SLAM Handbook

From Localization and Mapping to Spatial Intelligence

Edited by

Luca Carlone, Ayoung Kim, Frank Dellaert,
Timothy Barfoot, and Daniel Cremers

Together with all contributors

Henrik Andreasson
Arash Asgharivaskasi
Nikolay Atanasov
Timothy Barfoot
Jens Behley
Wolfram Burgard
Cesar Cadena
Teresa Vidal Calleja
Marco Camurri
Luca Carlone
Yun Chang
Margarita Chli
Henrik Christensen
Giovanni Cioffi
Javier Civera
Daniel Cremers
Andy Davison
Frank Dellaert
Jia Deng
Kevin Doherty
Jacob Engle
Maurice Fallon
Guillermo Gallego
Cedric Le Gentil
Christoffer Heckman
Connor Holmes
Paul Huang
Shoudong Huang
Nathan Hughes
Krishna Murthy Jatavallabhula
Michael Kaess
Kasra Khosoussi
Giseop Kim
Ayoung Kim
John Leonard
Stefan Leutenegger
Lahav Lipson
Martin Magnusson
Joshua Mangelson
Hidenobu Matsuki
Matias Mattamala
José M Martínez Montiel
Mustafa Mukadam
Paul Newman
Helen Oleynikova
Lionel Ott
Marc Pollefeys
Victor Reijgwart
David Rosen
Davide Scaramuzza
Lukas Schmid
Jingnan Shi
Cyrill Stachniss
Niko Sunderhauf
Zach Teed
Chen Wang
Heng Yang
Fu Zhang
Ji Zhang
Shibo Zhao

Contents

Notation

	– General Notation –
a	This font is used for real scalars
$\boldsymbol{a}$	This font is used for real column vectors
$\boldsymbol{A}$	This font is used for real matrices
X	This font is used for sets
$\mathbf{I}$	The identity matrix
$\mathbf{0}$	The zero matrix
$\boldsymbol{A}^\mathsf{T}$	The transpose of matrix $\boldsymbol{A}$
$\mathbb{R}^{M\times N}$	The vector space of real $M \times N$ matrices
$p(\boldsymbol{a})$	The probability density of $\boldsymbol{a}$
$p(\boldsymbol{a}\|\boldsymbol{b})$	The probability density of $\boldsymbol{a}$ given $\boldsymbol{b}$
$\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$	Gaussian probability density with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$
$\mathcal{GP}(\boldsymbol{\mu}(t), \boldsymbol{\mathcal{K}}(t,t'))$	Gaussian process with mean function, $\boldsymbol{\mu}(t)$, and covariance function, $\boldsymbol{\mathcal{K}}(t,t')$
$\hat{(\cdot)}$	A posterior (estimated) quantity
$\check{(\cdot)}$	A prior quantity
$(\cdot)_k$	The value of a quantity at timestep k
$(\cdot)_{k_1:k_2}$	The set of values of a quantity from timestep k_1 to timestep k_2, inclusive
$\|\cdot\|_1$	L1 norm $\|\boldsymbol{x}\|_1 = \sum \|x_i\|$
$\|\cdot\|_2$	L2 norm $\|\boldsymbol{x}\|_2 = \sqrt{\sum x_i^2}$

– 3D Geometry Notation –

$\mathcal{F}^a$	A reference frame in three dimensions
$\boldsymbol{v}^a$	The coordinates of a vector in frame $\mathcal{F}^a$
$\boldsymbol{R}_a^b$	A 3×3 rotation matrix (member of SO(3)) that takes points expressed in $\mathcal{F}^a$ and re-expresses them in (purely rotated) $\mathcal{F}^b$: $\boldsymbol{v}^b = \boldsymbol{R}_a^b \boldsymbol{v}^a$
$\boldsymbol{t}_a^b$	The three-dimensional position of the origin of frame $\mathcal{F}^a$ expressed in $\mathcal{F}^b$
$\tilde{\boldsymbol{v}}^a = \begin{bmatrix} \boldsymbol{v}^a \\ 1 \end{bmatrix}$	A 4×1 homogeneous point expressed in $\mathcal{F}^a$
$\boldsymbol{T}_a^b = \begin{bmatrix} \boldsymbol{R}_a^b & \boldsymbol{t}_a^b \\ \boldsymbol{0} & 1 \end{bmatrix}$	A 4×4 transformation matrix (member of SE(3)) that takes homogeneous points expressed in $\mathcal{F}^a$ and re-expresses them in (rotated and translated) $\mathcal{F}^b$: $\tilde{\boldsymbol{v}}^b = \boldsymbol{T}_a^b \tilde{\boldsymbol{v}}^a$
SO(3)	The special orthogonal group, a matrix Lie group used to represent rotations
so(3)	The Lie algebra associated with SO(3)
SE(3)	The special Euclidean group, a matrix Lie group used to represent poses
se(3)	The Lie algebra associated with SE(3)
$(\cdot)^\wedge$	An operator mapping a vector in $\mathbb{R}^3$ (resp. $\mathbb{R}^6$) to an element of the Lie algebra for rotations (resp. poses); implements the cross product for three-dimensional quantities, *i.e.,* for two vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^3$, $\boldsymbol{u}^\wedge \boldsymbol{v} = \boldsymbol{u} \times \boldsymbol{v}$
$(\cdot)^\vee$	An operator mapping an element of the Lie algebra for rotations (resp. poses) to a vector in $\mathbb{R}^3$ (resp. $\mathbb{R}^6$)

PART ONE
FOUNDATIONS

1

Prelude

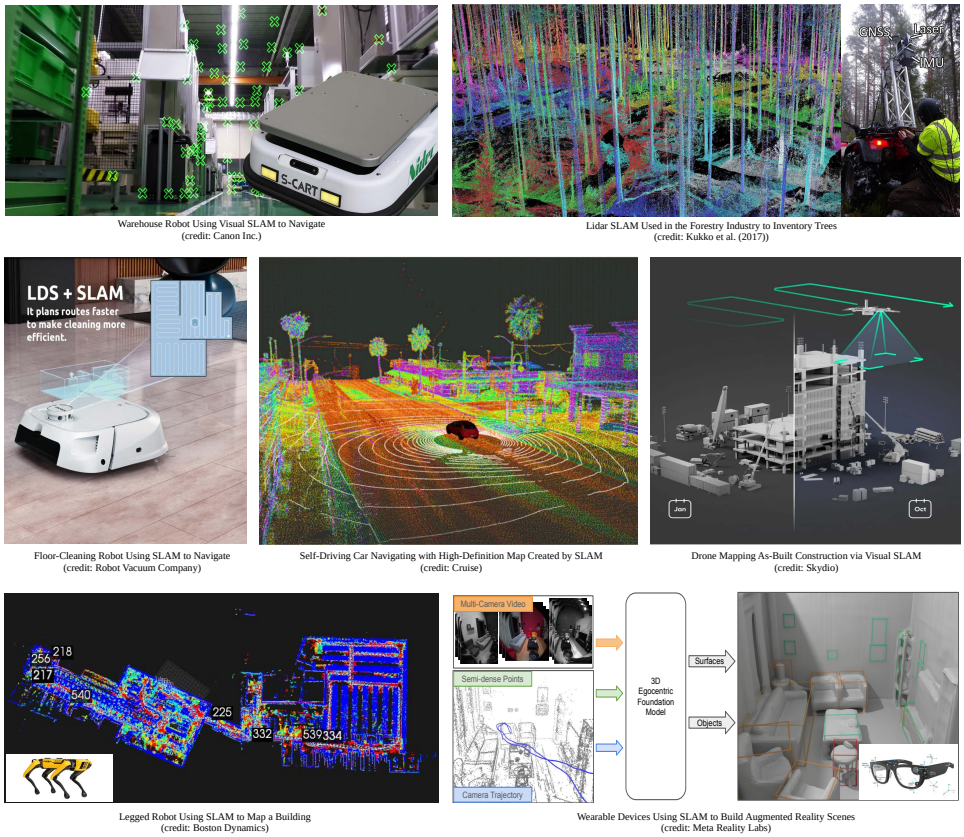
Luca Carlone, Ayoung Kim, Frank Dellaert, Timothy Barfoot, Daniel Cremers

This chapter introduces the Simultaneous Localization and Mapping (SLAM) problem, presents the modules that form a typical SLAM system, and explains the role of SLAM in the architecture of an autonomous system. The chapter also provides a short historical perspective of the topic and discusses how the traditional notion of SLAM is evolving to fully leverage new technological trends and opportunities. The ultimate goal of the chapter is to introduce basic terminology and motivations, and to describe the scope and structure of this handbook.

1.1 What is SLAM?

A necessary prerequisite for a robot to operate safely and effectively in an unknown environment is to form an internal representation of its surroundings. These type of representations can be used to support obstacle avoidance, low-level control, planning, and, more generally, the decision-making processes required for the robot to complete the task it has been assigned. The execution of simple tasks (*e.g.,* following a lane, or maintaining a certain distance to an object in front of the robot) may only require tracking entities of interest in the sensor data streams, and complex tasks (*e.g.,* large-scale navigation or mobile manipulation) require building and maintaining a *persistent representation* (a map) of the environment. Such a map describes the presence of obstacles, objects, and other entities of interest, and their relative location with respect to the robot's pose (position and orientation). For instance, the map might be used instruct the robot to reach a location of interest, to grasp a certain object, or to support the exploration of an initially unknown environment. Figure 1.1 provides some real-world examples of simultaneous localization and mapping (SLAM) in action.

For a robot operating in an initially unknown environment, the problem of building a map of the environment, while concurrently estimating its pose with respect to that map, is referred to as Simultaneous Localization and Mapping (SLAM). SLAM reduces to *localization* if the map is given, in which case the robot only has to estimate its pose with respect to the map. On the other hand, SLAM reduces to *mapping* if the pose of the robot is already known, for instance when an absolute



Warehouse Robot Using Visual SLAM to Navigate (credit: Canon Inc.)

Lidar SLAM Used in the Forestry Industry to Inventory Trees (credit: Kukko et al. (2017))

Floor-Cleaning Robot Using SLAM to Navigate (credit: Robot Vacuum Company)

Self-Driving Car Navigating with High-Definition Map Created by SLAM (credit: Cruise)

Drone Mapping As-Built Construction via Visual SLAM (credit: Skydio)

Legged Robot Using SLAM to Map a Building (credit: Boston Dynamics)

Wearable Devices Using SLAM to Build Augmented Reality Scenes (credit: Meta Reality Labs)

Figure 1.1 SLAM is rapidly becoming an enabling technology in a wide array of applications including warehouse robotics, forest inventories [141], floor-cleaning, self-driving cars, drones surveillance, legged-robot mapping, and augmented reality to name only a few.

positioning system is used (e.g., differential GPS or motion capture), in which case the robot only needs to model its surroundings using its sensor data.

The central role of SLAM in robotics research is due to the fact that robot poses are rarely known in practical applications. Differential GPS and motion capture systems are expensive and restricted to small areas, hence being unsuitable for large-scale robot deployments. Consumer-grade GPS is much more broadly available, but its accuracy (with errors typically in the order of meters) and its availability (which is limited to outdoor areas with line-of-sight to satellites) often makes it unsuitable as a source of localization; the consumer-grade GPS —when available— is typically used as an additional source of information for SLAM, rather than a replacement for the localization aspects of SLAM.

Similarly, in many robotics applications, the robot will not typically have access

to a prior map, hence it needs to perform SLAM rather than localization. Indeed, in certain applications, building a map is actually the *goal* of the robot deployment; for instance, when robots are used to support disaster response and search-and-rescue operations, they might be deployed to construct a map of the disaster site to help first-responders. In other cases, the map might be stale or not have enough detail. For instance, a domestic robot might have access to the floor plan of the apartment it has to operate in, but such a floor plan may not describe the furniture and objects actually present in the environment, nor the fact that these elements can be rearranged from day to day. In a similar manner, Mars exploration rovers have access to low-resolution satellite maps of the Martian surface, but they still need to perform local mapping to guide obstacle avoidance and motion planning.

The importance of the SLAM problem motivates the large amount of attention this topic has received, both within the research community and from practitioners interested in using SLAM technologies across multiple application domains from robotics, to virtual and augmented reality. At the same time, SLAM remains an exciting area of research, with many open problems and new opportunities.

1.2 Anatomy of a Modern SLAM System

The ultimate goal of SLAM is to infer a map representation and robot poses (*i.e.,* trajectory) from sensor data, including data from *proprioceptive* sensors (*e.g.,* wheel odometry or inertial measurement unit, IMU) and *exteroceptive* sensors (*e.g.,* cameras, light detection and ranging (LiDAR)s, radars). In mathematical terms this can be understood as an *inverse problem*: given a set of measurements, determine a model of the world (the map) and a set of robot poses (trajectory) that could have produced those measurements. There exist two alternative strategies to solve the SLAM problem: indirect and direct methods.

The vast majority of SLAM methods prefers pre-processing the raw sensory data in order to extract "intermediate representations" that are compact and easier to describe mathematically. Instead of using every pixel in an image, these methods extract a few distinctive *2D point features* (or keypoints) and then only model the geometry of how these keypoints depend on the pose of the camera and the geometry of the scene. In contrast, rather than computing an intermediate abstraction, direct methods aim to compute localization and mapping *directly* from the raw sensory data. This categorization is prominent in visual SLAM but is not limited to it as we will see in Chapter 9 and Chapter 10. Both indirect and direct methods have their advantages and shortcomings.

Indirect methods are often faster and more memory efficient. Rather than processing every single pixel of each camera image, for example, they merely process a small subset of keypoints for which the 3D location is determined. As a consequence, real-time capable systems for indirect visual SLAM were already available around the year 2000. To date, indirect methods are the preferred approach for real-time

robot vision on platforms with limited compute. Moreover, once the intermediate representation is determined, the subsequent computations are often mathematically simpler, making the resulting inference problems more tractable. In the case of visual SLAM, for example, once a set of corresponding points is identified across a set of images, the resulting problem of localization and mapping amounts to the classical bundle adjustment problem for which a multitude of powerful solvers and approximation methods exist.

In turn, direct methods have the potential to provide superior accuracy because they make use of all available input information. While the processing of all available input information (for example all pixels in each image) is computationally cumbersome and capturing the complex relationship between the quantities of interest (localization and mapping) and the raw input data (e.g. the brightness of each pixel) may create additional non-convexities in the overall loss function, there exist efficient approximation and inference strategies with first real-time capable methods for direct visual SLAM emerging in the 2010s. As we will see in Part II and III, the efficient processing of huge amounts of input data can be facilitated by using graphics processing units (GPUs) to parallelize computations.

In both direct and indirect methods the measurements are used to infer the robot pose and map representation. There is a well-established literature in estimation theory describing how we can infer quantities of interest (in our case, the robot poses and the surrounding map) from observations. This book particularly focuses on estimation theoretic tools —reviewed and tailored to the SLAM problem in Chapter 2 and Chapter 3— that have their foundations in probabilistic inference and that rephrase estimation in terms of solving optimization problems.

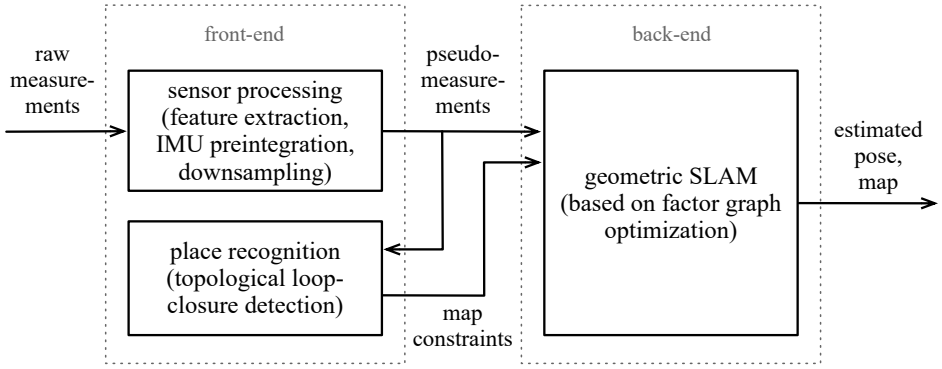


Figure 1.2 The anatomy of a typical SLAM is made up of a *front-end* (to process rich sensing data into more manageable information and to detect topological loop closures) and a *back-end* (to estimate the robot's pose and a geometric map). The back-end often has a number of helper modules aimed at helping with robustness, computational tractability, and map quality.

Indirect methods produce a natural split in common SLAM architectures (Figure 1.2): the raw sensor data is first passed to a set of algorithms (the *SLAM front-end*) in charge of extracting intermediate representations; then such intermediate representations are passed to an estimator (the *SLAM back-end*), that estimates the quantities of interest. The front-end is typically also in charge of building an *initial guess*: this is an initial estimate the back-end can use for iterative optimization, hence mitigating convergence issues due to non-convexity. Let us discuss a few examples to clarify the difference between the SLAM front-end and back-end.

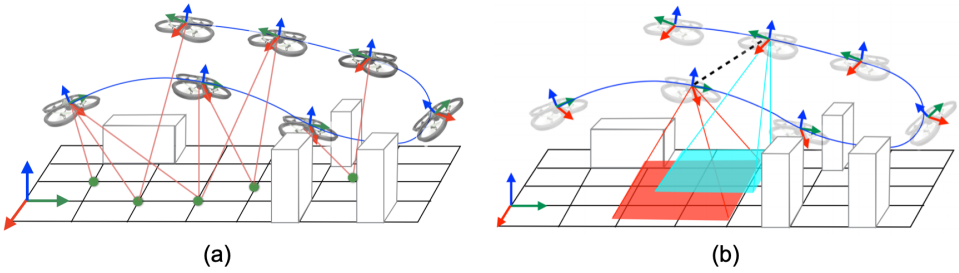


Figure 1.3 (a) In landmark-based SLAM models, the front-end produces measurements to 3D landmarks and the back-end estimates the robot trajectory (as a set of poses) and landmark positions. (b) In pose-graph-based SLAM models, the front-end abstracts the raw sensor measurements in terms of odometry and loop closure measurements (these are typically relative pose measurements) and the back-end estimates the overall robot trajectory.

Example 1.1 (Visual SLAM: from pixels to landmarks)**.** Visual SLAM uses camera images to estimate the robot trajectory and a sparse 3D point cloud map. The typical front-end of a visual SLAM system extracts 2D keypoints and matches them across frames such that each group (a feature track) corresponds to re-observations of the same 3D point (a landmark) across different camera views. The front-end will also compute rough estimates of the camera poses and 3D landmark positions by using computer vision techniques known as *minimal solvers*.[1] Then, the back-end is in charge of estimating (or refining) the unknown 3D position of the landmarks and the robot poses observing them by solving an optimization problem, known as *bundle adjustment*. This example leads to a landmark-based (of feature-based) SLAM model, visualized in Figure 1.3(a). We will discuss visual SLAM at length in Chapter 8.

Example 1.2 (Lidar SLAM: from scans to odometry and loop closures)**.** Lidar

[1] A more subtle point is that minimal solvers will also allow pruning away a large portion of outliers, *i.e.*, incorrect detections of a landmark. This makes the job of the back-end easier, while still allowing it to remove any remaining outlier. We discuss outlier rejection and the related problem of data association in Chapter 4.

SLAM uses lidar scans to estimate the robot trajectory and a map. A common front-end for lidar SLAM consists in using scan matching algorithms (*e.g.,* the Iterative Closest Point or ICP) to compute the relative pose between two lidar scans. In particular, the front-end will match scans taken at consecutive time instants to estimate the relative motion of the robot between them (the so called *odometry*) and will also match scans corresponding to multiple visits to the same place (the so called *loop closures*). Odometry and loop closure measurements are then passed to the back-end that optimizes the robot trajectory by solving an optimization problem, known as *pose-graph optimization*. This example leads to a pose-graph-based SLAM model, visualized in Figure 1.3(b). We discuss LiDAR SLAM in Chapter 9.

The previous examples showcase three popular examples of "intermediate representations" (or pseudo-measurements) that are produced by the front-end and passed to the back-end (Figure 1.2): landmark observations, odometry, and loop closures. In complex SLAM systems, these representations can be used in combination: for instance, in certain visual-SLAM systems one might extract keypoints corresponding to 3D landmarks, and further process them to compute relative poses corresponding to odometry and loop-closures, and finally use a pose-graph-based back-end. The choice of the front-end/back-end split is about selecting a desired trade-off between computation and accuracy. Extracting simpler representations might lead to much faster back-end solvers (*e.g.,* performing pose-graph optimization is typically much faster than doing bundle adjustment); but at the same time abstracting measurements induces approximation in how the measurements are modeled in the back-end, hence leading to small inaccuracies (*e.g.,* bundle adjustment is typically more accurate than pose-graph optimization).

We remark that loop closures are a key aspect of SLAM. If we only use odometry for trajectory estimation, the resulting estimate —obtained by accumulating odometry motion estimates— is bound to drift over time, leading to severe distorsion in the trajectory estimate. Revisiting already visited places is crucial to keep the trajectory estimation error bounded and obtain globally consistent maps. We also remark that loop closures are implicitly captured in landmark-based SLAM, where loop closures correspond to new observations of previously seen landmarks.

We conclude this section by observing how SLAM research cuts across multiple disciplines. The SLAM front-end extracts features from raw sensor data, hence touching disciplines ranging from signal processing, geometry, 2D computer vision, and machine learning. The SLAM back-end performs estimation given measurements from the front-end, hence touching estimation theory, optimization, and applied mathematics. This variety of ideas and influences contribute to making SLAM a fascinating and multi-faceted problem.

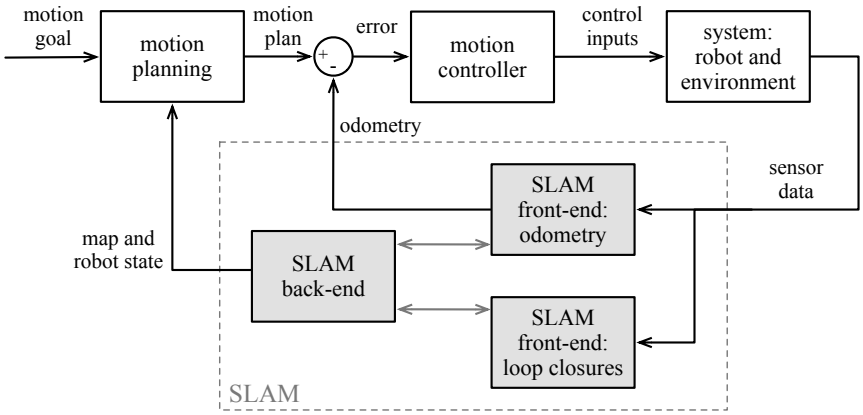


Figure 1.4 SLAM plays an important role in the overall autonomy pipeline of a robot that interacts with the world, and provides necessary information for control and motion planning.

1.3 The Role of SLAM in the Autonomy Architecture

The role of SLAM is to serve downstream tasks. For instance, the robot pose estimate can be used to control the robot to follow a desired trajectory, while the map (in combination with the current robot pose) can be used for motion planning (Figure 1.4). Here motion planning is used in a broad sense: while SLAM is typically used to build large-scale maps to support navigation tasks, it can also support building local 3D maps to enable manipulation and grasping.

While it would be tempting to think about SLAM as a monolithic system that takes sensor data in input and instantaneously outputs robot poses and map, the actual implementation of these systems and their integration in autonomy architectures is more complicated in practice. This is due to the fact that the robot needs to close different control and decision-making loops with different latency requirements. For instance, with reference to Figure 1.4, the robot will need to close low-level control loops over its trajectory (this is the standard feedback control loop at the top-right of the figure), which might require relatively high rates and low-latency to be stable; for instance, a UAV flying at high speed might need the front-end to produce odometry estimates with a latency of a few milliseconds. On the other hand, closing the loop over motion planning (the outer loop in Figure 1.4) can accommodate higher latencies, since global planning typically runs at lower rates; hence it might be acceptable for the back-end to provide global trajectory and map estimates with a latency of seconds. For these reasons, a typical implementation of a SLAM system involves multiple processes running in parallel and in a way that slower processes (*e.g.,* global pose and map optimization in the back-end) do not get in the way of faster processes (*e.g.,* odometry estimation). We

also observe that the processes involved in a SLAM system have complex interactions (as emphasized by the bi-directional edges in Figure 1.4): for instance, while the front-end feeds the odometry to the back-end, the back-end periodically applies global corrections to the odometric trajectory, which is then passed to the motion controller; similarly, while the front-end computes loop closures that are fed to the back-end, the back-end can also inform loop closure detection about plausible or implausible loop closure opportunities.

The problem of visual SLAM is closely related to the problem of Structure from Motion (SfM). While for some researchers both terms are equivalent, others distinguish and argue that visual SLAM systems will typically also integrate additional sensory information (IMUs, wheel odometry, etc) and focus on an online approach where data comes in sequentially whereas in SfM both online and offline versions are conceivable and the input is only images.

Overall, one can distinguish two complementary challenges: There is the *online challenge*, where a robot moves around, where sensory data streams in sequentially and while the SLAM back-end might run at a slower pace, vital estimates such as the localization of the robot must be determined in real-time, often even on embedded hardware with limited compute. These realtime constraints are vital for the robot to properly act in a complex environment, in particular with faster robots such as drones. They often dictate the choice of algorithms and processing steps.

And there is the *offline challenge* where the input data may not exhibit any sequential ordering (say an unordered dataset of images), where computations typically do not require real-time performance and where the compute hardware can be (arbitrarily) large (multiple powerful GPUs), see for example [8]. In such cases, accuracy of the estimated map and trajectories is more important than compute time.

In most applications, however, one will face a mix of these two extreme scenarios where certain quantities need to be determined fast whereas others can be determined offline. In practical applications of SLAM it is of utmost importance to carefully analyze which quantities need to be determined at which frequency and one may come up with an entire hierarchy of different temporal scales at which quantities are being estimated.

1.3.1 Do we really need SLAM for robotics?

From our description above, SLAM feels like an intriguing but very challenging problem, ranging from its complex implementation, to the need of fast runtime on resource-constrained platforms. Therefore, a fair question to ask is whether we can develop complex autonomous robots that do *not* rely on SLAM. We refine this question into three sub-questions.

Q1. Do we need SLAM for any robotics task? We started this section stating that SLAM is designed to support robotics tasks. Then a natural question is

whether it is necessary for *any* robotics task. The answer is clearly: no. More reactive tasks, for instance keeping a target in sight can be solved with simpler control strategies (*e.g.,* visual servoing).[2] Similarly, if the robot has to operate over small distances, relying on odometry estimates and local mapping might be acceptable. Moreover, if the environment the robot operates in has some infrastructure for localization, then we may not need to solve SLAM. Nevertheless, SLAM seems an indispensable component for long-term robot operation in unstructured (*i.e.,* infrastructure-free) environments: long-term operation typically requires *memory* (*e.g.,* to go back to previously seen objects or find suitable collision-free paths), and map representations built from SLAM provide such a long-term memory.

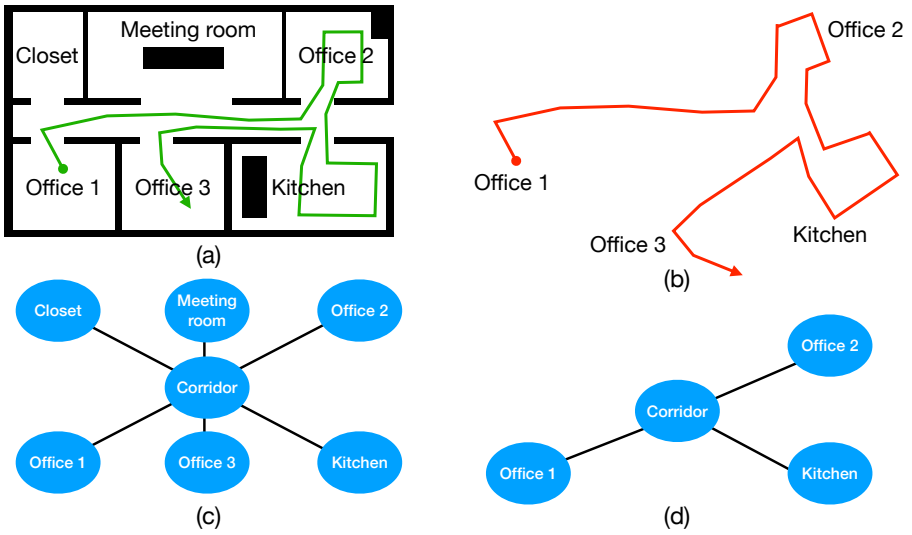


Figure 1.5 (a) Our robot visits Office 1 in a building and then —after exploring other areas (including Office 2 and the Kitchen)— it visits Office 3, which is just next door from Office 1. Obstacles are shown in black and ground truth trajectory is shown in green. (b) Odometric estimate of the trajectory, labeled with corresponding room labels. (c) Ground truth topological map of the environment. (d) Estimated topological map in the presence of perceptual aliasing, causing the robot to think that Office 1 and 3 are the same room.

Q2. Do we need globally consistent geometric maps for navigation? A major focus in SLAM is to optimize the trajectory and map representations such that they are metrically accurate (or *globally consistent*) – this is precisely the role of the SLAM back-end. One might ask whether metric accuracy is actually needed. One alternative that comes to mind is to just use odometry to get locally consistent trajectory and map estimates; this circumvents the need for loop closures and

[2] One could argue that while not being strictly necessary for tracking, SLAM and odometry might still be helpful to increase its robustness, *e.g.,* when the target gets out of sight.

back-end optimization. Unfortunately, due to its drift, odometry is unsuitable to support long-term operation: imagine that our robot visits Office 1 in a building and then, after exploring other areas of the building it visits Office 3, which is just next door from Office 1 (see Figure 1.5(a)). Using just odometry, the robot might be misled to conclude that Office 1 and Office 3 are quite far from each other (due to the odometry drift), hence being unable to realize there is a short path connecting the two offices (Figure 1.5(b)). A slightly more sophisticated alternative is to build a *topological map* instead. A topological map can be thought of as a graph where nodes are places the robot visited and edges represent traversability between the places connected by each edge (Figure 1.5(c)). The difference with the *metric* SLAM lens we adopt in this handbook is that nodes and edges in a topological map do not carry metric information (distances, bearing, positions), hence they do not require any optimization: one can simply add edges to a topological map when the robot traveled between two places (odometry) or when a place recognition module recognizes the places to overlap (loop closures). While this seems a perfectly reasonable approach, the main issue is that place recognition techniques are not perfect and, more fundamentally, two different places might look similar (a phenomenon known as *perceptual aliasing*). Therefore, going back to our example above, if Office 1 and Office 3 look very similar, a purely topological approach might be misled to think there is a single office instead (Figure 1.5(d)). On the other hand, metric SLAM approaches can use geometric information to conclude that the two offices are indeed two different rooms, by giving the user access to a more powerful set of tools to decide whether place recognition results are correct and if two observations correspond to the same place; we will discuss these tools at length in Chapter 4.

Q3. Do we need maps? SLAM builds a map that can be directly queried, inspected, and visualized. As we will see in Chapter 6, there are many ways to represent a map, including 3D point clouds, voxels, meshes, neural radiance fields, and others. On the other hand, one might take a completely different approach: in order for the robot to execute a task, the robot might be trained to translate raw sensor data directly to actions (*e.g.,* using Reinforcement Learning), hence circumventing the need to build a map. In such an approach, the neural network trained from sensor data to actions will arguably create an internal representation, but such an internal representation cannot be directly queried, inspected, or visualized. While the jury is still out on whether maps are indeed necessary, there is some initial evidence that using maps as an intermediate representation is at least beneficial in completing many visual tasks for robotics [218, 304]. Moreover, maps have the benefit of being useful across a wide variety of tasks, while a representation that is fully learned in the context of a single task might not be able to support new unseen tasks. Finally, we observe that there are several applications where the *goal* is to have a map that can be inspected. This is the case in search-and-rescue robotics applications where it is desirable to provide a map to help first-responders. Moreover, it is the case for several applications beyond robotics (*e.g.,* real-estate planning and

visualization, construction monitoring, virtual and augmented reality), where the goal is for a human to inspect or visualize the map.

1.4 Past, Present, and Future of SLAM, and Scope of this Handbook

The design of algorithms for spatial reasoning has been at the center-stage of robotics and computer vision research since their inception. At the same time, SLAM research keeps evolving and expanding to novel tools and problems.

1.4.1 Short History and Scope of this Handbook

As discussed across the various chapters of this book, SLAM has multiple facets. As a consequence, its history is also multi-faceted with origins that can be traced back across different scientific communities.

Creating maps of the world from observations and measurements is among the oldest challenges in history and leads to the fields of *geodesy* (the science measuring properties of the Earth) and *surveying*. There are many pioneers who contributed to this field. Carl Friedrich Gauss triangulated the Kingdom of Hannover in the years 1821-1825. Sir George Everest served as Surveyor General of India 1830-1843 in the Great Trigonometric Survey, efforts for which he was honored by having the world's largest mountain named after him. In 1856, Carl Maximilian von Bauernfeind published a standard book on "Elements of Surveying" [22]. He subsequently founded the Technical University of Munich in 1868 with a central focus on establishing geodesy as a scientific discipline. André-Louis Cholesky developed the well-known Cholesky matrix decomposition while surveying Crete and North Africa before the First World War.

The problem of visual SLAM is also closely related to the field of photogrammetry and the problems of Structure from Motion in computer vision. Its origins can be traced back to the 19th century. See Chapter 8.

In robotics, the origin of SLAM is typically traced back to the seminal work of Smith and Chessman [239] and Durrant-Whyte [79], as well as the parallel work by Crowley [60] and Chatila and Laumond [48]. The acronym SLAM was coined in 1995, as part of the survey paper [80]. These early works developed two fundamental insights. The first insight is that to avoid drift in unknown environments, one needs to simultaneously estimate the robot poses and the position of fixed external entities (*e.g.,* landmarks). The second insight is that existing tools from estimation theory, and in particular the celebrated Extended Kalman Filter (EKF), could be used to perform estimation over an extended state describing the robot poses and the landmark positions, leading to a family of *EKF-SLAM* approaches.

EKF-SLAM approaches have been extremely popular but face three main issues in practice. The first is that they are sensitive to outliers and data association errors. These errors may result from failures of place recognition or object detection,

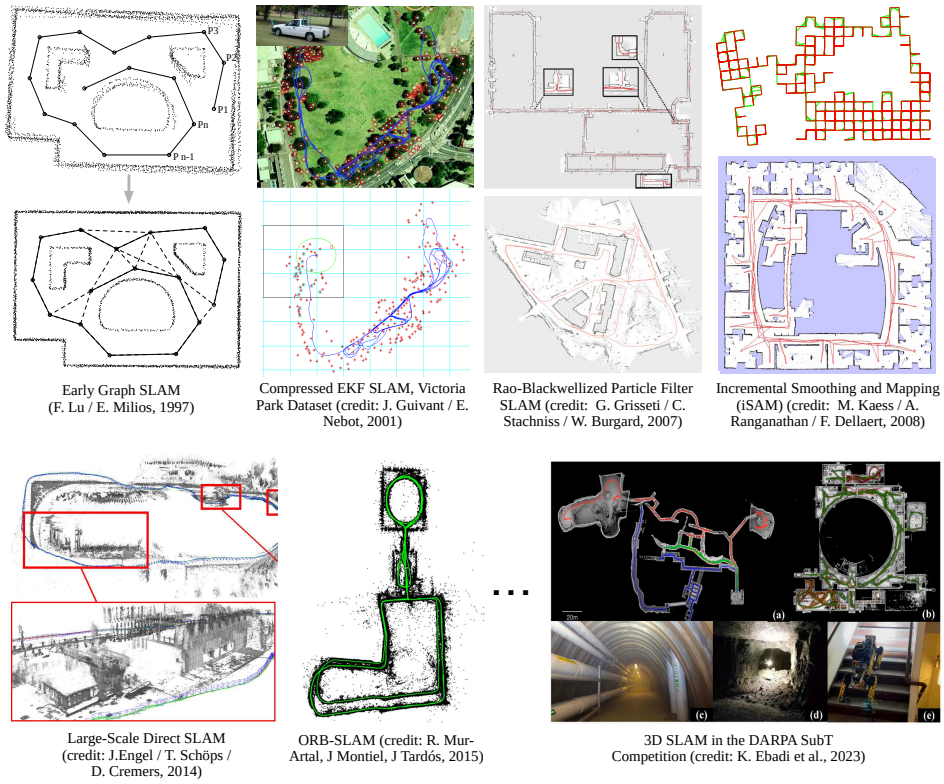


Figure 1.6 The history of SLAM is filled with numerous advances that have led to modern SLAM systems capable of localizing and mapping robots in challenging real-world environments. This image shows a selection of representative highlights.

where the robot believes it is observing a given object or place, but it is actually observing a different (but possibly similarly looking) one. If these spurious measurements are not properly handled, EKF-SLAM produces grossly incorrect estimates. The second issue is related to the fact that EKF relies on linearization of the equations describing the motion of the robot and sensor observations. In practice, the linearization point is typically built from odometry and when the latter drifts, the linearized system might be a poor approximation of the original nonlinear system. This leads EKF-SLAM to diverge when odometry accumulates substantial drift. The third problem is about computational complexity: a naive implementation of the Kalman Filter leads to a computational complexity that grows quadratically in the number of state variables, due to the need to manipulate a dense covariance matrix. In a landmark-based SLAM problem it is not uncommon to have thousands of landmark, which makes the naive approach prohibitive to run in real-time.

As a response to these issues, in the early 2000s, the community started focusing

on *particle-filter-based approaches* [178, 238, 103], which model the robot trajectory using a set of hypothesis (or *particles*), building on the theory of particle filtering in estimation theory.[3] When used in combination with landmark-based maps, these models allowed using a large number of landmarks (breaking through the quadratic complexity of the EKF); moreover, they allowed to more easily estimate dense map models, such as 2D occupancy grid maps. Also, these approaches did not rely on linearization and were less sensitive to outliers and incorrect data association. However, they still exhibited a trade-off between computation and accuracy: obtaining accurate trajectories and maps requires using many particles (in the thousands) but the more particles, the more computation. In particular, for a finite amount of particles, a particle filter may still diverge when none of the sampled particles are near the real trajectory of the robot (an issue known as *particle* depletion); this issue is exacerbated in 3D problems where one needs many particles to cover potential robot poses.

Between 2005 and 2015, a key insight pushed to the spotlight an alternative approach to SLAM. The insight is that while the covariance matrix appearing in the EKF is dense, its inverse (the so called *Information Matrix*) is very sparse and has a very predictable sparsity pattern when past robot poses are retained in the estimation [87]; this allows designing filtering algorithms that have close-to-linear complexity, as opposed to the quadratic complexity of the EKF. While this insight was initially applied to EKF-like approaches, such as EIF, it also paved the way for *optimization-based approaches*. Optimization-based approaches were first proposed in the early days of SLAM [162], but then disregarded as too slow to be practical. The sparsity structure mentioned above allowed rethinking these optimization methods and making them more scalable and solvable in online fashion [65, 126].[4] This new wave can be interpreted as a shift toward yet another estimation framework: *maximum likelihood* and *maximum a posteriori* estimation. These frameworks rephrase estimation problems in terms of optimization, while describing the structure of the problem in terms of a probabilistic graphical model, or, specifically, a *factor graph*. The resulting factor-graph-based approach to SLAM is still the dominant paradigm today, and has also shaped the way the community thinks about related problems, such as visual and visual-inertial odometry. The optimization lens is a powerful one and allows a much deeper theoretical analysis than previously possible (see Chapter 7). Moreover, it is fairly easy to show that the EKF (with suitable linearization points) can be understood as a single iteration of a nonlinear optimization solver, hence making the optimization lens strictly more powerful than its filtering-based counterpart. Finally, the optimization-based perspective appears more suitable for recent extensions of SLAM (described in the next section and

[3] The resulting algorithms are known with different names in different communities, *e.g.*, Sampling/Importance Re-sampling, Monte-Carlo filter Condensation algorithm, Survival of the fittest algorithm, and others.

[4] More details are in Chapter 2.

Part III of this handbook), where one wants to estimate both continuous variables (describing the scene geometry) and discrete variables (describing semantic aspects of the scene).

This short history review stops at 2015, while the goal of Part III of this handbook is to discuss more modern trends, including those triggered by the "deep learning revolution", which started around 2012 and slowly permeated to robotics. We also remark that the short history above mostly gravitates around what we called the SLAM back-end (essentially, the estimation engine), while the development of the SLAM front-end traces back to work done across multiple communities, including computer vision, signal processing, and machine learning.

As a result of the considerations mentioned above, this handbook will primarily focus on the factor-graph-based formulation of SLAM. This is a decision about scope and does not detract from the value of ongoing works using other technical tools. For instance, at the time of writing of this handbook, EKF-based tools are still popular for visual-inertial odometry applications (building on the seminal work from Mourikis and Roumeliotis [180]), and novel estimation formulations have been developed, including invariant [20] and equivariant filters [90], as well as alternative formulations based on random finite sets [181].

1.4.2 From SLAM to Spatial AI

SLAM essentially focuses on estimating geometric properties of the environment (and the robot). For instance, the SLAM map carries information about obstacles in the environment, distances and traversable paths between two locations, or geometric coordinates of distinctive landmarks. In this sense, SLAM is useful as a representation for the robot to understand and execute commands such as "robot: go to position $[x, y, z]$", where $[x, y, z]$ are the coordinates (in the map frame) of a place or object the robot has to reach. However, specifying goals in terms of coordinates is not suitable for non-expert human users and it is definitely not the way we interact or specify goals for humans. Therefore, it would be desirable for the next generation of robots to understand and execute high-level commands specified in natural language, such as "robot: pick up the clothes in the bathroom, and take them to the laundry room". Parsing these instructions requires the robot to understand both geometry (*e.g.,* where is the bathroom) and semantics (*e.g.,* what is a bathroom or laundry room, which objects are clothes) of the environment.

This realization has recently pushed the research community to think about SLAM as an integrated component of a broader *spatial perception* system, that simultaneously reasons about geometric, semantic, and possibly physical aspects of the scene, in order to build a multi-faceted map representation (a "world model"), that enables the robot to understand and execute complex instructions. The resulting *Spatial AI* algorithms and systems have the potential to increase robot autonomy and have rapidly progressed over the last decade. Intuitively, one can

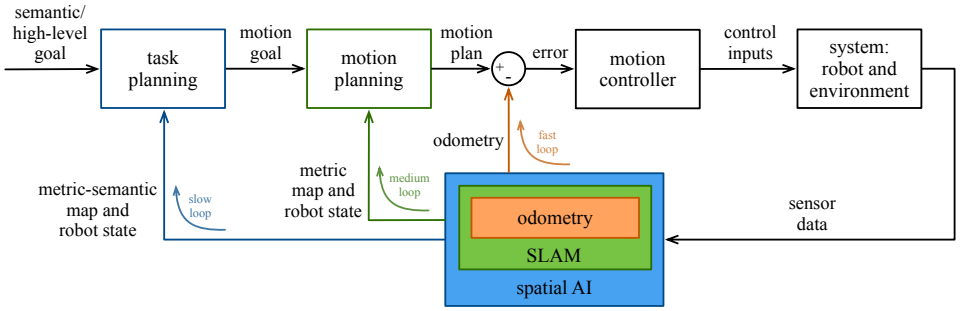


Figure 1.7 Spatial AI (or spatial perception) extends the geometric reasoning capabilities of SLAM to also perform semantic and physical reasoning. While the SLAM block is informed by odometry and provides a geometric understanding of the scene, the Spatial AI block is informed by the SLAM results and adds a scene understanding component, spanning semantics, affordances, dynamics, and more. This allows closing the loop over higher-level decision making modules, such as task planning, and allows the user to specify higher-level goals the robot has to achieve.

think that Spatial AI has SLAM as a submodule (to handle the geometric reasoning part), but provides extra semantic reasoning capabilities. This allows closing the loop over task planning, as shown in Figure 1.7, where now the robot can take high-level semantic goals instead of coordinates of motion goals. We will discuss Spatial AI at length in Part III of this handbook.

1.5 Handbook Structure

The chapters of this handbook are grouped into three parts.

Part I covers the foundations of SLAM, with particular focus on the estimation-theoretic machinery used in the SLAM back-end and the different types of map representations SLAM can produce. In particular, Chapter 2 introduces the factor-graph formulation of SLAM and reviews how to solve it via iterative nonlinear optimization methods. Then, Chapter 3 takes the indispensable step of extending the formulation to allow the estimation of variables belonging to smooth manifolds, such as rotation and poses. Chapter 4 discusses how to model and mitigate the impact of outliers and incorrect data association in the SLAM back-end. Chapter 5 reviews techniques to make the back-end optimization differentiable, a key step towards interfacing traditional SLAM methods with more recent deep learning architectures. Chapter 6 shifts the focus from the back-end to the question of dense map representations and discusses the most important representations used for SLAM. Finally, Chapter 7 discusses more advanced solvers and theoretical properties of the SLAM back-end.

Part II covers the "state of practice" in SLAM by discussing key approaches and

applications of SLAM using different sensing modalities. This part touches on the SLAM front-end design (which is heavily sensor dependent) and exposes what's feasible with modern SLAM algorithms and systems. Chapter 8 reviews the large body of literature on visual SLAM. Chapter 9 and Chapter 10 cover lidar-SLAM and radar-SLAM, respectively. Chapter 11 discusses recent work on SLAM using event-based cameras. Chapter 12 reviews how to model inertial measurements as part of a factor-graph SLAM system and discusses fundamental limits (*e.g.,* observability). Chapter 13 discussed how to model other sources of odometry information, including wheel and legged odometry.

Part III provides a future-looking view of the state of the art and recent trends. In particular, we touch on a variety of topics, ranging from computational architectures, to novel problems and representations, to the role of language and Foundation Models in SLAM. In particular, Chapter 14 focuses on future computational architectures for Spatial AI that could leverage more flexible and distributed computing hardware and better support spatial perception across many robotic platforms. Chapter 15 reviews recent improvements obtained by introducing deep learning modules in conjunction with differentiable optimization in SLAM. Chapter 16 discusses opportunities and challenges in using novel map presentations, including neural radiance fields (NeRFs) and Gaussian Splatting. Chapter 17 covers recent work on SLAM in highly dynamic and deformable environments, touching on real applications from mapping in crowded environments to surgical robotics. Chapter 18 discusses progress in Spatial AI and metric-semantic map representations. Finally, Chapter 19 considers new opportunities arising from the use of Foundation Models (*e.g.,* Large Vision-Language Models) and their role in creating novel map representation for Spatial AI that allow understanding and grounding "open-vocabulary" commands given in natural language.

2

Factor Graphs for SLAM

Frank Dellaert, Michael Kaess, Timothy Barfoot

In this chapter we introduce factor graphs and establish the connection with maximum a posteriori (MAP) inference and least-squares for the case of Gaussian priors and Gaussian measurement noise. We focus on the SLAM back-end, after measurements have been extracted by the front-end and data association has been accomplished. We discuss both linear and nonlinear optimization methods for the corresponding least-squares problems, and then make the connection between sparsity, factor graphs, and Bayes nets more explicit. Finally, we apply this to develop the Bayes tree and the incremental smoothing and mapping (iSAM) algorithm.

Historical Note

A *smoothing* approach to SLAM involves not just the most current robot location, but the entire robot trajectory up to the current time. A number of authors consider the problem of smoothing the robot trajectory only [48, 160, 161, 107, 140, 86], now known as *PoseSLAM*. This is particularly suited to sensors such as laser-range finders that yield pairwise constraints between nearby robot poses.

More generally, one can consider the *full SLAM problem* [259], i.e., the problem of optimally estimating the entire set of sensor poses along with the parameters of all features in the environment. This led to a flurry of work between 2000 and 2005 where these ideas were applied in the context of SLAM [76, 93, 92, 259]. From a computational view, this optimization-based smoothing was recognized as beneficial since (a) in contrast to the filtering-based covariance or information matrices, which *both* become fully dense over time [201, 258], the information matrix associated with smoothing is and stays sparse; (b) in typical mapping scenarios (i.e., not repeatedly traversing a small environment) this matrix is a much more compact representation of the map covariance structure.

Square-root smoothing and mapping (SAM), also known as the 'factor-graph approach', was introduced in [65, 68] based on the fact that the information matrix or measurement Jacobian can be efficiently factorized using sparse Cholesky or QR factorization, respectively. This yields a square-root information matrix that can be used to immediately obtain the optimal robot trajectory and map. Factoring the

information matrix is known in the sequential estimation literature as *square-root information filtering (SRIF)*, and was developed in 1969 for use in JPL's Mariner 10 missions to Venus [30]. The use of square roots results in more accurate and stable algorithms, and, quoting Maybeck [174], "a number of practitioners have argued, with considerable logic, that square root filters should *always* be adopted in preference to the standard Kalman filter recursion".

Below we discuss in detail how factor graphs are a natural representation for the sparsity inherent in SLAM problems, how (sparse) matrix factorization into a matrix-square root is at the heart of solving these problems, and finally how all this relates to the much more general *variable elimination algorithm*. Much of this chapter is an abridged version of a longer article by Dellaert et al. [69].

2.1 Visualizing SLAM With Factor Graphs

In this section we introduce factor graphs as a way of intuitively visualizing the sparse nature of the SLAM problem by first considering a toy example and its factor graph representation. We then show how many different flavors of SLAM can be represented as such, and how even in larger problems the sparse nature of many sparse problems is immediately apparent.

2.1.1 A Toy Example

We begin by examining a simple SLAM scenario to illustrate how factor graphs are constructed. Figure 2.1 shows a simple toy example illustrating the structure of the problem graphically. A robot moving across three successive poses $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, and $\boldsymbol{p}_3$ makes bearing observations on two landmarks $\boldsymbol{\ell}_1$ and $\boldsymbol{\ell}_2$. To anchor the solution in space, let us also assume there is an absolute position/orientation measurement on the first pose $\boldsymbol{p}_1$. Without this there would be no information about absolute position, as bearing measurements are all relative.[1]

Because of measurement uncertainty, we cannot hope to recover the true state of the world, but we can obtain a probabilistic description of what can be inferred from the measurements. In the Bayesian probability framework, we use the language of probability theory to assign a subjective degree of belief to uncertain events. We do this using *probability density functions (PDFs)* $p(\boldsymbol{x})$ over the unknown variables $\boldsymbol{x}$. PDFs are non-negative functions satisfying

$$\int p(\boldsymbol{x})\, d\boldsymbol{x} = 1, \tag{2.1}$$

which is the axiom of total probability. In the simple example of Figure 2.1, the

[1] Handling rotations properly is a bit more involved than our treatment in this first chapter lets on. However, the next chapter will put us on a proper footing for such quantities. For now, we will assume they are regular vector quantities and delay discussion of their subtleties.

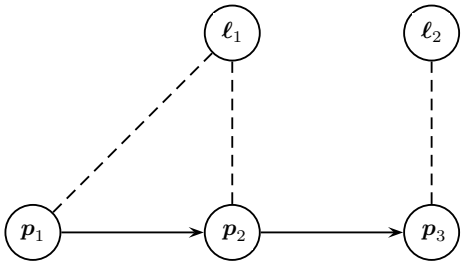


Figure 2.1 A toy simultaneous localization and mapping (SLAM) example with three robot poses and two landmarks. Above we schematically indicate the robot motion with arrows, while the dotted lines indicate bearing measurements.

state, $\boldsymbol{x}$, is

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{p}_1 \\ \boldsymbol{p}_2 \\ \boldsymbol{p}_3 \\ \boldsymbol{\ell}_1 \\ \boldsymbol{\ell}_2 \end{bmatrix}, \tag{2.2}$$

which is just a stacking of the individual unknowns.

In SLAM we want to characterize our knowledge about the unknowns $\boldsymbol{x}$, in this case robot poses and the unknown landmark positions, when given a set of *observed* measurements $\boldsymbol{z}$. Using the language of Bayesian probability, this is simply the conditional density

$$p(\boldsymbol{x}|\boldsymbol{z}), \tag{2.3}$$

and obtaining a description like this is called *probabilistic inference*. A prerequisite is to first specify a probabilistic model for the variables of interest and how they give rise to (uncertain) measurements. This is where *probabilistic graphical models* enter the picture.

Probabilistic graphical models provide a mechanism to compactly describe complex probability densities by exploiting the structure in them [139]. In particular, high-dimensional probability densities can often be factorized as a product of many *factors*, each of which is a probability density over a much smaller domain.

2.1.2 A Factor-Graph View

Factor graphs are probabilistic graphical models and they allow us to specify a joint density as a product of factors. However, they are more general in that they can be used to specify *any* factored function $\phi(\boldsymbol{x})$ over a set of variables $\boldsymbol{x}$, not just probability densities.

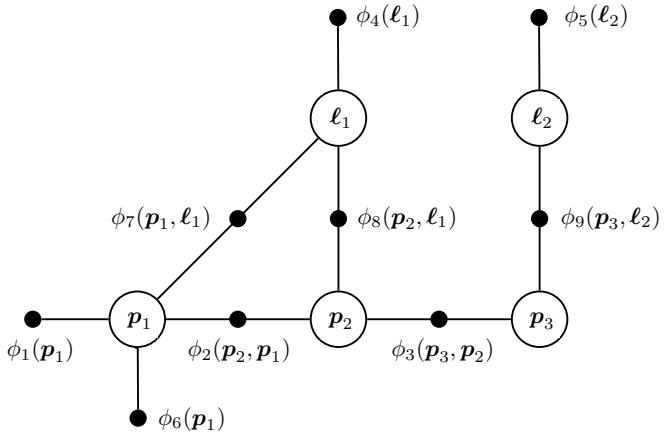


Figure 2.2 Factor graph resulting from the example in Figure 2.1.

To motivate this, consider performing inference for the toy SLAM example. The posterior $p(\boldsymbol{x}|\boldsymbol{z})$ can be re-written using Bayes' law, $p(\boldsymbol{x}|\boldsymbol{z}) \propto p(\boldsymbol{z}|\boldsymbol{x})p(\boldsymbol{x})$, as

$$
\begin{aligned}
p(\boldsymbol{x}|\boldsymbol{z}) \propto\ & p(\boldsymbol{p}_1)\, p(\boldsymbol{p}_2|\boldsymbol{p}_1)\, p(\boldsymbol{p}_3|\boldsymbol{p}_2) && \text{(2.4a)}\\
& \times p(\boldsymbol{\ell}_1)\, p(\boldsymbol{\ell}_2) && \text{(2.4b)}\\
& \times p(\boldsymbol{z}_1|\boldsymbol{p}_1) && \text{(2.4c)}\\
& \times p(\boldsymbol{z}_2|\boldsymbol{p}_1, \boldsymbol{\ell}_1)\, p(\boldsymbol{z}_3|\boldsymbol{p}_2, \boldsymbol{\ell}_1)\, p(\boldsymbol{z}_4|\boldsymbol{p}_3, \boldsymbol{\ell}_2). && \text{(2.4d)}
\end{aligned}
$$

where we assumed a typical Markov chain generative model for the pose trajectory. Each of the factors represents one piece of information about the unknowns, $\boldsymbol{x}$.

To visualize this factorization, we use a *factor graph*. Figure 2.2 introduces the corresponding factor graph by example: all unknown states $\boldsymbol{x}$, both poses and landmarks, have a node associated with them. Measurements are *not* represented explicitly as they are known, and hence not of interest. In factor graphs we explicitly introduce an additional node type to represent every *factor* in the posterior $p(\boldsymbol{x}|\boldsymbol{z})$. In the figure, each small black node represents a factor, and—importantly—is connected to only those state variables of which it is a function. For example, the factor $\phi_9(\boldsymbol{p}_3, \boldsymbol{\ell}_2)$ is connected only to the variable nodes $\boldsymbol{p}_3$ and $\boldsymbol{\ell}_2$. In more detail, we have

$$
\begin{aligned}
\phi(\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3, \boldsymbol{\ell}_1, \boldsymbol{\ell}_2) = \ & \phi_1(\boldsymbol{p}_1)\, \phi_2(\boldsymbol{p}_2, \boldsymbol{p}_1)\, \phi_3(\boldsymbol{p}_3, \boldsymbol{p}_2) && \text{(2.5a)}\\
& \times \phi_4(\boldsymbol{\ell}_1)\, \phi_5(\boldsymbol{\ell}_2) && \text{(2.5b)}\\
& \times \phi_6(\boldsymbol{p}_1) && \text{(2.5c)}\\
& \times \phi_7(\boldsymbol{p}_1, \boldsymbol{\ell}_1)\, \phi_8(\boldsymbol{p}_2, \boldsymbol{\ell}_1)\, \phi_9(\boldsymbol{p}_3, \boldsymbol{\ell}_2), && \text{(2.5d)}
\end{aligned}
$$

where the correspondence between the factors and the original probability densities in (2.4a)-(2.4d) should be obvious.

The factor values need only be *proportional* to the corresponding probability densities: any normalization constants that do not depend on the state variables may be omitted without consequence. Also, in this example, all factors above came either from a prior, e.g., $\phi_1(\boldsymbol{p}_1) \propto p(\boldsymbol{p}_1)$ or from a measurement, e.g., $\phi_9(\boldsymbol{p}_3, \boldsymbol{\ell}_2) \propto p(\boldsymbol{z}_4|\boldsymbol{p}_3, \boldsymbol{\ell}_2)$. Although the measurement variables $\boldsymbol{z}_1..\boldsymbol{z}_4$ are not explicitly shown in the factor graph, those factors are implicitly conditioned on them. Sometimes, when it helps to make this more explicit, factors can be written as (for example) $\phi_9(\boldsymbol{p}_3, \boldsymbol{\ell}_2; \boldsymbol{z}_4)$ or even $\phi_{\boldsymbol{z}_4}(\boldsymbol{p}_3, \boldsymbol{\ell}_2)$.

2.1.3 Factor Graphs as a Language

In addition to providing a formal basis for inference, factor graphs help visualize SLAM problems of many different flavors, give insight into the structure of the problem, and serve as a *lingua franca* that can help practitioners align across team boundaries. Each factor in a factor graph, such as those in Fig 2.2, can be thought of as an equation involving the variables it is connected to. There are typically many more equations than unknowns, which is why we need to quantify the uncertainty in both prior information and measurements. This will lead to a least-squares formulation, appropriately fusing the information from multiple sources.

Many different flavors of the SLAM problem are all easily represented as factor graphs. Figure 2.1 is an example of *landmark-based SLAM* because it involves both pose and landmark variables. Figure 2.3 illustrates several other variants including *bundle adjustment (BA)* (same as landmark-based SLAM but without motion model), *pose-graph optimization (PGO)* (no landmark variables but includes loop closures), and *simultaneous trajectory estimation and mapping (STEAM)* (poses are augmented to include derivatives such as velocity).

The factor graph for a more realistic landmark-based SLAM problem than the toy example could look something like Figure 2.4. This graph was created by simulating a 2D robot, moving in the plane for about 100 time steps, as it observes landmarks. For visualization purposes, each robot pose and landmark is rendered at its ground-truth position in 2D. With this, we see that the odometry factors form a prominent, chain-like backbone, whereas off to the sides binary likelihood factors are connected to the 20 or so landmarks. All factors in such SLAM problems are typically nonlinear, except for priors.

Examining the factor graph reveals a great deal of structure by which we can gain insight into a particular instance of the SLAM problem. First, there are landmarks with a great deal of measurements, which we expect to be pinned down very well. Others have only a tenuous connection to the graph, and hence we expect them to be less well determined. For example, the lone landmark near the bottom right has only a single measurement associated with it: if this is a bearing-only measurement,

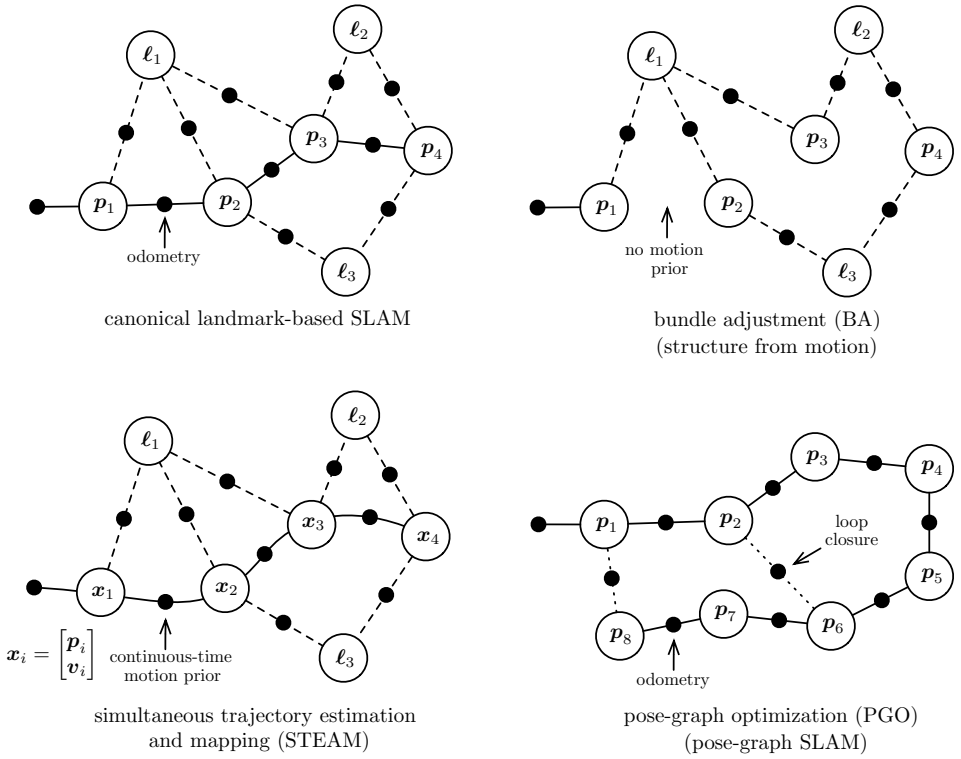


Figure 2.3 A few variants of SLAM problems that can all be viewed through the factor-graph lens. Canonical landmark-based SLAM has both pose and landmark variables; landmarks are measured from poses and there is some motion prior between poses typically based on odometry. BA is the same but without the motion prior. STEAM is similar but now poses can be replaced by higher-order states and a smooth continuous-time motion prior is used. PGO does not have landmark variables but enjoys extra loop-closure measurements between poses.

many assignments of a 2D location to the landmark will be equally 'correct'. This is the same as saying that we have infinite uncertainty in some subset of the domain of the unknowns, which is where prior knowledge should come to the rescue.

2.2 From MAP Inference to Least Squares

In SLAM, *maximum a posteriori (MAP)* inference is the process of determining the values for the unknowns $\boldsymbol{x}$ that maximally agree with the information present in the uncertain measurements. In real life we are *not* given the ground-truth locations for the landmarks, nor the time-varying pose of the robot, although in many practical cases we might have a good initial estimate. Below we review how to model both prior knowledge and measurements using probability densities, how the posterior

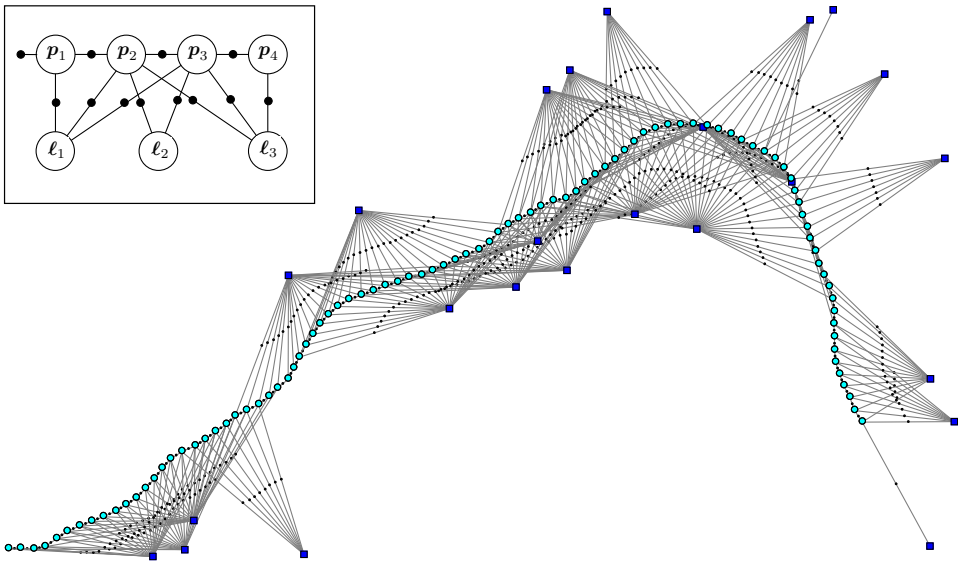


Figure 2.4 Factor graph for a larger, simulated SLAM example.

density given measurements is most conveniently represented as a factor graph, and how given Gaussian priors and Gaussian noise models the corresponding optimization problem is nothing but the familiar nonlinear least-squares problem.

2.2.1 Factor Graphs for MAP Inference

We are interested in the *unknown state variables* $\boldsymbol{x}$, such as poses and/or landmarks, *given* the measurements $\boldsymbol{z}$. The most-often-used *estimator* for these unknown state variables $\boldsymbol{x}$ is the *maximum a posteriori (MAP)* estimate, so named because it maximizes the posterior density $p(\boldsymbol{x}|\boldsymbol{z})$ of the states $\boldsymbol{x}$ given the measurements $\boldsymbol{z}$:

$$\boldsymbol{x}^{\text{MAP}} = \arg\max_{\boldsymbol{x}} p(\boldsymbol{x}|\boldsymbol{z}) \tag{2.6a}$$

$$= \arg\max_{\boldsymbol{x}} \frac{p(\boldsymbol{z}|\boldsymbol{x})p(\boldsymbol{x})}{p(\boldsymbol{z})} \tag{2.6b}$$

$$= \arg\max_{\boldsymbol{x}} p(\boldsymbol{z}|\boldsymbol{x})p(\boldsymbol{x}) \tag{2.6c}$$

The second equation above is Bayes' law, and expresses the posterior as the product of the measurement density $p(\boldsymbol{z}|\boldsymbol{x})$ and the prior $p(\boldsymbol{x})$ over the states, appropriately normalized by the factor $p(\boldsymbol{z})$. The third equation drops the $p(\boldsymbol{z})$ since this does not depend on the $\boldsymbol{x}$ and therefore will not impact the $\arg\max$ operation.

We use factor graphs to express the unnormalized posterior $p(\boldsymbol{z}|\boldsymbol{x})p(\boldsymbol{x})$. Formally a factor graph is a bipartite graph $F = (\mathcal{U}, \mathcal{V}, \mathcal{E})$ with two types of nodes: *factors*

$\phi_i \in \mathcal{U}$ and *variables* $\boldsymbol{x}_j \in \mathcal{V}$. Edges $e_{ij} \in \mathcal{E}$ are always between factor nodes and variables nodes. The set of variable nodes adjacent to a factor ϕ_i is written as $\mathcal{X}(\phi_i)$, and we write $\boldsymbol{x}_i$ for an assignment to this set. With these definitions, a factor graph F defines the factorization of a global function $\phi(\boldsymbol{x})$ as

$$\phi(\boldsymbol{x}) = \prod_i \phi_i(\boldsymbol{x}_i). \tag{2.7}$$

In other words, the independence relationships are encoded by the edges e_{ij} of the factor graph, with each factor ϕ_i a function of *only* the variables $\boldsymbol{x}_i$ in its adjacency set $\mathcal{X}(\phi_i)$.

In the rest of this chapter, we show how to find an optimal assignment, the MAP estimate, through optimization over the unknown variables in the factor graph. Indeed, for an arbitrary factor graph, MAP inference comes down to maximizing the product (2.7) of all factor-graph potentials:

$$\boldsymbol{x}^{\text{MAP}} = \arg\max_{\boldsymbol{x}} \phi(\boldsymbol{x}) \tag{2.8a}$$

$$= \arg\max_{\boldsymbol{x}} \prod_i \phi_i(\boldsymbol{x}_i). \tag{2.8b}$$

What is left now is to derive the exact form of the factors $\phi_i(\boldsymbol{x}_i)$, which depends very much on how we model the measurement models $p(\boldsymbol{z}|\boldsymbol{x})$ and the prior densities $p(\boldsymbol{x})$. We discuss this in detail next.

2.2.2 Specifying Probability Densities

The exact form of the densities $p(\boldsymbol{z}|\boldsymbol{x})$ and $p(\boldsymbol{x})$ above depends very much on the application and the sensors used. The most often used densities involve the *multivariate Gaussian density*, with probability density

$$\mathcal{N}(\boldsymbol{\theta}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{|2\pi\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2} \|\boldsymbol{\theta} - \boldsymbol{\mu}\|^2_{\boldsymbol{\Sigma}}\right), \tag{2.9}$$

where $\boldsymbol{\mu} \in \mathbb{R}^n$ is the mean, $\boldsymbol{\Sigma}$ is an $n \times n$ covariance matrix, and

$$\|\boldsymbol{\theta} - \boldsymbol{\mu}\|^2_{\boldsymbol{\Sigma}} \triangleq (\boldsymbol{\theta} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\boldsymbol{\theta} - \boldsymbol{\mu}) \tag{2.10}$$

denotes the squared Mahalanobis distance. The normalization constant $\sqrt{|2\pi\boldsymbol{\Sigma}|} = (2\pi)^{n/2} |\boldsymbol{\Sigma}|^{1/2}$, where $|.|$ denotes the matrix determinant, ensures the multivariate Gaussian density integrates to 1.0 over its domain.

Priors on unknown quantities are often specified using a Gaussian density, and in many cases it is both justified and convenient to model measurements as corrupted by zero-mean Gaussian noise. For example, a bearing measurement[2] from a given

[2] As a reminder, there are some subtleties associated with rotational state variables that we will discuss more thoroughly in the next chapter.

pose $\boldsymbol{p}$ to a given landmark $\boldsymbol{\ell}$ would be modeled as

$$\boldsymbol{z} = \boldsymbol{h}(\boldsymbol{p}, \boldsymbol{\ell}) + \boldsymbol{\eta}, \tag{2.11}$$

where $\boldsymbol{h}(\cdot)$ is a *measurement prediction function*, and the noise $\boldsymbol{\eta}$ is drawn from a zero-mean Gaussian density with measurement covariance $\boldsymbol{\Sigma_R}$. This yields the following conditional density $p(\boldsymbol{z}|\boldsymbol{p}, \boldsymbol{\ell})$ on the measurement $\boldsymbol{z}$:

$$p(\boldsymbol{z}|\boldsymbol{p}, \boldsymbol{\ell}) = \mathcal{N}(\boldsymbol{z}; \boldsymbol{h}(\boldsymbol{p}, \boldsymbol{\ell}), \boldsymbol{\Sigma_R}) = \frac{1}{\sqrt{|2\pi\boldsymbol{\Sigma_R}|}} \exp\left(-\frac{1}{2} \|\boldsymbol{z} - \boldsymbol{h}(\boldsymbol{p}, \boldsymbol{\ell})\|^2_{\boldsymbol{\Sigma_R}}\right). \tag{2.12}$$

The measurement functions $\boldsymbol{h}(\cdot)$ are often nonlinear in practical robotics applications. Still, while they depend on the sensor used and the SLAM front-end, they are typically not difficult to reason about or write down. The measurement function for a 2D bearing measurement is simply

$$\boldsymbol{h}(\boldsymbol{p}, \boldsymbol{\ell}) = \text{atan2}(\ell_y - p_y, \ell_x - p_x), \tag{2.13}$$

where atan2 is the well-known two-argument arctangent variant. Hence, the final *probabilistic measurement model* $p(\boldsymbol{z}|\boldsymbol{p}, \boldsymbol{\ell})$ is obtained as

$$p(\boldsymbol{z}|\boldsymbol{p}, \boldsymbol{\ell}) = \frac{1}{\sqrt{|2\pi\boldsymbol{\Sigma_R}|}} \exp\left(-\frac{1}{2} \|\boldsymbol{z} - \text{atan2}(\ell_y - p_y, \ell_x - p_x)\|^2_{\boldsymbol{\Sigma_R}}\right). \tag{2.14}$$

Note that we will not *always* assume Gaussian measurement noise: to cope with the occasional data association mistake, for example, many authors have proposed the use of robust measurement densities, with heavier tails than a Gaussian density; these are discussed in Chapter 4.

Not all probability densities involved are derived from measurements. For example, in the toy SLAM problem the prior $p(\boldsymbol{x})$ on the trajectory is made up of a prior $p(\boldsymbol{p}_1)$ and conditional densities $p(\boldsymbol{p}_{t+1}|\boldsymbol{p}_t)$, specifying a *probabilistic motion model* that the robot is assumed to obey given known control inputs $\boldsymbol{u}_t$. In practice, we often use a conditional Gaussian assumption,

$$p(\boldsymbol{p}_{t+1}|\boldsymbol{p}_t, \boldsymbol{u}_t) = \frac{1}{\sqrt{|2\pi\boldsymbol{\Sigma_Q}|}} \exp\left(-\frac{1}{2} \|\boldsymbol{p}_{t+1} - \boldsymbol{g}(\boldsymbol{p}_t, \boldsymbol{u}_t)\|^2_{\boldsymbol{\Sigma_Q}}\right), \tag{2.15}$$

where $\boldsymbol{g}(\cdot)$ is a motion model, and $\boldsymbol{\Sigma_Q}$ a covariance matrix of the appropriate dimensionality, e.g., 3×3 in the case of robots operating in the plane.

Often we have no known control inputs $\boldsymbol{u}_t$ but instead we *measure* how the robot moved, e.g., via an odometry measurement $\boldsymbol{o}_t$. For example, if we assume the odometry simply measures the difference between poses, subject to Gaussian noise with covariance $\boldsymbol{\Sigma_S}$, we obtain

$$p(\boldsymbol{o}_t|\boldsymbol{p}_{t+1}, \boldsymbol{p}_t) = \frac{1}{\sqrt{|2\pi\boldsymbol{\Sigma_S}|}} \exp\left(-\frac{1}{2} \|\boldsymbol{o}_t - (\boldsymbol{p}_{t+1} - \boldsymbol{p}_t)\|^2_{\boldsymbol{\Sigma_S}}\right). \tag{2.16}$$

If we have *both* known control inputs $\boldsymbol{u}_t$ and odometry measurements $\boldsymbol{o}_t$ we can combine (2.15) and (2.16).

Note that for robots operating in three-dimensional space, we will need slightly more sophisticated machinery to specify densities on nonlinear manifolds such as SE(3), as discussed in the next chapter.

2.2.3 Nonlinear Least Squares

We now show that MAP inference for SLAM problems with Gaussian noise models as above is equivalent to solving a nonlinear least-squares problem. If we assume that all factors are of the form

$$\phi_i(\boldsymbol{x}_i) \propto \exp\left(-\frac{1}{2}\left\|\boldsymbol{z}_i - \boldsymbol{h}_i(\boldsymbol{x}_i)\right\|^2_{\boldsymbol{\Sigma}_i}\right), \tag{2.17}$$

which include both simple Gaussian priors and likelihood factors derived from measurements corrupted by zero-mean, normally distributed noise. Taking the negative log of (2.8b) and dropping the factor $\frac{1}{2}$ allows us to instead minimize a sum of *nonlinear least-squares* terms:

$$\boldsymbol{x}^{\mathrm{MAP}} = \arg\min_{\boldsymbol{x}} \sum_i \left\|\boldsymbol{z}_i - \boldsymbol{h}_i(\boldsymbol{x}_i)\right\|^2_{\boldsymbol{\Sigma}_i}. \tag{2.18}$$

Minimizing this objective function performs sensor fusion through the process of combining several measurement-derived factors, and possibly several priors, to uniquely determine the MAP solution for the unknowns.

An important and non-obvious observation is that the factors in (2.18) typically represent rather *under-specified* densities on the involved unknown variables $\boldsymbol{x}_i$. Indeed, except for simple prior factors, the measurements $\boldsymbol{z}_i$ are typically of lower dimension than the unknowns $\boldsymbol{x}_i$. In those cases, the factor by itself accords the same likelihood to an infinite subset of the domain of $\boldsymbol{x}_i$. For example, a 2D measurement in a camera image is consistent with an entire ray of 3D points that project to the same image location.

Even though the functions $\boldsymbol{h}_i$ are nonlinear, *if* we have a decent initial guess available, then the nonlinear optimization methods we discuss in this chapter will be able to converge to the global minimum of (2.18). We should caution, however, that as our objective in (2.18) is *non-convex*, there is no guarantee that we will not get stuck in a local minimum if our initial guess is poor. This has led to so-called *certifiably optimal* solvers, which are the subject of a later chapter. Below, however, we focus on local methods rather than global solvers. We start off below by considering the the easier problem of solving a *linearized* version of the problem.

2.3 Solving Linear Least Squares

Before tackling the more difficult problem of nonlinear least squares, in this section we first show to *linearize* the problem, show how this leads to a *linear* least squares

problem, and review matrix factorization as computationally efficient way to solve the corresponding *normal equations*. A seminal reference for these methods is the book by Golub and Loan [101].

2.3.1 Linearization

We can linearize all measurement functions $\boldsymbol{h}_i(\cdot)$ in the nonlinear least-squares objective function (2.18) using a simple Taylor expansion,

$$\boldsymbol{h}_i(\boldsymbol{x}_i) = \boldsymbol{h}_i(\boldsymbol{x}_i^0 + \boldsymbol{\delta}_i) \approx \boldsymbol{h}_i(\boldsymbol{x}_i^0) + \boldsymbol{H}_i\boldsymbol{\delta}_i, \tag{2.19}$$

where the *measurement Jacobian* $\boldsymbol{H}_i$ is defined as the (multivariate) partial derivative of $\boldsymbol{h}_i(\cdot)$ at a given linearization point $\boldsymbol{x}_i^0$,

$$\boldsymbol{H}_i \triangleq \left.\frac{\partial \boldsymbol{h}_i(\boldsymbol{x}_i)}{\partial \boldsymbol{x}_i}\right|_{\boldsymbol{x}_i^0}, \tag{2.20}$$

and $\boldsymbol{\delta}_i \triangleq \boldsymbol{x}_i - \boldsymbol{x}_i^0$ is the *state update vector*. Note that we make an assumption that $\boldsymbol{x}_i$ lives in a vector space or, equivalently, can be represented by a *vector*. This is not always the case, e.g., when some of the unknown states in $\boldsymbol{x}$ represent 3D rotations or other more complex manifold types. We will revisit this issue in Chapter 3.

Substituting the Taylor expansion (2.19) into the nonlinear least-squares expression (2.18) we obtain a *linear* least-squares problem in the state update vector $\boldsymbol{\delta}$,

$$\boldsymbol{\delta}^* = \arg\min_{\boldsymbol{\delta}} \sum_i \left\|\boldsymbol{z}_i - \boldsymbol{h}_i(\boldsymbol{x}_i^0) - \boldsymbol{H}_i\boldsymbol{\delta}_i\right\|_{\boldsymbol{\Sigma}_i}^2 \tag{2.21a}$$

$$= \arg\min_{\boldsymbol{\delta}} \sum_i \left\|\left(\boldsymbol{z}_i - \boldsymbol{h}_i(\boldsymbol{x}_i^0)\right) - \boldsymbol{H}_i\boldsymbol{\delta}_i\right\|_{\boldsymbol{\Sigma}_i}^2, \tag{2.21b}$$

where $\boldsymbol{z}_i - \boldsymbol{h}_i(\boldsymbol{x}_i^0)$ is the *prediction error* at the linearization point, *i.e.,* the difference between actual and predicted measurement. Above, $\boldsymbol{\delta}^*$ denotes the solution to the locally linearized problem.

By a simple change of variables we can drop the covariance matrices $\boldsymbol{\Sigma}_i$ from this point forward: defining $\boldsymbol{\Sigma}^{1/2}$ as the matrix square root of $\boldsymbol{\Sigma}$, we can rewrite the square Mahalanobis norm as follows:

$$\|\boldsymbol{e}\|_{\boldsymbol{\Sigma}}^2 \triangleq \boldsymbol{e}^\top\boldsymbol{\Sigma}^{-1}\boldsymbol{e} = \left(\boldsymbol{\Sigma}^{-1/2}\boldsymbol{e}\right)^\top\left(\boldsymbol{\Sigma}^{-1/2}\boldsymbol{e}\right) = \left\|\boldsymbol{\Sigma}^{-1/2}\boldsymbol{e}\right\|_2^2. \tag{2.22}$$

Hence, we can eliminate the covariances $\boldsymbol{\Sigma}_i$ by pre-multiplying the Jacobian $\boldsymbol{H}_i$ and the prediction error in each term in (2.21b) with $\boldsymbol{\Sigma}_i^{-1/2}$:

$$\boldsymbol{A}_i = \boldsymbol{\Sigma}_i^{-1/2}\boldsymbol{H}_i \tag{2.23a}$$

$$\boldsymbol{b}_i = \boldsymbol{\Sigma}_i^{-1/2}\left(\boldsymbol{z}_i - \boldsymbol{h}_i(\boldsymbol{x}_i^0)\right). \tag{2.23b}$$

This process is a form of *whitening*. For example, in the case of scalar measurements

it simply means dividing each term by the measurement standard deviation σ_i. Note that this eliminates the units of the measurements (e.g., length, angles) so that the different rows can be combined into a single cost function.

2.3.2 SLAM as Least-Squares

After linearization, we finally obtain the following standard least-squares problem:

$$\boldsymbol{\delta}^* = \arg\min_{\boldsymbol{\delta}} \sum_i \|\boldsymbol{A}_i\boldsymbol{\delta}_i - \boldsymbol{b}_i\|_2^2 \tag{2.24a}$$

$$= \arg\min_{\boldsymbol{\delta}} \|\boldsymbol{A}\boldsymbol{\delta} - \boldsymbol{b}\|_2^2, \tag{2.24b}$$

Above $\boldsymbol{A}$ and $\boldsymbol{b}$ are obtained by collecting all whitened Jacobian matrices $\boldsymbol{A}_i$ and whitened prediction errors $\boldsymbol{b}_i$ into one large matrix $\boldsymbol{A}$ and right-hand-side (RHS) vector $\boldsymbol{b}$, respectively.

The Jacobian $\boldsymbol{A}$ is a large-but-sparse matrix, with a block structure that mirrors the structure of the underlying factor graph. We will examine this sparsity structure in detail below. First, however, we review the the classical linear algebra approach to solving this least-squares problem.

2.3.3 Matrix Factorization for Least-Squares

For a full-rank $m \times n$ matrix $\boldsymbol{A}$, with $m \geq n$, the unique least-squares solution to (2.24b) can be found by solving the *normal equations*:

$$\left(\boldsymbol{A}^\top \boldsymbol{A}\right) \boldsymbol{\delta}^* = \boldsymbol{A}^\top \boldsymbol{b}. \tag{2.25}$$

This is normally done by factoring the *information matrix* $\boldsymbol{\Lambda}$ (also called the Hessian matrix), defined and factored as follows:

$$\boldsymbol{\Lambda} \triangleq \boldsymbol{A}^\top \boldsymbol{A} = \boldsymbol{R}^\top \boldsymbol{R}. \tag{2.26}$$

Above, the *Cholesky triangle* $\boldsymbol{R}$ is an upper-triangular $n \times n$ matrix[3] and is computed using *Cholesky factorization*, a variant of lower-upper (LU) factorization for symmetric positive-definite matrices. After this, $\boldsymbol{\delta}^*$ can be found by solving first

$$\boldsymbol{R}^\top \boldsymbol{y} = \boldsymbol{A}^\top \boldsymbol{b} \tag{2.27}$$

for $\boldsymbol{y}$ and then

$$\boldsymbol{R}\,\boldsymbol{\delta}^* = \boldsymbol{y} \tag{2.28}$$

[3] Some treatments, including [101], define the Cholesky triangle as the lower-triangular matrix $\boldsymbol{L} = \boldsymbol{R}^\top$, but the other convention is more convenient here.

for $\boldsymbol{\delta}^*$ by forward and backward substitution, respectively. For dense matrices, Cholesky factorization requires $n^3/3$ flops, and the entire algorithm, including computing half of the symmetric $\boldsymbol{A}^\top \boldsymbol{A}$, requires $(m + n/3)n^2$ flops. One could also use lower-diagonal-upper (LDU) factorization, a variant of Cholesky decomposition that avoids the computation of square roots.

An alternative to Cholesky factorization that is more accurate and more numerically stable is to proceed via *QR-factorization*, which works *without* computing the information matrix $\boldsymbol{\Lambda}$. Instead, we compute the QR-factorization of $\boldsymbol{A}$ itself along with its corresponding RHS:

$$\boldsymbol{A} = \boldsymbol{Q} \begin{bmatrix} \boldsymbol{R} \\ \boldsymbol{0} \end{bmatrix}, \quad \begin{bmatrix} \boldsymbol{d} \\ \boldsymbol{e} \end{bmatrix} = \boldsymbol{Q}^\top \boldsymbol{b}. \tag{2.29}$$

Here $\boldsymbol{Q}$ is an $m \times m$ orthogonal matrix, $\boldsymbol{d} \in \mathbb{R}^n$, $\boldsymbol{e} \in \mathbb{R}^{m-n}$, and $\boldsymbol{R}$ is the *same* upper-triangular Cholesky triangle. The preferred method for factorizing a dense matrix $\boldsymbol{A}$ is to compute $\boldsymbol{R}$ column by column, proceeding from left to right. For each column j, all nonzero elements below the diagonal are zeroed out by multiplying $\boldsymbol{A}$ on the left with a *Householder reflection matrix* $\boldsymbol{H}_j$. After n iterations $\boldsymbol{A}$ is completely factorized:

$$\boldsymbol{H}_n \cdots \boldsymbol{H}_2 \boldsymbol{H}_1 \boldsymbol{A} = \boldsymbol{Q}^\top \boldsymbol{A} = \begin{bmatrix} \boldsymbol{R} \\ \boldsymbol{0} \end{bmatrix}. \tag{2.30}$$

The orthogonal matrix $\boldsymbol{Q}$ is not usually formed: instead, the transformed RHS $\boldsymbol{Q}^\top \boldsymbol{b}$ is computed by appending $\boldsymbol{b}$ as an extra column to $\boldsymbol{A}$. Because the $\boldsymbol{Q}$ factor is orthogonal, we have

$$\|\boldsymbol{A}\,\boldsymbol{\delta} - \boldsymbol{b}\|_2^2 = \left\|\boldsymbol{Q}^\top \boldsymbol{A}\boldsymbol{\delta} - \boldsymbol{Q}^\top \boldsymbol{b}\right\|_2^2 = \|\boldsymbol{R}\,\boldsymbol{\delta} - \boldsymbol{d}\|_2^2 + \|\boldsymbol{e}\|_2^2, \tag{2.31}$$

where we made use of the equalities from (2.29). Clearly, $\|\boldsymbol{e}\|_2^2$ will be the least-squares sum of squared residuals, and the least-squares solution $\boldsymbol{\delta}^*$ can be obtained by solving the triangular system

$$\boldsymbol{R}\,\boldsymbol{\delta}^* = \boldsymbol{d} \tag{2.32}$$

via back-substitution. Note that the upper-triangular factor $\boldsymbol{R}$ obtained using QR factorization is the same (up to possible sign changes on the diagonal) as would be obtained by Cholesky factorization, since

$$\boldsymbol{A}^\top \boldsymbol{A} = \begin{bmatrix} \boldsymbol{R} \\ \boldsymbol{0} \end{bmatrix}^\top \boldsymbol{Q}^\top \boldsymbol{Q} \begin{bmatrix} \boldsymbol{R} \\ \boldsymbol{0} \end{bmatrix} = \boldsymbol{R}^\top \boldsymbol{R}, \tag{2.33}$$

where we again made use of the fact that $\boldsymbol{Q}$ is orthogonal. The cost of QR is dominated by the cost of the Householder reflections, which is $2(m - n/3)n^2$. Comparing this with Cholesky, we see that both algorithms require $O(mn^2)$ operations when $m \gg n$, but that QR-factorization is slower by a factor of 2.

In summary, the linearized optimization problem associated with SLAM can be

concisely stated in terms of basic linear algebra. It comes down to factorizing either the information matrix $\boldsymbol{\Lambda}$ or the measurement Jacobian $\boldsymbol{A}$ into square-root form. Because they are based on matrix square roots derived from the *SAM* problem, we have referred to this family of approaches as *square-root SAM*, or $\sqrt{\text{SAM}}$ for short [65, 68].

2.4 Nonlinear Optimization

In this section, we discuss some classic optimization approaches to the nonlinear least-squares problem defined in (2.18). As a reminder, in SLAM the nonlinear least-squares objective function is given by

$$J(\boldsymbol{x}) \triangleq \sum_i \|\boldsymbol{z}_i - \boldsymbol{h}_i(\boldsymbol{x}_i)\|^2_{\boldsymbol{\Sigma}_i} \tag{2.34}$$

and corresponds to a nonlinear factor graph derived from the measurements along with prior densities on some or all unknowns.

Nonlinear least-squares problems cannot be solved directly in general, but require an iterative solution starting from a suitable initial estimate. Nonlinear optimization methods do so by solving a succession of linear approximations to (2.18) in order to approach the minimum [72]. A variety of algorithms exist that differ in how they locally approximate the cost function, and in how they find an improved estimate based on that local approximation. A general in-depth treatment of nonlinear solvers is provided by [189], while [101] focuses on the linear-algebra perspective.

All of the algorithms share the following basic structure: they start from an initial estimate $\boldsymbol{x}^0$. In each iteration, an update step $\boldsymbol{\delta}$ is calculated and applied to obtain the next estimate $\boldsymbol{x}^{t+1} = \boldsymbol{x}^t + \boldsymbol{\delta}$. This process ends when certain convergence criteria are reached, such as the norm of the change $\boldsymbol{\delta}$ falling below a small threshold.

2.4.1 Steepest Descent

Steepest Descent (SD) uses the direction of steepest descent at the current estimate to calculate the following update step:

$$\boldsymbol{\delta}^{\text{sd}} = -\alpha \, \nabla J(\boldsymbol{x})|_{\boldsymbol{x}=\boldsymbol{x}^t} . \tag{2.35}$$

Here the negative gradient is used to identify the direction of steepest descent. For the nonlinear least-squares objective function (2.34), we locally approximate the objective function as a quadratic, $J(\boldsymbol{x}) \approx \|\boldsymbol{A}(\boldsymbol{x} - \boldsymbol{x}^t) - \boldsymbol{b}\|^2_2$, and obtain the exact gradient $\nabla J(\boldsymbol{x})|_{\boldsymbol{x}=\boldsymbol{x}^t} = -2\boldsymbol{A}^\top \boldsymbol{b}$ at the linearization point $\boldsymbol{x}^t$.

The step size α needs to be carefully chosen to balance between safe updates and reasonable convergence speed. An explicit line search can be performed to find a minimum in the given direction. SD is a simple algorithm, but suffers from slow convergence near the minimum.

2.4.2 Gauss-Newton

Gauss-Newton (GN) provides faster convergence by using a second-order update. GN exploits the special structure of the nonlinear least-squares problem to approximate the Hessian using the Jacobian as $\boldsymbol{A}^\top \boldsymbol{A}$. The GN update step is obtained by solving the normal equations (2.25),

$$\boldsymbol{A}^\top \boldsymbol{A}\, \boldsymbol{\delta}^{\mathrm{gn}} = \boldsymbol{A}^\top \boldsymbol{b}, \tag{2.36}$$

by any of the methods in Section 2.3.3. For a well-behaved (i.e., nearly quadratic) objective function and a good initial estimate, Gauss-Newton exhibits nearly quadratic convergence. If the quadratic fit is poor, a GN step can lead to a new estimate that is further from the minimum and subsequent divergence.

2.4.3 Levenberg-Marquardt

The Levenberg-Marquardt (LM) algorithm allows for iterating multiple times to convergence while controlling in which region one is willing to trust the quadratic approximation made by Gauss-Newton. Hence, such a method is often called a *trust-region method*.

To combine the advantages of both the SD and GN methods, Levenberg [149] proposed to modify the normal equations (2.25) by adding a non-negative constant $\lambda \in \mathbb{R}^+ \cup \{0\}$ to the diagonal

$$\left(\boldsymbol{A}^\top \boldsymbol{A} + \lambda \mathbf{I}\right) \boldsymbol{\delta}^{\mathrm{lm}} = \boldsymbol{A}^\top \boldsymbol{b}. \tag{2.37}$$

Note that for $\lambda = 0$ we obtain GN, and for large λ we approximately obtain $\boldsymbol{\delta}^* \approx \frac{1}{\lambda}\boldsymbol{A}^\top \boldsymbol{b}$, an update in the negative gradient direction of the cost function J (2.34). Hence, LM can be seen to blend naturally between the GN and SD methods.

Marquardt [170] later proposed to take into account the scaling of the diagonal entries to provide faster convergence:

$$\left(\boldsymbol{A}^\top \boldsymbol{A} + \lambda\, \mathrm{diag}(\boldsymbol{A}^\top \boldsymbol{A})\right) \boldsymbol{\delta}^{\mathrm{lm}} = \boldsymbol{A}^\top \boldsymbol{b}. \tag{2.38}$$

This modification causes larger steps in the steepest-descent direction if the gradient is small (nearly flat directions of the objective function), because there the inverse of the diagonal entries will be large. Conversely, in steep directions of the objective function the algorithm becomes more cautious and takes smaller steps. Both modifications of the normal equations can be interpreted in Bayesian terms as adding a zero-mean prior to the system.

A key difference between GN and LM is that the latter rejects updates that would lead to a higher sum of squared residuals. A rejected update means that the nonlinear function is locally not well-behaved, and smaller steps are needed. This is achieved by heuristically increasing the value of λ, for example by multiplying its current value by a factor of 10, and resolving the modified normal equations.

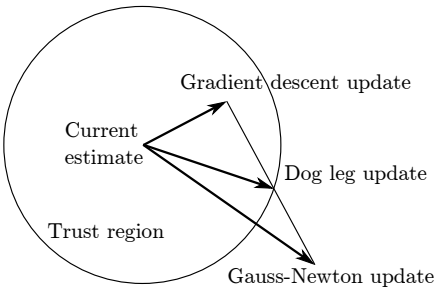


Figure 2.5 Powell's dogleg algorithm combines the separately computed Gauss-Newton and gradient descent update steps.

On the other hand, if a step leads to a reduction of the sum of squared residuals, it is accepted, and the state estimate is updated accordingly. In this case, λ is reduced (*e.g.,* by dividing by a factor of 10), and the algorithm repeats with a new linearization point, until convergence.

2.4.4 Dogleg Minimization

Powell's dogleg (PDL) algorithm [212] can be a more efficient alternative to LM [159]. A major disadvantage of the Levenberg-Marquardt algorithm is that in case a step gets rejected, the modified information matrix has to be refactored, which is the most expensive component of the algorithm. Hence, the key idea behind PDL is to separately compute the GN and SD steps, and then combine appropriately. If the LM step gets rejected, the directions of the GN and SD steps are still valid, and they can be combined in a different way until a reduction in the cost is achieved. Hence, each update of the state estimate only involves one matrix factorization, as opposed to several.

Figure 2.5 shows how the GN and SD steps are combined. The combined step starts with the SD update, followed by a sharp bend (hence the term dogleg) towards the GN update, but stopping at the trust region boundary. Unlike LM, PDL maintains an explicit trust region within which we trust the linear assumption. The appropriateness of the linear approximation is determined by the gain ratio

$$\rho = \frac{J(\boldsymbol{x}^t) - J(\boldsymbol{x}^t + \boldsymbol{\delta})}{L(\mathbf{0}) - L(\boldsymbol{\delta})}, \tag{2.39}$$

where $L(\boldsymbol{\delta}) = \boldsymbol{A}^\top \boldsymbol{A}\boldsymbol{\delta} - \boldsymbol{A}^\top \boldsymbol{b}$ is the linearization of the nonlinear quadratic cost function J from (2.34) at the current estimate $\boldsymbol{x}^t$. If ρ is small (i.e., $\rho < 0.25$) then the cost has not reduced as predicted by the linearization and the trust region is reduced. On the other hand, if the reduction is as predicted (or better, i.e., $\rho > 0.75$), then the trust region is increased depending on the magnitude of the update vector, and the step is accepted.

2.5 Factor Graphs and Sparsity

The solvers presented so far assume that the matrices involved may be dense. Dense methods will not scale to realistic problem sizes in SLAM. For the toy problem in Figure 2.1 a dense method will work fine. The larger simulation example, with its factor graph shown in Figure 2.4, is more representative of real-world problems. However, it is still relatively small as real SLAM problems go, where problems with thousands or even millions of unknowns are common. Yet, we are able to handle these without a problem because of sparsity.

The sparsity can be appreciated directly from looking at the factor graph. It is clear from Figure 2.4 that the graph is *sparse* (i.e., it is by no means a fully connected graph). The odometry chain linking the 100 unknown poses is a linear structure of 100 binary factors, instead of the possible 100^2 (binary) factors. In addition, with 20 landmarks we could have up to 2000 factors linking each landmark to each pose: the true number is closer to 400. And finally, there are no factors between landmarks at all. This reflects that we have not been given any information about their relative position. This structure is typical of most SLAM problems.

2.5.1 The Sparse Jacobian and its Factor Graph

The key to modern SLAM algorithms is exploiting sparsity, and an important property of factor graphs in SLAM is that they represent the sparse block structure in the resulting sparse Jacobian matrix $\boldsymbol{A}$. To see this, let us revisit the least-squares problem that is the key computation in the inner loop of the nonlinear SLAM problem:

$$\boldsymbol{\delta}^* = \arg\min_{\boldsymbol{\delta}} \sum_i \|\boldsymbol{A}_i\,\boldsymbol{\delta}_i - \boldsymbol{b}_i\|_2^2\,. \tag{2.40}$$

Each term above is derived from a factor in the original, nonlinear SLAM problem, linearized around the current linearization point (2.21b). The matrices $\boldsymbol{A}_i$ can be broken up in blocks corresponding to each variable, and collected in a large, block-sparse Jacobian whose sparsity structure is given exactly by the factor graph.

Even though these linear problems typically arise as inner iterations in nonlinear optimization, we drop the $\boldsymbol{\delta}$ notation below, as everything holds for general linear problems regardless of their origin.

Consider the factor graph for the small toy example in Figure 2.1. After linearization, we obtain a sparse system $[\boldsymbol{A}|\boldsymbol{b}]$ with the block structure in Figure 2.6. Comparing this with the factor graph, it is obvious that every factor corresponds to a block-row, and every variable corresponds to a block-column of $\boldsymbol{A}$. In total there are nine block-rows, one for every factor in the factorization of $\phi(\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3, \boldsymbol{\ell}_1, \boldsymbol{\ell}_2)$.

$$
[\boldsymbol{A}|\boldsymbol{b}] = \begin{array}{c} \\ \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \\ \phi_5 \\ \phi_6 \\ \phi_7 \\ \phi_8 \\ \phi_9 \end{array}
\begin{array}{c}
\begin{array}{ccccccc} \delta\boldsymbol{\ell}_1 & \delta\boldsymbol{\ell}_2 & \delta\boldsymbol{p}_1 & \delta\boldsymbol{p}_2 & \delta\boldsymbol{p}_3 & \boldsymbol{b} \end{array} \\
\left[\begin{array}{ccccc|c}
 & & \boldsymbol{A}_{13} & & & \boldsymbol{b}_1 \\
 & & \boldsymbol{A}_{23} & \boldsymbol{A}_{24} & & \boldsymbol{b}_2 \\
 & & & \boldsymbol{A}_{34} & \boldsymbol{A}_{35} & \boldsymbol{b}_3 \\
\boldsymbol{A}_{41} & & & & & \boldsymbol{b}_4 \\
 & \boldsymbol{A}_{52} & & & & \boldsymbol{b}_5 \\
 & & \boldsymbol{A}_{63} & & & \boldsymbol{b}_6 \\
\boldsymbol{A}_{71} & & \boldsymbol{A}_{73} & & & \boldsymbol{b}_7 \\
\boldsymbol{A}_{81} & & & \boldsymbol{A}_{84} & & \boldsymbol{b}_8 \\
 & \boldsymbol{A}_{92} & & & \boldsymbol{A}_{95} & \boldsymbol{b}_9
\end{array}\right]
\end{array}
$$

Figure 2.6 Block structure of the sparse Jacobian $\boldsymbol{A}$ for the toy SLAM example in Figure 2.1 with $\boldsymbol{\delta} = \left[\delta\boldsymbol{\ell}_1^\top \ \delta\boldsymbol{\ell}_2^\top \ \delta\boldsymbol{p}_1^\top \ \delta\boldsymbol{p}_2^\top \ \delta\boldsymbol{p}_3^\top\right]^\top$. Blank entries are zeros.

$$
\begin{bmatrix}
\boldsymbol{\Lambda}_{11} & & \boldsymbol{\Lambda}_{13} & \boldsymbol{\Lambda}_{14} & \\
 & \boldsymbol{\Lambda}_{22} & & & \boldsymbol{\Lambda}_{25} \\
\boldsymbol{\Lambda}_{31} & & \boldsymbol{\Lambda}_{33} & \boldsymbol{\Lambda}_{34} & \\
\boldsymbol{\Lambda}_{41} & & \boldsymbol{\Lambda}_{43} & \boldsymbol{\Lambda}_{44} & \boldsymbol{\Lambda}_{45} \\
 & \boldsymbol{\Lambda}_{52} & & \boldsymbol{\Lambda}_{54} & \boldsymbol{\Lambda}_{55}
\end{bmatrix}
$$

Figure 2.7 The information matrix $\boldsymbol{\Lambda} \triangleq \boldsymbol{A}^\top \boldsymbol{A}$ for the toy SLAM problem.

2.5.2 The Sparse Information Matrix and its Graph

When using Cholesky factorization for solving the normal equations, as explained in Section 2.3.3, we first form the Hessian or information matrix $\boldsymbol{\Lambda} = \boldsymbol{A}^\top \boldsymbol{A}$. [4] In general, since the Jacobian $\boldsymbol{A}$ is block-sparse, the Hessian $\boldsymbol{\Lambda}$ is expected to be sparse as well. By construction, the Hessian is a symmetric matrix, and if a unique MAP solution to the problem exists, it is also positive definite.

The information matrix $\boldsymbol{\Lambda}$ can be associated with another, *undirected* graphical model for the SLAM problem, namely a *Markov random field (MRF)*. In contrast to a factor graph, an MRF is a graphical model that involves only the variables. The graph G of an MRF is an undirected graph: the edges only indicate that there is *some* interaction between the variables involved. At the block level, the sparsity pattern of $\boldsymbol{\Lambda} = \boldsymbol{A}^\top \boldsymbol{A}$ is exactly the adjacency matrix of G.

Figure 2.7 shows the information matrix $\boldsymbol{\Lambda}$ associated with our running toy example. In this case, there are five variables that partition the Hessian as shown. The zero blocks indicate which variables do not interact (e.g., $\boldsymbol{\ell}_1$ and $\boldsymbol{\ell}_2$ have no direct interaction). Figure 2.8 shows the corresponding MRF.

In what follows, we will frequently refer to the undirected graph G of the MRF associated with an inference problem. However, we will not use the MRF graphical

[4] Note that $\boldsymbol{A}^\top \boldsymbol{A}$ is not true Hessian, but is often used to approximate Hessian by truncating a Taylor series of the residual.

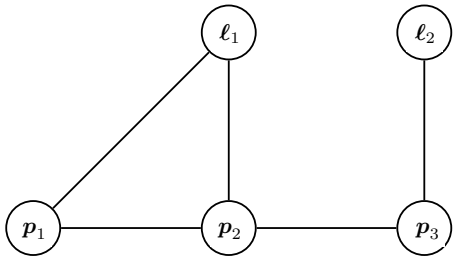


Figure 2.8 The Hessian matrix $\mathbf{\Lambda}$ can be interpreted as the matrix associated with the Markov random field representation for the problem.

model much beyond that as factor graphs are better suited to our needs. They are able to express a finer-grained factorization, and are more closely related to the original problem formulation. For example, if there exist ternary (or higher) factors in the factor graph, the graph G of the equivalent MRF connects those nodes in an undirected clique (a group of fully connected variables), but the origin of the corresponding clique potential is lost. In linear algebra, this reflects the fact that many matrices $\boldsymbol{A}$ can yield the same $\mathbf{\Lambda} = \boldsymbol{A}^\top \boldsymbol{A}$ matrix: important information on the sparsity is lost.

2.5.3 Sparse Factorization

We have seen MAP estimation amounts to solving a linear system of equations as described in Section 2.3.3. In the case of nonlinear least-squares problems, we solve such a system repeatedly in an iterative setup. We have seen in the previous two sections that both $\boldsymbol{A}$ and $\boldsymbol{A}^\top \boldsymbol{A}$ enjoy sparsity determined by the factor graph and MRF connectivity, respectively. Without going into detail, this known sparsity pattern can be used to greatly speed up either Cholesky factorization (in the case of working with $\boldsymbol{A}^\top \boldsymbol{A}$) or QR-factorization (in the case of working with $\boldsymbol{A}$). Efficient software implementations are available, e.g., CHOLMOD [54] and SuiteSparseQR [63], which are also used under the hood by several software packages. In practice, sparse Cholesky or LDU factorization outperform QR factorization on sparse problems as well, and not just by a constant factor.

The flop count for sparse factorization will be much lower than for a dense matrix. Crucially, the column ordering chosen for the sparse matrices can dramatically influence the total flop-count. While any order will ultimately produce an identical MAP estimate, the variable order determines the *fill-in* of matrix factors (i.e., the extra nonzero entries beyond the sparsity pattern of the matrix being factored). It is known that finding the variable ordering that minimizes fill-in during matrix fac-

torization is an NP-hard problem [301], so we must resort to using good heuristics. This will in turn affect the computational complexity of the factorization algorithm.

We demonstrate this by way of an example. Recall the larger simulation example, with its factor graph shown in Figure 2.4. The sparsity patterns for the corresponding sparse Jacobian matrix $\boldsymbol{A}$ is shown in Figure 2.9. Also shown is the pattern for the information matrix $\boldsymbol{\Lambda} \triangleq \boldsymbol{A}^\top \boldsymbol{A}$, in the top-right corner. On the right of Figure 2.9, we show the resulting upper triangular Choleksy factor $\boldsymbol{R}$ for two different orderings. Both of them are sparse, and both of them satisfy $\boldsymbol{R}^\top \boldsymbol{R} = \boldsymbol{A}^\top \boldsymbol{A}$ (up to a permutation of the variables), but they differ in the amount of sparsity they exhibit. It is exactly this that will determine how expensive it is to factorize $\boldsymbol{A}$. The first version of the ordering comes naturally: the poses come first and then the landmarks, leading to a sparse $\boldsymbol{R}$ factor with 9399 nonzero entries. In contrast, the sparse factor $\boldsymbol{R}$ in the bottom right was obtained by reordering the variables according to the Column approximate minimum degree permutation (COLAMD) heuristic [12, 64] and only has 4168 nonzero entries. Yet back-substitution gives exactly the same solution for both versions.

It is worth mentioning that other tools, like pre-conditioned conjugate gradient, can solve the normal equations *iteratively*. In visual SLAM, which has a very specific sparsity pattern, power iterations have also been used successfully [282]. However, sparse factorization is still the method of choice for most SLAM problems and has a nice graphical model interpretation, which we discuss next.

2.6 Elimination

We have so far restricted ourselves to a linear-algebra explanation of performing inference for SLAM. In this section, we expand our worldview by thinking about inference more abstractly using graphical models directly. This will ultimately lead us to current state-of-the-art SLAM solvers based on a concept called the *Bayes tree* for incremental smoothing and mapping in the next section.

2.6.1 Variable Elimination Algorithm

There exists a general algorithm that can, given any (preferably sparse) factor graph, compute the corresponding posterior density $p(\boldsymbol{x}|\boldsymbol{z})$ on the unknown variables $\boldsymbol{x}$ in a form that allows easy recovery of the MAP solution to the problem. As we saw, a factor graph represents the unnormalized posterior $\phi(\boldsymbol{x}) \propto p(\boldsymbol{x}|\boldsymbol{z})$ as a product of factors, and in SLAM problems this graph is typically generated directly from the measurements. The *variable elimination* algorithm is a recipe for converting a factor graph into another graphical model called a *Bayes net*, which depends only on the unknown variables $\boldsymbol{x}$. This then allows for easy MAP inference (as well as other operations such as sampling and/or marginalization).

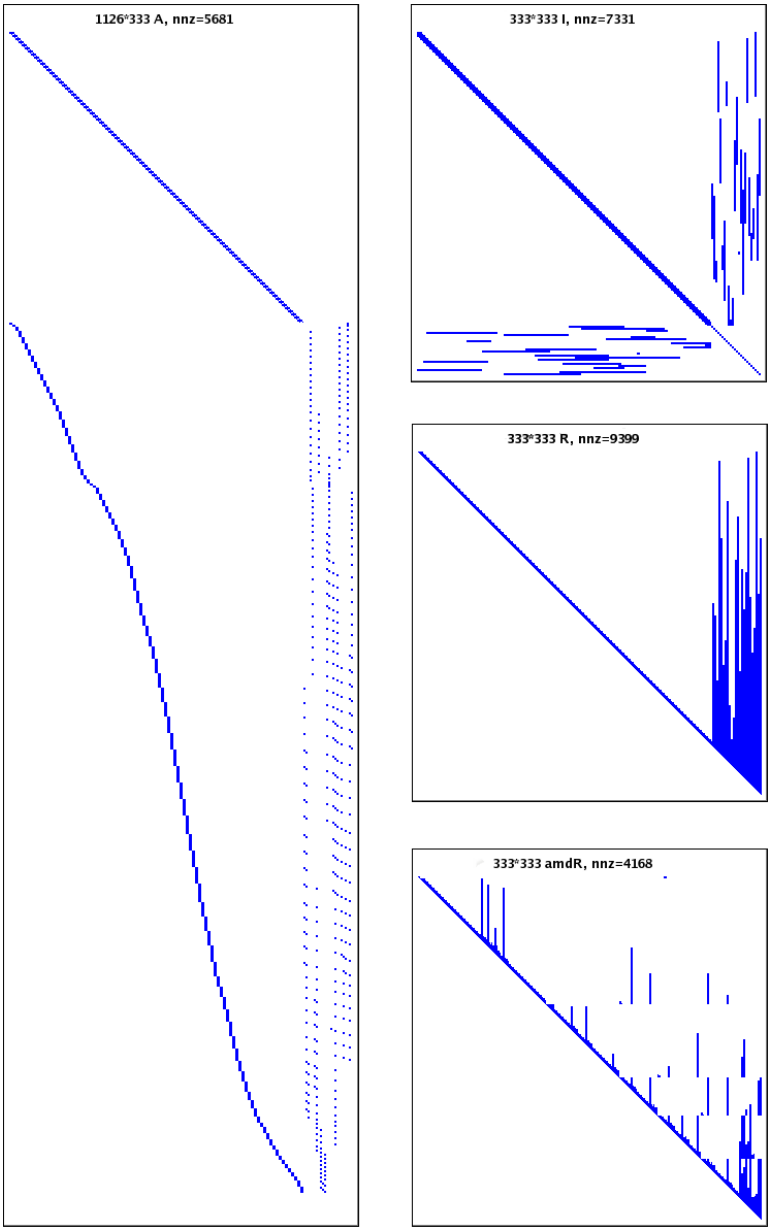


Figure 2.9 On the left, the measurement Jacobian $\boldsymbol{A}$ associated with the problem in Figure 2.4, which has $3 \times 95 + 2 \times 24 = 333$ unknowns. The number of rows, 1126, is equal to the number of (scalar) measurements. Also given is the number of nonzero entries "nnz". On the right: (top) the information matrix $\boldsymbol{\Lambda} \triangleq \boldsymbol{A}^\top \boldsymbol{A}$; (middle) its upper triangular Cholesky triangle $\boldsymbol{R}$; (bottom) an alternative factor $amdR$ obtained with a better variable ordering (COLAMD).

In particular, the variable elimination algorithm is a way to factorize any factor graph of the form

$$\phi(\boldsymbol{x}) = \phi(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \tag{2.41}$$

into a factored Bayes net probability density of the form

$$p(\boldsymbol{x}) = p(\boldsymbol{x}_1|\boldsymbol{s}_1)p(\boldsymbol{x}_2|\boldsymbol{s}_2)\ldots p(\boldsymbol{x}_n) = \prod_j p(\boldsymbol{x}_j|\boldsymbol{s}_j), \tag{2.42}$$

where $\boldsymbol{s}_j$ denotes an assignment to the *separator* $\boldsymbol{s}(\boldsymbol{x}_j)$ associated with variable $\boldsymbol{x}_j$ under the chosen variable ordering $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$. The separator is defined as the set of variables on which $\boldsymbol{x}_j$ is conditioned, after elimination. While this factorization is akin to the chain rule, eliminating a sparse factor graph will typically lead to small separators.

The elimination algorithm proceeds by eliminating one variable $\boldsymbol{x}_j$ at a time, starting with the complete factor graph $\phi_{1:n}$. As we eliminate each variable $\boldsymbol{x}_j$, we generate a single conditional $p(\boldsymbol{x}_j|\boldsymbol{s}_j)$, as well as a reduced factor graph $\phi_{j+1:n}$ on the remaining variables. After all variables have been eliminated, the algorithm returns the resulting Bayes net with the desired factorization.

To eliminate a single variable $\boldsymbol{x}_j$ given a partially eliminated factor graph $\phi_{j:n}$, we first remove all factors $\phi_i(\boldsymbol{x}_i)$ that are adjacent to $\boldsymbol{x}_j$ and multiply them into the product factor $\psi(\boldsymbol{x}_j, \boldsymbol{s}_j)$. We then factorize $\psi(\boldsymbol{x}_j, \boldsymbol{s}_j)$ into a conditional distribution $p(\boldsymbol{x}_j|\boldsymbol{s}_j)$ on the eliminated variable $\boldsymbol{x}_j$, and a new factor $\tau(\boldsymbol{s}_j)$ on the separator $\boldsymbol{s}_j$:

$$\psi(\boldsymbol{x}_j, \boldsymbol{s}_j) = p(\boldsymbol{x}_j|\boldsymbol{s}_j)\tau(\boldsymbol{s}_j). \tag{2.43}$$

Hence, *the entire factorization from $\phi(\boldsymbol{x})$ to $p(\boldsymbol{x})$ is seen to be a succession of n local factorization steps.* When eliminating the last variable $\boldsymbol{x}_n$ the separator $\boldsymbol{s}_n$ will be empty, and the conditional produced will simply be a prior $p(\boldsymbol{x}_n)$ on $\boldsymbol{x}_n$.

One possible elimination sequence for the toy example is shown in Figure 2.10, for the ordering $\boldsymbol{\ell}_1$, $\boldsymbol{\ell}_2$, $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, $\boldsymbol{p}_3$. In each step, the variable being eliminated is shaded gray, and the new factor $\tau(\boldsymbol{s}_j)$ on the separator $\boldsymbol{s}_j$ is shown in red. Taken as a whole, the variable elimination algorithm factorizes the factor graph $\phi(\boldsymbol{\ell}_1, \boldsymbol{\ell}_2, \boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3)$ into the Bayes net in Figure 2.10 (bottom right), corresponding to the factorization

$$p(\boldsymbol{\ell}_1, \boldsymbol{\ell}_2, \boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3) = p(\boldsymbol{\ell}_1|\boldsymbol{p}_1, \boldsymbol{p}_2)\, p(\boldsymbol{\ell}_2|\boldsymbol{p}_3)\, p(\boldsymbol{p}_1|\boldsymbol{p}_2)\, p(\boldsymbol{p}_2|\boldsymbol{p}_3)\, p(\boldsymbol{p}_3). \tag{2.44}$$

2.6.2 Linear-Gaussian Elimination

In the case of linear measurement functions and additive normally distributed noise, the *elimination algorithm is equivalent to sparse matrix factorization.* Both sparse Cholesky and QR factorization are a special case of the general algorithm.

As explained before, the elimination algorithm proceeds one variable at a time. For every variable $\boldsymbol{x}_j$ we remove all factors $\phi_i(\boldsymbol{x}_i)$ adjacent to $\boldsymbol{x}_j$ and form the

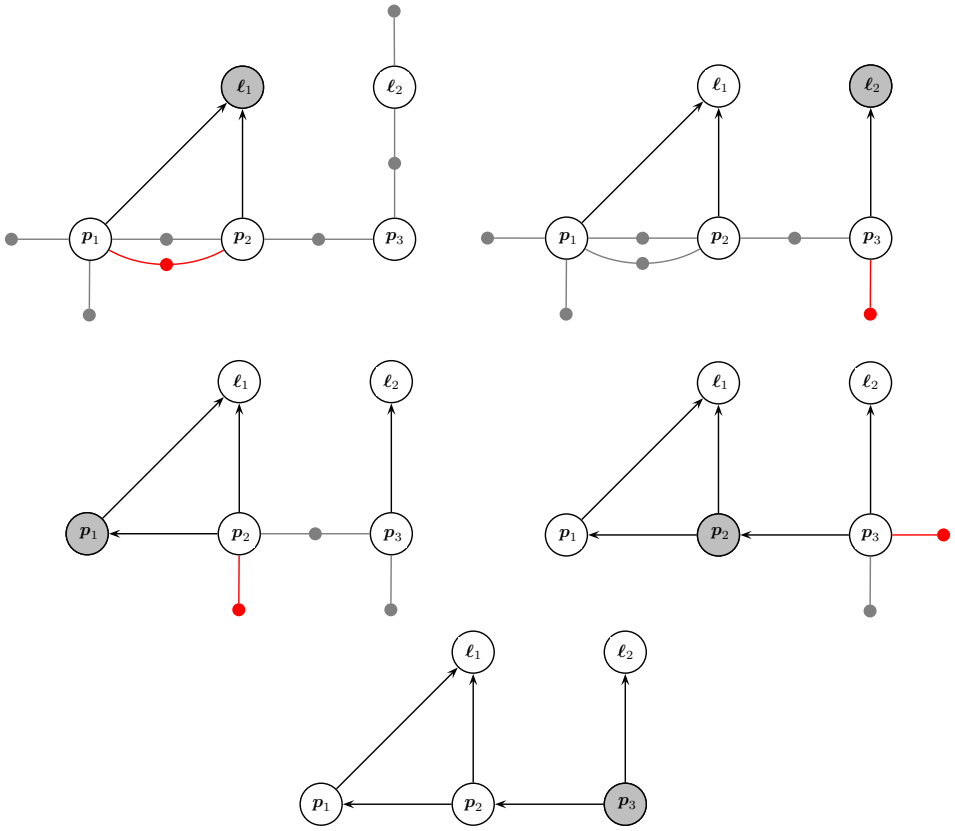


Figure 2.10 Variable elimination for the toy SLAM example, transforming the factor graph from Figure 2.2 into a Bayes net (bottom right), using the ordering $\boldsymbol{\ell}_1$, $\boldsymbol{\ell}_2$, $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, $\boldsymbol{p}_3$.

intermediate product factor $\psi(\boldsymbol{x}_j, \boldsymbol{s}_j)$. This can be done by accumulating all the matrices $\boldsymbol{A}_i$ into a new, larger block-matrix $\bar{\boldsymbol{A}}_j$, as we can write

$$\psi(\boldsymbol{x}_j, \boldsymbol{s}_j) \leftarrow \prod_{i \in \mathcal{N}_j} \phi_i(\boldsymbol{x}_i) \tag{2.45a}$$

$$= \exp\left(-\frac{1}{2} \sum_i \|\boldsymbol{A}_i \boldsymbol{x}_i - \boldsymbol{b}_i\|_2^2\right) \tag{2.45b}$$

$$= \exp\left(-\frac{1}{2} \left\|\bar{\boldsymbol{A}}_j [\boldsymbol{x}_j; \boldsymbol{s}_j] - \bar{\boldsymbol{b}}_j\right\|_2^2\right), \tag{2.45c}$$

where the new RHS vector $\bar{\boldsymbol{b}}_j$ stacks all $\boldsymbol{b}_i$ and ';' also denotes vertical stacking.

Consider eliminating the variable $\boldsymbol{\ell}_1$ in the toy example. The adjacent factors are ϕ_4, ϕ_7, and ϕ_8, in turn inducing the separator $\boldsymbol{s}_1 = [\boldsymbol{p}_1; \boldsymbol{p}_2]$. The product factor is

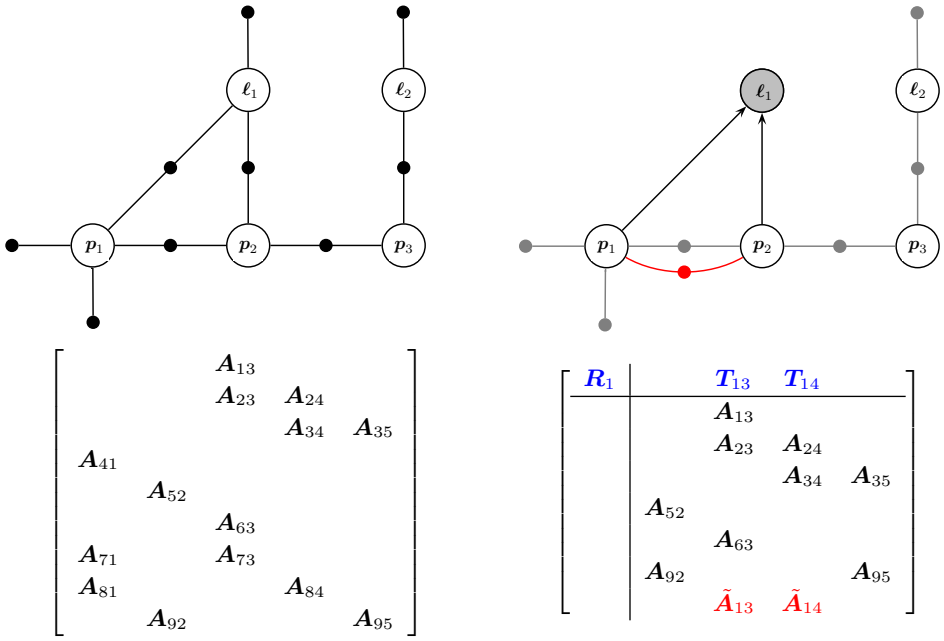


Figure 2.11 Eliminating the variable $\boldsymbol{\ell}_1$ as a partial sparse factorization step.

then equal to

$$\psi\left(\boldsymbol{\ell}_1, \boldsymbol{p}_1, \boldsymbol{p}_2\right) = \exp\left(-\frac{1}{2}\left\|\bar{\boldsymbol{A}}_1[\boldsymbol{\ell}_1; \boldsymbol{p}_1; \boldsymbol{p}_2] - \bar{\boldsymbol{b}}_1\right\|_2^2\right), \tag{2.46}$$

with

$$\bar{\boldsymbol{A}}_1 \triangleq \begin{bmatrix} \boldsymbol{A}_{41} & & \\ \boldsymbol{A}_{71} & \boldsymbol{A}_{73} & \\ \boldsymbol{A}_{81} & & \boldsymbol{A}_{84} \end{bmatrix}, \qquad \bar{\boldsymbol{b}}_1 \triangleq \begin{bmatrix} \boldsymbol{b}_4 \\ \boldsymbol{b}_7 \\ \boldsymbol{b}_8 \end{bmatrix}. \tag{2.47}$$

Looking at the sparse Jacobian in Figure 2.6, this simply boils down to taking out the block-rows with nonzero blocks in the first column, corresponding to the three factors adjacent to $\boldsymbol{\ell}_1$.

Next, factorizing the product $\psi(\boldsymbol{x}_j, \boldsymbol{s}_j)$ can be done in several different ways. We discuss the QR variant, as it more directly connects to the linearized factors. In particular, the augmented matrix $[\bar{\boldsymbol{A}}_j | \bar{\boldsymbol{b}}_j]$ corresponding to the product factor $\psi(\boldsymbol{x}_j, \boldsymbol{s}_j)$ can be rewritten using partial QR-factorization [101] as follows:

$$[\bar{\boldsymbol{A}}_j | \bar{\boldsymbol{b}}_j] = \boldsymbol{Q} \begin{bmatrix} \boldsymbol{R}_j & \boldsymbol{T}_j & \boldsymbol{d}_j \\ & \tilde{\boldsymbol{A}}_\tau & \tilde{\boldsymbol{b}}_\tau \end{bmatrix}, \tag{2.48}$$

where $\boldsymbol{R}_j$ is an upper-triangular matrix. This allows us to factor $\psi(\boldsymbol{x}_j, \boldsymbol{s}_j)$ as follows:

$$\psi(\boldsymbol{x}_j, \boldsymbol{s}_j) = \exp\left\{-\frac{1}{2}\left\|\bar{\boldsymbol{A}}_j[\boldsymbol{x}_j; \boldsymbol{s}_j] - \bar{\boldsymbol{b}}_j\right\|_2^2\right\} \tag{2.49a}$$

$$= \exp\left\{-\frac{1}{2}\left\|\boldsymbol{R}_j\boldsymbol{x}_j + \boldsymbol{T}_j\boldsymbol{s}_j - \boldsymbol{d}_j\right\|_2^2\right\} \exp\left\{-\frac{1}{2}\left\|\tilde{\boldsymbol{A}}_\tau \boldsymbol{s}_j - \tilde{\boldsymbol{b}}_\tau\right\|_2^2\right\}$$

$$= p(\boldsymbol{x}_j|\boldsymbol{s}_j)\tau(\boldsymbol{s}_j), \tag{2.49b}$$

where we used the fact that the rotation matrix $\boldsymbol{Q}$ does not alter the value of the norms involved.

In the toy example, Figure 2.11 shows the result of eliminating the first variable in the example, the landmark $\boldsymbol{\ell}_1$ with separator $[\boldsymbol{p}_1; \boldsymbol{p}_2]$. We show the operation on the factor graph *and* the corresponding effect on the sparse Jacobian from Figure 2.6, omitting the RHS. The partition above the line corresponds to a sparse, upper-triangular matrix $\boldsymbol{R}$ that is being formed. New contributions to the matrix are shown: blue for the contributions to $\boldsymbol{R}$, and red for newly created factors. For completeness, we show the four remaining variable elimination steps in Figure 2.12, showing an end-to-end example of how QR factorization proceeds on a small example. The final step shows the equivalence between the resulting Bayes net and the sparse upper-triangular factor $\boldsymbol{R}$.

The entire elimination algorithm, using partial QR to eliminate a single variable, is equivalent to *sparse QR factorization*. As the treatment above considers multi-dimensional variables $\boldsymbol{x}_j \in \mathbb{R}^{n_j}$, this is in fact an instance of *multi-frontal QR factorization* [77], as we eliminate several scalar variables at a time, which is beneficial for processor utilization. While in our case the scalar variables are grouped because of their semantic meaning in the inference problem, sparse linear algebra codes typically analyze the problem to group for maximum computational efficiency. In many cases these two strategies are closely aligned.

2.6.3 Sparse Cholesky Factor as a Bayes Net

The equivalence between variable elimination and sparse matrix factorization reveals that the graphical model associated with an upper triangular matrix is a Bayes net! Just like a factor graph is the graphical embodiment of a sparse Jacobian, and an MRF can be associated with the Hessian, a Bayes net reveals the sparsity structure of a Cholesky factor. In hindsight, this perhaps is not too surprising: a Bayes net is a directed acyclic graph (DAG), and that is exactly the 'upper-triangular' property for matrices.

What's more, the Cholesky factor corresponds to a *Gaussian Bayes net*, which we define as one made up of linear-Gaussian conditionals. The variable elimination algorithm holds for general densities, but in case the factor graph only contains linear measurement functions and Gaussian additive noise, the resulting Bayes net

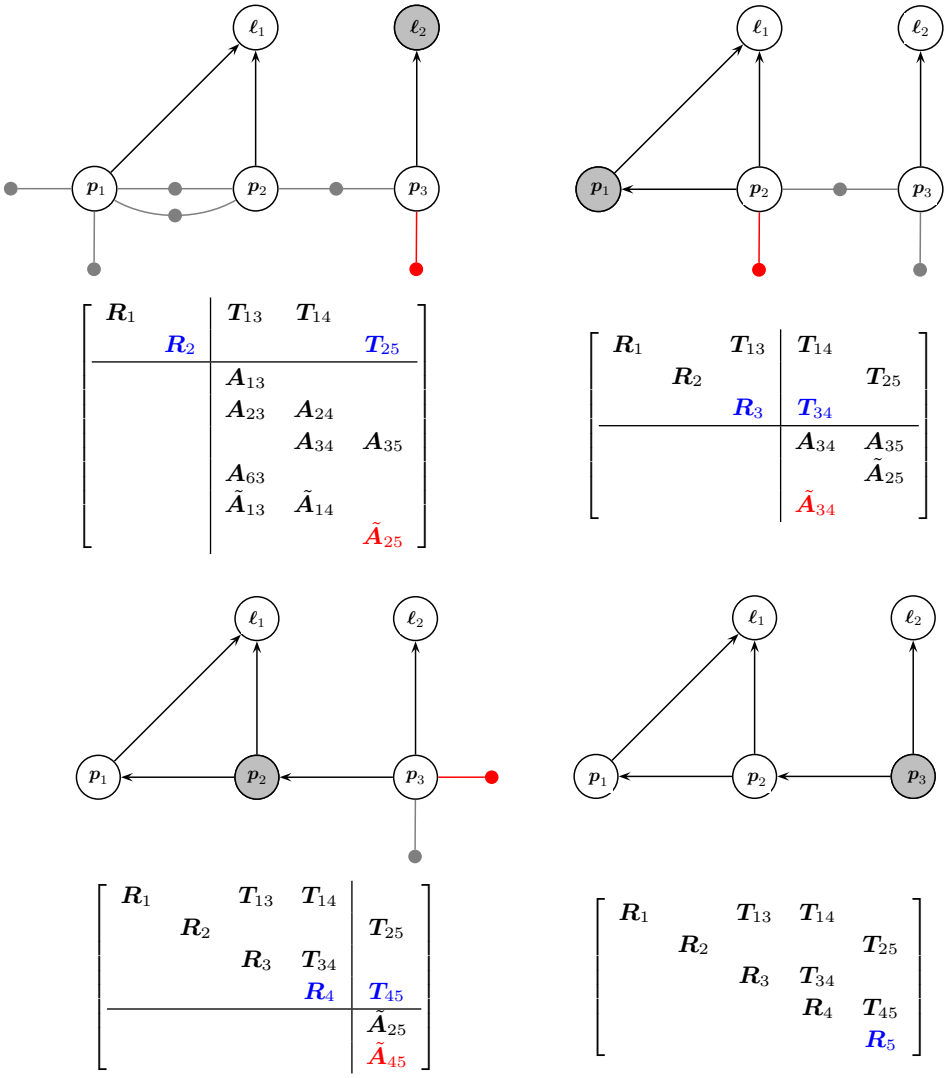


Figure 2.12 The remaining elimination steps for the toy example, completing a full QR factorization. The last step in the bottom right shows the equivalence between the resulting Bayes net and the sparse Cholesky factor $\boldsymbol{R}$.

has a very specific form. We discuss the details below, as well as how to solve for the MAP estimate in the linear case.

As we discussed, the Gaussian factor graph corresponding to the linearized nonlinear problem is transformed by elimination into the density $p(\boldsymbol{x})$ given by the

now-familiar Bayes-net factorization:

$$p(\boldsymbol{x}) = \prod_j p(\boldsymbol{x}_j|\boldsymbol{s}_j). \tag{2.50}$$

In both QR and Cholesky variants, the conditional densities $p(\boldsymbol{x}_j|\boldsymbol{s}_j)$ are given by

$$p(\boldsymbol{x}_j|\boldsymbol{s}_j) = k \exp\left(-\frac{1}{2}\left\|\boldsymbol{R}_j\boldsymbol{x}_j + \boldsymbol{T}_j\boldsymbol{s}_j - \boldsymbol{d}_j\right\|_2^2\right), \tag{2.51}$$

which is a linear-Gaussian density on the eliminated variable $\boldsymbol{x}_j$. Indeed, we have

$$\left\|\boldsymbol{R}_j\boldsymbol{x}_j + \boldsymbol{T}_j\boldsymbol{s}_j - \boldsymbol{d}_j\right\|_2^2 = (\boldsymbol{x}_j - \boldsymbol{\mu}_j)^\top \boldsymbol{R}_j^\top \boldsymbol{R}_j (\boldsymbol{x}_j - \boldsymbol{\mu}_j) \triangleq \|\boldsymbol{x}_j - \boldsymbol{\mu}_j\|_{\boldsymbol{\Sigma}_j}^2, \tag{2.52}$$

where the mean $\boldsymbol{\mu}_j = \boldsymbol{R}_j^{-1}(\boldsymbol{d}_j - \boldsymbol{T}_j\boldsymbol{s}_j)$ depends linearly on the separator $\boldsymbol{s}_j$, and the covariance matrix is given by $\boldsymbol{\Sigma}_j = (\boldsymbol{R}_j^\top \boldsymbol{R}_j)^{-1}$. Hence, the normalization constant $k = |2\pi\boldsymbol{\Sigma}_j|^{-\frac{1}{2}}$.

After the elimination step is complete, back-substitution is used to obtain the MAP estimate of each variable. As seen in Figure 2.12, the last variable eliminated does not depend on any other variables. Thus, the MAP estimate of the last variable can be directly extracted from the Bayes net. By proceeding in reverse elimination order, the values of all the separator variables for each conditional will always be available from the previous steps, allowing the estimate for the current frontal variable to be computed.

At every step, the MAP estimate for the variable $\boldsymbol{x}_j$ is the conditional mean,

$$\boldsymbol{x}_j^* = \boldsymbol{R}_j^{-1}(\boldsymbol{d}_j - \boldsymbol{T}_j\boldsymbol{s}_j^*), \tag{2.53}$$

since by construction the MAP estimate for the separator $\boldsymbol{s}_j^*$ is fully known by this point.

2.7 Incremental SLAM

In an incremental SLAM setting, we want to compute the optimal trajectory and map whenever we receive new measurements while traversing the environment, or at least at regular intervals. One way to do so is to *update* the most recent matrix factorization with the new measurements, to reuse the computation that already incorporated all previous measurements. In the linear case, this is possible through incremental factorization methods, the dense versions of which are also discussed at length in Golub and Loan [101]. However, matrix factorization operates on linear systems, but most SLAM problems of practical interest are *nonlinear*. Using incremental matrix factorization, it is far from obvious how re-linearization can be performed incrementally without refactoring the complete matrix. To overcome this problem we once again resort to graphical models, and introduce a new graphical model, the *Bayes tree*. We then show how to incrementally update the Bayes tree as

new measurements and states are added to the system, leading to the incremental smoothing and mapping (iSAM) algorithm.

2.7.1 The Bayes Tree

It is well known that inference in a tree-structured graph is efficient. In contrast, the factor graphs associated with typical robotics problems contain many loops. Still, we can construct a tree-structured graphical model in a two-step process: first, perform variable elimination on the factor graph to obtain a Bayes net with a special property. Second, exploit that special property to find a tree structure over *cliques* in this Bayes net.

In particular, a Bayes net obtained by running the elimination algorithm on a factor graph satisfies a special property: it is *chordal*, meaning that any undirected cycle of length greater than three has a *chord*, i.e., an edge connecting two non-consecutive vertices on the cycle. In AI and machine learning, a chordal graph is more commonly said to be *triangulated*. Because it is still a Bayes net, the corresponding joint density $p(\boldsymbol{x})$ is given by factorizing over the individual variables $\boldsymbol{x}_j$,

$$p(\boldsymbol{x}) = \prod_j p(\boldsymbol{x}_j|\boldsymbol{\pi}_j), \tag{2.54}$$

where $\boldsymbol{\pi}_j$ are the parent nodes of $\boldsymbol{x}_j$. However, although the Bayes net is chordal, at this variable level it is still a non-trivial graph: neither chain-like nor tree-structured. The chordal Bayes net for our running toy SLAM example is shown in the last step of Figure 2.10, and it is clear that there is an undirected cycle $\boldsymbol{p}_1 - \boldsymbol{p}_2 - \boldsymbol{\ell}_1$, implying it does not have a tree-structured form.

By identifying cliques in this chordal graph, the Bayes net may be rewritten as a *Bayes tree*. We introduce this new, tree-structured graphical model to capture the *clique structure* of the Bayes net. It is not obvious that cliques in the Bayes net should form a tree. They do so because of the chordal property, although we will not attempt to prove that here. Listing all these cliques in an undirected tree yields a *clique tree*, also known as a *junction tree* in AI and machine learning. The Bayes tree is just a directed version of this that preserves information about the elimination order.

More formally, a Bayes tree is a directed tree where the nodes represent *cliques* $\boldsymbol{c}_k$ of the underlying chordal Bayes net. In particular, we define one conditional density $p(\boldsymbol{f}_k|\boldsymbol{s}_k)$ per node, with the *separator* $\boldsymbol{s}_k$ as the intersection $\boldsymbol{c}_k \cap \boldsymbol{\varpi}_k$ of the clique $\boldsymbol{c}_k$ and its parent clique $\boldsymbol{\varpi}_k$. The *frontal variables* $\boldsymbol{f}_k$ are the remaining variables, i.e., $\boldsymbol{f}_k \triangleq \boldsymbol{c}_k \setminus \boldsymbol{s}_k$. Notationally, we write $\boldsymbol{c}_k = \boldsymbol{f}_k : \boldsymbol{s}_k$ for a clique. The following expression gives the joint density $p(\boldsymbol{x})$ on the variables $\boldsymbol{x}$ defined by a Bayes tree:

$$p(\boldsymbol{x}) = \prod_k p(\boldsymbol{f}_k|\boldsymbol{s}_k). \tag{2.55}$$

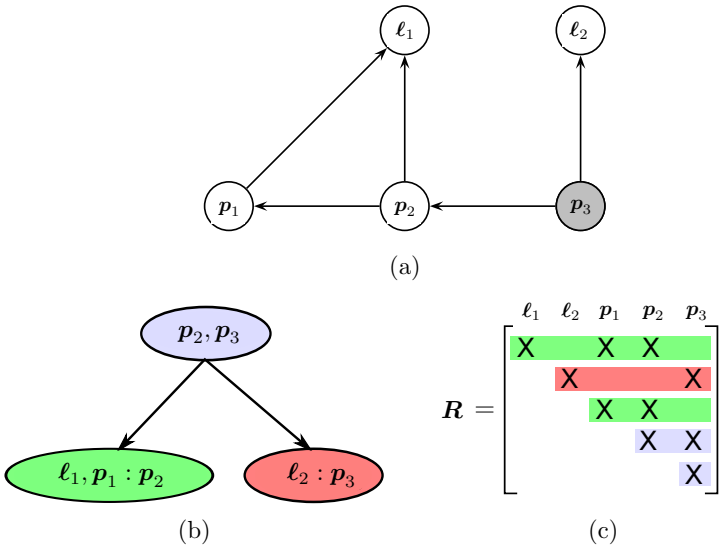


Figure 2.13 The Bayes tree (b) and the associated square root information matrix $\boldsymbol{R}$ (c) describing the clique structure in the chordal Bayes net (a) based on our canonical example from Figure 2.2. A Bayes tree is similar to a clique tree, but is better at capturing the formal equivalence between sparse linear algebra and inference in graphical models. The association of cliques with rows in the $\boldsymbol{R}$ factor is indicated by color.

For the root $\boldsymbol{f}_r$, the separator is empty, i.e., it is a simple prior $p(\boldsymbol{f}_r)$ on the root variables. The way Bayes trees are defined, the separator $\boldsymbol{s}_k$ for a clique $\boldsymbol{c}_k$ is always a subset of the parent clique $\boldsymbol{\varpi}_k$, and hence the directed edges in the graph have the same semantic meaning as in a Bayes net: conditioning.

The Bayes tree associated with our canonical toy SLAM problem (Figure 2.2) is shown in Figure 2.13. The root clique $\boldsymbol{c}_1 = \boldsymbol{p}_2, \boldsymbol{p}_3$ (shown in blue) comprises $\boldsymbol{p}_2$ and $\boldsymbol{p}_3$, which intersects with two other cliques, $\boldsymbol{c}_2 = \boldsymbol{\ell}_1, \boldsymbol{p}_1 : \boldsymbol{p}_2$ shown in green, and $\boldsymbol{c}_3 = \boldsymbol{\ell}_2 : \boldsymbol{p}_3$ shown in red. The colors also indicate how the rows of square-root information matrix $\boldsymbol{R}$ map to the different cliques, and how the Bayes tree captures independence relationships between them. For example, the green and red rows only intersect in variables that belong to the root clique, as predicted.

2.7.2 Updating the Bayes Tree

Incremental inference corresponds to a simple editing of the Bayes tree. This view provides a better explanation and understanding of the otherwise abstract incremental matrix factorization process. It also allows us to store and compute the square-root information matrix in the form of a Bayes tree, a deeply meaningful sparse storage scheme.

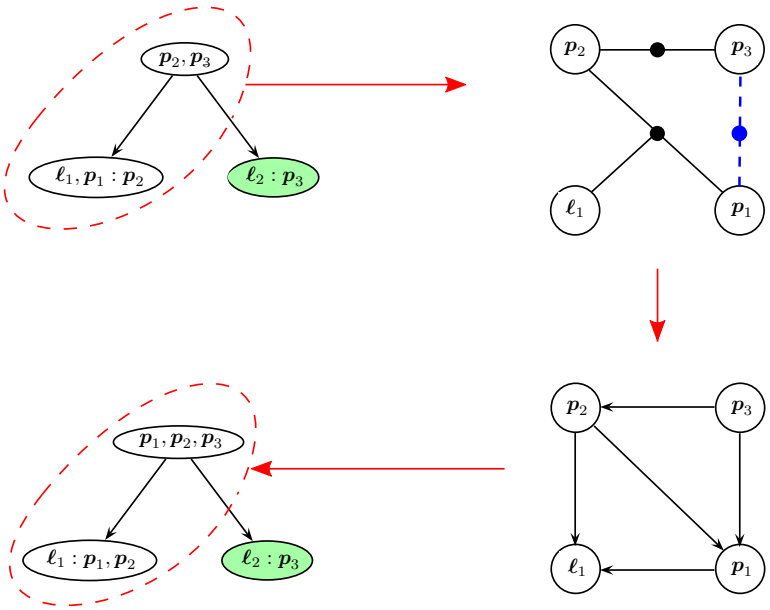


Figure 2.14 Updating a Bayes tree with a new factor, based on the example in Figure 2.13. The affected part of the Bayes tree is highlighted for the case of adding a new factor between $\boldsymbol{p}_1$ and $\boldsymbol{p}_3$. Note that the right branch (green) is not affected by the change. (top right) The factor graph generated from the affected part of the tree with the new factor (dashed blue) inserted. (bottom right) The chordal Bayes net resulting from eliminating the factor graph. (bottom left) The Bayes tree created from the chordal Bayes net, with the unmodified right 'orphan' sub-tree from the original Bayes tree added back in.

To incrementally update the Bayes tree, we selectively convert part of the Bayes tree back into factor-graph form. When a new measurement is added this corresponds to adding a factor, e.g., a measurement involving two variables will induce a new binary factor $\phi(\boldsymbol{x}_j, \boldsymbol{x}_{j'})$. In this case, *only* the paths in the Bayes tree between the cliques containing $\boldsymbol{x}_j$ and $\boldsymbol{x}_{j'}$ and the root will be affected. The sub-trees below these cliques are unaffected, as are any other sub-trees not containing $\boldsymbol{x}_j$ or $\boldsymbol{x}_{j'}$. Hence, to update the Bayes tree, the affected parts of the tree are converted back into a factor graph, and the new factor associated with the new measurement is added to it. By re-eliminating this temporary factor graph, using whatever elimination ordering is convenient, a new Bayes tree is formed and the unaffected sub-trees can be reattached.

In order to understand why only the top part of the tree is affected, we look at two important properties of the Bayes tree. These directly arise from the fact that it encodes the information flow during elimination. The Bayes tree is formed from the chordal Bayes net following the inverse elimination order. In this way, variables in each clique collect information from their child cliques via the elimination of

these children. Thus, information in any clique propagates only upwards to the root. Second, the information from a factor enters elimination only when the first variable connected to that factor is eliminated. Combining these two properties, we see that a new factor cannot influence any other variables that are not successors of the factor's variables. However, a factor involving variables having different (i.e., independent) paths to the root means that these paths must now be re-eliminated to express the new dependency between them.

Figure 2.14 shows how these incremental factorization/inference steps are applied to our canonical SLAM example. In this example, we add a new factor between $\boldsymbol{p}_1$ and $\boldsymbol{p}_3$, affecting only the left branch of the tree, marked by the red dashed line in to top-left figure. We then create the factor graph shown in the top-right figure by creating a factor for each of the clique densities, $p(\boldsymbol{p}_2, \boldsymbol{p}_3)$ and $p(\boldsymbol{\ell}_1, \boldsymbol{p}_1 | \boldsymbol{p}_2)$, and add the new factor $f(\boldsymbol{p}_1, \boldsymbol{p}_3)$. The bottom-right figure shows the eliminated graph using the ordering $\boldsymbol{\ell}_1, \boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3$. And finally, in the bottom-left figure, the reassembled Bayes tree is shown consisting of two parts: the Bayes tree derived from the eliminated graph, and the unaffected clique from the original Bayes tree (shown in green).

Figure 2.15 shows an example of the Bayes tree for a small SLAM sequence. Shown is the tree for step 400 of the well-known Manhattan world simulated sequence by Olson et al. [195]. As a robot explores the environment, new measurements only affect parts of the tree, and only those parts are re-calculated.

2.7.3 Incremental Smoothing and Mapping

Putting all of the above together and addressing some practical considerations about re-linearization yields a state-of-the-art incremental, nonlinear approach to MAP estimation in robotics, iSAM. The first version, iSAM1[125], used the incremental matrix factorization methods from Golub and Loan [101]. However, linearization in iSAM1 was handled in a sub-optimal way: it was done for the full factor graph at periodic instances and/or when matrix fill-in became unwieldy. The second version of the approach, iSAM2, uses a Bayes tree representation for the posterior density [127]. It then employs Bayes tree incremental updating as each new measurement comes in, as described above.

What variable ordering should we use in re-eliminating the affected cliques? Only the variables in the affected part of the Bayes tree are updated. One strategy then is to apply COLAMD locally to the affected variables. However, we can do better: we force recently accessed variables to the end of the ordering, i.e., into the root clique. For this incremental variable ordering strategy one can use the constrained COLAMD algorithm [64]. This both forces the most recently accessed variables to the end and still provides a good overall ordering. Generally, subsequent updates will then only affect a small part of the tree, and can therefore be expected to be efficient in most cases, except for large loop closures.

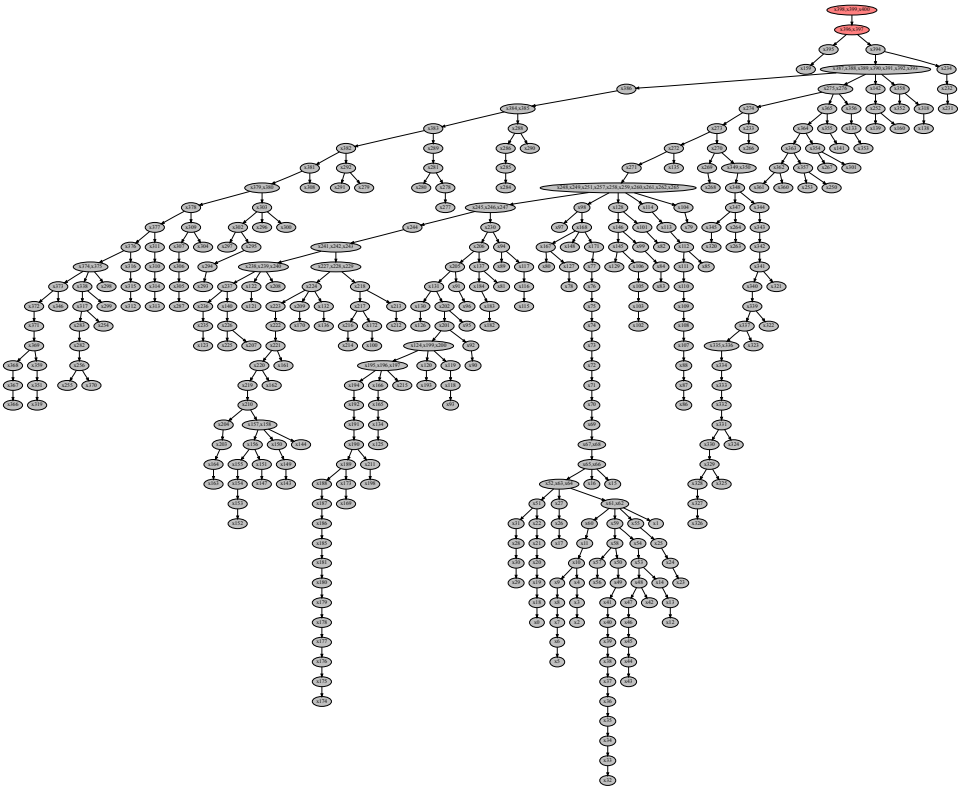


Figure 2.15 An example of the Bayes tree data structure for a small SLAM sequence. The incremental nonlinear least-squares estimation algorithm iSAM2 [127] is based on viewing incremental factorization as editing the graphical model corresponding to the posterior probability of the solution, the Bayes tree. As a robot explores the environment, new measurements often only affect small parts of the tree, and only those parts are recalculated (shown in red).

After updating the tree we also need to *update the solution*. Back-substitution in the Bayes tree proceeds from the root (which does not depend on any other variables) and proceeds to the leaves. However, it is typically not necessary to recompute a solution for all variables: local updates to the tree often do not affect variables in remote parts of the tree. Instead, at each clique we can check the difference in variable estimates that is propagated downwards and stop when this difference falls below a small threshold.

Our motivation for introducing the Bayes tree was to incrementally solve nonlinear optimization problems. For this we *selectively re-linearize* factors that contain variables whose deviation from the linearization point exceeds a small threshold. In contrast to the tree modification above, we now have to redo all cliques that contain

the affected variables, not just as frontal variables, but also as separator variables. This affects larger parts of the tree, but in most cases is still significantly cheaper than recomputing the complete tree. We also have to go back to the original factors, instead of directly turning the cliques into a factor graph. And that requires caching certain quantities during elimination. The overall incremental nonlinear algorithm, iSAM2, is described in much more detail in [127].

iSAM1 and iSAM2 have been applied successfully to many different robotics estimation problems with non-trivial constraints between variables that number into the millions, as will be discussed subsequent chapters. Both are implemented in the GTSAM library, which can be found at `https://github.com/borglab/gtsam`.

3

Advanced State Variable Representations

Timothy Barfoot, Frank Dellaert, Michael Kaess

The previous chapter detailed how to set up and solve a SLAM problem using the factor-graph paradigm. We deliberately avoided discussing some subtleties of the state variables we were estimating. In this chapter, we revisit the nature of our state variables and introduce two important topics that are prevalent in modern SLAM formulations. First and foremost, we need some better tools for handling state variables that have certain constraints associated with them; these constraints define a *manifold* for our variables, which subsequently then require special care during optimization. There are many examples of manifolds that appear in SLAM, the most common being those associated with the rotational aspects of a robot (especially in three dimensions, but even in the plane). A second aspect of state variables stems from the nature of time itself. In the previous chapter, we implicitly assumed that our robot moved in discrete-time steps through the world. In this chapter we introduce smooth, *continuous-time* representations of trajectories and discuss how these are fully compatible with our factor-graph formulation. We use Barfoot [17] as the primary reference with some streamlined notation from Sola et al. [240].

3.1 Optimization on Manifolds

While in some robotics problems we can get away with vector-valued unknowns, in most practical situations we have to deal with three-dimensional rotations and other non-vector manifolds. Loosely speaking, a manifold is collection of points forming a topologically closed surface (e.g., the perimeter of a circle, or the surface of a sphere); importantly, a manifold resembles Euclidean space locally near each point. Manifolds require a more sophisticated machinery that takes into account their special structure. In this section, we discuss how to perform optimization on manifolds, which will build upon the optimization framework for vector spaces from the previous chapter. As an example, Figure 3.1 visualizes a spherical manifold, $\mathcal{M}$, and its tangent space, $T_{\boldsymbol{\chi}}\mathcal{M}$, which can be used as a local coordinate system at $\boldsymbol{\chi} \in \mathcal{M}$ for optimization.

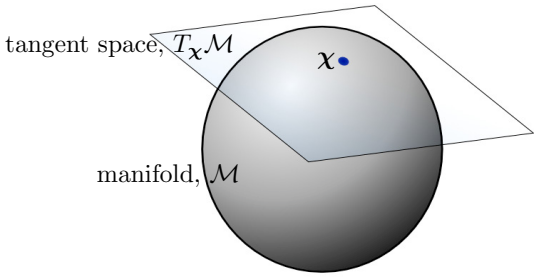


Figure 3.1 For the sphere manifold, $\mathcal{M}$, the local tangent plane, $T_{\boldsymbol{\chi}}\mathcal{M}$, with a local basis provides the notion of local coordinates.

3.1.1 Rotations and Poses

While there are several manifolds that can be discussed in the context of SLAM, the two most common are those used to represent rotations and poses. Rotations are typically either in two (planar) or three dimensions and we therefore refer to the manifold of rotations as the *special orthogonal group* $\mathrm{SO}(d)$, where $d = 2$ or 3, accordingly. A planar *rotation matrix*, $\boldsymbol{R}_a^b \in \mathrm{SO}(2)$, has the form

$$\boldsymbol{R}_a^b = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}, \tag{3.1}$$

where $\theta \in \mathbb{R}$, the angle of rotation, is the single degree of freedom in this case. Moreover, $\boldsymbol{R}_a^b$ allows us to rotate a two-dimensional vector (i.e., landmark) expressed in reference frame $\mathcal{F}^a$ to $\mathcal{F}^b$: $\boldsymbol{\ell}^b = \boldsymbol{R}_a^b \boldsymbol{\ell}^a$.

A rotation matrix in three dimensions, $\boldsymbol{R}_a^b \in \mathrm{SO}(3)$, again rotates vectors (this time in three dimensions) from one frame to another. Three-dimensional rotation matrices have nine entries but only three degrees of freedom (e.g., roll, pitch, yaw). Both two- and three-dimensional rotation matrices must satisfy the constraints $\boldsymbol{R}_a^{b\top} \boldsymbol{R}_a^b = \mathbf{I}$ and $\det(\boldsymbol{R}_a^b) = 1$ to limit their degrees of freedom appropriately.

The *pose* of a robot comprises both rotational, $\boldsymbol{R}_a^b \in \mathrm{SO}(d)$, and translational, $\boldsymbol{t}_a^b \in \mathbb{R}^d$, variables with $3(d-1)$ degrees of freedom in all. Sometimes we keep track of these quantities separately and then can use $\{\boldsymbol{R}_a^b, \boldsymbol{t}_a^b\} \in \mathrm{SO}(d) \times \mathbb{R}^d$ as the representation. Alternatively, these quantities can be assembled into a $(d{+}1){\times}(d{+}1)$ *transformation matrix*,

$$\boldsymbol{T}_a^b = \begin{bmatrix} \boldsymbol{R}_a^b & \boldsymbol{t}_a^b \\ \mathbf{0} & 1 \end{bmatrix}. \tag{3.2}$$

The manifold of all such transformation matrices is called the *special Euclidean group*, $\mathrm{SE}(d)$, where again $d = 2$ (planar motion) or 3 (three-dimensional motion). The benefit of using $\mathrm{SE}(d)$ is that we can easily translate and rotate landmarks

using a single matrix multiplication:

$$\underbrace{\begin{bmatrix}\boldsymbol{\ell}^b\\ 1\end{bmatrix}}_{\tilde{\boldsymbol{\ell}}^b} = \underbrace{\begin{bmatrix}\boldsymbol{R}_a^b & \boldsymbol{t}_a^b\\ \mathbf{0} & 1\end{bmatrix}}_{\boldsymbol{T}_a^b}\underbrace{\begin{bmatrix}\boldsymbol{\ell}^a\\ 1\end{bmatrix}}_{\tilde{\boldsymbol{\ell}}^a}. \tag{3.3}$$

We refer to $\tilde{\boldsymbol{\ell}}$ as the *homogeneous* representation of the landmark $\boldsymbol{\ell}$.

Due to the constraints imposed on the forms of rotation and transformation matrices, they are unfortunately not vectors. For example, we cannot simply add two rotation matrices together and arrive at another valid rotation matrix. However, it turns out that $\mathrm{SO}(d)$ and $\mathrm{SE}(d)$ are examples of manifolds that possess some extra useful properties called *matrix Lie groups*. Thankfully, we can exploit the structure of these manifolds to continue to perform unconstrained MAP optimization for factor-graph SLAM (see, for example, Dellaert et al. [69] or Boumal [36] or Barfoot [17]).

3.1.2 Matrix Lie Groups

The key to performing optimization on $\mathrm{SO}(d)$ and $\mathrm{SE}(d)$ is to exploit their group structure. For example, one nice property is that matrix Lie groups enjoy *closure* so that if we multiply two members, e.g., $\boldsymbol{R}_b^c, \boldsymbol{R}_a^b \in \mathrm{SO}(d)$, the result is also in the group: $\boldsymbol{R}_a^c = \boldsymbol{R}_b^c \boldsymbol{R}_a^b \in \mathrm{SO}(d)$.

Another nice property of matrix Lie groups is that they come along with a very useful companion structure called a *Lie algebra*, which is also the tangent space for the Lie group. For our purposes, the most important aspects of the Lie algebra are (i) that it comprises a vector space with dimension equal to the number of degrees of freedom of its Lie group, and (ii) there is a well-established mapping (the matrix exponential) from the Lie algebra to the Lie group. This allows us to construct elements of the Lie group with relative ease from elements of the Lie algebra. For example, for $\mathrm{SO}(2)$ we can build a rotation matrix (dropping super/subscripts for now) according to

$$\boldsymbol{R} = \mathrm{Exp}(\theta) = \exp(\theta^\wedge) = \sum_{n=0}^{\infty} \frac{1}{n!}(\theta^\wedge)^n \in \mathrm{SO}(2),\ \theta^\wedge = \begin{bmatrix}0 & -\theta\\ \theta & 0\end{bmatrix},\ \theta \in \mathbb{R}. \tag{3.4}$$

The quantity, $\theta^\wedge$, is a member of the Lie algebra, $\mathrm{so}(2)$, and it is mapped through the matrix exponential, $\exp(\cdot)$, to a member of the Lie group, $\boldsymbol{R}$. We can go the other way with the matrix logarithm: $\theta = \mathrm{Log}(\boldsymbol{R}) = (\log(\boldsymbol{R}))^\vee$.

Each matrix Lie group has its own linear $(\cdot)^\wedge$ operator used to construct a Lie algebra member from the standard vector space of appropriate dimension. For $\mathrm{SO}(3)$,

it is the skew-symmetric operator:

$$\boldsymbol{R} = \text{Exp}(\boldsymbol{\theta}) = \exp(\boldsymbol{\theta}^\wedge) = \sum_{n=0}^{\infty} \frac{1}{n!} \boldsymbol{\theta}^{\wedge^n} \in \text{SO}(3), \tag{3.5a}$$

$$\boldsymbol{\theta}^\wedge = \begin{bmatrix} 0 & -\theta_3 & \theta_2 \\ \theta_3 & 0 & -\theta_1 \\ -\theta_2 & \theta_1 & 0 \end{bmatrix} \in \text{so}(3), \quad \boldsymbol{\theta} = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix} \in \mathbb{R}^3. \tag{3.5b}$$

For $\text{SE}(d)$, we can use

$$\boldsymbol{T} = \text{Exp}(\boldsymbol{\xi}) = \exp(\boldsymbol{\xi}^\wedge) \in \text{SE}(d), \tag{3.6a}$$

$$\boldsymbol{\xi}^\wedge = \begin{bmatrix} \boldsymbol{\theta}^\wedge & \boldsymbol{\rho} \\ \boldsymbol{0} & 0 \end{bmatrix} \in \text{se}(d), \quad \boldsymbol{\xi} = \begin{bmatrix} \boldsymbol{\rho} \\ \boldsymbol{\theta} \end{bmatrix} \in \mathbb{R}^{3(d-1)}, \quad \boldsymbol{\theta} \in \mathbb{R}^{2d-3}, \quad \boldsymbol{\rho} \in \mathbb{R}^d, \tag{3.6b}$$

where $d = 2$ (planar) or 3 (three-dimensional). Note, the version of the $(\cdot)^\wedge$ operator can be determined by the size of the input vector.

For each of the matrix Lie groups discussed here, there are also well-known closed-form expressions for the mappings between the Lie algebra and the Lie group that can be used rather than the infinite series form of the matrix exponential [17].

3.1.3 Lie Group Optimization

Now that we have these matrix Lie groups established, we can use them to help 'linearize' our nonlinear least-squares terms in order to carry out MAP inference. Looking back to the discussion in Section 2.3.1, we still seek to linearize our measurement functions, $\boldsymbol{h}_i(\cdot)$, only now the input to these may involve a member of a Lie group.

For example, suppose $\boldsymbol{h}_i(\cdot)$ represents a camera model that takes as its input a homogeneous landmark expressed in the camera frame, $\tilde{\boldsymbol{\ell}}_i^c$, and returns the pixel coordinates of the landmark in an image, $\boldsymbol{z}_i \in \mathbb{R}^2$: $\boldsymbol{z}_i = \boldsymbol{h}_i(\tilde{\boldsymbol{\ell}}_i^c)$. We can write the generative sensor model therefore as

$$\boldsymbol{z}_i = \boldsymbol{h}_i\left(\boldsymbol{T}_w^c \tilde{\boldsymbol{\ell}}_i^w\right) + \boldsymbol{\eta}_i, \tag{3.7}$$

where $\boldsymbol{T}_w^c \in \text{SE}(3)$ is the pose of the camera with respect to a world frame, $\tilde{\boldsymbol{\ell}}_i^w$ is the homogeneous landmark expressed in the world frame, and $\boldsymbol{\eta}_i$ is the usual sensor noise. We then might like to solve the optimization problem

$$\boldsymbol{T}_w^{c^*} = \underset{\boldsymbol{T}_w^c}{\arg\min} = \sum_i \left\| \boldsymbol{z}_i - \boldsymbol{h}_i\left(\boldsymbol{T}_w^c \tilde{\boldsymbol{\ell}}_i^w\right) \right\|_{\boldsymbol{\Sigma}_i}^2, \tag{3.8}$$

which is known as the perspective-n-point (PNP) problem. For this example, we assume that the positions of the landmarks in the world frame are known but of course in SLAM we might like to estimate these as well.

To linearize our sensor model, we use the fact that we can produce a perturbed version of our pose through its Lie algebra according to[1]

$$\boldsymbol{T}_w^c = \boldsymbol{T}_w^{c^0} \operatorname{Exp}\left(\boldsymbol{\xi}_w^c\right). \tag{3.9}$$

Here, $\boldsymbol{\xi}_w^c \in \mathbb{R}^6$ is used to produce a 'small' pose change that perturbs an initial guess, $\boldsymbol{T}_w^{c^0} \in \mathrm{SE}(3)$. This perturbation is also sometimes written succinctly using the $\oplus$ operator so that

$$\boldsymbol{T}_w^c = \boldsymbol{T}_w^{c^0} \oplus \boldsymbol{\xi}_w^c \tag{3.10}$$

implies (3.9). Owing to the closure property discussed earlier, the product of these two quantities is guaranteed to be in SE(3). By using the Lie algebra to define our pose perturbation, we restrict its dimension to be equal to the actual number of degrees of freedom in a three-dimensional pose, which will mean that we can avoid introducing constraints during optimization.

We can also approximate the perturbed pose according to

$$\boldsymbol{T}_w^c \approx \boldsymbol{T}_w^{c^0} \left(\mathbf{I} + \boldsymbol{\xi}_w^{c^\wedge}\right), \tag{3.11}$$

where we have kept just the terms up to linear in $\boldsymbol{\xi}_w^c$ from the series form of the matrix exponential. Then, inserting (3.11) into our measurement function (3.7), we have

$$\boldsymbol{z}_i \approx \boldsymbol{h}_i\left(\boldsymbol{T}_w^{c^0}\left(\mathbf{I} + \boldsymbol{\xi}_w^{c^\wedge}\right)\tilde{\boldsymbol{\ell}}_i^w\right) + \boldsymbol{\eta}_i. \tag{3.12}$$

This can also be rewritten as

$$\boldsymbol{z}_i \approx \boldsymbol{h}_i\left(\boldsymbol{T}_w^{c^0}\tilde{\boldsymbol{\ell}}_i^w + \boldsymbol{T}_w^{c^0}\tilde{\boldsymbol{\ell}}_i^{w^\odot}\boldsymbol{\xi}_w^c\right) + \boldsymbol{\eta}_i, \tag{3.13}$$

where $\odot$ is a (linear) operator for homogeneous points [17]:

$$\tilde{\boldsymbol{\ell}}^\odot = \begin{bmatrix}\boldsymbol{\ell}\\ 1\end{bmatrix}^\odot = \begin{bmatrix}\mathbf{I} & -\boldsymbol{\ell}^\wedge\\ \mathbf{0} & \mathbf{0}\end{bmatrix}. \tag{3.14}$$

We have essentially 'linearized' the pose perturbation in (3.13) and now need to linearize the camera function $\boldsymbol{h}_i(\cdot)$ as well. We can use a standard first-order Taylor series approximation to write

$$\boldsymbol{z}_i \approx \boldsymbol{h}_i\left(\boldsymbol{T}_w^{c^0}\tilde{\boldsymbol{\ell}}_i^w\right) + \underbrace{\left.\frac{\partial \boldsymbol{h}_i}{\partial \boldsymbol{\ell}}\right|_{\boldsymbol{T}_w^{c^0}\tilde{\boldsymbol{\ell}}_i^w} \boldsymbol{T}_w^{c^0}\tilde{\boldsymbol{\ell}}_i^{w^\odot}}_{\boldsymbol{H}_i \text{ (chain rule)}}\boldsymbol{\xi}_w^c + \boldsymbol{\eta}_i, \tag{3.15}$$

where the chaining of two pieces into the overall Jacobian, $\boldsymbol{H}_i$, is now clear. Looking

[1] It is also possible to perturb on the left side rather than the right. A more subtle question is whether the perturbation is happening on the 'sensor' side or the 'world' side, which depends on whether the unknown transform is $\boldsymbol{T}_w^c$ or $\boldsymbol{T}_c^w$ and whether the perturbation is applied to the left or the right.

back to (2.21b), we can write the linearized least-squares term (i.e., negative-log factor) for this measurement as

$$\left\| \left(\boldsymbol{z}_i - \boldsymbol{h}_i \left(\boldsymbol{T}_w^{c^0} \tilde{\boldsymbol{\ell}}_i^w \right) \right) - \boldsymbol{H}_i \, \boldsymbol{\xi}_w^c \right\|_{\boldsymbol{\Sigma}_i}^2 , \tag{3.16}$$

where the only unknown is our pose perturbation, $\boldsymbol{\xi}_w^c$. After combining this with other factors and then solving for the optimal updates to our state variables, including $\boldsymbol{\xi}_w^{c^*}$, we need to update our initial guess, $\boldsymbol{T}_w^{c^0}$. For this, we must return to the perturbation scheme we chose in (3.9) and update according to

$$\boldsymbol{T}_w^{c^0} \leftarrow \boldsymbol{T}_w^{c^0} \oplus \boldsymbol{\xi}_w^{c^*}, \tag{3.17}$$

to ensure our solution, $\boldsymbol{T}_w^{c^0}$, remains in SE(3). As usual, optimization proceeds iteratively until the change in all the state variable updates (including $\boldsymbol{\xi}_w^c$) is sufficiently small.

To recap, we have shown how to carry out unconstrained optimization for a state variable that is a member of a Lie group. Although our example was specific to a three-dimensional pose variable, other Lie groups can be optimized in a similar manner. The key is to arrive at a situation as in (3.15) where the measurement function has been linearized with respect to a perturbation in the *Lie algebra*. Most times, as in our example, this can be done analytically. However, it is also straightforward to compute the required Jacobian, $\boldsymbol{H}_i$, numerically or through automatic differentiation (by exploiting the chain rule and some primitives for Lie groups). In (3.9), we perturbed our pose variable on the left side, but this was a choice and in some cases perturbing on the right may be preferable.

Stepping back a bit, this approach to optimizing a function of a Lie group member is an example of *Riemannian optimization* [36]. By exploiting the Lie algebra, which is also the *tangent space* of a manifold, we constrain the optimization to be *tangent* to the manifold of poses (or rotations). By carrying out the update according to (3.17), we are *retracting* our update back onto the manifold. Riemannian optimization is a very general concept that can be applied to quantities that live on manifolds that are not matrix Lie groups as well. Retractions other than the matrix exponential are also possible within the manifold-optimization framework (e.g., see Dellaert et al. [69] or Barfoot et al. [18]).

3.1.4 Lie Group Extras

There is a lot more that we could say about Lie groups [243, 36] but have so far restrained ourselves in the interest of keeping things simple. We use this section to collect a few more useful facts that come up later in this and other chapters. Further details of carrying out derivatives of functions of Lie group elements will be provided in Section 5.3.

3.1.4.1 The $\oplus$ and $\ominus$ Operators

We have already seen the use of the $\oplus$ operator to compose a Lie algebra vector with a Lie group member. For $\mathrm{SE}(d)$ we have

$$\boldsymbol{T} = \boldsymbol{T}^0 \oplus \boldsymbol{\xi} = \boldsymbol{T}^0 \mathrm{Exp}(\boldsymbol{\xi}) = \boldsymbol{T}^0 \exp(\boldsymbol{\xi}^\wedge) \in \mathrm{SE}(d). \tag{3.18}$$

We often have occasion to consider the 'difference' of two Lie group elements and for this we can also define the $\ominus$ operator. Again for $\mathrm{SE}(d)$ we have

$$\boldsymbol{\xi} = \boldsymbol{T}^0 \ominus \boldsymbol{T} = \mathrm{Log}\left(\boldsymbol{T}^0 \boldsymbol{T}^{-1}\right) = \log\left(\boldsymbol{T}^0 \boldsymbol{T}^{-1}\right)^\vee \in \mathrm{se}(d). \tag{3.19}$$

These operators are a nice way to abstract away the details of these operations.

3.1.4.2 Inverses

Sometimes when we are carrying out perturbations, we have need to perturb the inverse of a rotation or transformation matrix. In the $\mathrm{SE}(d)$ case, we simply have that

$$\left(\boldsymbol{T}^0 \mathrm{Exp}(\boldsymbol{\xi})\right)^{-1} = \mathrm{Exp}(\boldsymbol{\xi})^{-1} \boldsymbol{T}^{0^{-1}} = \mathrm{Exp}(-\boldsymbol{\xi}) \boldsymbol{T}^{0^{-1}}, \tag{3.20}$$

where we see the perturbation moves from the right to the left with a negative sign.

3.1.4.3 Adjoints

The *adjoint* of a Lie group is a way of describing the elements of that group as linear transformations of its Lie algebra, which we recall is a vector space. For $\mathrm{SO}(d)$, the adjoint representation is the same as the group itself, so we omit the details. For $\mathrm{SE}(d)$, the adjoint differs from the group's primary representation and so we use this section to provide some details. The adjoint will prove to be an essential tool when setting up state estimation problems, particularly for $\mathrm{SE}(d)$.

The *adjoint map* of $\mathrm{SE}(d)$ transforms a Lie algebra element $\boldsymbol{\xi}^\wedge \in \mathrm{se}(d)$ to another element of $\mathrm{se}(d)$ according to a map known as the *inner automorphism* or *conjugation*:

$$\mathrm{Ad}_{\boldsymbol{T}} \boldsymbol{\xi}^\wedge = \boldsymbol{T} \boldsymbol{\xi}^\wedge \boldsymbol{T}^{-1}. \tag{3.21}$$

We can equivalently express the output of this map as

$$\mathrm{Ad}_{\boldsymbol{T}} \boldsymbol{\xi}^\wedge = \left(\mathrm{Ad}(\boldsymbol{T}) \boldsymbol{\xi}\right)^\wedge, \tag{3.22}$$

where $\mathrm{Ad}(\boldsymbol{T})$ linearly transforms $\boldsymbol{\xi} \in \mathbb{R}^6$ to $\mathbb{R}^6$. We will refer to $\mathrm{Ad}(\boldsymbol{T})$ as the *adjoint representation* of $\mathrm{SE}(d)$.

The $(2d) \times (2d)$ transformation matrix, $\mathrm{Ad}(\boldsymbol{T})$, can be constructed directly from the components of the $(d+1) \times (d+1)$ transformation matrix:

$$\mathrm{Ad}(\boldsymbol{T}) = \mathrm{Ad}\left(\begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0} & 1 \end{bmatrix}\right) = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t}^\wedge \boldsymbol{R} \\ \boldsymbol{0} & \boldsymbol{R} \end{bmatrix}. \tag{3.23}$$

One situation in which adjoints are useful in our estimation problems is to manipulate perturbations from one side of a known transformation to another as in

$$\boldsymbol{T}\operatorname{Exp}(\boldsymbol{\xi}) = \operatorname{Exp}\left(\operatorname{Ad}(\boldsymbol{T})\boldsymbol{\xi}\right)\boldsymbol{T}, \tag{3.24}$$

which we emphasize does not require approximation.

3.1.4.4 Jacobians

Every Lie group also has a Jacobian associated with it, which allows us to relate changes in an element of the group to elements of its algebra. For the case of $\mathrm{SO}(d)$, for example, the common kinematic equation (i.e., Poisson's equation) relating a rotation matrix, $\boldsymbol{R} \in \mathrm{SO}(d)$, to angular velocity, $\boldsymbol{\omega} \in \mathbb{R}^{3(d-1)}$, is

$$\dot{\boldsymbol{R}} = \boldsymbol{\omega}^{\wedge}\boldsymbol{R}. \tag{3.25}$$

If we parameterize $\boldsymbol{R} = \operatorname{Exp}(\boldsymbol{\theta})$, then we can equivalently write

$$\dot{\boldsymbol{\theta}} = \boldsymbol{J}^{-1}(\boldsymbol{\theta})\,\boldsymbol{\omega}, \tag{3.26}$$

where $\boldsymbol{J}(\boldsymbol{\theta})$ is the (left) Jacobian of $\mathrm{SO}(d)$. A place where this Jacobian is quite useful is when combining expressions involving products of matrix exponentials. For example, we have that

$$\operatorname{Exp}(\boldsymbol{\theta}_1)\operatorname{Exp}(\boldsymbol{\theta}_2) \approx \operatorname{Exp}\left(\boldsymbol{\theta}_2 + \boldsymbol{J}(\boldsymbol{\theta}_2)^{-1}\boldsymbol{\theta}_1\right), \tag{3.27}$$

where $\boldsymbol{\theta}_1$ is assumed to be 'small'. The series expression for $\boldsymbol{J}(\theta)$ is

$$\boldsymbol{J}(\boldsymbol{\theta}) = \sum_{n=0}^{\infty} \frac{1}{(n+1)!}\left(\boldsymbol{\theta}^{\wedge}\right)^n, \tag{3.28}$$

and a closed-form expression can be found in Barfoot [17]. We will overload and write $\boldsymbol{J}(\boldsymbol{\xi})$ for the (left) Jacobian of $\mathrm{SE}(d)$ where the context should inform which is meant.

3.2 Continuous-Time Trajectories

Continuous-time trajectories offer a way to represent smooth robot motions. In our development so far, we have assumed a discrete sequence of poses along a trajectory is to be estimated. However, robots typically move fairly smoothly through the world, which motivates the use of a smoother representation of trajectory. Continuous-time trajectories come primarily in two varieties: *parametric* methods combine known temporal basis functions into a smooth trajectory. Typically, these temporal basis functions are chosen to have *local support* (e.g., piecewise polynomials / splines), which ensures the factor graph remains sparse, as we will see. *Nonparametric* methods have higher representational power by making use of *kernel functions*. Specifically, a one-dimensional *Gaussian process (GP)* with time as the

independent variable can be used to represent a trajectory. When an appropriate physically motivated kernel is chosen, we will see that the factor graph associated with a GP also remains very sparse.

In addition to trajectory smoothness, the use of a continuous-time trajectory can be particularly useful when working with high-rate and/or asynchronous sensors. In the factor-graph examples that we have considered so far, we added robot poses to the factor graph for each newly collected measurement (*e.g.,* to model that the current pose is taking a landmark measurement). This quickly leads to unwieldy factor graphs when using high-rate sensors or when different sensors collect measurements at different time instants. Below, we will see that we can easily represent the trajectory with a number of variables that is much smaller than the number of measurements, to keep things tractable. This is particularly useful for *motion-distorted* sensors such as spinning lidars and radars and even rolling-shutter cameras; using continuous-time trajectories we can account for the exact time stamp of each point or pixel and relate them to the trajectory at that instant.

Finally, after MAP inference, continuous-time trajectories allow us to efficiently query the trajectory at any time of interest, not just at the measurement times. We can both interpolate and extrapolate (with caution), which can be useful for consumers of our SLAM outputs. Separating the roles of measurements times, estimation variables, and query times, is a major advantage of both parametric and nonparametric continuous-time methods.

3.2.1 Splines

The idea with parametric continuous-time trajectory methods is to write the pose as a weighted sum of K known *temporal basis functions*, $\boldsymbol{\Psi}_k(t)$:

$$\boldsymbol{p}(t) = \sum_{k=1}^{K} \boldsymbol{\Psi}_k(t)\boldsymbol{c}_k, \tag{3.29}$$

where the $\boldsymbol{c}_k$ are the unknown *coefficients*. For now, we return to a vector-space explanation and discuss implementation on Lie groups in a later section. The basis functions are typically chosen to be *splines*, which are piecewise polynomials (e.g., B-splines, cubic Hermite polynomials); splines are advantageous because they have *local support* meaning outside of their local region of influence they go to zero. The setup is depicted in Figure 3.2. In this example, at each instant of time only four basis functions are nonzero, which we see results in a sparse factor graph.

The main difference, as compared to our earlier discrete-time development, is that we have coefficient variables instead of pose variables, but this is completely compatible with the general factor-graph approach. Now, when we observe a landmark, $\boldsymbol{\ell}$, at a particular time, t_i, the sensor model is

$$\boldsymbol{z}_i = \boldsymbol{h}_i\left(\boldsymbol{p}(t_i), \boldsymbol{\ell}\right) + \boldsymbol{\eta}_i. \tag{3.30}$$

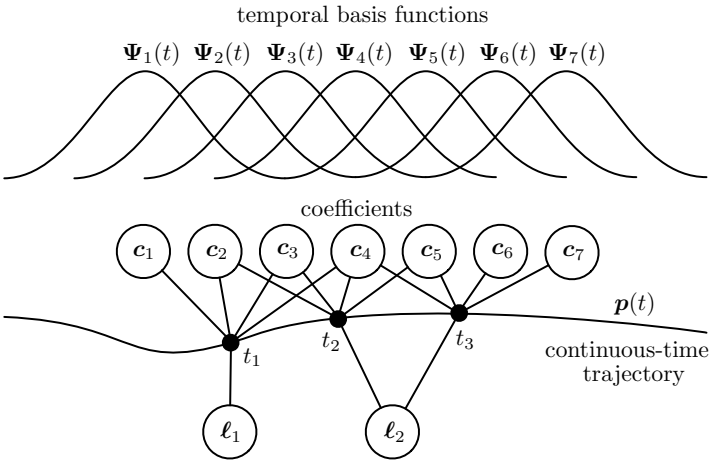


Figure 3.2 A parametric spline can be used to represent a continuous-time trajectory. In this example, the pose at a given time $\boldsymbol{p}(t)$ is assembled as a weighted sum of known temporal basis functions $\boldsymbol{\Psi}_k(t)$ with local support; at most four basis functions are nonzero at a given time. This results in each landmark measurement being represented by a *quinary* (five-way) factor between four coefficient variables and one landmark variable. The overall factor graph is still very sparse.

Inserting (3.29) we have

$$\boldsymbol{z}_i = \boldsymbol{h}_i \left(\sum_{k=1}^{K} \boldsymbol{\Psi}_k(t_i) \boldsymbol{c}_k, \boldsymbol{\ell} \right) + \boldsymbol{\eta}_i. \tag{3.31}$$

As mentioned above, if our basis functions are chosen to have local support, then only a small subset of the coefficients will be active at t_i. If we let $\boldsymbol{x}_i = \begin{bmatrix} \boldsymbol{c}_i^{\mathsf{T}} & \boldsymbol{\ell}^{\mathsf{T}} \end{bmatrix}^{\mathsf{T}}$ represent the active coefficient variables at t_i as well as the landmark variable, then we are back to being able to write the measurement function as

$$\boldsymbol{z}_i = \boldsymbol{h}_i \left(\boldsymbol{x}_i \right) + \boldsymbol{\eta}_i, \tag{3.32}$$

whereupon we can use our general approach to construct the nonlinear least-squares problem and optimize.

Moreover, if our basis functions are sufficiently differentiable, we can easily take the derivative of our pose trajectory,

$$\dot{\boldsymbol{p}}(t) = \sum_{k=1}^{K} \dot{\boldsymbol{\Psi}}_k(t) \boldsymbol{c}_k \tag{3.33}$$

so that we can handle sensor outputs that are functions of, say, velocity or even higher derivatives while still optimizing the same coefficient variables. We simply need to compute the derivatives of our basis functions, $\dot{\boldsymbol{\Psi}}_k(t)$.

Finally, once we have solved for the optimal coefficients through MAP inference, we can then query the trajectory (or its derivatives) at *any* time of interest using (3.29) or (3.33). If we compute the covariance of the estimated coefficients during inference (e.g., by inverting the information matrix), this can also be mapped through to covariance of a queried pose (or derivative) quite easily since (3.29) or (3.33) are linear relationships; and, local support in the basis functions implies only the appropriate marginal covariance is needed from the coefficients.

3.2.2 From Parametric to Nonparametric

The main challenge with basic parametric continuous-time methods is that we must decide what type and how many basis functions to use. If we have too many basis functions, it becomes very easy to overfit to the measurement data. If we have too few basis functions, we may not have sufficient capacity to represent the true shape of the trajectory, resulting in an overly smooth solution. This challenge is partly addressed by moving to a nonparametric method.

To simplify the explanation slightly, in this section we will assume for now that there are no landmark variables only pose variables. Using the parametric approach introduced in the previous section, our linearized least-squares term (negative-log factor) will have the form

$$\left\| \left(\boldsymbol{z}_i - \boldsymbol{h}_i \left(\boldsymbol{x}_i^0 \right) \right) - \boldsymbol{H}_i \boldsymbol{\Psi}_i \, \boldsymbol{\delta}_{c,i} \right\|_{\boldsymbol{\Sigma}_i}^2 , \tag{3.34}$$

where $\boldsymbol{x}_i^0$ is the current solution (active coefficients), $\boldsymbol{\delta}_{c,i}$ is the update (to the active coefficients), $\boldsymbol{\Psi}_i$ is the stacking of all basis functions active (and evaluted) at t_i, and the Jacobian, $\boldsymbol{H}_i$, is given by

$$\boldsymbol{H}_i = \left. \frac{\partial \boldsymbol{h}_i}{\partial \boldsymbol{x}} \right|_{\boldsymbol{x}_i^0} . \tag{3.35}$$

Gathering quantities into larger matrices as before, we can write our least-squares problem as

$$\boldsymbol{\delta}_c^* = \arg\min_{\boldsymbol{\delta}_c} \left(\left\| \boldsymbol{b} - \boldsymbol{A} \boldsymbol{\Psi} \boldsymbol{\delta}_c \right\|^2 + \left\| \boldsymbol{\delta}_c \right\|^2 \right), \tag{3.36}$$

where we now include a *regularizer term*, $\left\| \boldsymbol{\delta}_c \right\|^2$, that seeks to keep the *description length* of our solution reasonable (i.e., we prefer spline coefficients to be closer to zero). The regularizer term helps to avoid the over-fitting problem mentioned in the last section. The optimal solution will be given by

$$\left(\boldsymbol{\Psi}^\mathsf{T} \boldsymbol{A}^\mathsf{T} \boldsymbol{A} \boldsymbol{\Psi} + \mathbf{I} \right) \boldsymbol{\delta}_c^* = \boldsymbol{\Psi}^\mathsf{T} \boldsymbol{A}^\mathsf{T} \boldsymbol{b}, \tag{3.37}$$

which would allow us to compute the optimal update for the coefficients, $\boldsymbol{\delta}_c^*$. However, what we typically care about is to produce an estimate for the pose, not the spline coefficients (they are a means to an end). The optimal update to the

pose variables at the measurement times is actually $\boldsymbol{\delta}^* = \boldsymbol{\Psi}\boldsymbol{\delta}_c^*$. With a little bit of algebra, we can show that

$$\left(\boldsymbol{A}^\mathsf{T}\boldsymbol{A} + \boldsymbol{K}^{-1}\right)\boldsymbol{\delta}^* = \boldsymbol{A}^\mathsf{T}\boldsymbol{b}, \tag{3.38}$$

which is a modified version of the *normal equations*, first introduced in (2.25). The *kernel matrix*, $\boldsymbol{K} = \boldsymbol{\Psi}^\mathsf{T}\boldsymbol{\Psi}$, serves a regularization or smoothing function. The careful reader will notice that (3.38) represents a larger linear system of equations than (3.37) because there are more poses than basis function coefficients. However, in the end we will be able to reduce the size of the linear system we need to solve in our nonparametric approach by using built-in interpolation capabilities. For now, we will work with (3.38) and come back to this issue towards the end of the section.

To move away from explicit basis functions, we can employ the so-called *kernel trick*, which replaces the explicit inner product of basis functions with evaluations of a chosen *kernel function*, $\boldsymbol{\mathcal{K}}(t, t')$ (e.g., squared-exponential). We can see that in (3.38) it is only the *inner product* of the basis functions that is required to build the kernel matrix. The kernel matrix is then $\boldsymbol{K} = \left[\boldsymbol{\mathcal{K}}(t_i, t_j)\right]_{ij}$, which is to say we populate it with evaluations of the kernel function at every pairing of measurement times. We can now refer to this as a *nonparametric* method since we are no longer estimating the coefficients (i.e., parameters) of a spline. We do, however, have to tune the *hyperparameters* of our chosen kernel function (e.g., length scale for squared exponential) to achieve the desired trajectory smoothness.

Since we need the *inverse* kernel matrix right away in (3.38), it would seem to be expensive to formulate things this way. However, the next section shows how we can choose a kernel function that guarantees that we have a very sparse inverse kernel matrix and therefore a sparse factor graph.

3.2.3 Gaussian Processes

We will construct a family of kernel functions that by design results in a sparse inverse kernel matrix and corresponding factor graph. We saw in the last section that we could swap out our basis functions for a kernel function, creating a nonparametric continuous-time method. However, if done naively, this could result in a dense inverse kernel matrix, which is undesirable. In this section, we come at things from a slightly different direction. As a teaser, Figure 3.3 shows an example of a factor graph resulting from the ideas in this section, which we see remains sparse yet results in smooth trajectories.

We start by choosing a linear, time-invariant, stochastic differential equation (SDE) driven by white noise:[2]

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{A}\boldsymbol{x}(t) + \boldsymbol{L}\boldsymbol{w}(t), \tag{3.39}$$

[2] It is also possible to include control inputs in this equation but we omit them in the interest of simplicity.

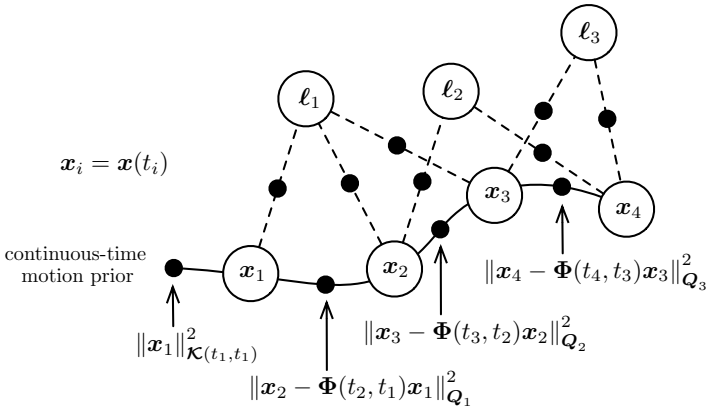


Figure 3.3 Example of Gaussian process (GP) continuous-time factor graph. The motion prior is based on a kernel function derived from a stochastic differential equation (SDE) for Markovian state $\boldsymbol{x}(t)$. This results in a very sparse set of factors: a single unary factor at the initial state and then binary factors linking consecutive states.

where $\boldsymbol{w}(t) = \mathcal{GP}(\boldsymbol{0}, \boldsymbol{\mathcal{Q}}\delta(t - t'))$ is a zero-mean white noise Gaussian process, $\boldsymbol{\mathcal{Q}}$ is a power-spectral density matrix, and $\delta(\cdot)$ is the Dirac delta function. The idea is that this will serve as a motion prior. We can integrate this SDE once in closed form:

$$\boldsymbol{x}(t) = \boldsymbol{\Phi}(t, t_1)\boldsymbol{x}(t_1) + \int_{t_1}^{t} \boldsymbol{\Phi}(t, s)\boldsymbol{L}\boldsymbol{w}(s)\, ds, \tag{3.40}$$

where $\boldsymbol{\Phi}(t, s) = \exp(\boldsymbol{A}(t - s))$ is known as the *transition function* and t_1 is the time stamp of the first measurement. The function, $\boldsymbol{x}(t)$, is also a Gaussian process. To keep the explanation simple, if we assume the mean of the initial state is zero, $E[\boldsymbol{x}(t_1)] = \boldsymbol{0}$, then the mean will remain zero for all subsequent times. The covariance function of the state (i.e., the kernel function), $\boldsymbol{\mathcal{K}}(t, t')$, can be calculated as

$$\boldsymbol{\mathcal{K}}(t, t') = \boldsymbol{\Phi}(t, t_1)\boldsymbol{\mathcal{K}}(t_1, t_1)\boldsymbol{\Phi}(t', t_1)^{\mathsf{T}} + \int_{t_1}^{\min(t,t')} \boldsymbol{\Phi}(t, s)\boldsymbol{L}\boldsymbol{\mathcal{Q}}\boldsymbol{L}^{\mathsf{T}}\boldsymbol{\Phi}(t', s)^{\mathsf{T}}\, ds, \tag{3.41}$$

which looks daunting. However, we can evaluate this kernel function at all pairs of measurement times (i.e., build the kernel matrix) using the tidy relation

$$\boldsymbol{K} = \boldsymbol{\Phi}\boldsymbol{Q}\boldsymbol{\Phi}^{\mathsf{T}}, \tag{3.42}$$

where $\boldsymbol{Q} = \operatorname{diag}(\boldsymbol{\mathcal{K}}(t_1, t_1), \boldsymbol{Q}_1, \ldots, \boldsymbol{Q}_M)$, $\boldsymbol{Q}_i = \int_{t_{i-1}}^{t_i} \boldsymbol{\Phi}(t_i, s) \boldsymbol{L} \boldsymbol{\mathcal{Q}} \boldsymbol{L}^{\mathsf{T}} \boldsymbol{\Phi}(t_i, s)^{\mathsf{T}} \, ds$, and

$$\boldsymbol{\Phi} = \begin{bmatrix} \mathbf{I} & & & & & \\ \boldsymbol{\Phi}(t_2, t_1) & \mathbf{I} & & & & \\ \boldsymbol{\Phi}(t_3, t_1) & \boldsymbol{\Phi}(t_3, t_2) & \mathbf{I} & & & \\ \vdots & \vdots & \vdots & \ddots & & \\ \boldsymbol{\Phi}(t_{M-1}, t_1) & \boldsymbol{\Phi}(t_{M-1}, t_2) & \boldsymbol{\Phi}(t_{M-1}, t_3) & \cdots & \mathbf{I} & \\ \boldsymbol{\Phi}(t_M, t_1) & \boldsymbol{\Phi}(t_M, t_2) & \boldsymbol{\Phi}(t_M, t_3) & \cdots & \boldsymbol{\Phi}(t_M, t_{M-1}) & \mathbf{I} \end{bmatrix}, \tag{3.43}$$

with M the last measurement time index. However, since it is the *inverse* kernel matrix that we want in (3.38), $\boldsymbol{K}^{-1} = \boldsymbol{\Phi}^{-\top} \boldsymbol{Q}^{-1} \boldsymbol{\Phi}^{-1}$, we can compute this directly. The middle matrix, $\boldsymbol{Q}$, is block-diagonal and so its inverse can be computed one diagonal block at a time. Importantly, when we compute the inverse of $\boldsymbol{\Phi}$, we find

$$\boldsymbol{\Phi}^{-1} = \begin{bmatrix} \mathbf{I} & & & & & \\ -\boldsymbol{\Phi}(t_2, t_1) & \mathbf{I} & & & & \\ & -\boldsymbol{\Phi}(t_3, t_2) & \mathbf{I} & & & \\ & & -\boldsymbol{\Phi}(t_4, t_3) & \ddots & & \\ & & & \ddots & \mathbf{I} & \\ & & & & -\boldsymbol{\Phi}(t_M, t_{M-1}) & \mathbf{I} \end{bmatrix}, \tag{3.44}$$

which is all zeros except for the main block-diagonal and one block-diagonal below. Thus, when we construct the inverse kernel matrix, $\boldsymbol{K}^{-1}$, it will be *block-tridiagonal*, for any length of trajectory. Based on our earlier discussions about factor graphs, we know that the sparsity of the left-hand side in (3.38) is closely tied to the factor-graph structure. In this case, $\boldsymbol{K}^{-1}$ serves as a motion prior over the entire trajectory, but it is easily described using a very sparse factor graph. Figure 3.3 shows how the block-tridiagonal structure of $\boldsymbol{K}^{-1}$ turns into a factor graph.

The reason $\boldsymbol{K}^{-1}$ has such a sparse factor graph is that we started from an SDE whose state, $\boldsymbol{x}(t)$, is *Markovian*. Practically speaking, what this means is that depending on the motion prior that we want to express using (3.39), we may need to use a higher-order state, i.e., not simply the pose but also some of its derivatives. For example, if we want to use the so-called 'constant-velocity' prior, our SDE can be chosen to be

$$\underbrace{\begin{bmatrix} \dot{\boldsymbol{p}}(t) \\ \dot{\boldsymbol{v}}(t) \end{bmatrix}}_{\dot{\boldsymbol{x}}(t)} = \underbrace{\begin{bmatrix} \mathbf{0} & \mathbf{I} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}}_{\boldsymbol{A}} \underbrace{\begin{bmatrix} \boldsymbol{p}(t) \\ \boldsymbol{v}(t) \end{bmatrix}}_{\boldsymbol{x}(t)} + \underbrace{\begin{bmatrix} \mathbf{0} \\ \mathbf{I} \end{bmatrix}}_{\boldsymbol{L}} \boldsymbol{w}(t), \tag{3.45}$$

where the state now comprises pose and its derivative, $\boldsymbol{v}(t) = \dot{\boldsymbol{p}}(t)$. Due to the use of this augmented state, this is sometimes this is referred to as *simultaneous trajectory estimation and mapping (STEAM)*, a variation of SLAM.

This formulation of continuous-time trajectory estimation is really an example of *Gaussian process regression* [217]. By making this connection, once we have solved

at the measurement times, we can easily query the trajectory at other times of interest using GP interpolation (for both mean and covariance); with our sparse kernel approach, the cost of each query is constant time with respect to the number of measurements, M, as it only involves the estimated states at the two times bracketing the query.

Importantly, we can also use the resulting GP interpolation scheme to reduce the number of control points needed (i.e., we do not need one at every measurement time), which is similar to the idea of *GP inducing points*. For example, we might put one control point per lidar scan but still make use of all the individual time stamps of each point gathered during a sweep. This last point is quite important because in contrast to discrete-time estimation, the measurement times, the estimation times, and the query times can now all be different in this continuous formulation. Moreover, in the GP approach we do not need to worry about overfitting by including too many estimation times as the kernel provides proper regularization. However, we still need enough estimation times to capture the detail of the trajectory.

3.2.4 Spline and GPs on Lie Groups

It is also possible to use both splines and GP continuous-time methods when the state lives on a manifold. In the case that the manifolds are Lie groups, both methods make use of the Lie algebra to accomplish this, but in different ways. We begin with splines and then move to Gaussian processes.

3.2.4.1 Splines on Lie Groups

The key to making splines work on Lie groups is to use a *cumulative formulation*. For a vector space, we can simplify (3.29) by assuming we are using the same basis functions for all degrees of freedom so that we can write

$$\boldsymbol{p}(t) = \sum_{k=1}^{K} \psi_k(t)\, \boldsymbol{p}_k, \tag{3.46}$$

where the $\boldsymbol{p}_k$ are now *control points* of our spline (replacing the earlier coefficients) and the basis functions, $\psi_k(t)$, are now scalar. Then, we can rewrite this in cumulative form as

$$\boldsymbol{p}(t) = \psi_1^c(t)\, \boldsymbol{p}_1 + \sum_{k=2}^{K} \psi_k^c(t)\, (\boldsymbol{p}_k - \boldsymbol{p}_{k-1})\,, \tag{3.47}$$

where

$$\psi_k^c(t) = \sum_{\ell=k}^{K} \psi_\ell(t) \tag{3.48}$$

are the *cumulative basis functions*.

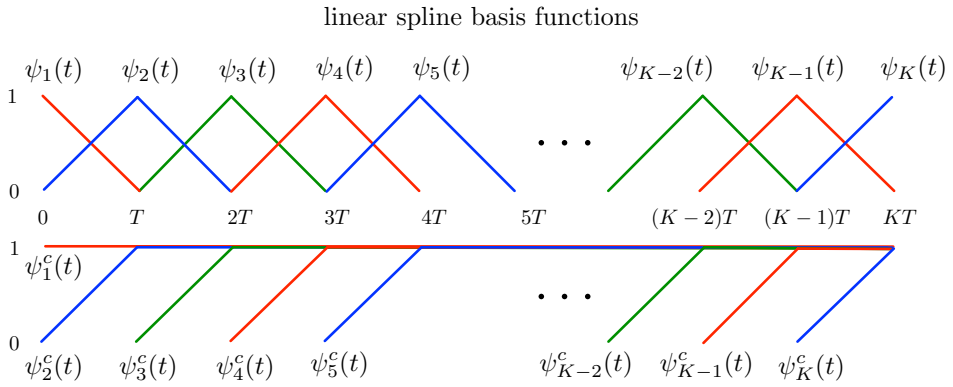


Figure 3.4 Example of linear spline basis functions both in (top) normal and (bottom) cumulative form.

For example, if we want to have *linear* interpolation with uniform temporal spacing, T, the basis functions are

$$\psi_1(t) = \begin{cases} 1-\alpha_1(t) & 0 \le t < T \\ 0 & \text{otherwise} \end{cases}, \quad \psi_K(t) = \begin{cases} \alpha_K(t) & (K-1)T \le t < KT \\ 0 & \text{otherwise} \end{cases}, \tag{3.49}$$

$$k = 2 \ldots K-1: \quad \psi_k(t) = \begin{cases} \alpha_{k-1}(t) & (k-2)T \le t < (k-1)T \\ 1-\alpha_k(t) & (k-1)T \le t < kT \\ 0 & \text{otherwise} \end{cases}, \tag{3.50}$$

where $\alpha_k(t) = \frac{t-(k-1)T}{T}$. The corresponding *cumulative* basis functions are

$$\psi_1^c(t) = 1, \quad \psi_K^c(t) = \begin{cases} 0 & \le t < (K-1)T \\ \alpha_k(t) & (k-1)T \le t \end{cases}, \tag{3.51}$$

$$k = 2 \ldots K-1: \quad \psi_k^c(t) = \begin{cases} 0 & t < (k-1)T \\ \alpha_k(t) & (k-1)T \le t < kT \\ 1 & kT \le t \end{cases}. \tag{3.52}$$

Figure 3.4 shows what these basis functions look like.

The key advantage of the cumulative basis functions is that at a given time stamp, most of the basis functions are inactive. In the case of our linear spline example, we can write

$$\boldsymbol{p}(t) = \boldsymbol{p}_{k-1} + \psi_k^c(t)\,(\boldsymbol{p}_k - \boldsymbol{p}_{k-1}) \tag{3.53}$$

when $(k-1)T \le t < kT$. We see that only a single basis function needs to be

evaluated. With higher-order splines, we will still have only a small active set at a particular time stamp.

To apply splines on a Lie group, the idea is to then use the cumulative formulation with the Lie group operator (matrix multiplication) replacing the summation. For example, in the case of a linear spline, an element of SE(d) can be written as

$$\boldsymbol{T}(t) = \operatorname{Exp}\left(\psi_k^c(t) \operatorname{Log}\left(\boldsymbol{T}_k \boldsymbol{T}_{k-1}^{-1}\right)\right) \cdot \boldsymbol{T}_{k-1}, \tag{3.54}$$

when $(k-1)T \leq t < kT$. We can now insert $\boldsymbol{T}(t_i)$ into any measurement expression at some time stamp t_i, linearize it with respect to the $\boldsymbol{T}_k$ control points (our estimation variables), and then use it within our MAP framework. Again, with compact-support basis functions, only a few are active at a given measurement time (one in the example of linear splines).

In a bit more detail for our linear spline example, we can rewrite (3.54) as

$$\boldsymbol{T}(t) = \left(\boldsymbol{T}_k \boldsymbol{T}_{k-1}^{-1}\right)^{\alpha_k(t)} \boldsymbol{T}_{k-1}. \tag{3.55}$$

When linearizing expressions involving $\boldsymbol{T}(t)$, we can make use of the optimization approach introduced in Section 3.1.3. We perturb each of the poses[3] so that

$$\operatorname{Exp}(\boldsymbol{\xi}(t))\boldsymbol{T}^0(t) = \left(\operatorname{Exp}(\boldsymbol{\xi}_k)\boldsymbol{T}_k^0\boldsymbol{T}_{k-1}^{0^{-1}}\operatorname{Exp}(-\boldsymbol{\xi}_{k-1})\right)^{\alpha_k(t)} \operatorname{Exp}(\boldsymbol{\xi}_{k-1})\boldsymbol{T}_{k-1}^0. \tag{3.56}$$

Our goal is to relate the perturbation of the interpolated pose, $\boldsymbol{\xi}(t)$, to those of the control points, $\boldsymbol{\xi}_k$ and $\boldsymbol{\xi}_{k-1}$. As shown by Barfoot [17], this relationship can be approximated (to first order in the perturbations) as

$$\boldsymbol{\xi}(t) \approx \left(\boldsymbol{I} - \boldsymbol{A}(\alpha_k(t))\right)\boldsymbol{\xi}_{k-1} + \boldsymbol{A}(\alpha_k(t))\,\boldsymbol{\xi}_k, \tag{3.57}$$

where

$$\boldsymbol{A}(\alpha_k(t)) = \alpha_k(t)\boldsymbol{J}\left(\alpha_k(t)\boldsymbol{T}_k^0\boldsymbol{T}_{k-1}^{0^{-1}}\right)\boldsymbol{J}\left(\boldsymbol{T}_k^0\boldsymbol{T}_{k-1}^{0^{-1}}\right)^{-1} \tag{3.58}$$

and $\boldsymbol{J}(\cdot)$ is the left Jacobian of SE(d). We can then use (3.57) to relate changes in our pose at a measurement time to the two bracketing control-point poses in order to form linearized error terms for use in MAP estimation. For example, consider the linearized measurement model in (3.15) again, where we rearrange it as an error with slightly simpler notation for the pose and its perturbation as a function of t:

$$\boldsymbol{e}_i(t) \approx \boldsymbol{z}_i - \boldsymbol{h}_i\left(\boldsymbol{T}^0(t)\tilde{\boldsymbol{\ell}}_i\right) - \boldsymbol{H}_i\boldsymbol{\xi}(t). \tag{3.59}$$

It is now a simple matter of substituting (3.57) in for $\boldsymbol{\xi}(t)$ to produce a linearized error in terms of the bracketing control points:

$$\boldsymbol{e}_i(t) \approx \boldsymbol{z}_i - \boldsymbol{h}_i\left(\boldsymbol{T}(t)^0\tilde{\boldsymbol{\ell}}_i\right) - \boldsymbol{H}_i\left(\boldsymbol{I} - \boldsymbol{A}(\alpha_k(t))\right)\boldsymbol{\xi}_{k-1} - \boldsymbol{H}_i\boldsymbol{A}(\alpha_k(t))\,\boldsymbol{\xi}_k. \tag{3.60}$$

[3] In this case, we are perturbing on the left side instead of the right as shown in Section 3.1.3. The reason is that if the unknown poses represent $\boldsymbol{T}_w^s(t)$ ('sensor' with respect to 'world'), we typically apply splines in the 'sensor' frame and so choose the perturbations to occur there as well.

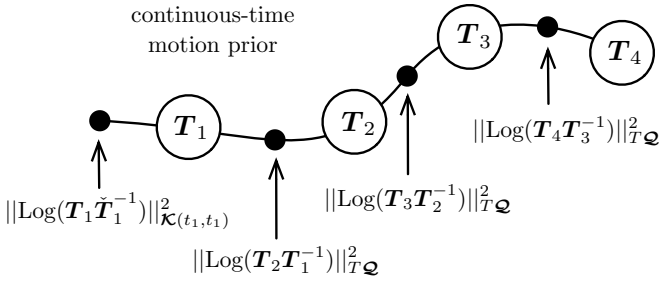


Figure 3.5 Example GP motion prior factors when using a 'random walk' model.

Note, we also need to substitute $\boldsymbol{T}^0(t) = \left(\boldsymbol{T}_k^0 \boldsymbol{T}_{k-1}^{0^{-1}}\right)^{\alpha_k(t)} \boldsymbol{T}_{k-1}^0$ for the nominal pose at t, both within $\boldsymbol{h}_i$ and $\boldsymbol{H}_i$. We have essentially chained the derivative through our linear spline. The same process can be followed for higher-order splines as well.

3.2.4.2 Gaussian Processes on Lie Groups

To use Gaussian processes on a Lie group, we will again exploit its Lie algebra to do so. Figure 3.5 provides a visual teaser of the GP motion-prior factors resulting from the ideas in this section. Note, as in the vector-space case, depending on the chosen motion prior, the control-point state may comprise additional trajectory derivatives as well.

To apply GPs on a Lie group, we will employ a local GP between a set of control-point states [17], similar to splines. Figure 3.6 provides a depiction of these local variables for SE(d). This means the SDE used to derive our kernel function operates on these local variables. For example, in the case of a 'random-walk' prior for SE(d),

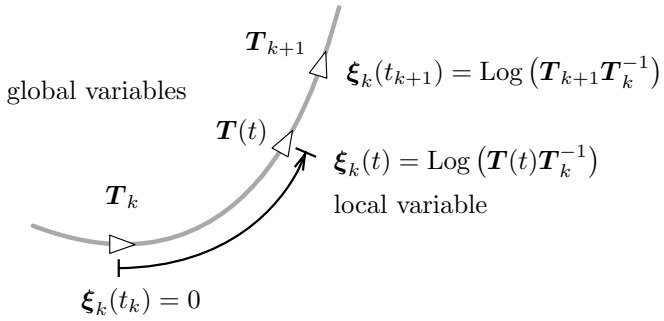


Figure 3.6 When using a GP for continuous-time estimation on Lie groups (e.g., SE(d)), a local variable, $\boldsymbol{\xi}_k(t)$, is defined between control-point states.

we could choose the SDE to be

$$\dot{\boldsymbol{\xi}}_k(t) = \boldsymbol{w}(t), \quad \boldsymbol{w}(t) = \mathcal{GP}(\mathbf{0}, \boldsymbol{\mathcal{Q}}\delta(t-t')), \tag{3.61}$$

where we note that we have defined it using the local variable (between control points $\boldsymbol{T}_k$ and $\boldsymbol{T}_{k+1}$). The transition function for this SDE is simply $\boldsymbol{\Phi}(t,s) = \mathbf{I}$ and so stochastically integrating we have

$$\boldsymbol{\xi}_k(t) = \underbrace{\boldsymbol{\xi}_k(t_k)}_{\mathbf{0}} + \int_{t_k}^{t} \boldsymbol{w}(s)\, ds \tag{3.62}$$

and then after taking the mean and covariance we can say that the motion prior is

$$\boldsymbol{\xi}_k(t) \sim \mathcal{GP}(\mathbf{0}, \min(t,t')\boldsymbol{\mathcal{Q}}). \tag{3.63}$$

If we place our control-point poses uniformly spaced every T seconds then our inverse kernel matrix will be simply $\boldsymbol{\mathcal{K}}^{-1} = \boldsymbol{\Phi}^{-\top}\boldsymbol{Q}^{-1}\boldsymbol{\Phi}^{-1}$ with

$$\boldsymbol{\Phi}^{-1} = \begin{bmatrix} \mathbf{I} & & & \\ -\mathbf{I} & \mathbf{I} & & \\ & \ddots & \ddots & \\ & & -\mathbf{I} & \mathbf{I} \end{bmatrix}, \quad \boldsymbol{Q} = \operatorname{diag}\left(\boldsymbol{\mathcal{K}}(t_1,t_1), T\boldsymbol{\mathcal{Q}}, \ldots, T\boldsymbol{\mathcal{Q}}\right). \tag{3.64}$$

The individual errors in terms of the local variables will be

$$\boldsymbol{e}_k = \begin{cases} \operatorname{Log}\left(\boldsymbol{T}_1 \check{\boldsymbol{T}}_1^{-1}\right) & k = 1 \\ \boldsymbol{\xi}_{k-1}(t_k) - \boldsymbol{\xi}_{k-1}(t_{k-1}) & k > 1 \end{cases}, \tag{3.65}$$

where $\check{\boldsymbol{T}}_1$ is some prior initial pose value. In terms of the global variables, these same errors are

$$\boldsymbol{e}_k = \begin{cases} \operatorname{Log}\left(\boldsymbol{T}_1 \check{\boldsymbol{T}}_1^{-1}\right) & k = 1 \\ \operatorname{Log}\left(\boldsymbol{T}_k \boldsymbol{T}_{k-1}^{-1}\right) & k > 1 \end{cases}. \tag{3.66}$$

Figure 3.5 shows what the 'random walk' GP motion prior looks like as a factor graph. Similarly to the previous section discussing linear splines, if we want to query the trajectory at other times of interest, we can do this using GP interpolation. For the 'random walk' prior, this results again in linear interpolation [17]:

$$\boldsymbol{T}(t) = \left(\boldsymbol{T}_k \boldsymbol{T}_{k-1}^{-1}\right)^{\alpha_k(t)} \boldsymbol{T}_{k-1}, \tag{3.67}$$

where $\alpha_k(t) = \frac{t-(k-1)T}{T}$ and $(k-1)T \leq t < kT$. In contrast to the spline method, this linear interpolation results indirectly from our choice of SDE at the beginning rather than an explicit choice. Choosing higher-order SDEs at the start will result in higher-order splines for interpolation.

The last part we need to understand is how to linearize our error terms for use in MAP estimation. To do this, we again make use of the Lie group perturbation

approach detailed earlier. For example, looking at the second case in (3.66) we can write

$$\begin{aligned}\boldsymbol{e}_k &= \operatorname{Log}\left(\operatorname{Exp}(\boldsymbol{\xi}_k)\boldsymbol{T}_k^0\boldsymbol{T}_{k-1}^{0^{-1}}\operatorname{Exp}(-\boldsymbol{\xi}_{k-1})\right) \\ &\approx \operatorname{Log}\left(\boldsymbol{T}_k^0\boldsymbol{T}_{k-1}^{0^{-1}}\right) + \boldsymbol{\xi}_k - \operatorname{Ad}\left(\boldsymbol{T}_k^0\boldsymbol{T}_{k-1}^{0^{-1}}\right)\boldsymbol{\xi}_{k-1}, \quad (3.68)\end{aligned}$$

where $\boldsymbol{T}_k^0$ and $\boldsymbol{T}_{k-1}^0$ are current guesses, $\boldsymbol{\xi}_k$ and $\boldsymbol{\xi}_{k-1}$ are the to-be-solved-for perturbations, and $\operatorname{Ad}(\cdot)$ is the adjoint for $\mathrm{SE}(d)$. This linearized form for $\boldsymbol{e}_k$ can be inserted in our standard MAP estimation framework at each iteration.

Additionally, if we want to use (3.67) to reduce the number of control points in this 'random walk' example, we can make use of the same approach developed for linear splines detailed in (3.60), since both methods boil down to linear interpolation between $\mathrm{SE}(d)$ control points. Ultimately, then, the big difference between the spline and GP approaches is that the GP approach employs motion-prior terms (see Figure 3.5) to regularize the problem, while the spline approach does not.[4]

[4] Johnson et al. [124] provide a detailed comparison between spline and GP approaches and shows that motion-prior terms can also be introduced to regularize spline methods.

4

Robustness to Incorrect Data Association and Outliers

Heng Yang, Josh Mangelson, Yun Chang, Jingnan Shi, and Luca Carlone

In Chapter 2, we have seen that factor graphs are a powerful representation to model and visualize SLAM problems, and that maximum a posteriori (MAP) estimation provides a grounded and general framework to infer variables of interest (*e.g.,* robot poses and landmark positions) given a set of measurements (*e.g.,* odometry and landmark measurements). For instance, we observed that when the measurements $\boldsymbol{z}_i$ are affected by additive and zero-mean Gaussian noise with covariance $\boldsymbol{\Sigma}_i$, MAP estimation leads to a *nonlinear least-squares* optimization:

$$\boldsymbol{x}^{\mathrm{MAP}} = \arg\min_{\boldsymbol{x}} \sum_i \|\boldsymbol{z}_i - \boldsymbol{h}_i(\boldsymbol{x}_i)\|^2_{\boldsymbol{\Sigma}_i}\,, \tag{4.1}$$

where $\boldsymbol{x}_i$ denotes the subset of the states involved in measurement i.[1] In this chapter we notice that in practice many measurements $\boldsymbol{z}_i$ —possibly due to incorrect data association— may have large errors, which are far from following a zero-mean Gaussian (Section 4.1); these measurements typically induce large perturbations in the estimate $\boldsymbol{x}^{\mathrm{MAP}}$ from eq. (4.1). Therefore, we discuss how to reject gross outliers in the SLAM front-end (Section 4.2) and then focus on how to increase robustness to remaining outliers in the SLAM back-end (Section 4.3). We close the chapter with a short review of recent trends and extra pointers to related work (Section 4.4).

4.1 What Causes Outliers and Why Are They a Problem?

This section argues that outliers are inevitable in most SLAM applications and that not handling them appropriately leads to grossly incorrect estimates.

4.1.1 Data Association and Outliers

To understand the cause of outlier measurements, let us consider two examples.

First, consider a landmark-based SLAM problem, where we have to reconstruct

[1] While for simplicity eq. (4.1) assumes that measurements belong to a vector space, the algorithms in this chapter apply to arbitrary SLAM problems where variables belong to manifolds, see Chapter 3.

the trajectory of the robot and the position of external landmarks from odometry measurements and relative observations of landmarks from certain robot poses. Assuming (as we did in Chapter 2) that the landmark measurements have zero-mean Gaussian noise leads to terms in the optimization in the form $\|\boldsymbol{z}_{ij} - \boldsymbol{h}(\boldsymbol{p}_i, \boldsymbol{\ell}_j)\|^2_{\boldsymbol{\Sigma}}$. These terms model the fact that a given measurement $\boldsymbol{z}_{ij}$ is an observation of landmark $\boldsymbol{\ell}_j$ from pose $\boldsymbol{p}_i$ up to Gaussian noise, where $\boldsymbol{h}(\cdot)$ is the function describing the type of relative measurement (*e.g.,* range, bearing, etc.). In practice, the measurements $\boldsymbol{z}_{ij}$ are obtained by pre-processing raw sensor data in the SLAM front-end. For instance, if the robot has an onboard camera and $\boldsymbol{z}_{ij}$ is a visual observation of the bearing to a landmark $\boldsymbol{\ell}_j$, the measurement $\boldsymbol{z}_{ij}$ might be extracted by performing object (or more generally, feature) detection and matching in the image, and then computing the bearing corresponding to the detected pixels. Now, the issue is that the detections are imperfect and a landmark detected as $\boldsymbol{\ell}_j$ in the image, might be actually a different landmark in reality. This causes $\boldsymbol{z}_{ij}$ to largely deviate from the assumed model. The problem of associating a measurement to a certain landmark is typically referred to as the *data association problem* and is common to many other estimation problems (*e.g.,* target tracking). Therefore, incorrect data association creates outliers in the estimation problem.

As a second example, consider a pose-graph optimization problem, where we are primarily interested in estimating the trajectory of the robot (represented as a set of poses), and the measurements are either odometry measurements (which relate consecutive poses along the trajectory) or *loop closures* (which relate non-consecutive and possibly temporally distant poses). In practice, the loop closures are detected using (vision-based or lidar-based) place recognition methods, which are in charge of detecting if a pair of poses $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ have observed the same portion of the environment. Unfortunately current place recognition methods are prone to making mistakes and detecting loop closures between poses that are *not* observing the same scene. This is partially due to limitations of current methods, but it is often due to *perceptual aliasing*, that is the situation where two similarly looking locations actually correspond to different locations (think of two classrooms in a university building, or similarly looking cubicles in an office environment). This can be again understood as a failure of data association, where we mistakenly associate the loop closure measurement to two incorrectly chosen robot poses.

Note that outliers are not only caused by incorrect data associations, but can also be caused by violations of the assumptions made in the SLAM approach. For instance, the majority of SLAM approaches assume landmarks to be static, hence detections of a moving object —even when correctly associated to that object— may lead to outlier measurements with large residuals. Similarly, sensor failure and degradation, *e.g.,* a faulty wheel encoder or dust on the camera lens, might contribute to creating outliers in the measurements.

4.1.2 Least-Squares in the Presence of Outliers

In the presence of outliers, the estimate resulting from the least-squares formulation (4.1) can be grossly incorrect. From the theoretical standpoint, the Gaussian noise we assumed for the measurement is "light-tailed", in that it essentially rules out the possibility of measurements with very large error. From a more practical perspective, the outliers lead to terms in the objective function where the residual error $\boldsymbol{r}_i(\boldsymbol{x}) := \|\boldsymbol{z}_{ij} - \boldsymbol{h}(\boldsymbol{p}_i, \boldsymbol{\ell}_j)\|_{\boldsymbol{\Sigma}}$ is very large, when evaluated near the ground truth. Since the residuals are squared in the objective of the optimization, *i.e.,* the objective is $\sum_i \boldsymbol{r}_i(\boldsymbol{x})^2$, these residuals have a disproportionately large impact on the cost, and the optimization focuses on minimizing the large terms induced by the outliers rather than making good use of the remaining (inlier) measurements.

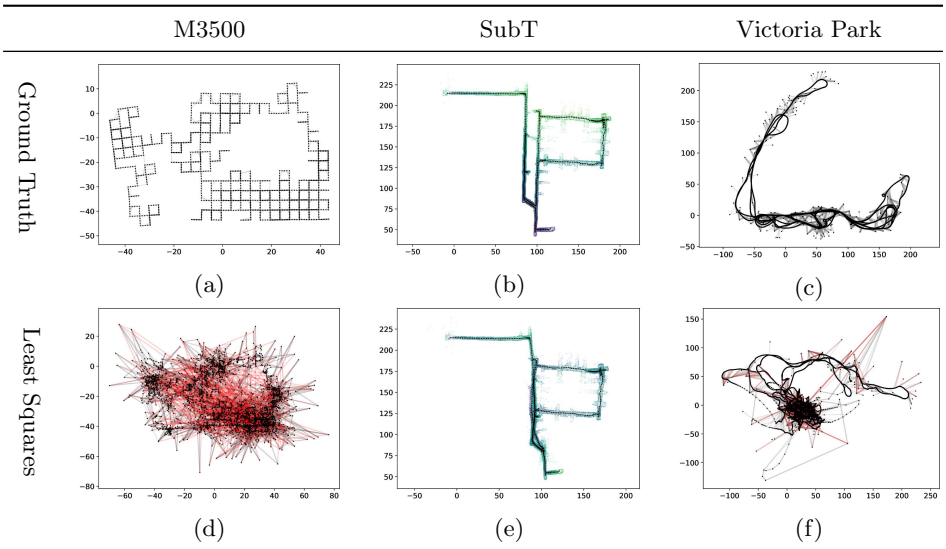


Figure 4.1 SLAM problems with outliers: (a)-(c) Ground truth trajectories for the M3500, SubT, and Victoria Park datasets. (d)-(f) Trajectory estimates obtained with the least-squares formulation in the presence of outliers. Inlier measurements are visualized as gray edges, while outliers are visualized as red edges. In the SubT dataset, we also visualize a dense map built from the SLAM pose estimate.

To illustrate this point, Figure 4.1 shows results for three SLAM problems with outliers. The first column is a simulated pose-graph optimization benchmark, known as M3500, with poses arranged in a grid-like configuration; the dataset includes 3500 2D poses and 8953 measurements. The second column is a real-world pose-graph dataset, denoted as SubT, collected in a tunnel during the DARPA Subterranean Challenge [82]; the dataset includes 682 3D poses and 3278 measurements. The third column is a real-world landmark-SLAM dataset, known as Victoria Park [187]; the dataset includes 7120 2D poses and landmarks, and 17728 measurements. Figure

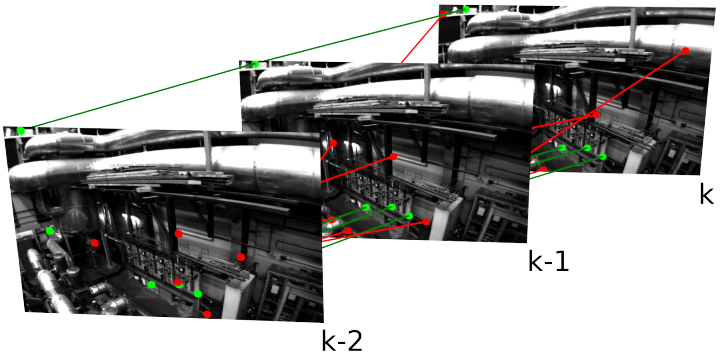


Figure 4.2 Feature tracking across three frames (collected at time $k-1$, k, and $k+1$) in a visual SLAM problem. Inliers are visualized in green, while outliers are visualized in red.

4.1(a)-(c) show the ground truth trajectories for the three problems. Figure 4.1 (d)-(f) show the estimate produced by the least-squares formulation in the presence of outliers. In particular, for M3500 and Victoria Park we add 15% random outliers to the (loop closures or landmark) measurements, while the SubT dataset already includes outliers. In the figure, we visualize outlier measurements in red. We observe that the presence of outliers leads to completely incorrect trajectories and map estimates. Moreover, the outliers often expose perceptual aliasing in the environment: for instance, the two similarly looking vertical corridors in the middle of the SubT dataset induce many spurious loop closures, which mislead the back-end to create a map with a single vertical corridor.

4.2 Detecting and Rejecting Outliers in the SLAM Front-end

The main role of the SLAM front-end is to extract intermediate representations or (pseudo-)measurements —which will be converted into factors for the back-end— from the raw sensor data. Typical SLAM front-ends accomplish this by first computing an initial set of measurements (possibly corrupted by many outliers) and then post-processing the initial set to remove outliers. This section discusses two approaches to reject outliers in the SLAM front-end.

4.2.1 RANdom SAmple Consensus (RANSAC)

RANSAC is a well-established tool for outlier rejection [89] and is a key component of many landmark-based SLAM systems. In order to understand what RANSAC is and its role in SLAM, consider a landmark-based visual SLAM approach.

Example 4.1 (Outliers in landmark-based visual SLAM)**.** A landmark-based (or

feature-based) visual-SLAM approach extracts 2D feature points in each image and then associates them across consecutive frames using either optical-flow-based feature tracking or descriptor-based feature matching (Figure 4.2). In particular, at time k, the approach detects 2D feature points and matches them with corresponding points observed in the previous frame (say, at time $k-1$); the matching pixels are typically referred to as *2D-2D correspondences*. Due to inaccuracies of optical flow or descriptor-based matching, this initial set of correspondences might contain outliers. Therefore, it is important to filter out gross outliers before passing them to the back-end, which estimates the robot poses and landmark positions.

RANSAC is a tool to quickly detect and remove outliers in the correspondences before passing them to the back-end. Detecting outliers relies on two key insights. The first insight is that in SLAM problems, inlier correspondences must satisfy geometric constraints. For instance, in our example, inlier correspondences picture the observed pixel motion of static 3D points as the camera moves. The resulting pixel motion cannot be arbitrary, but must follow a precise geometric constraint, known as the *epipolar constraint*, which dictates how corresponding pixels in two frames are related depending on the camera motion. In particular, for calibrated cameras, the epipolar constraint imposes that corresponding pixels $\mathbf{z}_i(k-1), \mathbf{z}_i(k)$ —picturing landmark i at time $k-1$ and k, respectively— satisfy

$$\mathbf{z}_i(k-1)^\mathsf{T} \left([\boldsymbol{t}_k^{k-1}]_\times \boldsymbol{R}_k^{k-1}\right) \mathbf{z}_i(k) = 0, \tag{4.2}$$

where $\boldsymbol{t}_k^{k-1}$ and $\boldsymbol{R}_k^{k-1}$ are the relative position and rotation describing the (unknown) motion of the camera between time $k-1$ and k.[2] More generally, if we denote the i-th correspondence as $\mathbf{z}_i$ (in the example above, $\mathbf{z}_i = \{\mathbf{z}_i(k-1), \mathbf{z}_i(k)\}$), these geometric constraints are in the form

$$C(\mathbf{z}_i, \boldsymbol{x}) \leq \gamma, \tag{4.3}$$

which states that the correspondences have to satisfy some inequality, which is possibly a function of the unknown state $\boldsymbol{x}$; in (4.3) the parameter γ on the right-hand-side is typically tuned to account for the presence of noise. For instance, while ideally the epipolar constraint in (4.2) is exactly satisfied, in practice it might have small errors since the pixel detections are noisy, and hence we would relax the constraint to only require $\left|\mathbf{z}_i(k-1)^\mathsf{T} \left([\boldsymbol{t}_k^{k-1}]_\times \boldsymbol{R}_k^{k-1}\right) \mathbf{z}_i(k)\right| \leq \gamma$, for some small γ.

The second insight is that —assuming we do not have too many outliers— we can find the inliers as the largest set of correspondences that satisfy the geometric

[2] Clearly, different problems will have different geometric constraints, but luckily there is a well-established literature in robotics and computer vision, that studies geometric constraints induced by different types of sensor measurements. The example in this section considers 2D-2D correspondences, and the corresponding constraints have been studied in the context of 2-view geometry in computer vision, see [111].

constraint (4.3) for some $\boldsymbol{x}$:

$$\begin{aligned} \mathsf{S}^*_{\mathrm{CM}} &= \underset{\boldsymbol{x}, \mathsf{S} \subset \mathsf{M}}{\operatorname{argmax}} |\mathsf{S}| \\ \text{s.t.} \;\; & C(\mathbf{z}_i, \boldsymbol{x}) \leq \gamma, \quad \forall i \in \mathsf{S} \end{aligned} \tag{4.4}$$

where M is the set of initial putative correspondences, and $|\mathsf{S}|$ denotes the cardinality (number of elements) in the subset S [169]. In words, the optimization (4.4) looks for the largest subset S of the set of putative correspondences M, such that measurements in S satisfy the geometric constraints for the same value of $\boldsymbol{x}$. Intuitively, problem (4.4) captures the intuition that the inliers (estimated by the set S) must "agree" on the same $\boldsymbol{x}$ (*e.g.,* they must all be consistent with the actual motion of the robot). Problem (4.4) is known as *consensus maximization* in computer vision. Note that (4.4) does not require solving the entire SLAM problem (which might involve many poses and landmarks), since it only involves a small portion of the SLAM state; for instance, the epipolar constraint (4.2) only involves the relative pose between two frames rather than the entire SLAM trajectory. At the same time, (4.4) is still a hard combinatorial problem, which clashes with the fast runtime requirements of typical SLAM front-ends. Therefore, rather than looking for exact solutions to (4.4), it is common to resort to quick heuristics to approximately solve (4.4).

RANdom SAmple Consensus (RANSAC) is probably the most well-known approach to find an approximate solution to the consensus maximization problem in (4.4). RANSAC builds on the key assumption that $\boldsymbol{x}$ in (4.4) is relatively low-dimensional and can be estimated from a small set of measurements (the so-called *minimal set*), using fast estimators (the so called *minimal solvers*).[3] For instance, in our visual SLAM example, one can estimate the relative motion between two camera frames using only 5 pixel correspondences, using Nister's 5-point method [188]. Then the key idea behind RANSAC is that, instead of exhaustively checking every possible subset $\mathsf{S} \subset \mathsf{M}$, one can *sample* minimal sets of measurements looking for inliers. More in detail, RANSAC iterates the following three steps:

1 Sample a subset of n correspondences, where n is the size of the minimal set for the problem at hand;[4]
2 Compute an estimate $\hat{\boldsymbol{x}}$ from the n sampled correspondences using a minimal solver;[5]
3 Select the correspondences $\mathsf{S} \subset \mathsf{M}$ that satisfy the geometric constraint $C(\mathbf{z}_i, \hat{\boldsymbol{x}}) \leq$

[3] The development of minimal solvers can be considered a sub-area of computer vision research, hence for typical problems it is well-understood what is the size of the minimal set and there are well-developed (and typically off-the-shelf) minimal solvers one can use.
[4] In our example with pixel correspondences between calibrated camera images, the minimal set has size $n = 5$, since 5 non-collinear measurements are sufficient to determine the pose between two cameras up to scale.
[5] In our example, this involves computing the relative motion (up to scale) between time $k-1$ and k using the 5-point method.

γ for the $\hat{\boldsymbol{x}}$ computed at the previous step. Store the set S if it is larger than the set computed at the previous iterations.

The set S computed in the last step is called the *consensus set* and RANSAC typically stops after computing a sufficiently large consensus set (as specified by a user parameter) or after a maximum number of iterations. RANSAC essentially attempts to sample n inliers from the set of measurements, since these are likely to "agree" with all the other inliers and hence have a large consensus set.

RANSAC is the go-to solution for many outlier-rejection problems. In particular, it quickly converges to good estimates (*i.e.,* good sets of correspondences) in problems with small number of outliers and small minimal sets. Assuming that the probability of sampling an inlier from the set of measurements is ω,[6] it is easy to conclude that the expected number of iterations RANSAC requires for finding a set of inliers is $\frac{1}{\omega^n}$. For instance, when $n = 5$ and $\omega = 0.7$ (*i.e.,* 70% of the measurements are inliers), the expected number of iterations is less then 10. This, combined with the fact that non-minimal solvers are extremely fast in practice (allowing even thousand of iterations in a handful of milliseconds), makes RANSAC extremely appealing. Moreover, RANSAC also provides an estimate $\hat{\boldsymbol{x}}$ (*e.g.,* the robot odometry), that can be useful as an initial guess for the back-end.

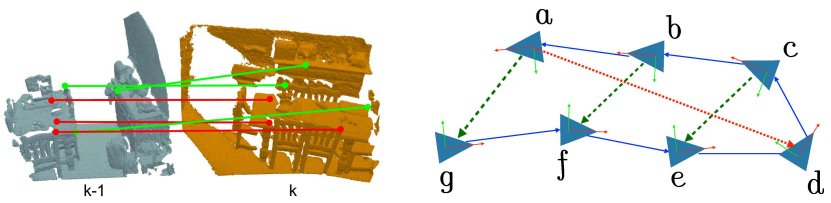
On the downside, RANSAC may not be the right approach for all problems. In particular, the expected number of iterations becomes impractically large when the number of inliers is small or when the minimal set is large; for instance, when $n = 10$ and $\omega = 0.1$, the expected number of iterations to find a set of inliers becomes 10^{10}, and terminating RANSAC after a smaller number of iterations is likely to return incorrect solutions (*i.e.,* incorrect $\hat{\boldsymbol{x}}$ and correspondences). As we discuss in the next section, in context of many SLAM problems, the assumptions of having many inliers and small minimal sets are not always valid.

4.2.2 Graph-theoretic Outlier Rejection and Pairwise Consistency Maximization

As we mentioned, RANSAC is very effective when the number of outliers is reasonable (say, below 70%) and the size of the minimal set is small (say, less than 8). However, environments with severe perceptual aliasing might have very high number of outliers. Moreover, not all the problems we are interested in have a fast minimal solver with a small minimal set. For instance, if we consider a pose-graph SLAM problem with N nodes, the minimal set must include at least $N - 1$ measurements (forming a spanning tree of the pose-graph), and N is typically in the thousands.

For these reasons, this section introduces an alternative approach, known as *pair-*

[6] Assuming that samples are drawn uniformly at random, ω can be thought of as the fraction of measurements that are inliers.



(a) (b)

Figure 4.3 (a) 3D-3D correspondences from two RGB-D scans representing two partial views of a scene. The green lines indicate inlier correspondences and the red lines indicate outlier correspondences. (b) Pose graph with outliers in the loop closures. The dashed green lines indicate inlier loop closures while the dotted red line is an outlier loop closure.

wise consistency maximization (PCM), that, rather than sampling minimal sets, seeks to find the largest set of measurements that are internally "consistent" with one another, using graph theory. This approach can be used to sort through sets of measurements with upwards of 90% outliers and prune gross outliers before passing them to the back-end. The approach was initially proposed in [169] and extended beyond pairwise consistency in [236, 91].

The key insight behind PCM is that for many problems one can define *consistency functions* that capture whether a pair of measurements are consistent with each other. Let's elucidate on this point with two examples.

Example 4.2 (Consistency Function in landmark-based visual SLAM with RGB-D cameras)**.** A landmark-based visual-SLAM approach with RGB-D cameras extracts 3D feature points in each RGB-D frame and then associates them across consecutive frames (Figure 4.3(a)). In particular, at time k, the approach detects 3D feature points and matches them with corresponding points observed in the previous frame (say, at time $k-1$); the matching 3D points are typically referred to as *3D-3D correspondences*. We observe that the 3D points collected at time k and $k-1$ ideally correspond to the same set of 3D static points observed from two different viewpoints; therefore, the distance between a pair of corresponding points $\{\boldsymbol{z}_i(k-1), \boldsymbol{z}_j(k-1)\}$ and $\{\boldsymbol{z}_i(k), \boldsymbol{z}_j(k)\}$ has to be constant over time (up to noise):

$$|\|\boldsymbol{z}_i(k-1) - \boldsymbol{z}_j(k-1)\| - \|\boldsymbol{z}_i(k) - \boldsymbol{z}_j(k)\|| \leq \gamma \tag{4.5}$$

We observe that contrary to the geometric constraints used in RANSAC, the consistency function (4.5) (i) does not depend on the state, hence it can be evaluated directly without the need for a minimal solver, and (ii) involves a pair of correspondences regardless of the size of the minimal set. While the previous example could also be solved with RANSAC,[7] let us now consider a higher dimensional problem.

[7] Motion estimation from 3D-3D correspondences admits a fast 3-point minimal solver, *e.g.,* Horn's method [115].

Example 4.3 (Consistency Function in pose-graph SLAM)**.** Consider a pose-graph SLAM problem where loop closures might contain outliers due to place recognition failure and perceptual aliasing; we assume the odometry is reliable and outlier free. In order to understand if two loop closures are consistent with each other, we observe that in the noiseless case, pose measurements along cycles in the graph must compose to the identity (Figure 4.3(b)).[8] Therefore, a pair of loop closures $\boldsymbol{T}_{ab}$ (between poses a and b) and $\boldsymbol{T}_{cd}$ (between poses c and d) must satisfy:

$$\text{dist}(\boldsymbol{T}_{ab} \cdot \bar{\boldsymbol{T}}_{bc} \cdot \boldsymbol{T}_{cd} \cdot \bar{\boldsymbol{T}}_{da}, \mathbf{I}) \leq \gamma \tag{4.6}$$

where $\bar{\boldsymbol{T}}_{bc}$ and $\bar{\boldsymbol{T}}_{da}$ are the chain of odometry measurements from node b to node c, and from node d to node a, respectively, and dist is a suitable distance function that measures how far is $\boldsymbol{T}_{ab} \cdot \bar{\boldsymbol{T}}_{bc} \cdot \boldsymbol{T}_{cd} \cdot \bar{\boldsymbol{T}}_{da}$ from the identity pose. As usual, γ is a parameter chosen to account for the noise: measurements along a loop might not compose to the identity due to noise in the odometry and loop closures.[9]

More generally, a *consistency function* is a function relating two measurements and that have to satisfy a given constraint. For a pair of measurements $\mathbf{z}_i$ and $\mathbf{z}_j$, the resulting *pairwise consistency constraints* are in the form:

$$F(\mathbf{z}_i, \mathbf{z}_j) \leq \gamma, \tag{4.7}$$

where F is the consistency function, and γ is a user-specified parameter that accounts for measurement noise. We remark that the pairwise consistency constraint are state independent, hence they can be efficiently checked without resorting to a minimal solver by just inspecting every pair of measurements.

Using (4.7), we can formulate an alternative approach for outlier rejection, which selects the largest set of measurements that are pairwise consistent:

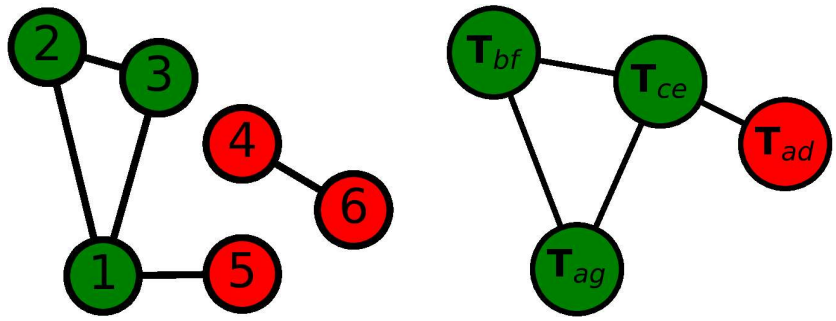
$$\begin{aligned} \mathsf{S}^*_{\text{PCM}} &= \underset{\mathsf{S} \subset \mathsf{M}}{\text{argmax}} \, |\mathsf{S}| \\ \text{s.t.} \quad & F(\mathbf{z}_i, \mathbf{z}_j) \leq \gamma, \quad \forall i, j \in \mathsf{S} \end{aligned} \tag{4.8}$$

Problem (4.8) looks for the largest subset S of measurements such that every pair of measurements in S are pairwise consistent. We refer to this as the *pairwise consistency maximization* (PCM) problem. This problem is still combinatorial in nature, but appears slightly easier than (4.4): the problem does not involve $\boldsymbol{x}$, and the constraints $F(\mathbf{z}_i, \mathbf{z}_j) \leq \gamma$ can be pre-computed for every pair (i, j) in M. Furthermore, the problem admits a graph-theoretic interpretation, which allows solving (4.4) using well-established tools from graph theory, namely, *maximum clique* algorithms.

In order to draw a connection between problem (4.8) and graph theory, let us visualize the outlier-rejection problem as a graph $\mathcal{G}$, where the nodes of the graph are the putative measurements $i \in \mathsf{M}$ and an edge exists between two nodes i and j if

[8] Intuitively, if we walk back along a loop in the environment we come back to our initial location.

[9] In practice, one would select γ to account for the size of the loop: intuitively, longer loops will accumulate more noise, see [169, 81].



(a) (b)

Figure 4.4 (a) Consistency graph of the 3D-3D correspondences example in Figure 4.3(a). (b) Consistency graph of the loop closures for the pose-graph example in Figure 4.3(b)

$F(\mathbf{z}_i, \mathbf{z}_j) \leq \gamma$. This is typically called the *consistency graph*. Now problem (4.8) asks to select the largest subset of nodes S such that every pair of nodes in S is connected by an edge: this is exactly the definition of *maximum clique* of a graph. More in detail, a *clique* in graph theory is a subset of nodes in which every pair of nodes has an edge between them, and the *maximum clique* is the largest such subset of nodes in the graph. Therefore, the solution to problem (4.8) is the maximum clique of the consistency graph $\mathcal{G}$. This graph theoretic connection is really useful in practice, since the problem of finding the maximum clique for a given graph is a well-studied problem in graph theory and is called the maximum clique problem. The maximum clique problem is an NP-hard problem [290] and hard to approximate [313, 88], meaning that finding a solution arbitrarily close to the true solution is also NP-hard. However, dozens of potential solutions have been proposed, some of which can handle significantly sized problems depending on the density of the graph. The majority of proposed methods can be classified as either exact or heuristic-based methods. All of the exact algorithms are exponential in complexity and are usually based on branch and bound, while the heuristic algorithms often try to exploit some type of structure in the problem, making them faster, at the expense of not necessarily guaranteeing the optimal solution [290]. Relatively recent works, *e.g.,* [203], propose maximum clique algorithms that are parallelizable and able to quickly find maximum cliques in sparse graphs.

In summary, solving the PCM problem using a maximum clique algorithm involves the following steps:

1 Select a Consistency Function F for the problem at hand.
2 Evaluate the Consistency Function for every pair of putative measurements $(i, j) \in \mathsf{M}$, and create a consistency graph with edges between i and j when $F(\mathbf{z}_i, \mathbf{z}_j) \leq \gamma$.
3 Solve for the Maximum Clique of the Consistency Graph using exact or approximate maximum clique algorithms.

4 Return measurements S in the (possibly approximate) maximum clique.

We remark that the choice of consistency function is problem-dependent. Moreover, choosing a good consistency function might largely influence the quality of the outlier rejection. For instance, one could select a dummy function $F(\mathbf{z}_i, \mathbf{z}_j) = 0$ which always returns zero regardless of the arguments; such a function would not allow rejecting any outliers, hence making PCM ineffective. On the other hand, if we make the function such that only the inliers can pass the test, then we would exactly reject all the outliers. A selection of potential consistency functions for broad variety of geometric problems is discussed in [236, 91].

Before concluding this section a few remarks are in order. While we observed that PCM has the ability to handle a large number of outliers compared to RANSAC and is more suitable for certain problems (*e.g.,* pose-graph SLAM), the trade-offs between PCM and RANSAC are more nuanced. RANSAC evaluates the consistency of individual measurements using an estimate computed by a minimal solver; PCM, on the other hand, evaluates the consistency of a set of measurements to each other in a pairwise manner. In certain cases, RANSAC's individual consistency is insufficient to evaluate the set of measurements as a whole: this is often the case in pose-graph optimization where individual consistency of a pair of loop-closure measurements does not necessarily ensure pairwise consistency of the two loop-closures.[10] On the other hand, for certain problems such as 3D-3D pose estimation (Example 4.2), the pairwise consistency function (4.5) used in PCM might be more permissive then RANSAC and lead to classifying certain outliers as inliers.

In the context of PCM, it is also important to note that exact maximum clique solvers tend to be slow in dense consistency graphs (*i.e.,* when many pairs of measurements are consistent), hence heuristic-based maximum-clique solutions may be a better choice for certain problems. Finally, for certain problems, it might be hard to design a suitable consistency function; for instance, for 2D-2D correspondences, there is no easy way to rigorously design a general consistency function due to the lack of suitable invariances (see discussion in [236]).

4.3 Increasing Robustness to Outliers in the SLAM Back-end

Front-end outlier rejection, including both RANSAC and PCM, might still miss outliers and pass an outlier-contaminated set of measurements to the back-end.[11] As we have seen in Section 4.1.2, a handful of outliers can lead to completely wrong

[10] This is especially pronounced in the context of the multi-robot pose-graph optimization — where the goal is to estimate the trajectory of two (or more) robots jointly within a single pose-graph. In these contexts in particular, PCM has been shown to dramatically outperform RANSAC [169].

[11] Intuitively, both the geometric constraints (4.3) —even when evaluated at the ground truth $\boldsymbol{x}$— and the pairwise consistency constraints (4.7) are necessary (but not sufficient) conditions for measurements to be inliers. Moreover, consensus maximization and PCM are often solved with approximation algorithms that do not guarantee an optimal selection of the inliers.

results when using standard least squares estimation. Therefore, it is important to enhance the back-end to be robust to remaining outliers.

In Section 4.1.2, we observed that the use of squared residuals "amplifies" the impact of outlying measurements on the cost function. In this section we slightly modify the objective function in the SLAM optimization to regain robustness to outliers, following the standard theory of M-Estimation in robust statistics [117].

M-Estimation ("Maximum-likelihood-type Estimation") is a framework for robust estimation and suggests replacing the squared loss in eq. (4.1) with a suitably chosen *robust loss* function ρ:

$$\boldsymbol{x}^{\mathrm{MAP}} = \arg\min_{\boldsymbol{x}} \sum_i \boldsymbol{r}_i(\boldsymbol{x})^2 \quad \Longrightarrow \quad \boldsymbol{x}^{\mathrm{MEST}} = \arg\min_{\boldsymbol{x}} \sum_i \rho\left(\boldsymbol{r}_i(\boldsymbol{x})\right). \tag{4.9}$$

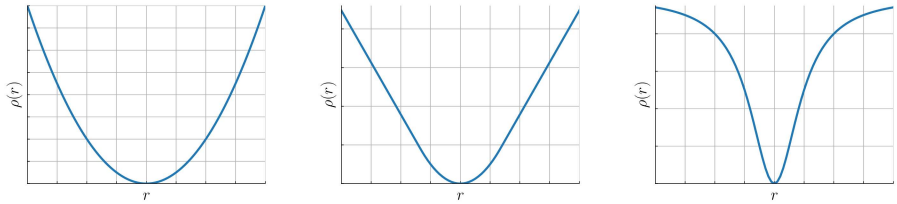
The key requirement for the robust loss ρ is to be a non-negative function and grow less than quadratically for large residuals; in other words, robust loss functions need to have derivative $\frac{\partial \rho(\boldsymbol{r}_i)}{\partial \boldsymbol{r}_i} \ll \frac{\partial \|\boldsymbol{r}_i\|^2}{\partial \boldsymbol{r}_i} = 2\boldsymbol{r}_i$ as $\boldsymbol{r}_i$ becomes large; in many cases, it is desirable for $\frac{\partial \rho(\boldsymbol{r}_i)}{\partial \boldsymbol{r}_i}$ to approach zero as $\boldsymbol{r}_i$ becomes large. To elucidate this requirement, consider the case where we solve $\min_{\boldsymbol{x}} \sum_i \rho\left(\boldsymbol{r}_i(\boldsymbol{x})\right)$ using gradient descent. Using the chain rule, the gradient of the objective $f(\boldsymbol{x}) \doteq \sum_i \rho\left(\boldsymbol{r}_i(\boldsymbol{x})\right)$ becomes:

$$\frac{\partial f}{\partial \boldsymbol{x}} = \sum_i \frac{\partial \rho\left(\boldsymbol{r}_i\right)}{\partial \boldsymbol{r}_i} \cdot \frac{\partial \boldsymbol{r}_i(\boldsymbol{x})}{\partial \boldsymbol{x}} \tag{4.10}$$

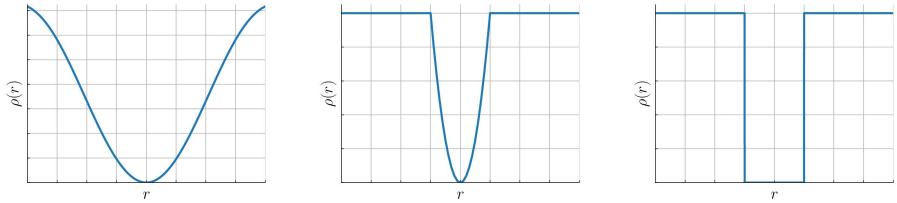
From (4.10), it is clear that if we start for a good initial guess (*i.e.,* relatively close to the ground truth), outlier measurements will have large residual and hence very small $\frac{\partial \rho(\boldsymbol{r}_i)}{\partial \boldsymbol{r}_i}$, thus having a minor influence on the overall descent direction. Hence they will have almost no influence in the estimation. Indeed, the function $\psi(\boldsymbol{r}_i) := \frac{\partial \rho(\boldsymbol{r}_i)}{\partial \boldsymbol{r}_i}$ is typically referred to as the *influence function* [31].

Rather than a single choice of robust loss function, the robust estimation literature provides a "menu" of potential choices. Figure 4.5 lists common choices of loss functions. This list includes common robust losses, such as Huber, Geman-McClure, Tukey's biweight and the truncated quadratic loss, and also includes a more radical choice, named *maximum consensus* loss. The latter is not typically listed among the loss functions in the robust estimation literature, but we mention it here, since it connects back to the consensus maximization problem we discussed in (4.4).[12] The choice of robust loss is fairly problem-dependent [133, 253]. For instance, loss functions with hard cut-offs (*e.g.,* the truncated quadratic loss, where there is a sudden transition between the quadratic and the "flat" portion of the function) are preferable when a reasonable threshold for the cut-off (*i.e.,* the maximum error expected from the inliers) is known. One also has to take into account computational

[12] Intuitively, the maximum consensus loss "counts" the outliers in the estimation problem, since it is constant (typically equal to 1) for large residuals and zero for small residuals. Therefore, minimizing such loss leads to an estimate that produces the least number of outliers. This is the same as solving the consensus maximization problem in (4.4).



(a) Quadratic loss (b) Huber loss (c) Geman-McClure



(d) Tukey's biweight (e) Truncated Quadratic (f) Maximum Consensus loss

Figure 4.5 Quadratic loss and examples of robust loss functions. The shape of the robust loss functions is controlled by a parameter that controls the separation between inliers and outliers.

considerations. For instance, Huber is often used in bundle adjustment problems since it is a convex function and is better-behaved during the optimization, despite leaving non-zero influence for the outliers (the influence becomes zero only if the loss is constant for large residuals). On the other hand, the truncated quadratic and maximum consensus losses are known to be particularly insensitive to outliers, but they often require ad-hoc solvers.[13]

Figure 4.6(g)-(l) show the SLAM trajectories obtained by applying gradient descent to two of the robust losses mentioned above: the Huber loss and the truncated quadratic loss. Here we consider the same datasets used in Figure 4.1. We implemented the gradient descent solver using GTSAM's NonlinearConjugateGradientOptimizer [67] with the gradientDescent flag enabled, and using robust noise models to instantiate the robust loss functions. We set the maximum number of iterations and the stopping conditions thresholds (relative and absolute tolerance) to 10000 and 10^{-7}, respectively. All other parameters were left to the default GTSAM values. In the figure, an edge is colored in gray if it is correctly classified as inlier or outlier by the optimization (*i.e.,* an inlier that falls in the quadratic region of the Huber or truncated quadratic loss); it is colored in red if it is an outlier incorrectly classified as an inlier (a "false positive"); it is colored in blue if it is an inlier incorrectly classified as an outlier (a "false negative"). Compared to

[13] For instance, RANSAC (Section 4.2.1) optimizes the maximum consensus loss via sampling (an option that is only viable for the low-dimensional optimization problems arising in the front-end), while graduated non-convexity allows optimizing a broad variety of losses including the truncated quadratic loss (more on this in Section 4.3.4 below).

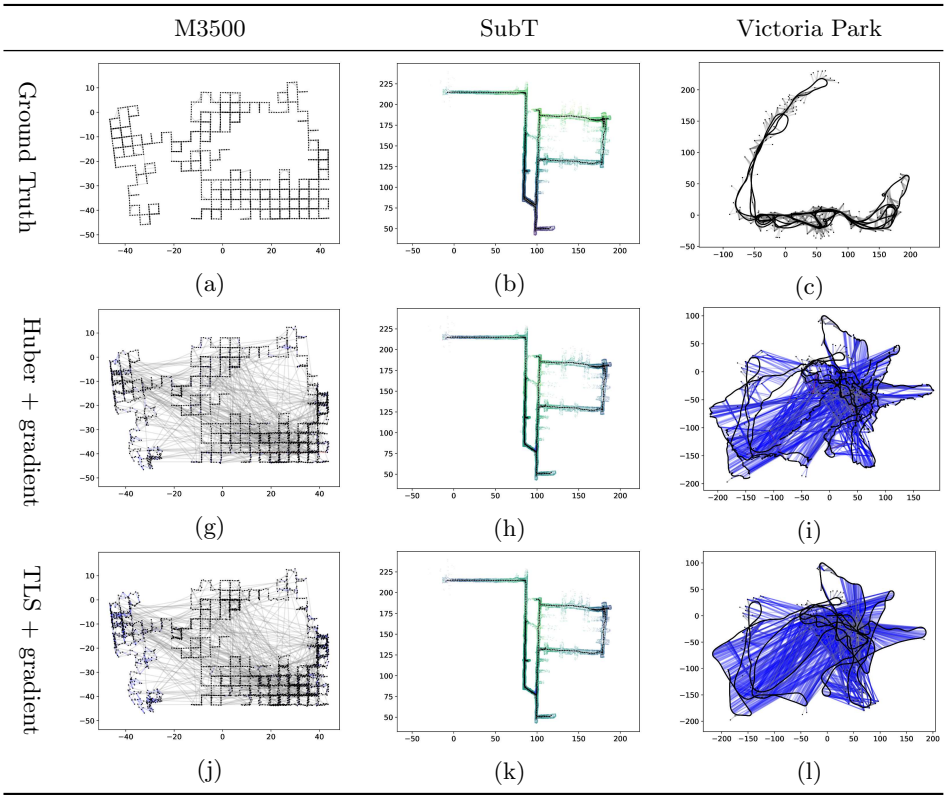


Figure 4.6 Solving SLAM problems with outliers using robust loss functions and gradient descent: (a)-(c) Ground truth trajectories for the M3500, SubT, and Victoria Park datasets. (d)-(f) Trajectory estimates obtained using gradient descent and Huber loss. (h)-(l) Trajectory estimates obtained using gradient descent and truncated quadratic loss. Measurements are visualized as colored edges. In particular, an edge is colored in gray if it is correctly classified as inlier or outlier by the optimization (*i.e.,* an inlier that falls in the quadratic region of the Huber or truncated quadratic loss); it is colored in red if it is an outlier incorrectly classified as an inlier (a "false positive"); it is colored in blue if it is an inlier incorrectly classified as an outlier (a "false negative").

(non-robust) least squares optimization (Figure 4.1(d)-(f)), we note that the use of the robust losses allows us to quickly regain robustness to outliers in the case of the M3500 and SubT datasets, but a simple gradient descent method may still fail to correctly optimize heavily non-convex functions such as the truncated quadratic cost, as in the case of the Victoria Park dataset. We will address this issue with a better solver, based on *graduated non-convexity*, below. Moreover, while gradient descent already improves performance in many of the instances as shown in Figure 4.6, it has slow convergence tails. For instance, in our experiments, it often takes

thousands of iterations to converge. Therefore, in the rest of this section we discuss more advanced solvers, that improve both convergence quality and speed.

As a concluding remark before delving into more advanced solvers, we observe that while it might seem that we gave up on our probabilistic framework when switching to robust loss functions, it is actually possible to derive several robust losses by applying MAP estimation to heavy-tailed noise distributions. For instance, the truncated quadratic loss results from MAP estimation when assuming the noise follow a max-mixture distribution between a Gaussian density (describing the inliers) and a uniform distribution (describing the outliers) [15].

4.3.1 Iteratively Reweighted Least Squares

M-Estimation replaces the least-squares loss by a robust loss ρ in (4.9) — a function that grows sub-quadratically for large residuals. This comes with two prices. First, we lose the efficient solutions already developed for least-squares formulations; for instance, the Gauss-Newton and the Levenberg-Marquardt methods are designed for least squares problems. Second, due to the typical non-convex landspace of M-Estimation, iterative solvers (*e.g.,* based on gradient descent) are sensitive to the quality of initialization and often converge to undesired suboptimal estimates. In this section, we introduce a popular algorithm for solving M-Estimation called *iteratively reweighted least squares* (IRLS) which, as the name suggests, allows reusing the efficient least-squares solvers. In the next section, we introduce graduated non-convexity as a technique to improve the convergence of IRLS.

The basic idea behind IRLS is to optimize (4.9) by solving a *weighted* least squares problem at each iteration

$$\boldsymbol{x}^{(t+1)} = \arg\min_{\boldsymbol{x}} \sum_i w_i(\boldsymbol{x}^{(t)}) r_i^2(\boldsymbol{x}), \tag{4.11}$$

where the weights w_i's depend on the estimate $\boldsymbol{x}^{(t)}$ from the last iteration. We wish the iterative solutions $\boldsymbol{x}^{(t)}$ to converge to the optimal solution of M-Estimation (4.9). This implies that the gradient of the robust loss ρ, shown in (4.10), must match the gradient of the loss in (4.11). By writing down the gradient of (4.11) as

$$\sum_i 2 w_i(\boldsymbol{x}^{(t)}) r_i(\boldsymbol{x}) \frac{\partial r_i(\boldsymbol{x})}{\partial \boldsymbol{x}}$$

and comparing it to (4.10), we obtain the IRLS weight update rule

$$w_i(\boldsymbol{x}^{(t)}) = \frac{1}{2 r_i(\boldsymbol{x}^{(t)})} \frac{\partial \rho(r_i(\boldsymbol{x}^{(t)}))}{\partial r_i(\boldsymbol{x}^{(t)})} = \frac{\psi(r_i(\boldsymbol{x}^{(t)}))}{2 r_i(\boldsymbol{x}^{(t)})}, \tag{4.12}$$

where we recall that $\psi(r_i) := \frac{\partial \rho(r_i)}{\partial r_i}$ is the influence function. Therefore, IRLS alternates computing the weights $w_i(\boldsymbol{x}^{(t)})$ for each measurement i, with performing

an optimization step (*i.e.,* a Gauss-Newton or Levengberg-Marquardt iteration) on the weighted least squares problem (4.11).

Figure 4.7 shows the performance of IRLS on the M3500, SubT, and Victoria Park datasets. IRLS converges in tens of iterations and is typically much faster than gradient descent; for instance, gradient descent requires around 5 seconds to optimize the Huber loss in our M3500 experiments, while IRLS took less than 1.5 seconds. On the other hand, this faster convergence often comes at the cost of a slightly decreased accuracy, as can be seen by comparing Figure 4.7 and Figure 4.6. The convergence properties of the update rule (4.12) has been studied in [6, 191].

4.3.2 Black-Rangarajan Duality

The weight update rule (4.12) is widely used in practice, but its derivation was somewhat heuristic. It also has the issues that (4.12) is not well-defined at the non-differentiable points of ρ (*e.g.,* the cut-off point of the truncated quadratic loss). We now introduce a more principled framework, namely the *Black-Rangarajan (B-R) duality* [31], to solve M-Estimation using IRLS.

Let us present the intuition of B-R duality using the truncated quadratic loss

$$\rho(r_i(\boldsymbol{x})) := \min\{r_i^2(\boldsymbol{x}), \beta_i^2\}, \tag{4.13}$$

where β_i^2 is a bound on the i-th residual such that the i-th measurement is an inlier if $r_i^2(\boldsymbol{x}) \leq \beta_i^2$ and an outlier otherwise. We observe that the cost in (4.13) can be equivalently written as a sum of two terms by introducing a new weight variable $w_i \in [0, 1]$

$$\rho(r_i(\boldsymbol{x})) := \min_{w_i \in [0,1]} w_i r_i^2(\boldsymbol{x}) + (1 - w_i)\beta_i^2, \tag{4.14}$$

where the first term is exactly the weighted least squares, and the second term is a function of w_i that does not depend on $\boldsymbol{x}$. With (4.14), the M-Estimation problem (4.9) with a truncated quadratic loss can be reformulated as

$$\min_{\substack{\boldsymbol{x} \\ w_i \in [0,1], i=1,\ldots,N}} \sum_i \left[w_i r_i^2(\boldsymbol{x}) + (1 - w_i)\beta_i^2 \right], \tag{4.15}$$

where we have introduced one w_i for each measurement residual $r_i(\boldsymbol{x})$. Problem (4.15) is easy to interpret: $w_i = 1$ implies $r_i^2(\boldsymbol{x}) \leq \beta_i^2$ and the i-th measurement is an inlier; $w_i = 0$ implies $r_i^2(\boldsymbol{x}) > \beta_i^2$ and the i-th measurement is an outlier. Moreover, all the residuals with $w_i = 0$ are effectively discarded from the optimization (4.15) and hence robustness is ensured.

B-R duality generalizes the derivation above to a family of robust losses.

Theorem 4.4 (Black-Rangarajan Duality [31])**.** *Given a robust loss function $\rho(\cdot)$, define $\phi(z) := \rho(\sqrt{z})$. If $\phi(z)$ satisfies $\lim_{z \to 0} \phi'(z) = 1$, $\lim_{z \to \infty} \phi'(z) = 0$, and*

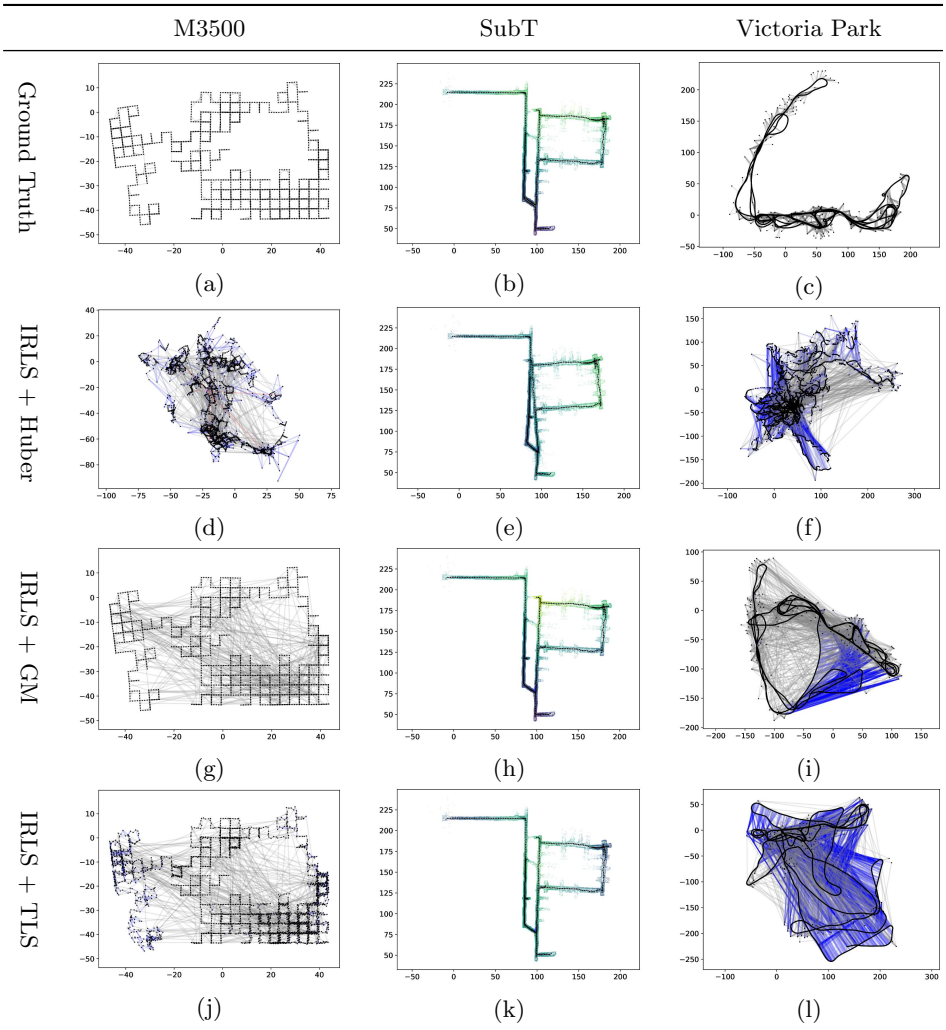


Figure 4.7 Solving SLAM problems with outliers using robust loss functions and Iteratively Re-weighted Least-Squares (IRLS): (a)-(c) Ground truth trajectories for the M3500, SubT, and Victoria Park datasets. (d)-(f) Trajectory estimates obtained using the Huber loss. (g)-(i) Trajectory estimates obtained using the Geman-McClure loss. (j)-(l) Trajectory estimates obtained using the truncated quadratic loss. Measurements are visualized as colored edges. In particular, an edge is colored in gray if it is correctly classified as inlier or outlier by the optimization (*i.e.,* an inlier that falls in the quadratic region of the Huber or truncated quadratic loss); it is colored in red if it is an outlier incorrectly classified as an inlier (a "false positive"); it is colored in blue if it is an inlier incorrectly classified as an outlier (a "false negative").

$\phi''(z) < 0$, then the M-Estimation problem (4.9) is equivalent to

$$\min_{\substack{\boldsymbol{x} \\ w_i \in [0,1], i=1,\ldots,N}} \sum_{i=1}^{N} \left[w_i r_i^2(\boldsymbol{x}) + \Phi_\rho(w_i) \right], \tag{4.16}$$

where $w_i \in [0,1], i = 1, \ldots, N$ are weight variables associated to each residual r_i, and the function $\Phi_\rho(w_i)$, referred to as an outlier process, defines a penalty on the weight w_i whose form is dependent on the choice of robust loss ρ.

In the case of ρ being the truncated quadratic loss, we easily derived from (4.14) that $\Phi_\rho(w_i) = (1-w_i)\beta_i^2$. When ρ takes other forms, [31] provides a recipe to derive $\Phi_\rho(w_i)$. We give an example for the Geman-McClure (G-M) robust loss.

Example 4.5 (B-R Duality for G-M Loss)**.** Consider the G-M robust loss function

$$\rho(r_i(\boldsymbol{x})) = \frac{\beta_i^2 r_i^2(\boldsymbol{x})}{\beta_i^2 + r_i^2(\boldsymbol{x})}, \tag{4.17}$$

where β_i^2 is a noise bound for the i-th residual similar to (4.13). The outlier process associated to (4.17) is

$$\Phi_\rho(w_i) = \beta_i^2(\sqrt{w_i} - 1)^2. \tag{4.18}$$

To verify the correctness of (4.18), consider

$$\min_{w_i \in [0,1]} w_i r_i^2(\boldsymbol{x}) + \Phi_\rho(w_i), \tag{4.19}$$

whose optimal solution is (via setting the gradient of (4.19) to zero)

$$w_i^\star = \left(\frac{\beta_i^2}{r_i^2(\boldsymbol{x}) + \beta_i^2} \right)^2. \tag{4.20}$$

Plugging (4.20) back to the objective of (4.19) recovers the G-M robust loss (4.17).

4.3.3 Alternating Minimization

With the introduction of B-R duality, the IRLS algorithm naturally comes out using a common optimization strategy called *alternating minimization* [267, 27]. The idea is that, although it is difficult to jointly optimize both $\boldsymbol{x}$ and $w_i \in [0,1], i = 1, \ldots, N$ in (4.16), optimization of either $\boldsymbol{x}$ or w_i's when fixing the other is easy. To see this, observe that when w_i's are fixed, problem (4.16) becomes a weighted least squares; analogously, when $\boldsymbol{x}$ is fixed, problem (4.16) becomes

$$\min_{w_i \in [0,1], i=1,\ldots,N} \sum_{i=1}^{N} \Phi_\rho(w_i) + w_i r_i^2(\boldsymbol{x}),$$

which splits into N subproblems, each optimizing a scalar w_i

$$\min_{w_i \in [0,1]} \Phi_\rho(w_i) + w_i r_i^2(\boldsymbol{x}). \tag{4.21}$$

Problem (4.21) is easy to solve and often admits a closed-form solution. In fact, for the G-M robust loss, the solution of (4.21) is just (4.20). For the truncated quadratic loss, problem (4.21) reads

$$\min_{w_i \in [0,1]} (1 - w_i)\beta_i^2 + w_i r_i^2(\boldsymbol{x})$$

and admits a closed-form solution

$$w_i^\star = \begin{cases} 1 & \text{if } r_i^2(\boldsymbol{x}) < \beta_i^2 \\ 0 & \text{if } r_i^2(\boldsymbol{x}) > \beta_i^2 \\ [0,1] & \text{otherwise} \end{cases}.$$

In summary, the t-th iteration of IRLS in the context of B-R duality alternates between two steps

1 **Variable update**: solve a weighted least squares problem using the current weights $w_i^{(t)}$

$$\boldsymbol{x}^{(t)} \in \arg\min_{\boldsymbol{x}} \sum_i w_i^{(t)} r_i^2(\boldsymbol{x}). \tag{4.22}$$

2 **Weight update**: update the weights using $\boldsymbol{x}^{(t)}$

$$w_i^{(t+1)} \in \arg\min_{w_i \in [0,1]} \Phi_\rho(w_i) + w_i r_i^2(\boldsymbol{x}^{(t)}), i = 1, \ldots, N. \tag{4.23}$$

The weight update rule obtained via B-R duality actually matches the popular weight update rule (4.12). Interestingly, instantiating the above IRLS algorithm in SLAM using the G-M robust loss leads to the *dynamic covariance scaling* algorithm [7], which has been proposed in the context of outlier-robust SLAM.

4.3.4 Graduated Non-Convexity

The previous section leveraged Black-Rangarajan duality and alternating minimization to derive the IRLS framework that alternates in solving (4.22) and (4.23). However, due to the non-convexity of common robust losses, the convergence of the IRLS framework can be highly sensitive to the quality of initialization, *i.e.,* how close is $\boldsymbol{x}^{(0)}$ to the optimal solution of (4.9) or how well does $w_i^{(0)}$ reflect the inlier-outlier membership of each measurement. For example, [237] showed that IRLS with the truncated quadratic loss and the Geman-McClure loss might fail when there are as little as 10% outliers in the measurements (*cf.* also with our results in Figure 4.7).

In this section, we introduce the *graduated non-convexity* (GNC) scheme that can make IRLS significantly less sensitive to the quality of initialization. Given a robust cost function ρ, the basic idea of GNC is to create a smooth version of ρ, denoted

as ρ_μ, using a scalar smoothing factor μ. Tuning μ controls the amount of non-convexity in ρ_μ: ρ_μ is convex at one end of the spectrum and recovers the original ρ at the other end of the spectrum. Let us illustrate this using two examples.

Example 4.6 (GNC Truncated Quadratic Loss)**.** Consider the GNC truncated quadratic loss function

$$\rho_\mu(r_i(\boldsymbol{x})) = \begin{cases} r_i^2(\boldsymbol{x}) & \text{if } r_i^2(\boldsymbol{x}) \in \left[0, \frac{\mu}{\mu+1}\beta_i^2\right] \\ 2\beta_i|r_i(\boldsymbol{x})|\sqrt{\mu(\mu+1)} - \mu(\beta_i^2 + r_i^2(\boldsymbol{x})) & \text{if } r_i^2(\boldsymbol{x}) \in \left[\frac{\mu}{\mu+1}\beta_i^2, \frac{\mu+1}{\mu}\beta_i^2\right] \\ \beta_i^2 & r_i^2(\boldsymbol{x}) \in \left[\frac{\mu+1}{\mu}\beta_i^2, +\infty\right]. \end{cases} \tag{4.24}$$

ρ_μ is convex for μ approaching zero and retrieves the truncated quadratic loss in (4.13) for μ approaching infinity.

Example 4.7 (GNC Geman-McClure Loss)**.** Consider the GNC Geman-McClure loss function

$$\rho_\mu(r_i(\boldsymbol{x})) = \frac{\mu\beta_i^2 r_i^2(\boldsymbol{x})}{\mu\beta_i^2 + r_i^2(\boldsymbol{x})}. \tag{4.25}$$

ρ_μ is convex for μ approaching ∞ and recovers the G-M loss (4.17) when $\mu = 1$.

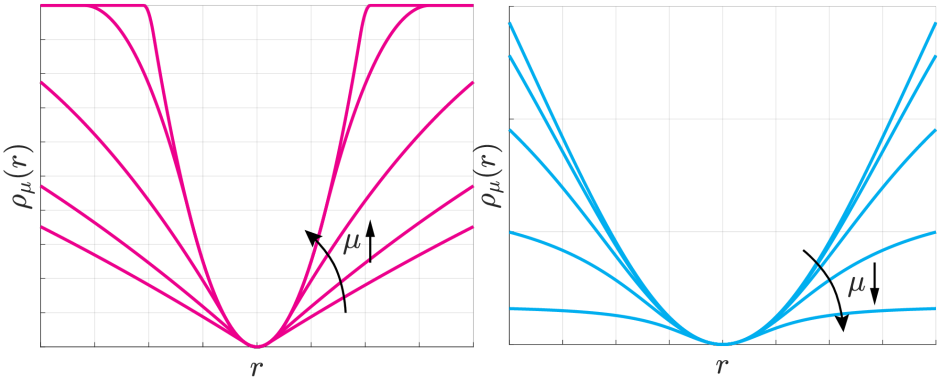


Figure 4.8 Graduated Non-Convexity with control parameter μ for (a) Truncated Quadratic loss and (b) Geman-McClure loss.

Figure 4.8(a) and (b) plot the GNC truncated quadratic loss and the GNC Geman-McClure loss, respectively. Observe how increasing or decreasing the control parameter μ adds more non-convexity to the function.

One nice property of the smoothed GNC functions in (4.24) and (4.25) is that the B-R duality still applies. For the GNC truncated quadratic loss (4.24), applying

B-R duality leads to the outlier process

$$\Phi_{\rho_\mu}(w_i) = \frac{\mu(1-w_i)}{\mu+w_i}\beta_i^2.$$

For the GNC Geman-McClure loss (4.25), applying B-R duality leads to the outlier process

$$\Phi_{\rho_\mu}(w_i) = \mu\beta_i^2(\sqrt{w_i}-1)^2.$$

We are now ready to state the GNC algorithm, which at each iteration performs three steps

1 **Variable update**: solve a weighted least squares problem using the current weights $w_i^{(t)}$

$$\boldsymbol{x}^{(t)} \in \arg\min_{\boldsymbol{x}} \sum_i w_i^{(t)} r_i^2(\boldsymbol{x}). \tag{4.26}$$

2 **Weight update**: update the weights using $\boldsymbol{x}^{(t)}$

$$w_i^{(t+1)} \in \arg\min_{w_i\in[0,1]} \Phi_{\rho_\mu}(w_i) + w_i r_i^2(\boldsymbol{x}^{(t)}), i = 1,\dots,N. \tag{4.27}$$

3 **Control parameter update**: Increase or decrease μ to add more nonconvexity to ρ_μ.

The GNC algorithm is similar to the IRLS algorithm, except that it starts with a convex, smoothed function ρ_μ and then iteratively updates the control parameter μ to gradually add more non-convexity to ρ_μ to approach the original loss function ρ. Depending on the definition of the smoothed loss ρ_μ, one would recover the original ρ by either increasing or decreasing μ. For instance, the smoother GNC truncated quadratic loss recovers the original truncated quadratic loss when μ is large, hence μ is increased by a constant factor $\gamma > 1$ at each GNC iteration (*e.g.,* $\gamma = 1.4$ in [296]). Conversely, the smoother Geman-McClure loss recovers the original GM loss when μ is close to 1, hence μ is *divided* by $\gamma > 1$ at each GNC iteration.

Figure 4.9 showcases the SLAM trajectories obtained by applying GNC on the same three datasets of Figure 4.6, with two different robust losses: the Geman-McClure loss and the truncated quadratic loss. We implemented GNC using GTSAM's GNCOptimizer. By comparing the figure with Figure 4.6 and Figure 4.7 we observe that GNC ensures better convergence (*i.e.,* it is less prone to being stuck in local minima) and recovers fairly accurate trajectories in all the three datasets. While GNC has been shown to be extremely resilient to outliers (*e.g.,* it has shown to tolerate around 80-90% incorrect loop closures in real-world problems [296, 47]), we remark that the approach does not provide any convergence guarantees. Moreover, its performance has been empirically seen to be problem-dependent, and while it leads to superb performance in pose-graph optimization problems, its performance largely degrades in other perception problems [236].

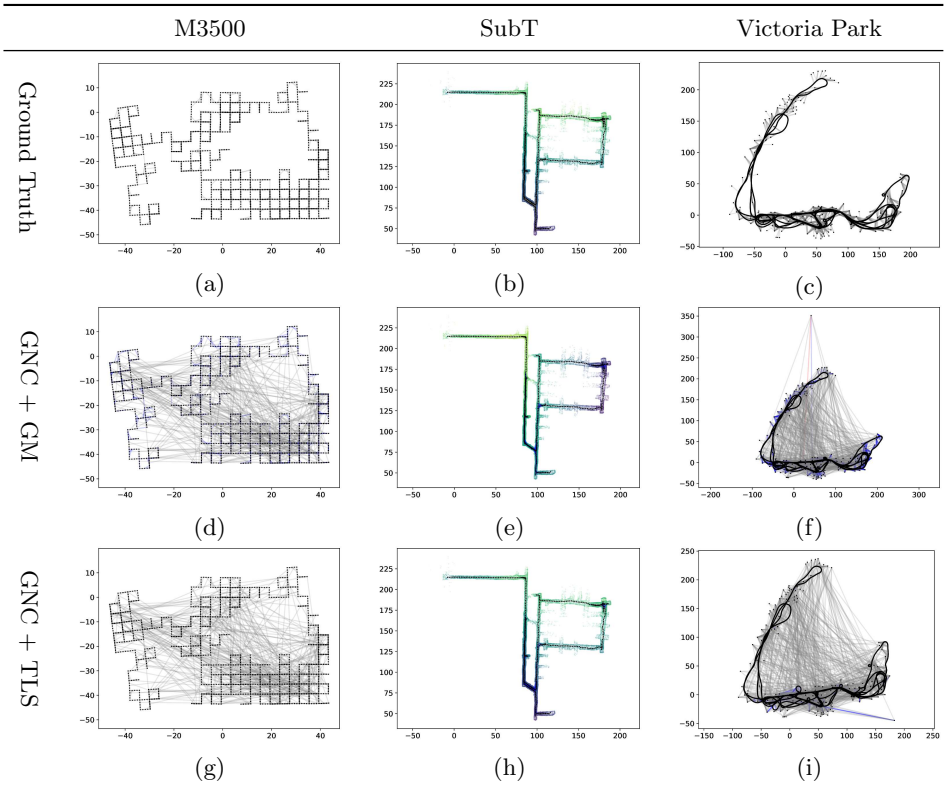


Figure 4.9 Solving SLAM problems with outliers using robust loss functions and graduated non-convexity (GNC): (a)-(c) Ground truth trajectories for the M3500, SubT, and Victoria Park datasets. (d)-(f) Trajectory estimates obtained using the Geman-McClure loss. (g)-(i) Trajectory estimates obtained using the truncated quadratic loss. Measurements are visualized as colored edges. In particular, an edge is colored in gray if it is correctly classified as inlier or outlier by the optimization (*i.e.,* an inlier that falls in the quadratic region of the Huber or truncated quadratic loss); it is colored in red if it is an outlier incorrectly classified as an inlier (a "false positive"); it is colored in blue if it is an inlier incorrectly classified as an outlier (a "false negative").

Below we showcase a problem when GNC fails and also show that combining front-end and back-end outlier rejection can be beneficial. Towards this goal, we are going to consider a slightly more challenging SLAM setup compared to the ones discussed above. Earlier in this chapter, we considered pose-graph optimization problems (*e.g.,* Figure 4.1) where the odometry is reliable but there might be outliers in the loop closures or in the landmark measurements. This setting essentially assumes the presence of an "odometry backbone" that largely simplifies the problem by providing a set of trusted measurements while also allowing building an initial guess for the robot poses. While this assumption is realistic in many

problems,[14] certain SLAM applications might lack an odometry backbone. For instance, certain odometry sensors might produce incorrect measurements, *e.g.,* due to wheel slippage in wheel odometry, or incorrect lidar alignment in lidar odometry. Another example arises in multi-robot SLAM, where each robot has an odometry backbone, but the overall SLAM problem (including the trajectory of all robots) does not [169].

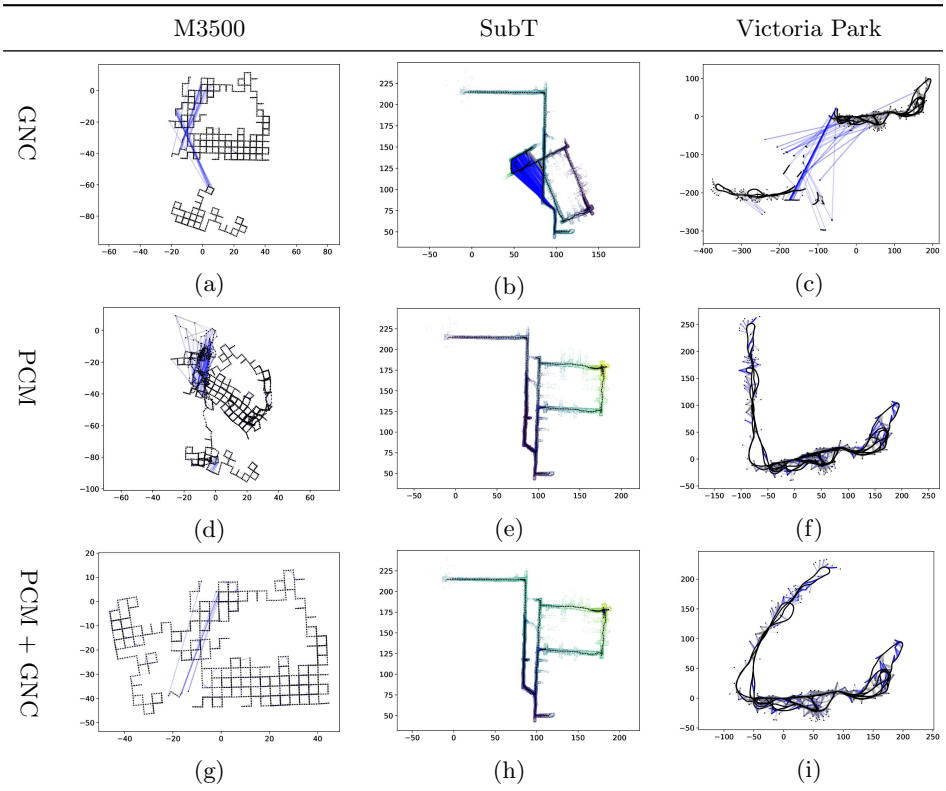


Figure 4.10 SLAM problems with outliers in both the loop closures and landmark measurements, as well as the odometry: (a)-(c) Trajectory estimates obtained using GNC with the truncated quadratic loss. (d)-(f) Trajectory estimates obtained using PCM for front-end outlier rejection followed by least squares optimization. (g)-(i) Trajectory estimates obtained using PCM for front-end outlier rejection followed by GNC with truncated quadratic loss.

Robustly solving SLAM problems where both odometry and loop closure measurements can be outliers is extremely hard. To showcase the shortcomings of the

[14] The fact that odometric measurements are computed between consecutive frames makes data association relatively simpler, in particular when the rate of the sensor (*e.g.,* camera framerate) is much faster than the motion of the robot. Intuitively, consecutive frames will provide snapshots of the scene from similar viewpoints, thus reducing sources of errors for feature matching, including illumination changes, occlusions, lack of viewpoint invariance, or perceptual aliasing.

approaches we discussed, in this more complex setting with potentially incorrect odometry measurements, we first modify the pose-graph problems in Figure 4.1 by corrupting two randomly selected odometry measurements. Then we attempt to solve the problem with GNC. In particular, in GNC we fix all odometry measurements except the two potentially corrupted measurements as inliers. Figure 4.10 shows the trajectories obtained by solving the problem with GNC. GNC fails due to the corrupted initialization and converges to a local minima by categorizing all measurements (including inliers) as outliers (blue edges). The same figure also shows the result of using PCM to filter out gross outliers before using a least-squares back-end. PCM is agnostic to the initial guess, hence is able to converge to better solutions in the case of the SubT and Victoria Park datasets (Figure 4.10 (e) and (f)). Interestingly, we get best results in the SubT and Victoria Park datasets by combining front-end outlier rejection (PCM) with GNC. At the same time, all three methods (GNC, PCM, and PCM+GNC) fail to converge to acceptable solutions in the M3500 dataset, confirming the hardness of this SLAM setup and the limitations of existing SLAM algorithms in terms of outlier rejection.

4.4 Further References and New Trends

Consensus Maximization. While in this chapter we discussed the most basic instantiation of a RANSAC algorithm (according to the initial proposal in [89]), it is worth mentioning that the literature offers many RANSAC variants, including variants that refine estimates through local optimization [57], use better scores (rather than the size of the consensus set) in the RANSAC iterations (*e.g.,* MLESAC [209]), or bias the sampling in the RANSAC iterations (*e.g.,* PROSAC [56]). The recent literature also includes differentiable variants of RANSAC [284] and variants that attempt to find the inliers when the parameter γ in (4.4) is unknown (*e.g.,* [16]). A recent survey and evaluation of RANSAC variants can be found in [265].

Beyond RANSAC, the literature also includes approaches for *exact* consensus maximization, typically based on *branch-and-bound* [23, 112, 309, 150, 241, 55, 120, 38, 299, 298]. Despite its global optimality guarantees, branch-and-bound has exponential runtime in the worst case and does not scale to high-dimensional problems.

Pairwise Consistency Maximization. The PCM approach described in Section 4.2.2 was originally proposed in the context of multi-robot SLAM in [169], where this approach showed particular promise. Graph-theoretic outlier rejection has also been investigated in computer vision. Segundo and Artieda [226] build an association graph and find the maximum clique for 2D image feature matching. Perera and Barnes [207] segment objects under rigid body motion with a clique formulation. Leordeanu and Hebert [148] establish image matches by finding strongly-connected clusters in the correspondence graph with an approximate spectral method. Enqvist *et al.* [85] develop an outlier rejection algorithm for 3D-3D and 2D-3D registration based on approximate vertex cover. Yang and Carlone [292, 297]

and Parra *et al.* [39] investigate graph-theoretic outlier rejection based on maximum clique for 3D-3D registration. The idea of checking consistency across a subset of measurements also arises in Latif *et al.* [143], which perform loop-closure outlier rejection by clustering measurements together and checking for consistency using a Chi-squared-based test. The PCM paper [169], similarly to the discussion in this chapter, focuses on pairwise consistency. More recently, PCM has been extended to group-k consistency (*i.e.,* the case where the consistency constraint (4.7) involves k measurements instead of only 2 measurements) in [236, 237, 91]. These papers essentially generalize the notion of consistency graphs to consistency *hypergraphs*, where each hyper-edge involves k nodes. Related work also considers soft variations of the maximum clique problem, where the binary condition (4.7) is relaxed to produce continuous weights on the edges of the consistency graph [164, 163]. These methods have been used in practical applications, including subterranean exploration [81], lidar point-cloud localization [165], multi-robot metrics-semantic mapping [261], and global localization in unstructured environments [14].

Alternating Minimization and Graduated Non-Convexity. M-estimation has been a popular approach for robust estimation in robotics [33] and vision [230, 49]. Tavish *et al.* [253] investigate the performance of different loss functions. Several papers investigate formulations with auxiliary variables as the one in (4.16), without realizing the connection to M-estimation provided by the Black-Rangarajan duality (Theorem 4.4). For instance, Sünderhauf and Protzel [249, 250] and Agarwal *et al.* [7] augment the problem with latent binary variables responsible for deactivating outliers. Lee *et al.* [147] use expectation maximization. Olson and Agarwal [196] use a max-mixture distribution to approximate multi-modal measurement noise. Recently, Barron [21] proposes a single parametrized function that generalizes a family of robust loss functions in M-estimation. Chebrolu *et al.* [50] design an expectation-maximization algorithm to simultaneously estimate the unknown quantity $\boldsymbol{x}$ and choose a suitable robust loss function ρ. The graduated non-convexity algorithm was first introduced in [32, 31] for outlier rejection in early vision applications; more recently, the algorithm was used for point cloud registration [311, 297], SLAM [296], and other applications [15]. Recently, Peng *et al.* [205] has proposed an algorithm similar to GNC and IRLS, that is based on the idea of *smooth majorization* in optimization and can be applied to a broad set of robust losses. Moreover, [205] derives global and local convergence guarantees for GNC.

Certifiable Algorithms. The algorithms described so far can be broadly divided into two categories: (i) *fast heuristics* (*e.g.,* RANSAC or local solvers for M-estimation), which are efficient but provide little performance guarantees, and (ii) *global solvers* (*e.g.,* branch-and-bound), which offer optimality guarantees but scale poorly with the problem size. Recent years have seen the advent of a new type of methods, called *certifiable algorithms*, that try to strike a balance between tractability and optimality. Certifiable algorithms relax non-convex robust estima-

tion problems into convex *semidefinite programs* (SDP),[15] whose solutions can be obtained in polynomial time and provide readily checkable *a posteriori* global optimality certificates. Certifiable algorithms for robust estimation have been proposed in the context of rotation estimation [293], 3D-3D registration [297], and pose-graph optimization [142, 43]. A fairly general approach to derive certifiable algorithms for problems with outliers is described in [294, 295], while connections with parallel work in statistics is discussed in [42]. With few notable exceptions, these algorithms, albeit running in polynomial time, are still computationally expensive and typically much slower than heuristics methods. In some cases, the insights behind these algorithms can be used to certify optimality of a solution obtained with a fast heuristic [294], hence getting the best of both worlds.

[15] We are going to review the notion of certifiable algorithms in the context of SLAM in Chapter 7.

5

Differentiable Optimization

Chen Wang, Krishna Murthy Jatavallabhula, and Mustafa Mukadam

5.1 Introduction

As presented in Chapter 2, the design of a contemporary SLAM system generally adheres to a front-end and back-end architecture. In this structure, the front-end is typically responsible for pre-processing sensor data and generating an initial estimate of the robot's trajectory and the map of the environment, while the back-end refines these initial estimates to improve overall accuracy. Recent advances in machine learning have provided new approaches, based on deep neural networks, that have the potential to enhance some of the functionalities in the SLAM front-end. For instance, deep learning-based methods can exhibit impressive performance in feature detection and matching [229, 73, 306] and front-end motion estimation [281, 256]. These methods train a neural network from a large dataset of examples, and then make estimations without being explicitly programmed to perform the task. Meanwhile, geometry-based techniques persist as an essential element for the SLAM back-end, primarily due to their generality and effectiveness in producing a globally consistent estimate by solving an optimization problem.

While in principle one could just "plug" a learning-based SLAM front-end in the SLAM architecture and feed the corresponding outputs to the back-end, the use of learning-based techniques opens the door for a less unidirectional information exchange. In particular, the back-end can now provide feedback to the front-end, enabling it to learn directly from the back-end estimates in a way that the two modules can more harmoniously cooperate to reduce the estimation errors. Reconciling geometric approaches with deep learning to leverage their complementary strengths is a common thread in a large body of recent work in SLAM. In particular, an emerging trend is to differentiate through geometry-based optimization problems arising in the SLAM back-end. Intuitively, differentiating through an optimization problem allows understanding how the optimal solution of that problem (*e.g.,* our SLAM estimate) depends on the parameters of that problem — in our case, the measurements produced by a learning-based front-end; this in turns allows optimizing the front-end to maximize the SLAM accuracy. One could think about this as a *bilevel optimization* problem, *i.e.,* an upper-level optimization process subject

to a lower-level optimization — in particular, a neural network based-optimization to train the front-end, subject to a geometry-based optimization that computes the SLAM solution for a given front-end output.

The ability to compute gradients end-to-end through an optimization is the core of solving a bilevel optimization problem, which allows neural models to take advantage of geometric priors captured by the optimization. The flexibility of such a scheme has led to promising state-of-the-art results in a wide range of applications such as structure from motion [255], motion planning [28, 291], SLAM [121, 256], bundle adjustment [252, 306], state estimation [302, 51], and image alignment [166].

In this chapter, we illustrate the basics of how to differentiate through nonlinear least squares problems, such as the ones arising in SLAM. Specifically, Section 5.1.1 restates the non-linear least square (NLS) problem. Section 5.2 describes how to differentiate through the NLS problem. Section 5.3 shows how to differentiate problems defined on manifold. Section 5.4 discusses numerical challenges of the above differentiation and introduces related machine learning libraries. Finally, Section 5.5 provides examples of differentiable optimization in contemporary SLAM systems.

5.1.1 Recap on Nonlinear Least Squares

Non-linear least squares (NLS) estimate the parameters of a model by minimizing the sum of the squares of the mismatch between observed values and those predicted by the model. Unlike linear least squares, NLS involves a model that is non-linear in the parameters. Beyond our factors graphs in Chapter 2, this approach is widely used in many fields such as statistics, physics, and engineering, where it is useful for fitting complex models to data when the relationship between variables is not straightforward, enabling more accurate and robust predictions.

Specifically, NLS aim to find variables $\boldsymbol{x} \in \mathbb{R}^n$ by solving:

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}} \mathcal{L}(\boldsymbol{x}) = \arg\min_{\boldsymbol{x}} = \frac{1}{2} \sum_i || \underbrace{w_i \boldsymbol{c}_i(\boldsymbol{x}_i)}_{\boldsymbol{r}_i(\boldsymbol{x}_i)} ||^2, \tag{5.1}$$

where the objective $\mathcal{L}(\boldsymbol{x})$ is a sum of squared vector-valued residual terms $\boldsymbol{r}_i$, each a function of $\boldsymbol{x}_i \subset \boldsymbol{x}$ that are (non-disjoint) subsets of the optimization variables $\boldsymbol{x} = \{\boldsymbol{x}_i\}$. While for now we assume $\boldsymbol{x}_i$ to be vectors, later in the chapter we generalize the discussion to the case where the variables belong to a manifold. For flexibility, here we represent a residual $\boldsymbol{r}_i(\boldsymbol{x}_i) = w_i \boldsymbol{c}_i(\boldsymbol{x}_i)$ as a product of a weight w_i and vector cost $\boldsymbol{c}_i$.

As explained in Chapter 2, a NLS is normally solved by iteratively linearizing the nonlinear objective around the current variables to get the linear system $(\sum_i \boldsymbol{J}_i^\top \boldsymbol{J}_i)\delta\boldsymbol{x} = (\sum_i \boldsymbol{J}_i^\top \boldsymbol{r}_i)$, then solving the linear system to find the update $\delta\boldsymbol{x}$, and finally updating the variables $\boldsymbol{x} \leftarrow \boldsymbol{x} + \delta\boldsymbol{x}$, until convergence. We have also commented in Chapter 3 that the addition in the update step is more generally a

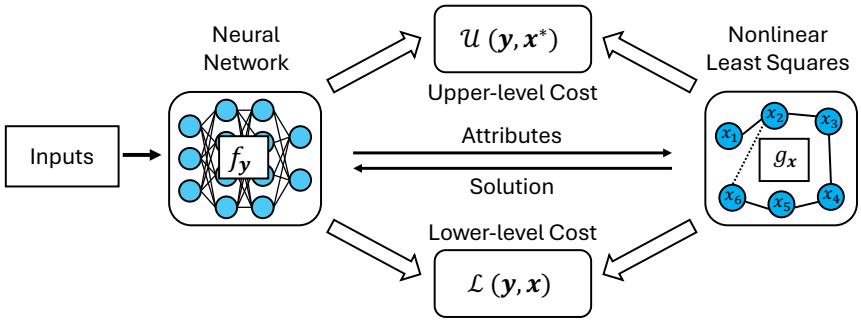


Figure 5.1 A modern SLAM system often involves both neural networks and nonlinear least squares. To eliminate compound errors introduced by optimizing the two modules separately, we can optimize the system in an end-to-end manner by formulating the entire system as a bilevel optimization, which involves an upper-level cost and a lower-level cost.

retraction mapping for variables that belong to a manifold. In the linear system, $\boldsymbol{J}_i = [\partial \boldsymbol{r}_i / \partial \boldsymbol{x}_i]$ are the Jacobians of residuals with respect to the variables. This iterative method above, called Gauss-Newton (GN), is a nonlinear optimizer that is (approximately) second-order, since $\sum_i \boldsymbol{J}_i^\top \boldsymbol{J}_i$ is an approximation of the Hessian. To improve robustness and convergence, variations like Levenberg-Marquardt (LM) dampen the linear system, while others adjust the step size for the update with line search, *e.g.,* Dogleg introduced in Chapter 2.

5.2 Differentiation Through Nonlinear Least Squares

To seamlessly merge deep learning with nonlinear least squares, differentiable nonlinear least squares (DNLS) are often required to solve the optimization problem illustrated in Figure 5.1. This necessitates gradients of the solution $\boldsymbol{x}^*$ with respect to any upper-level neural model parameters $\boldsymbol{y}$ that parameterize the objective $\mathcal{U}(\boldsymbol{x}; \boldsymbol{y})$ and, in turn, any costs $c_i(\boldsymbol{x}_i; \boldsymbol{y})$ or initialization for variables $\boldsymbol{x}_{\text{init}}(\boldsymbol{y})$. The goal is to learn these parameters $\boldsymbol{y}$ end-to-end with a lower-level learning objective $\mathcal{L}$ defined as a function of $\boldsymbol{x}$. This results in a bilevel optimization (BLO), which can be written as:

$$\boldsymbol{y}^* = \arg\min_{\boldsymbol{y} \in \Theta} \ \mathcal{U}(\boldsymbol{y}, \boldsymbol{x}^*), \tag{5.2a}$$

$$\text{s.t.} \quad \boldsymbol{x}^* = \arg\min_{\boldsymbol{x} \in \Psi} \ \mathcal{L}(\boldsymbol{y}, \boldsymbol{x}), \tag{5.2b}$$

where $\mathcal{L} : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}$ is a lower-level (LL) cost, $\mathcal{U} : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}$ is a upper-level (UL) cost, $\boldsymbol{x} \in \Psi$ and $\boldsymbol{y} \in \Theta$ are the feasible sets.

In practice, the variables $\boldsymbol{x}$ are often parameters with explicit physical meanings such as camera poses, while $\boldsymbol{y}$ are parameters without physical meanings such as weights in a neural network. We next present two examples to explain this.

Example 5.1. Imagine a SLAM system that leverages a neural network (parameterized by $\boldsymbol{y}$) for feature extraction/matching, while utilizing bundle adjustment (BA) for pose estimation (parameterized by $\boldsymbol{x}$), which take the feature matching as an input. In this example, the UL cost (5.2a) can be feature matching error for optimizing the network, while the LL cost L (5.2b) can be the reprojection error for BA. Intuitively, the optimal solution $\boldsymbol{x}^*$ for the camera poses and landmark positions plays the role of a supervisory signal in the neural network training. Therefore, optimizing the BLO (5.2) allows us to further reduce the matching error via back-propagating the BA reprojection errors [306].

Example 5.2. Imagine a full SLAM system that uses a neural network for front-end pose estimation, while leverages pose-graph optimization (PGO) as the back-end to eliminate odometry drifts. In this example, both UL and LL costs can be the pose-graph error. The difference is the UL cost optimizes the network parameterized by $\boldsymbol{y}$, while the LL cost optimizes the camera poses parameterized by $\boldsymbol{x}$. As a result, the front-end network can leverage global geometric knowledge obtained through pose-graph optimization by back-propagating the pose residuals from the back-end PGO [94].

BLO is a long-standing and well-researched problem [275, 122, 155]. Solving a BLO often relies on gradient-descent techniques. Specifically, the UL optimization performs updates in the form $\boldsymbol{y} \leftarrow \boldsymbol{y} + \delta\boldsymbol{y}$, where $\delta\boldsymbol{y}$ is a step in the direction of the negative gradient. Therefore, we need compute the gradient of $\mathcal{U}$ with respect to the UL variable $\boldsymbol{y}$, which can be written as

$$\nabla_{\boldsymbol{y}}\mathcal{U} = \frac{\partial \mathcal{U}(\boldsymbol{y}, \boldsymbol{x}^*)}{\partial \boldsymbol{y}} + \frac{\partial \mathcal{U}(\boldsymbol{y}, \boldsymbol{x}^*)}{\partial \boldsymbol{x}^*} \frac{\partial \boldsymbol{x}^*(\boldsymbol{y})}{\partial \boldsymbol{y}}, \tag{5.3}$$

where the term $\frac{\partial \boldsymbol{x}^*(\boldsymbol{y})}{\partial \boldsymbol{y}}$ involves indirect gradient computation. Since other direct gradient terms in (5.3) are easy to obtain, the challenge of solving a BLO (5.2) is to compute the term $\frac{\partial \boldsymbol{x}^*(\boldsymbol{y})}{\partial \boldsymbol{y}}$. For this purpose, a series of techniques have been developed from either explicit or implicit perspectives. This involves recurrent differentiation through dynamical systems or implicit differentiation theory, which are often referred to as **unrolled differentiation** and **implicit differentiation**, respectively. These algorithms have been summarized in [155, 275] and here we list a generic framework incorporating both methods in Algorithm 1. We next explain the unrolled differentiation and implicit differentiation, respectively.

5.2.1 Unrolled Differentiation

Unrolled Differentiation needs automatic differentiation (AutoDiff) through the LL optimization to solve a BLO problem. Specifically, given an initialization $\boldsymbol{x}_0 = \Phi_0(\boldsymbol{y})$ at step $t = 0$, the iterative process of unrolled LL optimization is

$$\boldsymbol{x}_t = \Phi_t(\boldsymbol{x}_{t-1}; \boldsymbol{y}), \quad t = 1, \cdots, T, \tag{5.4}$$

Algorithm 1 Solving BLO by *Unrolled Differentiation* or *Implicit Differentiation*.

1: **Initialization:** $\boldsymbol{y}_0$, $\boldsymbol{x}_0$.
2: **while** Not Convergent ($\|\boldsymbol{y}_{k+1} - \boldsymbol{y}_k\|$ is large enough) **do**
3: Obtain $\boldsymbol{x}_T$ by solving (5.2b) by a generic optimizer $\mathcal{O}$ with T steps.
4: Efficient estimation of upper-level gradients in (5.3) via
Unrolled Differentiation: $\hat{\nabla}_{\boldsymbol{y}_k} U = \frac{\partial U(\boldsymbol{y}_k, \boldsymbol{x}_T))}{\partial \boldsymbol{y}_k}$ via AutoDiff in (5.7).
Implicit Differentiation (Algorithm 2): Compute

$$\hat{\nabla}_{\boldsymbol{y}_k} \mathcal{U} = \frac{\partial \mathcal{U}}{\partial \boldsymbol{y}_k}\Big|_{\boldsymbol{x}_T} + \frac{\partial \mathcal{U}}{\partial \boldsymbol{x}^*} \frac{\partial \boldsymbol{x}^*}{\partial \boldsymbol{y}_k}\Big|_{\boldsymbol{x}_T},$$

where the implicit derivatives $\frac{\partial \boldsymbol{x}^*}{\partial \boldsymbol{y}_k}$ can be obtained by solving an equation derived via lower-level optimality conditions (surveyed in following sections).
5: Compute $\boldsymbol{y}_{k+1}$ via gradients using $\hat{\nabla}_{\boldsymbol{y}_k} U$.
6: **end while**

where Φ_t denotes an updating scheme based on the LL problem at the t-th step and T is the number of iterations. One updating scheme is the gradient descent:

$$\Phi_t(\boldsymbol{x}_{t-1}; \boldsymbol{y}) = \boldsymbol{x}_{t-1} - \eta_t \cdot \frac{\partial \mathcal{L}(\boldsymbol{x}_{t-1}, \boldsymbol{y})}{\partial \boldsymbol{x}_{t-1}}, \tag{5.5}$$

where η_t is a learning rate and the term $\frac{\partial \mathcal{L}(\boldsymbol{x}_{t-1}, \boldsymbol{y})}{\partial \boldsymbol{x}_{t-1}}$ can be computed from AutoDiff.[1] Therefore, we can compute the $\nabla_{\boldsymbol{y}} \mathcal{U}(\boldsymbol{y})$ by substituting $\boldsymbol{x}_T$ approximately for $\boldsymbol{x}^*$ and the full unrolled system can be defined as

$$\boldsymbol{x}^* \approx \boldsymbol{x}_T = \Phi(\boldsymbol{y}) = \left(\Phi_T \circ \cdots \circ \Phi_1 \circ \Phi_0\right)(\boldsymbol{y}), \tag{5.6}$$

where the symbol $\circ$ denotes the function composition. As a result, we only need to consider the following problem instead of a bilevel optimization in (5.2):

$$\min_{\boldsymbol{y} \in \Theta} \ \mathcal{U}(\boldsymbol{y}, \Phi(\boldsymbol{y})), \tag{5.7}$$

which needs to compute $\frac{\partial \Phi(\boldsymbol{y})}{\partial \boldsymbol{y}}$ via AutoDiff instead of calculating (5.3). It is worth noting that there exist two approaches for computing the recurrent gradients, one of which corresponds to backward propagation in a reverse-mode way [204], and the other corresponds to the forward-mode way [221]. We omit the details of the two approaches of AutoDiff and refer the readers to the AutoDiff libraries such as PyTorch [202] for deep learning and PyPose [274] and Theseus [210] for SLAM. A review of these approaches can also be found in Liu et al. [155].

[1] While here we mention gradient descent, the same ideas can be extended to other iterative optimization methods, such as Gauss-Newton.

5.2.2 Truncated Unrolled Differentiation

The reverse and forward modes are two precise recurrent gradient calculation methods but are time-consuming with the full iterative propagation. This is due to the complicated long-term dependencies of the UL problem on $\boldsymbol{x}_t$, where $t = 0, 1, \cdots, T$. This difficulty is further aggravated when both $\boldsymbol{y}$ and $\boldsymbol{x}$ are high-dimensional vectors. To overcome this challenge, the truncated unrolled differentiation has been investigated as a way to compute high-quality approximate gradients with significantly less computation time and memory. Specifically, by ignoring the long-term dependencies and approximating the gradient of (5.5) with partial history, i.e., storing only the last M iterations ($t = T, T-1, \cdots, T-M$), we can significantly reduce the time and space complexity. It has been proved by Shaban et al. [232] that using fewer backward steps to compute the gradients could perform comparably to optimization with the exact one, while requiring much less memory and computation.

In case of more stringent computational and memory constraints, truncated unrolled differentiation is still often a bottleneck in modern robotic applications. Therefore, researchers have also tried to further simplify the truncated differentiation by only performing a one-step iteration in (5.4) to remove the recursive structure [153], *i.e.,*

$$\nabla_{\boldsymbol{y}} \mathcal{U} = \frac{\partial \mathcal{U}(\boldsymbol{y}, \boldsymbol{x}_1(\boldsymbol{y}))}{\partial \boldsymbol{y}} + \frac{\partial \mathcal{U}(\boldsymbol{y}, \boldsymbol{x}_1(\boldsymbol{y}))}{\partial \boldsymbol{x}_1} \frac{\partial \boldsymbol{x}_1(\boldsymbol{y})}{\partial \boldsymbol{y}}, \tag{5.8}$$

where the term $\frac{\partial \boldsymbol{x}_1(\boldsymbol{y})}{\partial \boldsymbol{y}}$ is a Hessian that can be calculated from (5.5) as

$$\frac{\partial \boldsymbol{x}_1(\boldsymbol{y})}{\partial \boldsymbol{y}} = -\frac{\partial^2 \mathcal{L}(\boldsymbol{x}_0, \boldsymbol{y})}{\partial \boldsymbol{x}_0 \partial \boldsymbol{y}}. \tag{5.9}$$

Since calculating a Hessian is time-consuming in some applications, we can resort to numerical solutions that apply small perturbations to the variables $\boldsymbol{x}$ and calculate an approximation of the second term in (5.8) as a whole:

$$\frac{\partial \mathcal{U}(\boldsymbol{y}, \boldsymbol{x}_1(\boldsymbol{y}))}{\partial \boldsymbol{x}_1} \frac{\partial \boldsymbol{x}_1(\boldsymbol{y})}{\partial \boldsymbol{y}} \approx \frac{\frac{\partial \mathcal{L}(\boldsymbol{x}_0^+, \boldsymbol{y})}{\partial \boldsymbol{y}} - \frac{\mathcal{L}(\boldsymbol{x}_0^-, \boldsymbol{y})}{\partial \boldsymbol{y}}}{2\epsilon}, \tag{5.10}$$

where ϵ is a small scalar and $\boldsymbol{x}_0^{\pm} = \boldsymbol{x}_0 \pm \epsilon \frac{\partial \mathcal{U}(\boldsymbol{y}, \boldsymbol{x}_1(\boldsymbol{y}))}{\partial \boldsymbol{x}_1}$ is a small perturbation. This bypasses an explicit calculation of the Jacobian $\frac{\partial \boldsymbol{x}_1(\boldsymbol{y})}{\partial \boldsymbol{y}}$. Nevertheless, we need to pay attention to the perturbation model if non-Euclidean variables are involved, *e.g.,* variables belonging to Lie Groups. Fortunately, the AutoDiff of Lie Group for Hessian-vector and Jacobian-vector multiplications are supported in modern libraries, such as PyPose [274], which will be introduced in Section 5.4.

5.2.3 Implicit Differentiation

It is intuitive that the term $\frac{\partial \boldsymbol{x}^*(\boldsymbol{y})}{\partial \boldsymbol{y}}$ in (5.3) is dependent on the LL cost (5.2b), thus *implicit differentiation* can be used to derive a solution to the gradient.

Example 5.3. In calculus, *implicit differentiation* refers to the method makes use of the chain rule to differentiate implicit function. To differentiate an implicit function $y(x)$, defined by an equation $R(x, y) = 0$, it is not generally possible to solve it explicitly for y and then differentiate. Instead, one can totally differentiate $R(x, y) = 0$ with respect to x and then solve the resulting linear equation for $\frac{\mathrm{d}y}{\mathrm{d}x}$ to explicitly get the derivative in terms of x and y. For instance, consider an implicit function $x + y + 5 = 0$, differentiating it with respect to x on its both sides gives $\frac{\mathrm{d}y}{\mathrm{d}x} + \frac{\mathrm{d}x}{\mathrm{d}x} + \frac{\mathrm{d}}{\mathrm{d}x}(5) = 0 \Rightarrow \frac{\mathrm{d}y}{\mathrm{d}x} + 1 + 0 = 0$. Solving for $\frac{\mathrm{d}y}{\mathrm{d}x}$ gives $\frac{\mathrm{d}y}{\mathrm{d}x} = -1$.

Assume the LL cost $\mathcal{L}$ is at least twice differentiable w.r.t. both $\boldsymbol{y}$ and $\boldsymbol{x}$, then we have $\frac{\partial \mathcal{L}(\boldsymbol{x}^*(\boldsymbol{y}), \boldsymbol{y})}{\partial \boldsymbol{x}^*(\boldsymbol{y})} = 0$ due to the optimality condition where $\boldsymbol{x}^*$ is a stationary point. Derive the equation $\frac{\partial \mathcal{L}(\boldsymbol{x}^*(\boldsymbol{y}), \boldsymbol{y})}{\partial \boldsymbol{x}^*(\boldsymbol{y})} = 0$ on both sides w.r.t. $\boldsymbol{y}$ giving us

$$\frac{\partial^2 \mathcal{L}(\boldsymbol{x}^*(\boldsymbol{y}), \boldsymbol{y})}{\partial \boldsymbol{x}^*(\boldsymbol{y}) \partial \boldsymbol{y}} + \frac{\partial^2 \mathcal{L}(\boldsymbol{x}^*(\boldsymbol{y}), \boldsymbol{y})}{\partial \boldsymbol{x}^*(\boldsymbol{y}) \partial \boldsymbol{x}^*(\boldsymbol{y})} \cdot \frac{\partial \boldsymbol{x}^*(\boldsymbol{y})}{\partial \boldsymbol{y}} = 0. \tag{5.11}$$

This leads to the indirect gradient $\frac{\partial \boldsymbol{x}^*(\boldsymbol{y})}{\partial \boldsymbol{y}}$ as

$$\frac{\partial \boldsymbol{x}^*(\boldsymbol{y})}{\partial \boldsymbol{y}} = -\left(\frac{\partial^2 \mathcal{L}(\boldsymbol{x}^*(\boldsymbol{y}), \boldsymbol{y})}{\partial \boldsymbol{x}^*(\boldsymbol{y}) \partial \boldsymbol{x}^*(\boldsymbol{y})}\right)^{-1} \frac{\partial^2 \mathcal{L}(\boldsymbol{x}^*(\boldsymbol{y}), \boldsymbol{y})}{\partial \boldsymbol{x}^*(\boldsymbol{y}) \partial \boldsymbol{y}}, \tag{5.12}$$

The strength of (5.12) is that we convert the indirect gradient among the variables $\boldsymbol{y}$ and $\boldsymbol{x}$ to direct gradients of $\mathcal{L}$ at the cost of an inversion of Hessian matrix. However, the weakness is that a Hessian is often too large to compute, thus it is common to solve a linear system leveraging the fast Hessian-vector product.

Example 5.4. Assume both UL and LU costs have a network with merely 1 million (10^6) parameters (32-bit float numbers), thus each network only needs a space of $10^6 \times 4\text{Byte} = 4\text{M}B$ to store, while their Hessian matrix needs a space of $(10^6)^2 \times 4\text{Byte} = 4\text{T}B$ to store. This indicates that a Hessian matrix cannot even be explicitly stored in the memory of a low-power computer, thus directly calculating its inversion is more impractical.

Recollect that our goal is to compute the gradient in (5.3), substituting (5.12) into (5.3) gives us:

$$\begin{aligned} \nabla_{\boldsymbol{y}} \mathcal{U} &= \frac{\partial \mathcal{U}(\boldsymbol{y}, \boldsymbol{x}^*)}{\partial \boldsymbol{y}} - \underbrace{\frac{\partial \mathcal{U}(\boldsymbol{y}, \boldsymbol{x}^*)}{\partial \boldsymbol{x}^*}}_{\boldsymbol{v}^\mathsf{T}} \underbrace{\left(\frac{\partial^2 \mathcal{L}(\boldsymbol{x}^*(\boldsymbol{y}), \boldsymbol{y})}{\partial \boldsymbol{x}^*(\boldsymbol{y}) \partial \boldsymbol{x}^*(\boldsymbol{y})}\right)^{-1}}_{(\boldsymbol{H}^\mathsf{T})^{-1}} \frac{\partial^2 \mathcal{L}(\boldsymbol{x}^*(\boldsymbol{y}), \boldsymbol{y})}{\partial \boldsymbol{x}^*(\boldsymbol{y}) \partial \boldsymbol{y}} \\ &= \frac{\partial \mathcal{U}(\boldsymbol{y}, \boldsymbol{x}^*)}{\partial \boldsymbol{y}} - \boldsymbol{q}^\mathsf{T} \cdot \frac{\partial^2 \mathcal{L}(\boldsymbol{x}^*(\boldsymbol{y}), \boldsymbol{y})}{\partial \boldsymbol{x}^*(\boldsymbol{y}) \partial \boldsymbol{y}} \end{aligned}. \tag{5.13}$$

Then we can solve the linear system $\boldsymbol{H}\boldsymbol{q} = \boldsymbol{v}$ for $\boldsymbol{q}^\mathsf{T}$ by optimizing

$$\boldsymbol{q}^* = \min \arg_{\boldsymbol{q}} Q(\boldsymbol{q}) = \min \arg_{\boldsymbol{q}} \frac{1}{2}\boldsymbol{q}^\mathsf{T}\boldsymbol{H}\boldsymbol{q} - \boldsymbol{q}^\mathsf{T}\boldsymbol{v}, \tag{5.14}$$

using efficient linear solvers such as a simple gradient descent or conjugate gradient method [113]. For gradient descent, we need to compute the gradient of Q as $\frac{\partial Q(\boldsymbol{q})}{\partial \boldsymbol{q}} = \boldsymbol{H}\boldsymbol{q} - \boldsymbol{v}$, where $\boldsymbol{H}\boldsymbol{q}$ can be computed using the fast Hessian-vector product, i.e., a Hessian-vector product is the gradient of a gradient-vector product:

$$\boldsymbol{H}\boldsymbol{q} = \frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{x} \partial \boldsymbol{x}} \cdot \boldsymbol{q} = \frac{\partial \left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}} \cdot \boldsymbol{q} \right)}{\partial \boldsymbol{x}}, \tag{5.15}$$

where $\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}} \cdot \boldsymbol{q}$ is a scalar. This means that the Hessian matrix $\boldsymbol{H}$ is not explicitly computed or stored. We summarize the computation of implicit differentiation with linear systems in Algorithm 2. The algorithm using a conjugate gradient is similar.

Algorithm 2 Computing *Implicit Differentiation* via Linear System.

1: **Input:** The current UL variable $\boldsymbol{y}$ and the optimal LL variable $\boldsymbol{x}^*$.
2: **Initialization:** $k = 1$, learning rate η.
3: **while** Not Convergent ($\|\boldsymbol{q}_k - \boldsymbol{q}_{k-1}\|$ is large enough) **do**
4: Perform gradient descent:

$$\boldsymbol{q}_k = \boldsymbol{q}_{k-1} - \eta \left(\boldsymbol{H}\boldsymbol{q}_{k-1} - \boldsymbol{v} \right), \tag{5.16}$$

where $\boldsymbol{H}\boldsymbol{q}_{k-1}$ is computed via the fast Hessian-vector product.
5: **end while**
6: Assign $\boldsymbol{q} = \boldsymbol{q}_k$
7: Compute $\nabla_{\boldsymbol{y}} \mathcal{U}$ in (5.3) as:

$$\nabla_{\boldsymbol{y}} \mathcal{U} = \frac{\partial \mathcal{U}(\boldsymbol{y}, \boldsymbol{x}^*)}{\partial \boldsymbol{y}} - \underbrace{\left(\frac{\partial^2 \mathcal{L}(\boldsymbol{x}^*(\boldsymbol{y}), \boldsymbol{y})}{\partial \boldsymbol{y} \partial \boldsymbol{x}^*(\boldsymbol{y})} \cdot \boldsymbol{q} \right)^\mathsf{T}}_{(\boldsymbol{H}_{\boldsymbol{y}\boldsymbol{x}} \cdot \boldsymbol{q})^\mathsf{T}}, \tag{5.17}$$

where $\boldsymbol{H}_{\boldsymbol{y}\boldsymbol{x}} \cdot \boldsymbol{q}$ can also be computed efficiently using the Hessian-vector product.

Approximations. Implicit differentiation is complicated to implement but there is one approximation, which is to ignore the implicit components and only use the direct part $\hat{\nabla}_{\boldsymbol{y}} U \approx \frac{\partial U}{\partial \boldsymbol{y}}\Big|_{\boldsymbol{x}_T}$. This is equivalent to taking the solution $\boldsymbol{x}_T$ from the LL optimization as *constants* in the UL problem. Such an approximation is more efficient but introduces an error term

$$\epsilon \sim \left| \frac{\partial U}{\partial \boldsymbol{x}^*} \frac{\partial \boldsymbol{x}^*}{\partial \boldsymbol{y}} \right|. \tag{5.18}$$

Nevertheless, it is useful when the implicit gradients contain products of *small* second-order derivatives, which depends on the specific NLS problems.

5.3 Differentiation on Manifold

Given that the state of a system within SLAM is bound to evolve on specific manifolds, optimization on manifolds plays a crucial role in solving back-end SLAM problems. We next derive the Jacobians required to differentiate with respect to variables belonging to Lie groups, which is an essential step for differentiation on the manifold.

5.3.1 Derivatives on the Lie Group

Since we introduced the basic concepts of Lie group, Lie algebra, and their basic operations (*e.g.,* exponential and logarithmic maps) in Chapter 3, this section will briefly recap those concepts but mainly focus on the definition of their derivatives, which is essential for solving a differentiable optimization problem.

Consider a Lie group's manifold $\mathcal{M}$, each point $\boldsymbol{\chi}$ on this smooth manifold possesses a unique tangent space, denoted by $T_{\boldsymbol{\chi}}\mathcal{M}$, where the fundamental principles of calculus are valid. The Lie algebra, represented as $\mathfrak{m}$, is a vector space that can be locally defined to the point $\boldsymbol{\chi}$ as $\mathfrak{m} = T_{\boldsymbol{\chi}}\mathcal{M}$. The exponential map $\exp : \mathfrak{m} \to \mathcal{M}$ projects elements from the Lie algebra to the Lie group, while the logarithmic map $\log : \mathcal{M} \to \mathfrak{m}$ serves as its inverse, establishing a bi-directional relationship:

$$\boldsymbol{\chi} = \exp(\boldsymbol{\tau}^\wedge) \Leftrightarrow \boldsymbol{\tau}^\wedge = \log(\boldsymbol{\chi}), \tag{5.19}$$

where hat $^\wedge$ is a linear invertible map, and $\boldsymbol{\tau}^\wedge \in \mathfrak{m}$. By representing the coordinates within the Lie algebra as vectors $\boldsymbol{\tau}$ in $\mathbb{R}^n$, we can define mappings between vector $\boldsymbol{\tau}$ and the Lie group $\boldsymbol{\chi}$:

$$\boldsymbol{\chi} = \operatorname{Exp}(\boldsymbol{\tau}) \Leftrightarrow \boldsymbol{\tau} = \operatorname{Log}(\boldsymbol{\chi}), \tag{5.20}$$

where we redefined the exponential and logarithm maps to directly use a vector as input and output, respectively.

To calculate derivatives on Lie groups, it is crucial to first understand the relative change between two manifold elements, say $\boldsymbol{\chi}_1$ and $\boldsymbol{\chi}_2$. These changes are quantified by first defining the $\oplus$ and $\ominus$ operators, which capture the concept of displacement on the manifold, as described in (5.21) and (5.22) below:

$$\begin{aligned} \boldsymbol{\chi}_2 &= \boldsymbol{\chi}_1 \oplus \boldsymbol{\tau} \triangleq \boldsymbol{\chi}_1 \circ \operatorname{Exp}(\boldsymbol{\tau}), \\ \boldsymbol{\tau} &= \boldsymbol{\chi}_2 \ominus \boldsymbol{\chi}_1 \triangleq \operatorname{Log}\left(\boldsymbol{\chi}_1^{-1} \circ \boldsymbol{\chi}_2\right). \end{aligned} \tag{5.21}$$

The placement of $\boldsymbol{\tau}$ on the right-hand side in (5.21) signifies that it is expressed in the local frame at $\boldsymbol{\chi}_1$. Conversely, the left operators in (5.22) reflect a global frame perspective:

$$\begin{aligned} \boldsymbol{\chi}_2 &= \boldsymbol{\varepsilon} \oplus \boldsymbol{\chi}_1 \triangleq \operatorname{Exp}(\boldsymbol{\varepsilon}) \circ \boldsymbol{\chi}_1, \\ \boldsymbol{\varepsilon} &= \boldsymbol{\chi}_2 \ominus \boldsymbol{\chi}_1 \triangleq \operatorname{Log}\left(\boldsymbol{\chi}_2 \circ \boldsymbol{\chi}_1^{-1}\right), \end{aligned} \tag{5.22}$$

where $\boldsymbol{\varepsilon}$ is expressed in the global frame. Both $\boldsymbol{\tau}$ and $\boldsymbol{\varepsilon}$ can be viewed as incremental perturbations to the manifold elements. By using corresponding composition operators $\oplus$ and $\ominus$, the variations are expressed as vectors in the tangent space.

With the right $\oplus$ and $\ominus$ operators in place, we use the Jacobian matrix $\boldsymbol{J}$ to describe perturbations on manifolds. The Jacobian captures the essence of infinitesimal perturbations $\boldsymbol{\tau}$ within the tangent space $\mathfrak{m}$:

$$\begin{aligned}\frac{\partial f(\boldsymbol{\chi})}{\partial \boldsymbol{\chi}} &\triangleq \lim_{\boldsymbol{\tau}\to 0} \frac{f(\boldsymbol{\chi}\oplus\boldsymbol{\tau})\ominus f(\boldsymbol{\chi})}{\boldsymbol{\tau}} \\ &= \lim_{\boldsymbol{\tau}\to 0} \frac{f(\boldsymbol{\chi}\circ \operatorname{Exp}(\boldsymbol{\tau}))\ominus f(\boldsymbol{\chi})}{\boldsymbol{\tau}} \\ &= \lim_{\boldsymbol{\tau}\to 0} \frac{\operatorname{Log}\left(f(\boldsymbol{\chi})^{-1}\circ f(\boldsymbol{\chi}\circ \operatorname{Exp}(\boldsymbol{\tau}))\right)}{\boldsymbol{\tau}}.\end{aligned} \tag{5.23}$$

Let $g(\boldsymbol{\tau}) = \operatorname{Log}\left(f(\boldsymbol{\chi})^{-1}\circ f(\boldsymbol{\chi}\circ \operatorname{Exp}(\boldsymbol{\tau}))\right)$, then the right Jacobian $\boldsymbol{J}_R$ can be expressed as the derivative of $g(\boldsymbol{\tau})$ at $\boldsymbol{\tau}=0$:

$$\frac{\partial f(\boldsymbol{\chi})}{\partial \boldsymbol{\chi}} = \boldsymbol{J}_R = \left.\frac{\partial g(\boldsymbol{\tau})}{\partial \boldsymbol{\tau}}\right|_{\boldsymbol{\tau}=0}. \tag{5.24}$$

In this way, the derivatives of $f(\boldsymbol{\chi})$ with respect to $\boldsymbol{\chi}$ in the manifold are represented by the Jacobian matrix $\boldsymbol{J}_R \in \mathbb{R}^{m\times n}$, where m and n are the dimensions of the Lie groups $\mathcal{M}$ and $\mathcal{N}$, respectively. The right Jacobian matrix performs a linear mapping from the tangent space $\mathfrak{m}$ to the tangent space $\mathfrak{n} = T_{f(\boldsymbol{\chi})}\mathcal{N}$.

Similarly, consider an infinitesimal perturbation $\boldsymbol{\varepsilon} \in T_g\mathcal{M}$ applied to the Lie group element $\boldsymbol{\chi}$, the left Jacobian $\boldsymbol{J}_L$ can be defined with the left plus and minus operators:

$$\begin{aligned}\frac{\partial f(\boldsymbol{\chi})}{\partial \boldsymbol{\chi}} &\triangleq \lim_{\boldsymbol{\varepsilon}\to 0} \frac{f(\boldsymbol{\varepsilon}\oplus\boldsymbol{\chi})\ominus f(\boldsymbol{\chi})}{\boldsymbol{\varepsilon}} \\ &= \lim_{\boldsymbol{\varepsilon}\to 0} \frac{f(\operatorname{Exp}(\boldsymbol{\varepsilon})\circ\boldsymbol{\chi})\ominus f(\boldsymbol{\chi})}{\boldsymbol{\varepsilon}} \\ &= \lim_{\boldsymbol{\varepsilon}\to 0} \frac{\operatorname{Log}\left(f(\operatorname{Exp}(\boldsymbol{\varepsilon})\circ\boldsymbol{\chi})\circ f(\boldsymbol{\chi})^{-1}\right)}{\boldsymbol{\varepsilon}} \\ &= \left.\frac{\partial \operatorname{Log}\left(f(\operatorname{Exp}(\boldsymbol{\varepsilon})\circ\boldsymbol{\chi})\circ f(\boldsymbol{\chi})^{-1}\right)}{\partial \boldsymbol{\varepsilon}}\right|_{\boldsymbol{\varepsilon}=0}.\end{aligned} \tag{5.25}$$

The resulting left Jacobian $\boldsymbol{J}_L \in \mathbb{R}^{n\times m}$ is also a linear mapping, but in the global tangent space from $T_g\mathcal{M}$ to $T_g\mathcal{N}$.

To delve into the local perturbations around a point $\boldsymbol{\chi}_1$, we consider perturbations $\boldsymbol{\tau}$ as $\boldsymbol{\tau} = \boldsymbol{\chi}\ominus\boldsymbol{\chi}_1$, with $\boldsymbol{\chi}$ being a perturbed version of $\boldsymbol{\chi}_1$. The covariance matrices $\boldsymbol{\Sigma}_{\boldsymbol{\chi}}$ defined on the tangent space are derived using the expectation operator $\mathbb{E}$, enabling the representation of uncertainties and their propagation:

$$\boldsymbol{\Sigma}_{\boldsymbol{\chi}} \triangleq \mathbb{E}[\boldsymbol{\tau}\boldsymbol{\tau}^\top] = \mathbb{E}[(\boldsymbol{\chi}\ominus\boldsymbol{\chi}_1)(\boldsymbol{\chi}\ominus\boldsymbol{\chi}_1)^\top]. \tag{5.26}$$

These covariance matrices facilitate the establishment of Gaussian distributions on the manifold, expressed as $\boldsymbol{\chi} \sim \mathcal{N}(\boldsymbol{\chi}_1, \boldsymbol{\Sigma}_{\boldsymbol{\chi}})$. It is important to note that the covariance matrices $\boldsymbol{\Sigma}_{\boldsymbol{\chi}}$ are defined on the tangent space $T_{\boldsymbol{\chi}_1}\mathcal{M}$, which allows the uncertainty in the manifold to be represented by a vector and be propagated in the form of covariance matrices.

Example 5.5. Consider a robot equipped with an inertial measurement unit (IMU) and a camera. Given noisy observations $\boldsymbol{R}_{\text{IMU}}$ and $\boldsymbol{R}_{\text{Cam}}$ from both sensors, the orientation of the robot can be estimated by minimizing the discrepancy between the measurements, which can be formulated as a nonlinear least squares problem on the manifold $SO(3)$:

$$\hat{\boldsymbol{R}} = \arg\min_{\boldsymbol{R} \in SO(3)} f(\boldsymbol{R}, \boldsymbol{R}_{\text{IMU}}, \boldsymbol{R}_{\text{Cam}}), \tag{5.27}$$

where $f(\cdot)$ is the cost function that quantifies the differences between the estimated orientation $\boldsymbol{R}$ and the sensor measurements $\boldsymbol{R}_{\text{IMU}}$ and $\boldsymbol{R}_{\text{Cam}}$. With the Jacobian matrices in place, the optimization on the manifold $SO(3)$ for the pose estimation can be effectively managed. The cost function $f(\cdot)$ can be detailed as:

$$f(\boldsymbol{R}) = \| \text{Log}\left(\boldsymbol{R}_{\text{IMU}}^{-1}\boldsymbol{R}\right) \|^2 + \| \text{Log}\left(\boldsymbol{R}_{\text{Cam}}^{-1}\boldsymbol{R}\right) \|^2. \tag{5.28}$$

To minimize $f(\boldsymbol{R})$, we need to compute its gradient with respect to $\boldsymbol{R}$ on the manifold $SO(3)$. The gradient can be derived using the right Jacobian $\boldsymbol{J}_R$ as:

$$\begin{aligned} \nabla f(\boldsymbol{R}) =& 2 \left(\frac{\partial \text{Log}\left(\boldsymbol{R}_{\text{IMU}}^{-1}\boldsymbol{R}\right)}{\partial \boldsymbol{R}} \right)^{\top} \text{Log}\left(\boldsymbol{R}_{\text{IMU}}^{-1}\boldsymbol{R}\right) \\ &+ 2 \left(\frac{\partial \text{Log}\left(\boldsymbol{R}_{\text{Cam}}^{-1}\boldsymbol{R}\right)}{\partial \boldsymbol{R}} \right)^{\top} \text{Log}\left(\boldsymbol{R}_{\text{Cam}}^{-1}\boldsymbol{R}\right). \end{aligned} \tag{5.29}$$

The gradient $\nabla f(\boldsymbol{R})$ can be used in conjunction with optimization algorithms like gradient descent which moves along the tangent space and reprojecting back to the manifold to update the pose $\boldsymbol{R}$ iteratively:

$$\boldsymbol{R}_{k+1} = \boldsymbol{R}_k \text{Exp}\left(-\alpha \nabla f(\boldsymbol{R})\right), \tag{5.30}$$

where α is the step size. This iterative process continues until the cost function $f(\boldsymbol{R})$ converges to a minimum, providing an optimal pose estimate $\hat{\boldsymbol{R}}$ that aligns with the sensor measurements.

5.3.2 Differentiation Operations on Manifold

For typical manifold operations, we can derive closed-form expressions for the Jacobians associated with inversion, composition, and group actions. These expressions

facilitate a comprehensive approach to optimization in SLAM, by enabling the computation of function derivatives on manifolds with the chain rule:

$$\frac{\partial \mathcal{Z}}{\partial \boldsymbol{\chi}} = \frac{\partial \mathcal{Z}}{\partial \mathcal{Y}} \frac{\partial \mathcal{Y}}{\partial \boldsymbol{\chi}}, \tag{5.31}$$

where $\mathcal{Z} = g(\mathcal{Y})$, and $\mathcal{Y} = f(\boldsymbol{\chi})$.

Jacobians of inversion can be derived through the application of the function $f(\boldsymbol{\chi}) = \boldsymbol{\chi}^{-1}$ with (5.23) for the right Jacobian $\boldsymbol{J}_R$, which leads to:

$$\begin{aligned} \frac{\partial \boldsymbol{\chi}^{-1}}{\partial \boldsymbol{\chi}} &\triangleq \lim_{\boldsymbol{\tau} \to 0} \frac{\operatorname{Log}\left((\boldsymbol{\chi}^{-1})^{-1} (\boldsymbol{\chi} \operatorname{Exp}(\boldsymbol{\tau}))^{-1}\right)}{\boldsymbol{\tau}} \\ &= \lim_{\boldsymbol{\tau} \to 0} \frac{\operatorname{Log}\left(\boldsymbol{\chi} \operatorname{Exp}(\boldsymbol{\tau})^{-1} \boldsymbol{\chi}^{-1}\right)}{\boldsymbol{\tau}} \\ &= \lim_{\boldsymbol{\tau} \to 0} \frac{(\boldsymbol{\chi}(-\boldsymbol{\tau})^{\wedge} \boldsymbol{\chi}^{-1})^{\vee}}{\boldsymbol{\tau}}. \end{aligned} \tag{5.32}$$

Jacobians of composition can be derived through the application of the function $f(\boldsymbol{\chi}) = \boldsymbol{\chi} \circ \boldsymbol{\chi}_1$ with the Equation (5.23). The derivative of the composition operator $\boldsymbol{\chi} \circ \boldsymbol{\chi}_1$ with respect to $\boldsymbol{\chi}$ is:

$$\begin{aligned} \frac{\partial (\boldsymbol{\chi} \circ \boldsymbol{\chi}_1)}{\partial \boldsymbol{\chi}} &\triangleq \lim_{\boldsymbol{\tau} \to 0} \frac{\operatorname{Log}\left((\boldsymbol{\chi}\boldsymbol{\chi}_1)^{-1} (\boldsymbol{\chi} \operatorname{Exp}(\boldsymbol{\tau}) \boldsymbol{\chi}_1)\right)}{\boldsymbol{\tau}} \\ &= \lim_{\boldsymbol{\tau} \to 0} \frac{\operatorname{Log}\left(\boldsymbol{\chi}_1^{-1} \operatorname{Exp}(\boldsymbol{\tau}) \boldsymbol{\chi}_1\right)}{\boldsymbol{\tau}} \\ &= \lim_{\boldsymbol{\tau} \to 0} \frac{(\boldsymbol{\chi}_1^{-1} \boldsymbol{\tau}^{\wedge} \boldsymbol{\chi}_1)^{\vee}}{\boldsymbol{\tau}}. \end{aligned} \tag{5.33}$$

The derivative of the composition operator $\boldsymbol{\chi} \circ \boldsymbol{\chi}_1$ with respect to $\boldsymbol{\chi}_1$ is:

$$\begin{aligned} \frac{\partial (\boldsymbol{\chi} \circ \boldsymbol{\chi}_1)}{\partial \boldsymbol{\chi}_1} &\triangleq \lim_{\boldsymbol{\tau} \to 0} \frac{\operatorname{Log}\left((\boldsymbol{\chi}\boldsymbol{\chi}_1)^{-1} (\boldsymbol{\chi}\boldsymbol{\chi}_1 \operatorname{Exp}(\boldsymbol{\tau}))\right)}{\boldsymbol{\tau}} \\ &= \lim_{\boldsymbol{\tau} \to 0} \frac{\operatorname{Log}(\operatorname{Exp}(\boldsymbol{\tau}))}{\boldsymbol{\tau}} \\ &= \boldsymbol{I}. \end{aligned} \tag{5.34}$$

Jacobians of the manifold $\mathcal{M}$ are characterized by the right Jacobian of $\boldsymbol{\chi}$ which is derived from the exponential map of $\boldsymbol{\tau} \in \mathbb{R}^m$. This is expressed as:

$$\boldsymbol{J}_r(\boldsymbol{\tau}) \triangleq \frac{^{\boldsymbol{\tau}}\partial \operatorname{Exp}(\boldsymbol{\tau})}{\partial \boldsymbol{\tau}}. \tag{5.35}$$

The right Jacobian conveys minor changes in $\boldsymbol{\tau}$ to modifications in the local tangent space at $\operatorname{Exp}(\boldsymbol{\tau})$. Similarly, the left Jacobian of $\boldsymbol{\chi}$ maps changes of $\boldsymbol{\tau}$ to variations within the global tangent space of the manifold. This is expressed as:

$$\boldsymbol{J}_l(\boldsymbol{\tau}) \triangleq \frac{^{\boldsymbol{\epsilon}}\partial \operatorname{Exp}(\boldsymbol{\tau})}{\partial \boldsymbol{\tau}}. \tag{5.36}$$

Jacobians of group action depends on the specific group action set $v \in \mathcal{V}$. The group action is defined as:

$$\begin{aligned} \boldsymbol{J}_{\boldsymbol{\chi}}^{\boldsymbol{\chi} \cdot v} &\triangleq \frac{{}^{\boldsymbol{\chi}} D \boldsymbol{\chi} \cdot v}{D \boldsymbol{\chi}}, \\ \boldsymbol{J}_{v}^{\boldsymbol{\chi} \cdot v} &\triangleq \frac{{}^{v} D \boldsymbol{\chi} \cdot v}{D v}, \end{aligned} \tag{5.37}$$

where $\boldsymbol{\chi} \in \mathcal{M}$ and $v \in \mathcal{V}$.

Example 5.6. Consider a robotic arm with two joints, $\boldsymbol{R}_1$ and $\boldsymbol{R}_2$, each represented by an element in $SO(3)$. The final orientation of the robot's end-effector is determined by the composition of the joint rotations:

$$\boldsymbol{R} = \boldsymbol{R}_1 \circ \boldsymbol{R}_2. \tag{5.38}$$

To evaluate the impact of small perturbations $\boldsymbol{\tau}$ in $\boldsymbol{R}_1$ and $\boldsymbol{R}_2$ on the end-effector orientation $\boldsymbol{R}$. It can be quantified using the Jacobians of composition:

$$\begin{aligned} \frac{\partial (\boldsymbol{R}_1 \circ \boldsymbol{R}_2)}{\partial \boldsymbol{R}_1} &= \lim_{\boldsymbol{\tau} \to 0} \frac{(\boldsymbol{R}_2^{-1} \boldsymbol{\tau}^{\wedge} \boldsymbol{R}_2)^{\vee}}{\boldsymbol{\tau}}, \\ \frac{\partial (\boldsymbol{R}_1 \circ \boldsymbol{R}_2)}{\partial \boldsymbol{R}_2} &= \boldsymbol{I}. \end{aligned} \tag{5.39}$$

This example implies that adjustments to the first joint $\boldsymbol{R}_1$ affect the final orientation $\boldsymbol{R}$ through a transformation influenced by the current state of the second joint $\boldsymbol{R}_2$. However, changes in the second joint $\boldsymbol{R}_2$ directly impact $\boldsymbol{R}$ without being influenced by the first joint $\boldsymbol{R}_1$.

5.4 Modern Libraries

5.4.1 Numerical Challenges of Automatic Differentiation

Automatic Differentiation (AutoDiff) is a cornerstone technique for computing derivatives accurately and efficiently in various optimization contexts, including differentiation on manifolds. Differentiation on manifolds poses unique challenges due to the complex geometrical properties inherent in manifold structures, which can affect the performance and applicability of AutoDiff. In differential optimization, these challenges become pronounced as AutoDiff interacts with the curved space of manifolds, potentially introducing numerical instability and inaccuracies.

This section delves into the specific numerical issues that arise when using automatic differentiation for manifold-based optimization tasks. Particular attention will be paid to the complexities involved in maintaining numerical stability and precision in the presence of manifold constraints, such as those found in constrained optimization and in systems defined by differential equations on manifolds. For simplicity, we will take the PyPose library [274] as an example, which defines a general

data structure, `LieTensor` for Lie Group and Lie Algebra. Specifically, we will show its numerical challenges and how PyPose tackle this challenge.

Analytical Foundations of Exponential Mapping to Quaternions. The exponential map is a fundamental concept in the theory of Lie groups and is particularly critical when transitioning between Lie algebras and Lie groups represented by quaternions. This mapping enables the translation of angular velocities from the algebraic structure in $\mathbb{R}^3$ to rotational orientations in the group of unit quaternions $\mathbb{S}^3$. Analytically, the exponential map for quaternions is derived from the Rodrigues' rotation formula, which relates a vector in $\mathbb{R}^3$ to the corresponding rotation. Given a vector $\boldsymbol{x}$ in $\mathbb{R}^3$, representing the axis of rotation scaled by the rotation angle, the quaternion representation of the rotation is given by:

$$\texttt{Exp}(\boldsymbol{\nu}) = \left[\sin\left(\frac{\|\boldsymbol{\nu}\|}{2}\right)\frac{\boldsymbol{\nu}^\top}{\|\boldsymbol{\nu}\|},\ \cos\left(\frac{\|\boldsymbol{x}\|}{2}\right)\right]^\top \tag{5.40}$$

where $\|\boldsymbol{\nu}\|$ represents the magnitude of $\boldsymbol{\nu}$, corresponding to the angle of rotation, and $\frac{\boldsymbol{\nu}}{\|\boldsymbol{\nu}\|}$ is the unit vector in the direction of $\boldsymbol{\nu}$.

One of the challenges of implementing a differentiable `LieTensor` is that one often need to calculate numerically problematic terms such as $\frac{\sin\boldsymbol{\nu}}{\boldsymbol{\nu}}$ in (5.40) for the Exp and Log mapping [257]. The direct computation of sine and cosine functions for very small angles can lead to precision issues due to the finite representation of floating-point numbers in computer systems. To manage these issues and maintain numerical stability, PyPose takes the Taylor expansion to avoid calculating the division by zero.

$$\texttt{Exp}(\boldsymbol{\nu}) = \begin{cases} \left[\boldsymbol{\nu}^T\gamma_e,\ \cos(\frac{\|\boldsymbol{\nu}\|}{2})\right]^T & \text{if } \|\boldsymbol{\nu}\| > \text{eps} \\ \left[\boldsymbol{\nu}^T\gamma_o,\ 1-\frac{\|\boldsymbol{\nu}\|^2}{8}+\frac{\|\boldsymbol{\nu}\|^4}{384}\right]^T & \text{otherwise,} \end{cases} \tag{5.41}$$

where $\gamma_e = \frac{\sin(\frac{\|\boldsymbol{\nu}\|}{2})}{\|\boldsymbol{\nu}\|}$ when $\|\boldsymbol{\nu}\|$ is significant, and $\gamma_o = \frac{1}{2} - \frac{\|\boldsymbol{\nu}\|^2}{48} + \frac{\|\boldsymbol{\nu}\|^4}{3840}$ for small $\|\boldsymbol{\nu}\|$, ensuring precise calculations across all ranges of rotation magnitudes. Here, eps is the smallest machine number where $1 + \text{eps} \neq 1$. This analytical-to-numerical progression demonstrates the importance of accurate and stable methods for computing exponential maps in applications that require high fidelity in rotation representation, such as in 3D graphics, robotics, and aerospace engineering.

`LieTensor` is different from the existing libraries in several aspects: (**1**) PyPose supports auto-diff for any order gradient and is compatible with most popular devices, such as CPU, GPU, TPU, and Apple silicon GPU, while other libraries like LieTorch [257] implement customized CUDA kernels and only support 1^{st}-order gradient. (**2**) `LieTensor` supports parallel computing of gradient with the `vmap` operator, which allows it to compute Jacobian matrices much faster. (**3**) Libraries such as LieTorch, JaxLie [303], and Theseus only support Lie groups, while Py-

Pose supports both Lie groups and Lie algebras. As a result, one can directly call the `Exp` and `Log` maps from a `LieTensor` instance, which is more flexible and user-friendly. Moreover, the gradient with respect to both types can be automatically calculated and back-propagated. The readers may find a list of supported `LieTensor` operations in [2] and the tutorial of PyPose is available in [4]. The usages of a `LieTensor` and its automatic differentiation can be found at `https://github.com/pypose/slambook-snippets/blob/main/lietensor.ipynb`.

5.4.2 Implementation of Differentiable Optimization

To enable end-to-end learning with bilevel optimization, one need to integrate general optimizers beyond the gradient-based methods such as SGD [222] and Adam [137] required by neural methods, since many problems in SLAM such as bundle adjustment and factor graph optimization require other optimizations algorithms such as constrained or 2^{nd}-order optimization [19]. Moreover, practical problems have outliers, hence one needs to robustify the loss as described in Chapter 4. Next we consider an Iteratively Reweighted Least Squares (IRLS) approach to SLAM as introduced in Section 4.3, and present the intuition behind the optimization-oriented interfaces of PyPose, including `solver`, `kernel`, `corrector`, and `strategy` for using the 2^{nd}-order Levenberg-Marquardt (`LM`) optimizer.

Let us start by considering a weighted least square problem:

$$\min_{\boldsymbol{y}} \sum_i \left(\boldsymbol{h}_i(\boldsymbol{x}_i) - \boldsymbol{z}_i\right)^T \boldsymbol{\Sigma}_i \left(\boldsymbol{h}_i(\boldsymbol{x}_i) - \boldsymbol{z}_i\right), \tag{5.42}$$

where $\boldsymbol{h}(\cdot)$ is a regression model (`Module`), $\boldsymbol{x} \in \mathbb{R}^n$ is the parameters to be optimized, $\boldsymbol{h}_i$ deontes prediction for the i-th input sample, $\boldsymbol{\Sigma}_i \in \mathbb{R}^{d\times d}$ is a square information matrix. The solution to (5.42) of an `LM` algorithm is computed by iteratively updating an estimate $\boldsymbol{x}_t$ via $\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} + \boldsymbol{\delta}_t$, where the update step $\boldsymbol{\delta}_t$ is computed as:

$$\sum_i \left(\boldsymbol{\Lambda}_i + \lambda \cdot \operatorname{diag}(\boldsymbol{\Lambda}_i)\right) \boldsymbol{\delta}_t = -\sum_i \boldsymbol{J}_i^T \boldsymbol{\Sigma}_i \mathbf{r}_i, \tag{5.43}$$

where $\mathbf{r}_i = \boldsymbol{h}_i(\boldsymbol{x}_i) - \boldsymbol{z}_i$ is the i-th residual error, $\boldsymbol{J}_i$ is the Jacobian of $\boldsymbol{h}$ computed at $\boldsymbol{x}_{t-1}$, $\boldsymbol{\Lambda}_i$ is an approximated Hessian matrix computed as $\boldsymbol{\Lambda}_i = \boldsymbol{J}_i^T \boldsymbol{\Sigma}_i \boldsymbol{J}_i$, and λ is a damping factor. To find step $\boldsymbol{\delta}_t$, one needs a linear `solver`:

$$\mathbf{A} \cdot \boldsymbol{\delta}_t = \boldsymbol{\beta}, \tag{5.44}$$

where $\mathbf{A} = \sum_i \left(\boldsymbol{\Lambda}_i + \lambda \cdot \operatorname{diag}(\boldsymbol{\Lambda}_i)\right)$, $\boldsymbol{\beta} = -\sum_i \boldsymbol{J}_i^T \boldsymbol{\Sigma}_i \mathbf{r}_i$. In practice, the square matrix $\mathbf{A}$ is often positive-definite, so we could leverage standard linear solvers such as Cholesky. If the Jacobian $\boldsymbol{J}_i$ is large and sparse, we may also use sparse solvers such as sparse Cholesky [46] or preconditioned conjugate gradient (PCG) [113] solver. In practice, one often introduces robust `kernel` functions $\rho : \mathbb{R} \mapsto \mathbb{R}$ into (5.42) to

reduce the effect of outliers:

$$\min_{\boldsymbol{y}} \sum_i \rho\left(\mathbf{r}_i^T \boldsymbol{\Sigma}_i \mathbf{r}_i\right), \tag{5.45}$$

where ρ is designed to down-weigh measurements with large residuals $\mathbf{r}_i$. In this case, we need to adjust (5.43) to account for the presence of the robust kernel. A popular way is to use an IRLS method, Triggs' correction [264], which is also adopted by the Ceres [9] library. However, it needs 2^{nd}-order derivative of the kernel function ρ, which is always negative. This can lead 2^{nd}-order optimizers including LM to be unstable [264]. Alternatively, PyPose introduces an IRLS method, `FastTriggs`, which is faster yet more stable than `Triggs` by only involving the 1^{st}-order derivative:

$$\mathbf{r}_i^\rho = \sqrt{\rho'(c_i)}\mathbf{r}_i, \quad \boldsymbol{J}_i^\rho = \sqrt{\rho'(c_i)}\boldsymbol{J}_i, \tag{5.46}$$

where $c_i = \mathbf{r}_i^T \boldsymbol{\Sigma}_i \mathbf{r}_i$, $\mathbf{r}_i^\rho$ and $\boldsymbol{J}_i^\rho$ are the corrected model residual and Jacobian due to the introduction of kernel functions, respectively. More details about `FastTriggs` and its proof can be found in [1], while IRLS was introduced in Section 4.3.

A simple `LM` optimizer may not converge to the global optimum if the initial guess is too far from the optimum. For this reason, we often need other strategies such as adaptive damping, dogleg, and trust region methods [158] to restrict each step, preventing it from stepping "too far". To adopt those strategies, one may simply pass a `strategy` instance, e.g., `TrustRegion`, to an optimizer. In summary, PyPose supports easy extensions for the aforementioned algorithms by simply passing `optimizer` arguments to their constructor, including `solver`, `strategy`, `kernel`, and `corrector`. A list of available algorithms and examples can be found in [3]. The usages of a 2nd-order optimizatoin can be found at `https://github.com/pypose/slambook-snippets/blob/main/optimization.ipynb`.

5.4.3 Related Open-source Libraries

Open-source libraries related to differentiable optimization can be divided into three groups: (1) linear algebra, (2) machine learning libraries, and (3) specialized optimization libraries.

Linear Algebra Libraries are essential to machine learning and robotics research. NumPy [194], a linear algebra library for Python, offers comprehensive operations on vectors and matrices while enjoying higher running speed due to its underlying well-optimized C code. Eigen [105], a high performance C++ linear algebra library, has been used in many projects such as TensorFlow [5], Ceres [9], GTSAM [66], and g^{2}o [104]. ArrayFire [168], a GPU acceleration library for C, C++, Fortran, and Python, contains simple APIs and provides GPU-tuned functions.

Machine Learning Libraries focus more on operations on tensors (i.e., high-dimensional matrices) and automatic differentiation. Early machine learning frameworks, such as Torch [59], OpenNN [197], and MATLAB [172], provide primitive tools for researchers to develop neural networks. However, they only support CPU computation and lack concise APIs, which plague engineers using them in applications. A few years later, deep learning frameworks such as Chainer [262], Theano [11], and Caffe [123] arose to handle the increasing size and complexity of neural networks while supporting multi-GPU training with convenient APIs for users to build and train their neural networks. Furthermore, the recent frameworks, such as TensorFlow [5], PyTorch [202], and MXNet [52], provide a comprehensive and flexible ecosystem (e.g., APIs for multiple programming languages, distributed data parallel training, and facilitating tools for benchmark and deployment). Gvnn [109] introduced differentiable transformation layers into Torch-based framework, leading to end-to-end geometric learning. JAX [37] can automatically differentiate native Python and NumPy functions and is an extensible system for composable function transformations. In many ways, the existence of these frameworks facilitated and promoted the growth of deep learning. Recently, more efforts have been taken to combine standard optimization tools with deep learning. Recent work like Theseus [211] and CvxpyLayer [10] showed how to embed differentiable optimization within deep neural networks. PyPose [274] incorporates 2^{nd}-order optimizers such as Gaussian-Newton and Levenberg-Marquardt and can compute any order gradients of Lie groups and Lie algebras, which are essential to robotics.

Other Specialized Optimization Libraries have been developed and leveraged in robotics. To mention a few, Ceres [9] is an open-source C++ library for large-scale nonlinear least squares optimization problems and has been widely used in SLAM. Pyomo [110] and JuMP [78] are optimization frameworks that have been widely used due to their flexibility in supporting a diverse set of tools for constructing, solving, and analyzing optimization models. CasADi [13] has been used to solve many real-world control problems in robotics due to its fast and effective implementations of different numerical methods for optimal control. Pose- and factor-graph optimization also play an important role in robotics. For example, g^2o [104] and GTSAM [66] are open-source C++ frameworks for graph-based nonlinear optimization, which provide concise APIs for constructing new problems and have been leveraged to solve several optimization problems in SLAM.

Optimization libraries have also been widely used in robotic control problems. To name a few, IPOPT [273] is an open-source C++ solver for nonlinear programming problems based on interior-point methods and is widely used in robotics and control. Similarly, OpenOCL [138] supports a large class of optimization problems such as continuous time, discrete time, constrained, unconstrained, multi-phase, and trajectory optimization problems for real-time model-predictive control. Another library for large-scale optimal control and estimation problems is CT [100], which provides standard interfaces for different optimal control solvers and can be

extended to a broad class of dynamical systems in robotic applications. Drake [254] has solvers for common control problems and that can be directly integrated with its simulation tool boxes. Its system completeness made it favorable to researchers.

5.5 Final Considerations & Recent Trends

Deep learning methods have witnessed significant development in recent years [308]. As data-driven approaches, they are believed to perform better on visual tracking than traditional handcrafted features. Most studies on the subject employed end-to-end structures, including both supervised methods such DeepVO [279] and TartanVO [281] and unsupervised methods such as UnDeepVO [151] and Unsupervised VIO [283]. It is generally observed that the supervised approaches achieve higher performance compared to their unsupervised counterparts since they can learn from a diverse range of ground truths such as pose, flow, and depth. Nevertheless, obtaining such ground truths in the real world is a labor-consuming process [280].

Recently, hybrid methods have received increasing attention as they integrate the strengths of both geometry-based and deep-learning approaches. Several studies have explored the potential of integrating Bundle Adjustment (BA) with deep learning methods to impose topological consistency between frames, such as attaching a BA layer to a learning network such as BA-Net [251] and DROID-SLAM [256]. Additionally, some works focused on compressing image features into codes (embedded features) and optimizing the pose-code graph during inference such as DeepFactors [62]. Furthermore, DiffPoseNet [199] is proposed to predict poses and normal flows using networks and fine-tune the coarse predictions through a Cheirality layer. However, in these works, the learning-based methods and geometry-based optimization are decoupled and separately used in different sub-modules. The lack of integration between the front-end and back-end may result in sub-optimal performance. Besides, they only back-propagate the pose error "through" bundle adjustment, thus the supervision is from the ground truth poses. In this case, BA is just a special layer of the network. Recently, iSLAM [94] connects the front-end and back-end bidirectionally and enforces the learning model to learn from geometric optimization through a bilevel optimization framework, which achieves performance improvement without external supervision. Some other tasks can also be formulated as bilevel optimization, *e.g.,* reinforcement learning [242], local planning [291], global planning [53], feature matching [306], and multi-robot routing [106].

6

Dense Map Representations

Victor Reijgwart, Jens Behley, Teresa Vidal-Calleja, Helen Oleynikova,
Lionel Ott, Cyrill Stachniss and Ayoung Kim

We now shift our focus to a different aspect of SLAM, specifically its mapping component. The mapping problem is approached with the assumption that the robot's pose is known, and the objective is to construct a dense map of its surroundings. Indeed, typical approaches first solve for the robot trajectory using the SLAM back-end —as discussed in the previous chapters— and then reconstruct a dense map given the trajectory. In this chapter, we illustrate the details of the dense map representation, focusing on the map elements, data structures, and methods.

Early mapping approaches were predominantly based on sparse, landmark-based solutions as discussed in Chapter 2 that extract only a few salient features from the environment. However, the increase in compute capabilities paired with the advent of accurate 3D range sensors, such as mechanical 3D LiDARs or RGB-D cameras that provide detailed 3D range measurements at high frequencies, led to an increasing research interest in dense map representations. Dense maps are crucial for downstream tasks that require a detailed understanding of the environment, such as planning, navigation, manipulation and precise localization. This chapter explains how these dense methods leverage the full spectrum of range sensor data to refine and update comprehensive maps.

The chapter begins by presenting key sensor types that facilitate dense mapping, primarily focusing on range sensors that produce detailed range measurements. The chapter continues with an introduction to fundamental representation elements and data structures in Section 6.2, that we then tailor to specific applications in Section 6.4. Contrasting with sparse landmark-based mapping, the choice of a dense map representation hinges on the sensor types used and the intended downstream applications. Key factors influencing the selection of the representation type are summarized in Section 6.5.

6.1 Range Sensing Preliminaries

Before we delve into dense map representations, we briefly summarize a key sensing modality, range sensors, often used for SLAM, providing the necessary context for the following discussion. Such sensors produce range measurements to the objects

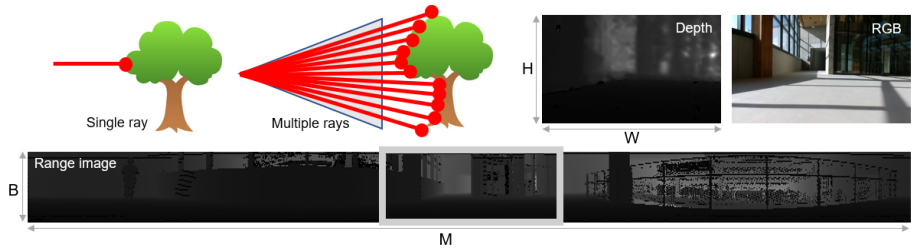


Figure 6.1 Single ray and multi-ray types for LiDAR sensors. Sample data from real-world is visualized to show RGB, depth, and LiDAR range image.

in the environment, including LiDAR sensors, time-of-flight (TOF) cameras, RGB-D cameras, and stereo cameras. Here, we concentrate on the most commonly used LiDAR sensors and RGB-D cameras that are predominately used in outdoor and indoor environments for SLAM and dense mapping.

6.1.1 Sensor Measurement Model

Let us start with a brief summary of the sensing mechanism and associated measurement model. In the case of LiDAR sensors,[1] the range measurements are generated using laser beams that are emitted, reflected by the environment, and then detected [224]. By measuring the time t_{emit} when the laser beam is emitted and the time of detection t_{detect}, we can derive the range r using the speed of light c as follows:

$$r = \frac{c\,(t_{\text{detect}} - t_{\text{emit}})}{2}. \tag{6.1}$$

A single ray measurement can be enhanced into a two-dimensional (2D) or three-dimensional (3D) collection of points by employing an array of rays that move in a designated pattern, such as a 360-degree rotation or a specific shape. The collection of points generated by the sensor is referred to as point clouds, which serve as the fundamental element for creating maps. The LiDAR measurements can also be represented using a range image $\boldsymbol{R} \in \mathbb{R}^{B \times M}$, where the range of each of the B beams is stored for a single complete turn of the sensor, *i.e.,* a complete 360° rotation. Thus, we have M measurements in the horizontal field of view of the sensor for each of the B beams.

Another commonly used range sensor in robotics applications are RGB-D cameras, such as Microsoft's Kinect and Azure Kinect DK, and Intel's RealSense. RGB-D cameras provide besides the RGB image $\boldsymbol{I}_{\text{RGB}} \in \mathbb{R}^{3 \times H \times W}$ of height H and width W, a depth map $\boldsymbol{I}_D \in \mathbb{R}^{H \times W}$ of the same dimension, where each pixel location contains the depth or range. To generate the depth map $\boldsymbol{I}_D$, early RGB-D cameras

[1] We illustrate this chapter using simple 2D and mechanically rotating 3D LiDARs, while other solid-state and flash LiDARs will be introduced in Chapter 9

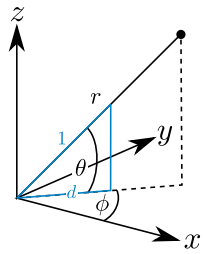
use a structured infrared (IR) light source with a known pattern that is projected onto the environment. The distance of individual pixels is then determined from the distortion of the known pattern. As sunlight usually interferes with this sensing mechanism, such RGB-D cameras using projected light are mainly used in indoor environments. Fortunately, newer generations of RGB-D sensors introduce an IR texture projector or TOF less affected by interferences, supporting outdoor applications.

6.1.2 Conversion to Point Cloud

Using the intrinsics of a range sensor (*e.g.,* a LiDAR or RGB-D camera), we can convert a range image $\boldsymbol{R}$ or a depth map $\boldsymbol{I}_D$ into a point cloud $\mathsf{P} = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_N\}$, where points $\boldsymbol{p}_i \in \mathbb{R}^3$ are expressed in the local coordinate frame of the sensor. The point is the most fundamental unit in the map representation and will be discussed further in this chapter.

For conversion from LiDAR, the sensors provide an intrinsic calibration for each beam $(\phi_{i,j}, \theta_{i,j})$, where $1 \leq i < B$ and $1 \leq j < M$, consisting of the azimuthal angle $\phi_{i,j} \in [0, 2\pi]$ and polar/inclination angle $\theta_{i,j} \in [-\pi, \pi]$ as depicted in Figure 6.2. Using these known angles of each beam, we can convert a range measurement $r_{i,j}$ at $\boldsymbol{R}_{i,j}$ into a three-dimensional point $\boldsymbol{p} = (x, y, z)$ as follows:

Figure 6.2



$$x = r_{i,j} \cos(\theta_{i,j}) \cos(\phi_{i,j}) \tag{6.2}$$

$$y = r_{i,j} \cos(\theta_{i,j}) \sin(\phi_{i,j}) \tag{6.3}$$

$$z = r_{i,j} \sin(\theta_{i,j}) \tag{6.4}$$

For an RGB-D camera, we commonly use a pinhole camera model to convert the ranges $r_{u,v} = \boldsymbol{R}_{u,v}$ at pixel location (u, v) into a three-dimensional coordinate. For this, we use the intrinsics of the camera $\boldsymbol{K} \in \Re^{3\times4}$ to convert a homogeneous coordinate $\boldsymbol{x} = (u, v, r_{u,v})$ in the image coordinate into a point $\boldsymbol{p}$ in the camera coordinate:

$$\boldsymbol{p} = \boldsymbol{K}^{-1}\boldsymbol{x}. \tag{6.5}$$

The resulting point cloud is said to be *organized* or *unorganized* depending on how the points are structured. When converted from a depth map, the point cloud is organized, and each point location is structured with respect to the pixel location of the associate depth map. This can be exploited to compute neighboring points by a simple indexing. On the other hand, the point cloud generated by a LiDAR sensor is more complicated. For example, the generated point cloud is organized in the static 3D mechanical LiDAR. However, the organization of the point cloud no longer

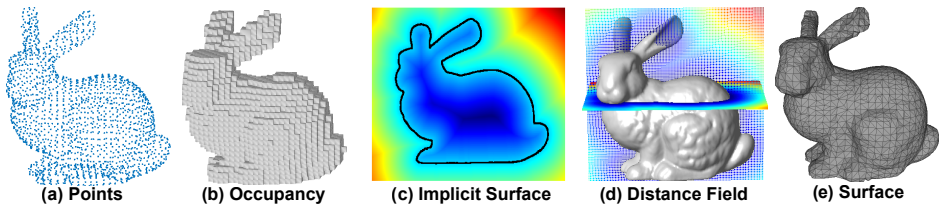


Figure 6.3 Examples of common dense representations.

holds in the case of non-repeated pattern solid-state, flash LiDARs, or mechanically rotating LiDAR under motion distortion. In many cases, the unorganized points are distorted due to the movement of the sensor and measurement neighborhood in the range image is not necessarily correlated to spatial neighborhood. Therefore, an estimate of the motion of the sensor while completing a sweep, *e.g.,* inertial measurement unit (IMU) measurements or odometry information, together with the information of the per-beam time is necessary to undistort an LiDAR point cloud to account for the motion of the sensor [272, 70, 300]. For more details on motion distortion and compensation, refer to Chapter 9.

6.2 Foundations of Mapping

A map, generated with sensors' information and data processing approaches, is a symbolic structure that models the environment [260, 41]. One thing to be noted here is that the map representation can be diverse, and many different representations exist for the same spatial information (as in Figure 6.3). The choice and accuracy of the scene representation strongly impact the performance of the task at hand, and thus the representation should be determined by the use case. For instance, motion estimation and localization in robotics favor sparse representation, such as 3D points features [182, 214] in order to exploit these features for consistent robot pose estimation. On the other hand, a key objective of scene reconstruction is an accurate, dense, and high-resolution map, for example, for inspection purposes [119, 193, 219]. Similarly, path planning tasks require dense information such as obstacle occupancy or closest distance to collision for obstacles avoidance [179, 83, 266]. Overall, this chapter examines the following three questions.

Q1. What quantity do we need to estimate for dense mapping? The most commonly used quantities to represent the environment are *occupancy* and *distance*. Occupancy is a key property in mapping for distinguishing between free and occupied space. Distance estimation provides a more robot-centric interpretation of free and occupied space by measuring the range to nearby surfaces or objects.

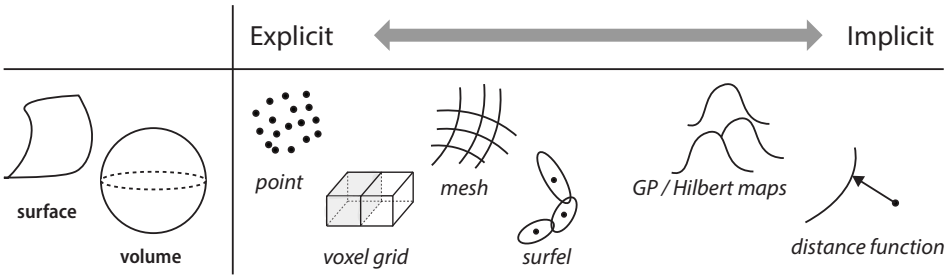


Figure 6.4 The representation can be either explicit or implicit. The illustration is simplified for clarity. In reality, the abstraction is not clearly separable and can often be applied in a combined manner.

Q2. How should the world be represented? This is the question of what space abstraction we use for representation. Broadly, representation can be either explicit or implicit. Explicit space abstractions are further classified based on the type of geometry they utilize, while implicit representation can be categorized based on their choice of functions. The list of abstraction types are illustrated in Figure 6.4.

Q3. What data structure and storage should be used? The chosen representation should be stored in memory for later use. The data structure and storage method should be selected based on the specific application and intended usage.

In the literature, a wide variety of approaches exist for generating a dense representation of the scene using range sensors, varying in terms of their estimated quantities, space abstraction, storage structure, continuity, and application areas.

Beginning the discussion on different representations, we first explore the primary quantities estimated from range measurements. The focus is on understanding the key quantities predominantly estimated in the mapping phase. We will provide a concise overview of the basic definitions of each quantity, elaborating their significance and the specific contexts in which they play a crucial role.

6.2.1 Occupancy Maps

Since their introduction over three decades ago by Elfes and Moravec [83, 179], occupancy grids have been widely used. Their simplicity and computational efficiency have made occupancy grids[2] popular when mapping indoor (and even outdoor) environments. In the simplest scenario, the estimated quantity is the *probability* of a cell being occupied. In this case, the occupancy of the cell is modeled as a probability of that cell containing an obstacle, with occupancy equal to 1 for occu-

[2] Occupancy mapping can be conducted without grid-based methods (e.g., GPOM [190]). In this chapter, we will focus on grid-based mapping for simplicity.

pied cells and 0 for cells deemed empty. Essentially occupancy mapping is a binary classification problem to predict the binary class probability of each cell.

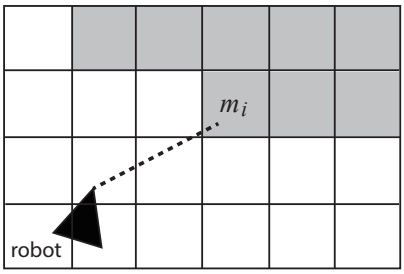


Figure 6.5 A simple ray-casting example in 2D. Given a range measurement at a certain (computed or estimated) azimuth, the return of the range measurement indicates the existence of an obstacle.

Given a set of sensor measurements $\boldsymbol{z}_{1:t}$ and a set of sensor poses $\boldsymbol{x}_{1:t}$, the probability of being occupied for each cell in the map $\boldsymbol{m}$ is modeled as $p(\boldsymbol{m}|\boldsymbol{z}_{1:t}, \boldsymbol{x}_{1:t})$. A sample map is shown in Figure 6.5. Assuming that each cell m_k is independent and that the measurements are conditionally independent, the update of occupancy can be formulated efficiently using the well-known log-odds form [179],

$$l(m_k|\boldsymbol{z}_{1:t}) = l(m_k|\boldsymbol{z}_{1:t} - 1) + l(m_k|z_t)\,, \tag{6.6}$$

where $l(\cdot|\cdot) = \log(o(\cdot|\cdot))$, and $o(\cdot|\cdot)$ is the odds form:

$$o(m_k|\boldsymbol{z}_{1:t}) = \frac{p(m_k|\boldsymbol{z}_{1:t})}{1 - p(m_k|\boldsymbol{z}_{1:t})}\,. \tag{6.7}$$

The main advantage of occupancy mapping is shown in (6.6), where only the previous occupancy value and the inverse sensor model $l(m_k|\boldsymbol{z}_t)$ are needed to update the probability through a simple addition. Despite these benefits, occupancy grids rely on very strong assumptions of the environment to be efficient. Notably, the assumption that the likelihood of occupancy in one cell is independent of other cells disregard spatial correlations that can be important to infer occupancy in unobserved nearby regions. Additionally, traditional occupancy grids require the discretization of the environment be defined a priori which makes the spatial resolution constant throughout the map.

6.2.2 Distance Fields

Another way of representing the geometry surrounding the robot is not through *probabilities* of occupancy, but rather by describing the boundary between free and occupied space. In an ideal world, we could describe the shape and location of this

boundary using an analytical function in three variables (x, y and z) which reaches 0 whenever a point $\boldsymbol{p} = (x, y, z)$ lies on the surface. In other words, the function's zero crossings correspond to the surface itself. Such a function is known as an *implicit surface*. By convention, the function's sign is negative when $\boldsymbol{p}$ lies *inside* an object and positive *outside*. A simple example in Figure 6.6 illustrates how this implicit function values are determined.

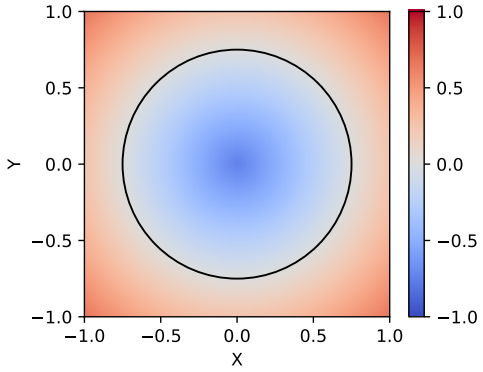


Figure 6.6 A solid 2D disk represented as an explicit surface (black outline) and implicit surface (colored field). The implicit surface function is negative inside the object (blue) and positive outside (red). Note how the function's zero crossings correspond to the surface itself.

There are many advantages to continuous implicit surface representations. Given that there is no fixed resolution, they can represent objects of arbitrary shapes at any level of detail. Furthermore, they make it possible to check if a given point in space is inside or outside an obstacle by simply evaluating the function's sign.

Different types of functions can be employed to model implicit surfaces, with the Euclidean Signed Distance Function (ESDF) being a prevalent option. At any query point, the ESDF expresses the distance to the nearest surface (indicated by the magnitude of the ESDF) and whether the point is inside an obstacle (indicated by the sign of the ESDF). ESDF representations are commonly used by accelerated geometric algorithms for tasks such as collision checking. Furthermore, high-quality proximity gradients can be derived from the ESDF for optimization-based motion planning and shape registration.

The ESDF is computed by finding the (Euclidean) closest surface point for each point in the map. For a known surface, this can efficiently be done using techniques such as Fast Marching. However, for surface estimation, the *projective* signed distance is more commonly used as it can efficiently be computed from measurements and is better suited for filtering. Given a measurement ray going through a query point, the projective distance is defined as the distance from the beam's endpoint to the query point *along the ray*. This eliminates the need to search the closest point

explicitly. Although the projective distance overestimates the Euclidean distance, its zero crossings (the estimated surface) remain correct. The standard approach of estimating implicit surfaces, proposed by Curless and Levoy [61] and popularized in the field of robotics by KinectFusion [185], combines the projective signed distances for all measurements using a simple weighted average. To reduce the impact of overestimates, the projective signed distance function is typically clamped to a fixed range, named the truncation band, in which case it is called the Truncated Signed Distance Function (TSDF). Note that ESDFs can efficiently be computed from TSDFs and occupancy maps, as will be described in Section 6.4.4.

6.2.3 Occupancy Maps or Distance Fields?

Being volumetric methods, a shared aspect of these estimated quantities in occupancy and implicit representations is that they model the geometry by estimating a quantity of interest everywhere in the observed volume. However, each representation fundamentally prioritizes different things. Which option is best, therefore, depends on the application. We will briefly summarize two key differences.

Directness in modeling: Given that measurement rays *directly* tell us which parts of space are free, occupied or unobserved, maps based on occupancy probabilities can be updated using fewer heuristics and assumptions. In contrast, implicit surfaces typically model the distance to the surface. This can be computed exactly for a known surface, but not from partial measurements. For surface estimation, they therefore rely on distance proxies such as the previously introduced TSDF.

Smoothness: Implicit surface maps are inherently smoother than occupancy maps, which model a binary property. The smoothness of implicit surfaces has many benefits. Most importantly, it makes them differentiable. The resulting proximity gradients are valuable for many applications. Smoothness also reduces the approximation errors resulting from discretization and makes it possible to obtain good, sub-pixel resolution estimates through interpolation. However, since discontinuities cannot be represented smoothly, implicit surfaces tend to miss thin obstacles.

6.3 Map Representations

6.3.1 Explicitness of Target Spatial Structures

As summarized in Figure 6.4, the representation can be classified based on their explicitness and target space. In 3D mapping, representing volume is straightforward; however, surfaces are equally important in robotic mapping for enabling downstream tasks. We can consider four major categorization: explicit surface, implicit surface, explicit volume, and implicit volume representations.

For surfaces, we can either *explicitly* or *implicitly* represent a surface. Defined as a 2D manifold, explicit type of representation aims to characterize the space

in terms of their boundary of the objects in the scene. The simplest abstraction that can represent the boundary is directly the point cloud produced by the range sensors. Another general representation of the surface is the polygon mesh (Section 6.4.3), which comprises vertices, edges, and faces. These meshes have the ability to encode the directed surfaces of a volume by forming connected closed polygons, more commonly triangles. Surfels (Section 6.4.2) are also popular abstraction widely used in mapping. Surface representations are a key for any visualization application, but also are used for rendering simulated environments, augmented reality or for computed aided design and 3D printing.

Similar strategies are employed in 3D volume modeling. Naive point-based representations are commonly used in LiDAR SLAM. Additionally, occupancy or distance-based voxels (Section 6.4.4) are popular choices for explicit representation. When storing volumetric maps, careful consideration of data storage is necessary to minimize computational costs. Implicit representations for volumetric mapping are also utilized, typically through functions. GP (Section 6.4.5) and Hilbert maps (Section 6.4.6) are well-known examples of implicit representations.

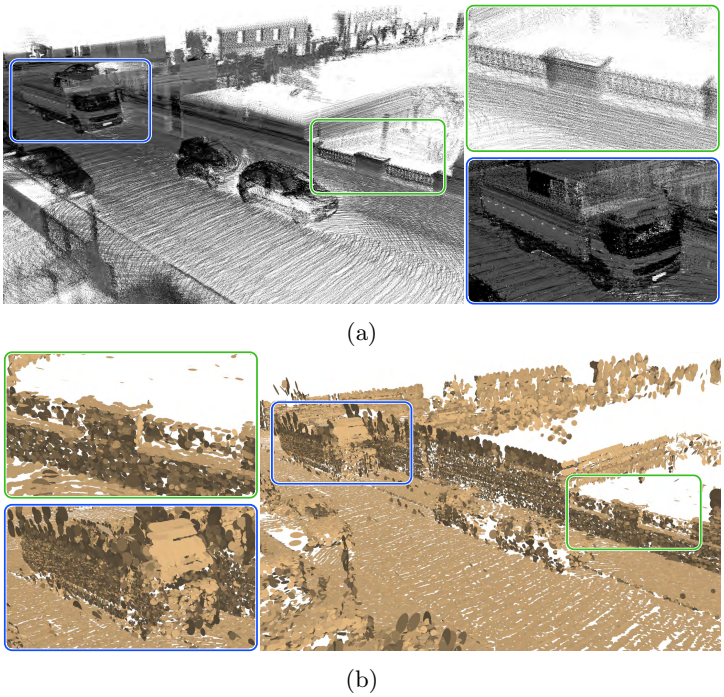
6.3.2 Types of Spatial Abstractions

6.3.2.1 Points

Given range measurements, a straight-forward dense map representation is using clouds. For generation, we accumulate the point clouds P_t recorded at time t in the local coordinate frame $\mathcal{F}^t$ using the estimated global pose $\boldsymbol{T}_t^1$ in a global map point cloud P_M. Also common practice is to assume our global coordinate frame of P_M is given in the coordinate frame of the first point cloud $\mathcal{F}^1$.

Since simply accumulating point clouds $\mathsf{P}_1, \ldots, \mathsf{P}_t$ does not scale to larger environments, a common strategy is to discard redundant measurements of the same spatial location. To this end, most methods [307, 70, 272] use efficient nearest neighbor search, such as voxel grids or hierarchical tree-based representations (see Section 6.3.3), to subsample and store the point clouds, *e.g.*,, store only a limited number of points per voxel [70, 272] or only specific points that meet a certain criterion are stored [307]. Additionally, a representation of only keyframes where only a few point clouds are explicitly stored is possible, but this requires to determine when a keyframe or submap needs to be generated.

Being the most elemental representation form, a point cloud map can be converted into other representations, *e.g.,* a mesh via Poisson surface reconstruction [270] or a Signed Distance Function (SDF) via marching cubes [271]. Unfortunately, this is feasible only with additional data, such as the viewpoint of a point's measurement. Yet, this information may be lost when merging multiple measurements into a point cloud map. Therefore, assumptions about the surface's direction are often required to discern inside or outside regions.



(a)

(b)

Figure 6.7 Qualitative comparison of maps generated by accumulating point clouds and surfels from a sequence of LiDAR scans from the KITTI dataset [96] Sequence 07. (a) Point cloud map. The brightness of points indicates the remission of the LiDAR measurements. (b) Corresponding surfel map based on circular disks. The complete map with all accumulated point clouds use 2.95 GB, while the corresponding surfel map by SuMa [24] uses only 160 MB.

6.3.2.2 Surfels

While point cloud maps directly represent the measurements, the stored points do not contain surface information or can represent from which direction a point has been measured. With surfels [208], we can encode such information by adding directional information to a point. Surfels are commonly represented via circular or elliptic discs [131, 24, 285, 35, 34], or more generally ellipsoids [246, 247, 74, 314] modeled with a a Gaussian. So-called splatting [314, 34] allows to integrate texture information but also blend overlapping surfels into coherent renderings of a specific viewpoint.

A commonly employed circular surfel representing a circular disk is defined by a location $\boldsymbol{p} \in \mathbb{R}^3$, a normal direction $\boldsymbol{n} \in \mathbb{R}^3$, and a radius $r \in \mathbb{R}$. As rendering primitive such surface patches can be efficiently rendered using the capabilities of modern graphics processing units (GPUs), which can be exploited to efficiently

render arbitrary views. This accelerates point-to-surfel associations and leads to the substantial memory reduction.

Figure 6.7 qualitatively compares a point cloud-based and a surfel-based map representation. A dense point cloud can accurately represent the environment with a high level of detail, but at the cost of memory. In contrast, while losing fine details as multiple measurements get aggregated into a single surfel, significant memory usage can be reduced while preserving the main structural details of larger surfaces.

Closely related to the explicit geometric representation of surfaces via surfels, *i.e.,* small circular surface patches, is the representation via a normal distributions transform (NDT) [29, 245]. Using NDT, the space is subdivided into voxels and the points inside a voxel are approximated via a normal distribution $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, having estimated mean $\boldsymbol{\mu}$ and covariance from the enclosed points $\boldsymbol{\Sigma}$. The eigenvalues $\lambda_1 < \lambda_2 < \lambda_3$ and corresponding eigenvectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3$ of the covariance can be used to estimate the surface properties inside a voxel. For planar surfaces ($\lambda_1 \ll \lambda_2$), the eigenvector $\boldsymbol{v}_1$ of the smallest eigenvalue λ_1 corresponds to the surface normal. Thus, for planar surfaces the NDT represents a surfel, and can also more accurately represent point distributions that cannot be approximated via a surfel. In that sense, the NDT is a hybrid representation that is explicit due to the space division into a voxel grid, but also implicit due to the representation of voxels via a normal distribution which continuously represents the space inside a voxel.

6.3.2.3 Meshes

While describing local surface properties, both point clouds and surfel maps are still relatively sparse as they do not model the surface's connectivity. One way to get a more complete understanding is to use meshes, which describe the surface as a set of points that are connected to form a collection of polygons. This, in turn, makes it possible to represent watertight surfaces, query and interpolate new surface points, and efficiently iterate along a connected surface.

In meshing terminology, each polygon is referred to as a *face*, and each corner point as a *vertex*. The most common types are triangle meshes, where each face is bounded by three vertices, and quad meshes, whose faces are bound by four vertices. Note that a polygon, or face, can always be broken down into an equivalent set of triangles; hence, triangle meshes are not only the simplest but also the most general.

A mesh is a very flexible and memory-efficient representation because the number of faces and vertices can be directly adapted to the surface complexity and required detail. For example, a plane of any size can be represented with just two triangular faces and four vertices. Furthermore, meshes are well-suited for parallel processing and rendering. Meshes are often used in applications that overlap with computer graphics, such as rendering, surface analysis, manipulation, and deformation, and more generally in applications involving digital models, simulation, or surface-based algorithms, such as path planning for ground robots.

6.3.2.4 Voxels

Point clouds and meshes are well suited to represent properties of the environment that are defined along surfaces. However, certain estimated quantities, including occupancy and Signed Distance, are defined throughout the entire volume. One straightforward way to store and process volumetric properties is to discretize them over a regular grid. Discretized occupancy and Signed Distance maps are called occupancy grids and Signed Distance Fields, respectively.

Generalizing the concept of 2D pixels, the cells in a 3D grid are referred to as voxels. Given a grid's regular structure, each voxel can easily be assigned a unique index and stored in a data structure. Note that a voxel is merely a container or, more formally, a space partition. The significance of its contents varies from one method to another. In a classic occupancy grid, a voxel's value represents the likelihood that any point in the voxel is occupied. Hence, the occupancy at an arbitrary point in the map is equal to the value of the voxel that contains the point. However, a voxel's value does not have to represent a constant little cube. For example, voxels in a Signed Distance Field estimate the signed distance at each voxel's center. To retrieve the signed distance at an arbitrary point, one would therefore query the voxels that neighbor the point and obtain the point's value using interpolation. Finally, some applications even use (sparse) voxel grids to store and efficiently query non-volumetric properties, such as points or surface colors.

6.3.2.5 Continuous Functions

Functions are a key abstraction for mapping in a continuous manner. The problem of mapping in this case is reduced to fitting a parametric or non-parametric function, *i.e.,* solving a regression problem. Most of the above-mentioned space abstractions require the discretization of the environment to be defined *a priori*, which usually makes the spatial resolution constant throughout the map. Continuous functions, however, parametric or non-parametric, give more flexibility allowing the resolution to be recomputed and also provide interpolation capabilities to fill up data gaps.

Some parametric functions such as infinite lines in 2D [263] and planes in 3D [128, 97, 263] require making strict assumptions about the environment and limit the representation of the scene. However, these representations are efficient in terms of memory consumption and computational complexity. Control points-based functions (*e.g.,* B-splines [223]) or non-parametric (*e.g.,* GP-based) have the ability to model the environment with fewer assumptions, still in a continuous manner. From occupancy [190], implicit surface [287, 146], distance fields [288], and surface itself [268], GP-based representations are a popular choice to represent the environment —despite their high computational complexity— because of their probabilistic nature, which enables uncertainty quantification and inference over both observed and unseen areas [99].

A key advantage of the continuous functions for mapping is that if they are

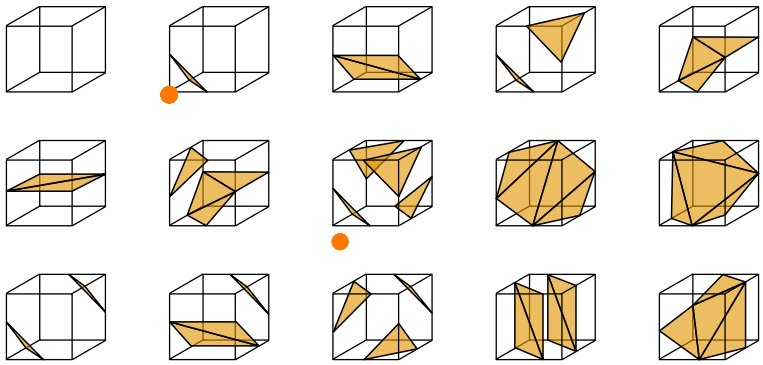


Figure 6.8 The algorithm works by projecting the cubes into the implicit surface, querying the sign of the values at the corners of the cubes, and looking up which one of the 15 configurations these values map to.

chosen to be at least once differentiable they will be able to provide gradients. Gradient information can be key for localisation to compute surface normals [289], loop closure to compute terrain features [98], for data fusion [146] and planning [289] applications.

6.3.2.6 Conversions

The abstractions listed above are not always used exclusively; they are often converted from one form to another or employed simultaneously in multiple forms.

For instance, explicit geometry, such as points and meshes, can be transformed into an implicit surface. One flexible method to compute the signed distance at any point in space is through a *closest point lookup*, which can be performed against any explicit geometry and on-demand, only when and where needed. Alternatively, the signed distance can be computed across all points on a regular grid using *wavefront propagation*, which can efficiently be implemented via the fast marching method [231].

Conversely, it is common to convert implicit surfaces into mesh representations. The original technique for converting distance fields into meshes is known as Marching Cubes [157]. The algorithm divides the implicit surface into a grid of fixed-size *cubes*, which it processes independently. Each cube generates a set of triangle elements based on the implicit surface's values at its eight corners (Figure 6.8). The positions of their vertices are then refined through linear interpolation. Meshes, including those from BIM or CAD models, can, in turn, be sampled to create points or surfels.

Occasionally, a discrete representation must be converted into a continuous one. This is typically achieved by solving an optimization problem over the parameters

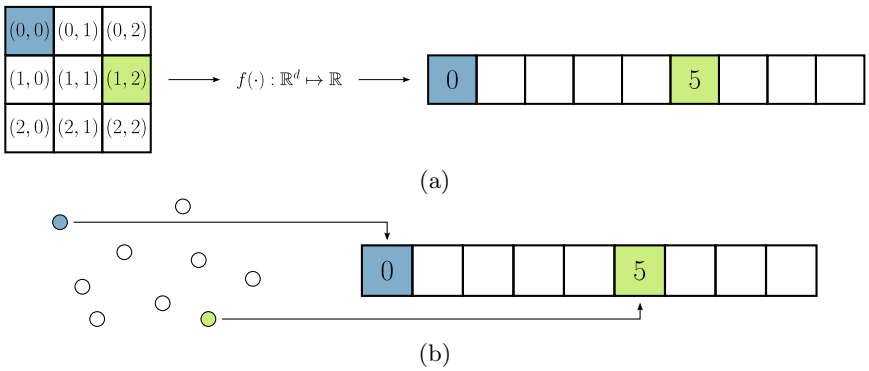


Figure 6.9 (a) Mapping of logical grid coordinates to an underlying naive array-based storage through a function $f(\cdot)$ that uniquely maps coordinates to linear indices. (b) Storage of unordered data directly into an array structure.

of the continuous representation, minimizing the fitting error with respect to the discrete data.

6.3.3 Data Structures and Storage

Previously introduced abstractions all need to be stored in memory. In this section, we explore how various abstractions are stored in memory by examining the choice of data structures along with their advantages and disadvantages.

6.3.3.1 Naive Data Storage

For many representations a simple dynamically resizeable array is a reasonable starting point. For data with a pre-defined spatial partitioning two things are needed, the type of data to store and a conversion function from a spatial coordinate to an index coordinate. This is often used when building maps representing occupancy or signed distance values. For irregular data, only the type of data to store is needed, for example point clouds or surfels. The naive storage of ordered data using a mapping function and unordered pointcloud data is illustrated in Figure 6.9.

The benefit of this naive approach is that it is simple and provides fast random access. The trade-off is, that large amounts of memory can be required for such a representation. Also while read and modify operations are fast, changing the spatial dimension of the representation can be very costly as the content of the entire data structure needs to be copied.

6.3.3.2 Hash Map

A natural extension to address the limitations of the naive storage method described above is to use a hash table. This approach divides the map into shards, applies

a *hash function* to convert the coordinates of each shard into a single value, and stores the sharded data in a table indexed by this hash value. The shards are typically chosen to represent map subregions with well-established coordinates, such as cubes in a regular grid. These cubes may correspond to individual voxels or fixed-size groups of voxels, referred to as voxel *blocks*. Alternatively, they can also store other elements like points, surfels, or mesh fragments contained in their respective subregions.

A hash table retains the fast $\mathcal{O}(1)$ look-up time of a fixed array while allowing the map to grow dynamically without reallocation. Three key considerations must be addressed when using a hash table for dense map storage:

1 **Granularity of the sharding:** Smaller shards improve sparsity by allocating data only where necessary. However, the number of shards should not grow too large, as this reduces the hash table's insertion performance and memory efficiency. This trade-off is particularly relevant when hash maps are used to store properties that only exist along the surface.
2 **Hash function:** An ideal hash function distributes keys evenly across the table, even when the data is spatially adjacent, as is often the case in mapping scenarios.
3 **Collision resolution:** The method for handling hash collisions, whether through linear chaining (where each entry contains a linked list) or open addressing, significantly affects the performance of the hash table.

In most cases, hash tables offer a good balance of fast access and efficient insertion and deletion of data. However, they may require initial tuning to perform well for a given application.

6.3.3.3 Tree-based Data Structures

Another option to efficiently store spatial data while only occupying memory for relevant parts of the environment is to use hierarchical, tree-based representations. Just like hash tables, trees generally enable efficient access and insertions. However, their unique strength is their hierarchical structure, which can be used to efficiently store multi-resolution data and speed up spatial operations such as nearest neighbor search. The most prominent tree variants are kD-trees [26], bounding volume hierarchies (BVH) [58], and octrees [175].

Among them, the octree efficiently searches neighbors with the capability to integrate novel measurements incrementally. The octree is a tree representing each node by a so-called *octant* that refers to a subspace. An octant is defined by a center $\boldsymbol{c} \in \mathbb{R}^3$ and an extent $e \in \mathbb{R}$, corresponding to an axis-aligned bounding-box. Each octant has potentially 8 child octants of extent $\frac{1}{2}e$, as depicted in Figure 6.10. Common practice is to store points only in the leaf octants (*i.e.,* octants without children) and determine subsets of points at inner octants of the tree structure by tree traversal.

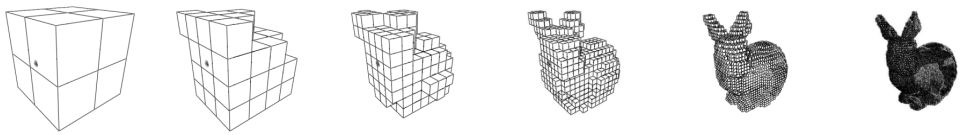


Figure 6.10 Example of an octree and its octants at different levels of the tree hierarchy. Each level of an octree subdivides the space in more fine-grained octants. Note that deeper levels of the octree only represent the occupied space.

To construct an octree, we iteratively divide space into octants within an axis-aligned bounding box encompassing a point cloud P. Each division splits P into subsets $\mathsf{P}_1, \ldots, \mathsf{P}_8$, corresponding to 8 child octants of half extent $\frac{1}{2}e$. Non-empty subsets P_i form child octants with center $\boldsymbol{c}$ and extent $\frac{1}{2}e$, stopping at a specific octant size or a minimal point count. Once constructed, updates and insertions can efficiently be performed by traversing the tree structure and adding inner nodes as needed. When new data is inserted that falls outside of the tree's root octant, the tree can be extended by creating a new root node and assigning the new data and the old root node to its children.

In contrast to voxel grids, an octree represents only data-containing subspaces, enabling efficient storage of occupied space. However, this memory efficiency requires tree traversal to access specific leaf octants, potentially leading to increased runtime to locate points. Additionally, the tree structure itself must be explicitly represented, incurring extra memory overhead. Several recent approaches address these memory overheads [84, 192, 25].

6.3.3.4 Hybrid Data Structures

To balance memory requirements and runtime for data access, several data structures combine the advantages of different data structures in specific ways, leading to hybrid representations. In this tradeoff, we accept less efficient memory usage but enable more efficient memory access.

For example, hashed voxel grids [186] combine the strengths of dense voxel grids and hash tables by splitting the environment into fixed-sized, dense blocks (e.g. $8 \times 8 \times 8$ voxels), which are in turn stored in a hash table. Thanks to the hash table's flexibility, blocks only need to be allocated in locations that contain meaningful information (e.g. near the surface). At the same time, using a plain 3D array to store the voxels inside each block ensures that operations remain simple, efficient, and even suitable to GPU acceleration.

Another option is to combine hash tables with trees. In a similar vein to hashed voxel grids, the VDB [3] data structure [184, 183] splits the space into hashed blocks, but stores a hierarchical tree inside each block. This data structure provides all the

[3] VDB refers to sparse volumetric data and stands for several different thing as Voxel Data Base or Volumetric Data Blocks. Here we follow the terminology used in [184].

Section	Space Abstraction Type	Representing Map Entities
6.4.1	Points	Surface
6.4.2	Surfels	Surface
6.4.3	Mesh	Surface (connected)
6.4.4	Voxels	Occupancy or Implicit surface
6.4.5 - 6.4.6	Continuous function	Occupancy or Implicit surface

Table 6.1 *Summary of presented mapping methods.*

benefits of hierarchical trees, including multi-resolution representation and efficient nearest neighbor lookups. However, since each block has a fixed size, the maximum tree height is constant regardless of the size of the environment. Lookups and insertions can therefore be performed in constant time, and significantly faster than when using pure trees.

6.4 Constructing Maps: Methods and Practices

So far, we have explored the target quantities to estimate and the various space abstractions available for mapping. In this section, we will examine in detail the methods used to construct these map elements. The approaches are categorized by their main space abstraction, as shown in Table 6.1. Note that some of the methods use additional space abstractions to improve performance, for example, by grouping points into voxels for more efficient storage and faster queries.

6.4.1 Points

As mentioned in Section 6.3.2.1, naively storing points by accumulating the measured point clouds will not scale to large-scale environments and will lead to redundantly represented measurements. Therefore, most approaches [307, 70, 272] adopt a point-based representation in combination with a voxel grid or octree to represent the dense map. Moreover, the selection of a data structure is driven by the requirement for efficient nearest neighbor searches, essential for conducting scan registration through iterative closest point (ICP), where point correspondences must be iteratively established.

In order to handle large-scale environments, some methods, such as the well-known LiDAR SLAM LOAM [307], filter the raw point clouds to extract corner and surface points thereby significantly reducing the point cloud size. A voxel grid is applied to store only a subset of points in the map representation, pruning redundant measurements. Stemming from the point-based voxelization used in LOAM, several follow-up approaches [234, 276, 198, 152, 213, 233] refine the extraction of points [234, 276, 198], improve the optimization pipeline [198, 152], or integrate information of an IMU [213, 233].

Another branch of methods handles the amount of point cloud data differently to avoid reliance on a capable feature extraction approach. Regularly sampling the point clouds via a voxel grid [70] significantly reduces the number of points per LiDAR scan and removes potentially redundant information. The key insight is here that points in the voxel grid are not aggregated and averaged, but original measurements are retained. Following these insights, Dellenbach *et al.* [70] and Vizzo *et al.* [272] use this strategy to downsample an input point cloud, only storing a restricted number of points inside a voxel grid map.

Overall, as also mentioned in Section 6.3.2.1, the (hashed) voxel grid serves dual purposes: it abstracts space by storing a limited number of point measurements per voxel, and it facilitates accelerated nearest neighbor search through direct indexing of neighboring voxels.

6.4.2 Surfels

For surfels, similar strategies can be applied as for point clouds, but notably Stückler *et al.* [246] and follow-up work by Dröschel *et al.* [74] use an octree to represent surfels at multiple levels in the octree hierarchy for data association. The so-called multi-resolution surfel maps indirectly represent the surfels via accumulated mean and covariance statistics, like a NDT.

In contrast, Whelan *et al.* [285] store surfels as a simple list and exploit efficient rendering techniques to produce a projection for data association for RGB-D SLAM in indoor environments. In this case, surfels are explicit geometric primitives and, therefore, need to be directly handled to update the surfel properties (*i.e.*,, size and direction) accordingly [131]. A key contribution of Whelan *et al.* is leveraging a map deformation that directly deforms the surfels instead of relying on a pose graph optimization, which enables the use of the measurements represented by the surfels to deform the map on a loop closure detection. A similar strategy for map deformation of surfels was employed by Park *et al.* [200].

Similarly, Behley *et al.* [24] target outdoor environments, which makes it necessary to represent the surfels via multiple submaps of $100\,\mathrm{m} \times 100\,\mathrm{m}$ spatial extent that can be off-loaded from GPU memory. In contrast to ElasticFusiuon [285], the approach relies on pose graph optimization but exploits that surfels can be freely positioned and ties surfels to poses enabling a straight-forward deformation of the map with pose-graph-optimized poses, which was also adopted by other approaches [278].

6.4.3 Meshes

As introduced ealier, meshes offer an expressive, flexible way to represent connected surfaces. Mesh generation methods can be split into two families of approaches.

The first family directly converts the measured points into a mesh. In contrast, the second family splits the problem into two steps: reconstructing an implicit surface, followed by iso-surface extraction to get the final mesh (see Section 6.3.2.6).

Methods in the first family typically work *directly* by computing the Delaunay triangulation of the input point set and identifying the subset of Delaunay triangles that lie on the surface. A detailed overview of such methods is provided in [45]. When building directly from points, the mesh implicitly adapts itself to the sampling density. This can be an advantage, as it provides adaptive resolution, but it also means these methods are more sensitive to sampling irregularities and holes. In practice, direct meshing methods are chosen when the entire surface can be sampled densely with a very accurate depth sensor, for example, using surveying equipment.

The second family of approaches uses an implicit surface as an *intermediate* step, to simplify the process of fusing and filtering the data before extracting the final surface mesh. One intuitive way to generate the implicit surface from data is to estimate the distance to the surface at each point on a regular grid. As described in Section 6.2.2, the implicit surface's sign must also be set according to whether each point is inside or outside an object. This information is often determined based on estimated surface normals, which can for example be obtained by applying Principal Component Analysis (PCA) over a small surrounding area. However, as indicated in [114], such methods may yield implicit surfaces that are discontinuous. Tackling this issue, Carr et al. [44] model the implicit surface using a collection of Radial Basis Functions (RBFs) and fit these to the input points by solving a global optimization problem. The resulting implicit surfaces are smooth by construction and faithfully fill holes based on the global context. Unfortunately, solving the underlying large, dense optimization problem is computationally expensive. Shen et al. [235] overcome this limitation by locally approximating the input points using moving least squares (MLS). Going one step further, Poisson Surface Reconstruction [129] fits the implicit function to the normals of the measured points by solving a partial differential equation (PDE), resulting in a sparse, computationally tractable optimization problem that is particularly robust to noise.

In robotics applications, constructing a mesh from a live sensor stream is often desirable. One way to make surface reconstruction efficient enough to run in real-time is to use incremental updates. TSDF-based surface reconstruction is particularly popular in practice given its inherently incremental nature and general simplicity. This method falls under the second family of approaches and estimates the implicit surface by averaging projective distances. Since the cost of updating the TSDF, or implicit surface in general, overshadows the cost of the mesh extraction, real-time methods primarily focus on optimizing the former.

6.4.4 Voxels

Voxel-based methods are among the most commonly used volumetric representations in 3D reconstruction and robotics. Instead of covering a swath of existing literature chronologically, this section will focus on concepts commonly encountered in practice and organize them according to three fundamental decision criteria: the chosen estimated quantity, data structure, and scalability considerations.

6.4.4.1 Methods by their Estimated Quantity

The first choice in a voxel-based mapping framework is which quantity to estimate, with the most common options being *occupancy* (see Section 6.2.1) or a *distance* metric (see Section 6.2.2). The previous discussion in Section 6.2.3 can be used to decide between the two.

Since the introduction of the original continuous probabilistic occupancy measurement model for sonar [179], simplified piecewise-constant models have been developed to reduce computational costs [116]. This shift was influenced by the advent of LiDAR technology and the growing interest in transitioning from 2D to 3D maps. More recently, Loop et al. [156] presented a continuous probabilistic model that, instead of inflating objects, converges to an occupancy probability of 0.5 along objects' surfaces. Occupancy estimation, popular for collision avoidance due to its superior recall, is limited by its discontinuous nature and uninformative gradients compared to distance-based methods (see Section 6.2.3).

For distance metrics, we must not only estimate the positive part of the distance field but also extrapolate negative distances behind the surface since the surface is represented by the signed distance field's zero-crossings. To limit the accuracy impact of fusing imperfect positive and negative distances estimates (see Section 6.2.2), the updates are typically clamped to a small *truncation band* around the surface boundary. However, distance-based methods remain prone to erasing geometry. For example, when thin objects are observed from opposing sides, averaging the observed positive and hallucinated negative distances makes the zero-crossings flip around or disappear. Some works have analyzed the effect of the truncation band and weight drop-offs on the quality of the final reconstruction [40]. Fundamentally, the problem can be reduced but not eliminated. Overall, the surfaces estimated by TSDFs outperform occupancy methods along smooth surfaces at the cost of lower *recall* on thin objects.

The distance information provided by TSDFs is inherently valuable. However, instead of being conservative, TSDFs strictly overestimate the *Euclidean* distance. To address this safety concern, voxblox [193] popularized incrementally building ESDFs. Voxblox fuses the sensor data into a TSDF and then updates its ESDF using a brushfire algorithm [144]. Subsequently, FIESTA [108] proposed a hybrid approach that incrementally updates an ESDF map from an occupancy map instead.

6.4.4.2 Methods by Data Structure

The simplest data structure for volumetric mapping is a static 3D array. As shown by KinectFusion [185], this data structure yields good results for small and fixed-size scenes. However, many applications require the ability to dynamically expand the map at runtime, while only allocating voxels where needed to save memory.

To address these concerns, Niessner proposed a voxel-block hashing scheme [186], which groups the voxels into blocks (e.g., $8 \times 8 \times 8$ voxels) that are stored in a hash-map. This data structure was quickly adopted for TSDFs, providing constant-time ($\mathcal{O}(1)$) lookups and dynamic insertions. Of course, it can also be used to store occupancy probabilities, as shown by FIESTA [108]. Compared to hashing voxels individually, grouping them in blocks offers an adjustable trade-off between the hash table's size and the granularity at which voxels are allocated.

Naturally, voxels can also be stored using tree structures. Octomap [116] first popularized using an *octree* to store occupancy probabilities and has been the *de facto* standard for volumetric mapping for many years. A significant advantage of using trees is that they inherently support multi-resolution, while a major limitation is that encoding the tree's structure introduces a significant memory overhead, and that the cell lookup time is proportional to the tree's height. Most recent approaches address this limitation by leveraging hybrid data structures. Supereight [269], for example, proposes to use a standard (dynamic) octree for the first levels and static octrees for the last few levels. These static octrees can be seen as octrees stored using a fixed-sized array. This removes the memory overhead of encoding parent-child relationships with pointers, at the cost of reducing granularity since static octrees are allocated as a block. The VDB [184] data structure was first introduced for the visual effects (VFX) industry and subsequently used by several volumetric mapping frameworks [167, 271]. As discussed in Section 6.3.3.4, it combines block-hashing with trees to obtain the best of both worlds: good memory efficiency, hash-like constant time lookups and insertions, and tree-like multi-resolution.

A practical consideration is that downstream tasks often demand storing additional information, such as colors, semantics [102, 225] or an ESDF [193, 108] alongside the occupancy probabilities or TSDF. Although virtually any data structure can be extended to support additional channels, the required implementation effort scales with how complicated the underlying data structure is. This further motivates using simple data structures (e.g. voxel-block hashing) or flexible, third-party libraries.

6.4.4.3 Methods by Measurement Integration Algorithm

The algorithm used to update the map based on depth measurements is referred to as the measurement integrator. It updates the estimated quantity for each observed voxel by applying the measurement model. The two main approaches used to integrate measurements are ray-tracing and projection-based methods.

For each measured point, ray tracing integrators cast a ray from the sensor to the point and update all the voxels intersected by the ray. An advantage of this approach is that it is very general, and only requires that the position of the sensor's origin is known. However, voxels may be hit by multiple rays, especially if they are near the sensor. This leads to duplicated efforts, and handling the resulting race conditions in parallel implementations creates implementation and performance overheads.

In contrast, projection-based methods directly iterate over the observed voxels and look up the ray(s) needed to compute their update by projecting each voxel into sensor coordinates. Iterating over the map instead of the rays inherently avoids race conditions. Projection-based methods are, therefore, prevalent in multi-threaded and GPU-accelerated volumetric mapping frameworks. The predictable access pattern resulting from directly iterating over the map also reduces memory bottlenecks. Yet, a major disadvantage is the need for explicit knowledge of the sensor's full pose and projection model. This method is also harder to use with disorganized point clouds, including the clouds obtained after applying LiDAR motion-undistortion.

6.4.4.4 Methods by Scalability

Memory and computational costs are two of the main bottlenecks in volumetric mapping. For fixed-resolution methods, the memory and computational complexities grow linearly with the map's total volume and cubically with the chosen resolution. Reducing these complexities is of significant research interest, as it is necessary to create detailed maps that scale beyond small, restricted volumes.

Early works in volumetric mapping mainly focused on reducing memory usage. For example, Octomap [116] proposes to use its octree's inner nodes to store their children's max or average occupancy. By recursively pruning out leaf nodes whose estimated quantities are close to their parent, constant areas in the map are automatically represented with fewer, lower resolution nodes. This adaptation to the environment's geometry is very effective in practice since environments predominantly consist of free space. Furthermore, storing min, max, or average values in the octree's inner nodes could be valuable for downstream tasks, as it enables map queries at lower resolutions and the use of hierarchical algorithms for tasks such as fast collision checking or exploration planning. Yet, a core limitation of Octomap is that it integrates all measurements at the highest resolution, meaning that the scaling of its computational complexity remains cubic.

Multi-resolution can also be leveraged to reduce the computational cost of measurement updates. Given that measurement rays are emitted at fixed angles, resulting in fewer rays hitting distant geometry, it seems logical to lower the update resolution as the distance increases. This can be achieved through multi-resolution ray-tracing [75] or multi-resolution projective integration [269]. Supereight2 [95] reduces the computational complexity further by adjusting the update resolution to the entropy of the measurement updates. Such methods significantly enhance the update performance, yet a remaining challenge is that the map's different resolution

levels still have to be synchronized explicitly. One way to eliminate this synchronization requirement is to encode only the differences between each resolution level, instead of storing absolute values in each octree node. This can formally be done by applying wavelet decomposition. Wavelet-encoded maps can efficiently be queried at any resolution at any time. Using this property, wavemap [220] reduces the computational complexity even further by updating the map in a coarse-to-fine manner. In addition to adjusting the update resolution to the measurement entropy, it also skips uninformative updates, such as when the occupancy for an area in the map has converged to being free, and all measurements agree.

6.4.5 GPs

As mentioned in Section 6.3.2.5, formulating the mapping problem as a regression problem is desired to obtain a continuous representation. Moreover, if the aim is to limit the number of assumptions about the environment, solving a non-linear regression problem with non-parametric methods is ideal. GP [217] is a stochastic, non-parametric, non-linear regression approach. It allows estimating the value of an unknown function at an arbitrary query point given noisy and sparse measurements at other points. We already learned in Chapter 3.2 how GP can be used for continuous time trajectory representation. As will be apparent, GP are also an appealing solution for mapping continuous quantities, and they have been extensively used in the robotics literature to model continuously spatial phenomena with depth sensors [268, 190, 98, 134].

The information in GP models is contained in its mean $\boldsymbol{m}(\boldsymbol{x})$ and kernel functions $\boldsymbol{\mathcal{K}}(\boldsymbol{x}, \boldsymbol{x}')$ and model the estimated continuous quantity as

$$f(\boldsymbol{x}) \sim \mathcal{GP}\left(\boldsymbol{m}(\boldsymbol{x}), \boldsymbol{\mathcal{K}}(\boldsymbol{x}, \boldsymbol{x}')\right). \tag{6.8}$$

Let $\mathsf{X} = \{\boldsymbol{x}_j \in \mathbb{R}^D\}$ be a set of locations with measurements $\boldsymbol{y}$, with $y_j = f(\boldsymbol{x}_j) + \epsilon_j$ of the estimated quantity taken at the locations $\boldsymbol{x}_j$. For J number of training pair $(\boldsymbol{x}_j, y_j)$, we assume the noise ϵ_j to be *i.i.d* following Gaussian $\epsilon_j \sim \mathcal{N}(\mathbf{0}, \sigma_j^2)$. Given a set of testing locations $\mathsf{X}^* = \{\boldsymbol{x}_n^* \in \mathbb{R}^D \mid n = 0, \ldots, N\}$, we can express the joint distribution of the function values and the observed target values as,

$$\begin{bmatrix} \boldsymbol{y} \\ \boldsymbol{f}_* \end{bmatrix} = \mathcal{N}\left(\mathbf{1}, \begin{bmatrix} \boldsymbol{K}(\mathsf{X}, \mathsf{X}) + \sigma_j^2 \mathbf{I} & \boldsymbol{K}(\mathsf{X}, \mathsf{X}^*) \\ \boldsymbol{K}(\mathsf{X}^*, \mathsf{X}) & \boldsymbol{K}(\mathsf{X}^*, \mathsf{X}^*) \end{bmatrix}\right), \tag{6.9}$$

where $\boldsymbol{K} = \left[\boldsymbol{\mathcal{K}}(\boldsymbol{x}_i, \boldsymbol{x}_j)\right]_{ij}$. Thus the conditional distribution of $(\boldsymbol{f}_* \mid \mathsf{X}, \boldsymbol{y}, \mathsf{X}^*) \sim \mathcal{N}\left(\overline{\boldsymbol{f}}_*, \operatorname{cov}(\boldsymbol{f}_*)\right)$, with the mean equation is given by,

$$\overline{\boldsymbol{f}_*} = \boldsymbol{K}(\mathsf{X}^*, \mathsf{X})\left[\boldsymbol{K}(\mathsf{X}, \mathsf{X}) + \sigma_j^2 \mathbf{I}\right]^{-1} \boldsymbol{y}, \tag{6.10}$$

and the covariance equation is,

$$\operatorname{cov}(\boldsymbol{f}_*) = \boldsymbol{K}(\mathsf{X}^*, \mathsf{X}^*) - \boldsymbol{K}(\mathsf{X}^*, \mathsf{X}) \left[\boldsymbol{K}(\mathsf{X}, \mathsf{X}) + \sigma_j^2 \mathbf{I}\right]^{-1} \boldsymbol{K}(\mathsf{X}, \mathsf{X}^*). \tag{6.11}$$

Here, (6.10) and (6.11) are the predictive equations for the estimated quantitative.

GPs have proven particularly powerful to represent spatially correlated data, hence overcoming the traditional assumption of independence between cells, characteristic of the occupancy grid method for mapping environments. Gaussian Process Occupancy Map (GPOM)s [190] collects sensor observations and the corresponding labels (free or occupied) as training data; the map cells comprise testing locations, which are related to the training data as shown in (6.9). After the regression is performed using (6.10) and (6.11), the cell's probability of occupancy is obtained by "squashing" regression outputs into occupancy probabilities using binary classification functions.

In its original formulation GPOM is a batch mapping technique with cubic computational complexity ($\mathcal{O}(J^3 + J^2 N)$). Approaches that aim to tacking this computational complexity especially for incremental GP map building have been proposed following this work, for example [134, 135, 277, 99].

A key advantage of mapping with GP-based functions is that the estimated quantity can be linearly operated [228] and still produce a GP as an output. Given that derivatives and, therefore gradients, are linear operations, the differentiation output of the estimated quantity is probabilistic. A continuous representation of the uncertainty in the environment can be used to highlight unexplored regions and optimize a robot's search plan [99, 154]. The continuity property of the GP map can improve the flexibility of a planner by inferring directly on collected sensor data without being limited by the resolution of a grid/voxel cell.

6.4.5.1 Gaussian Process Implicit Surface

Implicit surfaces can also be represented by a GP. Gaussian process implicit surface (GPIS) techniques [287, 171, 154, 118] use a GP approach to estimate a probabilistic and continuous representation of the implicit surface given noisy measurements. Furthermore, GPIS can be also used to estimate not only the surface but also the distance field in a continuous manner [136, 244, 146, 288].

In the GPIS formulation, let us consider the distance field $\boldsymbol{d}$ to be estimated from the distance to the nearest surface d_i given the the points on the surface and its corresponding gradient $\nabla \boldsymbol{d}$ computed through linear operators [228]. Then $\boldsymbol{d}$ with $\nabla \boldsymbol{d}$ can be modelled by the joint GP with zero mean (given that at the surface the distance is zero):

$$\begin{bmatrix} \boldsymbol{d} \\ \nabla \boldsymbol{d} \end{bmatrix} \sim \mathcal{GP}(\mathbf{0}, \boldsymbol{K}(\mathsf{X}, \mathsf{X}')). \tag{6.12}$$

GPIS approaches have the ability to estimate a continuous implicit surface and the normal of the surface through the gradient, both with uncertainty. Some works

have considered the use of parametric function priors to capture given shapes more accurately [171, 118]. Other approaches aimed to estimate not only the implicit surface but the full distance field. Given the nature of the vanilla version of the GPIS formulation, the distance is well approximated near the measurements, *i.e.,* on the surface, but falls back to the mean, which in this case is zero, faraway from the surface. To estimate the full distance field in a continuous and probabilistic formulation further away from the surface, works have considered applying a non-linear operation to a GPIS-like formulation [288, 289, 145].

All these works have to deal with the computational complexity of the GP-based formulation, but as an exchange, a continuous, generative, and probabilistic representation of the environment, given only point clouds can be achieved.

6.4.6 Hilbert Maps

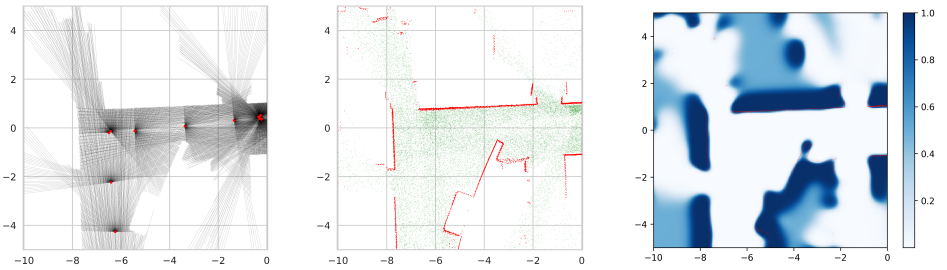
Hilbert Map (HM)s [216] are in many ways similar to GPOMs [190]. Both are continuous probabilistic models that do not discretize the space, unlike voxel-based methods, and in contrast to point-based methods are capable of interpolating missing data. As stated, the major challenge in GPOM is high computational expense. Thus the design goals of Hilbert maps were the following: (i) process data continuously in an online manner, (ii) model dependence between observations, and (iii) incorporate measurement uncertainty.

To achieve these goals, training a logistic regressor with stochastic gradient descent in a projected feature space is often leveraged. The classifier and optimizer combination enables online model updates using large amounts of data while the feature projection permits representing intricate spatial details with such a simple classifier.

The feature projection serves the same idea as the kernel in a GP, but instead of a full covariance we use an approximation. There are many options for this, including Nystroem [286], Random Fourier Features [215], and Sparse Kernel [176], which is what we will be using. To goal of the sparse kernel is to limit the range at which observations have an influence which improves convergence and computational efficiency. The outcome is a kernel that drops to exactly 0 at a specific distance.

This kernel allows us to project points in 2D or 3D space into significantly higher dimensions by placing inducing kernels at regular intervals over the space to be mapped. Furthermore, this enables computing high-dimensional feature space representations of input data to be trained the logistic regression classifier using mini-batch stochastic gradient descent. Lastly, training is done by sampling free space points along the range measurement, while adding the return as an obstacle point.

One challenge faced by HMs is the expressivity of the used kernel. A radial basis function (RBF), as used in Figure 6.11, is a circle or a sphere and their values need to be combined to reconstruct intricate details of the environment. Therefore there



(a) (b) (c)

Figure 6.11 (a) The observations by a robot using a 2D LiDAR sensor, which is turned into a dataset (b) Free (green) and occupied (red) points. These are then used to train a Hilbert Map as seen in (c).

is a tradeoff in the form of number of kernels and their lengthscale affecting the computational cost, reconstruction detail, and interpolation ability.

6.4.7 Deep Learning in Mapping

With the recent interest of the computer vision and robotics community in novel view synthesis using neural radiance fields (NeRFs) [177], which provided compelling results for image generation via a simple multi-layer perceptron (MLP), several approaches investigated the usage of neural representations to estimate a SDF. Learning to predict a SDF at arbitrary spatial location leads to a continuous representation that can be turned into meshes at arbitrary resolutions, but also can lead to more complete representation due to the interpolation capabilities of the learned function.

Similar to implict representations, the representation is learned from input data and approximated to provide a continuous function that can be queried at arbitary locations. While often these neural representations are learned offline with given poses, there has been recently also an interest in incremental approaches [248, 310] and approaches that estimate poses on-the-fly using a neural representation [248, 227].

In particular, the approach of Sucar *et al.* [248] uses a neural network to predict the SDF value of an arbitrary point in the scene based on RGB-D frames. Follow-up approaches extended this approach by separating the spatial representation of the features via voxel grids [206, 312], octrees [310], points [227, 71], etc. from the neural representation. In these approaches, small but descriptive features are stored in a spatial representation and used to determine with a small, neural network the SDF value of an arbitrary point in the scene. This allows to decouple the learned function from the spatial representation, which makes it possible to rely on small

neural decoders to turn features into signed distance values, but also being effective for large-scale scenes, such as outdoor environments.

The area of deep learning-based mapping, reconstruction, and SLAM is currently rapidly evolving and integrating ideas from classical representation, such as surfel splatting [130, 173], to achieve remarkable results in terms of reconstruction quality, but also capabilities. In particular, the ability to render novel views and generate new data at arbitary positions could be potentially exploited for robot learning without relying on simulated environments. For example, NeRF and Gaussian splatting [314, 34] have gained considerable popularity, demonstrating significant potential in various SLAM-related works. These will be further detailed in Chapter 16.

6.5 Usage Considerations

All map representations trade off distinctive, often complementary, strengths and weaknesses. When choosing a map representation for a given application or robotic system, it is therefore important to carefully consider how the map will be used in all downstream tasks. Further factors to consider are the operating environment and available sensors. We will start by discussing environmental factors, which motivate several clear-cut choices, followed by more nuanced task-dependent considerations. Finally, we conclude this chapter with a brief discussion on usage considerations related to the existing methods presented in Section 6.4.

6.5.1 Environmental Aspects

Operating environments can be categorized as either *structured* or *unstructured*. In tightly controlled spaces, such as automated factories, custom map representations – tailored to the robot's task and specific objects it will encounter – typically outperform general dense representations in terms of efficiency and accuracy. In contrast, the dense representations covered in this chapter can model objects of arbitrary shapes and work in any environment. When operating in changing or partially unknown environments, it is often important for robots to be able to distinguish observed free space from unobserved space. This information allows path planners to avoid unsafe motions through unobserved space, which could be occupied, and can also be used for exploration planning. Explicit surface representations, including points, surfels, and meshes, generally cannot distinguish between free and unobserved space, while all occupancy-based methods do. Implicit surface-based methods can also provide this distinction, though very often for more reconstruction-focused applications, this information is discarded farther from the surface to save computational and memory costs.

Another consideration is *scalability*. Explicit representations tend to be more memory efficient than implicit representations, as they only describe the surface

itself and their fidelity can easily be adapted to the detail required for each part of the scene. When free-space information is required, multi-resolution approaches can offer significant improvements over fixed-resolution voxelized representations in terms of accuracy, memory, and their ability to capture very thin objects.

One final consideration is whether the environment has a significant amount of *dynamic objects* and the degree to which these should be modeled. Most existing mapping frameworks can be grouped into one of three categories of approaches. The first set of approaches does not consider dynamics and directly fuses all measurements into one of the representations introduced in this chapter. In practice, this might already suffice when using implicit representations, since their free-space updates typically do a good job at erasing leftovers of objects after they moved. The second category of approaches tries to only integrate the environment's static elements into the map, by explicitly detecting and discarding all measurements corresponding to dynamic objects. This approach is particularly popular when using explicit maps, where leftovers are more tedious to remove, and generally makes it possible to generate clean maps even in highly dynamic spaces. The last set of solutions not only represents the background but also the moving elements in the scene. Note that this is commonly done using hybrid representations, mixing fundamental geometric representations introduced in this chapter with bespoke representations at the object level.

6.5.2 Downstream Task Types

In addition to the environment, it is equally important to consider what map information is necessary for the robot's required tasks. While any given operation can typically be performed on all representations, the efficiency and implementation complexity tend to vary greatly. The biggest difference lies in whether the operation is performed along the surface or in Cartesian space. As shown in Table 6.2, implicit representations generally allow for simple, efficient filtering of properties that are expressed in Cartesian coordinates, such as occupancy. In contrast, explicit representations are well suited to filter properties that are expressed along the surface, such as visual textures. This explains why explicit representations are generally more sensitive to the quality of the depth measurements, but can create very detailed, visually appealing 3D reconstructions. On the other hand, implicit methods are well suited for fusing noisy depth measurements, such as RGB-D camera data.

In terms of queries, explicit representations make it possible to directly iterate over the surface. This explains their popularity in rendering and graphics applications, and for tasks such as coverage path planning. However, they require additional steps, such as nearest neighbor lookups, to answer queries in Cartesian coordinates. The exact opposite is true for implicit representations, which are therefore commonly used for collision checking tasks.

	Efficient in	
Operation	Explicit representation	Implicit representation
Filter measurements	Along the surface (texture,...)	In Cartesian space (occupancy,...)
Query and iterate	In surface coordinates (coverage planning,...)	In Cartesian coordinates (collision checking,...)
Modify surface	Geometry (deformation,...)	Topology (merge, cut, simplify,...)

Table 6.2 *Complementary strengths and weaknesses of explicit and implicit surface representations.*

Finally, explicit representations allow for efficient modifications of the surface's geometry, including deformations. In practice, maps are often constructed by integrating depth measurements using pose estimates from an imperfect, drifting odometry system. Over time, the accumulated pose errors also lead to inconsistencies in the dense map. Just like in SLAM systems, these errors can be eliminated by deforming the dense map when detecting loop closures. Although both explicit and implicit surfaces can be deformed, this operation is inherently simpler and more efficient when using an explicit representation. In contrast, using an implicit representation simplifies and improves the efficiency of operations affecting the surface's topology, or connectivity. Implicit representations are therefore often used to merge surface estimates, combine or subtract object shapes, and simplify surfaces.

It is important to remember that different representations can also be used in tandem to leverage their respective strengths. One good example of a hybrid approach is TSDF-based meshing (Section 6.4.3), where noisy depth measurements are first conveniently filtered using an implicit surface representation (TSDF) which is then converted to an explicit representation (mesh) using Marching Cubes. When deciding whether the advantages of hybrid representations outweigh the overhead they introduce, it is worth considering how the conversions can be limited to only happen locally and infrequently.

6.5.3 Summary of Mapping Methods

We now conclude our discussion by summarizing the key differences between the existing methods presented in this chapter. Starting with the explicit representations, using a collection of points to describe the surface is simple and requires the fewest assumptions, but it is also the least informative. Beyond infinitesimal points, surfels represent the surface's properties over small neighborhoods, or patches. Finally, meshes explicitly represent the surface's connectivity and allow its properties to smoothly be interpolated. However, estimating the surface's connectivity requires the most assumptions and comes at a significant computational cost.

In terms of implicit representations, a particular advantage of implicit surfaces over occupancy maps is that they offer fast, high-quality distance information and gradients which are beneficial for optimization-based planning. However, filtering occupancy estimates requires less assumptions and, for voxel-based methods, occupancy maps are better at capturing thin obstacles. In cases with particularly noisy or sparse depth measurements, non-voxelized implicit representations, based on GPs and Hilbert Maps, provide particularly good uncertainty estimates. As they explicitly consider the geometry's spatial correlations, they are generally also better at interpolating partially observed surfaces.

One rapidly advancing research area is that of learning-based methods. In terms of learning-based implicit representations, NeRFs have been shown to enable promising new capabilities, particularly for semantic modeling and spatial reasoning. More recently, Gaussian splatting [132] – an explicit learning-based representation bearing similarities to surfels – lead to an increasing interest into approaches using splatting [130, 173, 305]. Researchers are actively working on improving the computational and memory footprint of these approaches, testing what new skills they can enable, and exploring how they can be integrated into complete robotic systems. Looking ahead, we suspect that learning-based methods can increase the generality and expressiveness of dense representation, while improving their ability to handle noisy measurements, incomplete observations and dynamic objects through learned priors.

7

Certifiably Optimal Solvers and Theoretical Properties of SLAM

Author I, Author II, and Author III

PART TWO

USE CASES / APPLICATIONS / STATE OF PRACTICE

8

Visual SLAM

Author I, Author II, and Author III

Aenean sem dolor, fermentum nec, gravida hendrerit, mattis eget, felis. Nullam non diam vitae mi lacinia consectetuer. Fusce non massa eget quam luctus posuere. Aenean vulputate velit. Quisque et dolor. Donec ipsum tortor, rutrum quis, mollis eu, mollis a, pede. Donec nulla. Duis molestie. Duis lobortis commodo purus. Pellentesque vel quam. Ut congue congue risus. Sed ligula. Aenean dictum pede vitae felis. Donec sit amet nibh. Maecenas eu orci. Quisque gravida quam sed massa.

9

LiDAR SLAM

Author I, Author II, and Author III

10

Radar SLAM

Author I, Author II, and Author III

Aenean sem dolor, fermentum nec, gravida hendrerit, mattis eget, felis. Nullam non diam vitae mi lacinia consectetuer. Fusce non massa eget quam luctus posuere. Aenean vulputate velit. Quisque et dolor. Donec ipsum tortor, rutrum quis, mollis eu, mollis a, pede. Donec nulla. Duis molestie. Duis lobortis commodo purus. Pellentesque vel quam. Ut congue congue risus. Sed ligula. Aenean dictum pede vitae felis. Donec sit amet nibh. Maecenas eu orci. Quisque gravida quam sed massa.

11

Event-based SLAM

Author I, Author II, and Author III

12

Inertial Odometry for SLAM

Author I, Author II, and Author III

13

Leg Odometry for SLAM

Marco Camurri and Matías Mattamala

PART THREE

FROM SLAM TO SPATIAL PERCEPTION AI

14

Spatial AI

Author I, Author II, and Author III

Aenean sem dolor, fermentum nec, gravida hendrerit, mattis eget, felis. Nullam non diam vitae mi lacinia consectetuer. Fusce non massa eget quam luctus posuere. Aenean vulputate velit. Quisque et dolor. Donec ipsum tortor, rutrum quis, mollis eu, mollis a, pede. Donec nulla. Duis molestie. Duis lobortis commodo purus. Pellentesque vel quam. Ut congue congue risus. Sed ligula. Aenean dictum pede vitae felis. Donec sit amet nibh. Maecenas eu orci. Quisque gravida quam sed massa.[307]

15

Boosting SLAM with Deep Learning

Author I, Author II, and Author III

Aenean sem dolor, fermentum nec, gravida hendrerit, mattis eget, felis. Nullam non diam vitae mi lacinia consectetuer. Fusce non massa eget quam luctus posuere. Aenean vulputate velit. Quisque et dolor. Donec ipsum tortor, rutrum quis, mollis eu, mollis a, pede. Donec nulla. Duis molestie. Duis lobortis commodo purus. Pellentesque vel quam. Ut congue congue risus. Sed ligula. Aenean dictum pede vitae felis. Donec sit amet nibh. Maecenas eu orci. Quisque gravida quam sed massa.

16

NeRF and GS

Author I, Author II, and Author III

Aenean sem dolor, fermentum nec, gravida hendrerit, mattis eget, felis. Nullam non diam vitae mi lacinia consectetuer. Fusce non massa eget quam luctus posuere. Aenean vulputate velit. Quisque et dolor. Donec ipsum tortor, rutrum quis, mollis eu, mollis a, pede. Donec nulla. Duis molestie. Duis lobortis commodo purus. Pellentesque vel quam. Ut congue congue risus. Sed ligula. Aenean dictum pede vitae felis. Donec sit amet nibh. Maecenas eu orci. Quisque gravida quam sed massa.[307]

17

Dynamic and Deformable SLAM

Author I, Author II, and Author III

Aenean sem dolor, fermentum nec, gravida hendrerit, mattis eget, felis. Nullam non diam vitae mi lacinia consectetuer. Fusce non massa eget quam luctus posuere. Aenean vulputate velit. Quisque et dolor. Donec ipsum tortor, rutrum quis, mollis eu, mollis a, pede. Donec nulla. Duis molestie. Duis lobortis commodo purus. Pellentesque vel quam. Ut congue congue risus. Sed ligula. Aenean dictum pede vitae felis. Donec sit amet nibh. Maecenas eu orci. Quisque gravida quam sed massa.

18

Metric-Semantic SLAM

Author I, Author II, and Author III

19

Foundation Models for SLAM

Author I, Author II, and Author III

Notes

References

[1] *FastTriggs*. `https://pypose.org/docs/main/generated/pypose.optim.corrector.FastTriggs/`.

[2] *LieTensor*. `https://pypose.org/docs/main/generated/pypose.LieTensor`.

[3] *PyPose Documents*. `https://pypose.org/docs/`.

[4] *PyPose Tutorial*. `https://github.com/pypose/tutorials`.

[5] Abadi, Martín, Barham, Paul, Chen, Jianmin, Chen, Zhifeng, Davis, Andy, Dean, Jeffrey, Devin, Matthieu, Ghemawat, Sanjay, Irving, Geoffrey, Isard, Michael, et al. 2016. TensorFlow: a system for Large-Scale machine learning. Pages 265–283 of: *12th USENIX symposium on operating systems design and implementation (OSDI 16)*.

[6] Aftab, Khurrum, and Hartley, Richard. 2015. Convergence of iteratively re-weighted least squares to robust m-estimators. Pages 480–487 of: *2015 IEEE Winter Conference on Applications of Computer Vision*. IEEE.

[7] Agarwal, P., Grisetti, G., Tipaldi, G. D., Spinello, L., Burgard, W., and Stachniss, C. 2014. Experimental Analysis of Dynamic Covariance Scaling for Robust Map Optimization Under Bad Initial Estimates. In: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[8] Agarwal, S., Snavely, N., Simon, I., Seitz, S. M., and Szeliski, R. 2009. Building Rome in a Day. In: *Intl. Conf. on Computer Vision (ICCV)*.

[9] Agarwal, Sameer, Mierle, Keir, and Team, The Ceres Solver. 2022 (3). *Ceres Solver*.

[10] Agrawal, A., Amos, B., Barratt, S., Boyd, S., Diamond, S., and Kolter, Z. 2019. Differentiable Convex Optimization Layers. In: *Advances in Neural Information Processing Systems*.

[11] Al-Rfou, Rami, Alain, Guillaume, Almahairi, Amjad, Angermueller, Christof, Bahdanau, Dzmitry, Ballas, Nicolas, Bastien, Frédéric, Bayer, Justin, Belikov, Anatoly, Belopolsky, Alexander, et al. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, arXiv–1605.

[12] Amestoy, P.R., Davis, T., and Duff, I.S. 1996. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, **17**(4), 886–905.

[13] Andersson, Joel AE, Gillis, Joris, Horn, Greg, Rawlings, James B, and Diehl, Moritz. 2019. CasADi: a software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, **11**(1), 1–36.

[14] Ankenbauer, Jacqueline, Lusk, Parker C., and How, Jonathan P. 2023 (Mar.). *Global Localization in Unstructured Environments Using Semantic Object Maps Built from Various Viewpoints.*

[15] Antonante, P., Tzoumas, V., Yang, H., and Carlone, L. 2021. Outlier-Robust Estimation: Hardness, Minimally Tuned Algorithms, and Applications. *IEEE Trans. Robotics*, **38**(1), 281–301. .

[16] Barath, Daniel, Noskova, Jana, Ivashechkin, Maksym, and Matas, Jiri. 2020. MAGSAC++, a fast, reliable and accurate robust estimator. Pages 1304–1312 of: *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR).*

[17] Barfoot, T D. 2024. *State Estimation for Robotics.* 2nd edn. Cambridge University Press.

[18] Barfoot, T D, Forbes, J R, and D'Eleuterio, G M T. 2022. Vectorial Parameterizations of Pose. *Robotica*, **40**(7), 2409–2427.

[19] Barfoot, Timothy D. 2017. *State estimation for robotics.* Cambridge University Press.

[20] Barrau, A., and Bonnabel, S. 2017. The Invariant Extended Kalman Filter as a Stable Observer. *IEEE Trans. on Automatic Control*, **62**(4), 1797–1812.

[21] Barron, Jonathan T. 2019. A general and adaptive robust loss function. Pages 4331–4339 of: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.*

[22] Bauernfeind, Carl Max v. 1856. *Elemente der Vermessungskunde: ein Lehrbuch der praktischen Geometrie.* Cotta.

[23] Bazin, J.C., Seo, Y., Hartley, R.I., and Pollefeys, M. 2014. Globally optimal inlier set maximization with unknown rotation and focal length. Pages 803–817 of: *European Conf. on Computer Vision (ECCV).*

[24] Behley, J., and Stachniss, C. 2018. Efficient Surfel-Based SLAM using 3D Laser Range Data in Urban Environments. In: *Robotics: Science and Systems (RSS).*

[25] Behley, Jens, Steinhage, Volker, and Cremers, Armin B. 2015. Efficient Radius Neighbor Search in Three-dimensional Point Clouds. In: *IEEE Intl. Conf. on Robotics and Automation (ICRA).*

[26] Bentley, J.L. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, **18**(9), 509–517.

[27] Bezdek, James C, and Hathaway, Richard J. 2003. Convergence of alternating optimization. *Neural, Parallel & Scientific Computations*, **11**(4), 351–368.

[28] Bhardwaj, Mohak, Boots, Byron, and Mukadam, Mustafa. 2020. Differentiable gaussian process motion planning. Pages 10598–10604 of: *2020 IEEE International Conference on Robotics and Automation (ICRA).* IEEE.

[29] Biber, Peter, and Straßer, Wolfgang. 2003. The normal distributions transform: A new approach to laser scan matching. Pages 2743–2748 of: *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, vol. 3. IEEE.

[30] Bierman, G.J. 1977. *Factorization methods for discrete sequential estimation.* Mathematics in Science and Engineering, vol. 128. New York: Academic Press.

[31] Black, Michael J., and Rangarajan, Anand. 1996. On the unification of line processes, outlier rejection, and robust statistics with applications in early vision. *Intl. J. of Computer Vision*, **19**(1), 57–91.

[32] Blake, Andrew, and Zisserman, Andrew. 1987. *Visual reconstruction.* MIT Press.

[33] Bosse, M., Agamennoni, G., and Gilitschenski, I. 2016. Robust Estimation and Applications in Robotics. *Foundations and Trends in Robotics*, **4**(4), 225–269.

[34] Botsch, M., Spernat, M., and Kobbelt, L. 2004. Phong splatting. In: *Symposium on Point-Based Graphics (PBG)*.

[35] Botsch, Mario, Hornung, Alexander, Zwicker, Matthias, and Kobbelt, Leif. 2005. High-Quality Surface Splatting on Today's GPUs. In: *Symposium on Point-Based Graphics (PBG)*.

[36] Boumal, Nicolas. 2022 (Mar). *An introduction to optimization on smooth manifolds*. To appear with Cambridge University Press.

[37] Bradbury, James, Frostig, Roy, Hawkins, Peter, Johnson, Matthew James, Leary, Chris, Maclaurin, Dougal, Necula, George, Paszke, Adam, VanderPlas, Jake, Wanderman-Milne, Skye, and Zhang, Qiao. 2018. *JAX: composable transformations of Python+NumPy programs*.

[38] Bustos, Á. P., and Chin, T. J. 2018. Guaranteed outlier removal for point cloud registration with correspondences. **40**(12), 2868–2882.

[39] Bustos, Alvaro Parra, Chin, Tat-Jun, Neumann, Frank, Friedrich, Tobias, and Katzmann, Maximilian. 2019. A Practical Maximum Clique Algorithm for Matching with Pairwise Constraints. *arXiv preprint arXiv:1902.01534*.

[40] Bylow, Erik, Sturm, Jürgen, Kerl, Christian, Kahl, Fredrik, and Cremers, Daniel. 2013. Real-time camera tracking and 3D reconstruction using signed distance functions. Page 2 of: *Robotics: Science and Systems (RSS)*, vol. 2.

[41] Cadena, C., Carlone, L., Carrillo, H., Latif, Y., Scaramuzza, D., Neira, J., Reid, I., and Leonard, J. J. 2016. Past, Present, and Future of Simultaneous Localization and Mapping: Toward the Robust-Perception Age. *IEEE Trans. Robotics*, **32**(6), 1309–1332.

[42] Carlone, L. 2023. Estimation Contracts for Outlier-Robust Geometric Perception. *Foundations and Trends (FnT) in Robotics, arXiv preprint: 2208.10521*. .

[43] Carlone, L., and Calafiore, G. 2018. Convex Relaxations for Pose Graph Optimization with Outliers. *IEEE Robotics and Automation Letters*, **3**(2), 1160–1167. arxiv preprint: 1801.02112, .

[44] Carr, J. C., Beatson, R. K., Cherrie, J. B., Mitchell, T. J., Fright, W. R., McCallum, B. C., and Evans, T. R. 2001. Reconstruction and representation of 3D objects with radial basis functions. Pages 67–76 of: *Intl. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*. Association for Computing Machinery.

[45] Cazals, Frédéric, and Giesen, Joachim. 2006. Delaunay Triangulation Based Surface Reconstruction. In: *Effective Computational Geometry for Curves and Surfaces*.

[46] Chadwick, Jeffrey N, and Bindel, David S. 2015. An efficient solver for sparse linear systems based on rank-structured Cholesky factorization. *arXiv preprint arXiv:1507.05593*.

[47] Chang, Y., Ebadi, K., Denniston, C., Ginting, M. Fadhil, Rosinol, A., Reinke, A., Palieri, M., Shi, J., A, Chatterjee, Morrell, B., Agha-mohammadi, A., and Carlone, L. 2022. LAMP 2.0: A Robust Multi-Robot SLAM System for Operation in Challenging Large-Scale Underground Environments. *IEEE Robotics and Automation Letters*, **7**(4), 9175–9182. .

[48] Chatila, R., and Laumond, J.-P. 1985. Position referencing and consistent world modeling for mobile robots. Pages 138–145 of: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[49] Chatterjee, A., and Govindu, V. M. 2013. Efficient and Robust Large-Scale Rotation Averaging. Pages 521–528 of: *Intl. Conf. on Computer Vision (ICCV)*.

[50] Chebrolu, Nived, Läbe, Thomas, Vysotska, Olga, Behley, Jens, and Stachniss, Cyrill. 2020. Adaptive Robust Kernels for Non-Linear Least Squares Problems. *arXiv preprint arXiv:2004.14938*.

[51] Chen, Hansheng, Wang, Pichao, Wang, Fan, Tian, Wei, Xiong, Lu, and Li, Hao. 2022. EPro-PnP: Generalized End-to-End Probabilistic Perspective-n-Points for Monocular Object Pose Estimation. Pages 2781–2790 of: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

[52] Chen, Tianqi, Li, Mu, Li, Yutian, Lin, Min, Wang, Naiyan, Wang, Minjie, Xiao, Tianjun, Xu, Bing, Zhang, Chiyuan, and Zhang, Zheng. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*.

[53] Chen, Xiangyu, Yang, Fan, and Wang, Chen. 2024. iA*: Imperative Learning-based A* Search for Pathfinding. *arXiv preprint arXiv:2403.15870*.

[54] Chen, Y., Davis, T.A., Hager, W.W., and Rajamanickam, S. 2008. Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/-Downdate. *ACM Trans. Math. Softw.*, **35**(3), 22:1–22:14.

[55] Chin, T., Kee, Y. H., Eriksson, A., and Neumann, F. 2016 (June). Guaranteed Outlier Removal with Mixed Integer Linear Programs. Pages 5858–5866 of: *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*.

[56] Chum, O., and Matas, J. 2005. Matching with PROSAC - Progressive Sample Consensus. In: *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*.

[57] Chum, Ondřej, Matas, Jiří, and Kittler, Josef. 2003. Locally optimized RANSAC. Pages 236–243 of: *Joint Pattern Recognition Symposium*. Springer.

[58] Clark, J.H. 1976. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, **19**(1), 547–554.

[59] Collobert, Ronan, Bengio, Samy, and Mariéthoz, Johnny. 2002. *Torch: a modular machine learning software library*. Tech. rept. Idiap.

[60] Crowley, J.L. 1989. World modeling and position estimation for a mobile robot using ultra-sonic ranging. Pages 674–680 of: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[61] Curless, Brian, and Levoy, Marc. 1996. A volumetric method for building complex models from range images. Pages 303–312 of: *Intl. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*.

[62] Czarnowski, Jan, Laidlow, Tristan, Clark, Ronald, and Davison, Andrew J. 2020. Deepfactors: Real-time probabilistic dense monocular slam. *IEEE Robotics and Automation Letters*, **5**(2), 721–728.

[63] Davis, T.A. 2011. Algorithm 915: SuiteSparseQR, A multifrontal multithreaded sparse QR factorization package. *ACM Trans. Math. Softw.*, **38**(1), 8:1–8:22.

[64] Davis, T.A., Gilbert, J.R., Larimore, S.I., and Ng, E.G. 2004. A column

approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.*, **30**(3), 353–376.

[65] Dellaert, F. 2005. Square Root SAM: Simultaneous Location and Mapping via Square Root Information Smoothing. In: *Robotics: Science and Systems (RSS)*.

[66] Dellaert, Frank. 2012. *Factor graphs and GTSAM: A hands-on introduction*. Tech. rept.

[67] Dellaert, Frank, and Contributors, GTSAM. 2022 (May). *borglab/gtsam*.

[68] Dellaert, Frank, and Kaess, Michael. 2006. Square Root SAM: Simultaneous localization and mapping via square root information smoothing. *The International Journal of Robotics Research*, **25**(12), 1181–1203.

[69] Dellaert, Frank, Kaess, Michael, et al. 2017. Factor graphs for robot perception. *Foundations and Trends® in Robotics*, **6**(1-2), 1–139.

[70] Dellenbach, P., Deschaud, J., Jacquet, B., and Goulette, F. 2022. CT-ICP Real-Time Elastic LiDAR Odometry with Loop Closure. In: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[71] Deng, Junyuan, Chen, Xieyuanli, Xia, Songpengcheng, Sun, Zhen, Liu, Guoqing, Yu, Wenxian, and Pei, Ling. 2023. NeRF-LOAM: Neural Implicit Representation for Large-Scale Incremental LiDAR Odometry and Mapping. In: *Intl. Conf. on Computer Vision (ICCV)*.

[72] Dennis, J.E., and Schnabel, R.B. 1983. *Numerical methods for unconstrained optimization and nonlinear equations*. Prentice-Hall.

[73] DeTone, Daniel, Malisiewicz, Tomasz, and Rabinovich, Andrew. 2018. SuperPoint: Self-Supervised Interest Point Detection and Description. *arXiv:1712.07629 [cs]*, Apr.

[74] Droeschel, D., and Behnke, S. 2018. Efficient Continuous-Time SLAM for 3D Lidar-Based Online Mapping. In: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[75] Duberg, D., and Jensfelt, P. 2020. UFOMap: An Efficient Probabilistic 3D Mapping Framework That Embraces the Unknown. *IEEE Robotics and Automation Letters*, **5**(4), 6411–6418.

[76] Duckett, T., Marsland, S., and Shapiro, J. 2002. Fast, On-line Learning of Globally Consistent Maps. *Autonomous Robots*, **12**(3), 287–300.

[77] Duff, I. S., and Reid, J. K. 1983. The Multifrontal Solution of Indefinite Sparse Symmetric Linear Systems. *ACM Trans. Math. Softw.*, **9**(3), 302–325.

[78] Dunning, Iain, Huchette, Joey, and Lubin, Miles. 2017. JuMP: A modeling language for mathematical optimization. *SIAM review*, **59**(2), 295–320.

[79] Durrant-Whyte, H.F. 1988. Uncertain geometry in robotics. *IEEE Trans. Robot. Automat.*, **4**(1), 23–31.

[80] Durrant-Whyte, H.F., Rye, D., and Nebot, E. 1996. Localisation of automatic guided vehicles. Pages 613–625 of: Giralt, G., and Hirzinger, G. (eds), *Robotics Research: The 7th International Symposium (ISRR 95)*. Springer-Verlag.

[81] Ebadi, K., Chang, Y., Palieri, M., Stephens, A., Hatteland, A., Heiden, E., Thakur, A., Morrell, B., Carlone, L., and Aghamohammadi, A. 2020. LAMP: Large-Scale Autonomous Mapping and Positioning for Exploration of Perceptually-Degraded Subterranean Environments. In: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[82] Ebadi, Kamak, Bernreiter, Lukas, Biggie, Harel, Catt, Gavin, Chang, Yun, Chatterjee, Arghya, Denniston, Christopher E., Deschênes, Simon-Pierre, Harlow, Kyle, Khattak, Shehryar, Nogueira, Lucas, Palieri, Matteo, Petráček, Pavel, Petrlík, Matěj, Reinke, Andrzej, Krátký, Vít, Zhao, Shibo, Aghamohammadi, Ali-akbar, Alexis, Kostas, Heckman, Christoffer, Khosoussi, Kasra, Kottege, Navinda, Morrell, Benjamin, Hutter, Marco, Pauling, Fred, Pomerleau, François, Saska, Martin, Scherer, Sebastian, Siegwart, Roland, Williams, Jason L., and Carlone, Luca. 2024. Present and Future of SLAM in Extreme Environments: The DARPA SubT Challenge. *IEEE Trans. Robotics*, **40**, 936–959.

[83] Elfes, A. 1989. Using occupancy grids for mobile robot perception and navigation. *Computer*, **22**(6), 46–57.

[84] Elseberg, J., Borrmann, D., and Nüchter, A. 2013. One billion points in the cloud – an octree for efficient processing of 3D laser scans. *ISPRS J. of Photogrammetry and Remote Sensing (JPRS)*, **76**, 76–88.

[85] Enqvist, O., Josephson, K., and Kahl, F. 2009. Optimal correspondences from pairwise constraints. Pages 1295–1302 of: *Intl. Conf. on Computer Vision (ICCV)*.

[86] Eustice, R., Singh, H., and Leonard, J. 2005a (April). Exactly Sparse Delayed-State Filters. Pages 2417–2424 of: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[87] Eustice, R., Walter, M., and Leonard, J. 2005b (Aug). Sparse Extended Information Filters: Insights into Sparsification. Pages 3281–3288 of: *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*.

[88] Feige, Uriel, Goldwasser, Shafi, Lovász, László, Safra, Shmuel, and Szegedy, Mario. 1991 (Sept.). Approximating clique is almost NP-complete. Pages 2–12 of: *Symp. on Foundations of Computer Science*.

[89] Fischler, M., and Bolles, R. 1981. Random sample consensus: a paradigm for model fitting with application to image analysis and automated cartography. *Commun. ACM*, **24**, 381–395.

[90] Fornasier, A., van Goor, P., Allak, E., Mahony, R., and Weiss, S. 2024. MSCEqF: A Multi State Constraint Equivariant Filter for Vision-Aided Inertial Navigation. *IEEE Robotics and Automation Letters*, **9**(1), 731–738.

[91] Forsgren, Brendon, Kaess, Michael, Vasudevan, Ram, McLain, Timothy W., and Mangelson, Joshua G. 2024. Group-k consistent measurement set maximization via maximum clique over k-uniform hypergraphs for robust multi-robot map merging. *Intl. J. of Robotics Research*.

[92] Frese, U. 2005. Treemap: An $O(\log n)$ Algorithm for Simultaneous Localization and Mapping. Pages 455–476 of: *Spatial Cognition IV*. Springer Verlag.

[93] Frese, U., Larsson, P., and Duckett, T. 2005. A Multilevel Relaxation Algorithm for Simultaneous Localisation and Mapping. *IEEE Trans. Robotics*, **21**(2), 196–207.

[94] Fu, Taimeng, Su, Shaoshu, Lu, Yiren, and Wang, Chen. 2024. iSLAM: Imperative SLAM. *IEEE Robotics and Automation Letters (RA-L)*.

[95] Funk, Nils, Tarrio, Juan, Papatheodorou, Sotiris, Popović, Marija, Alcantarilla, Pablo F., and Leutenegger, Stefan. 2021. Multi-Resolution 3D Mapping With Explicit Free Space Representation for Fast and Accurate Mobile Robot Motion Planning. *IEEE Robotics and Automation Letters*, **6**(2), 3553–3560.

[96] Geiger, A., Lenz, P., and Urtasun, R. 2012 (June). Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. Pages 3354–3361 of: *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*.

[97] Geneva, Patrick, Eckenhoff, Kevin, Yang, Yulin, and Huang, Guoquan. 2018. LIPS: LiDAR-Inertial 3D Plane SLAM. Pages 123–130 of: *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*.

[98] Gentil, Cédric Le, Vayugundla, Mallikarjuna, Giubilato, Riccardo, Sturzl, Wolfgang, Vidal-Calleja, Teresa, and Triebel, Rudolph. 2020. Gaussian Process Gradient Maps for Loop-Closure Detection in Unstructured Planetary Environments. Pages 1895–1902 of: *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*.

[99] Ghaffari Jadidi, Maani, Valls Miro, Jaime, and Dissanayake, Gamini. 2018. Gaussian processes autonomous mapping and exploration for range-sensing mobile robots. *Autonomous Robots*, **42**(2), 273–290.

[100] Giftthaler, Markus, Neunert, Michael, Stäuble, Markus, and Buchli, Jonas. 2018. The control toolbox-an open-source c++ library for robotics, optimal and model predictive control. Pages 123–129 of: *2018 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*. IEEE.

[101] Golub, G.H., and Loan, C.F. Van. 1996. *Matrix Computations*. Third edn. Baltimore: Johns Hopkins University Press.

[102] Grinvald, Margarita, Furrer, Fadri, Novkovic, Tonci, Chung, Jen Jen, Cadena, Cesar, Siegwart, Roland, and Nieto, Juan. 2019. Volumetric instance-aware semantic mapping and 3D object discovery. *IEEE Robotics and Automation Letters*, **4**(3), 3037–3044.

[103] Grisetti, G., Stachniss, C., and Burgard, W. 2007. Improved Techniques for Grid Mapping With Rao-Blackwellized particle filters. *IEEE Trans. Robotics*, **23**(1), 34–46.

[104] Grisetti, Giorgio, Kümmerle, Rainer, Strasdat, Hauke, and Konolige, Kurt. 2011. g2o: A general framework for (hyper) graph optimization. Pages 9–13 of: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.

[105] Guennebaud, Gaël, Jacob, Benoit, et al. 2010. Eigen. *URl: http://eigen. tuxfamily. org*, **3**.

[106] Guo, Yifan, Ren, Zhongqiang, and Wang, Chen. 2024. iMTSP: Solving Min-Max Multiple Traveling Salesman Problem with Imperative Learning. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

[107] Gutmann, J.-S., and Konolige, K. 2000 (November). Incremental Mapping of Large Cyclic Environments. Pages 318–325 of: *IEEE Intl. Symp. on Computational Intelligence in Robotics and Automation (CIRA)*.

[108] Han, Luxin, Gao, Fei, Zhou, Boyu, and Shen, Shaojie. 2019. Fiesta: Fast incremental euclidean distance fields for online motion planning of aerial robots. Pages 4423–4430 of: *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*.

[109] Handa, Ankur, Bloesch, Michael, Pătrăucean, Viorica, Stent, Simon, McCormac, John, and Davison, Andrew. 2016. gvnn: Neural network library for geometric computer vision. Pages 67–82 of: *Computer Vision–ECCV 2016*

Workshops: Amsterdam, The Netherlands, October 8-10 and 15-16, 2016, Proceedings, Part III 14. Springer.

[110] Hart, William E, Laird, Carl D, Watson, Jean-Paul, Woodruff, David L, Hackebeil, Gabriel A, Nicholson, Bethany L, Siirola, John D, et al. 2017. *Pyomo-optimization modeling in python*. Vol. 67. Springer.

[111] Hartley, R., and Zisserman, A. 2000. *Multiple View Geometry in Computer Vision*. Cambridge University Press.

[112] Hartley, R.I., and Kahl, F. 2009. Global optimization through rotation space search. *Intl. J. of Computer Vision*, **82**(1), 64–79.

[113] Hestenes, Magnus R, and Stiefel, Eduard. 1952. Methods of conjugate gradients for solving. *Journal of research of the National Bureau of Standards*, **49**(6), 409.

[114] Hoppe, Hugues, DeRose, Tony, Duchamp, Tom, McDonald, John, and Stuetzle, Werner. 1992. Surface reconstruction from unorganized points. Pages 71–78 of: *Intl. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*. Association for Computing Machinery.

[115] Horn, Berthold K. P. 1987. Closed-form solution of absolute orientation using unit quaternions. *J. Opt. Soc. Am. A*, **4**(4), 629–642.

[116] Hornung, Armin, Wurm, Kai M., Bennewitz, Maren, Stachniss, Cyrill, and Burgard, Wolfram. 2013. OctoMap: an efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots*, 189–206.

[117] Huber, P. 1981. *Robust Statistics*. John Wiley & Sons, New York, NY.

[118] Ivan, Jean-Paul A, Stoyanov, Todor, and Stork, Johannes A. 2022. Online Distance Field Priors for Gaussian Process Implicit Surfaces. *IEEE Robotics and Automation Letters*, **7**(4), 8996–9003.

[119] Izadi, Shahram, Kim, David, Hilliges, Otmar, Molyneaux, David, Newcombe, Richard, Kohli, Pushmeet, Shotton, Jamie, Hodges, Steve, Freeman, Dustin, Davison, Andrew, et al. 2011. KinectFusion: real-time 3D reconstruction and interaction using a moving depth camera. Pages 559–568 of: *ACM Symp. on User interface software and technology*.

[120] Izatt, G., Dai, H., and Tedrake, R. 2017. Globally Optimal Object Pose Estimation in Point Clouds with Mixed-Integer Programming. In: *Intl. Symp. of Robotics Research (ISRR)*.

[121] Jatavallabhula, Krishna Murthy, Iyer, Ganesh, and Paull, Liam. 2020. ∇ slam: Dense slam meets automatic differentiation. Pages 2130–2137 of: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE.

[122] Ji, Kaiyi, Yang, Junjie, and Liang, Yingbin. 2021. Bilevel optimization: Convergence analysis and enhanced design. Pages 4882–4892 of: *International conference on machine learning*. PMLR.

[123] Jia, Yangqing, Shelhamer, Evan, Donahue, Jeff, Karayev, Sergey, Long, Jonathan, Girshick, Ross, Guadarrama, Sergio, and Darrell, Trevor. 2014. Caffe: Convolutional architecture for fast feature embedding. Pages 675–678 of: *22nd ACM international conference on Multimedia*.

[124] Johnson, J, Mangelson, J, Barfoot, T D, and Beard, R. 2024. *Continuous-time Trajectory Estimation: A Comparative Study Between Gaussian Process and Spline-based Approaches*. (arXiv:2402.00399 [cs.RO]).

[125] Kaess, M., Ranganathan, A., and Dellaert, F. 2008. iSAM: Incremental Smoothing and Mapping. *IEEE Trans. Robotics*, **24**(6), 1365–1378.

[126] Kaess, M., Johannsson, H., Roberts, R., Ila, V., Leonard, J., and Dellaert, F. 2011 (May). iSAM2: Incremental Smoothing and Mapping with Fluid Relinearization and Incremental Variable Reordering. In: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[127] Kaess, M., Johannsson, H., Roberts, R., Ila, V., Leonard, J., and Dellaert, F. 2012. iSAM2: Incremental Smoothing and Mapping Using the Bayes Tree. *Intl. J. of Robotics Research*, **31**(Feb), 217–236.

[128] Kaess, Michael. 2015. Simultaneous localization and mapping with infinite planes. Pages 4605–4611 of: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[129] Kazhdan, Michael, Bolitho, Matthew, and Hoppe, Hugues. 2006. Poisson surface reconstruction. Pages 61–70 of: *Proceedings of the fourth Eurographics symposium on Geometry processing*. Eurographics Association.

[130] Keetha, Nikhil, Karhade, Jay, Jatavallabhula, Krishna Murthy, Yang, Gengshan, Scherer, Sebastian, Ramanan, Deva, and Luiten, Jonathon. 2023. SplaTAM: Splat, Track & Map 3D Gaussians for Dense RGB-D SLAM. *arXiv preprint*.

[131] Keller, M., Lefloch, D., Lambers, M., and Izadi, S. 2013. Real-time 3D Reconstruction in Dynamic Scenes using Point-based Fusion. In: *Intl. Conf. on 3D Vision (3DV)*.

[132] Kerbl, Bernhard, Kopanas, Georgios, Leimkühler, Thomas, and Drettakis, George. 2023. 3D Gaussian Splatting for Real-Time Radiance Field Rendering. *arXiv preprint*.

[133] Kerl, Christian, Sturm, Jürgen, and Cremers, Daniel. 2013. Robust odometry estimation for RGB-D cameras. Pages 3748–3754 of: *2013 IEEE international conference on robotics and automation*. IEEE.

[134] Kim, S., and Kim, J. 2012. Building occupancy maps with a mixture of Gaussian processes. Pages 4756–4761 of: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[135] Kim, S., and Kim, J. 2013. Occupancy Mapping and Surface Reconstruction Using Local Gaussian Processes With Kinect Sensors. Pages 1335–1346 of: *IEEE Trans. on Cybernetics*.

[136] Kim, Soohwan, and Kim, Jonghyuk. 2015. *GPmap: A Unified Framework for Robotic Mapping Based on Sparse Gaussian Processes*. Springer International Publishing. Pages 319–332.

[137] Kingma, Diederik P, and Ba, Jimmy. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[138] Koenemann, Jonas, Licitra, Giovanni, Alp, Mustafa, and Diehl, Moritz. 2017. *Openocl–open optimal control library*.

[139] Koller, D., and Friedman, N. 2009. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press.

[140] Konolige, K. 2004. Large-scale map-making. In: *Proc. 21^{th} AAAI National Conference on AI*.

[141] Kukko, Antero, Kaijaluoto, Risto, Kaartinen, Harri, Lehtola, Ville V, Jaakkola, Anttoni, and Hyyppä, Juha. 2017. Graph SLAM correction for single scanner MLS forest data under boreal forest canopy. *ISPRS Journal of Photogrammetry and Remote Sensing*, **132**, 199–209.

[142] Lajoie, P., Hu, S., Beltrame, G., and Carlone, L. 2019. Modeling Perceptual

Aliasing in SLAM via Discrete-Continuous Graphical Models. *IEEE Robotics and Automation Letters*. extended ArXiv version: , Supplemental Material: .

[143] Latif, Y., Lerma, C. D. C., and Neira, J. 2012. Robust Loop Closing Over Time. In: *Robotics: Science and Systems (RSS)*.

[144] Lau, Boris, Sprunk, Christoph, and Burgard, Wolfram. 2013. Efficient grid-based spatial representations for robot navigation in dynamic environments. *J. on Robotics and Autonomous Systems (RAS)*, **61**(10), 1116–1130.

[145] Le Gentil, Cedric, Ouabi, Othmane-Latif, Wu, Lan, Pradalier, Cedric, and Vidal-Calleja, Teresa. 2023. Accurate Gaussian-Process-based Distance Fields with Applications to Echolocation and Mapping. *IEEE Robotics and Automation Letters*, 1–8.

[146] Lee, Bhoram, Zhang, Clark, Huang, Zonghao, and Lee, Daniel D. 2019. Online continuous mapping using gaussian process implicit surfaces. In: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[147] Lee, G. H., Fraundorfer, F., and Pollefeys, M. 2013. Robust pose-graph loop-closures with expectation-maximization. In: *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*.

[148] Leordeanu, Marius, and Hebert, Martial. 2005. A spectral technique for correspondence problems using pairwise constraints. Pages 1482–1489 of: *Intl. Conf. on Computer Vision (ICCV)*, vol. 2. IEEE.

[149] Levenberg, K. 1944. A Method for the Solution of Certain Nonlinear Problems in Least Squares. *Quart. Appl. Math*, **2**(2), 164–168.

[150] Li, H. 2009. Consensus set maximization with guaranteed global optimality for robust geometry estimation. Pages 1074–1080 of: *Intl. Conf. on Computer Vision (ICCV)*.

[151] Li, Ruihao, Wang, Sen, Long, Zhiqiang, and Gu, Dongbing. 2018. Undeepvo: Monocular visual odometry through unsupervised deep learning. Pages 7286–7291 of: *2018 IEEE international conference on robotics and automation (ICRA)*. IEEE.

[152] Lin, J., and Zhang, F. 2019. Loam_livox A Robust LiDAR Odemetry and Mapping LOAM Package for Livox LiDAR. In: *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*.

[153] Liu, Hanxiao, Simonyan, Karen, and Yang, Yiming. 2019. Darts: Differentiable architecture search. In: *International Conference on Learning Representations (ICLR)*.

[154] Liu, Liyang, Fryc, Simon, Wu, Lan, Vu, Thanh Long, Paul, Gavin, and Vidal-Calleja, Teresa. 2021a. Active and interactive mapping with dynamic Gaussian process implicit surfaces for mobile manipulators. *IEEE Robotics and Automation Letters*, **6**(2), 3679–3686.

[155] Liu, Risheng, Gao, Jiaxin, Zhang, Jin, Meng, Deyu, and Lin, Zhouchen. 2021b. Investigating bi-level optimization for learning and vision from a unified perspective: A survey and beyond. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **44**(12), 10045–10067.

[156] Loop, C., Cai, Q., Orts-Escolano, S., and Chou, P. A. 2016. A Closed-Form Bayesian Fusion Equation Using Occupancy Probabilities. Pages 380–388 of: *Intl. Conf. on 3D Vision (3DV)*.

[157] Lorensen, William E, and Cline, Harvey E. 1987. Marching cubes: A high resolution 3D surface construction algorithm. *Intl. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*, **21**(4), 163–169.

[158] Lourakis, Manolis LA, and Argyros, Antonis A. 2005a. Is Levenberg-Marquardt the most efficient optimization algorithm for implementing bundle adjustment? Pages 1526–1531 of: *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, vol. 2. IEEE.

[159] Lourakis, MLA, and Argyros, Antonis A. 2005b. Is Levenberg-Marquardt the most efficient optimization algorithm for implementing bundle adjustment? Pages 1526–1531 of: *Intl. Conf. on Computer Vision (ICCV)*, vol. 2. IEEE.

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

[161] Lu, F., and Milios, E. 1997b. Robot pose estimation in unknown environments by matching 2D range scans. *J. of Intelligent and Robotic Systems*, April, 249:275.

[162] Lu, Feng, and Milios, Evangelos. 1997c. Globally consistent range scan alignment for environment mapping. *Autonomous Robots*, **4**, 333–349.

[163] Lusk, Parker C., Fathian, Kaveh, and How, Jonathan P. 2021a (May). CLIPPER: A Graph-Theoretic Framework for Robust Data Association. Pages 13828–13834 of: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[164] Lusk, Parker C., Roy, Ronak, Fathian, Kaveh, and How, Jonathan P. 2021b (Nov.). *MIXER: A Principled Framework for Multimodal, Multiway Data Association*.

[165] Lusk, Parker C., Parikh, Devarth, and How, Jonathan P. 2023. GraffMatch: Global Matching of 3D Lines and Planes for Wide Baseline LiDAR Registration. *IEEE Robotics and Automation Letters*, **8**(2), 632–639.

[166] Lv, Zhaoyang, Dellaert, Frank, Rehg, James M, and Geiger, Andreas. 2019. Taking a deeper look at the inverse compositional algorithm. Pages 4581–4590 of: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

[167] Macenski, Steve, Tsai, David, and Feinberg, Max. 2020. Spatio-temporal voxel layer: A view on robot perception for the dynamic world. *Intl. J. of Advanced Robotic Systems*, **17**(2).

[168] Malcolm, James, Yalamanchili, Pavan, McClanahan, Chris, Venugopalakrishnan, Vishwanath, Patel, Krunal, and Melonakos, John. 2012. ArrayFire: a GPU acceleration platform. Pages 49–56 of: *Modeling and simulation for defense systems and applications VII*, vol. 8403. SPIE.

[169] Mangelson, J. G., Dominic, D., Eustice, R. M., and Vasudevan, R. 2018. Pairwise Consistent Measurement Set Maximization for Robust Multi-robot Map Merging. Pages 2916–2923 of: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[170] Marquardt, D.W. 1963. An Algorithm for Least-Squares Estimation of Nonlinear Parameters. *J. Soc. Indust. Appl. Math.*, **11**(2), 431–441.

[171] Martens, W., Poffet, Y., Soria, P. R., Fitch, R., and Sukkarieh, S. 2017. Geometric Priors for Gaussian Process Implicit Surfaces. *IEEE Robotics and Automation Letters*, 373–380.

[172] MATLAB. 2010. *version 7.10.0 (R2010a)*. Natick, Massachusetts: The MathWorks Inc.

[173] Matsuki, Hidenobu, Murai2, Riku, Kelly, Paul H. J., and Davison, Andrew J. 2023. Gaussian Splatting SLAM.

[174] Maybeck, P. 1979. *Stochastic Models, Estimation and Control*. Vol. 1. New York: Academic Press.

[175] Meagher, D. 1980. Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer. *Technical Report*, **Image Processing Laboratory, Rensselaer Polytechnic Institute**(IPL-TR-80-111).

[176] Melkumyan, Arman, and Ramos, Fabio Tozeto. 2009. A sparse covariance function for exact Gaussian process inference in large datasets. In: *Intl. Joint Conf. on AI (IJCAI)*.

[177] Mildenhall, B., Srinivasan, P.P., Tancik, M., Barron, J.T., Ramamoorthi, R., and Ng, R. 2020. NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. In: *European Conf. on Computer Vision (ECCV)*.

[178] Montemerlo, M., Thrun, S., Koller, D., and Wegbreit, B. 2002. FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem. In: *Proc.* 19^{th} *AAAI National Conference on AI*.

[179] Moravec, Hans, and Elfes, Alberto. 1985. High resolution maps from wide angle sonar. Pages 116–121 of: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[180] Mourikis, A.I., and Roumeliotis, S.I. 2007 (April). A Multi-State Constraint Kalman Filter for Vision-aided Inertial Navigation. Pages 3565–3572 of: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[181] Mullane, J., Vo, B-N., Adams, M., and Vo, B-T. 2011. A Random-Finite-Set Approach to Bayesian SLAM. *IEEE Trans. Robotics*, **27**(2), 268–282.

[182] Mur-Artal, Raul, Montiel, Jose Maria Martinez, and Tardos, Juan D. 2015. ORB-SLAM: a versatile and accurate monocular SLAM system. *IEEE Trans. Robotics*, **31**(5), 1147–1163.

[183] Museth, Ken. 2021. NanoVDB: A GPU-Friendly and Portable VDB Data Structure For Real-Time Rendering And Simulation. In: *Intl. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*.

[184] Museth, Ken, Lait, Jeff, Johanson, John, Budsberg, Jeff, Henderson, Ron, Alden, Mihai, Cucka, Peter, Hill, David, and Pearce, Andrew. 2013. OpenVDB: an open-source data structure and toolkit for high-resolution volumes. In: *Intl. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*.

[185] Newcombe, R. A., Izadi, S., Hilliges, O., Molyneaux, D., Kim, D., Davison, A. J., Kohli, P., Shotton, J., Hodges, S., and Fitzgibbon, A. 2011. KinectFusion: Real-Time Dense Surface Mapping and Tracking. In: *IEEE and ACM Intl. Sym. on Mixed and Augmented Reality (ISMAR)*.

[186] Nießner, M., Zollhöfer, M., Izadi, S., and Stamminger, M. 2013. Real-time 3D Reconstruction at Scale using Voxel Hashing. In: *Intl. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH) Asia*.

[187] Nieto, J., Guivant, H., Nebot, E., and Thrun, S. 2003. Real Time Data Association for FastSLAM. In: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[188] Nistér, D. 2003. An Efficient Solution to the Five-Point Relative Pose Problem. In: *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*.

[189] Nocedal, Jorge, and Wright, Stephen J. 1999. *Numerical Optimization*. Springer Series in Operations Research. Springer-Verlag.

[190] O'Callaghan, Simon T, and Ramos, Fabio T. 2012. Gaussian process occupancy maps. *Intl. J. of Robotics Research*, **31**(1), 42–62.

[191] Ochs, Peter, Dosovitskiy, Alexey, Brox, Thomas, and Pock, Thomas. 2015. On iteratively reweighted algorithms for nonsmooth nonconvex optimization in computer vision. *SIAM Journal on Imaging Sciences*, **8**(1), 331–372.

[192] Ogayar-Anguita, C. J., Lopez-Ruiz, A., Rueda-Ruiz, A. J., and Segura-Sanchez, Rafael J. 2023. Nested spatial data structures for optimal indexing of LiDAR data. *ISPRS J. of Photogrammetry and Remote Sensing (JPRS)*, **195**, 287–297.

[193] Oleynikova, H., Taylor, Z., Fehr, M., Siegwart, R., and Nieto, J. 2017. Voxblox: Incremental 3D Euclidean Signed Distance Fields for on-board MAV planning. Pages 1366–1373 of: *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*.

[194] Oliphant, Travis E. 2006. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA.

[195] Olson, E., Leonard, J., and Teller, S. 2006 (May). Fast Iterative Alignment of Pose Graphs with Poor Initial Estimates. Pages 2262–2269 of: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[196] Olson, Edwin, and Agarwal, Pratik. 2012 (July). Inference on networks of mixtures for robust robot mapping. In: *Robotics: Science and Systems (RSS)*.

[197] Open, NN. 2016. An open source neural networks c++ library. *URL: http://opennn.cimne.com (2016)*.

[198] Pan, Yue, Xiao, Pengchuan, He, Yujie, Shao, Zhenlei, and Li, Zesong. 2021. MULLS: Versatile LiDAR SLAM via multi-metric linear least square. Pages 11633–11640 of: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*. IEEE.

[199] Parameshwara, Chethan M, Hari, Gokul, Fermüller, Cornelia, Sanket, Nitin J, and Aloimonos, Yiannis. 2022. DiffPoseNet: Direct differentiable camera pose estimation. Pages 6845–6854 of: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

[200] Park, C., Moghadam, P., Kim, S., Elfes, A., Fookes, C., and Sridharan, S. 2018. Elastic LiDAR Fusion: Dense Map-Centric Continuous-Time SLAM. In: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[201] Paskin, M.A. 2003. Thin Junction Tree Filters for Simultaneous Localization and Mapping. In: *Intl. Joint Conf. on AI (IJCAI)*.

[202] Paszke, Adam, Gross, Sam, Massa, Francisco, Lerer, Adam, Bradbury, James, Chanan, Gregory, Killeen, Trevor, Lin, Zeming, Gimelshein, Natalia, Antiga, Luca, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, **32**.

[203] Pattabiraman, Bharath, Patwary, Md. Mostofa Ali, Gebremedhin, Assefaw H., keng Liao, Wei, and Choudhary, Alok. 2015. Fast Algorithms for the Maximum Clique Problem on Massive Graphs with Applications to Overlapping Community Detection. *Internet Mathematics*, **11**(4-5), 421–448.

[204] Pearlmutter, Barak A, and Siskind, Jeffrey Mark. 2008. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **30**(2), 1–36.

[205] Peng, Liangzu, Kümmerle, Christian, and Vidal, René. 2023. On the Convergence of IRLS and Its Variants in Outlier-Robust Estimation. Pages 17808–17818 of: *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*.

[206] Peng, Songyou, Niemeyer, Michael, Mescheder, Lars, Pollefeys, Marc, and Geiger, Andreas. 2020. Convolutional occupancy networks. In: *European Conf. on Computer Vision (ECCV)*.

[207] Perera, Samunda, and Barnes, Nick. 2012. Maximal cliques based rigid body motion segmentation with a RGB-D camera. Pages 120–133 of: *Asian Conf. on Computer Vision*. Springer.

[208] Pfister, H., Zwickler, M., v. Baar, J., and Gross, M. 2000. Surfels: surface elements as rendering primitives. In: *Intl. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*.

[209] P.H.S. Torr, A. Zisserman. 2000. *MLESAC: A New Robust Estimator with Application to Estimating Image Geometry*. Tech. rept. MSR-TR-99-60. MSR.

[210] Pineda, Luis, Fan, Taosha, Monge, Maurizio, Venkataraman, Shobha, Sodhi, Paloma, Chen, Ricky T. Q., Ortiz, Joseph, DeTone, Daniel, Wang, Austin S., Anderson, Stuart, Dong, Jing, Amos, Brandon, and Mukadam, Mustafa. 2022a (Oct.). Theseus: A Library for Differentiable Nonlinear Optimization. In: *Conf. Neural Information Processing Systems (NIPS)*.

[211] Pineda, Luis, Fan, Taosha, Monge, Maurizio, Venkataraman, Shobha, Sodhi, Paloma, Chen, Ricky TQ, Ortiz, Joseph, DeTone, Daniel, Wang, Austin, Anderson, Stuart, Dong, Jing, Amos, Brandon, and Mukadam, Mustafa. 2022b. Theseus: A Library for Differentiable Nonlinear Optimization. *Advances in Neural Information Processing Systems*.

[212] Powell, M.J.D. 1970. A New Algorithm for Unconstrained Optimization. Pages 31–65 of: Rosen, J., Mangasarian, O., and Ritter, K. (eds), *Nonlinear Programming*. Academic Press.

[213] Qin, Chao, Ye, Haoyang, Pranata, Christian E, Han, Jun, Zhang, Shuyang, and Liu, Ming. 2020. LINS: A Lidar-Inertial State Estimator for Robust and Efficient Navigation. In: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[214] Qin, Tong, Li, Peiliang, and Shen, Shaojie. 2018. VINS-Mono: A Robust and Versatile Monocular Visual-Inertial State Estimator. *IEEE Trans. Robotics*, **34**(4), 1004–1020.

[215] Rahimi, Ali, and Recht, Benjamin. 2007. Random features for large-scale kernel machines. In: *Advances in Neural Information Processing Systems (NIPS)*, vol. 20.

[216] Ramos, Fabio, and Ott, Lionel. 2016. Hilbert maps: Scalable continuous occupancy mapping with stochastic gradient descent. *Intl. J. of Robotics Research*, **35**(14), 1717–1730.

[217] Rasmussen, Carl Edward, and Williams, Christopher KI. 2006. *Gaussian Processes for Machine Learning*. Cambridge, Mass.: MIT Press.

[218] Ravichandran, Z., Peng, L., Hughes, N., Griffith, J.D., and Carlone, L. 2022. Hierarchical Representations and Explicit Memory: Learning Effective Navigation Policies on 3D Scene Graphs using Graph Neural Networks. In: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*. .

[219] Reijgwart, V., Millane, A., Oleynikova, H., Siegwart, R., Cadena, C., and Nieto, J. 2020. Voxgraph: Globally Consistent, Volumetric Mapping Using Signed Distance Function Submaps. *IEEE Robotics and Automation Letters*.

[220] Reijgwart, Victor, Cadena, Cesar, Siegwart, Roland, and Ott, Lionel. 2023-07. Efficient volumetric mapping of multi-scale environments using wavelet-based compression.

[221] Revels, Jarrett, Lubin, Miles, and Papamarkou, Theodore. 2016. Forward-mode automatic differentiation in Julia. *arXiv preprint arXiv:1607.07892*.

[222] Robbins, Herbert, and Monro, Sutton. 1951. A stochastic approximation method. *The annals of mathematical statistics*, 400–407.

[223] Rodrigues, Rômulo T., Tsiogkas, Nikolaos, Pascoal, António, and Aguiar, A. Pedro. 2021. Online Range-Based SLAM Using B-Spline Surfaces. *IEEE Robotics and Automation Letters*, **6**(2), 1958–1965.

[224] Roriz, Ricardo, Cabral, Jorge, and Gomes, Tiago. 2022. Automotive LiDAR Technology: A Survey. *IEEE Trans. on Intelligent Transportation Systems (TITS)*, **23**(7), 6282–6297.

[225] Rosinol, Antoni, Abate, Marcus, Chang, Yun, and Carlone, Luca. 2020. Kimera: an open-source library for real-time metric-semantic localization and mapping. Pages 1689–1696 of: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[226] San Segundo, Pablo, and Artieda, Jorge. 2015. A novel clique formulation for the visual feature matching problem. *Appl. Intelligence*, **43**(2), 325–342.

[227] Sandström, Erik, Li, Yue, Van Gool, Luc, and R. Oswald, Martin. 2023. Point-SLAM: Dense Neural Point Cloud-based SLAM. In: *Intl. Conf. on Computer Vision (ICCV)*.

[228] Särkkä, Simo". 2011. Linear Operators and Stochastic Partial Differential Equations in Gaussian Process Regression. Pages 151–158 of: *International Conference on Artificial Neural Networks and Machine Learning*.

[229] Sarlin, Paul-Edouard, DeTone, Daniel, Malisiewicz, Tomasz, and Rabinovich, Andrew. 2020. SuperGlue: Learning Feature Matching with Graph Neural Networks. In: *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*.

[230] Schonberger, Johannes L, and Frahm, Jan-Michael. 2016. Structure-from-motion revisited. Pages 4104–4113 of: *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*.

[231] Sethian, James Albert. 1996. Level set methods: Evolving interfaces in geometry, fluid mechanics, computer vision, and materials science. *Cambridge monographs on applied and computational mathematics*, **3**.

[232] Shaban, Amirreza, Cheng, Ching-An, Hatch, Nathan, and Boots, Byron. 2019. Truncated back-propagation for bilevel optimization. Pages 1723–1732 of: *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR.

[233] Shan, T., Englot, B., Meyers, D., Wang, W., Ratti, C., and Rus, D. 2020. LIO-SAM: Tightly-coupled Lidar Inertial Odometry via Smoothing and Mapping. In: *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*.

[234] Shan, Tixiao, and Englot, Brendan. 2018. LeGO-LOAM: Lightweight and Ground-Optimized Lidar Odometry and Mapping on Variable Terrain. In: *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*.

[235] Shen, Chen, O'Brien, James F., and Shewchuk, Jonathan Richard. 2004. Interpolating and Approximating Implicit Surfaces from Polygon Soup. *Intl. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 9.

[236] Shi, J., Yang, H., and Carlone, L. 2021. ROBIN: a Graph-Theoretic Approach to Reject Outliers in Robust Estimation using Invariants. In: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*. arXiv preprint: 2011.03659, .

[237] Shi, J., Yang, H., and Carlone, L. 2023. Optimal and Robust Category-level Perception: Object Pose and Shape Estimation from 2D and 3D Semantic Keypoints. *IEEE Trans. Robotics*, **39**(5), 4131–4151. .

[238] Sim, R., Elinas, P., Griffin, M., Shyr, A., and Little, J.J. 2006 (Jun). Design and Analysis of a Framework for Real-time Vision-based SLAM using Rao-Blackwellised Particle Filters. In: *Proc. of the 3rd Canadian Conf. on Computer and Robotic Vision (CRV)*.

[239] Smith, R., and Cheeseman, P. 1987. On the representation and estimation of spatial uncertainty. *Intl. J. of Robotics Research*, **5**(4), 56–68.

[240] Sola, Joan, Deray, Jeremie, and Atchuthan, Dinesh. 2018. A micro Lie theory for state estimation in robotics. *arXiv preprint arXiv:1812.01537*.

[241] Speciale, P., Paudel, D. P., Oswald, M. R., Kroeger, T., Gool, L. V., and Pollefeys, M. 2017 (July). Consensus Maximization with Linear Matrix Inequality Constraints. Pages 5048–5056 of: *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*.

[242] Stadie, Bradly, Zhang, Lunjun, and Ba, Jimmy. 2020. Learning intrinsic rewards as a bi-level optimization problem. Pages 111–120 of: *Conference on Uncertainty in Artificial Intelligence*. PMLR.

[243] Stillwell, J. 2008. *Naive Lie Theory*. Springer.

[244] Stork, Johannes A, and Stoyanov, Todor. 2020. Ensemble of Sparse Gaussian Process Experts for Implicit Surface Mapping with Streaming Data. *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[245] Stoyanov, T., Saarinen, J.P., Andreasson, H., and Lilienthal, A.J. 2013. Normal Distributions Transform Occupancy Map Fusion: Simultaneous Mapping and Tracking in Large Scale Dynamic Environments. In: *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*.

[246] Stückler, J., and Behnke, S. 2014. Multi-Resolution Surfel Maps for Efficient Dense 3D Modeling and Tracking. *J. of Visual Communication and Image Representation*, **25**(1), 137–147.

[247] Stueckler, J., and Behnke, S. 2014. Efficient Deformable Registration of Multi-Resolution Surfel Maps for Object Manipulation Skill Transfer. In: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[248] Sucar, Edgar, Liu, Shikun, Ortiz, Joseph, and Davison, Andrew J. 2021. imap: Implicit mapping and positioning in real-time. In: *Intl. Conf. on Computer Vision (ICCV)*.

[249] Sünderhauf, N., and Protzel, P. 2012a. Switchable Constraints for Robust Pose Graph SLAM. In: *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*.

[250] Sünderhauf, N., and Protzel, P. 2012b. Towards a robust back-end for pose graph SLAM. Pages 1254–1261 of: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*. IEEE.

[251] Tang, Chengzhou, and Tan, Ping. 2019a. Ba-net: Dense bundle adjustment network. *International Conference on Learning Representations (ICLR)*.

[252] Tang, Chengzhou, and Tan, Ping. 2019b. BA-Net: Dense bundle adjustment networks. *7th International Conference on Learning Representations, ICLR 2019*.

[253] Tavish, K. Mac, and Barfoot, T. D. 2015. At all costs: A comparison of robust cost functions for camera correspondence outliers. Pages 62–69 of: *Conf. Computer and Robot Vision*. IEEE.

[254] Tedrake, Russ, and the Drake Development Team. 2019. *Drake: Model-based design and verification for robotics*.

[255] Teed, Zachary, and Deng, Jia. 2018. Deepv2d: Video to depth with differentiable structure from motion. *arXiv preprint arXiv:1812.04605*.

[256] Teed, Zachary, and Deng, Jia. 2021a. Droid-slam: Deep visual slam for monocular, stereo, and rgb-d cameras. *Advances in Neural Information Processing Systems*, **34**.

[257] Teed, Zachary, and Deng, Jia. 2021b. Tangent space backpropagation for 3d transformation groups. Pages 10338–10347 of: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

[258] Thrun, S., Liu, Y., Koller, D., Ng, A.Y., Ghahramani, Z., and Durrant-Whyte, H. 2004. Simultaneous Localization and Mapping With Sparse Extended Information Filters. *Intl. J. of Robotics Research*, **23**(7-8), 693–716.

[259] Thrun, S., Burgard, W., and Fox, D. 2005. *Probabilistic Robotics*. The MIT press, Cambridge, MA.

[260] Thrun, Sebastian, et al. 2002. Robotic mapping: A survey. *Exploring artificial intelligence in the new millennium*, **1**(1-35), 1.

[261] Tian, Y., Chang, Y., Arias, F. Herrera, Nieto-Granda, C., How, J.P., and Carlone, L. 2022. Kimera-Multi: Robust, Distributed, Dense Metric-Semantic SLAM for Multi-Robot Systems. *IEEE Trans. Robotics*. accepted, arXiv preprint: 2106.14386, .

[262] Tokui, Seiya, Oono, Kenta, Hido, Shohei, and Clayton, Justin. 2015. Chainer: a next-generation open source framework for deep learning. Pages 1–6 of: *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, vol. 5.

[263] Trevor, Alexander J. B., Rogers, John G., and Christensen, Henrik I. 2012. Planar surface SLAM with 3D and 2D sensors. Pages 3041–3048 of: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[264] Triggs, Bill, McLauchlan, Philip F, Hartley, Richard I, and Fitzgibbon, Andrew W. 1999. Bundle adjustment—a modern synthesis. Pages 298–372 of: *International workshop on vision algorithms*. Springer.

[265] Trulls, E., Jin, Y., Yi, K.M., Mishkin, D., and Matas, J. 2022. *Image matching challenge*. https://www.kaggle.com/competitions/image-matching-challenge-2022. Accessed: 2022.

[266] Tsardoulias, Emmanouil G, Iliakopoulou, A, Kargakos, Andreas, and Petrou, Loukas. 2016. A review of global path planning methods for occupancy grid maps regardless of obstacle density. *J. of Intelligent and Robotic Systems*, **84**(1), 829–858.

[267] Tseng, Paul. 2001. Convergence of a block coordinate descent method for nondifferentiable minimization. *Journal of optimization theory and applications*, **109**, 475–494.

[268] Vasudevan, Shrihari, Ramos, Fabio, Nettleton, Eric, and Durrant-Whyte, Hugh. 2009. Gaussian process modeling of large-scale terrain. Pages 812–840 of: *J. of Field Robotics*, vol. 26.

[269] Vespa, Emanuele, Funk, Nils, Kelly, Paul HJ, and Leutenegger, Stefan. 2019. Adaptive-resolution octree-based volumetric SLAM. Pages 654–662 of: *Intl. Conf. on 3D Vision (3DV)*.

[270] Vizzo, I., Chen, X., Chebrolu, N., Behley, J., and Stachniss, C. 2021. Poisson Surface Reconstruction for LiDAR Odometry and Mapping. In: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[271] Vizzo, Ignacio, Guadagnino, Tiziano, Behley, Jens, and Stachniss, Cyrill. 2022. VDBFusion: Flexible and Efficient TSDF Integration of Range Sensor Data. *IEEE Sensors*, **22**(3).

[272] Vizzo, Ignacio, Guadagnino, Tiziano, Mersch, Benedikt, Wiesmann, Louis, Behley, Jens, and Stachniss, Cyrill. 2023. KISS-ICP: In Defense of Point-to-Point ICP – Simple, Accurate, and Robust Registration If Done the Right Way. *IEEE Robotics and Automation Letters*, **8**(2), 1029–1036.

[273] Wächter, Andreas, and Biegler, Lorenz T. 2006. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical programming*, **106**(1), 25–57.

[274] Wang, Chen, Gao, Dasong, Xu, Kuan, Geng, Junyi, Hu, Yaoyu, Qiu, Yuheng, Li, Bowen, Yang, Fan, Moon, Brady, Pandey, Abhinav, Aryan, Xu, Jiahe, Wu, Tianhao, He, Haonan, Huang, Daning, Ren, Zhongqiang, Zhao, Shibo, Fu, Taimeng, Reddy, Pranay, Lin, Xiao, Wang, Wenshan, Shi, Jingnan, Talak, Rajat, Cao, Kun, Du, Yi, Wang, Han, Yu, Huai, Wang, Shanzhao, Chen, Siyu, Kashyap, Ananth, Bandaru, Rohan, Dantu, Karthik, Wu, Jiajun, Xie, Lihua, Carlone, Luca, Hutter, Marco, and Scherer, Sebastian. 2023. PyPose: A Library for Robot Learning with Physics-based Optimization. In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[275] Wang, Chen, Ji, Kaiyi, Geng, Junyi, Ren, Zhongqiang, Fu, Taimeng, Yang, Fan, Guo, Yifan, He, Haonan, Chen, Xiangyu, Zhan, Zitong, Du, Qiwei, Su, Shaoshu, Li, Bowen, Qiu, Yuheng, Lin, Xiao, Du, Yi, Li, Qihang, and Zhao, Zhipeng. 2024. Imperative Learning: A Self-supervised Neural-Symbolic Learning Framework for Robot Autonomy. *arXiv preprint*.

[276] Wang, H., Wang, C., Chen, C., and Xie, L. 2021a. F-LOAM: Fast LiDAR Odometry and Mapping. In: *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*.

[277] Wang, Jinkun, and Englot, Brendan. 2016. Fast, accurate gaussian process occupancy maps via test-data octrees and nested Bayesian fusion. Pages 1003–1010 of: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[278] Wang, K., Gao, F., and Shen, S. 2019. Real-Time Scalable Dense Surfel Mapping. In: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[279] Wang, Sen, Clark, Ronald, Wen, Hongkai, and Trigoni, Niki. 2017. Deepvo: Towards end-to-end visual odometry with deep recurrent convolutional neural networks. Pages 2043–2050 of: *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE.

[280] Wang, Wenshan, Zhu, Delong, Wang, Xiangwei, Hu, Yaoyu, Qiu, Yuheng, Wang, Chen, Hu, Yafei Hu, Kapoor, Ashish, and Scherer, Sebastian. 2020. TartanAir: A Dataset to Push the Limits of Visual SLAM. Pages 4909–4916 of: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

[281] Wang, Wenshan, Hu, Yaoyu, and Scherer, Sebastian. 2021b. Tartanvo: A generalizable learning-based vo. Pages 1761–1772 of: *Conference on Robot Learning*. PMLR.

[282] Weber, Simon, Demmel, Nikolaus, Chan, Tin Chon, and Cremers, Daniel. 2023. Power Bundle Adjustment for Large-Scale 3D Reconstruction. In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[283] Wei, Peng, Hua, Guoliang, Huang, Weibo, Meng, Fanyang, and Liu, Hong.

2021. Unsupervised monocular visual-inertial odometry network. Pages 2347–2354 of: *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*.

[284] Wei, T., Patel, Y., Shekhovtsov, A., Matas, J., and Barath, D. 2023. Generalized differentiable RANSAC. In: *Intl. Conf. on Computer Vision (ICCV)*.

[285] Whelan, T., Leutenegger, S., Moreno, R. S., Glocker, B., and Davison, A. 2015. ElasticFusion: Dense SLAM Without A Pose Graph. In: *Robotics: Science and Systems (RSS)*.

[286] Williams, Christopher, and Seeger, Matthias. 2000. Using the Nyström method to speed up kernel machines. In: *Advances in Neural Information Processing Systems (NIPS)*, vol. 13.

[287] Williams, Oliver, and Fitzgibbon, Andrew. 2007. *Gaussian Process Implicit Surfaces*.

[288] Wu, Lan, Lee, Ki Myung Brian, Liu, Liyang, and Vidal-Calleja, Teresa. 2021. Faithful Euclidean distance field from log-Gaussian process implicit surfaces. *IEEE Robotics and Automation Letters*, 2461–2468.

[289] Wu, Lan, Lee, Ki Myung Brian, Le Gentil, Cedric, and Vidal-Calleja, Teresa. 2023. Log-GPIS-MOP: A Unified Representation for Mapping, Odometry, and Planning. *IEEE Trans. Robotics*, **39**(5), 4078–4094.

[290] Wu, Qinghua, and Hao, Jin-Kao. 2015. A review on algorithms for maximum clique problems. *European Journal of Operational Research*, **242**(3), 693–709.

[291] Yang, Fan, Wang, Chen, Cadena, Cesar, and Hutter, Marco. 2023. iPlanner: Imperative Path Planning. In: *Robotics: Science and Systems (RSS)*.

[292] Yang, H., and Carlone, L. 2019a. A Polynomial-time Solution for Robust Registration with Extreme Outlier Rates. In: *Robotics: Science and Systems (RSS)*. , , , , .

[293] Yang, H., and Carlone, L. 2019b. A Quaternion-based Certifiably Optimal Solution to the Wahba Problem with Outliers. In: *Intl. Conf. on Computer Vision (ICCV)*. (Oral Presentation, accept rate: 4%), Arxiv version: 1905.12536, .

[294] Yang, H., and Carlone, L. 2020. One Ring to Rule Them All: Certifiably Robust Geometric Perception with Outliers. Pages 18846–18859 of: *Advances in Neural Information Processing Systems (NIPS)*, vol. 33. .

[295] Yang, H., and Carlone, L. 2022. Certifiably Optimal Outlier-Robust Geometric Perception: Semidefinite Relaxations and Scalable Global Optimization. *IEEE Trans. Pattern Anal. Machine Intell.* .

[296] Yang, H., Antonante, P., Tzoumas, V., and Carlone, L. 2020a. Graduated Non-Convexity for Robust Spatial Perception: From Non-Minimal Solvers to Global Outlier Rejection. *IEEE Robotics and Automation Letters*, **5**(2), 1127–1134. arXiv preprint:1909.08605 (with supplemental material), .

[297] Yang, H., Shi, J., and Carlone, L. 2020b. TEASER: Fast and Certifiable Point Cloud Registration. *IEEE Trans. Robotics*, **37**(2), 314–333. extended arXiv version 2001.07715 .

[298] Yang, J., Li, H., Campbell, D., and Jia, Y. 2016. Go-ICP: A Globally Optimal Solution to 3D ICP Point-Set Registration. **38**(11), 2241–2254.

[299] Yang, Jiaolong, Li, Hongdong, and Jia, Yunde. 2014. Optimal essential matrix estimation via inlier-set maximization. Pages 111–126 of: *European Conf. on Computer Vision (ECCV)*. Springer.

[300] Yang, Wen, Gong, Zheng, Huang, Baifu, and Hong, Xiaoping. 2022. Lidar With Velocity: Correcting Moving Objects Point Cloud Distortion From Oscillating Scanning Lidars by Fusion With Camera. *IEEE Robotics and Automation Letters*, **7**(3).

[301] Yannakakis, M. 1981. Computing the minimum fill-in is NP-complete. *SIAM J. Algebraic Discrete Methods*, **2**.

[302] Yi, Brent, Lee, Michelle A, Kloss, Alina, Martín-Martín, Roberto, and Bohg, Jeannette. 2021a. Differentiable factor graph optimization for learning smoothers. Pages 1339–1345 of: *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE.

[303] Yi, Brent, Lee, Michelle, Kloss, Alina, Martín-Martín, Roberto, and Bohg, Jeannette. 2021b. Differentiable Factor Graph Optimization for Learning Smoothers. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

[304] Yokoyama, N., Ha, S., Batra, D., Wang, J., and Bucher, B. 2024. VLFM: Vision-Language Frontier Maps for Zero-Shot Semantic Navigation. In: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[305] Yugay, Vladimir, Li, Yue, Gevers, Theo, and Oswald, Martin R. 2023. Gaussian-SLAM: Photo-realistic Dense SLAM with Gaussian Splatting. *arXiv preprint*.

[306] Zhan, Zitong, Gao, Dasong, Lin, Yun-Jou, Xia, Youjie, and Wang, Chen. 2024. iMatching: Imperative Correspondence Learning. In: *European Conference on Computer Vision (ECCV)*.

[307] Zhang, J., and Singh, S. 2014. LOAM: Lidar Odometry and Mapping in Real-time. In: *Robotics: Science and Systems (RSS)*.

[308] Zhao, Shibo, Wang, Peng, Zhang, Hengrui, Fang, Zheng, and Scherer, Sebastian. 2020. Tp-tio: A robust thermal-inertial odometry with deep thermalpoint. Pages 4505–4512 of: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE.

[309] Zheng, Y., Sugimoto, S., and Okutomi, M. 2011. Deterministically maximizing feasible subsystem for robust model fitting with unit norm constraint. Pages 1825–1832 of: *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*.

[310] Zhong, Xingguang, Pan, Yue, Behley, Jens, and Stachniss, Cyrill. 2023. SHINE-Mapping: Large-Scale 3D Mapping Using Sparse Hierarchical Implicit Neural Representations. In: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

[311] Zhou, Q.Y., Park, J., and Koltun, V. 2016. Fast global registration. Pages 766–782 of: *European Conf. on Computer Vision (ECCV)*. Springer.

[312] Zhu, Zihan, Peng, Songyou, Larsson, Viktor, Xu, Weiwei, Bao, Hujun, Cui, Zhaopeng, Oswald, Martin R, and Pollefeys, Marc. 2022. NICE-SLAM: Neural Implicit Scalable Encoding for SLAM. In: *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*.

[313] Zuckerman, David. 2006 (May). Linear degree extractors and the inapproximability of max clique and chromatic number. Pages 681–690 of: *ACM Symp. on Theory of Computing (STOC)*.

[314] Zwicker, Matthias, Pfister, Hanspeter, van Baar, Jeroen, and Gross, Markus. 2001. Surface Splatting. In: *Intl. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*.

Author index

Subject index