juliacon

Groups and smooth geometry using LieGroups.jl

Ronny Bergmann[1] and Mateusz Baran[2]

[1]Department of Mathematical Sciences, Norwegian University of Science and Technology, Trondheim, Norway
[2]AGH University of Science and Technology, Kraków, Poland

ABSTRACT

`LieGroups.jl` is a Julia package that provides an interface to define Lie groups as well as the corresponding Lie algebra and Lie group actions. The package also offers a well-documented, performant, and well-tested library of such objects, with a focus on numerical computations in engineering.
This paper presents the main features of the interfaces and how that is integrated within the `JuliaManifolds` ecosystem. We further present an overview on existing Lie groups implemented in `LieGroups.jl` as well as how to get started to use the package in practice.

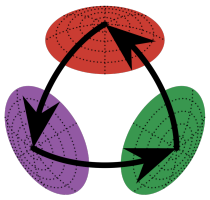
Keywords

Lie groups, differential geometry, Riemannian manifolds, numerical analysis, scientific computing

1. Introduction

In many situations, one encounters data that does not reside in a vector space. We can hence not use standard linear algebra tools to work with such data. For example in robotics, the configuration space of a rigid body in three-dimensional space is given by the special Euclidean group $\mathrm{SE}(3)$, consisting of all translations and rotations. A subset of these is the space of rotations, given by the special orthogonal group $\mathrm{SO}(3)$, or more generally $\mathrm{SO}(n)$ in n-dimensional space.
These are examples of Lie groups, formally defined as a smooth manifold equipped with a group structure. They have applications in physics, robotics, stochastic processes, information geometry, and many other areas see [6, 7], but are also interesting from their mathematical viewpoint [12] and their numerical aspects, for example when solving differential equations on Lie groups [13, 15].
The package `LieGroups.jl`[1] provides an easy access to both defining and using Lie groups within the Julia programming language [4] by defining an interface of Lie groups, as well as implementing a library of Lie groups, that can directly be used.
This paper provides an overview of the main features of `LieGroups.jl`, whose logo is shown on the right, as of version 0.1.9. After introducing some mathematical background in Section 2, we present the interface in Section 3 for the Lie group, the Lie algebra, and group actions. Section 4 provides an overview of all currently implemented Lie groups. Finally, in Section 5, we demonstrate how to get started and use `LieGroups.jl`.

[1]Available at juliamanifolds.github.io/LieGroups.jl/stable/, see also the zenodo archive [1].



Logo of `LieGroups.jl`.

Statement of Need

Several software packages for solving problems involving Lie groups exist, especially in the context of robotics and when solving differential equations or working with statistics on Lie groups. These then often contain Lie groups as a part or sub-module. Similar to the standalone packages like Lie++ [11], Sophus [16] in C++ or jaxlie [17] in Python/JAX or manif [8], these mainly focus on a small set of Lie groups, mostly $\mathrm{SO}(n)$ and $\mathrm{SE}(n)$ for $n = 2, 3$, sometimes also including simple product groups or the Gallilean group.
Two approaches that are a bit broader are the Lie groups defined within DiffMan [10] in MATLAB to solve differential equations on manifolds, and the Lie group implementations within the Python package geomstats [14], though they are a bit hard to explore within the documentation, since they are not distinguished from geometry interfaces, manifolds or metrics.
Formerly, the Julia package `Manifolds.jl` [2] contained some group operations, extending a few manifolds to Lie groups as well; the interface however ended up being too close to Riemannian features.
`LieGroups.jl` addresses this gap by providing both a flexible interface for defining new Lie groups and a comprehensive library of implementations, fully integrated within the `JuliaManifolds` ecosystem.

2. Mathematical Background

The following notation and definitions follow the text books [12], especially Chapters 2 and 9 therein. For more details on Riemannian manifolds, see also [9] or [5], especially Chapter 8 therein.

2.1 Lie groups

We denote a Lie group by $\mathcal{G} = (\mathcal{M}, \cdot)$ where $\mathcal{M}$ is a smooth manifold and $\cdot$ is the group operation. We call n the dimension of the manifold $\mathcal{M}$, denoted by $\dim(\mathcal{M}) = n$. A manifold $\mathcal{M}$ is a topological space that locally looks like an Euclidean space $\mathbb{R}^n$ for some

$n \in \mathbb{N}$, but globally may have a different topology. We further require $\mathcal{M}$ to be second-countable and Hausdorff, see [5, Sec. 8.2] for details.

That a manifold locally looks like $\mathbb{R}^n$ means that there exist a collection of *charts* $\varphi_i \colon U_i \to V_i \subseteq \mathbb{R}^n$ for open sets $U_i \subseteq \mathcal{M}$ and $V_i \subseteq \mathbb{R}^n$ such that the union of all U_i covers $\mathcal{M}$. Such a collection is called an *atlas* $\mathcal{A}$. If for any two charts φ_i, φ_j the transition map $\varphi_j \circ \varphi_i^{-1} \colon \varphi_i(U_i \cap U_j) \to \varphi_j(U_i \cap U_j)$ is smooth, then the atlas $\mathcal{A}$ is called a *smooth atlas* and the manifold $\mathcal{M}$ is called a *smooth manifold*. We refer to [5, Sec. 8.1] for more details on charts and atlases.

As an example, take the 2-dimensional sphere

$$\mathbb{S}^2 = \big\{p \in \mathbb{R}^3 \;\big|\; \|p\| = 1\big\},$$

which locally looks like $\mathbb{R}^2$, the corresponding charts φ_i are what you would expect in a classical atlas, but globally it is not homeomorphic to $\mathbb{R}^2$.

Finally we denote the *tangent space* at a point $p \in \mathcal{M}$ by $T_p\mathcal{M}$. This can be thought of as all "velocities" (direction and speed) in which a smooth curve $c(t)$ on the manifold can "pass through" the point $p = c(0)$ at time $t = 0$. Formally these velocities can be obtained by looking at the curve "through a chart" $\varphi \in \mathcal{A}$ and consider the derivative of $\varphi \circ c \colon \mathbb{R} \to \mathbb{R}^n$. Since we consider smooth manifolds, this derivative can be defined independent of the chosen chart due to the smoothness of the transition maps. Finally, the set of all "velocities" is set the equivalence classes of derivatives of these smooth curves [5, Sec. 8.4]. Each such tangent space $T_p\mathcal{M}$ is a n-dimensional vector space. We call the disjoint union of all tangent spaces

$$T\mathcal{M} = \dot{\bigcup}_{p\in\mathcal{M}} T_p\mathcal{M}$$

the *tangent bundle* of $\mathcal{M}$. For the example of the 2-dimensional sphere $\mathbb{S}^2$, the tangent space at a point $p \in \mathbb{S}^2$ is given by all vectors in $\mathbb{R}^3$ that are orthogonal to p, or in other words the plane that is tangent to the sphere at p.

A group operation $\cdot \colon \mathcal{G} \times \mathcal{G} \to \mathcal{G}$ is a function that satisfies the group axioms: associativity, existence of an identity element $e \in \mathcal{G}$, and existence of inverses $g^{-1} \in \mathcal{G}$ for all $g \in \mathcal{G}$. Furthermore the group operation $\cdot$ (on $\mathcal{G} \times \mathcal{G}$) and the inversion map $\iota \colon \mathcal{G} \to \mathcal{G}, g \mapsto g^{-1}$ have to be smooth maps. As an example, consider the special orthogonal group $\mathrm{SO}(n)$, consisting of all $n \times n$ orthogonal matrices, with determinant 1, that is, for $p \in \mathrm{SO}(n)$, we have $p^{\mathrm{T}}p = I$ and $\det(p) = 1$ together with the group operation $\cdot$ given by matrix multiplication. For $n = 2$ these are rotations in the plane, hence each operation can be identified with an angle $\alpha \in [-\pi, \pi)$[2], or, continuously, with the circle. The identity element is given by the identity matrix I (or the angle $\alpha = 0$) and the inverse of a rotation matrix is given by its transpose p^{T} (or an angle $-\alpha$ when $\alpha \neq -\pi$ and $-\pi$ otherwise).

2.2 The Lie algebra

The tangent space at the identity element $e \in \mathcal{G}$, denoted by $\mathfrak{g} = T_e\mathcal{G}$, plays a special role and is called the *Lie algebra* $\mathfrak{g}$ of the Lie group $\mathcal{G}$. The reason is that to represent arbitrary tangent vectors $X \in T_g\mathcal{G}$ at a point $g \in \mathcal{G}$ we can use the group operation: we denote by $\lambda_g(h) = g \cdot h$ the left multiplication with $g, h \in \mathcal{G}$. For a map $f \colon \mathcal{G} \to \mathcal{H}$ between two Lie groups, its *differential* (or push forward) at a point $g \in \mathcal{G}$ is a linear map $Df(g) \colon T_g\mathcal{G} \to T_{f(g)}\mathcal{H}$ between the corresponding tangent spaces. Using the differential $D\lambda_g(h) \colon T_h\mathcal{G} \to T_{g\cdot h}\mathcal{G}$, we can generate a so-called *left-invariant vector field* defined by $\mathcal{X}(g) \coloneqq D\lambda_g(e)[X]$ which is uniquely determined by the choice of $X \in \mathfrak{g}$[3]. Hence we can identify tangent vectors $\mathcal{X}(g) \in T_g\mathcal{G}$ at arbitrary points $g \in \mathcal{G}$ with X from the Lie algebra $\mathfrak{g}$.

As a first example, consider $\mathcal{G} = (\mathbb{R}^n, +)$, where the tangent space at any point g is again $\mathbb{R}^n$, especially at the identity $e = 0$. We further have $\lambda_g(h) = g + h$ and hence $D\lambda_g(h)[X] = X$ for all $g, h, X \in \mathbb{R}^n$. Here, a tangent vector X induces the constant vector field $\mathcal{X}(g) = X$ for all $g \in \mathbb{R}^n$.

As a second example, consider again the special orthogonal group $\mathrm{SO}(n)$. The tangent space at the identity element $e = I$ is given by the Lie algebra $\mathfrak{so}(n) = T_e\mathrm{SO}(n) = \{X \in \mathbb{R}^{n\times n} \mid X = -X^{\mathrm{T}}\}$ that consists of all skew-symmetric $n \times n$ matrices. For $g, h \in \mathrm{SO}(n)$ and $X \in \mathfrak{so}(n)$ we have $\lambda_g(h) = g \cdot h$ and hence $D\lambda_g(h)[X] = g \cdot X$. Here, the tangent vector $X \in \mathfrak{so}(n)$ induces the left-invariant vector field $\mathcal{X}(g) = g \cdot X \in T_g\mathrm{SO}(n)$, $g \in \mathrm{SO}(n)$. In other words, this formulation allows to represent tangent vectors $Y \in T_g\mathrm{SO}(n)$ also using $X = g^{-1}Y \in \mathfrak{g}$.

An important tool to "move around" on the Lie group is the exponential. The (Lie group) exponential (function) $\exp \colon \mathfrak{g} \to \mathcal{G}$ maps elements from the Lie algebra to the Lie group, and is formally defined [12, Def. 9.2.2] by evaluating the unique curve $\gamma \colon \mathbb{R} \to \mathcal{G}$ that fulfils the differential equation

$$\gamma'(t) = D\lambda_{\gamma(t)}(e)[X] \quad \text{with} \quad \gamma(0) = e \text{ and } \gamma'(0) = X$$

at time $t = 1$, that is $\exp(X) = \gamma(1)$. This can be interpreted as starting at the identity element $e \in \mathcal{G}$ and following the curve whose velocity at each point is given by the left-invariant vector field induced by $X \in \mathfrak{g}$ for one time unit.

For $(\mathbb{R}^n, +)$ the exponential is given by $\exp(X) = X$, $X \in \mathbb{R}^n$ and for $\mathrm{SO}(n)$ it is given by the matrix exponential $\exp(X) = \mathrm{e}^X$ for all $X \in \mathfrak{so}(n)$. Additionally for the unit circle in the complex plane, using the group operation of multiplication, the exponential function is given by the complex exponential $\exp(X) = \mathrm{e}^{\mathrm{i}X}$ for all $X \in \mathbb{R}$.

From the interpretation of the exponential function to follow a curve starting at the identity, we can define the (Lie group) *exponential map* to "start from an arbitrary point" $g \in \mathcal{G}$ by $\exp_g \colon \mathfrak{g} \to \mathcal{G}$, $\exp_g(X) \coloneqq g \cdot \exp(X)$ for all $X \in \mathfrak{g}$.

Concerning a metric on the tangent spaces, smooth manifolds are turned into Riemannian manifolds when they are equipped with a Riemannian metric $\langle \cdot, \cdot \rangle_p \colon T_p\mathcal{M} \times T_p\mathcal{M} \to \mathbb{R}$ for each point $p \in \mathcal{M}$ that smoothly varies with p. For Lie groups, we can use a single inner product $\langle \cdot, \cdot \rangle$ on the Lie algebra $\mathfrak{g}$ and use the change in representation as mentioned above to define

$$\langle X, Y \rangle_g = \langle D\lambda_{g^{-1}}(g)[X], D\lambda_{g^{-1}}(g)[Y] \rangle_e$$

for all $X, Y \in T_g\mathcal{G}$ and $g \in \mathcal{G}$. Representing tangent vectors at arbitrary points $g \in \mathcal{G}$ using the Lie algebra $\mathfrak{g}$ yields here, that we can use the single inner product directly to evaluate this Riemannian metric. Hence representing tangent vectors using the Lie algebra is the default in the following.

For the two examples above we obtain the Euclidean inner product $\langle X, Y \rangle = X^{\mathrm{T}}Y$ for $X, Y \in \mathbb{R}^n$ for $(\mathbb{R}^n, +)$ and the Frobenius inner product $\langle X, Y \rangle = \mathrm{trace}(X^{\mathrm{T}}Y)$, $X, Y \in \mathfrak{so}(n)$ can be used for $\mathrm{SO}(n)$.

[2] Note that the identification is not continuous.

[3] Analogously, one can use the right multiplication $\rho_g(h) = h \cdot g$ and its differential $D\rho_g(e)$ to define right-invariant vector fields. `LieGroups.jl` uses left-invariant vector fields as default.

Note that a Riemannian metric can also be used to define an exponential map using the Levi-Civita affine connection. This exponential map often differs from the Lie group exponential, in particular many Lie groups, such as the special Euclidean group in two or more dimensions, can not have a metric compatible with the Lie group exponential.

2.3 Group Actions

A *group action* of a Lie group $\mathcal{G}$ on a smooth manifold $\mathcal{M}$ is a smooth map $\sigma\colon \mathcal{G} \times \mathcal{M} \to \mathcal{M}$ such that for all $g, h \in \mathcal{G}$ and $p \in \mathcal{M}$ it holds that $\sigma(e, p) = p$ and $\sigma(g, \sigma(h, p)) = \sigma(g \cdot h, p)$.[4]
Informally a group action describes how elements of the Lie group $\mathcal{G}$ "act on" points on the manifold $\mathcal{M}$. As an example, think of the special orthogonal group $\mathrm{SO}(3)$ acting on points on Euclidean space $\mathbb{R}^3$ "moving" them somewhere by rotating them around the origin. We obtain the group action $\sigma\colon \mathrm{SO}(3) \times \mathbb{R}^3 \to \mathbb{R}^3$, $\sigma(R, x) \coloneqq Rx$.
The same action can also be applied to points from the sphere $\mathbb{S}^2$, resulting in a similar group action $\sigma\colon \mathrm{SO}(3) \times \mathbb{S}^2 \to \mathbb{S}^2$, $\sigma(R, p) \coloneqq Rp$.

3. The interface

Since a Lie group $\mathcal{G}$ consists of two main components, the smooth manifold $\mathcal{M}$ and the group operation $\cdot$, we can reuse existing functionality from the existing interface for manifolds provided by `ManifoldsBase.jl`, and later concrete manifolds provided by `Manifolds.jl` [2]. This is done in a transparent way, i.e. the `AbstractLieGroup` itself is a subtype of `AbstractManifold` from `ManifoldsBase.jl` and can hence also be used in all existing places, as for example optimization on manifolds provided by `Manopt.jl` [3]. In notation, we use typewriter font to denote functions in the interface, but we keep the same letters of notation as before, i.e. $\mathcal{G}$ for a Lie group is `G` in code, a point $g \in \mathcal{G}$ is `g` in code, and so on, just that for the Lie algebra we use `g` in both text and code.
The interface follows the philosophy of `ManifoldsBase.jl`, that the Lie group or algebra is the first argument and even a mutated argument comes second. Similarly, if a function computes something like a new point on the Lie group or a tangent vector, there also exists a variant, that computes this in-place. For example `identity_element(G)` returns the identity element e of a Lie group G in a default representation, `identity_element!(G, e)` writes the result into the pre-allocated variable `e`, which can also be used with other representation types. To generate the identity in a specific representation by objects of type `T`, one can use `identity_element(G, T)`. For example, a points on the special Euclidean group $\mathrm{SE}(n)$ can be represented either as a $(n+1) \times (n+1)$ matrix or a `ArrayPartition` with rotation and translation stored separately.

3.1 Lie groups

The main type for Lie Groups is the `LieGroup{𝔽, O, M} <: AbstractLieGroup{𝔽, O, M}`, which contains a manifold `M <: AbstractManifold{𝔽}` as well as the group operation `O <: AbstractGroupOperation`, where $\mathbb{F}$ is the set of scalars used in the representation of the manifold, usually $\mathbb{F} = \mathbb{R}$ and in some cases $\mathbb{F}$ is the field of complex numbers $\mathbb{C}$ or the ring of quaternions $\mathbb{H}$.

[4] This is the convention for left actions. Alternatively, right actions fulfil $\sigma(g, \sigma(h, p)) = \sigma(h \cdot g, p)$.

In general, when a function within the interfaces of `JuliaManifolds` returns a point, a tangent vector or any other mutable object, there always exist two variants of the function, one that allocates a new object and returns it, and one that takes an additional argument to write the result into. For example for a function `f(G, g)` that returns a point on the Lie group `G`, there also exists a variant `f!(G, h, g)` that computes the result in-place of `h`, possibly using the memory also for interims results. All methods are written such that the result is correct even if arguments `h` and `g` use the same memory. By default the allocating variant is implemented by allocating memory accordingly and calling the in-place variant. This default can be overridden, for example for performance reasons.

Topological functions. Points and tangent vectors can be represented by the same Julia objects, although they are distinct from a topological viewpoint.[5] Separate types are necessary when one Lie group has *different representations* of points, that have to be distinguished. When there are different representations, it is recommended to introduce a point type for each representation and make the default one fall back to (just) using arrays. Several functions in the interface return points or tangent vectors, where the type might not be known from the input argument. For such a function, for example again a function `f(G)` a positional argument `T` can be used to specify the desired return type, i.e. `f(G, T)` returns a point or tangent vector of type `T`. The in-place variant does not require this, since in its signature of the form `f!(G, g::T)` the type `T` is known from the argument `g`.
Alternatively to points, the same group structure can be defined over different topological manifolds. This is for example the case for the circle group $\mathbb{S}^1$, which can be represented as angles, points on the unit circle in $\mathbb{R}^2$ or as complex numbers with unit norm, all three of which are different manifolds to be used internally. To access the underlying manifold, the function `base_manifold(G)` is provided.
The following functions are available and pass directly to the underlying manifold interface from `ManifoldsBase.jl`: `is_approx(G, g, h)` to check for (approximate) equality of two points, `is_point(G, g; error=:none)` to check if a point is a valid point on the manifold, where the keyword can be used to throw an `:error`, a `:warn` or an `:info`. Similarly `manifold_dimension(G)` to get the dimension of the manifold, `project(G, q)`, to project a point onto the manifold, as well as `rand(G)` and `rand(rng, G)` to sample a random point from the manifold using a random number generator `rng` also pass directly on to the manifold. To access the underlying manifold of the Lie group one can use `base_manifold(G)`.

Group operation related functions. For the group operation $\cdot$ of a Lie group $\mathcal{G}$, the abstract supertype `AbstractGroupOperation` is mandatory. There are two main group operation types provided in `LieGroups.jl`. On the one hand operations that fall back to using $\cdot = +$ or $\cdot = *$, where the latter has two variants, the one where it is Abelian (like for numbers) and where it is not (like for matrices). On the other hand, specific meta groups like the (direct) product of two or more groups, the case where the product is taken for just one group, i.e. the power group, or semidirect products. The available

[5] `LieGroups.jl` introduces an abstract type `AbstractLieGroupPoint <: AbstractManifoldPoint`, that is used to represent points on the Lie group. This abstract supertype is not necessary, points can be represented by arrays or other types as well.

Table 1. : Group operations available in `LieGroups.jl`.

Group operation ·	comment/code
`AdditionGroupOperation`	falls back to `+`
`AbelianMultiplicationGroupOperation`	falls back to `*`
`MatrixMultiplicationGroupOperation`	falls back to `*`
`PowerGroupOperation{Op}`	`PowerLieGroup`
`ProductGroupOperation{Ops}`	product Lie groups
`SemidirectProductGroupOperation`	semidirect products

group operation types are summarized in Table 1. More on these meta groups is explained in Subsection 4.1.
For all these group operations, the following functions have default implementations. They might not be the most performant ones for every case, but provide working implementations out-of-the-box. It is always possible to override these by defining a new group operation type and implementing the following functions for a `LieGroup` with that new group operation type.
The identity element can be represented as a value of a special type, `e = Identity(G)` or `Identity(op)` where `op` is the group operation of `G`. This allows for dispatching on the type when defining functions and avoiding unnecessary allocations. If the actual value is needed, one can call `identity_element(G)` or `identity_element(G, typeof(g))`, where `g` is an element of the group. The first variant will always generate a point in the default representation of the group, while the latter will generate a point in the same representation as `g`.
The two central functions are `compose(G, g, h)` to compute the group operation $g \cdot h$ for two points $g, h \in \mathcal{G}$ and the inverse `inv(G, g)` which computes g^{-1} for a point $g \in \mathcal{G}$. Another function is $c_g(h) = g \cdot h \cdot g^{-1}$ called *conjugation*, which is available as `conjugate(G, g, h)`.
For these three functions, also the differentials are available: adding a `diff_` prefix to the function name and a final argument for the Lie algebra tangent vector, for compose additionally the argument with which to differentiate to, i.e. `diff_left_compose(G, g, h, X)` and `diff_right_compose(G, g, h, X)`, respectively.
Finally, there is a specific function for the differential of the conjugate at the identity $h = e$, called `adjoint(G, g, X)`, as well as the combinations `inv_left_compose(G, g, h)` for computing $g^{-1} \cdot h$ and `inv_right_compose(G, g, h)` for computing $g \cdot h^{-1}$. All three fall back to the previously defined functions, but provide an interface to possibly implement more efficient variants in case such exist.

Exponential and logarithm. In `LieGroups.jl`, the group exponential function is given by `exp(G, X)`. If we want to "start walking" from another point, we can "move" (or interpret X) as being from the tangent space at some point g and obtain by the chain rule the exponential map $\exp_g \colon \mathfrak{g} \to \mathcal{G}$ defined by $\exp_g(X) := g \cdot \exp(X)$. In `LieGroups.jl`, the exponential map is implemented as `exp(G, g, X)`. Note that on the underlying manifold, there is a further exponential map, the Riemannian exponential map. This usually differs from the Lie group exponential map. The Riemannian exponential map `exp(M, p, X)` can be distinguished in that the first argument is a manifold, and the tangent vector `X` has to be from the tangent space at the point `p` on the manifold. Here, again, to access the Riemannian exponential, one can use the base manifold of the Lie group, i.e. `exp(base_manifold(G), g, X)` and have to make sure that `X` is from the tangent space at `g`.
Locally around the identity element $e \in \mathcal{G}$, the exponential map is a diffeomorphism, i.e. there exists an inverse map on some neighbourhood of e. The logarithmic function $\log \colon \mathcal{G} \to \mathfrak{g}$ and logarithmic map $\log_g \colon \mathcal{G} \to \mathfrak{g}$have the function signatures `log(G, g)`and `log(G, h, g)`, respectively, with the same caveat to the Riemannian logarithmic map as for the exponential.
When the exponential and logarithmic map are not known in closed form, it might be beneficial to use retractions and inverse retractions instead, respectively. These are first or second order approximations of the exponential and logarithmic map, respectively and their interface is already provided in `ManifoldsBase.jl`[6]. In `LieGroups.jl`, one can either implement new variants based on a subtype of `AbstractRetractionMethod` and `AbstractInverseRetractionMethod`, resp., or use the wrappers `BaseManifoldRetraction` and `BaseManifoldInverseRetraction`, resp., to directly use the retraction and inverse retraction from the underlying manifold.

3.2 Lie algebras

Similar to points on the Lie group, when representing elements $X \in \mathfrak{g}$ from a Lie algebra, we do not type the general functions of the interface. This allows to use either plain arrays or own structures to represent these in code[7]. Keep in mind, that the Lie algebra is a vector space, so that addition, subtraction and scalar multiplication, as well as their broadcasted variants, are assumed to be defined in case you use an individual data type.
A major difference to the usual representation of tangent vectors on the underlying manifold is, that here the usual representation is done in the Lie algebra.
A central function on the Lie algebra is the Lie bracket $[\cdot, \cdot] \colon \mathfrak{g} \times \mathfrak{g} \to \mathfrak{g}$, which is available as `lie_bracket(g, X, Y)`. For more details on the Lie bracket see [12, Ch. 5].

Topological basics. Given a Lie group `G`, we obtain the Lie algebra by calling `g = LieAlgebra(G)`. To access the Lie group again, use `base_lie_group(g)`. Similarly `base_manifold(g)` returns the underlying manifold of the Lie group.
As a technical detail, the Lie algebra is modelled as a tangent space[8] using the already mentioned `Identity(G)` as base point. A zero vector is generated via `zero_vector(g)` for the default representation and `zero_vector(g, T)` for a specific representation type `T`.
The main topological function is to test the validity of a tangent vector `is_vector(g, X; error=:none)` using the same `error=` keyword as `is_point` on the Lie group.

Vector space related functions. For the following functions related to vector space features, using the Lie algebra $\mathfrak{g}$ as first argument is equivalent to specifying the Lie Group G and an arbitrary point $g \in \mathcal{G}$. This yields that the Lie group complies with the general interface for manifolds.
The inner product and norm on the Lie algebra are available as `inner(g, X, Y)` and `norm(g, X)`, respectively. Furthermore, there are two functions to convert between a coordinate-free representation of X as a tangent vector and its representation in coordinates of a basis.
Given a vector $c \in \mathbb{R}^{\dim(\mathfrak{g})}$, we obtain the corresponding tangent vector by calling `get_vector(g, c, B)` where `B` is a basis of the tangent space, i.e. a subtype of `AbstractBasis` and defaults within `LieGroups.jl` to

[6] see juliamanifolds.github.io/ManifoldsBase.jl/stable/retractions/.
[7] The optional abstract supertype `AbstractLieAlgebraTangentVector`, which is a `AbstractTangentVector`, is provided to also distinguish different representations here.
[8] see the documentation of ManifoldsBase.TangentSpace

Table 2. : Group actions types available in `LieGroups.jl`.

Group action type	comment/code
`AdditionGroupAction`	falls back to `+`
`ColumnwiseGroupAction{A}`	of a group action type `A`
`(Inverse)LeftGroupOperationAction`	requires $\mathcal{M} = \mathcal{G}$
`LeftMultiplicationAction`	falls back to `*`
`(Inverse)RightGroupOperationAction`	requires $\mathcal{M} = \mathcal{G}$
`RotationAroundAxisAction`	
`RowwiseGroupAction{A}`	of a group action type `A`

`DefaultLieAlgebraOrthogonalBasis()`. When a Lie group has different representations of points and tangent vectors, these are distinguished by calling `get_vector(g, c, B, T)` specifying the tangent vector type. Given a tangent vector $X \in \mathfrak{g}$, we obtain its coordinates by calling `get_coordinates(g, X, B)`, where the basis is again optional.
For the case of the default, the `DefaultLieAlgebraOrthogonalBasis()` the more commonly used names `hat(g, c)` (again with an optional vector type `T`) and `vee(g, X)` are implemented

Push forward and pull back of tangent vectors. To get a tangent vector $X_g \in T_g\mathcal{G}$ at point $g \in \mathcal{G}$ from its Lie algebra representation $X \in \mathfrak{g}$ we have to use the push forward of the left multiplication with g, i.e. $D\lambda_g(e)[X]$. This is implemented as `push_forward_tangent(G, g, X)`. Conversely, to represent a tangent vector $Y \in T_g\mathcal{G}$ back in the Lie algebra $\mathfrak{g}$, we have to use the pull back of the left multiplication with g^{-1}, i.e. $D\lambda_{g^{-1}}(g)[Y]$. This is implemented as `pull_back_tangent(G, g, Y)`.

Jacobians. On a Euclidean space, the terms differential and Jacobian are often used interchangeably. Within differential geometry, the differential (or push forward) of a smooth map $f\colon \mathcal{M} \to \mathcal{N}$ between two manifolds is a map between the corresponding tangent spaces. We denote it by $Df(p)\colon T_p\mathcal{M} \to T_{f(p)}\mathcal{N}$, where $p \in \mathcal{M}$. The differential is a linear map between the tangent spaces. However, in the coordinate-free representation of tangent vectors, no matrix representation of this linear map is constructed. But as soon as we choose bases for the tangent spaces and we represent tangent vectors in coordinates (cf. `get_coordinates` previously), we can represent the differential as a matrix, called the Jacobian matrix.
Currently for both `exp` as well as `conjugate`, the Jacobian is implemented. Note that for `exp`, the domain is the Lie algebra. Since that is a vector space, its tangent space(s) can be identified with the Lie algebra again. Hence both Jacobians are maps from the Lie algebra to itself. The signatures are `jacobian_exp(G, p, X, B)` and `jacobian_conjugate(G, g, X, B)`, where `B` is again the basis to represent the tangent vectors in coordinates and defaults to `DefaultLieAlgebraOrthogonalBasis()`.

3.3 Group actions

Group actions as defined in Subsection 2.3 are implemented as a struct `GroupAction{G, M, A} <: AbstractGroupAction` containing the Lie group `G <: AbstractLieGroup`, the manifold `M`, and a group action tpe `A` in order to distinguish possible different ways a group could act on a manifold. Table 2 summarizes the currently available group action types in `LieGroups.jl`.
The main functions then are `apply(a::GroupAction, g, p)` to compute $\sigma(g, p)$ for a group action `a`, where $g \in \mathcal{G}$ and $p \in \mathcal{M}$, as well as its differential `diff_apply(a::GroupAction, g, p, X)` to compute $D_{\mathcal{M}}\sigma(g, p)[X]$ for a tangent vector $X \in T_p\mathcal{M}$ with respect to the manifold argument as well as `diff_group_apply(a::GroupAction, g, p, Y)` to compute $D_{\mathcal{G}}\sigma(g, p)[Y]$ for a Lie algebra element $Y \in \mathfrak{g}$.
Due to the property that $\sigma(g^{-1}, \sigma(g, p)) = p$ for all $g \in \mathcal{G}$ and $p \in \mathcal{M}$, there is also an inverse group action available $\sigma^{-1}(g, p) = \sigma(g^{-1}, p)$, which can be obtained by calling `inv(a)`.
Note that while the group action $\sigma(g, \sigma(h, p)) = \sigma(g \cdot h, p)$ "appends" a new action (w.r.t. g) on the left, for the inverse we have

$$\begin{aligned}\sigma^{-1}(g, \sigma^{-1}(h, p)) &= \sigma(g^{-1} \cdot h^{-1}, p) = \sigma((h \cdot g)^{-1}, p)\\ &= \sigma^{-1}(h \cdot g, p)\end{aligned}$$

for all $g, h \in \mathcal{G}$. This is a so-called right action. Similarly for a right group action the inverse is a left group action. This is taken into account when using `apply`.

Table 3. : Implemented Lie groups in `LieGroups.jl` version 0.1.9.

Lie Group $\mathcal{G}$	**Symbol**	comment/code
`CircleGroup()`	$\mathbb{S}^1$	3 representations
`GeneralLinearGroup(n, F)`	$\mathrm{GL}(n, \mathbb{F})$	$\mathbb{F} \in \{\mathbb{R}, \mathbb{C}\}$
`HeisenbergGroup(n)`	$\mathrm{H}(n)$	
`OrthogonalGroup(n)`	$\mathrm{O}(n)$	
`PowerLieGroup(G, n)`	$\mathcal{G}^n$	`G^n`
`ProductLieGroup(G1, G2,...)`	$\mathcal{G}_1 \times \mathcal{G}_2 \times \dots$	`G1` $\times$ `G2` $\times \dots$
Semidirect product group	$\mathcal{G}_1 \ltimes \mathcal{G}_2$	`G1` $\ltimes$ `G2`
	$\mathcal{G}_1 \rtimes \mathcal{G}_2$	`G1` $\rtimes$ `G2`
`SpecialEuclideanGroup(n)`	$\mathrm{SE}(n)$	
`SpecialGalileanGroup(n)`	$\mathrm{SGal}(n)$	
`SpecialLinearGroup(n, F)`	$\mathrm{SL}(n, \mathbb{F})$	$\mathbb{F} \in \{\mathbb{R}, \mathbb{C}\}$
`SpecialOrthogonalGroup(n)`	$\mathrm{SO}(n)$	
`SpecialUnitaryGroup(n)`	$\mathrm{SU}(n)$	
`SymplecticGroup(n)`	$\mathrm{Sp}(2n)$	
`TranslationGroup(n; field=F)`	$(\mathbb{F}^n, +)$	$\mathbb{F} \in \{\mathbb{R}, \mathbb{C}, \mathbb{H}\}$
`UnitaryGroup(n, F)`	$\mathrm{U}(n)$	$\mathbb{F} \in \{\mathbb{C}, \mathbb{H}\}$
`ValidationLieGroup(G)`	wraps `G` for numerical verification	

4. Implemented Lie groups

Table 3 summarizes the currently implemented Lie groups in `LieGroups.jl` 0.1.9.

4.1 Meta Lie groups

There are three Lie groups that are built upon other Lie groups. We mention them here briefly and point out specific functions and features that are additionally available for these.

Product Lie group. Given two Lie groups $\mathcal{G} = (\mathcal{M}, *)$ and $\mathcal{H} = (\mathcal{N}, \diamond)$, their (direct) product $\mathcal{G} \times \mathcal{H}$ is again a Lie group $(\mathcal{M} \times \mathcal{N}, \cdot)$ with group operation defined component-wise, i.e.

$$(g_1, h_1) \cdot (g_2, h_2) = (g_1 * g_2, h_1 \diamond h_2)$$

for $g_1, g_2 \in \mathcal{G}$ and $h_1, h_2 \in \mathcal{H}$. Since $\times$ is a binary operator in Julia, given two Lie Groups `G,H`, their product Lie group can be constructed via `G` $\times$ `H`. The same applies for more than two groups, i.e. you can construct arbitrary long product Lie groups `G1` $\times$ `G2` $\times$ `G3` $\times \dots$.
As for the representation of points and tangent vectors, the default representation requires to load `RecursiveArrayTools.jl`[9].

[9] see docs.sciml.ai/RecursiveArrayTools/stable/

The binary operator $\times$ automatically flattens the input, so that the points are represented by non-nested `ArrayPartition` objects. If you want to construct products of Lie groups where points are represented by nested `ArrayPartition` objects, for example $(\mathcal{G}_1 \times \mathcal{G}_2) \times \mathcal{G}_3$, use `ProductLieGroup(ProductLieGroup(G1, G2), G3)` instead.

Power Lie groups. Mathematically power groups are product Lie groups where all manifolds are the same. Computationally we can sometimes benefit by exploiting this special structure. Therefore the `PowerManifold(G, n1, n2, ...)` is provided to construct the $n_1 \times n_2 \times \ldots$-fold Cartesian product of the Lie group $\mathcal{G}$ with itself. As a shortcut, you can use `G^n` to construct the n-fold Cartesian product of the Lie group $\mathcal{G}$ with itself.
Note that inheriting from the power manifold, there are two different data types to represent data, per default a single large array. Alternatively, one can use the constructor `PowerLieGroup(G, NestedPowerRepresentation(), n)` to use the other representation of points and tangent vectors, namely as a vector of points/tangent vectors.
To abstract the access of elements here, similar to the `PowerManifold` in `ManifoldsBase.jl`, the access functions `g[G, i]` are available.

Semidirect product Lie groups. Given two Lie groups $\mathcal{G} = (\mathcal{M}, *)$ and $\mathcal{H} = (\mathcal{N}, \diamond)$ as well as a group action $\sigma\colon \mathcal{G} \times \mathcal{H} \to \mathcal{H}$, the *(left) semidirect product Lie Group* $\mathcal{G} \ltimes \mathcal{H} = (\mathcal{M} \times \mathcal{N}, \cdot)$ is defined via the group operation given by

$$(g_1, h_1) \cdot (g_2, h_2) = (g_1 * g_2, h_1 \diamond \sigma(g_1, h_2))$$

This semidirect product Lie group is constructed calling `LeftSemidirectProductLieGroup(G, H, a)`, where the `GroupAction a` is optional, since one can define a default action by setting `default_left_action(G, H)`. With this default, also the shortcut `G` $\ltimes$ `H` is available. Similarly, the *right semidirect product Lie Group* $\mathcal{H} \rtimes \mathcal{G}$ is defined analogously with the positions of the groups are swapped in the group operation and the default for the group action is given by `default_right_action(G, H)`. This can be constructed via `H` $\rtimes$ `G` then as well.

4.2 Decorators for Lie groups

The decorator pattern is used within the `JuliaManifolds` ecosystem to add or modify existing functionality. For Lie groups there are currently two such decorators available: one for adding a custom metric structure (inner product) to the group and one for numerical validation.

Specifying a different inner product. By implementing `inner(g, X, Y)` and `norm(g, X)` for a Lie algebra `g = LieAlgebra(G)`, the Lie group `G` is equipped with a certain inner product on the Lie algebra and hence on all tangent spaces to the Lie group. This is an implicit choice of a default metric on the Lie group, that is considered the default thereafter.
If more than one metric is used in applications, a second metric can be introduced by wrapping the Lie group into a `MetricLieGroup`, i.e. `G2 = MetricLieGroup(G, m::M)`, where `M` has to be a subtype of `AbstractMetric` available from `Manifolds.jl`[10]. All functions unrelated to the metric are forwarded from `G2` to `G`, while all functions related to the inner product have to be specified for `G2` anew.

[10] where analogously a manifold of type `MetricManifold` is defined.

Adding numerical validation to a Lie group. The `ValidationLieGroup` is a special Lie group that is intended for numerical verification and debugging of code using Lie groups.
The `ValidationLieGroup` is implemented as a wrapper around any existing Lie group, `G2 = ValidationLieGroup(G)`. It provides additional functionality to check the correctness of computations involving the Lie group. Most prominently, all input and output of group functions are checked for validity, e.g. by calling `is_point` and `is_vector` on all points and tangent vectors, respectively. Similarly, the inner manifold is wrapped into the similar decorator from `ManifoldsBase.jl`, the `ValidationManifold`. While by default failures in these checks result in errors, this can be changed by setting the `error=` keyword of the `ValidationLieGroup` constructor to either `:warn` or `:info`.

4.3 Concrete Lie groups

From the list of available Lie groups in Table 3, we want to highlight some specific features of a few of them.
For the `CircleGroup()`, three different representations of points are available: as angles in $[-\pi, \pi)$, as complex numbers with unit norm, and as points on the unit circle in $\mathbb{R}^2$. These are constructed by specifying the representation of points, i.e. `CircleGroup(`$\mathbb{R}$`)`, `CircleGroup(`$\mathbb{C}$`)`, and `CircleGroup(`$\mathbb{R}^2$`)`, respectively. Note that each of these representations uses a different underlying manifold, namely `Circle(`$\mathbb{R}$`)`, `Circle(`$\mathbb{C}$`)`, and `Sphere(1)`, respectively. This is a case, where it is not only the representation of points and tangent vectors that differ, but indeed even the underlying manifold.
The special Euclidean group $\mathcal{G} = \mathrm{SE}(n)$, there are three different representations available:
The first one is the matrix representation, where points are represented as $(n+1) \times (n+1)$ matrices combining rotation and translation, i.e. as

$$g = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix},$$

where $R \in \mathrm{SO}(n)$ and $t \in \mathbb{R}^n$.
The second representation is the tuple representation, where points are represented as tuples $g' = (R, t)$ and the third is the tuple representation with the translation in the first component, i.e. $g'' = (t, R)$. These two are available when `RecursiveArrayTools.jl` is loaded and specifying the `variant=:left` (default) or `variant=:right`, respectively. The matrix representation works on either `variant`.
For all three representations, one can access the component semantically by writing `g[G, :Rotation]` and `g[G, :Translation]`, which works independent of the representation used.

5. Examples how to use `LieGroups.jl`

In this section, we provide examples of how to use the `LieGroups.jl` package in Julia.

5.1 A Beziér curve on a Lie group

In the first example we illustrate how to construct and plot a Beziér curve on a Lie group, here the special Euclidean group $\mathrm{SE}(2)$.
We first can generalize the de-Casteljau algorithm for Beziér curves to Lie groups by replacing the linear interpolation step with using the exponential and logarithmic map.

Code 1: de-Casteljau algorithm on a Lie group.

```
using LieGroups, ManifoldsBase, RecursiveArrayTools
using CairoMakie
"""
    deCasteljauLieGroup(G, cp, t)

Evaluate the Beziér curve at parameter `t` given a
vector of control points `cp` on the Lie group `G`.
"""
function deCasteljauLieGroup(
        G::AbstractLieGroup, cp::Vector{P}, t::Real
    ) where {P}
    n = length(cp)
    points = [copy(G, g) for g in cp]
    for r in 1:n-1
        for i in 1:n-r
            X = log(G, points[i], points[i+1])
            exp!(G, points[i], points[i], t * X)
        end
    end
    return points[1]
end
```

Code 2: Evaluating Bézier curves on $\mathrm{SE}(2)$ and $\mathrm{SO}(2) \times \mathcal{T}(2)$.

```
SO2 = SpecialOrthogonalGroup(2)
R2 = TranslationGroup(2)
SO2xR2 = SO2 × R2
ts = range(0, 1, length=35)
q1 = [
    deCasteljauLieGroup(SO2xT2, cp, t) for t in ts
]
SE2 = SpecialEuclideanGroup(2)
q2 = [deCasteljauLieGroup(SE2, cp, t) for t in ts]
```

In Code 1, we define the general de-Casteljau algorithm on a Lie group `G`, given a vector of control points `cp` and a parameter `t` in $[0, 1]$.
We can now compare the algorithm on the Lie group $\mathrm{SE}(2)$ with the corresponding Bézier curve on the (direct) product Lie group $\mathrm{SO}(2) \times \mathcal{T}(2)$, where $\mathcal{T}(2) = (\mathbb{R}^2, +)$ is the `TranslationGroup(2)`. Denote by R_α the rotation matrix in $\mathrm{SO}(2)$ for an angle $\alpha \in [0, 2\pi)$. We consider the control points

$$g_1 = \left(R_{\frac{\pi}{2}}, \begin{pmatrix} 0 \\ 0 \end{pmatrix}\right), \qquad g_2 = \left(R_{\frac{\pi}{4}}, \begin{pmatrix} 1/3 \\ 1/2 \end{pmatrix}\right),$$
$$g_3 = \left(R_{\frac{3\pi}{4}}, \begin{pmatrix} 2/3 \\ -1/2 \end{pmatrix}\right), \qquad g_4 = \left(R_{\pi}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}\right).$$

We evaluate the corresponding Bézier curves on both Lie groups using the Code 2.
The resulting Bézier curves on both Lie groups are shown in Figure 1. One can see that the product Lie group works the same as if one would have done that separately component-wise, while the Bézier curve on $\mathrm{SE}(2)$ behaves differently. A main feature here is, that due to the generic implementation on arbitrary Lie groups, we can easily switch between different Lie groups, where the data is still valid.

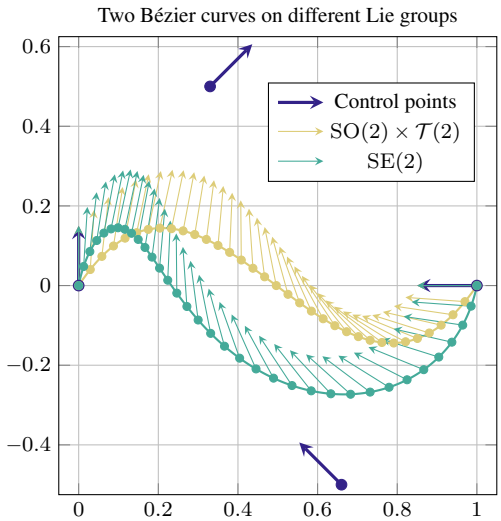


Fig. 1: Bézier curves on the Lie groups $\mathrm{SE}(2)$ (teal) and $\mathrm{SO}(2) \times \mathbb{R}^2$ (sand) based on the same control points $g_1, \ldots, g_4$ (blue).

5.2 Simulating a charged rod in a static electric field

Physical systems can often be described by ordinary differential equations on Lie groups. For example, the state of a charged metal rod moving on a plane can be specified by its position, orientation, linear velocity, and angular velocity. Such state naturally belongs to a Lie group: a direct product of $\mathcal{G}_1 = \mathrm{SO}(2) \times \mathbb{R}^2$ and its Lie algebra, $\mathfrak{g}_1$, denoted $\mathcal{G} = \mathcal{G}_1 \times \mathfrak{g}_1$. The ordinary differential equation governing the dynamics of the system can be derived from the electrostatic approximation, in particular the Coulomb's law. Electric point charges $q_i, i \in 1, \ldots, N$ at positions R_i generate a static electric field $E(p)$ at every point p in the plane. The force and the moment of force affecting the rod of length L, mass M, moment of inertia I_M linear charge density λ, with the center at position p, orientation θ, and where u is the direction along the rod can be expressed as the following two second order ODEs

$$M\ddot{p} = \lambda \int_{-L/2}^{L/2} E(p + su)ds,$$
$$I_M\ddot{\theta} = \int_{-L/2}^{L/2} \lambda s u_\perp \cdot E(p + su)ds,$$

where $u_\perp$ is the vector perpendicular to u. The pair $g = (p, \theta)$ represents a point on $\mathcal{G}_1$, while $\dot{g} = (\dot{p}, \dot{\theta})$ is an element of the Lie algebra $\mathfrak{g}_1$. This second order ODE on $\mathcal{G}_1$ can be converted to a first order ODE on $\mathcal{G} = \mathcal{G}_1 \times \mathfrak{g}_1$ by introducing $(\dot{p}, \dot{\theta})$ as additional state variables.
The overall ODE has the form

$$\frac{d}{dt}\begin{pmatrix} g \\ \dot{g} \end{pmatrix} = \begin{pmatrix} \dot{g} \\ f(g, \dot{g}, t) \end{pmatrix},$$

where $g \in \mathcal{G}_1$ and $f(g, \dot{g}, t)$ is derived from the above equations. We further require initial conditions for both g and $\dot{g}$ at time $t_0 = 0$. Then this can be solved using Lie group methods, such as the Runge-Kutta-Munthe-Kaas (RKMK) variant [15] of Heun's method [13, App. A.1].
Code 3 shows functions that calculate the right hand side of the ODE, where `E_field` is a function that calculates electric field strength from the point charges at the given point. The function

Code 3: Dynamics of a rod affected by point charges.

```
function force_and_torque(
        sp::SystemParameters, Fθ, r
    )
    u, u⊥ =eachrow(Fθ)
    F, τ =zeros(2), 0.0
    for s in range(-sp.L/2, sp.L/2; length=Nq)
        E = E_field(sp, r .+ s .* u)
        F .+= sp.λ *E * (sp.L/(Nq-1))
        τ +=sp.λ *s * dot(u⊥, E) *(sp.L/(Nq-1))
    end
    return F, τ
end
# f((p,θ), (dp,dθ), t) where (p,θ) and (dp,dθ) are
# passed as the first argument, system parameters
# as the second one and time as the last one
function dynamics(state, sp::SystemParameters, t)
    p, dp = state.x
    F, tau = force_and_torque(sp, p.x...)
    a = F ./ sp.M
    c = [tau / sp.Icm, a...]
    ddx = hat(sp.lie_algebra, c, ArrayPartition)
    return ArrayPartition(dp, ddx)
end
```

Code 4: Step calculation for the RKMK Heun method.

```
function integrator_step!(
        A::GroupAction, sp::SystemParameters,
        y, f, t, dt
) # The dynamics function is passed as f
    Ie = Identity(A.group)
    F1 = (dt / 2) * f(y, sp, t)
    tmp = zero_vector(A.manifold, y)
    diff_group_apply!(A, tmp, Ie, y, F1)
    y2 = exp(A.manifold, y, tmp)
    F2 = dt * f(y2, sp, t + dt/2)
    diff_group_apply!(A, tmp, Ie, y, F2)
    exp!(base_manifold(A), y, y, tmp)
end
```

`force_and_torque` calculates the force and torque affecting the rod, while `dynamics` converts them to the Lie algebra element can be used by the solver.
The RKMK Heun method is a second order method that can solve ordinary differential equations defined on Lie groups. A single step of an RKMK-type Heun solver is implemented in Code 4. For our special case the `manifold` is equal to the `group`, namely $\mathcal{G}$, and the group action is the left group operation.
The main advantage of using a Lie group solver for this problem is that we do not have to consider parametrization of the Lie group. The presented approach can be easily generalized to three spatial dimensions, where parametrization of rotations on $\mathrm{SO}(3)$ would be even more challenging numerically.
The presented method was applied to a case with two static charges: $1 \times 10^{-6}\,\mathrm{C}$ at position $(-1\,\mathrm{m},\ 0\,\mathrm{m})$ and $-1 \times 10^{-6}\,\mathrm{C}$ at position $(1\,\mathrm{m},\ 0\,\mathrm{m})$. A rod of length $0.67\,\mathrm{m}$ with charge density $1 \times 10^{-6}\,\mathrm{C/m}$ and mass $1\,\mathrm{kg}$ started from three different initial states:

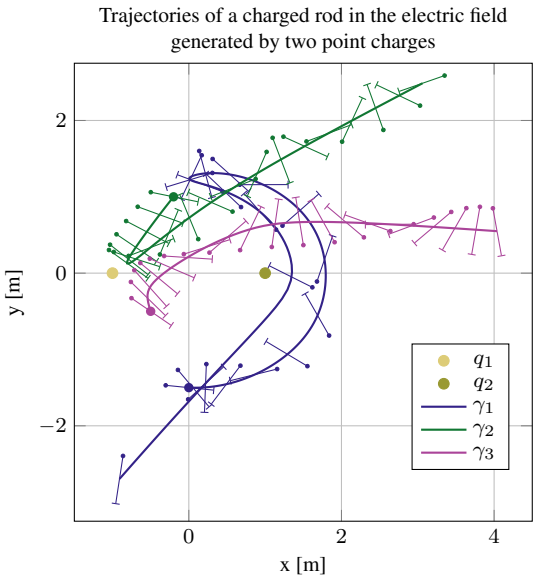


Fig. 2: Three solution curves γ_1 (blue), γ_2 (green), and γ_3 (purple) of a Lie group Heun method on the product of Lie groups $\mathrm{SO}(2) \times \mathbb{R}^2$ modelling the movement within an electric field with two point charges located at q_1 (sand) and q_2 (olive). The initial start points of the rods are indicated by circles, the rods themselves have two different end markers to indicate their orientation.

(1) position $(0\,\mathrm{m},\ -1.5\,\mathrm{m})$, angle $0.1\,\mathrm{rad}$, velocity $(0\,\mathrm{m/s},\ 0\,\mathrm{m/s})$, angular velocity $0\,\mathrm{rad/s}$,
(2) position $(-0.2\,\mathrm{m},\ 1\,\mathrm{m})$, angle $0.2\,\mathrm{rad}$, velocity $(-0.12\,\mathrm{m/s},\ -0.15\,\mathrm{m/s})$, angular velocity $0\,\mathrm{rad/s}$,
(3) position $(-0.5\,\mathrm{m},\ -0.5\,\mathrm{m})$, angle $0.6\,\mathrm{rad}$, velocity $(-0.04\,\mathrm{m/s},\ 0.12\,\mathrm{m/s})$, angular velocity $0\,\mathrm{rad/s}$.

The trajectories obtained in a simulation with time step $0.001\,\mathrm{s}$ are presented in Figure 2. Simulation times between $30\,\mathrm{s}$ and $135\,\mathrm{s}$ were chosen to have the rod remain relatively close to point charges. Position and orientation of the rod is indicated at 20 equally spaced times, along with a densely sampled trajectory of the center of the rod.
For more advanced methods of solving differential equations on Manifolds and Lie groups in Julia, see `ManifoldDiffEq.jl`.[11]

6. References

[1] Seth Axen, Mateusz Baran, Ronny Bergmann, Yueh-Hua Tu, and Olivier Verdier. LieGroups.jl, 2025. doi:10.5281/zenodo.15343362.

[2] Seth D. Axen, Mateusz Baran, Ronny Bergmann, and Krzysztof Rzecki. Manifolds.jl: An extensible julia framework for data analysis on manifolds. *ACM Transactions on Mathematical Software*, 49(4), dec 2023. doi:10.1145/3618296.

[3] Ronny Bergmann. Manopt.jl: Optimization on manifolds in Julia. *Journal of Open Source Software*, 7(70):3866, 2022. doi:10.21105/joss.03866.

[4] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017. doi:10.1137/141000671.

[11] see juliamanifolds.github.io/ManifoldDiffEq.jl/stable/.

[5] Nicolas Boumal. *An Introduction to Optimization on Smooth Manifolds*. Cambridge University Press, first edition, March 2023. doi:10.1017/9781009166164.

[6] Gregory S. Chirikjian. *Stochastic Models, Information Theory, and Lie Groups, Volume 1: Classical Results and Geometric Methods*. 2009. doi:10.1007/978-0-8176-4803-9.

[7] Gregory S. Chirikjian. *Stochastic Models, Information Theory, and Lie Groups, Volume 2: Analytic Methods and Modern Applications*. 2012. doi:10.1007/978-0-8176-4944-9.

[8] Jérémie Deray and Joan Solà. Manif: A micro Lie theory library for state estimation in robotics applications. *Journal of Open Source Software*, 5(46):1371, 2020. doi:10.21105/joss.01371.

[9] Manfredo Perdigão do Carmo. *Riemannian Geometry*. Mathematics: Theory & Applications. Birkhäuser Boston, Inc., 1992.

[10] Kenth Engø, Arne Marthinsen, and Hans Z. Munthe-Kaas. Diffman: An object-oriented matlab toolbox for solving differential equations on manifolds. *Applied Numerical Mathematics*, 39(3–4):323–347, 2001. doi:10.1016/s0168-9274(00)00042-8.

[11] Alessandro Fornasier, Yixiao Ge, Pieter van Goor, Robert Mahony, and Stephan Weiss. Equivariant symmetries for inertial navigation systems. *Automatica*, 181:112495, 2025. doi:10.1016/j.automatica.2025.112495.

[12] Joachim Hilgert and Karl-Hermann Neeb. *Structure and Geometry of Lie Groups*. Springer Monographs in Mathematics, 2012. doi:10.1007/978-0-387-84794-8.

[13] Arieh Iserles, Hans Z. Munthe-Kaas, Syvert P. Nørsett, and Antonella Zanna. Lie-group methods. *Acta Numerica*, 9:215–365, 2000. doi:10.1017/s0962492900002154.

[14] Nina Miolane, Nicolas Guigui, Alice Le Brigant, Johan Mathe, Benjamin Hou, Yann Thanwerdas, Stefan Heyder, Olivier Peltre, Niklas Koep, Hadi Zaatiti, Hatem Hajri, Yann Cabanes, Thomas Gerald, Paul Chauchat, Christian Shewmake, Daniel Brooks, Bernhard Kainz, Claire Donnat, Susan Holmes, and Xavier Pennec. Geomstats: A python package for riemannian geometry in machine learning. *Journal of Machine Learning Research*, 21(223):1–9, 2020. url: http://jmlr.org/papers/v21/19-027.html.

[15] Hans Munthe-Kaas. Runge-Kutta methods on Lie groups. *BIT Numerical Mathematics*, 38(1):92–111, 1998. doi:10.1007/bf02510919.

[16] Hauke Strasdat. Sophus: C++ implementation of lie groups using eigen, 2024. avaiable on GitHub: https://github.com/strasdat/Sophus.

[17] Brent Yi, Michelle A. Lee, Alina Kloss, Roberto Martin-Martin, and Jeannette Bohg. Differentiable factor graph optimization for learning smoothers. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, page 1339?1345, 2021. doi:10.1109/iros51168.2021.9636300.