2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 1-5, 2023. Detroit, USA

navlie: A Python Package for State Estimation on Lie Groups

Charles Champagne Cossette, Mitchell Cohen, Vassili Korotkine, Arturo del Castillo Bernal,
Mohammed Ayman Shalaby, James Richard Forbes[1]

***Abstract*— The ability to rapidly test a variety of algorithms for an arbitrary state estimation task is valuable in the prototyping phase of navigation systems. Lie group theory is now mainstream in the robotics community, and hence estimation prototyping tools should allow state definitions that belong to manifolds. A new package, called `navlie`, provides a framework that allows a user to model a large class of problems by implementing a set of classes complying with a generic interface. Once accomplished, `navlie` provides a variety of on-manifold estimation algorithms that can run directly on these classes. The package also provides a built-in library of common models, as well as many useful utilities. The open-source project can be found at**

`https://github.com/decargroup/navlie`

***Index Terms*— Localization, Sensor Fusion, Software Tools for Robot Programming**

I. Introduction

To achieve full autonomy, robotic systems require the robot to maintain a belief of the current system state, given noisy sensor observations. Typically, this problem is solved using recursive Bayesian filters [1], such as Kalman-like filters, or optimization-based smoothing methods. Variants of the Kalman filter often used for nonlinear state estimation include the Extended Kalman Filter (EKF) [1, Ch. 5.2] and the sigma-point Kalman filter (SPKF) [1, Ch. 5.6], while smoothing methods rely on batch estimation [2]. These traditional approaches assume that the system state is an element of Euclidean space. Difficulty arises in robotics applications, where the system state often evolves on a manifold. In recent years, significant progress had been made to formulate robotic state estimation problems using tools from Lie theory [3]. Early applications of Lie groups to navigation typically focused on orientation estimation [4], where attitude is represented as an element of the Special Orthogonal Group $SO(3)$. More recently, utilizing Lie groups has shown success in several robotics applications, including characterizing the uncertainty of poses [5, 6], addressing consistency issues in simultaneous localization and mapping (SLAM) problems [7, 8], and inertial navigation problems [9, 10]. The use of Lie groups is also a crucial part of the invariant EKF [11], and the equivariant filter [12].

Given these recent successes, several software packages have been developed to aid in designing on-manifold state estimators. The library `kalmanif`, based on [3], provides a collection of Kalman filters on Lie groups. The OpenVINS codebase [13] provides a foundation for developing on-manifold visual-inertial estimators. Additionanlly, libraries developed for optimization-based state estimation such as Ceres [14], GTSAM [15], and g2o [16] also support on-manifold estimation. These libraries are all implemented in C++, allowing for real-time application, at the cost of making their use for rapid algorithm prototyping more difficult. The library UKF-M [17] provides Python and MATLAB implementations of an on-manifold unscented Kalman filter (UKF). Recently, the PyTorch-based package `PyPose` [18] was released, providing a collection of differentiable estimation and control tools, meant to connect learning-based models with physics-based optimization.

The main contribution of this paper is to present the library `navlie`, which contains a collection of on-manifold estimation algorithms and tools for rapid prototyping. At the time of writing, `navlie` has the following features:

- Generic implementations of common filtering algorithms, namely the EKF, iterated EKF, invariant EKF, multiple sigma-point Kalman filters, including the unscented, spherical cubature, and Gauss-Hermite filters, and the interacting multiple model filter.
- Batch estimation in a maximum a posterori (MAP) framework.
- Implementations of common Lie groups such as $SO(2)$, $SO(3)$, $SE(2)$, $SE(3)$, $SE_2(3)$ and $SL(3)$.
- A collection of common process and measurement models.
- A preintegration module for linear, wheel odometry, and IMU process models.
- Utilities for numerical differentiation, plotting, error and consistency evaluation, and conducting Monte-Carlo experiments.

This extended feature set, accessible with the convenience of Python, makes `navlie` unique.

II. Preliminaries

A. Lie groups

A Lie group G is a smooth manifold whose elements, given a group operation $\circ : G \times G \to G$, satisfy the group axioms [3]. The application of this operation to two arbitrary Lie group elements $\mathcal{X}, \mathcal{Y} \in G$ is written as $\mathcal{X} \circ \mathcal{Y} \in G$. For

*This work was supported by the NSERC Discovery and Alliance programs (with Denso and ARA Robotics), as well as by the Innovation for Defence Excellence and Security (IDEaS) program from the Department of National Defence (DND). Any opinions and conclusions in this work are strictly those of the authors and do not reflect the views, positions, or policies of - and are not endorsed by - IDEaS, DND, or the Government of Canada.

[1]The authors are with the Department of Mechanical Engineering, McGill University, 817 Sherbrooke St. W., Montreal, QC H3A 0C3, Canada. Corresponding author is `charles.cossette@mail.mcgill.ca`

2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) | 978-1-6654-9190-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/IROS55552.2023.10342362

any G, there exists an associated Lie algebra $\mathfrak{g}$, a vector space identifiable with elements of $\mathbb{R}^m$, where m is referred to as the degrees of freedom of G. The Lie algebra is related to the group through the exponential and logarithmic maps, denoted $\exp : \mathfrak{g} \to G$ and $\log : G \to \mathfrak{g}$. The "vee" and "wedge" operators are denoted $(\cdot)^\vee : \mathfrak{g} \to \mathbb{R}^m$ and $(\cdot)^\wedge : \mathbb{R}^m \to \mathfrak{g}$, and are used to associate group elements with vectors with

$$\mathcal{X} = \exp(\boldsymbol{\xi}^\wedge) \triangleq \mathrm{Exp}(\boldsymbol{\xi}), \qquad \boldsymbol{\xi} = \log(\mathcal{X})^\vee \triangleq \mathrm{Log}(\mathcal{X}),$$

where $\mathcal{X} \in G, \boldsymbol{\xi} \in \mathbb{R}^m$, and the shorthand notation $\mathrm{Exp} : \mathbb{R}^m \to G$ and $\mathrm{Log} : G \to \mathbb{R}^m$ has been defined. The most common Lie groups appearing in robotics are $SO(n)$, representing rotations in n-dimensional space, $SE(n)$, representing poses, and $SE_2(3)$ representing "extended" poses that also contain velocity information. In these cases, the element $\mathcal{X}$ is an invertible matrix and the group operation $\circ$ is regular matrix multiplication.

1) $\oplus$ and $\ominus$ operators: Estimation theory for vector-space states and Lie groups can be elegantly aggregated into a single mathematical treatment by defining generalized "addition" $\oplus : G \times \mathbb{R}^m \to G$ and "subtraction" $\ominus : G \times G \to \mathbb{R}^m$ operators, whose precise definitions are problem dependant. For example, possible implementations include

$$\begin{aligned} \mathcal{X} \oplus \delta\mathbf{x} &= \mathcal{X} \circ \mathrm{Exp}(\delta\mathbf{x}) \quad &&\text{(Lie group right)}, \\ \mathcal{X} \oplus \delta\mathbf{x} &= \mathrm{Exp}(\delta\mathbf{x}) \circ \mathcal{X} \quad &&\text{(Lie group left)}, \\ \mathbf{x} \oplus \delta\mathbf{x} &= \mathbf{x} + \delta\mathbf{x} \quad &&\text{(vector space)}, \end{aligned} \tag{1}$$

for addition and, correspondingly,

$$\begin{aligned} \mathcal{X} \ominus \mathcal{Y} &= \mathrm{Log}(\mathcal{Y}^{-1} \circ \mathcal{X}) \quad &&\text{(Lie group right)}, \\ \mathcal{X} \ominus \mathcal{Y} &= \mathrm{Log}(\mathcal{X} \circ \mathcal{Y}^{-1}) \quad &&\text{(Lie group left)}, \\ \mathbf{x} \ominus \mathbf{y} &= \mathbf{x} - \mathbf{y} \quad &&\text{(vector space)}, \end{aligned} \tag{2}$$

for subtraction. This abstraction is natural since a linear vector space technically qualifies as a Lie group with regular addition $+$ as the group operation.

2) Derivatives on Lie groups: Again following [3], the Jacobian of a function $f : G \to G$, taken with respect to $\mathcal{X}$ can be defined as

$$\left.\frac{Df(\mathcal{X})}{D\mathcal{X}}\right|_{\bar{\mathcal{X}}} \triangleq \left.\frac{\partial f(\bar{\mathcal{X}} \oplus \delta\mathbf{x}) \ominus f(\bar{\mathcal{X}})}{\partial \delta\mathbf{x}}\right|_{\delta\mathbf{x}=\mathbf{0}},$$

where it should be noted that the function $f(\bar{\mathcal{X}} \oplus \delta\mathbf{x}) \ominus f(\bar{\mathcal{X}})$ of $\delta\mathbf{x}$ has $\mathbb{R}^m$ as both its domain and codomain, and can thus be differentiated using any standard technique. With the above general definition of a derivative, it is easy to define the so-called *Jacobian of* G as $\mathbf{J} = D\mathrm{Exp}(\mathbf{x})/D\mathbf{x}$, where left/right group Jacobians are obtained with left/right definitions of $\oplus$ and $\ominus$.

III. Estimation Algorithms on Manifolds

`navlie` operates on implementations of states, process, and measurement models given in the standard form of

$$\mathcal{X}_k = f(\mathcal{X}_{k-1}, \mathbf{u}_{k-1}) \oplus \mathbf{w}_{k-1}, \tag{3}$$

$$\mathbf{y}_k = \mathbf{g}(\mathcal{X}_k) + \mathbf{v}_k, \tag{4}$$

where $\mathbf{u}_{k-1}$ is a generic process model input, and $\mathbf{w}_{k-1} \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_{k-1})$, $\mathbf{v}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_k)$ are "additive" Gaussian noises.

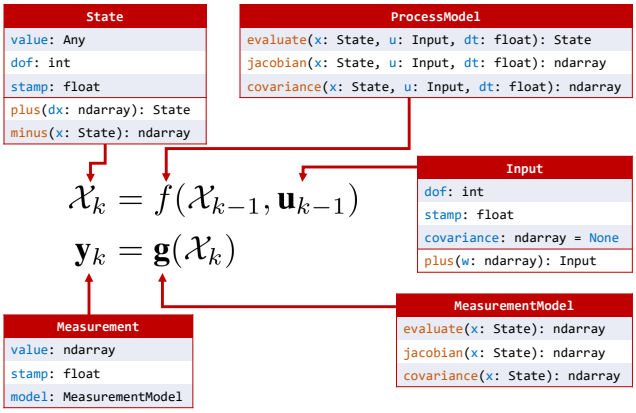

Fig. 1. Class diagram for the abstract types used in `navlie`. The methods shown should be implemented by the user.

Figure 1 shows the key classes and abstract methods that represent the system model. For the state, the user must implement $\oplus$ and $\ominus$ functions. For the process model, the user must implement the function f, and a computation of $\mathbf{Q}_{k-1}$. Finally, for the measurement model, the user must implement $\mathbf{g}$ and $\mathbf{R}_k$. Jacobians can optionally be implemented by the user for maximum performance, but a default finite-difference method is used otherwise.

The remainder of this section the mathematical describes the details of the algorithms found in `navlie`. The filter classes in this package are *stateless*, which allows the user to switch and combine estimators from timestep to timestep, or to use specific estimators for specific measurement types.

A. Extended Kalman Filter

The EKF in `navlie` follows a standard covariance-form implementation, and follows a predict-correct structure. The EKF is initialized with a mean $\hat{\mathcal{X}}_0$ and covariance $\hat{\mathbf{P}}_0$, and the prediction step is given by

$$\begin{aligned} \mathbf{F}_{k-1} &= \left.\frac{Df(\mathcal{X}, \mathbf{u}_{k-1})}{D\mathcal{X}}\right|_{\hat{\mathcal{X}}_{k-1}}, \\ \check{\mathcal{X}}_k &= f(\hat{\mathcal{X}}_{k-1}, \mathbf{u}_{k-1}), \\ \check{\mathbf{P}}_k &= \mathbf{F}_{k-1}\hat{\mathbf{P}}_{k-1}\mathbf{F}_{k-1}^\mathsf{T} + \mathbf{Q}_{k-1}, \end{aligned}$$

where $\check{(\cdot)}$ and $\hat{(\cdot)}$ refer to predicted and corrected variables, respectively. The correction step is given by

$$\begin{aligned} \mathbf{G}_k &= \left.\frac{D\mathbf{g}(\mathcal{X})}{D\mathcal{X}}\right|_{\check{\mathcal{X}}_k}, \\ \mathbf{K} &\triangleq \check{\mathbf{P}}_k\mathbf{G}_k(\mathbf{G}_k\check{\mathbf{P}}_k\mathbf{G}_k^\mathsf{T} + \mathbf{R}_k)^{-1}, \\ \mathbf{z} &\triangleq \mathbf{y}_k - \mathbf{g}(\check{\mathcal{X}}_k), \\ \hat{\mathcal{X}}_k &= \check{\mathcal{X}}_k \oplus \mathbf{K}_k\mathbf{z}, \\ \hat{\mathbf{P}}_k &= (\mathbf{1} - \mathbf{K}\mathbf{G}_k)\check{\mathbf{P}}_k. \end{aligned}$$

B. Iterated Extended Kalman Filter

The iterated EKF treats the correction step as a nonlinear least squares problem solved using Gauss-Newton in an

iterative manner [19]. In `navlie`, the iterated EKF inherits the EKF prediction step, but performs a correction step by first computing

$$\begin{aligned}\mathbf{K} &\triangleq \mathbf{J}_k\check{\mathbf{P}}_k\mathbf{J}_k^\mathsf{T}\mathbf{S}_k^\mathsf{T}(\mathbf{G}_k\mathbf{J}_k\check{\mathbf{P}}_k\mathbf{J}_k^\mathsf{T}\mathbf{G}_k^\mathsf{T}+\mathbf{R}_k)^{-1},\\ \mathbf{z} &\triangleq \mathbf{y}_k-\mathbf{g}(\hat{\mathcal{X}}_k)+\mathbf{G}_k\mathbf{J}_k(\hat{\mathcal{X}}_k\ominus\check{\mathcal{X}}_k),\\ \delta\mathbf{x}_k &= \mathbf{K}\mathbf{z}-\mathbf{J}_k(\hat{\mathcal{X}}_k\ominus\check{\mathcal{X}}_k),\end{aligned} \tag{5}$$

and then updating the current estimate $\hat{\mathcal{X}}_k \leftarrow \hat{\mathcal{X}}_k \oplus \delta\mathbf{x}_k$, initialized with $\hat{\mathcal{X}}_k \leftarrow \check{\mathcal{X}}_k$. Equations (5) are continuously iterated until convergence.

C. Invariant Extended Kalman Filter

The invariant EKF [11] defines the state and innovation such that the process and measurement model Jacobians are state estimate independent for a specific form of process and measurement models. The invariant EKF exploits measurement models with one of two forms,

$$\begin{aligned}\mathbf{y}_k &= \mathcal{X}_k\cdot\mathbf{b}_k+\mathbf{v}_k, &&\text{(left-invariant)}\\ \mathbf{y}_k &= \mathcal{X}_k^{-1}\cdot\mathbf{b}_k+\mathbf{v}_k &&\text{(right-invariant)}\end{aligned}$$

where $\mathbf{b}_k \in \mathbb{R}^n$ is an arbitrary known vector and $\cdot$ denotes the *group action* [3]. When the measurement model is of the right-invariant form, the invariant EKF defines the innovation to be used in the filter as

$$\begin{aligned}\mathbf{z} &= \check{\mathcal{X}}\cdot(\mathbf{y}-\mathbf{g}(\check{\mathcal{X}}))\\ &= \check{\mathcal{X}}\cdot(\mathbf{g}(\mathcal{X})+\mathbf{v}-\mathbf{g}(\check{\mathcal{X}}))\\ &\approx \check{\mathcal{X}}\cdot(\mathbf{g}(\check{\mathcal{X}})+\mathbf{G}\delta\mathbf{x}+\mathbf{v}-\mathbf{g}(\check{\mathcal{X}}))\\ &= \check{\mathcal{X}}\cdot\mathbf{G}\delta\mathbf{x}+\check{\mathcal{X}}\cdot\mathbf{v}\end{aligned}$$

where a Taylor series expansion of $\mathbf{g}(\mathcal{X})$ was used. The result is that the Jacobian of the innovation $\mathbf{z}$ is given by $\check{\mathcal{X}}\cdot\mathbf{G}$, which will be state-independent if a right-invariant measurement model is used with a left-definition of the $\oplus$ operator. Similarly, when the measurement model is of the left-invariant form, the innovation used in the invariant EKF is written

$$\mathbf{z} = \check{\mathcal{X}}^{-1}\cdot(\mathbf{y}-\mathbf{g}(\check{\mathcal{X}}))\approx\check{\mathcal{X}}^{-1}\cdot\mathbf{G}\delta\mathbf{x}+\check{\mathcal{X}}^{-1}\cdot\mathbf{v},$$

and the Jacobian of the left-invariant innovation is thus $\check{\mathcal{X}}^{-1}\cdot\mathbf{G}$, which will be state-independent with a left-invariant measurement model and right $\oplus$ definition. The implementation of the invariant EKF in `navlie` allows users to reuse existing measurement models with the specific definitions of the innovation presented above. The construction of the left- or right- invariant innovations is done automatically when possible.

D. Sigma-Point Kalman Filter

The sigma-point transform generates a set of sigma-points from a prior distribution, then passes them through a non-linearity to estimate the mean and covariance of the transformed distribution [1]. In `navlie`, sigma points can be generated by unscented (UKF), spherical cubature (CKF), and Gauss-Hermite (GHKF) transforms.

1) Prediction Step: For the sigma-point filters, noise is assumed to be additive to the input such that the process model is $f(\mathcal{X}_{k-1}, \mathbf{u}_{k-1}+\mathbf{w}_{k-1})$ with $\mathbf{w}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_k^u)$. The sigma points are then calculated by

$$\begin{aligned}\tilde{\mathbf{P}}_{k-1} &= \operatorname{diag}(\hat{\mathbf{P}}_{k-1}, \mathbf{Q}_{k-1}^u) \triangleq \mathbf{L}\mathbf{L}^\mathsf{T},\\ \begin{bmatrix}\delta\mathbf{x}^{(i)\mathsf{T}} & \delta\mathbf{w}^{(i)\mathsf{T}}\end{bmatrix}^\mathsf{T} &= \mathbf{L}\boldsymbol{\xi}^{(i)},\end{aligned}$$

where $\mathbf{L}$ is a Cholesky decomposition, $\boldsymbol{\xi}^{(i)}$ is the i^{th} unit sigma point generated by one of the transforms, and $w^{(i)}$ is a corresponding weight. Sigma points are propagated with

$$\check{\mathcal{X}}_k^{(i)} = f(\hat{\mathcal{X}}_{k-1}\oplus\delta\mathbf{x}^{(i)}, \mathbf{u}_{k-1}+\delta\mathbf{w}^{(i)}).$$

The mean $\check{\mathcal{X}}_k$ is computed in an iterated manner, where the first of the propagated states is set as the initial mean estimate, $\check{\mathcal{X}}_k \leftarrow \check{\mathcal{X}}_k^{(1)}$, and the error with the rest of the propagated states is computed with a weighted mean. The mean is updated until a convergence criterion is met,

$$\delta\mathbf{x}_j = \sum_i w^{(i)}\check{\mathcal{X}}_k^{(i)}\ominus\check{\mathcal{X}}_k, \quad \check{\mathcal{X}}_k \leftarrow \check{\mathcal{X}}_k\oplus\delta\mathbf{x}_j.$$

After convergence, the covariance is computed with

$$\check{\mathbf{P}}_k = \sum_i w^{(i)}\left(\check{\mathcal{X}}_k^{(i)}\ominus\check{\mathcal{X}}_k\right)\left(\check{\mathcal{X}}_k^{(i)}\ominus\check{\mathcal{X}}_k\right)^\mathsf{T}.$$

2) Correction Step: The correction step is handled the same way as in [17, Sec. II – C]. At the time of writing, sigma points are generated for the state only since the noise is assumed to be additive to the output in the measurement model. Thus,

$$\begin{aligned}\mathbf{y}_k^{(i)} &= \mathbf{g}(\check{\mathcal{X}}_k\oplus\delta\mathbf{x}^{(i)}), \quad \bar{\mathbf{y}}_k = \sum_i w^{(i)}\mathbf{y}_k^{(i)},\\ \mathbf{P}_{yy} &= \sum_i w^{(i)}\left(\mathbf{y}_k^{(i)}-\bar{\mathbf{y}}_k\right)\left(\mathbf{y}_k^{(i)}-\bar{\mathbf{y}}_k\right)^\mathsf{T}+\mathbf{R}_k,\\ \mathbf{P}_{xy} &= \sum_i w^{(i)}\delta\mathbf{x}^{(i)}\left(\mathbf{y}_k^{(i)}-\bar{\mathbf{y}}_k\right)^\mathsf{T},\\ \mathbf{K} &= \mathbf{P}_{xy}\mathbf{P}_{yy}^{-1},\\ \hat{\mathcal{X}}_k &= \hat{\mathcal{X}}_k\oplus\mathbf{K}\left(\mathbf{y}_k-\bar{\mathbf{y}}_k\right), \quad \hat{\mathbf{P}}_k = \check{\mathbf{P}}_k-\mathbf{K}\mathbf{P}_{yy}\mathbf{K}^\mathsf{T}.\end{aligned}$$

E. Batch Estimation via Nonlinear Least Squares

Batch estimation estimates the history of states from time $t = t_0$ to $t = t_K$, written as $\mathcal{X} = \mathcal{X}_{0:K} = \{\mathcal{X}_0, \ldots, \mathcal{X}_K\}$, that maximizes the posterior probability density function given a prior $\check{\mathcal{X}}_0$ and all the inputs and measurements received. Finding the MAP estimate can be equivalently posed as a weighted nonlinear least-squares problem as [5]

$$\begin{aligned}\hat{\mathcal{X}} &= \arg\min_{\mathcal{X}}\frac{1}{2}\left\|\mathcal{X}_0\ominus\check{\mathcal{X}}_0\right\|_{\check{\mathbf{P}}}^2\\ &+\frac{1}{2}\sum_{k=1}^{K}\left(\left\|\mathcal{X}_k\ominus f(\mathcal{X}_{k-1},\mathbf{u}_{k-1})\right\|_{\mathbf{Q}_{k-1}}^2+\left\|\mathbf{y}_k-\mathbf{g}(\mathcal{X}_k)\right\|_{\mathbf{R}_k}^2\right),\end{aligned}$$

which is a combination of a prior, process, and measurement errors. `navlie` provides on-manifold implementations of Gauss-Newton and Levenberg Marquardt [20] to solve non-

linear least squares problems that operate on generic error terms. Additionally, `navlie` implements standard forms of prior, process, and measurement errors, allowing users to easily construct and solve batch problems utilizing any chosen state, process, and measurement definitions.

F. Interacting Multiple-Model Filter

The Interacting Multiple Model Filter (IMM) [21] is a method for handling systems with discrete modes, modelled as a finite-state Markov chain with some probability transition matrix. The discrete modes can, for instance, correspond to different hypotheses for the process model. In this case, the overarching process model may be written

$$\mathcal{X}_k = f(\mathcal{X}_{k-1}, \mathbf{u}_{k-1}, \theta_k),$$

where $\theta_k \in \{1, \dots, N\}$ is modelled as a finite-state Markov chain. The values of θ_k correspond to a different process model, such that $\mathcal{X}_k = f_{\theta_k}(\mathcal{X}_{k-1}, \mathbf{u}_{k-1})$.

A key step in the IMM algorithm is the computation of means and covariances of Gaussian mixtures. For Gaussian mixtures defined on a vector space, the means and covariances are straightforward to compute. However, for a Lie group, the procedure to compute Gaussian mixtures involves expressing all of the components in the same tangent space and is more involved. `navlie` seamlessly handles the manifold structure using the approach in [22], reparametrizing the Gaussian mixture components about the component with the highest weight, carrying out the mixture in the corresponding tangent space, and projecting back to the manifold.

IV. Other On-Manifold Utilities

A. Finite-difference and complex-step differentiation

`navlie` also includes general-purpose numerical differentiation tools based on both finite differencing and the complex step [23]. The i^{th} column of the Jacobian of $f(\mathcal{X})$ can be approximated with a forward difference procedure

$$\left.\frac{Df(\mathcal{X})}{D\mathcal{X}}\right|_{\bar{\mathcal{X}}} \mathbf{e}_i \approx \frac{f(\bar{\mathcal{X}} \oplus h\mathbf{e}_i) \ominus f(\bar{\mathcal{X}})}{h},$$

or using a complex step

$$\left.\frac{Df(\mathcal{X})}{D\mathcal{X}}\right|_{\bar{\mathcal{X}}} \mathbf{e}_i \approx \frac{\text{Imag}\left(f(\bar{\mathcal{X}} \oplus hj\mathbf{e}_i) \ominus f(\bar{\mathcal{X}})\right)}{h},$$

where $j = \sqrt{-1}$ and $\mathbf{e}_i \in \mathbb{R}^m$ is all zeros except for a 1 in the i^{th} element.

B. Interpolation

`navlie` also provides a general interpolation method that works on any state implementation with well-defined $\oplus$ and $\ominus$ operators. Given two states $\mathcal{X}_k = \mathcal{X}(t_k)$, $\mathcal{X}_{k-1} = \mathcal{X}(t_{k-1})$, an interpolated state at time t can be obtained from

$$\begin{aligned} \alpha &= (t - t_{k-1})/(t_k - t_{k-1}) \in [0, 1], \\ \delta\mathbf{x} &= \mathcal{X}_k \ominus \mathcal{X}_{k-1}, \quad \mathcal{X}(t) = \mathcal{X}_{k-1} \oplus \alpha\delta\mathbf{x}. \end{aligned} \tag{6}$$

As usual, `navlie` provides a single implementation of the above that works with any state implementation complying with the abstract `State` interface.

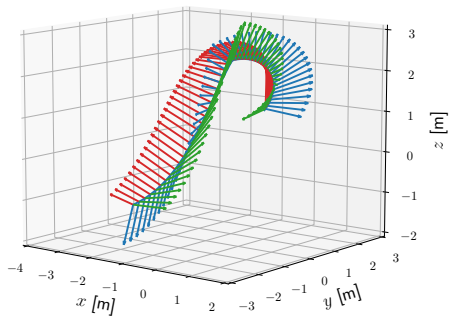

Fig. 2. $SE(3)$ pose interpolation using (6).

C. Composite States

A *composite group* or *bundle* refers to a collection of states serving as a single concatenated object of the form

$$\boldsymbol{\mathcal{X}} = (\mathcal{X}_1, \dots \mathcal{X}_N) \in G_1 \times \dots \times G_N.$$

Defining $\delta\mathbf{x} = [\delta\mathbf{x}_1^\mathsf{T} \ \dots \ \delta\mathbf{x}_N^\mathsf{T}]^\mathsf{T}$, the $\oplus$ operator for this new state is defined elementwise by

$$\boldsymbol{\mathcal{X}} \oplus \delta\mathbf{x} = (\mathcal{X}_1 \oplus \delta\mathbf{x}_1, \ \dots \ , \mathcal{X}_N \oplus \delta\mathbf{x}_N),$$

and a similar definition applies to $\ominus$. The `CompositeState` class in `navlie` allows users to freely combine and nest state implementations into a hierarchy of more complicated states, that all still comply with the `State` interface.

D. Preintegration

`navlie` provides preintegration classes for several common process models in robotics. For example, `navlie` provides the ability to preintegrate any linear process model

$$\mathbf{x}_k = \mathbf{F}_{k-1}\mathbf{x}_{k-1} + \mathbf{L}_{k-1}\mathbf{u}_{k-1}.$$

This is done by direct iteration to give

$$\begin{aligned} \mathbf{x}_j &= \left(\prod_{k=i}^{j-1} \mathbf{F}_k\right)\mathbf{x}_i + \sum_{k=i}^{j-1}\left(\prod_{\ell=k+1}^{j-1}\mathbf{F}_\ell\right)\mathbf{L}_k\mathbf{u}_k \\ &\triangleq \mathbf{F}_{ij}\mathbf{x}_i + \Delta\mathbf{x}_{ij}. \end{aligned}$$

Body-frame velocity-input process models of the form

$$\mathbf{X}_k = \mathbf{X}_{k-1}\text{Exp}(\Delta t\mathbf{u}_{k-1}),$$

where $\mathbf{X}_k$ belongs to a Lie group such as $SO(n)$ or $SE(n)$, can also be preintegrated with `navlie`, again done by direct iteration to produce

$$\mathbf{X}_j = \mathbf{X}_i \underbrace{\prod_{k=i}^{j-1} \text{Exp}(\Delta t\mathbf{u}_k)}_{\triangleq \Delta\mathbf{X}_{ij}}.$$

Finally, IMU preintegration is also included and is the first open-source Python implementation to do so directly on the $SE_2(3)$ manifold. Following [2, 9], discrete-time IMU kinematics are of the form

$$\mathbf{T}_k = \mathbf{G}_{k-1}\mathbf{T}_{k-1}\mathbf{U}_{k-1},$$

```python
import numpy as np
import navlie as nav
from scipy.linalg import expm, logm

def wedge_se3(x):
    return np.array([    0, -x[2],  x[1], x[3],
                      x[2],     0, -x[0], x[4],
                     -x[1],  x[0],     0, x[5],
                         0,     0,     0,    0])
def vee_se3(X):
    return np.array(
    [X[2, 1], X[0, 2], X[1, 0], X[0, 3], X[1, 3], X[2, 3]]
    )

class SE3State(nav.State):
    def __init__(self, value, stamp):
        self.value = value
        self.stamp = stamp
        self.dof = 6

    def plus(self, dx):
        new_value = self.value @ expm(wedge_se3(dx))
        return SE3State(new_value, self.stamp)

    def minus(self, other: "SE3State"):
        other_inv = np.linalg.inv(other.value)
        return vee_se3(logm(other_inv @ self.value))
```

Fig. 3. A minimal implementation of a state belonging to the $SE(3)$ manifold.

where $\mathbf{T}_k \in SE_2(3)$ and $\mathbf{G}_k, \mathbf{U}_k$ are 5×5 matrices constructed from gravity terms and IMU measurements. This is easily preintegrated with

$$\mathbf{T}_j = \left(\prod_{k=i}^{j-1} \mathbf{G}_{k-1}\right) \mathbf{T}_i \left(\prod_{k=i}^{j-1} \mathbf{U}_{k-1}\right) \triangleq \Delta\mathbf{G}_{ij} \mathbf{T}_i \Delta\mathbf{U}_{ij}$$

In all cases, `navlie` handles noise propagation through the preintegration process, and also provides the ability to augment these models with input bias estimates.

V. Example Usage

This section will show, concretely, an example implementation of a problem in `navlie`. Consider the problem of estimating the pose represented as an element of $SE(3)$ given by

$$\mathbf{T} = \begin{bmatrix} \mathbf{C}_{wb} & \mathbf{r}_w \\ \mathbf{0} & 1 \end{bmatrix} \in SE(3),$$

where $\mathbf{C}_{wb}$ is the direction-cosine-matrix (DCM) describing the attitude of a robot's body frame $\mathcal{F}_b$ relative to a world frame $\mathcal{F}_w$, and $\mathbf{r}_w$ is the position of the robot resolved in the world frame. In the considered example, a right-perturbation is chosen, and the definitions of the $\oplus$ and $\ominus$ operators are given by the "Lie group right" entries in (1) and (2) with $\delta\mathbf{x} \in \mathbb{R}^6$. Figure 3 shows how this state definition can be constructed using the `State` interface in `navlie`.

The robot has access to measurements of $\mathbf{u}_k = \begin{bmatrix} \boldsymbol{\omega}_{b_k}^\mathsf{T} & \mathbf{v}_{b_k}^\mathsf{T} \end{bmatrix}^\mathsf{T}$ where the translational velocity $\mathbf{v}_b$ and angular velocity $\boldsymbol{\omega}_b$ of the robot are both resolved in the body frame. The corresponding process model is

$$\mathbf{T}_k = \mathbf{T}_{k-1} \mathrm{Exp}\left(\Delta t \left(\mathbf{u}_{k-1} + \mathbf{w}_{k-1}\right)\right), \tag{7}$$

where $\mathbf{w}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_k^u)$ and $\mathrm{Exp}(\cdot)$ is the exponential map for $SE(3)$. Figure 4 shows how the `ProcessModel` interface can be utilized to define the process model, as well as the process model Jacobian and covariance. The robot is also equipped with a UWB tag that collects range measurements to multiple anchors with known position. Denoting the i^{th} known anchor position in the world frame as $\mathbf{a}_w^i$, the range measurements to the i^{th} anchor are in the form $y_k^i = g^i(\mathbf{T}_k) + v_k^i$, where $v_k^i \sim \mathcal{N}(0, R_k^i)$, and the measurement model is given by

$$g^i(\mathbf{T}) = \left\| \mathbf{r}_w - \mathbf{a}_w^i \right\|. \tag{8}$$

Figure 5 shows how the `MeasurementModel` interface can be used to implement the range measurement model (8).

```python
class BodyFrameVelocity(nav.ProcessModel):
    def __init__(self, input_covariance_matrix):
        self.Q = input_covariance_matrix

    def evaluate(self, x, u, dt):
        new_value = x.value @ expm(wedge_se3(u.value * dt))
        return SE3State(new_value, x.stamp + dt)

    def jacobian(self, x, u, dt):
        return adjoint_se3(expm(wedge_se3(-u.value * dt)))

    def covariance(self, x, u, dt):
        L = dt * left_jacobian_se3(-u.value * dt)
        return L @ self.Q @ L.T
```

Fig. 4. Implementation of the body-frame-velocity input process model in (7) using the `navlie` framework. Standard implementations of the adjoint matrix operator and the left Jacobian for $SE(3)$ are omitted.

```python
class RangePoseToAnchor(nav.MeasurementModel):
    def __init__(
        self,
        anchor_position: np.ndarray,
        measurement_covariance: float,
    ):
        self.a_pos = anchor_position
        self.R = measurement_covariance

    def evaluate(self, x: SE3State):
        pos = x.value[0:3, 3]
        return np.linalg.norm(pos - self.a_pos)

    def covariance(self, x: SE3State):
        return self.R
```

Fig. 5. Implementation of a ranging measurement model to a single landmark as described by (8). Here, the user has not implemented the `jacobian` method, meaning `navlie` automatically uses finite difference.

Once the state definition, process model, and measurement models have been implemented for a particular problem, the user can then fuse inputs and measurements using any of the generic implementations of the algorithms outlined in Section III. Figure 7 shows the norm of the position error and Figure 8 shows the NEES for the considered sample localization problem with three anchors, for several different estimators. If groundtruth data is available, a collection of utilities included in `navlie` can be utilized for error and consistency evaluation. Note that, despite not being showcased here, the implementations of the estimation algorithms make it simple to handle measurements from varying sensors, at varying rates, as they all share a common interface.

```python
import numpy as np
import navlie as nav

x0 = nav.lib.SE3State([0, 0, 0, 0, 0, 0],
                      stamp=0.0, direction="right")
P0 = np.diag([0.1**2, 0.1**2, 0.1**2, 1, 1, 1])
Q = np.diag([0.01**2, 0.01**2, 0.01**2, 0.1, 0.1, 0.1])

process_model = nav.lib.BodyFrameVelocity(Q)
data = nav.datasets.SimulatedPoseRangingData()
ground_truth = data.get_ground_truth()
input_data = data.get_input_data()
meas_data = data.get_meas_data()

## Choose a filter!
kf = nav.ExtendedKalmanFilter(process_model)
# kf = nav.IteratedKalmanFilter(process_model),
# kf = nav.SigmaPointKalmanFilter(process_model,
method="unscented")
# kf = nav.SigmaPointKalmanFilter(process_model,
method="cubature")

x = nav.StateWithCovariance(x0, P0)
for u, y in zip(input_data, meas_data):
    x = kf.predict(x, u)
    x = kf.correct(x, y, u)
```

Fig. 6. A typical top-level file using `navlie`. Here, an EKF is chosen, but choosing a different algorithm is a matter of changing a single line of code.

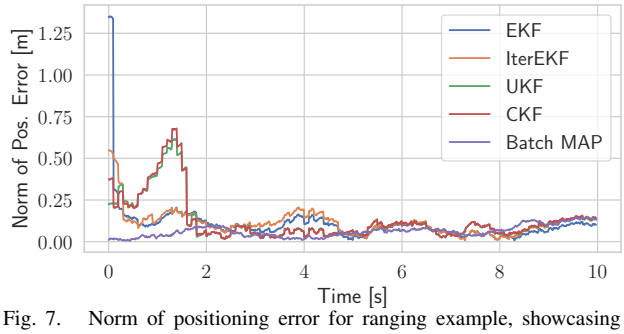

Fig. 7. Norm of positioning error for ranging example, showcasing the ease of running various algorithms on arbitrary problems in `navlie`.

VI. Conclusion

This paper presents `navlie`, a Python package that provides a collection of on-manifold estimation algorithms and utilities suitable for a variety of state estimation problems. `navlie` allows for users to rapidly prototype algorithms, compare various estimation approaches, and analyze fundamental properties of the developed algorithms, such as estimator consistency.

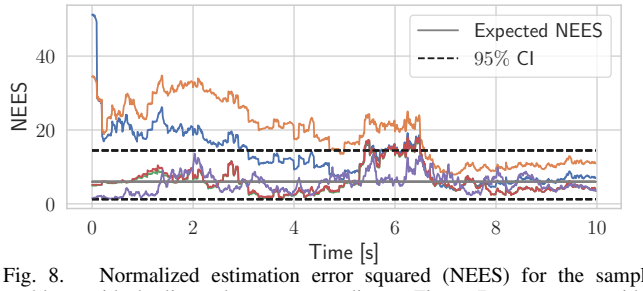

Fig. 8. Normalized estimation error squared (NEES) for the sample problem, with the line colors corresponding to Figure 7. `navlie` provides utility functions for standardized computation and plotting of the NEES.

References

[1] S. Särkkä, *Bayesian Filtering and Smoothing*. Cambridge University Press, 2013.

[2] T. D. Barfoot, *State Estimation for Robotics*, 2nd ed. 2023.

[3] J. Solà, J. Deray, and D. Atchuthan, "A Micro Lie Theory for State Estimation in Robotics," pp. 1–17, 2018. arXiv: 1812.01537. [Online]. Available: http://arxiv.org/abs/1812.01537.

[4] R. Mahony, T. Hamel, and J.-M. Pflimlin, "Nonlinear Complementary Filters on the Special Orthogonal Group," *IEEE Trans. on Auto. Control*, vol. 53, no. 5, pp. 1203–1218, 2008.

[5] T. D. Barfoot and P. T. Furgale, "Associating Uncertainty with Three-Dimensional Poses for use in Estimation Problems," *IEEE Trans. on Robotics*, vol. 30, no. 3, pp. 679–693, 2014.

[6] J. G. Mangelson, M. Ghaffari, R. Vasudevan, and R. M. Eustice, "Characterizing the Uncertainty of Jointly Distributed Poses in the Lie Algebra," *IEEE Trans. on Robotics*, vol. 36, no. 5, pp. 1371–1388, 2020.

[7] S. Heo and C. G. Park, "Consistent EKF-Based Visual-Inertial Odometry on Matrix Lie Group," *IEEE Sensors Journal*, vol. 18, no. 9, pp. 3780–3788, 2018.

[8] M. Brossard, A. Barrau, and S. Bonnabel, "Exploiting Symmetries to Design EKFs with Consistency Properties for Navigation and SLAM," *IEEE Sensors Journal*, vol. 19, no. 4, pp. 1572–1579, 2018.

[9] M. Brossard, A. Barrau, P. Chauchat, and S. Bonnabel, "Associating Uncertainty to Extended Poses for on Lie Group IMU Preintegration with Rotating Earth," *IEEE Trans. on Robotics*, vol. 38, no. 2, pp. 998–1015, 2021.

[10] M. Brossard, A. Barrau, and S. Bonnabel, "AI-IMU Dead-Reckoning," *IEEE Trans. on Intelligent Vehicles*, vol. 5, no. 4, pp. 585–595, 2020.

[11] A. Barrau and S. Bonnabel, "The Invariant Extended Kalman Filter as a Stable Observer," *IEEE Trans. on Auto. Control*, vol. 62, no. 4, pp. 1797–1812, 2016.

[12] P. van Goor, T. Hamel, and R. Mahony, "Equivariant Filter (EqF): A General Filter Design for Systems on Homogeneous Spaces," in *2020 59th IEEE Conf. on Decision and Control (CDC)*, IEEE, 2020, pp. 5401–5408.

[13] P. Geneva, K. Eckenhoff, W. Lee, Y. Yang, and G. Huang, "OpenVINS: A Research Platform for Visual-Inertial Estimation," in *2020 IEEE Intl. Conf. on Robotics and Auto.*, IEEE, 2020, pp. 4666–4672.

[14] S. Agarwal, K. Mierle, and T. C. S. Team, *Ceres Solver*, version 2.1, Mar. 2022. [Online]. Available: https://github.com/ceres-solver/ceres-solver.

[15] F. Dellaert, "Factor Graphs and GTSAM: A Hands-On Introduction," Georgia Institute of Technology, Tech. Rep., 2012.

[16] G. Grisetti, R. Kümmerle, H. Strasdat, and K. Konolige, "g2o: A General Framework for (Hyper) Graph Optimization," in *Proceedings of the IEEE Intl. Conf. on Robotics and Auto., Shanghai, China*, 2011, pp. 9–13.

[17] M. Brossard, A. Barrau, and S. Bonnabel, "A Code for Unscented Kalman Filtering on Manifolds (UKF-M)," in *IEEE Intl. Conf. on Robotics and Auto.*, 2020, pp. 5701–5708.

[18] C. Wang *et al.*, "PyPose: A Library for Robot Learning with Physics-based Optimization," *arXiv preprint arXiv:2209.15428*, 2022.

[19] G. Bourmaud, R. Mégret, A. Giremus, and Y. Berthoumieu, "From Intrinsic Optimization to Iterated Extended Kalman Filtering on Lie Groups," *Journal of Mathematical Imaging and Vision*, vol. 55, pp. 284–303, 2016.

[20] K Madsen, H. B. Nielsen, and O Tingleff, "Methods for Non-Linear Least Squares Problems," Technical University of Denmark, Tech. Rep., 2004, pp. 1–30.

[21] H. A. P. Blom and Y. Bar-Shalom, "The Interacting Multiple Model Algorithm for Systems with Markovian Switching Coefficients," *IEEE Trans. on Auto. Control*, vol. 33, no. 8, pp. 780–783, 1988.

[22] J. Ćesić, I. Marković, and I. Petrović, "Mixture Reduction on Matrix Lie Groups," *IEEE Signal Processing Letters*, vol. 24, no. 11, pp. 1719–1723, 2017.

[23] C. C. Cossette, A. Walsh, and J. R. Forbes, "The Complex-Step Derivative Approximation on Matrix Lie Groups," *IEEE Robotics and Auto. Letters*, vol. 5, no. 2, pp. 906–913, 2020.