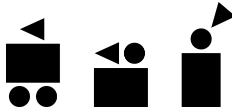
UNIVERSITÀ DI PISA



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis in Aerospace Engineering
University of Pisa
Year 2013/2014

Dense visual odometry and sensor fusion for UAV navigation

Nicolò Valigi

UNIVERSITY OF PISA TUTOR:

Prof. Roberto Galatolo

In the beginner's mind there are many possibilities, in the expert's mind there are few.

Alla mia famiglia, ad Angela, e a chi mi è stato vicino in questi anni.

ABSTRACT

We tackle the SLAM (Simultaneous Localization and Mapping) problem for Unmanned Aerial Vehicles with a combination of two promising technologies: incremental smoothing and dense visual odometry using stereo cameras. In constrast to filtering-based approaches, incremental smoothing refers to the possibility of performing efficient optimization of all the past system pose to improve accuracy and reliability. In dense visual odometry, camera images taken as the robot moves are aligned to compute the relative pose between the two viewpoints.

For an optimal fusion of information from the IMU and the cameras, we go further than the usual loosely coupled approach and directly integrate pixel-level information into the SLAM optimization problem.

ABBREVIATIONS

IMU Inertial Measurement Unit

VO Visual Odometry

DVO Dense Visual Odometry

GTSAM Georgia Tech Smoothing and Mapping

SE3 Special Euclidean 3

SO3 Special Orthogonal 3

PDF Probability Density Function

ROS Robot Operating System

LS Least Squares

CONTENTS

1 INTRODUCTION 1
2 INCREMENTAL SMOOTHING OF SENSOR DATA 4
  2.1 The SLAM Problem 4
  2.2 Incremental smoothing 9
  2.3 Optimization on non-Euclidean spaces 16
  2.4 Inertial navigation 20
3 DENSE VISION ODOMETRY 22
  3.1 The pinhole camera model 23
  3.2 The algorithm 24
  3.3 Using a stereo camera 29
  3.4 Hardware and other technologies used 31
4 THE LOOSE LOOP APPROACH 34
  4.1 Inertial navigation with ground data 34
  4.2 Smoothing for Visual Odometry 40
  4.3 Execution time 44
  4.4 The effect of keyframe selection 46
5 THE TIGHT LOOP APPROACH 51
  5.1 Leveraging the inertial platform 51
  5.2 Building a unified optimization problem 54
  5.3 Evaluation 62
  5.4 Selective marginalization 65
6 CONCLUSIONS 68
A THE LOOSELY COUPLED FACTOR 69
B THE CALIBRATED FACTOR 71

LIST OF FIGURES

Figure 1 A simple Bayes network for a SLAM problem 6
Figure 2 Bayes networks in the case of the full problem, filtering, and smoothing approaches. 7
Figure 3 A simple factor graph with unary and binary factors. 10
Figure 4 Factor graph solution process. 14
Figure 5 A Bayes network and its equivalent Bayes tree. 14
Figure 6 The pinhole camera model 23
Figure 7 Residual with respect to translation of the image in the X direction 28
Figure 8 Once the camera baseline is known, disparity can be used to compute the pixel depth. 30
Figure 9 The VI-sensor platform, with twin cameras and IMU. 31
Figure 10 Rviz interactive 3D pose visualization tool. 33
Figure 11 Coordinates frames as described in section 4.1. 35
Figure 12 Factor graph with absolute visual odometry measurements. 36
Figure 13 Online and offline estimations of the accelerometer bias. 37
Figure 14 Coordinates frames as described in section 4.1.2. 38
Figure 15 Online and offline estimations of the Body-Camera frames relative pose. 39
Figure 16 Online and offline estimates of the CB pose with inaccurate prior knowledge. 40
Figure 17 Coordinates frames as described in section 4.2. 41
Figure 18 Coordinates frames as described in section 4.2, in the case of no additional keyframes 42
Figure 19 Factor graphs for the visual odometry-based smoother. 43

Figure 20 Error resulting from translations in different directions. 44
Figure 21 Factor graph optimization time as function of the number of included frames. 45
Figure 22 Execution times for DVO and GTSAM. 46
Figure 23 Effect of the keyframe interval on the residual error at convergence. 48
Figure 24 Distribution of keyframe intervals with the rotation-based heuristic. 49
Figure 25 PDF of the residual error at convergence for the two keyframe selection approaches. 50
Figure 26 PDF of the number of iterations for both the "naive" and IMU-based pose prediction 52
Figure 27 PDF of the rotation prediction error for the "naive" and IMU-based pose prediction 53
Figure 28 Operation with tightly-coupled visual odometry. 55
Figure 29 Operation with loosely-coupled visual odometry. 56
Figure 30 Execution time (in number of CPU clocks) for the iSAM2 algorithm and the batch solver. 63
Figure 31 Number of relinearized factors for different thresholds 64
Figure 32 Optimized trajectory as obtained by the iSAM smoother with different re-linearization thresholds. 65
Figure 33 Translation component as optimized by the incremental smoother and the fixed lag smoother with two different threshold. 67

1 INTRODUCTION

Surveillance, mapping, and aerial photography are some of the many areas that have recently been starting to take advantage of Unmanned Aerial Vehicles (UAVs). Many more possibilities are bound to appear in the coming years, like mail delivery or industrial inspection. In particular, quadrocopters (i.e. helicopters with four independent propellers) are capable of hovering and flying at low speeds, making them suitable for centimeter-accurate indoor missions.

Due to their aerodynamics, however, autonomous quadrocopters are inherently unstable and thus require specially crafted controller systems. Part of the problem lies in accurately localizing the robot with respect to its environment. While outdoor applications usually take advantage of the satellites of the Global Positioning System (GPS), indoor-flying robots have no such possibility.

The robotics community has been tackling this *localization* problem for many decades now, using several different types of sensors, from cameras to laser beacons. Due to their widespread use and low cost, this thesis will be focusing on video cameras and Inertial Measurement Units (IMUs), which measure acceleration and rotational velocity of the robot. A familiar application of IMUs is found on smartphones, where they are used for sensing the screen orientation and to control some games by tilting the screen.

However, these two different sources of information can not be used together without a powerful framework for *sensor fusion*. Our bodies have incredibily advanced systems for this purpose: the vestibular system in the ear and eyes are tightly connected by the *vestibulo–ocular reflex* to optimize our perception of balance and space. In the first part of the thesis, we will discuss how the robotics community has been trying to emulate such a great achievement.

It is no wonder that man's visual cortex takes up the biggest part of its brain: cameras produce a wealth of visual information that is very difficult to make sense of, especially with a computer. For this reason, the second part of this thesis will discuss a state-of-the-art procedure for the vehicle to understand its own motion by looking at images of its sorroundings produced by the on-board cameras.

Finally, we present our solutions to the sensor fusion problem, first with a "loose" solution and then with tighter integration between image and motion sensing for optimal performance.

TECHNICAL INTRODUCTION

In the present work, we build on the literature about SLAM and visual odometry to implement a smoothing-based approach for UAV navigation using dense image information from a stereo camera and an inertial measurement unit.

As opposed to *feature-based* procedures, dense vision odometry requires no costly feature extraction step and operates instead by aligning images from two consecutive viewpoints. The alignment step is carried out by minimizing the least-squares error of the intensity of corresponding pixels in images taken during the motion. Compared to feature-based methods, dense visual odometry requires no fine tuning of the parameters and is thus more reliable.

Regarding the sensor fusion part, we implement both a "loosely" and "tightly" coupled solution. The former is the mainstay solution for its computational speed and versatility: visual odometry motion estimates are simply included as relative robot pose measurements (as could be obtained by wheel encoders).

Since both sensor fusion and visual odometry are formulated as least-squares optimizations, it is natural to combine everything into a single problem. This motivation leads to the "tightly coupled" approach, in which vision and IMU data are fed into the same SLAM problem.

For both "loosely" and "tightly" coupled systems, we eskew the traditional *filter-based* approach to localization and perform full optimization over all past robot poses instead (*smoothing*). We take advantage of recent developments in literature to per-

form the optimization *incrementally* and allow for real-time execution on limited hardware.

OUTLINE OF THE WORK

In the first chapter, we describe some of the approaches to the SLAM problem (Simultaneous Localization and Mapping). In particular, we devote some space to a promising line of action, namely *incremental smoothing*, that we will build on during the rest of the work.

In the second chapter, attention turns to the dense visual odometry algorithm for pose estimation, including some notions about its implementation on stereo RGB cameras (i.e. passive 3D).

In the third chapter, we implement and benchmark the loosely coupled approach using the GTSAM library for batch SLAM optimization and incremental smoothing. We specifically investigate the execution speed and the effect of keyframe selection.

In the fourth chapter, we conclude with a description of the tightly coupled localization solution and compare its performance with the loosely coupled approach.

2

INCREMENTAL SMOOTHING OF SENSOR DATA

2.1 THE SLAM PROBLEM

In the robotic community, SLAM stands for Simultaneous Localization and Mapping, and concerns the situation of an autonomous vehicle that finds itself in an unknown position in an unknown environment. Solving the SLAM problem means incrementally building a map of this environment while simultaneously using it to determine the absolute position of the robot.

Good SLAM performance is considered one of the most fundamental features of a truly autonomous robot, as it obviates the need for external infrastructure or prior mapping of the environment. Unfortunately, SLAM is also a clear chicken-and-egg situation, since on one hand a map is needed to localize the robot, while on the other hand an accurate position estimate is necessary for building such map. This means that every SLAM solution must proceed as a series of steps:

- First, the robot starts in a known absolute position;
- Once the robot starts to move, a *motion model* is used to predict the new position and its accuracy, which is bound to grow with time;
- The robot observes landmarks in the environment and estimates their position, thus building the *map*. The inaccuracy of the map is a combination of the inaccuracy in the robot position and the measurement errors.
- By remeasuring the same landmarks, the robot can improve the estimates of both its own position and the landmarks';
- The last two steps are repeated as appropriate.

As can be seen in the above description, a SLAM solution has to deal with a situation rife with uncertainty and noisy information that naturally lends itself to a probabilistic interpretation [7]. More specifically, our goal can be restated as estimating the *posterior* probability distribution:

$$Bel(x_t) = p(x_t|z_1, z_2, ..., z_t)$$

where x_t is the state of the system at time t and $z_1..z_t$ is the set of past measurements from sensors. In this model, the state x_t contains both the pose of the robot as well as an encoding of the map that the robot itself has been building. Such encoding can take different forms depending on the type of landmarks observed and of sensors used.

To simplify the computation of the posterior function, it is usually assumed that the system obeys the Markov property, meaning that each state of the system only depends on the previous one and that measurements at time t only depend on the state x_t at the same time. These simplifications allow the problem to be represented with a *Bayes network*, which is a directed acyclic graph having a node for each random variable in the system. Directed edges connect two nodes when there is a relation of conditional dependence between them. An example of a typical Bayes network is shown in figure 1 and will be discussed in the next section.

2.1.1 *A Bayesian interpretation*

Figure 1 is a graphical representation of information in the SLAM problem, and can be clearly explained with reference to a Bayesian interpretation of the estimation problem, as outlined generally in [25] and more specifically in [6]. The Bayesian approach provides powerful tools to draw inferences in the presence of uncertainty in the data. More specifically, every time a new measurement is available, we first compute our *prior* belief:

$$Bel^-(x_t) = \int p(x_t|x_{t-1}) Bel(x_{t-1}) dx_{t-1}$$

based on the previous state and a model of the system's evolution over time, as expressed in the *propagation model* $p(x_t|x_{t-1})$. Afterwards, a straightforward application of Bayes' theorem en-

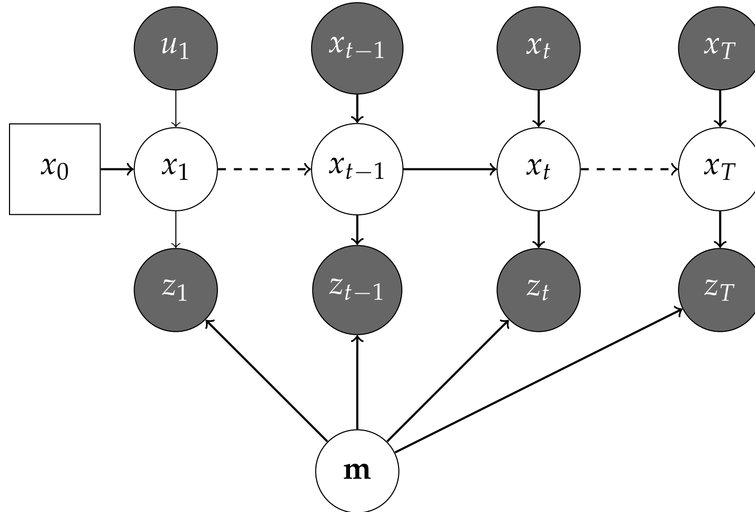


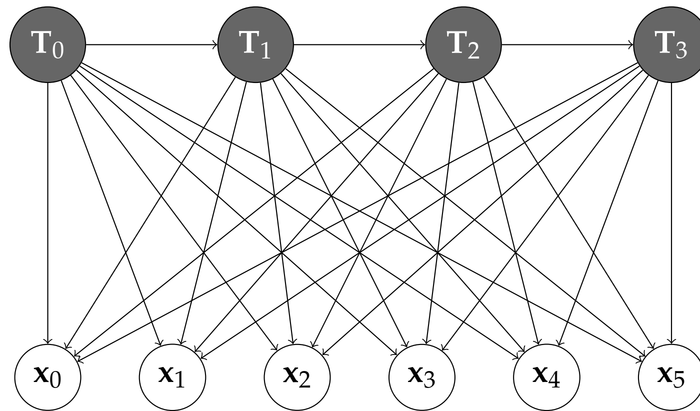
Figure 1.: A simple Bayes network for a SLAM problem

ables us to incorporate the new measurement z_t in the prior belief Bel^-.

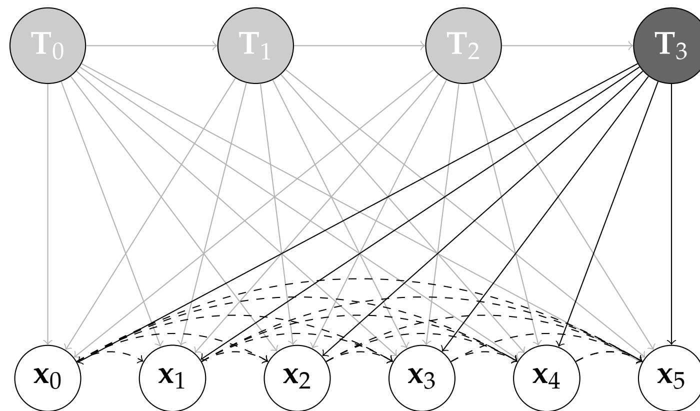
$$Bel(x_t) = \alpha p(z_t|x_t)Bel^-(x_t) \tag{1}$$

where α is just a normalizing constant. Equation 1 gives the *posterior* distribution and contains our best estimate of the system's state vector. We have used the *measurement model*, or *perceptual model*, $p(z_t|x_t)$ to define the probability of making an observation x_t given the current state x_t. The measurement model obviously depends on the type of sensor used, and captures its uncertainty characteristics.

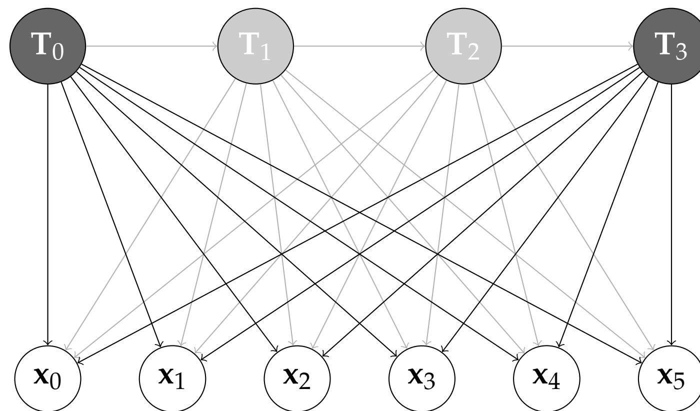
We can now link this mathematical framework to the entities in figure 1. The edges connecting nodes x_i to x_n represent the propagation (or state transition) model. We also see how the measurements z_i are influenced both by the environment map **m** and by the states x_i. In this model, the arrows pointing from the states x_i to the measurements z_i represent the measurement models $p(z_t|x_t)$. The Bayesian network thus describes how the system evolves in time and how new measurements and states interact with each other and the environment. It is also clear how performing inference on a large graph can become computationally intractable. This problem will be the subject of the next section.



(a) The full problem



(b) Filtering



(c) Keyframes

Figure 2.: Bayes networks in the case of the full problem, filtering, and smoothing approaches.

2.1.2 *Solving the Bayes network: filtering and smoothing*

This section contains a conceptual introduction to the problem of performing inference on a Bayes net to obtain the desired MAP distribution. A more exhaustive comparison of available solving methods is presented in [23], with a strong emphasis on their corresponding computational load.

We will refer to figure 2a, a specialization of figure 1 to a SLAM problem with states x_i and landmark observations T_j (odometry measurements are not pictured).

BUNDLE ADJUSTEMENT The optimal solution is called *Bundle Adjustement* (BA) and involves recomputing the MAP estimate of the full graph at each time step (i.e. every time a new measurement is added, represented by a new node in the network). While applicable offline, this approach becomes progressively hard to compute onboard as the robot spends a longer time interacting with the environment. BA is also often referred to as *smoothing* in literature. As outlined in [23], BA has to be abandoned in order to keep computational costs to a resonable level. To this end, there are several alternatives, depicted in figures 2b and 2c and described below.

FILTERING The most commonly employed solution is referred to as *filtering* and consists in marginalising out all the states but the current one, as shown in figure 2b. Since new features are added only when exploring new areas, the graph stays compact. On the other hand, the missing connections between the marginalised states and measurements are replaced by interactions between the states themselves, so that the graph becomes densely connected.

The most common implementation by far, the Extended Kalman Filter (EKF), represents all these interactions by a single covariance matrix, thus limiting the computational load. Even though their performance is satisfactory in computationally-constrained environments, EKFs do not deal well with non-linearities because past states are not kept in the filter (and re-linearization becomes less effective).

This problem can be somewhat alleviated by keeping a window of past states within the filter. Such a modified filter is usually referred to as *fixed lag smoother*, basically a conventional

Kalman filter working with an *augmented* state vector that includes delayed values of the system states. It is clear how fixed-lag smoothers are a stop-gap solution that increases the computational load without solving the underlying problem.

KEYFRAME APPROACH The smoothing-based *keyframe* approach is an alternative to filtering. Full BA is performed at each step on the simplified graph of figure 2c that only contains a strategically chosen subset of past states. The other poses and measurements are simply discarded, thus keeping the graph sparsely connected and easy to compute inference on.

INCREMENTAL SMOOTHING An innovative approach has been developed recently and has been named *incremental smoothing* by Kaess et al. in [14]. Instead of discarding or marginalising out past states, incremental smoothing takes advantage of the sparse nature of the Bayes network to solve the BA problem incrementally. Since only a small part of the graph changes at each time step, the algorithm only optimizes a small subset of the nodes, with correspondingly low computational load.

In light of the previous discussion, we have dedicated the present work to the exploration of incremental smoothing for on-board sensor fusion and localization purposes. The following section introduces the abstractions and mathematical background, with strong emphasis on the work by Dellaert et al.

2.2 INCREMENTAL SMOOTHING

As we mentioned above, incremental smoothing is a relatively recent advancement that significantly improves the performance of the Bundle Adjustement procedure by taking advantage of the intrinsic characteristics of the SLAM problem. Incremental smoothing as implemented by Kaess et al. relies on a well-known graphical model known as the *factor graph*, that we present in the next section.

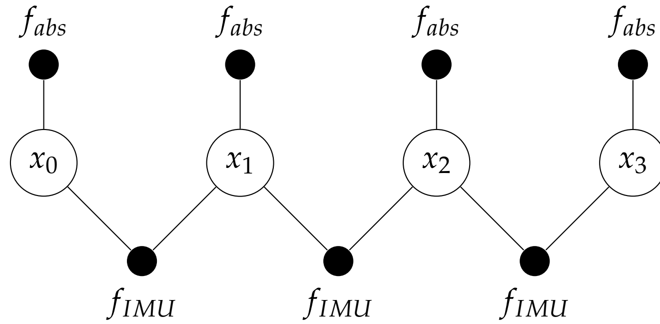


Figure 3.: A simple factor graph with unary and binary factors.

2.2.1 *Factor graphs*

What follows is a concise introduction to factor graphs and their use in the SLAM problem, based on references [11], [12], and [17].

For ease of notation, we will include the calibration parameters in the set x_i of navigation states at time t_i. Z_i represents the set of measurement carried out while the system was in state x_i. Furthermore, at a current time t_k, we denote the set of past states and measurements as $\mathcal{V}_k$ and $\mathcal{Z}_k$ respectively. The general joint probability distribution function 1 can always be factorized in terms of *a priori* knowledge and individual process and measurement models:

$$p(\mathcal{V}_k|\mathcal{Z}_k) = p(\mathcal{V}_0)\prod_i \left[p(x_i|x_{i-1}) \prod_{z_j \in Z_i} p(z_j|\mathcal{V}_i^j) \right]$$

where $\mathcal{V}_i^j$ is the set of variables appearing in the measurement model $p(z_j|\mathcal{V}_i^j)$ for measurement z_j at time t_i. The last equation naturally leads to a representation of the form:

$$p(\mathcal{V}_k) \propto \prod_i f_i(\mathcal{V}_k^i)$$

in which each factor f_i involves a different subset $\mathcal{V}_k^i$ of system states. The last equation can be graphically represented in a so-called *factor graph* with a *factor node* for each f_i and *variable nodes* that represent system states. If state variable x_j appears in factor f_i, $x_i \in \mathcal{V}_k^i$ and an *edge* connects the variable node for x_j with the factor f_i.

Figure 3 is an example of a simple factor graph that encodes different types of probabilistic information. Variable nodes are

shown in white, whereas factor nodes have a black background. In figure 3, some factors are connected to two states and thus represent the system evolution over time. Examples of this type of *binary* factors could be wheel rotation or IMU measurements. Factors that represent prior knowledge or external measurements (such as GPS) are only connected to a single variable node.

At this point, we reformulate our problem with respect to equation 1 by saying that we are actually interested in the single most probable configuration of the system states. This particular value is usually referred to as the *maximum a posteriori* (MAP) estimate:

$$\text{MAP} = \arg\max_{X} Bel(x_t) \tag{2}$$

where X denotes the set of all possible states x. In the factor graph formulation, the MAP estimate in equation 2 immediately takes the form:

$$\text{MAP} = \arg\max_{X} \prod_{i} f_i(\mathcal{V}_k^i) \tag{3}$$

Furthermore, the robotics community usually assumes that all noise distributions are Gaussian, so that we can write factors in the general form of a multivariate Gaussian distribution with the squared Mahalanobis distance $\|a\|_{\Sigma}^2 = a^T \Sigma^{-1} a$:

$$f_i(\mathcal{V}_k^i) = \exp\left(-\frac{1}{2}\|err_i(\mathcal{V}_k^i, z_i)\|_{\Sigma_i}^2\right) \tag{4}$$

where $err_i(x_i, z_i)$ is a factor-specific error function, and Σ_i is the covariance matrix. Under this assumption, we can reformulate equation 3 as an optimization problem by taking its negative logarithm:

$$\text{MAP} = \arg\max_{X} \prod_{i} f_i(\mathcal{V}_k^i) = \arg\min_{X} \sum_{i} \|err(\mathcal{V}_k^i, z_i)\|_{\Sigma_i}^2 \tag{5}$$

Each factor has now become an error term in a least squares problem, and must be minimized to obtain the MAP estimate.

Before describing how factors for different models are implemented, we turn to the algorithm for solving equation 5, with particular reference to the software we will be using throughout this thesis.

2.2.2 *Solving the factor graph*

The conventional procedure

As it stands, equation 5 can be conveniently solved with a traditional Gauss-Newton non-linear least squares solver that proceeds by repeated linearisation. Reference [7] outlines this procedure in detail. We begin by defining:

$$\mathbf{e}_i^k = err_i(\mathcal{V}_k^i, z_i) = err_i(\mathbf{x}, \mathbf{z}_i)$$

which is a function of the *full* set of state variables at time k, that we denoted by $\mathbf{x}$ for future convenience. By using the definition of a multivariate Gaussian distribution (or, equivalently, of Mahalanobis distance), we can rewrite equation 5 at time t_k as follows:

$$\text{MAP} = \arg\min \mathbf{F}(\mathbf{x}) = \arg\min_X \underbrace{\sum_i \mathbf{e}_i^T \Omega_i \mathbf{e}_i}_{F_i(\mathbf{x})}$$

where $\Omega_i = \Sigma^{-1}$ is the information matrix for factor i. We then take a first-order approximation of the error function around the guessed value $\hat{\mathbf{x}}$:

$$\mathbf{e}_i(\hat{\mathbf{x}} + \Delta\mathbf{x}) \sim \mathbf{e}_i + J_i \Delta\hat{\mathbf{x}}$$

where J_i is the Jacobian matrix of $\mathbf{e}_i$ computed in $\hat{\mathbf{x}}$. If we use this last expression in each $F_i(\mathbf{x})$, we obtain:

$$\begin{aligned}
F_i(\hat{\mathbf{x}} + \Delta\mathbf{x}) = & \\
&= (\mathbf{e}_i + J_i\Delta\mathbf{x})^T \Omega_i (\mathbf{e}_i + J_i \Delta\mathbf{x}) \\
&= \mathbf{e}_i^T + 2\mathbf{e}_j^T \Omega_i J_i \Delta\mathbf{x} + \Delta\mathbf{x}^T J_i^T \Omega_i J_i \Delta\mathbf{x} \\
&= \mathbf{c}_i + 2\mathbf{b}_i \Delta_{\mathbf{x}} + \Delta\mathbf{x}^T \mathbf{H}_i \Delta\mathbf{x}
\end{aligned}$$

By summing all the F_i together again, we obtain a linearised representation of the entire graph:

$$\begin{aligned}
F(\hat{x} + \Delta x) &= \sum_i F_i(\hat{x} + \Delta x) \\
&= c + 2b^T \Delta x + \Delta x^T H \Delta x
\end{aligned}$$

that can be minimized with respect to $\Delta\mathbf{x}$ by solving the linear system:

$$\mathbf{H}\Delta \star \mathbf{x} = -\mathbf{b} \tag{6}$$

The solution for this step of the iteration is recovered by adding the increment to the initial step:

$$\mathbf{x}\star = \hat{\mathbf{x}} + \Delta\mathbf{x}\star$$

The Gauss-Newton algorithm proceeds by iterating the linearization, solution, and update steps until convergence is accomplished. As a linearized version of the factor graph, matrix $\mathbf{H} = J_i^T \Omega_i J_i$ reflects its structure and is thus sparse. This fact greatly reduces the computation load, and equation 6 can be easily solved by Cholesky or QR matrix factorization.

Incremental smoothing

In this section, we compare the standard procedure outlined above with the *incremental smoothing* approach that was touched upon in section 2.1.2. In particular, we will be taking advantage of the *iSAM2* algorithm as presented in [14], which is heavily based upon graphical representations instead of matrix algebra. The main steps of the algorithm are outlined below.

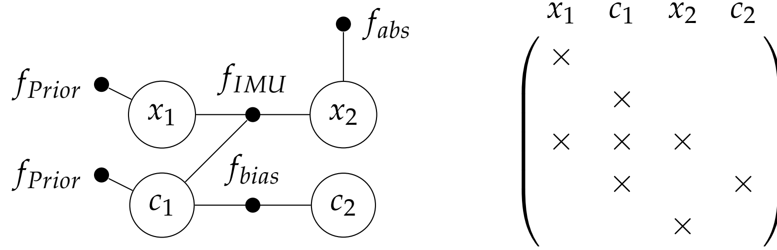
First, the factor graph is converted to a *Bayes network* such as the one in figure 1. The connection between a factor graph and its corresponding Bayes net is very direct: [14] proves how in the Gaussian case, elimination is equivalent to QR factorization of the measurement Jacobian $\mathbf{H}$.

Evidence of this fact is shown in figure 4. The simple factor graph in figure 4a produces the Jacobian shown on the right. The i-th factor appears as the i-th set of rows in the Jacobian, with the B_{ij} block being non-zero when the i-th factor involves the j-th system state.

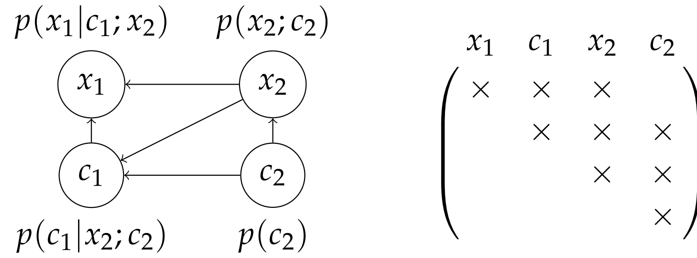
After choosing an (arbitrary) *elimination order*, the probability information embedded in the factor graph can be encoded directly in a Bayes net, as shown in 4b. The Jacobian matrix is now square (and upper-triangular), with the B_{ij} block being non-zero when $p(x_i)$ has a conditional dependece on state x_j.

It so happens that Bayes trees constructed from factor graphs are *chordal*, meaning that every cycle of four or more vertices has a chord (an edge that is not part of the cycle but connects two of its vertices). This fact enables a new graphical representation, which is the original contribution in [13].

The *Bayes tree* is a directed graph model in which each node represents a set of variables (called a clique). The Bayes tree must be variable-connected, meaning that if a variable appears

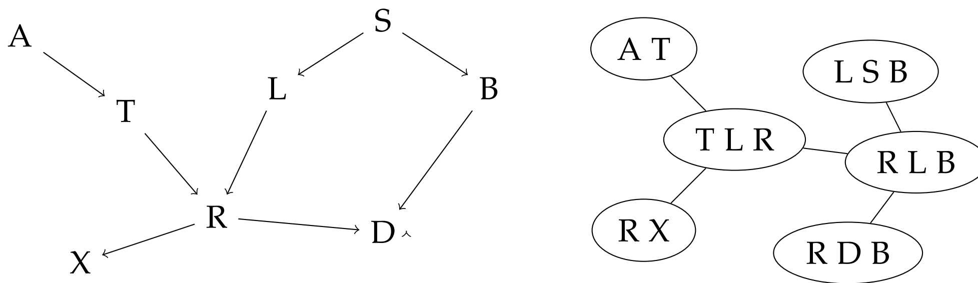


(a) Factor graph and its relative Jacobian matrix



(b) The equivalent Bayes network and its corresponding (upper triangular) Jacobian matrix.

Figure 4.: Factor graph solution process.



(a) The original Bayes network

(b) The Bayes tree version

Figure 5.: A Bayes network and its equivalent Bayes tree.

in two cliques, then it must appear in all cliques on the path between those two. A fictional example of a Bayes network with its corresponding Bayes tree is shown in figures 5a and 5b. A chordal Bayes net can be transformed to its corresponding Bayes tree by discovering its cliques through one of several algorithms, such as Maximum Cardinality Search.

The key insight that lead to the development of Bayes trees lies in the way they encode the information matrix of the factor graph. Since information can only travel "upwards" towards the root of tree, only a small part of it is affected when a new measurement is added. In particular, if a new factor $f(x_i, x_j)$ is added, only the paths between the cliques containing x_i and x_j and the root are affected. Neither the sub-tree below these cliques are affected, nor any other sub-trees not containing x_i or x_j.

Incremental additions to the Bayes tree can thus be conveniently carried out by detaching the affected part of the tree and transforming it back to a factor graph. The new measurement is added to the factor graph, which is again converted to a Bayes tree and re-attached to the unaffected part. Reference [14] has a more formal description of this process.

It is intuitively clear how the possibility to "re-use" most of the past calculations reduces the execution time and allows for a greater number of factors to be added within the envelope of real-time operation.

The GTSAM Library

This thesis makes extensive use of the tools provided by the software library GTSAM (Georgia Tech Smoothing and Mapping), developed by Kaess et al. at Georgia Tech. In the authors' own words:

> GTSAM is a library of C++ classes that implement smoothing and mapping (SAM) in robotics and vision, using factor graphs and Bayes networks as the underlying computing paradigm rather than sparse matrices.

GTSAM provides an object-oriented framework that facilitates the creation and manipulation of complex factor graphs such as those encountered in a SLAM problem. Taking advantage of the iSAM2 algorithm is also a nearly plug-and-play operation

that replaces standard solvers such as Gauss-Newton. Furthermore, the library has been designed to be easily inter-operable with other pieces of software, and can be integrated with the other components of the sensing solution.

GTSAM comes equipped with a series of factors that model common sensor types such as cameras, wheel odometry, and IMUs. A (non-exhaustive) list is reported below:

PRIOR FACTOR is the simplest factor, encoding a measurement model in which the value of the state is distributed according to the specified noise model. We will use this type of built-in factor to constrain the initial poses.

BETWEEN FACTOR provides information about the *relative* pose between two states, which is assumed distributed according to the specified noise model. We will be using a customized version of this factor for our visual odometry measurements.

RANGE FACTOR gives information about the scalar distance between two poses or a pose and a point. This factor could be useful for laser or infrared range finders.

IMU FACTOR constrains two relative poses (expressed in an inertial frame) given a series of IMU measurements (acceleration and angular velocity). We will describing this factor more extensively in the following sections.

PROJECTION FACTOR measures the projected (pixel) coordinates of the landmark in a camera's image plane given its pose. This factor is useful when using feature-based visual odometry.

Most importantly, a mechanism of class inheritance enables the user to write her own custom factors that respond to specific needs and integrate seamlessly in the rest of the library. In the present work, we have taken advantage of this possibility by writing custom factors while leveraging GTSAM optimizers and solution algorithms.

2.3 OPTIMIZATION ON NON-EUCLIDEAN SPACES

In our case, the optimization states referred to in the last section are mainly 3D pose transformations, for which some care must

be put in specifying an appropriate parametrization. The next pages explain how we carried out this process in the context of least-squares optimization. In particular, the *group* of 3D rotations is non-Euclidean, so that the concept of a *manifold* needs to be introduces.

2.3.1 *The representation of rotations*

Following the steps outlined in [21], we first define a *rotation matrix* from the inertial frame A to the body frame B as:

$$R_{ab} = [\mathbf{x}_{ab}\ \mathbf{y}_{ab}\ \mathbf{z}_{ab}]$$

where $\mathbf{x}_{ab}$, $\mathbf{x}_{ab}$, $\mathbf{x}_{ab}$ are the coordinates of the principal axes of B relative to A. We note that rotation matrices have two interesting properties: $RR^T = 1$ and $\det R = 1$. The set of all 3x3 matrices that obeys these properties is called SO(3), for *special orthogonal)*:

$$SO(3) = \{\mathbb{R} \in R^{3\times 3} : RR^T = I, detR = +1\}$$

It can be shown that SO(3) forms a group under matrix multiplication with the identity matrix as the identity element. A rotation matrix can also be interpreted as a mapping $\mathbb{R}^3 \to \mathbb{R}^3$ that rotates the coordinates of a point from frame B to frame A:

$$q_a = [\mathbf{x}_{ab}\ \mathbf{y}_{ab}\ \mathbf{z}_{ab}]q_b = R_{ab}q_b$$

The composition rule for rotation matrices is matrix multiplication:

$$R_{ac} = R_{ab}R_{bc}$$

where R_{ac} is again a map $\mathbb{R}^3 \to \mathbb{R}^3$ that rotates the coordinates of a point from frame C to frame A.

Rotations are described by an axis $\omega \in \mathbb{R}^3$ and angle θ: for this reason, we would like to write $R \in SO(3)$ as a function of ω and θ. This operation is carried out by the exponential map. It can be proved that for every $R \in SO(3)$, there exists $\omega \in \mathbb{R}^3$, $\|\omega\| = 1$ and $\theta \in \mathbb{R}$ such that $R = \exp(\hat{\omega}\theta)$, where $\hat{\omega}$ is the skew-symmetric matrix related to ω, and:

$$\exp(\hat{\omega}) = e^{\hat{\omega}\theta} = I + \theta\hat{\omega} + \frac{\theta^2}{2!}\hat{\omega}^2 + ...$$

Geometrically, θ gives the angle of rotation and $\hat{\omega}$ gives the rotation axis, in the spirit of the *equivalent axis representation*. Other

commonly used representations for rotations are Euler angles and quaternions, that will be discussed later when dealing with overparametrization of the SO(3) space.

In the case of general rigid body motion, we consider the position vector $p_{ab} \in \mathbb{R}^3$ from the origin of frame A to the origin of frame B, so that the configuration of the system can be described by the pair (p_{ab}, R_{ab}) in the space $SE(3)$ (special Euclidean):

$$SE(3) = \{(p, R) : [\in \mathbb{R}^3, R \in SO(3)\} = \mathbb{R}^3 \times SO(3)$$

By appending 1 to the coordinates of a point, we obtain a vector in $\mathbb{R}^4$ which is its *homogeneuos representation*:

$$\bar{q} = [q_1\ q_2\ q_3\ 1]^T$$

Vectors, being the difference of points, have the form:

$$\bar{v} = [v_1\ v_2\ v_3\ 0]^T$$

Using this notation, a transformation $g_{ab} \in SE(3)$ can be represented in linear form:

$$\bar{q}_a = \begin{bmatrix} q_a \\ 1 \end{bmatrix} = \begin{bmatrix} R_{ab} & p_{ab} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} q_b \\ 1 \end{bmatrix}$$

Again, composition can be carried out by straightforward matrix multiplication. We now generalize the skew-symmetric matrix $\hat{\omega}$ we introduced for rotations by defining the corresponding $se(3)$ group:

$$se(3) = \{(v, \hat{\omega}) : v \in \mathbb{R}^3, \hat{\omega} \in so(3)\} \tag{7}$$

In homogeneous coordinates, we write an element $\hat{\xi} \in se(3)$ as:

$$\hat{\xi} = \begin{bmatrix} \hat{\omega} & v \\ 0 & 0 \end{bmatrix} \in \mathbb{R}^4$$

Analogously to the exponential map for rotations, it can be shown how, given $\hat{\xi} \in se(3)$ and $\theta \in \mathbb{R}$, we have:

$$e^{\hat{\xi}\theta} \in SE(3)$$

and that every rigid transformation can be written as the exponential of some *twist* $\hat{\xi}\theta \in se(3)$. The exponential map for a twist gives the relative motion of a rigid body in the same reference frame:

$$p(\theta) = e^{\hat{\xi}\theta} p(0)$$

Least squares on a manifold

There are two main approaches to carrying out estimation problems in a non-Euclidean space such as $SO(3)$, which is the case of our UAV. The first uses a minimal representation such as Euler angles. Unfortunately, it can be proved how there exists no parametrization of $SO(3)$ in $\mathbb{R}^3$ that has no singularities. Singularities cause "gimbal-lock", and thus need to be dealt with accordingly.

The second approach makes use of overparametrized representations that have no singularities, such as quaternions in $\mathbb{R}^4$. However, this creates a new set of problems. First, the algorithm does not take into account the constraints on the representation (such as $\|q\| = 1$ for a quaternion q) and optimizes fictional degrees of freedom. Secondly, the parametrization may behave in a non-Euclidean way, in which parameters affect the rotation with varying magnitude.

The most flexible solution takes advantage of the concept of *manifolds*: even tough $SO(3)$ is not globally an Euclidean space, it can be considered as such locally. Correspondingly, a global overparametrized representation is used, whereas incremental changes are expressed in minimal coordinates that ideally behave as an Euclidean space.

To formalise this, we note that matrix multiplication is a continuous and differentiable function, as is its inverse. This means that $SE(3)$ is a differentiable manifold, and thus a *Lie group*. Even though a complete study of Lie groups is out of context here, we note that the tangent space to a Lie group at identity is a Lie algebra, which in the case of $SO(3)$ actually is $so(3)$ as given in equation 7.

This means that we can use the exponential map (and its inverse, the logarithm map) as mappings from increments in the Euclidean space to the global overparametrized space, as described in [10]. In practice, this is carried out by defining an operator $\boxplus$ that maps a local variation $\Delta\mathbf{x}$ in the Euclidean space to a variation on the manifold: $\Delta\mathbf{x} \rightarrow \mathbf{x} \boxplus \Delta\mathbf{x}$. The error function can now be defined as:

$$\breve{\mathbf{e}}_k(\Delta\tilde{\mathbf{x}}_k) = \mathbf{e}_k(\breve{\mathbf{x}}_k \boxplus \Delta\tilde{\mathbf{x}}_k)$$

where $\breve{\mathbf{x}}$ is expressed in the overparametrized representation, and $\tilde{\mathbf{x}}$ in a minimal one. A common choice in this respect is the vector part of the unit quaternion.

Operator $\boxplus$ is a convenient shorthand for a procedure that starts by converting the rotation part of the increment to a full quaternion, which is then applied to $\check{\mathbf{x}}$ together with the translation. In particular, during the Least Squares optimization, increments $\Delta\tilde{\mathbf{x}}*$ are computed in the local Euclidean space, and are then accumulated in the global non-Euclidean one:

$$\mathbf{x} = \check{\mathbf{x}} \boxplus \Delta\tilde{\mathbf{x}}*$$

Thanks to this theoretical framework, optimization in the SE3 can be carried out simply by exchanging the usual Euclidean *sum* with the $\boxplus$ operator described above.

2.4 INERTIAL NAVIGATION

Reference [11] describes two different approaches for incorporating IMU measurements in the factor graph. We will see how the second formulation enables usage of IMU measurements at faster rates by reducing the number of needed states.

CONVENTIONAL APPROACH In the conventional approach, each IMU measurement corresponds to a factor that relates the navigation states at two different times x_k and x_{k+1}. Such relation embodies the non-linear time-propagation of states in the form:

$$x_{k+1} = h(x_k, c_k, z_k^{IMU})$$

where z_k^{IMU} is the set of IMU measurements (acceleration and angular velocity) and c_k is the set of parameters for the IMU error model. In practice, c is estimated along with the navigation states and contains the biases for the gyroscope and accelerometer. Its time propagation model is usually assumed to be a random walk with zero mean. The information required for this assumption is taken from the manufacturers' specification of the IMU. It should also be noted that the navigation pose x_k must be given in an inertial frame of reference to correctly subtract the gravity vector.

Formally, the IMU measurement factor is the error function:

$$f^{IMU}(x_{k+1}, x_k, c_k) = d(x_{k+1} - h(x_k, c_k, z_k))$$

In this approach, a new navigation state x_k must be added every time a new IMU measurement is available. Since the

IMU is sampled at very high rates (up to 100Hz), such a procedure quickly becomes computationally intractable due to the high number of states and factors involved. The next paragraph presents an alternative solution that alleviates this problem.

EQUIVALENT IMU FACTOR Lupton et al. ([19]) developed an innovative inertial navigation algorithm that is especially suited for factor graph based smoothing. The conventional approach to inertial navigation involves carrying out the integration of IMU measurements in the inertial (navigation) frame.

However, this solution is inconvenient because the integration must be repeated every time the factor graph is re-linearized during the solving process. On the other hand, [19] suggests computing a *pre-integrated delta* $\Delta x_{i \to j}$ in the *body* frame instead. Since such precomputed increment depends linearly on the initial navigation pose, its calculation must only be carried out once.

This second approach is extremely advantageous in the current case, since visual odometry will be available at most at camera rates. It would then be wasteful to add new navigation states between one VO estimate and the next just to keep up with the IMU. By using the precomputed increment, the IMU measurements are "accumulated" and integrated until a new VO estimate is available. Once it is ready, the *equivalent IMU factor* is added to the graph as follows:

$$F^{Equiv}(x_j, x_i, c_i) = d(x_j - h^{Equiv}(x_i, c_i, \Delta_{x_i \to x_j}))$$

Since the IMU factor is used within the optimization framework, there is no need to use sophisticated integration algorithms: a simple Euler method is enough to constrain the relative pose between two consecutive system states. GTSAM ships with an implementation of [19] as a *CombinedIMUFactor* that will be used throughout the rest of the current work.

More specifically, we decided to add the bias factors c_k at camera rate, just like the navigation poses x_k, even though the dynamics of the bias evolution are much slower than the robot's motion.

3

DENSE VISION ODOMETRY

Visual Odometry is a technique for estimating the 3D motion of the robot using a sequence of images captured by on board cameras. More specifically, the motion of the robot is computed incrementally, as a difference between two consecutive images. Historically, Visual Odometry has successfully replaced wheel measurements as far away as Mars in the 2004 JPL mission.

There are two main approaches to compute the relative motion of the robot between consecutive image frames. *Feature-based* methods work by tracking salient features across camera frames, and have successfully been employed in the past due to their relatively low computational requirements. Implementations of visual-intertial sensor fusion in literature are mostly based on this technology. We refer the reader to Weiss ([27]) and Engel et al. ([5]) that employ the well-known PTAM algorithm on Unmanned Aerial Vehicles. The workflow for feature-based VO is as follows. First, a so-called *detector*, such as Harris or FAST, extracts a set of salient features. Secondly, features in the current image are linked to their correspondent in the past image. As a last step, the relative pose between the two images is found by minimizing the reprojection error within each pair of features.

The main drawbacks of feature-based methods lie in their low accuracy and many intermediate steps that require fine tuning. Thanks to the recent advancements in computing power, a second class of algorithms has emerged. *Dense* visual odometry works with all pixels in the original image, and computes the relative pose transform by minimizing the photometric error between each pair of corresponding pixels. Dense methods have reached a sufficient level of maturity and performance when implemented on GPUs (see [22]). Luckily, less computationally heavy implementations have started to appear that are more

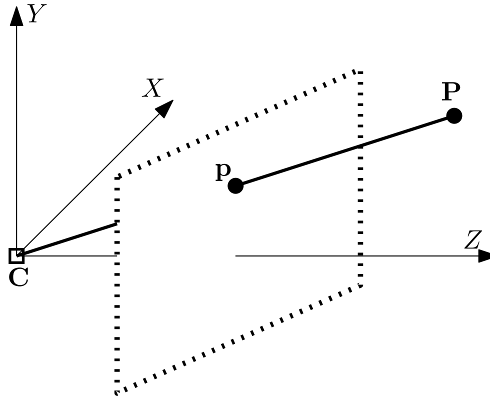


Figure 6.: The pinhole camera model

readily deployed within the limited resources of an Unmanned Aerial Vehicle.

In this respect, the present thesis is based on Kerl's work (see [15]). His implementation runs in real time on consumer CPUs, while retaining the accuracy and robustness that make dense VO a compelling choice. We will be taking advantage both of his code and theoretical framework. A more complete theoretical analysis of dense vision odometry concepts can be found in [2].

3.1 THE PINHOLE CAMERA MODEL

In the context of computer vision, a camera is a mapping π from 3D to 2D coordinates:

$$\pi : \mathbb{R}^3 \to \mathbb{R}^2$$

The most commonly used model is very simple: the whole camera system is represented by a single hole through which all light rays pass before forming the image on the sensor. The projection of a generic 3D point $\mathbf{X}$ is found by extending the line from $\mathbf{X}$ to the optical center $\mathbf{C}$ until it intersects the image plane in point $\mathbf{p}$, as shown in figure 6. The distance from the optical center to the image plane is referred to as the focal length f.

Given a point $\mathbf{X} = (X, Y, Z)$ in the camera reference frame, we denote the coordinates of its projection by $\mathbf{x} = (x, y)$. With simple geometrical optics relations, we obtain:

$$x = \frac{Xf_x}{Z} + o_x \qquad y = \frac{Yf_y}{Z} + o_y \tag{8}$$

where f_x and f_y are the equivalent focal lenghts in the x and y directions due to a possibly non-square sensor. In this simple pinhole model, the focal length is exactly the distance between the focal plane and the pinhole. The other constants o_x and o_y take into account the fact that the origin of the image plane is not coincident with the optical center. The set $\{f_x, f_y, o_x, o_y\}$ of *camera intrinsic parameters* must be estimated during the calibration process.

Projection of physical points to the camera plane can be expressed easily in the *projective space* by augmenting the pixel coordinate $\mathbf{x}$ with an additional component to make it an $\mathcal{R}^3$ vector. In this projective space, points are understood to be coincident if their corresponding vectors are proportional. With this notation, the projection function in 8 can be rewritten in matrix form:

$$q = MQ \qquad q = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \quad M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \quad Q = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

It should also be noted that the pinhole model is linear: further effects such as lens distorsion are disregarded. However, there exist several algorithms for preprocessing sensor data so that the pinhole model can still be applied successfully. In particular, optical distorsions of the lens can be accounted for with quadratic or cubic calibration functions.

Usually, camera calibration parameters are estimated with the help of special markers ("chessboards"). In our case, however, the VI sensor is already factory-calibrated.

3.2 THE ALGORITHM

Dense methods for visual odometry rely on the Lambertian assumption that the radiance of a surface does not change with the angle of observation. In practice, one assumes that the pixel to which a given 3D point is projected has the same intensity regardless of camera position. Mathematically, one defines a *warping function* $\tau(\xi, \mathbf{x})$ that maps a pixel $\mathbf{x} \in \mathbb{R}^2$ from the first image to the second image given the camera motion twist

$\xi \in \mathbb{R}^6$. With this formalism, the photo-consistency assumption for each pixel $\mathbf{x}$ can be rewritten as:

$$I_1(\mathbf{x}) = I_2(\tau(\xi, \mathbf{x})) \tag{9}$$

To express the warping function, we first find the 3D point $\mathbf{P} = (X, Y, Z)$ corresponding to pixel $\mathbf{p} = (x, y)$. To do this, we invert the projection function in 8 and obtain:

$$\begin{aligned} \mathbf{P} &= \pi^{-}1(\mathbf{p}, Z) \\ &= Z\left(\frac{x + o_x}{f_x}, \frac{y + o_y}{f_y}, 1)^T\right) \end{aligned}$$

Let $T(\xi, \mathbf{P})$ denote the coordinates of $\mathbf{P}$ after a camera motion with twist ξ. The full warping function is then given by:

$$\begin{aligned} \tau(\xi, \mathbf{p}) &= \pi(T(\xi, \mathbf{P})) \\ &= \pi(T(\xi, \pi^{-1}(\mathbf{x}, Z))) \end{aligned}$$

Looking at equation 9, we can define the *residual* of the i-th pixel:

$$r_i(\xi) = I_2(\tau(\xi, \mathbf{x}_i)) - I_1(\mathbf{x}_i) \tag{10}$$

We now formulate the image tracking process as a least-squares problem, by assuming that the residuals for each pixel are independent and identically distributed according to the *sensor model* $p(r_i|\xi)$. Under these assumptions, the probability distributions of each pixel can be multiplied together to obtain the probability of observing the whole residual image $\mathbf{r}$ given the pose ξ:

$$p(\mathbf{r}|\xi) = \prod_i p(r_i|\xi)$$

Following a procedure similar to that outlined for the solution of factor graphs, we use Bayes' theorem:

$$p(\xi|\mathbf{r}) = \frac{p(\mathbf{r}|\xi)|p(\xi)}{p(\mathbf{r})}$$

Our goal is finding the MAP estimate:

$$\xi^{\star} = \arg\max_{\xi} p(\xi|\mathbf{r}) = \arg\max_{\xi} \prod_i p(r_i|\xi)p(\xi)$$

by setting to zero the derivative of the log likelihood:

$$\frac{\partial r_i}{\partial \xi} w(r_i) r_i = 0 \qquad w(r_i) = \frac{\partial \log p(r_i)}{\partial r_i} \frac{1}{r_i} \tag{11}$$

The last equation corresponds to the following least-squares problem:

$$\xi_{MAP} = \arg\min_{\xi} \sum_i (r_i(\xi))^2 \tag{12}$$

3.2.1 *Linearization of the problem*

Since the residuals are non-linear in the pose coordinates, equation 12 must be linearized. The process is analogous to that outlined in section 2.2.2. We start with a first-order aproximation of the i-th residual:

$$r_{lin}(\xi, \mathbf{x}_i) = r(\mathbf{0}, \mathbf{x}_i) + J_i \Delta\xi \tag{13}$$

where J_i is a 6-dimensional vector Jacobian of the i-th residual with respect to the minimal representation of the relative pose between the two images.

With the linearized residuals, equation 11 can be rewritten in matrix form with the (diagonal) weighting matrix:

$$J^T W J \Delta\xi = -J^T W \mathbf{r}(0) \tag{14}$$

Here, J is the $n \times 6$ Jacobian matrix with the n stacked J_i Jacobians of each pixel, and $\mathbf{r}$ is the vector of residuals.

Inspection of the equations shows that both sides of the equations can be computed pixel-by-pixel. In fact, this is how Kerl's implementation proceeds, without storing the complete J matrix in memory. A simple 6-dimensional linear system is built incrementally:

$$\mathrm{A}\ \Delta\xi = \mathbf{b} \tag{15}$$

and solved by Cholesky decomposition.

As we have seen in the previous chapter, analytical computation of the pixel-by-pixel error Jacobian $J_i(\xi_k)$ is required for the optimization to perform well. We consider the i-th pixel at $\mathbf{x}_i$ in the first image, and its corresponding 3D point $\mathbf{p}_i = (x, y, z) = \pi^{-1}(\mathbf{x}_i,\ Z_1^{-1}(\mathbf{x}_i))$. Through the rigid transformation g described by ξ, $\mathbf{p}_i$ is transformed to $\mathbf{p} = (x', y', z') = g(\xi, \mathbf{p}_i)$.

Looking at equation 10, we see that we can use the chain rule to compose the pixel-by-pixel image derivative with the Jacobian of the warping function:

$$J_i(\xi_k) = J_I J_w$$

where J_I is the 1×2 Jacobian matrix of the image derivatives in the x and y directions (of the *second* image):

$$J_I = \frac{\partial I_2(\mathbf{x})}{\partial \pi}$$

The warp function Jacobian J_w can be decomposed in two parts, a 2×3 J_π matrix for the projection function π and a 3×6 Jacobian of the transformed point with respect to the minimal representation of the relative pose of the two cameras:

$$J_w = J_\pi J_g$$

where:

$$J_\pi = \frac{\partial \pi(\mathbf{p})}{\partial \mathbf{p}}\bigg|_{\mathbf{p}=g(\xi,\mathbf{p}_i)} = \begin{pmatrix} f_x \frac{1}{z'} & 0 & -f_x \frac{x'}{z'} \\ 0 & f_x \frac{1}{z'} & -f_y \frac{y'}{z'} \end{pmatrix}$$

Note that the derivative of the projection function π is evaluated at the *transformed* point $\mathbf{p}$. The other derivative,

$$J_g = \frac{\partial g(\xi, \mathbf{p}_i)}{\partial \xi}$$

can be found in the literature about Lie algebras.

As reported in [2], there exist different approaches for linearization of the problem, which differ in computational efficiency but are otherwise mathematically equivalent. We follow the classification in [1] and highlight the difference between the *additive* and *compositional* approaches. In the additive approach, we solve iteratively for increments $\Delta\xi$ to the parameters ξ:

$$r_i(\xi_{k+1}) = I_2(w(\xi_k \boxplus \Delta\xi)) - I_1(x_i)$$

The last equation is in fact the straightforwards product of our past discussion, but has the drawback that both the image gradient and the Jacobians depend on the current estimate ξ_k of the parameters.

Instead of solving for an increment to the parameters, we can solve for an incremental *warp* instead. This is the core concept of the *compositional* approach:

$$r_i(\xi_{k+1}) = I_2(w(\xi_k, w(\Delta\xi, x_i))) - I_1(x_i)$$

Since the Taylor expansion of the warping function with respect to $\Delta\xi$ is carried out at identity, the Jacobian can be computed only once, with obvious computation savings.

As the Jacobian depends on ξ, both approaches require its expensive re-computation for each image. Reference [1] builds on the seminal work in [8] and introduces the *inverse compositional* algorithm, which is much more efficient. The key insight in [8] is switching the roles of the reference and current image, so that the Jacobian can be pre-computed and does not need to be updated at each iteration.

3.2.2 *Image pyramids*

Figure 7 plots the sum of the residual errors over all the pixels as a function of translation in the x direction. In practice, we took an image, warped it to account for the translation, and then computed the residuals. Figure 7 shows three cases in which the original camera frame is downsampled once, twice, or three times, respectively corresponding to 320×240, 160×120 and 80×60 final images.

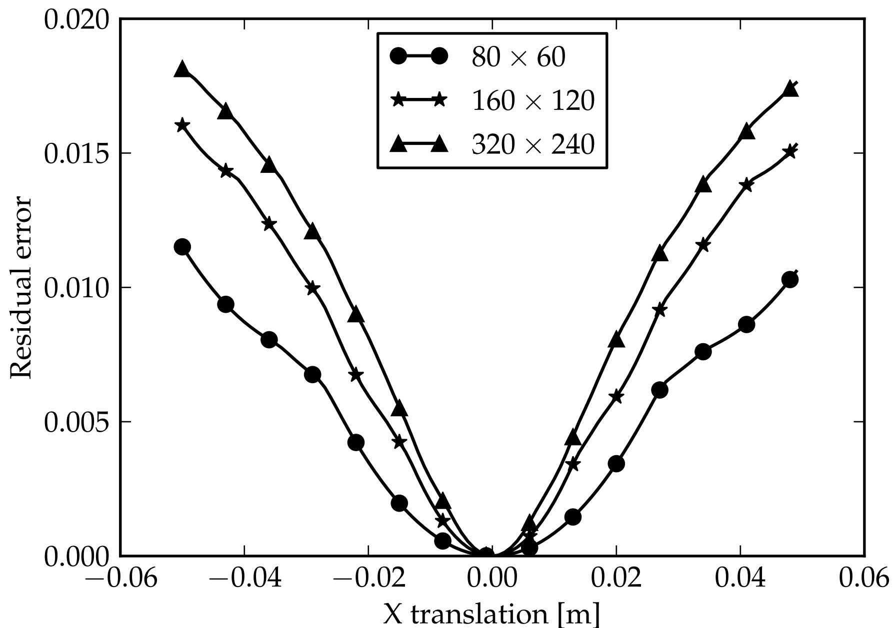


Figure 7.: Residual with respect to translation of the image in the X direction

While smallest images have a less distinct minimum (leading to lower accuracy), the convex area close to the minimum is larger. This suggests the possibility of building a *pyramid* of downsampled images to speed up processing. A rough estimate of the minimum point could be computed on the smaller

frames, improving speed and likelihood of convergence (thanks to the gentler shape of the function). Afterwards, optimization on the bigger images would obtain a more precise estimate thanks to the stepper gradient, at the expense of computation time.

3.3 using a stereo camera

Kerl's original work (see [15]) is intended for use with a RGBD camera such as the Microsoft Kinect, and is thus not directly applicable to our flight setup with a RGB stereo camera (see section 3.4.1 for more details about the hardware we are using). As we have seen earlier, the dense approach requires the knowledge of the depth of every pixel of the image. While an RGBD camera naturally provides this information at every frame, the image pair shot by our stereo setup requires further processing.

Since such processing is quite computationally intensive, our current implementation does not extract the depth information of every image, but routinely selects *keyframes* to which the stereo pipeline is applied. Each image is then tracked to the currently active keyframe, rather than to the immediately preceding image as in Kerl's original implementation. Obviously, a keyframe selection algorithm had to be developed. We will explore this challenge in more detail in section 4.4.

The stereo processing pipeline is based on the widely-used OpenCV library, and consists of the following steps:

rectification of the images This process makes use of the known calibration between the cameras to match the corresponding rows of pixels in the two images. The result corresponds to a particular arrangement of the cameras, known as *frontal parallel*, meaning that the optical planes of the two cameras are coplanar and the optical axes are parallel. Aligning the image rows improves the speed and reliability of the following steps.

stereo correspondence consists in matching a single 3D point to both images. We used OpenCV's *block match* function that exploits the presence of texture in the images. This means that the number of pixels for which depth can be computed strongly depends on the type of scene. Strongly-textured images will have many more

pixels matched, and thus convey more depth information. On the other hand, poorly-textured images will have "holes" in their depth image due to matching failures.

Rectification proves useful at this stage because matching points are located in the same row in both images. The output of this step is a *disparity image* that contains the $x^r - x^l$ difference between the pixel position in the two images.

TRIANGULATION Since the baseline between the two cameras is known, the disparity data can be triangulated to obtain depth information for each pixel (as shown in figure 8).

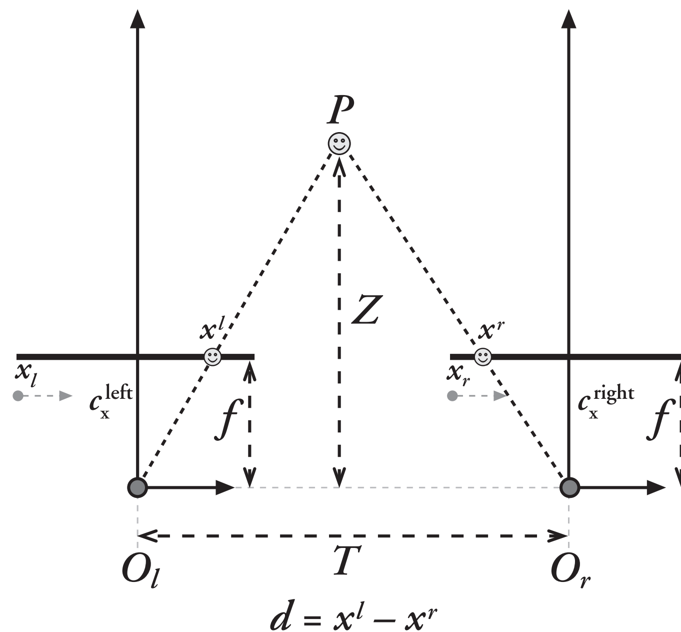


Figure 8.: Once the camera baseline is known, disparity can be used to compute the pixel depth.

The final output of this process is a *depth map* of the image that contains the depth of each pixel of the image. Depending on the camera baseline and on the presence of texture in the images, some pixels will not be able to be associated with their depth. In this case, they are marked as invalid and not used in the optimization process.

As noted earlier, due to the way the problem is linearised, depth information is only required for the keyframe, not for all individual images. Besides reducing the computation cost,

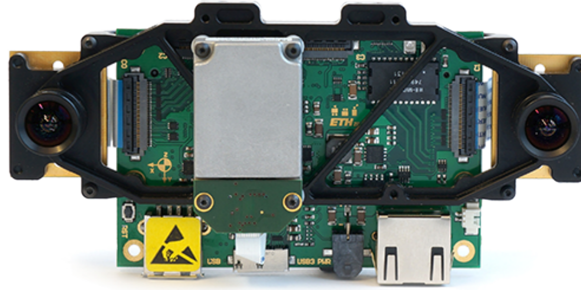


Figure 9.: The VI-sensor platform, with twin cameras and IMU.

this also opens up the possibility to use a monocular camera together with an initial 3D map of the environment. For more information on the depth extraction procedure, one can refer to [3].

3.4 HARDWARE AND OTHER TECHNOLOGIES USED

3.4.1 *The Visual-Inertial sensor*

Experimental data has been acquired with a visual-inertial platform by Skybotix AG, the *VI-sensor*. The *VI-sensor* is a self-contained embedded computer with ARM CPU and Artix FPGA, twin 752 x 480 cameras with 110mm stereo baseline and a tactical grade Analog Devices IMU.

With the aid of ROS drivers and a Gigabit Ethernet connection to the host PC, hardware timestamped camera images and IMU readings are available for further processing. The cameras operate at 30Hz, while the IMU is sampled at 200Hz. Other salient features include factory calibration of the cameras (intrinsics and extrinsics) and IMU (sensitivity, axis misalignment, and bias).

3.4.2 *The development platform*

By leveraging widely-used software packages, the resulting software implementation will be entirely platform-independent. In spite of this consideration, we aimed at a particular running target especially designed for UAV use, the *Asctec Mastermind*.

The Mastermind is a lightweight development board featuring an Intel Core 2 Duo processor, 4GB of RAM, and a Gigabit

Ethernet interface for connection with the VI sensor. While weighting less than 300g, the Mastermind offers desktop-class interface and thus the ability to use a familiar development environment.

In particular, the x86 instruction set requires no cross compilation of binaries, so that development can be carried out on a standard PC with Ubuntu Linux. Furthermore, the Core 2 Duo CPU supports modern instruction sets such as SSE and MMX that speed up vector operations. Kerl's implementation of the VO algorithm already includes such optimizations on compute-heavy operations such as image warping.

3.4.3 *ROS*

ROS (Robot Operating System) is a widely used middleware platform for robot software development. It provides libraries and well-tested code for hardware abstraction (device drivers), message-passing and other commonly used functionality.

More specifically, ROS provides a series of building blocks for complex robotics applications. Computations, such as our state estimation routines, can be implemented within a ROS "node" that receives messages from device drivers (such as cameras and IMU). The results can then easily be recorded or loaded in a plotting application.

One useful ROS feature that has been used extensively in the present work is the ability to record a flight dataset containing all sensor data and camera frames, to be played back later with accurate timing. Thanks to this convenience, algorithm development has been carried out iteratively with the same known input data. Recorded inputs can also be played back at slower or higher rates to test the execution time of the algorithm. This has been useful when implementing slower versions of the state estimator, that could be still run at slower-than-realtime speeds to benchmark their accuracy.

In pratice, most of the software described througout this thesis is developed as a ROS *node* that receives the camera images and the IMU measurements and outputs the estimated pose of the UAV. Evaluation of the software is also made easier by the availability of visualization tools such as *rviz* that show a 3D representation of the coordinate transforms available in the system.

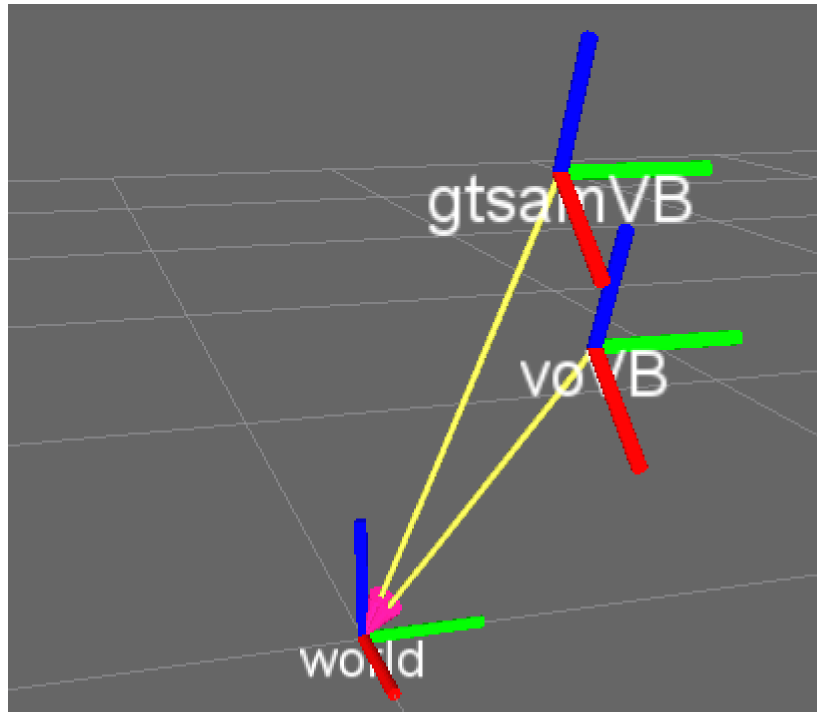


Figure 10.: Rviz interactive 3D pose visualization tool.

An example of the typical *rviz* screen is reproduced in figure 10, with two 3D poses shown as a set of Cartesian axes with respect to the world frame.

4

THE LOOSE LOOP APPROACH

This chapter is a first exploration of the feasibility of using the GTSAM library to implement a smoother for Visual Odometry and IMU data. Section 4.1 starts out by using the smoother to combine ground truth data from an external source with the IMU. After validation of the smoother performance, we start using VO estimates in place of the ground truth data.

In this chapter, the results of Kerl's VO framework are used *after* the optimization is completed, that is, as a pose estimates similar as what could be obtained with a GPS or a range sensor. In the next chapter, we will develop the framework for a tighter integration between the two systems.

4.1 INERTIAL NAVIGATION WITH GROUND DATA

As a first step, we investigated using GTSAM to smooth IMU measurements together with a ground truth source, namely a VICON motion capture system. The VICON motion capture system is a state-of-the-art infrared tracking system that can track 3D displacements with millimeter resolution. The procedure involves fitting infrared reflectors to the quadrocopter body, so that IR cameras can track its indoor flight within a room.

Let us attach a frame of reference B to the Vicon markers on the quadrocopter. IMU measurements are produced in the C frame (the body frame of the robot).

The GTSAM library comes with several built-in factors, including one to model relative pose measurements between two optimization states which comes close to be useful in the current situation. However, in the current case, the C and B frames are distinct, and thus a custom factor had to be developed.

The algorithms for inertial navigation involve removing the effect of Earth gravity on the accelerometer measurements and thus require the knowledge of the position of the body frame in an inertial and gravity aligned frame of reference. We denote such frame with the letter I and observe that the Vicon reference frame satisfies the afore-mentioned attributes.

With this considerations in mind, the natural choice for the set of system states was:

${}^{c}_{i}T$	Pose of the body frame (C) with respect to the inertial frame (I)
${}^{b}_{c}T$	Pose of the Vicon markers frame (B) with respect to the body frame (C)

Each VICON data point is thus a measurement of pose ${}^{b}_{i}T$ used by the smoother to update the system states.

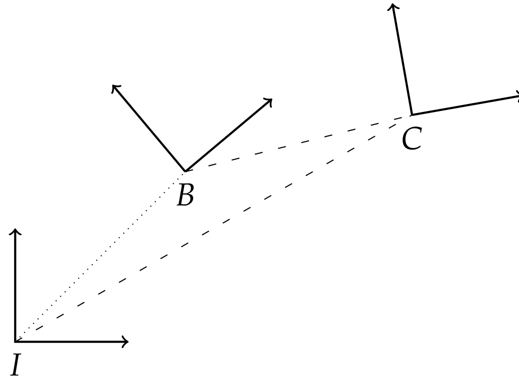


Figure 11.: Coordinates frames as described in section 4.1.

The first step for creating a custom factor is defining a measurement model as described in section 2.1.1. For our factor, this is a simple composition of poses to obtain ${}^{b}_{i}T$ as expected given the current system states:

$$ {}^{b}_{i}T = {}^{c}_{i}T \, {}^{b}_{c}T \tag{16} $$

This measurement model is implemented thanks to GTSAM's built-in handling of Lie algebras for SE3 pose compositions.

To complete the definition of a custom factor that could be used for optimization, the Jacobians of the error with respect to the states must also be defined (in our case, a Jacobian with respect to ${}^{i}_{c}T$ and one with respect to ${}^{c}_{b}T$). As always, these must be expressed in terms of the minimal representation of the SE3

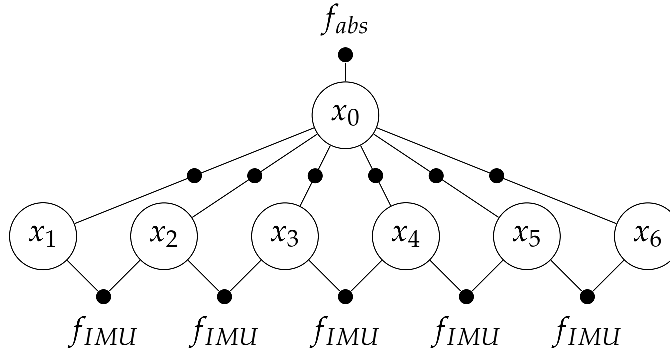


Figure 12.: Factor graph with absolute visual odometry measurements.

Lie algebra, as linear compositions of the *generators* of the Lie algebra.

Now that we have shown how the custom factor is implemented, we turn to the description of the execution flow. Each time a new Vicon measurement is ready, we define a set of new system states, corresponding to the poses at that time instant. Afterwards, a new factor is created linking these new states with the reference, as shown in the structure in figure 12.

At the same time, IMU measurements relative to the time interval between the current and the past Vicon measurement are accumulated and added to GTSAM's built-in *CombinedIMU* factor.

Both factors are then added to the factor graph. An initial guess for the new states is also needed, which is found by interpolation of the old states. Optimization is carried out incrementally on the existing graph, to which new states and factors are added at each iteration.

Since sensor fusion is carried out using a smoother, all past states are retained. This allows for a more accurate estimate of the itinerary to be retrieved at the end, once all the measurements are known. For example, plot 13 shows both the on-line and off-line results for the X component of the accelerometer bias. It is clear how the knowledge of all measurements makes the off-line results more accurate and stable compared to the on-line estimate. In fact, the offline estimate of the bias has very small variations, validating the good performance of the IMU hardware. With this evidence in mind, one could further optimize the performance of the system by reducing the number of bias states included in the factor graph. In the current

implementation, a new one is added for each Vicon measurement, but this number could be reduced very much in light of the very slow dynamics.

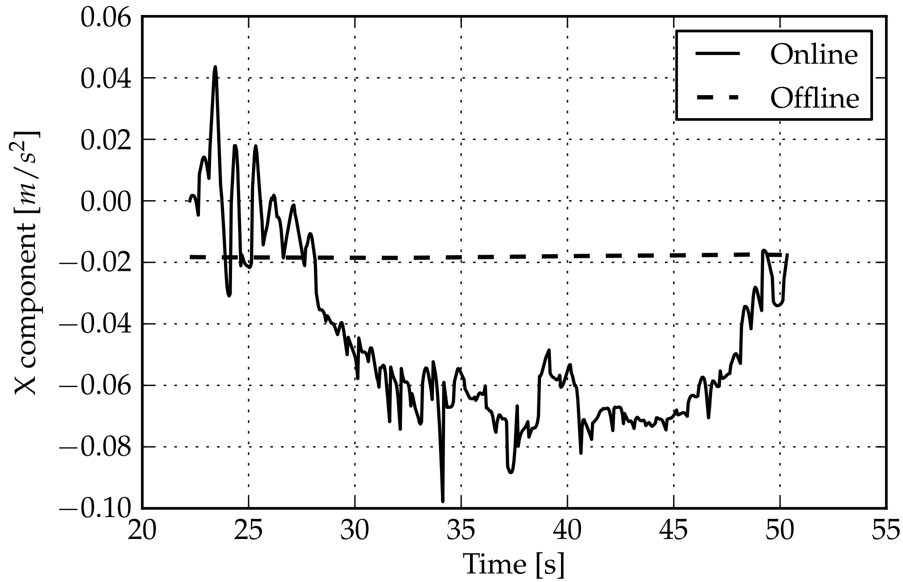


Figure 13.: Online and offline estimations of the accelerometer bias.

4.1.1 *The measurement noise models*

With reference to equation 4, a successful optimization of the factor graph requires that the noise models of the measurements be specified correctly. In the current case, we have to introduce two 6-dimensional noise models, one for the IMU measurements and one for the Lie algebra minimal representation of the pose. A further noise model is required for the accelerometer and gyroscope biases evolution.

The noise models are introduced in the form of a covariance matrix, so that a diagonal or other shapes can be used. In addition, Gaussian or other distributions are available.

The IMU noise model can be obtained from the manufacturer's data sheet. The same holds for the Vicon ground truth data we used in this section, with the additional fact that the system is generally very reliable thanks to the high number of cameras employed. Furthermore, the errors can be considered isotropic in nature, thanks to the random position of the cameras and the on-body reflectors.

The noise model for the VO estimate is more problematic, and will be discussed in the next section.

4.1.2 *Extending functionality: gravity calibration*

In the previous section we described how adding an additional pose to the system states could allow us to optimize the relative pose between the body frame and the Vicon markers. A natural extension of this configuration would allow for arbitrary initial orientation between the Vicon gravity and a gravity aligned inertial frame. This will be useful when using the framework with a visual estimator whose initial pose is not guaranteed to be gravity aligned at all.

In practice, we add a new frame of reference V that corresponds to the first Vicon pose, so that all subsequent measurements are relative to it. Figure 14 shows this setup, which will be generalized in the following section to account for visual odometry measurements whose orientation with respect to gravity is unknown.

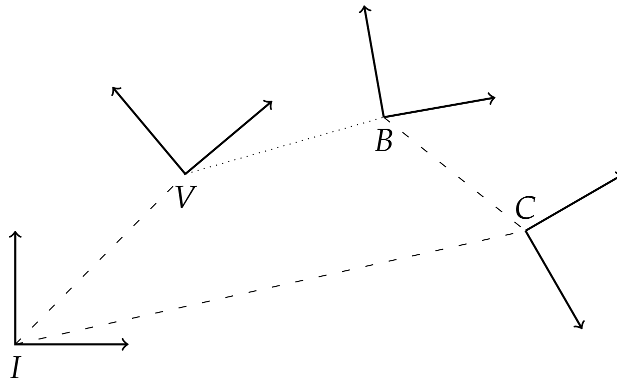


Figure 14.: Coordinates frames as described in section 4.1.2.

If the robot is kept stationary for a brief period of time in the start-up phase, the relative *orientation* between the V and I frames can be estimated by transforming the gravity vector as measured by the IMU to its known expression $[0, 0, -9.81m/s^2]$ in the inertial frame. Since the translation part of pose ${}^{i}_{v}T$ is arbitrary, we decided to have coincident origins for the V and I origins.

The measurement model in equation 16 must be modified to account for the additional system state, and becomes:

$$ {}^{b}_{v}T = {}^{i}_{v}T \, {}^{c}_{i}T \, {}^{b}_{c}T $$

Accordingly, the chain rule is used to compute the Jacobian of the complete composition with respect to each of the states.

Estimation of the calibration pose has proven resilient to mis-initialization, and also quite stable during the flight. Figure 15 compares the online result to the offline estimation which is extracted after the run is complete and the smoother has processed all data.

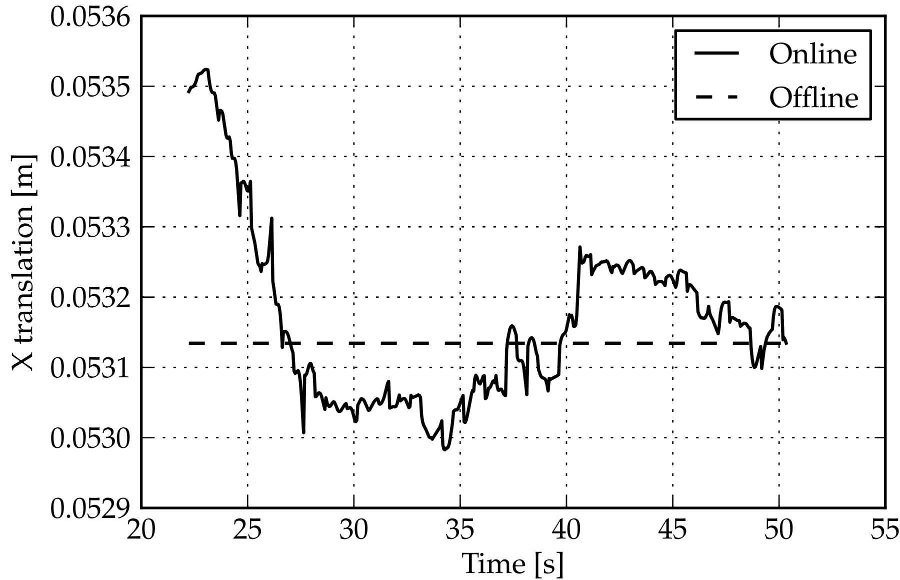


Figure 15.: Online and offline estimations of the Body-Camera frames relative pose.

It is easy to see how stable the pose estimation ends up to be, with accuracy in the mm range. For the sake of future plug-and-play applications, we also evaluated the smoother's ability to converge the calibration state CB without an accurate initialization. To this end, the initial prior was configured with a value incorrect by about $2cm$ in all axes. Such distance deliberately represents quite a substantial difference for an UAV which is only about $50cm$ wide. We also correspondingly decreased the confidence on the initial prior (increasing the σ on the corresponding noise model). Results for this test are shown in figure 16 created with a sub-minute flight during which the calibration pose converged nonetheless.

It should also be noted that the translation part of the CB pose is intrinsically harder to observe, so that the afore-described behaviour improves when considering the rotation part.

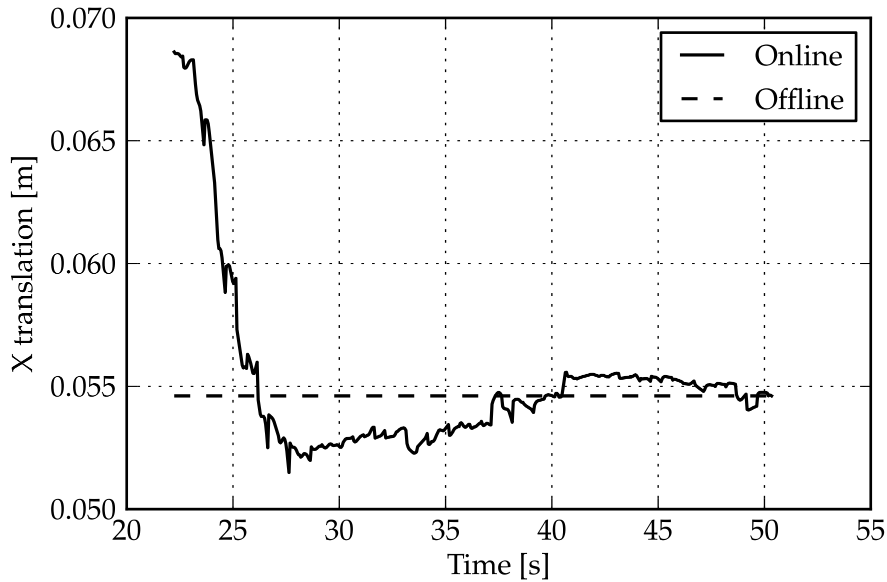


Figure 16.: Online and offline estimates of the CB pose with inaccurate prior knowledge.

4.2 SMOOTHING FOR VISUAL ODOMETRY

The real power of a smoothing approach comes through when we try to use it with actual visual odometry data. Until now, we have been using absolute pose measurements like those produced by the ground truth system. However, as described in the previous chapter, visual odometry instead tracks two images to each other. The concept of an absolute frame of reference is thus not applicable in this case.

This problem can be tackled by exploiting the flexibility of the smoothing approach. Since measurement factors can reference any arbitrary set of system states, the factor graph's topology can replicate the inner working of the DVO algorithm. To this end, we identified two alternative approaches, described in the following paragraphs.

ADDING THE KEYFRAME POSES AS STATES The first approach is a natural evolution of the implementation outlined for gravity calibration in section 4.1.2. During the operation of the Visual Odometry algorithm, additional states (such as K_i in figure 17) are added for each of the keyframes.

As illustrated in figure 17, the i-th keyframe is represented by an additional frame of reference K_i with a corresponding pose ${}^{b}_{ki}T$ from the VO reference frame V. Visual odometry measure-

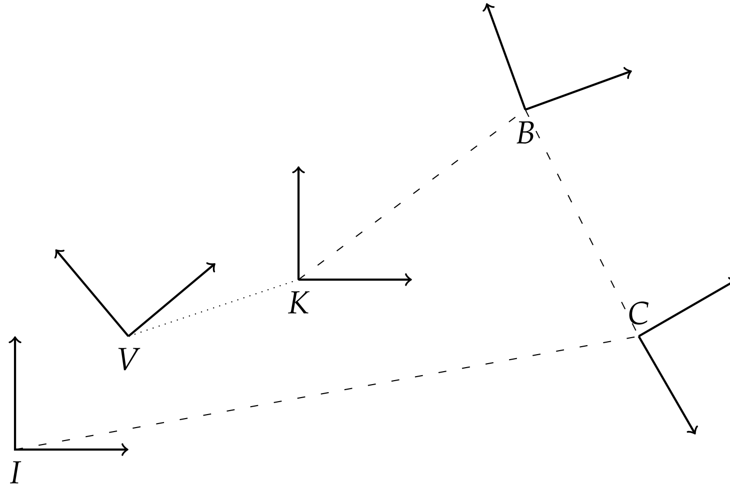


Figure 17.: Coordinates frames as described in section 4.2.

ments are then of the form ${}^{b}_{ki}T$, where K_i is the current (active) keyframe. Accordingly, the measurement model must be updated as follows:

$$ {}^{b}_{ki}T = {}^{v}_{ki}T\, {}^{i}_{v}T\, {}^{c}_{i}T\, {}^{b}_{c}T $$

The choice of storing keyframe poses as additional states in the smoother completely changes the graph structure, as can be seen by comparing figure 12 with 19a.

Thanks to the incremental nature of GTSAM's algorithms, adding these additional states does not incur a significant performance penalty. On the other hand, this allows the smoother to optimize over the set of all past keyframes, improving reliability and accuracy.

The advantage of this approach is its ability to optimize over the calibration poses VI and CB, since those are part of the measurement "chain". On the other hand, some system states are duplicated, since new ones are added for each keyframes. Even though this has not produced any problem during operation, an alternative is desiderable from a consistency point of view.

USING RELATIVE MEASUREMENTS WITHOUT ADDITIONAL STATES The second approach for VO-based smoothing sensor fusion does away with the necessity of introducing additional states since IC poses are directly connected to the IC pose corresponding to the active frame. In SLAM parlance, such a setup is quite common and is referred to as a *between*

measurement, as would be obtained from wheel odometry information.

In our case, however, direct usage of a simple *between* factor is not possibile because of the calibration pose CB. With reference to figure 18, the VO estimation produces pose B_iB_j but, for inertial navigation to work correctly, poses IC_i and IC_j must be used as states instead.

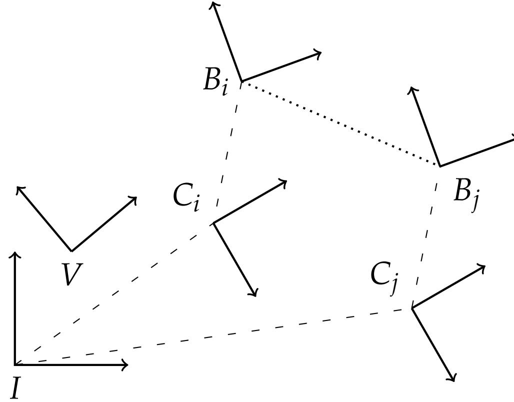


Figure 18.: Coordinates frames as described in section 4.2, in the case of no additional keyframes

To solve this problem, we perform the following coordinate transformations on the relative pose between B_i and B_j:

$$({}^{C_i}_{I}T)^{-1}\, {}^{C_j}_{I}T = {}^{B}_{C}T\, {}^{V}_{B_i}T\, {}^{B_j}_{V}T\, {}^{C}_{B}T$$

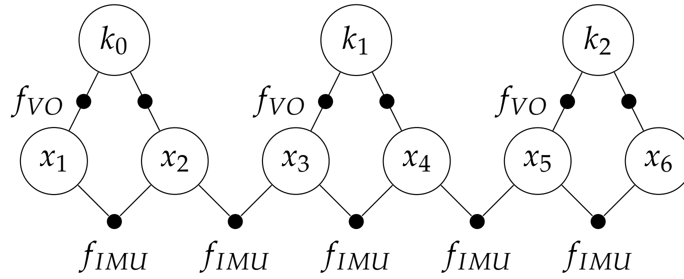
As shown in the last equation, we can use the "modified" relative pose:

$${}^{B}_{C}T\, {}^{B_2}_{B_1}T\, {}^{C}_{B}T$$

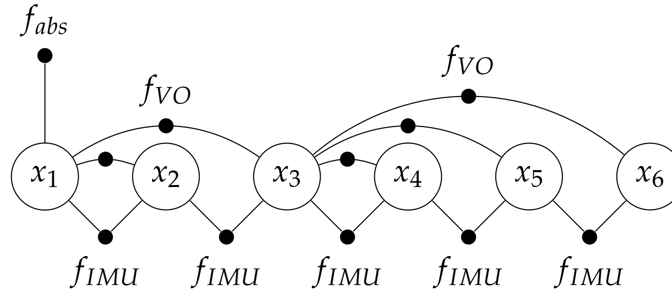
to constrain the relative poses between states IC_i and IC_j. This can easily be carried out thanks to GTSAM's built-in *Between-Factor* class.

When using the built-in *BetweenFactor*, the calibration pose CB from body (C) to camera (B) is assumed known. Alternatively, a custom factor can be implemented to perform the optimization over this pose as well. In our case, the relative calibration was known with rather high accuracy, so that this step was avoided for reasons explained in the following chapter.

In this configuration, the VI pose is not used as a system state during the optimization process. However, it is still estimated during the start-up phase, as it can be used to recover the VO estimate from the IC state.



(a) Factor graph with visual keyframes added as optimization states.



(b) Factor graph with relative VO measurements without adding further states for the keyframes.

Figure 19.: Factor graphs for the visual odometry-based smoother.

The factor graph resulting from this approach is shown in figure 19b, showing how each system state is connected to its corresponding keyframe by a relative VO measurement.

For the rest of thesis, we chose this second approach, since it is more straight-forward and coherent with the internal operation of the VO estimator. The reduction of around 20% of the number of system states is also attractive. Furthermore, the presence of a "chain" of measurements opens up new opportunities for future work. For example, an intelligent algorithm could decide to "coalesce" many system states into one to reduce the size of the graph, especially during times of low movement.

4.2.1 *The noise model*

As we mentioned in the last section, the choice of a noise model for the VO estimate is somewhat more arbitrary. For instance, the VO algorithm may fail with some image pairs, thus obtaining a pose estimate with an unusually high variance.

More importantly, the VO algorithm is intrinsically more accurate in some directions than in others. Figure 20 highlights this fact by comparing the residual error with an equal translation in different directions. The slope of the residual error when translating along the z direction is markedly smaller than that in the x and y directions. Correspondigly, we expect a larger error in this direction. Unfortunately, this is difficult to account for in the noise model, as the variances are expressed in the C frame, and not in the B frame. While we could use a coordinate transformation, in practice this is not really needed, as suitable values for the noise model can be found experimentally without much effort.

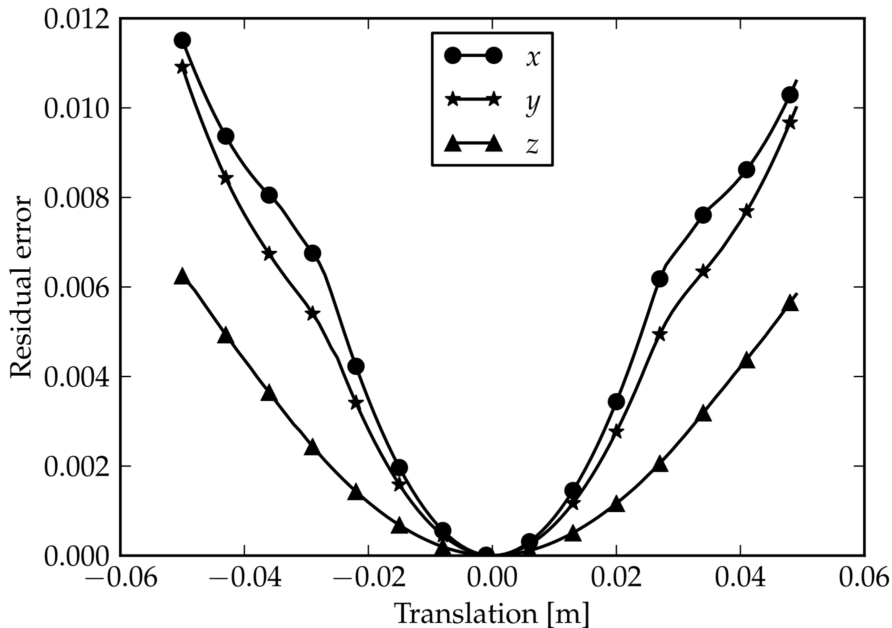


Figure 20.: Error resulting from translations in different directions.

While we will have more to say in this topic in the next chapter, for the loosely coupled approach we simply selected a diagonal noise model with two distinct variances, one for valid for all the translation components and one for the rotational ones.

4.3 EXECUTION TIME

The incremental algorithms in GTSAM are designed to be able to run in constant-time no matter the number of factors added to the graph. We put this claim to the test by using the setup in

section 4.2. In practice, a long dataset has been recorded while hand-holding the sensor board. The result is shown in plot 21.

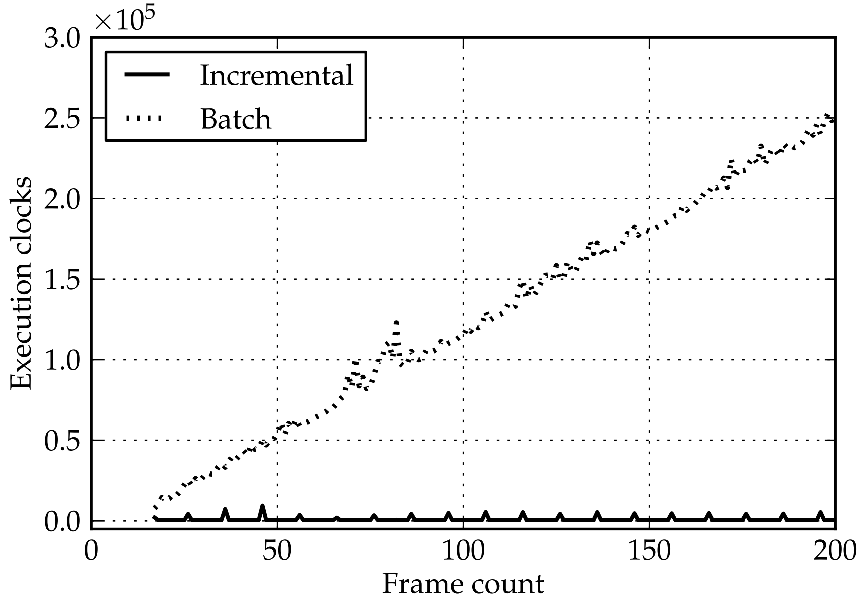


Figure 21.: Factor graph optimization time as function of the number of included frames.

We note how, in this case of "loose" loop between GTSAM and the VO algorithm, the computational load on the smoother is rather low, since most of the computationally-heavy work is done within the VO estimator instead. Thanks to the flexibility of the GTSAM library, comparison between the smoother and the batch solver is straight-forward, as identical measurement factors can be used.

In particular, we compared *iSAM2* with a standard Gauss-Newton solver with target error set at 10^{-6}. In figure 21, the execution time for iSAM2 is barely visible when stacked with the batch solver. While the incremental solver exhibits periodical peaks due to re-linearization, the batch solver execution time scales linearly with the number of constraints added. It is thus clear that the batch solver is not adequate for real-time usage on an autonomous robot, since the execution time is not at all bounded.

To put these numbers into perspective, we compared the execution time of the VO algorithm with the time required to update the GTSAM estimate. This analysis is useful to assess the relative computational loads of these two procedures while using a "loose" loop architecture. Figure 22 plots the aver-

age of the execution times for DVO and GTSAM in the first four sets of 100 images each in the dataset. One can see how the smoother execution time remains constant as the number of constraints increases. In any case, the time spent on the smoother is short compared to that devoted to the VO optimization.

In conclusion, we have seen how, in the loosely coupled formulation, the performance of the incremental routines in GTSAM are adequate for bounded real-time sensor fusion.

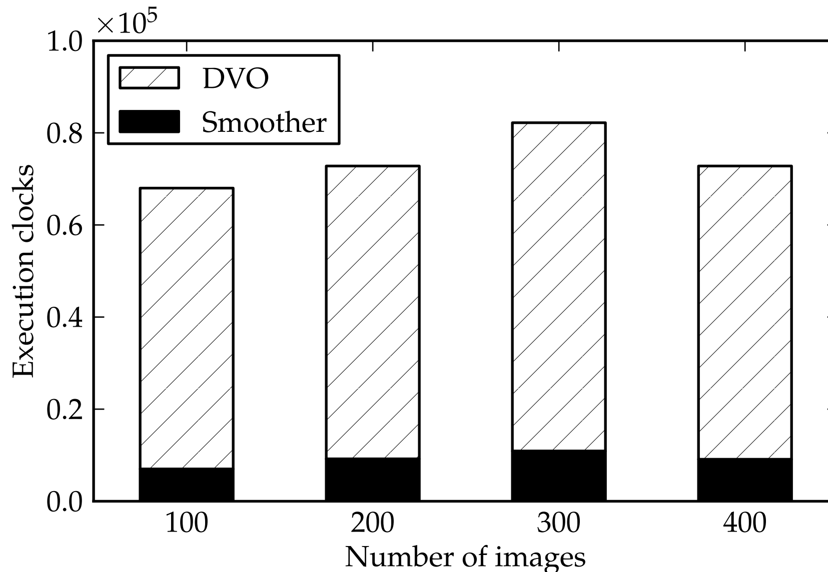


Figure 22.: Execution times for DVO and GTSAM.

4.4 THE EFFECT OF KEYFRAME SELECTION

As we have already discussed, in the case of RGB stereo images, the visual odometry algorithm tracks the relative motion between a *keyframe* and the current frame. The heuristics leading to the selection of such keyframes play an important role in the performance of the system.

For instance, if the relative motion between the current frame and the reference grows too large, the optimization algorithm may become slow or fail altogether. On the other hand, frequently changing keyframes requires an expensive computation of the depth map of the image, as discussed in section 3.3. In any case, due to the factor graph architecture chosen in the

preceding section, additional keyframes have no performance penalty on the GTSAM smoother.

A keyframe selection algorithm is supposed to take into account one or more of the following factors:

- the number of frames received since the keyframe had last been changed;
- the accumulated translation as measured in the last tracked frame;
- the accumulated rotation (in axis-angle representation) as measured in the last tracked frame

We performed an analytical analysis of the effect of the keyframe selection heuristic on the VO algorithm performance, though wary of the fact that performance may be dependent on the particular scenario and image features. We started out by investigating the effect of keyframe frequency on the number of iterations required to convergence.

For this analysis, we used a custom GTSAM factor built on top of Kerl's image warping routines and derivatives computation. We implemented a simple iteration control routine based on a simple factor graph with a single factor and a single state representing the relative pose between the two given images. The graph is repeatedly linearized and solved until the residual error is less than 10^{-6} or a given number of iterations is exceeded (20 in our case). Iterations are also stopped whenever the computed error results in an increase in the residual error.

With this setup, we investigated the simplest possible keyframe selection heuristic, consisting in choosing a new keyframe after a fixed amount of image frames, ranging from 1 to 6. Larger numbers were not tested because tracking reliability would suffer too much. We found that the keyframe interval had no effect on the number of performed iterations before convergence. We attribute this to the fact the iteration controller is set up to give up after an interations result in an increase in the error.

On the other hand, the keyframe interval was found to have a stronger effect on the residual error at convergence, as shown in figure 23.

In particular, the standard deviation of the residual error had a strong dependence on the keyframe error. Furthermore, val-

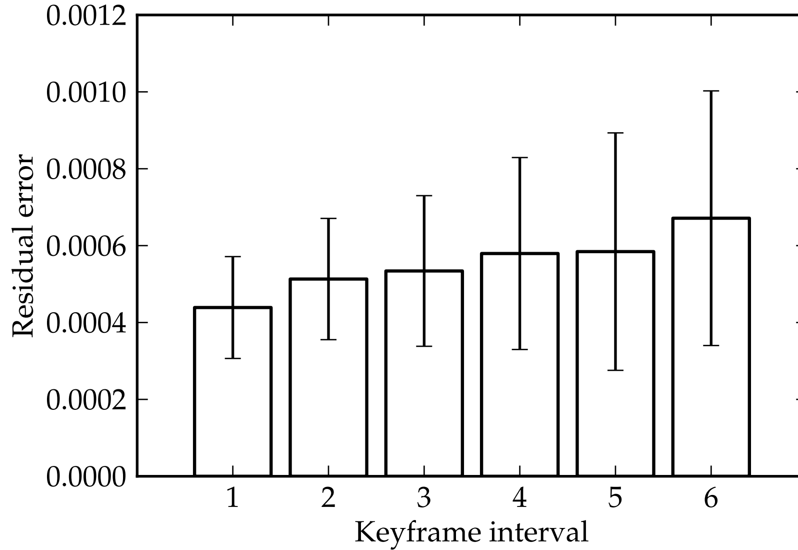


Figure 23.: Effect of the keyframe interval on the residual error at convergence.

ues larger than 6 are prone to tracking failures with this simple heuristic.

ROTATION BASED HEURISTIC Experimentally, it is found that the rotation part of the relative pose has the strongest impact on the reprojection error and thus on the tracking accuracy. For this reason, it makes sense to develop an heuristic based on some "magnitude" metric of the rotation component of the relative pose. It is assumed that, once such a metric reaches a certain threshold, a new keyframe should be selected and used for the following frames.

Euler's rotation theorem proves that any rigid rotation in SO3 can be expressed as a rotation of a certain angle around a fixed axis. This property motivates the *axis-angle* representation, through which a rotation is parametrized as a unit vector $\hat{e}$ representing the axis of rotation, and a scalar θ describing the angle of rotation. Thanks to the axis-angle representation, our rotation-based heuristic can be implemented as a conditional on the (absolute) value of the rotation angle to the active keyframe.

We compared the rotation-based heuristic with the fixed interval approach on the same dataset which represents typical operation of the UAV (with a brief starting-up period of low motion in which addition of new keyframes is useless). In the

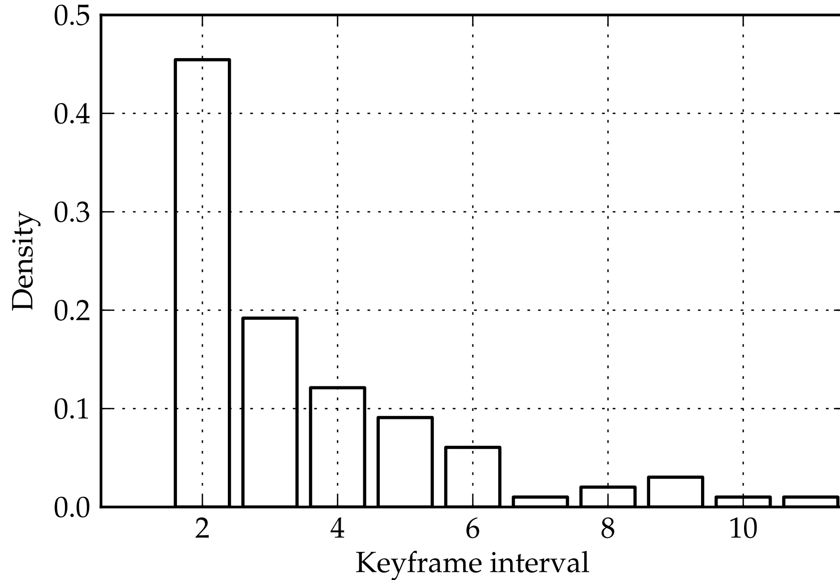


Figure 24.: Distribution of keyframe intervals with the rotation-based heuristic.

interest of a fair comparison, we adjusted the rotational threshold so that both approaches resulted in the same number of keyframes at the end of the dataset. For example, a fixed interval of 4 frames corresponded to a threshold of 0.165 rad.

The resulting probability distribution of keyframe intervals is shown in figure 24. We see that, in our sample dataset, the rotation-based heuristic increases the intervals during periods of slow motion. Besides reducing the computational load, this is also beneficial because it reduces estimate drift.

We found no appreciable difference on the number of iterations to convergence. On the other hand, figure 25 plots the probability density function of the residual error for the two different cases. It is clear how the distribution for the rotation-based heuristic is skewed to the left, validating the higher accuracy of the VO estimate. Numerically, we found a reduction of around 50% of the average residual error on our sample dataset.

OTHER APPROACHES Reference [20] sets a threshold on the reprojection error to trigger keyframe changes. We have not investigated this possibility due to our aim of implementing a tightly coupled visual odometry measurement model. In fact, using the residual error would introduce un-needed coupling between the SLAM and VO systems.

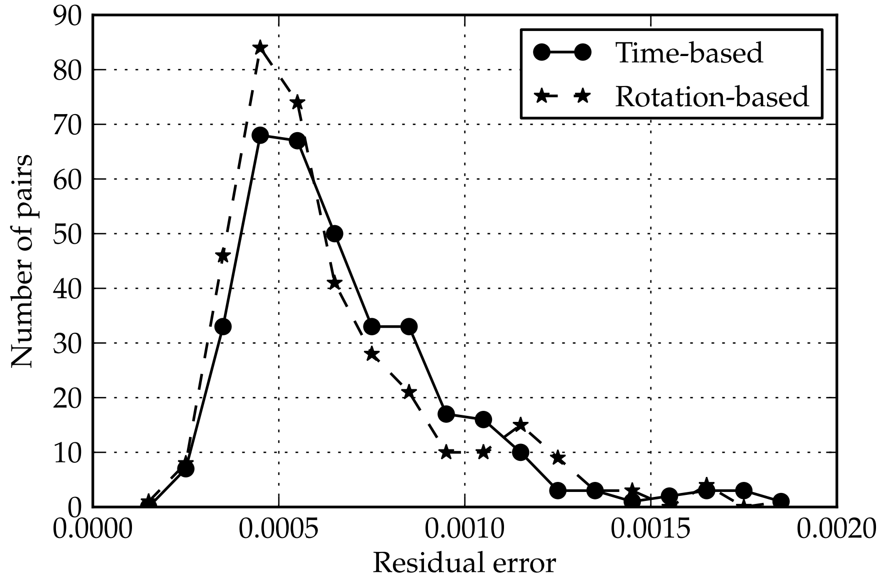


Figure 25.: PDF of the residual error at convergence for the two keyframe selection approaches.

A more interesting approach is presented in [16] and introduces the concept of *entropy* of the estimate. The entropy $H(\mathbf{x})$ is proportional to the natural logarithm of the determinant of the covariance matrix Σ:

$$H(\mathbf{x}) \sim \ln(|\Sigma|)$$

The authors observed a relationship between the entropy of the minimal representation $H(\xi)$ and the reprojection error and developed the concept of an *entropy ratio* to identify degradation in tracking performance. This approach is far more interesting in our case because GTSAM provides optimized functions to retrieve an approximation of the covariance matrix of states, so that *information about the complete graph* can be used when making decisions about keyframes.

5

THE TIGHT LOOP APPROACH

In this chapter, we describe two progressively more advanced approaches to improve the coupling between the GTSAM smoother and the VO algorithm. Most importantly, we develop a custom development factor which can directly incorporate image information in the GTSAM factor graph, for optimal results.

5.1 LEVERAGING THE INERTIAL PLATFORM

The VO optimization algorithm reduces its execution time by operating on a series of progressively-higher resolution images. This is a well-known solution in the image processing community, known as using "image pyramids". As shown in Kerl's thesis, the lower resolution images offer a steeper minimum in the optimization process, and correspondingly lower execution times. On the other hand, the higher-resolution images allow for more accuracy in the final pose estimate.

Following from this discussion, it is clear how the availability of an accurate first estimate of the pose would significantly reduce the time spent matching the lower-resolution images in the pyramids. A clear improvement could be made by taking advantage of the IMU measurements not only *after* the VO estimate is computed, but also *before*.

In practice, we decided to carry out an open-loop integration of the IMU measurements (rotation velocity and acceleration). Since the gyroscope and accelerometer biases are updated as system states, their most current estimate can be used for optimal performance of the open-loop integration, which would be likely to diverge.

We employed the canonical strap-on inertial navigation procedure, using the most current estimate of the body pose in the inertial frame (${}^{c}_{i}T$) to account for the gravitational acceleration.

After the straightforward integration, the current optimization states were again used to convert the updated ${}^{c}_{i}T$ pose to an estimate of the relative transform between the current image and the active keyframe (pose ${}^{v}_{k}T$).

In the following section, we benchmark this approach and compare it to a simpler prediction algorithm.

5.1.1 *Comparison to the naive prediction*

The simplest prediction algorithm assumes constant velocity (both rotational and translational). In practice, we compute the differential transform between states $i-1$ and i and apply it to predict the transform at time $i+1$. In this section, we compare this solution to the IMU-aided prediction in terms of their difference to the actual pose (as produced by the VO algorithm after convergence). We also assess their relative performance in terms of the number of Gauss-Newton steps required to reach convergence, a direct measure of the execution time.

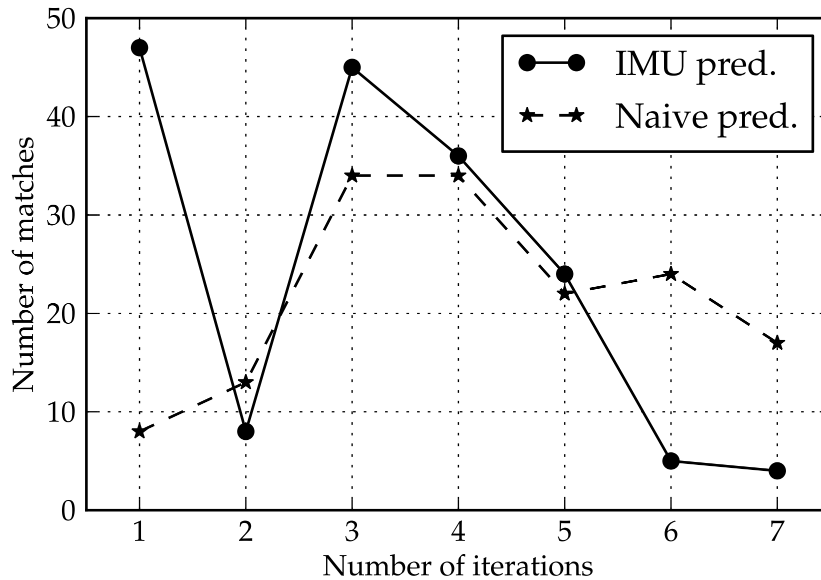


Figure 26.: PDF of the number of iterations for both the "naive" and IMU-based pose prediction

Results are summarized in plot 26 as a probability density distribution of the number of iterations required for the second-lowest image size. A comparison of the two distributions shows how the IMU-based prediction outperforms the "naive" approach. We also note that in the case of very slow movements,

the IMU-based prediction was observed to be of lower quality than the reference solution. We worked around this problem by using it instead of the IMU in these particular cases (which, anyhow, are not of particular interest).

To explain this improvement in performance, we compared both the "naive" and IMU-based predictions to the final estimate by the VO algorithm. We focused on the rotation part of the relative transformation, since it has the most impact on the image residuals. Using the axis-angle representation of the rotation, we obtained a single $\mathcal{R}$ metric for its prediction error.

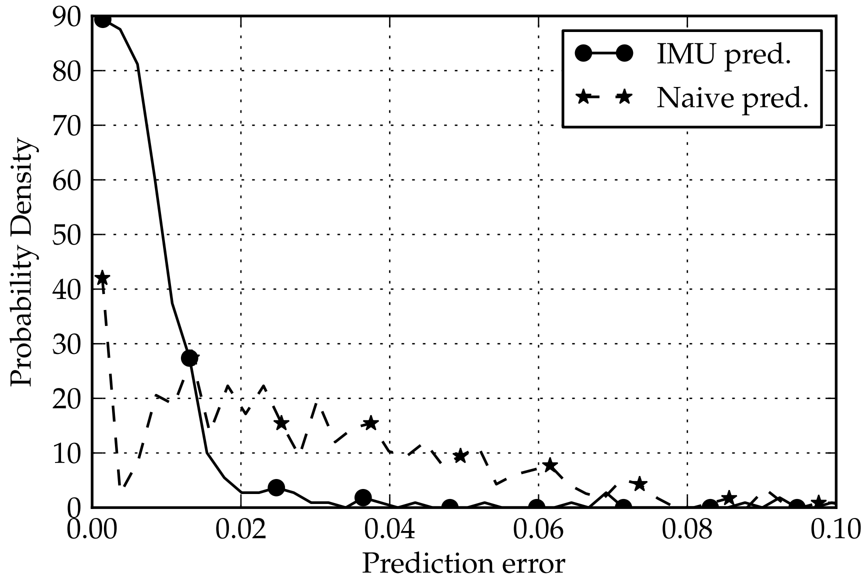


Figure 27.: PDF of the rotation prediction error for the "naive" and IMU-based pose prediction

As can be seen in figure 27, the probability distribution of the rotation prediction error is skewed towards the left in the case of the IMU based approach. In other words, compared to the "naive" approach, many more predictions are closer to their optimal value, thus resulting in fewer iterations within the VO algorithm.

5.1.2 *Collection of IMU sensor data*

Due to the way the ROS framework is implemented, messages of different types (such as camera images of IMU measurements) are not guaranteed to arrive to the *subscribed* in the order they are created from the *publisher*. This is due to various de-

lays in the system, as could be brought about by the need to transport large images. Luckily, each message is timestamped accurately, and the two cameras are synchronized in hardware, so that proper ordering and timing can be recovered using a buffer.

This is especially important when dealing with the IMU measurements, since the measurement factor is using straightforward Euler integration. Since the camera is synchronized in hardware, once a new camera image is available, the smoother should have already received a complete set of IMU measurements. With our particular setup, this means a set of 10 IMU measurements (sampled at 100Hz) for each pair of camera images (sampled at 10Hz).

In practice, the ROS middleware is not always perfectly synchronized, leading to our usage of a buffer for temporarily storing the IMU measurements. Furthermore, sometimes the full set of IMU measurements is not available until after the image pair is received from the cameras. Since it would not be rational to delay the state estimation just to wait for a complete packet of IMU measurements, we devised an "interpolation" algorithm to fill in the gaps.

With reference to the pre-integrated measurements introduced in section 2.4, we created additional fictional measurements assuming constant acceleration and angular velocity. In these particular cases, we also changed the noise model for less reliance on these "doctored" measurements.

5.2 BUILDING A UNIFIED OPTIMIZATION PROBLEM

As we have seen, the strength and power of using a graph-based smoother lies in its ability to easily incorporate measurements from different sources, as long as they can be formalized as least-squares problems.

In the last chapter, we introduced the VO information in the factor graph as an error term with respect to the final pose estimate produced by the VO algorithm. This approach, while a natural evolution of filtering, is not optimal because we need to solve two distinct optimization problems, one in the VO algorithm, and one in the GTSAM smoother.

On the other hand, the discussion on the dense vision algorithm has shown how minimization of the reprojection error is

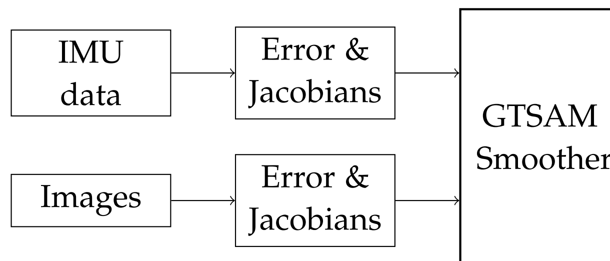


Figure 28.: Operation with tightly-coupled visual odometry.

itself formulated as a least-squares problem (see equation 12). A natural evolution of our past technique would then incorporate such information directly in the GTSAM factor graph, without a separate optimization step to converge on a pose estimate. In the case of the "loose" approach we are in fact iterating to convergence *twice*.

In fact, our "tight" approach is the opposite of what Weiss advocated in his thesis (see [27]). We are banking on the benefits of an incremental smoother to gain from an increased amount of information kept within the filter.

Some of the benefits we could expect from this change are the following:

- the noise model for the "loose" factor is somehow arbitrary, and bears no relation to the actual image tracking process. Even though we reached good results with a diagonal noise model with reasonable magnitudes, it was hard to account for the individual differences between each image pair. On the other hand, while using the "tight" factor, no such decision is needed because the pixel errors can be normalized while building the least-squares problem;

- ability to relinearize even past image pairs to improve the estimation accuracy. The process could be scheduled in periods of low CPU activity or carried out on-demand whenever more precision is required;

- possibility to choose keyframes "on-the-fly" depending on the VO optimization outcome. Detection of a tracking failure could be dealt with by removing the useless measurement factor and replacing it using a more recent keyframe. This could increase the likelihood of estimation success at the expense of computation time;

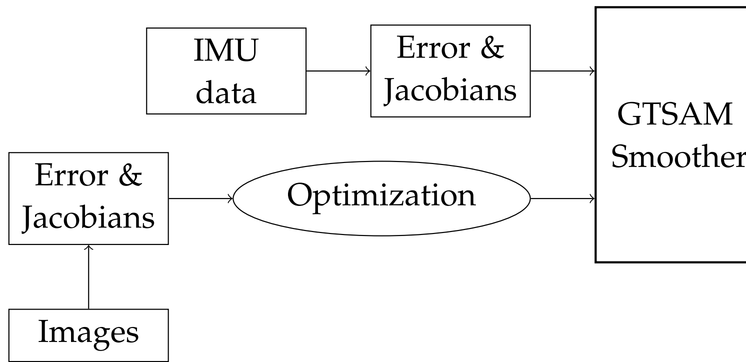


Figure 29.: Operation with loosely-coupled visual odometry.

- improvement of the off-line pose history estimate. This could b useful for surveying applications, for which improved precision can be recovered after the mission through repeated optimization.

Figure 28 and 29 highlight the difference between the "tight" and "loose" approaches: in the first case, the output from the VO stage to the GTSAM smoother is a final estimate on the relative pose between two consecutive images. In the second case, just the error and its Jacobians are passed from the VO stage to the smoother, which iterates this VO factor together with the IMU.

Regarding implementation, there are two mathematically equivalent alternatives that immediately spring to mind.

A FACTOR FOR EACH PIXEL PAIR a direct transposition of equation 12 in the context of factor graphs. For example, matching a pair of 64x64 images would produce around 4096 distinct factors, each with its 6×1 Jacobian matrix and scalar residual (error). While attractive from a theretical point of view, this solution has a number of disadvantages. First, even taking into account efficient methods for manipulating sparse matrices, the computing overhead in handling such a large number of factors is likely to be excessive. Secondly, any attempt at optimizing image manipulation routines (such as the warping function), would be made harder by GTSAM operating on a single pixel at a time.

A FACTOR FOR EACH PAIR OF IMAGES with a n-dimensional error vector and its corresponding $6 \times n$ Jacobian matrix.

While more efficient than the aforementioned option, this approach still does not take advantage of any optimizations built into the VO code. Furthermore, since GTSAM is optimized to work on large sparse matrices with relatively small non-zero blocks, using such large matrices (a row for each pixel) would be sub-optimal.

Most importantly, the least-squares normal equations can be built pixel-by-pixel in the linearized problem, with great memory and computation cost advantages. In other words, matrix $\mathbf{H}$ in equation 6 would have to be computed as the product of two matrices with n rows or columns (where n is the number of pixels in the image) rather than the sum of n outer products between 6-dimensional vectors. We are basically shifting the burden of computing $\mathbf{H}$ from DVO, where it can be done efficiently, to GTSAM, where it is done very slowly.

Since neither of these options is satisfactory for the reasons outlined above, the following section presents an alternative.

5.2.1 *Problem space reduction*

We propose using the Cholesky decomposition to reduce the dimensionality of the DVO problem while leveraging possible optimizations built into the existing code.

During each iteration, the DVO code builds the least squares normal equations to compute the increment $\mathbf{x}$ to be applied to current pose:

$$\mathbf{J}^T(\xi)\mathbf{J}(\xi)\ \mathbf{x} = -\mathbf{J}^T(\xi)\ \mathbf{r}(\xi) \qquad \Leftrightarrow \qquad \mathbf{A}\ \mathbf{x} = \mathbf{b} \tag{17}$$

As we mentioned earlier, the $\mathbf{A}$ matrix is efficiently computed pixel-by-pixel by summing up all the outer products of the pixel Jacobians. This process is obviously computationally more efficient than storing a big Jacobian in memory to be multiplied by its inverse. The same consideration applies for the right hand side of equation 17.

On the other hand, GTSAM requires the Jacobian of the error to linearize the factor. A convenient solution to this problem involves performing a *Cholesky decomposition* of the $\mathbf{A}$ matrix, as explained in the following paragraphs.

Given a symmetric, positive definite matrix such as $\mathbf{A}$ in equation 17, Cholesky decomposition computes an upper triangular matrix $\mathbf{U}$ such that:

$$\mathbf{A} = \mathbf{U}^T\mathbf{U} \tag{18}$$

By comparing equations 17 and 18, we see that we can build an equivalent least squares problem as follows:

$$\mathbf{U}^T\mathbf{U}\ \mathbf{x} = -\mathbf{U}^T\ \mathbf{c}$$

where $\mathbf{c}$ is, by comparison:

$$\mathbf{c} = -(\mathbf{U}^T)^{-1}\ \mathbf{b} \tag{19}$$

In conclusion, the Cholesky decomposition and equation 19 allow for a compact system of equations to be inserted into GTSAM for optimization in the factor graph. In this formulation, the non linear VO factor is linearized to a "Jacobian" factor, the Jacobians obtained as above.

5.2.2 *Hessian based linearization*

There is yet another interesting approach for implementing the tightly coupled factor that keeps all the advantages of the Cholesky decomposition and is even more mathematically appealing.

In the previous section, we took the default GTSAM route and linearized our factor using its Jacobian matrices. As explained in the GTSAM API documentation ([9]), such a "Jacobian" factor implements an error (negative log likelihood) of the following form):

$$E(\mathbf{x}) = \|J\mathbf{x} - \mathbf{b}\|^2 \tag{20}$$

by storing matrix J and vector $\mathbf{b}$. Optionally, non linear factors can also be linearized to an *Hessian factor* when convienient. In fact, the negative log-likelihood of a Gaussian is given by:

$$E(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \mu)^T P^{-1}(\mathbf{x} - \mu)$$

where μ is the mean and P the covariance matrix. Expanding the product we arrive at the generic quadratic factor implemented by the Hessian factor:

$$E(\mathbf{x}) = 0.5\mathbf{x}^T\Lambda\mathbf{x} - \mathbf{x}^T\eta + 0.5\mu^T\Lambda\mu$$

If we compare this last equation with the expansion of 20, we see how we could directly linearize to a Hessian factor and save the time required for the Cholesky decomposition. In other respects, the two forms are equivalent since GTSAM carries out the same operations during the solving procedure.

5.2.3 *Implementation*

We implemented the "tight" measurement factor as a drop-in replacement for the "loose" factor in the past chapter. In other words, instead of receiving the DVO measurement in the form of a 3D pose transformation, the "tight" factor works directly from a pair of images, thus achieving our objective. This setup allowed us to keep using the graph topology pictured in figure 19b while mixing and matching the different factors for comparison purposes.

We decided to plug in into Kerl's implementation at the optimization stage with minimal code changes, so that both the loosely and tightly coupled approaches could be used on the same codebase. Besides taking into account the different conventions for the minimal representation of the Lie algebra, some additional care has been taken in ensuring that the solver used a correct estimate of the scalar residual error.

Given a certain set of system states and a measurement factor, GTSAM computes the (scalar) residual error by using the factor noise model. For example, in the case of a Gaussian noise model with square root information matrix R, we have the Mahalanobis distance:

$$d = < Rv, Rv >$$

While this approach would have been correct had we used the full stacked Jacobian in the factor, it is no longer applicable with the selected solution (pre-multiplication of the A matrix and subsequent Cholesky decomposition). The $\mathcal{R}^6$ vector of errors has no relation to the pixel-wise residuals anymore, and thus can't be used for iteration control in the GTSAM smoother. In other words and referring to a simple Gauss Newton optimizer, GTSAM would iterate through the correct steps, but never successfully detect convergence if using the noise model as described above.

We solved this problem by using the scalar residual as computed by DVO. This also has the advantange of being able to leverage built-in weighting of the residuals to offset the effect of outliers on the estimation.

5.2.4 *Robustification and the error model*

The term *robust* estimation refers to the algorithm's ability to cope with the presence of *outliers* in the data. Outliers are pixels whose residual is very high, due to scene movement or global brightness changes (automatic esposition). Least squares estimation is intrinsically very sensitive to outliers since the residuals are squared, thus giving a disproportionate "importance" to big residuals that could skew the optimum.

As we have seen, in the conventional least squares approach the optimum estimate is obtained as follows:

$$\xi_{MAP} = \arg\min_{\xi} \sum_i (r_i(\xi))^2$$

To increase the robustness, the squared term can be replaced with an *error function* $\rho(x)$:

$$\xi_{WLS} = \arg\min_{\xi} \sum_i \rho(r_i(\xi))$$

We define the weight function $\omega(x) = \rho'(x)/x$ as described in [24], so that by taking the negative logarithm, we obtain the *weighted least squares* problem:

$$\xi_{IRLS} = \arg\min_{\xi} \sum_i \omega(r_i)(r_i)^2$$

We follow the approach implemented in [15] and choose weights based on the *t-distribution* given by:

$$\omega(x) = \frac{\nu + 1}{\nu + \left(\frac{x}{\sigma_t}\right)^2}$$

The parameter σ_t has the role of a scale factor and must be estimated iteratively based on the residuals themselves at each iteration. Reference [18] provides more information on this topic. ν is the number of degrees of freedom in the distribution and is set to 5 in our implementation.

In the past chapter, the noise models used in GTSAM were set experimentally based on the expected accuracy of the *final* relative pose estimated by the VO algorithm. On the other hand, the current tightly coupled approach requires a more careful investigation of the role of noise models in the SLAM optimization process.

As we have seen, GTSAM carries out the optimization of a factor graph iteratively by linearizing each *NonLinearFactor* to a *JacobianFactor*. This step is carried out by computing the error Jacobians with respect to the affected states and assembling a matrix. Afterwards, given a linearization point, the Jacobians and the error computed at that point are then *whitened*. Whitening refers to transforming a set of variables with a given covariance matrix so that their covariance matrix becomes the identity matrix (uncorrelated variables with unity variance). In the case of a Gaussian factor, whitening is simply done by pre-multipying the Jacobian matrix and the error vector by the square root information matrix R.

We see how choosing appropriate noise models ensures that all the factors in the linearized graph can be whitened correctly so that the optimization is carried out successfully. In other words, the noise models ensure that all information is taken into account with the correct "weights". Furthermore, we see how the weighting step in DVO can already be adjusted to whiten the system before it is handed over to GTSAM.

Speeding up the execution with image pyramids

As described earlier, DVO uses a pyramid approach to reduce the computational load for pose estimation. The first iterations are carried out on heavily-subsampled copies of the original image to reduce the number of pixels to be taken into consideration. After the residual error has converged on the i-th image level, the optimizer switches to the twice-bigger image at the $i-1$-th level. In practice, the original 640×480 images are downsampled 3 times to start with 80×60 pixel versions. It is usually found that accuracy-wise there is no need to go to the full-sized original frame, but that a 320×240 version is sufficient.

When switching from the 'loose" to the "tight" factor, some care must be taken to keep the advantages of this approach. One could set up the GTSAM factor with a similar behaviour,

so that it would start using bigger and bigger images after convergence. However, sound software design decisions mean that the factor itself has no information about the state of the optimization process. Furthermore, since the residuals at different image scales are not comparable directly, such a behaviour would create problems at the optimization stage, with errors increasing and decreasing without connection to the Jacobians.

For this reason, we decided to maintain the existing behaviour, so that DVO would separately optimize the smallest image sizes and provide an initial estimate to the GTSAM factor for it to complete the optimization on the full-sized image.

5.3 EVALUATION

In this section, we present our experiments using the tightly coupled factor described in the previous pages. In particular, we are concerned about the execution time and the accuracy when compared to the more widespread loosely coupled approach. We also devote some time to the analysis of the available solution methods for the factor graph, from batch solving to incremental smoothing to fixed lag smoothing.

5.3.1 *Execution time*

In the past chapter we have shown how the incremental smoother is fast enough for real-time operation of the loosely coupled system, since additional computation required by the graph optimizer is minimal when compared to the visual odometry estimation. On the other hand, now that the VO estimation is integrated directly into the factor graph, a small increase of the number of iterations in the GTSAM solver has a measurable effect on system performance. In particular, correct configuration of the solver settings (such as error tolerance and maximum number of iterations) is critical for performance evaluation.

We set up a factor graph with "tight" factors, a fixed keyframe interval equal to 5, and configured the factor to only iterate on 80×60 downsampled images. IMU measurement factors are of course also included. Among the various *iSAM2* algorithm configuration parameters, we deemed the *re-linearization threshold* to be most interesting for our present needs. This value controls when GTSAM should relinearize factors given

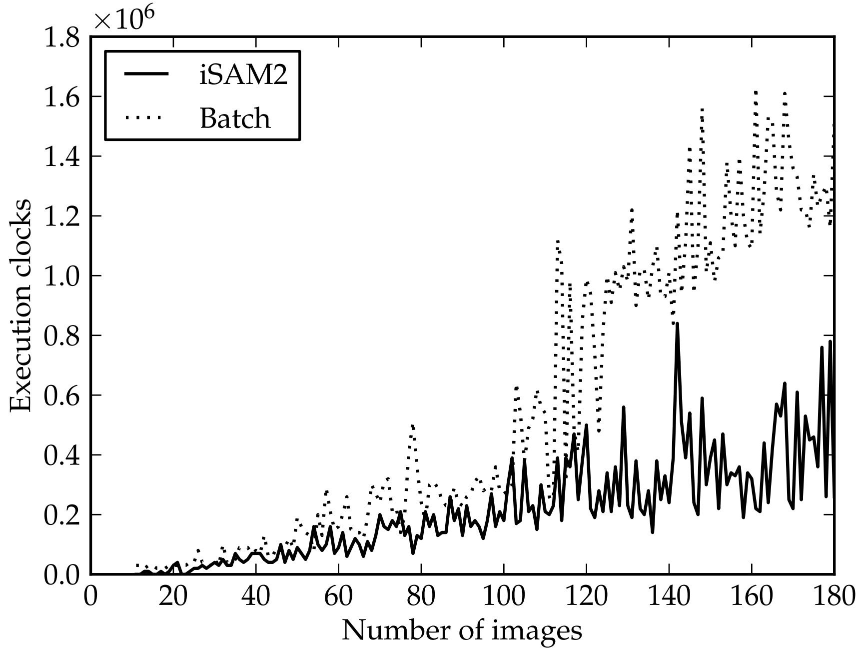


Figure 30.: Execution time (in number of CPU clocks) for the iSAM2 algorithm and the batch solver.

the linear increments in the last optimization step. Compared to standards factors such as those encoding relative poses or probability priors, the re-linearization of our "tight" factor is much more expensive, because of the required image processing. For this reason, the re-linearization threshold has a strong effect on the total execution time in the case of the tightly coupled system.

In the case of figure 30 we set the re-linearization threshold for the IC pose as 8×10^{-3} rad and 5×10^{-4} m for the rotation and translation respectively. iSAM appears quite effective in reducing the execution time as the factor graph grows larger. On the other hand, we suspect that a "chain" effect brings about the need to re-linearize a large amount of past factors, even though such an expensive process may have a limited effect on the current state estimate.

This chain-effect can be quantified with reference to the principle of operation of the incremental smoother. New factors and guesses for the values of the new states are added during the call to the smoother's *update* method. These factors are linearized and added to the graph. Further calls to the *update* method perform additional iterations of the algorithm, relin-

earizing factors when needed (as controlled by the relinearization threshold mentioned above).

5.3.2 *Effect of relinearization threshold*

From the discussion above, two metrics are available to quantify the extent of the chain effect brought about during the solution procedure. The first metric is the number of *update* calls needed for all states to converge within the relinearization threshold, i.e. the point at which further iterations do not require costly relinearization.

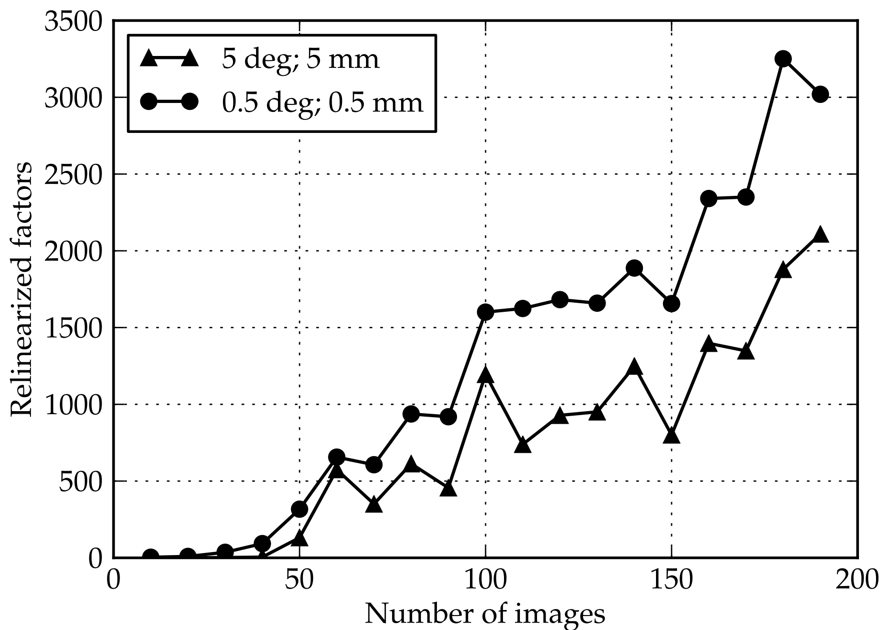


Figure 31.: Number of relinearized factors for different thresholds

The second available metric is the *total* number of factors relinearized during the iterations needed to reach the state described above. We expect this second metric to bear a closer relation to the execution time of the tightly coupled smoother and visual estimator.

We experimented with two different relinearization thresholds, both adequate for reliable tracking on the sample dataset. We decided to cap the maximum number of calls to the *update* method to 10. Figure 31 shows how the extent of the chain effect grows as the thresholds get smaller. As more images are

stored in the graph, the number of factors requiring relinearization grows, more so when the threshold is smaller.

We also investigated the effect of the relinearization threshold on the smoothing accuracy. To this end, figure 32 shows a part of the robot trajectory in the inertial frame, specifically its x component in the first ten seconds.

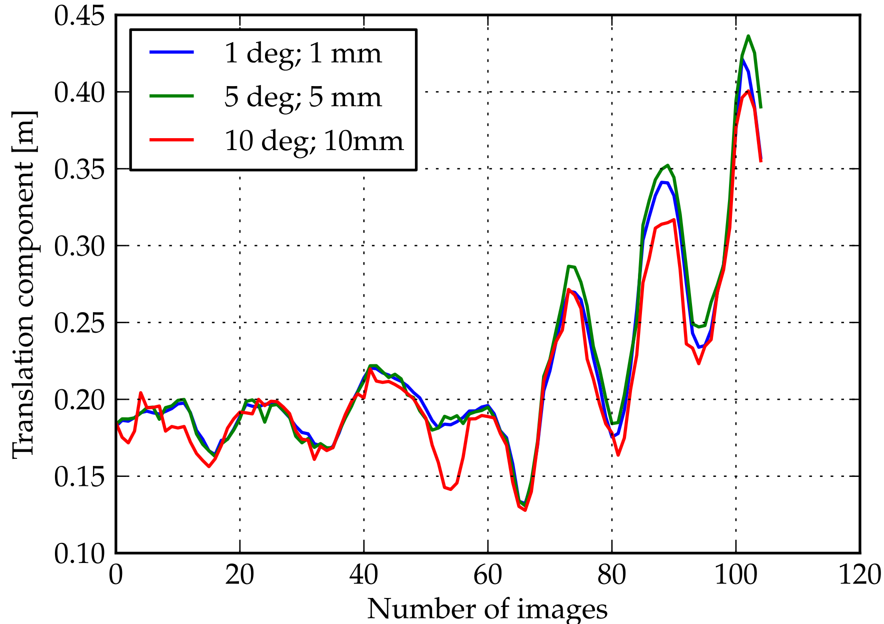


Figure 32.: Optimized trajectory as obtained by the iSAM smoother with different re-linearization thresholds.

We see how careful selection of the relinearization threshold can result in the optimal tradeoff between execution speed and accuracy. Similar conclusions can be found by examining the plots for the rotation components and with different datasets.

5.4 SELECTIVE MARGINALIZATION

As was expected, the incremental solver is much faster than the batch solver, but we nonetheless found it difficult to obtain good precision on the new states without setting off a "chain reaction" on the remaining states. New images can trigger a large number of expensive re-linearizations of past states. While this effect can be somehow mitigated by appropriately choosing the smoother threshold, real time operation is still not achievable with reasonable accuracy.

In the context of the Bayesian interpretation of the SLAM problem, past states are removed by *marginalisation*. Marginalisation is the complementary operation to *augmentation*, through which new states are added to the system (the "prediction" step in a Kalman filter). We take advantage of the sum rule to bring the *functional* dependence on x_2 out of $p(x_1, x_2)$:

$$p(x_1) = \int p(x_1, x_2)\mathrm{d}x_2 = \int p(x_1|x_2)p(x_2)\mathrm{d}x_2$$

In the case of Gaussian distributions, the joint distributions of all states are described by the covariance matrix, and marginalisation is carried out through Schur's complement. However, removal of a state creates many additional non-zero elements in the covariance matrix. For more information on the subject, together with a description on various state-of-the-art methods, one could refer to [26].

The discussion above suggests how *selective* marginalisation could be useful in improving the computational efficiency of the smoothing architecture. Though GTSAM offers a method to marginalize factors out of the graph, it is not usable in many cases since it can only operate on the "leaves" of the Bayes tree.

For a more general solution, we took advantage of the *fixed lag smoother* part of the library. Every time a new factor is added with images from the camera, the smoother marginalizes out factors older than a certain threshold. This is carried out by storing a timestamp when new factor are added to the smoother.

We tested this approach by comparing the incremental smoother with the fixed lag smoother. We selected two different values for the lag window, representing the time period for which all states are kept in the factor, 2.0 and 4.0 seconds.

The result is shown in figure 33, where the plots for the iSAM and the fixed lag smoother at 4 seconds are coincident. On the other hand, when the lag window is limited to 2 seconds, artifacts start to appear on the estimate. This shows how directly the duration of the window is connected to the quality of the estimate.

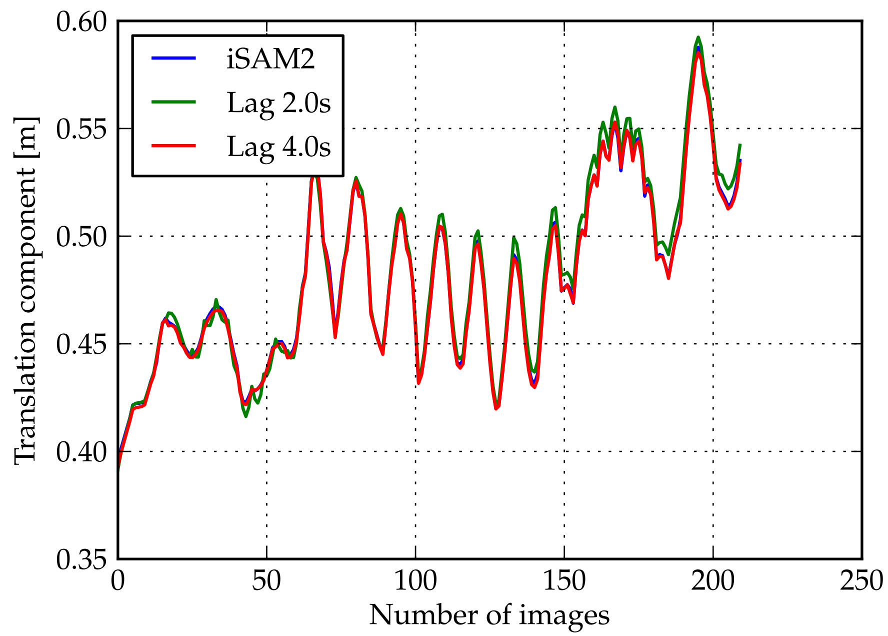


Figure 33.: Translation component as optimized by the incremental smoother and the fixed lag smoother with two different threshold.

6

CONCLUSIONS

The present work has demonstrated increasingly sophisticated methods for on-board sensor fusion using an IMU and visual odometry.

In the first part of the work, we have shown how the GTSAM framework can usefully replace a Kalman-based filter to better represent the intrinsical features of a dense-vision odometry information, namely the presence of keyframes and relative pose measurements. Also, a factor graph-based method for gravity calibration is presented.

Working towards a tighter integration of IMU and Visual Odometry, we have shown how accelerometer and gyroscope data can be used to improve the performance of the dense image matching algorithm.

Finally, we have shown how directly integrating the Visual Odometry equations in the SLAM factor graph achieves better performance and allows for more coherent error models to be used.

THE LOOSELY COUPLED FACTOR

Here we reproduce the (short) source code for the loosely coupled between factor described in the text. It is inherited from the *NoiseModelFactor2* base class since it optimizes over a pair of system states (the keyframe pose in the inertial frame and the current image's pose). The calibration pose from camera to center of gravity *CB* is passed to the constructor of the factor class. This is the factor we used for evaluation of the loosely coupled approach. More information on creating custom factors can be found in the authors' guide [4].

```
class LooseBetween: public NoiseModelFactor2<Pose3,
    Pose3> {

private:
  Pose3 measuredBK;
  Pose3 givenCB;

public:
  typedef boost::shared_ptr<LooseBetween> shared_ptr;

  LooseBetween(Key j, Key k, const Pose3& measured,
                             const Pose3& givenCB_,
                             const SharedNoiseModel&
                                 model):
    NoiseModelFactor2<Pose3,Pose3>(model, j, k),
      measuredBK(measured),
      givenCB(givenCB_)
       {}

  virtual ~LooseBetween() {}

  Vector evaluateError(const Pose3& pIC1, const Pose3&
      pIC2,
        boost::optional<Matrix&> H1 = boost::none,
        boost::optional<Matrix&> H2 = boost::none
      ) const {
```

```
  Pose3 hx = pIC2.between(pIC1, H2, H1); // h(x)
  Pose3 measuredC2C1 = givenCB * measuredBK * givenCB.
      inverse();
  return measuredC2C1.localCoordinates(hx);
  }

  virtual gtsam::NonlinearFactor::shared_ptr clone()
      const {
    return boost::static_pointer_cast<gtsam::
        NonlinearFactor>(
         gtsam::NonlinearFactor::shared_ptr(
            new LooseBetween(*this))); }
};
```

B

THE CALIBRATED FACTOR

In this appendix, we reproduce the code for the keyframe-based factor class. This factor optimizes over four distinct system states: the two calibration poses CB and VI, and two poses in the inertial frame (one for the current image and one for the keyframe it is matched against).

With this factor, one can recover optimized estimates of the relative pose between the gravity and the first image received, and also the updated calibration pose between the IMU and the camera.

```
class LooseWithKF: public NoiseModelFactor4<Pose3,
    Pose3, Pose3, Pose3> {

  private:
    Pose3 measured_;

  public:
    typedef boost::shared_ptr<VOfactorWithKF>
        shared_ptr;

    VOfactorWithKF(Key j, Key k, Key l, Key m, const
        Pose3& measured, const SharedNoiseModel& model):
    NoiseModelFactor4<Pose3,Pose3, Pose3, Pose3>(model,
         j, k, l, m), measured_(measured) {}

    virtual ~VOfactorWithKF() {}

    Vector evaluateError(const Pose3& pIC, const Pose3&
         pCB, const Pose3& pVI, const Pose3& pKV,
          boost::optional<Matrix&> H1 = boost::none,
          boost::optional<Matrix&> H2 = boost::none,
          boost::optional<Matrix&> H3 = boost::none,
          boost::optional<Matrix&> H4 = boost::none )
              const {

      Matrix Ha, Hb, Hc, Hd, He, Hf;
```

```
    Pose3 expectedKB = pKV.compose(pVI.compose(pIC.
        compose(pCB, Ha, Hb), Hc, Hd), He, Hf);

    if (H1) *H1 = Hf * Hd * Ha;
    if (H2) *H2 = Hf * Hd * Hb;
    if (H3) *H3 = Hf * Hc;
    if (H4) *H4 = He;

    return measured_.localCoordinates(expectedKB);
  }

  virtual gtsam::NonlinearFactor::shared_ptr clone()
      const {
  return boost::static_pointer_cast<gtsam::
      NonlinearFactor>(
    gtsam::NonlinearFactor::shared_ptr(new
        VOfactorWithKF(*this))); }
};
```

BIBLIOGRAPHY

[1] Simon Baker and Iain Matthews. Equivalence and Efficiency of Image Alignment Algorithms. 2001.

[2] Simon Baker and Iain Matthews. Lucas-Kanade 20 Years On : A Unifying Framework. 56(3):221–255, 2004.

[3] Gary Bradski and Adrian Kaehler. *Learning OpenCV*.

[4] Frank Dellaert. Factor Graphs and GTSAM: A Hands-on Introduction. (September):1–27, 2012.

[5] Jakob Engel and Daniel Cremers. Camera-Based Navigation of a Low-Cost Quadrocopter.

[6] D. Fox, J. Hightower, D. Schulz, and G. Borriello. Bayesian filtering for location estimation. *IEEE Pervasive Computing*, 2(3):24–33, July 2003.

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

[8] Gregory D Hager and Peter N Belhumeur. Efficient Region Tracking With Parametric Models of Geometry and Illumination. 20(10):1025–1039, 1998.

[9] Gregory D Hager and Peter N Belhumeur. Efficient Region Tracking With Parametric Models of Geometry and Illumination. 20(10):1025–1039, 1998.

[10] Christoph Hertzberg. A framework for sparse, non-linear least squares problems on manifolds, 2008.

[11] Vadim Indelman, Stephen Williams, Michael Kaess, and Frank Dellaert. Information fusion in navigation systems via factor graph based incremental smoothing. *Robotics and Autonomous Systems*, 61(8):721–738, August 2013.

[12] Vadim Indelman, Stephen Williams, Michael Kaess, Frank Dellaert, and Georgia Ins. Factor Graph Based Incremental Smoothing in Inertial Navigation Systems. (July):1–23, 2012.

[13] Michael Kaess, Viorela Ila, Richard Roberts, and Frank Dellaert. The Bayes Tree: An Algorithmic Foundation for Probabilistic Robot Mapping. pages 1–16, 2011.

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

[15] Christian Kerl. *Odometry from RGB-D Cameras for Autonomous Quadrocopters*. PhD thesis.

[16] Christian Kerl, Jurgen Sturm, and Daniel Cremers. Dense visual SLAM for RGB-D cameras. *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2100–2106, November 2013.

[17] Sven Lange, Peter Protzel, and Information Technology. Incremental sensor fusion in factor graphs with unknown delays. 2013.

[18] Chuanhai Liu and Donald B Rubin. ML estimation of the T distribution using EM and its extensions, ECM and ECME. 5:19–39, 1995.

[19] Todd Lupton and Salah Sukkarieh. Visual-Inertial-Aided Navigation for High-Dynamic Motion in Built Environments Without Initial Conditions. 28(1):61–76, 2012.

[20] Maxime Meilland, Andrew Ian Comport, and Patrick Rives. Dense visual mapping of large scale environments for real-time localisation.

[21] Richard Murray. *A mathematical introduction to robot manipulation*. 1994.

[22] Richard A Newcombe, Steven J Lovegrove, and Andrew J Davison. DTAM: Dense Tracking and Mapping in Real-Time.

[23] Hauke Strasdat, J.M.M. Montiel, and Andrew J. Davison. Visual SLAM: Why filter? *Image and Vision Computing*, 30(2):65–77, February 2012.

[24] Richard Szeliski. Computer Vision : Algorithms and Applications. 2010.

[25] Michael E Tipping. Bayesian Inference: An Introduction to Principles and Practice in Machine Learning. pages 1–19, 2006.

[26] Matthew R Walter, Ryan M Eustice, and John J Leonard. Exactly Sparse Extended Information Filters for Feature-Based SLAM.

[27] Stephan M Weiss. Vision Based Navigation for Micro Helicopters (PhD Thesis - Weiss 2012). (20305), 2012.