford Campus Dataset [23], the Kitti Vision Benchmark Dataset a proprietary dataset from Volvo Cars. These datasets had high quality LIDAR scans from Velodyne HDL-64E and high quality GPS measurements. As time was limited in the project no solution for lower quality sensors was attempted.

The Ford Campus Dataset included LIDAR data from a low industrial area with fewer features detected by the feature detection algorithm. Still the algorithm was able to perform SLAM on the dataset with some slight modification to the settings in the feature extraction algorithms. While the Ford Campus Dataset had the longest drive, the Kitti Vision Benchmark Suite Dataset had an almost as long drive with even more loop closures. It was decided to only include results from this dataset, thus results from the Ford Campus Dataset are omitted from the report.

9 Future Work

The ambition of this thesis was to investigate which SLAM algorithms are available and which ones are suitable for further development, and to investigate which challenges exists by implementing a solution for LIDAR data. It is the view of the authors that the ambition has been reached, but that the implemented solution will require a lot of future work before being suitable for commercial applications. In this chapter suggestions for future work are presented.

9.1 Registration

The implemented feature registration method works by projecting the point cloud onto a 2-dimensional plane. The algorithm struggles when the vehicles pitch is changing between the scans. If the change in pitch is known, for example measured with an accelerometer, the pitch can be used to transform the point cloud before features are extracted. It is the view of the authors that this should improve recognition and therefor also improve localization. Another potential improvement is using a keypoint extractor and feature descriptor more suitable for LIDAR data such as FLIRT.

By converting the point cloud into 2D, a lot of information is discarded and only the two-dimensional transformation can be estimated. A method that works in 3D can make use of all the information and might provide a more accurate transformation. When reflectivity information is available it can also be used to improve registration, but it is important that the algorithm remains robust to changes in reflectivity due to rain or other sources.

One interesting approach is using Convolution Neural Networks (CNN) to extract features or to estimate transformations. Estimating rigid body transformation from point clouds of human faces using CNN has been tried by C.Junfen et. al. [34]. As CNNs has been very successful in camera image analysis, it is the authors view that CNN has the potential to be very successful in point cloud analysis also. CNN also has the promise of performing well for combinations of different data sources, such as camera, LIDAR and IMU sensors.

9.2 SLAM

As the data was mostly from drives in flat areas it was decided to solve the SLAM problem in 2D. For other datasets it may be of an advantage to have the map and robot trajectory in 3D.

For large scale mappings there are available methods for faster optimization. Instead of updating the entire map, the map can be split into different hierarchical levels as proposed by G.Grisetti et. al. [20]. There are also methods for using Bayes Tree for optimization as proposed by M.Kaess [35].

Another important aspect is removing dynamical objects from maps such as other vehicles and for updating maps over time. Objects that were observed in previous drives but not the next should be removed. A marginalization algorithm was implemented, which could remove older nodes if new nodes were observed in the approximately same position. As the datasets did not include many drives at different times this was not tried.

9.3 Localization

The particle filter is a robust and well performing algorithm for localization. As time was limited and the datasets were not designed for testing of localization, little effort was put into tuning the algorithm. For future work it would be recommended to have a dataset that consists of a longer drive around in one area followed by a drive through the same area. The covariance matrices in the particle filter can be tuned better and a smarter resampling can be used. This would improve the performance and the computational complexity.

A way to further reduce the size of the map is to make a training set of features gathered for a larger set of data, and only store features which are similar to the features in the training set. By doing this only the identifiers to the feature in the training set have to be stored, not the entire feature descriptor for each feature in the map.

The implemented feature-matching algorithm struggles in feature-poor environments such as tunnel. Instead of relying on features another interesting approach is to use bitmaps of reflectivity as suggested by J.Levinsson [32]. A solution like this, may require LIDAR sensors with higher angular resolution than the sensors used in this project.

Another potentially interesting solution is to represent the spatial world with planes or other basic geometric figures. This could create a very efficient representation of the map and is suitable for usage in combination with a particle filter.

Bibliography

[1] Volvo Trucks. European Accident Research Safety Report. *Engineering*, pages 1–31, 2011.

[2] James M. Anderson, Nidhi Kalra, Karlyn D. Stanley, Paul Sorensen, Constantine Samaras, and Oluwatobi a. Oluwatola. *Autonomous Vehicle Technology - A Guide for Policymakers*. 2014.

[3] Wikipedia. Simultaneous localization and mapping, 2015. [Online; accessed 27-april-2015].

[4] S Thrun, W Burgard, and D Fox. *Probabilistic Robotics*. Intelligent robotics and autonomous agents. MIT Press, 2005.

[5] Cyril Stachniss. Robot Mapping WS 2013/2014, 2013.

[6] Ben Bellekens, Vincent Spruyt, Rafael Berkvens, and Maarten Weyn. A Survey of Rigid 3D Pointcloud Registration Algorithms. *The Fourth International Conference on Ambient Computing, Applications, Services and Technologies*, 2014.

[7] D.W. Eggert, a. Lorusso, and R.B. Fisher. Estimating 3-D rigid body transformations: a comparison of four major algorithms. *Machine Vision and Applications*, 9(5-6):272–290, 1997.

[8] Aleksandr V Segal, Dirk Haehnel, and Sebastian Thrun. Generalized-ICP. *Proc. of Robotics: Science and Systems*, 2:4, 2009.

[9] Hans Martin Kjer and Jakob Wilm. Evaluation of surface registration algorithms for PET motion correction Bachelor thesis. *Orbitdtudk*, 2010.

[10] Radu Bogdan Rusu. Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments. *KI - Künstliche Intelligenz*, pages 1–4, 2010.

[11] Davide Scaramuzza and Friedrich Fraundorfer. Visual Odometry [Tutorial]. *IEEE Robotics & Automation Magazine*, 18(4):80–92, 2011.

[12] Gian Diego Tipaldi and Kai O. Arras. FLIRT - Interest regions for 2D range data. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 3619–3622, 2010.

[13] R Hänch, T Weber, and O Hellwich. Comparison of 3D Interest Point Detectors and Descriptors for Point Cloud Fusion. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, II-3, 2014.

[14] Stefan Leutenegger, Margarita Chli, and Roland Y. Siegwart. BRISK: Binary Robust invariant scalable keypoints. *Proceedings of the IEEE International Conference on Computer Vision*, pages 2548–2555, 2011.

[15] Jorge Nocedal and Stephen J. Wright. Least-Squares Problems. *Numerical Optimization*, pages 254–257, 2006.

[16] Wikipedia. A* search algorithm, 2015. [Online; accessed 27-april-2015].

[17] W. Zeng and R. L. Church. Finding shortest paths on real road networks: the case for A*, 2009.

[18] D Koller B Wegbreit J Nieto E Nebot S. Thrun M. Montemerlo. FastSLAM: An Efficent Solution to the Simultaneous Localization and Mapping Problem with Unknown Data Association. *Journal of Machine Learning Research*, (July), 2004.

[19] S. Thrun. 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.

[20] Giorgio Grisetti, Rainer Kummerle, Cyrill Stachniss, and Wolfram Burgard. A tutorial on graph-based SLAM. *IEEE Intelligent Transportation Systems Magazine*, 2(4):31–43, 2010.

[21] Edwin Olson. Recognizing places using spectrally clustered local matches. *Robotics and Autonomous Systems*, 57(12):1157–1172, 2009.

[22] Pratik Agarwal, Gian Diego Tipaldi, Luciano Spinello, Cyrill Stachniss, and Wolfram Burgard. Robust map optimization using dynamic covariance scaling. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 62–69, 2013.

[23] G. Pandey, J. R. McBride, and R. M. Eustice. Ford Campus vision and lidar data set. *The International Journal of Robotics Research*, 30(13):1543–1552, 2011.

[24] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*, 2013.

[25] Press Release (Volvo Cars). Volvo Car Group initiates world unique Swedish pilot project with self-driving cars on public roads, 2013.

[26] Velodyne Acoustics. User's Manual and Programming Guide HDL-64E S2 and S2.1; Firmware Version 4.07. 4.07.

[27] OXTS. User manual Covers RT2000, RT3000 and RT4000 products. 2013.

[28] Nvidia. Whitepaper NVIDIA ® Tegra ® X1, NVIDIA's New Mobile Superchip. pages 1–41.

[29] Jakob Wilm. Iterative Closest Point, 2010.

[30] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[31] Yangming Li and Edwin B. Olson. Extracting general-purpose features from LIDAR data. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 1388–1393, 2010.

[32] Jesse Levinson. Automatic Laser Calibration, Mapping, and Localization for Autonomous Vehicles. (August), 2011.

[33] Rainer Kümmerle, Giorgio Grisetti, Hauke Strasdat, Kurt Konolige, and Wolfram Burgard. G2o: A general framework for graph optimization. In *Proceedings - IEEE International Conference on Robotics and Automation*, pages 3607–3613, 2011.

[34] Chen Junfen, Yi Liao Iman, Belaton Bahari, and Munir Zaman. A neural network-based point registration method for 3D rigid face image. *World Wide Web*, 18(2):197–214, 2015.

[35] Michael Kaess, Hordur Johannsson, Richard Roberts, Viorela Ila, John Leonard, and Frank Dellaert. iSAM2 : Incremental Smoothing and Mapping Using the Bayes Tree. 2008.