Theseus: A Library for Differentiable Nonlinear Optimization

Luis Pineda[1], **Taosha Fan**[1], **Maurizio Monge**[2], **Shobha Venkataraman**[1], **Paloma Sodhi**[1], **Ricky T. Q. Chen**[1], **Joseph Ortiz**[1], **Daniel DeTone**[2], **Austin Wang**[1], **Stuart Anderson**[1], **Jing Dong**[2], **Brandon Amos**[1], **Mustafa Mukadam**[1]

[1]Meta AI, [2]Reality Labs Research

Abstract

We present Theseus, an efficient application-agnostic open source library for differentiable nonlinear least squares (DNLS) optimization built on PyTorch, providing a common framework for end-to-end structured learning in robotics and vision. Existing DNLS implementations are application specific and do not always incorporate many ingredients important for efficiency. Theseus is application-agnostic, as we illustrate with several example applications that are built using the same underlying differentiable components, such as second-order optimizers, standard costs functions, and Lie groups. For efficiency, Theseus incorporates support for sparse solvers, automatic vectorization, batching, GPU acceleration, and gradient computation with implicit differentiation and direct loss minimization. We do extensive performance evaluation in a set of applications, demonstrating significant efficiency gains and better scalability when these features are incorporated. Project page: https://sites.google.com/view/theseus-ai/

1 Introduction

Reconciling traditional approaches with deep learning to leverage their complementary strengths is a common thread in a large body of recent work in robotics. In particular, an emerging trend is to differentiate through nonlinear least squares (NLS) [1] which is a second-order optimization formulation at the heart of many problems in robotics [2–7] and vision [8–13]. Optimization layers as inductive priors in neural models have been explored in machine learning with convex optimization [14, 15] and in meta learning with gradient descent [16, 17] based first-order optimization.

Differentiable nonlinear least squares (DNLS) provides a general scheme to encode inductive priors, as the objective function can be partly parameterized by neural models and partly with engineered domain-specific differentiable models. Here, as illustrated in Fig. 1, input tensors define a sum of weighted squares objective function and output tensors are minima of that objective. Such implicit layers [18] are in contrast to typical (explicit) layers that take input tensors through a linear transformation and some element-wise nonlinear activation function.

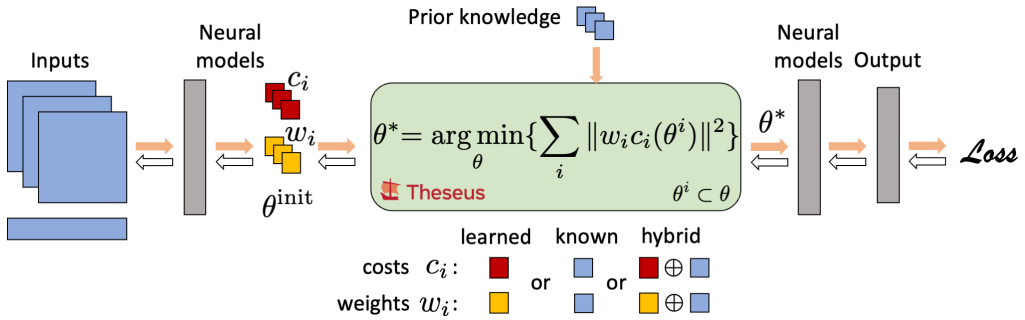


Figure 1: Theseus enables building custom, efficient DNLS layers that support end-to-end structured learning.

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

The ability to compute gradients end-to-end is retained by differentiating through the optimizer which allows neural models to train on the final task loss, while also taking advantage of priors captured by the optimizer. 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 [19], motion planning [20], SLAM [21, 22], bundle adjustment [23], state estimation [24, 25], image alignment [26] with other applications like manipulation and tactile sensing [27, 28], control [29], human pose tracking [30, 31] to be explored. However, existing implementations from above are application specific, common underlying tools like optimizers get reimplemented, and features like sparse solvers, batching, and GPU support that impact efficiency are not always included. This has led to a fragmented literature where it is difficult to start work on new ideas or to build on the progress of prior work.

To address this gap, we present `Theseus`, an open source library for differentiable nonlinear least squares optimization built on `PyTorch`. `Theseus` provides an efficient application-agnostic interface that consolidates recent efforts and catalyzes future progress in the domain of structured end-to-end learning for robotics and vision. Our contributions are summarized below.

Application agnostic interface. Our implementation provides an easy to use interface to build custom optimization layers and plug them into any neural architecture. (i) The layer can be constructed from a set of available second-order optimizers like Gauss-Newton and Levenberg–Marquardt and a nonlinear least squares objective. (ii) The objective can be constructed with learnable or hand-specified cost functions, either by applying one of many common costs already provided in the library, or by building custom costs in-place with support for automatic differentiation through `PyTorch` [32]. (iii) We also provide differentiable Lie groups for representing 2D/3D positions and rotations [33], and differentiable kinematics wrapping over an existing library [34] for representing robot models. More details are described in Sec. 3.

Efficiency based design. Efficiency is a central design consideration and we make several advancements in improving computation times and memory consumption. (i) As common in prior work, an optimizer implementation using `PyTorch`'s native linear solver would use a dense representation for solving the linear system within the nonlinear optimization. In practice, these optimization problems often have a considerable amount of sparsity that can be exploited [35–38]. In `Theseus`, we implement sparse linear solvers that are differentiable end-to-end and make them efficient with custom CPU and CUDA backends to support batching and GPU acceleration. (ii) Beyond sparse solvers, we extend batching and GPU support to all features in the library and add automatic vectorization of cost functions and other operations to significantly boost efficiency. (iii) Finally, we introduce implicit differentiation [39] and direct loss minimization [40, 41], which have been previously applied to only first order optimizers like gradient descent and convex optimization, to a new class of second-order optimizers. This goes beyond prior work with nonlinear least squares that currently only support differentiation with standard unrolling, which is known to have challenges with compute, memory, and vanishing gradients. More details are described in Sec. 4.

Highlights of results. Together, the application-agnostic features let users easily set up a variety of problems like pose graph optimization, tactile state estimation, bundle adjustment, motion planning, and homography estimation, all of which are included as examples in the open source code and described in Sec. 3.1. In evaluations, we find that on a standard GPU, `Theseus` with a sparse solver is much faster and requires significantly less memory than a dense solver, and when solving a batch of large problems the forward pass of Theseus is up to 20x faster than state-of-the-art C++ based solver `Ceres` that has limited GPU support and does not support batching and end-to-end learning. We also compare all backward modes to find that with increasing number of optimization iterations, compute and memory increases linearly for unrolling and stays constant for implicit differentiation, while the latter also provides better gradients. More details are described in Sec. 5.

2 Background and related work

Nonlinear least squares (NLS) is an optimization problem [1] that finds optimization variables θ

$$\theta^\star = \arg\min_\theta S(\theta), \qquad S(\theta) = \frac{1}{2}\sum_i ||r_i(\theta^i)||^2 = \frac{1}{2}\sum_i ||w_i c_i(\theta^i)||^2 \tag{1}$$

where the objective $S(\theta)$ is a sum of squared vector-valued residual terms r_i, each a function of $\theta^i \subset \theta$ that are (non-disjoint) subsets of the optimization variables $\theta = \{\theta_j\}$. Any variable θ_j is a manifold object; for example, a Euclidean vector or a matrix Lie group. For flexibility, we represent a residual $r_i(\theta^i) = w_i c_i(\theta^i)$ as a product of a matrix weight w_i and vector cost c_i. **Robotics and**

vision have used this general optimization formulation to tackle many applications [3, 4]. For example, costs capture sensor measurement errors and physical constraints to optimize camera, robot, object, or human poses in estimation and tracking problems like simultaneous localization and mapping (SLAM) [42], structure from motion [13], bundle adjustment [8], visual inertial odometry [2], articulated tracking [12], contact odometry in legged locomotion [27], 3D pose and shape reconstruction of humans [30, 31] or objects [10]. Similarly, costs can also capture constraints and desired future goals to find robot states or actions in motion planning [5], dynamics [6], and control [29] problems.

Solving NLS. Problems represented by Eq. (1) are solved by iteratively linearizing the nonlinear objective around the current variables to get the linear system $(\sum_i J_i^\top J_i)\delta\theta = (\sum_i J_i^\top r_i)$, then solving the linear system to find the update $\delta\theta$, and finally updating the variables $\theta \leftarrow \theta - \delta\theta$, until convergence. Note that in the update the minus operation is more generally a retraction mapping for non-Euclidean variables. In the linear system, $J_i = [\partial r_i / \partial \theta^i]$ are the Jacobians of residuals with respect to the variables and the iterative method above, called Gauss-Newton (GN), is a nonlinear optimizer that is (approximately) second-order, since $H = (\sum_i J_i^\top J_i)$ represents the approximate Hessian. To improve robustness and convergence, variations like Levenberg–Marquardt (LM) damp the linear system, while others use a trust region and adjust step size for the update with line search (e.g., Dogleg). Please refer to [1, 43] for an in-depth exploration. In most applications discussed above the objective structure gives rise to a sparse Hessian, since not all costs depend on all variables. Several general purpose frameworks [35–38] have been built that leverage this sparsity property to efficiently solve the sparse linear system in every iteration of the nonlinear optimization. While these frameworks were not built for deep learning, they are highly efficient and performant on CPU.

NLS with learning. Data driven learning has been explored to address challenges in hand crafting costs or features for costs, finding weights to balance different costs, or to find initializations that lead to better convergence. Some examples include, learning object shape code [44] or environment depth code [45] for SLAM [46], learning motion priors for planning to manipulate articulated objects [47], learning relative pose from tactile images to estimate object state during pushing [28], and semantic 2D segmentation fused in 3D mesh for semantic SLAM [7]. These approaches only train features on a surrogate or intermediate loss and then apply optimization at inference where the true downstream task loss is available but not utilized. To take full advantage of end-to-end learning, latest approaches thus are redesigning the optimization to be differentiable.

Differentiable NLS (DNLS) solves the optimization in Eq. (1) and also provides gradients of the solution $\theta^\star$ with respect to any upstream neural model parameters ϕ that parameterize the objective $S(\theta; \phi)$ and in turn any costs $c_i(\theta^i; \phi)$, weights $w_i(\phi)$, or initialization for variables $\theta_{init}(\phi)$. The goal is to learn these parameters ϕ end-to-end with a downstream learning objective L defined as a function of $\theta^\star$. This results in a bilevel optimization setup as shown in Fig. 1

$$\text{inner loop: } \theta^\star(\phi) = \arg\min_\theta S(\theta; \phi), \qquad \text{outer loop: } \phi^\star = \arg\min_\phi L(\theta^*(\phi)) \tag{2}$$

where the inner loop is DNLS and the outer loop is gradient descent class of optimization that is standard in deep learning. The outer loop performs update $\phi \leftarrow \phi + \delta\phi$ by computing $\delta\phi$ using gradients $\partial\theta^\star / \partial\phi$ through inner loop DNLS. Note that more generally the learning objective i.e. outer loss L can also depend on other quantities like neural model parameters downstream of θ^*, but we omit them here for clarity.

Recent works with DNLS have outperformed optimization only or learning only methods by combining the strengths of classical methods with deep learning. For example, learning features for costs to represent depth in bundle adjustment [23] and monocular stereo [48] where an initialization network also learns to predict depth and pose, learning cost weights like motion model weights in video to depth estimation [19], obstacle avoidance weights in 2D motion planning from occupancy images [20], learning robust loss weights in image alignment [26] and state-of-the-art dense SLAM [22], and confidence weights for feature matching to optimize camera pose [49]. Other works, backpropagate reconstruction error to sensor model in a SLAM system [21], solve large scale bundle adjustment on a GPU [50], and learn sensor and dynamics models for 2D visual object tracking and visual odometry [24]. These implementations however, are application specific which has led to repeated work in building DNLS where features like learnable costs and weights, Lie groups, and kinematics are not always present. Additionally, features that have a significant impact on performance, like sparsity and vectorization of costs are only considered by some [24, 50, 51] or in the case of implicit differentiation for NLS optimization, have not yet been explored.

3 Application agnostic interface

Given the lack of a common and efficient framework for DNLS an important goal of Theseus is to provide an application-agnostic interface. In this section, we describe how we enable this with an easy-to-use core API, standard cost functions, and features like Lie groups and kinematics, and illustrate several examples using this interface. We discuss design for efficiency in the next section.

```
1  x_true, y_true, v_true = read_data() # shapes (1, N), (1, N), (1, 1)
2
3  x = th.Variable(torch.randn_like(x_true), name="x")
4  y = th.Variable(y_true, name="y")
5  v = th.Vector(1, name="v") # a manifold subclass of Variable for optim_vars
6
7  def error_fn(optim_vars, aux_vars):  # returns y - v * exp(x)
8      x, y = aux_vars
9      return y.tensor - optim_vars[0].tensor * torch.exp(x.tensor)
10
11 objective = th.Objective()
12 cost_function = th.AutoDiffCostFunction(
13     [v], error_fn, y_true.shape[1], aux_vars=[x, y],
14     cost_weight=th.ScaleCostWeight(1.0))
15 objective.add(cost_function)
16 layer = th.TheseusLayer(th.GaussNewton(objective, max_iterations=10))
17
18 phi = torch.nn.Parameter(x_true + 0.1 * torch.ones_like(x_true))
19 outer_optimizer = torch.optim.Adam([phi], lr=0.001)
20 for epoch in range(10):
21     solution, info = layer.forward(
22         input_tensors={"x": phi.clone(), "v": torch.ones(1, 1)},
23         optimizer_kwargs={"backward_mode": "implicit"})
24     outer_loss = torch.nn.functional.mse_loss(solution["v"], v_true)
25     outer_loss.backward()
26     outer_optimizer.step()
```

Listing 1: Simple DNLS example with Theseus, see App. A for details.

The core API lets users focus on describing the DNLS problem and their interaction with the outer loss L and parameters ϕ within any broader PyTorch model, while the solution and differentiation are seamlessly taken care of under-the-hood. The basic components of the core API are described below with the help of a simple example in Listing 1 (see App. A for more details on the example):

- Variable: refers to either *optimization variables*, θ, or *auxiliary variables* (those constant with respect to S, e.g., parameters ϕ or data tensors), which are named wrappers of torch batched tensors stored in Variable.tensor (lines 3-5).
- CostFunction: defines costs c_i (lines 12-14) and are also responsible for declaring which of its variables are optimization and which are auxiliary (lines 8-9),
- CostWeight: defines weights w_i associated with cost c_i (line 14).
- Objective: defines $S(\theta; \phi)$, and thus the structure of an optimization problem (lines 11, 15) by holding all cost functions and weights, and their associated variables. These are implicitly obtained when a CostFunction is added to the Objective, and Variable names are used to infer which are shared by one or more CostFunction.
- Optimizer: is the inner loop optimization algorithm (e.g. Gauss-Newton) that finds the solution $\theta^\star$ given objective S (line 16).
- TheseusLayer: encapsulates Optimizer and Objective, and serves as the interface between the DNLS block and other torch modules upstream or downstream (line 16).

The interface between the inner loop optimization and the outer loop's parameters and loss occurs via TheseusLayer.forward (lines 21-23). This receives as input a dictionary mapping variable names to torch tensors, which Theseus then uses to populate the corresponding Variable with the tensor mapped to its name. With the input dictionary users can provide initial values for the optimization variables, data tensors, or current values for parameters ϕ before running the inner loop optimization. The output of forward is another dictionary that maps variable names to tensors with their optimal

values found in the inner loop (lines 21, 24); auxiliary variables are not modified during the forward pass. The output tensors can then be combined with other `torch` modules downstream to compute L, while maintaining the full differentiable computation graph (lines 24-26).

We currently provide Gauss-Newton and Levenberg-Marquardt as `Optimizer` for the inner loop, with the ability to easily add support for more optimizers in the future. Listing 1 uses `AutoDiffCostFunction` to construct an in-place `CostFunction` (line 12) which allows automatically calculating Jacobians J_i with `PyTorch` (see App. B). Beyond this, in the library we include standard cost functions with analytical Jacobians broadly used in many applications, like Gaussian measurements, reprojection error, relative pose measurement, motion models, and collision costs. We also include a variety of robust loss functions, useful for example in handling outliers [52], which can be easily integrated with `CostFunction`. Next we describe support for Lie groups and kinematics.

Differentiable Lie groups. Lie groups are widely used in robotics and vision to represent 2D/3D positions and rotations [33]. Due to their non-Euclidean geometry, it is difficult to apply them to deep learning, which primarily operates with Euclidean tensors, but recently there is growing interest in making them compatible [24, 53–57]. LieTorch [54] generalizes automatic differentiation on the Lie group tangent space through local parameterization around the identity, but the implementation is complex since every operation requires a custom kernel. In contrast, `Theseus` computes common Lie group operators, e.g., the exponential and logarithm map, inverse, composition, etc., in closed form, and provides their corresponding analytical derivatives on the tangent space. Following [58], we also implement a projection operator that allows us to project gradients computed by `PyTorch`'s autodiff to the tangent space and use them to easily compute Jacobians and update Lie group variables correctly; a similar strategy has also been implemented in [59]. Additionally, our Lie group implementation includes a heuristic extension that allows using any of `PyTorch`'s first-order optimizers on non-Euclidean manifolds with minimal code changes. All of these make it easy and straightforward to run optimization and train neural networks with Lie groups variables. More details in App. C.

Differentiable kinematics. Many problems such as motion planning or state estimation on high degree of freedom robots like arms or mobile manipulators, involve computation of robot kinematics for collision avoidance or computing distance of end effector to goal. `Theseus` provides a differentiable implementation of forward kinematics by wrapping over Differentiable Robot Model [34], which builds a differentiable kinematics function from a standard robot model file. Gradients are computed through autodiff, while we also provide a more efficient, analytical manipulator Jacobian. This module can be used within any `CostFunction` in `Theseus`.

3.1 Example applications

To illustrate the versatility of `Theseus`, we include a number of example DNLS applications below with more details in App. D. Crucially, to implement these with `Theseus`, most of the effort is only in defining application-specific components such as data management, neural models, or custom `CostFunction`. With these defined, putting the full DNLS block together is a few lines of code to setup a `TheseusLayer` and an outer loop, similar to the simple example in Listing 1.

Pose graph optimization (PGO) estimates poses from their noisy relative measurements [60]. With DNLS we learn the radius of a Welsh robust cost function for outlier rejection, using the difference between estimated and ground truth poses as the outer loss on a synthetic dataset.

Tactile state estimation follows [28], which estimates 2D poses of an object pushed by a robot hand with an image-based tactile sensor [61]. A neural network that predicts relative pose between hand and object from tactile images is learned end-to-end through the `TheseusLayer`.

Bundle adjustment is the problem of optimizing a 3D reconstruction formed by a set of camera images and a set of landmarks observed and matched across the images [62]. We learn the radius of a soft-kernel that penalizes outlier observations, using the average frame pose error as outer loss.

Motion planning considers a differentiable version of the GPMP2 planning algorithm, inspired by [20], where the outer loss tries to match expert demonstrations. Here we learn a model for initializing optimization variables, and we include the inner loop objective as a term in the outer loss.

Homography estimation. Homography is a linear transformation between corresponding points in two images and can be solved by minimising a dense photometric loss. Robustness to lighting and viewpoint change can be improved with a feature-metric loss based on CNN features [63–68]. In our outer loop, we train a CNN to produce robust features for image alignment.

4 Efficiency based design

Theseus enables several different applications with a general interface. Compute and memory efficiency are central to making its usage practical. Next, we explain design considerations to support batching and vectorization, sparsity, and backward modes for differentiation, which we demonstrate boost performance in the evaluations section.

4.1 Batching and vectorization

Parallel processing is important to improve computational efficiency in machine learning and optimization. In Theseus, we enable two levels of parallelization. First, Theseus natively supports solving a batch of DNLS in parallel, thus fitting seamlessly in the PyTorch framework, where training and inferences on batches is the standard. Second, inspired by DeepLM [50], and noting that lots of the operations such as costs, gradients/Jacobian computation, and variable updates only differ from each other in terms of the input data, we make use of the single-instruction-multiple-data (SIMD) protocol by automatically detecting and vectorizing operations of the same type, significantly reducing computation overhead. Using the PGO example, Fig. 2 shows that Theseus achieves significant speedup with automatic vectorization both for forward and backward pass. Note that there is an application-dependent trade-off between memory and speed; here the memory use increases by up to $\sim 82\%$ for forward and $\sim 55\%$ for backward.

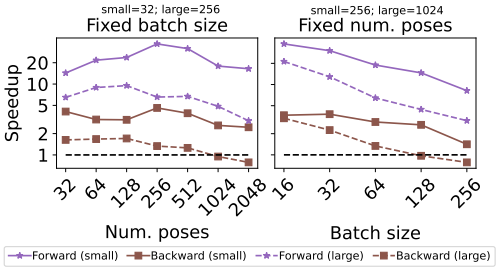


Figure 2: Speedup with automatic vectorization on PGO. Black dotted line is without vectorization.

4.2 Handling sparsity with linear solvers beyond PyTorch

Solving NLS requires solving a sequence of linear systems to obtain descent directions. As discussed in Sec. 2, these systems are generally sparse and can be solved much more efficiently if not treated as dense. Theseus includes differentiable sparse solvers that take advantage of the sparsity, complementing PyTorch's native dense solvers. Importantly, Theseus seamlessly takes care of assembling the cost functions and variables in the objective into sparse data structures that our linear solvers can consume, without any extra burden on the user. Currently, we provide three sparse solvers: (i) a CPU-based solver that relies on CHOLMOD [69], (ii) cudaLU, which is based on the cuSolverRF package that is part of Nvidia's cuSolver library provided with CUDA, and (iii) BaSpaCho, our novel batched sparse Cholesky solver with GPU support. As a bonus feature, we provide access to these solvers as standalone PyTorch functions, so they can be used to solve sparse matrices arising outside of NLS or DNLS optimization.

CHOLMOD-based solver. CHOLMOD [69] achieves state-of-the-art performance on computation of the Cholesky decomposition of sparse matrices. It exploits parallelism by grouping sparse entries to take advantage of high-performance multi-threaded dense matrix operations in BLAS/LAPACK libraries. CHOLMOD has some limited support for GPU for some of its operations, but the algorithm is strongly CPU-based, and the user is expected to provide matrix data on the CPU. One convenient feature is computing the symbolic analysis of a sparse matrix pattern as a separate step and creating a symbolic decomposition object that can be used for all subsequent factorizations. We also take advantage of builtin functionality for sparse multiplication and only provide the Jacobian matrix J to solve for the Hessian $H = J^\top J$. Two limitations of the library with respect to Theseus are, first, the lack of proper GPU support, which forces us to provide matrix data on the CPU, and, second, the lack of batching, which requires us to loop to solve every problem in the batch independently. On the other hand, since it runs on CPU, it has less memory restrictions than GPU-based solvers (see Sec. 5.1).

cudaLU solver. cuSolverRF is designed to accelerate the solution of sets of linear systems by fast LU refactorization when given new coefficients for the same sparsity pattern. To take advantage of this, we implemented custom CUDA kernels for batched sparse matrix-matrix and matrix-vector products, and for solving a batch of sparse linear systems using LU factorization from cuSolverRF. Although this solver leads to a substantial performance boost over PyTorch's dense solver (see Sec. 5.1), the closed-source nature of cuSOLVER results in some challenges and limitations: (i) cuSolverRF does not support separate symbolic decomposition and numeric contexts, so it's not possible to use the same symbolic decomposition to hold in memory separate factors. Since this is necessary in Theseus for unrolling of the inner loop, we work around this limitation by creating a pool of contexts, and we use the least recently used context for factorization. As a consequence, the number of contexts must

be set according to the number of iterations that need to be unrolled; (ii) The batch size is fixed once a context is created. Since recreating the contexts is an expensive operation, it means that the batch size has to be constant over the course of outer loop optimization; (iii) It relies on LU factorization, which for symmetric matrices (the case of `Theseus`) is less efficient than using Cholesky decomposition.

`BaSpaCho` solver. Batched Sparse Cholesky (`BaSpaCho`) is a novel open-source sparse Cholesky solver designed for `Theseus` with support for batching (https://github.com/facebookresearch/baspacho). `BaSpaCho` implements the *supernodal* Cholesky algorithm [70] to achieve state-of-the art performance by exploiting dense operations via `BLAS`/`cuBLAS`. This is achieved by building an elimination tree and then clustering column blocks with similar sparsity patterns. These blocks form nodes of the elimination tree and allow dense operations. In `BaSpaCho`, the dense operations are dispatched to `BLAS` (on CPU) or `cuBLAS` (on GPU), with additional support added on top for batching matrix operations with the same sparsity patterns. In problems with very sparse matrices, like bundle adjustment [8], the supernodal algorithm employed in state-of-the-art solvers [37] is unable to eliminate columns of parameter blocks simultaneously. Thus, past work has resorted to the Schur complement trick [71] to send a reduced problem to the sparse solver. However, this logic adds extra complexity to the nonlinear optimization, while essentially duplicating the work of the (mathematically equivalent) Cholesky decomposition. In `BaSpaCho`, we instead complement the supernodal algorithm with *sparse elimination* that removes the need to externally handle Schur complement as a workaround to the limitation of the supernodal algorithm. More details are described in App. E.

Backward for custom linear solvers. Obtaining derivatives of the linear system solve with respect to the parameters is a crucial operation for DNLS. In particular, we consider optimizing the parameters A and b of a linear system $y = A^{-1}b$ to minimize a downstream function $f(y)$. The derivatives of the loss with respect to the parameters of the linear system can be obtained with implicit differentiation, $\frac{\partial f}{\partial b} = A^{-1}\frac{\partial f}{\partial y}$ and $\frac{\partial f}{\partial A} = -A^{-1}\frac{\partial f}{\partial y}y^\top$, as done in Barron and Poole [72]. In `Theseus`, we implement this by connecting the Python interface of our sparse solvers with `PyTorch`'s `autograd.Function` classes that implement the gradients above in their `backward` methods. This connects the computation graph between the downstream function and any upstream parameters that modify the system via auxiliary variables or values for optimization variables. Furthermore, since the gradients require solving linear systems that use the same matrix as the forward pass, our backward pass can cache factorizations, resulting in it being significantly faster than the forward pass (see Fig. 3).

4.3 Backward modes for DNLS

The parameters ϕ upstream of DNLS can be learned end-to-end through the solution $\theta^\star(\phi)$ by using the *adjoint derivatives* $\partial\theta^\star(\phi)/\partial\phi$. We include four methods for computing them in `Theseus`.

Unrolling is the standard way in which past work in DNLS has computed the adjoint derivatives. This is often referred to as backpropagation through time or unrolled optimization and is explored in [16, 20, 73–82]. In practice, often only a few steps of unrolling are performed due to challenges with compute, memory, and vanishing gradients.

Truncated differentiation. Aside from unrolling a few steps, another way of approximating the derivatives is to use truncated backpropagation through time (TBPTT) [83, 84]. Truncation unfortunately results in biased derivatives and many works [85–89] seek to further theoretically understand the properties of TBPTT, including the bias of the estimator and how to unbias it.

Implicit differentiation. If $\theta^\star$ can be computed exactly, then the implicit function theorem provides a way of computing the adjoint derivatives, as done in related work in convex optimization and first-order gradient descent methods [14, 15, 90–95]. We apply the implicit function theorem from Dontchev and Rockafellar [39, Theorem 1B.1] (see App. G) to Eq. (2) to perform implicit differentiation on a new class of second-order NLS optimization. This first requires that we transform Eq. (2) into an implicit function that finds the roots. We do this via the first-order optimality condition, resulting in $g(\theta;\phi) := \nabla_\theta S(\theta;\phi)$. Finding $\Theta^\star(\phi) := \{\theta \mid g(\theta;\phi) = 0\}$ corresponds to solving Eq. (2). Under mild assumptions, the theorem above gives the adjoint derivative at $\bar{\phi}$

$$\mathrm{D}_\phi\theta^\star(\bar{\phi}) = -\mathrm{D}_\theta^{-1}g(\theta^\star(\bar{\phi});\bar{\phi})\mathrm{D}_\phi g(\theta^\star(\bar{\phi});\bar{\phi}). \tag{3}$$

As `Theseus` internally uses a (Gauss-)Newton solver, the following proposition provided in App. G shows how we can compute Eq. (3) by differentiating a single Newton step at an optimal solution.

Proposition 1. *Differentiating a single Newton iteration h at an optimal and unique $\theta^\star$ results in the exact computation of the implicit derivative in Eq. (3).*

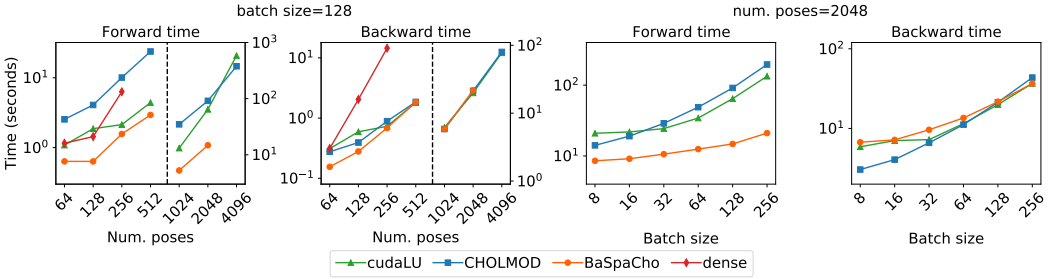


Figure 3: Forward/backward times of Theseus with sparse and dense solvers on different PGO problem scales.

Direct loss minimization. Suppose we have an outer loss as in Eq. (2). The direct loss minimization (DLM) approach uses this loss to augment the inner-loop optimization problem in order to define a finite difference scheme that approaches the true gradient $\nabla_\phi L = \lim_{\varepsilon \to 0} g^\varepsilon_{\text{DLM}}$, where $g^\varepsilon_{\text{DLM}} \triangleq \frac{1}{\varepsilon}\{\frac{\partial}{\partial \phi} S(\theta^*; \phi) - \frac{\partial}{\partial \phi} S(\theta_{\text{direct}}; \phi)\}$. This was used in prior works that solve optimization problems on structured discrete domains [40, 41, 96, 97], but has so far not seen much use in structured continuous settings. We modify the original DLM formulation to better suit its implementation within Theseus

$$\theta^\star = \arg\min_{\hat{\theta}} S(\hat{\theta}; \phi), \qquad \theta_{\text{direct}} = \arg\min_{\hat{\theta}} S(\hat{\theta}; \phi) + \left\| \varepsilon \hat{\theta} - \tfrac{1}{2} \nabla_\theta L(\theta^*) \right\|^2. \tag{4}$$

This is different from the original formulation in two ways: (i) we only assume access to the gradient vector $\nabla_\theta L(\theta^*)$, which helps formulate DLM as an algorithm for computing vector-Jacobian products, and (ii) we add a small regularization term to ensure the modified objective for θ_{direct} is a sum-of-squares without affecting the limit as $\varepsilon \to 0$. See App. G for more details.

5 Evaluation

We evaluate the performance of Theseus under different settings with PGO and tactile state estimation applications from Sec. 3.1. PGO allows us to easily control the problem scales for performance evaluation; in Sec. 5.1 we profile time and memory consumption of Theseus in an end-to-end setup and in Sec. 5.2 we evaluate timings of Theseus as a stand-alone NLS optimizer and compare with state-of-the-art Ceres [37]. The tactile state estimation application involves a more complex outer loop model that is useful for comparing all different backward modes, which we present in Sec. 5.3.

5.1 Profiling forward and backward pass of Theseus for DNLS

We study the performance of Theseus for DNLS on the PGO problem [60] with the synthetic Cube dataset, as described in App. D. We run 10 inner loop iterations and 20 outer loop epochs, and use implicit differentiation to compute gradients of the inner NLS optimization. For these experiments we used an Nvidia V100 GPU with 32GBs of memory for all Python computation, and Intel Xeon 2.2GHz CPU with 20 threads for the CPU-based CHOLMOD linear solver. We evaluate performance using our sparse solvers in Theseus and using PyTorch's Cholesky dense solver.

Fig. 3 shows the average time of a full forward and backward pass for a given batch size, taken by Theseus with different solvers (cudaLU, CHOLMOD, BaSpaCho and dense), for different problem scales (number of poses and batch size). The two left plots show time as a function of number of poses for a batch size of 128, while the two right plots show time as a function of batch size for 2048 poses. We find that dense does not scale well with poses or batch size. For a batch size of 128, the largest problem that it can solve before running out of GPU memory has 256 poses (left two plots). With 2048 poses, dense is unable to solve the problem regardless of batch size (right two plots). On the other hand with a batch size of 128, our solvers BaSpaCho scale to 2048 poses and cudaLU scale to 4096 poses. CHOLMOD can solve problems even larger, since the linear system is solved on CPU and we have successfully tested up to 8192 poses and batch size 256 (see App. F), for a total of 22GBs of GPU usage for residuals and Jacobian blocks computation.

In addition to being more memory efficient, running times of our sparse solvers are also smaller for large enough number of poses/batch size, especially for the backward pass. Even though dense's total time for forward+backward is comparable to cudaLU and faster than CHOLMOD for smaller problems: e.g., 1.47s (dense) vs. 1.32s (cudaLU) and 2.82s (CHOLMOD) for batch size 128 and 64 poses, dense is significantly slower or out of memory for larger problems. For the largest problem that dense can solve (batch size 128 and 256 poses) its total time is already much slower than all others methods:

20.81s (dense) vs. 10.96s (CHOLMOD), 2.86s (cudaLU), and 2.25s (BaSpaCho). Furthermore, BaSpaCho outperforms dense for any problem scale and is up to one order of magnitude faster, including for smaller batch sizes and number of poses (see App. F for more results and details). For the largest problem that we consider (batch size 256 and 2048 poses), the total times for our sparse solvers are 170.28s for cudaLU, 239.07s for CHOLMOD, and 57.67s for BaSpaCho.

5.2 Profiling Theseus as stand-alone NLS optimizer

DNLS typically involves solving numerous optimization problems each epoch where a fast NLS optimizer is essential. We compare Theseus as a stand-alone NLS optimizer with the state-of-the-art Ceres [37] library for solving a batch of PGO problems without any learning involved. We compare all solvers in terms of the total time required to perform 10 iterations on a set of 256 PGO problems. CPU/GPU configurations are same as before. For CHOLMOD, we also include a configuration that runs everything on CPU, including Jacobians and residual computation (labelled CHOLMOD-allcpu).

Fig. 4 shows speedup obtained by Theseus with batching, vectorization and sparse solvers, over Ceres as a function of increasing number of poses or batch size. We vary the number of poses for a fixed batch size of 256, and vary the batch size for a fixed number of poses of 2048. Although Ceres is faster than all of our solvers when the number of poses and batch size are small (for instance, Ceres is 25x faster with 256 poses and 16 batch size, see App. F), as these increase Theseus shows significant speedup by being able to solve larger batches of problems in parallel. For the largest setup that all our solvers can scale to (2048 poses, 256 batch size), BaSpaCho is ∼23x faster than Ceres, and our other solvers are ∼4x faster. CHOLMOD has a 6x speedup for its largest setting (4096 poses, 256 batch size).

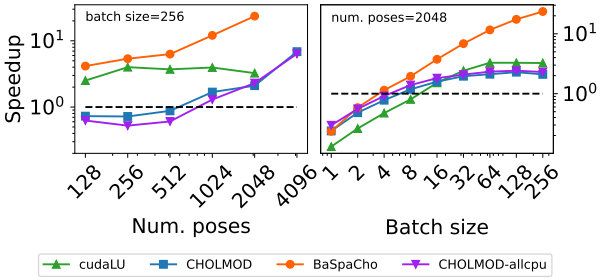


Figure 4: Speedup of Theseus (forward pass) over Ceres (black dashed) on different PGO problem scales.

Since typical use case of Theseus involves large batches and number of variables during end-to-end learning with DNLS, the speedups in this setting against a performant NLS solver highlights the significance of our efficiency-based design choices. See App. F for additional results of smaller fixed batch size and number of poses.

5.3 Backward modes analysis

We explore the trade-offs between our different backward modes using the tactile state estimation application in Sec. 3.1. The learnable components here include a neural network, and thus closely follow the type of applications that motivate Theseus. We compare the following backward modes: derivative unrolling (Unroll), implicit differentiation (Implicit), truncated differentiation (Trunc), and direct loss minimization (DLM); for Trunc we include results when truncating 5 and 10 steps. We compare all modes along 3 axis of performance: validation loss after 100 epochs (outer loop), run time during training, and peak GPU memory consumption of TheseusLayer. For these experiments we used Quadro GP100 GPUs with 16GB of memory. For time and memory we present separate results for forward and backward pass, and all numbers are averaged over 700 (7 batches for 100 epochs). Below we discuss our main findings from this analysis, and more results and details can be found in App. G.

Fig. 5 shows average run times for all backward modes as a function of the maximum number of iterations in the inner loop optimization. We observe that the time used in the forward pass (Fig. 5, far left) increases roughly linearly for all modes, all having similar times except for Unroll, which is slower than other modes. On the other hand, we observe stark differences in the backward pass time (Fig. 5, center left), where Unroll is the only method that has a linear dependence on the number of inner loop iterations. All other methods have a constant footprint for computing derivatives, independent of the number of inner loop iterations. As expected, increasing the number of iterations through which we backprop (5 or 10 for Trunc, all iterations for Unroll) increases the time necessary for a backward pass (Implicit = DLM $<$ Trunc-5 $<$ Trunc-10 $<<$ Unroll).

Figure 5 (center right) shows the average peak memory consumption of the backward modes. In this case, the trends observed for the backward pass memory consumption is similar to the trends

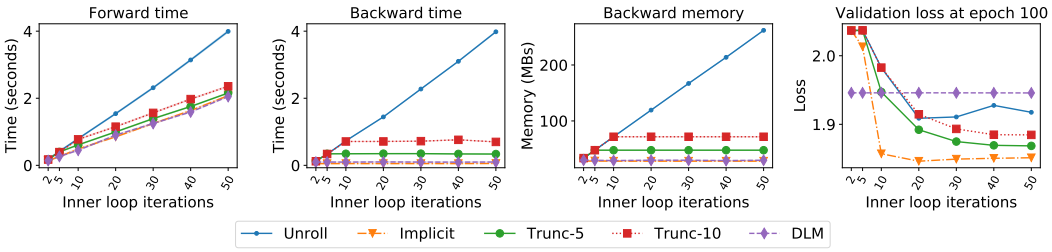


Figure 5: Time and memory consumption of different backward modes in tactile state estimation.

in time. In particular, `Unroll`'s memory footprint increases linearly with the number of inner loop iterations, from $\sim$ 34MBs to $\sim$ 262MBs; for all other methods the memory consumption remains constant. The best memory profiles in this example is obtained with `Implicit` and `DLM` backward modes, with $\sim$ 28MBs and $\sim$ 29MBs, respectively. These trends also hold for the forward pass memory consumption.

Figure 5 also shows the validation losses obtained with all backward modes (far right). The best validation loss, after 100 epochs of training, is obtained using `Implicit`, followed by `Trunc` variants. We notice that both variants of `Trunc` keep improving with increasing number of inner loop iterations, and that `Unroll` and `Implicit` achieve the best results with 20 iterations. One exception is `DLM`, which doesn't improve much with the number of iterations, but is also the best method when only 2 inner loop iterations are performed. As a point of caution, we stress that, unlike the timing and memory results, the relative training performance between different backward modes is likely to be application dependent, and is affected by hyperparameters such as the step size used for the inner loop optimizer (0.05 in this example), and the outer optimizer's learning rate. Our experiments suggest that implicit differentiation is a good default to use for differentiable optimization, considering its low time/memory footprint, and potential for better end-to-end performance with proper hyperparameter tuning.

6 Discussion

Summary. `Theseus` provides nonlinear least squares as a differentiable layer and enables easily building and training end-to-end architectures for robotics and vision applications. We illustrate several example applications using the same application-agnostic interface and demonstrate significant improvements in performance with our efficiency-based design. Following how autodiff and GPU acceleration (among others) have led to the evolution of `PyTorch` in contrast to `NumPy` [98], we can similarly view sparsity and implicit differentiation on top of autodiff and GPU acceleration as the key ingredients that power `Theseus`, in contrast to solvers like `Ceres` that typically only support sparsity. When solving a batch of large problems the forward pass of Theseus is up to 20x faster than `Ceres`.

Limitations. `Theseus` currently has a few limitations. The nonlinear solvers we currently support apply constraints in a soft manner (i.e., using weighted costs). Hard constraints can be handled with methods like augmented Lagrangian or sequential quadratic programs [99, 100], and differentiating through them are active research topics. The current implementation of LM does not support damping to be learnable. Some limitations and trade-offs with the sparse linear solvers are discussed in Sec. 4.2, and with backward modes are discussed in App. G. Online learning applications may require frequently editing the objective and depending on the problem size there may be a nontrivial overhead that is not currently optimized as we explored only non-incremental settings in this work. Additional performance gains can be extracted by moving some of our Python implementation to C++ but we prioritized flexibility in evolving the API in the short-term. We do not yet support distributed training beyond what `PyTorch` natively supports. We will explore these features and optimizations in the future as the library continues to evolve.

Acknowledgments and Disclosure of Funding

The authors would like to thank Dhruv Batra, Olivier Delalleau, Jessica Hodgins, and Mary Williamson for guidance and support on the project, Dhruv Batra and Sal Candido for feedback on early drafts of the paper, Franziska Meier for help with the differentiable robot model library, Paul-Edouard Sarlin for helpful discussion on the homography example, and Horace He, Richard Zou and Samantha Andow for help with `functorch` [101] library. The authors also thank community members for contributions to the open source code. Work by PS and JO was done while at Meta AI.

References

[1] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006. 1, 2, 3

[2] Christian Forster, Luca Carlone, Frank Dellaert, and Davide Scaramuzza. On-manifold preintegration for real-time visual–inertial odometry. *IEEE Transactions on Robotics*, 33(1): 1–21, 2016. 1, 3

[3] Frank Dellaert, Michael Kaess, et al. Factor graphs for robot perception. *Foundations and Trends® in Robotics*, 6(1-2):1–139, 2017. 3

[4] Timothy D. Barfoot. State estimation for robotics. *State Estimation for Robotics*, 2017. doi: 10.1017/9781316671528. 3

[5] Mustafa Mukadam, Jing Dong, Xinyan Yan, Frank Dellaert, and Byron Boots. Continuous-time gaussian process motion planning via probabilistic inference. *The International Journal of Robotics Research*, 37(11):1319–1340, 2018. 3, 21

[6] Mandy Xie and Frank Dellaert. A unified method for solving inverse, forward, and hybrid manipulator dynamics using factor graphs. *arXiv preprint arXiv:1911.10065*, 2019. 3

[7] Antoni Rosinol, Marcus Abate, Yun Chang, and Luca Carlone. Kimera: an open-source library for real-time metric-semantic localization and mapping. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1689–1696. IEEE, 2020. 1, 3

[8] Bill Triggs, Philip F McLauchlan, Richard I Hartley, and Andrew W Fitzgibbon. Bundle adjustment—a modern synthesis. In *International workshop on vision algorithms*, pages 298–372. Springer, 1999. 1, 3, 7, 21

[9] Hanna Pasula, Stuart Russell, Michael Ostland, and Yaacov Ritov. Tracking many objects with many sensors. In *IJCAI*, volume 99, pages 1160–1171. Citeseer, 1999.

[10] Richard Szeliski and Sing Bing Kang. Recovering 3d shape and motion from image streams using nonlinear least squares. *Journal of Visual Communication and Image Representation*, 5 (1):10–28, 1994. 3

[11] Aleksandr Segal, Dirk Haehnel, and Sebastian Thrun. Generalized-icp. In *Robotics: science and systems*, volume 2, page 435. Seattle, WA, 2009.

[12] Tanner Schmidt, Richard A Newcombe, and Dieter Fox. Dart: Dense articulated real-time tracking. In *Robotics: Science and Systems*, volume 2, pages 1–9. Berkeley, CA, 2014. 3

[13] Johannes L Schonberger and Jan-Michael Frahm. Structure-from-motion revisited. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4104–4113, 2016. 1, 3

[14] Brandon Amos and J Zico Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 136–145. JMLR. org, 2017. 1, 7

[15] Akshay Agrawal, Brandon Amos, Shane Barratt, Stephen Boyd, Steven Diamond, and J Zico Kolter. Differentiable convex optimization layers. In *Advances in Neural Information Processing Systems*, pages 9558–9570, 2019. 1, 7

[16] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pages 1126–1135. PMLR, 2017. 1, 7

[17] Edward Grefenstette, Brandon Amos, Denis Yarats, Phu Mon Htut, Artem Molchanov, Franziska Meier, Douwe Kiela, Kyunghyun Cho, and Soumith Chintala. Generalized inner loop meta-learning. *arXiv preprint arXiv:1910.01727*, 2019. 1

[18] David Duvenaud, Zico Kolter, and Matthew Johnson. Deep Implicit Layers: Neural ODEs, Equilibrium Models, and Differentiable Optimization. http://implicit-layers-tutorial.org/, 2020. 1

[19] Zachary Teed and Jia Deng. Deepv2d: Video to depth with differentiable structure from motion. *arXiv preprint arXiv:1812.04605*, 2018. 2, 3

[20] Mohak Bhardwaj, Byron Boots, and Mustafa Mukadam. Differentiable gaussian process motion planning. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10598–10604. IEEE, 2020. 2, 3, 5, 7, 21

[21] Krishna Murthy Jatavallabhula, Ganesh Iyer, and Liam Paull. ∇ slam: Dense slam meets automatic differentiation. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2130–2137. IEEE, 2020. 2, 3

[22] Zachary Teed and Jia Deng. Droid-slam: Deep visual slam for monocular, stereo, and rgb-d cameras. *Advances in Neural Information Processing Systems*, 34, 2021. 2, 3

[23] Chengzhou Tang and Ping Tan. BA-Net: Dense bundle adjustment networks. *7th International Conference on Learning Representations, ICLR 2019*, 2019. 2, 3

[24] Brent Yi, Michelle A Lee, Alina Kloss, Roberto Martín-Martín, and Jeannette Bohg. Differentiable factor graph optimization for learning smoothers. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1339–1345. IEEE, 2021. 2, 3, 5

[25] Hansheng Chen, Pichao Wang, Fan Wang, Wei Tian, Lu Xiong, and Hao Li. Epro-pnp: Generalized end-to-end probabilistic perspective-n-points for monocular object pose estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2781–2790, 2022. 2

[26] Zhaoyang Lv, Frank Dellaert, James M Rehg, and Andreas Geiger. Taking a deeper look at the inverse compositional algorithm. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4581–4590, 2019. 2, 3

[27] Ross Hartley, Maani Ghaffari Jadidi, Lu Gan, Jiunn-Kai Huang, Jessy W Grizzle, and Ryan M Eustice. Hybrid contact preintegration for visual-inertial-contact state estimation using factor graphs. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3783–3790. IEEE, 2018. 2, 3

[28] Paloma Sodhi, Michael Kaess, Mustafa Mukadam, and Stuart Anderson. Learning tactile models for factor graph-based estimation. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 13686–13692. IEEE, 2021. 2, 3, 5, 20, 21

[29] Markus Giftthaler, Michael Neunert, Markus Stäuble, Jonas Buchli, and Moritz Diehl. A family of iterative gauss-newton shooting methods for nonlinear optimal control. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1–9. IEEE, 2018. 2, 3

[30] Federica Bogo, Angjoo Kanazawa, Christoph Lassner, Peter Gehler, Javier Romero, and Michael J Black. Keep it smpl: Automatic estimation of 3d human pose and shape from a single image. In *European conference on computer vision*, pages 561–578. Springer, 2016. 2, 3

[31] Taosha Fan, Kalyan Vasudev Alwala, Donglai Xiang, Weipeng Xu, Todd Murphey, and Mustafa Mukadam. Revitalizing optimization for 3d human pose and shape estimation: A sparse constrained formulation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 11457–11466, 2021. 2, 3

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019. 2

[33] Joan Solà, Jeremie Deray, and Dinesh Atchuthan. A micro Lie theory for state estimation in robotics. *arXiv*, pages 1–17, 2018. URL http://arxiv.org/abs/1812.01537. 2, 5

[34] Franziska Meier, Austin Wang, Giovanni Sutanto, Yixin Lin, and Paarth Shah. Differentiable and learnable robot models. *arXiv preprint arXiv:2202.11217*, 2022. 2, 5

[35] Giorgio Grisetti, Rainer Kümmerle, Hauke Strasdat, and Kurt Konolige. g2o: A general framework for (hyper) graph optimization. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 9–13, 2011. 2, 3, 20

[36] Frank Dellaert. Factor graphs and gtsam: A hands-on introduction. Technical report, Georgia Institute of Technology, 2012.

[37] Sameer Agarwal and Keir Mierle. Ceres solver: Tutorial & reference. *Google Inc*, 2(72):8, 2012. 7, 8, 9, 20

[38] Jing Dong and Zhaoyang Lv. miniSAM: A flexible factor graph non-linear least squares optimization framework. *CoRR*, abs/1909.00903, 2019. URL http://arxiv.org/abs/1909.00903. 2, 3

[39] Asen L Dontchev and R Tyrrell Rockafellar. *Implicit functions and solution mappings*, volume 543. Springer, 2009. 2, 7, 26

[40] Tamir Hazan, Joseph Keshet, and David McAllester. Direct loss minimization for structured prediction. *Advances in neural information processing systems*, 23, 2010. 2, 8

[41] Yang Song, Alexander Schwing, Raquel Urtasun, et al. Training deep neural networks via direct loss minimization. In *International conference on machine learning*, pages 2169–2177. PMLR, 2016. 2, 8, 27

[42] Frank Dellaert and Michael Kaess. Square root sam: Simultaneous localization and mapping via square root information smoothing. *The International Journal of Robotics Research*, 25 (12):1181–1203, 2006. 3

[43] Carl T Kelley. *Iterative methods for optimization*. SIAM, 1999. 3

[44] Edgar Sucar, Kentaro Wada, and Andrew Davison. NodeSLAM: Neural Object Descriptors for Multi-View Shape Reconstruction. *Proceedings - 2020 International Conference on 3D Vision, 3DV 2020*, pages 949–958, 2020. doi: 10.1109/3DV50981.2020.00105. 3

[45] Michael Bloesch, Jan Czarnowski, Ronald Clark, Stefan Leutenegger, and Andrew J Davison. Codeslam—learning a compact, optimisable representation for dense visual slam. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2560–2568, 2018. 3

[46] Jan Czarnowski, Tristan Laidlow, Ronald Clark, and Andrew J. Davison. DeepFactors: Real-Time Probabilistic Dense Monocular SLAM. *IEEE Robotics and Automation Letters*, 5(2): 721–728, 2020. doi: 10.1109/LRA.2020.2965415. 3

[47] Muhammad Asif Rana, Mustafa Mukadam, Seyed Reza Ahmadzadeh, Sonia Chernova, and Byron Boots. Towards robust skill generalization: Unifying learning from demonstration and motion planning. In *Conference on Robot Learning*, pages 109–118. PMLR, 2017. 3

[48] Ronald Clark, Michael Bloesch, Jan Czarnowski, Stefan Leutenegger, and Andrew J Davison. Ls-net: Learning to solve nonlinear least squares for monocular stereo. *arXiv preprint arXiv:1809.02966*, 2018. 3

[49] Barbara Roessle and Matthias Nießner. End2end multi-view feature matching using differentiable pose optimization. *arXiv preprint arXiv:2205.01694*, 2022. 3

[50] Jingwei Huang, Shan Huang, and Mingwei Sun. Deeplm: Large-scale nonlinear least squares on deep learning frameworks using stochastic domain decomposition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10308–10317, 2021. 3, 6

[51] Borglab. Swiftfusion. https://github.com/borglab/SwiftFusion, 2020. 3

[52] Jonathan T Barron. A general and adaptive robust loss function. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4331–4339, 2019. 5

[53] Valentin Peretroukhin, Matthew Giamou, David M Rosen, W Nicholas Greene, Nicholas Roy, and Jonathan Kelly. A smooth representation of belief over so (3) for deep rotation learning with uncertainty. *Robotics: Science and Systems*, 2020. 5

[54] Zachary Teed and Jia Deng. Tangent Space Backpropagation for 3D Transformation Groups. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 10333–10342, 2021. doi: 10.1109/CVPR46437.2021.01020. URL http://arxiv.org/abs/2103.12032. 5

[55] Yi Zhou, Connelly Barnes, Jingwan Lu, Jimei Yang, and Hao Li. On the continuity of rotation representations in neural networks. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2019-June:5738–5746, 2019. doi: 10.1109/CVPR.2019.00589.

[56] Congyue Deng, Or Litany, Yueqi Duan, Adrien Poulenard, Andrea Tagliasacchi, and Leonidas J Guibas. Vector neurons: A general framework for so (3)-equivariant networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 12200–12209, 2021.

[57] Philippe Hansen-Estruch, Wenling Shang, Lerrel Pinto, Pieter Abbeel, and Stas Tiomkin. Gem: Group enhanced model for learning dynamical control systems. *arXiv preprint arXiv:2104.02844*, 2021. 5

[58] P-A Absil, Robert Mahony, and Rodolphe Sepulchre. *Optimization algorithms on matrix manifolds*. Princeton University Press, 2009. 5, 20

[59] Brent Yi. jaxlie. https://github.com/brentyi/jaxlie, 2021. 5

[60] D.M. Rosen, L. Carlone, A.S. Bandeira, and J.J. Leonard. SE-Sync: A certifiably correct algorithm for synchronization over the special Euclidean group. *Intl. J. of Robotics Research*, 38(2–3):95–125, March 2019. 5, 8, 20

[61] Mike Lambeta, Po-Wei Chou, Stephen Tian, Brian Yang, Benjamin Maloon, Victoria Rose Most, Dave Stroud, Raymond Santos, Ahmad Byagowi, Gregg Kammerer, et al. Digit: A novel design for a low-cost compact high-resolution tactile sensor with application to in-hand manipulation. *IEEE Robotics and Automation Letters*, 5(3):3838–3845, 2020. 5, 21

[62] Sameer Agarwal, Noah Snavely, Steven M Seitz, and Richard Szeliski. Bundle adjustment in the large. In *European conference on computer vision*, pages 29–42. Springer, 2010. 5, 21

[63] Jan Czarnowski, Stefan Leutenegger, and Andrew J. Davison. Semantic texture for robust dense tracking. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV) Workshops*, Oct 2017. 5, 22

[64] Jing Dong, Byron Boots, Frank Dellaert, Ranveer Chandra, and Sudipta Sinha. Learning to align images using weak geometric supervision. In *2018 International Conference on 3D Vision (3DV)*, pages 700–709. IEEE, 2018.

[65] Chengzhou Tang and Ping Tan. BA-Net: Dense Bundle Adjustment Network. In *ICLR*, 2019.

[66] Paul-Edouard Sarlin, Ajaykumar Unagar, Måns Larsson, Hugo Germain, Carl Toft, Viktor Larsson, Marc Pollefeys, Vincent Lepetit, Lars Hammarstrand, Fredrik Kahl, and Torsten Sattler. Back to the Feature: Learning robust camera localization from pixels to pose. In *CVPR*, 2021. URL https://arxiv.org/abs/2103.09213.

[67] Zhaoyang Lv, Frank Dellaert, James Rehg, and Andreas Geiger. Taking a deeper look at the inverse compositional algorithm. In *CVPR*, 2019.

[68] Chaoyang Wang, Hamed Kiani Galoogahi, Chen-Hsuan Lin, and Simon Lucey. Deep-lk for efficient adaptive object tracking. *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 627–634, 2018. 5

[69] Yanqing Chen, Timothy A Davis, William W Hager, and Sivasankaran Rajamanickam. Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software (TOMS)*, 35(3):1–14, 2008. 6

[70] Esmond Ng and Barry W Peyton. A supernodal cholesky factorization algorithm for shared-memory multiprocessors. *SIAM Journal on scientific computing*, 14(4):761–769, 1993. 7, 23

[71] Fuzhen Zhang. *The Schur complement and its applications*, volume 4. Springer Science & Business Media, 2006. 7

[72] Jonathan T Barron and Ben Poole. The fast bilateral solver. In *European conference on computer vision*, pages 617–632. Springer, 2016. 7

[73] Barak A Pearlmutter and Jeffrey Mark Siskind. Reverse-mode ad in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2):1–36, 2008. 7

[74] Chongjie Zhang and Victor Lesser. Multi-agent learning with policy prediction. In *Twenty-fourth AAAI conference on artificial intelligence*, 2010.

[75] Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In *International conference on machine learning*, pages 2113–2122. PMLR, 2015.

[76] David Belanger and Andrew McCallum. Structured prediction energy networks. In *International Conference on Machine Learning*, pages 983–992, 2016.

[77] Luke Metz, Ben Poole, David Pfau, and Jascha Sohl-Dickstein. Unrolled generative adversarial networks. *CoRR*, abs/1611.02163, 2016. URL http://arxiv.org/abs/1611.02163.

[78] Tian Han, Yang Lu, Song-Chun Zhu, and Ying Nian Wu. Alternating back-propagation for generator network. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[79] David Belanger, Bishan Yang, and Andrew McCallum. End-to-end learning for structured prediction energy networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 429–439. JMLR. org, 2017.

[80] David Belanger. *Deep energy-based models for structured prediction*. PhD thesis, University of Massachusetts Amherst, 2017.

[81] Jakob N Foerster, Richard Y Chen, Maruan Al-Shedivat, Shimon Whiteson, Pieter Abbeel, and Igor Mordatch. Learning with opponent-learning awareness. *arXiv preprint arXiv:1709.04326*, 2017.

[82] Vishal Monga, Yuelong Li, and Yonina C Eldar. Algorithm unrolling: Interpretable, efficient deep learning for signal and image processing. *IEEE Signal Processing Magazine*, 38(2): 18–44, 2021. 7

[83] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990. 7

[84] Herbert Jaeger. *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the" echo state network" approach*, volume 5. GMD-Forschungszentrum Informationstechnik Bonn, 2002. 7

[85] Corentin Tallec and Yann Ollivier. Unbiasing truncated backpropagation through time. *arXiv preprint arXiv:1705.08209*, 2017. 7

[86] Yuhuai Wu, Mengye Ren, Renjie Liao, and Roger Grosse. Understanding short-horizon bias in stochastic meta-optimization. *arXiv preprint arXiv:1803.02021*, 2018.

[87] Renjie Liao, Yuwen Xiong, Ethan Fetaya, Lisa Zhang, KiJung Yoon, Xaq Pitkow, Raquel Urtasun, and Richard Zemel. Reviving and improving recurrent back-propagation. In *International Conference on Machine Learning*, pages 3082–3091. PMLR, 2018.

[88] Amirreza Shaban, Ching-An Cheng, Nathan Hatch, and Byron Boots. Truncated back-propagation for bilevel optimization. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 1723–1732. PMLR, 2019.

[89] Paul Vicol, Luke Metz, and Jascha Sohl-Dickstein. Unbiased gradient estimation in unrolled computation graphs with persistent evolution strategies. In *International Conference on Machine Learning*, pages 10553–10563. PMLR, 2021. 7

[90] Brandon Amos. *Differentiable Optimization-Based Modeling for Machine Learning*. PhD thesis, Carnegie Mellon University, May 2019. 7

[91] Justin Domke. Generic methods for optimization-based modeling. In *Artificial Intelligence and Statistics*, pages 318–326. PMLR, 2012.

[92] Stephen Gould, Basura Fernando, Anoop Cherian, Peter Anderson, Rodrigo Santa Cruz, and Edison Guo. On differentiating parameterized argmin and argmax problems with application to bi-level optimization. *arXiv preprint arXiv:1607.05447*, 2016.

[93] Jonathan Lorraine, Paul Vicol, and David Duvenaud. Optimizing millions of hyperparameters by implicit differentiation. In *International Conference on Artificial Intelligence and Statistics*, pages 1540–1552. PMLR, 2020.

[94] Mathieu Blondel, Quentin Berthet, Marco Cuturi, Roy Frostig, Stephan Hoyer, Felipe Llinares-López, Fabian Pedregosa, and Jean-Philippe Vert. Efficient and modular implicit differentiation. *arXiv preprint arXiv:2105.15183*, 2021.

[95] Matan Atzmon, Niv Haim, Lior Yariv, Ofer Israelov, Haggai Maron, and Yaron Lipman. Controlling neural level sets. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/b20bb95ab626d93fd976af958fbc61ba-Paper.pdf. 7

[96] Marin Vlastelica, Anselm Paulus, Vít Musil, Georg Martius, and Michal Rolínek. Differentiation of blackbox combinatorial solvers. *arXiv preprint arXiv:1912.02175*, 2019. 8

[97] Mathias Niepert, Pasquale Minervini, and Luca Franceschi. Implicit mle: Backpropagating through discrete exponential family distributions. *Advances in Neural Information Processing Systems*, 34, 2021. 8

[98] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2. 10

[99] Paloma Sodhi, Sanjiban Choudhury, Joshua G. Mangelson, and Michael Kaess. ICS: Incremental Constrained Smoothing for State Estimation, 2020. 10

[100] André F.T. Martins, Mário A.T. Figueiredo, Pedro M.Q. Aguiar, Noah A. Smith, and Eric P. Xing. An augmented Lagrangian approach to constrained MAP inference. *Proceedings of the 28th International Conference on Machine Learning, ICML 2011*, pages 169–176, 2011. 10

[101] Richard Zou Horace He. functorch: Jax-like composable function transforms for pytorch. https://github.com/pytorch/functorch, 2021. 10, 18, 19

[102] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 19

[103] Gregory S Chirikjian. *Stochastic models, information theory, and Lie groups, volume 2: Analytic methods and modern applications*, volume 2. Springer Science & Business Media, 2011. 20

[104] Taosha Fan, Hanlin Wang, Michael Rubenstein, and Todd Murphey. Cpl-slam: Efficient and certifiably correct planar graph-based slam using the complex number representation. *IEEE Transactions on Robotics*, 36(6):1719–1737, 2020. 20

[105] Taosha Fan and Todd Murphey. Majorization minimization methods for distributed pose graph optimization. *arXiv preprint arXiv:2108.00083*, 2021. 20

[106] Daniel Martinec and Tomas Pajdla. Robust rotation and translation estimation in multiview reconstruction. In *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2007. 20

[107] Amit Singer and Yoel Shkolnisky. Three-dimensional structure determination from common lines in cryo-em by eigenvectors and semidefinite programming. *SIAM journal on imaging sciences*, 4(2):543–572, 2011. 20

[108] Mihai Cucuringu, Yaron Lipman, and Amit Singer. Sensor network localization by eigenvector synchronization over the euclidean group. *ACM Transactions on Sensor Networks (TOSN)*, 8 (3):1–42, 2012. 20

[109] Luca Carlone, Roberto Tron, Kostas Daniilidis, and Frank Dellaert. Initialization techniques for 3d slam: A survey on rotation estimation and its use in pose graph optimization. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4597–4604, 2015. 20

[110] Jiaji Zhou, James A. Bagnell, and Matthew T. Mason. A fast stochastic contact model for planar pushing and grasping: Theory and experimental validation. In Nancy M. Amato, Siddhartha S. Srinivasa, Nora Ayanian, and Scott Kuindersma, editors, *Robotics: Science and Systems XIII, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, July 12-16, 2017*, 2017. doi: 10.15607/RSS.2017.XIII.040. URL http://www.roboticsproceedings.org/rss13/p40.html. 21

[111] Paloma Sodhi, Eric Dexheimer, Mustafa Mukadam, Stuart Anderson, and Michael Kaess. Leo: Learning energy-based models in factor graph optimization. In *Conference on Robot Learning*, pages 234–244. PMLR, 2022. 21

[112] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'81, page 674–679, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc. 22

[113] Simon Baker and Iain A. Matthews. Lucas-kanade 20 years on: A unifying framework. *International Journal of Computer Vision*, 56(3):221–255, 2004. URL https://doi.org/10.1023/B:VISI.0000011205.11775.fd. 22

[114] Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. Deep image homography estimation. *RSS 2016 Workshop: Limits and Potentials of Deep Learning in Robotics*, 2016. URL http://arxiv.org/abs/1606.03798. 23

[115] Ulisse Dini. *Analisi infinitesimale*. Lithografia Gorani, 1878. 26

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [No] We do not foresee any beyond what the field of differentiable optimization already have.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [Yes]
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

Appendix

A Simple example description

In this section, we describe the example in Listing 1 in more detail. The example considers fitting the curve $y = ve^x$ to a dataset of N observations $(x, y) \sim \mathcal{D}$. A standard way to solve this is to minimize the least squares objective Eq. (1) with residuals $r_i(\hat{v}) := y^{(i)} - \hat{v}e^{x^{(i)}}$, for $i = 1, ..., N$, and where $\theta := \hat{v}$. We can model this in `Theseus` with a single `CostFunction` that computes the N-dimensional vector $R(\hat{v})$ of all residuals as a function of a single optimization `Variable` $\hat{v}$ and two auxiliary variables, x and y.

The code implementing this problem starts by creating uniquely named `Variable` containers in lines 3-5. We then create an objective with the cost function (lines 11-15). We use a `CostFunction` of type `AutoDiffCostFunction` (line 12), which relies on `torch.autograd` and vectorization via `functorch` [101] to automatically compute the residual Jacobians used by the inner optimizer (see App. B). `AutoDiffCostFunction` requires providing an error function that receives optimization

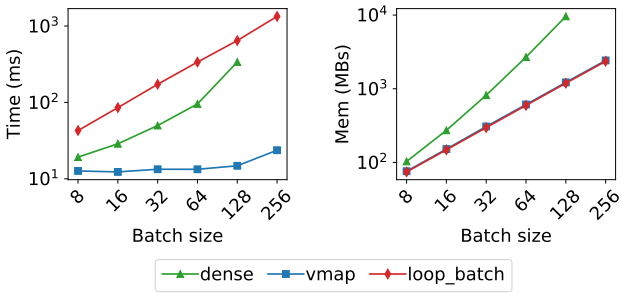


Figure 6: `AutodiffCostFunction` time (left) and memory (right) consumption in the homography estimation example for different autograd modes and batch sizes.

variables and auxiliary variables (defined in lines 7-9), and returns `torch` tensors computing the (unweighted) residual. Although not required by this problem, we also illustrate how to add a cost weight to the residuals by including a `ScaleCostWeight`, which simply scales all residuals in this cost function by a scalar (1.0 in this case). Finally, we encapsulate the objective and a Gauss-Newton optimizer into a differentiable `TheseusLayer` in line 16.

To illustrate how to differentiate through this layer, we perturb the x values in the dataset so that it becomes impossible to recover the correct value of v from NLS optimization alone, and then define $\phi := x$ and $L(\theta^*(\phi)) := (\hat{v}^*(x) - v)^2$. Thus, the outer loop optimization corrects the x tensor so that the solution of the inner loop matches the true value of v.

In the code, the outer parameter is defined in line 18, with initial value for ϕ set to a perturbed function of the true x, while the outer loss is defined in line 24. In lines 21-23 we solve the NLS problem, by calling `forward()` with the current value of ϕ as the value to set for auxiliary variable named "x", and an initial value $v = 1$ for the optimization variable named "v" (with a required batch dimension in the input); we also set the backward mode to `Implicit`. The optimum found can be recovered as a tensor by getting key "v" of the dictionary returned by `forward`, which we then use to compute the outer loss in line 24. Finally, outer loop optimization is done via `torch`'s well-known autograd engine, in lines 25-26, using the Adam optimizer [102] for ϕ, defined in line 19.

B `AutodiffCostFunction` autograd modes

In this section, we evaluate the effect of the three different autograd modes we provide for automatically computing the jacobians for `AutodiffCostFunction`:

- **dense** uses `torch.autograd.functional.jacobian`, which computes a dense jacobian that includes cross-batch derivatives; i.e., the derivative of j-th batch output with respect to variables in the i-th batch input. Since we only need per-sample gradients, we slice the result of this operation.
- **loop_batch** is also based on `torch.autograd.functional.jacobian`, but we manually loop over the batch before each call so that we obtain only per-sample gradients.
- **vmap** mode uses `functorch.vmap` [101] to compute per-sample gradients in a vectorized manner.

For evaluation, we use the homography example described in App. D.5, which uses `AutodiffCostFunction` and can have significant memory requirements when computing the jacobians. Fig. 6 illustrates the advantages of using **vmap** over the other two modes, both in terms of compute time and memory. The **loop_batch** mode has similar memory requirements to **vmap**, but the compute time is significantly slower than the other two methods. Finally, **dense** mode has substantially more memory requirements than the two other methods (up to an order of magnitude higher), and runs out of memory for the largest batch size 256 used in this experiment. For a batch size of 128, **vmap** results in a speedup of 22x over the next best method ($\sim$ 15ms vs. $\sim$ 337ms for **dense**), and almost 8x less memory ($\sim$ 1.2GBs vs. $\sim$ 9.6GBs).

C Differentiable Lie group details

While differentiation on the Euclidean space is straightforward, it remains challenging to do so on the non-Euclidean manifolds. In this section, we provide details about how to compute the derivatives

	Objective Value					
	Sphere	Torus	Cubicle	Rim	Grid	Garage
Initial	8.437×10^2	1.234×10^4	1.622×10^6	1.924×10^7	4.365×10^4	7.108×10^{-1}
Final	6.805×10^2	1.212×10^4	1.455×10^3	4.157×10^4	4.218×10^4	6.342×10^{-1}

Table 1: Initial and final objective values of `Theseus` on 3D benchmark datasets with PGO example.

on the tangent space of Lie groups using the projection operator [58]. The implementation of the projection operator is essential for automatic differentiation on the tangent space of Lie groups.

Suppose $F(g)$ is a function of $g \in G$ where G is a matrix Lie group and $\tau(\xi)$ a retraction map of G. For notational simplicity, let $\nabla_g F(g)$ denote the Euclidean gradient of $F(g)$ and $T_e G$ the Lie algebra of G. Following [103], the gradient on the tangent space of Lie group is a linear operator $\mathrm{D}_g F(g)$ such that

$$\mathrm{D}_g F(g) \cdot \xi = \left.\frac{\partial}{\partial s}\right|_{s=0} F\big(g\tau(s \cdot \xi)\big) \tag{5}$$

holds for any Lie algebra elements $\xi \in T_e G$. As a result of the chain rule, the right-hand side of the equation above is equivalent to

$$\left.\frac{\partial}{\partial s}\right|_{s=0} F\big(g\tau(s \cdot \xi)\big) = \nabla_g F(g) \cdot \left.\frac{\partial}{\partial s}\right|_{s=0} g\tau(s \cdot \xi) = \nabla_g F(g) \cdot g\xi \tag{6}$$

where the last equality results from properties of the retraction map. Then, we conclude from Eqs. (5) and (6) that

$$\mathrm{D}_g F(g) \cdot \xi = \nabla_g F(g) \cdot g\xi. \tag{7}$$

Therefore, there exists a projection operator $\mathrm{proj}_g(\cdot)$ such that

$$\mathrm{D}_g F(g) = \mathrm{proj}_g\big(\nabla_g F(g)\big) \tag{8}$$

for any gradients on the tangent space of Lie group and their corresponding Euclidean gradients [58]. Furthermore, note that the projection operator $\mathrm{proj}_g(\cdot)$ is a linear operator depending on $g \in G$ and can be computed in closed form.

D Example application details

D.1 Pose graph optimization

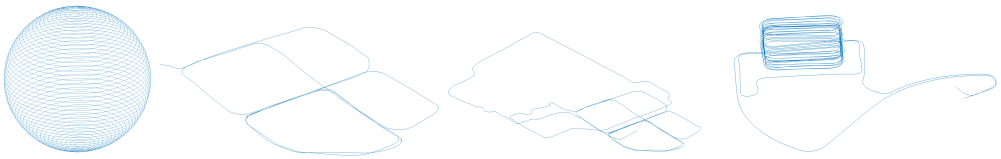
Pose graph optimization (PGO) [60, 104, 105] is the problem of recovering unknown poses of SE(2) and SE(3) from the noisy relative pose measurements. Pose graph optimization has extensive applications in robotics [35], computer vision [106], computational biology [107], sensor networks [108], etc. In pose graph optimization, we represent unknown poses as vertices and relative pose measurements as edges. Then, it is possible to compute the relative pose errors for each pair of neighboring vertices such that a nonlinear least-squares optimization problem can be formulated for pose estimation. A more detailed introduction to pose graph optimization can be found in [35, 60, 104, 105].

`Theseus` includes a differentiable and coordinate-independent version of the relative pose errors with which it is straightforward to solve pose graph optimization. We evaluated `Theseus` on the simulated Cube dataset and a number of benchmark datasets for pose graph optimization [35, 60]. The Cube dataset simulates the 3D odometry of a robot with varying numbers of poses, loop closure probabilities, and loop closure outlier ratios, which is used to profile the time and space complexities of the forward and backward passes in `Theseus`. Furthermore, the benchmark datasets indicate that `Theseus` is capable of solving large-scale differentiable nonlinear optimization problems with comparable accuracy and efficiency to existing state-of-the-art solver like `Ceres` [37].

`Theseus` and `Ceres` attain the same objective values for all the evaluated benchmark datasets [35, 60] using the chordal initialization [109]. The inital/final objective values and qualitative results for some benchmark datasets are shown in Table 1 and Fig. 7, respectively.

D.2 Tactile state estimation

Recent work [28] explored the use of NLS optimization with learned tactile sensor observations for tactile pose estimation. The goal is to incrementally estimate sequences of object poses that



(a) Sphere (b) Cubicle (c) Rim (d) Garage

Figure 7: Qualitative results of Theseus on 3D benchmark datasets with PGO example.

are moved by a robotic hand equipped with a DIGIT tactile tensor [61]. The key insight of Sodhi et al. [28] is to use learning to transform high dimensional tactile observations into relative poses between measurement pairs. Once relative poses are available, the sensor data can be integrated into an optimization problem that solves for object poses. The objective includes four types of cost functions. One penalizes inconsistencies with the measurement coming from the learned observation model. A second one encourages the predicted poses to be consistent with a quasi-static physics model [110]. A third type adds geometric constraints by penalizing intersections between the end effector and the object using a signed distance field. Finally, a fourth cost function incorporates pose priors from a camera.

In Theseus, we implement an offline and differentiable version of the tactile state estimation problem above, using a dataset of 63 trajectories of length 25 with known ground truth poses provided by the authors of [28]; we used 56 of these as training set and the other 7 as a test set. Optimization variables are object and end effector poses (modeled as SE(2) groups) for each point in the trajectory, and the outer loss objective is the difference between the optimized object poses and the ground truth in the dataset. The learnable component corresponds to the relative pose model, using a pre-trained encoder, and finetuning the final layer via end-to-end learning through the inner loop optimization. This approach is similar to how the tactile measurements model was trained in [111], with the two main differences being that we do not use an energy-based formulation and instead directly differentiate through the inner optimizer, and we also do not consider an incremental setting. Fig. 8 (left) shows an example of estimated trajectories before and after learning.

D.3 Bundle adjustment

Bundle adjustment is the problem of optimizing a 3D reconstruction formed by a set of camera images and a set of landmarks observed and matched across the images. In every camera image a 2D coordinate is identified for the position of all observed landmarks, and the problem is initialized with an estimate of the positions of the landmarks and the camera poses. We call *reprojection error* the image-offset between where the landmark was detected on the image, and the reprojection of the landmark according to the current parameter estimation. The optimization problem consists of simultaneously tweaking the cameras poses and landmark positions, while minimizing the square-sum of all the reprojection errors; see [8, 62] for in depth exploration on bundle adjustment and its state-of-the-art.

We provide a bundle adjustment application example in Theseus, adopting the same data format of [62], with functions to generate synthetic dataset, as well as load/save open source datasets. To test bundle adjustment in a differentiable optimization setting, we add soft-kernels to the reprojection errors and setup as outer loop parameter the radius of the soft-kernel, which represents the confidence radius for reprojection errors with respect to possible outlier observations. We use as outer loss the average frame pose error from a ground truth value, such that the outer loop's task is to set the radius to a value that will make the bundle adjustment problem set the ideal soft loss radius value.

D.4 Motion planning

NLS optimization can also be used for motion planning in robotics [5], where the objective variables are robot poses and velocities on a set of discrete time steps. Cost functions include terms representing smoothness constraints modeling forward kinematics, collision avoidance penalties, and boundary conditions on start and goal states. An end-to-end differentiable version of this formulation was proposed by Bhardwaj et al. [20], where a neural model predicts state-dependent cost weights for each step in the path, and the outer loss encourages the inner loop optimization to produce paths matching an expert in a dataset of trajectories. As part of Theseus, we include differentiable versions of cost functions like smoothness and collision in [5], and an example of how to setup end-to-end differentiation for optimization variable initialization. That is, the model proposes initial trajectories

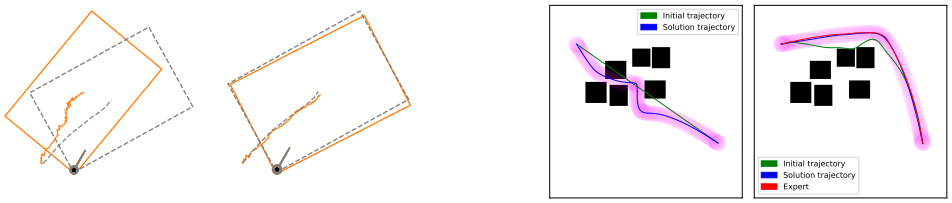


Figure 8: Examples of differentiable tactile state estimation and differentiable motion planning with Theseus. **Left:** Pose estimates before and after learning in the tactile state estimation example. Grey color indicates ground truth, and orange the estimate. The plot shows the trajectory as a curve, and the rectangle indicates the last object pose in the trajectory. **Right:** Trajectories generated by the planner before and after learning. The learned model generates initial trajectories for the optimizer to match an expert with only a few optimization steps.

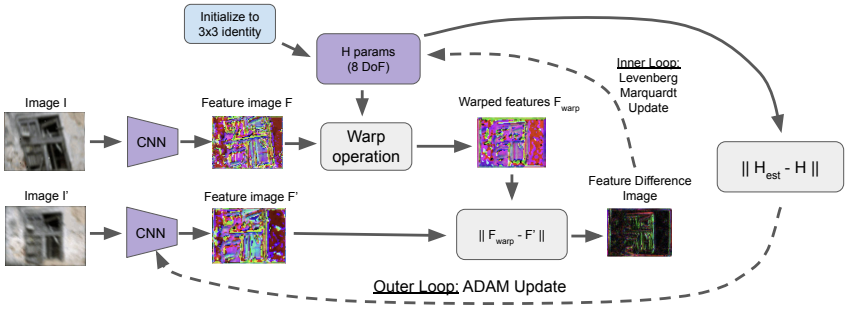


Figure 9: **Learning Robust Image Features for Homography Estimation with Theseus.** An inner loop optimization problem aligns two images via a feature-metric Levenberg-Marquardt optimization using features computed from a CNN. The outer loop uses Adam to update the weights the of the CNN that best minimize the final homography error.

for the optimizer, and the outer loss is set to a weighted sum of two terms, one computing closeness to the expert trajectory, and another equal to the inner loop's objective after only 2 iterations. This loss encourages the model to produce high-quality "proposals" that converge to good quality solutions quickly; an example of before/after training initial trajectories is illustrated in Fig. 8.

D.5 Homography estimation

A homography, also known as a projective transformation, is a linear mapping between a 2D point in one image x to a point in another image x', defined by a 3×3 matrix H, written as $x \sim Hx'$, where $\sim$ defines the equivalence up to scale. In addition to representing linear transformations across 2D images, the homography is also a valid approximation of the motion of 2D points observed from camera images in 3D scenes in certain scenarios such as (1) rotation-only motion between cameras (2) when the scene is planar and (3) when the scene structure is far from the camera.

One approach to solving for the parameters of the homography is through iterative optimization via dense alignment of RGB image pixels in the image through fast second order optimization methods, as is done in Lucas-Kanade optical flow algorithm [112, 113]. This approach is also known as photo-metric alignment. Though this technique performs well in many scenarios, photo-metric alignment struggles when the lighting in the scene changes significantly, because it assumes that the brightness of a pixel is constant across different views. Feature-metric optimization is an extension to photo-metric optimization that works by first passing the image I through a feature extractor function $f(\cdot)$, such as a convolutional neural network parameterized by weights w, $F = f(I; w)$. This function generates a feature map $F \in \mathbb{R}^{C \times H \times W}$, where C is a high dimensional channel number like 32, and H and W represent the image height and width respectively. In feature-metric alignment, the alignment is done at the feature map level, rather than the RGB image level.

One important question when designing a feature-metric optimization algorithm, is how to obtain the weights w that define the feature extractor. One approach used in works such as [63] uses an off-the-shelf CNN which has been trained for image classification. One benefit of using Theseus for such as task is that the learning problem can be written without deriving analytical gradients, making it much easier to rapidly prototype and explore various formulations. In our example, we demonstrate a use-case of Theseus by performing end-to-end training of a two-layer CNN using

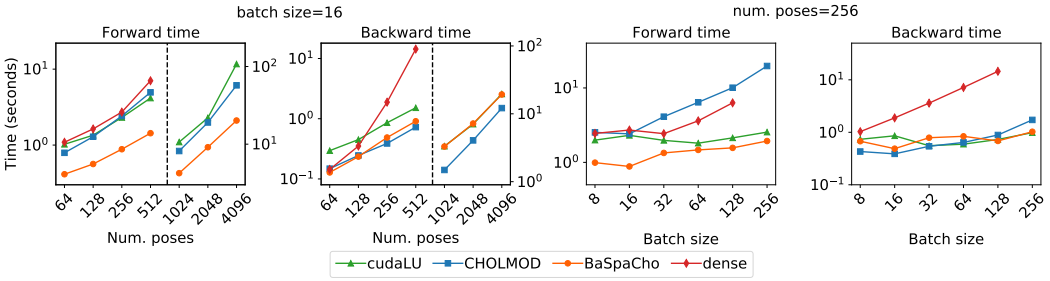


Figure 10: Forward/backward times of Theseus with sparse and dense solvers on different PGO problem scales.

gradients obtained through the homography optimization. A high level diagram of this learning problem is presented in Fig. 9. We optimize a dense feature-metric mean-squared error term in the inner loop and a four-corner homography error in the outer loop. The four-corner error is a simple measure that computes the L2 distance of four corners of the image after being transformed by the estimated and ground truth homography, as is used in [114] as the output parameterization.

E BaSpaCho: Batched Sparse Cholesky

In this section, we provide more details for our open-source novel BaSpaCho solver (https://github.com/facebookresearch/baspacho). BaSpaCho implements the *supernodal* Cholesky algorithm [70] to achieve state-of-the art performance by exploiting dense operations via BLAS/cuBLAS. The heuristics for clustering in the supernodal algorithm evaluate the trade-offs of fragmentation in sparse matrices against denser matrices with explicit zero-fill. These heuristics use a computation model that takes into account the architecture (batched/CPU/GPU) that can impact preference towards sparser or denser operations, and allows for further fine-tuning and customization. Apart from a minimal memory allocation needed for the symbolic factorization, BaSpaCho does not own any allocated memory allowing the user to fully manage memory arrays. This allow us to temporarily offload GPU arrays representing factorized matrix data to the CPU when necessary. Unlike existing solvers, BaSpaCho exposes lightweight random accessors that allows the user to read and write matrix blocks in the numeric factor data. This facilitates easy bookkeeping needed by optimization methods which often re-implement block-sparse matrix structures and convert between different matrix formats in order to invoke sparse solvers like CHOLMOD.

F Benchmark details and additional results

In this section, we present more profiling results for forward and backward pass of Theseus, using the same setup as Sec. 5.1. For evaluation, we used the cube datasets of PGO (see App. D.1) with different numbers of poses, batch sizes, and levels of sparsity. In addition to the forward and backward times as a function of the numbers of poses and batch sizes reported in Sec. 5.1, we further report and analyze the memory usage of Theseus with different solvers (cudaLU, CHOLMOD, BaSpaCho and dense) in various settings.

F.1 Forward and backward times with smaller batch size and number of poses

We profile PGO using different linear solvers (cudaLU, CHOLMOD, BaSpaCho and dense) for fixed batch size of 16 and number of poses of 256. The setup is the same as that in Sec. 5.1 except that fixed batch size and number of poses are smaller. Fig. 10 shows the average time of a full forward and backward pass. Similar to Fig. 3 with larger fixed batch size and number of poses, the sparse solvers are faster than dense. For the smallest problem considered (64 poses, 16 batch size), the total sum of average forward and backward times are 1.32s (cudaLU), 0.94s (CHOLMOD), 0.54s (BaSpaCho), 1.23s (dense) per batch. Increasing to 128 poses makes the sparse solvers noticeably faster than dense: 1.98s (dense) vs 1.80s (cudaLU), 1.53s (CHOLMOD), and 0.80s (BaSpaCho). As the problem scale increases, the gap between the sparse and dense solvers widens: for the largest problem solvable with dense (512 poses, 16 batch size) the average total times are 5.68s (cudaLU), 5.67s (CHOLMOD), 2.34s (BaSpaCho), and 21.37s (dense). The speedup over dense is ∼3.7x for cudaLU and CHOLMOD, and ∼9.1x for BaSpaCho.

F.2 Forward and backward times with respect to sparsity

We study the forward and backward pass times of Theseus with sparse and dense solvers for different levels of sparsity using the synthetic Cube dataset. In PGO, loop closure probability represents how

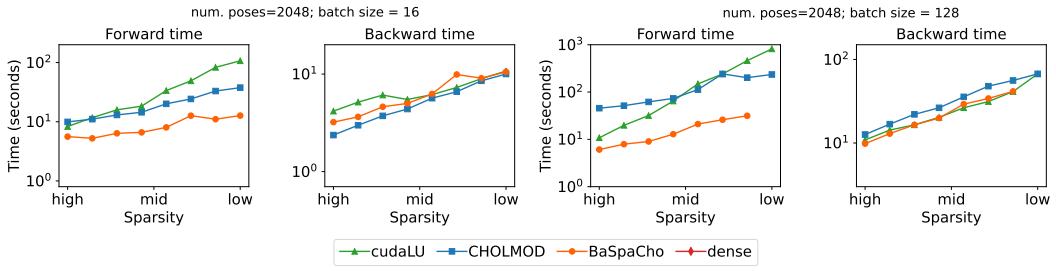


Figure 11: Forward and backward times of Theseus with sparse and dense solvers on PGO problems with 2048 poses and different levels of sparsity.

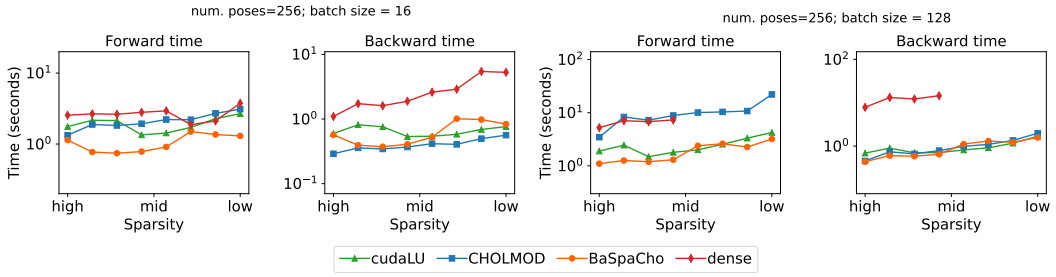


Figure 12: Forward and backward times of Theseus with sparse and dense solvers on PGO problems with 256 poses and different levels of sparsity.

likely a pose has a loop closure edge connected to the other poses, and thus, greater loop closure probability yields a less sparse Hessian. We use loop closure probabilities from 0.05 to 0.40 in increments of 0.05, to indicate the level of sparsity for Cube datasets from high (0.05) to low (0.40).

The average forward and backward times of Theseus on PGO problems with different levels of sparsity for fixed numbers of poses are shown in Fig. 11 for 2048 poses and in Fig. 12 for 256 poses. In both figures, the left two plots are with 16 batch size and the right two plots are with 128 batch size. As expected, it takes more time in most cases for PGO problems with lower sparsity. Since dense does not exploit the sparsity of optimization problems when solving the linear systems, forward pass of dense takes almost the same amount of time regardless of the levels of sparsity. There is still some overhead for dense as sparsity decreases, because more loop closure edges implies more cost function terms in the objective, so putting together the approximate Hessian is computationally more expensive.

F.3 Scalability of Theseus

In addition to forward and backward times in Sec. 5.1 and Apps. F.1 and F.2, we analyze the scalability of Theseus with different linear solvers (cudaLU, CHOLMOD, BaSpaCho and dense) following a similar setup to evaluation in Figs. 3 and 10.

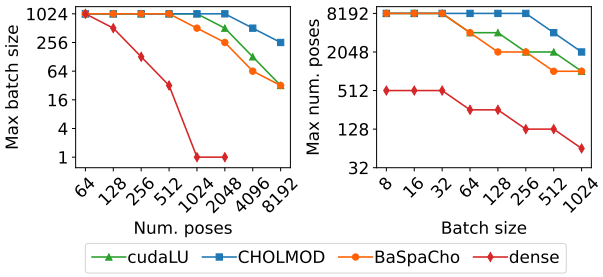


Figure 13: Largest PGO problems Theseus scales to for different numbers of poses and batch sizes.

We profile PGO with various numbers of poses from 64 to 8192 and batch sizes from 8 to 1024 in increments of power of 2. Fig. 13 shows the maximum batch sizes solvable for given numbers of poses (left) and the maximum numbers of poses solvable for given batch sizes (right). In Fig. 13, it can be seen that dense only scales to small PGO problems due to the memory limitation and fails to solve any PGO problems with 4096 poses or more, even with batch size of 1. In contrast, cudaLU, CHOLMOD and BaSpaCho successfully solve PGO problems with 8192 poses for a batch size of 32 (cudaLU, BaSpaCho) and 256 (CHOLMOD). As discussed in Sec. 5.1, cudaLU and BaSpaCho require extra GPU memory to solve linear systems, whereas CHOLMOD has all computation run on CPU, and thus can solve larger DNLS problems than cudaLU and BaSpaCho.

F.4 Comparison with Ceres on smaller batch size and number of poses

In addition to Sec. 5.2, we follow the same setup to compare Theseus as a stand-alone NLS optimizer with Ceres for PGO problems with a smaller fixed batch size of 16 and number of poses of 256. Fig. 14 shows the speedup of Theseus compared to Ceres (black dotted line). Similar to Fig. 4, Ceres is faster for PGO problems for small batch sizes and numbers of poses, and Theseus is faster as the problem scale increases.

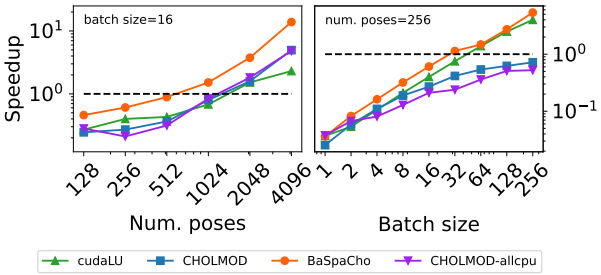


Figure 14: Speedup of Theseus (forward pass) over Ceres (black dashed) on different PGO problem scales.

G Backward mode details, additional results, and derivations

G.1 Experimental details

In Sec. 5.3 we use the tactile state estimation example to evaluate the performance of different backward modes. As mentioned in App. D.2, the dataset consists of 63 trajectories of length 25, 56 of which we use for training and 7 for test. We use a batch size of 8 and train for 100 epochs, resulting in 700 batches for averaging time and memory results. For the inner loop, we use Gauss-Newton with a step size of 0.05; in the test set we run the inner loop for 50 iterations, regardless of the number used during training. For the outer loop, we use the Adam optimizer with a learning rate of 10^{-4}, decayed exponentially by a factor of 0.98 after every epoch.

G.2 Additional results

Fig. 15 (left) shows the peak memory consumption during the forward pass. We observe the same trend from the backward pass (Fig. 5, center right), where Unroll's memory consumption increases linearly with the number of inner loop iterations, while for the other methods it remains constant. Implicit has the lowest peak memory requirement (∼22MBs), followed by DLM (∼29MBs).

Fig. 15 (center, right) also shows training curves for all methods. We observe that, despite higher performance in the test set, Implicit is more unstable during training and oscillates between low and high values; this suggests that careful use of early stopping and hyperparameter tuning might be required when using Implicit. The other methods are more stable, with the two truncated methods achieving the lowest training loss after Implicit. Fig. 15 (right) shows that Unroll's performance degradation, relative to other methods, with increasing number of inner loop iterations (also see Fig. 5, right) is not just a generalization issue, but also happens during training. This suggest possible numerical issues from unrolling gradients over a high number of optimization steps, as observed in prior work.

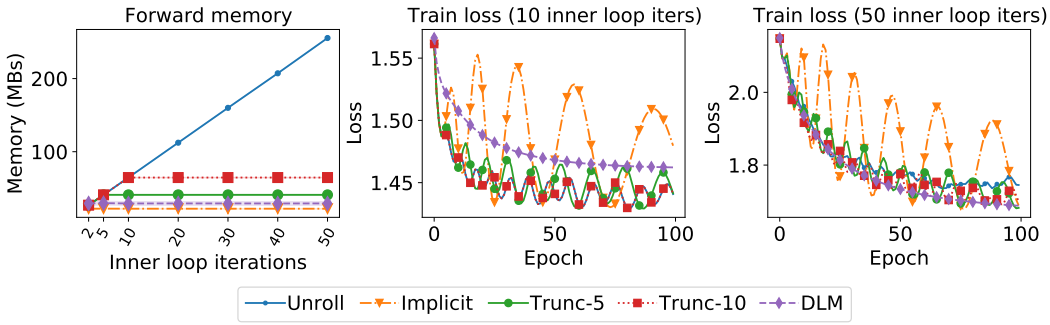


Figure 15: Additional results for backward modes comparison in tactile state estimation problem. **Left:** Memory consumption in forward pass. **Center:** Training loss when using 10 inner loop iterations. **Right:** Training loss when using 50 inner loop iterations.

G.3 Backward modes summary and limitations

Fig. 16 visualizes the backward modes and Table 2 contrasts their limitations. The table shows that all four modes can be used when learning parameters for cost functions or cost weights. However,

	Cannot be used for learning c_i or w_i	Cannot be used for learning θ_{init}	Requires accurate θ^* solution	Possible vanishing or exploding gradients	Requires tuning	Compute and memory usage
`Unroll`				✓		high
`Trunc`				✓	✓	medium
`Implicit`		✓	✓			low
`DLM`					✓	low

Table 2: Backward modes summary and limitations.

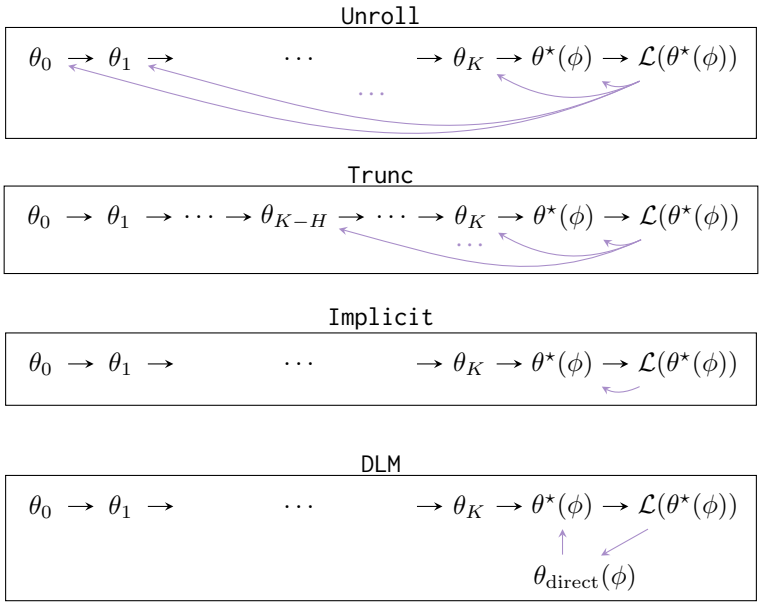


Figure 16: Illustration of the dependencies of the backward modes for computing $\nabla_\phi \mathcal{L}(\theta^\star)$.

unlike other approaches, `Implicit` cannot be used when learning initial values for the optimization variables, θ_{init}. Another limitation of `Implicit` is that the resulting gradients might be inaccurate in problems where it is not feasible to find the optimal solution to the inner optimization problem; other methods don't experience this limitation, since they compute gradients around the approximate solution found. On the other hand, both `Unroll` and `Trunc` could potentially experience vanishing or exploding gradient issues when the number of iterations to backpropagate through is large, a limitation that is not shared by `Implicit` and `DLM`. Finally, a limitation of `DLM` is that ϵ needs to be tuned (see Eq. (16)), which can greatly affect performance. Likewise, the number of backward iterations for `Trunc` may also require some tuning.

G.4 Derivations for backward modes

G.4.1 Implicit function theorem

For adjoint differentiation, we make use of the implicit function theorem, which is originally from Dini [115], and presented in Dontchev and Rockafellar [39, Theorem 1B.1] as:

Theorem 1 (Dini's implicit function theorem)**.** *Let the roots of $g(\theta;\phi)$ define an implicit mapping $\Theta^\star(\phi)$ given by $\Theta^\star(\phi) := \{\theta \mid g(\theta;\phi) = 0\}$, where $\theta \in \mathbb{R}^m$, $\phi \in \mathbb{R}^n$, and $g : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^m$. Let g be continuously differentiable in a neighborhood of $(\bar\theta, \bar\phi)$ such that $g(\bar\theta;\bar\phi) = 0$, and let the Jacobian of g with respect to θ at $(\bar\theta, \bar\phi)$, i.e. $\mathrm{D}_\theta g(\bar\theta;\bar\phi)$, be non-singular. Then $\Theta^\star$ has a single-valued localization $\theta^\star$ around $\bar\phi$ for $\bar\theta$ which is continuously differentiable in a neighborhood Q of $\bar\phi$ with Jacobian satisfying*

$$\mathrm{D}_\phi \theta^\star(\tilde\phi) = -\mathrm{D}_\theta^{-1} g(\theta^\star(\tilde\phi);\tilde\phi)\mathrm{D}_\phi g(\theta^\star(\tilde\phi);\tilde\phi) \qquad \text{for every } \tilde\phi \in Q. \tag{9}$$

G.4.2 Proof of Prop. 1

Proof. Let $\bar{\phi}$ be a hyper-parameter resulting in a unique $\theta^\star(\bar{\phi})$ and recall $g(\theta;\phi) := \nabla_\theta S(\theta,\phi)$ with $g(\theta^\star(\bar{\phi}),\bar{\phi}) = 0$. Applying the product rule to differentiate the Newton iteration h at $(\theta^\star(\bar{\phi}),\bar{\phi})$ gives:

$$\begin{aligned} \mathrm{D}_\phi h(\theta^\star(\bar{\phi});\bar{\phi}) &= \mathrm{D}_\phi \left[-\left(\nabla_\theta^2 S(\theta^\star(\bar{\phi});\bar{\phi})\right)^{-1} \nabla_\theta S(\theta^\star(\bar{\phi});\bar{\phi}) \right] \\ &= -\left(\nabla_\theta^2 S(\theta^\star(\bar{\phi});\bar{\phi})\right)^{-1} \mathrm{D}_\phi \nabla_\theta S(\theta^\star(\bar{\phi});\bar{\phi}) \\ &= -\mathrm{D}_\theta^{-1} g(\theta^\star(\bar{\phi});\bar{\phi}) \mathrm{D}_\phi g(\theta^\star(\bar{\phi});\bar{\phi}) \end{aligned} \tag{10}$$

□

G.4.3 Direct loss minimization for use in Theseus

Originally, DLM gradient for non-linear objective functions [41] can be expressed as

$$\nabla_\phi L = \lim_{\varepsilon \to 0} g_{\text{DLM}}^\varepsilon, \quad \text{where} \quad g_{\text{DLM}}^\varepsilon \triangleq \frac{1}{\varepsilon} \left[\frac{\partial}{\partial \phi} S(\theta^*;\phi) - \frac{\partial}{\partial \phi} S(\theta_{\text{direct}};\phi) \right] \tag{11}$$

where

$$\theta^* = \arg\min_{\hat{\theta}} S(\hat{\theta};\phi), \qquad \text{and} \qquad \theta_{\text{direct}} = \arg\min_{\hat{\theta}} S(\hat{\theta};\phi) - \varepsilon L(\hat{\theta}). \tag{12}$$

However, this dependence on the loss function fits poorly in a reverse-mode automatic differentiation framework like PyTorch. Instead, we can construct an equivalent formulation by noting that in continuous space, we can first linearize the loss function around the current solution θ^*,

$$\hat{L}(\theta) = L(\theta^*) + \nabla_\theta L(\theta^*)(\theta - \theta^*) \tag{13}$$

Let $v = \nabla_\theta L(\theta^*)$, then the perturbed solution becomes

$$\theta_{\text{direct}} = \arg\min_{\hat{\theta}} S(\hat{\theta};\phi) - \varepsilon \left(L(\theta^*) + v^T(\hat{\theta} - \theta^*) \right) = \arg\min_{\hat{\theta}} S(\hat{\theta};\phi) - \varepsilon v^T \hat{\theta}. \tag{14}$$

Plugging this back into Eq. (11), we see that this is an algorithm which takes in a gradient vector v and computes an approximation to the vector-Jacobian product $\nabla_\phi L(\theta^*) = v \frac{\partial \theta^*}{\partial \phi}$.

As Theseus is designed to solve optimization problems where S is expressed as sum of squares, it cannot readily handle solving θ_{direct} as this requires adding a linear term to the objective. Instead, let us consider the following "completing the square" approach:

$$\arg\min_{\hat{\theta}} \|\varepsilon\hat{\theta}\|^2 - \varepsilon v^T \theta = \arg\min_{\hat{\theta}} \varepsilon^2 \hat{\theta}^T \hat{\theta} - \varepsilon v^T \hat{\theta} + \left(\tfrac{1}{2}v\right)^T \left(\tfrac{1}{2}v\right) = \arg\min_{\hat{\theta}} \left\| \varepsilon\hat{\theta} - \tfrac{1}{2}v \right\|^2 \tag{15}$$

We can thus add this extra term and let

$$\theta_{\text{direct}} = \arg\min_{\hat{\theta}} S(\hat{\theta};\phi) + \left\| \varepsilon\hat{\theta} - \tfrac{1}{2}v \right\|^2 \tag{16}$$

This adds a small bias to the gradient due to the addition of $\|\varepsilon\hat{\theta}\|^2$ but when ε is small it shouldn't be problematic. In practice, we solve for θ_{direct} by starting from θ^* and using just one iteration of Gauss-Newton.