robotics

MDPI

Article

Least Squares Optimization: From Theory to Practice

Giorgio Grisetti [1,*], Tiziano Guadagnino [1], Irvin Aloise [1], Mirco Colosi [1,2], Bartolomeo Della Corte [1] and Dominik Schlegel [1]

[1] Department of Computer, Control, and Management Engineering "Antonio Ruberti", Sapienza University of Rome, 00185 Rome, Italy; guadagnino@diag.uniroma1.it (T.G.); ialoise@diag.uniroma1.it (I.A.); colosi@diag.uniroma1.it (M.C.); dellacorte@diag.uniroma1.it (B.D.C.); schlegel@diag.uniroma1.it (D.S.)

[2] Robot Navigation and Perception (CR/AER1), Robert Bosch Corporate Research, 70839 Stuttgart, Germany

* Correspondence: grisetti@diag.uniroma1.it

Received: 25 May 2020; Accepted: 27 June 2020; Published: 1 July 2020

Abstract: Nowadays, Nonlinear Least-Squares embodies the foundation of many Robotics and Computer Vision systems. The research community deeply investigated this topic in the last few years, and this resulted in the development of several open-source solvers to approach constantly increasing classes of problems. In this work, we propose a unified methodology to design and develop efficient Least-Squares Optimization algorithms, focusing on the structures and patterns of each specific domain. Furthermore, we present a novel open-source optimization system that addresses problems transparently with a different structure and designed to be easy to extend. The system is written in modern C++ and runs efficiently on embedded systemsWe validated our approach by conducting comparative experiments on several problems using standard datasets. The results show that our system achieves state-of-the-art performances in all tested scenarios.

Keywords: nonlinear optimization; SLAM; mapping; tutorial

1. Introduction

Iterative Least-Squares (ILS) solvers are core building blocks of many robotic applications, systems, and subsystems [1]. This technique has been traditionally used for calibration [2–4], registration [5–7], and global optimization [8–11]. In particular, modern Simultaneous Localization and Mapping (SLAM) systems typically employ multiple ILS solvers at different levels: in computing the incremental ego-motion of the sensor, in refining the localization of a robot upon loop closure and—most notably—to obtain a globally consistent map. Similarly, in several computer vision systems, ILS is used to compute/refine camera parameters, estimating the structure of a scene, the position of the camera or both. Many inference problems in robotics are effectively described by a factor graph [12], which is a graphical model expressing the joint likelihood of the *known* measurements with respect to a set of unknown conditional variables. Solving a factor graph requires finding the values of the variables that maximize the joint likelihood of the measurements. If the noise affecting the sensor data are Gaussian, the solution of a factor graph can be computed by an ILS solver implementing variants of the well known Gauss–Newton (GN) algorithm.

The relevance of the topic has been addressed by several works such as GTSAM [9], g^2o [8], SLAM++ [13], or the Ceres solver [10] by Google. These systems have grown over time to include comprehensive libraries of factors and variables that can tackle a large variety of problems, and, in most cases, these systems can be used as black boxes. Since they typically consist of an extended codebase, entailing them to a particular application/architecture to achieve maximum performance is a non-trivial task. In contrast, extending these systems to approach new problems is typically easier than customizing: in this case, the developer has to implement some additional functionalities/classes

according to the API of the system. In addition, in this case, however, an optimal implementation might require a reasonable knowledge of the solver internals.

We believe that, at the current time, a researcher working in robotics should possess the knowledge on how to design factor graph solvers for specific problems. Having this skill enables us to both effectively extend existing systems and realize custom software that leverages maximally on the available hardware. Accordingly, the primary goal of this paper is to provide the reader with a methodology on how to mathematically define such a solver for a problem. To this extent in Section 4, we start by revising the nonlinear least squares by highlighting the connections between inference on conditional Gaussian distributions and ILS. In the same section, we introduce the $\boxplus$ (i.e., boxplus) method introduced by Hertzberg et al. [14] to deal with non-Euclidean domains. Furthermore, we discuss how to cope with outliers in the measurements through robust cost functions and we outline the effects of sparsity in factor graphs. We conclude the section by presenting a general methodology on how to design factors and variables that describe a problem. In Section 5, we validate this methodology by providing examples that approach four prominent problems in Robotics: Iterative Closest Point (ICP), projective registration, Bundle Adjustment (BA), and Pose-Graph Optimization (PGO). When it comes to the implementation of a solver, several choices have to be made in the light of the problem structure, the compute architecture and the operating conditions (online or batch). In this work, we characterize ILS problems, distinguishing between dense and sparse, batch and incremental, stationary, and non-stationary based on their structure and application domain. In Section 2, we provide a more detailed description of these characteristics, while, in Section 3, we discuss how ILS has been used in the literature to approach various problems in Robotics and by highlighting how addressing a problem according to its traits leads to effective solutions.

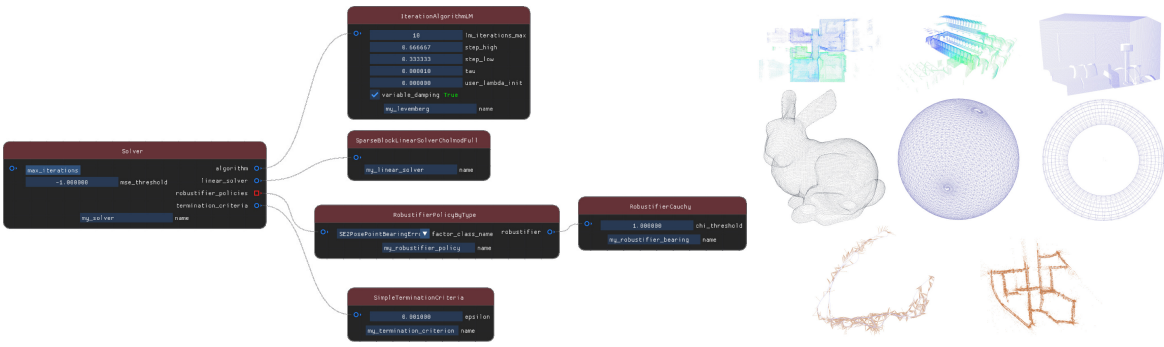
The second orthogonal goal of this work is to propose a unifying system that deals with dense/sparse, stationary/non-stationary, and batch problems, with no apparent performance loss compared to ad-hoc solutions. We build on the ideas that are at the base of the g^2o optimizer [8], in order to address some requirements arising from users and developers, namely: fast convergence, small runtime per iteration, rapid prototyping, trade-off between implementation effort and performances, and, finally, code compactness. In Section 6, we highlight from the general algorithm outlined 4 a set of functionalities that result in a modular, decoupled, and minimal design. This analysis ultimately leads to a modern compact and efficient C++ library released under BSD3 license for ILS of Factor Graphs that relies on a component model, presented in Section 7, that effectively runs on both on `x86-64` and `ARM` platforms (Source code: http://srrg.gitlab.io/srrg2-solver.html). To ease prototyping, we offer an interactive environment to graphically configure the solver (Figure 1a). The core library of our solver consists of no more than 6000 lines of C++ code, whereas the companion libraries implementing a large set of factors and variables for approaching problems -for example, 2D/3D ICP, projective registration, BA, 2D/3D PGO and Pose-Landmark Graph Optimization (PLGO), and many others—is at the time of this writing below 4000 lines. Our system relies on our visual component framework, image processing, and visualization libraries that contain no optimization code and consists of approximately 20,000 lines. To validate our claims, we conducted extensive comparative experiments on publicly available datasets (Figure 1b)—in dense and sparse scenarios. We compared our solver with sparse approaches such as GTSAM, g^2o and Ceres, and with dense ones, such as the well-known PCL library [15]. The experiments presented in Section 8 confirm that our system has performances that are on par with other state-of-the-art frameworks.

Summarizing, the contribution of this work is twofold:

— We present a methodology on how to design a solver for a generic class of problems, and we exemplify such a methodology by showing how it can be used to approach a relevant subset of problems in Robotics.
— We propose an open-source, component-based ILS system that aims to coherently address problems having a different structure, while providing state-of-the-art performances.

2. Taxonomy of ILS Problems

Whereas the theory on ILS is well-known, the effectiveness of an implementation greatly depends on the structure of the problem being addressed and on the operating conditions. We qualitatively distinguish between dense and sparse problems, by discriminating on the connectivity of the factor graph. A dense problem is characterized by many measurements affected by relatively few variables. This occurs in typical registration problems, where the likelihoods of the measurements (for example, the intensities of image pixels) depend on a single variable expressing the sensor position. In contrast, sparse problems are characterized by measurements that depend only on a small subset of variables. Examples of sparse problems include PGO or BA.



(**a**) Graphical solver configurator. (**b**) Datasets used in the evaluation.

Figure 1. (**a**) visual configuration manager implemented in our framework. Each block represents a configurable sub-module. (**b**) dataset used in the evaluation—dense and sparse.

A further orthogonal classification of the problems divides them into *stationary* and *non-stationary*. A problem is stationary when the associations do not change during the iterations of the optimization. This occurs when the data-association is known a priori with sufficient certainty. Conversely, non-stationary problems admit associations that might change during the optimization, as a result of a modification of the variables being estimated. A typical case of a non-stationary problem is point registration [16], when the associations between the points in the model and the references are computed at each iteration based on a heuristic that depends on the current estimate of their displacement.

Finally, the problem might be extended over time by adding new variables and measurements. Several Graph-Based SLAM systems exploit this intrinsic characteristic in online applications, reusing the computation done while solving the original problem to determine the solution for the augmented one. We refer to a solver with this capability as an *incremental* solver, in contrast to *batch* solvers that carry on all the computation from scratch once the factor graph is augmented.

In this taxonomy, we left out other crucial aspects that affect the convergence basin of the solver such as the linearity of the measurement function, or the domain of variables and measurements. Exploiting the structure of these domains has shown to provide even more efficient solutions [17], with the obvious shortcoming that they are restricted to the specific problem they are designed to address.

Using a sparse stationary solver on a dense non-stationary problem results in carrying on useless computation that hinders the usability of the system. Using a dense dynamic solver to approach a sparse stationary problem presents similar issues. State-of-the-art open-source solvers like the ones mentioned in Section 1 focus on sparse stationary or incremental problems. Dense solvers are usually within the application/library using it and tightly coupled to it. On the one hand, this allows reducing the time per iteration, while, on the other hand, it results in avoidable code replication when multiple systems are integrated. This might result in potential inconsistencies among program parts and consequent bugs.

3. Related Work

In this section, we revise the use of ILS in approaching several problems in robotics, in order to highlight the structure and the peculiarities that each problem presents to the solver according to the taxonomy presented in Section 2. Furthermore, we provide an overview of generic sparse solvers that are commonly used nowadays for factor graph optimization.

3.1. ILS in Robotics

In calibration, ILS has been used extensively since the first works appeared until these days [18–20]. Common works in batch calibration involve relatively small state spaces covering only the parameters to be estimated. Since these parameters condition directly or indirectly all measurements, this class of problems typically requires a dense stationary solver. When temporal calibration is required, however, the changing time offset might result in considering different data chunks at each iterations, thus requiring a dense, non-stationary solver, such as the one presented in [4].

Among the first works on pairwise shape registration relying on ILS, we find the ICP proposed by Besl and McKay [16], while Chen and Medioni [21] proposed the first ILS method for the incremental reconstruction of a 3D model from multiple range images. These methods constitute the foundation of many registration algorithms appearing during the subsequent years. In particular, Lu and Milios [22] specialized ICP to operate on 2D laser scans. All of these works employed dense non-stationary solvers, to estimate the robot pose that better explains the point measurements. The non-stationary aspect arises from the heuristic used to estimate the data association is based on the current pose estimate.

In the context of ICP, Censi [23] proposed an alternative metric to compute the distance between two points and an approach to estimate the information matrix from the set of correspondences [24]. Subsequently, Segal et al. [25] proposed the use of covariance matrices that better reflect the structure of the surface in computing the error. Registration has been addressed by Bieber et al. [26] for 2D scans and subsequently Magnusson et al. [27] for 3D point clouds by using a pure Newton's method relying on a Gaussian approximation of the point clouds to be registered, called Normal Distributed Transform (NDT). Serafin et al. [7] approached the problem of point cloud registration using a 6D error function encoding also the normal difference in the error vector. All of the approaches mentioned so far leverage on a dense ILS solver, with the notable exception of NDT that is a second-order approach that specializes the Newton's algorithm.

In the context of Computer Vision, the PnP algorithm [28,29] allows for finding the camera transformation that minimizes the reprojection error between a set of 3D points in the scene and their 2D projections in the image. The first stage of PnP is usually conducted according to a consensus scheme that relies on an ad-hoc minimal solver requiring only three correspondences. The final stage, however, typically uses a dense and stationary ILS approach, since the correspondences do not change during the iterations. When the initial guess of the camera is known with sufficient accuracy, like, in Visual Odometry (VO), only the latter stage is typically used. In contrast to these feature-based solvers, Engel et al. [30] approach VO by minimizing the reprojection error between two images through dense and non-stationary ILS. Using this method requires the system to possess a reasonably good estimate of the depth for a subset of the point in the scene. Such initialization is usually obtained by estimating the transformation between two images using a combination of RANSAC and direct solvers, and then computing the depth through triangulation between the stereo pair. Della Corte et al. [31] developed a registration algorithm called MPR, which was built on this idea. As a result, MPR is able to operate on depth images capturing different cues and obtained with arbitrary projection functions. To operate online, all the registration works mentioned so far rely on ad-hoc dense and non-stationary ILS solvers that exploit the specific problem's structure to reduce the computation.

The scan based ICP algorithm [22] has been subsequently employed by the same authors [32] as a building block for a system that estimates a globally consistent map. The core is to determine the relative transforms between pairwise scans through ICP. These transformations are known as

constraints, and a global map is obtained by finding the position of all the scans that better explain the constraints. The process can be visualized as a graph, whose nodes are the scan positions and whose edges are the constraints, hence this problem is called PGO. Constraints can exist only between spatially close nodes, due to the limited sensor range. Hence, PGO is inherently sparse. Additionally, in the online case, the graph is incrementally augmented as new measurements become available, rendering it incremental. We are unaware on these two aspects being exploited in the design of the underlying solver in [32]. The work of Lu and Milios inspired Borrman et al. [33] to produce an effective 3D extension.

For several years after the introduction of Graph-Based SLAM [32], the community put aside ILS approaches in favor of filtering methods relying on Gaussian [34–39] or Particle [40–43] representation of the posterior. Filtering approaches were preferred since they were regarded as more suitable to be run online on a moving robot with the available computational resources of the era, and the sparsity of the problem had not yet been fully exploited.

In a Graph-Based SLAM problem, it is common to have a number of variables in the order of hundreds or thousands. Such a high number of variables results in a large optimization problem that represented a challenge for the computers of the time, rendering global optimization a bottleneck of Graph-Based SLAM systems. In the remainder of this document, we will refer to the global optimization module in Graph-SLAM as the *back-end*, in contrast to the *front-end*, which is responsible to construct the factor graph based on the sensor measurements. Gutmann and Konolidge [44] addressed the problem of incrementally building a map, by finding topological relations and loop closures. This work was one of the first online implementations of Graph-Based SLAM. The core idea to reduce the computation in the back-end was to restrict the optimization to the portions of the graph having the larger errors. This insight has inspired several subsequent works [13,45].

3.2. Stand-Alone ILS Solvers

Whereas dense solvers are typically embedded in the specific application for performance reasons, sparse solvers are complex enough to motivate the design of generic libraries for ILS. The first work to explicitly consider the sparsity of SLAM in conjunction with a direct method to solve the linear system was $\sqrt{\text{SAM}}$, developed by Dallaert et al. [46]. Kaess et al. [45] exploited this aspect of the problem in iSAM, the second iteration of $\sqrt{\text{SAM}}$. Here, when a new edge is added to the graph, the system computes a new solution reusing part of the previous one and selectively updating the vertices. In the third iteration of the system—iSAM2—Kaess et al. [47] exploited the Bayes Tree to solve the optimization problem without explicitly constructing the linear system. This solution is in contrast with the general trend of decoupling linearization of the problem and solution of the linear system and highlights the connections between the elimination algorithms used in the solution of a linear system and inference on graphical models. This self-contained engine allows for dealing very efficiently with a dynamic graph that grows during time and Gaussian densities, two typical features of the SLAM problem. The final iteration of the system, called GTSAM [9], embeds all this concepts in a single framework.

Meanwhile, Hertzberg with his thesis [48] introduced the $\boxplus$ method to systematically deal with non-Euclidean spaces and sparse problems. This work has been at the root of the framework of Kümmerle et al. [8]—called g^2o. This system introduces a layered architecture that allows for easily exchanging sub-modules of the system—for example, the linear solver or optimization algorithm. A further paper of Hertzberg et al. [14] extends the $\boxplus$ method ILS to filtering.

Agarwal et al. proposed in their Ceres Solver [10] a generalized framework to perform nonlinear optimization. Ceres embeds state-of-the-art methodologies that take advantages of modern CPUs—for example, SIMD instructions and multi-threading—resulting in a complete and fast framework. One of its most relevant features is represented by the efficient use of Automatic Differentiation (AD) [49] that consists of the algorithmic computation of derivatives starting from the error function. Further information on the topic of AD can be found in [50,51].

In several contexts, knowing the optimal value of a solution is not sufficient, and also the covariance is required. In SLAM, knowing the marginal covariances relative to a variable is fundamental to approach data-association. To this extent, Kaess et al. [52] outlined the use of the elimination tree. Subsequently, Ila et al. [13] designed SLAM++, an optimization framework to estimate mean and covariance of the state by performing incremental Cholesky updates. This work takes advantage of the incremental aspect of the problem to selectively update the approximated Hessian matrix by using parallel computation.

ILS algorithms have several known drawbacks. Perhaps the most investigated aspect is the sensitivity of the solution to the initial guess that is reflected by the convergence basin. A wrong initial guess might lead a nonlinear solver to converge to an inconsistent local minimum. Convex optimization [53] is one of the possible strategies to overcome this problem; however, its use is highly domain dependent. Rosen et al. [17] explored this topic, proposing a system to perform optimization of generic $\mathrm{SE}(d)$ factor graphs. In their system, called SE-Sync, they use a Riemannian Truncated-Newton Trust-Region method to certifiably compute the global optimum in a two-step optimization (rotation and translation). Briales et al. [54] extended this approach to jointly optimize rotation and translation using the same concepts. Bai et al. [55] provided a formulation of the SLAM problem based on *constrained optimization*, where constraints are represented by loop-closure cycles. Still, those approaches are bounded to $\mathrm{SE}(d)$ sparse optimization. In contrast, Ni et al. [56] and Grisetti et al. [57] exploited respectively nested dissection and hierarchical local sub-graphs devising divide and conquer strategies to both increase the convergence basin and speed up the computation. More recently, Yang et al. [58] proposed a general approach to render non-minimal solvers more resilient to outliers. Their work exploits Graduated Non-Convexity (GNC) to solve the optimization problem without requiring a good initial guess.

4. Least Squares Minimization

This section describes the foundations of ILS minimization. We first present a formulation of the problem that highlights its probabilistic aspects (Section 4.1). In Section 4.2, we review some basic rules for manipulating the Normal distribution and we apply these rules to the definition presented in Section 4.1, leading to the initial definition of linear Least-Squares (LS). In Section 4.3, we discuss the effects of a nonlinear observation model, assuming that both the state space and the measurements space are Euclidean. Subsequently, we relax this assumption on the structure of state and measurement spaces, proposing a solution that uses smooth manifold encapsulation. In Section 4.4, we introduce effects of outliers in the optimization and we show commonly used methodologies to reject them. Finally, in Section 4.5, we address the case of a large, sparse problem characterized by measurement functions depending only on small subsets of the state. Classical problems such as SLAM or BA fall in this category and are characterized by a rather sparse structure.

4.1. Problem Formulation

Let $\mathcal{W}$ be a stationary system whose non-observable state variable is represented by $\mathbf{x}$ and let $\mathbf{z}$ be a measurement, i.e., a perception of the environment. The state is distributed according to a prior distribution $p(\mathbf{x})$, while the conditional distribution of the measurement given the state $p(\mathbf{z}|\mathbf{x})$ is known. $p(\mathbf{z}|\mathbf{x})$ is commonly referred to as the *observation model*. Our goal is to find the most likely distribution of states, given the measurements—i.e., $p(\mathbf{x}|\mathbf{z})$. A straightforward application of the Bayes rule results in the following:

$$p(\mathbf{x}|\mathbf{z}) = \frac{p(\mathbf{x})p(\mathbf{z}|\mathbf{x})}{p(\mathbf{z})} \propto p(\mathbf{x})\, p(\mathbf{z}|\mathbf{x}). \tag{1}$$

The proportionality is a consequence of the normalization factor $p(\mathbf{z})$, which is constant.

In the remainder of this work, we will consider two key assumptions:

– the prior about the states is uniform, i.e.,

$$p(\mathbf{x}) = \mathcal{N}(\mathbf{x}; \mu_x, \mathbf{\Sigma}_x = \inf) = \mathcal{N}(\mathbf{x}; \nu_x, \mathbf{\Omega}_x = 0), \tag{2}$$

– the observation model is Gaussian, i.e.,

$$p(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}; \mu_{z|x}, \mathbf{\Omega}_{z|x}^{-1}) \quad \text{where} \quad \mu_{z|x} = \mathbf{h}(\mathbf{x}). \tag{3}$$

Equation (2) expresses the uniform prior about the states using the canonical parameterization of the Gaussian. Alternatively, the moment parameterization characterizes the Gaussian by the information matrix—i.e., the inverse of the covariance matrix $\mathbf{\Omega}_x = \mathbf{\Sigma}_x^{-1}$—and the information vector $\nu_x = \mathbf{\Omega}_x \mu_x$. The canonical parameterization is better suited to represent a non-informative prior, since $\mathbf{\Omega}_x = 0$ does not lead to numerical instabilities while implementing the algorithm. In contrast, the moment parameterization can express in a stable manner situations of absolute certainty by setting $\mathbf{\Sigma}_x = 0$. In the remainder, we will use both representations upon convenience, their relation being clear.

In Equation (3), the mean $\mu_{z|x}$ of the predicted measurement distribution is controlled by a generic nonlinear function of the state $\mathbf{h}(\mathbf{x})$, commonly referred to as *measurement function*. In the next section, we will derive the solution for Equation (1), imposing that the measurement function is an *affine transformation* of the state—i.e., $\mathbf{h}(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}$—illustrated in Figure 2. In Section 4.3, we address the more general nonlinear case.

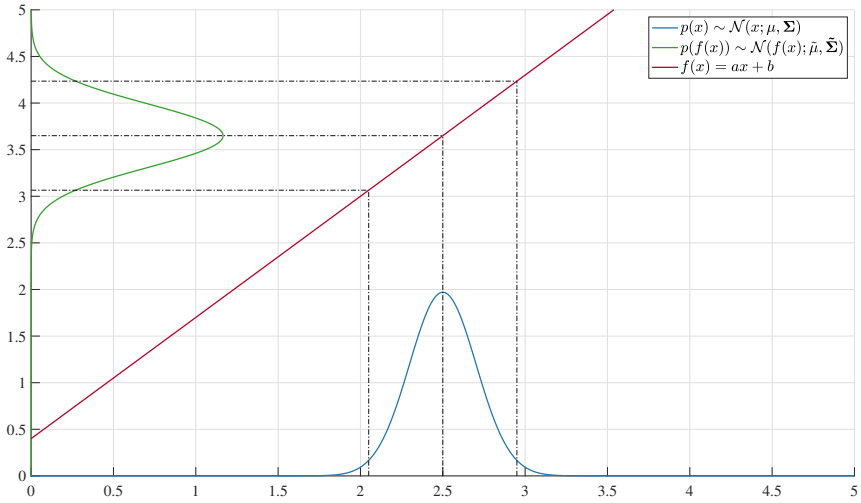


Figure 2. Affine transformation of a uni-variate Gaussian distribution. The blue curve represents the source PDF while in green we show the output PDF. The red line represents the affine transformation.

4.2. Linear Measurement Function

In the case of linear measurement function, expressed as $\mathbf{h}(\mathbf{x}) = \mathbf{A}(\mathbf{x} - \mu_x) + \hat{\mathbf{z}}$, the prediction model has the following form:

$$\begin{aligned} p(\mathbf{z}|\mathbf{x}) &= \mathcal{N}(\mathbf{z}; \mu_{z|x} = \mathbf{A}(\overbrace{\mathbf{x} - \mu_x}^{\mathbf{\Delta x}}) + \hat{\mathbf{z}}, \mathbf{\Omega}_{z|x}^{-1}) \\ p(\mathbf{z}|\mathbf{\Delta x}) &= \mathcal{N}(\mathbf{z}; \mu_{z|\Delta x} = \mathbf{A}\mathbf{\Delta x} + \hat{\mathbf{z}}, \mathbf{\Omega}_{z|x}^{-1}), \end{aligned} \tag{4}$$

with $\hat{\mathbf{z}}$ constant and μ_x being the mean of the prior. For convenience, we express the stochastic variable $\mathbf{\Delta x} = \mathbf{x} - \mu_x$ as

$$p(\mathbf{\Delta x}) = \mathcal{N}(\mathbf{\Delta x}; 0, \mathbf{\Sigma}_x = \inf) = \mathcal{N}(\mathbf{x}; \nu_{\Delta x}, \mathbf{\Omega}_x = 0). \tag{5}$$

Switching between $\mathbf{\Delta x}$ and $\mathbf{x}$ is achieved by summing or subtracting the mean. To retrieve a solution for Equation (1), we first compute the joint probability of $p(\mathbf{z}, \mathbf{\Delta x})$ using the chain rule and, subsequently, we condition this joint distribution with respect to the known measurement $\mathbf{z}$. For further details on the Gaussian manipulation, we refer the reader to [59].

4.2.1. Chain Rule

Under the Gaussian assumptions made in the previous section, the parameters of the joint distribution over states and measurements $p(\mathbf{\Delta x}, \mathbf{z})$ have the following block structure:

$$p(\mathbf{\Delta x}, \mathbf{z}) = \mathcal{N}\left(\mathbf{\Delta x}, \mathbf{z}; \mu_{\Delta x,z}, \mathbf{\Omega}^{-1}_{\Delta x,z}\right) \tag{6}$$

$$\mu_{\Delta x,z} = \begin{pmatrix} \mathbf{0} \\ \hat{\mathbf{z}} \end{pmatrix} \qquad \mathbf{\Omega}_{\Delta x,z} = \begin{pmatrix} \mathbf{\Omega}_{xx} & \mathbf{\Omega}_{xz} \\ \mathbf{\Omega}^T_{xz} & \mathbf{\Omega}_{zz} \end{pmatrix}. \tag{7}$$

The value of the terms in Equation (7) are obtained by applying the chain rule to multivariate Gaussians to Equations (4) and (5), according to [59], and they result in the following:

$$\begin{aligned} \mathbf{\Omega}_{xx} &= \mathbf{A}^T \mathbf{\Omega}_{z|x} \mathbf{A} + \mathbf{\Omega}_x \\ \mathbf{\Omega}_{xz} &= -\mathbf{A}^T \mathbf{\Omega}_{z|\Delta x} \\ \mathbf{\Omega}_{zz} &= \mathbf{\Omega}_{z|x}. \end{aligned}$$

Since we assumed the prior to be non-informative, we can set $\mathbf{\Omega}_x = 0$. As a result, the information vector $\nu_{\Delta x,z}$ of the joint distribution is computed as:

$$\nu_{\Delta x,z} = \begin{pmatrix} \nu_{\Delta x} \\ \nu_z \end{pmatrix} = \mathbf{\Omega}_{\Delta x,z}\, \mu_{\Delta x,z} = \begin{pmatrix} -\mathbf{A}^T \mathbf{\Omega}_{z|x} \hat{\mathbf{z}} \\ \mathbf{\Omega}_{z|x} \hat{\mathbf{z}} \end{pmatrix}. \tag{8}$$

Figure 3 shows visually the outcome distribution.

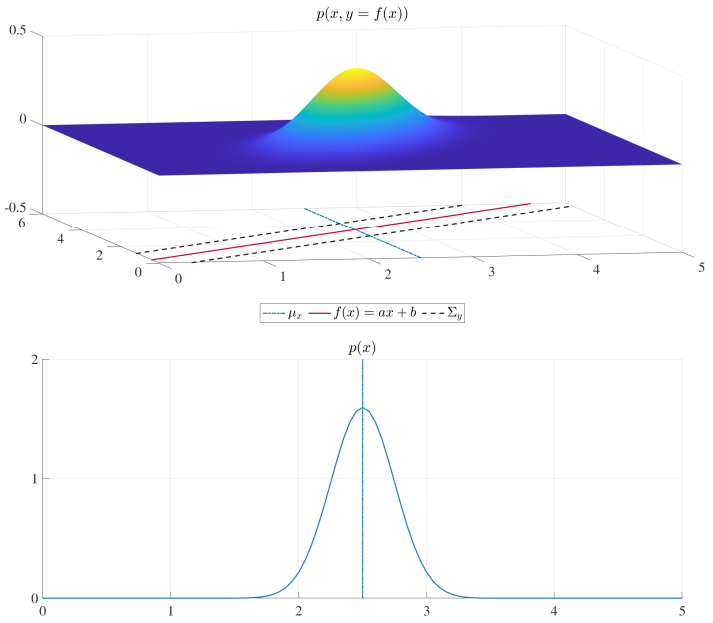


Figure 3. Given a uni-variate Gaussian PDF $p(x)$ and an affine trasformation $f(x)$—indicated in red—we computed the joint distribution $p(x, y = f(x))$ through the chain rule.

4.2.2. Conditioning

Integrating the known measurement $\mathbf{z}$ in the joint distribution $p(\mathbf{\Delta x}, \mathbf{z})$ of Equation (6) results in a new distribution $p(\mathbf{\Delta x}|\mathbf{z})$. This can be done by conditioning in the Gaussian domain. Once again, we refer the reader to [59] for the proofs, while we report here the results that the conditioning has on the Gaussian parameters:

$$p(\mathbf{\Delta x}|\mathbf{z}) \sim \mathcal{N}\left(\mathbf{\Delta x}; \nu_{\Delta x|z}, \mathbf{\Omega}_{\Delta x|z}\right) \tag{9}$$

where

$$\nu_{\Delta x|z} = \nu_{\Delta x} - \mathbf{\Omega}_{xz}\mathbf{z} = \nu_{\Delta x} - (-\mathbf{A}^\top \mathbf{\Omega}_{z|x})\mathbf{z} = \mathbf{A}^\top \mathbf{\Omega}_{z|x}(\underbrace{\mathbf{z} - \hat{\mathbf{z}}}_{-\mathbf{e}}) \tag{10}$$

$$\mathbf{\Omega}_{\Delta x|z} = \mathbf{\Omega}_{zz} = \underbrace{\mathbf{A}^\top \mathbf{\Omega}_{z|x}\mathbf{A}}_{\mathbf{H}}. \tag{11}$$

The conditioned mean $\mu_{\Delta x|z}$ is retrieved from the information matrix and the information vector as:

$$\mu_{\Delta x|z} = \mathbf{\Omega}^{-1}_{\Delta x|z}\nu_{x|z} = -\mathbf{H}^{-1}\underbrace{\mathbf{A}^\top \mathbf{\Omega}_{z|x}\mathbf{e}}_{\mathbf{b}} = -\mathbf{H}^{-1}\mathbf{b}. \tag{12}$$

Remembering that $\mathbf{\Delta x} = \mathbf{x} - \mu_x$, the Gaussian distribution over the conditioned states has the same information matrix, while the mean is obtained by summing the increment's mean $\mu_{\Delta x|z}$ as

$$\mu_{x|z} = \mu_x + \mu_{\Delta x|z}. \tag{13}$$

An important result in this derivation is that the matrix $\mathbf{H} = \mathbf{\Omega}_{\Delta x|z}$ is the information matrix of the estimate; therefore, we can estimate not only the optimal solution $\mu_{x|z}$, but also its uncertainty $\mathbf{\Sigma}_{x|z} = \mathbf{\Omega}^{-1}_{\Delta x|z}$. Figure 4 illustrates visually the conditioning of a bi-variate Gaussian distribution.

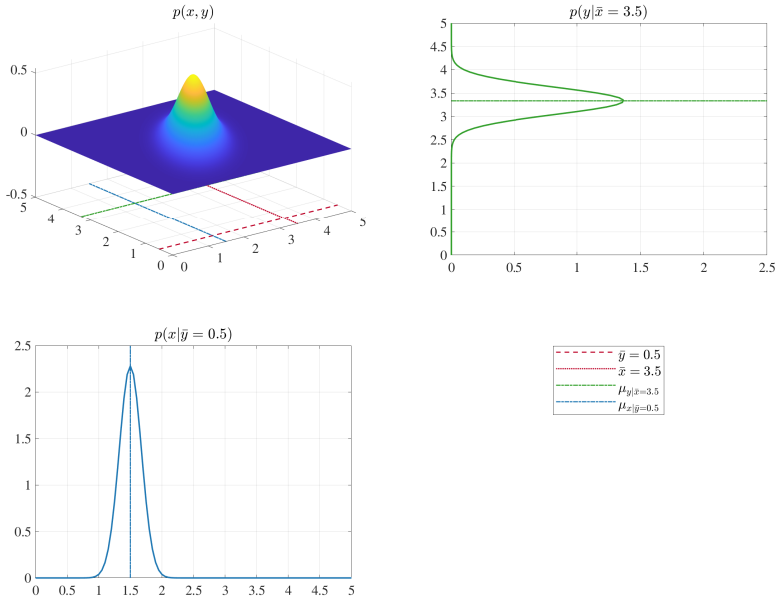


Figure 4. Conditioning of a bi-variate Gaussian. Top left: the source PDF; top right and bottom left indicate the conditioning over x and y respectively.

4.2.3. Integrating Multiple Measurements

Integrating multiple independent measurements $\mathbf{z}_{1:K}$ requires stacking them in a single vector. As a result, the observation model becomes

$$p(\mathbf{z}_{1:K}|\Delta\mathbf{x}) = \prod_{k=1}^{K} p(\mathbf{z}_k|\Delta\mathbf{x}) \sim \mathcal{N}(\mathbf{z}; \mu_{z|x}, \mathbf{\Omega}_{z|x}) = \tag{14}$$
$$= \left(\begin{bmatrix} \mathbf{z}_1 \\ \vdots \\ \mathbf{z}_K \end{bmatrix} ; \begin{bmatrix} \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_K \end{bmatrix} \Delta\mathbf{x} + \begin{bmatrix} \hat{\mathbf{z}}_1 \\ \vdots \\ \hat{\mathbf{z}}_K \end{bmatrix}, \begin{bmatrix} \mathbf{\Omega}_{z_1|x} & & \\ & \ddots & \\ & & \mathbf{\Omega}_{z_K|x} \end{bmatrix} \right).$$

Hence, matrix $\mathbf{H}$ and vector $\mathbf{b}$ are composed by the sum of each measurement's contribution; setting $\mathbf{e}_k = \hat{\mathbf{z}}_k - \mathbf{z}_k$, we compute them as follows:

$$\mathbf{H} = \sum_{k=1}^{K} \underbrace{\mathbf{A}_k^\top \mathbf{\Omega}_{z_k|x} \mathbf{A}_k}_{\mathbf{H}_k} \qquad \mathbf{b} = \sum_{k=1}^{K} \underbrace{\mathbf{A}_k^\top \mathbf{\Omega}_{z_k|x} \mathbf{e}_k}_{\mathbf{b}_k}. \tag{15}$$

4.3. Nonlinear Measurement Function

Equations (12), (13) and (15) allow us to find the exact mean of the conditional distribution, under the assumptions that (i) the measurement noise is Gaussian, (ii) the measurement function is an affine transform of the state, and (iii) both measurement and state spaces are Euclidean. In this section, we first relax the assumption on the affinity of the measurement function, leading to the common derivation of the GN algorithm. Subsequently, we address the case of non-Euclidean state and measurement spaces.

If the measurement model mean $\mu_{z|x}$ is controlled by a nonlinear but *smooth* function $\mathbf{h}(\mathbf{x})$, and that the prior mean $\mu_x = \check{\mathbf{x}}$ is reasonably close to the optimum, we can approximate the behavior of $\mu_{x|z}$ through the first-order Taylor expansion of $\mathbf{h}(\mathbf{x})$ around the mean, namely:

$$\mathbf{h}(\check{\mathbf{x}} + \Delta\mathbf{x}) \approx \underbrace{\mathbf{h}(\check{\mathbf{x}})}_{\hat{\mathbf{z}}} + \underbrace{\left.\frac{\partial \mathbf{h}(\mathbf{x})}{\partial \mathbf{x}}\right|_{\mathbf{x}=\check{\mathbf{x}}}}_{\mathbf{J}} \Delta\mathbf{x} = \mathbf{J}\Delta\mathbf{x} + \hat{\mathbf{z}}. \tag{16}$$

The Taylor expansion reduces conditional mean to an affine transform in $\Delta\mathbf{x}$. Whereas the conditional distribution will not be in general Gaussian, the parameters of a Gaussian approximation can still be obtained around the optimum through Equations (11) and (13). Thus, we can use the same algorithm described in Section 4.2, but we have to compute the linearization at each step. Summarizing, at each iteration, the GN algorithm:

- processes each measurement $\mathbf{z}_k$ by evaluating error $\mathbf{e}_k(\mathbf{x}) = \mathbf{h}_k(\mathbf{x}) - \mathbf{z}_k$ and Jacobian $\mathbf{J}_k$ at the current solution $\check{\mathbf{x}}$:

$$\mathbf{e}_k = \mathbf{h}(\check{\mathbf{x}}) - \mathbf{z} \tag{17}$$
$$\mathbf{J}_k = \left.\frac{\partial \mathbf{h}_k(\mathbf{x})}{\partial \mathbf{x}}\right|_{\mathbf{x}=\check{\mathbf{x}}}. \tag{18}$$

- builds a coefficient matrix and coefficient vector for the linear system in Equation (12), and computes the optimal perturbation $\Delta\mathbf{x}$ by solving a linear system:

$$\Delta\mathbf{x} = -\mathbf{H}^{-1}\mathbf{b} \qquad \mathbf{H} = \sum_{k=1}^{K} \mathbf{J}_k^\top \mathbf{\Omega}_{z_k|x} \mathbf{J}_k \qquad \mathbf{b} = \sum_{k=1}^{K} \mathbf{J}_k^\top \mathbf{\Omega}_{z_k|x} \mathbf{e}_k \tag{19}$$

– applies the computed perturbation to the current state as in Equation (13) to get an improved estimate

$$\check{\mathbf{x}} \leftarrow \check{\mathbf{x}} + \boldsymbol{\Delta}\mathbf{x}. \tag{20}$$

A smooth prediction function has lower-magnitude higher order terms in its Taylor expansion. The smaller these terms are, the better its linear approximation will be. This leads to the situations close to the ideal affine case. In general, the *smoother* the measurement function $\mathbf{h}(\mathbf{x})$ is and the closer the initial guess is to the optimum, the better the convergence properties of the problem.

Non-Euclidean Spaces

The previous formulation of GN algorithm uses vector addition and subtraction to compute the error $\mathbf{e}_k$ in Equation (17) and to apply the increments in Equation (20). However, these two operations are only defined in Euclidean spaces. When this assumption is violated—as it usually happens in Robotics and Computer Vision applications—the straightforward implementation does not generally provide satisfactory results. Rotation matrices or angles cannot be directly added or subtracted without performing subsequent non-trivial normalization. Still, typical continuous states involving rotational or similarity transformations are known to lie on a smooth manifold [60].

A smooth manifold $\mathbb{M}$ is a space that, albeit not homeomorphic to $\mathbb{R}^n$, admits a locally Euclidean parameterization around each element $\mathbf{M}$ of the domain, commonly referred to as *chart*—as illustrated in Figure 5. A chart computed in a manifold point $\mathbf{M}$ is a function from $\mathbb{R}^n$ to a new point $\mathbf{M}'$ on the manifold:

$$\text{chart}_{\mathbf{M}}(\boldsymbol{\Delta}\mathbf{m}) : \mathbb{R}^n \rightarrow \mathbb{M}. \tag{21}$$

Intuitively, $\mathbf{M}'$ is obtained by "walking" along the perturbation $\boldsymbol{\Delta}\mathbf{m}$ on the chart, starting from the origin. A null motion ($\boldsymbol{\Delta}\mathbf{m} = \mathbf{0}$) on the chart leaves us at the point where the chart is constructed—i.e., $\text{chart}_{\mathbf{M}}(\mathbf{0}) = \mathbf{M}$.

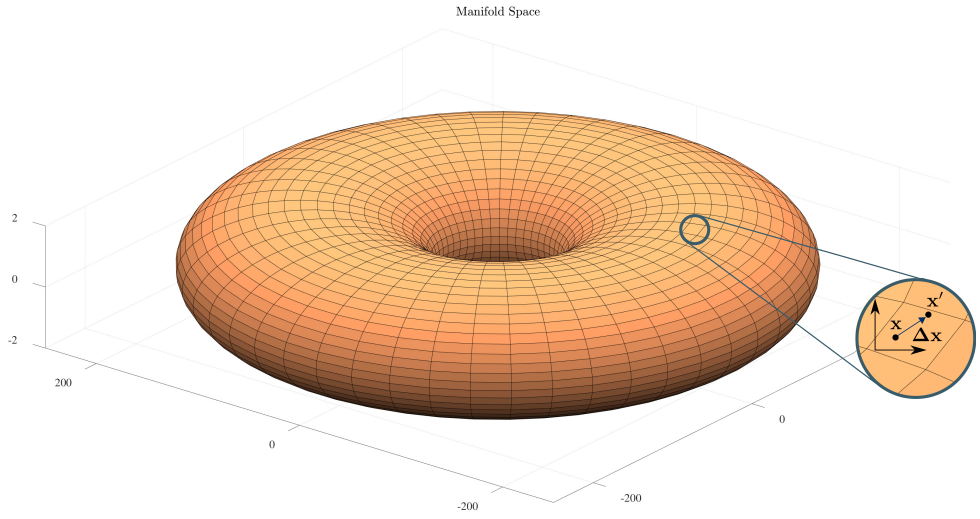


Figure 5. Illustration of a manifold space. Since the manifold is smooth, *local* perturbations—i.e., $\boldsymbol{\Delta}\mathbf{x}$ in the illustration—can be expressed with a suitable Euclidean vector.

Similarly, given two points $\mathbf{M}$ and $\mathbf{M}'$ on the manifold, we can determine the motion $\boldsymbol{\Delta}\mathbf{m}$ on the chart constructed on $\mathbf{M}$ that would bring us to $\mathbf{M}'$. Let this operation be the inverse chart, denoted as

$\text{chart}_{\mathbf{M}}^{-1}(\mathbf{M}')$. The direct and inverse charts allow us to define operators on the manifold that are analogous to the sum and subtraction. Those operators, referred to as $\boxplus$ and $\boxminus$, are, thence, defined as:

$$\mathbf{M} = \mathbf{M} \boxplus \boldsymbol{\Delta}\mathbf{m} \triangleq \text{chart}_{\mathbf{M}}(\boldsymbol{\Delta}\mathbf{m}) \tag{22}$$

$$\boldsymbol{\Delta}\mathbf{m} = \mathbf{M}' \boxminus \mathbf{M} \triangleq \text{chart}_{\mathbf{M}}^{-1}(\mathbf{M}'). \tag{23}$$

This notation—firstly introduced by Smith et al. [61] and then generalized by Hertzberg et al. [14,62]—allows us to straightforwardly adapt the Euclidean version of ILS to operate on manifold spaces. The dimension of the chart is chosen to be the minimal needed to represent a generic perturbation on the manifold. On the contrary, the manifold representation can be chosen arbitrarily.

A typical example of smooth manifold is the $SO(3)$ domain of 3D rotations. We represent an element $SO(3)$ on the manifold as a rotation matrix $\mathbf{R}$. In contrast, the for perturbation, we pick a minimal representation consisting on the three Euler angles $\boldsymbol{\Delta}\mathbf{r} = (\Delta\phi, \Delta\theta, \Delta\psi)^\top$. Accordingly, the operators become:

$$\mathbf{R}_A \boxplus \boldsymbol{\Delta}\mathbf{r} = \text{fromVector}(\boldsymbol{\Delta}\mathbf{r})\, \mathbf{R}_A \tag{24}$$

$$\mathbf{R}_A \boxminus \mathbf{R}_B = \text{toVector}(\mathbf{R}_B^{-1}\, \mathbf{R}_A). \tag{25}$$

The function $\text{fromVector}(\cdot)$ computes a rotation matrix as the composition of the rotation matrices relative to each Euler angle. In formulae:

$$\mathbf{R} = \text{fromVector}(\boldsymbol{\Delta}\mathbf{r}) = \mathbf{R}_x(\Delta\phi)\, \mathbf{R}_y(\Delta\theta)\, \mathbf{R}_z(\Delta\psi). \tag{26}$$

The function $\text{toVector}(\cdot)$ does the opposite by computing the value of each Euler angle starting from the matrix $\mathbf{R}$. It operates by equating each of its element to the corresponding one in the matrix product $\mathbf{R}_x(\Delta\phi)\, \mathbf{R}_y(\Delta\theta)\, \mathbf{R}_z(\Delta\psi)$, and by solving the resulting set of trigonometric equations. As a result, this operation is quite articulated. Around the origin the chart constructed in this manner is immune to singularities.

Once proper $\boxplus$ and $\boxminus$ operators are defined, we can reformulate our minimization problem in the manifold domain. To this extent, we can simply replace the $+$ with a $\boxplus$ in the computation of the Taylor expansion of Equation (16). Since we will compute an increment on the chart, we need to compute the expansion on the chart $\boldsymbol{\Delta}\mathbf{x}$ at the local optimum that is at the origin of the chart itself $\boldsymbol{\Delta}\mathbf{x} = 0$, in formulae:

$$\mathbf{h}_k(\breve{\mathbf{X}} \boxplus \boldsymbol{\Delta}\mathbf{x}) \approx \mathbf{h}_k(\breve{\mathbf{X}}) + \underbrace{\left.\frac{\partial \mathbf{h}_k(\breve{\mathbf{X}} \boxplus \boldsymbol{\Delta}\mathbf{x})}{\partial \boldsymbol{\Delta}\mathbf{x}}\right|_{\boldsymbol{\Delta}\mathbf{x}=0}}_{\breve{\mathbf{J}}_k} \boldsymbol{\Delta}\mathbf{x}. \tag{27}$$

The same holds when applying the increments in Equation (20), leading to:

$$\breve{\mathbf{X}} \leftarrow \breve{\mathbf{X}} \boxplus \boldsymbol{\Delta}\mathbf{x}. \tag{28}$$

Here, we denoted with capital letters the manifold representation of the state $\mathbf{X}$, and with $\boldsymbol{\Delta}\mathbf{x}$ the Euclidean perturbation. Since the optimization *within one iteration* is conducted on the chart, the origin of the chart $\breve{\mathbf{X}}$ on the manifold stays constant during this iteration.

If the measurements lie on a manifold too, a local $\boxminus$ operator is required to compute the error, namely:

$$\mathbf{e}_k(\mathbf{X}) = \hat{\mathbf{Z}}_k \boxminus \mathbf{Z}_k = \mathbf{h}_k(\mathbf{X}) \boxminus \mathbf{Z}_k. \tag{29}$$

To apply the previously defined optimization algorithm, we should linearize the error around the current estimate through its first-order Taylor expansion. Posing $\check{\mathbf{e}}_k = \mathbf{e}_k(\check{\mathbf{X}})$, we have the following relation:

$$\mathbf{e}_k(\check{\mathbf{X}} \boxplus \boldsymbol{\Delta}\mathbf{x}) = \mathbf{h}_k(\check{\mathbf{X}}) \boxminus \mathbf{Z}_k \approx \check{\mathbf{e}}_k + \left.\frac{\partial \mathbf{e}_k(\check{\mathbf{X}} \boxplus \boldsymbol{\Delta}\mathbf{x})}{\partial \boldsymbol{\Delta}\mathbf{x}}\right|_{\boldsymbol{\Delta}\mathbf{x}=0} \boldsymbol{\Delta}\mathbf{x} = \check{\mathbf{e}}_k + \check{\mathbf{J}}_k \boldsymbol{\Delta}\mathbf{x}. \tag{30}$$

The reader might notice that, in Equation (30), the error space may differ from the increments one, due to the $\boxminus$ operator. As reported in [63], having a different parametrization might enhance the convergence properties of the optimization in specific scenarios. Still, to avoid any inconsistencies, the information matrix $\boldsymbol{\Omega}_k$ should be expressed on a chart around the current measurement $\mathbf{Z}_k$.

4.4. Handling Outliers: Robust Cost Functions

In Section 4.3, we described a methodology to compute the parameters of the Gaussian distribution over the state $\mathbf{x}$, which minimizes the Omega-norm of the error between prediction and observation. More concisely, we compute the optimal state $\mathbf{x}^\star$ such that:

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmin}} \sum_{k=1}^{K} \|\mathbf{e}_k(\mathbf{x})\|^2_{\boldsymbol{\Omega}_k}. \tag{31}$$

The mean of our estimate $\mu_{x|z} = \mathbf{x}^*$ is the local optimum of the GN algorithm, and the information matrix $\boldsymbol{\Omega}^*_{x|z} = \mathbf{H}^*$ is the coefficient matrix of the system at convergence. The procedure reported in the previous section assumes all measurements *correct*, albeit affected by noise. Still, in many real cases, this is not the case. This is mainly due to aspects that are hard to model—i.e., multi-path phenomena or incorrect data associations. These *wrong* measurements are commonly referred to as *outliers*. On the contrary, *inliers* represent the *good* measurements.

A common assumption made by several techniques to reject outliers is that the inliers tend to agree towards a common solution, while outliers do not. This fact is at the root of consensus schemes such as RANSAC [29]. In the context of ILS, the quadratic nature of the error terms leads to over-accounting for measurement whose error is large, albeit those measurements are typically outliers. However, there are circumstances where all errors are quite large even if no outliers are present. A typical case occurs when we start the optimization from an initial guess which is far from the optimum.

A possible solution to this issue consists of carrying on the optimization under a cost function that grows sub-quadratically with the error. Indicating with $u_k(\mathbf{x})$ the L1 Omega-norm of the error term in Equation (17), its derivatives with respect to the state variable $\mathbf{x}$ can be computed as follows:

$$u_k(\mathbf{x}) = \sqrt{\mathbf{e}_k(\mathbf{x})^T \boldsymbol{\Omega}_k \mathbf{e}_k(\mathbf{x})} \tag{32}$$

$$\frac{\partial u_k(\mathbf{x})}{\partial \mathbf{x}} = \frac{1}{u_k(\mathbf{x})} \mathbf{e}_k(\mathbf{x})^T \boldsymbol{\Omega}_k \frac{\partial \mathbf{e}_k(\mathbf{x})}{\partial \mathbf{x}}. \tag{33}$$

We can generalize Equation (31) by introducing a scalar function $\rho(u)$ that computes a new error term as a function of the L1-norm. Equation (31) is a special case for $\rho(u) = \frac{1}{2}u^2$. Thence, our new problem will consist of minimizing the following function:

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmin}} \sum_{k=1}^{K} \rho(u_k(\mathbf{x})) \tag{34}$$

Going more into detail and analyzing the gradients of Equation (34), we have the following relation:

$$\begin{aligned}\frac{\partial \rho(u_k(\mathbf{x}))}{\partial \mathbf{x}} &= \left.\frac{\partial \rho(u)}{\partial u}\right|_{u=u_k(\mathbf{x})} \frac{\partial u_k(\mathbf{x})}{\partial \mathbf{x}} \\ &= \left.\frac{\partial \rho(u)}{\partial u}\right|_{u=u_k(\mathbf{x})} \frac{1}{u_k(\mathbf{x})} \mathbf{e}_k(\mathbf{x})^T \mathbf{\Omega}_k \frac{\partial \mathbf{e}_k(\mathbf{x})}{\partial \mathbf{x}} \\ &= \gamma_k(\mathbf{x}) \mathbf{e}_k(\mathbf{x})^T \mathbf{\Omega}_k \frac{\partial \mathbf{e}_k(\mathbf{x})}{\partial \mathbf{x}} \end{aligned} \tag{35}$$

where

$$\gamma_k(\mathbf{x}) = \left.\frac{\partial \rho(u)}{\partial u}\right|_{u=u_k(\mathbf{x})} \frac{1}{u_k(\mathbf{x})}. \tag{36}$$

The *robustifier* function $\rho(\cdot)$ acts on the gradient, modulating the magnitude of the error term through a scalar function $\gamma_k(\mathbf{x})$. Still, we can also compute the gradient of the Equation (31) as follows:

$$\frac{\partial \|\mathbf{e}_k(\mathbf{x})\|^2_{\mathbf{\Omega}_k}}{\partial \mathbf{x}} = 2\mathbf{e}_k(\mathbf{x})^T \mathbf{\Omega}_k \frac{\partial \mathbf{e}_k(\mathbf{x})}{\partial \mathbf{x}}. \tag{37}$$

We notice that Equations (35) and (37) differ by a scalar term $\gamma(\mathbf{x})$ that depends on $\mathbf{x}$. By absorbing this scalar term at each iteration in a new information matrix $\bar{\mathbf{\Omega}}_k(\mathbf{x}) = \gamma_k(\mathbf{x})\mathbf{\Omega}_k$, we can rely on the iterative algorithm illustrated in the previous sections to implement a robust estimator. In this sense, at each iteration, we compute $\gamma_k(\mathbf{x})$ based on the result of the previous iteration. Note that upon convergence the Equations (35) and (37) will be the same, therefore, they lead to the same optimum. This formalization of the problem is called Iterative Reweighed Least-Squares (IRLS).

The use of robust cost functions biases the information matrix of the system $\mathbf{H}$. Accordingly, if we want to recover an estimate of the solution uncertainty when using robust cost functions, we need to "undo" the effect of function $\rho(\cdot)$. This can be easily achieved recomputing $\mathbf{H}$ after convergence considering only inliers and disabling the robustifier—i.e., setting $\rho(u) = \frac{1}{2}u^2$. Figure 6 illustrates some of the most common cost function used in Robotics and Computer Vision.

Further information on modern robust cost function can be found in the work of MacTavish et al. [64].

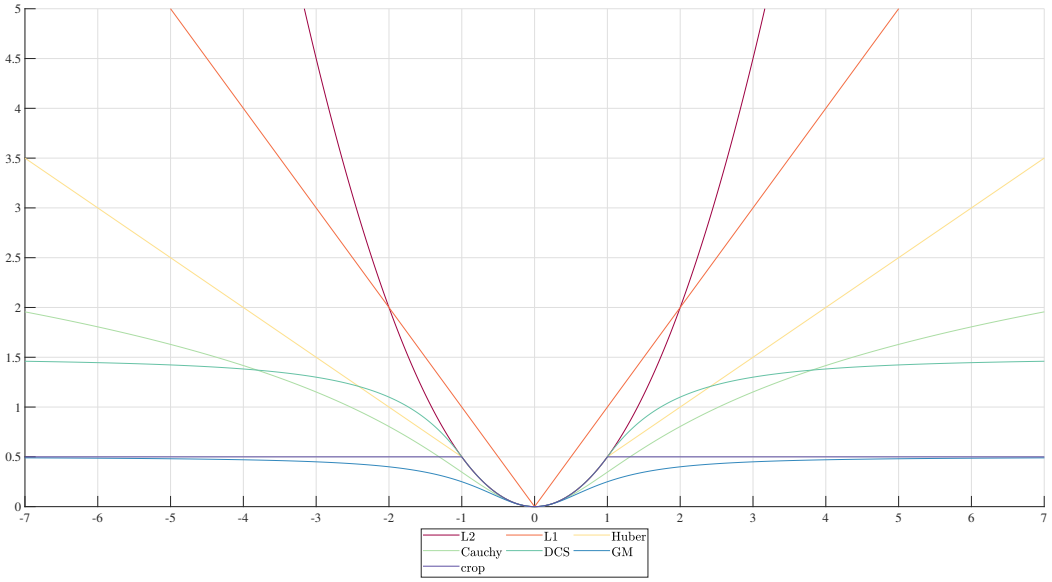


Figure 6. Commonly used robust kernel functions. The kernel threshold is set to 1 in all cases.

4.5. Sparsity

Minimization algorithms like GN or Levenberg-Marquardt (LM) lead to the repeated construction and solution of the linear system $\mathbf{H}\boldsymbol{\Delta}\mathbf{x} = -\mathbf{b}$. In many cases, each measurement $\mathbf{z}_k$ only involves a small subset of state variables, namely:

$$\mathbf{h}_k(\mathbf{x}) = \mathbf{h}_k(\mathbf{x}_k) \quad \text{where} \quad \mathbf{x}_k = \{\mathbf{x}_{k_1}, \dots, \mathbf{x}_{k_q}\} \in \mathbf{x}. \tag{38}$$

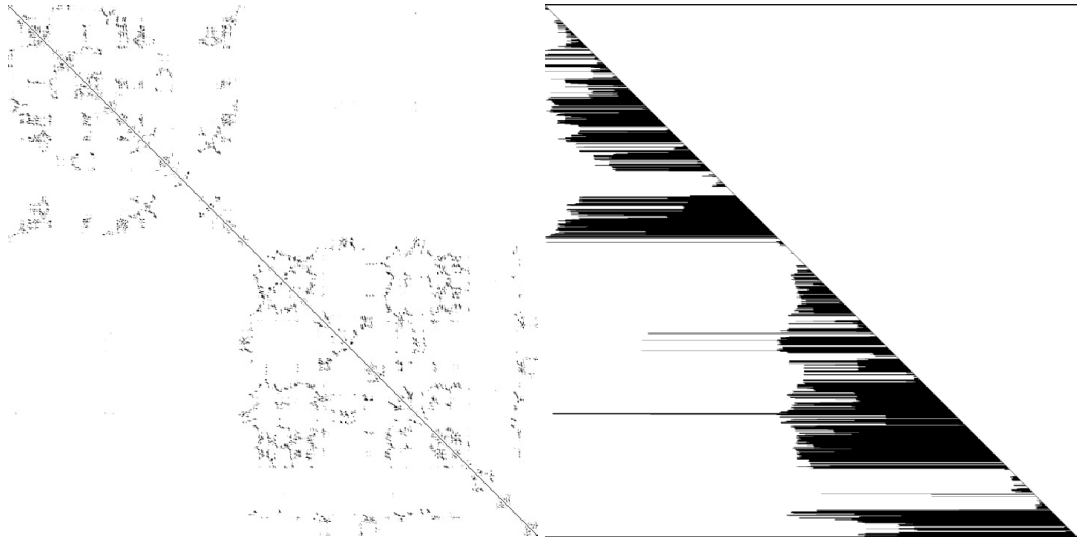
Therefore, the Jacobian for the error term k has the following structure:

$$\mathbf{J}_k = \begin{bmatrix} \mathbf{0} \cdots \mathbf{0}\, \mathbf{J}_{k_1}\, \mathbf{0} \cdots \mathbf{0}\, \mathbf{J}_{k_h}\, \mathbf{0} \cdots \mathbf{0}\, \mathbf{J}_{k_q}\, \mathbf{0} \cdots \mathbf{0} \end{bmatrix}. \tag{39}$$

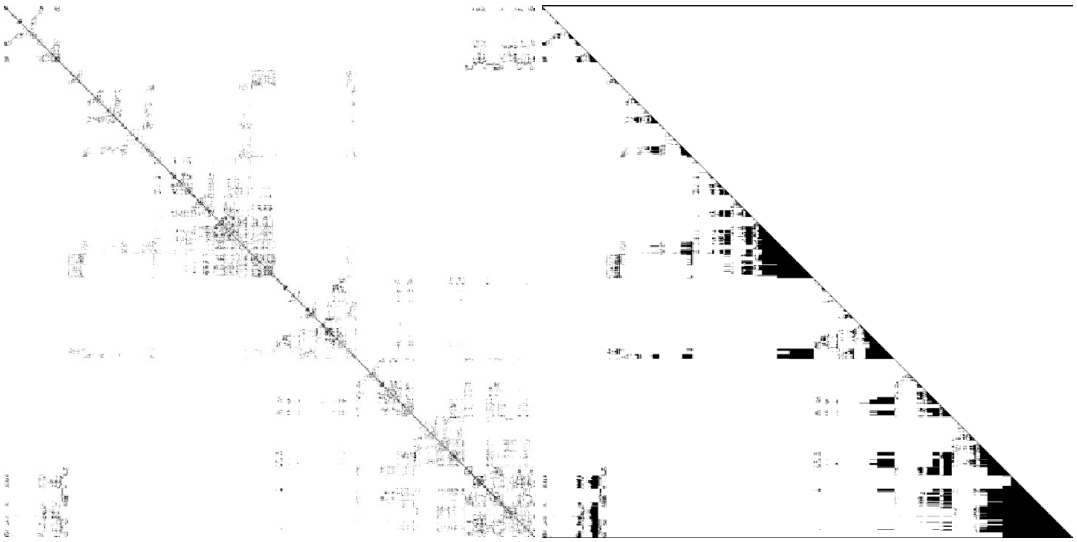
According to this, the contribution $\mathbf{H}_k = \mathbf{J}_k^T \boldsymbol{\Omega}_k \mathbf{J}_k$ of the kth measurement to the system matrix $\mathbf{H}$ exhibits the following pattern:

$$\mathbf{H}_k = \begin{pmatrix} \ddots & & & & & \\ & \mathbf{J}_{k_1}^T \boldsymbol{\Omega}_k \mathbf{J}_{k_1} & \cdots & \mathbf{J}_{k_1}^T \boldsymbol{\Omega}_k \mathbf{J}_{k_h} & \cdots & \mathbf{J}_{k_1}^T \boldsymbol{\Omega}_k \mathbf{J}_{k_q} \\ & \vdots & & \vdots & & \vdots \\ & \mathbf{J}_{k_h}^T \boldsymbol{\Omega}_k \mathbf{J}_{k_1} & \cdots & \mathbf{J}_{k_h}^T \boldsymbol{\Omega}_k \mathbf{J}_{k_h} & \cdots & \mathbf{J}_{k_h}^T \boldsymbol{\Omega}_k \mathbf{J}_{k_q} \\ & \vdots & & \vdots & & \vdots \\ & \mathbf{J}_{k_q}^T \boldsymbol{\Omega}_k \mathbf{J}_{k_1} & \cdots & \mathbf{J}_{k_q}^T \boldsymbol{\Omega}_k \mathbf{J}_{k_h} & \cdots & \mathbf{J}_{k_q}^T \boldsymbol{\Omega}_k \mathbf{J}_{k_q} \\ & & & & & \ddots \end{pmatrix}.$$

Each measurement introduces a finite number of non-diagonal components that depends quadratically on the number of variables that influence the measurement. Therefore, in the many cases when the number of measurements is proportional to the number of variables such as SLAM or BA, the system matrix $\mathbf{H}$ is sparse, symmetric and positive semi-definite *by construction*. Exploiting these intrinsic properties, we can efficiently solve the linear system in Equation (12). In fact, the literature provides many solutions to this kind of problem, which can be arranged in two main groups: (i) iterative methods and (ii) direct methods. The former computes an iterative solution to the linear system by following the gradient of the quadratic form. These techniques often use a *pre-conditioner* for example, Preconditioned Conjugate Gradient (PCG), which scales the matrix to achieve quicker convergence to take steps along the steepest directions. Further information about these approaches can be found in [65]. Iterative methods might require a quadratic time in computing the exact solution of a linear system, but they might be the only option when the dimension of the problem is very large due to their limited memory requirements. Direct methods, instead, compute return the exact solution of the linear system, usually leveraging on some matrix decomposition followed by backsubstitution. Typical methods include: the *Cholesky* factorization $\mathbf{H} = \mathbf{L}\mathbf{L}^T$ or the *QR-decomposition*. A crucial parameter controlling the performances of a sparse direct linear solver is the fill-in that is the number of new non-zero elements introduced by the specific factorization. A key aspect in reducing the fill in is the ordering of the variables. Since computing the optimal reordering is NP-hard, approximated techniques [66–68] are generally employed. Figure 7 shows the effect of different variable ordering on the *same* system matrix. A larger fill-in results in more demanding computations. We refer to [69,70] for a more detailed analysis about this topic.



(**a**) **H** matrix before reordering. (**b**) Cholesky decomposition before reordering.



(**c**) **H** matrix after reordering. (**d**) Cholesky decomposition after reordering.

Figure 7. Effects of the AMD variable reordering on the fill-in of the Cholesky decomposition of matrix **H**. Black pixels indicate non-zero blocks. As illustrated in Figure 7b,d, variable reordering dramatically reduces the fill-in of the decomposed matrix.

4.6. A Unifying Formalism: Factor Graphs

In this section, we introduce a formalism to represent a super-class of the minimization problems discussed so far: *factor graphs*. We recall that (i) our state $\mathbf{x} = \{\mathbf{x}_{1:N}\}$ is composed by N variables, (ii) the conditional probabilities $p(\mathbf{z}_k|\mathbf{x})$ might depend only on a subset of the state variables $\mathbf{x}_k = \{\mathbf{x}_{k_1}, \mathbf{x}_{k_2}, \ldots, \mathbf{x}_{k_q}\} \in \mathbf{x}$ and (iii) we have no prior about the state $p(\mathbf{x}) = \mathcal{U}(\mathbf{x})$. Given this, we can expand Equation (1) as follows:

$$p(\mathbf{x}|\mathbf{z}) \propto \prod_{k=1}^{K} p(\mathbf{z}_k|\mathbf{x}_k) = \prod_{k=1}^{K} p(\mathbf{z}_k|\mathbf{x}_{k_1}, \mathbf{x}_{k_2}, \ldots, \mathbf{x}_{k_q}). \tag{40}$$

Equation (40) expresses the likelihood of the measurements as a product of factors. This concept is elegantly captured in the *factor graph* formalism, which provides a graphical representation for these kinds of problems. A factor graph is a bipartite graph where each node represents either a variable

$\mathbf{x}_i \in \mathbf{x}$, or a *factor* $p(\mathbf{z}_k|\mathbf{x}_k)$. Figure 8 illustrates an example of factor graph. Edges connect a factor $p(\mathbf{z}_k|\mathbf{x}_k)$ with each of its variables $\mathbf{x}_k = \{\mathbf{x}_{k_1}, \mathbf{x}_{k_2}, \ldots, \mathbf{x}_{k_q}\}$. In the remainder of this document, we will stick to the factor graph notation and we will refer to the measurement likelihoods $p(\mathbf{z}_k|\mathbf{x}_k)$ as factors. Note that the main difference between Equation (40) and Equation (14) is that the former highlights the subset of state variables $\mathbf{x}_k$ from which the observation depends, while the latter considers *all* state variables and also those that have a null contribution.

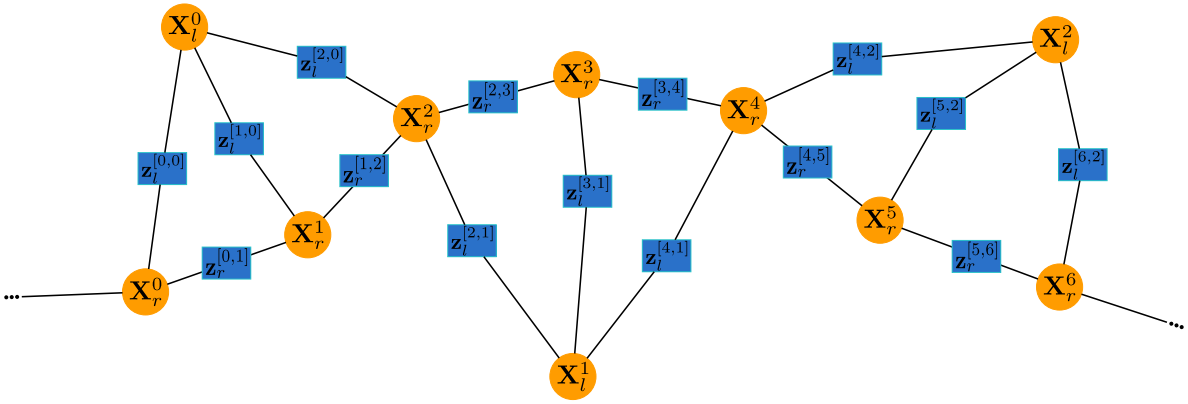


Figure 8. Illustration of a generic factor graph. Orange round nodes depict state variables $\mathbf{x}_{0:N}$. Each blue squared nodes, instead, represents a factor $p(\mathbf{z}|\mathbf{x}_{k_s}, \ldots, \mathbf{x}_{k_q})$; edges denote the variables in conditionals of a factor $\{\mathbf{x}_{k_s}, \ldots, \mathbf{x}_{k_q}\}$.

The aim of this section is to use the factor graph formulation to formalize the ILS minimization exposed so far. In this sense, Algorithm 1 reports a step-by-step expansion of the vanilla GN algorithm exploiting the factor graph formalism—supposing that both states and measurement belong to a smooth manifold. In the remainder of this document, we indicate with bold uppercase symbols elements lying on a manifold space—for example, $\mathbf{X} \in \mathrm{SE}(3)$; lowercase bold symbols specify their corresponding vector perturbations—for example, $\boldsymbol{\Delta}\mathbf{x} \in \mathbb{R}^n$. Going into more detail, at each iteration, the algorithm re-initializes its workspace to store the current linear system (lines 6–7). Subsequently, it processes each measurement $\mathbf{Z}_k$ (line 8), computing (i) the prediction (line 9), (ii) the error vector (line 10), and (iii) the coefficients to apply the robustifier (lines 13–15). While processing a measurement, it also computes the blocks $\mathbf{J}_{k,i}$ of the Jacobians with respect to the variables $\mathbf{X}_i \in \mathbf{X}_k$ involved in the factor. We denote with $\mathbf{H}_{i,j}$ the block i, j of the $\mathbf{H}$ matrix corresponding to the variables $\mathbf{X}_i$ and $\mathbf{X}_j$; similarly, we indicate with $\mathbf{b}_i$ the block of the coefficient vector for the variable $\mathbf{X}_i$. This operation is carried on in the lines 18–20. The contribution of each measurement to the linear system is added in a block fashion. Further efficiency can be achieved exploiting the symmetry of the system matrix $\mathbf{H}$, computing only its lower triangular part. Finally, once the linear system $\mathbf{H}\boldsymbol{\Delta}\mathbf{x} = -\mathbf{b}$ has been built, it is solved using a general sparse linear solver (line 21) and the perturbation $\boldsymbol{\Delta}\mathbf{x}$ is applied to the current state in a block-wise fashion (line 23). The algorithm proceeds until convergence is reached, namely when the delta of the cost-function F between two consecutive iterations is lower than a threshold ϵ—line 3.

Algorithm 1 Gauss–Newton minimization algorithm for manifold measurements and state spaces

Require: Initial guess $\breve{\mathbf{X}}$; Measurements $\mathcal{C} = \{\langle \mathbf{Z}_k, \mathbf{\Omega}_k \rangle\}$
Ensure: Optimal solution $\mathbf{X}^\star$

$F_{\text{old}} \leftarrow \inf$
$F_{\text{new}} \leftarrow 0$
while $F_{\text{old}} - F_{\text{new}} > \epsilon$ **do**
  $F_{\text{old}} \leftarrow F_{\text{new}}$
  $F_{\text{new}} \leftarrow 0$
  $\mathbf{b} \leftarrow 0$
  $\mathbf{H} \leftarrow 0$
  for all $k \in \{1 \dots K\}$ **do**
    $\hat{\mathbf{Z}}_k \leftarrow \mathbf{h}_k(\breve{\mathbf{X}}_k)$
    $\mathbf{e}_k \leftarrow \hat{\mathbf{Z}}_k \boxminus \mathbf{Z}_k$
    $\chi_k \leftarrow \mathbf{e}_k^T \mathbf{\Omega}_k \mathbf{e}_k$
    $F_{\text{new}} \leftarrow F_{\text{new}} + \chi_k$
    $u_k \leftarrow \sqrt{\chi_k}$
    $\gamma_k = \frac{1}{u_k} \frac{\partial \rho_k(u)}{\partial u}\Big|_{u=u_k}$.
    $\tilde{\mathbf{\Omega}}_k = \gamma_k \mathbf{\Omega}_k$
    for all $\mathbf{X}_i \in \{\mathbf{X}_{k_1} \dots \mathbf{X}_{k_q}\}$ **do**
      $\tilde{\mathbf{J}}_{k,i} \leftarrow \frac{\partial \mathbf{h}_k(\mathbf{X} \boxplus \mathbf{\Delta x}) \boxminus \mathbf{Z}_k}{\partial \mathbf{\Delta x}_i}\Big|_{\mathbf{\Delta x}=\mathbf{0}}$
      for all $\mathbf{X}_j \in \{\mathbf{X}_{k_1} \dots \mathbf{X}_{k_q}\}$ and $j <= i$ **do**
        $\mathbf{H}_{i,j} \leftarrow \mathbf{H}_{i,j} + \mathbf{J}_{k,i}^\top \tilde{\mathbf{\Omega}}_k \mathbf{J}_{k,j}$
        $\mathbf{b}_i \leftarrow \mathbf{b}_i + \mathbf{J}_{k,i}^\top \tilde{\mathbf{\Omega}}_i \mathbf{e}_k$
  $\mathbf{\Delta x} \leftarrow \text{solve}(\mathbf{H}\mathbf{\Delta x} = -\mathbf{b})$
  for all $\mathbf{X}_i \in \mathbf{X}$ **do**
    $\breve{\mathbf{X}}_i \leftarrow \breve{\mathbf{X}}_i \boxplus \mathbf{\Delta x}_i$
return $\breve{\mathbf{X}}$

Summarizing, instantiating Algorithm 1 on a specific problem requires to:

- Define for each type of variable $\mathbf{x}_i \in \mathbf{x}$ (i) an extended parametrization $\mathbf{X}_i$, (ii) a vector perturbation $\mathbf{\Delta x}_i$ and (iii) a $\boxplus$ operator that computes a new point on the manifold $\mathbf{X}_i' = \mathbf{X}_i \boxplus \mathbf{\Delta x}_i$. If the variable is Euclidean, the extended and the increment parametrization match and thus $\boxplus$ degenerates to vector addition.
- For each type of factor $p(\mathbf{z}_k|\mathbf{x}_k)$, specify (i) an extended parametrization $\mathbf{Z}_i$, (ii) a Euclidean representation $\mathbf{\Delta z}_i$ for the error vector and (ii) a $\boxminus$ operator such that, given two points on the manifold $\mathbf{Z}_i$ and $\mathbf{Z}_i'$, $\mathbf{\Delta z}_i = \mathbf{Z}_i' \boxminus \mathbf{Z}_i$ represents the motion on the chart that moves $\mathbf{Z}_i$ onto $\mathbf{Z}_i'$. If the measurement is Euclidean, the extended and perturbation parametrizations match and, thus, $\boxminus$ becomes a simple vector difference. Finally, it is necessary to define the measurement function $\mathbf{h}_k(\mathbf{X}_k)$ that, given a subset of state variables $\mathbf{X}_k$, computes the expected measurement $\hat{\mathbf{Z}}_k$.
- Choose a robustifier function $\rho_k(u)$ for each type of factor. The non-robust case is captured by choosing $\rho_k(u) = \frac{1}{2}u^2$.

Note that depending on the choices and on the application, not all of these steps indicated here are required. Furthermore, the value of some variables might be known beforehand—for example, the initial position of the robot in SLAM is typically set at the origin. Hence, these variables do not need to be estimated in the optimization process, since they are *constants* in this context.

In Algorithm 1, fixed variables can be handled in the solution step—i.e., line 21—suppressing all block rows and columns in the linear system that arise from these special nodes. In the next section, we present how to formalize several common SLAM problems through the factor graph formalization introduced so far.

5. Examples

In this section, we present examples on how to apply the methodology illustrated in Section 4.6 to typical problems in robotics, namely: Point-Cloud Registration, Projective Registration, BA, and PGO.

5.1. ICP

ICP represents a family of algorithms used to compute a transform that maximizes the overlap between two point clouds. Let $\mathcal{P}^{\mathrm{f}}$ be the cloud that stays *fixed* and $\mathcal{P}^{\mathrm{m}}$ be the one that is *moved* to maximize the overlap. ICP algorithms achieve this goal progressively refining an initial guess of the target transformation $\mathbf{X}$ by alternating two phases: data-association and optimization. The aim of the data-association is to find a point $\mathbf{p}_i^{\mathrm{f}} \in \mathcal{P}^{\mathrm{f}}$ that is likely to be the same as the point $\mathbf{p}_j^{\mathrm{m}} \in \mathcal{P}^{\mathrm{m}}$ being transformed according to $\mathbf{X}$, see Figure 9. Note that several heuristics to determine the data association have been proposed by the research community, depending on the context of the problem. The most common one are either geometry based—i.e., nearest neighbor, normal shooting, projective association—or rely on appearance-based evaluations. Discussing data-association strategies is out of the scope fo this work, still, we can generalize the *outcome* of data association by a selector function $\mathrm{j}(k) \in \{1, \ldots, |\mathcal{P}^{\mathrm{f}}|\}$ that maps a point index k in the moving cloud to an index j in the fixed cloud. In this way, we indicate a pair of corresponding points as $\langle \mathbf{p}_k^{\mathrm{m}}, \mathbf{p}_{\mathrm{j}(k)}^{\mathrm{f}} \rangle$. In contrast to data-association, the optimization step is naturally described as an ILS problem. The variable to be estimated is a transform $\mathbf{X}$ whose domain depends on the specific scenario—for example, $\mathrm{SE}(2)$, $\mathrm{SE}(3)$ or even a Similarity if the two clouds are at different scales. In the remaining of this section, we will use $\mathbf{X} \in \mathrm{SE}(3)$ to instantiate our factor-graph-based ILS problem.

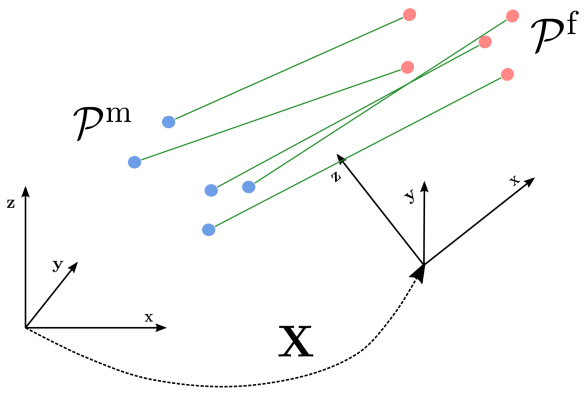


Figure 9. Registration of two point cloud through ICP. The red points represent entries of the *fixed* cloud, while the blue points belong to the *moving* one. Green lines emphasize the associations between points belonging to the two clouds.

5.1.1. Variables

Since the transformation we should estimate is a 3D Isometry $\mathbf{X} \in \mathrm{SE}(3)$, our state lies on a smooth manifold. Therefore, we should define all the entities specified in Section 4.6, namely:

– Extended Parameterization: we conveniently define a transformation $\mathbf{X} = [\mathbf{R} \mid \mathbf{t}] \in \mathrm{SE}(3)$ as a rotation matrix $\mathbf{R}$ and a translation vector $\mathbf{t}$. Using this notation, the following relations hold:

$$\mathbf{X}_{12} = \mathbf{X}_1 \mathbf{X}_2 \triangleq \begin{bmatrix} \mathbf{R}_1 \mathbf{R}_2 & \mathbf{t}_1 + \mathbf{R}_1 \mathbf{t}_2 \end{bmatrix} \tag{41}$$

$$\mathbf{X}^{-1} \triangleq \begin{bmatrix} \mathbf{R}^\top & -\mathbf{R}^\top \mathbf{t} \end{bmatrix}. \tag{42}$$

– Perturbation Vector: a commonly used vector parametrization is $\boldsymbol{\Delta x}^\top = [\boldsymbol{\Delta t}^\top \ \boldsymbol{\Delta a}^\top] \in \mathbb{R}^6$, where $\boldsymbol{\Delta t} \in \mathbb{R}^3$ represents a translation, while $\boldsymbol{\Delta a} \in \mathbb{R}^3$ is a minimal representation for the rotation. The latter might use Euler angles, unit-quaternion or the logarithm of the rotation matrix.

– $\mathbf{X} \boxplus \boldsymbol{\Delta x}$ Operator: this is straightforwardly implemented by first computing the transformation $\boldsymbol{\Delta X} = [\boldsymbol{\Delta R} \ \boldsymbol{\Delta t}] \in \mathrm{SE}(3)$ from the perturbation, and then applying such a perturbation to the previous transform. In formulae:

$$\mathbf{X} \boxplus \boldsymbol{\Delta x} = \mathrm{v2t}(\boldsymbol{\Delta x}) \mathbf{X} \tag{43}$$

where $\mathrm{v2t}(\boldsymbol{\Delta x})$ computes a transform $\boldsymbol{\Delta X}$ from a perturbation vector $\boldsymbol{\Delta x}$. Its implementation depends on the parameters chosen for the rotation part $\boldsymbol{\Delta a}$. Note that the perturbation might be applied to the left or to the right of the initial transformation. In this document, we will consistently apply it to the left. Finally, we define also the inverse function $\boldsymbol{\Delta x} = \mathrm{t2v}(\boldsymbol{\Delta X})$ that computes a perturbation vector from the transformation matrix such that $\boldsymbol{\Delta x} = \mathrm{t2v}(\mathrm{v2t}(\boldsymbol{\Delta x}))$.

5.1.2. Factors

In this problem, we have just one type of factor, which depends on the relative position between a pair of corresponding points, after applying the current transformation $\mathbf{X}$ to the moving cloud. Given a set of associations $\{\langle \mathbf{p}_s^{\mathrm{m}}, \mathbf{p}_{\mathrm{j}(s)}^{\mathrm{f}} \rangle, \ldots, \langle \mathbf{p}_K^{\mathrm{m}}, \mathbf{p}_{\mathrm{j}(K)}^{\mathrm{f}} \rangle\}$, each fixed point $\mathbf{p}_{\mathrm{j}(k)}^{\mathrm{f}}$ constitutes a measurement $\mathbf{z}_k$—since its value does not change during optimization. On the contrary, each moving point $\mathbf{p}_k^{\mathrm{m}}$ will be used to generate the prediction $\hat{\mathbf{z}}$. Note that the measurement space is Euclidean in this scenario—i.e., $\mathbb{R}^3$. Therefore, we only need to define the following entities:

– Measurement Function: it computes the position of a point $\mathbf{p}_k^{\mathrm{m}}$ that corresponds to the point $\mathbf{p}_{\mathrm{j}(k)}^{\mathrm{f}}$ in a fixed scene by applying the transformation $\mathbf{X}$, namely:

$$\mathbf{h}_k^{\mathrm{icp}}(\mathbf{X}) \triangleq \mathbf{X}^{-1} \mathbf{p}_k^{\mathrm{m}} = \mathbf{R}^\top (\mathbf{p}_{\mathrm{j}(k)}^{\mathrm{m}} - \mathbf{t}) \tag{44}$$

– Error Function: since both prediction and measurement are Euclidean, the $\boxminus$ operator boils down to simple vector difference. The error, thus, is a three-dimensional vector computed as:

$$\mathbf{e}_k^{\mathrm{icp}}(\mathbf{X}) = \mathbf{h}_k(\mathbf{X}) - \mathbf{p}_{\mathrm{j}(k)}^{\mathrm{f}}. \tag{45}$$

The Jacobians can be computed analytically very straightforwardly from Equation (45) as:

$$\mathbf{J}^{\mathrm{icp}}(\mathbf{X}, \mathbf{p}) = \left. \frac{\partial \left(\mathbf{X} \boxplus \boldsymbol{\Delta x} \right)^{-1} \mathbf{p}}{\partial \boldsymbol{\Delta x}} \right|_{\boldsymbol{\Delta x} = \mathbf{0}}. \tag{46}$$

With this in place, we can now fully instantiate Algorithm 1. For completeness, in Appendix A we report the functions $\mathrm{v2t}(\cdot)$ and $\mathrm{t2v}(\cdot)$ for $\mathrm{SE}(3)$ objects, while in Appendix B we provide the analytical derivation of the Jacobians. Since in this case the measurement is Euclidean, the Jacobians of error function and measurement function are the same.

5.2. Projective Registration

Projective Registration consists of determining the pose $\mathbf{X}$ of a camera in a known 3D scene from a set of 2D projections of these points on the image plane. In this case, our *fixed* point cloud $\mathcal{P}^{\mathrm{f}}$ will be

consisting of the image projections, while the *moving* one $\mathcal{P}^{\mathrm{m}}$ will be composed by the known location of the 3D points, see Figure 10. We use the notation for data-association defined in Section 5.1, in which the function $\mathrm{j}(i)$ retrieves the index of a 2D measurement on the image that corresponds to the 3D point $\mathbf{p}_i^{\mathrm{m}} \in \mathcal{P}^{\mathrm{m}}$. In addition, in this scenario, the only variable to estimate is the transformation $\mathbf{X} \in \mathrm{SE}(3)$; therefore, we will simply re-use the entities defined in Section 5.1.1 and focus only on the factors.

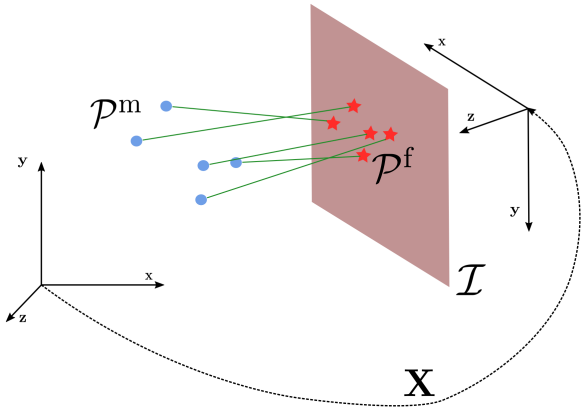


Figure 10. Projective Registration scenario: $\mathcal{I}$ represents the image plane; blue points represent 3D entries of the *moving* cloud, while red stars indicate the projection of each corresponding 3D point onto $\mathcal{I}$. Finally, green lines emphasize the associations between moving cloud and fixed image projections.

Factors

Given a set of 2D–3D associations $\{\langle \mathbf{p}_s^{\mathrm{m}}, \mathbf{p}_{\mathrm{j}(s)}^{\mathrm{f}} \rangle, \ldots, \langle \mathbf{p}_K^{\mathrm{m}}, \mathbf{p}_{\mathrm{j}(K)}^{\mathrm{f}} \rangle\}$, each fixed point $\mathbf{p}_k^{\mathrm{m}} \in \mathbb{R}^2$ will represent a measurement $\mathbf{z}_k$, each moving point will contribute to the prediction $\hat{\mathbf{z}}_k$. Therefore, we can define:

– Measurement Function: it is the projection on the image plane of a scene point $\mathbf{p}_k^{\mathrm{m}}$, assuming that the camera is at $\mathbf{X}$. Such a prediction is obtained by first mapping the point in the camera reference frame to get a new point $\mathbf{p}^{\mathrm{icp}}$, and then projecting this point on the image plane, in formulae:

$$\mathbf{p}^{\mathrm{icp}} \triangleq \mathbf{X}^{-1}\mathbf{p}^{\mathrm{m}} \tag{47}$$

$$\mathbf{p}^{\mathrm{cam}} \triangleq \mathbf{K}\mathbf{p}^{\mathrm{icp}} \tag{48}$$

$$\mathbf{p}^{\mathrm{img}} \triangleq \hom(\mathbf{p}^{\mathrm{cam}}) = \begin{pmatrix} p_x^{\mathrm{cam}} / p_z^{\mathrm{cam}} \\ p_y^{\mathrm{cam}} / p_z^{\mathrm{cam}} \end{pmatrix}. \tag{49}$$

Note that $\mathbf{p}^{\mathrm{cam}}$ is the point in homogeneous image coordinates, while $\mathbf{p}^{\mathrm{img}}$ is the 2D point in pixel coordinates obtained normalizing the point through homogeneous division. Finally, the complete measurement function is defined as:

$$\mathbf{h}_k^{\mathrm{reg}}(\mathbf{X}) \triangleq \hom(\mathbf{K}\mathbf{X}^{-1}\mathbf{p}_{\mathrm{j}(k)}^{\mathrm{m}}) = \hom(\mathbf{K}[\mathbf{h}^{icp}(\mathbf{p}_{\mathrm{j}(k)}^{\mathrm{m}})]). \tag{50}$$

– Error Function: In this case, both measurement and prediction are also Euclidean vectors and, thus, we can use the vector difference to compute the two-dimensional error as follows:

$$\mathbf{e}_k^{\mathrm{reg}}(\mathbf{X}) = \mathbf{h}_k^{\mathrm{reg}}(\mathbf{X}) - \mathbf{z}_k \tag{51}$$

Note that we can exploit the work done in Section 5.1.2 to easily compute Jacobians using the chain-rule, namely:

$$\mathbf{J}^{\text{reg}}(\mathbf{X},\mathbf{p}) = \overbrace{\frac{\partial \text{hom}(\mathbf{v})}{\partial \mathbf{v}}\bigg|_{\mathbf{v}=\mathbf{p}^{\text{cam}}}}^{\mathbf{J}^{\text{hom}}(\mathbf{p}^{\text{cam}})} \mathbf{K}\mathbf{J}^{ICP}(\mathbf{X},\mathbf{p}) = \mathbf{J}^{\text{hom}}(\mathbf{p}^{\text{cam}})\mathbf{K}\mathbf{J}^{ICP}(\mathbf{X},\mathbf{p}). \tag{52}$$

The complete derivation is provided in the Appendix C.

5.3. Structure from Motion and Bundle Adjustment

Structure from Motion (SfM) is the problem of determining the pose of N cameras and the position of M 3D points on a scene, from their image projections. The scenario is shown in Figure 11. This problem is highly non-convex, and tackling it with ILS requires starting from an initial guess not too far from the optimum. Such a guess is usually obtained by using Projective Geometry techniques to determine an initial layout of the camera poses. Subsequently, the points are triangulated to initialize all variables. A final step of the algorithm consists of performing a nonlinear refinement of such an initial guess—known as BA—which is traditionally approached as an ILS problem. Since typically each camera observes only a subset of points, and a point projection depends only on the relative pose between the observed point and the observing camers, BA is a good example of a sparse problem.

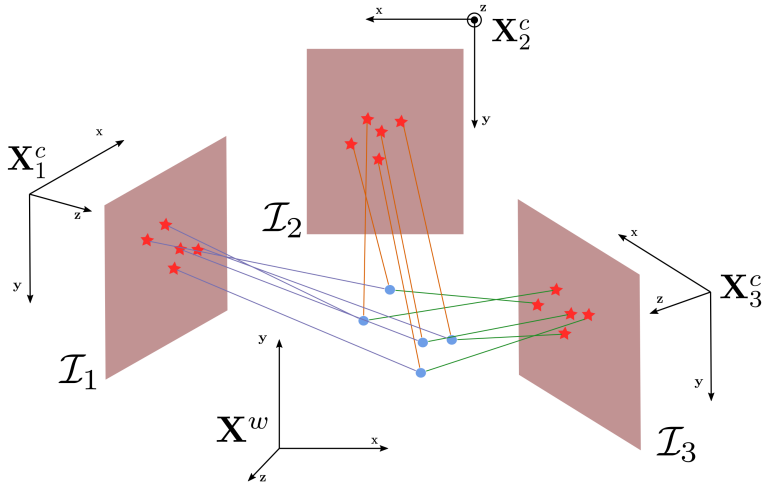


Figure 11. SfM scenario: blue dots represent 3D point in the world, while red dots indicate their projection onto a specific image plane $\mathcal{I}_k$. Colored lines emphasize the data association between 3D points and their corresponding image projections.

5.3.1. Variables

We want to estimate the pose of each camera $\mathbf{X}^{\text{c}}_{1:N}$, *and* the position of each point $\mathbf{x}^{\text{p}}_{1:M}$. The state will thus be a data structure $\mathbf{X} = \langle \mathbf{X}^{\text{c}}_{1:N}, \mathbf{x}^{\text{p}}_{1:M} \rangle$ storing all camera poses and all points. Given this, for the camera poses $\mathbf{X}^{\text{c}}_{1:N}$, the definitions in Section 5.1.1 will be used again. As for the points, we do not need a specific extended parametrization, since they lie on $\Re^3$. Therefore, we should define only:

- Perturbation Vector: the total perturbation vector is defined as

$$\boldsymbol{\Delta}\mathbf{x}^\top = \begin{pmatrix} \boldsymbol{\Delta}\mathbf{x}_1^{\text{c}\top} & \dots & \boldsymbol{\Delta}\mathbf{x}_N^{\text{c}\top} & | & \boldsymbol{\Delta}\mathbf{x}_1^{\text{p}\top} & \dots & \boldsymbol{\Delta}\mathbf{x}_M^{\text{p}\top} \end{pmatrix}. \tag{53}$$

  Otherwise speaking, it is a $6N + 3M$ vector obtained by stacking the individual perturbations.
- $\mathbf{X} \boxplus \boldsymbol{\Delta}\mathbf{x}$ Operator: the operator will use the same machinery introduced in Section 5.1.1 for the poses and the standard Euclidean addition for the point positions.

5.3.2. Factors

Similar to Projective Registration, in BA, a measurement $\mathbf{z}_k$ is a projection of a point on the 2D image plane. However, in this specific scenario, such a projection depends not only on the estimate of a camera pose but also on the estimate of the point. Note that this information was known in the case of Projective Registration, while now it becomes part of the state. For consistency with Algorithm 1, if the kth measurement arises from observing the point $\mathbf{x}^{\mathrm{p}}_m$ with the camera $\mathbf{X}^{\mathrm{c}}_n$, we will denote these indices with two selector functions $n = \mathrm{n}(k)$ and $m = \mathrm{m}(k)$ that map the factor index k respectively to the indices of the observing camera and the observed point. For the kth factor, the camera and point variables will then be $\mathbf{x}^{\mathrm{p}}_{\mathrm{m}(k)}$ and $\mathbf{X}^{\mathrm{c}}_{\mathrm{n}(k)}$. Note that, also in this case, a measurement $\mathbf{z}_k$ lies in an Euclidean space—i.e., $\mathbb{R}^2$. Given this, to instantiate a factor, we define:

– Measurement Function: the prediction $\hat{\mathbf{z}}_k$ can be easily obtained from Equation (50), namely:

$$\mathbf{h}^{\mathrm{ba}}_k(\mathbf{X}) \triangleq \mathbf{h}^{ba}_k(\mathbf{X}^{\mathrm{c}}_{\mathrm{n}(k)}, \mathbf{x}^{\mathrm{p}}_{\mathrm{m}(k)}) = \hom\left(\mathbf{K}({\mathbf{X}^{\mathrm{c}}_{\mathrm{n}(k)}}^{-1})\mathbf{x}^{\mathrm{p}}_{\mathrm{m}(k)}\right). \tag{54}$$

– Error Function: it is the Euclidean difference between prediction and measurement:

$$\mathbf{e}^{ba}_k(\mathbf{X}^{\mathrm{c}}_{\mathrm{n}(k)}, \mathbf{x}^{\mathrm{p}}_{\mathrm{m}(k)}) \triangleq \mathbf{h}^{ba}_k\left(\mathbf{X}^{\mathrm{c}}_{\mathrm{n}(k)}, \mathbf{x}^{\mathrm{p}}_{\mathrm{m}(k)}\right) - \mathbf{z}_k. \tag{55}$$

In this context, the Jacobian $\mathbf{J}^{\mathrm{ba}}_k(\mathbf{X})$ will be consisting of two blocks, corresponding to the perturbation of the camera pose and to the perturbation of the point position, in formulae:

$$\mathbf{J}^{\mathrm{ba}}_k = \begin{pmatrix} \mathbf{0} & \cdots & \mathbf{0} & \mathbf{J}^{\mathrm{ba}}_{k,\mathrm{n}(k)} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{J}^{\mathrm{ba}}_{k,\mathrm{m}(k)} & \mathbf{0} & \cdots & \mathbf{0} \end{pmatrix} \tag{56}$$

where

$$\mathbf{J}^{\mathrm{ba}}_{k,\mathrm{n}(k)}(\mathbf{X}) = \left.\frac{\partial \mathbf{e}^{\mathrm{ba}}_k(\mathbf{X}^{\mathrm{c}}_{\mathrm{n}(k)} \boxplus \Delta\mathbf{x}^{\mathrm{r}}, \mathbf{x}^{\mathrm{p}}_{\mathrm{m}(k)})}{\partial \Delta\mathbf{x}^{\mathrm{c}}}\right|_{\Delta\mathbf{x}^{\mathrm{c}}=\mathbf{0}} \tag{57}$$

$$\mathbf{J}^{\mathrm{ba}}_{k,\mathrm{m}(k)}(\mathbf{X}) = \left.\frac{\partial \mathbf{e}^{\mathrm{ba}}_k(\mathbf{X}^{\mathrm{r}}_{\mathrm{n}(k)}, \mathbf{x}^{\mathrm{p}}_{\mathrm{m}(k)} \boxplus \Delta\mathbf{x}^{\mathrm{p}})}{\partial \Delta\mathbf{x}^{\mathrm{p}}}\right|_{\Delta\mathbf{x}^{\mathrm{p}}=\mathbf{0}}. \tag{58}$$

Again, the measurement domain is Euclidean, thus the Jacobians of the error function and the measurement function match. For completeness, in the Appendix D of this document, we report a more in-depth derivation of the Jacobians.

Still, since all measurements are relative, given a particular solution $\mathbf{X}^\star$ all solutions $\mathbf{X}' = \mathbf{T}\mathbf{X}^\star$ obtained by applying a transformation $\mathbf{T} \in \mathrm{SE}(3)$ to *all* the variables in $\mathbf{X}^\star$ have the same residual χ^2 and, thus, are equivalent. Furthermore, all solutions $\mathbf{X}' = s\mathbf{X}^\star$ obtained by scaling all poses and landmarks by a constant s are equivalent too. This reflects the fact that observing an object that is twice as big from twice the distance results in the same projection. Thence, the problem of BA is under-determined by 7 Degrees-of-Freedom (DoF) and, thus, the vanilla GN algorithm requires fixing at least 7 DoF—typically, a camera pose (6 DoF), and the distance between two points or two camera poses (1 DoF).

5.4. Pose Graphs

A pose graph is a factor graph whose variables $\mathbf{X}^{\mathrm{r}}_{1:N}$ are poses and whose measurements $\mathbf{Z}_{1:K}$ are relative measurements between pairs of poses. Optimizing a pose graph means determining the configuration of poses that is maximally consistent with the measurements. PGO is very common in the SLAM community, and several ad-hoc approaches have been proposed. Similar to BA, PGO is highly non-convex, and its solution with ILS requires a reasonably good initial guess.

5.4.1. Variables

In addition, in PGO, each variable $\mathbf{X}_k^{\mathrm{r}}$ lies on the smooth manifold $\mathrm{SE}(3)$. Once again, we will make use of the formulation used in Section 5.1.1 to characterize the state $\mathbf{X} = \mathbf{X}_{1:N}^{\mathrm{r}}$ and the perturbation vector $\boldsymbol{\Delta x}^{\mathrm{r}T} = (\boldsymbol{\Delta x}_1^{\mathrm{r}T} \ \dots \ \boldsymbol{\Delta x}_N^{\mathrm{r}T})$.

5.4.2. Factors

Using the same index notation in Section 5.3.2, let $\mathbf{Z}_k$ be the kth relative pose measurement expressing the pose $\mathbf{X}_m$ in the reference frame of the pose $\mathbf{X}_n$. We denote the pair of poses as $\mathbf{X}_n = \mathbf{X}_{\mathrm{n}(k)}$, and $\mathbf{X}_m = \mathbf{X}_{\mathrm{m}(k)}$ using the two selector functions $\mathrm{n}(k)$ and $\mathrm{m}(k)$. In this scenario, a measurement $\mathbf{Z}_k$ expresses a relative pose between two variables and, consequently, $\mathbf{Z}_k$ also lies on the smooth manifold $\mathrm{SE}(3)$. Considering this, we define the following entities:

- Measurement Function: this is straightforwardly obtained by expressing the observed pose $\mathbf{X}_{\mathrm{m}(k)}^{\mathrm{r}}$ in the reference frame of the observing pose $\mathbf{X}_{\mathrm{n}(k)}^{\mathrm{r}}$, namely:

$$\mathbf{h}_k^{\mathrm{pgo}}(\mathbf{X}) \triangleq \mathbf{h}_k^{\mathrm{pgo}}\left(\mathbf{X}_{\mathrm{n}(k)}^{\mathrm{r}}, \mathbf{X}_{\mathrm{m}(k)}^{\mathrm{r}}\right) = \left(\mathbf{X}_{\mathrm{n}(k)}^{\mathrm{r}}\right)^{-1} \mathbf{X}_{\mathrm{m}(k)}^{\mathrm{r}}. \tag{59}$$

- Error Function: in this case, since the measurements are non-Euclidean too, we are required to specify a suitable parametrization for the error vector $\mathbf{e}_k$. In literature, many error vectorization are available [63], each one with different properties. Still, in this document, we will make use of the same six-dimensional parametrization used for the increments—i.e., $\mathbf{e}^\top = (\mathbf{e}^{xyz\top} \ \mathbf{e}^{rpy\top})$. Furthermore, we need to define a proper $\boxminus$ operator that expresses on a chart the relative pose between two $\mathrm{SE}(3)$ objects $\boldsymbol{\Delta z} = \hat{\mathbf{Z}} \boxminus \mathbf{Z}$. To achieve this goal, we (i) express $\hat{\mathbf{Z}}$ in the reference system of $\mathbf{Z}$ obtaining the relative transformation $\boldsymbol{\Delta Z}$ and then (ii) compute the chart coordinates of $\boldsymbol{\Delta Z}$ around the origin using the $\mathrm{t2v}(\cdot)$ function. In formulae:

$$\boldsymbol{\Delta z} \triangleq \hat{\mathbf{Z}} \boxminus \mathbf{Z} = \mathrm{t2v}(\boldsymbol{\Delta Z}) = \mathrm{t2v}\left(\mathbf{Z}^{-1} \hat{\mathbf{Z}}\right). \tag{60}$$

  Note that, since $\boldsymbol{\Delta Z}_k$ expresses a relative motion between prediction and measurement, its rotational component will be away from singularities. With this in place, the error vector is computed as the pose of the prediction $\hat{\mathbf{Z}}_k = \mathbf{h}_k^{\mathrm{pgo}}(\mathbf{X})$ on a chart centered in $\mathbf{Z}_k$; namely:

$$\mathbf{e}_k^{\mathrm{pgo}}(\mathbf{X}) \triangleq \mathbf{h}_k^{\mathrm{pgo}}(\mathbf{X}_{\mathrm{n}(k)}^{\mathrm{r}}, \mathbf{X}_{\mathrm{m}(k)}^{\mathrm{r}}) \boxminus \mathbf{Z}_k = \left(\left(\mathbf{X}_{\mathrm{n}(k)}^{\mathrm{r}}\right)^{-1} \mathbf{X}_{\mathrm{m}(k)}^{\mathrm{r}}\right) \boxminus \mathbf{Z}_k. \tag{61}$$

Similar to the BA case, in PGO, the Jacobian $\mathbf{J}_k^{\mathrm{pgo}}(\mathbf{X})$ will be consisting of two blocks, corresponding to the perturbation of observed and the observing poses. The measurements in this case are non-Euclidean, and, thus, we need to compute the Jacobians on the error function—as specified in Equation (61):

$$\mathbf{J}_k^{\mathrm{pgo}} = \begin{pmatrix} \mathbf{0} & \cdots & \mathbf{0} & \mathbf{J}_{k,\mathrm{n}(k)}^{\mathrm{pgo}} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{J}_{k,\mathrm{m}(k)}^{\mathrm{pgo}} & \mathbf{0} & \cdots & \mathbf{0} \end{pmatrix} \tag{62}$$

where

$$\mathbf{J}_{k,\mathrm{n}(k)}^{\mathrm{pgo}}(\mathbf{X}) = \left.\frac{\partial \mathbf{e}_k^{\mathrm{pgo}}(\mathbf{X}_{\mathrm{n}(k)}^{\mathrm{r}} \boxplus \boldsymbol{\Delta x}^{\mathrm{r}}, \mathbf{X}_{\mathrm{m}(k)}^{\mathrm{r}})}{\partial \boldsymbol{\Delta x}^{\mathrm{r}}}\right|_{\boldsymbol{\Delta x}^{\mathrm{r}}=\mathbf{0}} \tag{63}$$

$$\mathbf{J}_{k,\mathrm{m}(k)}^{\mathrm{pgo}}(\mathbf{X}) = \left.\frac{\partial \mathbf{e}_k^{\mathrm{pgo}}(\mathbf{X}_{\mathrm{n}(k)}^{\mathrm{r}}, \mathbf{X}_{\mathrm{m}(k)}^{\mathrm{r}} \boxplus \boldsymbol{\Delta x}^{\mathrm{r}})}{\partial \boldsymbol{\Delta x}^{\mathrm{r}}}\right|_{\boldsymbol{\Delta x}^{\mathrm{r}}=\mathbf{0}} \tag{64}$$

Analogous to the BA case, also in PGO, all measurements are relative, and, hence, all solutions that are related by a single transformation are equivalent. The scale invariance, however, does not apply in this context. As a result, PGO is under-determined by 6 DoF and using GN requires fixing at least one of the poses.

5.5. Considerations

In general, one can carry on the estimation by using an arbitrary number of heterogeneous factors and variables. As an instance, if we want to augment a BA problem with odometry, we can model the additional measurements with PGO factors connecting subsequent poses. Similarly, if we want to solve a Projective Registration problem where the world is observed with two cameras, and we guess the orientation from an inertial sensor, and we can extend the approach presented in Section 5.2 by conducting the optimization on a common origin of the rigid sensor system, instead of the camera position. We will have three types of factors, one for each camera, and one modeling the inertial measurements.

As a final remark, common presentations of ICP, Projective Registration and BA conduct the optimization by estimating *world-to-sensor* frame, rather than the *sensor-to-world*, as we have done in this document. This leads to a more compact formulation. This avoids inverting the transform to compute the prediction, and results in Jacobians being easier to compute in close form. We preferred to provide the solution for *sensor-to-world* to be consistent with the PGO formulation.

6. A Generic Sparse/Dense Modular Least Squares Solver

The methodology presented in Section 4.6 outlines a straight path to the design of an ILS optimization algorithm. Robotic applications often require running the system online, and, thus, they need efficient implementations. When extreme performances are needed, the ultimate strategy is to *overfit* the solution to the specific problem. This can be done both at an algorithmic level and at an implementation level. To improve the algorithm, one can leverage on the a priori known structure of the problem, by removing parts of the algorithm that are not needed or by exploiting domain-specific knowledge. A typical example is when the structure of the linear system is known in advance—for example, in BA—where it is common to use specialized methods to solve the linear system [71].

Focusing on the implementation, we reported two main bottlenecks: the computation of the linear system $\mathbf{H}\boldsymbol{\Delta}\mathbf{x} = \mathbf{b}$ and its solution. Dense problems such as ICP, Sensor Calibration, or Projective Registration are typically characterized by a small state space and many factors of the same type. In ICP, for instance, the state contains just a single $\mathrm{SE}(3)$ object—i.e., the robot pose. Still, this variable might be connected to hundreds of thousands of factors, one for each point correspondence. Between iterations, the ICP mechanism results in these factors to change, depending on the current status of the data association. As a consequence, these systems spend most of their time in *constructing* the linear system, while the time required to solve it is negligible. Notably, applications such as Position Tracking or VO require the system to run at the sensor frame-rate, and each new frame might take several ILS iterations to perform the registration. On the contrary, sparse problems like PGO or large scale BA are characterized by thousands of variables, and a number of factors which is typical in the same order of magnitude. In this context, a factor is connected to very few variables. As an example, in case of PGO, a single measurement depends only two variables that express mutually observable robot poses, whereas the complete problem might contain a number of variables proportional to the length of the trajectory. This results in a large-scale linear system, albeit most of its coefficients are null. In these scenarios, the time spent to *solve* the linear system dominates over the time required to build it.

A typical aspect that hinders the implementation of a ILS algorithm by a person approaching this task for the first time is the calculation of the Jacobians. The labor-intensive solution is to compute them analytically, potentially with the aid of some symbolic-manipulation package. An alternative solution is to evaluate them numerically, by calculating the Jacobians column-by-column with repeated evaluation of the error function around the linearization point. Whereas this practice might work in

many situations, numerical issues can arise when the derivation interval is not properly chosen. A third solution is to delegate the task of evaluating the analytic solution directly to the program, starting from the error function. This approach is called AD and Ceres Solver [10] is the most representative system to embed this feature—later also adopted by other optimization frameworks.

In the remainder of this section, we first revisit and generalize Algorithm 1 to support multiple solution strategies. Subsequently, we outline some design requirements that will finally lead to the presentation of the overall design of our approach proposed in Section 7.

6.1. Revisiting the Algorithm

In the previous section, we presented the implementation of a vanilla GN algorithm for generic factor graphs. This simplistic scheme suffers under high nonlinearities, or when the cost function is under-determined. Over time, alternatives to GN have been proposed, to address these issues, such as LM or Trusted-Region Method (TRM) [72]. All of these algorithms present some common aspects or patterns that can be exploited when designing an optimization system. Therefore, in this section, we reformulate Algorithm 1 to isolate different independent sub-modules. Finally, we present both the GN and the LM algorithms rewritten by using these sub-modules.

In Algorithm 2, we isolate the operations needed to compute the scaling factor γ_k for the information matrix $\mathbf{\Omega}_k$, knowing the current χ^2_k. Algorithm 3 performs the calculation of the error $\mathbf{e}_k$ and the Jacobian $\mathbf{J}_k$ for a factor $\langle \mathbf{Z}_k, \mathbf{\Omega}_k \rangle$ at the current linearization point. Algorithm 4 applies the robustifier to a factor, and updates the linear system. Algorithm 5 applies the perturbation $\mathbf{\Delta}x$ to the current solution $\check{\mathbf{X}}$ to obtain an updated estimate. Finally, in Algorithm 6, we present a revised version of Algorithm 1 that relies on the modules described so far. In Algorithm 7, we provide an implementation of the LM algorithm that makes use of the same core sub-algorithms used in Algorithm 6. The LM algorithm solves a damped version of the system, namely $(\mathbf{H} + \lambda \cdot \text{diag}(\mathbf{H}))\mathbf{\Delta x} = \mathbf{b}$. The magnitude of the damping factor λ is adjusted depending on the current variation of the χ^2. If the χ^2 increases upon an iteration, λ increases too. In contrast, if the solution improves, λ is decreased. Variants of these two algorithms—for example, damped GN that solves $(\mathbf{H} + \lambda \mathbf{I})\mathbf{\Delta x} = \mathbf{b}$—can be straightforwardly implemented by slight modification to the algorithm presented here.

Algorithm 2 robustify(χ^2_k)—computes the robustification coefficient γ_k

Require: Current χ^2_k.
Ensure: γ_k computed from the actual error,
1: $u_k \leftarrow \sqrt{\chi_k}$
2: $\gamma_k = \frac{1}{u_k} \left. \frac{\partial \rho_k(u)}{\partial u} \right|_{u=u_k}$
3: **return** γ_k

6.2. Design Requirements

While designing our system, we devised a set of requirements stemming from our experience both as developers and as users. Subsequently, we turned these requirements into some design choices that lead to our proposed optimization framework. Although most of these requirements indicate good practices to be followed in potentially any new development, we highlight here their role in the context of a solver design.

6.2.1. Easy to Use and Symmetric API

As users, we want to configure, instantiate, and run a solver in the same manner, regardless of the specific problem to which it is applied. Ideally, we do not want the user to care if the problem is dense or sparse. Furthermore, in several practical scenarios, one wants to change aspects of the solver while it runs—for example, the minimization algorithm chosen, the robust kernel, or the termination

criterion. Finally we want to save/retrieve the configuration of a solver and all of its sub-modules to/from disk. Thence, the expected usage pattern should be: (i) load the specific solver configuration from disk and eventually tune it, (ii) assign a problem to the solver or load it from file, (iii) compute a solution and eventually iv) provide statistics about the evolution of optimization.

Algorithm 3 linearize$(\breve{\mathbf{X}}_k, \breve{\mathbf{Z}}_k)$—computes the error $\mathbf{e}_k$ and the Jacobians $\mathbf{J}_k$ at the current linearization point $\breve{\mathbf{X}}$

Require: Initial guess $\breve{\mathbf{X}}_k$; Current measurement $\breve{\mathbf{Z}}_k$;

Ensure: Error: $\mathbf{e}_k$; Jacobians $\tilde{J}_k$;

$\hat{\mathbf{Z}}_k \leftarrow \mathbf{h}_k(\breve{\mathbf{X}}_k)$

$\mathbf{e}_k \leftarrow \hat{\mathbf{Z}}_k \boxminus \mathbf{Z}_k$

$\tilde{\mathbf{J}}_k = \{\}$

for all $\mathbf{X}_i \in \{\mathbf{X}_{k_1} ... \mathbf{X}_{k_q}\}$ **do**

  $\tilde{\mathbf{J}}_{k,i} \leftarrow \left. \frac{\partial \mathbf{h}_k(\mathbf{X} \boxplus \boldsymbol{\Delta}\mathbf{x}) \boxminus \mathbf{Z}_k}{\partial \boldsymbol{\Delta}\mathbf{x}_i} \right|_{\boldsymbol{\Delta}\mathbf{x}=\mathbf{0}}$

  $\tilde{\mathbf{J}}_k \leftarrow \tilde{\mathbf{J}}_k \cup \{\tilde{\mathbf{J}}_{k,i}\}$

return $< \mathbf{e}_k, \mathbf{J}_k >$

Algorithm 4 updateHb$(\mathbf{H}, \mathbf{b}, \breve{\mathbf{X}}_k, \mathbf{Z}_k, \boldsymbol{\Omega}_k)$ updates a linear system with a factor current linearization point $\breve{\mathbf{X}}$, and returns the χ^2_k of the factor

Require: Initial guess $\breve{\mathbf{X}}_k$; Coefficients of the linear system $\mathbf{H}$ and $\mathbf{b}$; Measurement $\langle \mathbf{Z}_k, \boldsymbol{\Omega}_k \rangle$

Ensure: Coefficients of the linear system after the update $\mathbf{H}$ and $\mathbf{b}$; Value of the cost function for this factor χ^2_k

$< \mathbf{e}_k, \mathbf{J}_k >= \text{linearize}(\breve{\mathbf{X}}_k, \mathbf{Z}_k)$

$\chi^2_k \leftarrow \mathbf{e}_k^T \boldsymbol{\Omega}_k \mathbf{e}_k$

$\gamma_k = \text{robustify}(\chi^2_k)$

$\tilde{\boldsymbol{\Omega}}_k = \gamma_k \boldsymbol{\Omega}_k$

for all $\mathbf{X}_i \in \{\mathbf{X}_{k_1} ... \mathbf{X}_{k_q}\}$ **do**

  for all $\mathbf{X}_j \in \{\mathbf{X}_{k_1} ... \mathbf{X}_{k_q}\}$ and $j <= i$ **do**

    $\mathbf{H}_{i,j} \leftarrow \mathbf{H}_{i,j} + \mathbf{J}_{k,i}^\top \tilde{\boldsymbol{\Omega}}_k \mathbf{J}_{k,j}$

    $\mathbf{b}_i \leftarrow \mathbf{b}_i + \mathbf{J}_{k,i}^\top \tilde{\boldsymbol{\Omega}}_i \mathbf{e}_k$

return $< \chi^2_k, \mathbf{H}, \mathbf{b} >$

Algorithm 5 updateSolution$(\breve{\mathbf{X}}, \boldsymbol{\Delta}\mathbf{x})$ applies a perturbation to the current system solution

Require: Current solution $\breve{\mathbf{X}}$; Perturbation $\boldsymbol{\Delta}\mathbf{x}$

Ensure: New solution $\breve{\mathbf{X}}$, moved according to $\boldsymbol{\Delta}\mathbf{x}$

for all $\mathbf{X}_i \in \mathbf{X}$ **do**

  $\breve{\mathbf{X}}_i \leftarrow \breve{\mathbf{X}}_i \boxplus \boldsymbol{\Delta}\mathbf{x}_i$

return $\breve{\mathbf{X}}$

Algorithm 6 gaussN($\check{\mathbf{X}}, \mathcal{C}$)—Gauss–Newton minimization algorithm for manifold measurements and state spaces

Require: Initial guess $\check{\mathbf{X}}$; Measurements $\mathcal{C} = \{\langle \mathbf{Z}_k, \mathbf{\Omega}_k \rangle\}$
Ensure: Optimal solution $\mathbf{X}^\star$

$F_{\text{old}} \leftarrow \text{inf}$, $F_{\text{new}} \leftarrow 0$
while $F_{\text{old}} - F_{\text{new}} > \epsilon$ **do**
  $F_{\text{old}} \leftarrow F_{\text{new}}$, $F_{\text{new}} \leftarrow 0$, $\mathbf{b} \leftarrow 0$, $\mathbf{H} \leftarrow 0$
  for all $k \in \{1 \dots K\}$ **do**
    $\langle \chi_k, \mathbf{H}_k, \mathbf{b}_k \rangle \leftarrow \text{updateHb}(\mathbf{H}_k, \mathbf{b}_k, \mathbf{X}_k, \mathbf{Z}_k, \mathbf{\Omega}_k)$
    $F_{\text{new}} \leftarrow \chi_k$
  $\mathbf{\Delta x} \leftarrow \text{solve}(\mathbf{H}\mathbf{\Delta x} = -\mathbf{b})$
  $\check{\mathbf{X}} \leftarrow \text{updateSolution}(\check{\mathbf{X}}, \mathbf{\Delta x})$
return $\check{\mathbf{X}}$

Algorithm 7 levenbergM($\check{\mathbf{X}}, \mathcal{C}$)—Levenberg–Marquardt minimization algorithm for manifold measurements and state spaces

Require: Initial guess $\check{\mathbf{X}}$; Measurements $\mathcal{C} = \{\langle \mathbf{Z}_k, \mathbf{\Omega}_k \rangle\}$; Maximum number of internal iteration t_{max}
Ensure: Optimal solution $\mathbf{X}^\star$

$F_{\text{old}} \leftarrow \text{inf}$, $F_{\text{new}} \leftarrow 0$, $F_{\text{internal}} \leftarrow 0$
$\check{\mathbf{X}}_{backup} \leftarrow \check{\mathbf{X}}$
$\lambda \leftarrow \text{initializeLambda}(\check{\mathbf{X}}, \mathcal{C})$
while $F_{\text{old}} - F_{\text{new}} < \epsilon$ **do**
  $F_{\text{old}} \leftarrow F_{\text{new}}$, $F_{\text{new}} \leftarrow 0$, $\mathbf{b} \leftarrow 0$, $\mathbf{H} \leftarrow 0$
  for all $k \in \{1 \dots K\}$ **do**
    $\langle \chi, \mathbf{H}, \mathbf{b} \rangle \leftarrow \text{updateHb}(\mathbf{H}, \mathbf{b}, \mathbf{X}_k, \mathbf{Z}_k, \mathbf{\Omega}_k)$
    $F_{\text{internal}} \leftarrow \chi$
  $t \leftarrow 0$
  while $t < t_{max} \wedge t > 0$ **do**
    $\mathbf{\Delta x} \leftarrow \text{solve}((\mathbf{H} + \lambda \mathbf{I})\mathbf{\Delta x} = -\mathbf{b})$
    $\check{\mathbf{X}} \leftarrow \text{updateSolution}(\check{\mathbf{X}}, \mathbf{\Delta x})$
    $\langle \chi, \mathbf{H}, \mathbf{b} \rangle \leftarrow \text{updateHb}(\mathbf{H}, \mathbf{b}, \mathbf{X}_k, \mathbf{Z}_k, \mathbf{\Omega}_k)$
    $F_{\text{new}} \leftarrow \chi$
    if $F_{\text{new}} - F_{\text{internal}} < 0$ **then**
      $\lambda \leftarrow \lambda / 2$
      $\check{\mathbf{X}}_{backup} \leftarrow \check{\mathbf{X}}$
      $t \leftarrow t - 1$
    else
      $\lambda \leftarrow \lambda \cdot 2$
      $\check{\mathbf{X}} \leftarrow \check{\mathbf{X}}_{backup}$
      $t \leftarrow t + 1$
return $\check{\mathbf{X}}$

Note that many current state-of-the-art ILS solver allow for easily performing the last three steps of this process; however, they do not provide the ability of permanently write/read their configuration on/from disk—as our system does.

6.2.2. Isolating Parameters, Working Variables and Problem Description/Solution

When a user is presented to a new potentially large code-base, having a clear distinction between what the variables represent and how they are used, which substantially reduces the learning curve. In particular, we distinguish between parameters, working variables, and the input/output. Parameters are those objects controlling the behavior of the algorithm, such as number of iterations or the thresholds in a robustifier. Parameters might also include processing sub-modules, such as the algorithm to use or the algebraic solver of the linear system. In summary, parameters characterize the behavior of the optimizer, independently from the input, and represent the configuration that can be stored/retrieved from disk.

In contrast to parameters, working variables are altered during the computation, and are not directly accessible to the end user. Finally, we have the description of the problem—i.e., the factor graph, where the factors and the variables expose an interface agnostic to the approach that will be used to solve the problem.

6.2.3. Trade-off Development Effort/Performance

Quickly developing a proof of concept is a valuable feature to have while designing a novel system. At the same time, once a way to approach the problem has been found, it becomes perfectly reasonable to invest more effort to enhance its efficiency.

Upon instantiation, the system should provide an off-the shelf generic and fair configuration. Obviously, this might be tweaked later for enhancing the performances on the specific class of problems. A possible way to enhance performances is by exploiting the special structure of a specific class of problems, overriding the general APIs to perform ad-hoc computations. This results in layered APIs, where the functionalities of a level rely only on those of the level below. As an example, in our architecture, the user can either specify the error function and let the system to compute the Jacobians using AD or provide the analytical expression of the Jacobians if more performances are needed. Finally, the user might intervene at a lower level, providing directly the contribution to the linear system $\mathbf{H}_k = \mathbf{J}_k^T \mathbf{\Omega}_k \mathbf{J}_k$ given by the factor. In a certain class of problems also computing this product represents a performance penalty. An additional benefit provided by this design is the direct support for *Newton's method*, which can be straightforwardly achieved by substituting the approximated Hessian $\mathbf{H}_k = \mathbf{J}_k^T \mathbf{\Omega}_k \mathbf{J}_k$ with the analytic one $\mathbf{H}_k = \frac{\partial^2 \mathbf{e}_k}{\partial \Delta \mathbf{x}_k^2}$. This feature captures second order approaches such as NDT [26] in the language of our API.

6.2.4. Minimize Codebase

The likelihood of bugs in the implementation grows with the size of the code-base. For small teams characterized by a high turnover—like the ones found in academic environments—maintaining the code becomes an issue. In this context, we choose to favor the code reuse in spite of a small performance gain. The same class used to implement an algorithm, a variable type or a robustifier should be used in all circumstances, namely sparse and dense problems—where it is needed.

7. Implementation

As support material for this tutorial, we offer an own implementation of a modular ILS optimization framework that has been designed around the methodology illustrated in Section 4.6. Our system is written in modern C++17 and provides static type checking, AD and a straightforward interface. The core of our framework fits in less than 6000 lines of code, while the companion libraries to support the most common problems—for example, 2D and 3D SLAM, ICP, Projective and Dense Registration, Sensor Calibration—are contained in 6500 lines of code. For comparison, g^2o is around 45,000 lines, GTSAM 303,000 and ceres 80,000. Albeit originally designed as a tool for rapid prototyping, our system achieves a high degree of customization and competes with other state-of-the-art systems in terms of performances.

Based on the requirements outlined in Section 6.2, we designed a component model, where the processing objects (named `Configurable`) can possess parameters, support dynamic loading, and can be transparently serialized. Our framework relies on a custom-built serialization infrastructure that supports format independent serialization of arbitrary data structures, named `Basic Object Serialization System` (`BOSS`). Furthermore, thanks to this foundation, we can provide both a graphical configurator—that allows for assembling and easily tuning the modules of a solver—and a command-line utility to edit and run configurations on the go.

The goal of this section is to provide the reader with a quick overview of the proposed system, focusing on how the user interacts with it. Given the class-diagram illustrated in Figure 12, in the remainder, we will first analyze the core modules of the solver and then provide two practical examples on how to use it.

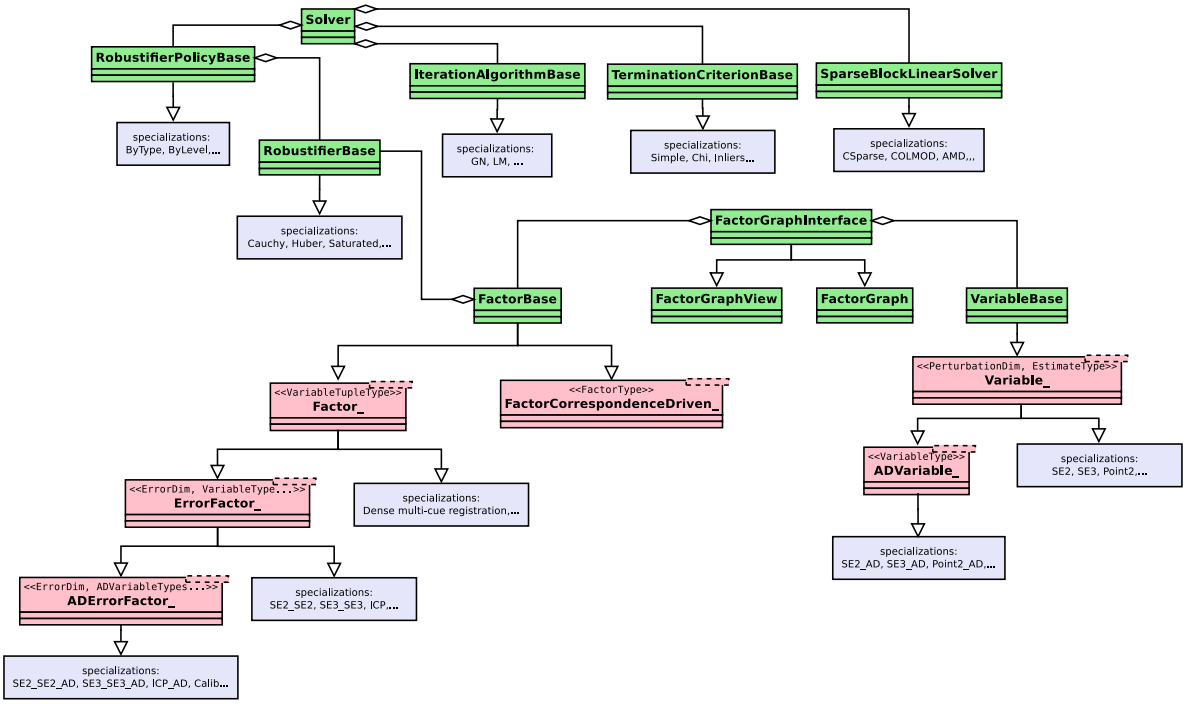


Figure 12. UML class diagram of our architecture. In green, we show the type-independent classes, in pink, the type-dependent classes and, in pale blue, we outline potential specializations. The template class names end with an underscore and the arguments are highlighted above the name, between angular brackets. Arrow lines denote inheritance, and diamonds aggregation/ownership.

7.1. Solver Core Classes

Our framework has been designed to satisfy the requirements stated in Section 6, embedding unified APIs to cover both dense and sparse problems symmetrically. Furthermore, thanks to the `BOSS` serialization library, the user can generate permanent configuration of the solver, to be later read and reused on the go. The configuration of a solver generally embeds the following parameters:

- *Optimization Algorithm*: the algorithm that performs the minimization; currently, only GN and LM are supported, still, we plan to also add TRM approaches
- *Linear Solver*: the algebraic solver that computes the solution of the linear system $\mathbf{H}\boldsymbol{\Delta}\mathbf{x} = -\mathbf{b}$; we embed a naive AMD-based [66] linear solver together with other approaches based on well-known highly-optimized linear algebra libraries—for example, SuiteSparse (http://faculty.cse.tamu.edu/davis/suitesparse.html)

- *Robustifier*: the robust kernel function applied to a specific factor; we provide several commonly used instances of robustifier, together with a modular mechanism to assign specific robustifier to different types of factor—called robustifier *policy*
- *Termination Criterion*: a simple modules that, based on optimization statistics, checks whether convergence has been reached.

Note that, in our architecture, there is a clear separation between solver and problem. In the next section, we will describe how we formalized the latter. In the remainder of this section, instead, we will focus on the solver classes, which are in charge of *computing* the problem solution.

The class `Solver` implements a unified interface for our optimization framework. It presents itself to the user with a unified data-structure to configure, control, run, and monitor the optimization. This class allows for selecting the type of algorithm used within one iteration, which the algorithm uses to solve the linear system, or which termination criterion to use. This mechanism is achieved by delegating the execution of these functions to specific interfaces. In more detail, the linear system is stored in a sparse-block-matrix structure that effectively separates the solution of the linear system from the rest of the optimization machinery. Furthermore, our solver supports incremental updates, and can provide an estimate of partial covariance blocks. Finally, our system supports hierarchical approaches. In this sense, the problem can be represented at different resolutions (levels), by using different factors. When the solution at a coarse level is computed, the optimization starts from the next denser level, enabling a new set of factors. In the new step, the initial guess is computed from the solution of the coarser level.

The `IterationAlgorithmBase` class defines an interface for the outer optimization algorithm—i.e., GN or LM. To carry on its operations, it relies on the interface exposed by the `Solver` class. The latter is in charge to invoke the `IterationAlgorithmBase`, which will run a *single* iteration of its algorithm.

Class `RobustifierBase` defines an interface for using arbitrary $\rho(u)$ functions—as illustrated in Section 4.4. Robust kernels can be directly assigned to factors or, alternatively, the user might define a *policy* that is based on the status of the actual factor decides on which robustifier to use. The definition of a policy is done by implementing the `RobustifierPolicyBase` interface.

Finally, `TerminationCriterionBase` defines an interface for a predicate that, exploiting the optimization statistics, detects whether the system has converged to a solution or a fatal error has occurred.

7.2. Factor Graph Classes

In this section, we provide an overview of the top-level classes constituting a factor graph—i.e., the optimization *problem*—in our framework. In specifying new variables or factors, the user can interact with the system through a layered interface. More specifically, factors can be defined using AD and, thus, contained in few lines of code for rapid prototyping or the user can directly provide how to compute analytic Jacobians if more speed is required. Furthermore, to achieve extreme efficiency, the user can choose to compute its own routines to update the quadratic form directly, consistently in line with our design requirement of *more-work/more-performance*. Note that we observed in our experiments that in large sparse problems the time required to linearize the system is marginal compared to the time required to solve it. Therefore, in most of these cases, AD can be used without significant performance losses.

7.2.1. Variables

The `VariableBase` implements a base abstract interface the variables in a factor graph, whereas `Variable_<PerturbationDim, EstimateType>` specializes the base interface on a specific type. The definition of a new variable extending the `Variable_` template requires the user to specify (i) the type `EstimateType` used to store the value of the variable $\mathbf{X}_i$, (ii) the dimension

`PerturbationDim` of the perturbation $\Delta\mathbf{x}_i$ and (iii) the $\boxplus$ operator. This is coherent with the methodology provided in Section 4.6. In addition to these fields, the variable has an integer key, to be uniquely identified within a factor graph. Furthermore, a variable can be in either one of these three states:

- **Active**: the variable will be estimated
- **Fixed**: the variable stays constant through the optimization
- **Disabled**: the variable is ignored and all factors that depend on it are ignored as well.

To provide roll-back operations—such as those required by LM—a variable also stores a stack of values.

To support AD, we introduce the `ADVariable_` template, that is instantiated on a variable *without* AD. Instantiating a variable with AD requires to define the $\boxplus$ operator by using the AD scalar type instead of the usual `float` or `double`. This mechanism allows us to mix in a problem factors that require AD with factors that do not.

7.2.2. Factors

The base level of the hierarchy is the `FactorBase`. It defines a common interface for this type of graph objects. It is responsible for (i) computing the error and, thus, the χ^2—(ii) updating the quadratic form $\mathbf{H}$ and the right-hand side vector $\mathbf{b}$ and (iii) invoking the robustifier function (if required). When an update is requested, a factor is provided with a structure on which to write the outcome of the operation. A factor can be *enabled* or *disabled*. In the latter case, it will be ignored during the computation. In addition, upon updating a factor might become *invalid*, if the result of the computation is meaningless. This occurs for instance in BA, when a a point is projected outside the image plane.

The `Factor_<VariableTupleType>` class implements a typed interface for the factor class. The user willing to extend the class at this level is responsible for implementing the entire `FactorBase` interface, relying on functions for typed access to the blocks of the system matrix $\mathbf{H}$ and of the coefficient vector $\mathbf{b}$. In this case, the block size is determined from the dimension of the perturbation vector of the variables in the template argument list. We extended the factors at this level to implement approaches such as dense multi-cue registration [31]. Special structures in the Jacobians can be exploited to speed up the calculation of $\mathbf{H}_k$ whose computation has a non negligible cost.

The `ErrorFactor_<ErrorDim, VariableTypes...>` class specializes a typed interface for the factor class, where the user has to implement both the error function $\mathbf{e}_k$ and the Jacobian blocks $\mathbf{J}_{k,i}$. The calculation of the $\mathbf{H}$ and the $\mathbf{b}$ blocks is done through loops unrolled at compile time since the types and the dimensions of the variables/errors are part of the type.

The `ADErrorFactor_<Dim, VariableTypes...>` class further specializes the `ErrorFactor_`. Extending the class at this level only required specifying *only* the error function. The Jacobians are computed through AD, and the updates of $\mathbf{H}$ and the $\mathbf{b}$ are done according to the base class.

Finally, the `FactorCorrespondenceDriven_<FactorType>` implements a mechanism that allows the solver to iterate over multiple factors of the same type and connecting the same set of variables, without the need of explicitly storing them in the graph. A `FactorCorrespondenceDriven_` is instantiated on a base type of factor, and it is specialized by defining which actions should be carried on as a consequence of the selection of the "next" factor in the pool by the solver. The solver sees this type of factor as multiple ones, albeit a `FactorCorrespondenceDriven_` is stored just once in memory. Each time a `FactorCorrespondenceDriven_` is accessed by the solver a callback changing the internal parameters is called. In its basic implementation, this class takes a container of corresponding indices, and two data containers: `Fixed` and `Moving`. Each time a new factor within the `FactorCorrespondenceDriven_` is requested, the factor is configured by: selecting the next pair of corresponding indices from the set, and by picking the elements in `Fixed` and `Moving` at those indices. As an instance, to use our solver within an ICP algorithm, the user has to configure the factor by setting

the `Fixed` and `Moving` point clouds. The correspondence vector can be changed anytime to reflect a new data association. This results in different correspondences being considered at each iteration.

7.2.3. FactorGraph

To carry on an iteration, the solver has to *iterate* over the factors and, hence, it requires randomly accessing the variables. Restricting the solver to access a graph through an interface of random access iterators enables us to decouple the way the graph is accessed from the way it is stored. This would allow us to support transparent off-core storage that can be useful on very large problems.

A `FactorGraphInterface` defines the way to access a graph. In our case, we use integer values as key for variables and factors. The solver accesses a graph only through the `FactorGraphInterface` and, thence, it can read/write the value of state variables, read the factors, but it is not allowed to modify the graph structure.

A heap-based concrete implementation of a factor graph is provided by the `FactorGraph` class that specializes the interface. The `FactorGraph` supports transparent serialization/deserialization. Our framework makes use of the open-source math library Eigen [73], which provides fast and easy matrix operation. The serialization/deserialization of variable and factors that are constructed on eigentypes is automatically handled by our `BOSS` library.

In sparse optimization, it is common to operate on a local portion of the entire problem. Instrumenting the solver with methods to specify the local portions would bloat the implementation. Alternatively, we rely on the concept of `FactorGraphView` that exposes an interface on a local portion of a `FactorGraph`—or of any other object implementing the `FactorGraphInterface`.

8. Experiments

In this section, we propose several comparisons between our framework and other state-of-the-art optimization systems. The aim of these experiments is to support the claims on the performance of our framework and, thus, we focused on the accuracy of the computed solution and the time required to achieve it. Experiments have been performed both on dense scenarios—such as ICP—and sparse ones—for example, PGO and PLGO.

8.1. Dense Problems

Many well-known SLAM problems related to the front-end can be solved exploiting the ILS formulation introduced before. In such scenarios—for example, point-clouds registration—the number of variables is small compared to the observations' one. Furthermore, at each registration step, the data-association is usually recomputed to take advantage of the new estimate. In this sense, one has to build the factor graph associated with the problem from scratch at each step. In such contexts, the most time-consuming part of the process is represented by the construction of linear system in Equation (19) and not its solution.

To perform dense experiments, we choose a well-known instance of these kinds of problems: ICP. We conducted multiple tests, comparing our framework to the current state-of-the-art PCL library [15] on the standard registration datasets summarized in Table 1.

Table 1. Specification of the datasets used to perform dense benchmarks.

Dataset	Sensor	Variables	Factors
`ICL-NUIM` [74]	RGB-D	1	307,200
`ETH-Hauptgebaude` [75]	Laser-scanner	1	189,202
`ETH-Apartment` [75]	Laser-scanner	1	370,276
`Stanford-Bunny` [76]	3D digitalizer	1	35,947

In all of the cases, we setup a controlled benchmarking environment, in order to have a precise ground-truth. In the `ETH-Hauptgebaude`, `ETH-Apartment`, and `Stanford-Bunny`, the raw data consist of a series of range scans. Therefore, in such cases, we constructed the ICP problem as follows:

- reading of the first raw scan and generate a point cloud
- transformation of the point cloud according to a known isometry $\mathbf{T}_{GT}$
- generation of perfect association between the two clouds
- registration starting from $\mathbf{T}_{init} = \mathbf{I}$.

Since our focus is on the ILS optimization, we used the same set of data-associations and the same initial guess for all approaches. This methodology renders the comparison fair and unbiased. As for the `ICL-NUIM` dataset, we obtained the raw point cloud unprojecting the range image of the first reading of the `lr-0` scene. After this initial preprocessing, the benchmark flow is the same as described before.

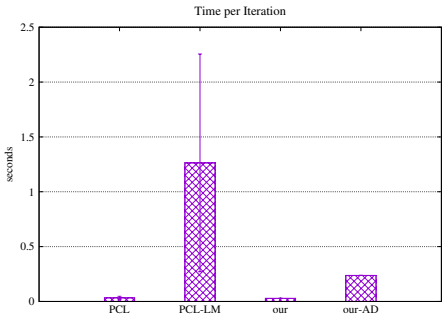
In this context, we compared (i) the accuracy of the solution obtained computing the translational and rotational error of the estimate and (ii) the time required to achieve that solution.

We compared the recommended PCL registration suite that uses the Horn formulas against our framework with and without AD. Furthermore, we also provide results obtained using PCL implementation of the LM optimization algorithm.

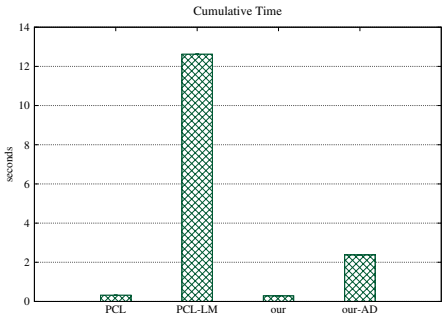
As reported in Table 2, the final registration error is almost negligible in all cases. Instead, in Figure 13, we document the speed of each solver. When using the full potential of our framework—i.e., using analytic Jacobians—it is able to achieve results in general equal or better than the off-the-shelf PCL registration algorithm. Using AD has a great impact on the iteration time, however, our system is able to be faster than the PCL implementation of LM also in this case.

Table 2. Comparison of the final registration error of the optimization result.

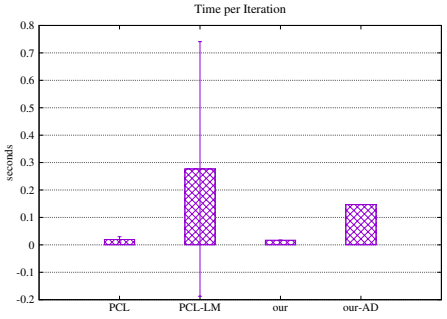
		PCL	PCL-LM	Our	Our-AD
`ICL-NUIM-lr-0`	$e_{pos}[m]$	6.525×10^{-06}	1.011×10^{-04}	1.390×10^{-06}	$\mathbf{6.743 \times 10^{-07}}$
	$e_{rot}[rad]$	1.294×10^{-08}	2.102×10^{-05}	$\mathbf{1.227 \times 10^{-08}}$	9.510×10^{-08}
`ETH-Haupt`	$e_{pos}[m]$	4.225×10^{-06}	2.662×10^{-05}	1.581×10^{-06}	$\mathbf{2.384 \times 10^{-07}}$
	$e_{rot}[rad]$	5.488×10^{-08}	8.183×10^{-06}	$\mathbf{1.986 \times 10^{-08}}$	1.952×10^{-07}
`ETH-Apart`	$e_{pos}[m]$	1.527×10^{-06}	5.252×10^{-05}	$\mathbf{6.743 \times 10^{-07}}$	2.023×10^{-06}
	$e_{rot}[rad]$	7.134×10^{-08}	1.125×10^{-04}	$\mathbf{1.548 \times 10^{-08}}$	1.564×10^{-07}
`bunny`	$e_{pos}[m]$	$\mathbf{1.000 \times 10^{-12}}$	1.352×10^{-05}	1.284×10^{-06}	9.076×10^{-06}
	$e_{rot}[rad]$	$\mathbf{1.515 \times 10^{-07}}$	2.665×10^{-04}	1.269×10^{-06}	5.660×10^{-07}



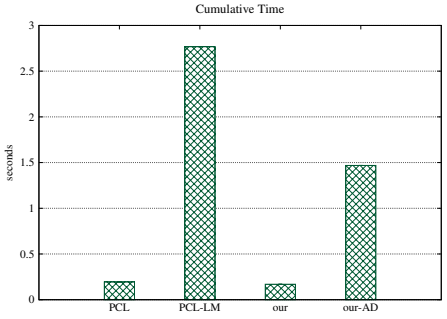
(**a**) Iteration time: `ICL-NUIM - lr-0`



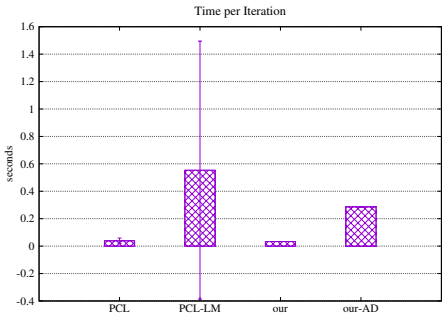
(**b**) Cumulative time: `ICL-NUIM - lr-0`



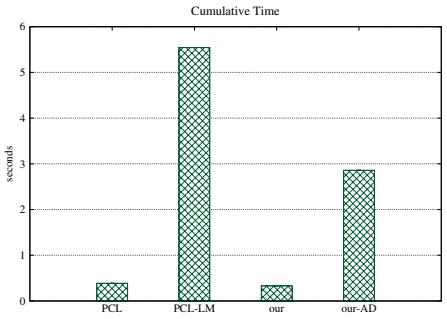
(**c**) Iteration time: `ETH-Hauptgebaude`



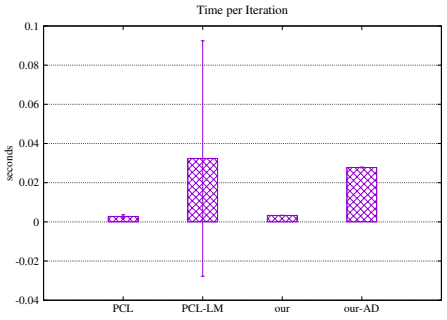
(**d**) Cumulative time: `ETH-Hauptgebaude`



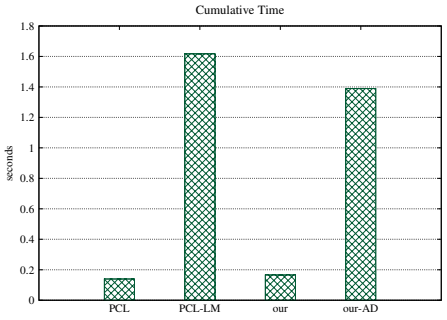
(**e**) Iteration time: `ETH-Apartment`



(**f**) Cumulative time: `ETH-Apartment`



(**g**) Iteration time: `Stanford-Bunny`



(**h**) Cumulative time: `Stanford-Bunny`

Figure 13. Timing analysis of the ILS optimization. In the left column, the mean and standard deviation of a full ILS iteration computed over the 10 total iterations are reported. In the right column, the cumulative time to perform all the iterations is reported instead.

8.2. Sparse Problems

Sparse problems are mostly represented by generic global optimization scenarios, in which the graph has a large number of variables while each factor connects a very small subset of those (typically two). In these kinds of problems, the graph remains unchanged during the iterations; therefore, the most time-consuming part of the optimization is the solution of the linear system, not its

construction. PGO and PLGO are two instances of this problem that are very common in the SLAM context and, therefore, we selected these two to perform comparative benchmarks.

8.2.1. Pose-Graph Optimization

PGO represents the backbone of SLAM systems and it has been well investigated by the research community. For these experiments, we employed standard 3D PGO benchmark datasets—all publicly available [63]. We added to the factors Additive White Gaussian Noise (AWGN) and we initialized the graph using the breadth-first initialization.

We report in Table 3 the complete specifications of the datasets employed together with the noise statistics used. Given the probabilistic nature of the noise imposed on the factors, we performed experiments over 10 noise realizations and we report here the statistics of the results obtained—i.e., mean and standard deviation. To avoid any bias in the comparison, we used the native LM implementation of each framework, since it was the only algorithm common to all candidates. Furthermore, we imposed a maximum number of 100 LM iterations. Still, each framework has its own termination criterion active, so that each one can detect when to stop the optimization. Finally, no robust kernel has been employed in these experiments.

Table 3. Specifications of PGO datasets.

Dataset	Variables	Factors	Noise Σ_t $[m]$	Noise Σ_R $[rad]$
`kitti-00`	4541	5595	$\text{diag}(0.05, 0.05, 0.05)$	$\text{diag}(0.01, 0.01, 0.01)$
`sphere-b`	2500	9799	$\text{diag}(0.10, 0.10, 0.10)$	$\text{diag}(0.05, 0.05, 0.05)$
`torus-b`	1000	1999	$\text{diag}(0.10, 0.10, 0.10)$	$\text{diag}(0.05, 0.05, 0.05)$

In Table 4, we illustrate the Absolute Trajectory Error (ATE) (RMSE) computed on the optimized graph with respect to the ground truth. The values reported refer to mean and standard deviation over all noise trials. As expected, the result obtained are in line with all other methods.

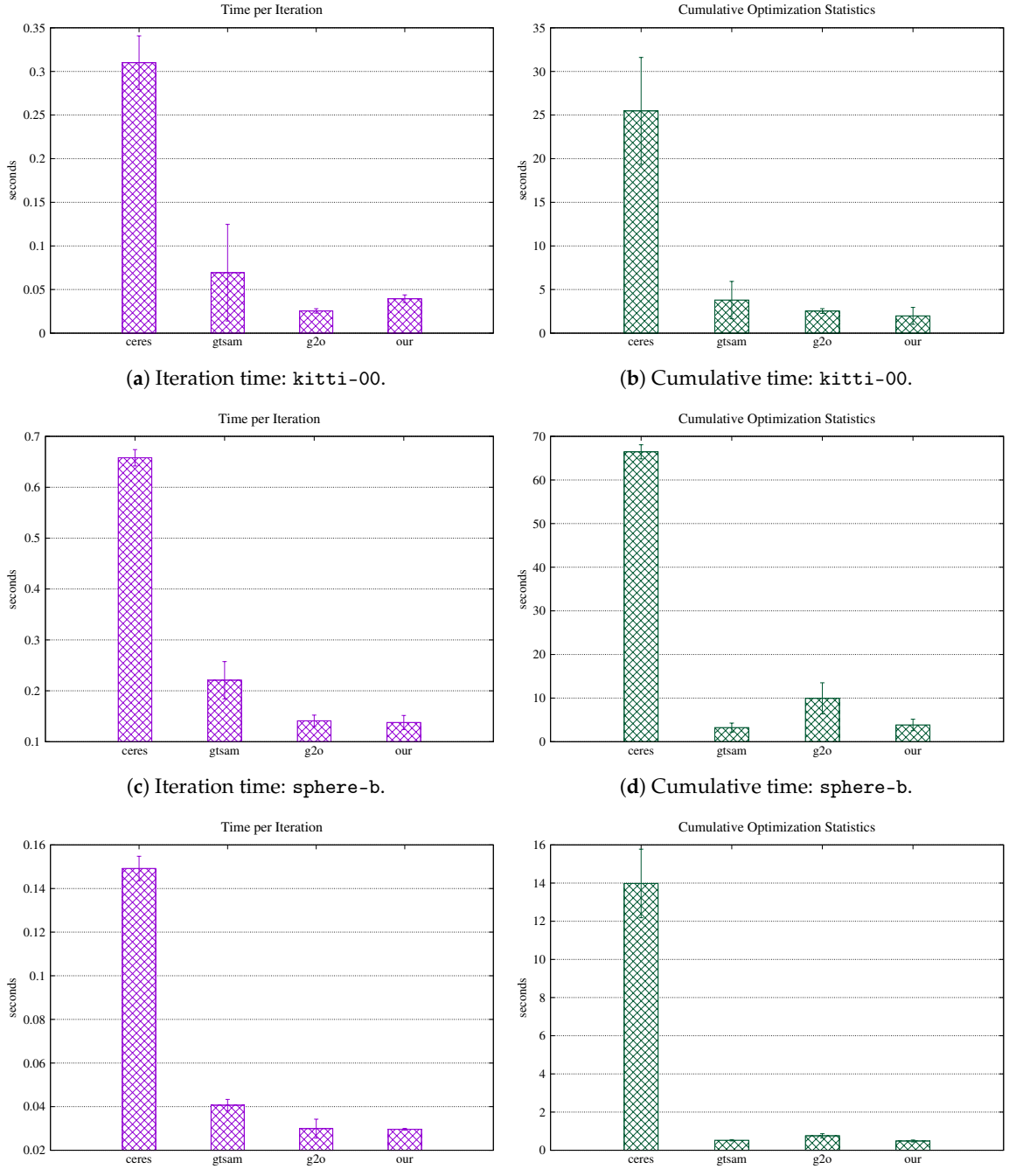
Figure 14, instead, reports a detailed timing analysis. The time to perform a complete LM iteration is always among the smallest, with a very narrow standard deviation. Furthermore, since the specific implementation of LM is slightly different in each framework, we also reported the total time to perform the full optimization, while the number of LM iterations elapsed are shown in Table 5. In addition, in this case, our system is able to achieve state-of-the-art performances that are better than or equal to the other approaches.

Table 4. Comparison of the ATE (RMSE) of the optimization result—mean and standard deviation.

		CERES	g^2o	GTSAM	Our
`kitti-00`	$\text{ATE}_{pos}[m]$	96.550 ± 36.680	94.370 ± 39.590	$\mathbf{77.110 \pm 41.870}$	95.290 ± 38.180
	$\text{ATE}_{rot}[rad]$	1.107 ± 0.270	0.726 ± 0.220	$\mathbf{0.579 \pm 0.310}$	0.720 ± 0.230
`sphere-b`	$\text{ATE}_{pos}[m]$	83.210 ± 7.928	$\mathbf{9.775 \pm 4.003}$	55.890 ± 12.180	26.060 ± 16.350
	$\text{ATE}_{rot}[rad]$	2.135 ± 0.282	$\mathbf{0.150 \pm 0.160}$	0.861 ± 0.170	0.402 ± 0.274
`torus-b`	$\text{ATE}_{pos}[m]$	14.130 ± 1.727	$\mathbf{2.232 \pm 0.746}$	8.041 ± 1.811	3.691 ± 1.128
	$\text{ATE}_{rot}[rad]$	2.209 ± 0.3188	$\mathbf{0.121 \pm 0.0169}$	0.548 ± 0.082	0.156 ± 0.0305

Table 5. Comparison of the number of LM iterations to reach convergence—mean values.

	CERES	g^2o	GTSAM	Our
`kitti-00`	81.70	99.50	69.40	**49.0**
`sphere-b`	101.0	70.90	**15.50**	27.40
`torus-b`	93.50	12.90	25.50	**16.40**



(**a**) Iteration time: `kitti-00`.

(**b**) Cumulative time: `kitti-00`.

(**c**) Iteration time: `sphere-b`.

(**d**) Cumulative time: `sphere-b`.

(**e**) Iteration time: `torus-b`.

(**f**) Cumulative time: `torus-b`.

Figure 14. Timing analysis of different optimization frameworks. The left column reports the mean and standard deviation of the time to perform a complete LM iteration. The right column, instead, illustrates the total time to reach convergence—mean and standard deviation.

8.2.2. Pose-Landmark Graph Optimization

PLGO is another common global optimization task in SLAM. In this case, the variables contain both robot (or camera) poses and landmarks' position in the world. Factors, instead, embody spatial constraints between either two poses or between a pose and a landmark. As a result, these kinds of factor graphs are the perfect representative of the SLAM problem, since they contain the robot trajectory *and* the map of the environment.

To perform the benchmarks, we used two datasets: Victoria Park [77] and KITTI-00 [78]. We obtained the last one running ProSLAM [79] on the stereo data and saving the full output graph. We super-imposed to the factors specific AWGN and we generated the initial guess through the breadth-first initialization technique. Table 6 summarizes the specification of the datasets used in these experiments. In addition, in this case, we sampled multiple noise trials (5 samples) and reported mean and standard deviation of the results obtained. The configuration of the framework is the same as the one used in PGO experiments—i.e., 100 LM iterations at most, with termination criterion active.

As reported in Table 7, the ATE (RMSE) that we obtain is compatible with the one of the other frameworks. The higher error in the `kitti-00-full` dataset is mainly due to the slow convergence of LM that triggers the termination criterion too early, as shown in Table 8. In such case, the use of GN leads to better results; however, in order to not bias the evaluation, we choose to not report results obtained with different ILS algorithms.

Table 6. Specification of PLGO datasets.

Dataset	Variables	Factors	Noise Σ_t $[m]$	Noise Σ_R $[rad]$	Noise Σ_{land} $[m]$
`victoria-park`	7120	10608	$\text{diag}(0.05, 0.05)$	0.01	$\text{diag}(0.05, 0.05)$
`kitti-00-full`	123215	911819	$\text{diag}(0.05, 0.05, 0.05)$	$\text{diag}(0.01, 0.01, 0.01)$	$\text{diag}(0.05, 0.05, 0.05)$

Table 7. Comparison of the ATE (RMSE) of the optimization result—mean and standard deviation.

		CERES	g^2o	GTSAM	Our
`victoria-park`	$\text{ATE}_{pos}[m]$	37.480 ± 21.950	29.160 ± 37.070	$\mathbf{2.268 \pm 0.938}$	5.459 ± 3.355
	$\text{ATE}_{rot}[rad]$	0.515 ± 0.207	0.401 ± 0.461	$\mathbf{0.030 \pm 0.007}$	0.056 ± 0.028
`kitti-00-full`	$\text{ATE}_{pos}[m]$	134.9 ± 29.160	31.14 ± 27.730	$\mathbf{30.97 \pm 18.150}$	135.4 ± 27.000
	$\text{ATE}_{rot}[rad]$	1.137 ± 0.268	$\mathbf{0.173 \pm 0.157}$	0.174 ± 0.104	0.850 ± 0.148

Table 8. Comparison of the number of LM iterations to reach convergence—mean values.

	CERES	CERES-Schur	g^2o	g^2o-Schur	GTSAM	GTSAM-Seq	Our
`victoria-pack`	101.0	101.0	66.8	66.0	43.6	43.6	**36.0**
`kitti-00-full`	101.0	101.0	100.0	100.0	100.0	100.0	**2.0**

As for the wall times to perform the optimization, the results are illustrated in Figure 15. In PLGO scenarios, given the fact that there are two types of factors, the linear system in Equation (19) can be rearranged as follows:

$$\begin{pmatrix} \mathbf{H}_{pp} & \mathbf{H}_{pl} \\ \mathbf{H}_{pl}^{\top} & \mathbf{H}_{ll} \end{pmatrix} \begin{pmatrix} \Delta\mathbf{x}_p \\ \Delta\mathbf{x}_l \end{pmatrix} = \begin{pmatrix} -\mathbf{b}_p \\ -\mathbf{b}_l \end{pmatrix}. \tag{65}$$

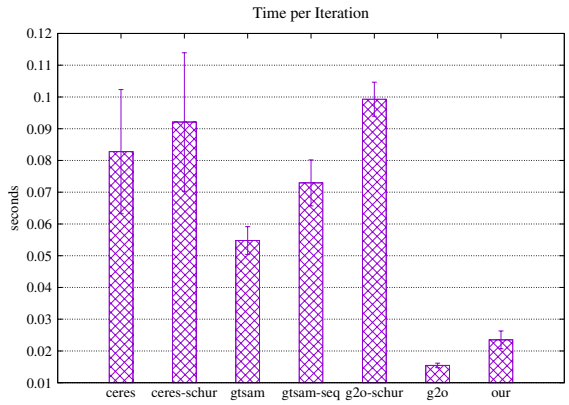
A linear system with this structure can be solved more efficiently through the Schur complement of the Hessian matrix [80], namely:

$$(\mathbf{H}_{pp} - \mathbf{H}_{pl}\mathbf{H}_{ll}^{-1}\mathbf{H}_{pl}^{\top})\Delta\mathbf{x}_p = -\mathbf{b}_p + \mathbf{H}_{pl}\mathbf{H}_{ll}^{-1}\mathbf{b}_l \tag{66}$$

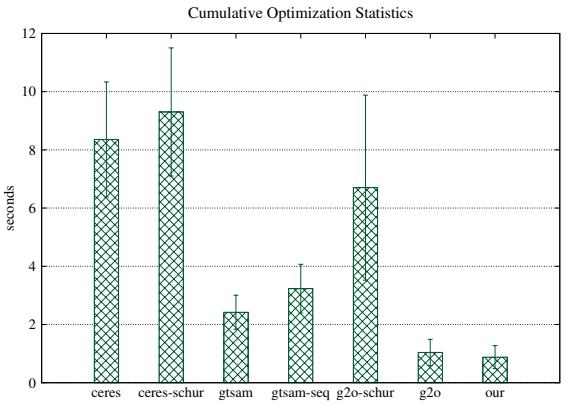
$$\mathbf{H}_{ll}\Delta\mathbf{x}_l = -\mathbf{b}_l + \mathbf{H}_{pl}^{\top}\Delta\mathbf{x}_p. \tag{67}$$

Ceres-Solver and g^2o can make use of the Schur complement to solve this kind of special problem; therefore, we also reported the wall times of the optimization when this technique is used. Obviously, using the Schur complement leads to a major improvement in the efficiency of the linear solver, leading to very low iteration times. For completeness, we reported the results of GTSAM with two different linear solvers: `cholesky_multifrontal` and `cholesky_sequential`. Our framework does

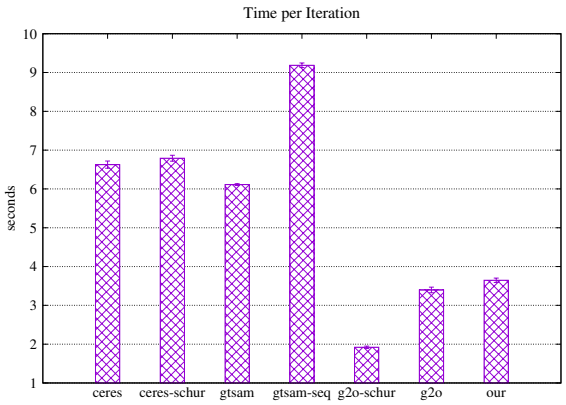
not provide at the moment any implementation of a Schur-complement-based linear solver; still, the performance achieved is in line with all the non-Schur methods, confirming our conjectures.



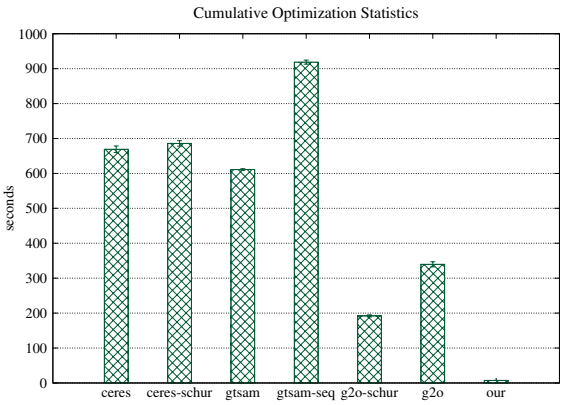
(**a**) Iteration time: `victoria-park`.



(**b**) Cumulative time: `victoria-park`.



(**c**) Iteration time: `kitti-00-full`.



(**d**) Cumulative time: `kitti-00-full`.

Figure 15. Timing analysis: the left column illustrates the time to perform a complete LM iteration; the right column reports the total time to complete the optimization. All values are mean and standard deviation computed over five noise realizations.

9. Conclusions

In this work, we propose a generic overview on ILS optimization for factor graphs in the fields of robotics and computer vision. Our primary contribution is providing a unified and complete methodology to design efficient solutions to generic problems. This paper analyzes in a probabilistic flavor the *mathematical fundamentals* of ILS, addressing also many important collateral aspects of the problem such as dealing with non-Euclidean spaces and with outliers, exploiting the sparsity or the density. Then, we propose a set of common use-cases that exploit the theoretic reasoning previously done.

In the second half of the work, we investigate how to *design* an efficient system that can work in all the possible scenarios depicted before. This analysis led us to the *development* of a novel ILS solver, focused on efficiency, flexibility, and compactness. The system is developed in modern C++ and almost entirely self-contained in less than 6000 lines of code. Our system can seamlessly deal with sparse/dense, static/dynamic problems with a unified consistent interface. Furthermore, thanks to specific implementation designs, it allows easy prototyping of new factors and variables or to intervene at a low level when performances are critical. Finally, we provide an extensive evaluation of the system's performances, both in dense—for example, ICP—and sparse—for example, batch global

optimization—scenarios. The evaluation shows that the performances achieved are in line with contemporary state-of-the-art frameworks, both in terms of accuracy and speed.

Author Contributions: All authors contributed to the design and development of the software. I.A. and M.C. performed the experiments presented in this paper. G.G. and I.A. wrote the manuscript. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. SE(3) Mappings

In this section, we will assume that a SE(3) variable $\mathbf{X}$ is composed as follows:

$$\mathbf{X} \in \mathrm{SE}(3) = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}_{1\times 3} & 1 \end{bmatrix}. \tag{A1}$$

A possible minimal representation for this object could be using three Cartesian coordinates for the position and the three Euler angles for the orientation, namely

$$\boldsymbol{\Delta}x = \begin{bmatrix} \Delta x & \Delta y & \Delta z & \Delta\phi & \Delta\gamma & \Delta\psi \end{bmatrix}^\top = \begin{bmatrix} \boldsymbol{\Delta}\mathbf{t} & \boldsymbol{\Delta}\theta \end{bmatrix}^\top \tag{A2}$$

To pass from one representation to the other, we should define suitable mapping functions. In this case, we use the following notation:

$$\boldsymbol{\Delta}\mathbf{X} = \mathrm{v2t}(\boldsymbol{\Delta}\mathbf{x}) \tag{A3}$$

$$\boldsymbol{\Delta}\mathbf{x} = \mathrm{t2v}(\boldsymbol{\Delta}\mathbf{X}). \tag{A4}$$

In more detail, the function v2t computes the SE(3) isometry reported in Equation (A1) from the six-dimensional vector $\boldsymbol{\Delta}\mathbf{x}$ in Equation (A2). While the translational component of $\mathbf{X}$ can be recovered straightforwardly from $\boldsymbol{\Delta}\mathbf{x}$, the rotational part requires composing the rotation matrices around each axis, leading to the following relation:

$$\mathbf{R}(\boldsymbol{\Delta}\theta) = \mathbf{R}(\Delta\phi, \Delta\gamma, \Delta\psi) = \mathbf{R}_x(\Delta\phi)\ \mathbf{R}_y(\Delta\gamma)\ \mathbf{R}_z(\Delta\psi). \tag{A5}$$

In Equation (A5), $\mathbf{R}_x, \mathbf{R}_y, \mathbf{R}_z$ represent elementary rotations around the x, y and z axis. Summarizing, we can expand Equation (A3) as:

$$\boldsymbol{\Delta}\mathbf{X} = \mathrm{v2t}(\boldsymbol{\Delta}x) = \begin{bmatrix} \mathbf{R}(\boldsymbol{\Delta}\theta) & \boldsymbol{\Delta}t \\ \mathbf{0}_{1\times 3} & 1 \end{bmatrix}$$

Expanding Equation (A5) and performing all the multiplications, the rotation matrix $\mathbf{R}(\boldsymbol{\Delta}\theta)$ is computed as follows:

$$\mathbf{R}(\boldsymbol{\Delta}\theta) = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} = \begin{bmatrix} c(\Delta\gamma)\, c(\Delta\psi) & -c(\Delta\gamma)\, s(\Delta\psi) & s(\Delta\gamma) \\ a & b & -c(\Delta\gamma)\, s(\Delta\phi) \\ c & d & c(\Delta\gamma)\, c(\Delta\phi) \end{bmatrix} \tag{A6}$$

where $c(\cdot)$ and $s(\cdot)$ indicate the cosine and sine of an angle respectively, while

$$\begin{aligned} a &= c(\Delta\phi)\, s(\Delta\psi) + s(\Delta\phi)\, c(\Delta\psi)\, s(\Delta\gamma) \\ b &= c(\Delta\phi)\, c(\Delta\psi) - s(\Delta\phi)\, s(\Delta\gamma)\, s(\Delta\psi) \\ c &= s(\Delta\phi)\, s(\Delta\psi) - c(\Delta\phi)\, c(\Delta\psi)\, s(\Delta\gamma) \\ d &= s(\Delta\phi)\, c(\Delta\psi) + c(\Delta\phi)\, s(\Delta\gamma)\, s(\Delta\psi). \end{aligned}$$

On the contrary, through Equation (A4), we compute the minimal parametrization $\mathbf{\Delta x}$ starting from $\mathbf{\Delta X}$. Again, while the translational component of $\mathbf{\Delta x}$ can be retrieved easily from the isometry. The rotational component $\mathbf{\Delta}\theta$—i.e., the three Euler angles—should be computed starting from the rotation matrix in Equation (A6), in formulae:

$$\Delta\phi = \operatorname{atan2}\left(\frac{-r_{23}}{r_{33}}\right) \quad \Delta\psi = \operatorname{atan2}\left(\frac{-r_{12}}{r_{11}}\right) \quad \Delta\gamma = \operatorname{atan2}\left(\frac{r_{13}}{r_{11} \cdot \frac{1}{c(\Delta\psi)}}\right).$$

Other minimal parametrizations of $\mathrm{SE}(3)$ can be used, and they typically differ on how they represent the rotational component of the $\mathrm{SE}(3)$ object. Common alternatives to Euler angles are unit quaternions and axis-angle. Clearly, changing the minimal parametrization will affect the Jacobians too, and thus the entire optimization process.

Appendix B. ICP Jacobian

In this section, we will provide the reader the full mathematical derivation of the Jacobian matrices reported in Section 5.1.2. To this end, we recall that the measurement function for the ICP problem is:

$$\mathbf{h}_k^{\text{icp}}(\mathbf{X}) \triangleq \mathbf{X}^{-1}\mathbf{p} = \mathbf{R}^\top(\mathbf{p} - \mathbf{t}). \tag{A7}$$

If we apply a small state perturbation using the $\boxplus$ operator defined in Equation (43), we obtain:

$$\mathbf{h}^{\text{icp}}(\mathbf{X} \boxplus \mathbf{\Delta}x) = (\text{v2t}(\mathbf{\Delta x}) \cdot \mathbf{X})^{-1}\mathbf{p} = \mathbf{X}^{-1} \cdot \text{v2t}(\mathbf{\Delta x})^{-1}\mathbf{p} = \mathbf{R}^\top(\text{v2t}(\mathbf{\Delta x})^{-1}\mathbf{p} - \mathbf{t}). \tag{A8}$$

To compute the Jacobian matrix $\mathbf{J}^{\text{icp}}$, we should derive Equation (A8) with respect to $\mathbf{\Delta x}$. Note that the translation vector $\mathbf{t}$ is constant with respect to the perturbation, so it will have no impact in the computation. Furthermore, $\mathbf{R}^\top$ represents a constant multiplicative factor. Since the state perturbation is local, its magnitude is small enough to make the following approximation hold:

$$\mathbf{R}(\mathbf{\Delta}\theta) \approx \begin{bmatrix} 1 & -\mathbf{\Delta}\psi & \mathbf{\Delta}\gamma \\ \mathbf{\Delta}\psi & 1 & -\mathbf{\Delta}\phi \\ -\mathbf{\Delta}\gamma & \mathbf{\Delta}\phi & 1 \end{bmatrix}. \tag{A9}$$

Finally, the Jacobian $\mathbf{J}^{\text{icp}}$ is computed as follows:

$$\begin{aligned} \mathbf{J}^{\text{icp}}(\mathbf{X}) &= \frac{\partial\left(\mathbf{h}^{\text{icp}}(\mathbf{X} \boxplus \mathbf{\Delta x})\right)}{\partial \mathbf{\Delta x}}\Big|_{\mathbf{\Delta x}=\mathbf{0}} = \mathbf{R}^\top \frac{\partial\left(\text{v2t}(\mathbf{\Delta x})^{-1}\mathbf{p}\right)}{\partial \mathbf{\Delta x}}\Big|_{\mathbf{\Delta x}=\mathbf{0}} = \\ &= -\mathbf{R}^\top \begin{bmatrix} \frac{\partial\left(\text{v2t}(\mathbf{\Delta x})^{-1}\mathbf{p}\right)}{\partial \mathbf{\Delta t}}\Big|_{\mathbf{\Delta x}=\mathbf{0}} & \frac{\partial\left(\text{v2t}(\mathbf{\Delta x})^{-1}\mathbf{p}\right)}{\partial \mathbf{\Delta}\theta}\Big|_{\mathbf{\Delta x}=\mathbf{0}} \end{bmatrix} = -\mathbf{R}^\top \begin{bmatrix} \mathbf{I}_{3\times 3} & -[\mathbf{p}]_\times \end{bmatrix} \end{aligned} \tag{A10}$$

where $[\mathbf{p}]_\times$ is the skew-symmetric matrix constructed out of $\mathbf{p}$.

Appendix C. Projective Registration Jacobian

In this section, we will provide the complete derivation of the Jacobians in the context of projective registration. From Equation (50), we know that the prediction is computed as

$$\mathbf{h}_k^{\text{reg}}(\mathbf{X}) = \hom(\mathbf{K}[\mathbf{h}^{icp}(\mathbf{p}^{\text{m}}_{\text{j}(k)})]).$$

Furthermore, as stated in Equation (52), we can compute the Jacobian for this factor through the chain rule, leading to the following relation:

$$\mathbf{J}^{\text{reg}}(\mathbf{X}) = \frac{\partial \hom(\mathbf{v})}{\partial \mathbf{v}}\bigg|_{\mathbf{v}=\mathbf{p}^{\text{cam}}} \mathbf{K}\,\mathbf{J}^{\text{icp}}(\mathbf{X}) = \mathbf{J}^{\hom}(\mathbf{p}^{\text{cam}})\,\mathbf{K}\,\mathbf{J}^{\text{icp}}(\mathbf{X}). \tag{A11}$$

where $\mathbf{p}^{\text{cam}} = \mathbf{K}\mathbf{X}^{-1}\mathbf{p}^{\text{m}}$. Since we already computed $\mathbf{J}^{\text{icp}}$ in Equation (A10), in the remainder of this section, we will focus only on $\mathbf{J}^{\hom}$—i.e., the contribution of the homogeneous division. Given that the function $\hom(\cdot)$ is defined as

$$\hom([x\ y\ z]^\top) = \begin{bmatrix} x/z \\ y/z \end{bmatrix}$$

the Jacobian $\mathbf{J}^{\hom}(\mathbf{p}^{cam})$ is computed as follows:

$$\mathbf{J}^{\hom}(\mathbf{p}^{\text{cam}}) = \begin{bmatrix} \frac{1}{p_z^{cam}} & 0 & \frac{-p_x^{\text{cam}}}{(p_z^{\text{cam}})^2} \\ 0 & \frac{1}{p_z^{cam}} & \frac{-p_y^{\text{cam}}}{(p_z^{\text{cam}})^2}. \end{bmatrix} \tag{A12}$$

Appendix D. Bundle Adjustment Jacobian

In this section, we address the computation of the Jacobians in the context of Bundle Adjustment. The scenario is the one described in Section 5.3. We recall that, in this case, each factor involves two state variables, namely a pose and a landmark. Therefore, $\mathbf{J}^{\text{ba}}$ has the following pattern:

$$\mathbf{J}_k^{\text{ba}} = \begin{pmatrix} \mathbf{0} & \cdots & \mathbf{0} & \mathbf{J}^{\text{ba}}_{k,\text{n}(k)} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{J}^{\text{ba}}_{k,\text{m}(k)} & \mathbf{0} & \cdots & \mathbf{0} \end{pmatrix}.$$

In more detail, the two non-zero blocks embody the derivatives computed with respect to the two active variables, namely:

$$\mathbf{J}^{\text{ba}}_{pose} = \frac{\partial \left(\mathbf{e}_k(\mathbf{X}^{cam} \boxplus \boldsymbol{\Delta}\mathbf{x}^{cam}, \mathbf{x}^{land}) \right)}{\partial \boldsymbol{\Delta}\mathbf{x}^{cam}} \qquad \mathbf{J}^{\text{ba}}_{land} = \frac{\partial \left(\mathbf{e}_k(\mathbf{X}^{cam}, \mathbf{x}^{land} \boxplus \boldsymbol{\Delta}\mathbf{x}^{land}) \right)}{\partial \boldsymbol{\Delta}\mathbf{x}^{land}} \tag{A13}$$

where $\mathbf{e}_k$ represents the error for the k-th factor, computed according to Equation (55).

Unrolling the multiplications, we note that $\mathbf{J}^{\text{ba}}_{pose}$ is the same as the one computed in Equation (A12)—i.e., in the projective registration example. The derivatives relative to the landmark—i.e., $\mathbf{J}^{\text{ba}}_{pose}$—can be straightforwardly computed from Equation (A8), considering that the derivation is with respect to the landmark perturbation this time. In formulae:

$$\mathbf{J}^{\text{ba}}_{land}(\mathbf{X}) = \mathbf{J}^{\hom}(\mathbf{p}^{\text{cam}})\mathbf{K}\mathbf{R}^\top. \tag{A14}$$

Summarizing, the complete Bundle Adjustment Jacobian is computed as:

$$\begin{aligned} \mathbf{J}_k^{\text{ba}} &= \begin{pmatrix} \mathbf{0} & \cdots & \mathbf{0} & \mathbf{J}^{\text{ba}}_{pose} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{J}^{\text{ba}}_{land} & \mathbf{0} & \cdots & \mathbf{0} \end{pmatrix} \\ \mathbf{J}^{\text{ba}}_{pose} &= \mathbf{J}^{\hom}(\mathbf{p}^{\text{cam}})\,\mathbf{K}\,\mathbf{J}^{\text{icp}} \qquad \mathbf{J}^{\text{ba}}_{land} = \mathbf{J}^{\hom}(\mathbf{p}^{\text{cam}})\mathbf{K}\mathbf{R}^\top. \end{aligned} \tag{A15}$$

References

1. Grisetti, G.; Kummerle, R.; Stachniss, C.; Burgard, W. A tutorial on graph-based SLAM. *IEEE Trans. Intell. Transp. Syst. Mag.* **2010**, *2*, 31–43.
2. Kümmerle, R.; Grisetti, G.; Burgard, W. Simultaneous calibration, localization, and mapping. In Proceedings of the 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, San Francisco, CA, USA, 25–30 September 2011; pp. 3716–3721.
3. Censi, A.; Franchi, A.; Marchionni, L.; Oriolo, G. Simultaneous calibration of odometry and sensor parameters for mobile robots. *IEEE Trans. Robot. (TRO)* **2013**, *29*, 475–492.
4. Della Corte, B.; Andreasson, H.; Stoyanov, T.; Grisetti, G. Unified motion-based calibration of mobile multi-sensor platforms with time delay estimation. *IEEE Robot. Autom. Lett. (RA-L)* **2019**, *4*, 902–909.
5. Newcombe, R.A.; Izadi, S.; Hilliges, O.; Molyneaux, D.; Kim, D.; Davison, A.J.; Kohli, P.; Shotton, J.; Hodges, S.; Fitzgibbon, A.W. Kinectfusion: Real-time dense surface mapping and tracking. In Proceedings of the 2011 10th IEEE International Symposium on Mixed and Augmented Reality, Basel, Switzerland, 26–29 October 2011; Volume 11, pp. 127–136.
6. Pomerleau, F.; Colas, F.; Siegwart, R.; Magnenat, S. Comparing ICP variants on real-world data sets. *Auton. Robot.* **2013**, *34*, 133–148.
7. Serafin, J.; Grisetti, G. Using extended measurements and scene merging for efficient and robust point cloud registration. *J. Robot. Auton. Syst. (RAS)* **2017**, *92*, 91–106.
8. Kümmerle, R.; Grisetti, G.; Strasdat, H.; Konolige, K.; Burgard, W. g 2 o: A general framework for graph optimization. In Proceedings of the 2011 IEEE International Conference on Robotics and Automation, Shanghai, China, 9–13 May 2011; pp. 3607–3613.
9. Dellaert, F. *Factor Graphs and GTSAM: A Hands-On Introduction*; Technical Report; Georgia Institute of Technology: Atlanta, GA, USA, 2012.
10. Agarwal, S.; Mierle, K.Ceres Solver. Available online: http://ceres-solver.org (accessed on 30 June 2020).
11. Ila, V.; Polok, L.; Solony, M.; Istenic, K. Fast incremental bundle adjustment with covariance recovery. In Proceedings of the 2017 International Conference on 3D Vision (3DV), Qingdao, China, 10–12 October 2017; pp. 175–184.
12. Dellaert, F.; Kaess, M. Factor Graphs for Robot Perception. *Found. Trends Robot.* **2017**, *6*, 1–139.
13. Ila, V.; Polok, L.; Solony, M.; Svoboda, P. SLAM++—A highly efficient and temporally scalable incremental SLAM framework. *Int. J. Robot. Res. (IJRR)* **2017**, *36*, 210–230.
14. Hertzberg, C.; Wagner, R.; Frese, U.; Schröder, L. Integrating generic sensor fusion algorithms with sound state representations through encapsulation of manifolds. *Inf. Fusion* **2013**, *14*, 57–77.
15. Rusu, R.B.; Cousins, S. 3D is here: Point Cloud Library (PCL). In Proceedings of the 2011 IEEE International Conference on Robotics and Automation, Shanghai, China, 9–13 May 2011.
16. Besl, P.J.; McKay, N.D. A method for registration of 3-D shapes. *IEEE Trans. Pattern Anal. Mach. Intell. (TPAMI)* **1992**, *14*, 239.
17. Rosen, D.M.; Carlone, L.; Bandeira, A.S.; Leonard, J.J. SE-Sync: A certifiably correct algorithm for synchronization over the special Euclidean group. *Int. J. Robot. Res. (IJRR)* **2019**, *38*, 95–125.
18. Zhang, Z. A flexible new technique for camera calibration. *IEEE Trans. Pattern Anal. Mach. Intell. (TPAMI)* **2000**, *22*, 1330–1334.
19. Censi, A.; Marchionni, L.; Oriolo, G. Simultaneous maximum-likelihood calibration of robot and sensor parameters. In Proceedings of the 2008 IEEE International Conference on Robotics and Automation, Pasadena, CA, USA, 19–23 May 2008.
20. Di Cicco, M.; Della Corte, B.; Grisetti, G. Unsupervised calibration of wheeled mobile platforms. In Proceedings of the 2016 IEEE International Conference on Robotics and Automation, Stockholm, Sweden, 16–21 May 2016; pp. 4328–4334.
21. Chen, J.; Medioni, G. Object modeling by registration of multiple range images. *Image Vis. Comput.* **1992**, *10*, 145–155.
22. Lu, F.; Milios, E. Robot Pose Estimation in Unknown Environments by Matching 2D Range Scans. *J. Intell. Robot. Syst.* **1997**, *18*, 249–275.
23. Censi, A. An ICP variant using a point-to-line metric. In Proceedings of the 2008 IEEE International Conference on Robotics and Automation, Pasadena, CA, USA, 19–23 May 2008.

24. Censi, A. An accurate closed-form estimate of ICP's covariance. In Proceedings of the 2007 IEEE International Conference on Robotics and Automation, Roma, Italy, 10–14 April 2007; pp. 3167–3172.
25. Segal, A.V.; Haehnel, D.; Thrun, S. Generalized-ICP. In Proceedings of the Robotics: Science and Systems, Seattle, WA, USA, 28 June–1 July 2009; Volume 2.
26. Biber, P.; Straßer, W. The normal distributions transform: A new approach to laser scan matching. In Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No. 03CH37453), Las Vegas, NV, USA, 27–31 October 2003; Volume 3, pp. 2743–2748.
27. Magnusson, M.; Duckett, T.; Lilienthal, A.J. Scan Registration for Autonomous Mining Vehicles Using 3D-NDT. *J. Field Robot. (JFR)* **2007**, *24*, 803–827.
28. Wolf, P.R.; Dewitt, B.A. *Elements of Photogrammetry: With Applications in GIS*; McGraw-Hill: New York, NY, USA, 2000; Volume 3.
29. Fischler, M.A.; Bolles, R.C. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM* **1981**, *24*, 381–395.
30. Engel, J.; Sturm, J.; Cremers, D. Semi-dense visual odometry for a monocular camera. In Proceedings of the IEEE International Conference on Computer Vision, Sydney, NSW, Australia, 1–8 December 2013; pp. 1449–1456.
31. Della Corte, B.; Bogoslavskyi, I.; Stachniss, C.; Grisetti, G. A general framework for flexible multi-cue photometric point cloud registration. In Proceedings of the 2018 IEEE International Conference on Robotics and Automation (ICRA), Brisbane, QLD, Australia, 21–25 May 2018; pp. 1–8.
32. Lu, F.; Milios, E. Globally consistent range scan alignment for environment mapping. *Auton. Robot.* **1997**, *4*, 333–349.
33. Borrmann, D.; Elseberg, J.; Lingemann, K.; Nüchter, A.; Hertzberg, J. Globally consistent 3D mapping with scan matching. *J. Robot. Auton. Syst. (RAS)* **2008**, *56*, 130–142.
34. Dissanayake, M.G.; Newman, P.; Clark, S.; Durrant-Whyte, H.; Csorba, M. A solution to the simultaneous localization and map building (SLAM) problem. *IEEE Trans. Robot. Autom.* **2001**, *17*, 229–241.
35. Davison, A.J.; Murray, D.W. Simultaneous localization and map-building using active vision. *IEEE Trans. Pattern Anal. Mach. Intell.* **2002**, *24*, 865–880.
36. Leonard, J.; Newman, P. Consistent, convergent, and constant-time SLAM. In Proceedings of the International Conference on Artificial Intelligence (IJCAI), Acapulco, Mexico, 9–15 August 2003; pp. 1143–1150.
37. Se, S.; Lowe, D.; Little, J. Mobile robot localization and mapping with uncertainty using scale-invariant visual landmarks. *Int. J. Robot. Res. (IJRR)* **2002**, *21*, 735–758.
38. Castellanos, J.A.; Neira, J.; Tardós, J.D. Limits to the consistency of EKF-based SLAM. *IFAC Proc. Vol.* **2004**, *37*, 716–721.
39. Clemente, L.A.; Davison, A.J.; Reid, I.D.; Neira, J.; Tardós, J.D. Mapping Large Loops with a Single Hand-Held Camera. In Proceedings of the Robotics: Science and Systems (RSS), Atlanta, GA, USA, 27–30 June 2007; Volume 2.
40. Montemerlo, M.; Thrun, S.; Koller, D.; Wegbreit, B. FastSLAM: A factored solution to the simultaneous localization and mapping problem. In Proceedings of the AAAI National Conference on Artificial Intelligence 2002, Edmonton, AB, Canada, 28 July–1 August 2002; pp. 593–598.
41. Montemerlo, M.; Thrun, S.; Koller, D.; Wegbreit, B. FastSLAM 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. In Proceedings of the International Conference on Artificial Intelligence (IJCAI), Acapulco, Mexico, 9–15 August 2003; pp. 1151–1156.
42. Grisetti, G.; Stachniss, C.; Burgard, W. Improved techniques for grid mapping with rao-blackwellized particle filters. *IEEE Trans. Robot. (TRO)* **2007**, *23*, 34.
43. Stachniss, C.; Grisetti, G.; Burgard, W.; Roy, N. Analyzing gaussian proposal distributions for mapping with rao-blackwellized particle filters. In Proceedings of the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems, San Diego, CA, USA, 29 October–2 Novembrt 2007; pp. 3485–3490.
44. Gutmann, J.; Konolige, K. Incremental mapping of large cyclic environments. In Proceedings of the 1999 IEEE International Symposium on Computational Intelligence in Robotics and Automation, CIRA'99 (Cat. No. 99EX375), Monterey, CA, USA, 8–9 November 1999; pp. 318–325.

45. Kaess, M.; Ranganathan, A.; Dellaert, F. iSAM: Fast incremental smoothing and mapping with efficient data association. In Proceedings of the 2007 IEEE International Conference on Robotics and Automation, Roma, Italy, 10–14 April 2007; pp. 1670–1677.
46. Dellaert, F.; Kaess, M. Square Root SAM: Simultaneous localization and mapping via square root information smoothing. *Int. J. Robot. Res. (IJRR)* **2006**, *25*, 1181–1203.
47. Kaess, M.; Johannsson, H.; Roberts, R.; Ila, V.; Leonard, J.J.; Dellaert, F. iSAM2: Incremental smoothing and mapping using the Bayes tree. *Int. J. Robot. Res. (IJRR)* **2012**, *31*, 216–235.
48. Hertzberg, C. *A Framework for Sparse, Nonlinear Least Squares Problems on Manifolds*; Universität Bremen: Bremen, Germany, 2008; Citeseer.
49. Griewank, A.; Juedes, D.; Utke, J. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Softw. (TOMS)* **1996**, *22*, 131–167.
50. Griewank, A. On automatic differentiation. *Math. Program. Recent Dev. Appl.* **1989**, *6*, 83–107.
51. Griewank, A.; Walther, A. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*; SIAM: Philadelphia, PA, USA, 2008; Volume 105.
52. Kaess, M.; Dellaert, F. Covariance recovery from a square root information matrix for data association. *J. Robot. Auton. Syst. (RAS)* **2009**, *57*, 1198–1210.
53. Boyd, S.; Vandenberghe, L. *Convex optimization*; Cambridge University Press: Cambridge, UK, 2004.
54. Briales, J.; Gonzalez-Jimenez, J. Cartan-sync: Fast and global SE (d)-synchronization. *IEEE Robot. Autom. Lett. (RA-L)* **2017**, *2*, 2127–2134.
55. Bai, F.; Vidal-Calleja, T.; Huang, S. Robust incremental SLAM under constrained optimization formulation. *IEEE Robot. Autom. Lett. (RA-L)* **2018**, *3*, 1207–1214.
56. Ni, K.; Dellaert, F. Multi-level submap based slam using nested dissection. In Proceedings of the 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems, Taipei, Taiwan, 18–22 October 2010; pp. 2558–2565.
57. Grisetti, G.; Kümmerle, R.; Ni, K. Robust optimization of factor graphs by using condensed measurements. In Proceedings of the 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, Vilamoura, Portugal, 7–12 October 2012; pp. 581–588.
58. Yang, H.; Antonante, P.; Tzoumas, V.; Carlone, L. Graduated non-convexity for robust spatial perception: From non-minimal solvers to global outlier rejection. *IEEE Robot. Autom. Lett. (RA-L)* **2020**, *5*, 1127–1134.
59. Schön, T.B.; Lindsten, F. Manipulating the multivariate Gaussian density. *Div. Automat. Control Linköping Univ. Linköping Sweden Tech. Rep* **2011**, Linköping, Sweden.
60. Lee, J.M. Smooth manifolds. In *Introduction to Smooth Manifolds*; Springer: New York, NY, USA 2013; pp. 1–31.
61. Smith, R.C.; Cheeseman, P. On the representation and estimation of spatial uncertainty. *Int. J. Robot. Res. (IJRR)* **1986**, *5*, 56–68.
62. Hertzberg, C.; Wagner, R.; Frese, U. Tutorial on quick and easy model fitting using the SLoM framework. In *International Conference on Spatial Cognition*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 128–142.
63. Aloise, I.; Grisetti, G. Chordal Based Error Function for 3D Pose-Graph Optimization. *IEEE Robot. Autom. Lett.* **2019**, *5*, 274–281.
64. MacTavish, K.; Barfoot, T.D. At all costs: A comparison of robust cost functions for camera correspondence outliers. In Proceedings of the 2015 12th Conference on Computer and Robot Vision, Halifax, NS, Canada, 3–5 June 2015; pp. 62–69.
65. Saad, Y. *Iterative Methods for Sparse Linear Systems*; SIAM: Philadelphia, PA, USA, 2003; Volume 82.
66. Amestoy, P.R.; Davis, T.A.; Duff, I.S. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl.* **1996**, *17*, 886–905.
67. Davis, T.A. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw. (TOMS)* **2004**, *30*, 165–195.
68. Karypis, G.; Kumar, V. Multilevelk-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.* **1998**, *48*, 96–129.
69. Davis, T.A. *Direct Methods for Sparse Linear Systems*; SIAM: Philadelphia, PA, USA, 2006; Volume 2.
70. Agarwal, P.; Olson, E. Variable reordering strategies for SLAM. In Proceedings of the 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, Vilamoura, Portugal, 7–12 October 2012; pp. 3844–3850.

71. Lourakis, M.I.; Argyros, A.A. SBA: A software package for generic sparse bundle adjustment. *ACM Trans. Math. Softw. (TOMS)* **2009**, *36*, 2.
72. Conn, A.R.; Gould, N.I.; Toint, P.L. *Trust Region Methods*; SIAM: Philadelphia, PA, USA, 2000; Volume 1.
73. Guennebaud, G.; Jacob, B. Eigen v3. 2010. Available online: http://eigen.tuxfamily.org (accessed on 30 June 2020).
74. Handa, A.; Whelan, T.; McDonald, J.; Davison, A.J. A benchmark for RGB-D visual odometry, 3D reconstruction and SLAM. In Proceedings of the 2014 IEEE international conference on Robotics and automation (ICRA), Hong Kong, China, 31 May–7 June 2014; pp. 1524–1531.
75. Pomerleau, F.; Liu, M.; Colas, F.; Siegwart, R. Challenging data sets for point cloud registration algorithms. *Int. J. Robot. Res. (IJRR)* **2012**, *31*, 1705–1711.
76. Curless, B.; Levoy, M. A volumetric method for building complex models from range images. In Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, New Orleans, LA, USA, 4–9 August 1996.
77. Guivant, J.E.; Nebot, E.M. Optimization of the simultaneous localization and map-building algorithm for real-time implementation. *IEEE Trans. Robot. Autom.* **2001**, *17*, 242–257.
78. Geiger, A.; Lenz, P.; Urtasun, R. Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. In Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition, Providence, RI, USA, 16–21 June 2012.
79. Schlegel, D.; Colosi, M.; Grisetti, G. Proslam: Graph SLAM from a programmer's perspective. In Proceedings of the 2018 IEEE International Conference on Robotics and Automation (ICRA), Brisbane, QLD, Australia, 21–25 May 2018; pp. 1–9.
80. Frese, U. A proof for the approximate sparsity of SLAM information matrices. In Proceedings of the 2005 IEEE International Conference on Robotics and Automation, Barcelona, Spain, 18–22 April 2005; pp. 329–335.