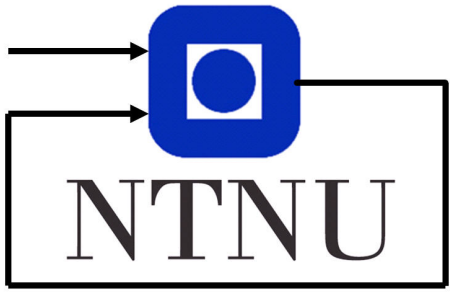
SLAM-based Tunnel Navigation System for Autonomous Ground Vehicles

Sjur Grønnevik Wroldsen

December 17, 2020



Department of Engineering Cybernetics

Problem Description

Inertial navigation is often paired with measurments from the global navigation satellite systems(GNSS) in order to avoid drift in position estimation. However, GNSS measurements may not be available or uncertain under several conditions, like when the vehicle is in an indoor environment. This has raised the need for other navigation methods that can control the vehicle in situations where the classical approach cannot be relied upon.

This thesis investigates the problem of navigating a slowly moving autonomous vehicle through a tunnel autonomously. With the end goal of enabling the autonomous vehicle to drive from an environment where GNSS is available to navigate through a tunnel performing object avoidance and path planning simultaneously, this thesis proposes a LiDAR-based simultaneous localization and mapping (SLAM) front-end estimating the odometry and building a map.

Abstract

Autonomous vehicles have seen an enormous growth in recent years. Moreover, autonomous vehicles require robust systems for navigation. Typically they employ inertial navigation systems(INS) aided with measurements from global navigation satellite systems(GNSS) to keep their estimates globally consistent. In the case of bad satellite coverage, however, INS tend to drift quickly due to drift that occur in inertial sensors.

To provide a backup to such a system, a feature-based LiDAR odometry system is proposed. The proposed system extracts intrinsic shape signatures(ISS) keypoints from the point cloud in order to reduce the dimensionality of the problem. The pose-invariant fast point feature histogram(FPFH) feature descriptor is used for matching. Further the scan match is done following the fast pairwise global registration(FPGR) method proposed in [25]. In order to create a robust backup system, the challenges of LiDAR based navigation must be identified. This is done through several simulated scenarios.

The proposed system shows that LiDAR navigation systems are prone to drift, especially in the travelled distance. Following a simple path the system yields a root mean square error(RMSE) of 20.2m in travelled distance and a RMSE of 2.5° in heading. For a more complicated path the system yielded a RMSE of 18.9m in travelled distance and a RMSE of 3° in heading. The experiments also showed that the proposed system is unable to provide estimates in real-time, raising the need for better and more efficient matching methods.

Feature-based LiDAR SLAM has shown great promise in 6 degrees of freedom(DOF) estimation for both aerial and ground vehicles. To keep global consistency and suppress drift it is possible to fuse GNSS measurements into the estimation. Other methods include implementing smoothing over a set of poses and the detection of loop closures. Inertial navigation also include inertial sensors, which makes it natural to investigate the fusion of this into the estimation problem.

Preface

This report is a result of a study performed at the 9th semester towards the degree of Master in Technology at the Department of Engineering Cybernetics at the Norwegian Institute of Science and Technology (NTNU). The study has been completed in collaboration with Semcon Norge to investigate different technology.

The result of this report is largely in the form of a literature review, in addition to some testing and simulation of different implementations. The implementations are a mixture of self-implemented methods and methods implemented using open-source libraries.

Supervisor: Annette Stahl

Co-supervisor: Eirik Hexeberg Henriksen

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Inertial navigation systems(INS) are the de facto standard for most autonomous navigation purposes. However, INS heavily relies on bona-fide measurements such as GNSS in order to avoid drift in the inertial sensors. This creates issues whenever such measurements are unavailable, as is the case when an autonomous vehicle drives through a tunnel. The issue arises since the vehicle has no means to localize itself in the environment.

Simultaneous localization and mapping(SLAM) is a technique where the goal is to estimate the motion of the ego-vehicle while simultaneously mapping an unknown environment. In order to solve this problem, extensive computer power and mapping sensors are required.

The SLAM problem has seen solutions based on the traditional inertial navigation techniques. The first example is the Kalman filter solution, namely EKF-SLAM, which solved the *online SLAM* problem. The ground-breaker for solving the *full SLAM* problem was the particle filter solution called factored solution to SLAM, abbreviated FAST-SLAM[21]. Typically these solutions involved odometry sensors such as wheel encoders.

The recent solutions to the SLAM problem has involved using cameras to estimate motion. This is called visual odometry(VO), and has either replaced or been combined with other odometry sensors. VO has shown promise compared to classical odometry sensors in terms of accuracy and drift. These SLAM systems typically solve the bundle adjustment(BA) problem to estimate structure and/or motion. Modern systems based on these ideologies include ORB-SLAM[13] and its predecessors, LSD-SLAM[6] and PTAM[12].

However, cameras as sensors suffer under different illumination conditions. Additionally, the situational awareness is restricted to the field of view(FOV) of the camera. This has motivated for the use of other sensors for autonomous navigation. A technology which has shown promise in combination with SLAM are LiDAR sensors (both solid state and scanning) to map the surroundings of a vehicle. LiDARs offer real-time scanning of the environment with up to 360° FOV.

Unlike VO methods, LiDAR odometry methods can observe scale using only a single sensor which in turn could give a more consistent map of the surroundings. Methods such as LiDAR odometry and mapping(LOAM)[23] is method of estimating motion in 6 degrees of freedom(DOF) and creating 3D maps based on 3D LiDARs. However, as it only is an odometry method, it is prone to drift eventually. To minimize the effect of drift, a LiDAR-inertial odometry (LIO) method was proposed in [18]. LOAM was also extended to include loop closures in [17]. Other state-of-the-art LiDAR SLAM methods include Google Cartographer[9]. Unlike LOAM, Google Cartographer also requires IMU for 6DOF estimation of pose.

LiDAR-based motion estimation techniques are typically divided into two categories; scan-matching and feature-based methods. Scan-matching methods are typically based off the iterative closest point(ICP) algorithm. Feature-based

methods, however, aims to find robust points, match those and calculate the transformation between the point clouds.

One of the largest issues in solving the SLAM problem is the computational complexity. This is why graph-based SLAM methods may be the most popular approach to modern SLAM, and is heavily researched. The main benefit of graph-based methods is that they exploit the sparsity of the SLAM problem optimally. Motivated by the similarities to the QR-decomposition, these methods typically involve representing the problem in the form of a *factor graph* and exploit other graphical models such as the *Bayes net*. The state of the art graph-based SLAM solution may be iSAM2[11], which also utilizes *Bayes trees* to optimally structure the data in the factor graph.

A challenge when it comes to recording indoor datasets using any type of exteroceptive sensors is the, as mentioned, lack of bona-fide measurements that can be used as ground truth. To overcome this problem, simulators are typically used to maintain full control of both the environment and the motion of the vehicle. Popular simulation software for robotics include Gazebo[14]. Recently, MATLAB has also introduced a simulation framework for autonomous navigation purposes, in cooperation with Unreal Engine[7]. The simulator in this thesis is set up using MATLAB and Unreal Engine through the `Automated Driving Toolbox`.

For the problem of navigating an autonomous ground vehicle through a tunnel I propose a system based on using LiDARs as the main sensors. LiDARs are more suitable for the problem than other mapping sensors such as cameras because tunnels typically have varied illumination conditions. Additionally, LiDARs provide a much larger FOV than cameras, opening up for a higher situational awareness without adding more complexity. This is desirable for path-planning and collision-avoidance systems, both of which are desirable features of an autonomous system. To create different scenarios I provide several simulation environments in which one or more LiDARs can be placed for simulating an autonomous ground-vehicle navigating through a tunnel. The simulations are carried out in MATLAB, inside of a custom-made environment built in Unreal Engine. The data sets are then extracted from MATLAB and written to a rosbag. This way it is possible to benchmark the performance of a SLAM system in terms of both accuracy and speed. Furthermore I propose SLAM front-end containing a LiDAR odometry and mapping method implemented in Python using ISS keypoints and FPFH features following the idea of the fast global registration method as proposed in [25].

1.2 Goal of the project

This thesis investigates the problem of navigating an autonomous ground vehicle through a tunnel. The aim of this pre-project is to develop a realistic simulation environment for autonomous ground vehicle purposes. One of the criterias for the simulator is that it is possible to construct a wide variety of different scenarios where it is possible to evaluate the performance of localization techniques. Ultimately, the goal is to use this environment to investigate the challenges related to using feature based LiDAR SLAM methods for the purpose of navigating through a tunnel.

1.3 Challenges

There are many challenges related to feature-based LiDAR SLAM. As for all navigation systems, they have to be consistent in order to provide good estimates. They have to be robust towards failure, as for example when loosing track of its position, and towards noisy measurements. The system also needs to be reliable in that it needs to be able to provide positional estimates with real-time performance in order to be used for real-world applications.

1.4 Contributions

In order to address the challenges of LiDAR based SLAM I made an extensive literature overview of graph-based SLAM optimization and current state-of-the-art LiDAR SLAM methods. Initially the literature seemed overwhelming and I struggled to get an overview, leading to a stagnation in progress.

To verify the performance of the proposed SLAM system I created a simulation environment in MATLAB and Unreal Engine in order to conjure data from a wide spectra of scenarios where I had access to the ground truth. The MATLAB solution was chosen because of their extensive examples, good tools for confirmation and visualization and since I personally had no experience with ROS and Gazebo.

After speaking to Kostas Alexis at NTNU early in November, however, I received a strong recommendation to develop the SLAM system in the ROS framework. Thereafter some time went to learning the ROS basics, and migrating some of the already created modules to use ROS.

The LiDAR odometry system proposed in this thesis is based on the fast global registration method[25]. This method was chosen because of its robust scan-matching techniques, as the matching process proved to be a challenge initially.

My contributions can be summarized as follows

- Literature overview of graph-based SLAM
- Overview of current state-of-the-art feature-based LiDAR SLAM methods
- A modifiable simulation environment in which to evaluate SLAM implementations
- A simple LiDAR odometry and mapping module to address the challenges of LiDAR navigation made in Python with ROS

1.5 Outline

The remainder of this thesis is organized as follows. Section 2 gives an overview of the preliminaries relevant to the thesis. Section 3 gives an overview of current state-of-the art SLAM methods. Section 4 gives an introduction to the development platform, including the simulator and the software used in the development of the LiDAR odometry and mapping module. Section 5 shows an overview of the proposed system in terms of software architecture. Section 6

then presents the results before Section 7 discusses the results in light of the goal proposed in Section 1.2. Finally Section 8 then provides a conclusion.

2 Preliminaries

This section describes the preliminary information relevant for the thesis. The opening sections give some preliminary information relevant for the thesis, before the focus turns to SLAM-specific theory and considerations. Section 2.1 gives an introduction to some of the relevant sensors. Section 2.2 introduces some of the simulation frameworks for autonomous navigation.

Section 2.3 and Section 2.4 introduces the theory behind the classical SLAM problem, and why it is difficult. Section 2.5 - 2.7 introduces the relevant theory for modern SLAM. Section 2.8 introduces graph-based SLAM methods, putting much concern on the back-end, and why they have become the standard way of formulating a SLAM problem. Lastly, sections 2.9 - 2.11 focuses on the front-end, describing methods used for registrating point clouds.

Much of the theory about SLAM is inspired by [2], [5] and [3].

2.1 Sensors

Typical for modern navigation systems is the use of satellites, IMUs and *exteroceptive* sensors. Exteroceptive sensors are sensors that provide information about the environment outside the system itself. Typically used exteroceptive sensors include LiDARs and cameras.

2.1.1 GNSS

Global navigation satellite systems(GNSS) are a common term for satellite-based systems for navigation and positioning with global coverage. In total there are four established systems: the American system GPS, the Russian system GLONASS, the Chinese system BeiDou and the European system Galileo. Together they provide the global location of GNSS-receivers.

The GNSS calculates position by triangulating the distance measure from three different satellites simultaneously. The distance is calculated by measuring the time it takes to transfer a radio signal from the satellite to the receiver on Earth. The transfer time is then calculated by an atomic clock inside of the satellite. Using these clocks give an extremely accurate measure of position, which is why the GNSS measurements are often used as bona-fide measurements, or in other words, treated as ground truth.

2.1.2 IMU

Inertial measurement unit(IMU) is a sensor measuring inertial forces. IMUs have three axes, which are typically mounted along the body axes of the vehicle to be controlled.

IMUs typically consists of accelerometers, gyroscopes and sometimes compasses. The accelerometer is able to measure linear accelerations along the axis of the IMU, and the gyroscope measure the angular rates. Compasses measure the heading. Using these measurements, it is possible to get an estimate of the current position, velocity and orientation of the vehicle. This is done by integrating the measured linear acceleration and angular rates. However, IMUs tend

to drift over time and also has variable performance under different conditions. This makes it undesirable to use IMUs on their own for long-time operations, which is why it is often paired with GNSS measurements.

2.1.3 LiDAR

LiDAR is an acronym for "Light Detection and Ranging" and is a sensor based off the time-of-flight(TOF) principle. To do so it uses a light source, a detector circuit and the fact that light speed is constant in a given medium.

LiDAR sensors employ optical signals to measure the range to objects. This is done by emitting the optical signal onto an object, called the target, before detecting and processing the backscattered signal to determine the range. For a TOF lidar, the range is calculated according to

$$R = \frac{c}{2} t_{OF} \tag{2.1}$$

where t_{OF} is he time of flight and c is the speed of light in the given medium. Also keeping the azimuth angle at which the laser was fired, and employing several such lasers in an organized manner create what is known as the scanning lidar.

2.2 Simulation Frameworks

There exists several simulation frameworks for robotics. Many of these are open-source and contain alot of pre made content and environments. The two that will be presented here are Gazebo and Unreal Engine, but other renowned open-source frameworks like Webots[22] also exist.

2.2.1 Gazebo

Gazebo[14] is an open-source 3D dynamics simulator especially created to simulate a wide variety of robots in complex indoor and outdoor environments. Additionally, the simulator has the opportunity to use different physics engines and comes equipped with a vast suite of sensors and interfaces for a alot of different applications.

Gazebo is a highly regarded simulator in the robotics community. In 2012 the Open Source Robotics Foundation(today called Open Robotics), the same people who created ROS, took over the responsibility for the simulator. As a result of this all data that comes from the simulator follows the ROS standards. Open Robotics are also responsible for several highly renowned robotics competitions, such as the DARPA challenge, all of which employ Gazebo as the simulation environment. This has further contributed to giving Gazebo a huge user community and pre-made scenarios for robotics.

2.2.2 Unreal Engine

Unreal Engine[7], more specifically Unreal Engine 4, is an open-source game engine developed by Epic Games written in C++. It comes equipped with high-quality graphics, a lot of pre-made building blocks with modifiable physics and a GUI with both drag-and-drop and landscaping opportunities.

Unreal Engine is originally a game engine equipped with a GUI for designing and creating game scenarios for a wide variety of developers. Therefore it comes with a vast community and many different pre-made worlds and props. However, it does not come equipped with typical navigation-purpose sensors such as IMUs, LiDARs and cameras. This changed when MATLAB R2019b introduced the opportunity to establish a connection between Simulink and Unreal Engine.

2.3 SLAM

Simultaneous Localization and Mapping (SLAM) is a problem that was introduced in the 1980's. It consists of two sub-problems; mapping the environment and estimating the motion of an ego-vehicle inside it. However, its heavy computational needs has forced it to grow alongside the development of more powerful computers, and effective nonlinear optimization methods. In addition, since it relies on exteroceptive sensors it is clear that the popularity of SLAM has grown with the development of better and cheaper exteroceptive sensors such as the LiDAR as well.

2.4 Classical SLAM Problem

The classical SLAM problem is based on estimating pose, known as localization, in an environment with an unknown prior. This requires an estimation of the structure of the environment, often called a map. Mathematically this can be expressed as estimating the joint pose-landmark vector $\mathbf{X}_k = [\mathbf{x}_k^T, \mathbf{m}^T]^T$ given the measurement vector $\mathbf{z}_k$ at each time step, k. $\mathbf{x}_k$ and $\mathbf{m}$ are known as the poses and the map, respectively. The vector $\mathbf{m} = [\mathbf{l}_1^T, \mathbf{l}_2^T, ...\mathbf{l}_m^T]^T$ contains the m landmarks inside the map at all times, and $\mathbf{z}_k$ contains measurements of the landmarks, $\mathbf{l}_i$. Repeatedly estimating $\mathbf{X}_k$ for each time step allows for building a map of the environment while exploring a scene. This procedure is often referred to as SLAM[2].

2.4.1 EKF-SLAM

As SLAM was introduced to increase the robustness of inertial navigation, the early solutions to the SLAM problem involve Kalman Filters(KF), more precisely the Extended Kalman Filter(EKF). EKF-SLAM solves what is called the *online SLAM problem*. What this refers to is the fact that at every instance in time, the EKF performs marginalization over all the previous poses, effectively "forgetting" all about its trajectory in the estimation process. This allows for fast computation times, as the complexity of the pose estimation does not increase in size with time. However, the marginalization step also allows for linearization errors to accumulate over time. Additionally, the innovation covariance matrix of the EKF tends to become very dense since the independence structures involved in the SLAM problem is not exploited. As a result of this, the EKF solution to the SLAM problem scales very badly as the number of landmarks increases.

2.4.2 FAST-SLAM

As a solution to this the *Rao-Blackwellized* particle filter was introduced in FAST-SLAM[21]. Rao-Blackwellization is a method of decreasing the dimen-

sionality of a particle filtering problem. It involves separating the state space into a linear part and a nonlinear part. The reason why a particle filter is needed in the first place is because the estimation of the nonlinear states does not have a closed-form solution. This solution is approximated by sampling particles from a proposal density. With a Rao-Blackwellized particle filter only the nonlinear states needs to be sampled by the particle filter. It is therefore desirable to keep this as small as possible. For navigation purposes it is typically chosen to be the state of the vehicle. The linear state can be optimally calculated by a linear Kalman filter.

The particle filter approach samples trajectories, indirectly smoothing the state vector of the vehicle. In turn this reduces the linearization errors that tend to accumulate in the EKF-SLAM method. SLAM solutions that do not only concern the current state, but also all previous states are said to solve the *full SLAM problem*. However, as all particle filter solutions, Fast-SLAM involves random sampling to some extent. This raised the question whether or not there were other, more structured techniques to solving the problem. This is what paved the way for the modern graph-based SLAM methods.

2.5 Poses

Poses in the SLAM problem consists of a position and an orientation. Poses are often manipulated using *transformation matrices* which are part of the *special Euclidean group* in three dimensions, often abbreviated $SE(3)$. It is highly nontrivial to express uncertainty and derivatives of these matrices, leading to the necessity of *Lie algebra*. By using Lie algebra we can represent the pose as a six-dimensional vector $\mathbf{X} = [\mathbf{t}^T \boldsymbol{\omega}^T]^T$ consisting of a translational part $\mathbf{t}$ and a rotational part $\boldsymbol{\omega}$. The theory in this section is inspired by and follows the notation from [8].

2.5.1 SO(3)

The *special orthogonal group* in three dimensions, abbreviated $SO(3)$, is the set of rotation matrices fulfilling

$$SO(3) = \{\mathbf{R} \in \mathcal{R}^{3\times 3} | \mathbf{R}\mathbf{R}^T = \mathbf{I}, \det(\mathbf{R}) = 1\} \tag{2.2}$$

As matrices in the $SO(3)$ are orthogonal, they fullfill $\mathbf{R}^{-1} = \mathbf{R}^T$

The $SO(3)$ matrices has a set of operations that describes how they compose each other and how they affect vectors. The composition of two rotation matrices are given by

$$\mathbf{R}_a \circ \mathbf{R}_b = \mathbf{R}_a \mathbf{R}_b \tag{2.3}$$

and an action on vectors given by

$$\mathbf{R} \cdot \mathbf{v} = \mathbf{R}\mathbf{v} \tag{2.4}$$

2.5.2 SE(3)

The *special Euclidean group* in three dimensions, abbreviated $SE(3)$, is the set of transformation matrices fullfilling

$$SE(3) = \{\mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \in \mathcal{R}^{4\times4} | \mathbf{R} \in SO(3), \mathbf{t} \in \mathcal{R}^3\} \tag{2.5}$$

where the inverse of a transformation matrix is given by

$$\mathbf{T}^{-1} = \begin{bmatrix} \mathbf{R}^T & -\mathbf{R}^T\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{2.6}$$

Like the $SO(3)$ group, $SE(3)$ group also contains a set of operation for composition with other transformation matrices and action on vectors. The composition is given by

$$\mathbf{T}_a \circ \mathbf{T}_b = \mathbf{T}_a\mathbf{T}_b \tag{2.7}$$

and actions on vectors given by

$$\mathbf{T} \cdot \mathbf{v} = \mathbf{R}\mathbf{v} + \mathbf{t} \tag{2.8}$$

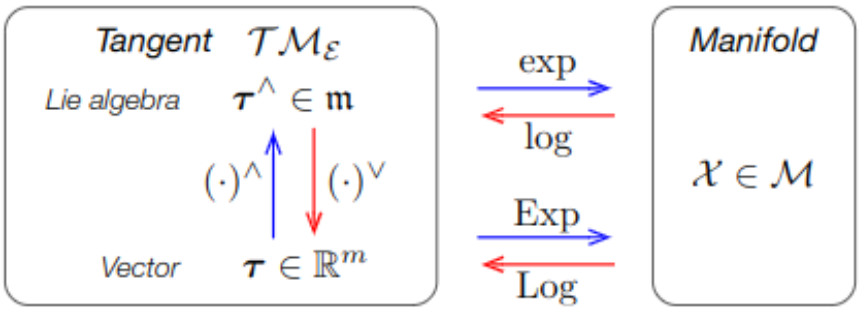


Figure 1: Summary of Lie algebra operations. Image taken from [8].

2.5.3 Lie Algebra

Lie algebra is a branch of algebra that concerns manipulating vectors that lie on strict, high-dimensional vector spaces such as $SO(3)$ and $SE(3)$, referred to as *manifolds*, in terms of operations familiar from linear algebra. Generally it is not simple to express perturbations on these strict spaces, and at least not in a way that allows us to express their derivatives and uncertainties. The Lie algebra provides a framework for manipulating vectors on these spaces, such as orientations and poses, in terms of operations familiar from linear algebra.

Lie algebra is defined in terms of Lie groups. A Lie group is defined as smooth and differentiable manifold consisting of a set $\mathcal{G}$ and a composition operation, $\circ$ that satisfies the group axioms

$$\begin{aligned} \text{Closure under } \circ &: \mathcal{X} \circ \mathcal{Y} \in \mathcal{G} \\ \text{Identity } \mathcal{E} &: \mathcal{E} \circ \mathcal{X} = \mathcal{X} \circ \mathcal{E} = \mathcal{X} \\ \text{Inverse } \mathcal{X}^{-1} &: \mathcal{X}^{-1} \circ \mathcal{X} = \mathcal{X} \circ \mathcal{X}^{-1} = \mathcal{E} \\ \text{Associativity} &: (\mathcal{X} \circ \mathcal{Y}) \circ \mathcal{Z} = \mathcal{X} \circ (\mathcal{Y} \circ \mathcal{Z}) \end{aligned} \tag{2.9}$$

for elements $\mathcal{X}, \mathcal{Y}, \mathcal{Z} \in \mathcal{G}$. Lie groups can also transform elements of other sets through what is called a group action. A group action must fullfill the axioms

$$\begin{aligned} \text{Identity} &: \mathcal{E} \cdot v = v \\ \text{Compatibility} &: (\mathcal{X} \circ \mathcal{Y}) \cdot v = \mathcal{X} \circ (\mathcal{Y} \cdot v) \end{aligned} \tag{2.10}$$

The most important functionality of Lie algebra is the ability to move between the manifold $\mathcal{M}$, the tangent manifold $\mathcal{TM}$ and the vector space $\mathcal{R}^m$, summarized in Figure 1. The conversion between vectors in $\mathcal{R}^m$ and Lie algebra elements in $\mathcal{TM}$ is done through the operators $(\,\cdot\,)^\vee$ and $(\,\cdot\,)^\wedge$. They are defined by

$$\text{Hat: } (\,\cdot\,)^\wedge : \mathcal{R}^m \to \mathfrak{m}; \quad \boldsymbol{\tau}^\wedge = \sum_{i=1}^{m} \tau_i \mathbf{E}_i \tag{2.11}$$

$$\text{Vee: } (\,\cdot\,)^\vee : \mathfrak{m} \to \mathcal{R}^m; \quad (\boldsymbol{\tau}^\wedge)^\vee = \sum_{i=1}^{m} \tau_i \mathbf{e}_i \tag{2.12}$$

where $\mathbf{E}_i$ are the generators of $\mathfrak{m}$ and $\mathbf{e}_i$ are the generators of $\mathcal{R}^m$, also known as the basis vectors. Further, the conversion between the lie algebra elements and the manifold is given by the exponential and logarithmic maps. Additionally, the `Exp` operator and the `Log` operator has been defined as composite operations converting between the vector space, $\mathcal{R}^m$ and the manifold $\mathcal{M}$. These operations are given by.

$$\exp : \mathfrak{m} \to \mathcal{M}; \quad \mathcal{X} = \exp(\boldsymbol{\tau}^\wedge) \tag{2.13}$$

$$\log : \mathcal{M} \to \mathfrak{m}; \quad \boldsymbol{\tau}^\wedge = \log(\mathcal{X}) \tag{2.14}$$

$$\text{Exp} : \mathcal{R}^m \to \mathcal{M}; \quad \mathcal{X} = \text{Exp}(\boldsymbol{\tau}) := \exp(\boldsymbol{\tau}^\wedge) \tag{2.15}$$

$$\text{Log} : \mathcal{M} \to \mathcal{R}^{\updownarrow}; \quad \boldsymbol{\tau} = \text{Log}(\mathcal{X}) := \log(\mathcal{X})^\vee \tag{2.16}$$

2.6 Nonlinear solvers

Typically the optimization back-end of a SLAM system boils down to maximum a-posteriori (MAP) estimation. These optimization problems are generally nonlinear, of raising the need for efficient nonlinear solvers. Nonlinear solvers typically follow the scheme depicted in Figure 2.

2.6.1 Gauss-Newton

The Gauss-Newton method approximates the Hessian of a quadratic problem of the form $(a - g(x))^T F(a - g(x)))$ as $H \approx G^T F G$ where G is the Jacobian of $g(x)$. Thus a single update step of the Gauss-Newton scheme is given by

$$x_{k+1} = x_k + (G^T F G)^{-1} G^T F (a - g(x_k)) \tag{2.17}$$

where $a - g(x_k)$ is often referred to as the residual, r.

2.6.2 Levenberg-Marquardt

The Levenberg-Marquardt algorithm is a trust-region method which takes basis in the Gauss-Newton method above and tries to fix its convergence problems by

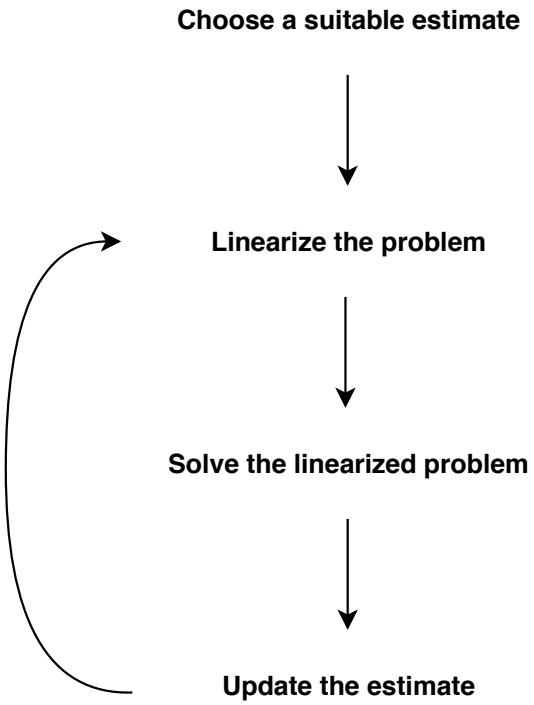


Figure 2: Nonlinear solver procedure

combining it with the well-known gradient-descent method. The minimization scheme is given by

$$x_{k+1} \leftarrow (G^T F G + \lambda D)^{-1} G^T F r \tag{2.18}$$

Notice how if we let $\lambda \to \infty$ we get the gradient descent method, while if we set $\lambda = 0$ we get the Gauss-Newton method. This compromise makes Levenberg Marquardt a more stable method than Gauss-Newton, but a well-initialized Gauss-Newton will outperform Levenberg-Marquardt.

2.7 Bayes Net

Bayes nets are directed acyclic graphs consisting of nodes $\mathcal{V}$ and egdes $\mathcal{E}$. Each node represents a random variable, x_i, and egdes represent the conditioning that expresses their relationship. Bayes nets are inspired by the probability chain rule

$$\begin{aligned} p(x_1, x_2, ..., x_n) &= p(x_1)p(x_2|x_1)...p(x_n|x_{n-1}, ..., x_1) \\ &= \prod_i p(x_i|\pi_i) \end{aligned} \tag{2.19}$$

where π_i are the parents of x_i.

2.7.1 Bayes Trees

Bayes trees, also called *junction trees* are data structures for representing cyclic graphs with a Bayes net tree structure. This is done by creating "super-nodes" in places where there are cycles. Typically this is done by utilizing *maximal cliques* as nodes and their shared nodes as *separators*. This data structure constructs the basis for iSAM2[11], which is the state-of-the-art factor graph algorithm.

2.8 Graph-based SLAM

Graph-based SLAM methods are perhaps the most popular solution to the SLAM problem in recent years. Like most other modern day SLAM methods they propose a solution to the full SLAM problem. The key idea is to represent the dependencies between the poses and landmarks involved using probabilistic graphical models, such as Bayes nets or Markov random fields. This is the base of what is known as factor graphs.

2.8.1 Front-end and Back-end

In graph-based SLAM the estimation problem is normally divided into two parts; the front-end and the back-end. The front-end takes care of all pre-processing of the data in order for it to fit into the factor graph. This includes feature extraction, short-term data association, long-term data association(for loop closure) and possibly pre-processing of other data, as for example data from an IMU in order to fit it into the graph. The back-end, on the other hand, handles state estimation. For the full SLAM problem this involves performing inference in the form of batch optimization on all the poses, which typically is done through a MAP estimator[3]. The relationship between front-end and back-end is shown in figure 3.

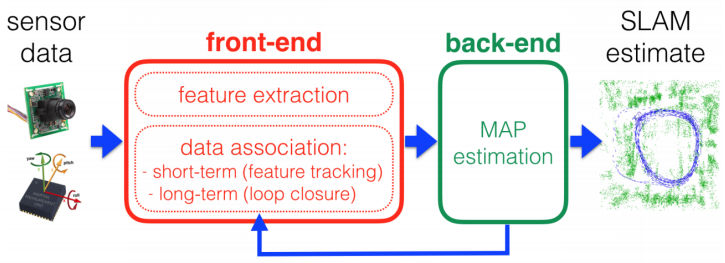


Figure 3: Front-end and back-end in a modern SLAM system. Figure from [3].

For the full SLAM problem, the goal is to estimate the set of poses and landmarks $\mathbf{X}_{1:k} = [\mathbf{X}_1^T ... \mathbf{X}_k^T]^T$ for each incoming measurement $\mathbf{z_k}$, where $\mathbf{z_k} = h_k(\mathbf{X}_k) + \mathbf{w}_k$. The set of such measurements are defined as $\mathbf{Z} = [\mathbf{z}_1^T ... \mathbf{z}_k^T]^T$. Thus, the MAP estimation can be expressed mathematically as

$$\mathbf{X}_{1:k}^{MAP} = \underset{\mathbf{X}_{1:k}}{\operatorname{argmax}}\, p(\mathbf{X}_{1:k}|\mathbf{Z}) = \underset{\mathbf{X}_{1:k}}{\operatorname{argmax}}\, \frac{p(\mathbf{Z}|\mathbf{X}_{1:k})p(\mathbf{X}_{1:k})}{p(\mathbf{Z})} \tag{2.20}$$

where $p(\mathbf{X}_{1:k})$ is the joint probability distribution between all the $\mathbf{X}_i$'s. The factor $p(\mathbf{Z})$ is simply a normalization factor and is irrelevant for the maximization problem. Therefore we can write

$$\mathbf{X}_{1:k}^{MAP} = \underset{\mathbf{X}_{1:k}}{\operatorname{argmax}}\, p(\mathbf{Z}|\mathbf{X}_{1:k})p(\mathbf{X}_{1:k}) \tag{2.21}$$

Invoking an independence assumption on the measurement yields

$$\mathbf{X}_{1:k}^{MAP} = \underset{\mathbf{X}_{1:k}}{\operatorname{argmax}}\, p(\mathbf{X}_{1:k}) \prod_{i=1}^{k} p(\mathbf{z}_i|\mathbf{X}_i) \tag{2.22}$$

since a single measurement $\mathbf{z}_i$ only depends on the state $\mathbf{X}_i$. Notice how the MAP estimator does not invoke any assumptions on either the measurement model, $h_k(\,\cdot\,)$ or the prior $p(\mathbf{X}_{1:k})$. Actually, MAP estimators, unlike Kalman filters, do not need a distinction between a motion model and an estimation model. This is a huge advantage, since this allows us to represent all of these as factors. As we'll see next, this leads to a more computationally efficient and more visually pleasing representation of the SLAM problem.

2.8.2 Factor graphs

Factor graphs are a highly convenient representation of a problem when it comes to performing inference. Formally, a factor graph is a bipartite graph $F = (\mathcal{U}, \mathcal{V}, \mathcal{E})$ with two types of nodes: factors $\phi_i \in \mathcal{U}$ and variables $x_j \in \mathcal{V}$. Egdes $e_{ij} \in \mathcal{E}$ are between factor nodes and variable nodes[5]. An example of a factor graph is shown in Figure 4. The nodes x_1, x_2, x_3 represent poses, while l_1, l_2 represent landmarks. The black dots represent the factors ϕ_i. Notice that the factors represent different sensor measurements and other constraints, such as priors in the exactly same manner. Thus factor graphs can model arbitrary interconnections between states in a general manner. The factors in a factor graph can intuitively be viewed as springs relating landmarks and poses. The goal of the MAP estimation is to find poses and landmarks that minimize the tension of the overall graph.

Actually, the factor graph framework can be extracted directly from the MAP framework, which will be investigated in this section.

The joint probability $p(\mathbf{X}_{1:k})$ in (2.22) can be rewritten using the general product rule in probability

$$\begin{aligned} p(\mathbf{X}_{1:k}) &= p(\mathbf{X}_1)p(\mathbf{X}_2|\mathbf{X}_1)p(\mathbf{X}_3|\mathbf{X}_2, \mathbf{X}_1), ..., p(\mathbf{X}_k|\mathbf{X}_{k-1}, ..., \mathbf{X}_1) \\ &= p(\mathbf{X}_1)p(\mathbf{X}_2|\mathbf{X}_1)p(\mathbf{X}_3|\mathbf{X}_2), ..., p(\mathbf{X}_k|\mathbf{X}_{k-1}) \end{aligned} \tag{2.23}$$

where the latter equation comes from the fact that all information about the next state is given completely by the previous state. It is important to note that without conditioning on the measurements, the poses $\mathbf{x}_i$ are independent of the map $\mathbf{m}$. In addition the landmarks inside the map at time step i, $\mathbf{l}_{1:m}$, are assumed independent of each other. With this information (2.23) can be

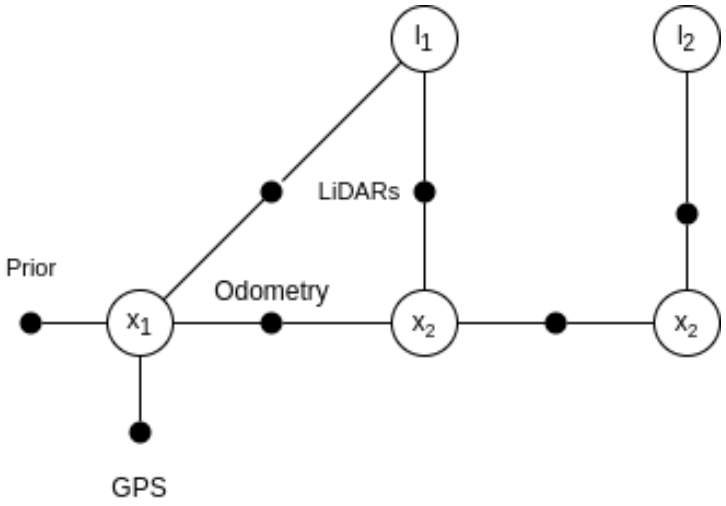


Figure 4: Example of a factor graph. Figure inspired by [5].

rewritten as

$$\begin{aligned} p(\mathbf{X}_{1:k}) =& p(\mathbf{x}_1)p(\mathbf{x}_2|\mathbf{x}_1)p(\mathbf{x}_3|\mathbf{x}_2)...p(\mathbf{x}_k|\mathbf{x}_{k-1}) \\ & \cdot p(\mathbf{m}) \\ =& p(\mathbf{x}_1)p(\mathbf{x}_2|\mathbf{x}_1)p(\mathbf{x}_3|\mathbf{x}_2)...p(\mathbf{x}_k|\mathbf{x}_{k-1}) \\ & \prod_{\mathbf{l}_j \in Z} p(\mathbf{l}_j) \end{aligned} \tag{2.24}$$

where the latter equation is means that only measured landmarks affect the states. Clearly this system is starting to look causal and on a form that can be described in a graph. In order to emphasize that it is in reality the state $\mathbf{X}_{1:k}$ that is the goal of the optimization, the measurement likelihood $p(\mathbf{z}_i|\mathbf{X}_i)$ can be expressed by the likelihood function $l(\mathbf{X}_i; \mathbf{z}_i)$ which is defined as a function proportional to the measurement likelihood:

$$l(\mathbf{X}_i; \mathbf{z}_i) \propto p(\mathbf{z}_i|\mathbf{X}_i) \tag{2.25}$$

Combining (2.23) and (2.25) with (2.22) gives the expression

$$\begin{aligned} \mathbf{X}_{1:k}^{MAP} = \underset{\mathbf{X}_{1:k}}{\operatorname{argmax}} \; & p(\mathbf{x}_1)p(\mathbf{x}_2|\mathbf{x}_1)p(\mathbf{x}_3|\mathbf{x}_2)...p(\mathbf{x}_k|\mathbf{x}_{k-1}) \\ & \cdot \prod_{\mathbf{l}_j \in Z} p(\mathbf{l}_j) \\ & \cdot \prod_{i=1}^{k} l(\mathbf{X}_i; \mathbf{z}_i) \end{aligned} \tag{2.26}$$

where the maximization still holds due to the proportionality assumption made in (2.25). Comparing these results to the factor graph in Figure 4, the prior $p(\mathbf{x}_1)$ can be seen as the factor with the "prior" text above it. The "odometry" dots are described by the chain of conditional probabilities $\prod_{i=1}^{k} p(\mathbf{x}_k|\mathbf{x}_{k-1})$. Any

measurement factors are described through $\prod_{i=1}^{k} l(\mathbf{X}_i; \mathbf{z}_i)$. Any non-existing priors can simply be modeled as uniform probability over all possibilites. Defining each of these factors by a common function $\phi_i(\mathbf{X}_i)$ gives the global factor function

$$\phi(\mathbf{X}) = \prod_i \phi_i(\mathbf{X}_i) \tag{2.27}$$

Inserting this into our MAP estimator gives

$$\begin{aligned} \mathbf{X}_{1:k}^{MAP} &= \underset{\mathbf{X}_{1:k}}{\operatorname{argmax}}\, \phi(\mathbf{X}) \\ &= \underset{\mathbf{X}_{1:k}}{\operatorname{argmax}} \prod_i \phi_i(\mathbf{X}_i) \end{aligned} \tag{2.28}$$

which enlightens the connection between the factor graph and MAP estimation.

2.8.3 Solution to the SLAM problem

Solving the MAP inference problem involves invoking some assumptions on our model. Assuming all the factors $\phi_i(\mathbf{X}_i)$ are normally distributed they can be expressed:

$$\phi_i(\mathbf{X}_i) \propto \exp\{-\frac{1}{2}||h_i(\mathbf{X}_i) - \mathbf{z}_i||^2_{\Sigma_i}\} \tag{2.29}$$

where $||e||^2_\Sigma$ denotes the Mahalanobis norm. Inserting into the MAP estimator in (2.28), it can be expressed by the nonlinear least squares problem

$$\mathbf{X}_{1:k}^{MAP} = \underset{\mathbf{X}_{1:k}}{\operatorname{argmin}} \sum_i ||h_i(\mathbf{X}_i) - \mathbf{z}_i||^2_{\Sigma_i} \tag{2.30}$$

where the argmin comes from the fact that we are instead using the negative log posterior instead. Assuming all the factors $\phi_i(\mathbf{X}_i)$ are normally distributed is equivalent to assuming that all measurement models, the process model and the priors are all distributed according to a Gaussian. Typically this problem is solved using Gauss-Newton, Levenberg-Marquardt (LM) or other nonlinear solvers, most of which involve successive linearization.

Linearization can be done by means of the Taylor expansion. Thus we can express

$$h_i(\mathbf{X}_i) = h_i(\mathbf{X}_i^0 + \boldsymbol{\Delta_i}) \approx h_i(\mathbf{X}_i^0) + H_i \Delta \mathbf{X}_i \tag{2.31}$$

where H_i is the jacobian of h_i with respect to $\mathbf{X}_i$ evaluated at the working point $\mathbf{X}_i^0$, and $\Delta \mathbf{X}_i = \mathbf{X}_i - \mathbf{X}_i^0$ is state update vector. Substituting into (2.30) gives the linear least squares

$$\Delta \mathbf{X}_{1:k}^{MAP} = \underset{\Delta \mathbf{X}_{1:k}}{\operatorname{argmin}} \sum_i ||h_i(\mathbf{X}_i^0) - (\mathbf{z}_i - H_i \Delta \mathbf{X}_i)||^2_{\Sigma_i} \tag{2.32}$$

where $(\mathbf{z}_i - H_i \Delta \mathbf{X}_i)$ is the prediction error. By rewriting the Mahalanobis norm into a regular 2-norm expression, the linear least squares become

$$\begin{aligned} \Delta \mathbf{X}_{1:k}^{MAP} &= \underset{\Delta \mathbf{X}_{1:k}}{\operatorname{argmin}} \sum_i ||(A_i \Delta \mathbf{X}_i - \mathbf{b}_i||^2_2 \\ &= \underset{\Delta \mathbf{X}_{1:k}}{\operatorname{argmin}} ||A \Delta \mathbf{X}_{1:k} - \mathbf{b}||^2_2 \end{aligned} \tag{2.33}$$

where

$$A_i = \Sigma_i^{-1/2} H_i \tag{2.34}$$

$$\mathbf{b}_i = \Sigma_i^{-1/2}(\mathbf{z}_i - h_i(\mathbf{X}_i^0)) \tag{2.35}$$

and A, known as the *measurement Jacobian*, contains the A_i-matrices and $\mathbf{b}$ contains each individual *measurement prediction error*, $\mathbf{b}_i$. With this problem, Guass-Newton gives an iterative update given by

$$\Delta\mathbf{X}_{1:k} = (A^T A)^{-1} A^T \mathbf{b} \tag{2.36}$$

while Levenberg-Marquardt gives

$$\Delta\mathbf{X}_{1:k} = (A^T A + \lambda I)^{-1} A^T \mathbf{b} \tag{2.37}$$

Clearly A will have many more rows than columns, as a single landmark often gives rise to a measurement at several time steps. As a result of this, $(A^T A)^{-1}$ is much larger than A. It is also likely that it is much more dense, which makes it desirable to calculate $\Delta\mathbf{X}_{1:k}$ without forming $(A^T A)^{-1}$. As the problem grows with time, the brute force solution suggested above will become computationally ineffective. This raises the need for efficient sparse solvers, such as QR- or Cholesky decomposition. As will be shown, exploiting the graph representation of the problem opens for intuitive and efficient formulations of these parametrizations. In graph-based methods this is done through what is called the *variable elimination algorithm*.

2.8.4 Variable Elimination

The variable elimination algorithm converts the factor graph into a Bayes net only dependent on the unknown variables $\mathbf{X}_{1:k}$. The main advantage of this is that MAP inference becomes easy, but it simplifies sampling and/or marginalization. In fact, the variable elimination algorithm is equivalent to performing sparse matrix factorization, but this will not be shown in detail in this thesis.

The variable elimination algorithm converts a factor graph of the form

$$\phi(\mathbf{X}_{1:k}) = \phi(\mathbf{X}_1, ..., \mathbf{X}_k) \tag{2.38}$$

into a factored Bayes net probability density of the form

$$p(\mathbf{X}_{1:k}) = p(\mathbf{X}_1|S_1)p(\mathbf{X}_2|S_2)...p(\mathbf{X}_k) = \prod_j p(\mathbf{X}_j|S_j) \tag{2.39}$$

where the result above is achieved by factorization by choosing the variable ordering as given by the set $\mathbf{X}_{1:k}$. S_j is the set of variables on which $\mathbf{X}_j$ is conditioned on after elimination. For the last variable, the separator will be empty, simply giving the prior $p(\mathbf{X}_k)$ for the final variable. However, it is not necessarily given that the states must be arranged in this order. In fact, the sparseness of the resulting problem is dependent on the ordering. Selecting the ordering is equivalent to reordering the columns of the A matrix in (2.33) before decomposing. Choosing an optimal ordering is a NP-hard problem, and can

be done by for example the COLAMD[4] algorithm. The variable elimination algorithm is given in algorithm 1.

Algorithm 1: Variable Elimination Algorithm

Result: Bayes net representation of factor graph
Specify elimination order $\mathcal{O}(\mathbf{X})$;
for *each node* $\mathbf{X}_i$ *according to* $\mathcal{O}(\mathbf{X})$ **do**
  $S_i \leftarrow$ all nodes involved in factors adjacent to i except i;
  $\psi(\mathbf{X}_i, \mathbf{X}_{S_i}) \leftarrow \prod_{c \in ne(i)} \phi(\mathbf{X}_i)$;
  $p(\mathbf{X}_i|S_i)\tau(S_i) \leftarrow \psi(\mathbf{X}_i, \mathbf{X}_{S_i})$;
  Add new factor $\tau(S_i)$ back into the graph;
  Insert $p(\mathbf{X}_i|S_i)$ into the Bayes net;
end

After having assembled the Bayes net as above, the following inference problem can be described by an upper triangular matrix.

$$\begin{bmatrix} \mathbf{R}_1 & \mathbf{T}_{1,2} & \mathbf{T}_{1,3} & \dots & \mathbf{T}_{1,k} \\ \mathbf{0} & \mathbf{R}_2 & \mathbf{T}_{2,3} & \dots & \vdots \\ \mathbf{0} & \dots & \ddots & \dots & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{R}_k \end{bmatrix}$$

This may be recognized as the $\mathbf{R}$-matrix of a QR-decomposition. In navigation theory it is also often referred to as the *square root information matrix*. In a general SLAM problem, many of the conditional factors, noted $\mathbf{T}_{i,j}$ will indeed be zero for a well chosen variable ordering. The conditional densities can be expressed as

$$p(\mathbf{X}_j|S_j) = k \exp\{-\frac{1}{2}||\mathbf{R}_j\mathbf{X}_j + \mathbf{T}_j S_j - \mathbf{d}_j||_2^2\} \tag{2.40}$$

where $\mathbf{d}_j$ is the new error after the factorization, $\mathbf{T}_j$ is a row vector containing the residuals and the separator S_j is a column vector containing the separator variables. This can be solved using back-substitution. For every iteration, the MAP estimate for the separator S_j will be known, so the solution is given by

$$\Delta\mathbf{X}_j^{MAP} = \mathbf{R}_j^{-1}(\mathbf{d}_j - \mathbf{T}_j S_j^{MAP}) \tag{2.41}$$

It is also worth noting that the covariance matrix is given by $\Sigma_j = (\mathbf{R}_j^T\mathbf{R}_j)^{-1}$.

2.9 Point Cloud Registration

Recall that the SLAM problem consists of a front-end and a back-end. The MAP estimation performed through factor graphs discussed so far concerns the back end. The front-end of a SLAM system concerns building the factor graph, which involves registrating how the different point cloud scans relate to each other and performing data association to recognize landmarks. The registration of point clouds are based off two different principles; *scan-matching* and *feature-based methods*.

The scan-matching problem uses entire point clouds to compute the best rigid-body transformation between them. As point clouds tend to be rather large, this becomes computationally heavy, and is prone to converge to a local minima without proper initialization. Popular scan-matching methods include the iterative closest points(ICP) algorithm and the normal distributions transform(NDT).

Feature based methods extract keypoints in the point cloud, like egdes, corners, lines or curves to calculate the transformation. It is therefore necessary that these points are repeatable so they can be utilized in consecutive scans. This also involves data association, typically by the means of *descriptors* to recognize the keypoints between scans. Using features instead of the entire point cloud significantly reduces the computational load on the system. In fact, feature based methods can be both faster and more accurate than scan-matching methods. However, they require repeatable distinct features and are therefore less robust in this manner.

2.10 Point Cloud Keypoints

Feature-based point cloud registration require robust, repeatable points in order to find correspondences between point clouds. These robust points are often referred to as keypoints. The extraction of key points associates a fixed-time cost to each point cloud, but may severely decrease the amount of points that are processed in the matching process. The keypoints in this section are discussed on the basis of whats available in the PCL[16] library.

2.10.1 Harris Corners

Harris corner detector is perhaps the most classic detection algorithm that exists. For images it calculates the image gradients to estimate the eigenvalues of each pixel in the image, before evaluating what is defined as the corner response function (CRF). The Harris corner detector has an extension to estimate corners in 3D point clouds. This is done by replacing the image gradients from the classic corner detector with the surface normals in the point cloud, and inserting into the same CRF.

2.10.2 ISS3D

Intrinsic-shape signatures(ISS)[24] 3D keypoints are based off a view-invariant 3D shape descriptor. ISS keypoints are caluclated through a *saliency measure*, given by the magnitude of the smallest eigenvalue of a point. The calculation consists of three steps:

1. Calculation of the eigenvalue decomposition of the scatter matrix, which describes the variance of the neighboring points.
2. Pruning of points with undesirable spread, given by a ratio test between the eigenvalues.
3. Picking the points with the highest saliency measure in a neighborhood.

2.11 Point Cloud Feature Descriptors

A descriptor is a structure containing information that describes data. For image systems, a descriptor is typically a representation of different features that a point contains. Such descriptions may include information about the shape, texture or intensity, among others. Descriptors are also separated into *local descriptors* and *global descriptors*.

A local descriptor contains information about the features for a single point in the point cloud, while global descriptors try to encapsulate information the entire point cloud. Typically local descriptors are used for recognition purposes while global descriptors are used for detection of objects. Therefore the rest of this section will focus on local descriptors.

2.11.1 PFH

Point feature histograms (PFH)[15] is a pose-invariant local feature descriptor which represents the surface model properties of a point. The goal is to quantify the underlying geometric properties of a point. It does this by analyzing the geometric relationship between the point and all neighbors within a sphere with radius r. The geometrical properties are calculated for all points within the neighborhood, giving a fully interconnected mesh within the radius. The fully interconnectedness is necessary to precisely model the surface around the point. The main downside with this descriptor is its computational complexity. The complexity is given by $O(nk^2)$ where n is the number of points in the point cloud and k is the number of neighbors for each point.

2.11.2 FPFH

Fast point feature histograms (FPFH)[15] is an extension to the aforementioned PFH descriptor. The main benefit of using FPFH over a normal PFH is the computational complexity. Compared to the complexity of $O(nk^2)$ for the PFH, FPFH achieves a complexity of $O(nk)$. The complexity is reduced by relaxing the interconnectedness of the PFH. This simplification makes the FPFH unable to precisely capture the surface around the point, however according to [15] it still retains most of the discriminative power. Additionally, the FPFH histograms have alot fewer histograms, severely decreasing the computational complexity associated with matching. This makes FPFH a much more desirable descriptor for navigation purposes.

3 State-of-the-art SLAM

When building a SLAM system it is important to consider what problems other state-of-the-art solutions have and what problems they have solved. Section 3.1 contains a quick description of a few state-of-the-art LiDAR SLAM systems. Afterwards, Section 3.2 contains an introduction to ISAM2, which might be the state-of-the-art graph-SLAM system when it comes to optimization.

3.1 LiDAR SLAM Systems

3.1.1 LOAM

LiDAR odometry and mapping(LOAM)[23] proposes a real-time SLAM method by dividing the algorithm into two parts. One part handles the odometry and runs on a high frequency. The other part handles the map creation, and runs on a frequency at an order of magnitude lower than the odometry part to perform fine matching and mapping.

LOAM is a feature-based method, so keypoints needs to be extracted in order to estimate motion. The keypoints are carefully selected to contain both sharp egdes and planar surface patches. Points are chosen according to a score defined by

$$c = \frac{1}{|\mathcal{S}| \cdot ||\mathbf{X}^L_{(k,i)}||} || \sum_{j \in \mathcal{S}, j \neq i} (\mathbf{X}^L_{(k,i)} - \mathbf{X}^L_{(k,j)}) || \tag{3.1}$$

where i is a point in the scan, $\mathcal{S}$ is the set of consecutive points around i and $\mathbf{X}^L_{(k,i)}$ are the coordinates of point i in LiDAR sweep k defined in a LiDAR coordinate system L. After the scores have been calculated, the points in a scan are sorted according to their c's. Egde points are selected as the points with the highest c's and planar points as the points with the lowest c's.

To ensure that points are distributed evenly within the environment, the scan is separated into four identically sized subregions. Each subregion contributes with 2 edge points and 4 planar points. To ensure good point selection, egde and planar points can only be chosen if their scores are above or below a threshold, respectively. To avoid ambiguity in point matching, selected points cannot lie in the same neighborhood. Moreover, unreliable points are avoided by requiring that no points lie on a surface patch parallell to the laser beam or on the boundary of an occluded region. The egde points and planar points are then added to the sets $\mathcal{E}$ and $\mathcal{H}$.

Let the point cloud of the previous sweep be defined as $\mathcal{P}_k$ and the reprojected point cloud as $\bar{\mathcal{P}}_k$. After identifying the egde and planar points of $\mathcal{P}_{k+1}$, the sets $\mathcal{E}_{k+1}$ and $\mathcal{H}_{k+1}$ are reprojected to the start of the sweep. Let the reprojected sets be denoted $\tilde{\mathcal{E}}_{k+1}$ and $\tilde{\mathcal{H}}_{k+1}$ This is done to minimize motion distortion, and is valid as long as constant angular and linear velocities are assumed during a sweep.

After the reprojection, egde lines are found by finding the closest neighbors of an egde point in the current sweep, $i \in \tilde{\mathcal{E}}_{k+1}$ to a point, j in the previous sweep $\bar{\mathcal{P}}_k$. Thereafter, a point l is picked as the point closest to i in the two consecutive scans to point j. The egde line correspondence of i can then be identified as (j, l) after checking that both l and j are egdes. For the planar matching, three

points are needed. As when identifying egdes, points are matched by finding the closest neigbor of a planar point in the current sweep, $i \in \tilde{\mathcal{H}}_{k+1}$, to a point, j, in the previous sweep $\bar{\mathcal{P}}_k$. Thereafter, a point l is picked as the point closest to i in the two consecutive scans to point j. The third point is found by finding the closest neigbor to i in the same scan as j, denoted m. After verifying that all the points are planar, (j, l, m) are identified as the planar correspondences of i.

After performing the matching, the point-to-line distance is given by

$$d_{\mathcal{E}} = \frac{\left|\left(\tilde{\boldsymbol{X}}^L_{(k+1,i)} - \overline{\boldsymbol{X}}^L_{(k,j)}\right) \times \left(\tilde{\boldsymbol{X}}^L_{(k+1,i)} - \overline{\boldsymbol{X}}^L_{(k,l)}\right)\right|}{\left|\overline{\boldsymbol{X}}^L_{(k,j)} - \overline{\boldsymbol{X}}^L_{(k,l)}\right|} \tag{3.2}$$

where $i \in \tilde{\mathcal{E}}_{k+1}$ and (j, l) is the corresponding egde line. The point to plane distance is given by

$$d_{\mathcal{E}} = \frac{\left|\begin{array}{c}\left(\tilde{\boldsymbol{X}}^L_{(k+1,i)} - \bar{\boldsymbol{X}}^L_{(k,j)}\right) \\ \left(\tilde{\boldsymbol{X}}^L_{(k,j)} - \bar{\boldsymbol{X}}^L_{(k,l)}\right) \times \left(\tilde{\boldsymbol{X}}^L_{(k,j)} - \bar{\boldsymbol{X}}^L_{(k,m)}\right)\end{array}\right|}{\left|\left(\tilde{\boldsymbol{X}}^L_{(k,j)} - \bar{\boldsymbol{X}}^L_{(k,l)}\right) \times \left(\tilde{\boldsymbol{X}}^L_{(k,j)} - \bar{\boldsymbol{X}}^L_{(k,m)}\right)\right|} \tag{3.3}$$

This relationship between distance and pose can be minimized using Levenberg-Marquardt to obtain T^L_{k+1}.

The mapping algorithm in LOAM maps the point cloud in world coordinates. The mapping algorithm extracts points in the exactly same manner as the odometry algorithm, however, ten times as many feature points are used. Each point cloud is then downfiltered using a voxel grid and stored in 10m cubic areas. When correspondences are found, in the same manner as above, the distances between them are minimized using Levenberg-Marquardt to fine-grain the map and the transformation.

The voxel grid downfiltering of the map, however, makes it hard to perform loop closure detection and integrate other absolute measurements such as GNSS into the system. As a result of this, LOAM performs no loop closures and the method will drift over time.

3.1.2 LeGO-LOAM

Lightweight and Ground-Optimized Lidar Odometry and Mapping (LeGO-LOAM)[17] is an extension of LOAM adapted to use on UGVs. LeGO-LOAM uses the same features as LOAM, however it differs in the way it uses these features. LeGO-LOAM also includes support for loop-closures and point cloud segmentation to robustify the selected features.

Since UGVs often have their LiDAR sensor close to the ground it is quick to identify unreliable features such as from grass and tree leaves as good features. These features need to be omitted in the estimation process. To do this, the point cloud is transformed to a range image. From the range image all the points in the ground plane is labeled as ground points. The remaining points

are then passed through a clustering process. To only keep the reliable features, only clusters containing more than 30 points are used for estimation.

LeGO-LOAM divides the issue of pose estimation into two separate optimization problems; one for the lateral states and one for the longitudinal states. Like LOAM, it extracts planar and egde features into two sets, denoted F_p^t and F_e^t respectively. The points are then matched with respect to the feature sets of the previous point cloud, denoted $\mathbb{F}_p^{t-1}$ and $\mathbb{F}_e^{t-1}$. To make the matching more robust, only points that kept its label over two estimation cycles are considered in the matching. The matched points are then used for optimization. The first step in the optimization process is to compute the lateral transformation $[t_z, \theta_{roll}, \theta_{pitch}]$ using the features matched between F_p^t and $\mathbb{F}_p^{t-1}$. In the second step the longitudinal transformation $[t_x, t_y, \theta_{yaw}]$ is computed using the the features matched between F_e^t and $\mathbb{F}_e^{t-1}$, with the estimated lateral transformation as constraints in the optimization. Both optimizations are solved using Levenberg-Marquardt.

The mapping module of LeGO-LOAM matches the set of extracted features, denoted $\{\mathbb{F}_e^t, \mathbb{F}_p^t\}$, to a surrounding point cloud map, denoted $\bar{Q}^{t-1}$, in order to give a refined pose transformation. Like in LOAM it runs on a lower frequency than the pose estimation, and uses Levenberg-Marquardt to obtain the final transformation. LeGO-LOAM differs from LOAM in the way it stores the map. LOAM stores the map as one large continuous map, while LeGO-LOAM stores the map as each individual feature set, $M^t = \{\{\mathbb{F}_e^t, \mathbb{F}_p^t\} \forall t\}$, for each pose. The surrounding map used for refining the pose can be found by choosing the k most recent feature sets, assuming no drift. Loop closure detection is then performed by using ICP to find matches in the set of individual features for each pose.

3.1.3 LIO-SAM

LiDAR-Inertial Odometry Smoothing And Mapping(LIO-SAM)[18] is a odometry estimation method that fuses LiDAR and IMU data in the estimation process. In addition to fusing IMU information it has support for maintaining global consistency through GNSS measurements and loop closures.

As LIO-SAM fuses IMU data, its internal state vector is modified to include both body velocities and IMU-biases. To fuse the IMU information into a factor graph, it applies preintegration methods. In short, this is done by collecting all the IMU measurements between two poses in the factor graph and gathering them as a single factor. IMU preintegration into the factor graph makes the optimization problem jointly optimize the IMU bias as well.

The features used in LIO-SAM is equal to the features used in LOAM. However, unlike LOAM it doesn't perform any cloud-to-cloud optimization, but rather perform fixed-lag smoothing over the most recent n *keyframes*. A keyframe, $\mathbb{F}_i$, is a LiDAR frame represented in body frame, consisting of both the feature sets extracted, meaning $\mathbb{F}_i = \{F_i^e, F_i^p\}$, where F_i^e denotes the egde features and F_i^p denotes the planar features at time step i. Keyframes are chosen when the change in robot pose, $\mathbf{x}_{i+1}$ exceeds a user-defined threshold with respect to the previous pose, $\mathbf{x}_i$. Any LiDAR frame between two keyframes are discarded. The idea of keyframes helps keep both the factor graph and the map sparse, and the memory consumption to a minimum. This is ideal for a system

to operate in real-time.

After the n most recent keyframes are extracted, $\{\mathbb{F}_{i-n}, ..., \mathbb{F}_i\}$, they are transformed into the world coordinates according to the transformations $\{\mathbf{T}_{n-1}, ..., \mathbf{T}_n\}$. The feature maps are then merged, forming a voxel map, denoted $\mathbf{M}_i$. After the map is created, the newly obtained LiDAR frame $\mathbb{F}_{i+1}$ is matched to $\mathbf{M}_i$ using the predicted transformation from the IMU, $\tilde{\mathbf{T}}_{i+1}$ as the initial transform. The rest of the matching follows that of LOAM.

In addition to the preintegrated IMU factors, LIO-SAM contains support for GPS factors. If the timestamps of the GPS measurements do not match those of the LiDAR frames, they are linearly interpolated to match the time stamp of the LiDAR frames. In addition to using GPS for global consistency, LIO-SAM also has support for loop closures. It uses the same loop closure method as discussed in the LeGO-LOAM method.

3.2 iSAM2

iSAM2 is an algorithm for doing nonlinear incremental optimization. It exploits the expressivity of factor graphs to manipulate the square root information matrix in a visual and intuitive manner, as we showed in Section 2.8.4. As the name suggests, iSAM2 is an improvement of the original algorithm iSAM[10]. The original iSAM library argued purely based on linear algebra, and the contributions of iSAM2 is a result of the data being structured and visualized in another form, namely the *Bayes tree*. Using the Bayes tree, ISAM2 is able to update only the affected variables by investigating cliques. Further, the ordering given to the variable elimination algorithm given in algorithm 1 can be found through a constrained COLAMD algorithm.

In all, the Bayes tree representation gave the opportunity to introduce *incremental variable ordering* and *fluid relinearization* as opposed to the *periodic variable reordering* and *periodic relinearization* that was used in iSAM[11]. *Incremental inference* and *Partial state updates* are additional features in ISAM2 that contributed to reducing the computational cost of the algorithm.

3.2.1 Incremental Inference

By storing the square root information matrix in the form of a Bayes tree, incremental inference can be performed simply by editing the top of the tree. This can be done because all the information flows upwards towards the root of the tree. Also, the information from a factor will not enter the variable elimination process before the first variable of that factor is eliminated. This means that a new factor cannot influence any other variables that are not beneath its variables in the Bayes tree.

3.2.2 Incremental Variable Ordering

A good variable ordering results in minimal fill-in, as mentioned in Section 2.8.4. For the Bayes tree, variable reordering can be done incrementally at each step performing incremental inference. This eliminates the need for periodic variable reordering. ISAM2 uses a constrained COLAMD algorithm to find the optimal variable ordering. The constraint is present by forcing the recently accessed

variable to the end of the ordering. This is expected to be efficient in most cases, except for large loop closures.

3.2.3 Fluid Relinearization

Fluid relinearization is performed by keeping track of the validity of the linearization point for each variable. By doing this, ISAM2 can perform the expensive step of relinearizing the model only when the current estimate of a variable deviates from its linearization point by more than a threshold, β. The fluid relinearization follows algorithm

Algorithm 2: Fluid Relinearization

In: Linearization point Θ, delta Δ Out: Updated linearization point Θ, marked cliques M

1. Mark variables in Δ above threshold $\beta : J = \{\Delta_j \in \Delta || \Delta_j | > \beta\}$.
2. Update linearization point for marked variables $\Theta_J := \Theta_J \oplus \Delta_J$.
3. Mark all cliques M that involve marked variables Θ_J and all their ancestors.

3.2.4 Partial State Updates

Partial state updates involves only updating the states necessary to recover a nearly exact solution. It is inspired by the fact that changes to the top of the Bayes tree is often limited. For example, new measurements has only local effect as it only does not give much information on old variables. Partial state updates is performed by following algorithm 3. The major benefit of partial state updates is a significant saving in computation time without having a large effect on the result.

Algorithm 3: Partial State Update

In: Bayes tree $\mathcal{F}$ Out: Update Δ Starting from the root clique $C_r = F_r$:

1. The current clique $C_k = F_k : S_k$
   compute update Δ_k of frontal variables F_k from the local conditional density $P(F_k|S_k)$
2. For all variables Δ_{k_j} in Δ_k that change more than a threshold α:
   recursively process each descendant containing such a variable

4 Development Platform

4.1 Simulation Framework

Recorded indoor dataset often lack ground truth as a result of lacking bona-fide measurements as GNSS measurements. This raises the need for a realistic simulator in order to test the performance of SLAM implementations. Without some form of ground truth, there is no saying how consistent or accurate an estimate is, making it hard to quantify the performance of a SLAM implementation. It is necessary to have a simple interface for extracting and storing data and it is advantageous if the setup is simple. Since NTNU students have access to the proprietary program MATLAB, the decision of a simulation framework fell on the combination of MATLAB and Unreal Engine. This Section describes how the simulator is set up in terms of the environment and how the point cloud is generated.

Section 4.1.1 sheds some light on the different possibilities that MATLAB introduced to the simulation environment. Section 4.1.2 describe how the environment itself is set up. Section 4.1.3 describes how the the point cloud is created. Further, Section 4.1.4 will explain how the simulation is executed and how the data is represented in the simulation. Section 4.1.5 describes the process of storing the data sets in a rosbag after the simulation is complete. Lastly, Section 4.1.6 will discuss some of the challenges and the possible shortcomings of this specific simulator.

4.1.1 MATLAB and Unreal Engine

When MATLAB released MATLAB R2019b they introduced functionality for instantiating actors inside a custom Unreal Engine environment using a combination of `Automated Driving Toolbox` and `Automated Driving Toolbox Interface for Unreal Engine 4 Projects`. This made it possible to simulate "real-world" scenarios with different exteroceptive sensors and evaluate the data real-time. The supported sensors as of MATLAB R2020b are LiDARs, probabilistic radars and both fisheye and normal cameras. All sensors come with modifiable parameters and is easily exported out of the simulation environment through Simulinks `To Workspace`-block. MATLAB also provides several pre-made scenarios directed towards autonomous navigation, in addition to examples and tools for visualization.

4.1.2 Environment

The environment itself is created in Unreal Engine, whilst the initialization of the ego-vehicle and the sensor is done through Simulink. In order to create the tunnel itself a hollow cylinder is placed around the road. However, this hollow cylinder was very poor in features, which is unrealistic for most man-made tunnels. To fix this issue, the walls were covered with stone meshes with random rotation in order to create a feature-rich environment. For realism, a sidewalk was also added.

In order to verify that the SLAM algorithm works under different circumstances, two different environments are created. The simplest one describes a straight road tunnel. It is created with base in MATLABs pre-created scenario

"Straight Highway", in which the tunnel is simply wrapped around the road. In addition to the feature rich walls, several actors are placed inside the tunnel. These are namely two cones, a traffic barrel and a stationary SUV. These were placed to verify that the point cloud came out as expected. The result can be seen in Figure 5.

Figure 5: Straight tunnel scenario

The other environment is created with basis in MATLABS pre-created scenario "Curved Highway". In order to create a curved tunnel, many copies of the tunnel is made, shrinked and slightly rotated in order to simulate a curvature. The result is shown in Figure 6.

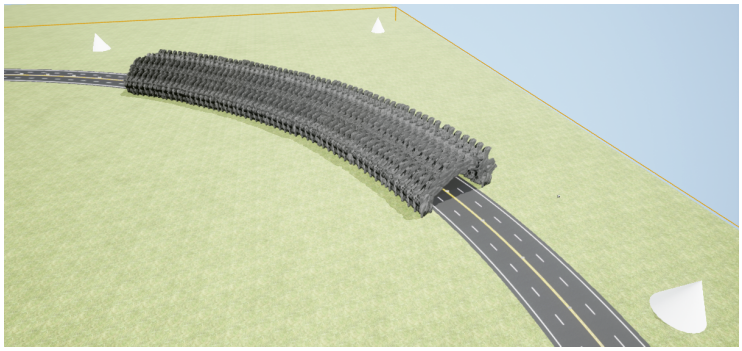


Figure 6: Curved tunnel scenario

4.1.3 Point Cloud Representation

How the point cloud is built varies for different LiDAR vendors dependent on the software of the sensor itself. Some choose to represent their measurements as a series of range-bearing measurements, while other apply some preprocessing before it is streamed out of the sensor. Some data representations include, but are not restricted to, range-bearing intensity points, XYZ intensity points and XYZ points.

For the MATLAB simulator, each point cloud is represented as a VnxHnx3 where Vn is the decided by the vertical resolution and FOV of the LiDAR, Hn

is decided by the horizontal resolution and FOV. Each point is represented as an XYZ coordinate relative to the body frame of the vehicle. Note that the simulator does not include intensity, making the usage of descriptors exploiting intensity, such as SIFT, impossible.

4.1.4 Simulation

Once the environment is created, the Simulink model required to instantiate the simulation is very simple, as shown in Figure 7. It consists of a `Simulation 3D Scene Configuration` block, and an arbitrary amount of actor and sensor blocks. The former block is the most important, and is recognized as the middle block in Figure 7. This block handles the communication with the Unreal Engine environment. It is important to configure the priority of the blocks correctly to enable communication. This is explained thoroughly in MATLABs tutorials[1]. The remaining blocks are `From`- and `To Workspace`-blocks and simply export the data for further use.

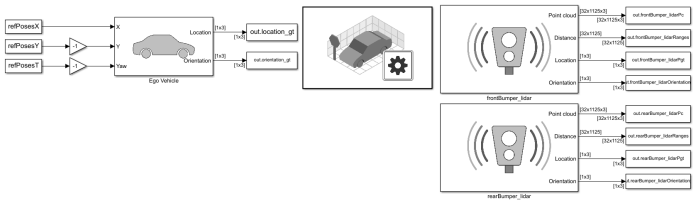


Figure 7: Simulink connected to Unreal Engine

The simulated scenario is specified through certain waypoints from which a trajectory is created. To make a smooth trajectory from the specified waypoints, they are interpolated using the `smoothPathSpline`[2]. An example of a trajectory is shown in 8. This function gives poses in 2D, which are fed as input into the $[x, y, \psi]^T$ states of the vehicle, where x and y are the x^w and y^w positions of the vehicle origin defined in reference to the world coordinates, whilst ψ is the heading of the vehicle relative to the world frame. The simulator follows a left-hand coordinate system with the x-axis pointing forward, the y-axis pointing rightwards and z-axis pointing upwards. This is the reason for the negative gain blocks input into the `Ego Vehicle`-block in Figure 7.

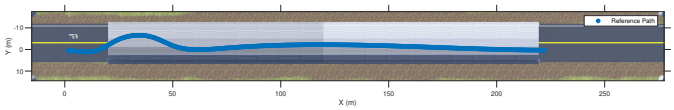


Figure 8: Trajectory of the ego-vehicle

[1] https://se.mathworks.com/help/driving/ug/how-3d-simulation-for-automated-driving-works.html

[2] https://se.mathworks.com/help/driving/ug/select-waypoints-for-3d-simulation.html

To make the simulator as realistic as possible, a truck is used as the ego-vehicle. The vehicle is equipped with two 3D-lidars in order for the vehicle to be aware of its surroundings at all times. Both LiDARs technical specifications as given in table 1. Visualized in MATLAB, the point cloud at a specific time is shown in 9. The car that can be seen in the point cloud is one of the static actors as mentioned earlier.

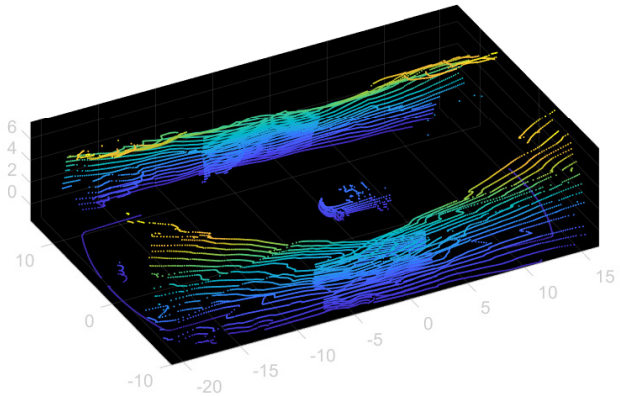


Figure 9: Point cloud in the middle of the tunnel

4.1.5 Point Cloud storage

After the simulation is complete and the data is recorded it is stored in .mat-files. In order to utilize this data optimally for our SLAM implementations the .mat-files are then read using Pythons scipy library and further stored in rosbags. In order to mimic the time delay between each full scan from the simulator, each point cloud is stored with the same frequency as specified in the simulator. This can be done since the data set also contains specs about the LiDAR itself, and the frequency is ensured using ROS' inbuilt rate class. The entire pipeline of how the simulation is performed and the data is stored is shown in Figure 10

4.1.6 Challenges and shortcomings

Unreal Engine is not a classical robotics simulator, but it is made to create games in a wide variety with a simple GUI. Additionally it is a relatively new feature, meaning it has not gone through many iterations of development. A shortcoming of the simulator is that LiDAR scans aren't provided as continuous streams, as a real-world LiDAR would have. This makes it harder to quantify the effect of motion distortion, which is an issue to deal with for LiDAR odometry methods.

Another challenge was the coordinate system of the simulator. Initially this led to confusion since the simulation environment did not act as expected.

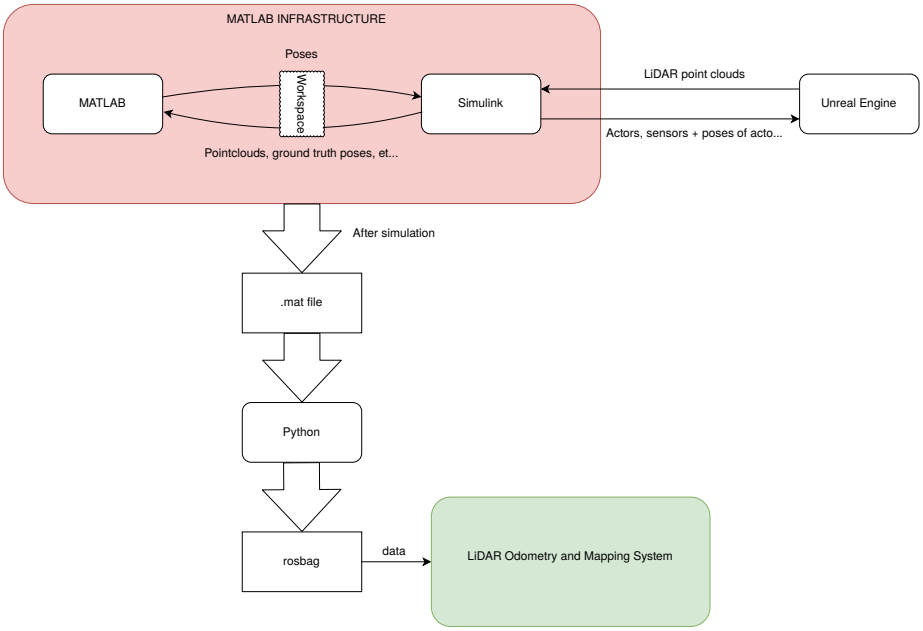


Figure 10: Pipeline showing data flow during and after simulation

4.2 Software Libraries

The software of the SLAM system is implemented in Python 3.6.12. Most SLAM systems are implemented in C++ for optimized speed, but to investigate the theory and to give a proof-of-concept this thesis is implemented in the more user-friendly Python. The rest of this section gives an overview of the different Python libraries used in the SLAM system.

4.2.1 GTSAM

GTSAM is a factor graph library implementing the iSAM2 algorithm. It exploits factor graphs and Bayes networks instead of direct sparse matrix algebra to optimize the structure of the problem for a most probable configuration. As it is meant for back-end optimization, it does not come equipped with a sensor front-end.

4.2.2 ROS

Robotic operating system (ROS)[19] is an open-source distributed and modular platform for developing robotics software. It is perhaps the most popular framework for SLAM (and many other applications) due to its extensive collection of tools, libraries and conventions combined with the fact that it can be equipped across a wide variety of platforms through a simple communication interface.

Another benefit of using ROS is that it has a standardized inbuilt data-storage type called rosbag. Rosbags allows us to read data as if it was real-time using time-stamps.

4.2.3 Point Cloud Library

Point cloud library (PCL)[16] is a highly established open-source library for point cloud manipulation. It contains functions for extracting keypoints, computing descriptors, point cloud registration and alot more, which makes it highly relevant for LiDAR SLAM. It is also compatible with ROS and has several open-source bindings to Python.

4.2.4 OpenCV

OpenCV[1] is an open-source library for image manipulation. The reason for including the OpenCV library for this thesis is because it has good descriptor matching functionality.

4.2.5 Pylie

Pylie is a small Lie-algebra library for Python created by Trym Haavardsholm. It contains classes and functions for manipulating poses and composing vectors lying on both the $SE(3)$ and $SO(3)$ groups.

4.2.6 Open3D

Open3D[26] is an open-source library for manipulating 3D data, such as point clouds. It is used supplementary to PCL since the Python bindings for PCL aren't very comprehensive. The main functionality for the open3D library is to calculate the ISS3D keypoints in the feature extraction module.

5 System Overview

The goal of this section is to give an overview of the system and show how the estimation process is executed. It starts by giving an overview of the system architecture in Section 5.1. Section 5.2 gives a brief introduction to the fast pairwise global registration algorithm.

5.1 System Architecture

An overview of the system is shown in Figure 11. It consists of three main threads running in parallell: A feature extraction thread, an odometry thread and a mapping thread. The figure shows an additional visualization module, which is set up using the 3D visualization tool `rviz`, which is a part of the ROS framework.

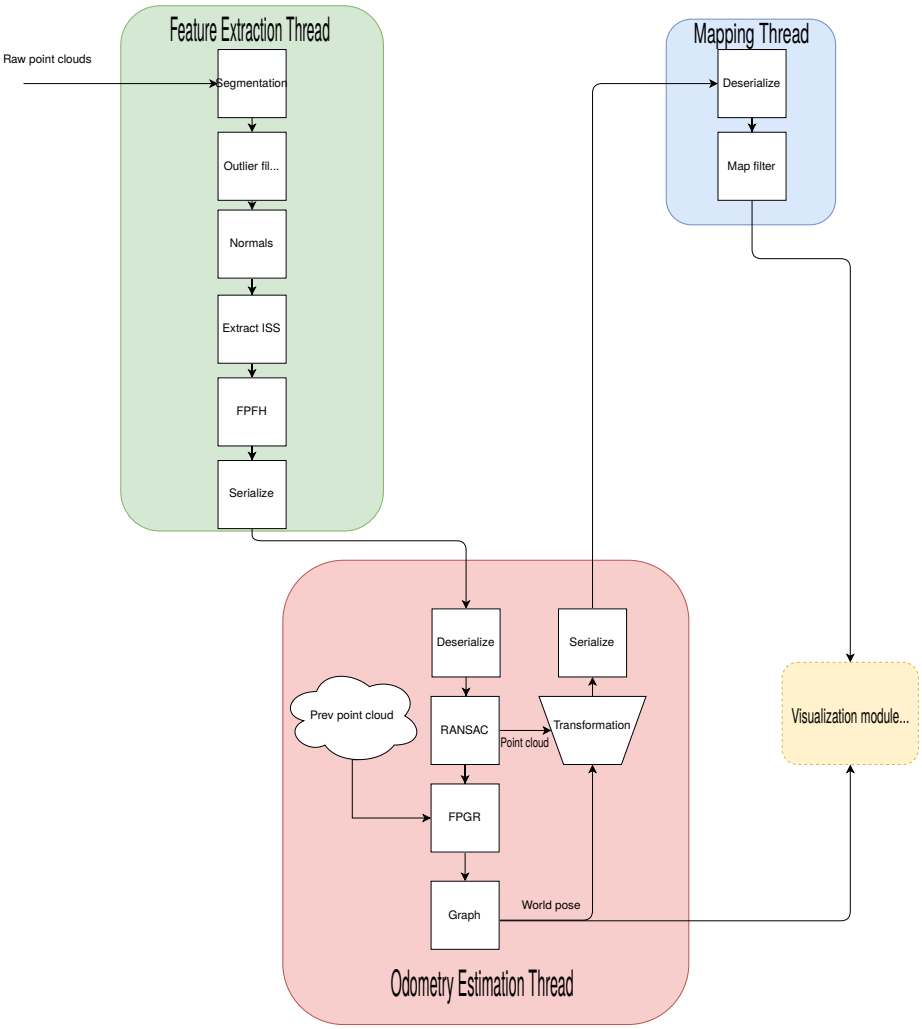


Figure 11: System architecture

5.1.1 Feature Extraction Thread

Feature extraction is an essential part of every feature-based LiDAR navigation system. The main task of the feature extraction thread is to extract robust features for the system to use for matching purposes. Finding few, but representative features can severely decrease the computational complexity of the cloud-to-cloud tracking problem. The extraction consists of several steps, with the pipeline visualized in the feature extraction thread in Figure 11.

The first step is *segmentation* of the point cloud. The goal of this step is to find the ground plane of the point cloud and remove it for the remainder of the process. After this the point cloud goes through a *statistical outlier removal* filter which removes all points where standard deviation of the average distances to its k-nearest neighbors surpasses that of the point cloud in general. This step is especially important in real-world applications to remove noisy points from the point cloud, however it showed very little effect in the simulations.

The remaining steps of the feature extraction thread concerns extracting robust features and descriptors to be used for motion tracking. Firstly, the normals are computed. The normals are necessary for the computation of alot of robust keypoints as it contains information about the surface around a point. Afterwards, the ISS3D keypoints introduced in Section 2.10.2 are calculated. This step is very important for computational complexity, as it reduces the amount of points by an order of magnitude. The ISS3D keypoints have also shown to be robust enough to still provide enough information to perform motion tracking. The final step of feature extraction is to calculate the FPFH features discussed in Section 2.11.2. The message is then serialized and published to a ROS topic for it to be used in motion tracking in the odometry estimation thread.

5.1.2 Odometry Estimation Thread

The odometry estimation thread handles the cloud-to-cloud estimation of motion. It maintains a copy of the previous point clouds as well as its feature descriptors. First, the point cloud is refined by running it through the model-fitting algorithm RANSAC. The model that showed the best performance was a cylinder-model. Afterwards, the features are matched with respect to the previous frame and motion is estimated through the fast pairwise global registration(FPGR) as suggested in [25]. The algorithm ensures robust feature matching through a *reciprocal test* and a *tuple test* before estimating the odometry using Gauss-Newton on a *Geman-McClure* penalty function. Details are provided in Section 5.2. After the odometry is estimated it is passed to a factor graph implemented in GTSAM, and pose is estimated. As a final step, the current refined point cloud is transformed to world coordinates and published to a ROS topic for the mapping thread to use.

5.1.3 Mapping Thread

The goal of the mapping thread is to refine the map for visualization purposes. It does this by representing the map as a KDTree and comparing the points in the map with the points in the incoming point cloud. If the map contains no

points in a $2m$ radius of point from the incoming point, it is added to the map. The map is then published to be used for visualization purposes.

5.2 FPGR

Fast pairwise global registration(FPGR)[25] is an algorithm for registration of three-dimensional surfaces with only partial overlapping. It requires no initialization and [25] argues that it can align surfaces at an accuracy comparable to ICP while being an order of magnitude faster. The method consists of two steps; Pruning the correspondence set and solving the optimization problem.

5.2.1 Pruning of Correspondences

The pruning of the correspondence test is done in three parts; calculating the initial correspondence set using FPFH features, finding good matches by the reciprocal test and checking compatibility using the tuple test. Let P and Q be the sets of which we wish to calculate the transformation between. Further let p and q be points in the sets, respectively. Finding the initial correspondence set is done by finding the nearest neighbor for each descriptor of a point in P, defined as $F(p)$, in Q.

Performing the reciprocal test involves calculating the the nearest neighbor for every point in Q, $F(q)$. Any correspondence from the initial correspondence set is only valid if the nearest neighbor of a descriptor in point p, $F(p)$ is $F(q)$ and the nearest neighbor of a descriptor in point q, $F(q)$ is $F(p)$. This way correspondences are only valid if the proposal of a nearest neighbor is reciprocal.

The tuple test involves checking compatibility of sets of points within the new correspondence set. This is done by randomly picking 3 correspondence pairs, $(p_1, p_2, p_3), (q_1, q_2, q_3)$ from the correspondence set, and calculating the ratio

$$\forall i \neq j \quad \tau < \frac{||p_i - p_j||}{||q_i - q_j||} < \frac{1}{\tau} \tag{5.1}$$

[25] also suggest $\tau = 0.9$. Any tuples of correspondences that pass this test is added to a final correspondence set used for the optimization.

5.2.2 Optimization

The objective function proposed by [25] is given by

$$E(\mathbf{T}, L) = \sum_{p,q \in \mathcal{K}} (l_{p,q}||p - \mathbf{T}q||^2 + \psi(l_{p,q})) \tag{5.2}$$

where

$$\psi(l_{p,q}) = \mu(\sqrt{l_{p,q}} - 1)^2 \tag{5.3}$$

$$l_{p,q} = (\frac{\mu}{\mu + ||p - \mathbf{T}q||^2})^2 \tag{5.4}$$

and μ is a tuning coefficient that increases the convexity of the problem.

To be able to express (5.2) in terms of Gauss-Newton optimization $\mathbf{T}$ is linearized locally as a 6-dimensional vector according to the Lie algebra covered

in Section 2.5.3. This gives the vector $\xi = (\omega, t)$ where ω are the rotational components and t are the translational components of the transformation. The Gauss-Newton is then solved as

$$\mathbf{T}_{k+1} = \text{Exp}(-\mathbf{J}_\mathbf{r}^T \mathbf{J}_\mathbf{r} \xi - \mathbf{J}_\mathbf{r}^T \mathbf{r}) \circ \mathbf{T}_k \tag{5.5}$$

where $\text{Exp}(\cdot)$ is defined as in Section 2.5.3, $\mathbf{r}$ is the residual vector and $\mathbf{J}_\mathbf{r}^T$ is its Jacobian. Notice that the system only optimizes the pose, meaning that it treats the structure as given.

Experimentally it was found that the Jacobian should be constricted to prioritize residuals in the longitudinal directions, otherwise the system tended to drift in the roll, pitch and altitude.

6 Results

This section represents the results achieved by the LiDAR odometry system on several simulated scenarios of a truck driving through a tunnel. The simulation environment is introduced in Section 4.1.2. The goal of the experiments is to investigate whether or not LiDAR odometry estimation is able to provide a solution to the problem, and to unveil some of the challenges related to the problem.

The experiments are shown in Section 6.1. The qualitative results are represented in Section 6.2, and the quantitative results are represented in 6.3.

6.1 Experiments

The two experiments are supposed to investigate two different spectra of challenges related to LiDAR odometry estimation. The first experiment is a relatively simple scenario of a truck driving straight through a completely empty tunnel with no actors. It will prove as a proof-of-concept and highlight some of the "basic" challenges of the proposed solution.

The second experiment shows a truck driving through a tunnel following a more complicated path with several props placed inside the tunnel. It serves to show that the proposed solution has the potential to solve more complex scenarios, and also highlight some challenges that must be resolved before the solution can be deployed in practice.

6.1.1 Simple Path Experiment

The simulated trajectory is depicted in 12. It shows the trajectory of a vehicle driving in a straight line through a tunnel. The vehicle covers a total of 220m over a time period of 60 seconds. No props were placed inside the tunnel. The experiment uses data only from the LiDAR mounted on the front bumper, which has the specifications given in 1.

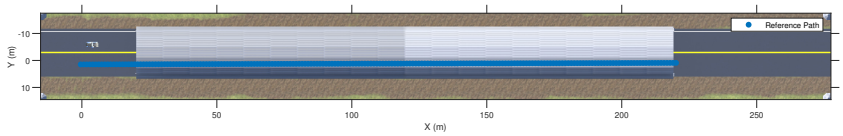


Figure 12: Experiment trajectory

6.1.2 Challenging Path Experiment

The simulated trajectory is depicted in Figure 13. It shows the trajectory of a vehicle following a path involving several turns approximately covering 330m over 60 seconds. Inside of the tunnel there are placed several props, including two small traffic cones, a traffic barrel and a static car. Notice how the path starts outside the tunnel, goes through the tunnel before turning and driving halfway back through the same tunnel. The vehicle remains inside the tunnel for approximately the first 40 seconds. After the vehicle has driven out of the

tunnel, it makes a crude turn before driving back through the same tunnel. The experiment uses data only from the LiDAR mounted on the front bumper, which has the specifications given in 1.

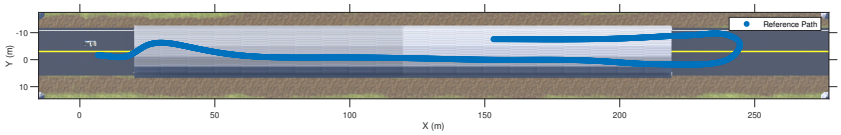


Figure 13: Experiment trajectory

Table 1: Technical specifications for the LiDAR mounted on the front bumper of the vehicle

Parameter name	**Numerical value**
Detection range	50m
Range resolution	0.002m
Vertical FOV	40°
Vertical resolution	1.25°
Horizontal FOV	180°
Horizontal resolution	0.16°
Frequency	15Hz

6.2 Qualitative results

This section investigates the qualitative results of the proposed LiDAR-based odometry estimation solution on two different data sets. The qualitative results focus on the consistency of the map and the estimated trajectory.

6.2.1 Simple Path

The map created when following the simple path is shown in 14. Notice that neither the trajectory nor the map contains any sudden skips or discontinuities.

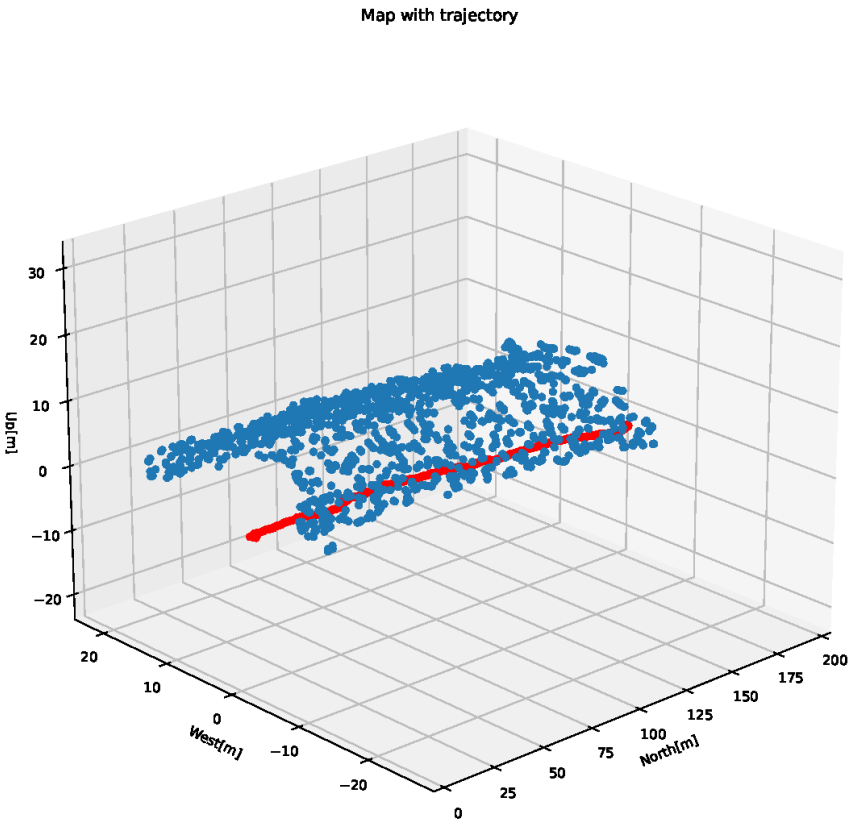


Figure 14: Estimated map with trajectory in the simple path experiment

6.2.2 Challenging Path

The map created is shown in full in Figure 15. Notice that after the vehicle has driven out of the tunnel at approximately 200m north, there is a sudden shift in the trajectory. Also notice that the map is completely continuous up until this point.

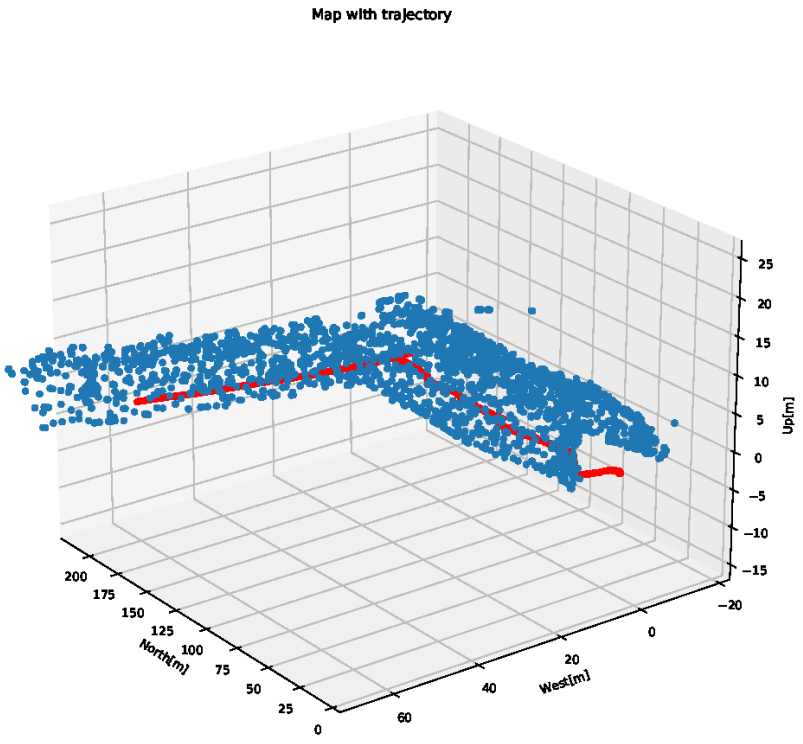


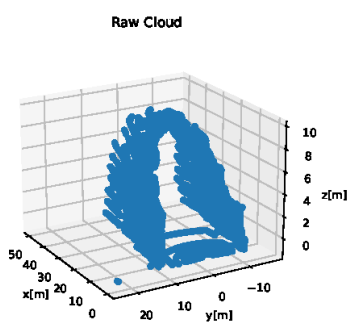
Figure 15: Estimated map with trajectory in the challenging path experiment

6.3 Quantitative results

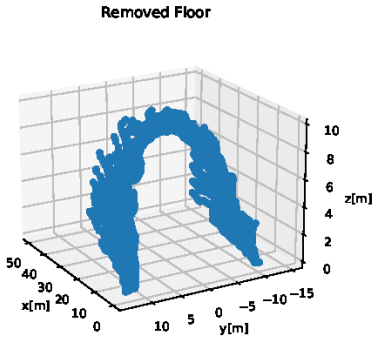
This section investigates the quantitative results of the proposed LiDAR-based odometry estimation solution on two different data sets. Section 6.3.1 shows how the feature extraction module and the RANSAC procedure severely decreases the amount of points in the point cloud while and Section 6.3.2 and Section 6.3.3 represents the results on the individual experiments.

6.3.1 Dimensionality Reduction

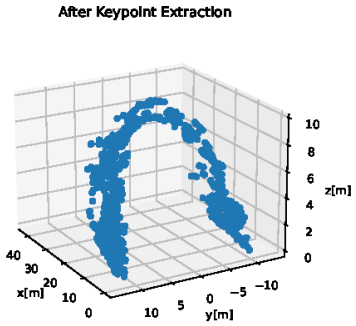
The feature extraction module decreases the dimensionality of the point cloud for each filtering step. The density of each cloud is shown in Figure 16. The dimension of the clouds is shown in table 2.



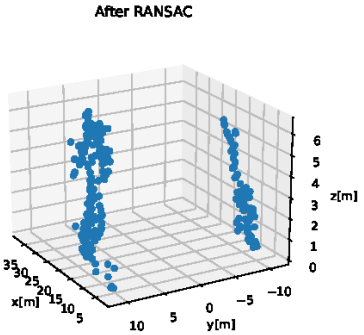
(a) Raw point cloud



(b) Cloud after removing ground plane



(c) Cloud after ISS keypoint extraction



(d) Cloud after RANSAC

Figure 16: Clouds inbetween each step in feature extraction

Table 2: The amount of 3D points in the point cloud after each step

Extraction step	**# of points**
Raw cloud	36000
Removed floor	16093
ISS extraction	683
RANSAC fit	226

The dimensionality of the problem is thereby reduced from 36000 three-dimensional points in the raw point cloud to 226 three-dimensional points to be processed in the matching procedure.

6.3.2 Simple Path

Figure 17 shows the estimated trajectory plotted against the true trajectory for the simple path experiment. The first point clouds are empty, so the origin of the estimate is set to match that of the real system. Figure 18 shows how the planar errors develop over time. The upper plot shows the error in travelled distance, while the lower plot shows the error for the heading estimate.

The upper plot in Figure 18 shows a linear trend for the development of the covered distance throughout the entire trajectory. This resulted in an root mean square error(RMSE) of 20.2m, with the error reaching its maximum of 33.4m after approximately 58 seconds. The lower plot in Figure 18 shows that the error in heading fluctuates alot around zero. The heading error has an RMSE of 2.5°, and reaches its maximum of 6.4° after 60 seconds, which is at the end of the trajectory. Notice that both of the maximum errors come at the end of the trajectory, which is when the vehicle is at the end of the tunnel and the LiDAR point cloud becomes sparse.

The timing statistics of the simple path problem is shown in table 3. The most important statistics for a system operating in the real world are the maximum execution time and the standard deviation.

Table 3: Estimation time statistics from odometry estimation system for the simple path experiment

Statistic	**Numerical value**
Mean	0.871s
Median	0.918s
Standard deviation	0.292s
Maximum	2.034s
Minimum	0.216s

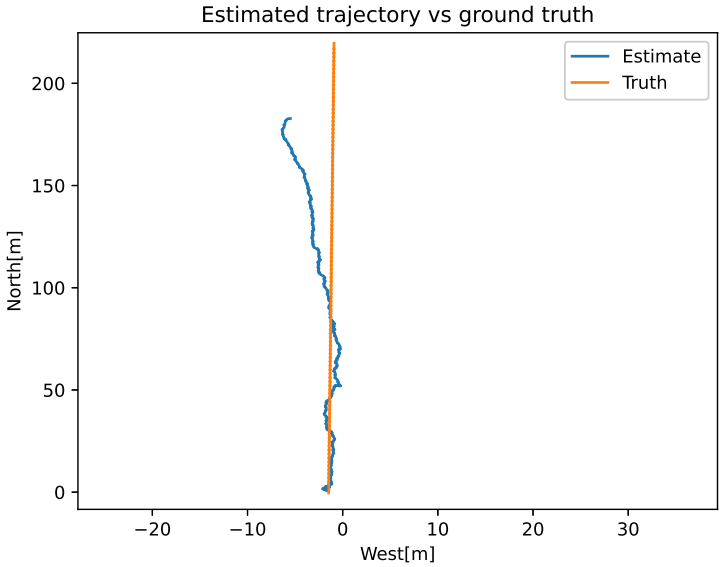


Figure 17: Estimated trajectory plotted against ground truth in the simple path experiment. The origin of the system is shifted to match that of the real trajectory.

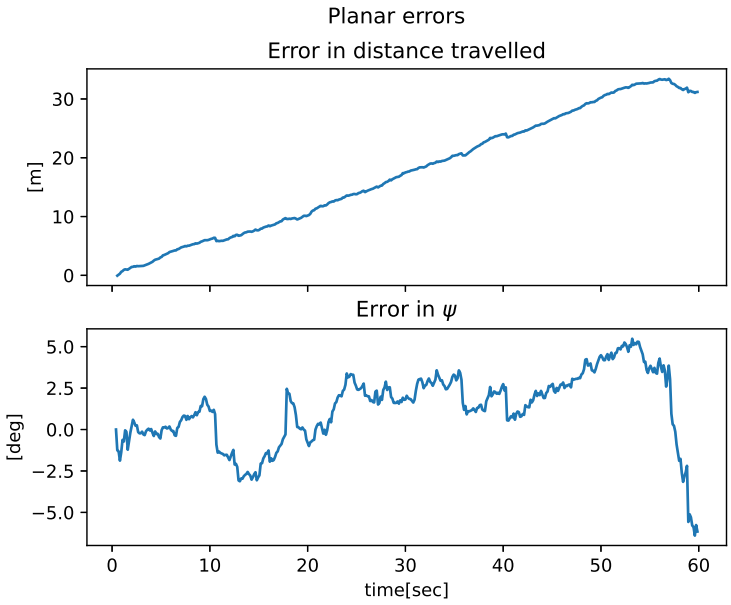


Figure 18: Errors in travelled distance and in heading for the simple path experiment

6.3.3 Challenging Path

Figure 19 shows the estimated trajectory plotted against the true trajectory for the challenging path experiment. As the first three point clouds given to the system were empty, the origin of the estimated trajectory has been shifted to match that of the true trajectory. Figure 20 shows how the planar errors develop over time. The upper plot shows how the error develops for the travelled distance and the bottom plot shows how the error develops for the heading estimates.

For approximately the first 40 seconds, the vehicle remains inside the tunnel.This is the situation where one could expect a LiDAR odometry system to function optimally. Figure 21 shows how the planar errors develop during this part of the trajectory. The RMSE in terms of distance travelled for the 40 seconds is approximately 18.9m. The trend is linearly increasing, reaching a plateau at approximately 29.5m after 35 seconds. Investigating the lower plot of Figure 21 shows that the peak error in the first 40 seconds is approximately 6.9° after 2 seconds. At this part of the trajectory the vehicle performs a turning maneuver. The RMSE of the heading estimation in the first 40 seconds is 3.0°. After the first 40 seconds, the heading estimation error exceeds 180°, as seen in Figure 20.

The timing statistics of the odometry estimation system is shown in table 4. The most important statistics for a system to be used in the real world are the maximum time and the standard deviation, as they hold information about the worst-case scenario and how large the deviation from the mean one could expect most of the time.

Table 4: Estimation time statistics from odometry estimation system for the challenging path experiment

Statistic	Numerical value
Mean	0.588s
Median	0.582s
Standard deviation	0.307s
Maximum	1.450s
Minimum	0.006s

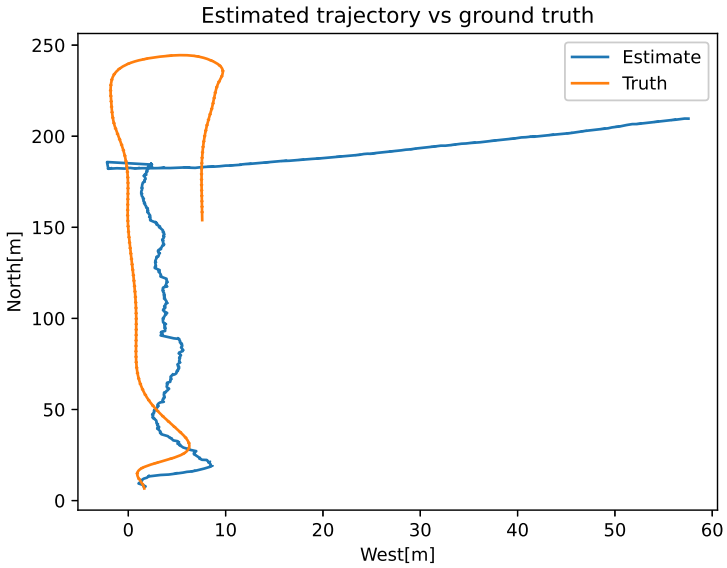


Figure 19: Estimated trajectory plotted against ground truth in the challenging path experiment. The origin of the system is shifted to match that of the real trajectory.

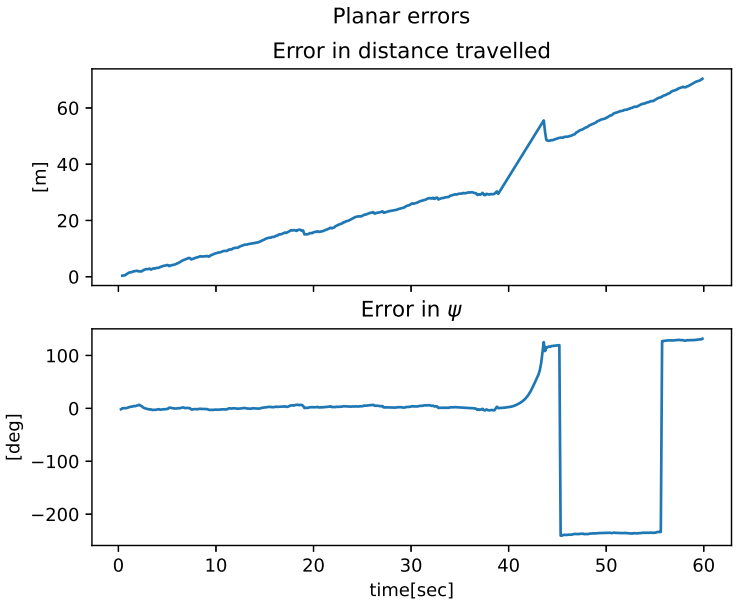


Figure 20: Errors in travelled distance and in heading for the challenging path experiment.

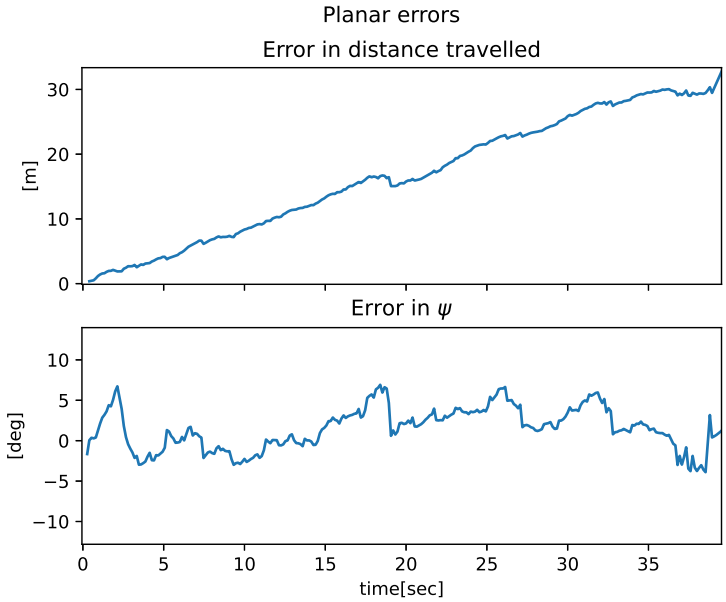


Figure 21: Errors in travelled distance and in heading the first 40 seconds of the challenging path experiment.

7 Discussion

This section discusses the results represented in Section 6. The discussion aims to provide insight to the strengths and weaknesses of the odometry system according to the challenges introduces in Section 1.3 and its applicability as part of a larger SLAM system.

7.1 Consistency

The proposed odometry estimation system is supposed to be part of a larger SLAM system providing back up to an inertial navigation system in the case of bad GNSS coverage, illustrated in this case by the vehicle driving through a tunnel. It is necessary for a backup system to be consistent, at least locally. Looking at Figure 14 shows that the system is able to produce a continuous estimate of the trajectory and map of the tunnel. However, Figure 15 shows that when the system looses track, such as when driving out of the tunnel, the vehicle suddenly estimates that it drives through the tunnel wall it mapped earlier. This situation arises because the estimation is purely only an odometry estimation, and does not consider the map at all. The consistency of the system could be increased by smoothing over a history of the last poses, and possibly also matching towards old point clouds.

7.2 Robustness

The upper part of Figure 17 and Figure 19 shows that the system initially is able to provide a somewhat good estimate of the trajectory. Investigating Figure 17 shows that the estimation has a tendency to drift even in the simplest case of driving through an empty tunnel. A reason for this might be that a single bad estimation of orientation can throw the system completely off course. The robustness could be increased by introducing smoothing over a history of poses into the estimation system.

Figure 19 shows another challenge of the estimation problem. This problem, however, is more general for all estimation systems; namely how to handle the case of lost track. After the vehicle has left the tunnel, the underlying connectivity assumption of the cloud-to-cloud odometry estimation system falls through. As a result of this it is almost completely random how it reinitializes. A system for resuming or reinitalizing track in a reasonable must be developed in order for the system to be used in practice.

The error in distance travelled is apparent in both experiments. Comparing the performance of the system on the simple scenario to the first 40 seconds of the challenging scenario should yield approximately equal results in terms of traversed distance, as it is equal in both scenarios except for the little maneuver in the beginning of the challenging trajectory. The RMSEs in travelled distance achieved in the simple scenario and the challenging scenario were 20.2m and 18.9m respectively. RANSAC introduces some form of randomness into the system, which may be why the two differ. However, the difference may also come from the fact that it is easier to predict transformations when more significant maneuvers are made, as is the case for much of the challenging scenario. The drift in distance travelled might be reduces by introducing smoothing, as men-

tioned earlier. Another way to reduce this error is to introduce other odometry information. Typically, autonomous ground vehicles include sensors such as IMUs or wheel encoders, and therefore it might be beneficial to investigate the inclusion of this information as well.

7.3 Reliability

Another important aspect of the system is to be reliable for real-time operation. It is interesting to investigate the timing statistics in the two experiments. The timing statistics of the two experiments are shown in table 3 and table 4. These tables show that there is a large difference between the amount of time needed for estimation in the two experiments. The simple trajectory experiment require almost 1.5x for estimation compared to the challenging trajectory experiment on average. A reason for this might be that the simple trajectory covers less distance in more time, giving room for more matches in each iteration. Additionally, the challenging trajectory scenario has a period where the point cloud becomes very sparse, and even completely empty. This drives down the average, but looking at the median and the standard deviation in the two experiments shows that it is the case in general as well. Investigating the maximum time needed to produce a single estimate shows that the simple scenario require 2.034 seconds while the challenging scenario require 1.450 seconds. This might be because the problem converges faster for larger maneuvers.

As a measure of reliability, the most important statistics are the worst-case execution time and the standard deviation, which in this case are 2.034 seconds and 0.292 seconds for the simple trajectory experiment, respectively. For the challenging trajectory scenario they are 1.450 seconds and 0.307 seconds, respectively. LOAM[23] argues that their system, which delivers odometry estimates at 10Hz is suitable for real-time operation. For the challenging scenario one could expect the most of odometry estimates to arrive every 0.588 ± 0.307 seconds. This heavily exceeds the delivery time of LOAM. For the simple scenario this consideration is even worse. Therefore, more efficient implementations must be investigated before it is ready to be used for real-time operation. Additionally, the proposed LiDAR odometry system does only consider cloud-to-cloud matching. As mentioned earlier, smoothing could improve the performance of the system. This would incur even more processing time needed for estimation, so the effectiveness of the proposed system needs improving.

7.4 Uncertainty

Uncertainty is an important part of estimation. The LiDAR odometry system proposed does not include any sort of uncertainty measure in its estimation. This is very undesirable for any sort of navigation system as it makes the inclusion of other sensors suboptimal. The inclusion of GPS or IMU measurements is useless without any measure of what estimates are considered "bad" and what are considered "good". Therefore, to be used in a SLAM system the LiDAR odometry estimation needs to include an uncertainty measure. Methods for estimating uncertainty between cloud matches are proposed in [20].

7.5 Future Work

Currently the motion estimation is only cloud-to-cloud based, meaning it is prone to drift. Moreover, the optimization problem in 5.2 treats the surrounding structure as a constant. Performance could probably be improved by also optimizing the structure. Also, although odometry systems have tendencies to drift since they do not optimize over the entire map, they typically implement some sort of smoothing over a history of poses. Since the end goal is to build a SLAM system for an autonomous ground vehicle, smoothing over both map and a history of poses needs to be implemented.

A SLAM system also needs to include loop closures. This involves investigation of different LiDAR-based loop closure methods. For example, methods such as LeGO-LOAM[17] implement loop closures by comparing ICP between the current and old views.

As the system is to prove as a backup system for an inertial navigation system, it would be natural to extend the integrate IMUs in the odometry estimation process. A necessity for the integration of other sensors is to provide an estimate of uncertainty in the LiDAR odometry. Integrating other sensor information could also help improve the robustness of the system in either feature-poor environments or when the point cloud becomes very sparse. Inspiration on how to do this could be found in methods such as LIO-SAM[18].

In order to be used for real-world applications the system needs to be able to provide motion estimates on a stable high frequency. To be able to provide estimates on a higher frequency, more efficient implementations need to be investigated. This could include restructuring the architecture of the system. For example, improvements could be made by moving the RANSAC step shown in the odometry estimation thread in Figure 11 into the feature extraction thread.

The main limiting factor of the work has been the programming language used for development. The PCL library bindings towards Python are community-made and none of them are especially comprehensive. This severily limits the opportunities the library originally has. Additionally, although GTSAM has bindings towards Python that are created by authors of the library itself, its bindings aren't complete either. Therefore a possible improvement is to migrate the system to C++, which is the most commonly used programming language for SLAM applications. Migration to C++ could even improve the real-time performance of the system.

8 Conclusion

This thesis has presented an analysis of some challenges related creating a SLAM system in the context of ground-based tunnel navigation. They have been investigated using a simple feature-based odometry estimation mapping system in a simulated environment. The possibility of using this system as the front-end of a larger SLAM system has also been investigated. The proposed system consists of three modules, one for feature extraction, one for odometry estimation and one for mapping. The feature extraction module severely decreases the amount of points to be processed in the other modules by an order of magnitude.

The results shows that the simulated environment provides a feature rich environment where feature-based LiDAR odometry estimation is able to provide a reasonable estimate of the trajectory. However, the proposed system tends to drift quickly in the estimation of travelled distance, and is not robust towards single bad estimates in heading. Moreover, the proposed system is not suitable for real-time operations, as the odometry estimates arrive too infrequent. Additionally, the system needs robust methods for reinitialization after track is lost.

Future work should focus on decreasing drift and improving real-time performance. A way to decrease the drift of the travelled distance estimate is to include smoothing over a history of old poses and point clouds. This could also provide robustness in the heading estimates. Drift could also be suppressed by introducing loop closures and GNSS measurements as global consistency measures into the system. Other sensory information such as from inertial sensors like IMUs are used in state-of-the art methods such as LIO-SAM, and are natural extensions to our system since it is to be part of an INS. Real-time performance could be improved by dividing the system into several estimation cycles, such as done in methods like LOAM, where the first cycle provides a rough estimate and the later cycles improve the previous estimations.

References

[1] G. Bradski. "The OpenCV Library." In: *Dr. Dobb's Journal of Software Tools* (2000).

[2] Edmund Brekke. *Fundamentals of Sensor Fusion.*

[3] C. Cadena et al. "Past, Present, and Future of Simultaneous Localization And Mapping: Towards the Robust-Perception Age." In: *IEEE Transactions on Robotics* 32.6 (2016), 1309–1332.

[4] Timothy A. Davis et al. "Algorithm 836: COLAMD, a Column Approximate Minimum Degree Ordering Algorithm." In: *ACM Trans. Math. Softw.* 30.3 (Sept. 2004), 377–380. URL: https://doi.org/10.1145/1024074.1024080.

[5] Framl Dellaert and Michael Kaess. *Factor Graphs for Robot Perception.* Vol. 6. 2017.

[6] J. Engel, T. Schöps, and D. Cremers. "LSD-SLAM: Large-Scale Direct Monocular SLAM." In: *European Conference on Computer Vision (ECCV).* 2014.

[7] Epic Games. *Unreal Engine.* Version 4.23. Jan. 30, 2019. URL: https://www.unrealengine.com/en-US/.

[8] Trym Vegard Haavardsholm. *A handbook in Visual SLAM.* 2019.

[9] Wolfgang Hess et al. "Real-Time Loop Closure in 2D LIDAR SLAM." In: *2016 IEEE International Conference on Robotics and Automation (ICRA).* 2016, pp. 1271–1278.

[10] Michael Kaess, Ananth Ranganathan, and Frank Dellaert. "iSAM: Incremental Smoothing and Mapping." In: *Robotics, IEEE Transactions on* 24 (Jan. 2009), pp. 1365 –1378.

[11] Michael Kaess et al. "ISAM2: Incremental Smoothing and Mapping Using the Bayes Tree." In: *Int. J. Rob. Res.* 31.2 (Feb. 2012), 216–235. URL: https://doi.org/10.1177/0278364911430419.

[12] Georg Klein and David Murray. "Parallel Tracking and Mapping for Small AR Workspaces." In: *Proc. Sixth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'07).* Nara, Japan, 2007.

[13] Raul Mur-Artal, J. Montiel, and Juan Tardos. "ORB-SLAM: a versatile and accurate monocular SLAM system." In: *IEEE Transactions on Robotics* 31 (Oct. 2015), pp. 1147 –1163.

[14] Open Robotics. *Gazebo.* Version 11.0.0. Jan. 30, 2019. URL: http://gazebosim.org/.

[15] Radu Bogdan Rusu, Nico Blodow, and Michael Beetz. "Fast point feature histograms (FPFH) for 3D registration." In: *2009 IEEE international conference on robotics and automation.* IEEE. 2009, pp. 3212–3217.

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

[17] Tixiao Shan and Brendan Englot. "LeGO-LOAM: Lightweight and Ground-Optimized Lidar Odometry and Mapping on Variable Terrain." In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2018, pp. 4758–4765.

[18] Tixiao Shan et al. "LIO-SAM: Tightly-coupled Lidar Inertial Odometry via Smoothing and Mapping." In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2020, pp. 5135–5142.

[19] Stanford Artificial Intelligence Laboratory et al. *Robotic Operating System*. Version ROS Melodic Morenia. May 23, 2018. URL: `https://www.ros.org`.

[20] Zuolei Sun et al. "Inferring laser-scan matching uncertainty with conditional random fields." In: *Robotics and Autonomous Systems* 60.1 (2012), pp. 83 –94. URL: `http://www.sciencedirect.com/science/article/pii/S0921889011001746`.

[21] Sebastian Thrun et al. "FastSLAM: An Efficient Solution to the Simultaneous Localization And Mapping Problem with Unknown Data." In: *Journal of Machine Learning Research* 4 (May 2004).

[22] Webots. *http://www.cyberbotics.com*. Ed. by Cyberbotics Ltd. Open-source Mobile Robot Simulation Software. URL: `http://www.cyberbotics.com`.

[23] Ji Zhang and Sanjiv Singh. "LOAM: Lidar Odometry and Mapping in Real-time." In: *Robotics: Science and Systems*. Vol. 2. 9. 2014.

[24] Y. Zhong. "Intrinsic shape signatures: A shape descriptor for 3D object recognition." In: *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops*. 2009, pp. 689–696.

[25] Qian-Yi Zhou, Jaesik Park, and V. Koltun. "Fast Global Registration." In: *ECCV*. 2016.

[26] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. "Open3D: A Modern Library for 3D Data Processing." In: *arXiv:1801.09847* (2018).