UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

A Full 2D/3D GraphSLAM System for Globally Consistent Mapping based on Manifolds

Diplomarbeit
zur Erlangung des Grades
Diplom-Informatiker
im Studiengang Informatik

vorgelegt von

Frank Neuhaus

Betreuer: Dipl.-Math.(FH) Dagmar Lang, Institut für Computervisualistik, Fachbereich Informatik, Universität Koblenz-Landau
Erstgutachter: Prof. Dr.-Ing. Dietrich Paulus, Institut für Computervisualistik, Fachbereich Informatik, Universität Koblenz-Landau
Zweitgutachter: Dipl.-Math.(FH) Dagmar Lang, Institut für Computervisualistik, Fachbereich Informatik, Universität Koblenz-Landau

Koblenz, im September 2011

Kurzfassung

In der heutigen Robotik-Forschung soll hauptsächlich die Interaktion von autonomen, mobilen Robotern mit vorher nicht bekannten Umgebungen ermöglicht werden. Eines der grundlegendsten Probleme, das in diesem Kontext gelöst werden muss, ist die Frage, wo der Roboter ist und wie seine Umgebung in unmittelbarer Nähe, aber auch an bereits besuchten Orten aussieht – das sogenannte SLAM Problem. In dieser Arbeit wird ein GraphSLAM System vorgestellt, das einen graphbasierten Lösungsansatz für dieses Problem darstellt. Ein solches System besteht aus einem Frontend und einem Backend. Das Frontend hat die Aufgabe, aus den Sensordaten einen Graphen zu konstruieren, der die relative Lage der Messungen zueinander widerspiegelt. Da sich Messungen widersprechen können, ist ein solcher Graph im Allgemeinen inkonsistent. Das Backend hat nun die Aufgabe, diesen Graphen zu optimieren, d. h. eine Konfiguration der Knoten zu bestimmen, die sich nur minimal widerspricht. Knoten repräsentieren Roboterposen, die aufgrund der enthaltenen Rotationen sog. Mannigfaltigkeiten sind und keinen gewöhnlichen Vektorraum bilden. Dies wird in der Arbeit konsequent berücksichtigt, was zu einem sehr effizienten und eleganten Optimierungsverfahren führt.

Abstract

Robotics research today is primarily about enabling autonomous, mobile robots to seamlessly interact with arbitrary, previously unknown environments. One of the most basic problems to be solved in this context is the question of where the robot is, and what the world around it, and in previously visited places looks like – the so-called simultaneous localization and mapping (SLAM) problem. We present a GraphSLAM system, which is a graph-based approach to this problem. This system consists of a frontend and a backend: The frontend's task is to incrementally construct a graph from the sensor data that models the spatial relationship between measurements. These measurements may be contradicting and therefore the graph is inconsistent in general. The backend is responsible for optimizing this graph, i. e. finding a configuration of the nodes that is least contradicting. The nodes represent poses, which do not form a regular vector space due to the contained rotations. We respect this fact by treating them as what they really are mathematically: manifolds. This leads to a very efficient and elegant optimization algorithm.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Die Vereinbarung der Arbeitsgruppe für Studien- und Abschlussarbeiten habe ich gelesen und anerkannt, insbesondere die Regelung des Nutzungsrechts.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.	ja ☒	nein ☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	ja ☒	nein ☐

Koblenz, den 28. September 2011

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem Statement

Robotics research today is primarily about enabling autonomous, mobile robots to seamlessly interact with arbitrary, previously unknown environments. This includes knowing where things are located in the world, which areas have been visited by the robot before and the question how the robot can reach a desired location. To achieve any of these goals, a robot needs to maintain a map of the environment, and it needs to know its current position inside this map at all times. This is the so-called simultaneous localization and mapping (SLAM) problem.

Variants of this problem can also occur when no actual robot is present. For example there are companies manufacturing highly-precise 3D laser range finders, such as Z+F[1] or RIEGL[2]. These laser range finders are used by architects, in forensic applications or for planning the layout of factories for instance. Up to now, the measurement process typically starts by manually attaching numerous markers to the walls, which can be detected and identified in the infra-red reflectance data provided by the devices. After that a surveyor sets the device up at carefully thought out and previously planned locations, and performs the actual measurement. When the measurements are done, the actual registration is performed offline on a computer which identifies the markers in the images and computes the required rigid-body transformations to align the markers and thus the scans. A large part of the total time required for the measurements goes into the planning phase, which includes attaching markers to the walls. It seems possible to reduce this time by avoiding the use of markers.

Note how this registration problem is essentially the same as the typical SLAM problem. The difference is that there is typically a larger distance between subse-

[1] Website: `http://www.zf-laser.com/`
[2] Website: `http://www.riegl.com/`

quent scans, and that nothing is known about any motion that occured between the scans. In other words, there is no real motion model or control input.

In this work, we will develop a system to build globally consistent maps using a graph-based formulation of this variation of the SLAM problem. We explain all relevant steps including pre-processing, scan matching and data-association, which together form the SLAM frontend, as well as the optimization of the graph using a maximum-likelihood-based approach. This optimization step is part of the SLAM backend.

Throughout this work, we take special care of the fact that the optimization is performed on poses, which are so-called manifolds, and do not form a regular $\mathbb{R}^n$ vectorspace. This leads to a very elegant and efficient optimization algorithm.

Our backend is essentially going to be an implementation of SLoM [Her]. Our frontend is an adaption of algorithms by Olson [Ols08] to make them compatible with manifolds and to make them able to deal with 3D data.

1.2 State of the Art

Algorithms for solving the (metrical) SLAM problem can coarsely be separated into two groups: online and offline methods. Online methods compute a map at the most recent timestamp, discarding all previous measurements after incorporating the contained information. In contrast, offline methods consider all the sensor readings at once and try to build a map from that.

Online methods have inherent disadvantages: Since they only see the measurements up to a certain time, it is much harder to tell whether the current measurements are outliers or not. Also since sensor readings are discarded after every time step, it is generally not possible to revise the data-association at a later point in time, which can cause the algorithm to get stuck in a local minimum. Offline methods are typically computationally much more expensive, yet their increased flexibility gives them the potential to deliver much higher quality maps than online methods.

The term *offline* has recently become a little misleading. In principle any offline approach can be used online by running the full mapping algorithm after every time step. This is inefficient of course. Yet efficiency can be regained by exploiting the problem's coherency and avoiding update rules that keep working on a majority of the previous measurements. Taking offline methods and adapting them for online use is an idea that is becoming increasingly popular, and was used in [KM07] and [GKS+10] for instance.

1.2.1 Online Methods

Virtually all online methods are instances of the Bayes filter [TBF05], which is a probabilistic framework for incorporating measurements into the current probabilistic belief of the world.

The first real solutions of the SLAM problem were typically feature-based online methods based on the Kalman filter. This approach is commonly known as EKF-SLAM and was first described by Cheeseman and Smith in [SC86], and later refined. Variants are still used today for example in [Dav03]. Every feature, as well as the robot pose is represented in the Kalman filter's state, which is modeled as a multivariate Gaussian. Therefore all covariances between the features and the robot pose are modeled. Modeling all these covariances between all features requires huge covariance matrices, which results in scalability issues. Incorrect data-associations can permanently corrupt the state or even make the filter diverge. Therefore high quality features are required, which is often an issue. Another shortcoming of the EKF-SLAM method is that the distribution of the robot position is modeled as a Gaussian, effectively restricting the algorithm to *one* data-association hypothesis only.

The FastSLAM algorithm developed by Montemerlo et al. in [MTKW02] addresses many of the issues associated with the EKF-SLAM algorithm. The key insight of the method is a clever factorization of the SLAM posterior, together with the observation the landmarks are conditionally independent given the path of the robot. This allows the problem to be decomposed into the estimation of the trajectory, and the estimation of the landmarks. The conditional independence of the landmarks allows each landmark to be estimated with its own Kalman filter. No covariances between the features have to be considered. This is the reason for the massively improved computational complexity in comparison to EKF-SLAM. The estimation of the trajectory is done with a particle filter, which is better suited for complex non-linear motion- and measurement models. It also supports multiple hypotheses on the data-association, which makes it much more capable of closing loops.

Further improvements to FastSLAM have been done by Grisetti et al. [GSB07] in an approach called GMapping: The feature-based map representation was replaced with a grid map. Additionally the number of particles needed for proper operation of the filter has been reduced significantly by incorporating measurements into the proposal distribution.

The GMapping approach is specific to 2D since it uses grid maps, which are not easily (i. e. efficiently) extensible to 3D. FastSLAM is usable in 3D in principle, however it would require a lot of particles to cover the high number of degrees of freedom of the robot pose, making it rather inefficient. EKF-SLAM easiest to use in 3D, and is in fact often used there, for example in [Wei05] or the aforementioned

[Dav03]. The core problem there remains the lack of robustness to false data-associations.

Note that all of the above mentioned algorithms can theoretically close arbitrary large loops. The only requirement is perfect data-association. For the particle-filter-based approaches, this means that at least one particle needs to have the correct data-association. This means that the larger the loop is, the more particles are needed to close it. In practice, the number of particles is always limited, and perfect data-association can not be achieved in any of the approaches.

1.2.2 Offline Methods

Being able to associate a current sensor reading with any previous sensor reading, as well as being able to revisit the data-association at a later point in time introduces a whole new sub problem to the SLAM problem: The frontend. The part that is focused on the optimization, given the data-association is called the backend.

Backends

Offline methods are dominated by graph-based approaches. This is due to the fact that graphs are naturally able to capture all sensor data collected by a robot, as well as the relations between them. The first graph-based approach was originally developed by Lu and Milios in [LM97] and later extended to 3D by Borrman, Elseberg, Nüchter et al. [BE, Nüc06, BEL+08]. They constructed a graph containing robot poses and nonlinear constraints between the poses, which were derived from scan matching. In a second step, this graph was globally optimized using least-squares.

Instead of using least-squares to minimize the error of the whole graph at once, Olson suggested [Ols08] using a variant of stochastic gradient descent. The idea was to pick individual constraints in a random order, each time distributing the error to a loop connecting the two nodes. The error distribution exploits a clever relative parametrization of the robot poses. The idea is that in each step, the error of the picked constraint is reduced, which may of course slightly increase the error of other constraints. Since a different constraint is picked every step and since the step size is decreases with every step, the method converges against some equilibrium, i. e. a point where all forces cancel out. Olson's method is amongst the most efficient ones for 2D.

Olson's error distribution scheme is not directly usable in 3D due to the non-commutativity of rotations. Yet Grisetti et al. [GGS+07] managed to adapt the algorithm to 3D in an approach called TORO. They found a different way to distribute the error which is based on the spherical linear interpolation operation

on unit quaternions. While the approach is mathematically a little different from a regular stochastic gradient descent, it still follows its spirit. Later on Grisetti's approach was refined to use a tree parametrization for the robot poses [GSB09], further improving the convergence behavior over Olson's algorithm.

All of the methods above need to perform numerical optimization on poses. While this is more or less straightforward in 2D, it is not in 3D. The reason for that is the topological structure of the space of 3D rotations, $SO(3)$, which is in fact not a normal $\mathbb{R}^n$ vector space but a so-called manifold. Hertzberg, Frese et al. [Her, FL06] developed an encapsulation of manifolds, that allows them to be treated as regular vector spaces by many existing algorithms. Hertzberg implemented a framework called SLoM that allows solving arbitrary least-squares problems on manifolds, including many 2D and 3D SLAM problems. The implementation does not include a SLAM frontend however.

Grisetti et al. have extended Hertzbergs approach by a hierarchical component in their algorithm called HOG-Man [GKS+10]. The idea of the algorithm is to maintain a hierarchy of graphs during online operation of the robot. Each hierarchy level represents a different abstraction level of the graph. The algorithm then proceeds to optimize the coarsest level. The poses in areas that were subject to significant changes are propagated to the next level. Thus the algorithm effectively limits the amount of poses that have to be optimized during online use. Note though that the computational advantages of the algorithm only shine when there are a *lot* (i. e. thousands) of poses to be optimized. Also note that in pure offline use, the algorithm offers no qualitative advantages to Hertzberg's approach [Her]: They locally converge to the same solution. However there are some computational advantages, as the hierarchy allows quickly deriving an initial guess for the higher resolution hierarchy levels.

Frontends

Gutmann et al. [GK99] propose a 2D approach which they call "Local Registration and Global Correlation". Edges between subsequent scans are automatically generated via scan matching. In order to generate loop-closing edges, they correlate a small patch of the local map with the global map, noting that a single scan alone typically does not provide enough information to reliably close loops.

The probably most common frontend for the 2D case was proposed by Olson. In order to generate edges between pairs of scans, he proposes an algorithm called "Real-Time Correlative Scan Matching" [Ols09a]. It is essentially a scan matching algorithm which can handle high amounts of uncertainty both in translation and rotation. It also yields an uncertainty estimate which can be used for the measurements on the edges of the graph.

To generate loop-closing edges, Olson [Ols09b] first estimates the uncertainty of other nodes in the graph, conditioned on the current robot pose, with an algorithm called Dijkstra projection. After that, the likelihood that the current scan overlaps with other scans is computed. Loop-closing edges are inserted when this value is above some threshold. Since the process of adding these edges is error prone, he suggests a spectral clustering approach in order to determine mutually agreeing edges. He notes that correct edges will always agree with each other, since there is only one truth. Incorrect edges however are typically wrong in many different ways, and thus these edges do not agree with each other.

Apart from these frontends for metrical, graph-based SLAM, there are also SLAM methods which only aim to construct topologically correct maps only. An example for this is an approach developed by Cummins et al. which is called FAB-MAP. It is an appearance-based method which aims to recognize previously visited places using camera images. While such topological SLAM methods are not exactly the same as the metrical frontends presented above, one can certainly consider these approaches when trying to close large loops.

1.3 Structure of this Work

The next chapter contains some mathematical background on things that are going to occur throughout the work, such as manifolds or the representation of poses.

The third chapter focuses on scan matching, i. e. the determination of a rigid-body transformation which aligns two laserscans or a laserscan and a reference map, as well as an uncertainty estimate for this transformation.

The fourth chapter is about the actual GraphSLAM system. It makes use of these local, scan-to-scan matching algorithms presented earlier in order to construct a graph describing the constraints between the scans. The chapter also explains how to optimize this graph and how to detect and handle loops.

The fifth chapter presents the results and experiments we have done to evaluate our system.

The last chapter concludes our work and presents ideas for improvements and possibilities for future work.

Chapter 2

Mathematical Basics

This chapter presents a number of mathematical basics which are needed in this work. We proceed to present an introduction to manifolds, the mathematical name for structures which locally behave like typical $\mathbb{R}^n$ vector spaces. After that, we elaborate on two particularly important manifolds, which are used in this work: the 2D and 3D pose representations. In the final section of this chapter, we present the probabilistically motivated derivation of the maximum-likelihood method, and present a way to make this method work on manifolds.

2.1 Manifolds

A wide range of algorithms including (non-)linear optimization techniques, but also the Kalman filter, particle filters and many others are designed to work on $\mathbb{R}^n$ vector spaces, i. e. they make implicit assumptions especially on the continuity and linearity of the space they work on. When these assumptions are not met, which is most commonly the case for angles or other rotational representations for example, those algorithms generally fail in some way, when no special care is taken. They may show suboptimal performance, produce states that do not make sense to the user or even diverge. To some extent, these problems can be circumvented by the user (for example by repeated renormalization of angles, quaternions etc.), but this is very cumbersome, problem-specific and does not always work. The reason for all the above mentioned problems is that some spaces are not $\mathbb{R}^n$ vector spaces, and thus should not be treated as such. In most cases, these spaces are so-called *manifolds*. Respecting this fact leads to a very elegant solution to the above mentioned problems.

A manifold is a space that locally behaves like a $\mathbb{R}^n$ vector, but globally behaves (or rather may behave) very differently. The number n denotes the *dimension* of the manifold. It is the number of independent parameters that are needed to

describe a point – sometimes also called the degrees of freedom. A trivial example for a manifold is any Euclidean $\mathbb{R}^n$ space itself. A more interesting example is a line inside any $\mathbb{R}^n$ space: It forms a one dimensional manifold, because the space has one degree of freedom, even when the line resides in three dimensional space for example.

In general all one dimensional manifolds are differently shaped curves. Even a closed curve forms a one dimensional manifold. A simple example for this is the surface of a unit circle: To unambiguously describe any point on the circle, a single angle $\theta \in [0, 2\pi[$ is required. In contrast to the line example though, this parameter itself does not form a typical $\mathbb{R}^1$ vector space: The difference between two angles is supposed to be the shortest angle between the two angles for instance, and it is not possible to achieve this using a simple difference of two elements of the space. Instead of using an angle, one could also attempt to represent each point as a two dimensional vector with a unit constraint. It does not get any better though, because now the problem is that one can not simply add or subtract any two members of the space without violating the unit constraint.

As stated above, the main characteristic of manifolds is that they locally *behave* like a $\mathbb{R}^n$ space. The mathematical term for "behaving like $\mathbb{R}^n$" is being *locally homeomorphic* to $\mathbb{R}^n$:

Definition 1. *Let X and Y be topological spaces. X is locally homeomorphic to Y if every point $x \in X$ has an open set U containing x, for which there exists a function $\varphi : U \to Y$, which is a homeomorphism, i. e. φ is a bijective, continuous map with a continuous inverse.*

Along with its domain, a function like this is called a *coordinate chart* or just *chart* [Lee02]:

Definition 2. *Let M be the manifold space. A (coordinate) chart on M is a pair (U, φ), where U is an open subset of M and φ is a local homeomorphism from U to $\mathbb{R}^n$.*

A visual explanation of these definitions can be found in Figure 2.1. Charts basically describe how points from the manifold can be mapped to $\mathbb{R}^n$ and vice versa. In order to cover the whole manifold space, multiple charts may be required. A set of these charts, whose combined domains cover the whole manifold space is called an *atlas*:

Definition 3. *Let $\mathcal{A} = \{(U_i, \varphi_i)\}$ be a set of charts corresponding to a manifold M. $\mathcal{A}$ is called an atlas of M iff $M = \bigcup_i U_i$.*

In order to obtain a smooth manifold, coordinate charts actually need to be "smoothly compatible" to each other, which means that transitioning between the

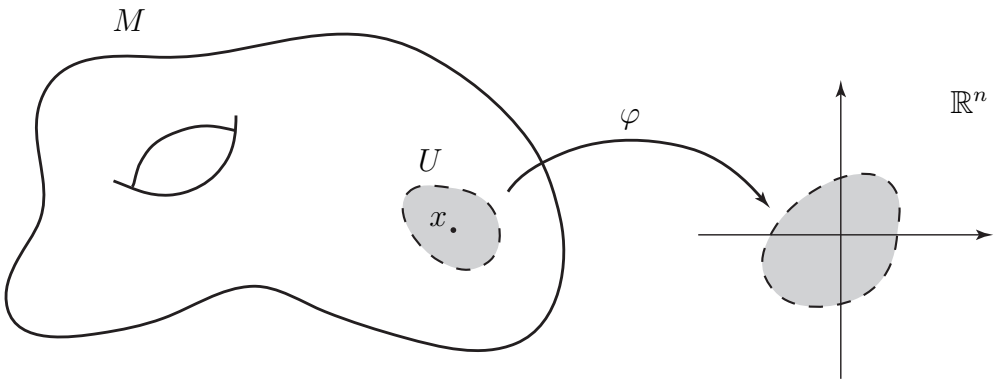


Figure 2.1: Visualization of a coordinate chart of a manifold M. An open set U around a point x is mapped to an open set in a $\mathbb{R}^n$ space.

local $\mathbb{R}^n$ spaces of different charts is smooth. We do not want to go too much into detail about that though, as we are only going to use manifolds where this requirement is fulfilled. More details on smooth manifolds can be found in [Lee02].

2.1.1 Product Manifold

It can be shown [Her] that the Cartesian product of two manifolds is a manifold again. This allows combining different manifolds to one joint manifold in a straightforward way. The manifold elements as well as the local $\mathbb{R}^n$ spaces are simply concatenated. Thus, the dimensions of the two manifolds add up: For example if a manifold with the local space $\mathbb{R}^3$ is combined with a manifold with the local space $\mathbb{R}^4$ the local space of the combined manifold is simply $\mathbb{R}^7$.

2.1.2 Operators

In order to facilitate working with the manifold, we follow [Her] and define the following operators:

$$\begin{aligned} \boxplus : M \times \mathbb{R}^n \to M \qquad & \qquad \boxminus : M \times M \to \mathbb{R}^n \\ x \boxplus \delta := \varphi_x^{-1}(\varphi_x(x) + \delta) \qquad & \qquad y \boxminus x := \varphi_x(y) - \varphi_x(x) \end{aligned} \tag{2.1}$$

The $\boxplus$ operator basically allows moving through the manifold space using small Euclidean increments, whereas the $\boxminus$ operator does the inverse: it recovers the Euclidean "difference" between two locations in the manifold space.

The idea behind these operators is to hide the mathematical details of the manifold behind an interface that resembles the typical $+$ and $-$ operators in

Euclidean spaces, thus allowing a more or less straightforward adaption of many existing methods and algorithms to manifolds.

Note that the $\boxplus$ operator is only well defined for reasonably small values of δ and the $\boxminus$ operator is only defined for x and y which are close enough together. This follows from the local homeomorphism requirement, which states that around every x there *exists* an open set U which is homeomorphic to $\mathbb{R}^n$. While U has to exist, nothing was said about its size, so it could be arbitrarily small.

2.1.3 Lie Groups

An important special case arises when the manifold together with some operator "$\cdot$" also satisfies the group axioms. A group like this is called a *Lie group*. Using the group's inverse and concatenation operators, one can choose φ_x for any $x \in M$ as:

$$\varphi_x(y) := \varphi_{id}(x^{-1}y) \tag{2.2}$$

Here, φ_{id} is a chart map centered around the group's identity element. The advantage of this definition is that one only needs to define φ_{id} instead of an whole atlas of charts. Using this definition, the $\boxplus$ and $\boxminus$ operators simplify to:

$$x \boxplus \delta := x\varphi_{id}^{-1}(\delta) \qquad\qquad x \boxminus y := \varphi_{id}(x^{-1}y) \tag{2.3}$$

2.1.4 SO(2)

The two dimensional special orthogonal group SO(2) is the set of 2D rotation matrices, which can be written as:

$$SO(2) = \{\mathbf{R} \in \mathbb{R}^{2\times 2} \,|\, \mathbf{R}\mathbf{R}^{\mathrm{T}} = \mathbf{R}^{\mathrm{T}}\mathbf{R} = \mathbf{I}_2 \wedge \det \mathbf{R} = 1\} \tag{2.4}$$

Since $SO(2)$ together with the matrix multiplication forms a Lie group, we only need to define a chart φ_{id}. Since the group only has one degree of freedom, this chart maps to $\mathbb{R}^1$. We define it as follows:

$$\varphi_{\mathrm{id}} : SO(2) \to \mathbb{R} : \varphi_{\mathrm{id}}(\mathbf{R}) = \operatorname{atan2}(R_{2,1}, R_{1,1}) \tag{2.5}$$

$$\varphi_{\mathrm{id}}^{-1} : \mathbb{R} \to SO(2) : \varphi_{\mathrm{id}}^{-1}(\delta) = \begin{pmatrix} \cos\delta & -\sin\delta \\ \sin\delta & \cos\delta \end{pmatrix} \tag{2.6}$$

The function φ_{id} maps a rotation matrix to the rotation angle $\delta \in]-\pi, \pi]$ encoded in the matrix.

Typically, one does not represent members of $SO(2)$ by a matrix, but simply by the encoded rotation angle. In that case, φ_{id} changes to:

$$\varphi_{\mathrm{id}} : SO(2) \to \mathbb{R} : \varphi_{\mathrm{id}}(\delta) = \operatorname{atan2}(\sin\delta, \cos\delta) = \delta - 2\pi \left\lfloor \frac{\delta + \pi}{2\pi} \right\rfloor \tag{2.7}$$

$$\varphi_{\mathrm{id}}^{-1} : \mathbb{R} \to SO(2) : \varphi_{\mathrm{id}}^{-1}(\delta) = \delta \tag{2.8}$$

Inserting into the definitions of the $\boxplus$ and $\boxminus$ operators yields:

$$x \boxplus \delta := x + \delta \qquad x \boxminus y := \operatorname{atan2}(\sin(x-y), \cos(x-y)) \tag{2.9}$$

Note how φ_{id} acts as a "normalizer" for the angle: All angles are mapped to $]-\pi, \pi]$, which ensures the well-behavedness of the $\boxminus$ operator.

2.1.5 SO(3)

In analogy to $SO(2)$, the space of rotations in 3 dimensional space, $SO(3)$, is defined as follows:

$$SO(3) = \{\mathbf{R} \in \mathbb{R}^{3\times 3} \,|\, \mathbf{R}\mathbf{R}^{\mathrm{T}} = \mathbf{R}^{\mathrm{T}}\mathbf{R} = \mathbf{I}_3 \wedge \det \mathbf{R} = 1\} \tag{2.10}$$

Just like its two dimensional counterpart, $SO(3)$ forms a Lie group and thus we only need to choose a φ_{id}.[1] Again we do not want to represent rotations directly using rotation matrices. We will use unit quaternions instead, which can be used to represent all possible rotations. We define φ_{id} as:

$$\varphi_{\mathrm{id}} : SO(3) \to \mathbb{R}^3 : \varphi_{\mathrm{id}}(\mathbf{q}) = \frac{\mathbf{q}_u}{||\mathbf{q}_u||} 2 \arctan \frac{||\mathbf{q}_u||}{\mathbf{q}_w} \tag{2.11}$$

$$\varphi_{\mathrm{id}}^{-1} : \mathbb{R}^3 \to SO(3) : \varphi_{\mathrm{id}}^{-1}(\mathbf{v}) = \begin{cases} \left(\cos(\frac{||\mathbf{v}||}{2}), \sin(\frac{||\mathbf{v}||}{2}) \frac{\mathbf{v}^{\mathrm{T}}}{||\mathbf{v}||}\right)^{\mathrm{T}} & \text{if } ||\mathbf{v}|| > 0 \\ \left(1,0,0,0\right)^{\mathrm{T}} & \text{otherwise} \end{cases} \tag{2.12}$$

This choice of φ_{id} maps rotations to a vector whose direction defines their rotation axis, and whose length defines the amount to rotate around that axis. Note that this is essentially the quaternion logarithm. However, the latter is typically defined using the arccos of $\mathbf{q}_w$. Quaternions are in fact a double cover of $SO(3)$, meaning that there are two different quaternions, representing the same rotation. These quaternions are $\mathbf{q}$ and $-\mathbf{q}$. One can imagine one to rotate in clockwise and the other in counter clockwise direction. Our definition using arctan ensures that both of these quaternions are mapped to the same vector in $\mathbb{R}^3$ [HWFS].

2.2 Pose Representations

Poses are a combination of a position and an orientation. They are ubiquitous in computer vision and robotics to describe the kinematic state [TBF05] of robots,

[1] In principle, one can choose a different φ_{id} than the one we provided, as long as it satisfies the requirements for charts. However we have not evaluated the effect of different choices on the algorithms developed later on this work.

cameras, or other sensors. Poses are essentially rigid-body transformations and thus the obvious way to represent them are transformation matrices. However there are also a number of alternatives, which are more comfortable to work with, and which are better for algorithms.

This section presents such alternatives to represent poses in both 2D and 3D, along with operators that allow composing poses, finding the relative pose between two absolute poses and inverting a pose.

Note that the positional part of a pose is a typical $\mathbb{R}^n$ vector space. The rotational part is an $SO(2)$ or $SO(3)$ space. In the previous section we have shown all of these are manifolds. Thus a pose, which can be considered as the Cartesian product of the position and the rotation is a manifold again according to the product manifold rule from Section 2.1.1.

2.2.1 2D

In 2D, a pose is most commonly [LM97] represented as a three dimensional vector containing the position and orientation of the robot:

$$\mathbf{x} = (x, y, \theta)^{\mathrm{T}} = (\mathbf{t}^{\mathrm{T}}, \theta)^{\mathrm{T}} \tag{2.13}$$

This pose can be written equivalently as the following transformation matrix:

$$\mathbf{X} = \begin{pmatrix} \cos\theta & -\sin\theta & x \\ \sin\theta & \cos\theta & y \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \mathbf{R}_\theta & \mathbf{t} \\ 0 & 1 \end{pmatrix} \tag{2.14}$$

The pose's inverse is:

$$\mathbf{x}^{-1} = (-\mathbf{R}_\theta^{\mathrm{T}}\mathbf{t}, -\theta)^{\mathrm{T}} \qquad \text{with} \qquad \mathbf{X}^{-1} = \begin{pmatrix} \mathbf{R}_\theta^{\mathrm{T}} & -\mathbf{R}_\theta^{\mathrm{T}}\mathbf{t} \\ 0 & 1 \end{pmatrix} \tag{2.15}$$

Note that $\mathbf{X}$ represents a transformation that transforms points from the local coordinate system of the robot to world coordinates (and not vice versa as one may tend to think).

The pose corresponding to the identity transformation is:

$$\mathbf{id} = \mathbf{0} \tag{2.16}$$

2.2.2 3D

In 3D, there are several different commonly used representations. In this work we use a seven dimensional vector containing the position and orientation of the robot, where the orientation is represented by a unit quaternion:

$$\mathbf{x} = (x, y, z, q_w, q_x, q_y, q_z)^{\mathrm{T}} = (\mathbf{t}^{\mathrm{T}}, \mathbf{q}^{\mathrm{T}})^{\mathrm{T}} \tag{2.17}$$

Just like in the 2D case, the equivalent transformation matrix is:

$$\mathbf{X} = \begin{pmatrix} \mathbf{R_q} & \mathbf{t} \\ 0 & 1 \end{pmatrix} \tag{2.18}$$

The pose's inverse is

$$\mathbf{x}^{-1} = (-\mathbf{R}_\mathbf{q}^{\mathrm{T}}\mathbf{t}, \mathbf{q}^{*\mathrm{T}})^{\mathrm{T}} \tag{2.19}$$

where $\mathbf{R_q}$ is the rotation matrix corresponding to the quaternion $\mathbf{q}$:

$$\mathbf{R_q} = \begin{pmatrix} 1-2q_y^2-2q_z^2 & 2q_xq_y-2q_zq_w & 2q_xq_z+2q_yq_w \\ 2q_xq_y+2q_zq_w & 1-2q_x^2-2q_z^2 & 2q_yq_z-2q_xq_w \\ 2q_xq_z-2q_yq_w & 2q_yq_z+2q_xq_w & 1-2q_x^2-2q_y^2 \end{pmatrix} \tag{2.20}$$

The pose corresponding to the identity transformation is:

$$\mathbf{id} = (0,0,0,1,0,0,0)^{\mathrm{T}} \tag{2.21}$$

An alternative way of storing the orientation would be to represent the orientation by three Euler angles for example. Since the space of 3D rotations, $SO(3)$ has three degrees of freedom, this representation is minimal. Yet this representation has a number of disadvantages: It is very difficult to compose and invert and additionally suffers from a well known singularity called the gimbal lock problem.

The unit quaternion representation we use does not expose any of these disadvantages. It is not minimal however, since it technically has four degrees of freedom, which are constrained to three by the unit constraint. This is not a problem though, since we are going to respect the fact that poses are a manifold by only accessing it via the previously defined operators $\boxplus$ and $\boxminus$.

2.2.3 Operations on Poses

Two important operations on poses are the pose compounding and differencing operations, represented by the $\oplus$ and $\ominus$ symbols respectively.

Both operators can be defined in terms of the underlying transformation matrices:

$$\mathbf{a} \oplus \mathbf{d} = \mathbf{b} \iff \mathbf{d} = \mathbf{b} \ominus \mathbf{a} \iff \mathbf{AD} = \mathbf{B} \tag{2.22}$$

Where $\mathbf{A}, \mathbf{D}$ and $\mathbf{B}$ are the transformation matrices belonging to the poses $\mathbf{a}, \mathbf{d}$ and $\mathbf{b}$ respectively. Here, $\mathbf{d}$ can be interpreted as the relative pose transformation that has to be executed in $\mathbf{a}$ to get to $\mathbf{b}$.

In the 2D case, this results in:

$$\mathbf{B} = \begin{pmatrix} \mathbf{R}_{\theta_a} & \mathbf{t}_a \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{R}_{\theta_d} & \mathbf{t}_d \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} \mathbf{R}_{\theta_a+\theta_d} & \mathbf{R}_a\mathbf{t}_d + \mathbf{t}_a \\ 0 & 1 \end{pmatrix} \tag{2.23}$$

Consequently:

$$\mathbf{b} = \mathbf{a} \oplus \mathbf{d} = \begin{pmatrix} \mathbf{R}_a \mathbf{t}_d + \mathbf{t}_a \\ \theta_a + \theta_d \end{pmatrix} = \begin{pmatrix} x_a + x_d \cos\theta_a - y_d \sin\theta_a \\ y_a + x_d \sin\theta_a + y_d \cos\theta_a \\ \theta_a + \theta_d \end{pmatrix} \tag{2.24}$$

In the 3D case, the same derivation results in

$$\mathbf{b} = \mathbf{a} \oplus \mathbf{d} = \begin{pmatrix} \mathbf{R}_{\mathbf{q}_a} \mathbf{t}_d + \mathbf{t}_a \\ \mathbf{q}_a \mathbf{q}_d \end{pmatrix} \tag{2.25}$$

where $\mathbf{q}_a$ and $\mathbf{q}_d$ are the quaternions corresponding to pose $\mathbf{a}$ and $\mathbf{b}$ and $\mathbf{R}_{\mathbf{q}_a}$ is again the rotation matrix corresponding to the quaternion $\mathbf{q}_a$.

$$\mathbf{D} = \mathbf{A}^{-1}\mathbf{B} = \begin{pmatrix} \mathbf{R}_{\theta_a}^{\mathrm{T}} & -\mathbf{R}_{\theta_a}^{\mathrm{T}} \mathbf{t}_a \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{R}_{\theta_b} & \mathbf{t}_b \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} \mathbf{R}_{\theta_b - \theta_a} & \mathbf{R}_{\theta_a}^{\mathrm{T}} (\mathbf{t}_b - \mathbf{t}_a) \\ 0 & 1 \end{pmatrix} \tag{2.26}$$

It follows that:

$$\mathbf{d} = \mathbf{b} \ominus \mathbf{a} = \begin{pmatrix} \mathbf{R}_{\theta_a}^{\mathrm{T}} (\mathbf{t}_b - \mathbf{t}_a) \\ \theta_b - \theta_a \end{pmatrix} = \begin{pmatrix} (x_b - x_a)\cos\theta_a + (y_b - y_a)\sin\theta_a \\ -(x_b - x_a)\sin\theta_a + (y_b - y_a)\cos\theta_a \\ \theta_b - \theta_a \end{pmatrix} \tag{2.27}$$

Again in the 3D case, the same derivation results in:

$$\mathbf{d} = \mathbf{b} \ominus \mathbf{a} = \begin{pmatrix} \mathbf{R}_{\mathbf{q}_a}^{\mathrm{T}} (\mathbf{t}_b - \mathbf{t}_a) \\ \mathbf{q}_a^* \mathbf{q}_b \end{pmatrix} \tag{2.28}$$

2.3 Maximum-Likelihood Method

The maximum-likelihood method is the problem of determining the parameters $\mathbf{x}$ which maximize the likelihood function $\mathrm{p}(\mathbf{x} \,|\, \mathbf{z})$, where $\mathbf{z}$ is a measurement of the parameters.[2] In other words, we are interested the parameters, that best explain the measurements we are seeing. We start off by rewriting this term as:

$$\mathrm{p}(\mathbf{x} \,|\, \mathbf{z}) = \eta \, \mathrm{p}(\mathbf{z} \,|\, \mathbf{x}) \, \mathrm{p}(\mathbf{x}) \tag{2.29}$$

$$\propto \mathrm{p}(\mathbf{z} \,|\, \mathbf{x}) \tag{2.30}$$

The first step follows from Bayes' rule. The newly introduced η is a normalizer, which is constant for all $\mathbf{x}$. The second step follows from an uninformed prior for $\mathbf{x}$,

[2] In fact sometimes one is not only interested in the likeliest parameters, but also in the whole distribution $\mathrm{p}(\mathbf{x} \,|\, \mathbf{z})$. In the Gaussian case, that is both the mean and the covariance matrix of the distribution.

which means that we assume we do not know anything about $\mathbf{x}$ and thus assume it to be uniformly distributed.

The remaining term $\mathrm{p}(\mathbf{z} \mid \mathbf{x})$, which is called the *measurement model*, is the probabilistic formulation of the so-called *measurement function* f, which explains how parameters are mapped to measurements. We model this relationship with additive Gaussian noise:

$$\mathbf{z} = \mathrm{f}(\mathbf{x}) + \omega \quad \text{with} \quad \omega \sim \mathcal{N}(\mathbf{0}, \mathbf{\Omega}^{-1}) \tag{2.31}$$

This can be written equivalently in terms of an error function e.[3]

$$\mathrm{e}_{\mathbf{z}}(\mathbf{x}) := \mathbf{z} - \mathrm{f}(\mathbf{x}) \quad \text{with} \quad \mathrm{e}_{\mathbf{z}}(\mathbf{x}) = \omega \tag{2.32}$$

We only want to consider the case where $\mathbf{x}$ and $\mathbf{z}$ are normally distributed. This is only the case when f and thus $\mathrm{e}_{\mathbf{z}}(\mathbf{x})$ is linear with Gaussian, additive noise. Unfortunately, both functions are non-linear in general. Therefore we first linearize it using a first-order Taylor expansion around $\mathbf{x} = \mathbf{x}_0$:

$$\mathrm{e}_{\mathbf{z}}(\mathbf{x}_0 + \mathbf{h}) \approx \mathrm{e}_{\mathbf{z}}(\mathbf{x}_0) + \mathbf{J}\mathbf{h} \quad \text{with} \quad \mathbf{J} = \left.\frac{\partial\, \mathrm{e}_{\mathbf{z}}(\mathbf{x})}{\partial\, \mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}_0} \tag{2.33}$$

This approximation of is exact when $\mathrm{e}_{\mathbf{z}}$ is already linear. Since $\mathrm{e}_{\mathbf{z}}(\mathbf{x}) = \omega \sim \mathcal{N}(\mathbf{0}, \mathbf{\Omega}^{-1})$, we write can write $\mathrm{p}(\mathbf{z} \mid \mathbf{x})$ as:

$$M = \mathrm{p}(\mathbf{z} \mid \mathbf{x}) = \mathrm{g}(\mathrm{e}_{\mathbf{z}}(\mathbf{x}_0) + \mathbf{J}\mathbf{h}, \mathbf{0}, \mathbf{\Theta}) \tag{2.34}$$

where $\mathrm{g}(\mathbf{x}, \mathbf{m}, \mathbf{\Theta})$ is a multivariate Gaussian with mean $\mathbf{m}$, and covariance matrix $\mathbf{\Theta}^{-1}$. For the sake of brevity, we simply call this term M from now on.

We actually want to find an $\mathbf{x}$ that maximizes M. Since we had to replace $\mathrm{e}_{\mathbf{z}}(\mathbf{x})$ by its Taylor expansion though, we can no longer directly compute *the* optimal $\mathbf{x}$. Instead, we can only compute an increment $\mathbf{h}$ which maximizes this linearized measurement model. Finding an increment $\mathbf{h}$ that maximizes M, is equivalent to minimizing its natural logarithm:

$$\begin{aligned}
\ln(M) &= \ln(\mathrm{g}(\mathrm{e}_{\mathbf{z}}(\mathbf{x_0}) + \mathbf{J}\mathbf{h}, \mathbf{0}, \mathbf{\Theta})) && (2.35)\\
&= -\frac{1}{2}(\mathrm{e}_{\mathbf{z}}(\mathbf{x}_0) + \mathbf{J}\mathbf{h})^{\mathrm{T}}\mathbf{\Theta}(\mathrm{e}_{\mathbf{z}}(\mathbf{x}_0) + \mathbf{J}\mathbf{h}) + const. && (2.36)\\
&= -\frac{1}{2}(\mathrm{e}_{\mathbf{z}}(\mathbf{x}_0)^{\mathrm{T}}\mathbf{\Theta}\mathrm{e}_{\mathbf{z}}(\mathbf{x}_0) + 2\mathbf{h}^{\mathrm{T}}\mathbf{J}^{\mathrm{T}}\mathbf{\Theta}\mathrm{e}_{\mathbf{z}}(\mathbf{x}_0) + \mathbf{h}^{\mathrm{T}}\mathbf{J}^{\mathrm{T}}\mathbf{\Theta}\mathbf{J}\mathbf{h}) + const. && (2.37)
\end{aligned}$$

[3]This formulation has the advantage that the $-$ operator, which essentially compares $\mathrm{f}(\mathbf{x})$ and $\mathbf{z}$, can be non-linear. We will make use of this in Section 2.3.2

Derive by $\mathbf{h}$ and set derivative to $\mathbf{0}$:

$$\mathbf{0} \stackrel{!}{=} \frac{\partial \ln(M)}{\partial \mathbf{h}} = -\mathbf{J}^{\mathrm{T}}\mathbf{\Theta}\mathrm{e}_{\mathbf{z}}(\mathbf{x}_0) - \mathbf{J}^{\mathrm{T}}\mathbf{\Theta}\mathbf{J}\mathbf{h} \tag{2.38}$$

$$\iff \quad \mathbf{J}^{\mathrm{T}}\mathbf{\Theta}\mathbf{J}\mathbf{h} = -\mathbf{J}^{\mathrm{T}}\mathbf{\Theta}\mathrm{e}_{\mathbf{z}}(\mathbf{x}_0) \tag{2.39}$$

The last line is a regular linear equation system of the form $\mathbf{A}\mathbf{x} = \mathbf{b}$, with $\mathbf{A}$ being symmetric in this case. Thus $\mathbf{h}$ can be computed using the Cholesky decomposition for example.

Having found this increment $\mathbf{h}$, we can now improve our guess for $\mathbf{x}$ to $\mathbf{x}_1 = \mathbf{x}_0 + \mathbf{h}$. Thus we iteratively compute a series of increments $\mathbf{h}$, until the method converges, i. e. $\mathbf{h}$ becomes sufficiently small. As a consequence, the method only converges to a local optimum, which can coincide with the global optimum if the initial guess is close enough.

2.3.1 Multiple Simultaneous Independent Measurements

So far we only considered the case where we are in possession of one measurement $\mathbf{z}$ of the parameters $\mathbf{x}$. Another important case is when we have multiple, conditionally independent measurements $\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n$: The likelihood function to be maximized then becomes:

$$\mathrm{p}(\mathbf{x} \,|\, \mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n) \tag{2.40}$$

This can be rewritten as:

$$\mathrm{p}(\mathbf{x} \,|\, \mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n) = \eta\, \mathrm{p}(\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n \,|\, \mathbf{x})\, \mathrm{p}(\mathbf{x}) \tag{2.41}$$

$$\propto \mathrm{p}(\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n \,|\, \mathbf{x}) \tag{2.42}$$

$$= \prod_{i=1}^{n} \mathrm{p}(\mathbf{z}_i \,|\, \mathbf{x}) \tag{2.43}$$

The first two steps are the same as in the previous subsection. The last step follows from the conditional independence of the measurements $\mathbf{z}_i$. The result is that the likelihood to be maximized is merely a product of the individual measurement models. In the last subsection we then proceeded to take the log-likelihood of the individual measurement models. The product thus transforms into a sum and can be simplified to:

$$\left(\sum_{i=1}^{n} \mathbf{J}_i^{\mathrm{T}}\mathbf{\Theta}_i\mathbf{J}_i\right)\mathbf{h} = \sum_{i=1}^{n} \mathbf{J}_i^{\mathrm{T}}\mathbf{\Theta}_i\mathrm{e}_{\mathbf{z}_i}(\mathbf{x}_0) \tag{2.44}$$

$$\text{with} \quad \mathbf{J}_i = \left.\frac{\partial\, \mathrm{e}_{\mathbf{z}_i}(\mathbf{x})}{\partial\, \mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}_0}$$

This is again a linear system of the form $\mathbf{A}\mathbf{x} = \mathbf{b}$ and can be solved the same fashion as in the single-measurement case.

2.3.2 Maximum-Likelihood on Manifolds

At the beginning of this section, we assumed that the parameters and the measurement are Gaussian random variables. When these spaces are no longer regular $\mathbb{R}^n$ vector spaces, the "additive" nature of this noise no longer makes sense, as one can not meaningfully add a small offset to an element of a manifold in general. Instead we can use the $\boxplus$ operator we defined in Section 2.1.2 to extend the notion of a Gaussian distribution to manifolds [HWFS]:

$$\mathcal{N}(\mu, \Sigma) := \mu \boxplus \mathcal{N}(\mathbf{0}, \Sigma) \tag{2.45}$$

Note that Σ is no longer in the space of the manifold, but in the space of the locally linear $\mathbb{R}^n$ space.

In the derivation at the beginning of this section, we introduced the error function in Equation (2.32), which we linearized using the Taylor expansion. This expansion uses the + operator which is not available on manifolds. Thus, instead of doing a Taylor expansion of $\mathrm{e}_{\mathbf{z}}(\mathbf{x})$ around $\mathbf{x} = \mathbf{x}_0$ directly in the parameter space, we will perform a Taylor expansion of $\mathrm{e}(\mathbf{x}_0 \boxplus \mathbf{h})$ around $\mathbf{h} = \mathbf{0}$:

$$\mathrm{e}_{\mathbf{z}}(\mathbf{x} \boxplus \mathbf{h}) \approx \mathrm{e}_{\mathbf{z}}(\mathbf{x} \boxplus \mathbf{0}) + \mathbf{J}\mathbf{h} = \mathrm{e}_{\mathbf{z}}(\mathbf{x}) + \mathbf{J}\mathbf{h} \tag{2.46}$$

In this case however, $\mathbf{J}$ is defined as

$$\mathbf{J} = \left.\frac{\partial\, \mathrm{e}_{\mathbf{z}}(\mathbf{x} \boxplus \mathbf{h})}{\partial\, \mathbf{h}}\right|_{\mathbf{h}=\mathbf{0}} \tag{2.47}$$

Using this new linearization of $\mathrm{e}_{\mathbf{z}}$, we can basically perform the same derivation as at the beginning of the section. There are two essential differences though:

- The increment $\mathbf{h}$ that is computed in every iteration, is now added to the current parameter estimate using the $\boxplus$ operator instead of the ordinary + operator.
- As explained at the beginning of the section, the involved normal distribution now respects the structure of the manifold. Thus, the dimension of the term itself as well as the information matrix Θ is now equal to the dimension of the $\mathbb{R}^n$ space, that the chart corresponding to the manifold maps to.

The pseudo code of the revised algorithm is given in Algorithm 1.

Algorithm 1 Algorithm for Maximum-Likelihood on Manifolds

Input: $\mathbf{x}_0$ – Initial Guess
Input: $maxIterations, \epsilon$ – Stopping criteria
Output: $\mathbf{x}_i$ – Maximum-Likelihood Solution

for $(i = 1 \rightarrow maxIterations)$
{
  $\mathbf{J} = \frac{\partial \mathrm{f}(\mathbf{x}_i \boxplus \mathbf{d})}{\partial \mathbf{d}}\Big|_{\mathbf{d}=\mathbf{0}}$
  Solve $\mathbf{J}^{\mathrm{T}}\boldsymbol{\Theta}\mathbf{J}\mathbf{h} = -\mathbf{J}^{\mathrm{T}}\boldsymbol{\Theta}\mathrm{e}(\mathbf{x}_i)$ for $\mathbf{h}$
  $\mathbf{x}_i = \mathbf{x}_{i-1} \boxplus \mathbf{h}$
  if $(||\mathbf{h}|| < \epsilon)$
    break
}
return $\mathbf{x}_i$

Chapter 3

Scan Matching

Scan matching algorithms have the primary purpose of computing a rigid-body transformation (or a relative pose) which aligns two laserscans, or a laserscan with an existing map. We proceed to present the well-known iterative closest point (ICP) algorithm, which provides local convergence for 2D and 3D point clouds. After that we will look at pre-processing algorithms that can be combined with the ICP algorithm. Finally we will look at an algorithm which also works when only a very bad initial guess is known.

For both algorithms, we will also derive Gaussian uncertainty estimates. These will be useful later on because they define a Gaussian (pseudo-)measurement of the relative pose of one scan to another. We will use these as building blocks of our SLAM graph in the next chapter.

3.1 Iterative Closest Point Algorithm

This section presents a summary of the iterative closest point (ICP) algorithm as presented in [Nüc06]. It is an algorithm to find a rigid-body transformation that best aligns two sets of points, namely the model set $\mathcal{M} = \{\mathbf{m}_i\}$ and the data set $\mathcal{D} = \{\mathbf{d}_i\}$. Given we knew which points of the one set correspond which points of the other set, the error function to be minimized would be:

$$\mathrm{E}(\mathbf{R}, \mathbf{t}) = \sum_{i=1}^{|\mathcal{M}|} \sum_{j=1}^{|\mathcal{D}|} w_{i,j} ||\mathbf{m}_i - (\mathbf{R}\mathbf{d}_j + \mathbf{t})||^2 \tag{3.1}$$

Here $\mathbf{R}$ and $\mathbf{t}$ define a rigid-body transformation and $w_{i,j}$ is a binary weight which is set to 1 if point i from the model set corresponds with point j from the data set, and 0 otherwise. An alternative but equivalent formulation considers only corresponding point pairs from the two sets instead of using a binary weight. Let

$\mathcal{C} = \{(i,j) \mid w_{i,j} = 1\}$ be the set of corresponding point tuple indices. Then we can rewrite E as:

$$\mathrm{E}(\mathbf{R}, \mathbf{t}) = \sum_{(i,j)\in\mathcal{C}} ||\mathbf{m}_i - (\mathbf{R}\mathbf{d}_j + \mathbf{t})||^2 \tag{3.2}$$

The function is obviously minimal for values of $\mathbf{R}$, $\mathbf{t}$ which transform the data set in a way that the sum of squared distances between the corresponding points is minimal.

There are a multitude of possibilities in minimizing E including generic methods such as gradient descent and the Levenberg-Marquardt algorithm [Fit01] but also problem-specific methods based on the singular-value-decomposition (SVD), quaternions, dual quaternions or the QR-decomposition. We will focus on the probably simplest method, which is based on the SVD of the correlation matrix $\mathbf{H}$ between the correspondence sets. The latter is defined as:

$$\mathbf{H} = \sum_{(i,j)\in\mathcal{C}} (\mathbf{d}_j - \bar{\mathbf{d}})(\mathbf{m}_i - \bar{\mathbf{m}})^{\mathrm{T}} \tag{3.3}$$

Here $\bar{\mathbf{d}}$ and $\bar{\mathbf{m}}$ are the means of the points in the correspondence set:

$$\bar{\mathbf{d}} = \frac{1}{|\mathcal{C}|} \sum_{(i,j)\in\mathcal{C}} \mathbf{d}_j \quad \bar{\mathbf{m}} = \frac{1}{|\mathcal{C}|} \sum_{(i,j)\in\mathcal{C}} \mathbf{m}_j \tag{3.4}$$

Now $\mathbf{R}$ and $\mathbf{t}$ can be found using:

$$\begin{aligned} &\mathbf{R} = \mathbf{V}\mathbf{U}^{\mathrm{T}} \quad \mathbf{t} = \bar{\mathbf{m}} - \mathbf{R}\bar{\mathbf{d}} \\ &\text{where} \quad \mathbf{V}\mathbf{\Lambda}\mathbf{U}^{\mathrm{T}} = \mathrm{svd}(\mathbf{H}) \end{aligned} \tag{3.5}$$

The method above immediately yields the optimal transformation to minimize E in closed form, given the correspondences are known. The problem is that the point correspondences are typically unknown. The key idea of the algorithm is to iteratively compute the point correspondences:

1. Given an initial guess for the alignment, find the tentative correspondence of each data point as its nearest neighbor in the model set. If there is no nearest neighbor within a radius of $d_{\max}$, the point is considered not to have any correspondences.

2. Find $\mathbf{R}$,$\mathbf{t}$ that minimize E using Equations (3.3), (3.4) and (3.5). Transform the data set by $\mathbf{R}$,$\mathbf{t}$.

3. Repeat the two steps above until the error is below a threshold ϵ.

The correspondence set is expected to improve over time, ultimately ending up with the correct correspondence set.

The ICP algorithm can be shown to converge *locally* with $d_{\max} = \infty$, however practically it usually also converges for finite values of $d_{\max}$. The main issue of the algorithm is that it needs a good initial guess in order to converge to the *correct* local minimum of E. How good the initial guess needs to be strongly depends on the geometry of the two point sets and also the choice of $d_{\max}$. High values of this parameter tend to increase the convergence radius of the solution, however they also lower the precision of the computed transformation because of the increased likelihood of false data-associations.

One heuristic to choose $d_{\max}$ is to use a high value in the early iterations of the algorithm to exploit the high convergence radius, gradually decreasing it in later iterations in order to increase the precision of the solution.

Other variants

There are a lot of variants of the ICP algorithm which refine certain aspects of the algorithm, such as the "trimmed ICP" (TrICP) [CSSK02] or the "metric based ICP" (MbICP) [Min05].

In a fashion similar to a k-trimmed mean filter in image processing which tries to avoid outliers in the computation of the mean of a number of values, the trimmed ICP aims to avoid false data-associations by sorting all correspondences by distance and only accepting a certain number of the closest matches.

The metric-based ICP suggests different metrics in order to find the nearest neighbor of a point. In the description of the algorithm above, we simply assumed an Euclidean metric. However it is quite obvious that this method has disadvantages in presence of rotational errors, as small rotational errors quickly result in large Euclidean distances between points.

All of the variants tend to improve the convergence behavior and/or the precision of the result. Yet the convergence stays local at all times.

3.1.1 Covariance Estimation

Borrmann et. al presented a way to compute the covariance matrix of the ICP alignment in [BEL+08] for the 3D case. Their algorithm yields a covariance estimate Σ around the computed relative pose $\mathbf{x}$. Since no manifolds were used in the cited work though, this covariance estimate is in the 7-dimensional pose space itself, and not in the locally linear 6-dimensional space of the pose manifold. In order to "convert" this covariance estimate, we propagate the pose Gaussian through the linearization of function $(\mathbf{x}, \delta) \to \mathbf{x} \boxplus \delta$ at $\delta = \mathbf{0}$. This ends up doing nothing to the mean pose itself since $\mathbf{x} \boxplus \mathbf{0} = \mathbf{x}$, but it allows to transform the Σ

into the locally linearized space of the manifold. The matrix resulting matrix Σ' is given by:

$$\Sigma' = \mathbf{J}^{\mathrm{T}} \Sigma \mathbf{J} \quad \text{with} \quad \mathbf{J} = \left. \frac{\partial \, \mathbf{x} \boxplus \delta}{\partial \, \delta} \right|_{\delta = \mathbf{0}} \tag{3.6}$$

Note that we are only going to need this covariance estimation (and in fact the whole ICP) in the 3D case.

3.1.2 Normals of Point Clouds

One problem of the ICP algorithm as presented in the previous section is, that due to the way it searches for nearest neighbors, it is likely to make point correspondences, which can not possibly be correct due to the way the point clouds were captured. An example for this is a wall which was scanned from two sides. Now the points from one side of the wall are potentially associated with points from the other side of the wall. Since all our point clouds are the result of a single 3D laser scan, we can determine the normal of each point in the point cloud. These normals make it possible to check whether the surface in the two query points faces in roughly the same direction. If they are facing in opposite directions, the correspondence can be rejected.

Normals are quite simple to compute given surface information of an object. When dealing with point clouds however, surface information is generally unavailable.

In order to estimate the normal at a certain point, it is required to first estimate the surface at that point. One possibility to do that is to first approximate the surface of the whole point cloud and then computing the normals from this approximation.

Since this is quite complex, we follow a much simpler alternative method, that only considers the local neighborhood of every point by computing its nearest neighbors. Given the set of N nearest neighbors $\mathbf{x}_i$ with $i = 1..N$, we compute the mean $\bar{\mathbf{x}}$ and the covariance matrix $\mathbf{\Sigma}$ using:

$$\bar{\mathbf{x}} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{x}_i \tag{3.7}$$

$$\mathbf{\Sigma} = \sum_{i=1}^{N} (\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^{\mathrm{T}} \tag{3.8}$$

Assuming local planarity of the surface, the eigenvector corresponding to the smallest eigenvalue of the eigen decomposition of the covariance matrix $\mathbf{\Sigma}$, is a vector that is normal to the surface.

A small issue is that the sign of the normal is undetermined. Fortunately we can use domain specific knowledge to determine it: The point clouds considered

in this work are all a result of laser scans. Consequently all surface normals have to point towards the position of the laser scanner (which is usually the origin).

The pseudo code for the described algorithm is given in Algorithm 2. Some implementation notes:

- The function getNeighbors mentioned in Algorithm 2 could either be implemented as a radius search or as a k-NN search. It is to be expected that a radius search makes more sense in this case and yields higher quality normals, because all normals are computed from a local surface patch of the same size. Yet we typically use a k-NN search with $k = 10$, because it is computationally cheaper, and because there are no problems in areas with low point densities. The downside of this method is that it suffers artifacts in areas with extremely high point densities, as it will attempt to compute normals from tiny clouds of points with no distinct shape in that case. These artifacts are also visible at the top of the left picture of Figure 3.1. We chose not to deal with these artifacts because only a very small part of the point cloud is affected, which does not interfere with our algorithm.
- The main loop inside of Algorithm 2 is in fact embarrassingly parallel, meaning that it is extremely easy to parallelize. This optimization has been applied in the implementation.

Algorithm 2 Computing normals from point clouds

Input: **origin** – Position of the laserscanner
Input: *pointSet* – Point Cloud
Output: *normals* – Normals for every point

foreach (**p** in *pointSet*)
{
  $\{\mathbf{x}_i\}$ = getNeighbors(p, *pointSet*)
  $\bar{\mathbf{x}} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}_i$
  $\mathbf{\Sigma} = \sum_{i=1}^{N}(\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^{\mathrm{T}}$
  $\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{\mathrm{T}} = \mathrm{eig}(\mathbf{\Sigma})$ // compute eigendecomposition
  $\mathbf{n} = \mathbf{Q}.\mathrm{col}(\arg\min_v \mathbf{\Lambda}_{v,v})$
  if $(\mathbf{n}^{\mathrm{T}}(\mathbf{p} - \mathbf{origin}) > 0)$
    $\mathbf{n} = -\mathbf{n}$
  $normals[\mathbf{p}] = \mathbf{n}$
}
return *normals*

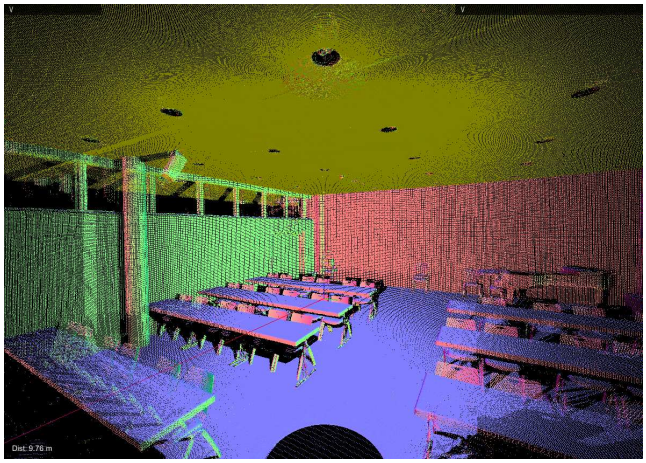


Figure 3.1: Visualization of the results of Algorithm 2 applied to a typical point cloud. The colors are subject to the mapping $XYZ \rightarrow RGB : \mathbf{n} \rightarrow \frac{1}{2}(\mathbf{n} + \mathbf{1})$. More examples can be found throughout this work, as we frequently use this color mapping since it yields well-perceivable images of point clouds.

	# of neighbors				
	$k = 5$	$k = 10$	$k = 20$	$k = 40$	$k = 80$
1 Thread	5234 ms	6583 ms	9307 ms	15042 ms	29544 ms
6 Threads	1598 ms	1830 ms	2278 ms	3333 ms	6113 ms

Table 3.1: Processing time in ms of a typical scan with 970 k points.

Results

An example for the computed normals can be found in Figure 3.1. To get an idea of the time required to compute the normals, the total computation time for a typical scan with around 970 k points for different amounts neighbors are shown in Table 3.1. The measurements were performed on a machine with a Intel Core-i7 920 Quad Core processor with 2.66 Ghz and enabled HyperThreading. In order to solve the k-NN problem, we use the FLANN library[1].

3.1.3 Downsampling of Point Clouds

During the ICP algorithm, every considered point pair usually has an equal influence on the result of the computation. The consequence is that the algorithm is biased towards areas having a high density of points. Such areas occur in ev-

[1]The FLANN library is available at `http://people.cs.ubc.ca/~mariusm/index.php/FLANN/FLANN`

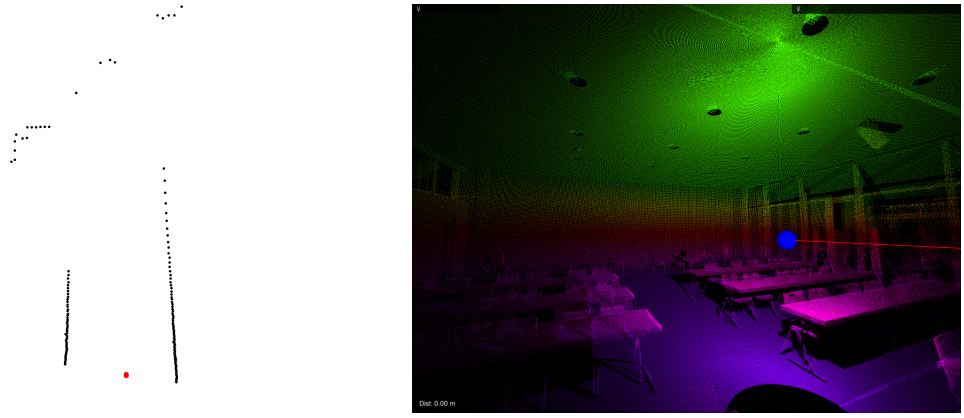


Figure 3.2: Uneven point densities in 2d (left) and 3d (right). The colors in the right hand side image indicate the height. The area right above the location from where the scan has been recorded shows a much higher point density.

ery laser scan due to the working principle of laser range finders. Uneven point densities can also occur when 3D laser range finders are built by rotating a 2D laser range finder: When the scanner is simply rotated around a certain axis for example, the area right above/below has a significantly higher density than the rest (see Figure 3.2).

What is needed to make the ICP algorithm unbiased to point densities is a subset of the point cloud in which all points are uniformly spaced in terms of geodesic distance. The geodesic distance is the shortest distance between two points on the surface on an object when only movement on this surface is allowed.

Poisson Disk Sampling

The computation of the aforementioned geodesic distance requires surface information, which is generally not readily available for laser scanned point clouds. Thus we approximate this distance using an Euclidean distance metric. This approximation is reasonable, though not perfect and especially complex areas will suffer undersampling (see Figure 3.3 for an illustration of the difference between geodesic and Euclidean distance). However, especially in the early iterations where the transformation estimate is not very good, complex areas are rarely very helpful for ICP anyway because of the high probability of false data-associations.

Therefore we want a point cloud that is evenly sampled in terms of Euclidean distance. A good way to guarantee an even sampling is to define a certain radius r and to make sure that there is no point in a sphere with that radius around any other point. This sampling policy is called Poisson disk sampling. It is related to techniques originating from computer graphics [Coo86] and is also known under

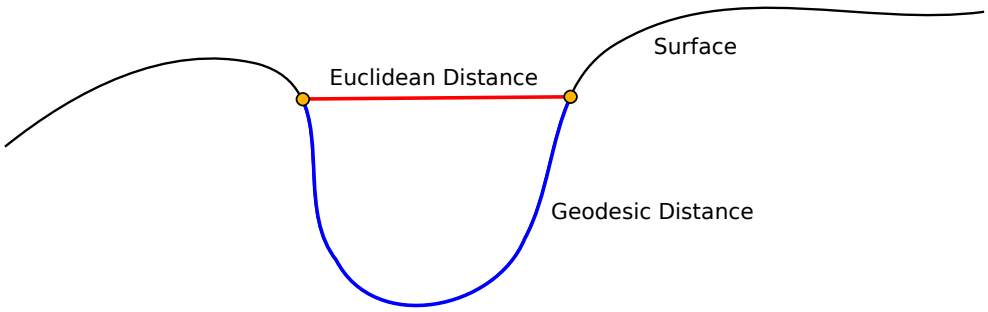


Figure 3.3: Illustration of the difference between Euclidean and geodesic distances.

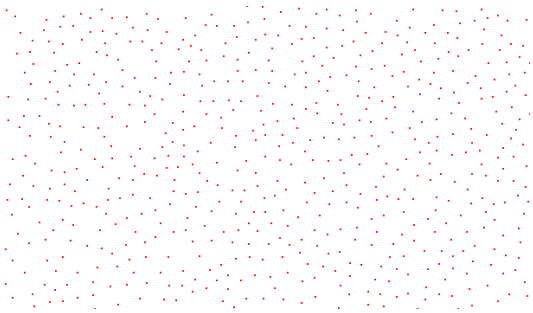


Figure 3.4: A typical pattern encountered after performing Poisson downsampling. The structure of the points is in a way regular, yet the pattern appears quite random. The regularity originates from the fact that there is always a certain minimum distance enforced between any two points. The randomness is due to the random selection of points from the input point set.

the name "Dart Throwing" [CJR+09]. It is said to have good statistical properties in the cited papers.

A straightforward implementation of this definition would be to keep a kD-tree for all points already in the downsampled point set. One could then loop over all points and perform a sphere query in the kD-tree to find out whether any point is in this sphere, and thus violates the above mentioned constraint. If this is the case, the current point is discarded, otherwise it is accepted, and inserted both into the downsampled point set and into the kD-tree.

While this method basically works, linearly iterating through the point cloud has a number of disadvantages. First note that point clouds are typically sorted in some way, i. e. they are spatially coherent. This could lead to potential aliasing effects in the downsampled result set. Another issue is that after accepting a point, subsequent points are very likely to be located in the sphere around the just accepted point. This can lead to very poor performance.

To counteract these issues, one could first shuffle the points, or iterate over them in a randomized order. Since the implementation of these two alternatives is a bit cumbersome and inefficient, we chose to simply randomly pick points from the point cloud, aborting the selection process either when we have found the desired number of points, or once a certain degree of "oversampling" has occured. Oversampling happens when a point is picked which is inside the sphere of a previously picked point. We simply proceed picking another random point in that case. Limiting the amount of oversampling makes sure that the selection process is terminated once a sufficiently good result has been reached. Without this, the algorithm could potentially loop forever, never happening to find an acceptable point.

The pseudo code for this algorithm can be found in Algorithm 3. Figure 3.4 shows typical "patterns" encountered when downsampling point clouds using this algorithm (or with Poisson downsampling in general).

Algorithm 3 Poisson Disk Sampling

Input: pc – Point Cloud
Input: r – Sphere Radius
Input: d – # of points to sample down to
Input: $oversampling$ – Oversampling Coefficient
Output: dpc – Downsampled Point Cloud

kd = **new** KDTree

for ($i = 1 \rightarrow oversampling \cdot d$)
{
    p = selectRandomPoint(pc)
    if (kd.radiusSearch(**p**,r)==$\emptyset$)
    {
        kd.insert(**p**)
        dpc.append(**p**)
        if (dpc.size()== d)
            break // desired # of pts found
    }
}

return dpc

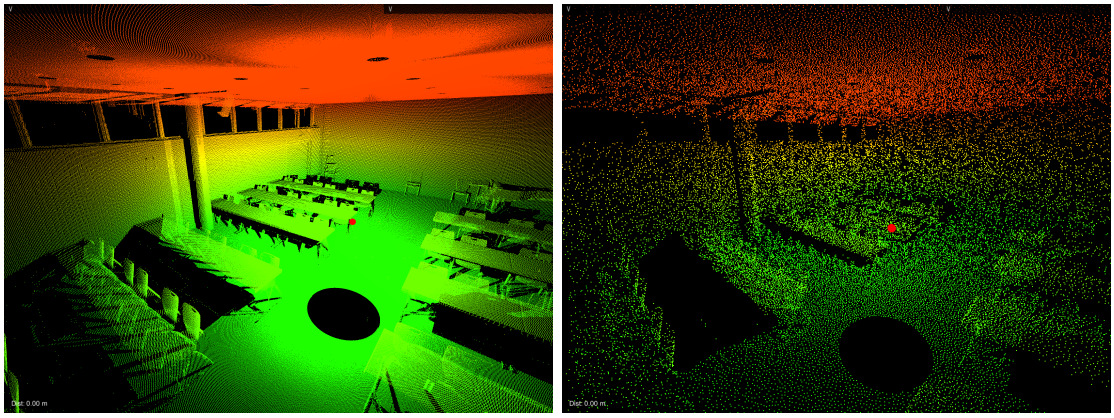


Figure 3.5: The image on the left shows the input point cloud with about 1.7 million points. The image on the right shows the result after applying the Poisson downsampling algorithm. The point cloud was downsampled to 80 k points and the radius defining the minimum allowed distance between two points was set to 3 cm.

Results

Figure 3.5 shows a point cloud before and after downsampling using our algorithm. Note that it is obviously quite hard to perceive, since uneven point densities, which normally make it possible to see what kind of scenario a point cloud shows, are exactly what the poisson sampling strives to remove. When taking a close-up look at any downsampled surface, they typically look like in Figure 3.4. To get an idea of the time required for this operation: Downsampling the point cloud in Figure 3.5, which originally had about 1.7 million points, to 80 k points with a minimum allowed distance between two points of 3 cm and an oversampling coefficient of 15, the algorithm requires about 290 ms on our machine.

3.2 Correlative Scan Matching

The previous section discussed the ICP algorithm and we mentioned that the algorithm only works properly if the initial guess for the transformation between the two scans is good enough. Unfortunately, this is the main problem of the algorithm. As we are going to see in the next chapter, there are some cases where a good prior is unavailable. Therefore we need a method to perform the registration of two scans *without* a good prior.

This section first presents the summary of an algorithm, which is called "Real-Time Correlative Scan Matching", originally developed by Olson in [Ols09a] to align 2D point clouds, and then proceeds to present our extension of this algorithm to 3D.

Olson's algorithm strives to find the distribution $\mathrm{p}(\mathbf{x}_i \mid \mathbf{x}_{i-1}, \mathbf{u}, \mathbf{m}, \mathbf{z})$, which is essentially the typical SLAM posterior. The underlying Bayes net states that the robot pose at time i, $\mathbf{x}_i$, is affected by the control input $\mathbf{u}$. In addition the robot makes a measurement $\mathbf{z}$ at time i, which is affected by the current robot pose $\mathbf{x}_i$ as well as the map $\mathbf{m}$.

The application of Bayes' rule as well the use of conditional independence assumptions from the described Bayes net, allows transforming the distribution as follows:

$$\mathrm{p}(\mathbf{x}_i \mid \mathbf{x}_{i-1}, \mathbf{u}, \mathbf{m}, \mathbf{z}) \propto \mathrm{p}(\mathbf{z} \mid \mathbf{x}_i, \mathbf{m})\, \mathrm{p}(\mathbf{x}_i \mid \mathbf{x}_{i-1}, \mathbf{u}) \tag{3.9}$$

The second factor is the motion model of the robot. It is typically known in terms of a Gaussian. In case of not knowing anything about the motion of the robot, it can be assumed to be uniformly distributed. The first factor, the observation model, is usually a very complex function with many local maxima, which is the reason why locally operating algorithms such as ICP or other methods struggle finding the maximum-likelihood solution of the posterior. Finding the distribution of the observation model is the "hard part" of finding the full posterior, and thus the following algorithm is focused on it. The main objective is to compute the $\mathbf{x}_i$, which maximizes the observation model. A desired byproduct is the computation of an estimate for the uncertainty of the solution. In order to be able to search for the maximum-likelihood solution, we first need a way to evaluate the observation model for a specific $\mathbf{x}_i$.

A common assumption is to assume that all rays in the laserscan are independent measurements. This assumption allows simplifying the observation model:

$$\mathrm{p}(\mathbf{z} \mid \mathbf{x}_i, \mathbf{m}) = \prod_k \mathrm{p}(\mathbf{z}_k \mid \mathbf{x}_i, \mathbf{m}) \tag{3.10}$$

Here $\mathbf{z}_k$ corresponds to a single measurement of the laserscan and $\mathrm{p}(\mathbf{z}_k \mid \mathbf{x}_i, \mathbf{m})$ is the likelihood of observing that specific measurement. In practice, the evaluation of the individual $\mathrm{p}(\mathbf{z}_k \mid \mathbf{x}_i, \mathbf{m})$ is done via a lookup table: It contains the log probabilities of making a laser measurement at each position in the world. Thus, computing $\log \mathrm{p}(\mathbf{z} \mid \mathbf{x}_i, \mathbf{m})$ means transforming the scan $\mathbf{z}$ by the pose $\mathbf{x}_i$, and summing up the values in the lookup table at the positions corresponding to the laser endpoints.

The lookup table is basically a 2D array where each cell is assigned the log likelihood of a Gaussian of the nearest point in the map. If no point is near, a low constant value is assigned to the cell. A visualization of a lookup table, which was constructed in this way can be found in Figure 3.6.

A naive/brute-force way to find the distribution $\mathrm{p}(\mathbf{z} \mid \mathbf{x}_i, \mathbf{m})$ would be to loop over all possible $\mathbf{x}_i$, computing the value of the likelihood function everywhere. This allows finding the covariance, and the maximum-likelihood solution. Since $\mathbf{x}_i$ refers to a two dimensional pose having three degrees of freedom, three nested

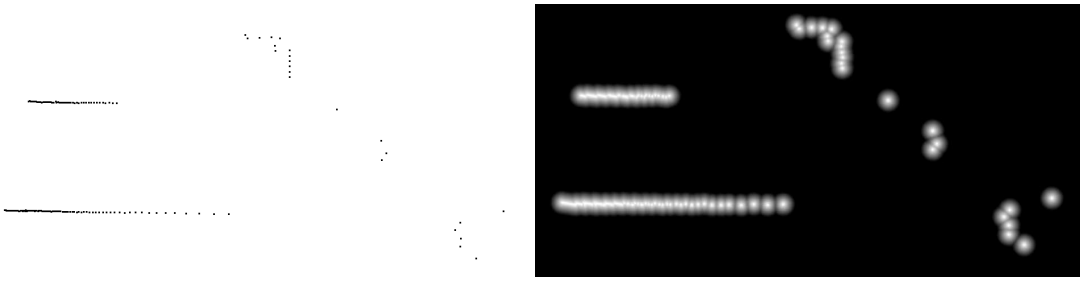


Figure 3.6: The image on the left shows a typical 2D laser scan, serving as the map $\mathbf{m}$. The image on the right shows a lookup table corresponding to this laser scan, which can be used to compute the measurement likelihood of a certain laser scan $\mathbf{z}$ being $\mathrm{p}(\mathbf{z} \mid \mathbf{x}_i, \mathbf{m})$. Every position in the image corresponds to the log likelihood of observing a laser point at that position.

loops are needed for this (in fact four since one has to loop over the scan points as well). While this may be acceptable for small search spaces, it is generally too inefficient for large ones consisting of large rotational search ranges in combination with translational search windows in the order of several meters.

Olson suggests using a multi-level resolution approach to speed up the computation. He suggests using two resolution levels, but the algorithm basically works for arbitrary amounts of resolution levels. The idea is to downsample the lookup table in a way that each cell in the low resolution table equals the maximum of the corresponding cells in the high resolution table. The suggested downsampling factor is 10, so 10×10 fine cell are mapped to one coarse cell. This methodology has an interesting property: When computing $\mathrm{p}(\mathbf{z} \mid \mathbf{x}_i, \mathbf{m})$ at a specific $\mathbf{x}_i$ using the coarse lookup table, the result is conservative in the sense that the probability is always overestimated in comparison to a lookup at the finest resolution level. This means that the coarse lookup table can be used to identify areas with a *potentially* high probability mass, which can then, in a second step, be further analyzed using the fine lookup table. If during this step, a probability is discovered that is higher than the maximum reachable probability of a coarse block, then this coarse block can be skipped. Therefore areas with a low probability mass are skipped at the highest resolution.

Since a big part of the total probability mass is sampled this way, one can compute a good uncertainty estimate Σ using:

$$\mathbf{K} = \sum_j \mathbf{x}_i^{(j)} \mathbf{x}_i^{(j)T} \, \mathrm{p}(\mathbf{x}_i^{(j)} \,|\, \mathbf{x}_{i-1}, \mathbf{u}, \mathbf{m}, \mathbf{z}) \tag{3.11}$$

$$\mathbf{u} = \sum_j \mathbf{x}_i^{(j)T} \, \mathrm{p}(\mathbf{x}_i^{(j)} \,|\, \mathbf{x}_{i-1}, \mathbf{u}, \mathbf{m}, \mathbf{z}) \tag{3.12}$$

$$s = \sum_j \mathrm{p}(\mathbf{x}_i^{(j)} \,|\, \mathbf{x}_{i-1}, \mathbf{u}, \mathbf{m}, \mathbf{z}) \tag{3.13}$$

$$\Sigma = \frac{1}{s} K - \frac{1}{s^2} \mathbf{u}\mathbf{u}^T \tag{3.14}$$

The summations cover all the samples done at the highest resolution. The key advantage of this estimate in comparison to an ICP based covariance estimate is that it accounts for a large part of the probability mass of the density function – even parts that are not close to the maximum-likelihood solution. Thus it not only takes the sensor noise into account, but also the possibility of wrong data-association. In contrast, the ICP based covariance only accounts for uncertainty introduced by sensor noise, which is reflected in the probability distribution in immediate vicinity of the computed transformation. This generally makes ICP covariance estimates overconfident.

The pseudo code for the algorithm can be found in Algorithm 4.

Extension to 3D

The algorithm we just presented is designed to work on 2D laser scans only. Naively extending it to 3D seems infeasible, because the volume of the six dimensional search space would typically be huge. We try to deal with the problem by essentially reducing the 3D alignment problem to 2D.

We first note that since almost all robots have integrated IMUs (inertial measurement units), or simply 3-axis-accelerometers, a good guess for the roll and pitch angle of the scans is typically known. Therefore only x, y, z as well as the yaw angle remain to be estimated.

Virtually all 3D laser scans contain "vertical structures" such as walls, pillars, bushes or the sides of cars. The next idea is to first filter vertical structures (which do even not have to be exactly straight), and then project them on the xy plane. The result is a set of 2D points, which resembles a 2D scan. A point is present in this scan at every position that contains a vertical structure somewhere at that x,y position. The interesting property of this approach is, that the result is almost z independent, i. e. scans taken at different heights (z coordinates) exhibit almost the same vertical structure points in the xy plane. This essentially decouples the

Algorithm 4 Correlative Scan Matching

Input: $\mathbf{m}$ – Reference scan or map
Input: $\mathbf{z}$ – Scan to be aligned with the map
Output: $\mathbf{x}_i$ – Maximum-likelihood solution
Output: Σ – Covariance estimate of the solution

- Build fine lookup table
- Downsample fine lookup table to get a coarse lookup table
- Evaluate estimates for $\mathrm{p}(\mathbf{z} \mid \mathbf{x}_i, \mathbf{m})$ using the coarse lookup table for all $\mathbf{x}_i$ in the search window
- Sort cells by the likelihood

$Best = 0$ // stores highest likelihood found at the fine resolution so far
forever
{
  - Pick the coarse cell with the highest likelihood L
  - If $L<Best$
    break
  - Evaluate $\mathrm{p}(\mathbf{z} \mid \mathbf{x}_i, \mathbf{m})$ using the fine lookup table for all $\mathbf{x}_i$ in the current coarse cell, keep $Best$ and corresponding $\mathbf{x}_{\mathrm{Best}}$ updated

}
- Compute Σ from all lookups in the fine table

return $\mathbf{x}_{\mathrm{Best}}, \Sigma$

estimation of x, y and yaw from the z coordinate. Therefore the general idea of our algorithm is:

1. Compute projection of vertical structures
2. Compute x, y and yaw using correlative scan matching
3. Estimate z
4. Given the initial guess, perform the actual registration using ICP

Step one is done with an extremely simple algorithm. We simply create a zero-initialized 2D grid, project the points down on the xy plane and increment the corresponding grid cell value. After all points are processed that way, a second pass filters out the 2D positions of cells that contain more than a certain number of points, and whose z range is larger than a certain threshold. The grid we use has a cell size of 12 cm.

Step three assumes the scans are approximately aligned in x, y, yaw, pitch and roll already. In order to estimate z, we compute the number of "inliers" for different values of z, and choose the z which has the maximum amount of inliers. We consider an inlier as a point from one scan that has a neighbor in the other scan within a certain range. We typically sample a z range of ± 3 m in steps of 10 cm. Our threshold for accepting a point as an inlier is 30 cm. The query points are taken from a Poisson downsampled version of the query scan.

Having estimated x, y, z and yaw this way, we perform a final ICP step with this initial guess. This is required for several reasons:

- The discretization and inaccuracies introduced in step 1
- The discrete sampling of z in step 3
- The values for pitch and roll are often known approximately, but not exactly.
- We need a covariance matrix of the solution which covers all six degrees of freedom, which could not be obtained otherwise. As a consequence, we are unfortunately stuck with the overconfident ICP covariance estimate once again in the 3D case.

Chapter 4

GraphSLAM System

This chapter presents the actual graph-based SLAM system, which serves to generate globally consistent maps from 2D or 3D laserscans. The main idea is to represent scans, or rather the poses from which they were taken, as nodes in a graph. Constraints between these nodes are generated in form of Gaussian (pseudo-)measurements using the scan matching techniques derived in the previous chapter. Since measurements generally do not fully agree with each other, the result is a contradictory graph.

As indicated previously, a SLAM system is basically a composition of two subsystems: The frontend and the backend. We will first focus on the probabilistic formulation of the problem, which represents the mathematical basis for the backend. The backend allows finding a configuration of graph's nodes, which is maximally consistent. It does not deal with the construction of the maintenance of this graph though. This part is done by the frontend, which will be presented afterwards. The frontend essentially creates a new node for every laserscan passed on to the system and derives the constaints between them.

4.1 Probabilistic Problem Formulation

The SLAM problem comprises the estimation of both the trajectory of the robot $\mathbf{x}$ as well as all feature positions θ given the measurements $\mathbf{z}$. Mathematically the (full) solution to the problem can be described as the parameters which maximize the following likelihood function:

$$\mathrm{p}(\mathbf{x}, \theta \,|\, \mathbf{z}) \tag{4.1}$$

The downside of this formulation of the problem is the huge number of unknowns [OLT06]: Both the true robot poses and the feature positions are unknown and need to be estimated at the same time. The fact that there is usually a large

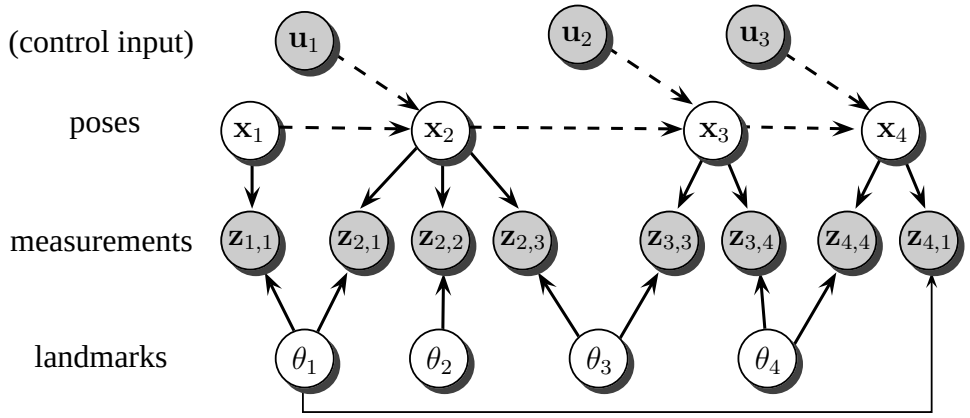


Figure 4.1: The image shows an example situation of the SLAM Bayes net. Circles correspond to random variables and edges indicate conditional dependence. Gray circles are known quantities, white circles are unknown. The $\mathbf{x}_i$ represent random variables for different poses of the robot. At every pose, the robot observes a number of landmarks. Each observation is a random variable on its own. The random variable $\mathbf{u}_i$ represent control input given to the robot. In this work we do not assume to have any control input, which is why the dependence is drawn with dashed lines. The lack of control input however enforces that subsequent poses have common landmark measurements. A particularly interesting measurement is $\mathbf{z}_{4,1}$, as it is a re-observation of the landmark θ_1 and closes a loop with the first and the second pose.

amount of landmarks makes the problem very high-dimensional and thus very hard to solve. To facilitate the computation, this equation can be decomposed into two factors using the chain rule: [MTKW02]

$$\mathrm{p}(\mathbf{x}, \theta \,|\, \mathbf{z}) = \mathrm{p}(\mathbf{x} \,|\, \mathbf{z})\ \mathrm{p}(\theta \,|\, \mathbf{x}, \mathbf{z}) \tag{4.2}$$

Instead of jointly estimating both $\mathbf{x}$ and θ, this factorization allows for a sequential estimation of $\mathbf{x}$ and θ, thus greatly reducing the search space.

Using the Bayes ball algorithm on the underlying dynamic Bayesian network (an example situation is depicted in Figure 4.1), distinct θ_i can be found to be conditionally independent given $\mathbf{x}$ and $\mathbf{z}$ [MTKW02]. Therefore the second term can be further decomposed:

$$\mathrm{p}(\mathbf{x}, \theta \,|\, \mathbf{z}) = \mathrm{p}(\mathbf{x} \,|\, \mathbf{z}) \prod_i \mathrm{p}(\theta_i \,|\, \mathbf{x}, \mathbf{z}) \tag{4.3}$$

The first term, $\mathrm{p}(\mathbf{x} \,|\, \mathbf{z})$ estimates the robot's trajectory, given the measurements.[1] The second term estimates the location of the features, given both the robot's

[1] It is equivalent to Equation (4.1) in which the features have been marginalized out: $\mathrm{p}(\mathbf{x} \,|\, \mathbf{z}) = \int \mathrm{p}(\mathbf{x}, \theta \,|\, \mathbf{z})\ \mathrm{d}\theta$

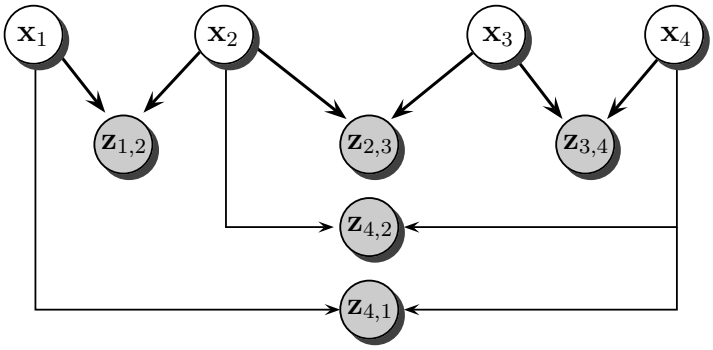


Figure 4.2: The same Bayes net as in Figure 4.1 except that the landmarks have been marginalized out and the control inputs have been omitted. Note that the $\mathbf{z}_{i,j}$ above are new random variables and do not correspond to the "old ones" in Figure 4.1.

trajectory and the measurements. Due to the mutual conditional independence of the features, it is relatively easy to compute since each feature can be considered separately.

The first term can also be considered as a marginalized version of the first term from Equation (4.1) where the features θ were removed. As a result of that, the Bayes net is altered in a way that *new* (pseudo-)measurements are added between two poses which previously observed common features.[2] As an example, marginalizing out the landmarks from the Bayes net in Figure 4.1 results in the Bayes net given in Figure 4.2.

In the context of this work, we are primarily interested in this first term $\mathrm{p}(\mathbf{x} \mid \mathbf{z})$, i. e. the localization part of the SLAM problem, and we seek to find the maximum of that likelihood function:

$$\arg\max_{\mathbf{x}} \mathrm{p}(\mathbf{x} \mid \mathbf{z}) \tag{4.4}$$

Once the optimal trajectory has been found, it could be used to find the corresponding optimal feature positions, by computing the following term for each θ_i:

$$\arg\max_{\theta_i} \mathrm{p}(\theta_i \mid \mathbf{x}, \mathbf{z}) \tag{4.5}$$

This last step is out-of-scope of this work though.

Pose Graphs

Before proceeding to the computation of Equation (4.4), we are going to present an alternative view on the dynamic Bayesian network introduced in the previous

[2]This is a result from the theory of graphical models. The prefix *pseudo* indicates that these measurements are not actual measurements made by some device, but they are the result of aggregating the actual measurements.

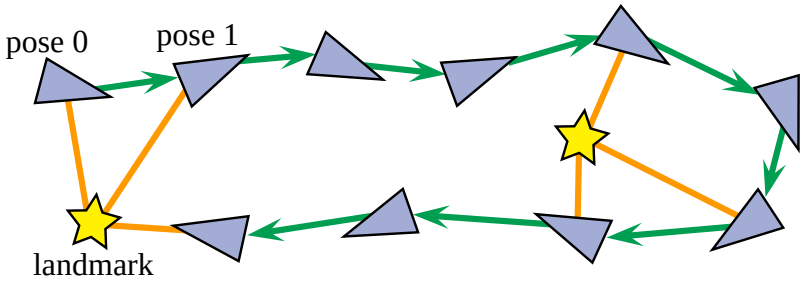


Figure 4.3: Example of a pose/feature graph [Ols08]

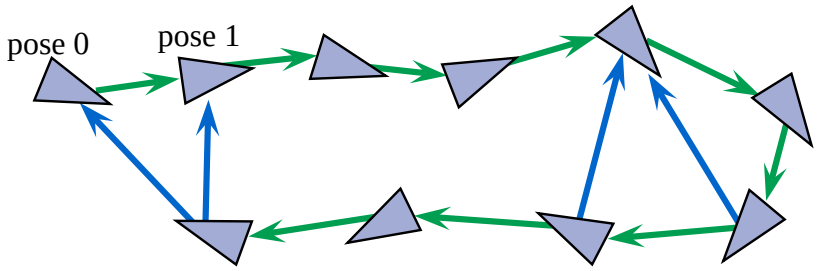


Figure 4.4: Pose graph after marginalizing out the features of the pose/feature graph from Figure 4.3. The new edges are highlighted in blue. Their direction is arbitrarily chosen and does not matter.

section, which has a slightly more intuitive visualization, which is also closer to the actual implementation.

Nodes are again random variables and represent the pose of the robot, or the position of other landmarks and features in the world. Edges between nodes are also random variables and correspond to estimated or measured relative motion (for example from odometry). Edges from nodes to landmarks are measurements of the relative position of the landmark to the respective pose of the robot. In the context of this work, we will assume all of these measurements to be Gaussian.

Figure 4.3 shows an example a graph constructed this way. Figure 4.4 shows a graph that corresponds to the marginalization of the previous graph.

4.2 Backend

We will now present the backend, which is *only* concerned with the computation of Equation (4.4). Therefore it is fairly detached from the real world: It neither directly works with sensor data nor does it do any data-association or outlier rejection. It only tries to find the maximum-likelihood solution to the problem.

The construction and maintenance of the graph required by the backend is going to be discussed in the next section.

4.2.1 Maximum-Likelihood Estimation

This section presents a way to compute the optimal, i. e. maximum-likelihood trajectory $\mathbf{x}$ given the set of measurements $\mathbf{z}$ and an initial guess $\mathbf{x}_0$ for the trajectory. The vector $\mathbf{x}$ is a concatenation of the individual poses of the robot's trajectory:

$$\mathbf{x} = (\mathbf{x}_1^{\mathrm{T}}, \mathbf{x}_2^{\mathrm{T}}, \dots)^{\mathrm{T}} \tag{4.6}$$

The individual poses $\mathbf{x}_i$ use the representations derived in Section 2.2. As shown in Section 2.1, both the positions and the orientations contained in them are manifolds, which together form a manifold again.

The term $\mathbf{z} = \{\mathbf{z}_{ij}\}$ denotes the set of all Gaussian measurements attached to the edges of the underlying graph. An edge from node i to node j has the covariance matrix $\mathbf{\Theta}_{ij}^{-1}$.

As shown in Equation (4.4) we need to find an $\mathbf{x}$ which maximizes:

$$\mathrm{p}(\mathbf{x} \,|\, \mathbf{z}) = \mathrm{p}(\mathbf{x} \,|\, \mathbf{z}_{i_1 j_1}, \mathbf{z}_{i_2 j_2}, \dots, \mathbf{z}_{i_n j_n}) \tag{4.7}$$

As we found out in Section 2.3, this requires iterated solving of:

$$\left(\sum_{ij} \mathbf{J}_{ij}^{\mathrm{T}} \mathbf{\Theta}_{ij} \mathbf{J}_{ij} \right) \mathbf{h} = \sum_{ij} \mathbf{J}_{ij}^{\mathrm{T}} \mathbf{\Theta}_{ij} \mathrm{e}_{\mathbf{z}_{ij}}(\mathbf{x}_0) \tag{4.8}$$

$$\text{with} \quad \mathbf{J}_{ij} = \left. \frac{\partial \, \mathrm{e}_{\mathbf{z}_{ij}}(\mathbf{x} \boxplus \mathbf{h})}{\partial \, \mathbf{h}} \right|_{\mathbf{h}=\mathbf{0}}$$

The function $\mathrm{e}_{\mathbf{z}_{ij}}$ computes the error of the constraint between node i and j. We will choose it as:

$$\mathrm{e}_{\mathbf{z}_{ij}}(\mathbf{x}) = (\mathbf{x}_j \ominus \mathbf{x}_i) \boxminus \mathbf{z}_{ij} \tag{4.9}$$

A visual explanation of this function can be found in Figure 4.5. The first part, $\mathbf{x}_j \ominus \mathbf{x}_i$ computes the relative pose between the *currently* best guess for their absolute poses. $\mathbf{z}_{ij}$ is a measurement of the same quantity.

Taking the difference between them with the $\boxminus$ operator computes a difference vector between the two in the locally linear $\mathbb{R}^n$ space of the manifold. In a world with perfect measurements, both would be equal and the error would simply be $\mathbf{0}$. It is sane to use the $\boxminus$ operator since the two quantities are supposedly close together given a reasonable guess for $\mathbf{x}$.

The Jacobian $\mathbf{J}_{ij}$ is:

$$\mathbf{J}_{ij} = \begin{pmatrix} \mathbf{0} & \dots & \overset{j}{\mathbf{A}} & \dots & \overset{i}{\mathbf{B}} & \dots & \mathbf{0} \end{pmatrix} \tag{4.10}$$

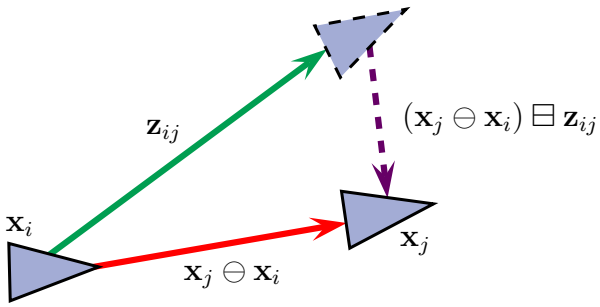


Figure 4.5: $\mathbf{x}_i$ and $\mathbf{x}_j$ are the poses of node i and node j in the current state $\mathbf{x}$. The term $\mathbf{x}_j \ominus \mathbf{x}_i$ describes the relative pose from node i to node j. $\mathbf{z}_{ij}$ is a measurement of this relative pose. The $\boxminus$ operator calculates the difference between the two in the locally linear $\mathbb{R}^n$ space of the pose manifold. If the current configuration of the nodes and the measurement are equal, this difference would become zero.

Here $\mathbf{A}$ and $\mathbf{B}$ represent the partial derivatives of $\mathrm{e}_{\mathbf{z}_{ij}}$ by $\mathbf{x}_i$ and $\mathbf{x}_j$ respectively. The actual formulas depend on whether we work in 2D or 3D and are not given in this work. Our implementation can compute them either numerically or via AutoDiff[3]. A symbolic derivation is possible though.

The important part is that only two blocks of the Jacobian are actually occupied, while the majority of entries is zero. This is obvious because deriving by anything other than $\mathbf{x}_i$ and $\mathbf{x}_j$ yields zero since they do not even occur in $\mathrm{e}_{\mathbf{z}_{ij}}$. Thus the term $\mathbf{J}_{ij}^{\mathrm{T}} \mathbf{\Theta}_{ij} \mathbf{J}_{ij}$ from Equation (4.8) becomes:

$$\mathbf{J}_{ij}^{\mathrm{T}} \mathbf{\Theta}_{ij} \mathbf{J}_{ij} = \begin{array}{c} \\ j \\ \\ \\ i \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccccc} & & & j & & i & \end{array} \\ \begin{pmatrix} \mathbf{0} & \dots & \mathbf{0} & \dots & \mathbf{0} & \dots & \mathbf{0} \\ \vdots & \dots & \vdots & \dots & \vdots & \dots & \vdots \\ \mathbf{0} & \dots & \mathbf{A}^{\mathrm{T}} \mathbf{\Theta}_{ij} \mathbf{A} & \dots & \mathbf{A}^{\mathrm{T}} \mathbf{\Theta}_{ij} \mathbf{B} & \dots & \mathbf{0} \\ \vdots & \dots & \vdots & \dots & \vdots & \dots & \vdots \\ \mathbf{0} & \dots & \mathbf{B}^{\mathrm{T}} \mathbf{\Theta}_{ij} \mathbf{A} & \dots & \mathbf{B}^{\mathrm{T}} \mathbf{\Theta}_{ij} \mathbf{B} & \dots & \mathbf{0} \\ \vdots & \dots & \vdots & \dots & \vdots & \dots & \vdots \\ \mathbf{0} & \dots & \mathbf{0} & \dots & \mathbf{0} & \dots & \mathbf{0} \end{pmatrix} \end{array} \tag{4.11}$$

[3]We use the NumericalDiff and AutoDiff modules from the Eigen math library: http://eigen.tuxfamily.org

The term $\mathbf{J}_{\mathbf{z}_{ij}}^{\mathrm{T}} \mathbf{\Theta}_{ij} \mathrm{e}_{\mathbf{z}_{ij}}(\mathbf{x}_0)$ becomes:

$$\mathbf{J}_{ij}(\mathbf{x})^{\mathrm{T}} \mathbf{\Theta}_{ij} \mathrm{e}_{ij}(\mathbf{x}_0) = \begin{matrix} \\ j \\ \\ i \\ \\ \end{matrix} \begin{pmatrix} \mathbf{0} \\ \vdots \\ \mathbf{A}^{\mathrm{T}} \mathbf{\Theta}_{ij} \mathrm{e}_{\mathbf{z}_{ij}}(\mathbf{x}_0) \\ \vdots \\ \mathbf{B}^{\mathrm{T}} \mathbf{\Theta}_{ij} \mathrm{e}_{\mathbf{z}_{ij}}(\mathbf{x}_0) \\ \vdots \\ \mathbf{0} \end{pmatrix} \tag{4.12}$$

Looking at the non-zero structure of this matrix, we can simply start with a zero matrix and a zero vector, loop over all edges and add up the non-zero blocks, in order to efficiently compute the sum in Equation (4.8). Some further notes on the occupancy of the matrix:

- Two blocks i,j are non-zero if node i and node j are connected, which is the same as in the adjacency matrix of the underlying graph. In addition to that, all diagonal blocks are non-zero, given there are no nodes in the graph without any edges. We have to disallow this case though, because it would make the equation system underconstrained.

- The matrix is typically very sparse: Consider a loop around a building that consists of 30 scans and 31 edges (one edge between each subsequent scan, and one from the last to the first scan). The matrix would consist of $30 \times 30 = 900$ blocks, of which only 90 would be occupied. The occupied fraction of blocks gets even lower as the size of the loop is increased. Higher occupancy only occurs when there are a lot of loop-closing edges. This happens for example when a robot keeps revisiting the same places: Every node added to the graph could potentially result in dozens of loop-closing edges. For practical use, the frontend needs to make sure that the number of loop-closing edges does not get out of control.

The sparsity of the matrix makes it necessary to use sparse matrix methods to solve the equation system. More on this can be found in Section 4.4.

4.3 Frontend

The iteration derived the previous section allows us to find the optimal configuration of the nodes in the graph. The task of the frontend, which we will discuss

now, is to construct and update this graph so that the backend can perform its optimization on it.

Note that being able to minimize the error of a given, inconsistent graph is a very important part of solving the GraphSLAM problem. At first glance, it might even seem like the backend was *the* main component of a GraphSLAM system. However the frontend is maybe even more important: If the backend does not get a proper graph to begin with, the results can be bad or even catastrophic. Just like with any least-squares optimization, a single outlier can drastically degrade or even downright ruin the result.

Therefore it is extremely important that the frontend ensures that no 'bad' graphs are passed on to the backend.

4.3.1 Working Principle

The frontend essentially has to accomplish four main tasks:

Incremental Scan Matching Once a new scan is received by the frontend, it needs to be aligned with the previous scan. Due to the fact that we do not assume to have any control input or motion model, this step is already non-trivial.

Loop Detection After adding a new scan (or a number of new scans) to the graph via incremental scan matching, we need to check whether any loops have been closed. If so, the relative pose between the two scans needs to be estimated and the actual loop closing is started.

Loop Closing After two potentially overlapping poses have been found, they are attempted to be registered given the relative pose estimate from the loop detection. If successful, an edge is added to the graph.

Evaluation/Monitoring The loop closing step can be error-prone. In a subsequent step, the consistency of all loop-closing edges should be monitored and evaluated as suggested by Olson in [Ols09b]. Inconsistent loop closing hypotheses can be disabled or deleted. Note that this step has not been implemented in this work due to time constraints. More details on the consequences of this can be found in Section 6.1.

After adding any nodes or edges to the graph, we run the backend in order to make the graph stay the maximum-likelihood solution at all times. Strictly speaking, running the backend is only needed after adding a loop-closing edge though.

We will now elaborate on the implementation of the individual tasks highlighted above.

4.3.2 Incremental Scan Matching

Whenever a new scan is received by the frontend, we need to add it to the graph and connect it to existing nodes in the graph by deriving Gaussian relationships between the nodes. We assume not to know anything about the motion model, and instead assume that subsequent scans were recorded in a way that they have a sufficient amount of overlap. This assumption effectively restricts the magnitude of the translation between the scans by about $\pm 30\,\mathrm{m}$ for all practical scenarios. It also allows the registration of each scan with its predecessor.

To perform this registration specific registration step, which we call incremental scan matching, we use the correlative scan matching algorithm explained in Section 3.2. For 2D scans, we simply use the original algorithm by Olson. For 3D scans, we use our extension explained in the same section. The 3D extension makes use of the ICP algorithm, which was explained in Section 3.1. In order to speed up this step, and in order to make it unbiased to uneven sampling, we use the Poisson sampling explained in Section 3.1.3.

3D scans are typically quite far apart, which means that even though the scans may have significant overlap, the appearance of different elements of the world may drastically change. A common example for this is that a wall or a pillar is observed from different sides in the two scans. We observed that due to the fact that the ICP algorithm will search for its nearest neighbors in a sphere, the data-association often fails by associating points from one side of a wall in the first scan, with points from the other side of the wall in the second scan. Because this affects many points systematically in the same way, the result of the algorithm is influenced and the two sides of the wall are "collapsed" to a single wall without thickness.

To cope with this problem, we also compute the normals for each scan as explained in Section 3.1.2, and use them to reject attempts to associated points having opposingly faced normals, i. e. cases where $\mathbf{n}_1^\mathrm{T}\mathbf{n}_2 < 0$.

Having applied either Olson's algorithm or our extension of the latter, depending on whether we are dealing with 2D or 3D data, we have found a relative pose along with a covariance matrix describing the uncertainty of the registration. They form a Gaussian pseudo measurement, which can be attached to an edge between the two nodes.

4.3.3 Loop Detection

After a scan was added to the graph via incremental scan matching, we need to check whether this scan closes a loop. In general, closing a loop in the graph means finding pairs of poses which are likely to have sufficiently overlapping sensor readings, even though they may be topologically far away from each other.

We will now derive a method that allows determining whether two poses are potentially overlapping. If they are, the corresponding scans can be matched and an additional edge can be added to the pose graph. We call the first part of this procedure loop *detection*, the second part of deriving the edge to be inserted into the graph is actual loop closing, which is explained in the next section.

At first we will look at two poses, that already have an edge connecting them. The edge corresponds to a Gaussian measurement of the associated relative pose or transformation. This measurement can be used to compute the probability that a point from the first scan corresponds with a point of the second scan, by inserting the points into the positional part of the Gaussian. Using this methodology, we could determine the number of points that are likely (e. g. probability $\geq 95\%$) to have a correspondence in the other scan.

The method above only works when we consider two poses that already have a Gaussian measurement between them. If we want to apply the same logic to two poses $\mathbf{x}_i$ and $\mathbf{x}_j$ in general, we first need to compute a Gaussian representing a hypothetical measurement between the two nodes. Mathematically this is the distribution $\mathrm{p}(\mathbf{x}_j \,|\, \mathbf{z}, \mathbf{x}_i)$, i. e. the distribution of $\mathbf{x}_j$ given all measurements and the pose $\mathbf{x}_i$. One way to compute this would be to take the graph, fix $\mathbf{x}_i$ using an absolutely certain measurement and run the backend on it. While this yields the correct means for all j, we lack the corresponding covariances. They could be computed by inverting the information matrix $\mathbf{J}^{\mathrm{T}}\mathbf{\Theta}\mathbf{J}$ from Section 2.3. Since the backend actually computes $\mathrm{p}(\mathbf{x} \,|\, \mathbf{z})$ it would contain *all* covariances[4], and not just the covariance of $\mathbf{x}_j$. The problem is that inverting the whole information matrix just to retrieve a single block from the diagonal is computationally very inefficient (and in fact infeasible, because the covariance matrix of a sparse information matrix is quite dense).

Therefore we consider an approximation for $\mathrm{p}(\mathbf{x}_j \,|\, \mathbf{z}, \mathbf{x}_i)$ which is called the *Dijkstra projection* [BNLT04, Ols09b, Ols08]. The idea of the algorithm is: Instead of considering all paths from node $\mathbf{x}_i$ to $\mathbf{x}_j$, as the just described correct method would do, we only consider the single most certain path. If there is just a single path to $\mathbf{x}_j$, the results are identical to the correct method. Otherwise, if there are multiple paths, the computed covariance is conservative in the sense that the covariance is always overestimated. This is because multiple paths to $\mathbf{x}_j$ means there are more measurements of that pose, and measurements will always decrease the covariance.

As the name suggests, the algorithm is implemented as a variant of the Dijkstra algorithm. Recall that the latter is parametrized with a start node, and then proceeds to compute the shortest path from this node to every other node in the graph. The key data structure is a sorted list (priority queue) of nodes that

[4]That is $\mathbf{cov}(\mathbf{x}_{k_1}, \mathbf{x}_{k_2})$ for *all* possible values of k_1 and k_2

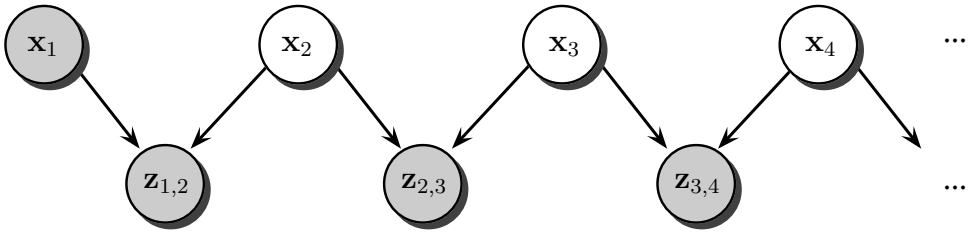


Figure 4.6: Bayes net in the situation where only a single chain of poses is considered, without any loop-closing edges.

are still to be expanded. The sorting order is determined by the distance to the node. The queue is initialized with the start node and the associated total distance zero. In every step, the algorithm greedily picks the node with the lowest distance and inserts its undiscovered neighbors into the queue until all nodes have been discovered. The distance attributed to newly inserted nodes is determined as the sum of distance of the current node and the "length" of the edge that is being followed.

This algorithm is now adapted as follows: Instead of associating a distance with each node, we now store the covariance matrix of the hypothetical measurement between the start node and the current node with each node. Instead of sorting the queue by distance, we now sort the queue by certainty, where certainty is expressed as the determinant of the associated covariance matrix. The reasoning behind that is that the determinant is related to the area/volume of the covariance matrix, which in turn influences the potential search area for the scan matcher [Ols09b]. This sorting order ensures, that we will always find the "most certain" path to every node. What remains is the question how to determine the covariance matrix of the nodes that are inserted into the queue, i. e. how the covariance of a node is propagated through an edge.

Covariance Propagation

First note that each entry in the queue corresponds to a certain path through the graph, i. e. every node on the path has one predecessor. Assuming the poses on the path are numbered $\mathbf{x}_1$ to $\mathbf{x}_t$, we want to know the parameters (more specifically the covariance) of the Gaussian distribution $\mathrm{p}(\mathbf{x}_t \mid \mathbf{z}_{2:t}, \mathbf{x}_{t-1})$. We can expand this term as follows[5]:

[5]Note that the derivation is similar to the inclusion of the motion model in the classical Bayes filter algorithm [TBF05].

$$\mathrm{p}(\mathbf{x}_t \,|\, \mathbf{z}_{2:t}, \mathbf{x}_1) = \int \mathrm{p}(\mathbf{x}_t \,|\, \mathbf{z}_{2:t}, \mathbf{x}_{t-1}, \mathbf{x}_1)\ \mathrm{p}(\mathbf{x}_{t-1} \,|\, \mathbf{z}_{2:t-1}, \mathbf{x}_1)\, d\mathbf{x}_{t-1} \tag{4.13}$$

$$= \int \mathrm{p}(\mathbf{x}_t \,|\, \mathbf{z}_t, \mathbf{x}_{t-1})\ \mathrm{p}(\mathbf{x}_{t-1} \,|\, \mathbf{z}_{2:t-1}, \mathbf{x}_1)\, d\mathbf{x}_{t-1} \tag{4.14}$$

The second step follows from conditional independence assumptions encoded in the Bayes net of this situation, which is given in Figure 4.6. The term $\mathrm{p}(\mathbf{x}_t \,|\, \mathbf{z}_t, \mathbf{x}_{t-1})$ is a Gaussian describing how the state t results from the last state $t-1$ and the current measurements. In our case this is simply:

$$\mathrm{p}(\mathbf{x}_t \,|\, \mathbf{z}_t, \mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \mathbf{x}_{t-1} \oplus \mathbf{z}_t, \Sigma_{\mathbf{z}_t}) \tag{4.15}$$

The second term, $\mathrm{p}(\mathbf{x}_{t-1} \,|\, \mathbf{z}_{2:t-1}, \mathbf{x}_1)$ is exactly the same likelihood as we are trying to compute, only up to node $t-1$ instead of t. Note that at $t=1$ it is a Gaussian with infinitely small covariance. Therefore both terms are Gaussians and the result can be expressed recursively as:[6]

$$\mathrm{p}(\mathbf{x}_t \,|\, \mathbf{z}_{2:t}, \mathbf{x}_1) = \mathcal{N}(\mathbf{x}_t; \mathbf{x}_{t-1} \oplus \mathbf{z}_t, \mathbf{J}_1 \Sigma_{t-1} \mathbf{J}_1^{\mathrm{T}} + \mathbf{J}_2 \Sigma_{\mathbf{z}_t} \mathbf{J}_2^{\mathrm{T}}) \tag{4.16}$$

$$\text{with } \mathbf{J}_1 = \left. \frac{\partial \left((\mathbf{x}_{t-1} \boxplus \delta) \oplus (\mathbf{z}_t \boxplus \epsilon) \right) \boxminus (\mathbf{x}_{t-1} \oplus \mathbf{z}_t)}{\partial\, \delta} \right|_{\delta = \mathbf{0}}$$

$$\text{and} \quad \mathbf{J}_2 = \left. \frac{\partial \left((\mathbf{x}_{t-1} \boxplus \delta) \oplus (\mathbf{z}_t \boxplus \epsilon) \right) \boxminus (\mathbf{x}_{t-1} \oplus \mathbf{z}_t)}{\partial\, \epsilon} \right|_{\epsilon = \mathbf{0}}$$

Note that the notation $\mathcal{N}(\mathbf{x}_t; \dots)$ indicates that the variable inside the Gaussian is $\mathbf{x}_t$ instead of the typical $\mathbf{x}$.

The last equation yields the actual propagation rule for the covariance: When we have the covariance Σ_{t-1} of the current node $t-1$, and want to follow an edge, we can compute the covariance of the node t by computing $\Sigma_t = \mathbf{J}_1 \Sigma_{t-1} \mathbf{J}_1^{\mathrm{T}} + \mathbf{J}_2 \Sigma_{\mathbf{z}_t} \mathbf{J}_2^{\mathrm{T}}$. This is the main step for enabling the Dijkstra projection to use Gaussians on manifolds.

This concludes the Dijktra projection algorithm. The pseudo code is given in Algorithm 5. Note that the derivation above and the pseudo code are slightly simplified, as they ignore the fact that sometimes edges have to be traversed against their edge direction. To handle this case properly, one can simply invert the edge measurement and then proceed as if all edges pointed away from the start node.

Further Simplifications

In the introduction to this section, we suggested computing the number of potentially corresponding scan points in order to determine whether the sensor readings

[6] The derivation for why the integral of the product of the two Gaussians results in the given Gaussian can be found in the derivation of the Kalman filter prediction in [TBF05]

Algorithm 5 Dijkstra Projection

Input: $startNode$ – Start Node
Input: $graph$ – Graph
Output: $sigmas$ – Covariance matrices for each node

Q=[(startNode,$\mathbf{0}$)] // init queue with startNode, zero covariance matrix
$visited$=$\emptyset$

while (!isEmpty(Q))
{
    $(curNode, \Sigma_{t-1})$ = extractMin(Q) // select min. determinant
    $sigmas[curNode] = \Sigma_{t-1}$
    $visited = visited \bigcup \{curNode\}$
    foreach ($(src,dst,\mathbf{z}_t,\Sigma_{\mathbf{z}_t})$ **in** getOutEdges($graph$,$curNode$))
    {
        if ($dst \in visited$)
            continue
        $\Sigma_t = \mathbf{J}_1\Sigma_{t-1}\mathbf{J}_1^{\mathrm{T}} + \mathbf{J}_2\Sigma_{\mathbf{z}_t}\mathbf{J}_2^{\mathrm{T}}$
        insert(Q,(dst,Σ_t))
    }
}
return $sigmas$

of two poses overlap. A computationally cheaper variant suggested in [Ols09b] is to not consider the individual laser scan points. Instead, the idea is to reduce every scan to a sphere whose center $\mathbf{c}$ equals the centroid of the laser scan points, and whose radius r is defined by the mean distance to the centroid.

In order to test whether the two spheres potentially overlap, we first compute shortest vector $\mathbf{s}$ connecting the two spheres:

$$\Delta\mathbf{c} = \frac{\mathbf{c}_j - \mathbf{c}_i}{||\mathbf{c}_j - \mathbf{c}_i||} \tag{4.17}$$

$$\mathbf{s} = \Delta\mathbf{c} \cdot \max(0, ||\mathbf{c}_j - \mathbf{c}_i|| - r_i - r_j) \tag{4.18}$$

This vector is now inserted into the previously computed Gaussian, or more practically into the Mahalanobis distance equation in the exponent. The idea is to check whether the sphere around $\mathbf{c}_i$ intersects with the ellipsoid around $\mathbf{c}_j$, which has previously been extruded by an amount of r_j:

$$\mathbf{s}^{\mathrm{T}} \, \Sigma|_{\mathrm{pos}} \, \mathbf{s} \leq \chi^2_{\max} \tag{4.19}$$

The parameter $\chi^2_{\max}$ is used to vary the size, i. e. the total contained probability mass, of the ellipsoid. We use $\chi^2_{\max} = 3$. The variable $\Sigma|_{\mathrm{pos}}$ is the positional part of the covariance matrix. If the 3D pose representation derived in Section 2.2.2 is used for example this would be the upper left 3×3 block of the actual covariance matrix.

A visual explanation of this overlap test can be found in Figure 4.7.

4.3.4 Loop Closing

The actual loop closing comprises the derivation of a Gaussian pseudo measurement, given the information that two poses/scans are potentially overlapping and given an estimated relative pose between them along with uncertainty estimate. Just like during the incremental scan matching step, we use the correlative scan matching algorithms described in Section 3.2 to perform the registration. In this case however, we have a prior and an uncertainty estimate of the latter, which are derived from the Dijkstra projection algorithm.

In the 2D case, we again simply stick to the original correlative scan matching algorithm by Olson. In the 3D case, everything depends on the uncertainty estimate. If it indicates a large uncertainty, e. g. translational uncertainties of $\geq 0.5\,\mathrm{m}$, we perform exactly the same thing as in the incremental scan matching step, which is our 3D extension of Olson's algorithm. If the estimated uncertainty is small, we apply the ICP algorithm only, since the initial guess of the relative pose is most likely good enough for the ICP algorithm to converge to the correct local minimum.

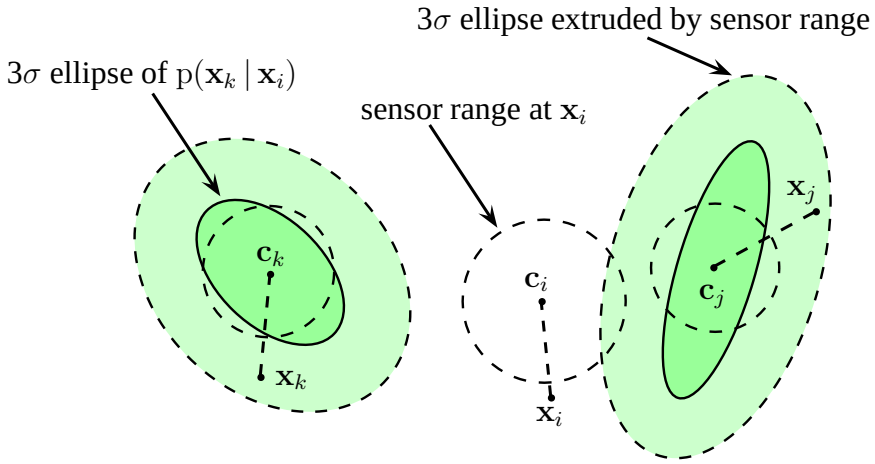


Figure 4.7: The image shows three poses $\mathbf{x}_i$, $\mathbf{x}_j$ and $\mathbf{x}_k$. At each pose, the laserscanner performed a measurement whose centroid was located at $\mathbf{c}_i$, $\mathbf{c}_j$ and $\mathbf{c}_k$ respectively. Additionally, an average measurement distance from this measurement was determined, and is visualized with the dashed circles around the centroids. For $\mathbf{x}_k$ and $\mathbf{x}_j$, the 3σ uncertainty ellipses of $\mathrm{p}(\mathbf{x}_k \,|\, \mathbf{x}_i)$ and $\mathrm{p}(\mathbf{x}_j \,|\, \mathbf{x}_i)$ are shown in dark green around the sensor data centroid corresponding the poses. They represent an area where the true location of the respective centroid can be found with high certainty. The light green dashed ellipses are variants of these uncertainty ellipses, which have been extruded the sensor range. Thus this area represents an area that possibly contains sensor data for each respective pose. — We consider a pose to be potentially overlapping with the current pose $\mathbf{x}_i$, if the extruded ellipses around the centroids intersect with the circle around the sensor range of $\mathbf{c}_i$. In the depicted sample, $\mathbf{x}_j$ is potentially overlapping with $\mathbf{x}_i$ while $\mathbf{x}_k$ is not.

4.4 Implementation

The implementation of our GraphSLAM system was done in C++, making heavy use of templates. Therefore, large parts of our implementation are header-only. The reason for this is that one of our goals was to have a very generic implementation where the 2D and 3D variants share as much code as possible.

In order to represent the pose graphs, we use the Boost Graph Library (BGL)[7]. All of our linear algebra is done with Eigen[8], which is a truly excellent templated math library.

4.4.1 Backend

The maximum-likelihood estimation in our backend is carried out using the sparse matrix classes included in the Eigen library. Before starting the iteration, we initialize a sparse matrix with the expected non-zero blocks. We then iterate over the edges in the graph, summing up the corresponding non-zero blocks as indicated in the Equation (4.11) in Section 4.2.1.

The equation system is solved with a sparse Cholesky solver. We use the SparseLLT class in Eigen's SparseExtra module, together with the CholMod backend.

The Jacobians of the error function $\mathrm{e}_{\mathbf{z}_{ij}}$ can either be computed with numerical differentiation provided by Eigen's NumericalDiff module, or with the AutoDiff method provided in Eigen's AutoDiff module. Making them work on the same functor ended up a being a little tricky and requiring heavy template use. Yet we chose to do so in order to avoid duplicate code.

Note that all of the modules mentioned here except the "core" sparse matrix functionality is currently marked as "unsupported" which indicates that their interfaces are subject to change or that the implementation is incomplete or immature. We have encountered several issues in AutoDiff for example, but worked around them in order to be able to test this exciting feature.

4.4.2 Frontend

Our extension of the correlative scan matching algorithm to 3D used the ICP algorithm as one of it's sub steps, which again needed the normals of the point clouds to be aligned. As indicated earlier, the nearest neighbor queries, which are needed for computing the normals and during the operation of the ICP are done with the FLANN library[9].

[7] Website: `http://www.boost.org/doc/libs/release/libs/graph/`
[8] Website: `http://eigen.tuxfamily.org`
[9] Website: `http://people.cs.ubc.ca/~mariusm/index.php/FLANN/FLANN`

In the loop detection step, we needed to estimate the covariance of a node relative to another node using the Dijkstra projection. The Dijkstra algorithm included in the BGL turned out to be unsuitable for this purpose. The main problem was its inability to traverse edges against their edge direction. While our edges did have a meaningful direction, these directions were not supposed to restrict traversability. Additionally it was not quite clear how one could make it pick the path with the highest total certainty in every step.

In general, the BGL turned out to be a little clumsy and immature in some aspects, which regularly slowed down or halted the development. Yet it is still an interesting library, with and we actually adopted some of their ideas such as the use of property maps for our own algorithms.

In order to speed up the computations, we developed multi-threaded implementations of the normal computation and the ICP algorithm. All other algorithms run in a single thread only, but we are considering to make use of parallel programming in the correlative scan matching as well.

Chapter 5

Experiments and Results

This chapter presents the results of the experiments with our GraphSLAM system. We divided this chapter into a section which is only concerned with open datasets, where the data-association problem has been solved already. This allows evaluating the backend separately without the need of any frontend. After that we will analyze the performance of the whole system on real-world datasets.

Even though all of our algorithms work both in 2D and 3D in principle, the frontend is focused on 3D worlds. One reason is that 2D scans tend to have more ambiguities than 3D scans and also the achievable accuracy of scan-to-scan matching is lower, since there is a much lower amount of matched points. In turn 2D datasets typically have much more frequent measurements. All these 2D-specific characteristics require proper treatment in the frontend. Olson for example suggests locally grouping the scans before passing them on to the actual frontend [Ols08]. Though we have done some basic experiments with real 2D data, we eventually decided to focus on 3D data only as they represent our primary area of interest.

We would also like to point out that the evaluation of SLAM problems in general is quite difficult: There are hardly any datasets with ground truth pose information available, which explains the lack of quantitative comparisons the second section of this chapter.

All time measurements were performed on a machine with a Intel Core-i7 920 Quad Core processor with 2.66 Ghz and enabled HyperThreading.

5.1 Synthetic Datasets

In order to evaluate the backend, we will use a number of publicly available graphs that need to be optimized. For these graphs, the data-association problem, which is normally the task of the frontend, has been solved already.

All of these datasets are simulated trajectories of an imaginary robot, where subsequent poses are connected by constraints. In addition to that, there are loop closing constraints, which were generated when a robot reaches a position similar to a previously visited location. All of the constraints are corrupted by noise.

Note that despite these datasets being synthetic, the obtained χ^2[1] error can never reach zero. Even with the "perfect" node configuration, each constraint will report a slight error corresponding to the residual of the individual measurements. Also the "ground truth" node configuration that was used to create the dataset does not correspond to the minimum χ^2 error. However with a dense graph, where there are multiple paths connecting any two vertices, the χ^2 error of the ground truth node configuration gets closer and closer to minimum χ^2 error.

These properties make the evaluation of the results a bit difficult. It does not make much sense to compare the results directly to the ground truth node positions. But also the χ^2 error alone is not a good indicator of the overall quality of the result. It is only an indicator for how well the constraints defined through the individual measurements are satisfied.

5.1.1 2D

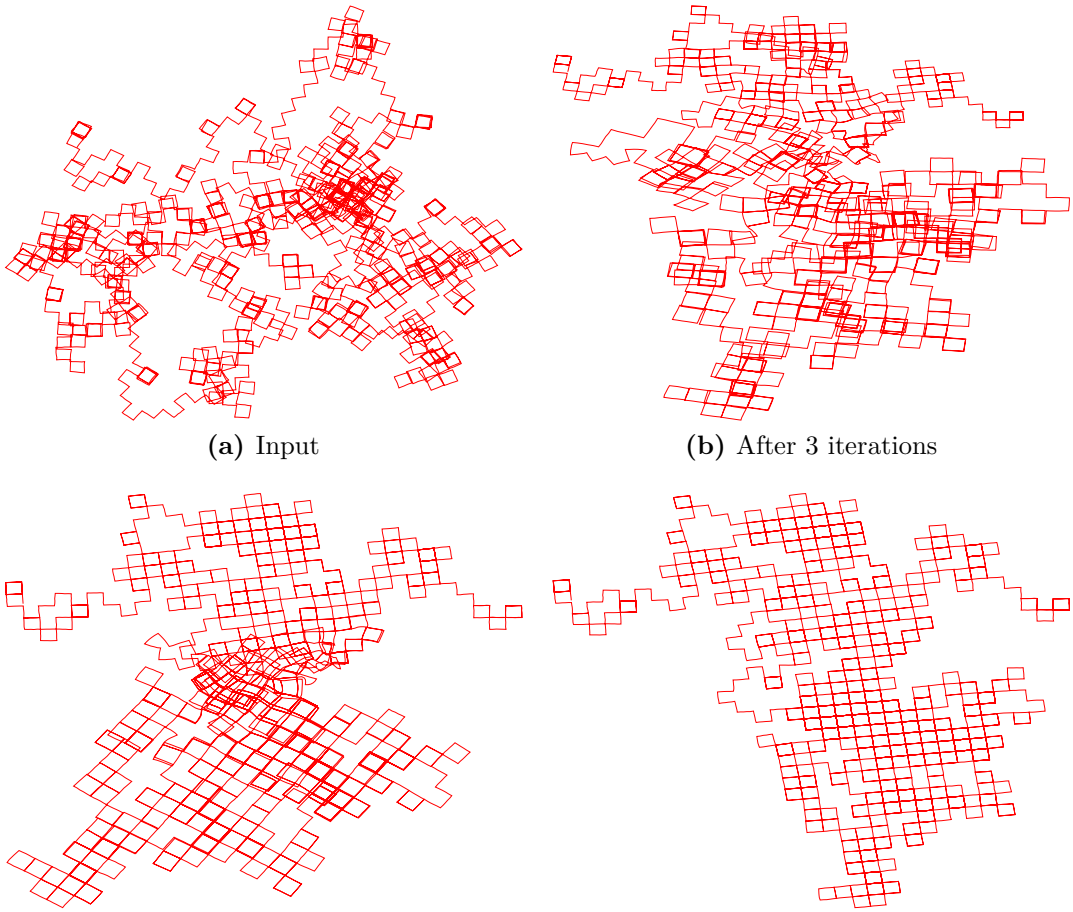
The first dataset we consider is the "W10 k" dataset created by Grisetti et al. [GSGB07][2], representing a simulated trajectory in a grid-world, consisting of a total of 10000 poses and around 64000 edges connecting them. As explained above, loop closing constraints were generated once the simulated robot reached a position similar to a previously visited position once again. Additionally each constraint is corrupted by noise.

Our implementation of the backend is able to optimize the dataset within 12 iterations. Each iteration takes between 693 ms and 705 ms, which is unsurprisingly constant. This is because at every iteration, the same computations have to be carried out – the only difference is the actual numbers. An average of about 460 ms is spent updating the linear system, while about 196 ms are spent solving the linear system (see Section 2.3). The rest of the time is used to update the current state estimate, compute error estimates and other minor things. The total computation time is around 11 seconds, including the time to load the graph into memory. When the derivatives are computed using AutoDiff instead of numerically, the time to update the system reduces to about 176 ms, which lowers the total computation time to around 8 s.

A visualization of the graph at various steps during the optimization can be found in Figure 5.1, the development of the χ^2 error can be found in Figure 5.2.

[1] The χ^2 error is equal to $\mathbf{e}_{\mathbf{z}}(\mathbf{x})\Theta\mathbf{e}_{\mathbf{z}}(\mathbf{x})$ from Section 2.3. It is the quantity which is minimized during the maximum-likelihood estimation.

[2] It is available at http://www.openslam.org/toro.html



(a) Input **(b)** After 3 iterations

(c) After 5 iterations **(d)** After 12 iterations (converged)

Figure 5.1: Graph of the "W10 k" dataset at different points during the execution of the optimization algorithm. Note that only the trajectory is shown and not the edges between the poses.

This figure also shows a comparison of our backend with the SLoM [Her] and TORO [GGS+07] backends. Since the used approach is more or less the same as SLoM and only the implementation differs, development is almost identical (the actual numbers differ a bit of course).

TORO in comparison is able to reduce the error much quicker at the beginning, and then struggles to find a good solution. Also the best solution found by TORO is worse than the solution by SLoM or our implementation.

As for the computation time, TORO required a total of about 13 s for the 30 iterations. It did not report convergence though. SLoM performed slightly faster than our approach requiring around 6 seconds. At least a part of the reason for the better performance seemed to be the quicker load time for the data files though.

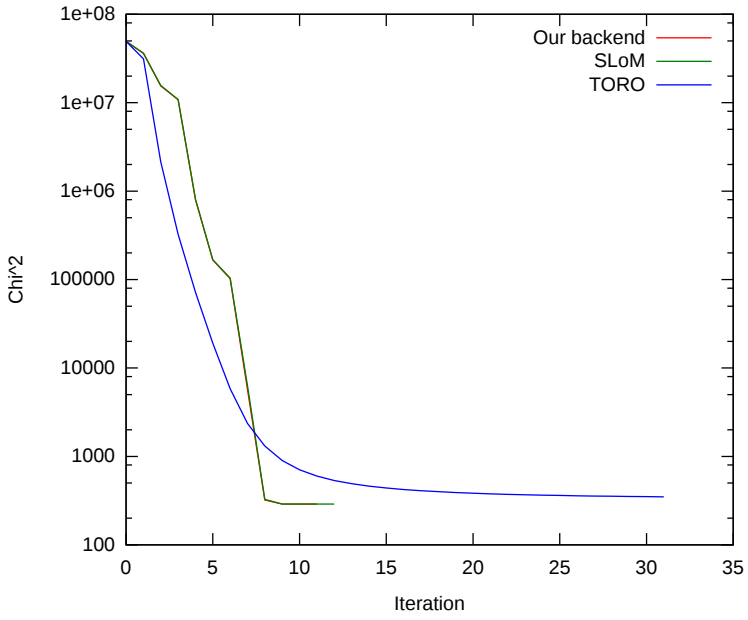


Figure 5.2: Comparison of the χ^2 errors of different optimizers on the "W10 k" dataset. Note that the graphs for our approach and for SLoM are almost identical and thus hard to distinguish.

5.1.2 3D

The next dataset is a 3D dataset called "sphere-mednoise" created by Grisetti et al. [GSB09][3]. It is a simulated trajectory on a sphere, which has been corrupted with noise. It consists of 2200 nodes with 8647 edges, therefore representing a relatively dense dataset, where every node has around 8 neighboring nodes. The high connectivity makes it a little unrealistic, though it is probably a good test case for optimization algorithms, because it is supposedly quite difficult to optimize.

Using numerical differentiation, our algorithm needed 23 iterations to optimize the dataset, which took about 15 seconds. In every iteration around 167 ms were spent on updating the linear system and 439 ms were spent solving it. A visualization of the graph at different points in the iteration can be found in Figure 5.4.

In this dataset, we found an interesting deviation between AutoDiff and numerical differentiation. One would actually expect AutoDiff to perform better in general, simply because it always computes the exact Jacobians, whereas numerical differentiation only yields a finite-difference approximation. As shown in Figure 5.3 though, AutoDiff appears to get almost "stuck" for several iterations, hardly improving on the χ^2 error at all. In fact in all of our experiments, we have never found a case where AutoDiff performed significantly better than numerical

[3]It is available at http://www.openslam.org/toro.html

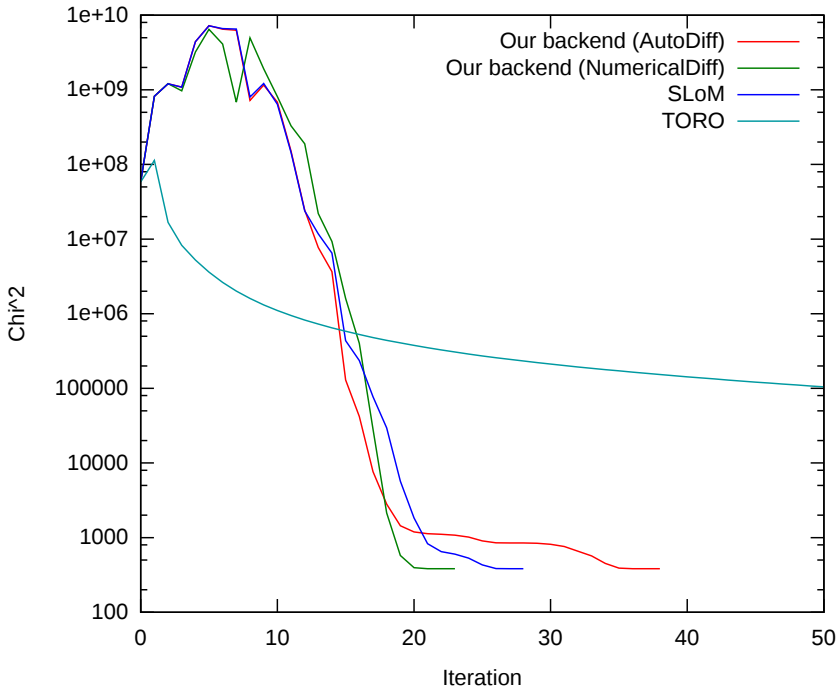


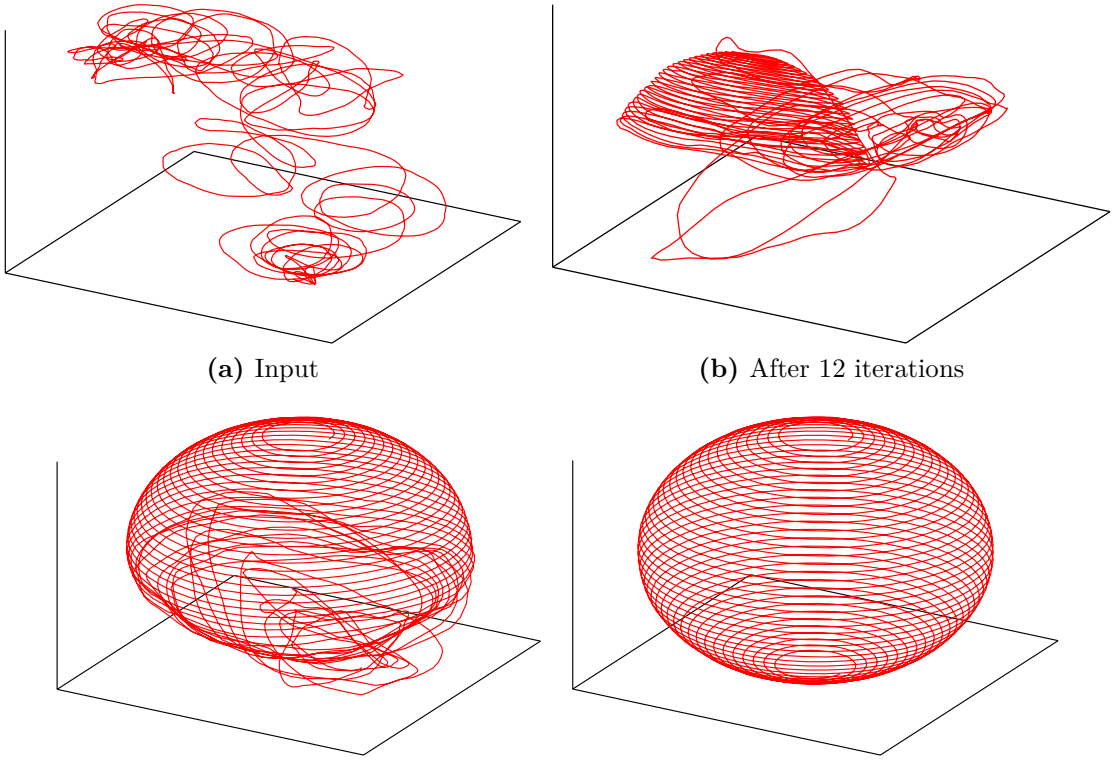
Figure 5.3: Comparison of the χ^2 errors of different optimizers on the "sphere-mednoise" dataset

differentiation. Our *guess* is that the finite differencing actually has a smoothing effect on the cost function, providing better guidance to the maximum-likelihood solution.

In all cases however, a single iteration using AutoDiff is faster than using numerical differentiation. In this case AutoDiff needed only 129 ms to update the linear system.

As expected, SLoM is again on par with our implementation. The slightly different convergence behavior can be blamed on different libraries for solving sparse linear systems – we use CholMod whereas SLoM uses CSparse – as well as different methods or step sizes to compute the required Jacobians. What is interesting though is that while it takes 29 iterations for it to converge, the computation time of about 40 seconds is much higher than ours.

TORO in comparison performs much worse in this case due to the higher non-linearity and complexity of the problem. Even after 300 iterations taking about 3 minutes of computing time, the χ^2 error is still sits at about 9600.



(a) Input **(b)** After 12 iterations

(c) After 15 iterations **(d)** After 23 iterations (converged)

Figure 5.4: Visualization of the optimization steps of the "sphere-mednoise" dataset. The first image shows the unaltered input. The second image was taken after 12 iterations, the third one after 15 iterations and the last one after the algorithm converged, which was after 23 iterations. Note that only the main trajectory is shown, i. e. loop closing edges are omitted.

5.2 Real Datasets

Real datasets evaluate the whole SLAM system. All parts have to work together in order to produce a globally consistent map. Yet our analysis focuses on the frontend, since we already evaluated the backend in the previous section and since we noticed that it is typically not the cause of problems.

For both datasets presented here, we have set the search window of the correlative scan matching algorithm, which is needed for the incremental alignment of the scans, to $\pm 20\,\mathrm{m}$ in x and y direction and to $\pm\pi$ for the yaw angle. Before applying the ICP algorithm, the point clouds were downsampled to 80 k points.

In some cases, the correlative scan matching algorithm failed and we needed to provide initial guesses for x, y and yaw in these cases. The rest of the incremental alignment steps, being the z estimation and the subsequent ICP algorithm work correctly at all times. An explanation for potential reasons for the failure of this algorithm can be found in Section 5.3.

Also note that we do not provide runtimes for the backend, even though they are contained in the total runtime of the algorithm. This is because it is almost negligible as the graphs in these datasets are so small.

5.2.1 "House" Dataset

The "House" dataset was recorded with a MappingCube (see Figure 5.5) by the V&R Vision&Robotics GmbH[4]. It is essentially a 3D laser scanner, which performs 3D scans by rotating a commercially available 2D laser scanner. The scan resolution is variable, but we set it up to capture around one million points per scan. In order to measure a large area, the scanner is first set up at a certain location. After that, the scanning process is started via an Android-based remote control. When the scanning procedure is finished, the device is carried on to the next location. At the chosen resolution, recording one scan requires about 30 seconds.

Our dataset consists of a total of 117 scans and took about 2 hours to capture (which means that roughly another 30 seconds per scan were needed to move the scanner to the next location). It is a mixed in- and outdoor scenario which was recorded in and around a typical house with a surrounding garden. It contains quite a few loops which even cover multiple floors of the building. The scans cover a total of three floors, and there are even scans which were taken on the roof of the garage which is adjacent to the house in order to close loops with scans which were taken in the area around the garage.

[4]Website: http://www.vision-robotics.de

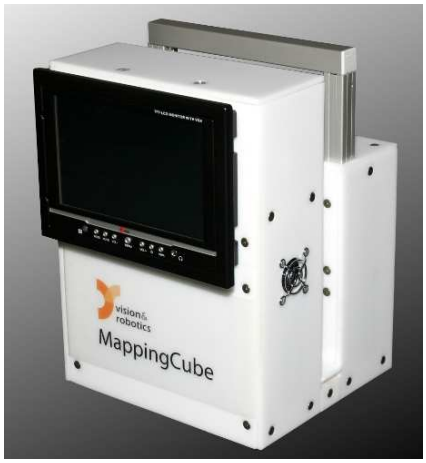


Figure 5.5: MappingCube by the V&R Vision & Robotics GmbH

Figure 5.6 shows pictures of the point cloud after running our algorithm on the data. The generated graph is shown in Figure 5.7. Figure 5.8 shows the current point cloud before and after closing the first loop.

Our system required around 23 minutes to fully process the dataset, including the start of the application, reading the scan files, computation of normals, incremental scan matching, loop closing, and the execution of the backend. 19 loop closing edges were inserted by the frontend. The correlative scan matching which is part of the incremental scan matching failed for 5 scan pairs.

The whole incremental scan matching typically took around 9 seconds. A detailed analysis of the runtimes of the sub algorithms involved in the incremental scan matching can be found in Figure 5.9. The time for loop closing is very similar, except that the normals do not have to be computed again, and the estimation is slightly faster because a prior is available.

5.2.2 Freiburg

The "Freiburg" dataset was recorded by Bastian Steder at the University of Freiburg and is available online[5]. It features a total of 77 laserscans taken on a trajectory of about 700 m. The trajectory consists of two large loops with a quite large overlapping area in the middle. The individual laser scans were taken with a SICK laser scanner on a pan-tilt unit and consist of about 150 k to 200 k points each. An aerial image of the area can be seen in Figure 5.10. The dataset also contains odometry information, which we did not end up using though, as we decided not to use any control input.

[5]The dataset is available at: `http://ais.informatik.uni-freiburg.de/projects/datasets/fr360/`

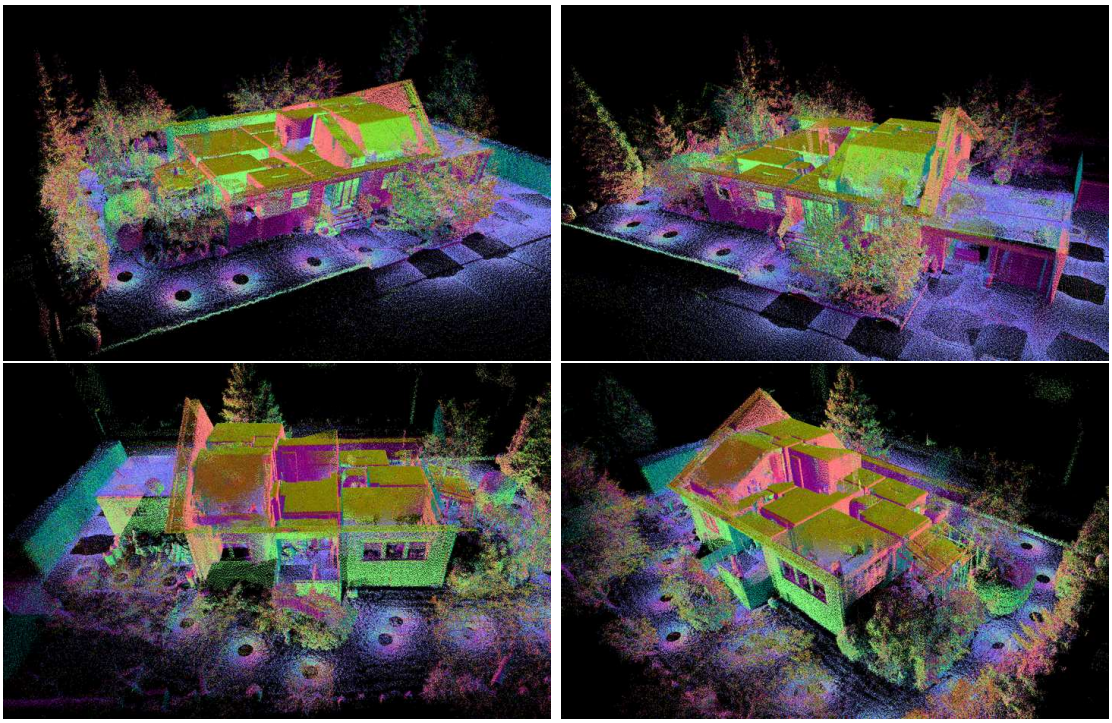


Figure 5.6: Resulting point cloud of the "House" dataset after optimization from different point of views. Only $\frac{1}{10}$ of the points are shown, which is still around 12 million points.

Figure 5.7: Orthographic projection of the graph generated in the "House" dataset. The colors ranging from purple to orange encode the height of the edges, where purple indicates the lowest height.

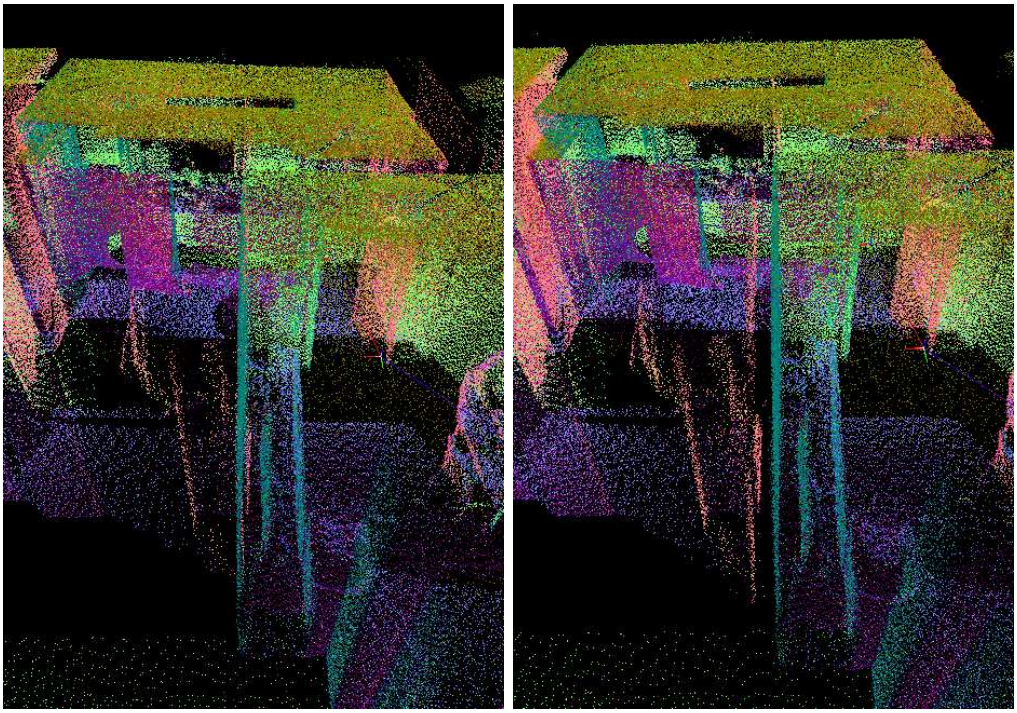


Figure 5.8: Point cloud of the "House" dataset before and after closing the first loop. One can (hopefully) see the room align with the previous measurements.

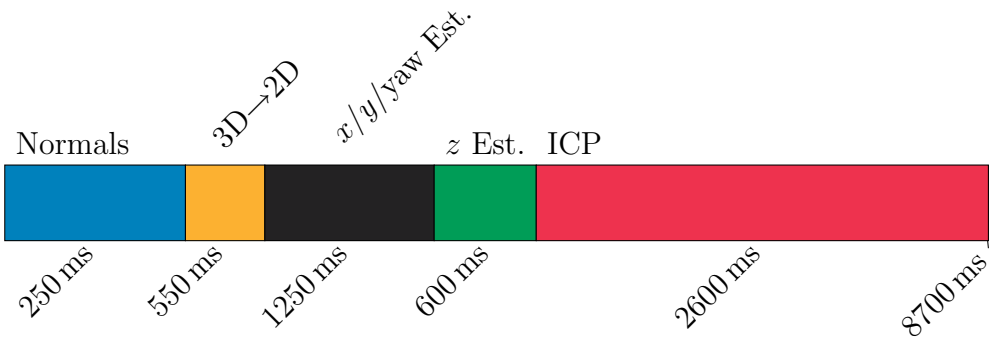


Figure 5.9: Execution times for the algorithms which are part of incremental scan matching in the "House" dataset. Note that the ICP step includes downsampling the scans, as well as the computation of the covariance matrix. The term "3D→2D" refers to the algorithm which projects vertical structures down to the xy-plane which was explained in Section 3.2

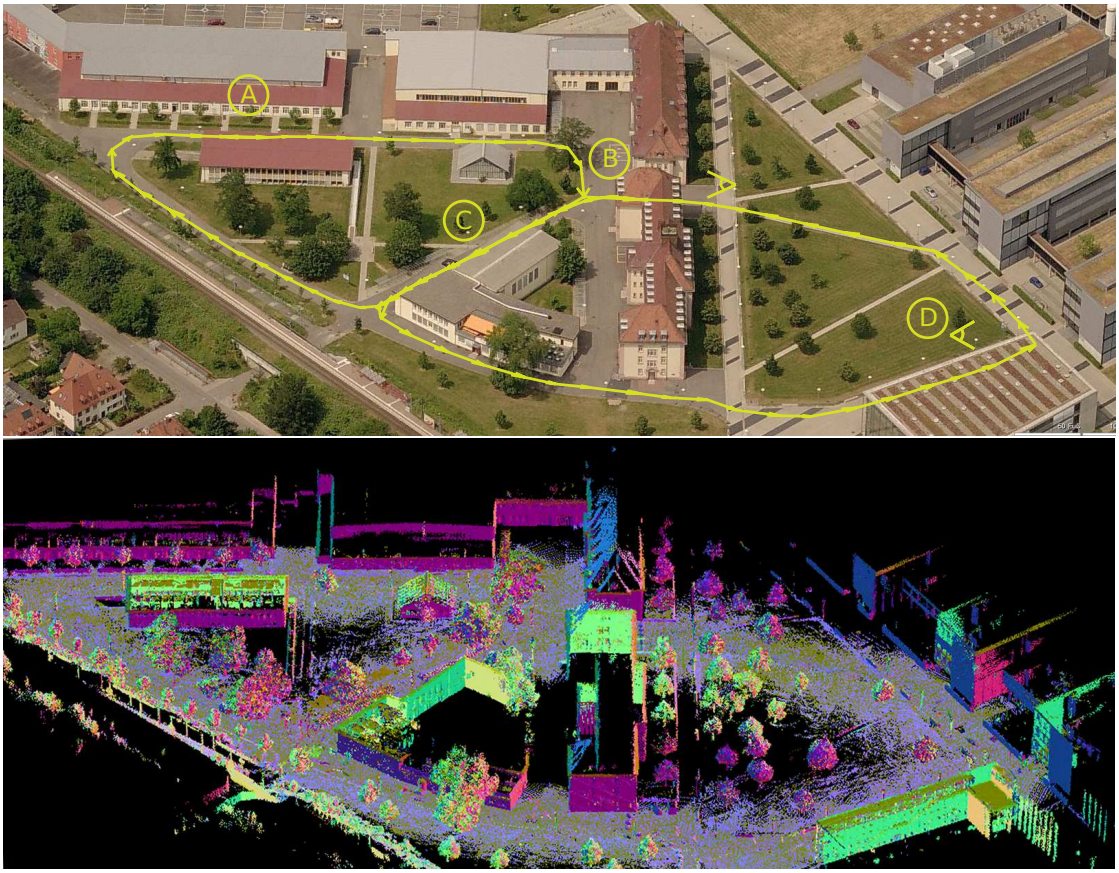


Figure 5.10: Top: An aerial image of the mapped area, taken from Microsoft Bing Maps (http://maps.bing.com). Bottom: An orthographic projection of the combined point cloud of all scans after optimization from a similar point of view.

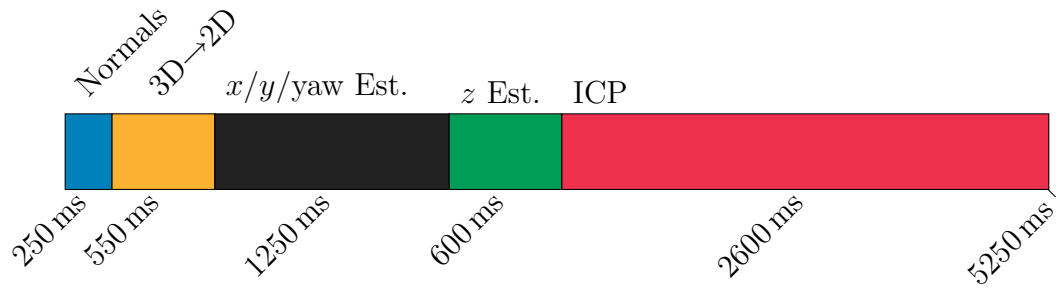


Figure 5.11: Execution times for the algorithms which are part of incremental scan matching in the "Freiburg" dataset. Note that the ICP step includes downsampling the scans, as well as the computation of the covariance matrix.

The registration process took a total of 6 minutes and 37 seconds. Again, this includes the start of the application, reading the scan files, computation of normals, incremental scan matching, loop closing, and the execution of the backend. A total of 12 loop closing edges were inserted in the process. The correlative scan matching algorithm failed for three scan pairs. The resulting point cloud is shown in Figure 5.10. The computed graph is overlaid on top of this image.

The whole incremental scan matching typically took around 5 seconds. A detailed analysis of the runtimes of the sub algorithms involved in the incremental scan matching can be found in Figure 5.11. In comparison to the previous dataset, our algorithms performed a bit faster. The reason for that is mainly that the point clouds are more than 5 times as big as in the "House" dataset, and not all sub algorithms – especially the computation of the normals – are immune to this.

Loop closing edges are present at the areas A, B and C (see top of Figure 5.10). However actual loops are only closed at A and B. Figure 5.12 shows two images taken at position B before and after closing the loop at that position. One can clearly see the wall suddenly lines up with the previous measurements.

Note that when viewed from the side, the area D on the right seems to be slightly bent downwards (see top of Figure 5.13), which is supposedly wrong. Indicators for that are also the covariance ellipsoids visualized at the bottom of Figure 5.13, which have been computed with the Dijkstra projection explained earlier. They show a high uncertainty in the height in the considered area. Imprecision in the height are quite common in outdoor scenarios. Roughly vertical walls or other obstacles help to constrain both the x, y and the yaw angle. In order to constrain pitch, roll and z, horizontal structures are needed. While indoor scenarios have both the floor, the ceiling and other miscellaneous objects, outdoor scenarios often have the ground as more or less the only horizontal structure. To make things worse, the ground is often undersampled due to the low distance of

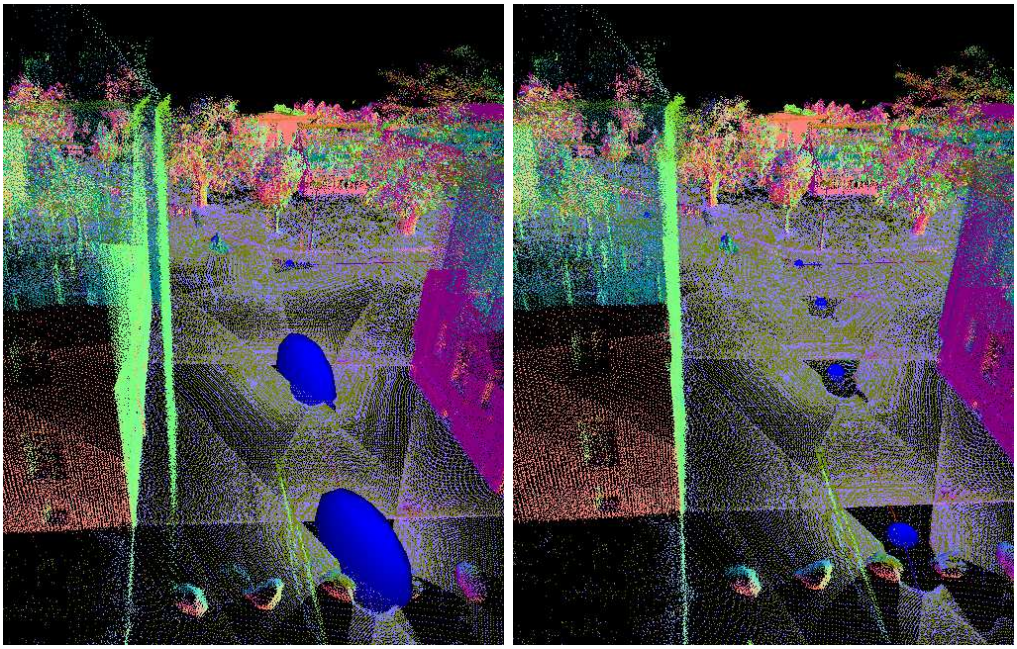


Figure 5.12: These pictures were taken at the stylized eye near position B before (left) and after (right) closing the loop. The blue ellipsoids rendered at each pose indicate the uncertainty relative to the first pose and were computed using the Dijkstra projection algorithm. After the loop is closed, the uncertainty of the last poses in the loop abruptly decreases, since there is now a new, more certain path to that location.

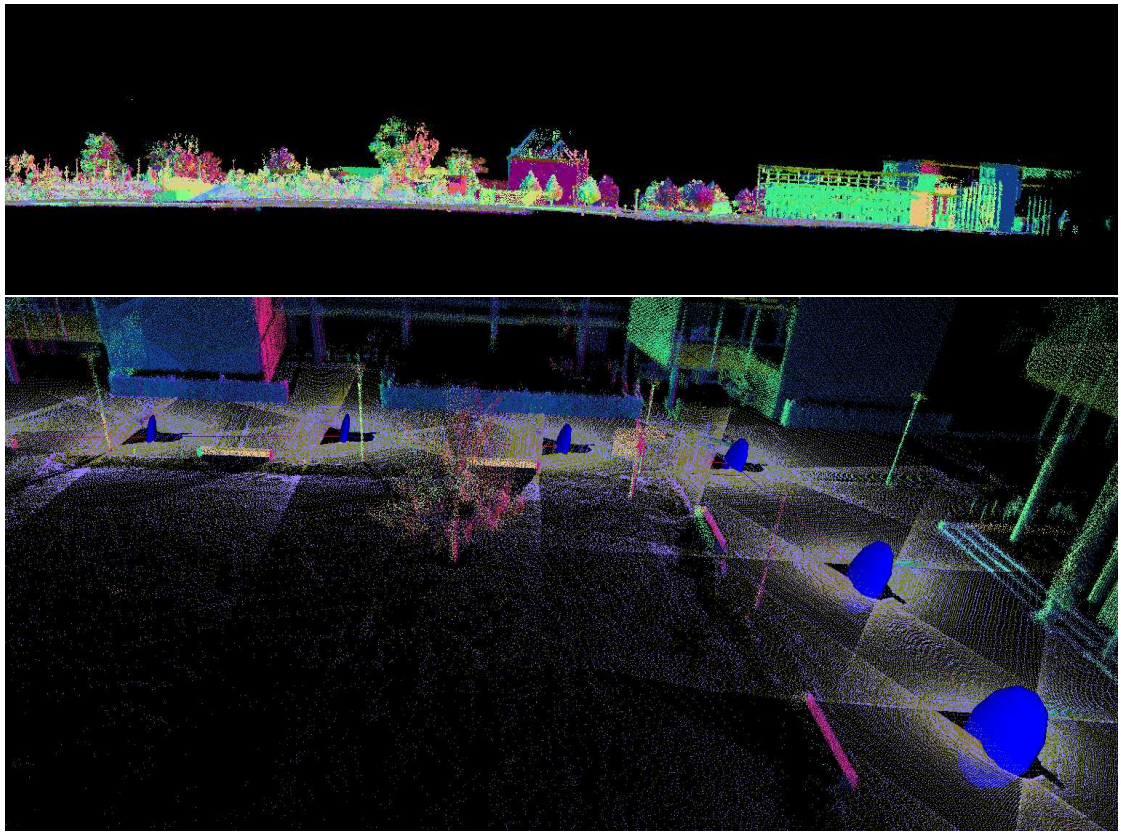


Figure 5.13: Point cloud of the "Freiburg" dataset after optimization taken at position D. Top: The right part of point cloud appears to be slightly bent downwards. Bottom: The σ-covariance ellipsoids in these areas. They have been computed using the Dijkstra projection.

the laserscanner to the ground plane, and noisy due to grass or other asperities of the surface.

5.3 Fail Cases

As indicated earlier in this chapter, the backend appears to be extremely reliable, given the frontend provides a locally correct graph, i. e. a graph where any edge approximately correctly aligns the scans it connects. The critical part which is much more likely to fail is the frontend. There are two things which typically go wrong in case *something* goes wrong:

5.3.1 Incremental Scan Matching

In the previous chapter we assumed not to use any control input, which led to the fact that we do not really have a motion model, which in turn means that we are completely reliant on the scan matching between two subsequent scans to succeed – there is nothing to fall back on. The algorithm we use for incremental scan matching is the extension of the correlative scan matching algorithm presented in Section 3.2 to 3D. The critical part of this algorithm is the estimation of the x, y translations and yaw angle: If any part of the algorithm fails, it's typically this part. The main reasons for it to fail are:

- The two scans to be matched are in fact too far apart. Therefore the amount of overlap is not sufficient.
- There are geometrical ambiguities in the scans such as repeating structures
- The heuristic to reduce the 3D world to only the 2D vertical structures fails. An example for this would be an outdoor scenario with only sloped hills, but no clear objects such as trees or bushes.

An inherent problem of this type of matching is that one has to find a trade off between forcing the algorithm to overlap the scans and allowing the algorithm to accept a low amount of overlap in favor of good matches. For example one can always find relative poses between the scans where all points with a neighbor in the other scan match perfectly, simply by moving them so far apart that only very few points actually have neighbors. Increasing the penalty for points without neighbors forces the algorithm to make the scans overlap more so that more points have neighbors. Increasing the penalty by too much makes the algorithm simply align regions with a lot of points, even if they dont fit well, just in order to avoid the penalty.

The cause of this problem is the fact that the reference scan (or map) is incomplete. This is always the case in a SLAM like context and thus this problem is probably unavoidable at this point.

5.3.2 Loop Closing

In order to detect loops, we use the potential overlap test together with the Dijkstra projection presented in Section 4.3.3. After identifying pairs of scans which can potentially be matched, we again use the correlative scan matching algorithm. The problem is that it is hard to tell whether two scans have a sufficient amount overlap

in order to be matched, and even after matching them it is hard to tell whether the matching actually succeeded[6].

This can lead to incorrect edges being inserted into the graph, which breaks the least-squares nature of the optimization algorithm, which is not robust to outliers. Also note that the covariance associated with these edges does often not account the uncertainty induced by potentially wrong data-association. Therefore the edge is wrong and yet fairly confident to be right.

[6]After all if one could define a measure that tells the "true" quality of the matching result, one could have used this for optimization in the first place.

Chapter 6

Conclusion

In this work, we have presented a full graph-based SLAM system, which does not use a real robot motion model or control input – a variant of the problem which is particularly interesting for commercial 3D laser range finders used in surveying or in forensic applications. It could also be used as a black-box slam system on robots, or be extended with a motion model and used like a traditional SLAM system. While all our algorithms are essentially usable in 2D and 3D and in fact our implementation actually uses the same code for a large part of the algorithms, we eventually focused the implementation of our frontend on 3D data.

All necessary algorithms and steps needed to implement a similar system were explained thoroughly and without shying away from mathematical details. Following just recently published work by Hertzberg, Frese et al., our whole system is based on the idea of representing poses as manifolds in order to have a mathematically sound framework for running optimization algorithms on poses. In order to consequently use manifolds, adaptions of existing algorithms such as the computation of the ICP covariance of the Dijkstra projection were needed.

Our system comprises both the SLAM front- and backend. The latter is based on the maximum-likelihood method and is essentially an implementation very similar to SLoM. This "reimplementation" has been done mostly for learning purposes and in order to explore potential improvements.

We have examined the possibility of using AutoDiff techniques instead of numerical differentiation or manual symbolic differentiation. While we observed a slightly worse performance of AutoDiff in this case, we still believe there are numerous interesting applications for it, especially in even higher dimensional spaces where numerical differentiation becomes even more inefficient. The use of C++ templates, as well as the Eigen library even allowed using the same code for both differentiation techniques (even though it was admittedly not an easy thing to do).

In our frontend, we mostly extended existing 2D methods from Olson to 3D and made them respect the pose manifold.

Our implementation is based on C++, focussing on generic code in order to be able to support both the 2D and the 3D case, as well as numerical differentiation and Autodiff without unnecessary duplication of code.

6.1 Future Work

While working on this thesis, we have found numerous interesting opportunities for future research on this topic:

- A number of recent publications such as the work by Rusu et al. [RBMB08] for example, explore the use of point cloud features in order to perform the registration. This could be used instead of – or maybe even parallel to – our extension of the correlative scan matching algorithm to 3D in order to make the alignment of point clouds with no or a bad initial guess even more robust.

- In our work we used a very simple method of not introducing too many loop-closing edges: We artificially restricted the number of loop-closing edges that can be added for each node to one. This becomes a problem as the robot keeps coming back to the same locations. Thus it is not a very good and scalable thing to do and certainly calls for more sophisticated solutions.

- The number of nodes/scans in our graph keep growing and growing as more measurements are being done. This is unsuitable for achieving the goal of "life-long-learning". A robot should be able to drive around for hours in the same area, without struggling ever-growing memory, space or time requirements. Our idea to achieve that would be to take clusters of scans having a high coherency (where coherency is defined by registration accuracy), and to join them together to a new node, potentially discarding unnecessary or repeated measurements in the process.

- It seems like it should be possible to use appearance-based, topological SLAM methods such as [CN10] as a way to determine potentially overlapping poses in the frontend. This could be used instead of or in addition to the rather crude method used in this work, especially in very large outdoor datasets which are getting more and more common.

- As already mentioned in Section 4.3.1, Olson suggested the use of a spectral clustering approach in order to validate the correctness of loop-closing edges. We started the implementation of this method but were not able to finish it due to time constraints. Yet we feel like this is a crucial thing to implement for this method, since the backend is unable to deal with outliers and no

matter how good the frontend is – it can never guarantee not to make mistakes. Robustness and perfection are not achieved by not making mistakes, they are achieved through the ability to properly deal with mistakes and the ability to rethink and revise previously made decisions.

6.2 Acknowledgements

We would like to thank Bastian Steder and Giorgio Grisetti for publishing the datasets we used in our evaluation, as well as the V&R Vision & Robotics GmbH for providing their MappingCube for our measurements. We would also like to thank Giorgio Grisetti and Christoph Hertzberg for open-sourcing their implementations, which allowed a comparison of the results. Finally we would like to thank all contributors of the FLANN, Eigen and Boost libraries for their excellent work on these projects.

Bibliography

[BE] D. Borrmann and J. Elseberg. Global konsistente 3D Kartierung am Beispiel des Botanischen Gartens in Osnabrück. Bachelor's thesis, University of Osnabrück, 2006.

[BEL+08] D. Borrmann, J. Elseberg, K. Lingemann, A. Nüchter, and J. Hertzberg. Globally consistent 3D mapping with scan matching. *Robotics and Autonomous Systems*, 56:130–142, 2008.

[BNLT04] M. Bosse, P. M. Newman, J. Leonard, and S. Teller. SLAM in Large-scale Cyclic Environments using the Atlas Framework. *International Journal of Robotics Research*, 23(12):1113–1139, 2004.

[CJR+09] D. Cline, S. Jeschke, A. Razdan, K. White, and P. Wonka. Dart throwing on surfaces. *Computer Graphics Forum*, 28:1217–1226, 2009.

[CN10] M. Cummins and P. Newman. Appearance-only SLAM at large scale with FAB-MAP 2.0. *International Journal of Robotics Research*, 2010.

[Coo86] R. L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics (TOG)*, 5:51–72, 1986.

[CSSK02] D. Chetverikov, D. Svirko, D. Stepanov, and P. Krsek. The trimmed iterative closest point algorithm. *Object recognition supported by user interaction for service robots*, 3(c):545–548, 2002.

[Dav03] A. J. Davison. Real-time simultaneous localisation and mapping with a single camera. In *Proc. of the Int. Conf. on Computer Vision (ICCV)*, 2003.

[Fit01] A. W. Fitzgibbon. Robust registration of 2D and 3D point sets. In *Proc. of British Machine Vision Conference*, pages 662–670, 2001.

[FL06] U. Frese and Schröder L. Vorlesungsskript: Theorie der Sensorfusion. `http://www.informatik.uni-bremen.de/agebv/de/VeranstaltungTDS06`, Version: 2006. Last accessed: 20.09.2011.

[GGS+07] G. Grisetti, S. Grzonka, C. Stachniss, P. Pfaff, and W. Burgard. Efficient estimation of accurate maximum likelihood maps in 3D. In *Proc. of the IEEE Int. Conf. on Intelligent Robots & Systems (IROS)*, pages 3472–3478, 2007.

[GK99] J. S. Gutmann and K. Konolige. Incremental mapping of large cyclic environments. In *Proc. of the 1999 IEEE International Symposium on Computational Intelligence in Robotics and Automation. (CIRA)*, pages 318–325, 1999.

[GKS+10] G. Grisetti, R. Kümmerle, C. Stachniss, U. Frese, and C. Hertzberg. Hierarchical optimization on manifolds for online 2D and 3D mapping. In *Proc. of the IEEE Int. Conf. on Robotics & Automation (ICRA)*, pages 273–278, 2010.

[GSB07] G. Grisetti, C. Stachniss, and W. Burgard. Improved techniques for grid mapping with rao-blackwellized particle filters. *IEEE Transactions on Robotics*, 23:2007, 2007.

[GSB09] G. Grisetti, C. Stachniss, and W. Burgard. Non-linear constraint network optimization for efficient map learning. *IEEE Transactions on Intelligent Transportation Systems*, 10:428–439, 2009.

[GSGB07] G. Grisetti, C. Stachniss, S. Grzonka, and W. Burgard. A tree parameterization for efficiently computing maximum likelihood maps using gradient descent. In *Proc. of Robotics: Science and Systems (RSS)*, 2007.

[Her] C. Hertzberg. A framework for sparse, non-linear least squares problems on manifolds. Diploma thesis, University of Bremen, 2008.

[HWFS] C. Hertzberg, R. Wagner, U. Frese, and L. Schröder. Integrating generic sensor fusion algorithms with sound state representation through encapsulation of manifolds. *Information Fusion*. To appear. Preprint available on arXiv as eprint 1107.1119.

[KM07] G. Klein and D. Murray. Parallel tracking and mapping for small AR workspaces. In *Proc. Sixth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR)*, 2007.

[Lee02] J. M. Lee. *Introduction to Smooth Manifolds*. Springer, 2002.

[LM97] F. Lu and E. Milios. Globally consistent range scan alignment for environment mapping. *Autonomous Robots*, 4:333–349, 1997.

[Min05] J. Minguez. Metric-based scan matching algorithms for mobile robot displacement estimation. In *Proc. of the IEEE Int. Conf. on Robotics & Automation (ICRA)*, 2005.

[MTKW02] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit. FastSLAM: A factored solution to the simultaneous localization and mapping problem. In *Proc. of the AAAI National Conference on Artificial Intelligence*, 2002.

[Nüc06] A. Nüchter. *Semantische dreidimensionale Karten für autonome mobile Roboter*, volume 303 of *Dissertationen zur künstlichen Intelligenz*. Akad. Verl.-Ges. Aka, Berlin, 2006. Zugl.: Bonn, Univ., Diss., 2006.

[Ols08] E. Olson. *Robust and Efficient Robotic Mapping*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, June 2008.

[Ols09a] E. Olson. Real-time correlative scan matching. In *Proc. of the IEEE Int. Conf. on Robotics & Automation (ICRA)*, pages 4387–4393, 2009.

[Ols09b] E. Olson. Recognizing places using spectrally clustered local matches. *Robotics and Autonomous Systems*, 57:1157–1172, 2009.

[OLT06] E. Olson, J. Leonard, and S. Teller. Fast iterative alignment of pose graphs with poor estimates. In *Proc. of the IEEE Int. Conf. on Robotics & Automation (ICRA)*, pages 2262–2269, 2006.

[RBMB08] R. B. Rusu, N. Blodow, Z. C. Marton, and M. Beetz. Aligning Point Cloud Views using Persistent Feature Histograms. In *Proc. of the IEEE Int. Conf. on Intelligent Robots & Systems (IROS)*, 2008.

[SC86] R. C. Smith and P. Cheeseman. On the representation and estimation of spatial uncertainty. *International Journal of Robotics Research*, 5:56–68, 1986.

[TBF05] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[Wei05] J. Weingarten. EKF-based 3D slam for structured environment reconstruction. In *Proc. of the IEEE Int. Conf. on Intelligent Robots & Systems (IROS)*, pages 2–6, 2005.