geoopt Documentation

Release 0.5.0

Max Kochurov

Jun 29, 2022

Contents

CHAPTER 1

geoopt

Manifold aware `pytorch.optim`.

Unofficial implementation for "Riemannian Adaptive Optimization Methods" ICLR2019 and more.

1.1 Installation

Make sure you have pytorch>=1.10.2 installed

There are two ways to install geoopt:

1. GitHub (preferred so far) due to active development

```
pip install git+https://github.com/geoopt/geoopt.git
```

2. pypi (this might be significantly behind master branch)

```
pip install geoopt
```

The preferred way to install geoopt will change once stable project stage is achieved. Now, pypi is behind master as we actively develop and implement new features.

1.1.1 PyTorch Support

Geoopt officially supports **2 latest stable versions** of pytorch upstream or the latest major release.

1.2 What is done so far

Work is in progress but you can already use this. Note that API might change in future releases.

1.2.1 Tensors

- `geoopt.ManifoldTensor` - just as torch.Tensor with additional `manifold` keyword argument.
- `geoopt.ManifoldParameter` - same as above, recognized in `torch.nn.Module.parameters` as correctly subclassed.

All above containers have special methods to work with them as with points on a certain manifold

- `.proj_()` - inplace projection on the manifold.
- `.proju(u)` - project vector `u` on the tangent space. You need to project all vectors for all methods below.
- `.egrad2rgrad(u)` - project gradient `u` on Riemannian manifold
- `.inner(u, v=None)` - inner product at this point for two **tangent** vectors at this point. The passed vectors are not projected, they are assumed to be already projected.
- `.retr(u)` - retraction map following vector `u`
- `.expmap(u)` - exponential map following vector `u` (if expmap is not available in closed form, best approximation is used)
- `.transp(v, u)` - transport vector `v` with direction `u`
- `.retr_transp(v, u)` - transport `self`, vector `v` (and possibly more vectors) with direction `u` (returns are plain tensors)

1.2.2 Manifolds

- `geoopt.Euclidean` - unconstrained manifold in `R` with Euclidean metric
- `geoopt.Stiefel` - Stiefel manifold on matrices `A in R^{n x p} :  A^t A=I, n >= p`
- `geoopt.Sphere` - Sphere manifold `||x||=1`
- `geoopt.BirkhoffPolytope` - manifold of Doubly Stochastic matrices
- `geoopt.Stereographic` - Constant curvature stereographic projection model
- `geoopt.SphereProjection` - Sphere stereographic projection model
- `geoopt.PoincareBall` - Poincare ball model
- `geoopt.Lorentz` - Hyperboloid model
- `geoopt.ProductManifold` - Product manifold constructor
- `geoopt.Scaled` - Scaled version of the manifold. Similar to Learning Mixed-Curvature Representations in Product Spaces if combined with `ProductManifold`
- `geoopt.SymmetricPositiveDefinite` - SPD matrix manifold
- `geoopt.UpperHalf` - Siegel Upper half manifold. Supports Riemannian and Finsler metrics, as in Symmetric Spaces for Graph Embeddings: A Finsler-Riemannian Approach.
- `geoopt.BoundedDomain` - Siegel Bounded domain manifold. Supports Riemannian and Finsler metrics.

All manifolds implement methods necessary to manipulate tensors on manifolds and tangent vectors to be used in general purpose. See more in documentation.

1.2.3 Optimizers

- `geoopt.optim.RiemannianSGD` - a subclass of `torch.optim.SGD` with the same API
- `geoopt.optim.RiemannianAdam` - a subclass of `torch.optim.Adam`

1.2.4 Samplers

- `geoopt.samplers.RSGLD` - Riemannian Stochastic Gradient Langevin Dynamics
- `geoopt.samplers.RHMC` - Riemannian Hamiltonian Monte-Carlo
- `geoopt.samplers.SGRHMC` - Stochastic Gradient Riemannian Hamiltonian Monte-Carlo

1.2.5 Layers

Experimental `geoopt.layers` module allows to embed geoopt into deep learning

1.2.6 Citing Geoopt

If you find this project useful in your research, please kindly add this bibtex entry in references and cite.

```
@misc{geoopt2020kochurov,
    title={Geoopt: Riemannian Optimization in PyTorch},
    author={Max Kochurov and Rasul Karimov and Serge Kozlukov},
    year={2020},
    eprint={2005.02819},
    archivePrefix={arXiv},
    primaryClass={cs.CG}
}
```

1.2.7 Donations

ETH: 0x008319973D4017414FdF5B3beF1369bA78275C6A

CHAPTER 2

API

2.1 Manifolds

All manifolds share same API. Some manifols may have several implementations of retraction operation, every implementation has a corresponding class.

class geoopt.manifolds.**Euclidean**(*ndim=0*)

Simple Euclidean manifold, every coordinate is treated as an independent element.

Parameters **ndim** (*int*) – number of trailing dimensions treated as manifold dimensions. All the operations acting on cuch as inner products, etc will respect the `ndim`.

component_inner(*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor = None*) → torch.Tensor

Inner product for tangent vectors at point x according to components of the manifold.

The result of the function is same as `inner` with `keepdim=True` for all the manifolds except ProductManifold. For this manifold it acts different way computing inner product for each component and then building an output correctly tiling and reshaping the result.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*Optional[torch.Tensor]*) – tangent vector at point x

Returns inner product component wise (broadcasted)

Return type torch.Tensor

Notes

The purpose of this method is better adaptive properties in optimization since ProductManifold will "hide" the structure in public API.

dist (*x: torch.Tensor*, *y: torch.Tensor*, *, *keepdim=False*) → torch.Tensor
Compute distance between 2 points on the manifold that is the shortest path along geodesics.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **y** (*torch.Tensor*) – point on the manifold
- **keepdim** (*bool*) – keep the last dim?

Returns distance between two points

Return type torch.Tensor

dist2 (*x: torch.Tensor*, *y: torch.Tensor*, *, *keepdim=False*) → torch.Tensor
Compute squared distance between 2 points on the manifold that is the shortest path along geodesics.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **y** (*torch.Tensor*) – point on the manifold
- **keepdim** (*bool*) – keep the last dim?

Returns squared distance between two points

Return type torch.Tensor

egrad2rgrad (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Transform gradient computed using autodiff to the correct Riemannian gradient for the point x.

Parameters

- **torch.Tensor** (u) – point on the manifold
- **torch.Tensor** – gradient to be projected

Returns grad vector in the Riemannian manifold

Return type torch.Tensor

expmap (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Perform an exponential map $\operatorname{Exp}_x(u)$.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

extra_repr ()
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

inner (*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor = None*, *, *keepdim=False*) → torch.Tensor
Inner product for tangent vectors at point x.

Parameters

- **x** (*torch.Tensor*) – point on the manifold

- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*Optional[torch.Tensor]*) – tangent vector at point x
- **keepdim** (*bool*) – keep the last dim?

Returns inner product (broadcasted)

Return type torch.Tensor

logmap (*x: torch.Tensor*, *y: torch.Tensor*) → torch.Tensor
Perform an logarithmic map $\operatorname{Log}_x(y)$.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **y** (*torch.Tensor*) – point on the manifold

Returns tangent vector

Return type torch.Tensor

norm (*x: torch.Tensor*, *u: torch.Tensor*, *, *keepdim=False*)
Norm of a tangent vector at point x.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **keepdim** (*bool*) – keep the last dim?

Returns inner product (broadcasted)

Return type torch.Tensor

origin (**size*, *dtype=None*, *device=None*, *seed=42*) → geoopt.tensor.ManifoldTensor
Zero point origin.

Parameters

- **size** (*shape*) – the desired shape
- **device** (*torch.device*) – the desired device
- **dtype** (*torch.dtype*) – the desired dtype
- **seed** (*int*) – ignored

Returns

Return type *ManifoldTensor*

proju (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Project vector u on a tangent space for x, usually is the same as *egrad2rgrad()*.

Parameters

- **torch.Tensor** (u) – point on the manifold
- **torch.Tensor** – vector to be projected

Returns projected vector

Return type torch.Tensor

projx (*x: torch.Tensor*) → torch.Tensor
Project point x on the manifold.

Parameters **torch.Tensor** (x) – point to be projected

Returns projected point

Return type torch.Tensor

random(**size*, *mean=0.0*, *std=1.0*, *device=None*, *dtype=None*) → geoopt.tensor.ManifoldTensor
Create a point on the manifold, measure is induced by Normal distribution.

Parameters

- **size** (*shape*) – the desired shape
- **mean** (*float|tensor*) – mean value for the Normal distribution
- **std** (*float|tensor*) – std value for the Normal distribution
- **device** (*torch.device*) – the desired device
- **dtype** (*torch.dtype*) – the desired dtype

Returns random point on the manifold

Return type *ManifoldTensor*

random_normal(**size*, *mean=0.0*, *std=1.0*, *device=None*, *dtype=None*) → geoopt.tensor.ManifoldTensor
Create a point on the manifold, measure is induced by Normal distribution.

Parameters

- **size** (*shape*) – the desired shape
- **mean** (*float|tensor*) – mean value for the Normal distribution
- **std** (*float|tensor*) – std value for the Normal distribution
- **device** (*torch.device*) – the desired device
- **dtype** (*torch.dtype*) – the desired dtype

Returns random point on the manifold

Return type *ManifoldTensor*

retr(*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Perform a retraction from point x with given direction u.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

transp(*x: torch.Tensor*, *y: torch.Tensor*, *v: torch.Tensor*) → torch.Tensor
Perform vector transport $\mathfrak{T}_{x \to y}(v)$.

Parameters

- **x** (*torch.Tensor*) – start point on the manifold
- **y** (*torch.Tensor*) – target point on the manifold
- **v** (*torch.Tensor*) – tangent vector at point x

Returns transported tensor

Return type torch.Tensor

class geoopt.manifolds.**Stiefel**(***kwargs*)

Manifold induced by the following matrix constraint:

$$X^\top X = I$$
$$X \in \mathrm{R}^{n \times m}$$
$$n \geq m$$

Parameters **canonical** (*bool*) – Use canonical inner product instead of euclidean one (defaults to canonical)

See also:

CanonicalStiefel, *EuclideanStiefel*, *EuclideanStiefelExact*

origin(**size*, *dtype=None*, *device=None*, *seed=42*) → torch.Tensor

Identity matrix point origin.

Parameters

- **size** (*shape*) – the desired shape
- **device** (*torch.device*) – the desired device
- **dtype** (*torch.dtype*) – the desired dtype
- **seed** (*int*) – ignored

Returns

Return type *ManifoldTensor*

projx(*x: torch.Tensor*) → torch.Tensor

Project point x on the manifold.

Parameters **torch.Tensor** (*x*) – point to be projected

Returns projected point

Return type torch.Tensor

random(**size*, *dtype=None*, *device=None*) → torch.Tensor

Naive approach to get random matrix on Stiefel manifold.

A helper function to sample a random point on the Stiefel manifold. The measure is non-uniform for this method, but fast to compute.

Parameters

- **size** (*shape*) – the desired output shape
- **dtype** (*torch.dtype*) – desired dtype
- **device** (*torch.device*) – desired device

Returns random point on Stiefel manifold

Return type *ManifoldTensor*

random_naive(**size*, *dtype=None*, *device=None*) → torch.Tensor

Naive approach to get random matrix on Stiefel manifold.

A helper function to sample a random point on the Stiefel manifold. The measure is non-uniform for this method, but fast to compute.

Parameters

- **size** (*shape*) – the desired output shape
- **dtype** (*torch.dtype*) – desired dtype
- **device** (*torch.device*) – desired device

Returns random point on Stiefel manifold

Return type *ManifoldTensor*

class geoopt.manifolds.**CanonicalStiefel**(***kwargs*)

Stiefel Manifold with Canonical inner product

Manifold induced by the following matrix constraint:

$$X^\top X = I$$
$$X \in \mathrm{R}^{n \times m}$$
$$n \geq m$$

egrad2rgrad(*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor

Project vector u on a tangent space for x, usually is the same as *egrad2rgrad()*.

Parameters

- **torch.Tensor** (u) – point on the manifold
- **torch.Tensor** – vector to be projected

Returns projected vector

Return type torch.Tensor

expmap(*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor

Perform a retraction from point x with given direction u.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

expmap_transp(*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

Perform a retraction + vector transport at once.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported point and vectors

Return type Tuple[torch.Tensor, torch.Tensor]

Notes

Sometimes this is a far more optimal way to preform retraction + vector transport

inner (*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor = None*, *, *keepdim=False*) → torch.Tensor
Inner product for tangent vectors at point x.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*Optional[torch.Tensor]*) – tangent vector at point x
- **keepdim** (*bool*) – keep the last dim?

Returns inner product (broadcasted)

Return type torch.Tensor

proju (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Project vector u on a tangent space for x, usually is the same as *egrad2rgrad()*.

Parameters

- **torch.Tensor** (u) – point on the manifold
- **torch.Tensor** – vector to be projected

Returns projected vector

Return type torch.Tensor

retr (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Perform a retraction from point x with given direction u.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

retr_transp (*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]
Perform a retraction + vector transport at once.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported point and vectors

Return type Tuple[torch.Tensor, torch.Tensor]

Notes

Sometimes this is a far more optimal way to preform retraction + vector transport

transp_follow_expmap (*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*) → torch.Tensor
Perform vector transport following u: $\mathfrak{T}_{x \to \operatorname{retr}(x,u)}(v)$.

This operation is sometimes is much more simpler and can be optimized.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported tensor

Return type torch.Tensor

transp_follow_retr (*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*) → torch.Tensor
Perform vector transport following u: $\mathfrak{T}_{x \to \operatorname{retr}(x,u)}(v)$.

This operation is sometimes is much more simpler and can be optimized.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported tensor

Return type torch.Tensor

class geoopt.manifolds.**EuclideanStiefel**(***kwargs*)
Stiefel Manifold with Euclidean inner product

Manifold induced by the following matrix constraint:

$$X^\top X = I$$
$$X \in \mathrm{R}^{n \times m}$$
$$n \geq m$$

egrad2rgrad (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Project vector u on a tangent space for x, usually is the same as *egrad2rgrad()*.

Parameters

- **torch.Tensor** (u) – point on the manifold
- **torch.Tensor** – vector to be projected

Returns projected vector

Return type torch.Tensor

expmap (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Perform an exponential map $\operatorname{Exp}_x(u)$.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

inner (*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor = None*, **, keepdim=False*) → torch.Tensor
Inner product for tangent vectors at point x.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*Optional[torch.Tensor]*) – tangent vector at point x
- **keepdim** (*bool*) – keep the last dim?

Returns inner product (broadcasted)

Return type torch.Tensor

proju (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Project vector u on a tangent space for x, usually is the same as *egrad2rgrad()*.

Parameters

- **torch.Tensor** (u) – point on the manifold
- **torch.Tensor** – vector to be projected

Returns projected vector

Return type torch.Tensor

retr (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Perform a retraction from point x with given direction u.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

transp (*x: torch.Tensor*, *y: torch.Tensor*, *v: torch.Tensor*) → torch.Tensor
Perform vector transport $\mathfrak{T}_{x \to y}(v)$.

Parameters

- **x** (*torch.Tensor*) – start point on the manifold
- **y** (*torch.Tensor*) – target point on the manifold
- **v** (*torch.Tensor*) – tangent vector at point x

Returns transported tensor

Return type torch.Tensor

class geoopt.manifolds.**EuclideanStiefelExact**(***kwargs*)
Stiefel Manifold with Euclidean inner product

Manifold induced by the following matrix constraint:

$$X^\top X = I$$
$$X \in \mathrm{R}^{n \times m}$$
$$n \geq m$$

Notes

The implementation of retraction is an exact exponential map, this retraction will be used in optimization

extra_repr()
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

retr(*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Perform an exponential map $\operatorname{Exp}_x(u)$.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

retr_transp(*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]
Perform an exponential map and vector transport from point x with given direction u.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported point

Return type torch.Tensor

transp_follow_retr(*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*) → torch.Tensor
Perform vector transport following u: $\mathfrak{T}_{x \to \operatorname{Exp}(x,u)}(v)$.

Here, Exp is the best possible approximation of the true exponential map. There are cases when the exact variant is hard or impossible implement, therefore a fallback, non-exact, implementation is used.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported tensor

Return type torch.Tensor

class geoopt.manifolds.**Sphere**(*intersection: torch.Tensor = None*, *complement: torch.Tensor = None*)
Sphere manifold induced by the following constraint

$$\|x\| = 1$$
$$x \in \sim \backslash \supset \ltimes(U)$$

where U can be parametrized with compliment space or intersection.

Parameters

- **intersection** (*tensor*) – shape (..., dim, K), subspace to intersect with

- **complement** (*tensor*) – shape `(..., dim, K)`, subspace to compliment

See also:

SphereExact

dist (*x: torch.Tensor*, *y: torch.Tensor*, *, *keepdim=False*) → torch.Tensor
Compute distance between 2 points on the manifold that is the shortest path along geodesics.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **y** (*torch.Tensor*) – point on the manifold
- **keepdim** (*bool*) – keep the last dim?

Returns distance between two points

Return type torch.Tensor

egrad2rgrad (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Project vector u on a tangent space for x, usually is the same as *egrad2rgrad()*.

Parameters

- **torch.Tensor** (u) – point on the manifold
- **torch.Tensor** – vector to be projected

Returns projected vector

Return type torch.Tensor

expmap (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Perform an exponential map $\operatorname{Exp}_x(u)$.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

inner (*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor = None*, *, *keepdim=False*) → torch.Tensor
Inner product for tangent vectors at point x.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*Optional[torch.Tensor]*) – tangent vector at point x
- **keepdim** (*bool*) – keep the last dim?

Returns inner product (broadcasted)

Return type torch.Tensor

logmap (*x: torch.Tensor*, *y: torch.Tensor*) → torch.Tensor
Perform an logarithmic map $\operatorname{Log}_x(y)$.

Parameters

- **x** (*torch.Tensor*) – point on the manifold

- **y** (*torch.Tensor*) – point on the manifold

Returns tangent vector

Return type torch.Tensor

proju (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Project vector u on a tangent space for x, usually is the same as *egrad2rgrad()*.

Parameters

- **torch.Tensor** (u) – point on the manifold
- **torch.Tensor** – vector to be projected

Returns projected vector

Return type torch.Tensor

projx (*x: torch.Tensor*) → torch.Tensor
Project point x on the manifold.

Parameters **torch.Tensor** (x) – point to be projected

Returns projected point

Return type torch.Tensor

random (**size*, *dtype=None*, *device=None*) → torch.Tensor
Uniform random measure on Sphere manifold.

Parameters

- **size** (*shape*) – the desired output shape
- **dtype** (*torch.dtype*) – desired dtype
- **device** (*torch.device*) – desired device

Returns random point on Sphere manifold

Return type *ManifoldTensor*

Notes

In case of projector on the manifold, dtype and device are set automatically and shouldn't be provided. If you provide them, they are checked to match the projector device and dtype

random_uniform (**size*, *dtype=None*, *device=None*) → torch.Tensor
Uniform random measure on Sphere manifold.

Parameters

- **size** (*shape*) – the desired output shape
- **dtype** (*torch.dtype*) – desired dtype
- **device** (*torch.device*) – desired device

Returns random point on Sphere manifold

Return type *ManifoldTensor*

Notes

In case of projector on the manifold, dtype and device are set automatically and shouldn't be provided. If you provide them, they are checked to match the projector device and dtype

retr (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Perform a retraction from point x with given direction u.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

transp (*x: torch.Tensor*, *y: torch.Tensor*, *v: torch.Tensor*) → torch.Tensor
Perform vector transport $\mathfrak{T}_{x \to y}(v)$.

Parameters

- **x** (*torch.Tensor*) – start point on the manifold
- **y** (*torch.Tensor*) – target point on the manifold
- **v** (*torch.Tensor*) – tangent vector at point x

Returns transported tensor

Return type torch.Tensor

class geoopt.manifolds.**SphereExact**(*intersection: torch.Tensor = None*, *complement: torch.Tensor = None*)
Sphere manifold induced by the following constraint

$$\|x\| = 1$$
$$x \in \sim \supset \ltimes (U)$$

where U can be parametrized with compliment space or intersection.

Parameters

- **intersection** (*tensor*) – shape `(..., dim, K)`, subspace to intersect with
- **complement** (*tensor*) – shape `(..., dim, K)`, subspace to compliment

See also:

Sphere

Notes

The implementation of retraction is an exact exponential map, this retraction will be used in optimization

extra_repr()
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

retr (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Perform an exponential map $\operatorname{Exp}_x(u)$.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

retr_transp(*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]
Perform an exponential map and vector transport from point x with given direction u.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported point

Return type torch.Tensor

transp_follow_retr(*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*) → torch.Tensor
Perform vector transport following u: $\mathfrak{T}_{x\to\operatorname{Exp}(x,u)}(v)$.

Here, Exp is the best possible approximation of the true exponential map. There are cases when the exact variant is hard or impossible implement, therefore a fallback, non-exact, implementation is used.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported tensor

Return type torch.Tensor

class geoopt.manifolds.**Stereographic**(*k=0.0*, *learnable=False*)
κ-Stereographic model.

Parameters **k** (*float|tensor*) – sectional curvature κ of the manifold - k<0: Poincaré ball (stereographic projection of hyperboloid) - k>0: Stereographic projection of sphere - k=0: Euclidean geometry

Notes

It is extremely recommended to work with this manifold in double precision.

http://andbloch.github.io/K-Stereographic-Model/ or *\kappa-Stereographic Projection model*

References

The functions for the mathematics in gyrovector spaces are taken from the following resources:

[1] Ganea, Octavian, Gary Bécigneul, and Thomas Hofmann. "Hyperbolic neural networks." Advances in neural information processing systems. 2018.

[2] Bachmann, Gregor, Gary Bécigneul, and Octavian-Eugen Ganea. "Constant Curvature Graph Convolutional Networks." arXiv preprint arXiv:1911.05076 (2019).

[3] Skopek, Ondrej, Octavian-Eugen Ganea, and Gary Bécigneul. "Mixed-curvature Variational Autoencoders." arXiv preprint arXiv:1911.08411 (2019).

[4] Ungar, Abraham A. Analytic hyperbolic geometry: Mathematical foundations and applications. World Scientific, 2005.

[5] Albert, Ungar Abraham. Barycentric calculus in Euclidean and hyperbolic geometry: A comparative introduction. World Scientific, 2010.

See also:

StereographicExact, *PoincareBall*, *PoincareBallExact*, *SphereProjection*, *SphereProjectionExact*

dist (*x: torch.Tensor*, *y: torch.Tensor*, ***, *keepdim=False*, *dim=-1*) → torch.Tensor
Compute distance between 2 points on the manifold that is the shortest path along geodesics.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **y** (*torch.Tensor*) – point on the manifold
- **keepdim** (*bool*) – keep the last dim?

Returns distance between two points

Return type torch.Tensor

dist2 (*x: torch.Tensor*, *y: torch.Tensor*, ***, *keepdim=False*, *dim=-1*) → torch.Tensor
Compute squared distance between 2 points on the manifold that is the shortest path along geodesics.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **y** (*torch.Tensor*) – point on the manifold
- **keepdim** (*bool*) – keep the last dim?

Returns squared distance between two points

Return type torch.Tensor

egrad2rgrad (*x: torch.Tensor*, *u: torch.Tensor*, ***, *dim=-1*) → torch.Tensor
Transform gradient computed using autodiff to the correct Riemannian gradient for the point x.

Parameters

- **torch.Tensor** (u) – point on the manifold
- **torch.Tensor** – gradient to be projected

Returns grad vector in the Riemannian manifold

Return type torch.Tensor

expmap (*x: torch.Tensor*, *u: torch.Tensor*, ***, *project=True*, *dim=-1*) → torch.Tensor
Perform an exponential map $\operatorname{Exp}_x(u)$.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

expmap_transp(*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*, *, *dim=-1*, *project=True*) → Tuple[torch.Tensor, torch.Tensor]
Perform an exponential map and vector transport from point x with given direction u.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported point

Return type torch.Tensor

inner(*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor = None*, *, *keepdim=False*, *dim=-1*) → torch.Tensor
Inner product for tangent vectors at point x.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*Optional[torch.Tensor]*) – tangent vector at point x
- **keepdim** (*bool*) – keep the last dim?

Returns inner product (broadcasted)

Return type torch.Tensor

logmap(*x: torch.Tensor*, *y: torch.Tensor*, *, *dim=-1*) → torch.Tensor
Perform an logarithmic map $\operatorname{Log}_x(y)$.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **y** (*torch.Tensor*) – point on the manifold

Returns tangent vector

Return type torch.Tensor

norm(*x: torch.Tensor*, *u: torch.Tensor*, *, *keepdim=False*, *dim=-1*) → torch.Tensor
Norm of a tangent vector at point x.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **keepdim** (*bool*) – keep the last dim?

Returns inner product (broadcasted)

Return type torch.Tensor

origin(**size*, *dtype=None*, *device=None*, *seed=42*) → geoopt.tensor.ManifoldTensor
Zero point origin.

Parameters

- **size** (*shape*) – the desired shape
- **device** (*torch.device*) – the desired device

- **dtype** (*torch.dtype*) – the desired dtype
- **seed** (*int*) – ignored

Returns random point on the manifold

Return type *ManifoldTensor*

proju (*x: torch.Tensor*, *u: torch.Tensor*, *, *dim=-1*) → torch.Tensor
Project vector u on a tangent space for x, usually is the same as *egrad2rgrad()*.

Parameters

- **torch.Tensor** (u) – point on the manifold
- **torch.Tensor** – vector to be projected

Returns projected vector

Return type torch.Tensor

projx (*x: torch.Tensor*, *, *dim=-1*) → torch.Tensor
Project point x on the manifold.

Parameters **torch.Tensor** (x) – point to be projected

Returns projected point

Return type torch.Tensor

random (**size*, *mean=0*, *std=1*, *dtype=None*, *device=None*) → geoopt.tensor.ManifoldTensor
Create a point on the manifold, measure is induced by Normal distribution on the tangent space of zero.

Parameters

- **size** (*shape*) – the desired shape
- **mean** (*float|tensor*) – mean value for the Normal distribution
- **std** (*float|tensor*) – std value for the Normal distribution
- **dtype** (*torch.dtype*) – target dtype for sample, if not None, should match Manifold dtype
- **device** (*torch.device*) – target device for sample, if not None, should match Manifold device

Returns random point on the PoincareBall manifold

Return type *ManifoldTensor*

Notes

The device and dtype will match the device and dtype of the Manifold

random_normal (**size*, *mean=0*, *std=1*, *dtype=None*, *device=None*) → geoopt.tensor.ManifoldTensor
Create a point on the manifold, measure is induced by Normal distribution on the tangent space of zero.

Parameters

- **size** (*shape*) – the desired shape
- **mean** (*float|tensor*) – mean value for the Normal distribution
- **std** (*float|tensor*) – std value for the Normal distribution

- **dtype** (*torch.dtype*) – target dtype for sample, if not None, should match Manifold dtype
- **device** (*torch.device*) – target device for sample, if not None, should match Manifold device

Returns random point on the PoincareBall manifold

Return type *ManifoldTensor*

Notes

The device and dtype will match the device and dtype of the Manifold

retr (*x: torch.Tensor*, *u: torch.Tensor*, *, *dim=-1*) → torch.Tensor
Perform a retraction from point x with given direction u.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

retr_transp (*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*, *, *dim=-1*) → Tuple[torch.Tensor, torch.Tensor]
Perform a retraction + vector transport at once.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported point and vectors

Return type Tuple[torch.Tensor, torch.Tensor]

Notes

Sometimes this is a far more optimal way to preform retraction + vector transport

transp (*x: torch.Tensor*, *y: torch.Tensor*, *v: torch.Tensor*, *, *dim=-1*)
Perform vector transport $\mathfrak{T}_{x \to y}(v)$.

Parameters

- **x** (*torch.Tensor*) – start point on the manifold
- **y** (*torch.Tensor*) – target point on the manifold
- **v** (*torch.Tensor*) – tangent vector at point x

Returns transported tensor

Return type torch.Tensor

transp_follow_expmap(*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*, *, *dim=-1*, *project=True*) → torch.Tensor

Perform vector transport following u: $\mathfrak{T}_{x \to \operatorname{Exp}(x,u)}(v)$.

Here, Exp is the best possible approximation of the true exponential map. There are cases when the exact variant is hard or impossible implement, therefore a fallback, non-exact, implementation is used.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported tensor

Return type torch.Tensor

transp_follow_retr(*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*, *, *dim=-1*) → torch.Tensor

Perform vector transport following u: $\mathfrak{T}_{x \to \operatorname{retr}(x,u)}(v)$.

This operation is sometimes is much more simpler and can be optimized.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported tensor

Return type torch.Tensor

wrapped_normal(**size*, *mean: torch.Tensor*, *std=1*, *dtype=None*, *device=None*) → geoopt.tensor.ManifoldTensor

Create a point on the manifold, measure is induced by Normal distribution on the tangent space of mean.

Definition is taken from [1] Mathieu, Emile et. al. "Continuous Hierarchical Representations with Poincaré Variational Auto-Encoders." arXiv preprint arxiv:1901.06033 (2019).

Parameters

- **size** (*shape*) – the desired shape
- **mean** (*float|tensor*) – mean value for the Normal distribution
- **std** (*float|tensor*) – std value for the Normal distribution
- **dtype** (*torch.dtype*) – target dtype for sample, if not None, should match Manifold dtype
- **device** (*torch.device*) – target device for sample, if not None, should match Manifold device

Returns random point on the PoincareBall manifold

Return type *ManifoldTensor*

Notes

The device and dtype will match the device and dtype of the Manifold

class geoopt.manifolds.**StereographicExact**(*k=0.0*, *learnable=False*)
κ-Stereographic model.

Parameters **k** (*float|tensor*) – sectional curvature κ of the manifold - k<0: Poincaré ball (stereographic projection of hyperboloid) - k>0: Stereographic projection of sphere - k=0: Euclidean geometry

Notes

It is extremely recommended to work with this manifold in double precision.

http://andbloch.github.io/K-Stereographic-Model/ or *\kappa-Stereographic Projection model*

The implementation of retraction is an exact exponential map, this retraction will be used in optimization.

See also:

Stereographic, *PoincareBall*, *PoincareBallExact*, *SphereProjection*, *SphereProjectionExact*

extra_repr()
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

retr(*x: torch.Tensor*, *u: torch.Tensor*, *, *project=True*, *dim=-1*) → torch.Tensor
Perform an exponential map $\operatorname{Exp}_x(u)$.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

retr_transp(*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*, *, *dim=-1*, *project=True*) → Tuple[torch.Tensor, torch.Tensor]
Perform an exponential map and vector transport from point x with given direction u.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported point

Return type torch.Tensor

transp_follow_retr(*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*, *, *dim=-1*, *project=True*) → torch.Tensor
Perform vector transport following u: $\mathfrak{T}_{x\to\operatorname{Exp}(x,u)}(v)$.

Here, Exp is the best possible approximation of the true exponential map. There are cases when the exact variant is hard or impossible implement, therefore a fallback, non-exact, implementation is used.

Parameters

- **x** (*torch.Tensor*) – point on the manifold

- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported tensor

Return type torch.Tensor

class geoopt.manifolds.**PoincareBall**(*c=1.0*, *learnable=False*)
Poincare ball model.

See more in *\kappa-Stereographic Projection model*

Parameters **c** (*float|tensor*) – ball's negative curvature. The parametrization is constrained to have positive c

Notes

It is extremely recommended to work with this manifold in double precision

See also:

Stereographic, *StereographicExact*, *PoincareBallExact*, *SphereProjection*, *SphereProjectionExact*

class geoopt.manifolds.**PoincareBallExact**(*c=1.0*, *learnable=False*)
Poincare ball model.

See more in *\kappa-Stereographic Projection model*

Parameters **c** (*float|tensor*) – ball's negative curvature. The parametrization is constrained to have positive c

Notes

It is extremely recommended to work with this manifold in double precision

The implementation of retraction is an exact exponential map, this retraction will be used in optimization.

See also:

Stereographic, *StereographicExact*, *PoincareBall*, *SphereProjection*, *SphereProjectionExact*

class geoopt.manifolds.**SphereProjection**(*k=1.0*, *learnable=False*)
Stereographic Projection Spherical model.

See more in *\kappa-Stereographic Projection model*

Parameters **k** (*float|tensor*) – sphere's positive curvature. The parametrization is constrained to have positive k

Notes

It is extremely recommended to work with this manifold in double precision

See also:

Stereographic, *StereographicExact*, *PoincareBall*, *PoincareBallExact*, *SphereProjectionExact*, *Sphere*

class geoopt.manifolds.**SphereProjectionExact**(*k=1.0*, *learnable=False*)
Stereographic Projection Spherical model.

See more in *\kappa-Stereographic Projection model*

Parameters **k** (*float|tensor*) – sphere's positive curvature. The parametrization is constrained to have positive k

Notes

It is extremely recommended to work with this manifold in double precision

The implementation of retraction is an exact exponential map, this retraction will be used in optimization.

See also:

Stereographic, *StereographicExact*, *PoincareBall*, *PoincareBallExact*, *SphereProjectionExact*, *Sphere*

class geoopt.manifolds.**Scaled**(*manifold: geoopt.manifolds.base.Manifold*, *scale=1.0*, *learnable=False*)
Scaled manifold.

Scales all the distances on tha manifold by a constant factor. Scaling may be learnable since the underlying representation is canonical.

Examples

Here is a simple example of radius 2 Sphere

```
>>> import geoopt, torch, numpy as np
>>> sphere = geoopt.Sphere()
>>> radius_2_sphere = Scaled(sphere, 2)
>>> p1 = torch.tensor([-1., 0.])
>>> p2 = torch.tensor([0., 1.])
>>> np.testing.assert_allclose(sphere.dist(p1, p2), np.pi / 2)
>>> np.testing.assert_allclose(radius_2_sphere.dist(p1, p2), np.pi)
```

egrad2rgrad(*x: torch.Tensor*, *u: torch.Tensor*, ***kwargs*) → torch.Tensor
Transform gradient computed using autodiff to the correct Riemannian gradient for the point x.

Parameters

- **torch.Tensor** (u) – point on the manifold
- **torch.Tensor** – gradient to be projected

Returns grad vector in the Riemannian manifold

Return type torch.Tensor

inner(*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor = None*, ***, *keepdim=False*, ***kwargs*) → torch.Tensor
Inner product for tangent vectors at point x.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*Optional[torch.Tensor]*) – tangent vector at point x

- **keepdim** (*bool*) – keep the last dim?

Returns inner product (broadcasted)

Return type torch.Tensor

norm(*x: torch.Tensor*, *u: torch.Tensor*, *, *keepdim=False*, ***kwargs*) → torch.Tensor
Norm of a tangent vector at point x.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **keepdim** (*bool*) – keep the last dim?

Returns inner product (broadcasted)

Return type torch.Tensor

proju(*x: torch.Tensor*, *u: torch.Tensor*, ***kwargs*) → torch.Tensor
Project vector u on a tangent space for x, usually is the same as *egrad2rgrad()*.

Parameters

- **torch.Tensor** (u) – point on the manifold
- **torch.Tensor** – vector to be projected

Returns projected vector

Return type torch.Tensor

projx(*x: torch.Tensor*, ***kwargs*) → torch.Tensor
Project point x on the manifold.

Parameters **torch.Tensor** (x) – point to be projected

Returns projected point

Return type torch.Tensor

random(**size*, *dtype=None*, *device=None*, ***kwargs*) → torch.Tensor
Random sampling on the manifold.

The exact implementation depends on manifold and usually does not follow all assumptions about uniform measure, etc.

transp(*x: torch.Tensor*, *y: torch.Tensor*, *v: torch.Tensor*, ***kwargs*) → torch.Tensor
Perform vector transport $\mathfrak{T}_{x\to y}(v)$.

Parameters

- **x** (*torch.Tensor*) – start point on the manifold
- **y** (*torch.Tensor*) – target point on the manifold
- **v** (*torch.Tensor*) – tangent vector at point x

Returns transported tensor

Return type torch.Tensor

class geoopt.manifolds.**ProductManifold**(**manifolds_with_shape*)
Product Manifold.

Examples

A Torus

```
>>> import geoopt
>>> sphere = geoopt.Sphere()
>>> torus = ProductManifold((sphere, 2), (sphere, 2))
```

component_inner (*x: torch.Tensor*, *u: torch.Tensor*, *v=None*) → torch.Tensor
Inner product for tangent vectors at point x according to components of the manifold.

The result of the function is same as `inner` with `keepdim=True` for all the manifolds except ProductManifold. For this manifold it acts different way computing inner product for each component and then building an output correctly tiling and reshaping the result.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*Optional[torch.Tensor]*) – tangent vector at point x

Returns inner product component wise (broadcasted)

Return type torch.Tensor

Notes

The purpose of this method is better adaptive properties in optimization since ProductManifold will "hide" the structure in public API.

dist (*x*, *y*, ***, *keepdim=False*)
Compute distance between 2 points on the manifold that is the shortest path along geodesics.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **y** (*torch.Tensor*) – point on the manifold
- **keepdim** (*bool*) – keep the last dim?

Returns distance between two points

Return type torch.Tensor

dist2 (*x: torch.Tensor*, *y: torch.Tensor*, ***, *keepdim=False*)
Compute squared distance between 2 points on the manifold that is the shortest path along geodesics.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **y** (*torch.Tensor*) – point on the manifold
- **keepdim** (*bool*) – keep the last dim?

Returns squared distance between two points

Return type torch.Tensor

egrad2rgrad (*x: torch.Tensor*, *u: torch.Tensor*)
Transform gradient computed using autodiff to the correct Riemannian gradient for the point x.

Parameters

- **torch.Tensor** (u) – point on the manifold
- **torch.Tensor** – gradient to be projected

Returns grad vector in the Riemannian manifold

Return type torch.Tensor

expmap (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Perform an exponential map $\operatorname{Exp}_x(u)$.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

expmap_transp (*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]
Perform an exponential map and vector transport from point x with given direction u.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported point

Return type torch.Tensor

classmethod from_point (**parts*, *batch_dims=0*)
Construct Product manifold from given points.

Parameters

- **parts** (*tuple[geoopt.ManifoldTensor]*) – Manifold tensors to construct Product manifold from
- **batch_dims** (*int*) – number of first dims to treat as batch dims and not include in the Product manifold

Returns

Return type *ProductManifold*

inner (*x: torch.Tensor*, *u: torch.Tensor*, *v=None*, ***, *keepdim=False*) → torch.Tensor
Inner product for tangent vectors at point x.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*Optional[torch.Tensor]*) – tangent vector at point x
- **keepdim** (*bool*) – keep the last dim?

Returns inner product (broadcasted)

Return type torch.Tensor

logmap (*x: torch.Tensor*, *y: torch.Tensor*) → torch.Tensor
Perform an logarithmic map $\operatorname{Log}_x(y)$.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **y** (*torch.Tensor*) – point on the manifold

Returns tangent vector

Return type torch.Tensor

origin (**size*, *dtype=None*, *device=None*, *seed=42*) → geoopt.tensor.ManifoldTensor
Create some reasonable point on the manifold in a deterministic way.

For some manifolds there may exist e.g. zero vector or some analogy. In case it is possible to define this special point, this point is returned with the desired size. In other case, the returned point is sampled on the manifold in a deterministic way.

Parameters

- **size** (*Union[int, Tuple[int]]*) – the desired shape
- **device** (*torch.device*) – the desired device
- **dtype** (*torch.dtype*) – the desired dtype
- **seed** (*Optional[int]*) – A parameter controlling deterministic randomness for manifolds that do not provide `.origin`, but provide `.random`. (default: 42)

Returns

Return type torch.Tensor

pack_point (**tensors*) → torch.Tensor
Construct a tensor representation of a manifold point.

In case of regular manifolds this will return the same tensor. However, for e.g. Product manifold this function will pack all non-batch dimensions.

Parameters **tensors** (*Tuple[torch.Tensor]*) –

Returns

Return type torch.Tensor

proju (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Project vector u on a tangent space for x, usually is the same as *egrad2rgrad()*.

Parameters

- **torch.Tensor** (u) – point on the manifold
- **torch.Tensor** – vector to be projected

Returns projected vector

Return type torch.Tensor

projx (*x: torch.Tensor*) → torch.Tensor
Project point x on the manifold.

Parameters **torch.Tensor** (x) – point to be projected

Returns projected point

Return type torch.Tensor

retr (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Perform a retraction from point x with given direction u.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

retr_transp (*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*)
Perform a retraction + vector transport at once.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported point and vectors

Return type Tuple[torch.Tensor, torch.Tensor]

Notes

Sometimes this is a far more optimal way to preform retraction + vector transport

take_submanifold_value (*x: torch.Tensor*, *i: int*, *reshape=True*) → torch.Tensor
Take i'th slice of the ambient tensor and possibly reshape.

Parameters

- **x** (*tensor*) – Ambient tensor
- **i** (*int*) – submanifold index
- **reshape** (*bool*) – reshape the slice?

Returns

Return type torch.Tensor

transp (*x: torch.Tensor*, *y: torch.Tensor*, *v: torch.Tensor*) → torch.Tensor
Perform vector transport $\mathfrak{T}_{x \to y}(v)$.

Parameters

- **x** (*torch.Tensor*) – start point on the manifold
- **y** (*torch.Tensor*) – target point on the manifold
- **v** (*torch.Tensor*) – tangent vector at point x

Returns transported tensor

Return type torch.Tensor

transp_follow_expmap (*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*) → torch.Tensor
Perform vector transport following u: $\mathfrak{T}_{x \to \operatorname{Exp}(x,u)}(v)$.

Here, Exp is the best possible approximation of the true exponential map. There are cases when the exact variant is hard or impossible implement, therefore a fallback, non-exact, implementation is used.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported tensor

Return type torch.Tensor

transp_follow_retr (*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*) → torch.Tensor
Perform vector transport following u: $\mathfrak{T}_{x\to\mathrm{retr}(x,u)}(v)$.

This operation is sometimes is much more simpler and can be optimized.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported tensor

Return type torch.Tensor

unpack_tensor (*tensor: torch.Tensor*) → Tuple[torch.Tensor]
Construct a point on the manifold.

This method should help to work with product and compound manifolds. Internally all points on the manifold are stored in an intuitive format. However, there might be cases, when this representation is simpler or more efficient to store in a different way that is hard to use in practice.

Parameters **tensor** (*torch.Tensor*) –

Returns

Return type torch.Tensor

class geoopt.manifolds.**Lorentz**(*k=1.0*, *learnable=False*)
Lorentz model

Parameters **k** (*float|tensor*) – manifold negative curvature

Notes

It is extremely recommended to work with this manifold in double precision

dist (*x: torch.Tensor*, *y: torch.Tensor*, *, *keepdim=False*, *dim=-1*) → torch.Tensor
Compute distance between 2 points on the manifold that is the shortest path along geodesics.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **y** (*torch.Tensor*) – point on the manifold
- **keepdim** (*bool*) – keep the last dim?

Returns distance between two points

Return type torch.Tensor

egrad2rgrad(*x: torch.Tensor*, *u: torch.Tensor*, **, dim=-1*) → torch.Tensor
Transform gradient computed using autodiff to the correct Riemannian gradient for the point x.

Parameters

- **torch.Tensor** (u) – point on the manifold
- **torch.Tensor** – gradient to be projected

Returns grad vector in the Riemannian manifold

Return type torch.Tensor

expmap(*x: torch.Tensor*, *u: torch.Tensor*, **, norm_tan=True*, *project=True*, *dim=-1*) → torch.Tensor
Perform an exponential map $\operatorname{Exp}_x(u)$.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

inner(*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor = None*, **, keepdim=False*, *dim=-1*) → torch.Tensor
Inner product for tangent vectors at point x.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*Optional[torch.Tensor]*) – tangent vector at point x
- **keepdim** (*bool*) – keep the last dim?

Returns inner product (broadcasted)

Return type torch.Tensor

logmap(*x: torch.Tensor*, *y: torch.Tensor*, **, dim=-1*) → torch.Tensor
Perform an logarithmic map $\operatorname{Log}_x(y)$.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **y** (*torch.Tensor*) – point on the manifold

Returns tangent vector

Return type torch.Tensor

norm(*u: torch.Tensor*, **, keepdim=False*, *dim=-1*) → torch.Tensor
Norm of a tangent vector at point x.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **keepdim** (*bool*) – keep the last dim?

Returns inner product (broadcasted)

Return type torch.Tensor

origin (**size*, *dtype=None*, *device=None*, *seed=42*) → geoopt.tensor.ManifoldTensor
Zero point origin.

Parameters

- **size** (*shape*) – the desired shape
- **device** (*torch.device*) – the desired device
- **dtype** (*torch.dtype*) – the desired dtype
- **seed** (*int*) – ignored

Returns zero point on the manifold

Return type *ManifoldTensor*

proju (*x: torch.Tensor*, *v: torch.Tensor*, ***, *dim=-1*) → torch.Tensor
Project vector u on a tangent space for x, usually is the same as *egrad2rgrad()*.

Parameters

- **torch.Tensor** (u) – point on the manifold
- **torch.Tensor** – vector to be projected

Returns projected vector

Return type torch.Tensor

projx (*x: torch.Tensor*, ***, *dim=-1*) → torch.Tensor
Project point x on the manifold.

Parameters **torch.Tensor** (x) – point to be projected

Returns projected point

Return type torch.Tensor

random_normal (**size*, *mean=0*, *std=1*, *dtype=None*, *device=None*) → geoopt.tensor.ManifoldTensor
Create a point on the manifold, measure is induced by Normal distribution on the tangent space of zero.

Parameters

- **size** (*shape*) – the desired shape
- **mean** (*float|tensor*) – mean value for the Normal distribution
- **std** (*float|tensor*) – std value for the Normal distribution
- **dtype** (*torch.dtype*) – target dtype for sample, if not None, should match Manifold dtype
- **device** (*torch.device*) – target device for sample, if not None, should match Manifold device

Returns random points on Hyperboloid

Return type *ManifoldTensor*

Notes

The device and dtype will match the device and dtype of the Manifold

retr (*x: torch.Tensor*, *u: torch.Tensor*, *, *norm_tan=True*, *project=True*, *dim=-1*) → torch.Tensor
Perform an exponential map $\operatorname{Exp}_x(u)$.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

transp (*x: torch.Tensor*, *y: torch.Tensor*, *v: torch.Tensor*, *, *dim=-1*) → torch.Tensor
Perform vector transport $\mathfrak{T}_{x\to y}(v)$.

Parameters

- **x** (*torch.Tensor*) – start point on the manifold
- **y** (*torch.Tensor*) – target point on the manifold
- **v** (*torch.Tensor*) – tangent vector at point x

Returns transported tensor

Return type torch.Tensor

transp_follow_expmap (*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*, *, *dim=-1*, *project=True*) → torch.Tensor
Perform vector transport following u: $\mathfrak{T}_{x\to \operatorname{Exp}(x,u)}(v)$.

Here, Exp is the best possible approximation of the true exponential map. There are cases when the exact variant is hard or impossible implement, therefore a fallback, non-exact, implementation is used.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported tensor

Return type torch.Tensor

class geoopt.manifolds.**SymmetricPositiveDefinite**(*default_metric: Union[str, geoopt.manifolds.symmetric_positive_definite.SPDMetric] = 'AIM'*)

Manifold of symmetric positive definite matrices.

$$\begin{aligned} A &= A^T \\ \langle x, Ax\rangle > 0 \quad &, \forall x \in \mathrm{R}^n, x \neq 0 \\ A &\in \mathrm{R}^{n\times m} \end{aligned}$$

The tangent space of the manifold contains all symmetric matrices.

References

- https://github.com/pymanopt/pymanopt/blob/master/pymanopt/manifolds/psd.py
- https://github.com/dalab/matrix-manifolds/blob/master/graphembed/graphembed/manifolds/spd.py

Parameters **default_metric** (*Union[str, SPDMetric]*) – one of AIM, SM, LEM. So far only AIM is fully implemented.

dist (*x: torch.Tensor*, *y: torch.Tensor*, *keepdim=False*) → torch.Tensor
Compute distance between 2 points on the manifold that is the shortest path along geodesics.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **y** (*torch.Tensor*) – point on the manifold
- **keepdim** (*bool, optional*) – keep the last dim?, by default False

Returns distance between two points

Return type torch.Tensor

Raises ValueError – if *mode* isn't in *_dist_metric*

egrad2rgrad (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Transform gradient computed using autodiff to the correct Riemannian gradient for the point x.

Parameters

- **torch.Tensor** (u) – point on the manifold
- **torch.Tensor** – gradient to be projected

Returns grad vector in the Riemannian manifold

Return type torch.Tensor

expmap (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Perform an exponential map $\operatorname{Exp}_x(u)$.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

extra_repr () → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

inner (*x: torch.Tensor*, *u: torch.Tensor*, *v: Optional[torch.Tensor] = None*, *keepdim=False*) → torch.Tensor
Inner product for tangent vectors at point x.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*Optional[torch.Tensor]*) – tangent vector at point x
- **keepdim** (*bool*) – keep the last dim?

Returns inner product (broadcasted)

Return type torch.Tensor

Raises `ValueError` – if *keepdim* sine *torch.trace* doesn't support keepdim

logmap (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Perform an logarithmic map $\operatorname{Log}_x(y)$.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **y** (*torch.Tensor*) – point on the manifold

Returns tangent vector

Return type torch.Tensor

origin (**size*, *dtype=None*, *device=None*, *seed: Optional[int] = 42*) → torch.Tensor
Create some reasonable point on the manifold in a deterministic way.

For some manifolds there may exist e.g. zero vector or some analogy. In case it is possible to define this special point, this point is returned with the desired size. In other case, the returned point is sampled on the manifold in a deterministic way.

Parameters

- **size** (*Union[int, Tuple[int]]*) – the desired shape
- **device** (*torch.device*) – the desired device
- **dtype** (*torch.dtype*) – the desired dtype
- **seed** (*Optional[int]*) – A parameter controlling deterministic randomness for manifolds that do not provide `.origin`, but provide `.random`. (default: 42)

Returns

Return type torch.Tensor

proju (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Project vector u on a tangent space for x, usually is the same as *egrad2rgrad()*.

Parameters

- **torch.Tensor** (u) – point on the manifold
- **torch.Tensor** – vector to be projected

Returns projected vector

Return type torch.Tensor

projx (*x: torch.Tensor*) → torch.Tensor
Project point x on the manifold.

Parameters **torch.Tensor** (x) – point to be projected

Returns projected point

Return type torch.Tensor

random (**size*, *dtype=None*, *device=None*, ***kwargs*) → torch.Tensor
Random sampling on the manifold.

The exact implementation depends on manifold and usually does not follow all assumptions about uniform measure, etc.

retr (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Perform a retraction from point x with given direction u.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

transp(*x: torch.Tensor*, *y: torch.Tensor*, *v: torch.Tensor*) → torch.Tensor
Perform vector transport $\mathfrak{T}_{x\to y}(v)$.

Parameters

- **x** (*torch.Tensor*) – start point on the manifold
- **y** (*torch.Tensor*) – target point on the manifold
- **v** (*torch.Tensor*) – tangent vector at point x

Returns transported tensor

Return type torch.Tensor

class geoopt.manifolds.**UpperHalf**(*metric: geoopt.manifolds.siegel.vvd_metrics.SiegelMetricType = <SiegelMetricType.RIEMANNIAN: 'riem'>*, *rank: int = None*)
Upper Half Space Manifold.

This model generalizes the upper half plane model of the hyperbolic plane. Points in the space are complex symmetric matrices.

$$\mathcal{S}_n = \{Z = X + iY \in \mathrm{Sym}(n, \mathbb{C}) | Y >> 0\}.$$

Parameters

- **metric** (*SiegelMetricType*) – one of Riemannian, Finsler One, Finsler Infinity, Finsler metric of minimum entropy, or learnable weighted sum.
- **rank** (*int*) – Rank of the space. Only mandatory for "fmin" and "wsum" metrics.

egrad2rgrad(*z: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Transform gradient computed using autodiff to the correct Riemannian gradient for the point Z.

For a function $f(Z)$ on $\mathcal{S}_n$, the gradient is:

$$\mathrm{grad}_R(f(Z)) = Y \cdot \mathrm{grad}_E(f(Z)) \cdot Y$$

where Y is the imaginary part of Z.

Parameters

- **z** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – gradient to be projected

Returns Riemannian gradient

Return type torch.Tensor

inner(*z: torch.Tensor*, *u: torch.Tensor*, *v=None*, ***, *keepdim=False*) → torch.Tensor
Inner product for tangent vectors at point Z.

The inner product at point $Z = X + iY$ of the vectors U, V is:

$$g_Z(U, V) = \mathrm{Tr}[Y^{-1}UY^{-1}\overline{V}]$$

Parameters

- **z** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point z
- **v** (*torch.Tensor*) – tangent vector at point z
- **keepdim** (*bool*) – keep the last dim?

Returns inner product (broadcasted)

Return type torch.Tensor

origin(**size*, *dtype=None*, *device=None*, *seed: Optional[int] = 42*) → torch.Tensor
Create points at the origin of the manifold in a deterministic way.

For the Upper half model, the origin is the imaginary identity. This is, a matrix whose real part is all zeros, and the identity as the imaginary part.

Parameters

- **size** (*Union[int, Tuple[int]]*) – the desired shape
- **device** (*torch.device*) – the desired device
- **dtype** (*torch.dtype*) – the desired dtype
- **seed** (*Optional[int]*) – A parameter controlling deterministic randomness for manifolds that do not provide `.origin`, but provide `.random`. (default: 42)

Returns

Return type torch.Tensor

projx(*z: torch.Tensor*) → torch.Tensor
Project point Z on the manifold.

In this space, we need to ensure that $Y = Im(Z)$ is positive definite. Since the matrix Y is symmetric, it is possible to diagonalize it. For a diagonal matrix the condition is just that all diagonal entries are positive, so we clamp the values that are <= 0 in the diagonal to an epsilon, and then restore the matrix back into non-diagonal form using the base change matrix that was obtained from the diagonalization.

Parameters **z** (*torch.Tensor*) – point on the manifold

Returns Projected points

Return type torch.Tensor

random(**size*, *dtype=None*, *device=None*, ***kwargs*) → torch.Tensor
Random sampling on the manifold.

The exact implementation depends on manifold and usually does not follow all assumptions about uniform measure, etc.

class geoopt.manifolds.**BoundedDomain**(*metric: geoopt.manifolds.siegel.vvd_metrics.SiegelMetricType = <SiegelMetricType.RIEMANNIAN: 'riem'>*, *rank: int = None*)
Bounded domain Manifold.

This model generalizes the Poincare ball model. Points in the space are complex symmetric matrices.

$$\mathcal{B}_n := \{Z \in \mathrm{Sym}(n, \mathbb{C}) | Id - Z^* Z >> 0\}$$

Parameters

- **metric** (*SiegelMetricType*) – one of Riemannian, Finsler One, Finsler Infinity, Finsler metric of minimum entropy, or learnable weighted sum.
- **rank** (*int*) – Rank of the space. Only mandatory for "fmin" and "wsum" metrics.

dist (*z1: torch.Tensor*, *z2: torch.Tensor*, ***, *keepdim=False*) → torch.Tensor
Compute distance in the Bounded domain model.

To compute distances in the Bounded Domain Model we need to map the elements to the Upper Half Space Model by means of the Cayley Transform, and then compute distances in that domain.

Parameters

- **z1** (*torch.Tensor*) – point on the manifold
- **z2** (*torch.Tensor*) – point on the manifold
- **keepdim** (*bool, optional*) – keep the last dim?, by default False

Returns distance between two points

Return type torch.Tensor

egrad2rgrad (*z: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Transform gradient computed using autodiff to the correct Riemannian gradient for the point Z.

For a function $f(Z)$ on $\mathcal{B}_n$, the gradient is:

$$\operatorname{grad}_R(f(Z)) = A \cdot \operatorname{grad}_E(f(Z)) \cdot A$$

where $A = Id - \overline{Z}Z$

Parameters

- **z** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – gradient to be projected

Returns Riemannian gradient

Return type torch.Tensor

inner (*z: torch.Tensor*, *u: torch.Tensor*, *v=None*, ***, *keepdim=False*) → torch.Tensor
Inner product for tangent vectors at point Z.

The inner product at point $Z = X + iY$ of the vectors U, V is:

$$g_Z(U, V) = \operatorname{Tr}[(Id - \overline{Z}Z)^{-1}U(Id - Z\overline{Z})^{-1}\overline{V}]$$

Parameters

- **z** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point z
- **v** (*torch.Tensor*) – tangent vector at point z
- **keepdim** (*bool*) – keep the last dim?

Returns inner product (broadcasted)

Return type torch.Tensor

origin (**size*, *dtype=None*, *device=None*, *seed: Optional[int] = 42*) → torch.Tensor
Create points at the origin of the manifold in a deterministic way.

For the Bounded domain model, the origin is the zero matrix. This is, a matrix whose real and imaginary parts are all zeros.

Parameters

- **size** (*Union[int, Tuple[int]]*) – the desired shape
- **device** (*torch.device*) – the desired device
- **dtype** (*torch.dtype*) – the desired dtype
- **seed** (*Optional[int]*) – A parameter controlling deterministic randomness for manifolds that do not provide `.origin`, but provide `.random`. (default: 42)

Returns

Return type torch.Tensor

projx(*z: torch.Tensor*) → torch.Tensor
Project point Z on the manifold.

In the Bounded domain model, we need to ensure that $Id - \overline{(}Z)Z$ is positive definite.

Steps to project: Z complex symmetric matrix 1) Diagonalize Z: $Z = \overline{S}DS^*$ 2) Clamp eigenvalues: $D' = clamp(D, max = 1 - epsilon)$ 3) Rebuild Z: $Z' = \overline{S}D'S^*$

Parameters **z** (*torch.Tensor*) – point on the manifold

Returns Projected points

Return type torch.Tensor

random(**size*, *dtype=None*, *device=None*, ***kwargs*) → torch.Tensor
Random sampling on the manifold.

The exact implementation depends on manifold and usually does not follow all assumptions about uniform measure, etc.

2.2 Optimizers

class geoopt.optim.**RiemannianAdam**(**args*, *stabilize=None*, ***kwargs*)
Riemannian Adam with the same API as `torch.optim.Adam`.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float (optional)*) – learning rate (default: 1e-3)
- **betas** (*Tuple[float, float] (optional)*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps** (*float (optional)*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **weight_decay** (*float (optional)*) – weight decay (L2 penalty) (default: 0)
- **amsgrad** (*bool (optional)*) – whether to use the AMSGrad variant of this algorithm from the paper On the Convergence of Adam and Beyond (default: False)

Other Parameters **stabilize** (*int*) – Stabilize parameters if they are off-manifold due to numerical reasons every `stabilize` steps (default: `None` – no stabilize)

step(*closure=None*)
Performs a single optimization step.

Parameters **closure** (*callable, optional*) – A closure that reevaluates the model and returns the loss.

class geoopt.optim.**RiemannianLineSearch**(*params*, *line_search_method='armijo'*, *line_search_params=None*, *cg_method='steepest'*, *cg_kwargs=None*, *compute_derphi=True*, *transport_grad=False*, *transport_search_direction=True*, *fallback_stepsize=1*, *stabilize=None*)

Riemannian line search optimizer.

We try to minimize objective $f\colon M \to \mathbb{R}$, in a search direction η. This is done by minimizing the line search objective

$$\phi(\alpha) = f(R_x(\alpha\eta)),$$

where R_x is the retraction at x. Its derivative is given by

$$\phi'(\alpha) = \langle \mathrm{grad} f(R_x(\alpha\eta)),\, \mathcal{T}_{\alpha\eta}(\eta)\rangle_{R_x(\alpha\eta)},$$

where $\mathcal{T}_\xi(\eta)$ denotes the vector transport of η to the point $R_x(\xi)$.

The search direction η is defined recursively by

$$\eta_{k+1} = -\mathrm{grad} f(R_{x_k}(\alpha_k\eta_k)) + \beta\mathcal{T}_{\alpha_k\eta_k}(\eta_k)$$

Here β is the scale parameter. If $\beta = 0$ this is steepest descent, other choices are Riemannian version of Fletcher-Reeves and Polak-Ribière scale parameters.

Common conditions to accept the new point are the Armijo / sufficient decrease condition:

$$\phi(\alpha) \leq \phi(0) + c_1\alpha\phi'(0)$$

And additionally the curvature / (strong) Wolfe condition

$$\phi'(\alpha) \geq c_2\phi'(0)$$

The Wolfe conditions are more restrictive, but guarantee that search direction η is a descent direction.

The constants c_1 and c_2 satisfy $c_1 \in (0, 1)$ and $c_2 \in (c_1, 1)$.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **line_search_method** (*('wolfe', 'armijo', or callable)*) – Which line_search_method to use. If callable it should be any method of signature *(phi, derphi, **kwargs)* -> *step_size*, where phi is scalar line search objective, and derphi is its derivative. If no suitable step size can be found, the method should return *None*. The following arguments are always passed in ***kwargs*: * **phi0:** float, Value of phi at 0 * **old_phi0:** float, Value of phi at previous point * **derphi0:** float, Value derphi at 0 * **old_derphi0:** float, Value of derphi at previous point * **old_step_size:** float, Stepsize at previous point If any of these arguments are undefined, they default to *None*. Additional arguments can be supplied through the *line_search_params* parameter
- **line_search_params** (*dict*) – Extra parameters to pass to *line_search_method*, for the parameters available to strong Wolfe see `strong_wolfe_line_search()`. For Armijo backtracking parameters see `armijo_backtracking()`.

- **cg_method** (*('steepest', 'fr', 'pr', or callable)*) – Method used to compute the conjugate gradient scale parameter beta. If 'steepest', set the scale parameter to zero, which is equivalent to doing steepest descent. Use 'fr' for Fletcher-Reeves, or 'pr' for Polak-Ribière (NB: this setting requires an additional vector transport). If callable, it should be a function of signature *(params, states, **kwargs)* -> *beta*, where params are the parameters of this optimizer, states are the states associated to the parameters (self._states), and beta is a float giving the scale parameter. The keyword arguments are specified in optional parameter *cg_kwargs*.
- **Paremeters** (*Other*) –
- **----------** –
- **compute_derphi** (*bool, optional*) – If True, compute the derivative of the line search objective phi for every trial step_size alpha. If alpha is not zero, this requires a vector transport and an extra gradient computation. This is always set True if *line_search_method='wolfe'* and False if *'armijo'*, but needs to be manually set for a user implemented line search method.
- **transport_grad** (*bool, optional*) – If True, the transport of the gradient to the new point is computed at the end of every step. Set to *True* if Polak-Ribière is used, otherwise defaults to *False*.
- **transport_search_direction** (*bool, optional*) – If True, transport the search direction to new point at end of every step. Set to False if steepest descent is used, True Otherwise.
- **fallback_stepsize** (*float*) – fallback_stepsize to take if no point can be found satisfying line search conditions. See also *step()* (default: 1)
- **stabilize** (*int*) – Stabilize parameters if they are off-manifold due to numerical reasons every *stabilize* steps (default: *None* – no stabilize)
- **cg_kwargs** (*dict*) – Additional parameters to pass to the method used to compute the conjugate gradient scale parameter.

last_step_size
Last step size taken. If *None* no suitable step size was found, and consequently no step was taken.

Type int or *None*

step_size_history
List of all step sizes taken so far.

Type List[int or *None*]

line_search_method

Type callable

line_search_params

Type dict

cg_method

Type callable

cg_kwargs

Type dict

fallback_stepsize

Type float

step(*closure*, *force_step=False*, *recompute_gradients=False*, *no_step=False*)
Do a linesearch step.

Parameters

- **closure** (*callable*) – A closure that reevaluates the model and returns the loss.
- **force_step** (*bool (optional)*) – If *True*, take a unit step of size *self.fallback_stepsize* if no suitable step size can be found. If *False*, no step is taken in this situation. (default: *False*)
- **recompute_gradients** (*bool (optional)*) – If True, recompute the gradients. Use this if the parameters have changed in between consecutive steps. (default: False)
- **no_step** (*bool (optional)*) – If True, just compute step size and do not perform the step. (default: False)

class geoopt.optim.**RiemannianSGD**(*params*, *lr*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*, *stabilize=None*)
Riemannian Stochastic Gradient Descent with the same API as `torch.optim.SGD`.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*) – learning rate
- **momentum** (*float (optional)*) – momentum factor (default: 0)
- **weight_decay** (*float (optional)*) – weight decay (L2 penalty) (default: 0)
- **dampening** (*float (optional)*) – dampening for momentum (default: 0)
- **nesterov** (*bool (optional)*) – enables Nesterov momentum (default: False)

Other Parameters **stabilize** (*int*) – Stabilize parameters if they are off-manifold due to numerical reasons every `stabilize` steps (default: `None` – no stabilize)

step(*closure=None*)
Performs a single optimization step (parameter update).

Parameters **closure** (*callable*) – A closure that reevaluates the model and returns the loss. Optional for most optimizers.

Note: Unless otherwise specified, this function should not modify the `.grad` field of the parameters.

---

class geoopt.optim.**SparseRiemannianAdam**(*params*, *lr=0.001*, *betas=(0.9*, *0.999)*, *eps=1e-08*, *amsgrad=False*)
Implements lazy version of Adam algorithm suitable for sparse gradients.

In this variant, only moments that show up in the gradient get updated, and only those portions of the gradient get applied to the parameters.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float (optional)*) – learning rate (default: 1e-3)
- **betas** (*Tuple[float, float] (optional)*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))

- **eps** (*float (optional)*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **amsgrad** (*bool (optional)*) – whether to use the AMSGrad variant of this algorithm from the paper On the Convergence of Adam and Beyond (default: False)

Other Parameters **stabilize** (*int*) – Stabilize parameters if they are off-manifold due to numerical reasons every `stabilize` steps (default: `None` – no stabilize)

step(*closure=None*)
Performs a single optimization step (parameter update).

Parameters **closure** (*callable*) – A closure that reevaluates the model and returns the loss. Optional for most optimizers.

Note: Unless otherwise specified, this function should not modify the `.grad` field of the parameters.

class geoopt.optim.**SparseRiemannianSGD**(*params*, *lr*, *momentum=0*, *dampening=0*, *nesterov=False*, *stabilize=None*)
Implements lazy version of SGD algorithm suitable for sparse gradients.

In this variant, only moments that show up in the gradient get updated, and only those portions of the gradient get applied to the parameters.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*) – learning rate
- **momentum** (*float (optional)*) – momentum factor (default: 0)
- **dampening** (*float (optional)*) – dampening for momentum (default: 0)
- **nesterov** (*bool (optional)*) – enables Nesterov momentum (default: False)

Other Parameters **stabilize** (*int*) – Stabilize parameters if they are off-manifold due to numerical reasons every `stabilize` steps (default: `None` – no stabilize)

step(*closure=None*)
Performs a single optimization step (parameter update).

Parameters **closure** (*callable*) – A closure that reevaluates the model and returns the loss. Optional for most optimizers.

Note: Unless otherwise specified, this function should not modify the `.grad` field of the parameters.

2.3 Tensors

class geoopt.tensor.**ManifoldTensor**
Same as `torch.Tensor` that has information about its manifold.

Other Parameters **manifold** (`geoopt.Manifold`) – A manifold for the tensor, (default: `geoopt.Euclidean`)

dist(*other: torch.Tensor*, *p: Union[int, float, bool, str] = 2*, ***kwargs*) → torch.Tensor
Return euclidean or geodesic distance between points on the manifold. Allows broadcasting.

Parameters

- **other** (*tensor*) –
- **p** (*str | int*) – The norm to use. The default behaviour is not changed and is just euclidean distance. To compute geodesic distance, `p` should be set to `"g"`

Returns

Return type scalar

expmap (*u: torch.Tensor*, ***kwargs*) → torch.Tensor
Perform an exponential map $\operatorname{Exp}_x(u)$.

Parameters **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

expmap_transp (*u: torch.Tensor*, *v: torch.Tensor*, ***kwargs*) → torch.Tensor
Perform an exponential map and vector transport from point x with given direction u.

Parameters

- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported point

Return type torch.Tensor

inner (*u: torch.Tensor*, *v: torch.Tensor = None*, ***kwargs*) → torch.Tensor
Inner product for tangent vectors at point x.

Parameters

- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*Optional[torch.Tensor]*) – tangent vector at point x
- **keepdim** (*bool*) – keep the last dim?

Returns inner product (broadcasted)

Return type torch.Tensor

logmap (*y: torch.Tensor*, ***kwargs*) → torch.Tensor
Perform an logarithmic map $\operatorname{Log}_x(y)$.

Parameters **y** (*torch.Tensor*) – point on the manifold

Returns tangent vector

Return type torch.Tensor

proj_ () → torch.Tensor
Inplace projection to the manifold.

Returns same instance

Return type tensor

proju (*u: torch.Tensor*, ***kwargs*) → torch.Tensor
Project vector u on a tangent space for x, usually is the same as `egrad2rgrad()`.

Parameters

- **torch.Tensor** (u) – point on the manifold
- **torch.Tensor** – vector to be projected

Returns projected vector

Return type torch.Tensor

retr(*u: torch.Tensor*, ***kwargs*) → torch.Tensor
Perform a retraction from point x with given direction u.

Parameters **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

retr_transp(*u: torch.Tensor*, *v: torch.Tensor*, ***kwargs*) → Tuple[torch.Tensor, torch.Tensor]
Perform a retraction + vector transport at once.

Parameters

- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported point and vectors

Return type Tuple[torch.Tensor, torch.Tensor]

Notes

Sometimes this is a far more optimal way to preform retraction + vector transport

transp(*y: torch.Tensor*, *v: torch.Tensor*, ***kwargs*) → torch.Tensor
Perform vector transport $\mathfrak{T}_{x\to y}(v)$.

Parameters

- **y** (*torch.Tensor*) – target point on the manifold
- **v** (*torch.Tensor*) – tangent vector at point x

Returns transported tensor

Return type torch.Tensor

transp_follow_expmap(*u: torch.Tensor*, *v: torch.Tensor*, ***kwargs*) → torch.Tensor
Perform vector transport following u: $\mathfrak{T}_{x\to \mathrm{Exp}(x,u)}(v)$.

Here, Exp is the best possible approximation of the true exponential map. There are cases when the exact variant is hard or impossible implement, therefore a fallback, non-exact, implementation is used.

Parameters

- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported tensor

Return type torch.Tensor

transp_follow_retr(*u: torch.Tensor*, *v: torch.Tensor*, ***kwargs*) → torch.Tensor
Perform vector transport following u: $\mathfrak{T}_{x\to \mathrm{retr}(x,u)}(v)$.

This operation is sometimes is much more simpler and can be optimized.

Parameters

- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported tensor

Return type torch.Tensor

unpack_tensor() → Union[torch.Tensor, Tuple[torch.Tensor]]
Construct a point on the manifold.

This method should help to work with product and compound manifolds. Internally all points on the manifold are stored in an intuitive format. However, there might be cases, when this representation is simpler or more efficient to store in a different way that is hard to use in practice.

Parameters **tensor** (*torch.Tensor*) –

Returns

Return type torch.Tensor

class geoopt.tensor.**ManifoldParameter**
Same as `torch.nn.Parameter` that has information about its manifold.

It should be used within `torch.nn.Module` to be recognized in parameter collection.

Other Parameters **manifold** (`geoopt.Manifold` (optional)) – A manifold for the tensor if `data` is not a `geoopt.ManifoldTensor`

2.4 Samplers

class geoopt.samplers.**RHMC**(*params*, *epsilon=0.001*, *n_steps=1*)
Riemannian Hamiltonian Monte-Carlo.

Parameters

- **params** (*iterable*) – iterables of tensors for which to perform sampling
- **epsilon** (*float*) – step size
- **n_steps** (*int*) – number of leapfrog steps

step(*closure*)
Perform a single sampling step.

Parameters **closure** (*callable*) – A closure that reevaluates the model and returns the log probability.

class geoopt.samplers.**RSGLD**(*params*, *epsilon=0.001*)
Riemannian Stochastic Gradient Langevin Dynamics.

Parameters

- **params** (*iterable*) – iterables of tensors for which to perform sampling
- **epsilon** (*float*) – step size

step(*closure*)
Perform a single sampling step.

Parameters **closure** (*callable*) – A closure that reevaluates the model and returns the log probability.

class geoopt.samplers.**SGRHMC**(*params*, *epsilon=0.001*, *n_steps=1*, *alpha=0.1*)
Stochastic Gradient Riemannian Hamiltonian Monte-Carlo.

Parameters

- **params** (*iterable*) – iterables of tensors for which to perform sampling
- **epsilon** (*float*) – step size
- **n_steps** (*int*) – number of leapfrog steps
- **alpha** (*float*) – $(1 - alpha)$ – momentum term

step(*closure*)
Perform a single sampling step.

Parameters **closure** (*callable*) – A closure that reevaluates the model and returns the log probability.

2.5 Extended Guide

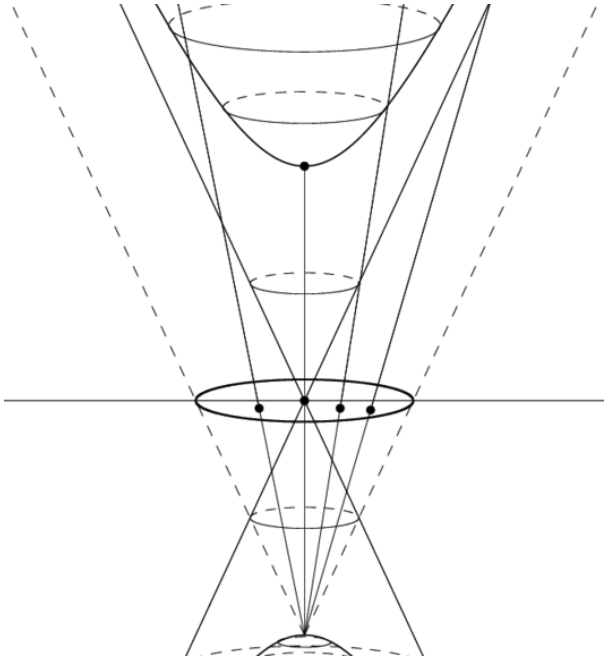
2.5.1 κ-Stereographic Projection model

Stereographic projection models comes to bind constant curvature spaces. Such as spheres, hyperboloids and regular Euclidean manifold. Let's look at what does this mean. As we mentioned constant curvature, let's name this constant κ.

Hyperbolic spaces

Hyperbolic space is a constant negative curvature ($\kappa < 0$) Riemannian manifold. (A very simple example of Riemannian manifold with constant, but positive curvature is sphere, we'll be back to it later)

An (N+1)-dimensional hyperboloid spans the manifold that can be embedded into N-dimensional space via projections.



Originally, the distance between points on the hyperboloid is defined as

$$d(x, y) = \operatorname{arccosh}(x, y)$$

Not to work with this manifold, it is convenient to project the hyperboloid onto a plane. We can do it in many ways recovering embedded manifolds with different properties (usually numerical). To connect constant curvature manifolds we better use Poincare ball model (aka stereographic projection model).

Poincare Model

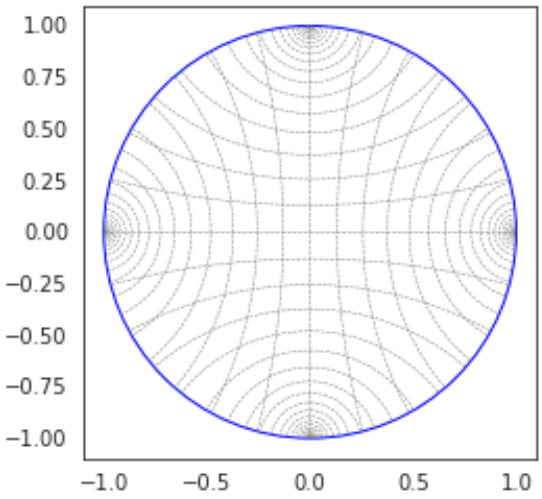


Fig. 1: Grid of Geodesics for $\kappa = -1$, credits to Andreas Bloch

First of all we note, that Poincare ball is embedded in a Sphere of radius $r = 1/\sqrt{\kappa}$, where c is negative curvature. We also note, as κ goes to 0, we recover infinite radius ball. We should expect this limiting behaviour recovers Euclidean geometry.

Spherical Spaces

Another case of constant curvature manifolds is sphere. Unlike Hyperboloid this manifold is compact and has positive κ. But still we can embed a sphere onto a plane ignoring one of the poles.

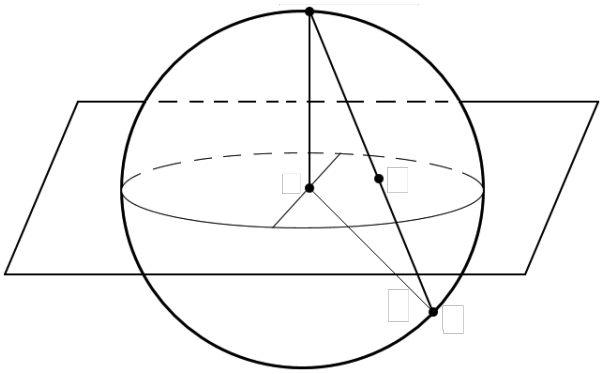
Once we project sphere on the plane we have the following geodesics

Again, similarly to Poincare ball case, we have Euclidean geometry limiting κ to 0.

Universal Curvature Manifold

To connect Euclidean space with its embedded manifold we need to get g_x^κ. It is done via *conformal factor* λ_x^κ. Note, that the metric tensor is conformal, which means all angles between tangent vectors are remained the same compared to what we calculate ignoring manifold structure.

The functions for the mathematics in gyrovector spaces are taken from the following resources:



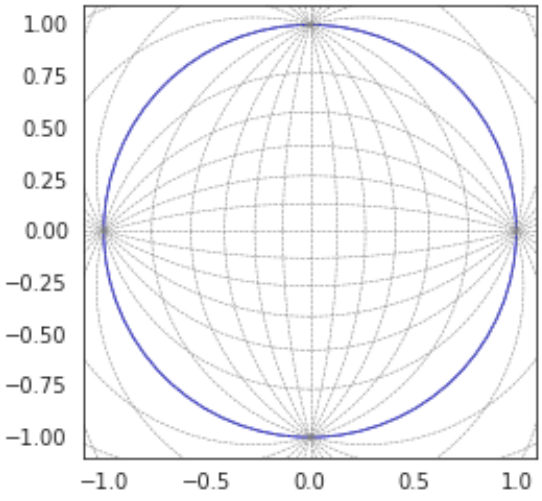


Fig. 2: Grid of Geodesics for $\kappa = 1$, credits to Andreas Bloch

[1] Ganea, Octavian, Gary Bécigneul, and Thomas Hofmann. "Hyperbolic neural networks." Advances in neural information processing systems. 2018.

[2] Bachmann, Gregor, Gary Bécigneul, and Octavian-Eugen Ganea. "Constant Curvature Graph Convolutional Networks." arXiv preprint arXiv:1911.05076 (2019).

[3] Skopek, Ondrej, Octavian-Eugen Ganea, and Gary Bécigneul. "Mixed-curvature Variational Autoencoders." arXiv preprint arXiv:1911.08411 (2019).

[4] Ungar, Abraham A. Analytic hyperbolic geometry: Mathematical foundations and applications. World Scientific, 2005.

[5] Albert, Ungar Abraham. Barycentric calculus in Euclidean and hyperbolic geometry: A comparative introduction. World Scientific, 2010.

geoopt.manifolds.stereographic.math.**lambda_x**(*x: torch.Tensor*, **, k: torch.Tensor*, *keepdim=False*, *dim=-1*)

Compute the conformal factor λ_x^κ for a point on the ball.

$$\lambda_x^\kappa = \frac{2}{1 + \kappa\|x\|_2^2}$$

Parameters

- **x** (*tensor*) – point on the Poincare ball
- **k** (*tensor*) – sectional curvature of manifold
- **keepdim** (*bool*) – retain the last dim? (default: false)
- **dim** (*int*) – reduction dimension

Returns conformal factor

Return type tensor

λ_x^κ connects Euclidean inner product with Riemannian one

geoopt.manifolds.stereographic.math.**inner**(*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*, **, k*, *keepdim=False*, *dim=-1*)

Compute inner product for two vectors on the tangent space w.r.t Riemannian metric on the Poincare ball.

$$\langle u, v\rangle_x = (\lambda_x^\kappa)^2 \langle u, v\rangle$$

Parameters

- **x** (*tensor*) – point on the Poincare ball
- **u** (*tensor*) – tangent vector to x on Poincare ball
- **v** (*tensor*) – tangent vector to x on Poincare ball
- **k** (*tensor*) – sectional curvature of manifold
- **keepdim** (*bool*) – retain the last dim? (default: false)
- **dim** (*int*) – reduction dimension

Returns inner product

Return type tensor

geoopt.manifolds.stereographic.math.**norm**(*x: torch.Tensor*, *u: torch.Tensor*, **, k: torch.Tensor*, *keepdim=False*, *dim=-1*)

Compute vector norm on the tangent space w.r.t Riemannian metric on the Poincare ball.

$$\|u\|_x = \lambda_x^\kappa \|u\|_2$$

Parameters

- **x** (*tensor*) – point on the Poincare ball
- **u** (*tensor*) – tangent vector to x on Poincare ball
- **k** (*tensor*) – sectional curvature of manifold
- **keepdim** (*bool*) – retain the last dim? (default: false)
- **dim** (*int*) – reduction dimension

Returns norm of vector

Return type tensor

geoopt.manifolds.stereographic.math.**egrad2rgrad**(*x: torch.Tensor*, *grad: torch.Tensor*, ***, *k: torch.Tensor*, *dim=-1*)

Convert the Euclidean gradient to the Riemannian gradient.

$$\nabla_x = \nabla_x^E / (\lambda_x^\kappa)^2$$

Parameters

- **x** (*tensor*) – point on the manifold
- **grad** (*tensor*) – Euclidean gradient for x
- **k** (*tensor*) – sectional curvature of manifold
- **dim** (*int*) – reduction dimension for operations

Returns Riemannian gradient $u \in T_x\mathcal{M}_\kappa^n$

Return type tensor

Math

The good thing about Poincare ball is that it forms a Gyrogroup. Minimal definition of a Gyrogroup assumes a binary operation $*$ defined that satisfies a set of properties.

Left identity For every element $a \in G$ there exist $e \in G$ such that $e * a = a$.

Left Inverse For every element $a \in G$ there exist $b \in G$ such that $b * a = e$

Gyroassociativity For any $a, b, c \in G$ there exist $gyr[a, b]c \in G$ such that $a * (b * c) = (a * b) * gyr[a, b]c$

Gyroautomorphism $gyr[a, b]$ is a magma automorphism in G

Left loop $gyr[a, b] = gyr[a * b, b]$

As mentioned above, hyperbolic space forms a Gyrogroup equipped with

geoopt.manifolds.stereographic.math.**mobius_add**(*x: torch.Tensor*, *y: torch.Tensor*, ***, *k: torch.Tensor*, *dim=-1*)

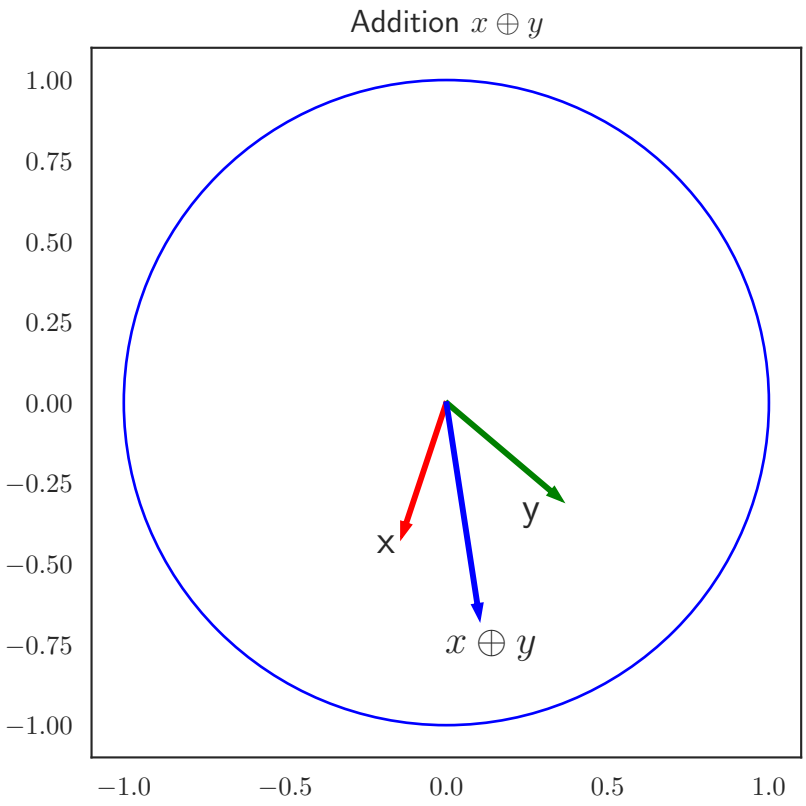
Compute the Möbius gyrovector addition.

$$x \oplus_\kappa y = \frac{(1 - 2\kappa\langle x, y\rangle - \kappa\|y\|_2^2)x + (1 + \kappa\|x\|_2^2)y}{1 - 2\kappa\langle x, y\rangle + \kappa^2\|x\|_2^2\|y\|_2^2}$$

In general this operation is not commutative:

$$x \oplus_\kappa y \neq y \oplus_\kappa x$$

But in some cases this property holds:


Addition $x \oplus y$
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0
x
y
$x \oplus y$

- zero vector case

$$\mathbf{0} \oplus_\kappa x = x \oplus_\kappa \mathbf{0}$$

- zero curvature case that is same as Euclidean addition

$$x \oplus_0 y = y \oplus_0 x$$

Another useful property is so called left-cancellation law:

$$(-x) \oplus_\kappa (x \oplus_\kappa y) = y$$

Parameters

- **x** (*tensor*) – point on the manifold
- **y** (*tensor*) – point on the manifold
- **k** (*tensor*) – sectional curvature of manifold
- **dim** (*int*) – reduction dimension for operations

Returns the result of the Möbius addition

Return type tensor

geoopt.manifolds.stereographic.math.**gyration**(*a: torch.Tensor*, *b: torch.Tensor*, *u: torch.Tensor*, **, k: torch.Tensor, dim=-1*)

Compute the gyration of u by $[a, b]$.

The gyration is a special operation of gyrovector spaces. The gyrovector space addition operation $\oplus_\kappa$ is not associative (as mentioned in *mobius_add()*), but it is gyroassociative, which means

$$u \oplus_\kappa (v \oplus_\kappa w) = (u \oplus_\kappa v) \oplus_\kappa \operatorname{gyr}[u, v]w,$$

where

$$\operatorname{gyr}[u, v]w = \ominus(u \oplus_\kappa v) \oplus (u \oplus_\kappa (v \oplus_\kappa w))$$

We can simplify this equation using the explicit formula for the Möbius addition [1]. Recall,

$$\begin{aligned} A &= -\kappa^2\langle u, w\rangle\langle v, v\rangle - \kappa\langle v, w\rangle + 2\kappa^2\langle u, v\rangle\langle v, w\rangle \\ B &= -\kappa^2\langle v, w\rangle\langle u, u\rangle + \kappa\langle u, w\rangle \\ D &= 1 - 2\kappa\langle u, v\rangle + \kappa^2\langle u, u\rangle\langle v, v\rangle \end{aligned}$$

$$\operatorname{gyr}[u, v]w = w + 2\frac{Au + Bv}{D}.$$

Parameters

- **a** (*tensor*) – first point on manifold
- **b** (*tensor*) – second point on manifold
- **u** (*tensor*) – vector field for operation
- **k** (*tensor*) – sectional curvature of manifold
- **dim** (*int*) – reduction dimension for operations

Returns the result of automorphism

Return type tensor

References

[1] A. A. Ungar (2009), A Gyrovector Space Approach to Hyperbolic Geometry

Using this math, it is possible to define another useful operations

geoopt.manifolds.stereographic.math.**mobius_sub**(*x: torch.Tensor*, *y: torch.Tensor*, ***, *k: torch.Tensor*, *dim=-1*)

Compute the Möbius gyrovector subtraction.

The Möbius subtraction can be represented via the Möbius addition as follows:

$$x \ominus_\kappa y = x \oplus_\kappa (-y)$$

Parameters

- **x** (*tensor*) – point on manifold
- **y** (*tensor*) – point on manifold
- **k** (*tensor*) – sectional curvature of manifold
- **dim** (*int*) – reduction dimension for operations

Returns the result of the Möbius subtraction

Return type tensor

geoopt.manifolds.stereographic.math.**mobius_scalar_mul**(*r: torch.Tensor*, *x: torch.Tensor*, ***, *k: torch.Tensor*, *dim=-1*)

Compute the Möbius scalar multiplication.

$$r \otimes_\kappa x = \tan_\kappa(r \tan_\kappa^{-1}(\|x\|_2)) \frac{x}{\|x\|_2}$$

This operation has properties similar to the Euclidean scalar multiplication

- *n-addition* property

$$r \otimes_\kappa x = x \oplus_\kappa \cdots \oplus_\kappa x$$

- Distributive property

$$(r_1 + r_2) \otimes_\kappa x = r_1 \otimes_\kappa x \oplus r_2 \otimes_\kappa x$$

- Scalar associativity

$$(r_1 r_2) \otimes_\kappa x = r_1 \otimes_\kappa (r_2 \otimes_\kappa x)$$

- Monodistributivity

$$r \otimes_\kappa (r_1 \otimes x \oplus r_2 \otimes x) = r \otimes_\kappa (r_1 \otimes x) \oplus r \otimes (r_2 \otimes x)$$

- Scaling property

$$|r| \otimes_\kappa x / \|r \otimes_\kappa x\|_2 = x / \|x\|_2$$

Parameters

- **r** (*tensor*) – scalar for multiplication
- **x** (*tensor*) – point on manifold
- **k** (*tensor*) – sectional curvature of manifold
- **dim** (*int*) – reduction dimension for operations

Returns the result of the Möbius scalar multiplication

Return type tensor

geoopt.manifolds.stereographic.math.**mobius_pointwise_mul**(*w: torch.Tensor*, *x: torch.Tensor*, **, k: torch.Tensor*, *dim=-1*)

Compute the generalization for point-wise multiplication in gyrovector spaces.

The Möbius pointwise multiplication is defined as follows

$$\operatorname{diag}(w) \otimes_\kappa x = \tan_\kappa\left(\frac{\|\operatorname{diag}(w)x\|_2}{x}\tanh^{-1}(\|x\|_2)\right)\frac{\|\operatorname{diag}(w)x\|_2}{\|x\|_2}$$

Parameters

- **w** (*tensor*) – weights for multiplication (should be broadcastable to x)
- **x** (*tensor*) – point on manifold
- **k** (*tensor*) – sectional curvature of manifold
- **dim** (*int*) – reduction dimension for operations

Returns Möbius point-wise mul result

Return type tensor

geoopt.manifolds.stereographic.math.**mobius_matvec**(*m: torch.Tensor*, *x: torch.Tensor*, **, k: torch.Tensor*, *dim=-1*)

Compute the generalization of matrix-vector multiplication in gyrovector spaces.

The Möbius matrix vector operation is defined as follows:

$$M \otimes_\kappa x = \tan_\kappa\left(\frac{\|Mx\|_2}{\|x\|_2}\tan_\kappa^{-1}(\|x\|_2)\right)\frac{Mx}{\|Mx\|_2}$$

Parameters

- **m** (*tensor*) – matrix for multiplication. Batched matmul is performed if `m.dim() > 2`, but only last dim reduction is supported
- **x** (*tensor*) – point on manifold
- **k** (*tensor*) – sectional curvature of manifold
- **dim** (*int*) – reduction dimension for operations

Returns Möbius matvec result

Return type tensor

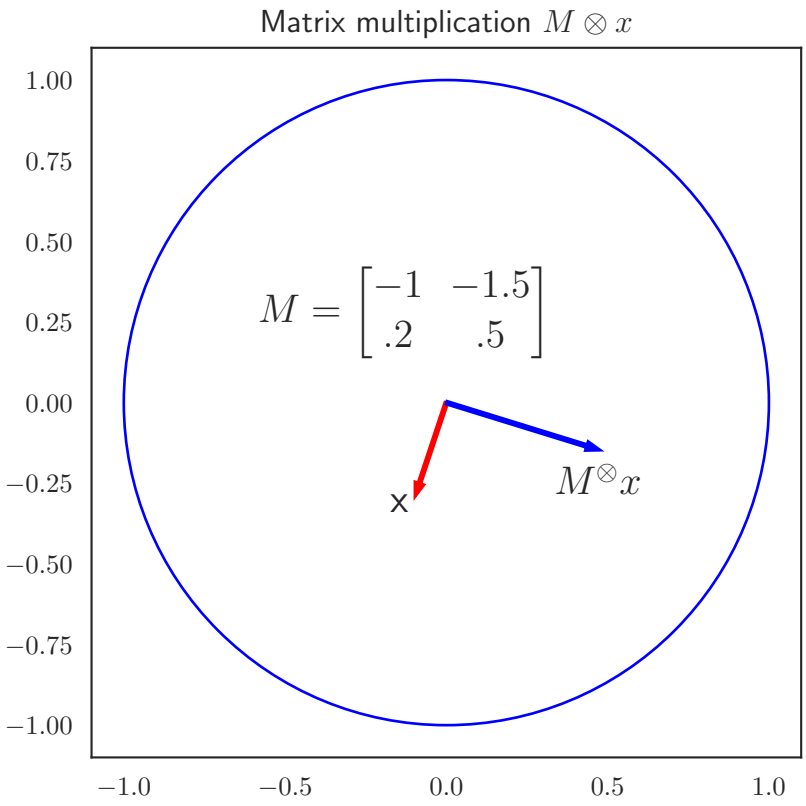
geoopt.manifolds.stereographic.math.**mobius_fn_apply**(*fn: callable*, *x: torch.Tensor*, **args*, *k: torch.Tensor*, *dim=-1*, ***kwargs*)

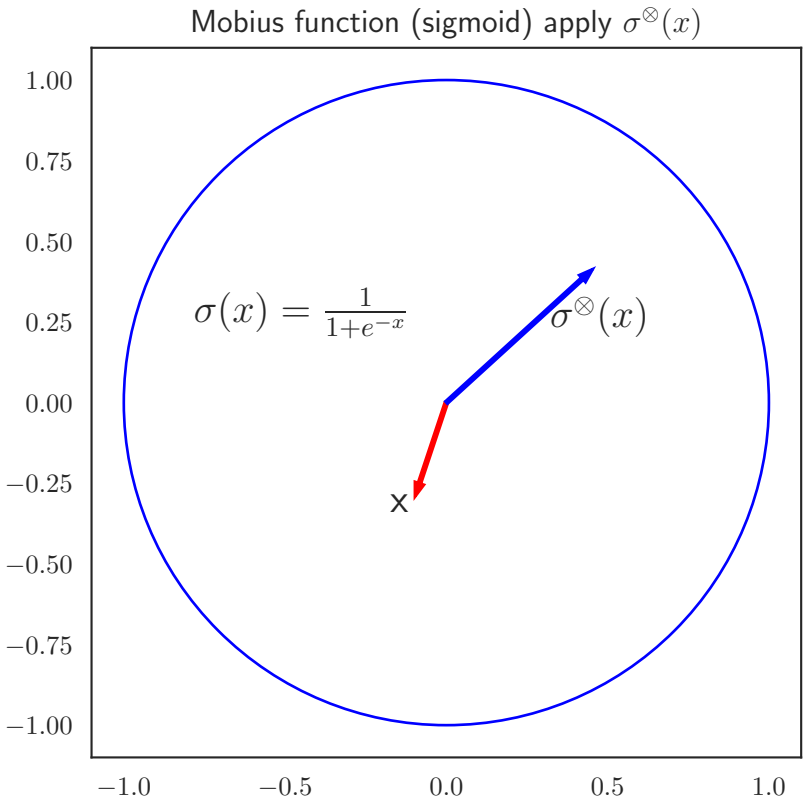
Compute the generalization of function application in gyrovector spaces.

First, a gyrovector is mapped to the tangent space (first-order approx.) via $\log_0^\kappa$ and then the function is applied to the vector in the tangent space. The resulting tangent vector is then mapped back with $\exp_0^\kappa$.

$$f^{\otimes_\kappa}(x) = \exp_0^\kappa(f(\log_0^\kappa(y)))$$

Parameters


Matrix multiplication $M \otimes x$
$M = \begin{bmatrix} -1 & -1.5 \\ .2 & .5 \end{bmatrix}$
x
$M \otimes x$
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0


Mobius function (sigmoid) apply $\sigma^{\otimes}(x)$
$\sigma(x) = \frac{1}{1+e^{-x}}$
$\sigma^{\otimes}(x)$
x
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0

- **x** (*tensor*) – point on manifold
- **fn** (*callable*) – function to apply
- **k** (*tensor*) – sectional curvature of manifold
- **dim** (*int*) – reduction dimension for operations

Returns Result of function in hyperbolic space

Return type tensor

geoopt.manifolds.stereographic.math.**mobius_fn_apply_chain**(*x: torch.Tensor*, **fns*, *k: torch.Tensor*, *dim=-1*)

Compute the generalization of sequential function application in gyrovector spaces.

First, a gyrovector is mapped to the tangent space (first-order approx.) via $\log_0^\kappa$ and then the sequence of functions is applied to the vector in the tangent space. The resulting tangent vector is then mapped back with $\exp_0^\kappa$.

$$f^{\otimes_\kappa}(x) = \exp_0^\kappa(f(\log_0^\kappa(y)))$$

The definition of mobius function application allows chaining as

$$y = \exp_0^\kappa(\log_0^\kappa(y))$$

Resulting in

$$(f \circ g)^{\otimes_\kappa}(x) = \exp_0^\kappa((f \circ g)(\log_0^\kappa(y)))$$

Parameters

- **x** (*tensor*) – point on manifold
- **fns** (*callable[]*) – functions to apply
- **k** (*tensor*) – sectional curvature of manifold
- **dim** (*int*) – reduction dimension for operations

Returns Apply chain result

Return type tensor

Manifold

Now we are ready to proceed with studying distances, geodesics, exponential maps and more

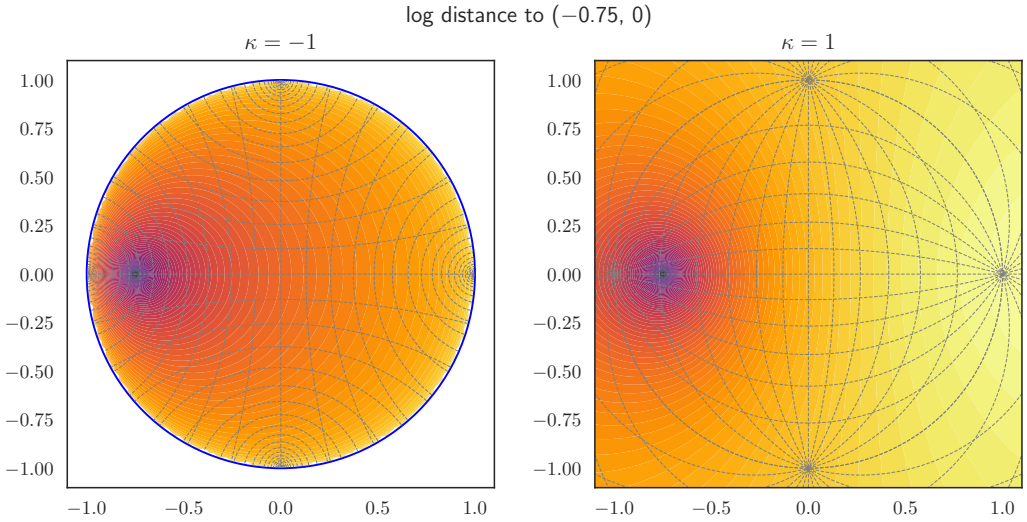
geoopt.manifolds.stereographic.math.**dist**(*x: torch.Tensor*, *y: torch.Tensor*, ***, *k: torch.Tensor*, *keepdim=False*, *dim=-1*)

Compute the geodesic distance between x and y on the manifold.

$$d_\kappa(x, y) = 2 \tan_\kappa^{-1}(\|(-x) \oplus_\kappa y\|_2)$$

Parameters

- **x** (*tensor*) – point on manifold
- **y** (*tensor*) – point on manifold
- **k** (*tensor*) – sectional curvature of manifold



- **keepdim** (*bool*) – retain the last dim? (default: false)
- **dim** (*int*) – reduction dimension

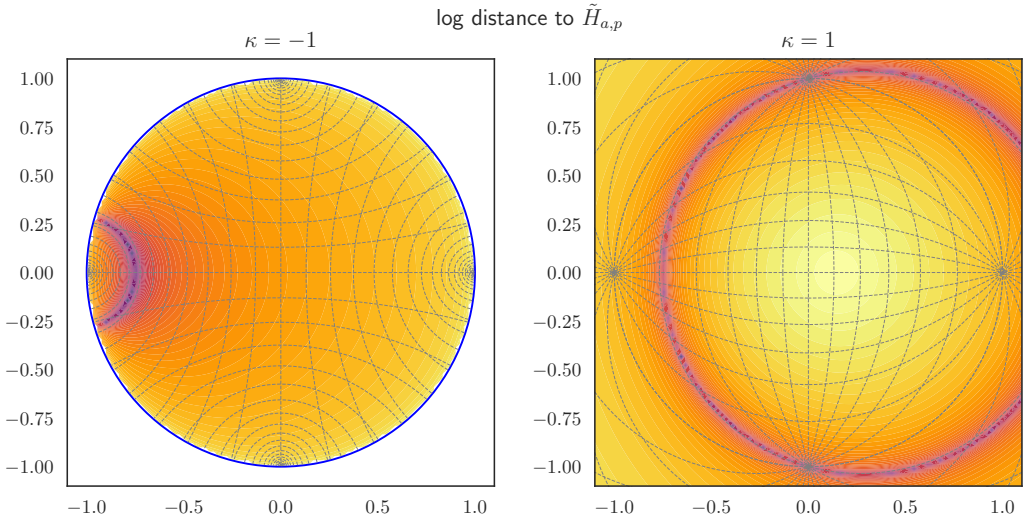
Returns geodesic distance between x and y

Return type tensor

geoopt.manifolds.stereographic.math.**dist2plane**(*x: torch.Tensor*, *p: torch.Tensor*, *a: torch.Tensor*, **, k: torch.Tensor*, *keepdim=False*, *signed=False*, *scaled=False*, *dim=-1*)

Geodesic distance from x to a hyperplane $H_{a,b}$.

The hyperplane is such that its set of points is orthogonal to a and contains p.



To form an intuition what is a hyperplane in gyrovector spaces, let's first consider an Euclidean hyperplane

$$H_{a,b} = \{x \in \mathbb{R}^n \ : \ \langle x, a\rangle - b = 0\}\,,$$

where $a \in \mathbb{R}^n \backslash \{\mathbf{0}\}$ and $b \in \mathbb{R}^n$.

This formulation of a hyperplane is hard to generalize, therefore we can rewrite $\langle x, a\rangle - b$ utilizing orthogonal completion. Setting any p s.t. $b = \langle a, p\rangle$ we have

$$\begin{aligned} H_{a,b} &= \{x \in \mathbb{R}^n \ : \ \langle x, a\rangle - b = 0\} \\ &= H_{a,\langle a,p\rangle} = \tilde{H}_{a,p} \\ &= \{x \in \mathbb{R}^n \ : \ \langle x, a\rangle - \langle a, p\rangle = 0\} \\ &= \{x \in \mathbb{R}^n \ : \ \langle -p + x, a\rangle = 0\} \\ &= p + \{a\}^\perp \end{aligned}$$

Naturally we have a set $\{a\}^\perp$ with applied $+$ operator to each element. Generalizing a notion of summation to the gyrovector space we replace $+$ with $\oplus_\kappa$.

Next, we should figure out what is $\{a\}^\perp$ in the gyrovector space.

First thing that we should acknowledge is that notion of orthogonality is defined for vectors in tangent spaces. Let's consider now $p \in \mathcal{M}^n_\kappa$ and $a \in T_p\mathcal{M}^n_\kappa \backslash \{\mathbf{0}\}$.

Slightly deviating from traditional notation let's write $\{a\}^\perp_p$ highlighting the tight relationship of $a \in T_p\mathcal{M}^n_\kappa \backslash \{\mathbf{0}\}$ with $p \in \mathcal{M}^n_\kappa$. We then define

$$\{a\}^\perp_p := \{z \in T_p\mathcal{M}^n_\kappa \ : \ \langle z, a\rangle_p = 0\}$$

Recalling that a tangent vector z for point p yields $x = \exp^\kappa_p(z)$ we rewrite the above equation as

$$\{a\}^\perp_p := \left\{x \in \mathcal{M}^n_\kappa \ : \ \langle \log^\kappa_p(x), a\rangle_p = 0\right\}$$

This formulation is something more pleasant to work with. Putting all together

$$\begin{aligned} \tilde{H}^\kappa_{a,p} &= p + \{a\}^\perp_p \\ &= \left\{x \in \mathcal{M}^n_\kappa \ : \ \langle \log^\kappa_p(x), a\rangle_p = 0\right\} \\ &= \{x \in \mathcal{M}^n_\kappa \ : \ \langle -p \oplus_\kappa x, a\rangle = 0\} \end{aligned}$$

To compute the distance $d_\kappa(x, \tilde{H}^\kappa_{a,p})$ we find

$$\begin{aligned} d_\kappa(x, \tilde{H}^\kappa_{a,p}) &= \inf_{w \in \tilde{H}^\kappa_{a,p}} d_\kappa(x, w) \\ &= \sin^{-1}_\kappa \left\{ \frac{2|\langle (-p) \oplus_\kappa x, a\rangle|}{(1 + \kappa\|(-p) \oplus_\kappa \|x\|^2_2)\|a\|_2} \right\} \end{aligned}$$

Parameters

- **x** (*tensor*) – point on manifold to compute distance for
- **a** (*tensor*) – hyperplane normal vector in tangent space of p
- **p** (*tensor*) – point on manifold lying on the hyperplane
- **k** (*tensor*) – sectional curvature of manifold
- **keepdim** (*bool*) – retain the last dim? (default: false)
- **signed** (*bool*) – return signed distance
- **scaled** (*bool*) – scale distance by tangent norm
- **dim** (*int*) – reduction dimension for operations

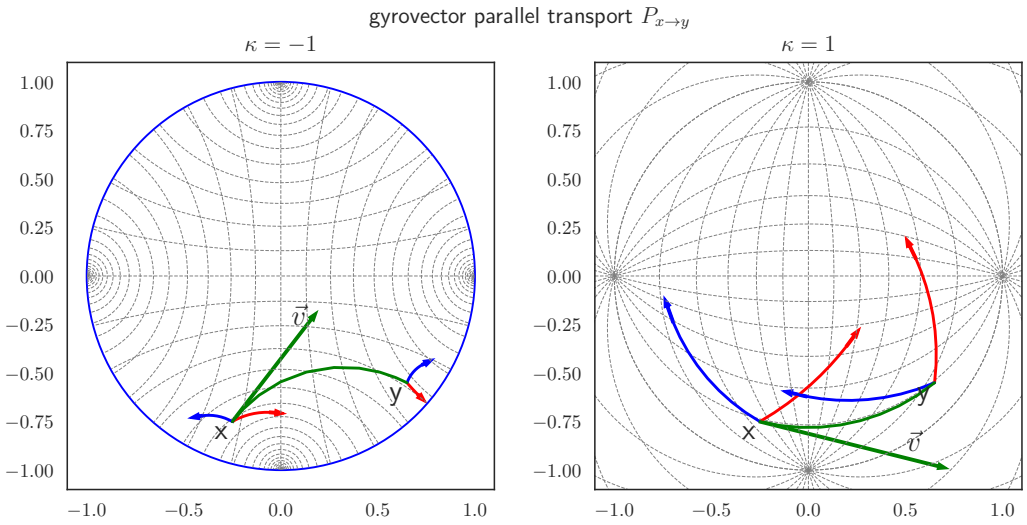
Returns distance to the hyperplane

Return type tensor

geoopt.manifolds.stereographic.math.**parallel_transport**(*x: torch.Tensor*, *y: torch.Tensor*, *v: torch.Tensor*, ***, *k: torch.Tensor*, *dim=-1*)

Compute the parallel transport of v from x to y.

The parallel transport is essential for adaptive algorithms on Riemannian manifolds. For gyrovector spaces the parallel transport is expressed through the gyration.



To recover parallel transport we first need to study isomorphisms between gyrovectors and vectors. The reason is that originally, parallel transport is well defined for gyrovectors as

$$P_{x\to y}(z) = \operatorname{gyr}[y, -x]z,$$

where $x,\ y,\ z \in \mathcal{M}_\kappa^n$ and $\operatorname{gyr}[a, b]c = \ominus(a \oplus_\kappa b) \oplus_\kappa (a \oplus_\kappa (b \oplus_\kappa c))$

But we want to obtain parallel transport for vectors, not for gyrovectors. The blessing is the isomorphism mentioned above. This mapping is given by

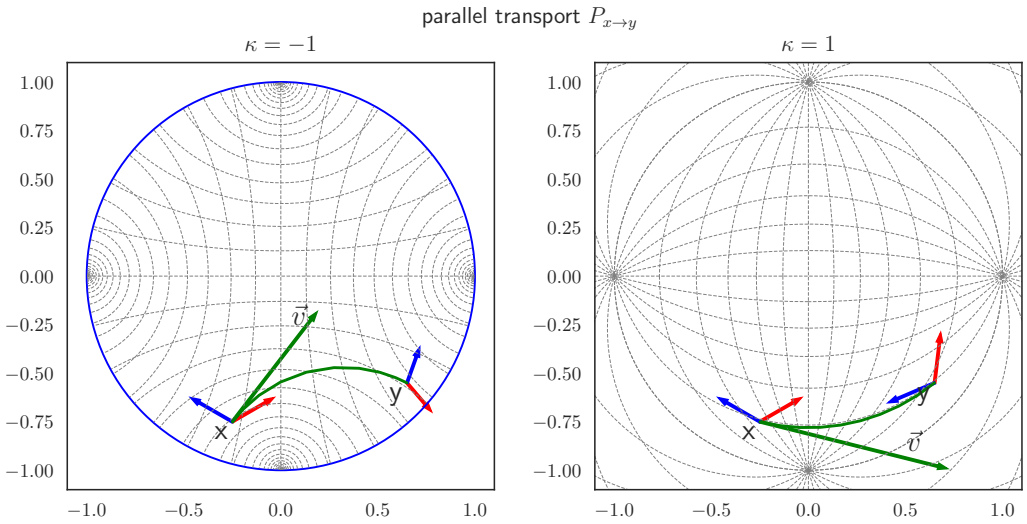
$$U_p^\kappa \ : \ T_p\mathcal{M}_\kappa^n \to \mathbb{G} = v \mapsto \lambda_p^\kappa v$$

Finally, having the points $x,\ y \in \mathcal{M}_\kappa^n$ and a tangent vector $u \in T_x\mathcal{M}_\kappa^n$ we obtain

$$\begin{aligned} P_{x\to y}^\kappa(v) &= (U_y^\kappa)^{-1}\left(\operatorname{gyr}[y, -x]U_x^\kappa(v)\right) \\ &= \operatorname{gyr}[y, -x]v\lambda_x^\kappa/\lambda_y^\kappa \end{aligned}$$

Parameters

- **x** (*tensor*) – starting point
- **y** (*tensor*) – end point
- **v** (*tensor*) – tangent vector at x to be transported to y
- **k** (*tensor*) – sectional curvature of manifold
- **dim** (*int*) – reduction dimension for operations



Returns transported vector

Return type tensor

geoopt.manifolds.stereographic.math.**geodesic**(*t: torch.Tensor*, *x: torch.Tensor*, *y: torch.Tensor*, *, *k: torch.Tensor*, *dim=-1*)

Compute the point on the path connecting x and y at time x.

The path can also be treated as an extension of the line segment to an unbounded geodesic that goes through x and y. The equation of the geodesic is given as:

$$\gamma_{x\to y}(t) = x \oplus_\kappa t \otimes_\kappa ((-x) \oplus_\kappa y)$$

The properties of the geodesic are the following:

$$\gamma_{x\to y}(0) = x$$
$$\gamma_{x\to y}(1) = y$$
$$\dot{\gamma}_{x\to y}(t) = v$$

Furthermore, the geodesic also satisfies the property of local distance minimization:

$$d_\kappa(\gamma_{x\to y}(t_1), \gamma_{x\to y}(t_2)) = v|t_1 - t_2|$$

"Natural parametrization" of the curve ensures unit speed geodesics which yields the above formula with $v = 1$.

However, we can always compute the constant speed v from the points that the particular path connects:

$$v = d_\kappa(\gamma_{x\to y}(0), \gamma_{x\to y}(1)) = d_\kappa(x, y)$$

Parameters

- **t** (*tensor*) – travelling time
- **x** (*tensor*) – starting point on manifold
- **y** (*tensor*) – target point on manifold
- **k** (*tensor*) – sectional curvature of manifold
- **dim** (*int*) – reduction dimension for operations

Returns point on the geodesic going through x and y

Return type tensor

geoopt.manifolds.stereographic.math.**geodesic_unit**(*t: torch.Tensor*, *x: torch.Tensor*, *u: torch.Tensor*, **, k: torch.Tensor*, *dim=-1*)

Compute the point on the unit speed geodesic.

The point on the unit speed geodesic at time t, starting from x with initial direction $u/\|u\|_x$ is computed as follows:

$$\gamma_{x,u}(t) = x \oplus_\kappa \tan_\kappa(t/2)\frac{u}{\|u\|_2}$$

Parameters

- **t** (*tensor*) – travelling time
- **x** (*tensor*) – initial point on manifold
- **u** (*tensor*) – initial direction in tangent space at x
- **k** (*tensor*) – sectional curvature of manifold
- **dim** (*int*) – reduction dimension for operations

Returns the point on the unit speed geodesic

Return type tensor

geoopt.manifolds.stereographic.math.**expmap**(*x: torch.Tensor*, *u: torch.Tensor*, **, k: torch.Tensor*, *dim=-1*)

Compute the exponential map of u at x.

The expmap is tightly related with *geodesic()*. Intuitively, the expmap represents a smooth travel along a geodesic from the starting point x, into the initial direction u at speed $\|u\|_x$ for the duration of one time unit. In formulas one can express this as the travel along the curve $\gamma_{x,u}(t)$ such that

$$\begin{aligned}\gamma_{x,u}(0) &= x\\ \dot\gamma_{x,u}(0) &= u\\ \|\dot\gamma_{x,u}(t)\|_{\gamma_{x,u}(t)} &= \|u\|_x\end{aligned}$$

The existence of this curve relies on uniqueness of the differential equation solution, that is local. For the universal manifold the solution is well defined globally and we have.

$$\begin{aligned}\exp_x^\kappa(u) = \gamma_{x,u}(1) =\\ x \oplus_\kappa \tan_\kappa(\|u\|_x/2)\frac{u}{\|u\|_2}\end{aligned}$$

Parameters

- **x** (*tensor*) – starting point on manifold
- **u** (*tensor*) – speed vector in tangent space at x
- **k** (*tensor*) – sectional curvature of manifold
- **dim** (*int*) – reduction dimension for operations

Returns $\gamma_{x,u}(1)$ end point

Return type tensor

geoopt.manifolds.stereographic.math.**expmap0**(*u: torch.Tensor*, **, k: torch.Tensor*, *dim=-1*)

Compute the exponential map of u at the origin 0.

$$\exp_0^\kappa(u) = \tan_\kappa(\|u\|_2/2)\frac{u}{\|u\|_2}$$

Parameters

- **u** (*tensor*) – speed vector on manifold
- **k** (*tensor*) – sectional curvature of manifold
- **dim** (*int*) – reduction dimension for operations

Returns $\gamma_{0,u}(1)$ end point

Return type tensor

geoopt.manifolds.stereographic.math.**logmap**(*x: torch.Tensor*, *y: torch.Tensor*, **, k: torch.Tensor*, *dim=-1*)

Compute the logarithmic map of y at x.

$$\log_x^\kappa(y) = \frac{2}{\lambda_x^\kappa}\tan_\kappa^{-1}(\|(-x)\oplus_\kappa y\|_2) * \frac{(-x)\oplus_\kappa y}{\|(-x)\oplus_\kappa y\|_2}$$

The result of the logmap is a vector u in the tangent space of x such that

$$y = \exp_x^\kappa(\log_x^\kappa(y))$$

Parameters

- **x** (*tensor*) – starting point on manifold
- **y** (*tensor*) – target point on manifold
- **k** (*tensor*) – sectional curvature of manifold
- **dim** (*int*) – reduction dimension for operations

Returns tangent vector $u \in T_x M$ that transports x to y

Return type tensor

geoopt.manifolds.stereographic.math.**logmap0**(*y: torch.Tensor*, **, k: torch.Tensor*, *dim=-1*)

Compute the logarithmic map of y at the origin 0.

$$\log_0^\kappa(y) = \tan_\kappa^{-1}(\|y\|_2)\frac{y}{\|y\|_2}$$

The result of the logmap at the origin is a vector u in the tangent space of the origin 0 such that

$$y = \exp_0^\kappa(\log_0^\kappa(y))$$

Parameters

- **y** (*tensor*) – target point on manifold
- **k** (*tensor*) – sectional curvature of manifold
- **dim** (*int*) – reduction dimension for operations

Returns tangent vector $u \in T_0 M$ that transports 0 to y

Return type tensor

Stability

Numerical stability is a pain in this model. It is strongly recommended to work in `float64`, so expect adventures in `float32` (but this is not certain).

`geoopt.manifolds.stereographic.math.`**`project`**(*x: torch.Tensor*, ***, *k: torch.Tensor*, *dim=-1*, *eps=-1*)

Safe projection on the manifold for numerical stability.

Parameters

- **x** (*tensor*) – point on the Poincare ball
- **k** (*tensor*) – sectional curvature of manifold
- **dim** (*int*) – reduction dimension to compute norm
- **eps** (*float*) – stability parameter, uses default for dtype if not provided

Returns projected vector on the manifold

Return type tensor

2.6 Developer Guide

2.6.1 Base Manifold

The common base class for all manifolds is `geoopt.manifolds.base.Manifold`.

class `geoopt.manifolds.base.`**`Manifold`**(***kwargs*)

_assert_check_shape(*shape: Tuple[int], name: str*)

Util to check shape and raise an error if needed.

Exhaustive implementation for checking if a given point has valid dimension size, shape, etc. It will raise a ValueError if check is not passed

Parameters

- **shape** (*tuple*) – shape of point on the manifold
- **name** (*str*) – name to be present in errors

Raises `ValueError`

_check_point_on_manifold(*x: torch.Tensor*, ***, *atol=1e-05*, *rtol=1e-05*) → Union[Tuple[bool, Optional[str]], bool]

Util to check point lies on the manifold.

Exhaustive implementation for checking if a given point lies on the manifold. It should return boolean and a reason of failure if check is not passed. You can assume assert_check_point is already passed beforehand

Parameters

- **torch.Tensor** (*x*) – point on the manifold
- **atol** (*float*) – absolute tolerance as in `numpy.allclose()`
- **rtol** (*float*) – relative tolerance as in `numpy.allclose()`

Returns check result and the reason of fail if any

Return type bool, str or None

_check_shape(*shape: Tuple[int], name: str*) → Union[Tuple[bool, Optional[str]], bool]
Util to check shape.

Exhaustive implementation for checking if a given point has valid dimension size, shape, etc. It should return boolean and a reason of failure if check is not passed

Parameters

- **shape** (*Tuple[int]*) – shape of point on the manifold
- **name** (*str*) – name to be present in errors

Returns check result and the reason of fail if any

Return type bool, str or None

_check_vector_on_tangent(*x: torch.Tensor*, *u: torch.Tensor*, ***, *atol=1e-05*, *rtol=1e-05*) → Union[Tuple[bool, Optional[str]], bool]
Util to check a vector belongs to the tangent space of a point.

Exhaustive implementation for checking if a given point lies in the tangent space at x of the manifold. It should return a boolean indicating whether the test was passed and a reason of failure if check is not passed. You can assume assert_check_point is already passed beforehand

Parameters

- **torch.Tensor** (*u*) –
- **torch.Tensor** –
- **atol** (*float*) – absolute tolerance
- **rtol** – relative tolerance

Returns check result and the reason of fail if any

Return type bool, str or None

assert_check_point(*x: torch.Tensor*)
Check if point is valid to be used with the manifold and raise an error with informative message on failure.

Parameters **x** (*torch.Tensor*) – point on the manifold

Notes

This check is compatible to what optimizer expects, last dimensions are treated as manifold dimensions

assert_check_point_on_manifold(*x: torch.Tensor*, ***, *atol=1e-05*, *rtol=1e-05*)
Check if point :math‘x‘ is lying on the manifold and raise an error with informative message on failure.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **atol** (*float*) – absolute tolerance as in numpy.allclose()
- **rtol** (*float*) – relative tolerance as in numpy.allclose()

assert_check_vector(*u: torch.Tensor*)
Check if vector is valid to be used with the manifold and raise an error with informative message on failure.

Parameters **u** (*torch.Tensor*) – vector on the tangent plane

Notes

This check is compatible to what optimizer expects, last dimensions are treated as manifold dimensions

assert_check_vector_on_tangent(*x: torch.Tensor*, *u: torch.Tensor*, ***, *ok_point=False*, *atol=1e-05*, *rtol=1e-05*)

Check if u u is lying on the tangent space to x and raise an error on fail.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – vector on the tangent space to x
- **atol** (*float*) – absolute tolerance as in `numpy.allclose()`
- **rtol** (*float*) – relative tolerance as in `numpy.allclose()`
- **ok_point** (*bool*) – is a check for point required?

check_point(*x: torch.Tensor*, ***, *explain=False*) → Union[Tuple[bool, Optional[str]], bool]

Check if point is valid to be used with the manifold.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **explain** (*bool*) – return an additional information on check

Returns boolean indicating if tensor is valid and reason of failure if False

Return type bool

Notes

This check is compatible to what optimizer expects, last dimensions are treated as manifold dimensions

check_point_on_manifold(*x: torch.Tensor*, ***, *explain=False*, *atol=1e-05*, *rtol=1e-05*) → Union[Tuple[bool, Optional[str]], bool]

Check if point x is lying on the manifold.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **atol** (*float*) – absolute tolerance as in `numpy.allclose()`
- **rtol** (*float*) – relative tolerance as in `numpy.allclose()`
- **explain** (*bool*) – return an additional information on check

Returns boolean indicating if tensor is valid and reason of failure if False

Return type bool

Notes

This check is compatible to what optimizer expects, last dimensions are treated as manifold dimensions

check_vector(*u: torch.Tensor*, ***, *explain=False*)

Check if vector is valid to be used with the manifold.

Parameters

- **u** (*torch.Tensor*) – vector on the tangent plane

- **explain** (*bool*) – return an additional information on check

Returns boolean indicating if tensor is valid and reason of failure if False

Return type bool

Notes

This check is compatible to what optimizer expects, last dimensions are treated as manifold dimensions

check_vector_on_tangent(*x: torch.Tensor*, *u: torch.Tensor*, *, *ok_point=False*, *explain=False*, *atol=1e-05*, *rtol=1e-05*) → Union[Tuple[bool, Optional[str]], bool]

Check if u is lying on the tangent space to x.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – vector on the tangent space to x
- **atol** (*float*) – absolute tolerance as in `numpy.allclose()`
- **rtol** (*float*) – relative tolerance as in `numpy.allclose()`
- **explain** (*bool*) – return an additional information on check
- **ok_point** (*bool*) – is a check for point required?

Returns boolean indicating if tensor is valid and reason of failure if False

Return type bool

component_inner(*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor = None*) → torch.Tensor

Inner product for tangent vectors at point x according to components of the manifold.

The result of the function is same as `inner` with `keepdim=True` for all the manifolds except ProductManifold. For this manifold it acts different way computing inner product for each component and then building an output correctly tiling and reshaping the result.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*Optional[torch.Tensor]*) – tangent vector at point x

Returns inner product component wise (broadcasted)

Return type torch.Tensor

Notes

The purpose of this method is better adaptive properties in optimization since ProductManifold will "hide" the structure in public API.

device

Manifold device.

Returns

Return type Optional[torch.device]

dist(*x: torch.Tensor*, *y: torch.Tensor*, *, *keepdim=False*) → torch.Tensor

Compute distance between 2 points on the manifold that is the shortest path along geodesics.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **y** (*torch.Tensor*) – point on the manifold
- **keepdim** (*bool*) – keep the last dim?

Returns distance between two points

Return type torch.Tensor

dist2(*x: torch.Tensor*, *y: torch.Tensor*, *, *keepdim=False*) → torch.Tensor
Compute squared distance between 2 points on the manifold that is the shortest path along geodesics.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **y** (*torch.Tensor*) – point on the manifold
- **keepdim** (*bool*) – keep the last dim?

Returns squared distance between two points

Return type torch.Tensor

dtype
Manifold dtype.

Returns

Return type Optional[torch.dtype]

egrad2rgrad(*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Transform gradient computed using autodiff to the correct Riemannian gradient for the point x.

Parameters

- **torch.Tensor** (u) – point on the manifold
- **torch.Tensor** – gradient to be projected

Returns grad vector in the Riemannian manifold

Return type torch.Tensor

expmap(*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Perform an exponential map $\operatorname{Exp}_x(u)$.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

expmap_transp(*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]
Perform an exponential map and vector transport from point x with given direction u.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x

- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported point

Return type torch.Tensor

extra_repr()
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

inner(*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor = None*, **, keepdim=False*) → torch.Tensor
Inner product for tangent vectors at point x.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*Optional[torch.Tensor]*) – tangent vector at point x
- **keepdim** (*bool*) – keep the last dim?

Returns inner product (broadcasted)

Return type torch.Tensor

logmap(*x: torch.Tensor*, *y: torch.Tensor*) → torch.Tensor
Perform an logarithmic map $\operatorname{Log}_x(y)$.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **y** (*torch.Tensor*) – point on the manifold

Returns tangent vector

Return type torch.Tensor

norm(*x: torch.Tensor*, *u: torch.Tensor*, **, keepdim=False*) → torch.Tensor
Norm of a tangent vector at point x.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **keepdim** (*bool*) – keep the last dim?

Returns inner product (broadcasted)

Return type torch.Tensor

origin(**size*, *dtype=None*, *device=None*, *seed: Optional[int] = 42*) → torch.Tensor
Create some reasonable point on the manifold in a deterministic way.

For some manifolds there may exist e.g. zero vector or some analogy. In case it is possible to define this special point, this point is returned with the desired size. In other case, the returned point is sampled on the manifold in a deterministic way.

Parameters

- **size** (*Union[int, Tuple[int]]*) – the desired shape
- **device** (*torch.device*) – the desired device

- **dtype** (*torch.dtype*) – the desired dtype
- **seed** (*Optional[int]*) – A parameter controlling deterministic randomness for manifolds that do not provide `.origin`, but provide `.random`. (default: 42)

Returns

Return type torch.Tensor

pack_point (**tensors*) → torch.Tensor
Construct a tensor representation of a manifold point.

In case of regular manifolds this will return the same tensor. However, for e.g. Product manifold this function will pack all non-batch dimensions.

Parameters **tensors** (*Tuple[torch.Tensor]*) –

Returns

Return type torch.Tensor

proju (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Project vector u on a tangent space for x, usually is the same as `egrad2rgrad()`.

Parameters

- **torch.Tensor** (u) – point on the manifold
- **torch.Tensor** – vector to be projected

Returns projected vector

Return type torch.Tensor

projx (*x: torch.Tensor*) → torch.Tensor
Project point x on the manifold.

Parameters **torch.Tensor** (x) – point to be projected

Returns projected point

Return type torch.Tensor

random (**size*, *dtype=None*, *device=None*, ***kwargs*) → torch.Tensor
Random sampling on the manifold.

The exact implementation depends on manifold and usually does not follow all assumptions about uniform measure, etc.

retr (*x: torch.Tensor*, *u: torch.Tensor*) → torch.Tensor
Perform a retraction from point x with given direction u.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x

Returns transported point

Return type torch.Tensor

retr_transp (*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]
Perform a retraction + vector transport at once.

Parameters

- **x** (*torch.Tensor*) – point on the manifold

- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported point and vectors

Return type Tuple[torch.Tensor, torch.Tensor]

Notes

Sometimes this is a far more optimal way to preform retraction + vector transport

transp(*x: torch.Tensor*, *y: torch.Tensor*, *v: torch.Tensor*) → torch.Tensor
Perform vector transport $\mathfrak{T}_{x \to y}(v)$.

Parameters

- **x** (*torch.Tensor*) – start point on the manifold
- **y** (*torch.Tensor*) – target point on the manifold
- **v** (*torch.Tensor*) – tangent vector at point x

Returns transported tensor

Return type torch.Tensor

transp_follow_expmap(*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*) → torch.Tensor
Perform vector transport following u: $\mathfrak{T}_{x \to \operatorname{Exp}(x,u)}(v)$.

Here, Exp is the best possible approximation of the true exponential map. There are cases when the exact variant is hard or impossible implement, therefore a fallback, non-exact, implementation is used.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported tensor

Return type torch.Tensor

transp_follow_retr(*x: torch.Tensor*, *u: torch.Tensor*, *v: torch.Tensor*) → torch.Tensor
Perform vector transport following u: $\mathfrak{T}_{x \to \operatorname{retr}(x,u)}(v)$.

This operation is sometimes is much more simpler and can be optimized.

Parameters

- **x** (*torch.Tensor*) – point on the manifold
- **u** (*torch.Tensor*) – tangent vector at point x
- **v** (*torch.Tensor*) – tangent vector at point x to be transported

Returns transported tensor

Return type torch.Tensor

unpack_tensor(*tensor: torch.Tensor*) → torch.Tensor
Construct a point on the manifold.

This method should help to work with product and compound manifolds. Internally all points on the manifold are stored in an intuitive format. However, there might be cases, when this representation is simpler or more efficient to store in a different way that is hard to use in practice.

Parameters **tensor** (*torch.Tensor*) –

Returns

Return type torch.Tensor

class geoopt.manifolds.base.**ScalingStorage**

Helper class to make implementation transparent.

This is just a dictionary with additional overriden `__call__` for more explicit and elegant API to declare members. A usage example may be found in `Manifold`.

Methods that require rescaling when wrapped into `Scaled` should be defined as follows

1. Regular methods like `dist`, `dist2`, `expmap`, `retr` etc. that are already present in the base class do not require registration, it has already happened in the base `Manifold` class.

2. New methods (like in `PoincareBall`) should be treated with care.

```
class PoincareBall(Manifold):
    # make a class copy of __scaling__ info. Default methods are already present␣
↪there
    __scaling__ = Manifold.__scaling__.copy()
    ... # here come regular implementation of the required methods

    @__scaling__(ScalingInfo(1))  # rescale output according to rule `out *␣
↪scaling ** 1`
    def dist0(self, x: torch.Tensor, *, dim=-1, keepdim=False):
        return math.dist0(x, c=self.c, dim=dim, keepdim=keepdim)

    @__scaling__(ScalingInfo(u=-1))  # rescale argument `u` according to the rule␣
↪`out * scaling ** -1`
    def expmap0(self, u: torch.Tensor, *, dim=-1, project=True):
        res = math.expmap0(u, c=self.c, dim=dim)
        if project:
            return math.project(res, c=self.c, dim=dim)
        else:
            return res
    ... # other special methods implementation
```

3. Some methods are not compliant with the above rescaling rules. We should mark them as *NotCompatible*

```
# continuation of the PoincareBall definition
@__scaling__(ScalingInfo.NotCompatible)
def mobius_fn_apply(
    self, fn: callable, x: torch.Tensor, *args, dim=-1, project=True, **kwargs
):
    res = math.mobius_fn_apply(fn, x, *args, c=self.c, dim=dim, **kwargs)
    if project:
        return math.project(res, c=self.c, dim=dim)
    else:
        return res
```

copy() → a shallow copy of D

class geoopt.manifolds.base.**ScalingInfo**(**results*, ***kwargs*)

Scaling info for each argument that requires rescaling.

```
scaled_value = value * scaling ** power if power != 0 else value
```

For results it is not always required to set powers of scaling, then it is no-op.

The convention for this info is the following. The output of a function is either a tuple or a single object. In any case, outputs are treated as positionals. Function inputs, in contrast, are treated by keywords. It is a common practice to maintain function signature when overriding, so this way may be considered as a sufficient in this particular scenario. The only required info for formula above is `power`.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index

F

G

I

L

M

N

O

P

R

S

T

U

W