Diplomarbeit
FB3 Mathematik und Informatik

A Framework for Sparse, Non-Linear Least Squares Problems on Manifolds

Ein Rahmen für dünnbesetzte, nichtlineare quadratische Ausgleichsrechnung auf Mannigfaltigkeiten

Gutachter: Prof. Dr.-Ing. Udo Frese
PD Dr. Lutz Schröder

vorgelegt durch: Christoph Hertzberg
Universität Bremen
November 2008

Acknowledgements

First of all I would like to thank Udo Frese for offering me this complex but also interesting and exiting topic, for the constructive discussions, the many useful suggestions and informations, as well as for providing me with several data sets needed to evaluate this thesis.

I like to thank Jörg Kurlbaum for implementing an early example using this framework, and his feedback which led to many interface improvements.

Furthermore I thank Lutz Schröder for volunteering to be the second reviewer for this thesis.

Finally I thank my family for supporting me during my studies and during my whole life.

Bremen, November 2008

Christoph Hertzberg

Contents

There is always a well-known solution to every human problem – neat, plausible, and wrong.

H. L. Mencken (1880-1956)

0 Introduction

This thesis provides a framework for sparse, non-linear least square problems on manifolds.

Least square problems appear in many areas of computer science today. A well-known example in robotics is the SLAM-problem [TBF05].

The motivation for this topic arose due to the fact, that today there exist many least squares algorithms, which all usually make some approximations and trade-offs, to keep their running time reasonable or to reduce their memory load. The goal of this thesis however is to find the "best possible" solution to any given least squares problem. This is useful when evaluating other algorithms to determine how good their approximation is.

In some areas in computer science like visual-SLAM there are already efforts to find best possible solutions as a final step, after performing an approximate algorithms. There the method is known as "bundle adjustment" and there already exist some specialized methods for that [HZ03].

The term manifold in combination with least squares problems often appears when trying to estimate orientations in 3D. It is known, that there is no way to represent an orientation with just three parameters, which doesn't get problems at some point.

0.1 Goals of the Thesis

The main goal of this thesis is to provide a framework which makes it easy to solve arbitrary least square problems. The framework will be a big support when solving problems which are already solved in an approximated way, and it offers a possibility to solve calibration problems.

This thesis *does not* provide an efficient online-SLAM algorithm, although it is useable as one theoretically. Furthermore doesn't it solve data association problems, which often arise simultaneously with least squares problem. And as this is only a framework it doesn't provide any application specific code, like sensor or motion models.

0.2 Guide to the Thesis

The first four chapters mainly describe the terms of the title in reverse direction.

The first chapter will give an introduction to manifolds, which are necessary to model non-euclidean variables. It requires some basic knowledge in topology.

The second chapter will describe least squares problems and will give some algorithms to solve them.

The third chapter will deal with sparse matrices (matrices which consist mainly of zero entries), and explain their importance.

The fourth chapter explains the general design of the framework and explains some design decissions.

Afterwards, the fifth chapter deals with the actual implementation, the sixth chapter presents some example usages of the framework and evaluates their results.

Finally the seventh chapter draws a conclusion over the thesis.

0.3 Acronyms

CRTP Curiously Recurring Template Pattern, see [Cop95]

DOF degrees of freedom, see definition 1.1.

LM Levenberg-Marquardt algorithm, see section 2.4.4.

LS Least Squares problem, see **??**.

RSS residual sum of squares

SLAM Simultaneos Localisation And Mapping

We must maintain that mathematical geometry is not a science of space insofar as we understand by space a visual structure that can be filled with objects – it is a pure theory of manifolds.

Hans Reichenbach

1 Manifolds

1.1 Motivation

Most standard estimation algorithms work properly only on euclidean vector spaces (i. e. spaces isomorphic to $\mathbb{R}^n$). But sometimes variables to be estimated do not form a euclidean vector space.

A simple example are the orientations in $\mathbb{R}^2$, more formally called $SO(2)$. They can be represented using a single angle α with $\alpha \in [-\pi, \pi)$. Adding small changes to α is no problem, as α can be "renormalized" if it gets out of $[-\pi, \pi)$ by simply adding or subtracting 2π.

But a problem arises when comparing two angles which are near $-\pi$ and π respectively. Their difference is almost 2π although they are actually very close. Some algorithms solve this manually by normalizing this difference to $[-\pi, \pi)$.

Another frequent example in practice is $SO(3)$, the group of rotations in $\mathbb{R}^3$. To handle $SO(3)$ there exist basically two suboptimal approaches.

One approach is to use a representation with three parameters (like e. g. Euler Angles). The problem arises that there exist no parameterization of $SO(3)$ with three parameters which has no singularities (cf. [FS]). Singularities cause problems called "gimbal lock". When the variable to be estimated approaches such a singularity the inverse of the parameterization becomes discontinuous, which means that some small changes of the variable require big changes in its representation. Some algorithms using this approach avoid this problem by changing the parameterization if the variable gets to close to a singularity.

Another approach is to "overparameterize" the variable (for $SO(3)$ e. g. by using quaternions, which have four values) and normalize the parameterization in some way [MYB+01, WI95]. But this usually causes significant problems:

As standard estimation algorithms expect their variables to be from euclidean vector spaces they are not aware of the inner constraints between the parameters. Therefore they are optimizing some degrees of freedom (DOF) which actually do not exist. This is obvisously a problem when the parameterization of the variable has more parameters than the dimension

of the measurement space, as there can't exist a unique result then (ignoring the constraints). But even otherwise the result has undefined DOF and any modification tends to denormalize the representation, which therefore has to be renormalized. Another related problem is that the parameterization sometimes behaves non-euclidean. That means changes of the same magnitude to different parameters yield to changes of the variable of quite different magnitude.

To solve these problems a combination of both approaches can be used. The variable is globally overparameterized, but local changes are represented with a minimal representation, which ideally behaves like a euclidean space as much as possible, for small values. A formalization of this idea leads to the theory of manifolds.

So informally a manifold can be seen as a space, which locally behaves like a euclidean space, but in general does not globally.

1.2 Definition

This section will give a more formal description of manifolds. The definition in this section is not the broadest possible, but it will suffice for this thesis. For a more general definition and deeper introduction see [Lee03].

As a general note, the actual definition of *smooth* in this whole chapter will depend on its context, but will generally mean C^k-differentiable, with k being constant within each definition or theorem.

Definition 1.1 (Manifold)**.** Let $M \subset \mathbb{R}^n$ be connected, and $\mathcal{F} = (U_\alpha, \varphi_\alpha)_{\alpha \in A}$ be a family such that for a given $d \in \mathbb{N}$:

1. $\forall_{\alpha \in A} : U_\alpha$ is an open subset of M (open in M means there is an open set $\tilde{U} \subset \mathbb{R}^n$, such that $U = M \cap \tilde{U}$), and $M = \bigcup_{\alpha \in A} U_\alpha$.
2. $\varphi_\alpha : U_\alpha \to V_\alpha$ is a homeomorphism to an open subset $V_\alpha \subset \mathbb{R}^d$.
3. If $U_\alpha \cap U_\beta \neq \emptyset$, the *transition map*

$$\varphi_\alpha \circ \varphi_\beta^{-1} : \varphi_\beta(U_\alpha \cap U_\beta) \to \varphi_\alpha(U_\alpha \cap U_\beta) \tag{1.1}$$

   is a C^k diffeomorphism.

Then $(M, \mathcal{F})$ (or just M) is called a (C^k-) *manifold*. The tuples $(U_\alpha, \varphi_\alpha)$ are called *chart*s, and the family $\mathcal{F}$ is called *atlas* of M. The number d is called the *dimension*, or *degrees of freedom* of M.

The first condition ensures that M is completely covered by open sets. The second condition makes sure that the charts are continous. Intuitively that means that the chart has no holes or gaps. The last condition ensures that the charts are smoothly compatible. This is needed when defining smooth functions on M.

Definition 1.2 (Smooth Function)**.** Let $(M, (U_\alpha, \varphi_\alpha))$, $(N, (W_\beta, \psi_\beta))$ be C^k-manifolds. A function $f : M \to N$ is called C^k-differentiable in $x \in M$ iff for $x \in U_\alpha$ and $f(x) \in W_\beta$:

$$\begin{aligned} f_\alpha^\beta : \varphi_\alpha(U_\alpha) &\to \psi_\beta(W_\beta) \\ x &\mapsto \psi_\beta(f(\varphi_\alpha^{-1}(x))) \end{aligned} \tag{1.2}$$

is C^k-differentiable in $\varphi_\alpha(x)$. [Note that $\varphi_\alpha(U_\alpha) \subset \mathbb{R}^d$, so the usual definition for smoothness is applicable.] f is called C^k-differentiable iff it is C^k-differentiable in every point $x \in M$.

Due to the third condition of definition 1.1 the smoothness of f does not depend on the particular choices of φ_α and ψ_β:

Proof. Let $x \in U_\alpha \cap U_{\alpha'}$, $f(x) \in W_\beta \cap W_{\beta'}$ and f_α^β be smooth in $\varphi_\alpha(x)$, then:

$$f_{\alpha'}^{\beta'} = (\psi_{\beta'} \circ \psi_\beta^{-1}) \circ f_\alpha^\beta \circ (\varphi_\alpha \circ \varphi_{\alpha'}^{-1}) \tag{1.3}$$

is smooth in $\varphi_{\alpha'}(x)$, because $(\varphi_\alpha \circ \varphi_{\alpha'}^{-1})(\varphi_{\alpha'}(x)) = \varphi_\alpha(x)$ and the composition of smooth functions is smooth. □

1.3 Encapsulation

When using manifolds in standard algorithms (like Kalman filters), it would be very impractical if the main algorithm had to care about charts and applying homeomorphisms. To avoid this the entire handling of manifolds can be encapsulated, with an approach proposed in [FS].

The general idea is that an estimation algorithm handles the manifold as a "black box". The algorithm has only two possibilities to access the manifold.

The first is to add small changes to the manifold:

$$\boxplus : M \times \mathbb{R}^m \to M \tag{1.4}$$

where $\delta \mapsto x \boxplus \delta$ is a homeomorphism from a neighborhood of $0 \in \mathbb{R}^m$ to a neighborhood of $x \in M$. The other possibility is to find the difference between two elements in M:

$$\boxminus : M \times M \to \mathbb{R}^m, \tag{1.5}$$

where $y \mapsto y \boxminus x$ is the inverse of $\boxplus$, or formally for all $x, y \in M$:

$$x \boxplus (y \boxminus x) = y \tag{1.6}$$

As $\delta \mapsto x \boxplus \delta$ is supposed to be a homeomorphism, the dimension of M has to be m. Furthermore as the domain of the homeomorphism is a neighborhood of 0, it follows that $x \boxplus 0 = x$.

This approach has some quite remarkable advantages. Most standard algorithms working on $\mathbb{R}^m$ now work essentially the same way on M, after replacing $+$ with $\boxplus$ when adding small updates to the state, and $-$ with $\boxminus$ when calculating the difference between two states.

In particular they do not have to deal with singularities or denormalized overparameterizations. And, after adapting an algorithm, no further adaptions are needed, if other manifolds are processed (or the internal representation of the manifold changes), as long as these manifolds implement both these operators.

Note that $\boxplus$ might often be defined only for "small" δ and that $\boxminus$ might behave uncontinous or become undefined if minuend and subtrahend are "far apart". Within this thesis it is always assumed that this doesn't happen. In practice adding too big differences should be avoided anyway, especially if non-linear functions are linearized by their derivation.

1.3.1 Generic Definition

One way to define $x \boxplus \delta$ is to choose a chart including x, within which a vector x can be added, and project the result back to M. $y \boxminus x$ essentially does the inverse of this operation, i. e. it finds the value y in a chart including x and subtracts their coordinates. To make these functions unique, the choice of the chart has to be determined by x.

More formally with φ_x describing the chosen chart around x, one can define:

$$x \boxplus \delta := \varphi_x^{-1}(\varphi_x(x) + \delta) \tag{1.7}$$

$$y \boxminus x := \varphi_x(y) - \varphi_x(x) \tag{1.8}$$

One can easily verify, that (1.6) holds for this definition:

Proof.

$$x \boxplus (y \boxminus x) = x \boxplus \big(\varphi_x(y) - \varphi_x(x)\big) \tag{1.9}$$

$$= \varphi_x^{-1}\big(\varphi_x(y) + \varphi_x(y) - \varphi_x(x)\big) \tag{1.10}$$

$$= \varphi_x^{-1}\big(\varphi_x(y)\big) = y \qquad \square$$

One can also prove, that using aboves definitions $\boxplus$ and $\boxminus$ propagate smoothness:

Proposition 1.1. *Let M, N be smooth manifolds, and $f : M \to N$ be smooth. Then*

$$g(\delta) := f(x \boxplus \delta) \boxminus f(x) \tag{1.11}$$

is smooth in δ.

Proof.

$$g(\delta) = f(x \boxplus \delta) \boxminus f(x) = \psi_{f(x)}(f(\varphi_x^{-1}(\varphi_x(x) + \delta))) - \psi_{f(x)}(f(x)), \tag{1.12}$$

and $\psi_{f(x)} \circ f \circ \varphi_x^{-1}$ is smooth by definition, whereas $\varphi_x(x)$ and $\psi_{f(x)}(f(x))$ are constant with regard to δ. $\square$

1.3.2 Encapsulation for Lie-Groups

M is called *Lie-Group*, if it is a manifold and has a group structure.

If M is a Lie-Group one can use a simpler definition, needing only a single chart around the unit element $\mathrm{id} \in M$ with $\varphi(\mathrm{id}) = 0$. Using the group's $\cdot$ and $^{-1}$ operators, one can then define:

$$x \boxplus \delta := x \cdot \varphi^{-1}(\delta) \tag{1.13}$$

$$y \boxminus x := \varphi(x^{-1} \cdot y) \tag{1.14}$$

or alternatively (but not necessarily equivalent):

$$x \boxplus \delta := \varphi^{-1}(\delta) \cdot x \tag{1.15}$$

$$y \boxminus x := \varphi(y \cdot x^{-1}) \tag{1.16}$$

Note that this definition is still a special case of the generic definition, as it is possible to recalculate φ_x from $\boxminus$, which satisfies (1.7) and (1.8):

$$\varphi_x(y) := y \boxminus x, \tag{1.17}$$

where $\varphi_{\mathrm{id}} \neq \varphi$ in general.

1.4 Cartesian Product

One can show, that with M_1, M_2 being manifolds with dimensions d_1 and d_2, their cartesian product $M := M_1 \times M_2$ is a manifold as well, having dimension $d := d_1 + d_2$ [Lee03, p. 8]. Intuitively this means just writing elements x of M as a tupel (x_1, x_2) with $x_i \in M_i$. It is now possible to extend the definition of $\boxplus$ and $\boxminus$ to M in the obvious way:

$$\begin{aligned} \boxplus : \quad & M \times \mathbb{R}^d \quad \to M \\ & \left((x_1, x_2), \begin{bmatrix} \delta_1 \\ \delta_2 \end{bmatrix}\right) \mapsto (x_1 \boxplus \delta_1, x_2 \boxplus \delta_2) \end{aligned} \tag{1.18}$$

$$\begin{aligned} \boxminus : \quad & M \times M \quad \to \mathbb{R}^d \\ & ((x_1, x_2), (y_1, y_2)) \mapsto \begin{bmatrix} x_1 \boxminus y_1 \\ x_2 \boxminus y_2 \end{bmatrix}, \end{aligned} \tag{1.19}$$

with $x \boxplus (y \boxminus x) = y$ and $x \boxplus 0 = x$ still holding, as they hold by component. This can of course be extended to any finite number of manifolds.

Analog to that a set of functions can be combined to a single function. For $f_1 : N_1 \times N_2 \to M_1$ and $f_2 : N_2 \times N_3 \to M_2$ one can define a function $f := f_1 \times f_2$ from $N := N_1 \times N_2 \times N_3$ to $M := M_1 \times M_2$ as follows:

$$\begin{aligned} f : \quad & N \quad \to M \\ & (x_1, x_2, x_3) \mapsto (f_1(x_1, x_2), f_2(x_2, x_3)), \end{aligned} \tag{1.20}$$

which of course can also be extended to any finite number of functions. Note that the smoothness of f is determined by the least smoothest function f_i, as different subfunctions do not affect the smoothness of others.

1.5 Examples

This section will give some examples for manifolds and prove their smoothness. The actual implementation will be handled in section 5.2.

The most trivial example would be $\mathbb{R}^d$ itself. As one can easily verify, replacing $\boxplus$, $\boxminus$ by their vector space equivalents $+$, $-$ complies with (1.6).

This shows that restricting a variable to be from a manifold doesn't let it lose its generality. In the following there will be given some other useful examples, which justify the effort of introducing manifolds.

1.5.1 $SO(2)$

A still simple example is the special orthogonal group of degree 2, which is informally the group of rotations in the plane (or angles with periodicity). Formally it can be described as the following set of matrices:

$$SO(2) := \left\{ Q \in \mathbb{R}^{2\times 2};\ Q^\top Q = \mathrm{I}_2 = Q Q^\top \wedge \det Q = 1 \right\} \tag{1.21}$$

which is a subgroup of the general linear group (the invertable matrices) using the standard matrix multiplication. One can verify, that $SO(2)$ has one DOF[1] and in fact it can be parameterized with one variable:

$$SO(2) = \left\{ \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix};\ \alpha \in \mathbb{R} \right\}. \tag{1.22}$$

So a simple parameterization is to just store a single angle α and define $\alpha \boxplus \delta := \alpha + \delta$. When defining $\alpha \boxminus \beta$ care has to be taken that $\alpha + 2\pi k$, for integer k all represent the same rotation. So simply defining $\alpha \boxminus \beta \stackrel{?}{:=} \alpha - \beta$ won't work. A working solution is to normalize this difference to the intervall $(-\pi, \pi)$, e. g. with $\varphi(\delta) := \delta - 2\pi\lfloor \frac{\delta+\pi}{2\pi} \rfloor$, defining:

$$\alpha \boxminus \beta := \varphi(\alpha - \beta). \tag{1.23}$$

Note that this directly corresponds to the encapsulation of Lie groups proposed in section 1.3.2 with $+$ and $-$ as group operations.

This is a procedure which is often done "manually" in algorithms not using a manifold approach.

1.5.2 $SO(3)$

A more important application for manifolds is the representation of rotations in 3D-space, formally the special orthogonal group of degree 3. Its matrix representation is similar to that of $SO(2)$:

$$SO(3) := \left\{ Q \in \mathbb{R}^{3\times 3};\ Q^\top Q = \mathrm{I}_3 = QQ^\top \wedge \det Q = 1 \right\}, \tag{1.24}$$

but in contrast it does not have a singularity free covering with only three parameters [FS].

However a covering with with four parameters is possible, namely using quaternions. As $SO(3)$ is a Lie group, it is only necessary to provide a homeomorphism from a neighborhood of the identity $\mathrm{id} \in SO(3)$ to $\mathbb{R}^3$ in order to define $\boxplus$ and $\boxminus$. A possible homeomorphism is given by the *quaternion logarithm* $\varphi(q) := \log(q)$. For $q = (w, u)$, with $w \in \mathbb{R}$, $u \in \mathbb{R}^3$ and $w^2 + \|u\|^2 = 1$ it is defined as:

$$\begin{aligned} \log : SO(3) &\to \mathbb{R}^3 \\ q &\mapsto \begin{cases} \frac{u}{\|u\|} \arccos w & u \neq 0 \\ 0 & u = 0, \end{cases} \end{aligned} \tag{1.25}$$

the inverse is given by the *exponential map* $\varphi^{-1}(\delta) = \exp\delta$, with

$$\begin{aligned} \exp : \mathbb{R}^3 &\to SO(3) \\ \delta &\mapsto (\cos\|\delta\|, \delta \cdot \operatorname{sinc}\|\delta\|). \end{aligned} \tag{1.26}$$

A derivation for these formulas can be found in [Ude99]. A more general introduction to quaternions can be found in [Vic01].

[1] Viewed as matrix it has 4 variables and the equation $Q^\top Q = \mathrm{I}_2$ gives 3 constraints.

1.5.3 S^2

S^n is the unit sphere in $\mathbb{R}^{n+1}$, that is

$$S^n := \left\{ x \in \mathbb{R}^{n+1};\ \|x\|^2 = 1 \right\}. \tag{1.27}$$

As this parameterisation has $n+1$ components and 1 constraint, it has n DOF. For $n = 0$ $S^0 = -1, 1$ is a two point set, which is not connected and therefore not a smooth manifold. For $n = 1$ it is essentially the same as $SO(2)$, using the parameterisation $S^1 = \{[\begin{smallmatrix}\cos\alpha \\ \sin\alpha\end{smallmatrix}];\ \alpha \in \mathbb{R}\}$, which can be mapped to $SO(2)$ isomorphically.

Furthermore one can see that S^3 is a cover of $SO(3)$ (the unit quaternions are isomorphic to S^3 and a cover of $SO(3)$ as seen previously).

The 2-sphere S^2 (the surface of the globe or the directions in $\mathbb{R}^3$) is another manifold and one example which does not have a group structure.[2] It can be parameterized by 2 angles *latitude* and *longitude* (like the earth's surface) but this leads to singularities at the poles as they have arbitrary longitude.

A common way to parameterize S^2 is to store it using its 3 coordinates in $\mathbb{R}^3$ and use *stereographic projection* [Wikb] to map it to $\mathbb{R}^2$. For $U := S^2 \setminus [0,0,1]^\top$ the following definition for φ is a homeomorphism:

$$\varphi : U \to \mathbb{R}^2 \tag{1.28}$$

$$\varphi([x,y,z]^\top) = \frac{1}{1-z}[\begin{smallmatrix}x \\ y\end{smallmatrix}] \tag{1.29}$$

with the inverse

$$\varphi^{-1}([a,b]^\top) = \frac{1}{1+a^2+b^2}\begin{bmatrix}2a \\ 2b \\ -1+a^2+b^2\end{bmatrix} \tag{1.30}$$

To get a complete cover of S^2 one can use the sets $U_{\pm i} := S^2 \setminus \pm e_i$, $1 \le i \le 3$ and e_i being the ith unit vector, thus $U = U_3$ and define $\varphi_{\pm i}$ analogously to φ.

When defining $\boxplus$ and $\boxminus$ one can use the generic definitions (1.7) and (1.8) with $\varphi_{[x,y,z]}$ chosen such that $[x,y,z]^\top$ has the biggest distance to $\pm e_i$ (with an arbitrary but defined strategy for ties). Due to the fact that $\varphi : \mathbb{R}^3 \setminus e_3 \to \mathbb{R}^2$ is smooth and $\varphi^{-1} : \mathbb{R}^2 \to \mathbb{R}^3$ as well, it is easy to see that any transistion map $\varphi_{\pm i} \circ \varphi_{\pm j}^{-1}$ is smooth as well thus holding condition 3 of definition 1.1.

This is however an example where $x \boxplus \delta$ is not smooth in x (and not even continuous), but it is in δ, as required.

[2] This is a consequence of the so-called "Hairy Ball Theorem" [Wika].

It is better to be approximately right than exactly wrong.

Old adage

2 Least Squares Problems

The problem to be solved in this chapter is to determine the distribution of a random variable $X \in N$, given a noisy measurement $Z \in M$ and a *measurement function* f describing the relation between them.

Before giving different solutions to that problem, it is discussed how exactly noise can be modeled into this relation. Afterwards a method to normalize Least Squares problems (LSs) on manifolds is introduced, which is then used to apply standard least squares solvers to the problem.

2.1 Modeling Least Squares Problems

For $M = \mathbb{R}^m$ a common way to describe a measurement with noise $\varepsilon \sim \mathcal{N}(0, \Sigma)$ and known covariance Σ is the following equation:

$$Z = f(X) + \varepsilon, \tag{2.1}$$

which is equivalent to:

$$Z + (-\varepsilon) = f(X). \tag{2.2}$$

Note that $\varepsilon \sim \mathcal{N}(0, \Sigma)$ is often just an approximation, which usually doesn't hold exactly especially when f is non-linear.

For M being an arbitrary manifold one can write analogously:

$$Z = f(X) \boxplus \varepsilon_1, \tag{2.3}$$

or:

$$Z \boxplus \varepsilon_2 = f(X). \tag{2.4}$$

Since $Z \boxminus f(X) \neq -(f(X) \boxminus Z)$ in general, these equations are no longer equivalent to each other as (2.1) and (2.2).

On the first view (2.3) makes more sense: After applying the measurement function f on X noise ε_1 is added and the "noisy" result Z is measured. But this has a big drawback: As the actual value of ε_1 can depend discontinuously on the unknown value of $f(X)$ it is hard to determine $\Sigma = \mathrm{Cov}(\varepsilon_1)$ in general.

When using (2.4) instead, the actual value of ε_2 depends smoothly on $f(X)$ and discontinuously only on the known value of Z. In fact with known $Z = z$ one can define:

$$\tilde{f}(X) := f(X) \boxminus z = \varepsilon_2 \sim \mathcal{N}(0, \Sigma), \tag{2.5}$$

so for now it is assumed that

$$f(X) = Z \boxplus \varepsilon, \tag{2.6}$$

with $\varepsilon \sim \mathcal{N}(0, \Sigma)$, known Σ and known $Z = z$.

2.2 Normalization

In order to simplify algorithms this section develops a method to normalize a Gaussian distribution, such that the problem will simplify to

$$f(X) \sim \mathcal{N}(0, \mathrm{I}_m). \tag{2.7}$$

First of all lemma 2.1 shows that Gaussian distributions are closed under linear transformations. It is usually proven in stochastics.

Lemma 2.1. *Let $\mu \in \mathbb{R}^m$, $\Sigma \in \mathbb{R}^{m \times m}$. Furthermore let $C \in \mathbb{R}^{n \times m}$ with C having full rank and $b \in \mathbb{R}^n$, then for $X \sim \mathcal{N}(\mu, \Sigma)$:*

$$Z = CX + b \sim \mathcal{N}\left(C\mu + b,\, C\Sigma C^\top\right) \tag{2.8}$$

Using this lemma, one can always normalize a normally distributed variable:

Proposition 2.1. *If $\mathbb{R}^m \ni Z \sim \mathcal{N}(\mu, \Sigma)$ and $LL^\top$ is the Cholesky decomposition of Σ, then*

$$Y := L^{-1}(Z - \mu) \sim \mathcal{N}(0, \mathrm{I}_m). \tag{2.9}$$

Proof.

$$L^{-1}(Z - \mu) \sim \mathcal{N}\left(L^{-1}(\mu - \mu),\, L^{-1}\Sigma L^{-\top}\right) = \mathcal{N}\left(0,\, L^{-1}(LL^\top)L^{-\top}\right) \tag{2.10}$$

$$= \mathcal{N}(0, \mathrm{I}_m) \qquad \square$$

This can be used to normalize $f(X)$ as in (2.6), by setting

$$\tilde{f}(X) := L^{-1}(f(X) \boxminus z) \sim L^{-1}(\mathcal{N}(0, \Sigma)) \tag{2.11}$$

$$\sim \mathcal{N}(0, \mathrm{I}_m), \tag{2.12}$$

using (2.5) and proposition 2.1.

For the rest of this chapter it is assumed without loss of generality, that $f(X)$ is normalized as above.

Also note that if $\tilde{f}$ is the cartesian product of smaller functions as shown in section 1.4, the normalization can be done function-wise.

2.3 Linear Least Squares Problems

The next goal is to determine X for different kinds of f. For linear $f : \mathbb{R}^n \to \mathbb{R}^m$, the following theorem gives an exact solution. It is assumed that $A \in \mathbb{R}^{m \times n}$ has maximal rank (therefore $A^\top A$ is invertable).

Theorem 2.1. *Let $f(X) = AX + b = Z \sim \mathcal{N}(0, \mathrm{I})$, then*

$$X \sim \mathcal{N}\left((A^\top A)^{-1} A^\top (-b),\, (A^\top A)^{-1}\right) \tag{2.13}$$

Proof. Using lemma 2.1 with $h(Z) = (A^\top A)^{-1} A^\top (Z - b)$ it follows:

$$\begin{aligned} X &= (A^\top A)^{-1} A^\top (AX + b - b) = h(AX + b) && (2.14)\\ &= h(Z) \sim \mathcal{N}\left((A^\top A)^{-1} A^\top (-b),\, (A^\top A)^{-1}\right) && \square \end{aligned}$$

To justify the term "least squares" the following theorem shows, that the expectation value from (2.13) also has the smallest squared residual:

Theorem 2.2. *Let $f(X) = AX + b$, then*

$$(A^\top A)^{-1} A^\top (-b) = \operatorname*{argmin}_x \|f(x)\|^2 . \tag{2.15}$$

Proof. Let $F(x) := \|f(x)\|^2$. The minimum is determined by finding the root of $\frac{\partial}{\partial x} F(x)$:

$$\begin{aligned} 0 \overset{!}{=} \frac{\partial}{\partial x} F(x) &= 2 f(x)^\top f'(x) = 2(x^\top A^\top A + b^\top A) = 2(A^\top A x + A^\top b)^\top && (2.16)\\ \Leftrightarrow \quad A^\top A x &= -A^\top b && (2.17)\\ \Leftrightarrow \quad x &= -(A^\top A)^{-1} A^\top b && (2.18) \end{aligned}$$

Since this is the only root and $F(x) \geq 0$, this has to be a minimum. $\square$

When computing x it is usually very inefficient to actually compute $(A^\top A)^{-1}$, unless one really needs the complete covariance matrix. Instead x can be calculated by solving the linear system (2.17). Since $A^\top A$ is symmetric and positive-definite, (2.17) can be solved using the Cholesky decomposition of $A^\top A$.

Another, sometimes faster, method to solve (2.17) is to use the QR decomposition of A:

Theorem 2.3. *If $A = Q\tilde{R}$ is the QR decomposition of A with $\tilde{R} = \begin{bmatrix} R & 0 \end{bmatrix}^\top$, solving (2.17) is equivalent to solving:*

$$Rx = -\begin{bmatrix} \mathrm{I}_n & 0 \end{bmatrix} Q^\top b. \tag{2.19}$$

(Note that multiplying by $\begin{bmatrix} \mathrm{I}_n & 0 \end{bmatrix}$ essentially means only to consider the first n entries $Q^\top b$.)

Proof. Substituting $A = Q\tilde{R}$ in (2.18) leads to:

$$\begin{aligned} x &= -(\tilde{R}^\top Q^\top Q \tilde{R})^{-1} \tilde{R}^\top Q^\top b = -\left(R^\top \begin{bmatrix} \mathrm{I}_n & 0 \end{bmatrix} \begin{bmatrix} \mathrm{I}_n \\ 0 \end{bmatrix} R\right)^{-1} R^\top \begin{bmatrix} \mathrm{I}_n & 0 \end{bmatrix} Q^\top b && (2.20)\\ &= -(R^\top R)^{-1} R^\top \begin{bmatrix} \mathrm{I}_n & 0 \end{bmatrix} Q^\top b = -R^{-1} R^{-\top} R^\top \begin{bmatrix} \mathrm{I}_n & 0 \end{bmatrix} Q^\top b = -R^{-1} \begin{bmatrix} \mathrm{I}_n & 0 \end{bmatrix} Q^\top b && (2.21)\\ \Leftrightarrow Rx &= -\begin{bmatrix} \mathrm{I}_n & 0 \end{bmatrix} Q^\top b && \square \end{aligned}$$

2.4 Non-Linear Least Squares Problems

In this section the goal is to find a least squares solution for (2.7), with f not necessarily linear.

In most cases it is not possible to find a closed form solution to a non-linear LS, especially without having exact information about f. Therefore all algorithms presented here are iterative algorithms, which means they start at an estimate x_0 and find for every x_k a (hopefully) better approximation x_{k+1} until the sequence x_k converges.

As a consequence, all methods described below only converge to a local optimum. That means, to find the global optimum, the start estimate x_0 has to be close enough to this optimum.

The method above can easily be extended to functions defined on manifolds. The idea is that for a fixed $x \in N$ the function $g(\delta) := f(x \boxplus \delta)$ behaves locally in 0 like f does in x. In particular $\|f(x)\|^2$ has a minimum in x if and only if $\|g(\delta)\|^2$ has a minimum in 0. Therefore finding a local optimum of g, $\delta = \operatorname{argmin}_\delta \|g(\delta)\|^2$ implies $x \boxplus \delta = \operatorname{argmin}_\xi \|f(\xi)\|^2$.

As a matter of fact, g is now a usual function $g : \mathbb{R}^n \to \mathbb{R}^m$, so any standard optimization algorithm can be applied to g now. This completely hides the manifolds from the actual optimization method.

There exist algorithms for solving non-linear LSs, which do not require f to be differentiable. One such algorithm is the Nelder-Mead method [NM65], but these algorithms usually don't perform well on high dimensional problems [PTVF92, p. 408]. Therefore they will not be considered in this thesis and throughout this section it is assumed that f near its optimum is (at least) continuously differentiable with:

$$f(x \boxplus \delta) = f(x) + J_x \delta + \mathcal{O}(\|\delta\|^2), \tag{2.22}$$

with J_x being the Jacobian of f at x or more explicitly the Jacobian of $f(x \boxplus \delta)$ at $\delta = 0$.

In algorithm 1 the basic concept of all following algorithms is shown. In line 4 the function FindIncrement is called, whose implementation depends on the actually used algorithm.

Additionally a stepsize control parameter λ can be used, which increases or decreases depending on the outcome of the last increment (see algorithm 2). The advantage of this approach is that the residual is guaranteed to decrease eventually. The disadvantage however is that convergence might become very slow and a lot of unnecessary loop cycles might occour just to increase enough.

The decreasement and increasement in line 8 and 10 could be a division or multiplication by a constant factor or may depend e. g. on the last gain $\|f(x)\|^2 - \|f(\tilde{x})\|^2$.

Algorithm 1 Find local optimum of $\|f\|^2$ for a non-linear function f

Require: f is continuously differentiable
Require: Start value x_0 is close to optimum of f
1: $x \leftarrow x_0$
2: **while** not converged **do**
3: $\quad J \leftarrow \left.\dfrac{\partial f(x \boxplus \delta)}{\partial \delta}\right|_{\delta=0}$ *// Calculate Jacobian*
4: $\quad \delta \leftarrow$ FindIncrement$(f(x), J)$ *// Note that $f(x)$ and J are plain vectors or matrices*
5: $\quad x \leftarrow x \boxplus \delta$
6: **end while**

Algorithm 2 Find local optimum using a stepsize control parameter

Require: Same as for algorithm 1.
Require: Reasonable initialization of $\lambda > 0$

$x \leftarrow x_0$
while not converged **do**
  $J \leftarrow \left.\frac{\partial f(x \boxplus \delta)}{\partial \delta}\right|_{\delta=0}$ // *Calculate Jacobian*
  $\delta \leftarrow \text{FindIncrement}(f(x), J, \lambda)$ // *Note that $f(x)$ and J are plain vectors or matrices*
  $\tilde{x} \leftarrow x \boxplus \delta$
  if $\|f(\tilde{x})\|^2 < \|f(x)\|^2$ **then**
    $x \leftarrow \tilde{x}$
    decrease λ
  else
    increase λ
  end if
end while

The rest of this chapter gives different options for FindIncrement. Since f behaves like an ordinary function for FindIncrement, for the rest of this chapter $f : \mathbb{R}^n \to \mathbb{R}^m$ is assumed.

2.4.1 Gradient Descent

One of the simplest methods for finding a minimum of a function $F : \mathbb{R}^n \to \mathbb{R}$ is the *Gradient Descent* method (sometimes called *Steepest Descent*). It can also be used to optimize function $f : \mathbb{R}^n \to \mathbb{R}^m$ by setting $F(x) := \|f(x)\|^2$. The basic idea is that the gradient of F points towards bigger values of F and its negative points to smaller values. By using a parameter $\alpha > 0$ one sets $\delta_k := -\alpha_k F'(x_k)$. For choosing α_k one can either use a linear search at every step to find an optimal value or use a stepsize control, which increases or decreases α_k at every step depending on how good the last estimation was (like algorithm 2 with $\alpha_k = \lambda^{-1}$ at each step).

If F stems from a multidimensional function, i. e. $F(x) := \|f(x)\|^2$ one can find an optimal α_k assuming f behaves linear near x_k

$$f(x+\delta) = f(x) + J_x\delta \tag{2.23}$$

$$\Rightarrow \quad F(x - \alpha F'(x)) = F(x) - 4\alpha \left\| J_x^\top f(x) \right\|^2 + \alpha^2 \left\| J_x J_x^\top f(x) \right\|^2 \tag{2.24}$$

$$\Rightarrow \quad \operatorname*{argmin}_\alpha F(x - \alpha F'(x)) = \frac{4 \left\| J_x^\top f(x) \right\|^2}{2 \left\| J_x J_x^\top f(x) \right\|^2} \tag{2.25}$$

The Gradient Descent method is an indirect method, as it does not involve solving a linear system. It does perform poorly in practice and usually a *Conjugated Gradient* method should be prefered [PTVF92, p. 420].

This will not be examined in this thesis though, because it is not that easy to apply on manifolds, and for problems handled in this thesis direct methods (which will be covered in the following sections) work satisfactorily.

2.4.2 Newton's Method

Newton's method usually is a method to find roots of a function $g : \mathbb{R}^n \to \mathbb{R}^n$, but can also be used to find local extrema of a function F like given above, provided that F is twice-differentiable. The general idea is to find the roots of F' using Newton's method.

Assuming $F(x)$ has the gradient $F'(x)$ and Hessian $F''(x)$ and higher order derivations are neglectible one can write:

$$F(x+\delta) = F(x) + F'(x)\delta + \frac{1}{2}\delta^\top F''(x)\delta \tag{2.26}$$

$$\Rightarrow \quad F'(x+\delta) = F'(x) + \delta^\top F''(x) \stackrel{!}{=} 0 \tag{2.27}$$

$$\Leftrightarrow \quad F''(x)\delta = -F'(x)^\top, \tag{2.28}$$

which can be solved for δ.

For $F = \|f\|^2$ the Jacobian of F calculates as $F'(x)^\top = 2J_x^\top f(x)$. The Hessian calculates as follows:

$$(F''(x))_{ij} = \frac{\partial}{\partial x_i}(F'(x))_j = 2\sum_{k=1}^{m} \frac{\partial}{\partial x_i}\left(f_k(x)\frac{\partial}{\partial x_j}f_k(x)\right) \tag{2.29}$$

$$= 2\sum_{k=1}^{m}\left(\frac{\partial}{\partial x_i}f_k(x)\frac{\partial}{\partial x_j}f_k(x) + f_k(x)\frac{\partial}{\partial x_i}\frac{\partial}{\partial x_j}f_k(x)\right), \tag{2.30}$$

therefore:

$$F''(x) = 2\left(J_x^\top J_x + \sum_{k=1}^{m} f_k(x) f_k''(x)\right) \tag{2.31}$$

In most practical cases Newton's method converges locally quadratic. A downside of Newton's method is that it requires the second order derivation of F (or f), which sometimes doesn't exist or is difficult to obtain, especially if it has to be calculated numerically.

2.4.3 Gauss-Newton Method

The Gauss-Newton method can be seen as a simplification of Newton's method, neglecting second order derivations of f. Equivalently it can be derived by iteratively applying linear least squares methods to f linearized at x_k (cf. theorem 2.1). With J_x being the Jacobian of f both lead to the same update formula:

$$(J_x^\top J_x)\delta = -J_x^\top f(x), \tag{2.32}$$

which has to be solved for δ. Unlike Newton's method the Gauss-Newton algorithm is not guaranteed to converge. Especially if f is highly non-linear (i. e. has big higher order derivations).

2.4.4 Levenberg-Marquardt Algorithm

A quite popular algorithm for LSs is the Levenberg-Marquardt algorithm (LM) [PTVF92, p. 683]. It can be seen as a combination of the Gauss-Newton algorithm and gradient descent. It uses a control parameter as in algorithm 2 and its update formula is:

$$(J_x^\top J_x + \lambda D)\delta = -J_x^\top f(x), \tag{2.33}$$

with D being a positive definite diagonal matrix. Common choices for D are $D = \mathrm{I}$ (the original proposal of Levenberg [Lev44]) or $D_{ii} = (J_x^\top J_x)_{ii}$ as proposed by Marquardt [Mar63].

The big advantage of LM is, that due to its control parameter it can't diverge and it almost always converges eventually, as for increased λ it behaves like gradient descent. Furthermore locally λ usually gets small, thus behaving much like the Gauss-Newton method, with nearly quadratic convergence.

A downside however is that it sometimes converges really slow especially with a highly non-linear f, as it then has to avoid increasing the residual by choosing a very big λ, which makes the update step δ very small.

3 Sparse Matrices

In many LSs that arise in practice one has a (probably big) set of random variables, each having a low dimension. Furthermore there is a (most likely even bigger) set of independent measurements, from which each depends only on a few variables.

One common example would be the Simultaneos Localisation And Mapping (SLAM) problem, where one has a set of poses and landmarks (the random variables) and a set of measurements (usually odometry and some kind of relation between the poses and the landmarks).

This chapter will deal with how this "sparseness" can be utilized and why it is crucial to do so. There will be no deep insight in how the actual matrix algorithms work, as for the implementation of this thesis CSparse [Dav] is used, which is described thoroughly in [Dav06].

3.1 Motivation

The straight-forward solution to solve problems as above is to stack all variables together and also all functions, as shown in section 1.4. Then there is one big function, which depends on one big random variable. For real life problems the dimension of the random variable can easily reach $n = 20000$, and for the function say $m = 50000$.

Storing the Jacobian J of f alone would take $8nm = 8 \cdot 10^9$ bytes (when using doubles), which would not be storable on a 32bit architecture. Calculating $J^\top J$ would require about $n^2 m = 20 \cdot 10^{12}$ FLOPS, which alone would take about 1000 s on a 20 GFLOPS CPU in each iteration. For the above problem this could easily lead to execution times of an hour or more even on state-of-the-art hardware.

As it will be shown later in this chapter, a big proportion of J and $J^\top J$ consists only of zero entries making these matrices *sparse*. There is no exact definition of a sparse matrix, but a commonly used is the following, first given by J. H. Wilkinson:

Definition 3.1 (Sparse Matrix)**.** "The matrix may be *sparse*, either with the non-zero elements concentrated on a narrow band centered on the diagonal or alternatively they may be

distributed in a less systematic manner. We shall refer to a matrix as *dense* if the percentage of zero elements or its distribution is such as to make it uneconomic to take advantage of their presence."[WR71]

It is clear, that by simply saying a matrix is sparse, if only its non-zero entries are stored, every matrix would be sparse. The above definition will surely depend on the methods how a sparse matrix is stored and on the efficiency of algorithms exploiting it. However, sparse matrices in practice are likely to give benefits in order of magnitude, even with off-the-shelf sparse algorithms, so the definition will be sufficient for this thesis.

3.2 Sparsity of Least Square Problems

First of all in this section it will be shown that the Jacobian J that arises in a LS is sparse and that the matrix $J^\top J$ is sparse as well. At the end some methods for solving sparse systems are mentioned.

3.2.1 Sparsity of the Jacobian

The measurement function f is assumed to be the combination of several smaller normalized subfunctions, that depends on a big vector $x \in N$, with $N = N_1 \times \ldots \times N_n$, but with each subfunction f_i depending only on a few components of $x = (x_1, \ldots, x_n)$. Since all f_i are normalized their individual components are independent, so every f_i can be split into its components. Therefore it is assumed in the rest of the chapter that every f_i is single dimensional with:

$$\begin{aligned} f_i : N &\to \mathbb{R} \\ x &\mapsto f_i(x_{i_1}, \ldots, x_{i_K}) \end{aligned} \tag{3.1}$$

The gradient of f in x is now defined as the gradient of $g_i(\delta) := f_i(x \boxplus \delta)$ (cf. algorithm 1, line 3).

Let $\mathcal{X}_i$ be the set of indexes that correspond to x_i in δ, that means with d_i being the dimension of N_i:

$$\mathcal{X}_1 := \{1, \ldots, d_1\}, \tag{3.2}$$
$$\mathcal{X}_2 := \{d_1 + 1, \ldots, d_1 + d_2\} \tag{3.3}$$
$$\vdots$$
$$\mathcal{X}_n := \left\{1 + \textstyle\sum_{i=1}^{n-1} d_i, \ldots, \sum_{i=1}^{n} d_i\right\}. \tag{3.4}$$

By using the definition of $\boxplus$ in section 1.4 one can see, that values of δ in positions which do not affect any x_{i_k} don't affect $g_i(\delta)$ either. Therefore the gradient of g_i can be non-zero only at positions that affect any x_{i_k}. That means $\frac{\partial}{\partial \delta} g_i(\delta)$ can be non-zero only at $\bigcup_{k=1}^{K} \mathcal{X}_{i_k} =: \mathcal{F}_i$. The Jacobian J now only consists of the lines g_i, i.e. $J_{i\bullet} = g_i$. This sums up to the following conclusion:

Proposition 3.1. *With J defined as above $J_{ij} \neq 0$ only if $j \in \mathcal{F}_i$.*

Going back to the introducing example from section 3.1 the space required for storing the entries of J now drops to mk entries assuming that each measurement depends on variables with a total dimension of k. For a typical case of 2D-SLAM this would be like 6 (a measurement between two poses) or 5 (a measurement between a pose and a landmark), requirering now only $2.4{\cdot}10^6$ bytes. Of course additional memory is needed to store the structure of J, but this is only a factor of about 1.5 asymptotically. What matters is that the space consumption now only grows linear in m (in the dense case n would usually grow linear with m as well, because in SLAM new measurements usually introduce new variables, and therefore letting the matrix grow quadratically.)

3.2.2 Sparsity and Calculation of $J^\top J$

The product of two sparse matrices is not necessarily sparse. The simplest counter-example would be a matrix $J \in \mathbb{R}^{n\times n}$ with only the first row filled with ones. Now the product $J^\top J$ would also be a matrix in $\mathbb{R}^{n\times n}$ but completely filled with ones, which surely can't be considered as sparse anymore.

However this will generally not happen for LSs if every sub-function depends only on a few variables. To see whether $(J^\top J)_{ij}$ is non-zero one has to look at the definition of the product:

$$(J^\top J)_{ij} = \sum_{k=1}^{m} J_{ki} J_{kj}, \tag{3.5}$$

which is non-zero only if there is a function f_k with $\{i, j\} \subset \mathcal{F}_k$.

It is now possible to find an upper bound for the density of $J^\top J$ for the introducing example. Suppose again every function depends on two variables having a dimension of 3. The diagonal blocks of $J^\top J$ are set as soon as one function depends on the corresponding variable.[1]

Now a measurement depending on N_i and N_j sets the blocks $\mathcal{X}_i \times \mathcal{X}_j$ and $\mathcal{X}_j \times \mathcal{X}_i$ which adds $2 \cdot 3 \cdot 3 = 18$ elements to $J^\top J$. Therefore $J^\top J$ in this example has a maximum of $9 \cdot n + 18 \cdot m$ entries, which can be reduced to its half by only saving the upper half of $J^\top J$ as it is symmetric. But again, most important here is that the number of non-zeroes grows only linear with m.

The number of FLOPS to calculate $J^\top J$ can be estimated by looking at another way to express this product:

$$J^\top J = \sum_{k=1}^{m} J_{k\bullet} J_{k\bullet}^\top, \tag{3.6}$$

with $J_{k\bullet}$ having 6 entries so each summand requires $\frac{6\cdot(6+1)}{2} = 21$ multiplications and the same amount of additions (taking advantage of the symmetry). This leads to a total sum of about $42m$ FLOPS needing only $2.1{\cdot}10^6$ FLOPS instead of $20{\cdot}10^{12}$ FLOPS for the naïve approach, which makes a tremendous difference needing on the same CPU only about 0.1 ms instead of 1000 s. This is of course a gain that can't be achieved in practice as for the sparse matrix multiplication some additional overhead appears. But again, what's crucial, is that the number of FLOPS grows only linear with m.

[1] Which should be the case, because otherwise there would be no way to estimate the variable anyway.

3.2.3 Solving Sparse Systems

In section 2.3 two methods were given for solving an equation like

$$(J^\top J)\delta = J^\top z, \tag{3.7}$$

which is needed to calculate a new increment δ.

One possibility is to calculate the Cholesky decomposition L of $J^\top J$ and then solve the systems

$$L\hat{\delta} = J^\top z \qquad \text{and} \tag{3.8}$$

$$L^\top \delta = \hat{\delta}. \tag{3.9}$$

The other possibility is to calculate the QR decomposition of $J = QR$ and then solve the linear system (cf. theorem 2.3):

$$R\delta = Q^\top z. \tag{3.10}$$

Although the latter looks like needing far less operations, this isn't always the case. For example for m much bigger than n the QR decomposition of $J \in \mathbb{R}^{m\times n}$ usually takes much more effort than the cholesky decomposition of $J^\top J \in \mathbb{R}^{n\times n}$, which can compensate for the extra work to calculate $J^\top J$ and $J^\top z$ and the extra linear system to be solved.

Also and more important is the amount of *fill-in* which occours when computing a matrix decomposition of a sparse matrix.

When the decomposition of a sparse matrix is calculated, ideally the factors (either L, or Q and R) would have the same sparsity pattern as the original matrix. Unfortunately this is only the case in some special cases. For most matrices additional entries called "fill-in" have to be introduced. To determine how much fill-in is needed and what possibilities exist to reduce fill-in is a rather complex task and will not be covered in this thesis. An in-depth analysis of it can be found in [Dav06].

4 Framework

A main objective of this thesis is to provide a framework, which makes it easy to implement and optimize arbitrary least square problems. Several design decisions had to be made to keep it reasonably fast, without increasing the complexity of implementation unreasonably.

This chapter will only describe the design decisions not the actual implementation, which will be covered in the next chapter.

4.1 Basic Types

Besides the main class `Estimator` the framework distuinguishes between three basic types:

- A `Manifold` represents just a simple manifold.
- A `RandomVariable` represents a random variable, which is to be optimized. It contains a manifold which represents the current value (the currently best approximation) of the random variable.
- A `Measurement` is a function which describes the relation between several `RandomVariable`s. The function has to be normalized as described in section 2.2.

There are several basic requirements for each type.

A `Manifold` has to implement the $\boxplus$ and $\boxminus$ operators, and its DOF have to be known. Additionally it can have several access methods to enable user code to initialize it and to calculate with it, but these are irrelevant for the main algorithm.

In order to evaluate a `Measurement` it has to hold references to the variables it depends on. In order to be normalized it needs to hold its value of expectation and covariance.[1] Additionally the framework needs to know the dimension of each measurement.

[1] They can be given implizit, e. g. if they are known to be 0 or I, respectively.

A `RandomVariable` needs to propagate the dimension of its inherent manifold and its $\boxplus$ operator.[2] It also has to give read access to its manifold to make it possible for functions to be evaluated, and it must be possible to initialize it (usually via a manifold).

During the next section additional requirements will arise.

4.2 Cartesian Product of Manifolds

As shown in section 1.4 manifolds can be combined to bigger manifolds by forming their cartesian product. A user could do this manually by implementing $\boxplus$, $\boxminus$ and the DOF of a combined manifold, but this would be error-prone and cumbersome. Therefore the framework provides a macro `BUILD_RANDOMVAR` which does the necessary work. It will be described in section 5.1.2.3.

4.3 Data holding

The main class needs to hold references or pointers to each variable which has to be optimized, and to each measurement function which describes the relation between the variables. Variables and measuremts are stored into a single container each.

When evaluating the whole function each subfunction can be evaluated by iterating the measurement container and writing the results into some array consecutively.

At some points it is however not necessary to evaluate the complete function, but only parts of it. In order to do so it is mandatory to know the index of the subfunction in the "big function". To simplify matters this is achieved by simply storing the index of the function into the corresponding `Measurement` object, as soon as it gets registered to the algorithm.

4.4 Evaluation of Jacobian

One of the main tasks for the framework is to calculate the Jacobian of the complete measurement function. In this framework this is always done numerically, as there isn't always a practical way to do so symbolically.

There are two common ways to compute a Jacobian numerically:

$$J_{\bullet j} = \frac{1}{d} f(X \boxplus d e_j) - f(X), \tag{4.1}$$

$$\text{or} \qquad J_{\bullet j} = \frac{1}{2d} (f(X \boxplus d e_j) - f(X \boxplus -d e_j)), \tag{4.2}$$

with $d > 0$ being a small scalar value and e_j being the jth unit vector. The first variant needs less function evaluations because $f(x)$ has to be calculated only once. The second variant however usually approximates the exact Jacobian better and it also has some advantages when the Jacobian is sparse.

The naïve way to evaluate the Jacobian would be to consecutively evaluate $f(X \boxplus d e_j)$ and subtract $f(X)$ (or evaluate $f(x \boxplus -d e_j)$ and subtract this). As observed in chapter 3, it is crucial to exploit the sparseness of J, therefore zero values have to be dropped then.

It is however possible to exploit the sparseness of J already when evaluating it. As observed in section 3.2.1 entry J_{ij} can be non-zero only if the ith subfunction depends on the variable

[2] Note that $\boxminus$ is only needed to normalize functions, which hold their expectation value as a manifold.

which goes over index j. With $\dim(x_k)$ denoting the dimension of the kth subvariable, $\mathrm{ind}(x_k)$ denoting its index in the complete variable vector, and $\mathcal{F}_{\mathrm{ind}(x_k)}$ holding the indexes of functions depending on x_k algorithm 3 shows how to evaluate the Jacobian efficiently.

To make this algorithm actually run efficiently, the values $\dim(x_k)$ and $\mathrm{ind}(x_k)$ have to be available. $\dim(x_k)$ is already required, and $\mathrm{ind}(x_k)$ can easily be generated by summing up the $\dim(x_k)$s. Furthermore the list $\mathcal{F}_{\mathrm{ind}\, x_k}$ is required for every k, i. e. the list of functions that depend on x_k. As there is no efficient way to generate this online, it is stored within each `RandomVariable`. As soon as a function is registered to the main class, it registers itself to every variable it depends on.

Note that algorithm 3 fills the Jacobian strictly column-wise and as long as the dependencies between functions and variables do not change or new functions or variables are inserted the non-zero pattern of J stays exactly the same. Therefore it is possible to evaluate the non-zero pattern just once, and afterwards just recalculate the actual values.

4.5 Building Measurements

There are several requirements for a `Measurement`. It must hold references to its variables, and it must usually hold data to evaluate and normalize its value. Furthermore it must register itself at the variables it depends on. And especially the references to the variables must be initialized within the constructor.

Like generating the cartesian product of manifolds, this would also be an error-prone task to do manually. Therefore a macro called `BUILD_MEASUREMENT` was implemented for that task. It is described in section 5.1.3.

Algorithm 3 Evaluating the sparse Jacobian

Require: Temporary array *temp*
Require: $d > 0$
for all $x_k \in X$ **do**
  $x_{backup} \leftarrow x_k$
  for $j = 0, j < \dim(x_k)$ **do**
    $x_k \leftarrow x_{backup} \boxplus de_j$
    for all $i \in \mathcal{F}_{\mathrm{ind}(x_k)}$ **do**
      $temp[i] \leftarrow f_i(X)$
    end for
    $x_k \leftarrow x_{backup} \boxplus (-de_j)$
    for all $i \in \mathcal{F}_{\mathrm{ind}(x_k)}$ **do**
      $J_{i,\mathrm{ind}(x_k)+j} \leftarrow \frac{1}{2d}(temp[i] - f_i(X)$
    end for
  end for
  $x_k \leftarrow x_{backup}$
end for

5 Implementation

This chapter will deal with the actual implementation of the algorithms described and explain some internal design decisions. A major design goal was to avoid virtual polymorphism as much as possible by using static polymorphism.

5.1 Basic Types

This section will describe the interfaces of the basic types for describing manifolds, random variables, and measurement functions.

5.1.1 Manifolds

`Manifold`s are completely defined using only static polymorphism. To be a `Manifold` a class needs to have a public `enum {DOF}` describing its DOF, and must implement a $\boxplus$ and $\boxminus$ method (see section 1.3). These methods are called `add` and `sub` and have the following interface:

Listing 5.1: Manifold interface

```
const double * add(const double *vec, double scale=1);
double * sub(double *res, const Derived& oth) const;
```

`m.add(vec, scale)` lets $m \leftarrow m \boxplus scale \cdot vec$, where `vec` is handled as standard C array. It returns the address of the first element after the vector, i. e. `vec+DOF`.

`m.sub(res, oth)` lets $res \leftarrow m \boxminus oth$ and returns the address of the first element of `res`, that isn't used, i. e. `res+DOF`.

The templated `struct Manifold` defined in `types/Manifold.h` implements `add` and `sub` and defines `DOF`, when given a baseclass, that implements `add_` and `sub_` which have the same interfaces but do not return a value. `Manifold` uses the Curiously Recurring Template Pattern (CRTP) to do this without virtual inheritence.

The idea of letting `add` and `sub` return the next adress is to make the cartesian product of manifolds be easier to implement (see section 1.4). When combining a list of manifolds to single manifold, one has to make sure that the resulting manifold has a `DOF` set to the sum of `DOF`s of its members, and that `add` and `sub` call the corresponding methods of the members in a defined order. This can be done automatically by the macro `BUILD_RANDOMVAR(name, entries)`, described in section 5.1.2.3.

5.1.2 Random Variables

Random variables are modeled using a virtual interface, since using static polymorphism would be too complicated for this purpose, as the main algorithm must be able to handle arbitrary random variables registered in any order. Random variables are defined in `types/RandomVariable.h`.

5.1.2.1 Interface

The interface of a random variable is:

Listing 5.2: Random Variable interface

```
struct IRVWrapper : public std::deque<const IMeasurement*>{
  virtual int getDOF() const = 0;
  virtual const double* add(const double* vec, double scale=1) = 0;
  virtual void store() = 0;
  virtual void restore() = 0;
  int registerMeasurement(const IMeasurement* m);
};
```

The `IRVWrapper` interface provides (basically) everything the main algorithm needs to know about a random variable. Via `getDOF()` it can determine its DOF, using `add` it can modify the random variable (note that the main algorithm itself only needs the $\boxplus$ operator but not $\boxminus$). Via `store` and `restore` it can store and restore values of random variables (to do so internally every variable holds a backup of itself), which is needed when modifying variables temporarly e. g. to numerically calculate a Jacobian, or when reverting unsuccessful optimization steps.

The `registerMeasurement` method finally lets each variable keep track what measurements depend on it. This is necessary to efficiently calculate the Jacobian.

5.1.2.2 Implementation

To implement the interface from the previous section there is a templated wrapper class `RVWrapper`. Besides implementing the virtual functions of `IRVWrapper` it has some methods for user access:

Listing 5.3: RVWrapper class

```
template<typename RV>
class RVWrapper : public IRVWrapper{
  RV var;
  RV backup;
public:
  enum {DOF = RV::DOF};
```

```
  RVWrapper(const RV& v=RV(), bool optimize=true);

  // Getters and setters:
  const RV& operator*() const;
  const RV* operator->() const;
  const RV& operator=(const RV& v);
};
```

The template parameter `RV` is supposed to be a manifold, thus having an `enum {DOF}` and implementing the `add` method. `RVWrapper` always holds two instances of `RV` in order to implement the `store` and `restore` methods.

An `RVWrapper` can be generated using the constructor of the same name, which initializes it with a passed `RV`. Via the `optimize` parameter it can be decided, whether the random variable should actually be optimized.

The `*` and `->` operators give a const reference or pointer to the inlaying `RV var` and via the `=` operator both `var` and `backup` can be overwritten.

5.1.2.3 Cartesian Product of Manifolds

Very often a random variable will consist of a combination of manifolds. To help implementing such a combination a macro `BUILD_RANDOMVAR(name, entries)` is defined in `tools/AutoConstruct.h`. Its parameters are a name for the random variable and a list of entries, each consisting of a pair of a type and a name of a manifold.

`BUILD_RANDOMVAR` first defines a manifold `name_T` generated of the passed `entries`, and then `typedef`s a random variable `name` as `RVWrapper<name_T>`. For `name_T` the methods `add`, `sub`, and `getDOF` are implemented, in order to be compatible to a `Manifold` (i. e. `name_T` could be used within the `entries` of another `BUILD_RANDOMVAR`). Furthermore a constructor is generated, with every item from `entries` as parameter as seen here:

Listing 5.4: Example of constructing a random variable

```
BUILD_RANDOMVAR(AB, ((A,a)) ((B,b)))
// does the same as:
struct AB_T{
  A a;
  B b;
  enum {DOF = A::DOF + B::DOF};
  AB_T( const A& a=A(), const B& b=B())
    : a(a), b(b) {}
  int getDOF() const { return DOF; }
  const double* add(const double* vec, double scale=1) {
    vec = a.add(vec, scale);
    vec = b.add(vec, scale);
    return vec;
  }
  double* sub(double *res, const AB_T& oth) const {
    res = a.sub(res, oth.a);
    res = b.sub(res, oth.b);
    return res;
  }
};
typedef RVWrapper<AB_T> AB;
```

BUILD_RANDOMVAR is able to process up to 255 entries (although this hasn't been tested yet and is unlikely to be useful). It uses the Boost preprocessor macros [KM].

An object of AB can now be constructed as follows:

```
AB x;                 // x.a, x.b, initialised by standard constructor
AB y(AB_T(), false); // also std constructor, but y will not be optimized
AB z(AB_T(a));        // z.a = a, z.b=B();
// Furthermore it is possible to overwrite the entire variable:
x = *z;       // Note that *z returns the value of z's variable
z = AB_T();
// But it is not possible to modify single elements of AB:
// x->a = y->a; // error!
```

5.1.3 Measurements

A measurement must implement the following interface:

```
struct IMeasurement {
  virtual int getDim() const = 0;
  virtual int registerVariables() const = 0;
  virtual double* eval(double* res) const = 0;
};
```

getDim returns the dimension of the measurement. registerVariables registers itself at all variables it depends on and returns the sum of DOFs of variables, which are to be optimized.

The eval function does the actual work of the measurement, writing its normalized measurement value into res.

To implement an IMeasurement besides holding references to the variables it depends on and optionally further data these methods have to be implemented. As this is an error-prone (and somewhat cumbersome) a helper macro BUILD_MEASUREMENT similar to BUILD_RANDOMVAR is defined in tools/AutoConstruct.h. Its interface is as follows:

```
BUILD_MEASUREMENT(name, dim, variables, data)
```

where name is the name of the measurement, dim its dimension, variables the list of variables it depends on, and data a (possibly empty) list of additional data. The variables are stored as references to the corresponding RVWrapper class, the data is stored by value. To make a measurement storeable in std container classes, the = operator is overloaded.[1] However it must be avoided, that the address of a measurement changes once it is registered to the main algorithm.

BUILD_MEASUREMENT only declares the eval function, so the user has to define it later on as

```
double* name::eval(double* res);
```

It is supposed to changed exactly dim values starting at res[0] and return the address of the first element after that, i. e. res+dim.

[1] This is necessary as it holds references, which can't be assigned directly.

5.2 Manifolds

Some commonly used manifolds are already implemented by the framework. They are described in this section.

5.2.1 Vector

The most simple manifold is a common $\mathbb{R}^n$ vector, which is implemented as `Vect<D>` (with `D` $= n$)

Listing 5.5: Implementation of the Vect manifold

```
template<int D>
struct Vect : public Manifold<Vect<D>,D>{
  double data[D];

  Vect();
  Vect(const double* src);
  void add_(const double vec[D], double scale=1);
  void sub_(double res[D], const Vect<D>& oth) const;
  double& operator[](int idx);
  const double& operator[](int idx);
};
```

The vector elements can be accessed either by using the `[]` operators or by directly accessing the `data` member. It can be initialised via a `double` pointer or (by default) is set to zero.

5.2.2 $SO(2)$

The rotation group $SO(2)$ is implemented as `SO2`:

Listing 5.6: Implementation of the rotation group $SO(2)$

```
struct SO2 : public Manifold<SO2, 1>, public RotationGroup<SO2,2>{
  double angle;
  SO2(double angle = 0);
  SO2(const Vect<2> &dir);

  void add_(const double vec[1], double scale=1);
  void sub_(double res[1], const SO2& oth) const;
  void mult(const SO2& oth, bool invThis, bool invOth);
  void rotate(double res[2], const double vec[2], bool back=false) const;
  void rotate(Vect<2> &res, const Vect<2> &vec, bool back=false) const;
};
```

It just stores an angle (which doesn't have to be normalized). It can be initialised either by directly passing an angle or by passing a direction encoded as a two-dimensional vector `Vect<2>`.

Besides the required methods `add_` and `sub_` for the `Manifold` interface, it implements the methods `mult` and `rotate`, which can be used to concatenate rotations or to rotate vectors. These methods are also used by the `RotationGroup`, which uses them to define the operators `*`, `/`, and `%`. These operators behave as if `SO2` was a matrix between `SO2` and `Vect<2>`. The `%`-operator is implemented like Matlab's \-operator, i. e. multiplying from left by its inverse.

5.2.3 $SO(3)$

$SO(3)$ has almost the same interface as $SO(2)$. The difference is of course that its DOF is 3, and that `rotate` processes `Vect<3>` rather than `Vect<2>`. Internally it is implemented using unit quaternions.

5.2.4 MakePose

A very commonly used random variable is a pose, i. e. a combination of a position and a rotation. To easily implement this two macros `MAKE_POSE2D` and `MAKE_POSE3D` are implemented, which implement such a pose, providing methods to transform vectors and other poses from local coordinates to world coordinates and vice versa. The interface for the macro `MAKE_POSE2D`[2] is:

```
MAKE_POSE2D(name, posN, orientN, otherVars)
```

with `name` being the name of the pose, `posN` the name of the position vector, `orientN` the name of the orientation, and `otherVars` a list of further variables.

The implemented transformations are:

```
  name_T local2World(const name_T& oth) const;
  posT local2World(const posT& oth) const;
  name_T world2Local(const name_T& oth) const;
  posT world2Local(const posT& oth) const;
```

5.3 Helper Classes

To normalize a measurement as described in Equation 2.7, one often needs the Cholesky decomposition of a matrix. To do this a helper class `CholeskyCovariance` is implemented, which holds the Cholesky decomposition of a covariance- or information-matrix.

It can be initialized either by just copying an already decomposited matrix or perform the decomposition itself. Additionally one can choose whether the source matrix is stored as full matrix or just its upper half. Here all matrices are supposed to be saved in row-major form, as it was easier to implement internally and because multi dimensional plain C arrays are stored row-major as well. Row/column order doesn't matter of course, when a full matrix is passed and the lower half of a column major matrix is identical to the upper half of an row major matrix.

Listing 5.7: CholeskyMode

```
namespace CholeskyMode{
/**
 * How to initialize a CholeskyCovariance.
 * COPY_* means source already is a Cholesky factor,
 * CHOLESKY_* does the decomposition itself.
 * *_UPPER means only the upper half of the matrix is stored,
 * *_FULL means zeros or symetric values from lower half are stored.
 */
enum CM {
```

[2] `MAKE_POSE3D` is completely analogous.

```
  COPY_UPPER,
  COPY_FULL,
  CHOLESKY_UPPER,
  CHOLESKY_FULL
};
};

template<int dim>
struct CholeskyCovariance
{
  enum {DIM = dim, SIZE = (dim*(dim+1))/2};
  // chol is an lower triangular matrix, saved in row major order
  double chol[SIZE];

  CholeskyCovariance(const double *A, CholeskyMode::CM mode);
  void invApply(double* arr) const;
  void apply(double *arr) const;
};
```

Besides the constructor two methods `invApply` and `apply` are implemented. They multiply the array starting at address `arr` with L^{-1} or L respectively. The latter is useful if not the covariance but the information matrix was decomposited.

5.4 Main Algorithm

This section will describe the main algorithm. To optimize a LS an object of `Estimator` has to be instantiated and used as described in the next subsection. The subsequent subsections will describe, how the algorithm works internally.

5.4.1 Interface

In listing 5.8 the important parts of the interface are shown.

5.4.1.1 Constructor

The constructor gets three parameters. First of all `solver` specifies how the arising linear systems are to be solved, either by a `QR` decomposition, or by a `Cholesky` decomposition. `Cholesky` turned out to be a good choice, especially when having much more measurements than variables.

The `Algorithm alg` parameter chooses the mode of finding the next update. Essentially it is possible to choose different damping terms D for the equation:

$$(J^\top J + D)\delta = J^\top [y - f(\beta)] \tag{5.1}$$

With $D = 0$ (i.e. no damping term) for `GaussNewton`, $D = \lambda^2 \mathrm{I}$ for `Levenberg`, and $D = \lambda^2 diag(J^\top J)$ for `LevenbergMarquardt` (cf. section 2.4.3 and section 2.4.4). Usually `GaussNewton` is a good first choice, unless the initialisation of the variables is very poor.

Finally `lamda0` sets the start value of the damping parameter λ, which is required for `Levenberg` and `LevenbergMarquardt`. For `GaussNewton` this value is just ignored.

Listing 5.8: Interface of the main algorithm

```
class Estimator
{
public:
// use the following "damping term" D in (J^T J + D)δ = J^T [y − f(β)]
  enum Algorithm{
    GaussNewton,        // D = 0
    Levenberg,          // D = λ^2 * I
    LevenbergMarquardt  // D = λ^2 * diag(J^T J)
  };

  enum Solver{
    QR,
    Cholesky
  };

public:
  Estimator(Solver solver, Algorithm alg=GaussNewton, double lamda0=1e-3);

  void insertRV(IRVWrapper *var);
  void insertMeasurement(IMeasurement* meas);

  void initialize();

  double optimizeStep();
};
```

5.4.1.2 Inserting Random Variables and Measurements

After constructing an `Estimator` random variables and measurements have to be inserted (or "registered"). This is done via the `insertRV` and `insertMeasurement` methods, which require a pointer to an `IRVWrapper` or `IMeasurement` respectively.

Two things are very important here. The address of the variables and measurements isn't allowed to change after they have been inserted.[3]

Furthermore variables always have to be inserted before any measurements, that depend on them. Apart from that the order in which variables are inserted doesn't matter.

5.4.1.3 Initializing and Optimizing

After all variables and measurements are registered the algorithms has to be initialized, via the method `initialize()`. Within this method memory is allocated and the sparsity structure of the Jacobian is precalculated.

Finally via `optimizeStep()` the problem can be optimized successively. The method directly modifies the registered random variables. It returns the last optimization gain i. e.

$$gain := \frac{\|f(x_k)\|^2 - \|f(x_{k+1})\|^2}{\|f(x_{k+1})\|^2}. \tag{5.2}$$

The user then has to decide whether he wants to continue optimizing or not.

[3] That means when storing variables or measurements in a `std::vector` it has to be sufficiently preallocated.

6 Results

The framework was successfully tested with several data sets, from which two are presented in this chapter, with various variations. This chapter demonstrates the simplicity in which LS can be implemented and also shows the enormous convergence speed of the algorithm.

The first data set is a syntetic data set provided by Edwin Olson [OLT06]. It consists of 2D poses and relations between them.

The second data set is Udo Frese's DLR-dataset [Fre], which contains odometry and landmark measurements (with known data association). For this data set some experiments were made like optimizing a calibration problem or just optimizing poses while keeping the landmarks fixed. The former is useful especially in real-life problems where it is often not possible to perfectly calibrate sensors. The latter is useful, e. g. when evaluating the results of other algorithms which only output landmarks.

6.1 Syntetic Data Set

Edwin Olson's simulated data set consists of poses moving along a grid-world path, which makes it easy to visually rate the obtained result, as only orthogonal corners appear and paths frequently overlap.

This data set consists only of poses and measurements between them, therefore only one type of random variable and one type of measurement had to be implemented (see listing 6.1). The odometry measurement transforms the endpose `t1` in the local coordinate system of the start pose `t0` and then subtracts the measured odometry (using `sub`, which corresponds to the $\boxminus$ operator). Afterwards the result is normalized by multiplying with the inverse Cholesky factor of the covariance.

In the main routine a logfile is read and for each relation an `Odo` object is created and stored. Furthermore whenever a new `frameID` is encountered a new `Pose` is created, which is initialized by the inverse odometry model (see listing 6.2).

Listing 6.1: Variable and Measurement for Relationgraph

```
MAKE_POSE(Pose, Vect<2>, pos, SO2, orientation, )

BUILD_MEASUREMENT(Odo, 3, ((Pose, t0)) ((Pose, t1)),
        ((Pose_T, odo)) ((CholeskyCovariance<3>, cov)) )
double* Odo::eval(double ret[3]) const
{
  Pose_T diff = t0->world2Local(*t1);
  diff.sub(ret, odo);
  cov.invApply(ret);
  return ret+3;
}
```

Listing 6.2: Initialization for Relationgraph

```
  // Instantiate Estimator:
  Estimator e(Estimator::Cholesky, Estimator::GaussNewton);

  // Data holding for variables and measurements:
  deque<Pose> poses; deque<Odo> odo;

  // Set 0th pose as (0,0,0) and don't optimize it
  poses.push_back(Pose(Pose_T(), false));
  e.insertRV(&poses.back());

  std::string line;
  while(getline(logfile, line)){
    // read a line:
    unsigned int frameA, frameB;
    double xEst[3], xCov[3][3];
    if(sscanf(line.c_str(),/*...*/)!=14) continue;
    scale(xEst, xCov, 1, M_PI/180);  //scale to radian
    Pose_T delta(xEst, xEst[2]);
    if(frameA>=poses.size()){
      poses.push_back(poses[frameB]->local2World(delta));
      e.insertRV(&poses.back());
    }
    // Create Measurement between frameB and frameA:
    odo.push_back(Odo(poses[frameB], poses[frameA], delta,
        CholeskyCovariance<3>(&xCov[0][0],CholeskyMode::CHOLESKY_FULL)));
    e.insertMeasurement(&odo.back());
  }
```

After initializing the variables and measurements, the `Estimator` itself has to be initialized (allocating workspace and precalculating the structure of the Jacobian) by calling its `initialize()` method. Afterwards the method `optimizeStep()` is called several times until the gain drops below a certain level (10^{-9}). Between optimization steps an intermediate result is stored in an output file. The optimization loop is shown in listing 6.3.

The residual sum of squares (RSS) drops from $1.13 \cdot 10^8$ after initialization to 6415.57 in four steps, afterwards it slowly drops further to 6413.96, where the optimization converges after a total of seven steps. The data set has 3500 poses and about 5600 measurements between them. The total optimization time (excluding data output) was about 0.95 s on a 1.86 GHz CPU. The initialization and final graph can be seen in figure 6.1 and figure 6.2.

6.2 DLR Data Set

The second data set was provided by Udo Frese [Fre]. The data set consists of odometry and landmark measurements. Both measurements are given in relative coordinates and with covariance information.

6.2.1 Preparations

Like in previous example, first of all the variables and measurements have to be defined. The `Pose` and the `Odo` types are implemented the same way as in listing 6.1. This example adds landmarks, which can be modeled as a `Vect<2>`. The landmark measurement is implemented similar to the odometry measurement by transforming the landmark from world coordinates into the local coordinate system of the pose. Afterwards the difference to the measured landmark is calculated and normalized by the inverse covariance (see listing 6.4).

6.2.2 Data Initialization

As the parser for this data set is a bit more complex, only the pure data initialization is demonstrated here. The landmarks do not necessarily come in order, therefore they are stored

Listing 6.3: Optimization Loop for Relationgraph

```
// Initialize Estimator:
e.initialize();

int kMax = 20; // limit nr of iterations
for(int k=0; k<kMax; k++){
  // optimize:
  double gain=e.optimizeStep();
  // Output intermediate result:
  outputPoses(poses, make_filename("output",k,".pos").c_str());
  // break if gain < 1e-9:
  if(0<=gain && gain < 1e-9) break;
}
```

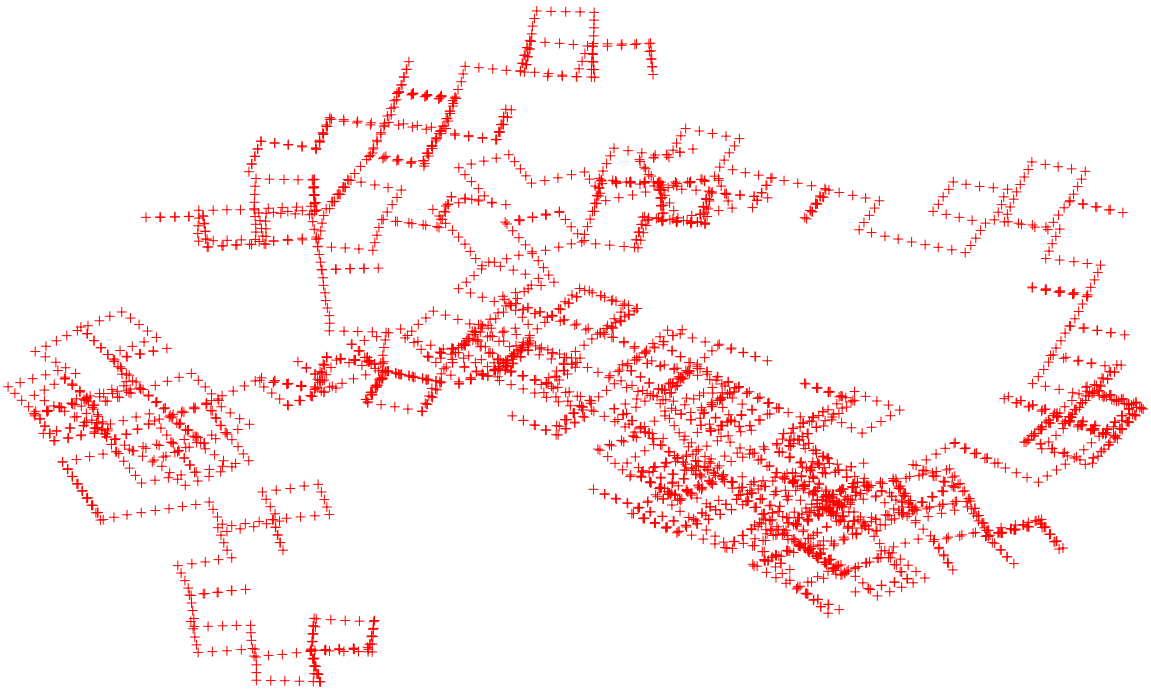

Figure 6.1: Initialization for Olson's data set.

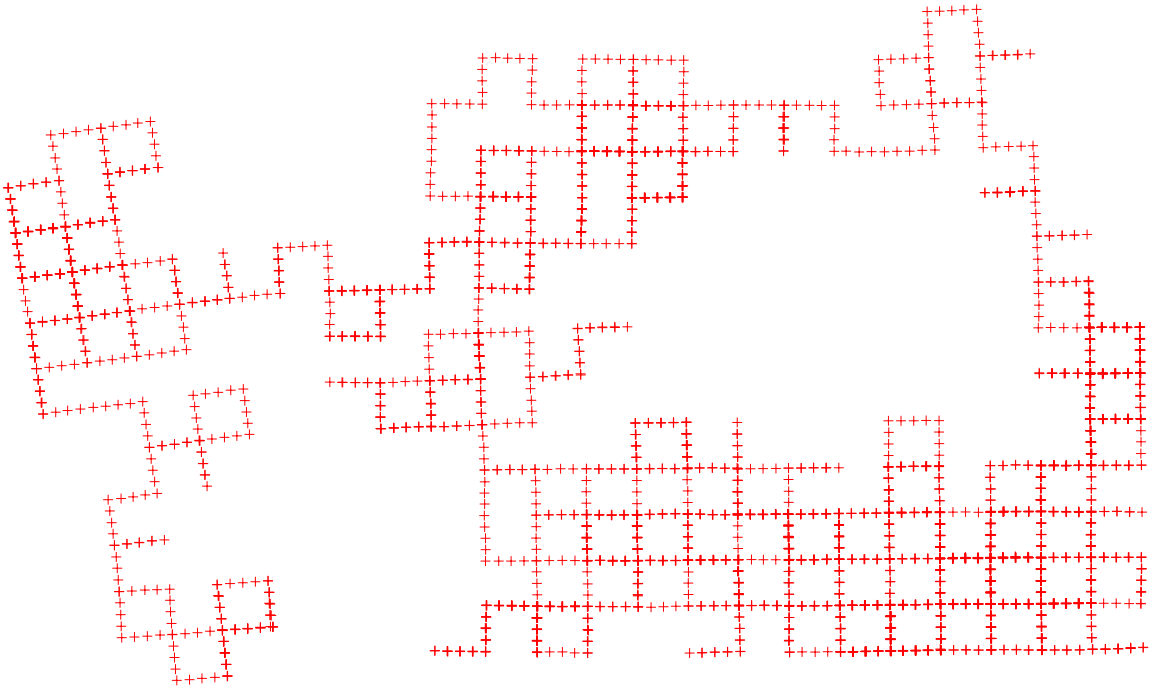

Figure 6.2: Olson's data set after optimization.

Listing 6.4: Landmarks for DLR data set

```
typedef RVWrapper<Vect<2> > LandMark;

BUILD_MEASUREMENT(LM_observation, 2, ((Pose, pose)) ((LandMark, lm)),
     ((Vect<2>, rel_coord )) ((CholeskyCovariance<2>, cov)) )
double* LM_observation::eval(double ret[2]) const
{
  Vect<2> landmark = pose->world2Local(*lm);
  rel_coord.sub(ret, landmark);
  cov.invApply(ret);
  return ret+2;
}
```

in a `std::map<int, *LandMark>` (note that the pointer is necessary because the position of the `LandMark` object isn't allowed to change).

The initialization for the poses is again similar to the previous example, except this time at every step a new pose is generated. The initialization of the landmarks is shown in listing 6.5.

6.2.3 Optimization Result

The optimization loop is again identical to listing 6.3 apart from the different data output.

The data set has 3298 poses, 576 landmarks, and 14309 landmark observations. The RSS drops in eight steps from $3.7 \cdot 10^8$ to 56871.5, where it converges. The optimization time was about 2.5 s on a 1.86 GHz CPU.

The initial map is shown in figure 6.3, the optimized map is shown in figure 6.4.

Listing 6.5: DLR data set landmark initialisation

```
  // obtained from parser:
  int id;              // Landmark id
  Vect<2> lm_local;    // Local coordinates of landmark
  double *covariance; // Covariance of landmark measurement

  // Last added pose:
  Pose& curr_pos=poses.back();

  // Look for id in landmark map:
  if (landmarks.find(id) == landmarks.end() ) {
    Vect<2> lm_global = curr_pos.local2World( lm_local );
    LandMark* n = new LandMark(lm_global);
    landmarks.insert(make_pair( id, n ) );
    e.insertRV( landmarks[id]);  // register landmark
  }
  // Generate observation from current pose to landmark "id":
  lm_obs.push_back( LM_observation(curr_pos, *landmarks[id], lm_local,
      CholeskyCovariance<2>(covariance, CholeskyMode::CHOLESKY_UPPER ) ));
  e.insertMeasurement( &lm_obs.back() );
```

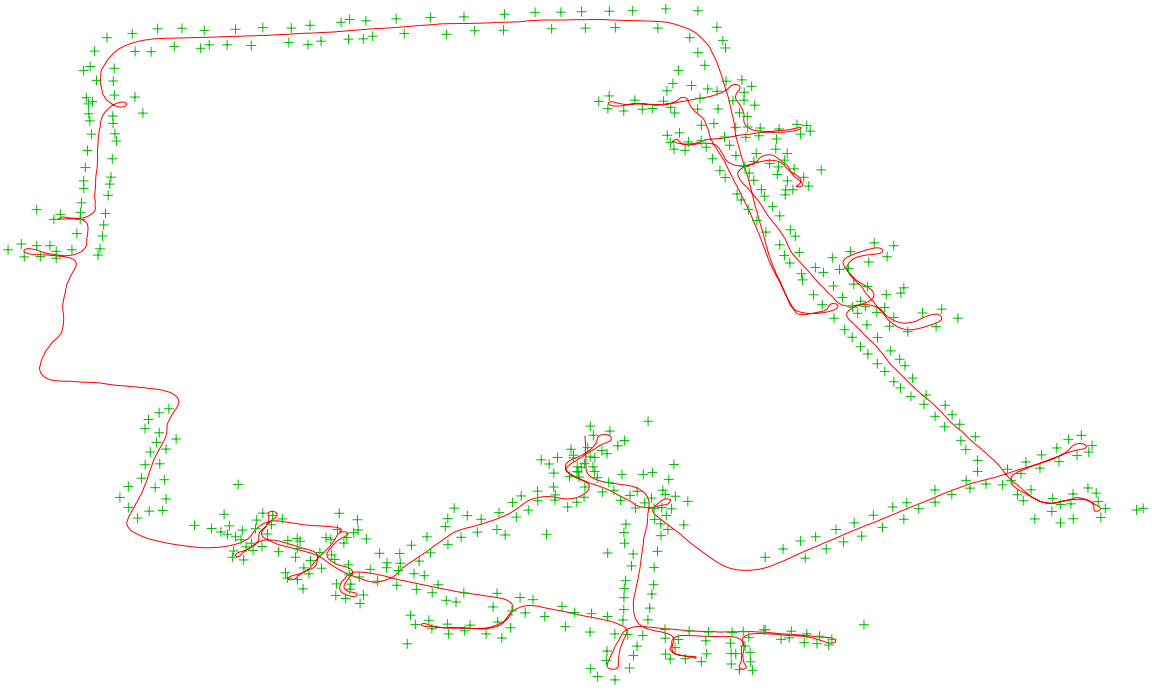

Figure 6.3: DLR data set with raw odometry.

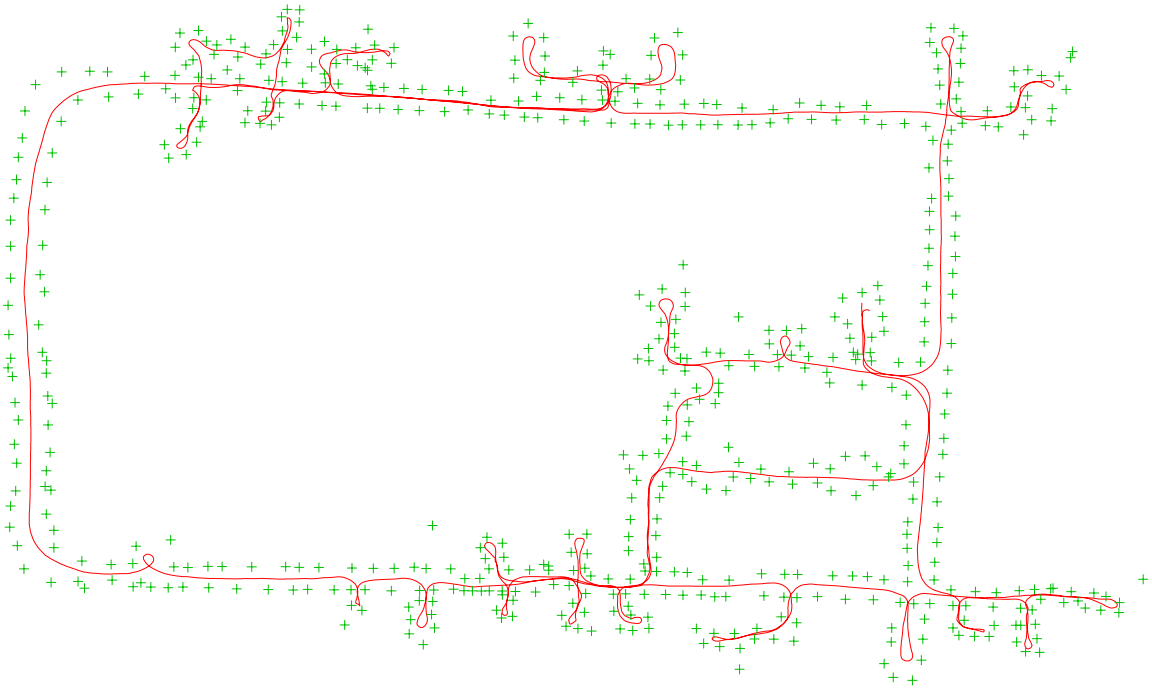

Figure 6.4: DLR data set after optimization.

6.2.4 Partial Optimization

A nice feature of the framework is to optimize only parts of a data set. A possible use case for this is the evaluation of algorithms which perform some kind of data reduction, e. g. combine local sub maps and lose the pose information therein before doing a global optimization.

If the result of such an algorithm shall be compared with the optimal solution, one has to reestimate the poses afterwards. This can be done by the framework, by inserting the landmarks as fixed variables into the algorithm:[1]

```
int id;
Vect<2> data;  // read from logfile
// Do not optimize n:
LandMark* n=new LandMark(data, false);
// store and register to algorithm
landmarks.insert(make_pair( id, n ) );
e.insertRV(n);
```

This was done for several data sets. The optimization time was slightly faster than the full optimization. The RSS was not surprisingly always bigger than the optimal result. The actual results are not that important within this thesis.

6.2.5 Calibration Problems

A class of problems which can't be solved by standard SLAM algorithms are calibration problems. When miscalibrated sensors are used, they can be calibrated simultaneously with optimizing the SLAM problem, using this framework.

Suppose for example, the odometry has an unknown scale factor for each component, i. e. for the measured odometry $\tilde{u} = [\tilde{x}, \tilde{y}, \tilde{\alpha}]^\top$, the real odometry would be (neglecting noise):

$$\hat{u} = [c_0\tilde{x}, c_1\tilde{y}, c_2\tilde{\alpha}]^\top. \tag{6.1}$$

This can easily be modelled in the framework by introducing a new variable `Calibration` which holds a `Vect<3>`, and adapting the `Odo` measurement as shown in listing 6.6.

During initialization this time a `Calibration` object has to be instantiated and registered to the framework. This object has then to be passed to every odometry measurement, as shown in listing 6.7.

The initialization of landmarks and the optimization is still the same. Amazingly the RSS of the optimized problem dropped significantly from 56871.5 to 42933.1 using the original data.

After optimizing, the values from `calib` were displayed:

$$c_0 = 1.05704 \qquad c_1 = 0.924149 \qquad c_3 = 1.00269, \tag{6.2}$$

which means that there might be a systematic error of over 5% in the translation, and 0.2% in the orientation. However this can't be said for sure, as the error theoretically might be caused by miscalibrated landmark measurements as well. Note that simultaneously applying the same

[1] This has to be done before the "other" initialization. Note that the later initialization does not overwrite the landmarks.

Listing 6.6: Odometry with calibration

```
typedef RVWrapper<Vect<3> > Calibration;

BUILD_MEASUREMENT(Odo, 3, ((Pose, t0)) ((Pose, t1)) ((Calibration, calib)),
        ((Pose_T, odo)) ((CholeskyCovariance<3>, cov)) )
double* Odo::eval(double ret[3]) const{
  // Apply calibration to odo:
  Pose_T odoC = odo;
  const double* cal = calib->data;
  odoC.pos[0] *= cal[0];
  odoC.pos[1] *= cal[1];
  odoC.orientation.angle *= cal[2];

  Pose_T diff = t0->world2Local(*t1);
  diff.sub(ret, odoC);   // subtract calibrated data.
  cov.invApply(ret);
  return ret+3;
}
```

Listing 6.7: Registering odometry with calibration

```
  Estimator e(Estimator::Cholesky, Estimator::GaussNewton);

  Calibration calib;    // Calibration object
  e.insertRV(&calib);

  while(1){
    Pose_T rel_pose;     //
    double *covariance;  // read from logfile

    Pose_T next_pos = poses.back()->local2World( tmp_pose );
    poses.push_back( next_pos );
    odometry.push_back( Odo( *(poses.end()-2), *(poses.end()-1), calib,
        rel_pose,
        CholeskyCovariance<3>(cov, CholeskyMode::CHOLESKY_UPPER) )
    );
    e.insertRV( &poses.back());
    e.insertMeasurement( &odometry.back() );

    // process landmarks ...
  }
```

calibration strategy to the landmarks will not give meaningful results, as this gives the whole system at least one undefined DOF namely an over all scale factor.[2]

The calibration problem was repeated by intentionally scaling each odometry value after reading it from the logfile. For the translational scale quite arbitrary factors could be used, while still obtaining the same results (when scaled by the same factor). For the orientational part however too big errors could not be compensated and let the algorithm get stuck in local minima.

[2] In that case the optimal result would be to set every factor to 0.

7 Conclusion

7.1 Achieve Goals

This thesis provided a formalization of manifolds and the encapsulation operators $\boxplus$, $\boxminus$ proposed in [FS].

It was shown that using these encapsulation operators functions between manifolds can be normalized such that they behave like general functions locally and standard optimization algorithms can be applied.

This was used this to solve arbitrary least squares problems on manifolds, first theoretically, and later in practice.

Exploiting the sparseness of the least squares problem and using a sparse library package for direct methods leads to rather fast optimization times, which even outperforms some specialized offline SLAM algorithms.

It was showed that using this framework it is possible to

- solve an offline SLAM problem with very little implementation necessary.
- just partly optimize problems (e. g. keep some variables fixed).
- solve calibration problems (which is hard to do with standard SLAM algorithms).
- test the accuracy of other LS algorithms.

7.2 Outlook

Unfortunately it wasn't possible in the end to implement a good 3D example, which would have demonstrated the usefulness of manifolds better.

Some possibilities for further research arose during the development of this thesis:

- Give the framework the possibility to handle online problems better

- Make it possible to change correspondences between variables and measurements between optimizations
- maybe use another backend like Treemap [Fre07] for the optimization.

Bibliography

[Cop95] Coplien, James O.: Curiously recurring template patterns. In: *C++ Rep.* 7 (1995), Nr. 2, S. 24–27. – ISSN 1040–6042

[Dav] Davis, Timothy A.: *Direct Methods for Sparse Linear Systems, and the CSparse Package.* `http://www.cise.ufl.edu/research/sparse/CSparse/`, Last checked: 2008-11-01

[Dav06] Davis, Timothy A.: *Direct Methods for Sparse Linear Systems.* Philadelphia : siam, 2006 (Fundamentals of Algorithms)

[Fre] Frese, Udo: *Deutsches Zentrum für Luft- und Raumfahrt (DLR) dataset.* `http://www.sfbtr8.spatial-cognition.de/insidedataassociation/data.html`, Last checked: 2008-11-01

[Fre07] Frese, U.: Efficient 6-DOF SLAM with Treemap as a Generic Backend. In: *Procedings of International Conference on Robotics and Automation, Rome*, 2007

[FS] Frese, Udo ; Schröder, Lutz: *Theorie der Sensorfusion Skript 2006.* `http://www.informatik.uni-bremen.de/agebv/de/VeranstaltungTDS06`, Last checked: 2008-11-01

[HZ03] Hartley, Richard ; Zisserman, Andrew: *Multiple View Geometry in Computer Vision.* Cambridge University Press, 2003. – ISBN 0521540518

[KM] Karvonen, Vesa ; Mensonides, Paul: *The Boost Library, Preprocessor Subset for C/C++.* `http://www.boost.org/doc/libs/1_36_0/libs/preprocessor/doc/index.html`, Last checked: 2008-11-01

[Lee03] Lee, John M.: *Introduction to Smooth Manifolds.* Springer Verlag, 2003

[Lev44] Levenberg, Kenneth: A Method for the Solution of Certain Non-Linear Problems in Least Squares. In: *The Quarterly of Applied Mathematics* (1944), Nr. 2, S. 164–168

[Mar63] Marquardt, D. W.: An Algorithm for Least-Squares Estimation of Nonlinear Parameters. In: *Journal of the Society for Industrial and Applied Mathematics* (1963)

[MYB+01] Marins, João L. ; Yun, Xiaoping ; Bachmann, Eric R. ; Mcghee, Robert B. ; Zyda, Michael J.: An extended Kalman filter for quaternion-based orientation estimation using MARG sensors. In: *Engineer's Thesis, Naval Postgraduate School*, 2001, 2003–2011

[NM65] Nelder, J. A. ; Mead, R.: A Simplex Method for Function Minimization. In: *The Computer Journal* 7 (1965), January, Nr. 4, S. 308–313

[OLT06] Olson, Edwin ; Leonard, John ; Teller, Seth: Fast Iterative Optimization of Pose Graphs with Poor Initial Estimates, 2006, S. 2262–2269

[PTVF92] Press, William ; Teukolsky, Saul ; Vetterling, William ; Flannery, Brian: *Numerical Recipes in C*. 2nd. Cambridge, UK : Cambridge University Press, 1992

[TBF05] Thrun, S. ; Burgard, W. ; Fox, D.: *Probabilistic Robotics*. MIT Press, 2005

[Ude99] Ude, Aleš: Filtering in a unit quaternion space for model-based object tracking. In: *Robotics and Autonomous Systems* 28 (1999), August, Nr. 2-3, S. 163–172

[Vic01] Vicci, L.: Quaternions and Rotations in 3-Space: The Algebra and its Geometric Interpretation / Dept. of Computer Science, University of North Carolina at Chapel Hill. 2001 (TR01-014). – Technical Report

[WI95] Wheeler, Mark D. ; Ikeuchi, Katsushi: Iterative estimation of rotation and translation using the quaternion / Carnegie Mellon University, Pittsburgh, PA. Version: 1995. `http://dx.doi.org/10.1.1.71.6291`. 1995. – Forschungsbericht

[Wika] Wikipedia: *Hairy Ball Theorem*. `http://en.wikipedia.org/wiki/Hairy_ball_theorem`, Last checked: 2008-11-01

[Wikb] Wikipedia: *Stereographic Projection*. `http://en.wikipedia.org/wiki/Stereographic_projection`, Last checked: 2008-11-01

[WR71] Wilkinson, James H. ; Reinsch, Christian: *Handbook for Automatic Computation*. Vol. II: Linear Algebra. Springer-Verlag, 1971. – Source of this quote: `http://www.netlib.org/na-digest-html/07/v07n12.html#1`