PyPose: A Library for Robot Learning with Physics-based Optimization

Chen Wang[1,2,✉], Dasong Gao[1,3], Kuan Xu[4], Junyi Geng[1], Yaoyu Hu[1], Yuheng Qiu[1],
Bowen Li[1], Fan Yang[5], Brady Moon[1], Abhinav Pandey[6], Aryan[1,7], Jiahe Xu[1], Tianhao Wu[8],
Haonan He[1], Daning Huang[6], Zhongqiang Ren[1], Shibo Zhao[1], Taimeng Fu[9], Pranay Reddy[10],
Xiao Lin[11], Wenshan Wang[1], Jingnan Shi[3], Rajat Talak[3], Kun Cao[4], Yi Du[2], Han Wang[4], Huai Yu[12],
Shanzhao Wang[13], Siyu Chen[4], Ananth Kashyap[14], Rohan Bandaru[15], Karthik Dantu[2],
Jiajun Wu[16], Lihua Xie[4], Luca Carlone[3], Marco Hutter[5], Sebastian Scherer[1]

https://pypose.org

Abstract

Deep learning has had remarkable success in robotic perception, but its data-centric nature suffers when it comes to generalizing to ever-changing environments. By contrast, physics-based optimization generalizes better, but it does not perform as well in complicated tasks due to the lack of high-level semantic information and reliance on manual parametric tuning. To take advantage of these two complementary worlds, we present PyPose: a robotics-oriented, PyTorch-based library that combines deep perceptual models with physics-based optimization. PyPose's architecture is tidy and well-organized, it has an imperative style interface and is efficient and user-friendly, making it easy to integrate into real-world robotic applications. Besides, it supports parallel computing of any order gradients of Lie groups and Lie algebras and 2nd-order optimizers, such as trust region methods. Experiments show that PyPose achieves more than $10\times$ speedup in computation compared to the state-of-the-art libraries. To boost future research, we provide concrete examples for several fields of robot learning, including SLAM, planning, control, and inertial navigation.

[✉]Corresponding Author. chenwang@dr.com
[1]Carnegie Mellon University, Pittsburgh, PA 15213, USA.
[2]State University of New York at Buffalo, NY 14260, USA.
[3]Massachusetts Institute of Technology, Cambridge, MA 02139, USA.
[4]Nanyang Technological University, Singapore 639798.
[5]ETH Zürich, 8092 Zürich, Switzerland.
[6]Pennsylvania State University, University Park, PA 16801, USA.
[7]Delhi Technological University, Delhi, India.
[8]University of Virginia, Charlottesville, VA 22904, USA.
[9]The Chinese University of Hong Kong, Shenzhen, China.
[10]University of Massachusetts Amherst, MA 01003, USA.
[11]Georgia Institute of Technology, Atlanta, GA 30332, USA
[12]Wuhan University, Hubei 430072, China.
[13]University of Michigan, Ann Arbor, MI 48109, USA.
[14]Fox Chapel Area High School, Pittsburgh, PA 15238, USA.
[15]Lexington High School, Lexington, MA 02421, USA.
[16]Stanford University, Stanford, CA 94305, USA.

1. Introduction

Deep learning has made great inroads in visual perception tasks such as classification [66], segmentation [45], and detection [38]. However, it is still often unsatisfactory in some robotic applications due to the lack of training data [65], ever-changing environments [76], and limited computational resources [67]. On the other hand, physics-based optimization has shown great generalization ability and high accuracy in many vision and robotic tasks, such as control [24], planning [70], and simultaneous localization and mapping (SLAM) [76]. Nevertheless, it relies on problem-specific parameter tuning and suffers from the lack of semantic information. Since both methods have shown their own merits, more and more efforts have been made to take advantage of the two complementary worlds [75].

Currently, learning-based methods and physics-based optimization are typically used separately in different modules of a robotic system [23]. For example, in semantic SLAM, learning-based methods showed promising results in scenarios where high-level semantic information is needed or as a replacement for hand-crafted features and descriptors, e.g., feature matching in the front-end [53], while physics-based optimization plays a vital role in cases where a well-defined physical model can be established, e.g., pose graph optimization in the back-end [12]. Researchers usually first execute the front end and then pass the results to the back end for optimization. Despite the tremendous progress in the past decades, such a two-stage, decoupled paradigm may only achieve sub-optimal solutions, which in turn limits system performance and generalization ability. Hence, developing integrated methods with end-to-end differentiation through optimization is an emerging trend [59–61].

A variety of applications in perception, motion planning, and automatic control have been explored for end-to-end learning [30,37,59]. However, most of these applications rely

on problem-specific implementations that are often coded from scratch, which makes it difficult for researchers to build upon prior work and explore new ideas. This hinders the development cycle due to the lack of a unified and systematic development framework. For example, people usually leverage PyTorch-based [48] models for developing learning-based perceptual models, but then have to use C(++)-based optimization libraries, such as GTSAM [20], OMPL [54], and CT [27], for physics-based optimization. The mixed usage of Python and C++ libraries increases the system complexity and slows down the development cycle as it is time-consuming for cross-language debugging and inefficient to transfer data among different processes, e.g., ROS nodes [51]. Therefore, there is an urgent need for a systematic development tool in a single language, accelerating end-to-end learning for physics-based optimization.

Some researchers have spent effort towards this objective. For example, LieTorch exploits the group structure of 3D transformations and performs back-propagation in the tangent spaces of manifolds [60]. However, only 1st-order differentiable operations are currently implemented, which limits its practical use, since higher order derivatives provide additional local information about the data distribution and enable new applications [44]. CvxpyLayer [3] takes convex optimization as a differentiable neural network layer, while it doesn't support operation for Lie groups and 2nd-order optimizers. Similarly, Theseus [49] takes non-linear optimization as network layers; however, it adopts rotation matrices for transformation representation, which is memory inefficient for practical robotic applications.

To address the above limitations, we present PyPose, an open-source library based on PyTorch to connect learning-based perceptual models with classical algorithms that can be formulated as physics-based optimization, e.g., geometry problem, factor-graph optimization, and optimal control. In summary, our main contributions are:

- We present a new python-based open-source library, Py-Pose, to further enable end-to-end learning with physics-based optimization and accelerate the next generation of developments in robotics. PyPose is designed to be easily interpretable, user-friendly, and efficient with a tidy and well-organized architecture. It provides an imperative programming style for the convenience of real-world robotic applications. PyPose supports any order gradient computation of Lie groups and Lie algebras, and 2nd-order optimizers such as Levenberg-Marquardt with trust region steps. As demonstrated in Figure 1, our experiments show that PyPose achieves more than $10\times$ faster compared to state-of-the-art libraries.
- We provide sample uses of PyPose. Users can easily build upon existing functionalities for various robotic applications. To the best of our knowledge, PyPose is one of the first Python libraries to comprehensively cover several sub-fields of robotics, such as perception, SLAM, and control, where optimization is involved.

2. Related Work

Existing open source libraries related to PyPose can mainly be divided into two groups: (1) linear algebra and machine learning frameworks and (2) optimization libraries.

2.1. Linear Algebra & Machine Learning

Linear algebra libraries are essential to machine learning and robotics research. To name a few, NumPy [46], a linear algebra library for Python, offers comprehensive operations on vectors and matrices while enjoying higher running speed due to its underlying well-optimized C code. Eigen [29], a high performance C++ linear algebra library, has been used in many projects such as TensorFlow [1], Ceres [2], GT-SAM [20], and g^2o [28]. ArrayFire [42], a GPU acceleration library for C, C++, Fortran, and Python, contains simple APIs and provides thousands of GPU-tuned functions.

While relying heavily on high-performance numerical linear algebra techniques, machine learning libraries focus more on operations on tensors (i.e., high-dimensional matrices) and automatic differentiation. Early machine learning frameworks, such as Torch [19], OpenNN [47], and MATLAB [43], provide primitive tools for researchers to develop neural networks. However, they only support CPU computation and lack concise APIs, which plague engineers using them in applications. A few years later, deep learning frameworks such as Chainer [62], Theano [4], and Caffe [34] arose to handle the increasing size and complexity of neural networks while supporting multi-GPU training with convenient APIs for users to build and train their neural networks. Furthermore, the recent frameworks, such as TensorFlow [1], PyTorch [48], and MXNet [18], provide a comprehensive and flexible ecosystem (e.g., APIs for multiple programming languages, distributed data parallel training, and facilitating tools for benchmark and deployment). Gvnn [31] introduced differentiable transformation layers into Torch-based framework, leading to end-to-end geometric learning. JAX [9] can automatically differentiate native Python and NumPy functions and is an extensible system for composable function transformations. In many ways, the existence of these frameworks facilitated and promoted the growth of deep learning. Recently, more efforts have been taken to combine standard optimization tools with deep learning. Recent work like Theseus [49] and CvxpyLayer [3] showed how to embed differentiable optimization within deep neural networks.

By leveraging PyTorch, our proposed library, PyPose, enjoys the same benefits as the existing state-of-the-art deep learning frameworks. Additionally, PyPose provides new features, such as 2nd-order optimizers and computing any order gradients of Lie groups and Lie algebras, which are essential to geometrical optimization problems and robotics.

Table 1. A list of supported class alias of `LieTensor`.

Transformation	Lie Group	Lie Algebra
Rotation	SO3()	so3()
Rotation & Translation	SE3()	se3()
Rotation & Translation & Scale	Sim3()	sim3()
Rotation & Scale	RxSO3()	rxso3()

2.2. Open Source Optimization Libraries

Numerous optimization solvers and frameworks have been developed and leveraged in robotics. To mention a few, Ceres [2] is an open-source C++ library for large-scale nonlinear least squares optimization problems and has been widely used in SLAM. Pyomo [32] and JuMP [22] are optimization frameworks that have been widely used due to their flexibility in supporting a diverse set of tools for constructing, solving, and analyzing optimization models. CasADi [6] has been used to solve many real-world control problems in robotics due to its fast and effective implementations of different numerical methods for optimal control.

Pose- and factor-graph optimization also play an important role in robotics. For example, g^{2}o [28] and GTSAM [20] are open-source C++ frameworks for graph-based nonlinear optimization, which provide concise APIs for constructing new problems and have been leveraged to solve several optimization problems in SLAM and bundle adjustment.

Optimization libraries have also been widely used in robotic control problems. To name a few, IPOPT [64] is an open-source C++ solver for nonlinear programming problems based on interior-point methods and is widely used in robotics and control. Similarly, OpenOCL [36] supports a large class of optimization problems such as continuous time, discrete time, constrained, unconstrained, multi-phase, and trajectory optimization problems for real-time model-predictive control. Another library for large-scale optimal control and estimation problems is CT [27], which provides standard interfaces for different optimal control solvers and can be extended to a broad class of dynamical systems in robotic applications. In addition, Drake [58] also has solvers for common control problems and that can be directly integrated with its simulation tool boxes. Its system completeness made it favorable to many researchers.

3. Method

To reduce the learning curve of new users, we leverage—to the greatest possible extent—existing structures in PyTorch. Our design principles are to keep the implementation logical, modular, and transparent so that users can focus on their own applications and invest effort in the mathematical aspects of their problems rather than the implementation details. Similar to PyTorch, we believe that trading a bit of efficiency for interpretability and ease of development is acceptable. However, we are still careful in maintaining a good balance between interpretability and efficiency by leveraging advanced PyTorch functionalities.

Following these principles, we mainly provide four concepts to enable end-to-end learning with physics-based optimization, i.e., `LieTensor`, `Module`, `Function`, and `Optimizer`. We briefly present their motivation, mathematical underpinning, and the interfaces in PyPose.

3.1. `LieTensor`

In robotics, 3D transformations are crucial for many applications, such as SLAM, control, and motion planning. However, most machine learning libraries, including PyTorch, assume that the computation graph operates in Euclidean space, while a 3D transformation lies on a smooth manifold. Simply neglecting the manifold structure will lead to inconsistent gradient computation and numerical issues. Therefore, we need a specific data structure to represent 3D transformations in learning models.

To address the above challenge while keeping PyTorch's excellent features, we resort to Lie theory and define `LieTensor` as a subclass of PyTorch's `Tensor` to represent 3D transformations. One of the challenges of implementing a differentiable `LieTensor` is that we often need to calculate numerically problematic terms such as $\frac{\sin x}{x}$ for the conversion between a Lie group and Lie algebra [60]. To solve this problem, we take the Taylor expansion to avoid calculating the division by zero. To illustrate this, consider the exponential map from $\mathbb{R}^3$ to $\mathbb{S}^3$ as an example, where $\mathbb{S}^3$ is the set of unit-quaternions. For numerical stability, the exponential map from $\mathbb{R}^3$ to $\mathbb{S}^3$ is formulated as follows:

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

where $\gamma_e = \frac{1}{\|\boldsymbol{x}\|} \sin\left(\frac{\|\boldsymbol{x}\|}{2}\right)$, $\gamma_o = \frac{1}{2} - \frac{1}{48}\|\boldsymbol{x}\|^2 + \frac{1}{3840}\|\boldsymbol{x}\|^4$, and eps is the smallest machine number where $1 + \text{eps} \neq 1$.

`LieTensor` is different from the existing libraries in several aspects: (**1**) PyPose supports auto-diff for any order gradient and is compatible with most popular devices, such as CPU, GPU, TPU, and Apple silicon GPU, while other libraries like LieTorch [60] implement customized CUDA kernels and only support 1$^{\text{st}}$-order gradient. (**2**) `LieTensor` supports parallel computing of gradient with the `vmap` operator, which allows it to compute Jacobian matrices much faster. (**3**) Theseus [49] adopts rotation matrices, which require more memory, while PyPose uses quaternions, which only require storing four scalars, have no gimbal lock issues, and have better numerical properties. (**4**) Libraries such as LieTorch, JaxLie [71], and Theseus only support Lie groups, while PyPose supports both Lie groups and Lie algebras. As a result, one can directly call the `Exp` and `Log` maps
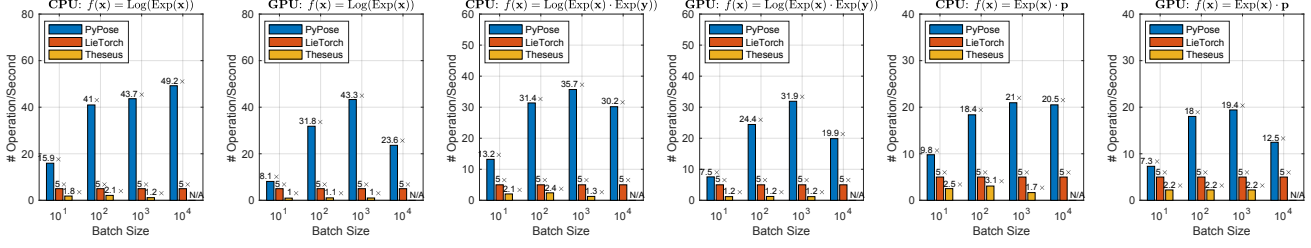
Figure 1. Efficiency comparison of dense Jacobian for Lie group operations on CPU/GPU, where "N/A" means performance is unavailable with that batch size. The performance of Theseus [49] is computed with its `AutoDiffCostFunction` method. We take LieTorch performance as $5\times$ to be the baseline since Theseus runs out of memory at the batch size of 10^4.

from a `LieTensor` instance, which is more flexible and user-friendly. Moreover, the gradient with respect to both types can be automatically calculated and back-propagated.

For other types of supported 3D transformations, the reader may refer to Table 1 for more details. The readers may find a list of supported `LieTensor` operations in the documentation.[1] A sample code for how to use a `LieTensor` is given in the supplementary and PyPose tutorials.[2]

3.2. `Module` **and** `Function`

PyPose leverages the existing concept of `Function` and `Module` in PyTorch to implement differentiable robotics-related functionalities. Concretely, a `Function` performs a specific computation given inputs and returns outputs; a `Module` has the same functionality as a `Function`, but it also stores data, such as `Parameter`, which is of type `Tensor` or `LieTensor`, that can be updated by an optimizer as discussed in the next section.

PyPose provides many useful modules, such as the system transition function, model predictive control (MPC), Kalman filter, and IMU preintegration.

A list of supported `Module` can be found in the documentation.[3] Users can easily integrate them into their own systems and perform a specific task, e.g., a `System` module can be used by both `EKF` and `MPC`. A few practical examples in SLAM, planning, control, and inertial navigation are presented in Section 4.3.

3.3. `Optimizer`

To enable end-to-end learning with physics-based optimization, we need to integrate general optimizers beyond the basic gradient-based methods such as SGD [52] and Adam [35], since many practical problems in robotics require more advanced optimization such as constrained or 2nd-order optimization [8]. In this section, we take a robust non-linear least square problem as an example and present the intuition behind the optimization-oriented interfaces of PyPose, including `solver`, `kernel`, `corrector`, and `strategy` for using the 2nd-order Levenberg-Marquardt

---

(`LM`) optimizer. Consider a weighted least square problem:

$$\min_{\boldsymbol{\theta}} \sum_i \left(\boldsymbol{f}(\boldsymbol{\theta}, \boldsymbol{x}_i) - \boldsymbol{y}_i\right)^T \mathbf{W}_i \left(\boldsymbol{f}(\boldsymbol{\theta}, \boldsymbol{x}_i) - \boldsymbol{y}_i\right), \quad (1)$$

where $\boldsymbol{f}(\cdot)$ is a regression model (`Module`), $\boldsymbol{\theta} \in \mathbb{R}^n$ is the parameters to be optimized, $\boldsymbol{x}_i \in \mathbb{R}^m$ is an input, $\boldsymbol{y}_i \in \mathbb{R}^d$ is the target, $\mathbf{W}_i \in \mathbb{R}^{d \times d}$ is a square information matrix.

The solution to (1) of an `LM` algorithm is computed by iteratively updating an estimate $\boldsymbol{\theta}_t$ via $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} + \boldsymbol{\delta}_t$, where the update step $\boldsymbol{\delta}_t$ is computed as:

$$\sum_i \left(\mathbf{H}_i + \lambda \cdot \text{diag}(\mathbf{H}_i)\right) \boldsymbol{\delta}_t = -\sum_i \mathbf{J}_i^T \mathbf{W}_i \mathbf{R}_i, \quad (2)$$

where $\mathbf{R}_i = \boldsymbol{f}(\boldsymbol{\theta}_{t-1}, \boldsymbol{x}_i) - \boldsymbol{y}_i$ is the i-th residual error, $\mathbf{J}_i$ is the Jacobian of $\boldsymbol{f}$ computed at $\boldsymbol{\theta}_{t-1}$, $\mathbf{H}_i$ is an approximate Hessian matrix computed as $\mathbf{H}_i = \mathbf{J}_i^T \mathbf{W}_i \mathbf{J}_i$, and λ is a damping factor. To find step $\boldsymbol{\delta}_t$, one needs a linear `solver`:

$$\mathbf{A} \cdot \boldsymbol{\delta}_t = \boldsymbol{b}, \quad (3)$$

where $\mathbf{A} = \sum_i \left(\mathbf{H}_i + \lambda \cdot \text{diag}(\mathbf{H}_i)\right)$, $\boldsymbol{b} = -\sum_i \mathbf{J}_i^T \mathbf{W}_i \mathbf{R}_i$. In practice, the square matrix $\mathbf{A}$ is often positive-definite, so we could leverage standard linear solvers such as Cholesky. If the Jacobian $\mathbf{J}_i$ is large and sparse, we may also use sparse solvers such as sparse Cholesky [15] or preconditioned conjugate gradient (PCG) [33] solver.

In practice, one often introduces robust `kernel` functions $\rho : \mathbb{R} \mapsto \mathbb{R}$ into (1) to reduce the effect of outliers:

$$\min_{\boldsymbol{\theta}} \sum_i \rho \left(\mathbf{R}_i^T \mathbf{W}_i \mathbf{R}_i\right), \quad (4)$$

where ρ is designed to down-weigh measurements with large residuals $\mathbf{R}_i$. In this case, we need to adjust (2) to account for the presence of the robust kernel. A popular way is to use Triggs' correction [63], which is also adopted by the Ceres [2] library. However, it needs 2nd-order derivative of the kernel function ρ, which is always negative. This can lead 2nd-order optimizers including LM to be unstable [63]. Alternatively, PyPose introduces `FastTriggs` corrector, which is faster yet more stable than Triggs' correction by only involving the 1st-order derivative:

$$\mathbf{R}_i^{\rho} = \sqrt{\rho'(c_i)}\mathbf{R}_i, \quad \mathbf{J}_i^{\rho} = \sqrt{\rho'(c_i)}\mathbf{J}_i, \quad (5)$$
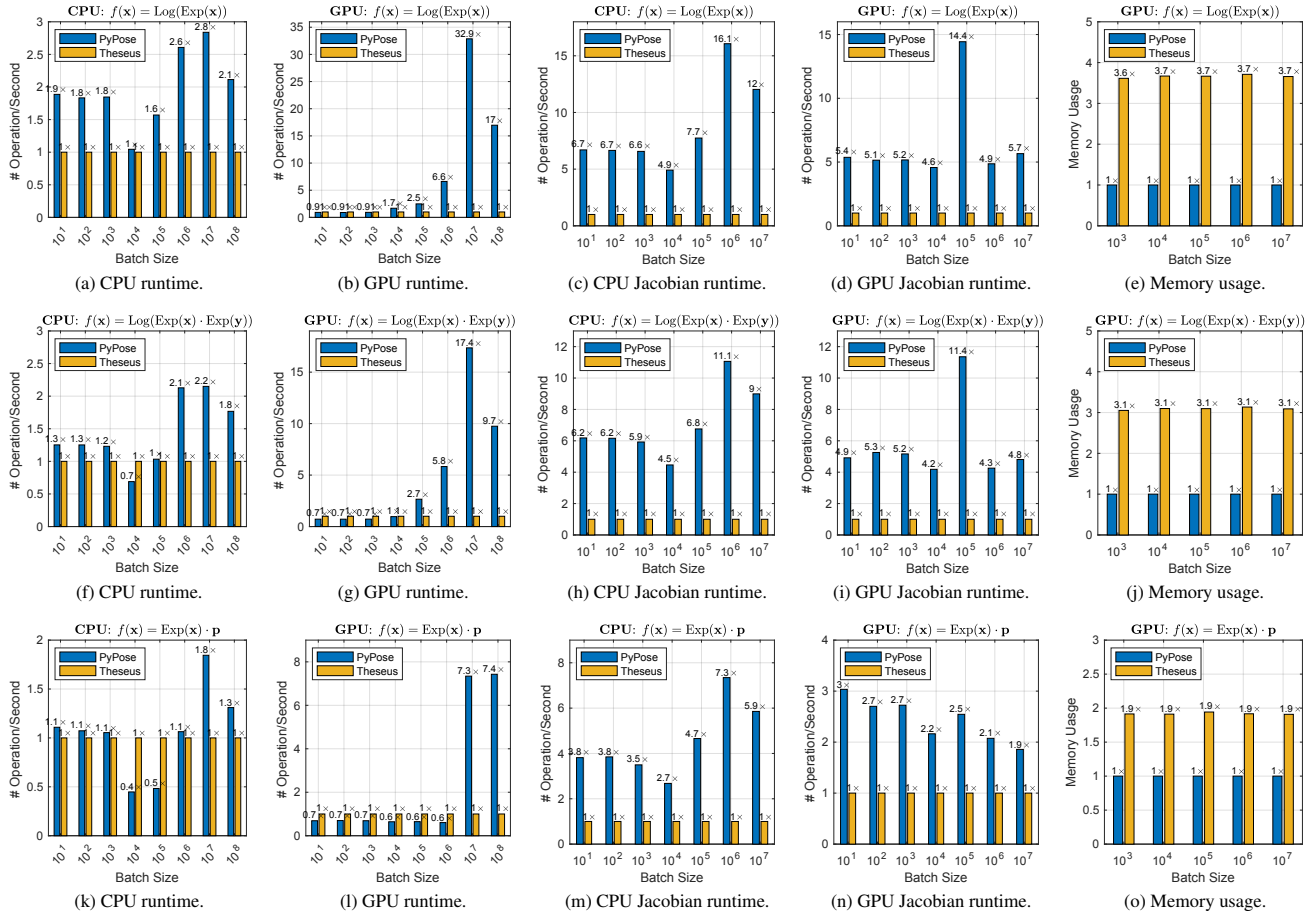
Figure 2. Efficiency and memory comparison of batched Lie group operations (we take Theseus [49] performance as 1×). Note that Theseus provides multiple ways to calculate Jacobian, and we adopted `AutoDiffCostFunction` for comparison. This is because this method is used in its optimizers and can reflect the performance of an optimization problem. Theseus also provides analytical Jacobian for some single operators, which could be faster, but one needs to manually compose the Jacobian of each operator for user-defined functions.

where $c_i = \mathbf{R}_i^T \mathbf{W}_i \mathbf{R}_i$, $\mathbf{R}_i^\rho$ and $\mathbf{J}_i^\rho$ are the corrected model residual and Jacobian due to the introduction of kernel functions, respectively. More details about `FastTriggs` and its proof can be found in the documentation.[4]

A simple `LM` optimizer may not converge to the global optimum if the initial guess is too far from the optimum. For this reason, we often need other strategies such as adaptive damping, dog leg, and trust region methods [41] to restrict each step, preventing it from stepping "too far". To adopt those strategies, one may simply pass a `strategy` instance, e.g., `TrustRegion`, to an optimizer. In summary, PyPose supports easy extensions for the aforementioned algorithms by simply passing `optimizer` arguments to their constructor, including `solver`, `strategy`, `kernel`, and `corrector`. A list of available algorithms and examples can be found in the documentation.[5] A sample code showing how to use an optimizer is provided in the supplementary.

---

[4]https://pypose.org/docs/main/generated/pypose.optim.corrector.FastTriggs
[5]https://pypose.org/docs/main/optim

4. Experiments

In this section, we showcase PyPose's performance and present practical examples of its use for vision and robotics.

4.1. Runtime Efficiency

We report the runtime efficiency of `LieTensor` and compare it with the state-of-the-art libraries, including LieTorch [60] and Theseus [49]. Since PyPose and LieTorch use quaternions and Theseus uses rotation matrices to represent transformation, they are not directly comparable. To have an extensive and fair comparison, we construct three widely-used functions with the same input/output shapes for all libraries, including exponential and logarithm mapping $f_1(\mathbf{x}) = \text{Log}(\text{Exp}(\mathbf{x}))$, $\mathbf{x} \in \mathfrak{so}(3)$, rotation composition $f_2(\mathbf{x}) = \text{Log}(\text{Exp}(\mathbf{x}) \cdot \text{Exp}(\mathbf{y}))$, $\mathbf{x}, \mathbf{y} \in \mathfrak{so}(3)$, and rotating points $f_3(\mathbf{x}) = \text{Exp}(\mathbf{x}) \cdot \mathbf{p}$, $\mathbf{x} \in \mathfrak{so}(3), \mathbf{p} \in \mathbb{R}^3$. We take the number of operations per second for the operation f_i and its Jacobian $J_i = \frac{\partial f_i}{\partial \mathbf{x}}$ as the metric.

We first report the performance of calculating dense Jaco-

Table 2. The final error and runtime of optimizers.

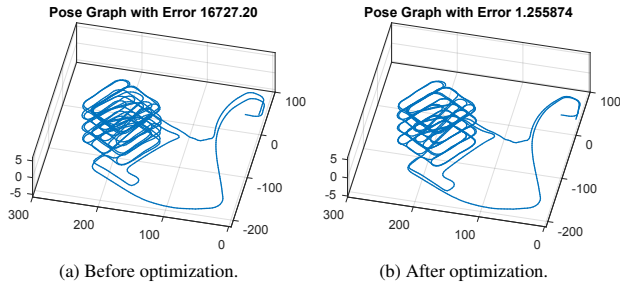| Batch	10^1		10^2		10^3		10^4	
	Error	Time	Error	Time	Error	Time	Error	Time
SGD	5.10E+1	0.46	4.96E+2	0.95	4.09E+3	1.45	6.86E+3	31.5
Adam	9.81E+1	2.96	1.25E+3	9.28	1.13E+4	16.9	1.16E+5	40.5
LM	**5.96E-5**	**0.57**	**5.80E-4**	**0.60**	**3.72E-3**	**0.92**	**3.77E-2**	**51.6**

(a) Before optimization. (b) After optimization.

Figure 3. A PGO example "Garage" using the LM optimizer. PyPose has the same final error as Ceres [2] and GTSAM [20].

bian of operator f_1 in Figure 1. PyPose is up to $37.9\times$ faster on a CPU than Theseus and up to $8.8\times$ faster than LieTorch. On a GPU, PyPose achieves up to $42.9\times$ faster than Theseus and up to $8.6\times$ faster than LieTorch.

Figure 2 shows the performance of calculating batched operations, where the function inputs $\mathbf{x}$ are assumed to be independent, and the batched Jacobian can be calculated in parallel. In this case, we have to use "loop" for calculating Jacobian with LieTorch; thus, we don't compare with it for fairness, as loop in Python is slow. On a CPU, PyPose is up to $1.8\times$, $2.8\times$, and $2.2\times$ faster than Theseus for the three operations, respectively. On a GPU, PyPose shows even better performance, up to $7.4\times$, $32.9\times$, and $17.4\times$ faster. For computing the batched Jacobian, PyPose achieves up to $7.3\times$, $16.1\times$, and $11.1\times$ on a CPU and $3.0\times$, $14.4\times$, and $11.4\times$ faster on a GPU for the three operations, respectively.

We noticed that PyPose only needs about $^1/_4$ to $^1/_2$ memory space compared to Theseus, which further demonstrates its efficiency. We report a maximum batch size of 10^7 because bigger one takes too much memory to handle for our platform. It is worth mentioning that the runtime usually has a fluctuation within 10%, and we report the average performance of multiple tests. A relative performance drop for certain batch size is seen (Figure 2k), which might be caused by cache miss or imbalanced Linux OS resource managing.

4.2. Optimization

We next report the performance of PyPose's 2nd-order optimizer Levenberg-Marquardt (LM) and compare with PyTorch's 1st-order optimizers such as SGD and Adam. Specifically, we construct a Module to learn transformation inverse, as provided in the PyPose tutorials and report their performance in Table 2. Note that PyPose provides analytical solutions for inversion, and this module is only to benchmark the optimizers. It can be seen that this problem

Table 3. Mean matching accuracy improves after self-supervised training on TartanAir [69].

Difficulty	# Sequence	Original	Trained	Improvement
Easy	19	21.37%	32.04%	+ 49.94%
Hard	19	18.10%	24.50%	+ 35.33%
Overall	38	19.74%	28.27%	+ 43.24%

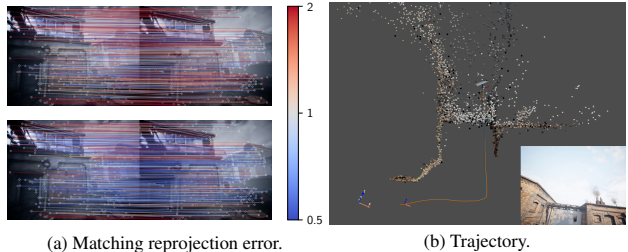(a) Matching reprojection error. (b) Trajectory.

Figure 4. An example of visual SLAM using PyPose. (a) Matching reprojection error (pixels) (top) is improved after self-supervised tuning using PyPose (bottom). (b) Input sequence (bottom right), resulting trajectory, and point cloud. The estimated and ground truth trajectories are shown in blue and orange, respectively.

is quite challenging for 1st-order optimizers, while 2nd-order optimizers can find the best solution with a much lower final error quickly, which verifies the effectiveness of PyPose's implementation.

One of the drawbacks of 2nd-order optimizers is that they need to compute Jacobians, which requires significant memory and may increase the runtime. Leveraging the sparsity of Jacobian matrices could significantly alleviate this challenge [72]. One example is pose graph optimization (PGO), which is challenging for 1st-order optimizers and also requires pose inversion for every edge in the graph [7]. We report the performance of PGO using the LM optimizer with a TrustRegion [56] strategy in Figure 3, where the sample is from the g^2o dataset [14]. PyPose achieves the same final error as Ceres [2] and GTSAM [20]. An example using PGO for inertial navigation is presented in Section 4.3.4.

4.3. Practical Examples
4.3.1 SLAM

To showcase PyPose's ability to bridge learning and optimization tasks, we develop a method for learning the SLAM front-end in a self-supervised manner. Due to difficulty in collecting accurate camera pose and/or dense depth, few dataset options are available to researchers for training correspondence models offline. To this end, we leverage the estimated poses and landmark positions as pseudo-labels for supervision. At the front end, we employ two recent works, SuperPoint [21] and CAPS [68], for feature extraction and matching, respectively. Implemented with the LM optimizer of PyPose, the backend performs bundle adjustment and back-propagates the final projection error to update the CAPS feature matcher. In the experiment, we start with a

(a) Planning runtime.

(b) Executed trajectory.
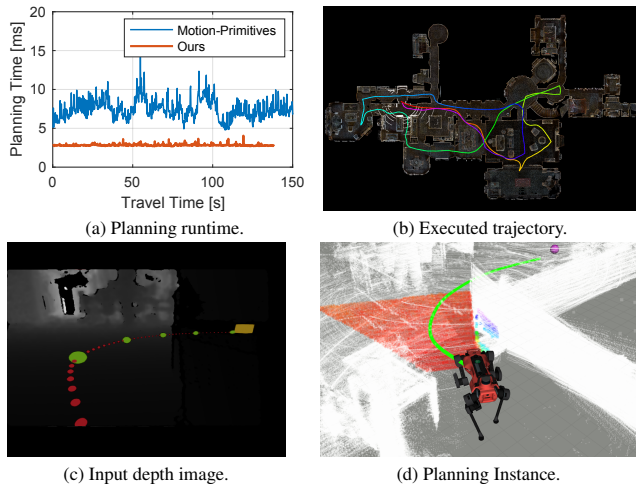
(c) Input depth image.

(d) Planning Instance.

Figure 5. An example of an end-to-end planning using PyPose library. (a) and (b) show a local planning experiment conducted in the Matterport3D simulated environment. (a) shows the comparison plot of the algorithm planning time. (b) shows the trajectory executed by the proposed method and the outline of the environment. (c) and (d) show an instance of the real-world experiment conducted with ANYmal-legged robot. A kinodynamically feasible trajectory, shown in (d), is directly generated from the input depth image (c). The red dots in (c) show the trajectory projected back into the image frame for visualization.

CAPS network pretrained on MegaDepth [39] and finetune on the TartanAir [69] dataset with only image input. We segment each TartanAir sequence into batches of 6 frames and train the CAPS network with the estimated poses. To assess the effect of unsupervised training, we evaluate the CAPS model on the two-frame matching task and report the results in Table 3. It can be seen that the matching accuracy (reprojection error ≤ 1 pixel) increased by up to 50% on unseen sequences after self-supervised training. We show a visual comparison of the reprojection error distribution before and after training in Figure 4a and the resulting trajectory and point cloud on a test sequence in Figure 4b. While the original model quickly loses track, the trained model runs to completion with an ATE of 0.63 m. This verifies the feasibility of PyPose for optimization in the SLAM backend.

4.3.2 Planning

The PyPose library has been used to develop a novel end-to-end planning policy that maps the depth sensor inputs directly into kinodynamically feasible trajectories. Here, we combined learning and trajectory optimization to achieve efficient planning and enforce smoothness and kinodynamic feasibility for the output paths. The SE3 LieTensor of PyPose library is utilized to form a differential layer for trajectory optimization. As shown in Figure 5a, our method achieves around $3\times$ speedup on average compared to a traditional planning framework [13], which utilizes a combined pipeline

(a) Model loss.

(b) Trajectory loss.

(c) Overall runtime.
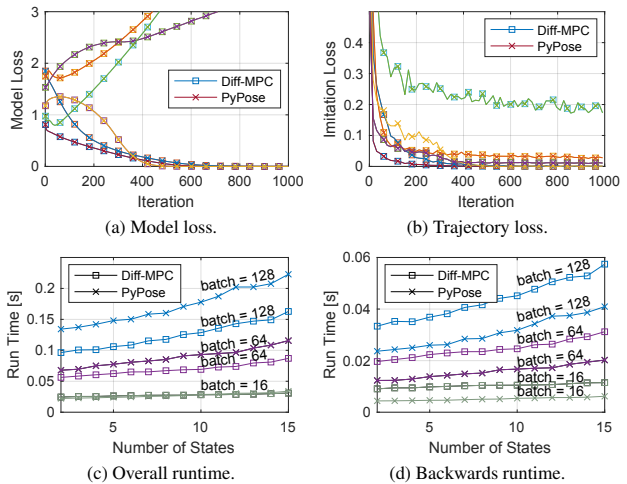
(d) Backwards runtime.

Figure 6. An example of MPC with imitation learning using PyPose library. (a) and (b) show the learning performance comparison of PyPose and Diff-MPC [5]. The two methods achieve the same learning performance. (c) and (d) show the computation cost comparison, where different colors represent different batch sizes.

of geometry-based terrain analysis and motion-primitives-based planning [73]. The experiment is conducted in the Matterport3D [16] environment. The robot follows 20 manually inputted waypoints for global guidance but plans fully autonomously by searching feasible paths and avoiding local obstacles. The trajectory executed by the proposed method is shown in Figure 5b. The efficiency of this method benefits from both the end-to-end planning pipeline and the efficiency of the PyPose library for training and deployment. Furthermore, this end-to-end policy has been integrated and tested on a real-legged robot, ANYmal. A planning instance during the field test is shown in Figure 5d using the current depth observation, shown in Figure 5c.

4.3.3 Control

PyPose integrates the dynamics and control tasks into the end-to-end learning framework. We demonstrate this capability using a learning-based MPC for an imitation learning problem described in Diff-MPC [5], where both the expert and learner employ a linear-quadratic regulator (LQR) that share all information except for linear system parameters and the learner tries to recover the dynamics using only expert controls. We treat MPC as a generic policy class with parameterized cost functions and dynamics, which can be learned by automatic differentiating (AD) through LQR optimization. Unlike the existing method of simply unrolling and differentiating through the optimization iterations [57], which incurs significant computational costs for both the forward and backward passes, or the method of computing the derivative analytically by solving the KKT condition of the LQR problem at a fix point [5], which is problem-specific, in PyPose, we use a *problem-agnostic* AD implementation that

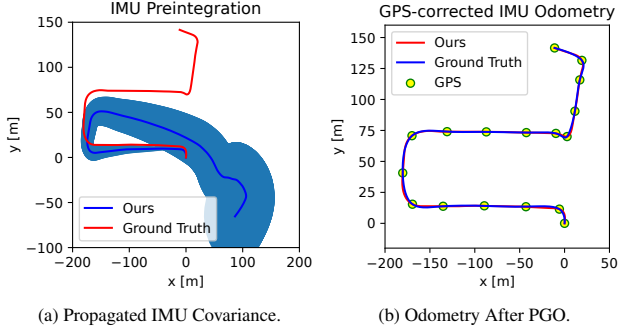(a) Propagated IMU Covariance.   (b) Odometry After PGO.

Figure 7. Examples of IMU integration and the IMU-centric PGO on a sequence from the KITTI dataset. Compared with the raw signal, the optimized result is very close to the ground truth.

applies one extra optimization iteration at the optimal point in the forward pass to enable the automatic gradient computation for the backward pass. Thus, our method does not require the computation of gradients for the entire unrolled chain or analytical derivative for the specific problem.

The preliminary results are shown in Figure 6 for a benchmarking problem [5]. Figure 6a and 6b show the model and trajectory losses, respectively, of four trajectories with randomly sampled initial conditions. We can see both methods achieve the same learning performance for different sample dynamics. We then compared the two methods' overall runtime and backward time. Our backward time is always shorter because the method in [5] needs to solve an additional iteration LQR in the backward pass. Overall, when the system is low-dimensional, our method shows a faster learning process. As the system dimension increases, the runtime of our method gets longer. The overhead is due to the extra evaluation iteration in the forward pass needs to create and maintain variables for the backward pass. However, we noticed that the overall runtime difference is not significant. The advantage of our method is more prominent as more complex optimization problems are involved.

4.3.4   Inertial Navigation

IMU preintegration [25] is one of the most popular strategies for many inertial navigation systems, e.g., VIO [50], LIO [77], and multi-sensor fusion [76]. A growing trend consists in leveraging the IMU's physical prior in learning-based inertial navigation, e.g., IMU odometry network [17], data-driven calibration [10, 74], and EKF-based odometry [40, 55]. To boost future research, PyPose develops an `IMUPreintegrator` module for differentiable IMU preintegration with covariance propagation. It supports batched operation and integration on the manifold space.

In Figure 7a, we present an IMU integration example on the KITTI dataset [26], where the blue shadows indicate the propagated IMU covariance. In this example, the trajectory drifts due to the accumulated sensor noise. However, as shown Figure 7b, it can be improved by PGO with GPS
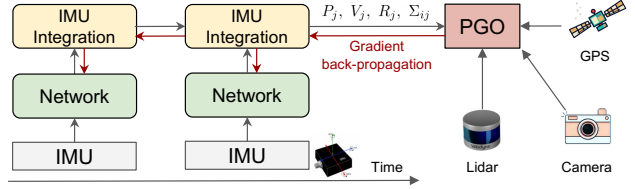

Figure 8. The framework of IMU calibration network using Py-Pose's `IMUPreintegrator` with `LieTensor`.

Table 4. The results of IMU pre-integration in $1\mathrm{s}$ (200 frames) on the testing sequences of the EuRoC dataset.

| Seq | MH_02_easy | | MH_04_hard | | V1_03_hard | | V2_02_med | | V1_01_easy | |
	pos	rot	pos	rot	pos	rot	pos	rot	pos	rot
Preinte [25]	0.050	0.659	0.047	0.652	0.062	0.649	0.041	0.677	0.161	0.649
Ours	**0.012**	**0.012**	**0.017**	**0.016**	**0.025**	**0.036**	**0.038**	**0.054**	**0.124**	**0.056**

(loop closing) for every $5\mathrm{s}$ segment, which also verifies the effectiveness of our 2$^\text{nd}$-order optimizers.

We next train an IMU calibration network shown in Figure 8 to denoise the IMU signals, after which we integrate the denoised signal and calculate IMU's state expressed with `LieTensor` including position, orientation, and velocity. To learn the parameters in the network, we supervise the integrated pose and back-propagate the gradient through the integrator module. To validate the effectiveness of this method, we train the network on 5 sequences and evaluate on 5 testing sequences from EuRoC dataset [11] with root-mean-square-error (RMSE) on position (m) and rotation (rad). As shown in Table 4, our method significantly improved orientation and translation compared to the traditional method [25]. This experiment shows that PyPose can benefit future research on the learning-based multi-modal SLAM and inertial navigation systems.

5. Discussion

We present PyPose, a novel pythonic open source library to boost the development of the next generation of robotics. PyPose enables end-to-end learning with physics-based optimization and provides an imperative programming style for the convenience of real-world robotics applications. It is designed to be easy-interpretable, user-friendly, and efficient with a tidy and well-organized architecture. To the best of our knowledge, PyPose is one of the first comprehensive pythonic libraries covering most sub-fields of robotics where physics-based optimization is involved.

Despite its promising features, PyPose is still in its early stage; its overall stability is not yet mature for practical vision and robotic applications compared to PyTorch. Moreover, its efficiency is still not comparable with C(++)-based libraries, although it provides an alternative solution for robot learning-based applications. In the future, we will keep adding more features, including sparse block tensors, sparse Jacobian computation, and constrained optimization.

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. 2

[2] Sameer Agarwal, Keir Mierle, and The Ceres Solver Team. Ceres Solver, 3 2022. 2, 3, 4, 6

[3] A. Agrawal, B. Amos, S. Barratt, S. Boyd, S. Diamond, and Z. Kolter. Differentiable convex optimization layers. In *Advances in Neural Information Processing Systems*, 2019. 2

[4] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv e-prints*, pages arXiv–1605, 2016. 2

[5] Brandon Amos, Ivan Jimenez, Jacob Sacks, Byron Boots, and J Zico Kolter. Differentiable mpc for end-to-end planning and control. *Advances in neural information processing systems*, 31, 2018. 7, 8

[6] Joel AE Andersson, Joris Gillis, Greg Horn, James B Rawlings, and Moritz Diehl. Casadi: a software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, 11(1):1–36, 2019. 3

[7] Fang Bai, Teresa Vidal-Calleja, and Giorgio Grisetti. Sparse pose graph optimization in cycle space. *IEEE Transactions on Robotics*, 37(5):1381–1400, 2021. 6

[8] Timothy D Barfoot. *State estimation for robotics*. Cambridge University Press, 2017. 4

[9] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. 2

[10] Martin Brossard, Silvere Bonnabel, and Axel Barrau. Denoising imu gyroscopes with deep learning for open-loop attitude estimation. *IEEE Robotics and Automation Letters*, 5(3):4796–4803, 2020. 8

[11] Michael Burri, Janosch Nikolic, Pascal Gohl, Thomas Schneider, Joern Rehder, Sammy Omari, Markus W Achtelik, and Roland Siegwart. The euroc micro aerial vehicle datasets. *The International Journal of Robotics Research*, 35(10):1157–1163, 2016. 8

[12] Carlos Campos, Richard Elvira, Juan J Gómez Rodríguez, José MM Montiel, and Juan D Tardós. Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam. *IEEE Transactions on Robotics*, 37(6):1874–1890, 2021. 1

[13] Chao Cao, Hongbiao Zhu, Fan Yang, Yukun Xia, Howie Choset, Jean Oh, and Ji Zhang. Autonomous exploration development environment and the planning algorithms. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 8921–8928. IEEE, 2022. 7

[14] Luca Carlone, David M Rosen, Giuseppe Calafiore, John J Leonard, and Frank Dellaert. Lagrangian duality in 3d slam: Verification techniques and optimal solutions. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 125–132. IEEE, 2015. 6

[15] Jeffrey N Chadwick and David S Bindel. An efficient solver for sparse linear systems based on rank-structured cholesky factorization. *arXiv preprint arXiv:1507.05593*, 2015. 4

[16] Angel Chang, Angela Dai, Thomas Funkhouser, Maciej Halber, Matthias Niessner, Manolis Savva, Shuran Song, Andy Zeng, and Yinda Zhang. Matterport3D: Learning from RGB-D data in indoor environments. *International Conference on 3D Vision (3DV)*, 2017. 7

[17] Changhao Chen, Xiaoxuan Lu, Andrew Markham, and Niki Trigoni. Ionet: Learning to cure the curse of drift in inertial odometry. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018. 8

[18] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015. 2

[19] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Technical report, Idiap, 2002. 2

[20] Frank Dellaert. Factor graphs and gtsam: A hands-on introduction. Technical report, Georgia Institute of Technology, 2012. 2, 3, 6

[21] Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. Superpoint: Self-supervised interest point detection and description. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 224–236, 2018. 6

[22] Iain Dunning, Joey Huchette, and Miles Lubin. Jump: A modeling language for mathematical optimization. *SIAM review*, 59(2):295–320, 2017. 3

[23] Kamak Ebadi, Lukas Bernreiter, Harel Biggie, Gavin Catt, Yun Chang, Arghya Chatterjee, Christopher E Denniston, Simon-Pierre Deschênes, Kyle Harlow, Shehryar Khattak, et al. Present and future of slam in extreme underground environments. *arXiv preprint arXiv:2208.01787*, 2022. 1

[24] Xu Fang, Chen Wang, Lihua Xie, and Jie Chen. Cooperative pursuit with multi-pursuer and one faster free-moving evader. *IEEE Transactions on Cybernetics*, 2020. 1

[25] Christian Forster, Luca Carlone, Frank Dellaert, and Davide Scaramuzza. Imu preintegration on manifold for efficient visual-inertial maximum-a-posteriori estimation, 2015. 8

[26] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, 2013. 8

[27] Markus Giftthaler, Michael Neunert, Markus Stäuble, and Jonas Buchli. The control toolbox—an open-source c++ library for robotics, optimal and model predictive control. In *2018 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pages 123–129. IEEE, 2018. 2, 3

[28] Giorgio Grisetti, Rainer Kümmerle, Hauke Strasdat, and Kurt Konolige. g2o: A general framework for (hyper) graph opti-

mization. In *Proceedings of the IEEE international conference on robotics and automation (ICRA), Shanghai, China*, pages 9–13, 2011. 2, 3

[29] Gaël Guennebaud, Benoit Jacob, et al. Eigen. *URl: http://eigen. tuxfamily. org*, 3, 2010. 2

[30] Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. In *International conference on machine learning*, pages 2555–2565, 2019. 1

[31] Ankur Handa, Michael Bloesch, Viorica Pătrăucean, Simon Stent, John McCormac, and Andrew Davison. gvnn: Neural network library for geometric computer vision. In *Computer Vision–ECCV 2016 Workshops: Amsterdam, The Netherlands, October 8-10 and 15-16, 2016, Proceedings, Part III 14*, pages 67–82. Springer, 2016. 2

[32] William E Hart, Carl D Laird, Jean-Paul Watson, David L Woodruff, Gabriel A Hackebeil, Bethany L Nicholson, John D Siirola, et al. *Pyomo-optimization modeling in python*, volume 67. Springer, 2017. 3

[33] Magnus R Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving. *Journal of research of the National Bureau of Standards*, 49(6):409, 1952. 4

[34] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *22nd ACM international conference on Multimedia*, pages 675–678, 2014. 2

[35] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 4

[36] Jonas Koenemann, Giovanni Licitra, Mustafa Alp, and Moritz Diehl. Openocl–open optimal control library, 2017. 3

[37] Ian Lenz, Ross A Knepper, and Ashutosh Saxena. Deepmpc: Learning deep latent features for model predictive control. In *Robotics: Science and Systems*, volume 10, 2015. 1

[38] Bowen Li, Chen Wang, Pranay Reddy, Seungchan Kim, and Sebastian Scherer. Airdet: Few-shot detection without fine-tuning for autonomous exploration. In *European Conference on Computer Vision (ECCV)*, 2021. 1

[39] Zhengqi Li and Noah Snavely. Megadepth: Learning single-view depth prediction from internet photos. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2041–2050, 2018. 7

[40] Wenxin Liu, David Caruso, Eddy Ilg, Jing Dong, Anastasios I Mourikis, Kostas Daniilidis, Vijay Kumar, and Jakob Engel. Tlio: Tight learned inertial odometry. *IEEE Robotics and Automation Letters*, 5(4):5653–5660, 2020. 8

[41] Manolis LA Lourakis and Antonis A Argyros. Is levenberg-marquardt the most efficient optimization algorithm for implementing bundle adjustment? In *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, volume 2, pages 1526–1531. IEEE, 2005. 5

[42] James Malcolm, Pavan Yalamanchili, Chris McClanahan, Vishwanath Venugopalakrishnan, Krunal Patel, and John Melonakos. Arrayfire: a gpu acceleration platform. In *Modeling and simulation for defense systems and applications VII*, volume 8403, pages 49–56. SPIE, 2012. 2

[43] MATLAB. *version 7.10.0 (R2010a)*. The MathWorks Inc., Natick, Massachusetts, 2010. 2

[44] Chenlin Meng, Yang Song, Wenzhe Li, and Stefano Ermon. Estimating high order gradients of the data distribution by denoising. *Advances in Neural Information Processing Systems*, 34:25359–25369, 2021. 2

[45] Shervin Minaee, Yuri Y Boykov, Fatih Porikli, Antonio J Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos. Image segmentation using deep learning: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 2021. 1

[46] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006. 2

[47] NN Open. An open source neural networks c++ library. *URL: http://opennn.cimne.com (2016)*, 2016. 2

[48] 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, 2019. 2

[49] Luis Pineda, Taosha Fan, Maurizio Monge, Shobha Venkataraman, Paloma Sodhi, Ricky TQ Chen, Joseph Ortiz, Daniel DeTone, Austin Wang, Stuart Anderson, Jing Dong, Brandon Amos, and Mustafa Mukadam. Theseus: A Library for Differentiable Nonlinear Optimization. *Advances in Neural Information Processing Systems*, 2022. 2, 3, 4, 5

[50] Tong Qin, Peiliang Li, and Shaojie Shen. Vins-mono: A robust and versatile monocular visual-inertial state estimator. *IEEE Transactions on Robotics*, 34(4):1004–1020, 2018. 8

[51] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, page 5. Kobe, Japan, 2009. 2

[52] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951. 4

[53] 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

[54] Ioan A Sucan, Mark Moll, and Lydia E Kavraki. The open motion planning library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, 2012. 2

[55] Scott Sun, Dennis Melamed, and Kris Kitani. Idol: Inertial deep orientation-estimation and localization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 6128–6137, 2021. 8

[56] Wenyu Sun and Ya-Xiang Yuan. *Optimization theory and methods: nonlinear programming*, volume 1. Springer Science & Business Media, 2006. 6

[57] Aviv Tamar, Garrett Thomas, Tianhao Zhang, Sergey Levine, and Pieter Abbeel. Learning from the hindsight plan—episodic mpc improvement. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 336–343. IEEE, 2017. 7

[58] Russ Tedrake and the Drake Development Team. Drake: Model-based design and verification for robotics, 2019. 3

[59] Zachary Teed and Jia Deng. DROID-SLAM: Deep Visual SLAM for Monocular, Stereo, and RGB-D Cameras. *Advances in neural information processing systems*, 2021. 1

[60] Zachary Teed and Jia Deng. Tangent space backpropagation for 3d transformation groups. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10338–10347, 2021. 1, 2, 3, 5

[61] Zachary Teed, Lahav Lipson, and Jia Deng. Deep patch visual odometry. *arXiv preprint arXiv:2208.04726*, 2022. 1

[62] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, pages 1–6, 2015. 2

[63] 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. 4

[64] Andreas Wächter and Lorenz T Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical programming*, 106(1):25–57, 2006. 3

[65] Chen Wang, Wenshan Wang, Yuheng Qiu, Yafei Hu, and Sebastian Scherer. Unsupervised online learning for robotic interestingness with visual memory. *IEEE Transactions on Robotics (TRO)*, 2021. 1

[66] Chen Wang, Jianfei Yang, Lihua Xie, and Junsong Yuan. Kervolutional neural networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 31–40, 2019. 1

[67] Chen Wang, Junsong Yuan, and Lihua Xie. Non-iterative slam. In *International Conference on Advanced Robotics (ICAR)*, pages 83–90. IEEE, 2017. 1

[68] Qianqian Wang, Xiaowei Zhou, Bharath Hariharan, and Noah Snavely. Learning feature descriptors using camera pose supervision. In *European Conference on Computer Vision*, pages 757–774. Springer, 2020. 6

[69] 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. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020. 6, 7

[70] Fan Yang, Chao Cao, Hongbiao Zhu, Jean Oh, and Ji Zhang. Far planner: Fast, attemptable route planner using dynamic visibility update. *arXiv preprint arXiv:2110.09460*, 2021. 1

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

[72] Christopher Zach. Robust bundle adjustment revisited. In *European Conference on Computer Vision*, pages 772–787. Springer, 2014. 6

[73] Ji Zhang, Chen Hu, Rushat Gupta Chadha, and Sanjiv Singh. Falco: Fast likelihood-based collision avoidance with extension to human-guided navigation. *Journal of Field Robotics*, 37(8):1300–1313, 2020. 7

[74] Ming Zhang, Mingming Zhang, Yiming Chen, and Mingyang Li. Imu data processing for inertial aided navigation: A recurrent neural network based approach. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3992–3998. IEEE, 2021. 8

[75] Shibo Zhao, Peng Wang, Hengrui Zhang, Zheng Fang, and Sebastian Scherer. Tp-tio: A robust thermal-inertial odometry with deep thermalpoint. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4505–4512. IEEE, 2020. 1

[76] Shibo Zhao, Hengrui Zhang, Peng Wang, Lucas Nogueira, and Sebastian Scherer. Super odometry: Imu-centric lidar-visual-inertial estimator for challenging environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 8729–8736. IEEE, 2021. 1, 8

[77] Xingxing Zuo, Yulin Yang, Patrick Geneva, Jiajun Lv, Yong Liu, Guoquan Huang, and Marc Pollefeys. Lic-fusion 2.0: Lidar-inertial-camera odometry with sliding-window plane-feature tracking. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020. 8

A. Sample Code of `LieTensor`

The following code sample shows how to rotate random points and compute the gradient of batched rotation.

```
>>> import torch, pypose as pp

>>> # A random so(3) LieTensor
>>> r = pp.randn_so3(2, requires_grad=True)
so3Type LieTensor:
tensor([[ 0.1606,  0.0232, -1.5516],
        [-0.0807, -0.7184, -0.1102]],
       requires_grad=True)

>>> R = r.Exp() # Equivalent to: R = pp.Exp(r)
SO3Type LieTensor:
tensor([[ 0.0724,  0.0104, -0.6995,  0.7109],
        [-0.0395, -0.3513, -0.0539,  0.9339]],
       grad_fn=<AliasBackward0>)

>>> p = R @ torch.randn(3) # Rotate random point
tensor([[ 0.8045, -0.8555,  0.5260],
        [ 0.3502,  0.8337,  0.9154]],
       grad_fn=<ViewBackward0>)

>>> p.sum().backward()      # Compute gradient
>>> r.grad                  # Print gradient
tensor([[-0.7920, -0.9510,  1.7110],
        [-0.2659,  0.5709, -0.3855]])
```

B. Sample Code of `Optimizer`

We show how to estimate batched transform inverse by a 2nd-order optimizer. Two usage options for a `scheduler` are provided, each of which can work independently.

```
>>> from torch import nn
>>> import torch, pypose as pp
>>> from pypose.optim import LM
>>> from pypose.optim.strategy import Constant
>>> from pypose.optim.scheduler \
    import StopOnPlateau

>>> class InvNet(nn.Module):

        def __init__(self, *dim):
            super().__init__()
            init = pp.randn_SE3(*dim)
            self.pose = pp.Parameter(init)

        def forward(self, input):
            error = (self.pose @ input).Log()
            return error.tensor()

>>> device = torch.device("cuda")
>>> input = pp.randn_SE3(2, 2, device=device)
>>> invnet = InvNet(2, 2).to(device)
>>> strategy = Constant(damping=1e-4)
>>> optimizer = LM(invnet, strategy=strategy)
>>> scheduler = StopOnPlateau(optimizer,
                              steps=10,
                              patience=3,
                              decreasing=1e-3,
                              verbose=True)

>>> # 1st option, full optimization
>>> scheduler.optimize(input=input)

>>> # 2nd option, step optimization
>>> while scheduler.continual():
        loss = optimizer.step(input)
        scheduler.step(loss)
```