

# Groups and smooth geometry using LieGroups.jl

Ronny Bergmann<sup>1</sup> and Mateusz Baran<sup>2</sup>

<sup>1</sup>Department of Mathematical Sciences, Norwegian University of Science and Technology, Trondheim, Norway

<sup>2</sup>AGH University of Krakow, 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 existing Lie groups, their algebra and corresponding group actions.

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

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

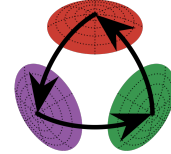
## 1. Introduction

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  $SE(3)$ , consisting of all translations and rotations. A subset of these is the space of rotations, given by the special orthogonal group  $SO(3)$ , or more generally  $SO(n)$  in  $n$ -dimensional space.

These spaces 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 [5, 6], but are also interesting from their mathematical viewpoint [8] and their numerical aspects, for example when solving differential equations on Lie groups [9]. The package **LieGroups.jl**<sup>1</sup> 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 as of version 0.1.7. After introducing the necessary mathematical background and notation in Section 2, we present the interface in Section 3, split into the Lie group, the Lie algebra, and group actions. In Section 4, we present an overview of currently implemented Lie groups. Finally, in Section 5, we demonstrate how to get started and use LieGroups.jl.

<sup>1</sup>Available at [juliamanifolds.github.io/LieGroups.jl/stable/](https://juliamanifolds.github.io/LieGroups.jl/stable/), see also the zenodo archive [1].



Logo of LieGroups.jl.

## 2. Mathematical Background

The following notation and definitions follow the text book [8], for more details on Riemannian manifolds, see also [7].

### 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. A smooth manifold  $\mathcal{M}$  is a topological space that is locally isomorphic to an Euclidean space  $\mathbb{R}^n$  for some  $n \in \mathbb{N}$ , but globally may have a different topology. We call  $n$  the dimension of the manifold  $\mathcal{M}$ , denoted by  $\dim(\mathcal{M}) = n$ . As an example, take the 2-dimensional sphere

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

which locally looks like  $\mathbb{R}^2$ , think of charts in an atlas, but globally it is not. 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 curve can “pass through” a point. Formally it is set the equivalence classes of derivatives of smooth curves. Each tangent space  $T_p\mathcal{M}$  is a  $n$ -dimensional vector space and we call the disjoint union of all tangent spaces

$$T\mathcal{M} = \bigcup_{p \in \mathcal{M}} T_p\mathcal{M}$$

the *tangent bundle* of  $\mathcal{M}$ .

As a group operation  $\cdot : \mathcal{G} \times \mathcal{G} \rightarrow \mathcal{G}$  has to satisfy 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  $\iota : \mathcal{G} \rightarrow \mathcal{G}, g \mapsto g^{-1}$  have to be smooth maps. As an example, consider the special orthogonal group  $SO(n)$ , consisting of all  $n \times n$  orthogonal matrices, with determinant 1, that is for  $p \in SO(n)$ , we have  $p^T p = I$  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

<sup>2</sup>Note that the identification is not continuous.

identity matrix  $I$  (or the angle  $\alpha = 0$ ) and the inverse of a rotation matrix is given by its transpose (or an angle  $-\alpha$ ).

## 2.2 Tangent spaces and 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* 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  $g, h \in \mathcal{G}$ . Then, using the differential (or push forward)  $D\lambda_g(h): T_h\mathcal{G} \rightarrow T_{g \cdot h}\mathcal{G}$ <sup>3</sup>, we can generate a so-called *left-invariant vector field*  $\mathcal{X}(g) := D\lambda_g(e)[X]$  which is uniquely determined by the choice of  $X \in \mathfrak{g}$ .<sup>4</sup> 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 constraint vector field  $\mathcal{X}(g) = X$  for all  $g \in \mathbb{R}^n$ .

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

An important tool to “move around” on the Lie group is the exponential map. The (Lie group) exponential function  $\exp: \mathfrak{g} \rightarrow \mathcal{G}$  maps elements from the Lie algebra to the Lie group, and is formally defined [8, Def. 9.2.2] by evaluating the unique curve  $\gamma: \mathbb{R} \rightarrow \mathcal{G}$  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 map is given by  $\exp(X) = X$ ,  $X \in \mathbb{R}^n$  and for  $\text{SO}(n)$  it is given by the matrix exponential  $\exp(X) = e^X$  for all  $X \in \mathfrak{so}(n)$ . Additionally for the unit circle in the complex plane, using the group operation of multiplication, the exponential map is given by the complex exponential  $\exp(X) = e^{iX}$  for all  $X \in \mathbb{R}$ .

Concerning a metric on the tangent spaces, manifolds are turned into Riemannian manifolds when they are equipped with a Riemannian metric  $\langle \cdot, \cdot \rangle_p: T_p\mathcal{M} \times T_p\mathcal{M} \rightarrow \mathbb{R}$  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

<sup>3</sup>MB: TODO: does the reader know that a differential is? (RB) Sure I can expand on that, I wrote a sentence more further below to get from differential to Jacobian, so I can also add a sentence there as well.

<sup>4</sup>Analogously, one can use the right multiplication  $\rho_g(h) = h \cdot g$  and its differential  $D\rho_g(h)$  to define right-invariant vector fields. LieGroups.jl uses left-invariant vector fields as default.

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  $\langle X, Y \rangle = X^T Y$  for  $X, Y \in \mathbb{R}^n$  for  $(\mathbb{R}^n, +)$  and the Frobenius inner product  $\langle X, Y \rangle = \text{trace}(X^T Y)$ ,  $X, Y \in \mathfrak{so}(n)$  can be used for  $\text{SO}(n)$ .

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  $\sigma: \mathcal{G} \times \mathcal{M} \rightarrow \mathcal{M}$  such that for all  $g, h \in \mathcal{G}$  and  $p \in \mathcal{M}$  it holds that  $\sigma(e, p) = p$  and  $\sigma(g, \sigma(h, p)) = \sigma(g \cdot h, p)$ .<sup>5</sup> 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  $\text{SO}(3)$  acting on points on Euclidean space  $\mathbb{R}^3$  “moving” them somewhere by rotating them around the origin. We obtain the group action  $\sigma: \text{SO}(3) \times \mathbb{R}^3 \rightarrow \mathbb{R}^3$ ,  $\sigma(R, x) := Rx$ .

The same action can also be applied to points from the sphere  $\mathbb{S}^2$ , resulting in a similar group action  $\sigma: \text{SO}(3) \times \mathbb{S}^2 \rightarrow \mathbb{S}^2$ ,  $\sigma(R, p) := Rp$ .

## 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  $\mathfrak{g}$  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.

### 3.1 Lie groups

The main type for Lie Groups is the `LieGroup{F, O, M} <: AbstractLieGroup`, which contains a manifold `M <: AbstractManifold{F}` as well as the group operation `O <: AbstractGroupOperation`, where `F` is

<sup>5</sup>This is the convention for left actions. Alternatively, right actions fulfil  $\sigma(g, \sigma(h, p)) = \sigma(h \cdot g, p)$ .

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  $\mathbb{H}$ .

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 interim 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.<sup>6</sup> 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, also different manifolds can be used. 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)`, `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

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

Group operation $\cdot$	comment/code
<code>AdditionGroupOperation</code>	falls back to $+$
<code>AbelianMultiplicationGroupOperation</code>	falls back to $*$
<code>MatrixMultiplicationGroupOperation</code>	falls back to $*$
<code>PowerGroupOperation{Op}</code>	<code>PowerLieGroup</code>
<code>ProductGroupOperation{Ops}</code>	product Lie groups
<code>SemidirectProductGroupOperation</code>	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 function*  $\exp_g: \mathfrak{g} \rightarrow \mathcal{G}$  defined by  $\exp_g(X) := g \cdot \exp(X)$ . In `LieGroups.jl`, the exponential map is implemented as `exp(G, g, X)`. As a word of carefulness, note that on the underlying manifold, there is a further exponential map, the Riemannian 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:  $\mathcal{G} \rightarrow \mathfrak{g}$`  and logarithmic

<sup>6</sup>`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.

mic map  $\log_g : \mathcal{G} \rightarrow \mathfrak{g}$  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`<sup>7</sup>. 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.<sup>8</sup> Keep in mind, that the Lie algebra is a vector space, so that addition and scalar multiplication 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  $[\cdot, \cdot] : \mathfrak{g} \times \mathfrak{g} \rightarrow \mathfrak{g}$ , which is available as `lie_bracket(g, X, Y)`. For more details on the Lie bracket see [8, 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<sup>9</sup> 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  $c \in \mathbb{R}^n$ , we obtain the 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 `DefaultLieAlgebraOrthogonalBasis()`. When a Lie group has different representations of points and tangent vectors, these

<sup>7</sup>see [juliamanifolds.github.io/ManifoldsBase.jl/stable/retractions/](https://juliamanifolds.github.io/ManifoldsBase.jl/stable/retractions/).

<sup>8</sup>The optional abstract supertype `AbstractLieAlgebraTangentVector`, which is a `AbstractTangentVector`, is provided to also distinguish different representations here.

<sup>9</sup>see the documentation of `ManifoldsBase.TangentSpace`

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

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

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 the representation of a tangent vector  $X \in \mathfrak{g}$  to be represented in the tangent space at  $g \in \mathcal{G}$  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 Euclidean space, the terms differential and Jacobian are often used interchangeably. Within differential geometry, the differential (or push forward) of a smooth map  $f : \mathcal{M} \rightarrow \mathcal{N}$  between two manifolds is a map between the corresponding tangent spaces. We denote it by  $Df(p) : T_p\mathcal{M} \rightarrow T_{f(p)}\mathcal{N}$ , where  $p \in \mathcal{M}$ . The differential is a linear map between the tangent spaces. However, in the coordinate-free representation of tangent vectors, there is no matrix representation of this linear map. 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 ?? are implemented as a struct `GroupAction{G, M, A} <: AbstractGroupAction` containing the Lie group  $G <: AbstractLieGroup$ , the manifold  $M$ , and a group action type  $A$  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  $D\sigma(g, p)[X]$  for a tangent vector  $X \in T_p\mathcal{M}$  with respect to the manifold argument as well as



Table 3. : Implemented Lie groups in LieGroups.jl version 0.1.7.

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

`diff_group_apply(a::GroupAction, g, p, Y)` to compute  $D\sigma(g, p)[Y]$  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`.

## 4. Implemented Lie groups

Table 3 summarizes the currently implemented Lie groups in LieGroups.jl 0.1.7.

### 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  $\mathbf{G}, \mathbf{H}$ , their product Lie group can be constructed via  $\mathbf{G} \times \mathbf{H}$ . The same applies for more than two groups, i.e. you can construct arbitrary long product Lie groups  $\mathbf{G1} \times \mathbf{G2} \times \mathbf{G3} \times \dots$ .

As for the representation of points and tangent vectors, the default representation requires to load `RecursiveArrayTools.jl`<sup>10</sup>.

<sup>10</sup>see [docs.sciml.ai/RecursiveArrayTools/stable/](https://docs.sciml.ai/RecursiveArrayTools/stable/)

If you want to construct nested Products of Lie groups, e.g.  $(\mathcal{G}_1 \times \mathcal{G}_2) \times \mathcal{G}_3$ , the binary operator would not work, since it automatically flattens the input; use `ProductLieGroup(ProductLieGroup(G1, G2), G3)` can be used instead.

*Power Lie groups.* Though mathematically the case of a product Lie Group, where  $G = H$  is also (just) a product, computationally we can benefit from the special structure. Therefore the `Powermanifold(G, n1, n2, ...)` is provided to construct the  $n_1 \times n_2 \times \dots$ -fold Cartesian product of the Lie group  $G$  with itself. As a shortcut, you can use  $\mathcal{G}^n$  to construct the  $n$ -fold Cartesian product of the Lie group  $G$  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 represent points and tangent vectors as a vector of points. 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  $\sigma: \mathcal{G} \times \mathcal{H} \rightarrow \mathcal{H}$ , 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  $\mathbf{G} \ltimes \mathbf{H}$  is available. Similarly, the *right semidirect product Lie Group*  $\mathcal{H} \rtimes \mathcal{G}$  is defined analogously with the positions of the groups 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  $\mathbf{H} \rtimes \mathbf{G}$  then as well.

### 4.2 A Lie group for Numerical verification

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,  $\mathbf{G2} = \text{ValidationLieGroup}(\mathbf{G})$ . It provides additional functionality to check the correctness of computations involving the Lie group. Most prominently, all input and output of a function is 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 concept from `ManifoldsBase.jl`, the `ValidationManifold`. While by default these checks do result in errors, this can be changed setting the `error=` keyword 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  $[0, 2\pi)$ , 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)`, respec-

tively. 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  $\mathcal{G} = \text{SE}(n)$ , 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  $R \in \text{SO}(n)$  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 first, matrix representation work 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.

**TODO: Check whether there are details for other Lie groups that should be mentioned here.**

## 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 Bézier curve on a Lie group

In the first example we illustrate how to construct and plot a Bézier curve on a Lie group, here the special Euclidean group  $\text{SE}(2)$ .

We first can generalize the de-Casteljau algorithm for Bézier 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.

```
1 using LieGroups, ManifoldsBase, RecursiveArrayTools
2 using CairoMakie
3 """
4     deCasteljauLieGroup(G, cp, t)
5
6 Evaluate the Bézier curve at parameter `t` given
7 a vector of control points `cp` on the Lie group `G`.
8 """
9 function deCasteljauLieGroup(
10     G::AbstractLieGroup, cp::Vector{P}, t::Real
11 ) where {P}
12     n = length(cp)
13     points = [copy(G, g) for g in cp]
14     for r in 1:n-1
15         for i in 1:n-r
16             X = log(G, points[i], points[i+1])
17             exp!(G, points[i], points[i], t * X)
18         end
19     end
20     return points[1]
21 end
```

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  $\text{SE}(2)$  with the corresponding Bézier curve on the (direct) product

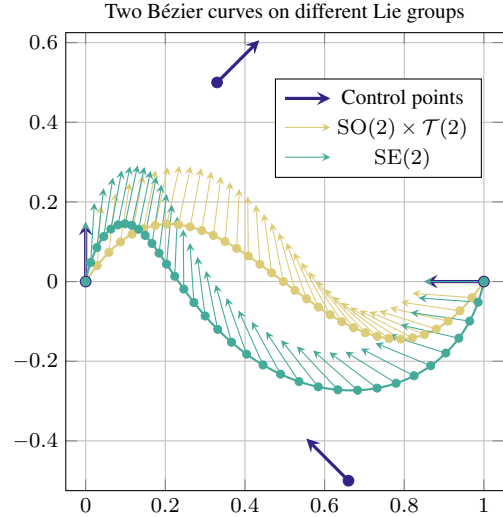


Fig. 1: Bézier curves on the Lie groups  $\text{SE}(2)$  (teal) and  $\text{SO}(2) \times \mathbb{T}(2)$  (sand) based on the same control points  $g_1, \dots, g_4$  (blue).

Lie group  $\text{SO}(2) \times \mathcal{T}(2)$ , where  $\mathcal{T}(2) = (\mathbb{R}^2, +)$  is the TranslationGroup(2). Denote by  $R_\alpha$  the rotation matrix in  $\text{SO}(2)$  for an angle  $\alpha \in [0, 2\pi)$ . We consider the control points

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

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

We evaluate the corresponding Bézier curves on both Lie groups using the Code 2.

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

```
1 S02 = SpecialOrthogonalGroup(2)
2 R2 = TranslationGroup(2)
3 S02xR2 = S02 × R2
4 ts = range(0, 1, length=35)
5 q1 = [
6     deCasteljauLieGroup(S02xR2, cp, t) for t in ts
7 ]
8 SE2 = SpecialEuclideanGroup(2)
9 q2 = [deCasteljauLieGroup(SE2, cp, t) for t in ts]
```

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  $\text{SE}(2)$  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.

### 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  $G_1 = \text{SO}(2) \times \mathbb{R}^2$  and its Lie algebra,  $\mathfrak{g}_1$ , denoted  $G = G_1 \times \mathfrak{g}_1$ . 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  $q_i, i \in 1, \dots, N$  at positions  $R_i$  generate a static electric field  $E(p)$  at point  $p$ . The force and the moment of force affecting the rod of length  $L$ , mass  $M$ , moment of inertia  $I_M$  linear charge density  $\lambda$ , with the center at position  $p$ , orientation  $\theta$  and where  $u$  is the direction along the rod can be expressed as

$$M\ddot{p} = \lambda \int_{-L/2}^{L/2} E(p + su) ds, \quad (1)$$

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

where  $u_{\perp}$  is the vector perpendicular to  $u$ . This can be understood as an ODE on  $G$  and solved using Lie groups methods, such as the RKMK variant of the Heun method [?].

Listing 3 shows functions that calculate the right hand side of the ODE. `E_field` is a function that calculates electric field strength from the point charges at the given point. The function `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.

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

```
1 function force_and_torque(sp::SystemParameters, Fθ, r)
2   u, u⊥ = eachrow(Fθ)
3   F, τ = zeros(2), 0.0
4   for s in range(-sp.L/2, sp.L/2; length=Nq)
5     E = E_field(sp, r .+ s .* u)
6     F .+= sp.λ * E * (sp.L/(Nq-1))
7     τ += sp.λ * s * dot(u⊥, E) * (sp.L/(Nq-1))
8   end
9   return F, τ
10 end
11
12 function dynamics(state, sp::SystemParameters, t)
13   p, dp = state.x
14   F, tau = force_and_torque(sp, p.x...)
15   a = F ./ sp.M
16   coords = [tau / sp.Icm, a...]
17   ddx = hat(se2, coords, ArrayPartition)
18   return ArrayPartition(dp, ddx)
19 end
```

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 4. 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 would be even more challenging numerically.

Code 4: Step calculation for the RKMK Heun method.

```
1 function integrator_step!(A::GroupAction,
2   sp::SystemParameters, y, f, t, dt)
3   Ie = Identity(A.group)
4   F1 = (dt / 2) * f(y, sp, t)
5   tmp = zero_vector(A.manifold, y)
6   diff_group_apply!(A, tmp, Ie, y, F1)
7   y2 = exp(A.manifold, y, tmp)
8   F2 = dt * f(y2, sp, t + dt/2)
9   diff_group_apply!(A, tmp, Ie, y, F2)
10  exp!(base_manifold(A), y, y, tmp)
11 end
```

Trajectories of a charged rod between two point charges

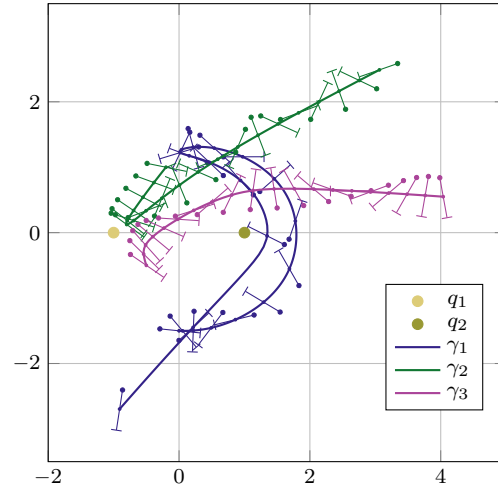


Fig. 2: Three solution curves  $\gamma_1$  (blue),  $\gamma_2$  (green), and  $\gamma_3$  (purple) of a Lie group Heun method on the product of Lie groups  $SO(2) \times \mathbb{R}^2$  modelling the movement within an electric field with two point charges located at  $q_1$  (sand) and  $q_2$  (olive).

The presented method was applied to a case with two static charges:  $1 \times 10^{-6} \text{ C}$  at position  $(-1 \text{ m}, 0 \text{ m})$  and  $-1 \times 10^{-6} \text{ C}$  at position  $(1 \text{ m}, 0 \text{ m})$ . A rod of length  $0.67 \text{ m}$  with charge density  $1 \times 10^{-6} \text{ C/m}$  and mass  $1 \text{ kg}$  started from three different initial positions. The trajectories obtained in a simulation with time step  $0.001 \text{ s}$  are presented in Figure 2. Simulation times between  $30 \text{ s}$  and  $135 \text{ s}$  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 Lie groups in Julia, see `ManifoldDiffEq.jl`.

## 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.
- [5] 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.
- [6] 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.
- [7] Manfredo Perdigão do Carmo. *Riemannian Geometry*. Mathematics: Theory & Applications. Birkhäuser Boston, Inc., 1992.

- [8] 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](https://doi.org/10.1007/978-0-387-84794-8).
- [9] Hans Munthe-Kaas. Runge-Kutta methods on Lie groups. *BIT Numerical Mathematics*, 38(1):92–111, 1998. doi:[10.1007/bf02510919](https://doi.org/10.1007/bf02510919).