4

Differentiable Optimization

Chen Wang, Krishna Murthy Jatavallabhula, and Mustafa Mukadam

4.1 Introduction

As presented in Chapter 1, 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 [976, 275, 1262] and front-end motion estimation [1165, 1086]. 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 [1084], motion planning [88, 1212], SLAM [519, 1086], bundle adjustment [1072, 1262], state estimation [1239, 188], and image alignment [724].

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 4.1.1 restates the non-linear least square (NLS) problem. Section 4.2 describes how to differentiate through the NLS problem. Section 4.3 shows how to differentiate problems defined on manifold. Section 4.4 discusses numerical challenges of the above differentiation and introduces related machine learning libraries. Finally, Section 4.5 provides examples of differentiable optimization in contemporary SLAM systems.

4.1.1 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 1, 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:

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}} \mathcal{L}(\boldsymbol{x}) = \arg\min_{\boldsymbol{x}} = \frac{1}{2} \sum_i || \underbrace{w_i \boldsymbol{c}_i(\boldsymbol{x}_i)}_{\boldsymbol{r}_i(\boldsymbol{x}_i)} ||^2, \tag{4.1}$$

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 1, 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

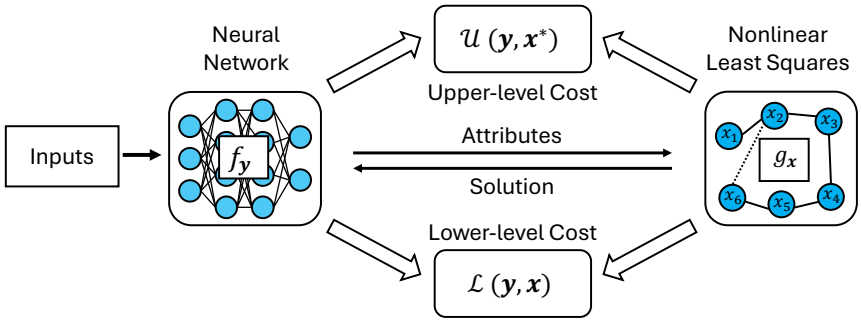

Figure 4.1 A modern SLAM system often involves both neural networks and nonlinear least squares. To eliminate compound errors introduced by optimizing the two modules separately, we can optimize the system in an end-to-end manner by formulating the entire system as a bilevel optimization, which involves an upper-level cost and a lower-level cost.

commented in Chapter 2 that the addition in the update step is more generally a 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 1.

4.2 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 4.1. 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:

$$\boldsymbol{y}^* = \arg\min_{\boldsymbol{y} \in \Theta} \ \mathcal{U}(\boldsymbol{y}, \boldsymbol{x}^*), \tag{4.2a}$$

$$\text{s.t.} \quad \boldsymbol{x}^* = \arg\min_{\boldsymbol{x} \in \Psi} \ \mathcal{L}(\boldsymbol{y}, \boldsymbol{x}), \tag{4.2b}$$

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 4.1 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 (4.2a) can be feature matching error for optimizing the network, while the LL cost L (4.2b) 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 (4.2) allows us to further reduce the matching error via back-propagating the BA reprojection errors [1262].

Example 4.2 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 [354].

BLO is a long-standing and well-researched problem [1148, 524, 686]. 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

$$\nabla_{\boldsymbol{y}}\mathcal{U} = \frac{\partial \mathcal{U}(\boldsymbol{y}, \boldsymbol{x}^*)}{\partial \boldsymbol{y}} + \frac{\partial \mathcal{U}(\boldsymbol{y}, \boldsymbol{x}^*)}{\partial \boldsymbol{x}^*} \frac{\partial \boldsymbol{x}^*(\boldsymbol{y})}{\partial \boldsymbol{y}}, \tag{4.3}$$

where the term $\frac{\partial \boldsymbol{x}^*(\boldsymbol{y})}{\partial \boldsymbol{y}}$ involves indirect gradient computation. Since other direct gradient terms in (4.3) are easy to obtain, the challenge of solving a BLO (4.2) 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 [686, 1148] and here we list a generic framework incorporating both methods in Algorithm 1. We next explain the unrolled differentiation and implicit differentiation, respectively.

4.2.1 Unrolled Differentiation

Unrolled Differentiation needs automatic differentiation (AutoDiff) through the LL optimization to solve a BLO problem. Specifically, given an initialization $\boldsymbol{x}_0 =$

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

Initialization: $\boldsymbol{y}_0$, $\boldsymbol{x}_0$.
while Not Convergent ($\|\boldsymbol{y}_{k+1} - \boldsymbol{y}_k\|$ is large enough) **do**
  Obtain $\boldsymbol{x}_T$ by solving (4.2b) by a generic optimizer $\mathcal{O}$ with T steps.
  Efficient estimation of upper-level gradients in (4.3) 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 (4.7).
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).

  Compute $\boldsymbol{y}_{k+1}$ via gradients using $\hat{\nabla}_{\boldsymbol{y}_k} U$.
end while

$\Phi_0(\boldsymbol{y})$ at step $t = 0$, the iterative process of unrolled LL optimization is

$$\boldsymbol{x}_t = \Phi_t(\boldsymbol{x}_{t-1}; \boldsymbol{y}), \quad t = 1, \cdots, T, \tag{4.4}$$

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:

$$\Phi_t(\boldsymbol{x}_{t-1}; \boldsymbol{y}) = \boldsymbol{x}_{t-1} - \eta_t \cdot \frac{\partial \mathcal{L}(\boldsymbol{x}_{t-1}, \boldsymbol{y})}{\partial \boldsymbol{x}_{t-1}}, \tag{4.5}$$

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

$$\boldsymbol{x}^* \approx \boldsymbol{x}_T = \Phi(\boldsymbol{y}) = \left(\Phi_T \circ \cdots \circ \Phi_1 \circ \Phi_0\right)(\boldsymbol{y}), \tag{4.6}$$

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 (4.2):

$$\min_{\boldsymbol{y} \in \Theta} \ \mathcal{U}(\boldsymbol{y}, \Phi(\boldsymbol{y})), \tag{4.7}$$

which needs to compute $\frac{\partial \Phi(\boldsymbol{y})}{\partial \boldsymbol{y}}$ via AutoDiff instead of calculating (4.3). 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 [864], and the other corresponds to the forward-mode way [934]. We omit the details of the two approaches of AutoDiff and refer the readers to the AutoDiff libraries such as

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

PyTorch [856] for deep learning and PyPose [1147] and Theseus [880] for SLAM. A review of these approaches can also be found in Liu et al. [686].

4.2.2 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 (4.5) 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. [997] 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 (4.4) to remove the recursive structure [683], *i.e.,*

$$\nabla_{\boldsymbol{y}} \mathcal{U} = \frac{\partial \mathcal{U}(\boldsymbol{y}, \boldsymbol{x}_1(\boldsymbol{y}))}{\partial \boldsymbol{y}} + \frac{\partial \mathcal{U}(\boldsymbol{y}, \boldsymbol{x}_1(\boldsymbol{y}))}{\partial \boldsymbol{x}_1} \frac{\partial \boldsymbol{x}_1(\boldsymbol{y})}{\partial \boldsymbol{y}}, \tag{4.8}$$

where the term $\frac{\partial \boldsymbol{x}_1(\boldsymbol{y})}{\partial \boldsymbol{y}}$ is a Hessian that can be calculated from (4.5) as

$$\frac{\partial \boldsymbol{x}_1(\boldsymbol{y})}{\partial \boldsymbol{y}} = -\frac{\partial^2 \mathcal{L}(\boldsymbol{x}_0, \boldsymbol{y})}{\partial \boldsymbol{x}_0 \partial \boldsymbol{y}}. \tag{4.9}$$

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 (4.8) as a whole:

$$\frac{\partial \mathcal{U}(\boldsymbol{y}, \boldsymbol{x}_1(\boldsymbol{y}))}{\partial \boldsymbol{x}_1} \frac{\partial \boldsymbol{x}_1(\boldsymbol{y})}{\partial \boldsymbol{y}} \approx \frac{\frac{\partial \mathcal{L}(\boldsymbol{x}_0^+, \boldsymbol{y})}{\partial \boldsymbol{y}} - \frac{\mathcal{L}(\boldsymbol{x}_0^-, \boldsymbol{y})}{\partial \boldsymbol{y}}}{2\epsilon}, \tag{4.10}$$

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 [1147], which will be introduced in Section 4.4.

4.2.3 Implicit Differentiation

It is intuitive that the term $\frac{\partial \boldsymbol{x}^*(\boldsymbol{y})}{\partial \boldsymbol{y}}$ in (4.3) is dependent on the LL cost (4.2b), thus *implicit differentiation* can be used to derive a solution to the gradient.

Example 4.3 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

$$\frac{\partial^2 \mathcal{L}(\boldsymbol{x}^*(\boldsymbol{y}), \boldsymbol{y})}{\partial \boldsymbol{x}^*(\boldsymbol{y}) \partial \boldsymbol{y}} + \frac{\partial^2 \mathcal{L}(\boldsymbol{x}^*(\boldsymbol{y}), \boldsymbol{y})}{\partial \boldsymbol{x}^*(\boldsymbol{y}) \partial \boldsymbol{x}^*(\boldsymbol{y})} \cdot \frac{\partial \boldsymbol{x}^*(\boldsymbol{y})}{\partial \boldsymbol{y}} = 0. \tag{4.11}$$

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

$$\frac{\partial \boldsymbol{x}^*(\boldsymbol{y})}{\partial \boldsymbol{y}} = -\left(\frac{\partial^2 \mathcal{L}(\boldsymbol{x}^*(\boldsymbol{y}), \boldsymbol{y})}{\partial \boldsymbol{x}^*(\boldsymbol{y}) \partial \boldsymbol{x}^*(\boldsymbol{y})}\right)^{-1} \frac{\partial^2 \mathcal{L}(\boldsymbol{x}^*(\boldsymbol{y}), \boldsymbol{y})}{\partial \boldsymbol{x}^*(\boldsymbol{y}) \partial \boldsymbol{y}}, \tag{4.12}$$

The strength of (4.12) 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 4.4 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 (4.3), substituting (4.12) into (4.3) gives us:

$$\begin{aligned} \nabla_{\boldsymbol{y}} \mathcal{U} &= \frac{\partial \mathcal{U}(\boldsymbol{y}, \boldsymbol{x}^*)}{\partial \boldsymbol{y}} - \underbrace{\frac{\partial \mathcal{U}(\boldsymbol{y}, \boldsymbol{x}^*)}{\partial \boldsymbol{x}^*}}_{\boldsymbol{v}^\top} \underbrace{\left(\frac{\partial^2 \mathcal{L}(\boldsymbol{x}^*(\boldsymbol{y}), \boldsymbol{y})}{\partial \boldsymbol{x}^*(\boldsymbol{y}) \partial \boldsymbol{x}^*(\boldsymbol{y})}\right)^{-1}}_{(\boldsymbol{H}^\top)^{-1}} \frac{\partial^2 \mathcal{L}(\boldsymbol{x}^*(\boldsymbol{y}), \boldsymbol{y})}{\partial \boldsymbol{x}^*(\boldsymbol{y}) \partial \boldsymbol{y}} \\ &= \frac{\partial \mathcal{U}(\boldsymbol{y}, \boldsymbol{x}^*)}{\partial \boldsymbol{y}} - \boldsymbol{q}^\top \cdot \frac{\partial^2 \mathcal{L}(\boldsymbol{x}^*(\boldsymbol{y}), \boldsymbol{y})}{\partial \boldsymbol{x}^*(\boldsymbol{y}) \partial \boldsymbol{y}} \end{aligned}. \tag{4.13}$$

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

$$\boldsymbol{q}^* = \min \arg_{\boldsymbol{q}} Q(\boldsymbol{q}) = \min \arg_{\boldsymbol{q}} \frac{1}{2}\boldsymbol{q}^\mathsf{T}\boldsymbol{H}\boldsymbol{q} - \boldsymbol{q}^\mathsf{T}\boldsymbol{v}, \tag{4.14}$$

using efficient linear solvers such as a simple gradient descent or conjugate gradient method [467]. 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:

$$\boldsymbol{H}\boldsymbol{q} = \frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{x} \partial \boldsymbol{x}} \cdot \boldsymbol{q} = \frac{\partial \left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}} \cdot \boldsymbol{q} \right)}{\partial \boldsymbol{x}}, \tag{4.15}$$

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:

$$\boldsymbol{q}_k = \boldsymbol{q}_{k-1} - \eta \left(\boldsymbol{H}\boldsymbol{q}_{k-1} - \boldsymbol{v} \right), \tag{4.16}$$

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 (4.3) as:

$$\nabla_{\boldsymbol{y}} \mathcal{U} = \frac{\partial \mathcal{U}(\boldsymbol{y}, \boldsymbol{x}^*)}{\partial \boldsymbol{y}} - \underbrace{\left(\frac{\partial^2 \mathcal{L}(\boldsymbol{x}^*(\boldsymbol{y}), \boldsymbol{y})}{\partial \boldsymbol{y} \partial \boldsymbol{x}^*(\boldsymbol{y})} \cdot \boldsymbol{q} \right)^\mathsf{T}}_{(\boldsymbol{H}_{\boldsymbol{y}\boldsymbol{x}} \cdot \boldsymbol{q})^\mathsf{T}}, \tag{4.17}$$

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

$$\epsilon \sim \left| \frac{\partial U}{\partial \boldsymbol{x}^*} \frac{\partial \boldsymbol{x}^*}{\partial \boldsymbol{y}} \right|. \tag{4.18}$$

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

4.3 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.

4.3.1 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 2, 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:

$$\boldsymbol{\chi} = \exp(\boldsymbol{\tau}^\wedge) \Leftrightarrow \boldsymbol{\tau}^\wedge = \log(\boldsymbol{\chi}), \tag{4.19}$$

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 $\boldsymbol{\chi}$:

$$\boldsymbol{\chi} = \operatorname{Exp}(\boldsymbol{\tau}) \Leftrightarrow \boldsymbol{\tau} = \operatorname{Log}(\boldsymbol{\chi}), \tag{4.20}$$

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 (4.21) and (4.22) below:

$$\begin{aligned} \boldsymbol{\chi}_2 &= \boldsymbol{\chi}_1 \oplus \boldsymbol{\tau} \triangleq \boldsymbol{\chi}_1 \circ \operatorname{Exp}(\boldsymbol{\tau}), \\ \boldsymbol{\tau} &= \boldsymbol{\chi}_2 \ominus \boldsymbol{\chi}_1 \triangleq \operatorname{Log}\left(\boldsymbol{\chi}_1^{-1} \circ \boldsymbol{\chi}_2\right). \end{aligned} \tag{4.21}$$

The placement of $\boldsymbol{\tau}$ on the right-hand side in (4.21) signifies that it is expressed in the local frame at $\boldsymbol{\chi}_1$. Conversely, the left operators in (4.22) reflect a global frame perspective:

$$\begin{aligned} \boldsymbol{\chi}_2 &= \boldsymbol{\varepsilon} \oplus \boldsymbol{\chi}_1 \triangleq \operatorname{Exp}(\boldsymbol{\varepsilon}) \circ \boldsymbol{\chi}_1, \\ \boldsymbol{\varepsilon} &= \boldsymbol{\chi}_2 \ominus \boldsymbol{\chi}_1 \triangleq \operatorname{Log}\left(\boldsymbol{\chi}_2 \circ \boldsymbol{\chi}_1^{-1}\right), \end{aligned} \tag{4.22}$$

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 $\mathfrak{m}$:

$$\begin{aligned}\frac{\partial f(\boldsymbol{\chi})}{\partial \boldsymbol{\chi}} &\triangleq \lim_{\boldsymbol{\tau}\to 0} \frac{f(\boldsymbol{\chi}\oplus\boldsymbol{\tau})\ominus f(\boldsymbol{\chi})}{\boldsymbol{\tau}} \\ &= \lim_{\boldsymbol{\tau}\to 0} \frac{f(\boldsymbol{\chi}\circ \operatorname{Exp}(\boldsymbol{\tau}))\ominus f(\boldsymbol{\chi})}{\boldsymbol{\tau}} \\ &= \lim_{\boldsymbol{\tau}\to 0} \frac{\operatorname{Log}\left(f(\boldsymbol{\chi})^{-1}\circ f(\boldsymbol{\chi}\circ \operatorname{Exp}(\boldsymbol{\tau}))\right)}{\boldsymbol{\tau}}.\end{aligned} \tag{4.23}$$

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 $\boldsymbol{\tau}=0$:

$$\frac{\partial f(\boldsymbol{\chi})}{\partial \boldsymbol{\chi}} = \boldsymbol{J}_R = \left.\frac{\partial g(\boldsymbol{\tau})}{\partial \boldsymbol{\tau}}\right|_{\boldsymbol{\tau}=0}. \tag{4.24}$$

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:

$$\begin{aligned}\frac{\partial f(\boldsymbol{\chi})}{\partial \boldsymbol{\chi}} &\triangleq \lim_{\boldsymbol{\varepsilon}\to 0} \frac{f(\boldsymbol{\varepsilon}\oplus\boldsymbol{\chi})\ominus f(\boldsymbol{\chi})}{\boldsymbol{\varepsilon}} \\ &= \lim_{\boldsymbol{\varepsilon}\to 0} \frac{f(\operatorname{Exp}(\boldsymbol{\varepsilon})\circ\boldsymbol{\chi})\ominus f(\boldsymbol{\chi})}{\boldsymbol{\varepsilon}} \\ &= \lim_{\boldsymbol{\varepsilon}\to 0} \frac{\operatorname{Log}\left(f(\operatorname{Exp}(\boldsymbol{\varepsilon})\circ\boldsymbol{\chi})\circ f(\boldsymbol{\chi})^{-1}\right)}{\boldsymbol{\varepsilon}} \\ &= \left.\frac{\partial \operatorname{Log}\left(f(\operatorname{Exp}(\boldsymbol{\varepsilon})\circ\boldsymbol{\chi})\circ f(\boldsymbol{\chi})^{-1}\right)}{\partial \boldsymbol{\varepsilon}}\right|_{\boldsymbol{\varepsilon}=0}.\end{aligned} \tag{4.25}$$

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:

$$\boldsymbol{\Sigma}_{\boldsymbol{\chi}} \triangleq \mathbb{E}[\boldsymbol{\tau}\boldsymbol{\tau}^\top] = \mathbb{E}[(\boldsymbol{\chi}\ominus\boldsymbol{\chi}_1)(\boldsymbol{\chi}\ominus\boldsymbol{\chi}_1)^\top]. \tag{4.26}$$

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 4.5 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 $SO(3)$:

$$\hat{\boldsymbol{R}} = \arg\min_{\boldsymbol{R} \in SO(3)} f(\boldsymbol{R}, \boldsymbol{R}_{\text{IMU}}, \boldsymbol{R}_{\text{Cam}}), \tag{4.27}$$

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:

$$f(\boldsymbol{R}) = \|\text{Log}\left(\boldsymbol{R}_{\text{IMU}}^{-1}\boldsymbol{R}\right)\|^2 + \|\text{Log}\left(\boldsymbol{R}_{\text{Cam}}^{-1}\boldsymbol{R}\right)\|^2. \tag{4.28}$$

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:

$$\begin{aligned} \nabla f(\boldsymbol{R}) =& 2\left(\frac{\partial \text{Log}\left(\boldsymbol{R}_{\text{IMU}}^{-1}\boldsymbol{R}\right)}{\partial \boldsymbol{R}}\right)^{\top} \text{Log}\left(\boldsymbol{R}_{\text{IMU}}^{-1}\boldsymbol{R}\right) \\ &+ 2\left(\frac{\partial \text{Log}\left(\boldsymbol{R}_{\text{Cam}}^{-1}\boldsymbol{R}\right)}{\partial \boldsymbol{R}}\right)^{\top} \text{Log}\left(\boldsymbol{R}_{\text{Cam}}^{-1}\boldsymbol{R}\right). \end{aligned} \tag{4.29}$$

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:

$$\boldsymbol{R}_{k+1} = \boldsymbol{R}_k \text{Exp}\left(-\alpha \nabla f(\boldsymbol{R})\right), \tag{4.30}$$

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.

4.3.2 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:

$$\frac{\partial \mathcal{Z}}{\partial \boldsymbol{\chi}} = \frac{\partial \mathcal{Z}}{\partial \mathcal{Y}} \frac{\partial \mathcal{Y}}{\partial \boldsymbol{\chi}}, \tag{4.31}$$

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 (4.23) for the right Jacobian $\boldsymbol{J}_R$, which leads to:

$$\begin{aligned} \frac{\partial \boldsymbol{\chi}^{-1}}{\partial \boldsymbol{\chi}} &\triangleq \lim_{\boldsymbol{\tau} \to 0} \frac{\operatorname{Log}\left((\boldsymbol{\chi}^{-1})^{-1} (\boldsymbol{\chi} \operatorname{Exp}(\boldsymbol{\tau}))^{-1}\right)}{\boldsymbol{\tau}} \\ &= \lim_{\boldsymbol{\tau} \to 0} \frac{\operatorname{Log}\left(\boldsymbol{\chi} \operatorname{Exp}(\boldsymbol{\tau})^{-1} \boldsymbol{\chi}^{-1}\right)}{\boldsymbol{\tau}} \\ &= \lim_{\boldsymbol{\tau} \to 0} \frac{(\boldsymbol{\chi}(-\boldsymbol{\tau})^{\wedge} \boldsymbol{\chi}^{-1})^{\vee}}{\boldsymbol{\tau}}. \end{aligned} \tag{4.32}$$

Jacobians of composition can be derived through the application of the function $f(\boldsymbol{\chi}) = \boldsymbol{\chi} \circ \boldsymbol{\chi}_1$ with the Equation (4.23). The derivative of the composition operator $\boldsymbol{\chi} \circ \boldsymbol{\chi}_1$ with respect to $\boldsymbol{\chi}$ is:

$$\begin{aligned} \frac{\partial (\boldsymbol{\chi} \circ \boldsymbol{\chi}_1)}{\partial \boldsymbol{\chi}} &\triangleq \lim_{\boldsymbol{\tau} \to 0} \frac{\operatorname{Log}\left((\boldsymbol{\chi}\boldsymbol{\chi}_1)^{-1} (\boldsymbol{\chi} \operatorname{Exp}(\boldsymbol{\tau}) \boldsymbol{\chi}_1)\right)}{\boldsymbol{\tau}} \\ &= \lim_{\boldsymbol{\tau} \to 0} \frac{\operatorname{Log}\left(\boldsymbol{\chi}_1^{-1} \operatorname{Exp}(\boldsymbol{\tau}) \boldsymbol{\chi}_1\right)}{\boldsymbol{\tau}} \\ &= \lim_{\boldsymbol{\tau} \to 0} \frac{(\boldsymbol{\chi}_1^{-1} \boldsymbol{\tau}^{\wedge} \boldsymbol{\chi}_1)^{\vee}}{\boldsymbol{\tau}}. \end{aligned} \tag{4.33}$$

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

$$\begin{aligned} \frac{\partial (\boldsymbol{\chi} \circ \boldsymbol{\chi}_1)}{\partial \boldsymbol{\chi}_1} &\triangleq \lim_{\boldsymbol{\tau} \to 0} \frac{\operatorname{Log}\left((\boldsymbol{\chi}\boldsymbol{\chi}_1)^{-1} (\boldsymbol{\chi}\boldsymbol{\chi}_1 \operatorname{Exp}(\boldsymbol{\tau}))\right)}{\boldsymbol{\tau}} \\ &= \lim_{\boldsymbol{\tau} \to 0} \frac{\operatorname{Log}(\operatorname{Exp}(\boldsymbol{\tau}))}{\boldsymbol{\tau}} \\ &= \boldsymbol{I}. \end{aligned} \tag{4.34}$$

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:

$$\boldsymbol{J}_r(\boldsymbol{\tau}) \triangleq \frac{^{\boldsymbol{\tau}}\partial \operatorname{Exp}(\boldsymbol{\tau})}{\partial \boldsymbol{\tau}}. \tag{4.35}$$

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:

$$\boldsymbol{J}_l(\boldsymbol{\tau}) \triangleq \frac{^{\boldsymbol{\epsilon}}\partial \operatorname{Exp}(\boldsymbol{\tau})}{\partial \boldsymbol{\tau}}. \tag{4.36}$$

Jacobians of group action depends on the specific group action set $v \in \mathcal{V}$. The group action is defined as:

$$\begin{aligned} \boldsymbol{J}_{\boldsymbol{\chi}}^{\boldsymbol{\chi} \cdot v} &\triangleq \frac{{}^{\boldsymbol{\chi}} D \boldsymbol{\chi} \cdot v}{D \boldsymbol{\chi}}, \\ \boldsymbol{J}_{v}^{\boldsymbol{\chi} \cdot v} &\triangleq \frac{{}^{v} D \boldsymbol{\chi} \cdot v}{D v}, \end{aligned} \tag{4.37}$$

where $\boldsymbol{\chi} \in \mathcal{M}$ and $v \in \mathcal{V}$.

Example 4.6 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:

$$\boldsymbol{R} = \boldsymbol{R}_1 \circ \boldsymbol{R}_2. \tag{4.38}$$

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:

$$\begin{aligned} \frac{\partial (\boldsymbol{R}_1 \circ \boldsymbol{R}_2)}{\partial \boldsymbol{R}_1} &= \lim_{\boldsymbol{\tau} \to 0} \frac{(\boldsymbol{R}_2^{-1} \boldsymbol{\tau}^{\wedge} \boldsymbol{R}_2)^{\vee}}{\boldsymbol{\tau}}, \\ \frac{\partial (\boldsymbol{R}_1 \circ \boldsymbol{R}_2)}{\partial \boldsymbol{R}_2} &= \boldsymbol{I}. \end{aligned} \tag{4.39}$$

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 $\boldsymbol{R}_1$.

4.4 Modern Libraries

4.4.1 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 [1147] 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:

$$\texttt{Exp}(\boldsymbol{\nu}) = \left[\sin\left(\frac{\|\boldsymbol{\nu}\|}{2}\right)\frac{\boldsymbol{\nu}^\top}{\|\boldsymbol{\nu}\|},\ \cos\left(\frac{\|\boldsymbol{x}\|}{2}\right)\right]^\top \tag{4.40}$$

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 (4.40) for the Exp and Log mapping [1087]. 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.

$$\texttt{Exp}(\boldsymbol{\nu}) = \begin{cases} \left[\boldsymbol{\nu}^T\gamma_e,\ \cos(\frac{\|\boldsymbol{\nu}\|}{2})\right]^T & \text{if } \|\boldsymbol{\nu}\| > \text{eps} \\ \left[\boldsymbol{\nu}^T\gamma_o,\ 1-\frac{\|\boldsymbol{\nu}\|^2}{8}+\frac{\|\boldsymbol{\nu}\|^4}{384}\right]^T & \text{otherwise,} \end{cases} \tag{4.41}$$

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 [1087] 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 [1239], 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`.

4.4.2 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 [936] and Adam [587] 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 [58]. Moreover, practical problems have outliers, hence one needs to robustify the loss as described in Chapter 3. Next we consider an Iteratively Reweighted Least Squares (IRLS) approach to SLAM as introduced in Section 3.3, 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:

$$\min_{\boldsymbol{y}} \sum_i \left(\boldsymbol{h}_i(\boldsymbol{x}_i) - \boldsymbol{z}_i\right)^T \boldsymbol{\Sigma}_i \left(\boldsymbol{h}_i(\boldsymbol{x}_i) - \boldsymbol{z}_i\right), \tag{4.42}$$

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 (4.42) 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:

$$\sum_i \left(\boldsymbol{\Lambda}_i + \lambda \cdot \operatorname{diag}(\boldsymbol{\Lambda}_i)\right) \boldsymbol{\delta}_t = -\sum_i \boldsymbol{J}_i^T \boldsymbol{\Sigma}_i \mathbf{r}_i, \tag{4.43}$$

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`:

$$\mathbf{A} \cdot \boldsymbol{\delta}_t = \boldsymbol{\beta}, \tag{4.44}$$

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 [167] or preconditioned conjugate gradient (PCG) [467] solver. In practice, one often introduces robust `kernel` functions $\rho : \mathbb{R} \mapsto \mathbb{R}$ into (4.42) to

reduce the effect of outliers:

$$\min_{\boldsymbol{y}} \sum_i \rho\left(\mathbf{r}_i^T \boldsymbol{\Sigma}_i \mathbf{r}_i\right), \tag{4.45}$$

where ρ is designed to down-weigh measurements with large residuals $\mathbf{r}_i$. In this case, we need to adjust (4.43) to account for the presence of the robust kernel. A popular way is to use an IRLS method, Triggs' correction [1111], which is also adopted by the Ceres [22] 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 [1111]. Alternatively, PyPose introduces an IRLS method, `FastTriggs`, which is faster yet more stable than `Triggs` by only involving the 1^{st}-order derivative:

$$\mathbf{r}_i^\rho = \sqrt{\rho'(c_i)}\mathbf{r}_i, \quad \boldsymbol{J}_i^\rho = \sqrt{\rho'(c_i)}\boldsymbol{J}_i, \tag{4.46}$$

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 3.3.

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 [704] 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`.

4.4.3 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 [829], 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 [420], a high performance C++ linear algebra library, has been used in many projects such as TensorFlow [9], Ceres [22], GTSAM [260], and g^{2}o [410]. ArrayFire [734], 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 [236], OpenNN [836], and MATLAB [745], 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 [1105], Theano [31], and Caffe [525] 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 [9], PyTorch [856], and MXNet [190], 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 [437] introduced differentiable transformation layers into Torch-based framework, leading to end-to-end geometric learning. JAX [116] 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 [880] and CvxpyLayer [25] showed how to embed differentiable optimization within deep neural networks. PyPose [1147] 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 [22] is an open-source C++ library for large-scale nonlinear least squares optimization problems and has been widely used in SLAM. Pyomo [441] and JuMP [300] 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 [39] 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 [410] and GTSAM [260] 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 [1143] 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 [598] 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 [391], which provides standard interfaces for different optimal control solvers and can be

extended to a broad class of dynamical systems in robotic applications. Drake [1083] 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.

4.5 Final Considerations & Recent Trends

Deep learning methods have witnessed significant development in recent years [1281]. 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 [1161] and TartanVO [1165] and unsupervised methods such as UnDeepVO [663] and Unsupervised VIO [1171]. 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 [1164].

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 [1072] and DROID-SLAM [1086]. Additionally, some works focused on compressing image features into codes (embedded features) and optimizing the pose-code graph during inference such as DeepFactors [244]. Furthermore, DiffPoseNet [848] 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 [354] 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 [1033], local planning [1212], global planning [193], feature matching [1262], and multi-robot routing [426].