Tangent Space Backpropagation for 3D Transformation Groups

Zachary Teed and Jia Deng
Princeton University
{zteed,jiadeng}@cs.princeton.edu

Abstract

We address the problem of performing backpropagation for computation graphs involving 3D transformation groups $SO(3)$, $SE(3)$, and $Sim(3)$. 3D transformation groups are widely used in 3D vision and robotics, but they do not form vector spaces and instead lie on smooth manifolds. The standard backpropagation approach, which embeds 3D transformations in Euclidean spaces, suffers from numerical difficulties. We introduce a new library, which exploits the group structure of 3D transformations and performs backpropagation in the tangent spaces of manifolds. We show that our approach is numerically more stable, easier to implement, and beneficial to a diverse set of tasks. Our plug-and-play PyTorch library is available at https://github.com/princeton-vl/lietorch

1. Introduction

3D transformation groups—$SO(3)$, $SE(3)$, $Sim(3)$—have been extensively used in a wide range of computer vision and robotics problems. Important applications include SLAM, 6-dof pose estimation, multiview reconstruction, inverse kinematics, pose graph optimization, geometric registration, and scene flow. In these domains, the state of the system—configuration of robotic joints, camera poses, non-rigid deformations—can be naturally represented as 3D transformations.

Recently, many of these problems have been approached using deep learning, either in an end-to-end manner[31, 10, 40, 8, 38, 22, 18] or composed as hybrid systems[21, 11, 29, 19]. A key ingredient of deep learning is auto-differentiation, in particular, backpropagation through a computation graph. A variety of general-purpose deep learning libraries such as PyTorch [27] and TensorFlow [1] have been developed such that backpropagation is automatically performed by the library—users only need to specify the computation graph and supply any custom operations.

A basic assumption of existing deep learning libraries is that a computation graph represents a composition of mappings between Euclidean spaces. That is, each node of the graph represents a differentiable mappings between Euclidean spaces, e.g. from $\mathbb{R}^m$ to $\mathbb{R}^n$. This assumption allows us to use the standard definition of the gradient of a function $f(x) : \mathbb{R}^n \rightarrow \mathbb{R}$ as $\frac{\partial f}{\partial x} = [\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}]$, and use the chain rule to backpropagate the gradient: $\frac{\partial f}{\partial y} = \frac{\partial f}{\partial x} \cdot \frac{\partial x}{\partial y}$, where $\frac{\partial x}{\partial y}$ is the Jacobian matrix $(\frac{\partial x_i}{\partial y_j})$.

However, this assumption breaks down when the computation graph involves 3D transformations, such as when a network iteratively updates its estimate of the camera pose [31, 40, 33, 10, 20]. 3D transformations do not form vector spaces and instead lie on smooth manifolds; the notion of addition is undefined and the standard notion of gradient defined in Euclidean spaces does not apply.

A typical approach in prior work is to embed the 3D transforms in a Euclidean space. For example, a rigid body transform from $SE(3)$ is represented as a 4×4 matrix and treated as a vector in $\mathbf{R}^{16}$. That is, $SE(3)$ corresponds to a subset of R^{16}, or more specifically, an embedded submanifold of R^{16}. Functions involving $SE(3)$ such as the exponential map and the inverse are replaced with their extensions in the embedding space, i.e. the matrix exponential and the matrix inverse:

$$\begin{aligned} &\exp : \mathfrak{se}(3) \mapsto SE(3) \longrightarrow \overline{\exp} : \mathbb{R}^6 \mapsto \mathbb{R}^{4\times4} \\ &\text{inv} : SE(3) \mapsto SE(3) \longrightarrow \overline{\text{inv}} : \mathbb{R}^{4\times4} \mapsto \mathbb{R}^{4\times4}. \end{aligned} \tag{1}$$

Now, the computation graph involves only Euclidean objects, and backpropagation can be performed as usual as long as the Jacobian of $\overline{\exp}$ and $\overline{\text{inv}}$ can be calculated.

There are several problems with this approach. First, while the extended functions such as the matrix exponential are smooth as a whole, the individual substeps needed for computing the backward passes often contain singularities. As a result, small deviations off the manifold can result in numerical instabilities causing gradients to explode. It can often be quite difficult to implement these functions in libraries such as PyTorch[27] and Tensorflow[1] in a way that numerically stable gradients are achieved through automatic differentiation. As a case in point, the commonly used PyTorch3D library[28] returns nan-gradients when the identity matrix is given as input the matrix log. Furthermore, some

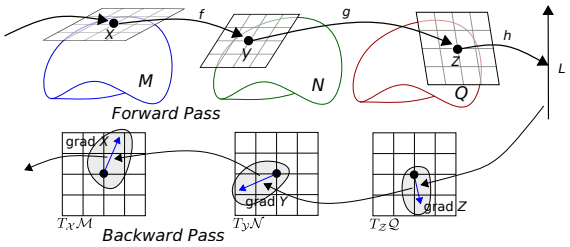

Figure 1. Tangent space Backpropagation. The forward pass is a composition of mappings between Lie groups. The backward pass propogates gradients in the tangent space of each element.

extended functions have very complicated backward pass, which leads to extremely large computation graphs, especially for groups like $Sim(3)$ which involve complex expressions for the matrix exponential and logarithm. Partly for this reason, to our knowledge, no prior work has performed backpropagation on $Sim(3)$.

Due to the aforementioned problems, existing deep learning libraries are unable to handle 3D transformations reliably and transparently. And incorporating 3D transformations into a computation graph typically requires a substantial amount of painstaking, ad-hoc manual effort.

In this work, we seek to make backpropagation on 3D transformations robust and "painless". We introduce a new approach for performing backpropagation though mixed computation graphs consisting of both real vectors and 3D transformations. Instead of embedding transformation groups in Euclidean spaces, we retain the group structure and perform backpropagation directly in the *tangent space* of each group element (Fig. 1). For example, while a rigid body transformation $T \in SE(3)$ may be represented as a $\mathbb{R}^{4\times4}$ matrix, we backpropagate the gradient in the tangent space $\mathfrak{se}(3)$, in particular, as a 6-dimensional vector in a local coordinate system centered at T.

We show that performing differentiation in the tangent space has several advantages

- *Numerical Stability:* By performing backpropagation in the tangent space, we avoid needing to differentiate through singularity-ridden substeps required for the embedding approach, guaranteeing numerically stable gradients. This allows us to provide groups such as $Sim(3)$ which do not give stable gradients when automatic differentiation is directly applied.

- *Representation Agnostic:* The computation of the gradients does not depend on how the 3D transformations are represented. Both quaternions and 3×3 matrices can be used to represent rotations without changing how the backward pass is computed.

- *Reduced Computation Graphs:* When functions such as `exp` and `log` are implemented directly in PyTorch, the output of each individual steps within `exp` and `log` need to be stored for the backward pass, leading to unnecessarily large computation graphs. We avoid the need to store intermediate values by differentiating through these functions using the group structure.

- *Manifold Optimization:* Our library can be directly used for problems where the variables we want to optimize are 3D transforms. Since we compute gradients directly in the tangent space, we avoid the need to reproject gradients.

We demonstrate use cases of our approach on a wide range of vision and robotics tasks. We show our approach can be used for pose graph optimization, inverse kinematics, RGB-D scan registration, and RGB-D SLAM.

Our approach is implemented as an easy-to-use, plug-and-play PyTorch library. It allows users to insert 3D transformations, either as parameters or activations, into a computation graph, just as regular tensors; backpropagation is taken care of transparently. Our 3D transformation objects expose an interface similar to the `Tensor` object, supporting arbitrary batch shapes, indexing, and reshaping operations.

Our contributions are two-fold. First, we introduce a new method of auto-differentiation involving 3D transformation groups by performing backpropagation in the tangent space. To our knowledge, this is the first time backpropagation is performed in the tangent space of Lie groups for training neural networks. Second, we introduce LieTorch, an open-source, easy-to-use PyTorch library that implements tangent space backpropagation. We expect our library to be a useful tool for researchers in 3D vision and robotics.

2. Related Work

Automatic Differentiation: Automatic Differentiation (AD) is a family of algorithms for evaluating derivatives of a program. AD frameworks expose a set of elementary operators (e.g. matrix multiplication, convolution, and pooling) and programs can be constructed by composing operations. Modern implementations can handle complex computations graphs with branching, loops, and recursion[4, 27].

AD has two common forms: *forward mode differentiation* applies the chain rule to each elementary operator in the forward pass. Optimization frameworks such as Ceres[2] implement forward mode differentiation using dual numbers. *Reverse mode differentiation* is a general form of backpropagation and works by complementing each intermediate value with an gradient. During the forward pass, the intermediate values are populated. In the backward pass, gradients are propagated in reverse. Reverse mode differentiation is well suited for differentiating functions with a single objective $f : \mathbb{R}^n \mapsto \mathbb{R}$, such as the loss used to

train a neural network. Deep learning frameworks such as Theano[32], Tensorflow[1], Autograd[23], PyTorch[27], and JAX[7] all support reverse mode differentation.

Existing frameworks do not directly support manifold elements in the computation graph. These libraries assume every variable belongs to an Euclidean space and every function maps from one Euclidean space to another. However, 3D transformations groups such as rotations do not form a vector space, and the usual notions of derivatives do not apply. We use a more general notation of gradient defined in tangent spaces and show that we can support 3D transformation groups by performing differentiation in the tangent space. We build our library on top of PyTorch, and expose an interface similar to the `Tensor` object, supporting arbitrary batch shapes, indexing, and reshaping operations. By building on PyTorch, we can compose Lie group and tensor operations in a shared computation graph.

Numeric issues often arise in AD if operations are naively implemented. For example, expressions such as the L2 norm $||\cdot||_2$ or the `LogSumExp` are problematic if implemented directly, and various tricks are required in order to ensure numeric stability. The exponential and logarithmic maps for 3D transformation groups contain many problematic expressions. We implement group operations (e.g. `exp`, `log`, `act`, `adj`) as the elementary operations.

Manifold Optimization: Many vision and robotics problems requiring optimizing variables which lie on a manifold. Libraries such as GTSAM[12] and g2o[14] provide general frameworks for solving nonlinear least-squares and MAP inference problems involving manifold elements such as camera poses. GTSAM and Koppel et al.[17] provide frameworks which can perform automatic differentiation over lie groups. However, these frameworks are tailored to the computation of Jacobian matrices and cannot be readily used within the computation graphs for training neural networks.

There are several libraries which provide tools for optimization on manifolds, such as Manopt[6], and PyManopt includes autodifferentiation capabilities provided by PyTorch[27], Tensorflow[1], and Autograd [23]. Several extensions have been proposed to PyTorch which allow optimization over smooth manifolds such as McTorch[25] and Geopt[16]. These libraries work by embedding manifolds in the Euclidean space $\mathbb{R}^n$, and manifold functions $f : \mathcal{N} \mapsto \mathcal{M}$ are implemented as the extension $\overline{f} : \mathbb{R}^n \mapsto \mathbb{R}^m$. Automatic differentiation can be used to differentiate $\overline{f}$, and the gradient on $\mathcal{M}$ is obtained using the orthogonal projection[5].

This strategy can be prone to large computation graphs and numerical instabilities. Our library avoids the need to differentiate $\overline{f} : \mathbb{R}^n \mapsto \mathbb{R}^m$ by differentiating the original function $f : \mathcal{N} \mapsto \mathcal{M}$ directly, i.e. obtaining the differential of f, which is a map from the tangent space of $\mathcal{N}$ to the tangent space of $\mathcal{M}$. We note that this is not possible for all manifolds, but using the additional structure provided by Lie Groups, we demonstrate that AD can be performed directly in the tangent space. An additional advantage of our approach is that we never need to perform a projection step, since gradients are already defined in the tangent space.

3. Preliminaries

A matrix Lie group $\mathcal{M}$ is both a group and a smooth manifold. Each element $X \in \mathcal{M}$ can be represented as a matrix in $\mathbb{R}^{n\times n}$. The group operator is identical to matrix multiplication and the group inverse is identical to matrix inversion. Being a smooth manifold, each element $X \in \mathcal{M}$ has a unique tangent space. Moreover, the tangent space of each group element is a vector space isomorphic to the tangent space of the identity element.

Lie Algebra: The lie algebra $\mathfrak{g}$ is defined as the tangent space at the identity element, and each group has an associated lie algebra. The lie algebra $\mathfrak{g}$ forms a vector space with a set of basis elements $\{G_1, \ldots, G_k\}$. The lie algebra is isomorphic to $\mathbb{R}^k$, and we can map between elements of $\mathfrak{g}$ and elements of $\mathbb{R}^k$ using the hat $^\wedge$

$$^\wedge : \mathbb{R}^k \to \mathfrak{g} : \qquad \boldsymbol{\tau}^\wedge = \sum_i^k \tau_i G_i \tag{2}$$

and the vee operator $^\vee : \mathfrak{g} \mapsto \mathbb{R}^k$ which is the inverse of the wedge operator, such that $(\boldsymbol{\tau}^\wedge)^\vee = \boldsymbol{\tau}$. The lie algebra $\mathfrak{g}$ is isomorphic to $\mathbb{R}^k$, $\mathfrak{g} \cong \mathbb{R}^k$. Since it is often easier to work in $\mathbb{R}^k$, we perform differentiating using vectors in $\mathbb{R}^k$, but it would be equivalent to representing gradients in $\mathfrak{g}$. The definitions of the $\wedge$ and $\vee$ operators for 3D transformation groups are given in the appendix.

Exponential and Logarithm Map: Elements of the lie algebra $\boldsymbol{\phi}^\wedge \in \mathfrak{g}$ can be exactly mapped to the manifold through the exponential map.

$$\exp(\boldsymbol{\phi}^\wedge) = \mathbf{I} + \boldsymbol{\phi}^\wedge + \frac{1}{2!}(\boldsymbol{\phi}^\wedge)^2 + \frac{1}{3!}(\boldsymbol{\phi}^\wedge)^3 + \ldots \tag{3}$$

The logarithm map is the inverse of the exponential map and takes elements from the manifold to the lie algebra. For convenience, we use vectorized versions of the exponential and logarithm map which map directly between $\mathbb{R}^k$ and $\mathcal{M}$

$$\mathrm{Exp} : \mathbb{R}^k \to \mathcal{M} \qquad \mathrm{Log} : \mathcal{M} \to \mathbb{R}^k \tag{4}$$

Group Multiplication: The group is endowed with a binary operator $\circ$ such that two group elements can be combined to form a third element $X \circ Y \in \mathcal{M}$. Using the notation of Sola et al. [30], we can overload the the addition and subtraction operations

$$\oplus : \mathbb{R}^k \times \mathcal{M} \to \mathcal{M} \qquad \xi \oplus X = \mathrm{Exp}(\xi) \circ X \tag{5}$$

$$\ominus : \mathcal{M} \times \mathcal{M} \to \mathbb{R}^k \qquad X \ominus Y = \mathrm{Log}(X \cdot Y^{-1}) \tag{6}$$

Operation	Map	Description
Exp	$\mathfrak{g} \mapsto \mathcal{M}$	exponential map
Log	$\mathcal{M} \mapsto \mathfrak{g}$	logarithm map
Inv	$\mathcal{M} \mapsto \mathcal{M}$	group inverse
Mul	$\mathcal{M} \times \mathcal{M} \mapsto \mathcal{M}$	group multiplication
Adj	$\mathcal{M} \times \mathfrak{g} \mapsto \mathfrak{g}$	adjoint operator
AdjT	$\mathcal{M} \times \mathfrak{g}^* \mapsto \mathfrak{g}^*$	dual adjoint operator
Act	$\mathcal{M} \times \mathbb{R}^3 \mapsto \mathbb{R}^3$	action on point (set)
ActP	$\mathcal{M} \times \mathbb{P}^3 \mapsto \mathbb{P}^3$	action on homogeneous point (set)

Table 1. Summary of operations supported by our library. Each operation is differentiable with respect to all the input arguments. Both the lie algebra $\mathbf{g}$ and its dual $\mathbf{g}^*$ are embedded in $\mathbb{R}^k$.

If the manifold $\mathcal{M}$ is a Euclidean space, then $\oplus$ and $\ominus$ are identical to standard vector addition and subtraction.

Adjoint: The Lie algebra and exponential map give us two possible local parameterizations of a neighborhood around a group element X: the "right action parameterization" $X \circ \mathrm{Exp}(a)$ and the "left action parameterization" $\mathrm{Exp}(b) \circ X$, where a and b represent the local coordinates. For a group element around X, its coordinates a and b are related by a linear map, which is the adjoint of X:

$$\mathrm{Adj}_X(a) = (Xa^{\wedge}X^{-1})^{\vee}, \tag{7}$$

from which it is easy to verify that

$$X \circ \mathrm{Exp}(a) = \mathrm{Exp}(\mathrm{Adj}_X(a)) \circ X. \tag{8}$$

Because Adj_X is a linear map, we $\mathbf{Adj}_X$ to denote its matrix representation, i.e. $\mathbf{Adj}_X$ is a $n \times n$ matrix where n is the dimension of the tangent space of X. The expressions of the adjoint for 3D transformation groups are given in the appendix.

3D Transformation Groups In this paper we are particularly concerned a special class of Lie groups that perform 3D transformation—SO(3), SE(3), and Sim(3). A description of the supported group operations is given in Tab. 1.

Differentials and Gradients: Given a function $f : \mathbb{R}^n \to \mathbb{R}^m$ and a point $x \in \mathbb{R}^n$ the differential of f is the linear operator

$$Df(x)[v] = \lim_{t \to 0} \frac{f(x + tv) - f(x)}{t} \tag{9}$$

However, this is problematic for functions on Lie groups $f : \mathcal{M} \to \mathcal{M}'$ since in general $\mathcal{M}$ is not closed under addition. However, we can generalize the differential as perturbations in the tangent space

$$Df(X)[\mathbf{v}] = \lim_{t \to 0} \frac{f(t\mathbf{v} \oplus X) \ominus f(X)}{t} \tag{10}$$

where v belongs to the tangent space of X. Eqn. 10 relates perturbations in the tangent space of X to perturbations in the tangent space of $f(X)$. Plugging in the (orthonormal) basis vectors into Eqn. 10 we can recover the Jacobian matrix $\mathbf{J} \in \mathbb{R}^{n \times m}$

$$\mathbf{J}_{ij} = \lim_{t \to 0} \frac{< f(t\mathbf{e}_j \oplus X) \ominus f(X), \mathbf{e}'_i >}{t}, \tag{11}$$

where $<,>$ is an inner product defined on the tangent space of $\mathcal{M}'$.

Throughout this paper we will use $\frac{\partial Y}{\partial X}$ to denote the Jacobian of a mapping $f : X \mapsto Y$ between two Lie groups, under the basis vectors given by the left action parameterizations of the tangent spaces of Y and X. That is, the dimension of $\frac{\partial Y}{\partial X}$ is $m \times n$, where m is the dimension of the tangent space of Y and n is the dimension of the tangent space of X.

Note that under vector addition, a Euclidean space is a Lie group whose tangent space is itself, so this definition subsumes the standard notion of Jacobian. In particular, for a loss function $L(X) \in \mathbb{R}$, $\frac{\partial L}{\partial X}$ is a row vector representing the gradient of L in the tangent space of X.

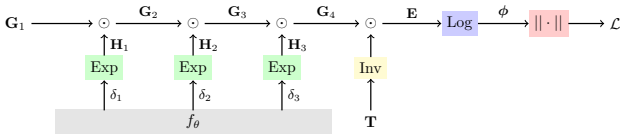

Figure 2. Computation graph involving Lie groups. A network f_θ producs a series of updates $\delta_1, \delta_2, \delta_2$ which are applied to pose $\mathbf{G}_1$. The loss function is defined on the geodesic distance between the estimated pose and the ground truth pose $\mathbf{T}$.

4. Approach

Our approach performs backpropagation through computation graphs consisting of Lie group elements and operations which map between groups. As an example, consider the computation graph shown in Fig. 2. The computation graph in Fig. 2 can be written as

$$\mathcal{L} = ||\mathbf{T}^{-1} \circ (e^{\delta_1} e^{\delta_2} e^{\delta_3} \mathbf{G}_1)|| \tag{12}$$

where a series of updates $\delta_1, \delta_2, \delta_3$ are predicted by a network f_θ. The loss is then taken to be the norm of the geodesic distance between $e^{\delta_1} e^{\delta_2} e^{\delta_3} \mathbf{G}_1$ and the ground truth pose $\mathbf{T}$. This type of computation graph shows up in many different applications where incremental updates are applied to an initial estimate.

4.1. Reverse Mode Autodifferentation

Each node in the computation graph (Fig. 2) represents an element X on some Lie group $\mathcal{M}$ and each edge in the computation graph represents a function which maps from one Lie group $\mathcal{N}$ to another $\mathcal{M}$.

$$f : \mathcal{M} \to \mathcal{N}, \qquad X \in \mathcal{M},\ f(X) \in \mathcal{N} \tag{13}$$

This notation subsumes standard Euclidean elements such as vectors and matrices, which also form a Lie groups under vector addition.

During the forward pass, we evaluate each function in the computation graph in topological order. The output of each function is stored for the backward pass. The backward pass propagates gradients in reverse topological order. Given a function as defined in Eqn. 13, $Y = f(X)$, we can backpropagate the gradient using the chain rule

$$\frac{\partial \mathcal{L}}{\partial X} = \frac{\partial \mathcal{L}}{\partial Y}\frac{\partial Y}{\partial X} = \frac{\partial \mathcal{L}}{\partial Y}\mathbf{J} \tag{14}$$

where $\frac{\partial \mathcal{L}}{\partial X}$ is a row vector with the same dimension as the tangent space of X and $\mathbf{J}$ is the Jacobian in Eqn 11. Eqn. 14 computes a Jacobian-vector product.

4.2. Computing the Jacobians

We use Eqn. 10 to derive analytical expressions for the Jacobians. As an example, we show how the Jacobians can be computed for two functions: group multiplication and the logarithm map. The derivations for other functions including the group inverse and the exponential map are included in the appendix.

Group Multiplication: Consider the first group multiplication in Fig. 2: $Z = X \circ Y$. Using the definition of the differential (Eqn. 10), we first compute the differential with respect to X

$$Df(X)[\mathbf{v}] = \lim_{t\to 0} \frac{\mathrm{Log}((e^{t\mathbf{v}}XY)(XY)^{-1})}{t} \tag{15}$$

$$= \lim_{t\to 0} \frac{\mathrm{Log}(e^{t\mathbf{v}})}{t} = \lim_{t\to 0} \frac{t\mathbf{v}}{t} = \mathbf{v} \tag{16}$$

The differential with respect to $\mathcal{Y}$ can be derived in a similiar manner

$$Df(Y)[\mathbf{v}] = \lim_{t\to 0} \frac{\mathrm{Log}((Xe^{t\mathbf{v}}Y)(XY)^{-1})}{t} \tag{17}$$

$$\tag{18}$$

applying the adjoint (Eqn. 8)

$$= \lim_{t\to 0} \frac{\mathrm{Log}(e^{\mathbf{Adj}_X\cdot t\mathbf{v}}(XY)(XY)^{-1})}{t} \tag{19}$$

$$= \lim_{t\to 0} \frac{\mathrm{Log}(e^{\mathbf{Adj}_X\cdot t\mathbf{v}})}{t} = \mathbf{Adj}_X \cdot \mathbf{v} \tag{20}$$

Using equation Eqn, 14, we can propogate the gradients as

$$\frac{\partial \mathcal{L}}{\partial X} = \frac{\partial \mathcal{L}}{\partial Z} \qquad \frac{\partial \mathcal{L}}{\partial Y} = \frac{\partial \mathcal{L}}{\partial Z}\mathbf{Adj}_X. \tag{21}$$

As an example, for $R \in SO(3)$, $\mathbf{Adj}_R = R$.

Logarithm Map: The logarithm map $\phi = \mathrm{Log}(X)$ takes a group element to its Lie algebra. As example, for $SO(3)$ its logarithm map $\mathrm{Log} : SO(3) \to \mathbb{R}^3$ can be expressed as

$$\mathrm{Log}(X) = \frac{\psi(X - X^T)^\vee}{2\sin(\psi)}, \psi = \cos^{-1}\left(\frac{tr(X)-1}{2}\right) \tag{22}$$

Using Eqn. 10, we can express the differential of the logarithm map (between tangent spaces) as

$$D\,\mathrm{Log}(X)[\mathbf{v}] = \lim_{t\to 0} \frac{\mathrm{Log}(\mathrm{Exp}(t\mathbf{v}) \circ X) - \mathrm{Log}(X)}{t} \tag{23}$$

From the Baker-Campbell-Hausdorff formula [3] we have

$$\log(\exp(\delta\xi)\exp(\phi)) \approx \phi + \mathbf{J}_l^{-1}(\phi)\delta\phi, \text{ when } \delta\phi \text{ is small,}$$

where $\mathbf{J}_l(\phi)$ is a matrix called "the left Jacobian" [30] which maps a perturbation in the tangent space to a perturbations on the manifold. As an example, $\mathbf{J}_l^{-1}$ of $SO(3)$ has a closed form expression [30]:

$$\mathbf{J}_l^{-1}(\phi) = \mathbf{I}_{3\times 3} - \frac{1}{2}\phi^\wedge + \left(\frac{1}{\phi^2} - \frac{1+\cos\phi}{2\phi\sin\phi}\right)(\phi^\wedge)^2 \tag{24}$$

We can then derive the backpropagated gradient as (full derivation in the appendix):

$$\frac{\partial \mathcal{L}}{\partial X} = \frac{\partial \mathcal{L}}{\partial \phi}\cdot \mathbf{J}_l^{-1}(\phi), \tag{25}$$

which is a vector in the tangent space. For SO(3), the gradient is 3-dimensional.

There also exists a closed form for $\mathbf{J}_l^{-1}$ for the $SE(3)$ group. For Lie groups such as $Sim(3)$ without an analytic expression for the left Jacobian, we can numerically approximate the gradient (to some desired precision) using the series expansion [3]

$$\mathbf{J}_l^{-1}(\phi) = \sum_{n=0} (-1)^n \frac{B_n}{n!}(\phi^\curlywedge)^n. \tag{26}$$

where $\phi^\curlywedge = \mathbf{adj}(\phi^\wedge)$ and $\mathbf{adj}$ is the adjoint of the Lie algebra $\mathfrak{sim}(3)$. B_n are the Bernoulli numbers. For small ϕ, we can numerically approximate $\mathbf{J}_l(\phi) \approx \mathbf{I}$.

Embedding Space vs. Tangent Space The logarithm map in $SO(3)$ is a good example to show the advantages of our approach over the standard embedding space backpropagation (e.g. Autograd in Pytorch), which simply autodifferentiates the expressions given in Eqn. 22 and obtains a 9-dimensional gradient, as opposed to a 3-dimensional gradient in our approach.

The forward pass is the same for our approach and the standard approach—both use Eqn. 22. Eqn. 22 as a whole is smooth but contains numerically problematic terms such as $\frac{\psi}{\sin\psi}$. The solution is to use approximations given by its

Taylor expansion $\frac{x}{\sin\psi} \approx 1 + \frac{1}{6}\psi^2 + \ldots$, when ψ is small. This replacement is done only around singular points, corresponding to a dynamic modification of the computation graph during the forward pass.

The backward pass, however, causes two difficulties for the standard backpropagation. First, it needs to handle backpropagating through numerically unstable terms such as $\frac{\psi}{\sin\psi}$. In existing work, this is done by simply backpropagating through the Taylor approximation formula. However, the gradient of the Taylor approximation is not necessarily a good approximation of the true gradient. As a result, when to use and how many terms to use need to be very carefully tuned. For example, it is common to have the term $\frac{1}{x^2}$ in a Taylor approximation; including it risks division by zero in the term $\frac{1}{x^3}$ in the backward pass, but excluding it risks deviation from the manifold in the forward pass.

Second, the standard backpropagation needs to handle terms with singular gradients. In Eqn. 22, the term $\cos^{-1}\left(\frac{tr(X)-1}{2}\right)$ has a singular gradient when X is identity because $\frac{\partial}{\partial x}\cos^{-1}(x)$ is undefined at $x = 1$. This issue in fact remains unaddressed in existing libraries. For example, PyTorch3D [28] returns a NaN gradient for its matrix logarithm when the identity matrix is given as input.

In contrast, our approach suffers from none of these difficulties. For the backward pass we simply use Eqn. 24, which is as straightforward as computing the forward pass.

4.3. Implementation

One of the technical challenges is integrating our tangent space representation into automatic differentiation software. We implement our library as an extension to PyTorch. We define a new type to represent group elements, and subclass this type for different groups. The resulting computation graph consists of mixed types, both Euclidean vectors and group elements. We implement a custom gradient for any function with group inputs or outputs, including the exponential map, logarithm map, adjoint, group inverse, group multiplication, and action on a point set.

Our implementation is a plug-and-play extension on PyTorch. To enable general use cases, we support arbitrary batch shapes, and common tensor operations such as indexing, reshaping, slicing, and repeating dimensions. We use unit quaternions to represent rotations since they are compact and have desirable numeric properties. All operations involving groups include both Cuda and c++ kernels to leverage the GPU if available.

5. Experiments

We show that our library can help a wide range of tasks.

Inverse Kinematics We first evaluate our library on a toy inverse kinematics task. Given a robot arm with joint lengths $d_1, ..., d_N \in \mathbb{R}^+$ and a target x^*, the task is to find a set of relative rotations $\Delta\mathbf{R}_1, \Delta\mathbf{R}_2, ..., \Delta\mathbf{R}_N$ such that the end of the arm is positioned at x^*. Given the relative joint angles, we can use forward kinematics to compute the arm position

$$\mathbf{R}_i = \Delta\mathbf{R}_i \cdot \Delta\mathbf{R}_{-1} \cdot \cdots \cdot \Delta\mathbf{R}_1 \tag{27}$$

$$y = \sum_i^N \mathbf{R}_i \cdot \begin{pmatrix} d_i & 0 & 0 \end{pmatrix}^T, \quad \mathcal{L} = ||y - x^*||_2^2 \tag{28}$$

We also experiment with extendable joints. These can be represented as the group of rotation and scaling in 3D or the $\mathbb{R}^+ \times SO(3)$ group, which can be represented as a rotation $\mathbb{R} \in SO(3)$ and a scaling $s \in \mathbb{R}^+$, $s\mathbf{R} \in \mathbb{R}^+ \times SO(3)$

We compare our approach to two different PyTorch/Autograd implementations. First, we directly backpropagate through each group operator, using the implementions provided by our library. For small angles, Backpropagation is performed through a Taylor approximation of the functions.We also show the performance of Autograd when the operations are explictly tuned for better stability. This includes reimplementing normalization functions, decreasing the threshold when Taylor approximations are used, and making division operations safe for small angles by adding a small epsilon to the denominator.

	$SO(3)$	$\mathbb{R}^+ \times SO(3)$
PyTorch+Autograd	0.0	0.0
PyTorch+Autograd (tuned)	99.8	**100.0**
Ours	**100.0**	**100.0**

The results from these experiments are provided in Tab. 5. We perform 1000 runs for both the $SO(3)$ and $\mathbb{R}^+ \times SO(3)$ experiments. We report the portion of runs which converge to the correct solution within 1000 iterations using a tolerance threshold of 1×10^{-4}.

We see that without tuning the forward pass, PyTorch+Autograd diverges on every single problem. By modifying the forward expression to make it safe for automatic differentation, we can get near 100% convergence. Our library converges on all the problems without modification.

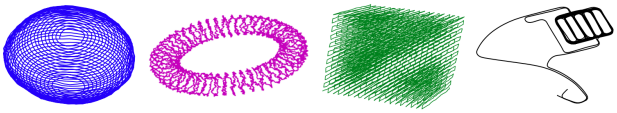

Figure 3. Optimized pose graphs on (left to right) sphere, torus, grid, and parking-garage problems. We perform gradient descent on the rotation group as initialization, followed by 7 Gauss-Newton updates.

Pose Graph Optimization Pose graph optimization is the problem of recovering the trajectory of a robot given a

	g2o[14]	gtsam[12]	chordal+gtsam[9]	gradient+gtsam[9]	PyTorch+Autograd	Ours
Parking-Garage ($n = 1661, m = 6275$)	6.42×10^{-1} -	$\mathbf{6.35 \times 10^{-1}}$ -	$\mathbf{6.35 \times 10^{-1}}$ 0.23	$\mathbf{6.35 \times 10^{-1}}$ 1.29	$\mathbf{6.35 \times 10^{-1}}$ 16.5	$\mathbf{6.35 \times 10^{-1}}$ 1.18
Sphere-A ($n = 2200, m = 8647$)	5.32×10^{10} -	5.71×10^{10} -	$\mathbf{1.49 \times 10^{6}}$ 0.88	1.9×10^{6} 46.3	$\mathbf{1.49 \times 10^{6}}$ 16.9	$\mathbf{1.49 \times 10^{6}}$ 1.19
Torus ($n = 5000, m = 9048$)	6.04×10^{8} -	4.71×10^{10} -	$\mathbf{1.21 \times 10^{4}}$ 1.18	2.81×10^{4} 20.2	$\mathbf{1.21 \times 10^{4}}$ 17.2	$\mathbf{1.21 \times 10^{4}}$ 1.17
Cube ($n = 8000, m = 22236$)	5.39×10^{7} -	6.59×10^{11} -	$\mathbf{4.22 \times 10^{4}}$ 17.9	$\mathbf{4.22 \times 10^{4}}$ 26.4	$\mathbf{4.22 \times 10^{4}}$ 18.3	$\mathbf{4.22 \times 10^{4}}$ 1.21

Table 2. Error (top row) and time (bottom row) on pose graph optimization. Time (seconds) is reported for the initialization method; the first two columns are run without any initiliation.

set of noisy measurements. Conventional methods use iterative solvers such as Gauss-Newton[12] or Levenberg-Marquardt[14]. However, due to rotation, pose graph optimization is a non-convex optimization problem, and second order methods are prone to local minimum [9].

One solution for overcoming local minimum is to use a good initialization. *Riemannian gradient descent*[37] provides an initialization for rotation by performing gradient descent using a reshaped cost function. These rotations are then used as initialization for second order methods.

We perform optimization over all rotations jointly, using the reshaped cost function over the geodesic distance using the reshaping function proposed by Tron et al. [36]

$$\theta = ||\log(\mathbf{R}_i^{-1} \cdot \mathbf{R}_j \cdot \mathbf{R}_{ij}^{-1})||_2 \tag{29}$$

$$\mathcal{L}(\theta) = 1/b - (1/b + \theta)\exp(-b\theta) \tag{30}$$

where $\mathbf{R}_{ij}$ are the noisy measurements and we set $b = 1.5$. For optimization we use SGD with momentum set to 0.5. We perform 1000 gradient steps and a exponentially decaying learning rate $.995^{\gamma}$. Riemannian gradient descent can be easily implemented using our library with only a few lines of code, as shown in the sample below

```
# (ii, jj) edges in pose graph
from lietorch import SO3

Eij = SO3(torch.from_numpy(Eij)).to('cuda')
R = SO3(torch.from_numpy(R)).to('cuda'))
optimizer = optim.SGD([R], lr=1.0, momentum=0.5)

for i in range(n_steps):
    optimizer.zero_grad()
    dE = (R[ii].inv() * R[jj]) * Eij.inv()
    loss = reshaping_fn(dE)

    loss.backward()
    optimizer.step()
```

We follow the setup by Carlone et al.[9] and report convergence and timing results in Tab. 5; optimized pose graphs are shown in Fig. 3. We compare to g2o[14] and gtsam[12]; gtsam also provides implementations of different initialization strategies such as chordal relaxation[24] and Riemannian gradient descent[36]. We test both our approach and a default PyTorch/Autograd implementation where backpropagation is performed directly in the embedding space.

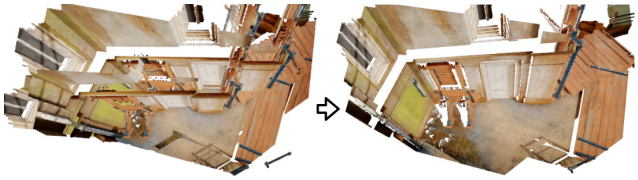

Figure 4. Example Sim3 registration, the network takes two RGB-D scans as input and predicts a similiarity transformation which aligns the two scans.

On all datasets, we find that our gradient based initialization converges to the global minimum, matching the performance of chordal relaxation. On two datasets, the gradient based initializer in gtsam gets stuck in a local minimum, while our implementation is able to converge. On all datasets, our implementation is much faster than gradient+gtsam since our library can leverage the GPU. chordal+gtsam is faster on the smaller problems, but our gradient based initializer is much faster on larger problems. We find that our method converges to the same solution as Autograd, but since our library performs a much simpler backward pass, it can properly leverage the GPU, consistently providing a 10-15x speedup.

RGB-D Sim3 Registration Given two RGB-D scans captured from different poses, we want to find a similarity transformation which aligns the two scans. This registration problem shows up in scan-to-CAD registration and loop closure in monocular SLAM[26]. While several recent works have used deep networks for registration [10, 13], they have focused on recovering a $SE(3)$ transformation. Here, we demonstrate that our library can recover a $Sim(3)$ transformation that allows scale change.

We run our experiments on the synthetic TartanAir dataset[39], taking the scenes *westerndesert, seasidetown, seasonsforest_winter, office2* and *gascola* for testing and the remaining scenes for validation and training. The network is trained to predict a $SE(3)$ or $Sim(3)$ transformation between a pair of frames.

In order to select training pairs which have sufficient dif-

	$SE(3)$		$Sim(3)$		
	tr. (<1cm)	rot. (< 0.1°)	tr. (<1cm)	rot. (< 0.1°)	scale (<1%)
Identity	0.05	0.05	0.05	0.05	0.65
PyTorch+Autograd	0.0 (NaN)	0.0 (NaN)	0.0 (NaN)	0.0 (NaN)	0.0 (NaN)
PyTorch+Autograd (tuned)	78.75 ± 0.25	90.9 ± 0.25	77.0 ± 0.25	90.8 ± 0.25	98.0 ± 0.25
Ours (1st order)	78.55 ± 0.28	91.1 ± 0.40	77.4 ± 0.28	90.9 ± 0.40	98.1 ± 0.43
Ours (2nd order)	$\mathbf{79.0} \pm 0.59$	91.0 ± 0.39	77.3 ± 0.59	91.1 ± 0.39	$\mathbf{98.3} \pm 0.35$
Ours (3rd order)	78.6 ± 0.48	$\mathbf{91.2} \pm 0.58$	$\mathbf{77.4} \pm 0.48$	$\mathbf{91.2} \pm 0.58$	97.2 ± 0.25
Ours (Analytic)	78.7 ± 0.33	91.1 ± 0.36	-	-	-

Table 3. Results on RGB-D registration task. We provide the mean and standard deviation over 3 runs each.

	360	desk	desk2	floor	plant	room	rpy	teddy	xyz	avg
DeepTAM [40]	0.111	0.053	0.103	**0.206**	0.064	0.239	0.093	0.144	0.036	0.116
DeepV2D[33]	**0.072**	**0.069**	0.074	0.317	0.046	0.213	0.082	**0.114**	**0.028**	0.113
DeepV2D (ours)	0.096	0.077	**0.072**	0.268	**0.024**	**0.173**	**0.057**	0.136	0.040	**0.105**

Table 4. Tracking results in the RGB-D benchmark (ATE rmse [m]).

ficulty, we compute the magnitude of the optical flow between all pairs of frames and uniformly sample pairs where the mean flow magnitude is between 16 and 120 pixels, resulting in a median translation of 54cm and a median rotation angle of 4.7°.. For Sim(3), we sample scaling uniformly in the range $[0.5, 2.0]$ and rescale the depth maps accordingly.

We use a network architecture based on RAFT[35] and build a full 4D correlation volume by computing the visual similarity between all-pairs of pixels in the input images (more details on the network architecture are provided in the appendix). We start by initializing the estimate of the 3D similarity transform to be the identity element $\mathbf{I} \in Sim(3)$. The network predicts a series of updates $\Delta\mathbf{T}_k \in Sim(3)$ which are applied to the current estimate.

In each iteration, we use the current estimate of the transformation to compute the optical flow and inverse depth change. We use the optical flow to index from the correlation volume similiar to RAFT. A GRU uses the correlation features and inverse depth error to output pixelwise residuals $r_x, r_y, r_z \in \mathbb{R}$ along with respective confidence weights $w_x, w_y, w_z \in [0, 1]$. r_x and r_y is the residual flow in the x and y directions, while r_z is the residual inverse depth.

The residual terms are used as input to a differentiable least squares optimization layer, which performs 3 Gauss-Newton updates which update the estimated transformation through $T_{k+1} = \Delta x \oplus T_k$. During training, we use our library to backprop through the Gauss-Newton updates to train the weights of the network. We train the network using the geodesic loss

$$\mathcal{L}(\mathbf{T}_1, \ldots, \mathbf{T}_K) = \sum_k || \operatorname{Log}(\mathbf{T}_k^{-1} \cdot \mathbf{T}^*)|| \qquad (31)$$

where $\mathbf{T}^*$ is the ground truth transformation.

We provide results from this experiment in Tab. 3, and qualitative results are shown in Fig. 4. As we note earlier, there isn't a closed form expression for the left Jacobian of $Sim(3)$. Hence, we use a series approximation based on the Taylor expansion. We find that the order of the approximation makes little difference in the final accuracy. To the best of our knowledge, this is the first time backpropagation has been performed on similarity transformations, which is easily enabled by our approach.

RGB-D SLAM In our final experiment, we use our library to implement an deep RGB-D SLAM system. We reimplement the approach from DeepV2D[33] in PyTorch so that it can be directly used with our library. DeepV2D maintains a set of keyframes as it processes the video. It estimates optical flow between all pairs of keyframes, then the optical flow is used to solve for a set of pose updates over all pairs jointly. This is done using a differentiable least squares solver which minimizes the reprojection error.

Like DeepV2D, we train on a combination of the NYU and ScanNet datasets using 4 frame video sequences as input. Pose estimation in DeepV2D was trained using an indirect proxy loss on the optical flow induced by the estimated poses. We train using the more direct geodesic error (Eq. 31), made possible by our library.

We provide results in Tab. 4 and compare to the classical method RGBD-SLAM[15] along with deep learning methods DeepTAM[41] and DeepV2D[34]. We see that the new loss, difficult to implement with standard backpropagation but made easy by our approach, resulted in improved tracking performance.

6. Conclusions

We have proposed a new approach for backpropagation through computation graphs containing 3D transformation groups. We have shown that our approach can benefit a wide range of vision and robotics tasks.

Acknowledgments: This research is partially supported by a grant (IIS-1942981) from the National Science Foundation and a grant from Samsung.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016. 1, 3

[2] Sameer Agarwal, Keir Mierle, et al. Ceres solver. 2012. 2

[3] Timothy D Barfoot. *State estimation for robotics*. Cambridge University Press, 2017. 5, 11, 14

[4] Atılım Günes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *The Journal of Machine Learning Research*, 18(1):5595–5637, 2017. 2

[5] Nicolas Boumal. An introduction to optimization on smooth manifolds. *Lie bracket*, 2020. 3

[6] N. Boumal, B. Mishra, P.-A. Absil, and R. Sepulchre. Manopt, a Matlab toolbox for optimization on manifolds. *Journal of Machine Learning Research*, 15(42):1455–1459, 2014. 3

[7] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018. 3

[8] Arunkumar Byravan and Dieter Fox. Se3-nets: Learning rigid body motion using deep neural networks. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 173–180. IEEE, 2017. 1

[9] 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. IEEE, 2015. 7

[10] Christopher Choy, Wei Dong, and Vladlen Koltun. Deep global registration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2514–2523, 2020. 1, 7

[11] 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. 1

[12] Frank Dellaert. Factor graphs and gtsam: A hands-on introduction. Technical report, Georgia Institute of Technology, 2012. 3, 7

[13] Zan Gojcic, Caifa Zhou, Jan D Wegner, Leonidas J Guibas, and Tolga Birdal. Learning multiview 3d point cloud registration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1759–1769, 2020. 7

[14] 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), Shanghai, China*, pages 9–13, 2011. 3, 7

[15] Christian Kerl, Jürgen Sturm, and Daniel Cremers. Dense visual slam for rgb-d cameras. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2100–2106. IEEE, 2013. 8

[16] Max Kochurov, Rasul Karimov, and Serge Kozlukov. Geoopt: Riemannian optimization in pytorch, 2020. 3

[17] Leonid Koppel and Steven L. Waslander. Manifold geometry with fast automatic derivatives and coordinate frame semantics checking in C++. In *15th Conference on Computer and Robot Vision (CRV)*, 2018. 3

[18] Jatavallabhula Krishna Murthy, Soroush Saryazdi, Ganesh Iyer, and Liam Paull. gradslam: Dense slam meets automatic differentiation. *arXiv*, 2020. 1

[19] Xuan Luo, Jia-Bin Huang, Richard Szeliski, Kevin Matzen, and Johannes Kopf. Consistent video depth estimation. *arXiv preprint arXiv:2004.15021*, 2020. 1

[20] Zhaoyang Lv, Frank Dellaert, James M Rehg, and Andreas Geiger. Taking a deeper look at the inverse compositional algorithm. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4581–4590, 2019. 1

[21] Zhaoyang Lv, Kihwan Kim, Alejandro Troccoli, Deqing Sun, James M Rehg, and Jan Kautz. Learning rigidity in dynamic scenes with a moving camera for 3d motion field estimation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 468–484, 2018. 1

[22] Wei-Chiu Ma, Shenlong Wang, Rui Hu, Yuwen Xiong, and Raquel Urtasun. Deep rigid instance scene flow. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3614–3622, 2019. 1

[23] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, volume 238, page 5, 2015. 3

[24] 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. 7

[25] Mayank Meghwanshi, Pratik Jawanpuria, Anoop Kunchukuttan, Hiroyuki Kasai, and Bamdev Mishra. Mctorch, a manifold optimization library for deep learning. *arXiv preprint arXiv:1810.01811*, 2018. 3

[26] Raul Mur-Artal, Jose Maria Martinez Montiel, and Juan D Tardos. Orb-slam: a versatile and accurate monocular slam system. *IEEE transactions on robotics*, 31(5):1147–1163, 2015. 7

[27] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017. 1, 2, 3

[28] Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari. Accelerating 3d deep learning with pytorch3d. *arXiv preprint arXiv:2007.08501*, 2020. 1, 6

[29] Paul-Edouard Sarlin, Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. Superglue: Learning feature

matching with graph neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4938–4947, 2020. 1

[30] Joan Sola, Jeremie Deray, and Dinesh Atchuthan. A micro lie theory for state estimation in robotics. *arXiv preprint arXiv:1812.01537*, 2018. 3, 5

[31] Chengzhou Tang and Ping Tan. Ba-net: Dense bundle adjustment network. *arXiv preprint arXiv:1806.04807*, 2018. 1

[32] The Theano Development Team, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*, 2016. 3

[33] Zachary Teed and Jia Deng. Deepv2d: Video to depth with differentiable structure from motion. *arXiv preprint arXiv:1812.04605*, 2018. 1, 8

[34] Zachary Teed and Jia Deng. Deepv2d: Video to depth with differentiable structure from motion. *arXiv preprint arXiv:1812.04605*, 2018. 8

[35] Zachary Teed and Jia Deng. Raft: Recurrent all-pairs field transforms for optical flow. *arXiv preprint arXiv:2003.12039*, 2020. 8, 15

[36] Roberto Tron, Bijan Afsari, and René Vidal. Intrinsic consensus on so (3) with almost-global convergence. In *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, pages 2052–2058. IEEE, 2012. 7

[37] Roberto Tron and Rene Vidal. Distributed 3-d localization of camera sensor networks from 2-d image measurements. *IEEE Transactions on Automatic Control*, 59(12):3325–3340, 2014. 7

[38] Chen Wang, Danfei Xu, Yuke Zhu, Roberto Martín-Martín, Cewu Lu, Li Fei-Fei, and Silvio Savarese. Densefusion: 6d object pose estimation by iterative dense fusion. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3343–3352, 2019. 1

[39] Wenshan Wang, Delong Zhu, Xiangwei Wang, Yaoyu Hu, Yuheng Qiu, Chen Wang, Yafei Hu, Ashish Kapoor, and Sebastian Scherer. Tartanair: A dataset to push the limits of visual slam. *arXiv preprint arXiv:2003.14338*, 2020. 7

[40] Huizhong Zhou, Benjamin Ummenhofer, and Thomas Brox. Deeptam: Deep tracking and mapping. In *Proceedings of the European conference on computer vision (ECCV)*, pages 822–838, 2018. 1, 8

[41] Huizhong Zhou, Benjamin Ummenhofer, and Thomas Brox. Deeptam: Deep tracking and mapping. In *Proceedings of the European conference on computer vision (ECCV)*, pages 822–838, 2018. 8

Tangent Space Backpropagation for 3D Transformation Groups Appendix

A. SO(3), SE(3) and Sim(3) Formulas

Given a Lie group $\mathcal{G}$ with Lie algebra $\mathfrak{g}$, we provide the expressions for $SO(3)$, $SE(3)$, and $Sim(3)$

$\wedge$ **Operator:** The $\wedge$ operator takes elements from $\mathbb{R}^k$ to the lie algebra $\mathfrak{g}$. For $\boldsymbol{\phi} \in \mathbb{R}^3$

$$\boldsymbol{\phi}^\wedge = \begin{pmatrix} 0 & -\phi_z & \phi_y \\ \phi_z & 0 & -\phi_x \\ -\phi_y & \phi_x & 0 \end{pmatrix} \in \mathfrak{so}(3) \tag{32}$$

for $\boldsymbol{\xi} = (\boldsymbol{\tau}, \boldsymbol{\phi}) \in \mathbb{R}^6$

$$\boldsymbol{\xi}^\wedge = \begin{pmatrix} \boldsymbol{\phi}^\wedge & \boldsymbol{\tau} \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & -\phi_z & \phi_y & \tau_x \\ \phi_z & 0 & -\phi_x & \tau_y \\ -\phi_y & \phi_x & 0 & \tau_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \in \mathfrak{se}(3) \tag{33}$$

and for $\boldsymbol{\eta} = (\boldsymbol{\tau}, \boldsymbol{\phi}, \sigma) \in \mathbb{R}^7$

$$\boldsymbol{\eta}^\wedge = \begin{pmatrix} \boldsymbol{\phi}^\wedge + \sigma \mathbf{I}_{3\times 3} & \boldsymbol{\tau} \\ \sigma & 1 \end{pmatrix} = \begin{pmatrix} \sigma & -\phi_z & \phi_y & \tau_x \\ \phi_z & \sigma & -\phi_x & \tau_y \\ -\phi_y & \phi_x & \sigma & \tau_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \in \mathfrak{sim}(3) \tag{34}$$

$\curlywedge$ **Operator:** The $\curlywedge$ operator takes elements from $\mathbb{R}^k$ to the lie algebra $\mathrm{adj}(\mathfrak{g})$, where $\mathrm{adj}(\mathfrak{g})$ is the Lie algebra associated with the group $\mathrm{Adj}(\mathcal{G}) = \{\mathrm{Adj}(X) \mid X \in \mathcal{G}\}$, It can be shown that $\mathrm{Adj}(\mathcal{G})$ also forms a Lie group [3].

For $\boldsymbol{\phi} \in \mathbb{R}^3$

$$\boldsymbol{\phi}^\curlywedge = \boldsymbol{\phi}^\wedge \in \mathrm{adj}(\mathfrak{so}(3)) \tag{35}$$

for $\boldsymbol{\xi} = (\boldsymbol{\tau}, \boldsymbol{\phi}) \in \mathbb{R}^6$

$$\boldsymbol{\xi}^\curlywedge = \begin{pmatrix} \boldsymbol{\phi}^\wedge & \boldsymbol{\tau}^\wedge \\ \mathbf{0} & \boldsymbol{\phi}^\wedge \end{pmatrix} \in \mathrm{adj}(\mathfrak{se}(3)) \tag{36}$$

and for $\boldsymbol{\eta} = (\boldsymbol{\tau}, \boldsymbol{\phi}, \sigma) \in \mathbb{R}^7$

$$\boldsymbol{\eta}^\curlywedge = \begin{pmatrix} \boldsymbol{\phi}^\wedge + \sigma \mathbf{I}_{3\times 3} & \boldsymbol{\tau}^\wedge & -\boldsymbol{\tau} \\ \mathbf{0} & \boldsymbol{\phi}^\wedge & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & 0 \end{pmatrix} \in \mathrm{adj}(\mathfrak{sim}(3)) \tag{37}$$

Exp **Map:** The exponential map takes elements from the Lie algebra to the Lie group. For $SO(3)$, $SE(3)$, and $Sim(3)$ the exponential map has a closed form expressions. For $\boldsymbol{\phi} \in \mathbb{R}^3$

$$\mathrm{Exp}(\boldsymbol{\phi}) = \exp(\boldsymbol{\phi}^\wedge) = \mathbf{I}_{3\times 3} + \frac{\sin(\theta)}{\theta}\boldsymbol{\phi}^\wedge + \frac{1-\cos(\theta)}{\theta^2}(\boldsymbol{\phi}^\wedge)^2, \qquad \theta = ||\boldsymbol{\phi}|| \tag{38}$$

for $\boldsymbol{\xi} = (\boldsymbol{\tau}, \boldsymbol{\phi}) \in \mathbb{R}^6$

$$\mathrm{Exp}(\boldsymbol{\xi}) = \begin{pmatrix} \mathbf{R} & \mathbf{V}\boldsymbol{\tau} \\ \mathbf{0} & 1 \end{pmatrix}, \qquad \mathbf{R} = \mathrm{Exp}(\boldsymbol{\phi}) \tag{39}$$

$$\mathbf{V} = \mathbf{I}_{3\times 3} + \frac{1-\cos(\theta)}{\theta^2}\boldsymbol{\phi}^\wedge + \frac{\theta - \sin(\theta)}{\theta^3}(\boldsymbol{\phi}^\wedge)^2, \qquad \theta = ||\boldsymbol{\phi}|| \tag{40}$$

and for $\boldsymbol{\eta} = (\boldsymbol{\tau}, \boldsymbol{\phi}, \sigma) \in \mathbb{R}^7$

$$\mathrm{Exp}(\boldsymbol{\xi}) = \begin{pmatrix} e^\sigma \mathbf{R} & \mathbf{W}\boldsymbol{\tau} \\ \mathbf{0} & 1 \end{pmatrix}, \qquad \mathbf{R} = \mathrm{Exp}(\boldsymbol{\phi}) \tag{41}$$

$$\mathbf{W} = \left(\frac{e^\sigma - 1}{\sigma}\right)\mathbf{I}_{3\times 3} + \frac{1}{\theta}\left(\frac{e^\sigma \sin(\theta)\sigma + (1 - e^\sigma\cos(\theta))\theta}{\sigma^2 + \theta^2}\right)\boldsymbol{\phi}^\wedge + \tag{42}$$

$$\frac{1}{\theta^2}\left(\frac{e^\sigma - 1}{\sigma} + \frac{(e^\sigma\cos(\theta) - 1)\sigma + e^\sigma\sin(\theta)\theta}{\sigma^2 + \theta^2}\right)(\boldsymbol{\phi}^\wedge)^2, \qquad \theta = ||\boldsymbol{\phi}|| \tag{43}$$

When θ or σ is small, we use second order Taylor approximations of the exponential maps to avoid numerical issues.

Log **Map:** The logarithm map takes elements from the Lie group to the Lie algebra. For $SO(3)$, $SE(3)$, and $Sim(3)$ the logarithm map can be computed in closed form. For a rotation $\mathbf{R} \in SO(3)$

$$\text{Log}(\mathbf{R}) = \frac{\psi(\mathbf{R} - \mathbf{R}^T)^\vee}{2\sin(\psi)}, \ \psi = \cos^{-1}\left(\frac{tr(\mathbf{R}) - 1}{2}\right) \tag{44}$$

for $\mathbf{G} = \begin{pmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{pmatrix} \in SE(3)$ we have

$$\boldsymbol{\xi} = \begin{pmatrix} \boldsymbol{\tau} \\ \boldsymbol{\phi} \end{pmatrix} = \begin{pmatrix} \mathbf{V}^{-1}\mathbf{t} \\ \text{Log}(\mathbf{R}) \end{pmatrix} = \text{Log}(\mathbf{G}) \tag{45}$$

$$\mathbf{V}^{-1} = \mathbf{I}_{3\times 3} - \frac{1}{2}\phi^\wedge + \left(\frac{1}{\phi^2} - \frac{1+\cos\theta}{2\theta\sin\theta}\right)(\phi^\wedge)^2, \qquad \theta = ||\boldsymbol{\phi}|| \tag{46}$$

and for $\mathbf{T} = \begin{pmatrix} s\mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{pmatrix} \in Sim(3)$

$$\boldsymbol{\eta} = \begin{pmatrix} \boldsymbol{\tau} \\ \boldsymbol{\phi} \\ \sigma \end{pmatrix} = \begin{pmatrix} \mathbf{W}^{-1}\mathbf{t} \\ \text{Log}(\mathbf{R}) \\ \ln(s) \end{pmatrix} = \text{Log}(\mathbf{T}) \tag{47}$$

where $\mathbf{W}^{-1}$ can be computed by taking the inverse Eqn. 43.

Adj **Operator:** The ajoint operator is a linear map which allows us to move an element $\nu \in \mathfrak{g}$ in the right tangent space of $X \in \mathcal{G}$ to the left tangent space

$$\text{Exp}(\text{Adj}_X\, \nu) \circ X = X \circ \text{Exp}(\nu) \tag{48}$$

For $\mathbf{R} \in SO(3)$

$$\text{Adj}_{\mathbf{R}} = \mathbf{R} \tag{49}$$

for $\mathbf{G} = \begin{pmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{pmatrix} \in SE(3)$

$$\text{Adj}_{\mathbf{G}} = \begin{pmatrix} \mathbf{R} & \boldsymbol{\tau}^\wedge\mathbf{R} \\ \mathbf{0} & \mathbf{R} \end{pmatrix} \tag{50}$$

and for $\mathbf{T} = \begin{pmatrix} s\mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{pmatrix} \in Sim(3)$ we have

$$\text{Adj}_{\mathbf{T}} = \begin{pmatrix} s\mathbf{R} & \boldsymbol{\tau}^\wedge\mathbf{R} & -\mathbf{t} \\ \mathbf{0} & \mathbf{R} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & 1 \end{pmatrix} \tag{51}$$

Inv **Operator:** Since $SO(3)$, $SE(3)$, and $Sim(3)$ all form a group, each element has a unique inverse. For $\mathbf{R} \in SO(3)$

$$\mathbf{R}^{-1} = \mathbf{R}^T \tag{52}$$

for $\mathbf{G} = \begin{pmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{pmatrix} \in SE(3)$

$$\mathbf{G}^{-1} = \begin{pmatrix} \mathbf{R}^T & -\mathbf{R}^T\mathbf{t} \\ \mathbf{0} & 1 \end{pmatrix} \tag{53}$$

and for $\mathbf{T} = \begin{pmatrix} s\mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{pmatrix} \in Sim(3)$ we have

$$\mathbf{T}^{-1} = \begin{pmatrix} s^{-1}\mathbf{R}^T & -s^{-1}\mathbf{R}^T\mathbf{t} \\ \mathbf{0} & 1 \end{pmatrix} \tag{54}$$

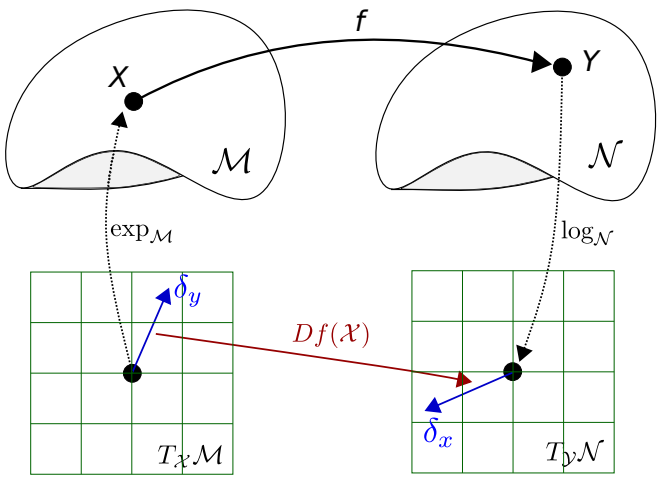

Figure 5. Illistration of the differential between two Lie groups.

B. Differentials and Jacobians

In the main paper, we derived the gradients for group multiplication. Here we provide derivations of the gradients for the remaining operators

Group Inverse: Using the definition of the differential

$$Df(X)[\mathbf{v}] = \lim_{t\to 0} \frac{\mathrm{Log}((e^{t\mathbf{v}}X)^{-1}(X^{-1})^{-1})}{t} \tag{55}$$

$$= \lim_{t\to 0} \frac{\mathrm{Log}(X^{-1}e^{-t\mathbf{v}}X)}{t} \tag{56}$$

using the adjoint

$$= \lim_{t\to 0} \frac{\mathrm{Log}(\mathrm{Exp}(-\mathrm{Adj}_{X^{-1}}(t\mathbf{v}))X^{-1}X)}{t} \tag{57}$$

$$= \lim_{t\to 0} \frac{-\mathrm{Adj}_{X^{-1}}\, t\mathbf{v}}{t} = -\mathrm{Adj}_{X^{-1}}\,\mathbf{v} \tag{58}$$

This can be used to recover the Jacobian $-\mathrm{Adj}_{X^{-1}}$.

Action on a Point: We can use elements from the 3D transformation groups to transform a 3D point or set of points. Given a homogeneous point $p = (X, Y, Z, 1)^T$ we can transform p using a transformation X

$$\mathbf{p}' = X\mathbf{p} \tag{59}$$

To make the notation consistent for all groups, a rotation can be represented as the 4×4 matrix $X = \begin{pmatrix}\mathbf{R} & 0\\ 0 & 1\end{pmatrix}$. X is a linear operator on p, so the Jacobian is simply the matrix representation of X

$$\frac{\partial \mathbf{p}'}{\partial \mathbf{p}} = X \tag{60}$$

We can also get the differential with respect to the transformation

$$Df(X)[\mathbf{v}] = \frac{d}{dt}(e^{t\mathbf{v}}X\mathbf{p})\Big|_{t=0} = \frac{d}{dt}(e^{t\mathbf{v}}\mathbf{p}')\Big|_{t=0} = \mathbf{v}^{\wedge}\mathbf{p}' \tag{61}$$

Adjoint: We consider the adjoint as the function $Adj : \mathcal{G}\times\mathfrak{g}\mapsto\mathfrak{g}$, $Adj_X(\boldsymbol{\omega}) = \boldsymbol{\upsilon}$. We need the Jacobians with respect to both X and ω. Since the adjoint is a linear map in terms of υ then

$$\frac{\partial \boldsymbol{\upsilon}}{\partial \boldsymbol{\omega}} = \mathrm{Adj}_X, \qquad \frac{\partial L}{\partial \omega} = \frac{\partial L}{\partial \upsilon}\mathrm{Adj}_X \tag{62}$$

where $\mathrm{Adj}_X \in \mathbb{R}^{6\times 6}$ is the matrix representation of the adjoint. The gradient with respect to X can be found

$$D\,\mathrm{Adj}_X(\boldsymbol{\omega})[\mathbf{v}] = \frac{\partial}{\partial t}\left(e^{t\mathbf{v}} X \boldsymbol{\omega}^\wedge (e^{t\mathbf{v}} X)^{-1}\right)^\vee \Big|_{t=0} \tag{63}$$

$$= \frac{\partial}{\partial t}\left(e^{t\mathbf{v}} X \boldsymbol{\omega}^\wedge X^{-1} e^{-t\mathbf{v}}\right)^\vee \Big|_{t=0} \tag{64}$$

$$= \frac{\partial}{\partial t}\left(e^{t\mathbf{v}} \boldsymbol{\upsilon}^\wedge e^{-t\mathbf{v}}\right)^\vee \Big|_{t=0} \tag{65}$$

$$= \left(\frac{\partial}{\partial t} e^{t\mathbf{v}} \boldsymbol{\upsilon}^\wedge e^{-t\mathbf{v}}\right)^\vee \Big|_{t=0} \tag{66}$$

$$= \left(\mathbf{v}^\wedge \boldsymbol{\upsilon}^\wedge - \boldsymbol{\upsilon}^\wedge \mathbf{v}^\wedge\right)^\vee = \boldsymbol{\upsilon}^\curlywedge \mathbf{v} \tag{67}$$

Where $\curlywedge$ is defined in Sec A. We note that the differential is linear in $\mathbf{v}$ allowing us to write the Jacobian and gradients as

$$\frac{\partial \boldsymbol{\upsilon}}{\partial X} = \boldsymbol{\upsilon}^\curlywedge, \qquad \frac{\partial L}{\partial X} = \frac{\partial L}{\partial \upsilon} \boldsymbol{\upsilon}^\curlywedge \tag{68}$$

Exponential and Logarithm Maps: The Jacobian of the exponential map $\mathbf{J}_l = \frac{\partial}{\partial \mathbf{x}} \mathrm{Exp}(\mathbf{x})$ is referred to as the left-Jacobian and can be written using the series [3] (page 235)

$$\mathbf{J}_l(\phi) = \sum_{n=0}^{\infty} \frac{1}{(n+1)!} (\phi^\curlywedge)^n \tag{69}$$

For $SO(3)$ and $SE(3)$ closed form expressions exist for Eqn. 69, otherwise we use the first 3 terms.

The Jacobian of the logarithm map $\mathbf{J}_l^{-1} = \frac{\partial}{\partial X} \mathrm{Log}(X)$ and can be computed using the series

$$\mathbf{J}_l^{-1}(\phi) = \sum_{n=0}^{\infty} \frac{B_n}{n!} (\phi^\curlywedge)^n \tag{70}$$

where B_n are the Bernoulli numbers [3](page 234). Again, we used analytic expressions of $\mathbf{J}_l^{-1}$ for $SO(3)$ and $SE(3)$, and the first 3 terms for $Sim(3)$.

C. Sim(3) Network Architecture

A overview of the Sim(3) network architecture is shown in Fig. 6. The context and feature encoders are identical to RAFT. We replace the 5×1, 1×5 GRU used in RAFT with a single 3×3 convolutional GRU, using a hidden state size of 128 channels. We apply 12 iterations during both training and testing.

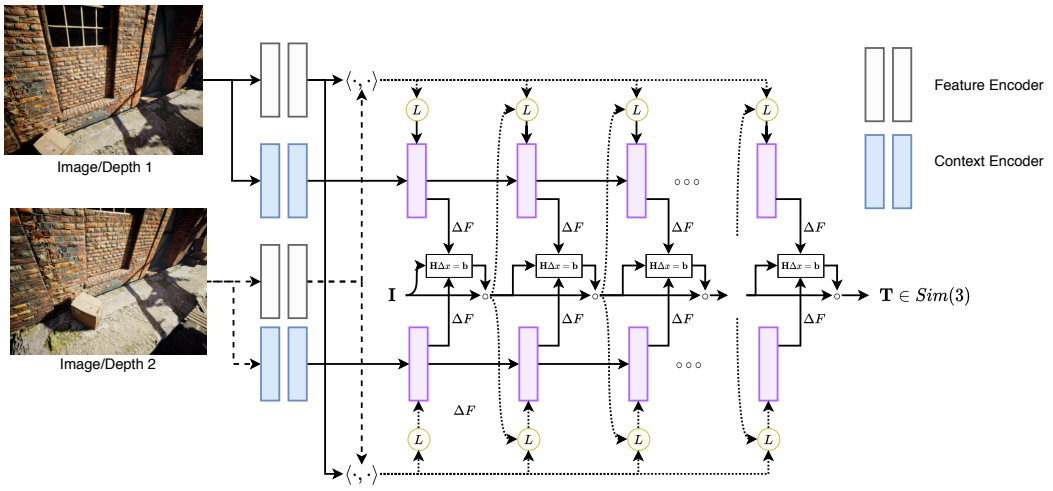

Figure 6. Network architecture used for our Sim(3) registration experiments. The network architecture is based on RAFT[35]. The top branch estimates motion from $I_1 \rightarrow I_2$ and the bottom branch estimates motion in the opposite direction $I_2 \rightarrow I_1$. Features are first extracted from each of the two input images and used to construct two 4D correlation volumes, which are pooled at multiple resolutions according to RAFT. During each iteration, the current estimate of the transformation $\mathbf{T}$ is used to index from each of the correlation volumes. The correlation features are processed by the GRU which outputs a residual flow estimate (optical flow not explained by the current transformation $\mathbf{T}$. Both bidirectional residual flow estimates are used as input to a optimization layer, which unrolls 3 Gauss-Newton iterations to produced a transformation update $\Delta\mathbf{T}$, which is applied to the current transformation estimate.