The Journal of Open Source Software

Manopt.jl: Optimization on Manifolds in Julia

Ronny Bergmann[1]

1 Norwegian University of Science and Technology, Department of Mathematical Sciences, Trondheim, Norway

DOI: 10.21105/joss.03866

Software

- Review
- Repository
- Archive

Editor: David P. Sanders

Reviewers:

- @krystophny
- @sweichwald

Submitted: 22 July 2021
Published: 04 November 2021

Summary

`Manopt.jl` provides a set of optimization algorithms for optimization problems given on a Riemannian manifold $\mathcal{M}$. Based on a generic optimization framework together with the interface `ManifoldsBase.jl` for Riemannian manifolds, classical and recently developed methods are provided in an efficient implementation. Algorithms include the derivative free Particle Swarm and Nelder–Mead algorithms as well as a classical gradient, conjugate gradient and stochastic gradient descent. Furthermore, quasi-Newton methods like a Riemannian L-BFGS (Huang et al., 2015) and nonsmooth optimization algorithms like a Cyclic Proximal Point Algorithm (Bačák, 2014), a (parallel) Douglas-Rachford algorithm (Bergmann et al., 2016) and a Chambolle-Pock algorithm (Bergmann et al., 2021) are provided together with several basic cost functions, gradients and proximal maps as well as debug and record capabilities.

Statement of Need

In many applications and optimization tasks, nonlinear data appears naturally. For example when data on the sphere is measured, diffusion data can be captured as a signal or even multivariate data of symmetric positive definite matrices, and orientations like they appear for electron backscattered diffraction (EBSD) data. Another example are fixed rank matrices, appearing in dictionary learning. Working on these data, for example doing data interpolation, data approximation, denoising, inpainting, or performing matrix completion, can usually phrased as an optimization problem

$$\text{Minimize} \quad f(x) \quad \text{where } x \in \mathcal{M},$$

where the optimization problem is phrased on a Riemannian manifold $\mathcal{M}$.

A main challenge of these algorithms is that, compared to the (classical) Euclidean case, there is no addition available. For example on the unit sphere $\mathbb{S}^2$ of unit vectors in $\mathbb{R}^3$, adding two vectors of unit lengths yields a vector that is not of unit norm. The resolution is to generalize the notion of a shortest path from the straight line to what is called a (shortest) geodesic, or acceleration free curves. In the same sense, other features and properties have to be rephrased and generalized, when performing optimization on a Riemannian manifold. Algorithms to perform the optimization can still often be stated in the generic form, i.e. on an arbitrary Riemannian manifold $\mathcal{M}$.

Further examples and a thorough introduction can be found in (Absil et al., 2008), (Boumal, 2020).

For a user facing an optimization problem on a manifold, there are two obstacles to the actual numerical optimization: on the one hand, a suitable implementation of the manifold at hand is required, for example how to evaluate the above mentioned geodesics. On the other hand, an implementation of the optimization algorithm that employs said methods from the manifold, such that the algorithm can be applied to the cost function f a user already has.

Using the interface for manifolds, `ManifoldsBase.jl`, the algorithms are implemented in the optimization framework. They can therefore be used with any manifold from `Manifolds.jl` (Axen et al., 2021), a library of efficiently implemented Riemannian manifolds. `Manopt.jl` provides a low level entry to optimization on manifolds while also providing efficient implementations, that can easily be extended to cover own manifolds.

Functionality

`Manopt.jl` provides a comprehensive framework for optimization on Riemannian manifolds and a variety of algorithms using this framework. The framework includes a generic way to specify a step size and a stopping criterion as well as enhance the algorithm with debug and recording capabilities. Each of the algorithms has a high level interface to make it easy to use the algorithms directly.

An optimization task in `Manopt.jl` consists of a `Problem p` and `Options o`, The `Problem` consists of all static information like the cost function and a potential gradient of the optimization task. The `Options` specify the type of algorithm and the settings and data required to run the algorithm. For example by default most options specify that the exponential map, which generalizes the notion of addition to the manifold, should be used and the algorithm steps are performed following an acceleration free curve on the manifold. This might not be known in closed form for some manifolds, e.g. the `Spectrahedron` does not have – to the best of the authors knowledge – a closed form expression for the exponential map. Hence also more general arbitrary retractions can be specified for this instead. Retractions are first order approximations for the exponential map. They provide an alternative to the acceleration free form, if no closed form solution is known. Otherwise, a retraction might also be chosen, when their evaluation is computationally cheaper than to use the exponential map, especially if their approximation error can be stated, see e.g. (Bendokat & Zimmermann, 2021).

Similarly, tangent vectors at different points are identified by a vector transport, which by default is the parallel transport. By providing always a default, a user can start right away without thinking about these details. They can then modify these settings to improve speed or accuracy by specifying other retractions or vector transport to their needs.

The main methods to implement for an own solver are the `initialize_solver!(p,o)` which should fill the data in the options with an initial state. The second method to implement is the `step_solver!(p,o,i)` performing the ith iteration.

Using a decorator pattern, the `Options` can be encapsulated in `DebugOptions` and `RecordOptions` which print and record arbitrary data stored within the `Options`, respectively. This enables to investigate how the optimization is performed in detail and use the algorithms from within this package also for numerical analysis.

In the current version `Manopt.jl` version 0.3.12 the following algorithms are available

- Alternating Gradient Descent (‘alternating_gradient_descent)
- Chambolle-Pock (`ChambollePock`) (Bergmann et al., 2021)
- Conjugate Gradient Descent (`conjugate_gradient_descent`), which includes eight direction update rules using the `coefficient` keyword: `SteepestDirectionUpdateRule`, `ConjugateDescentCoefficient`. `DaiYuanCoefficient`, `FletcherReevesCoefficient`, `HagerZhangCoefficient`, `HeestenesStiefelCoefficient`, `LiuStoreyCoefficient`, and`PolakRibiereCoefficient`

- Cyclic Proximal Point (`cyclic_proximal_point`) (Bačák, 2014)
- (parallel) Douglas–Rachford (`DouglasRachford`) (Bergmann et al., 2016)
- Gradient Descent (`gradient_descent`), including direction update rules (`IdentityUpdateRule` for the classical gradient descent) rules to perform `MomentumGradient`, `AverageGradient`, and `Nesterov` including Momentum, Average, and a Nestorv-type one
- Nelder-Mead (`NelderMead`)
- Particle Swarm Optimization (`particle_swarm`) (Borckmans et al., 2010)
- Quasi-Newton (`quasi_Newton`), with the `BFGS`, `DFP`, `Broyden` and a symmetric rank 1 (`SR1`) update, their inverse updates as well as a limited memory variant of (inverse) BFGS (using the `memory` keyword) (Huang et al., 2015)
- Stochastic Gradient Descent (`stochastic_gradient_descent`)
- Subgradient Method (`subgradient_method`)
- Trust Regions (`trust_regions`), with inner Steihaug-Toint (`truncated_conjugate_gradient_descent`) solver (Absil et al., 2006)

Example

`Manopt.jl` is registered in the general Julia registry and can hence be installed typing `]add Manopt` in Julia REPL. Given the `Sphere` from `Manifolds.jl` and a set of unit vectors $p_1, ..., p_N \in \mathbb{R}^3$, where N is the number of data points. we can compute the generalization of the mean, called the Riemannian Center of Mass (Karcher, 1977), which is defined as the minimizer of the squared distances to the given datan – a property the mean in vector spaces fulfills –

$$\operatorname*{arg\,min}_{x\in\mathcal{M}} \quad \sum_{k=1}^{N} d_{\mathcal{M}}(x, p_k)^2,$$

where $d_{\mathcal{M}}$ denotes length of a shortest geodesic connecting the two points in the arguments. It is called the Riemannian distance. For the sphere this `distance` is given by the length of the shorter great arc connecting the two points.

```julia
using Manopt, Manifolds, LinearAlgebra, Random
Random.seed!(42)
M = Sphere(2)
n = 40
p = 1/sqrt(3) .* ones(3)
B = DefaultOrthonormalBasis()
pts = [ exp(M, p, get_vector(M, p, 0.425*randn(2), B)) for _ in 1:n ]

F(M, y) = sum(1/(2*n) * distance.(Ref(M), pts, Ref(y)).^2)
gradF(M, y) = sum(1/n * grad_distance.(Ref(M), pts, Ref(y)))

x_mean = gradient_descent(M, F, gradF, pts[1])
```

The resulting `x_mean` minimizes the (Riemannian) distances squared but is especially a point of unit norm. Compared to `mean(pts)`, which computes the mean in the embedding of the sphere, the $\mathbb{R}^3$, yields a point "inside" the sphere, since its norm is approximately `0.858`. But even projecting this back onto the sphere, yields a point that does not fulfill the property of minimizing the squared distances.

In the following figure the data `pts` (teal) and the resulting mean (orange) as well as the projected Euclidean mean (small, cyan) are shown.

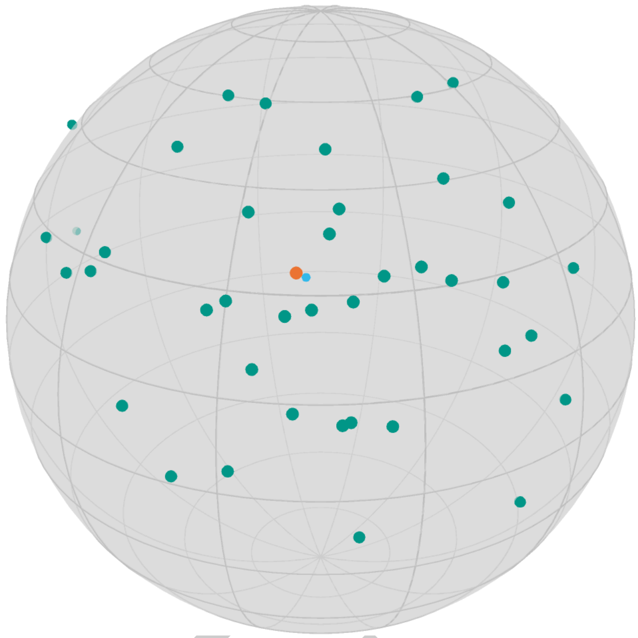

Figure 1: 40 random points `pts` and the result from the gradient descent to compute the `x_mean` (orange) compared to a projection of their (Euclean) mean onto the sphere (cyan).

In order to print the current iteration number, change and cost every iteration as well as the stopping reason, you can provide an `debug` keyword with the corresponding symbols interleaved with strings. The Symbol `:Stop` indicates the stopping reason should be printed in the end. The last integer in this array introduces that only every ith iteration a debug is printed. While `:x` could be used to also print the current iterate, this usually takes up too much space. It might be more reasonable to record these data. The `record` keyword can be used for this, for example to record the current iterate `:x`, the `:Change` from one iterate to the next and the current function value or `:Cost`. To access the recorded values, set `return_options` to `true`, to obtain not only the resulting value as in the example before, but the whole `Options` structure. Then the values can be accessed using the `get_record` function. Just calling `get_record` returns an array of tuples, where each tuple stores the values of one iteration. To obtain an array of values for one recorded value, use the access per symbol, i.e. from the `Iterations` we want to access the recorded iterates `:x` as follows:

```
o = gradient_descent(M, F, gradF, pts[1],
    debug=[:Iteration, " | ", :Change, " | ", :Cost, "\n", :Stop],
    record=[:x, :Change, :Cost],
    return_options=true
)
x_mean_2 = get_solver_result(o) # the solver result
all_values = get_record(o) # a tuple of recorded data per iteration
iterates = get_record(o, :Iteration, :x) # iterates recorded per iteration
```

The debug output of this example looks as follows:

```
Initial |  | F(x): 0.20638171781316278
# 1 | Last Change: 0.22025631624261213 | F(x): 0.18071614247165613
# 2 | Last Change: 0.014654955252636971 | F(x): 0.1805990319857418
# 3 | Last Change: 0.0013696682667046617 | F(x): 0.18059800144857607
# 4 | Last Change: 0.00013562945413135856 | F(x): 0.1805979913344784
```

```
# 5 | Last Change: 1.3519139571830234e-5 | F(x): 0.1805979912339798
# 6 | Last Change: 1.348534506171897e-6 | F(x): 0.18059799123297982
# 7 | Last Change: 1.3493575361575816e-7 | F(x): 0.1805979912329699
# 8 | Last Change: 2.580956827951785e-8 | F(x): 0.18059799123296988
# 9 | Last Change: 2.9802322387695312e-8 | F(x): 0.18059799123296993
The algorithm reached approximately critical point after 9 iterations;
    the gradient norm (1.3387605239861564e-9) is less than 1.0e-8.
```

For more details on more algorithms to compute the mean and other statistical functions on manifolds like the median see https://juliamanifolds.github.io/Manifolds.jl/v0.7/features/statistics.html.

Related research and software

The two projects that are most similar to `Manopt.jl` are `Manopt` (Boumal et al., 2014) in Matlab and `pymanopt` (Townsend et al., 2016) in Python. Similarly `ROPTLIB` (Huang et al., 2018) is a package for optimization on Manifolds in C++. While all three packages cover some algorithms, most are less flexible for example in stating the stopping criterion, which is fixed to mainly maximal number of iterations or a small gradient. Most prominently, `Manopt.jl` is the first package that also covers methods for high-performance and high-dimensional nonsmooth optimization on manifolds.

The Riemannian Chambolle-Pock algorithm presented in (Bergmann et al., 2021) was developed using Manopt.jl. Based on this theory and algorithm, a higher order algorithm was introduced in (Diepeveen & Lellmann, 2021). Optimized examples from (Bergmann & Gousenbourger, 2018) performing data interpolation and approximation with manifold-valued Bézier curves, are also included in `Manopt.jl`.

References

Absil, P.-A., Baker, C. G., & Gallivan, K. A. (2006). Trust-region methods on riemannian manifolds. *Foundations of Computational Mathematics*, *7*(3), 303–330. https://doi.org/10.1007/s10208-005-0179-9

Absil, P.-A., Mahony, R., & Sepulchre, R. (2008). *Optimization algorithms on matrix manifolds*. Princeton University Press. https://doi.org/10.1515/9781400830244

Axen, S. D., Baran, M., Bergmann, R., & Rzecki, K. (2021). *Manifolds.jl: An extensible Julia framework for data analysis on manifolds*. http://arxiv.org/abs/2106.08777

Bačák, M. (2014). Computing medians and means in hadamard spaces. *SIAM Journal on Optimization*, *24*(3), 1542–1566. https://doi.org/10.1137/140953393

Bendokat, T., & Zimmermann, R. (2021). *Efficient quasi-geodesics on the stiefel manifold* (B. F. Nielsen F., Ed.; pp. 763–771). Springer International Publishing. https://doi.org/10.1007/978-3-030-80209-7_82

Bergmann, R., & Gousenbourger, P.-Y. (2018). A variational model for data fitting on manifolds by minimizing the acceleration of a bézier curve. *Frontiers in Applied Mathematics and Statistics*, *4*. https://doi.org/10.3389/fams.2018.00059

Bergmann, R., Herzog, R., Silva Louzeiro, M., Tenbrinck, D., & Vidal-Núñez, J. (2021). Fenchel duality theory and a primal-dual algorithm on riemannian manifolds. *Foundations of Computational Mathematics*. https://doi.org/10.1007/s10208-020-09486-5

Bergmann, R., Persch, J., & Steidl, G. (2016). A parallel douglas rachford algorithm for minimizing ROF-like functionals on images with values in symmetric hadamard manifolds. *SIAM Journal on Imaging Sciences*, *9*(4), 901–937. https://doi.org/10.1137/15M1052858

Borckmans, P. B., Ishteva, M., & Absil, P.-A. (2010). A modified particle swarm optimization algorithm for the best low multilinear rank approximation of higher-order tensors. In *Lecture notes in computer science* (pp. 13–23). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-15461-4_2

Boumal, N. (2020, August). *An introduction to optimization on smooth manifolds*. http://www.nicolasboumal.net/book

Boumal, N., Mishra, B., Absil, P.-A., & Sepulchre, R. (2014). Manopt, a Matlab toolbox for optimization on manifolds. *Journal of Machine Learning Research*, *15*(42), 1455–1459. https://www.manopt.org

Diepeveen, W., & Lellmann, J. (2021). *Duality-based higher-order non-smooth optimization on manifolds*. http://arxiv.org/abs/2102.10309

Huang, W., Absil, P.-A., Gallivan, K. A., & Hand, P. (2018). ROPTLIB: An object-oriented C++ library for optimization on riemannian manifolds. *Association for Computing Machinery. Transactions on Mathematical Software*, *44*(4), Art. 43, 21. https://doi.org/10.1145/3218822

Huang, W., Gallivan, K. A., & Absil, P.-A. (2015). A broyden class of quasi-newton methods for riemannian optimization. *SIAM Journal on Optimization*, *25*(3), 1660–1685. https://doi.org/10.1137/140955483

Huang, W., Gallivan, K. A., & Absil, P.-A. (2015). A broyden class of quasi-newton methods for riemannian optimization. *SIAM Journal on Optimization*, *25*(3), 1660–1685. https://doi.org/10.1137/140955483

Karcher, H. (1977). Riemannian center of mass and mollifier smoothing. *Communications on Pure and Applied Mathematics*, *30*(5), 509–541. https://doi.org/10.1002/cpa.3160300502

Townsend, T., Koep, N., & Weichwald, S. (2016). Pymanopt: A python toolbox for optimization on manifolds using automatic differentiation. *Journal of Machine Learning Research*, *17*(137), 1–5. http://jmlr.org/papers/v17/16-177.html