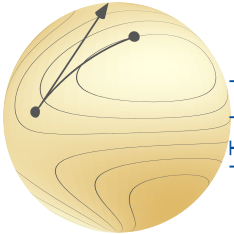


Manopt.jl

Optimization on Riemannian Manifolds

Ronny Bergmann
@ronnybergmann_

Norwegian University of Science and Technology, Trondheim, Norway.

Extended Lightning Talk
JuliaCon 2022, online and everywhere

July, 2022

Norwegian University of Science and Technology

NTNU

Optimization

(Constrained) Optimization aims to find for a function $f\colon \mathbb{R}^m \to \mathbb{R}$ a point

$$\arg\min_{x \in \mathbb{R}^m} f(x)$$

Challenges:

- constrained to some $\mathcal{C} \subset \mathbb{R}^m$, e. g. unit vectors
- symmetries / invariances

Geometric Optimization aims to find

$$\arg\min_{p \in \mathcal{M}} F(p)$$

where F is defined on a Riemannian manifold $\mathcal{M}$, e. g. the sphere $\mathbb{S}^d \subset \mathbb{R}^{d+1}$.
$\Rightarrow$ the problem is unconstrained (again).

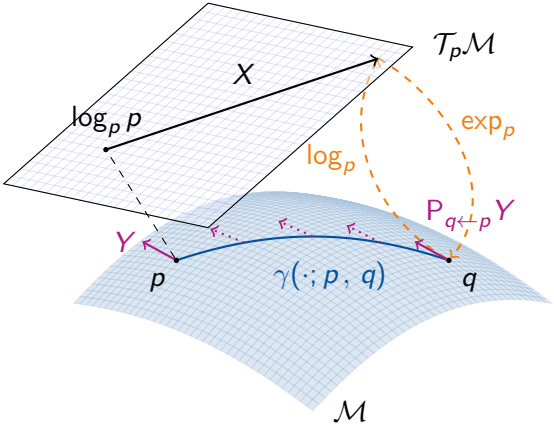
A Riemannian manifold $\mathcal{M}$

A d-dimensional Riemannian manifold can be informally defined as a set $\mathcal{M}$ covered with a 'suitable' collection of charts, that identify subsets of $\mathcal{M}$ with open subsets of $\mathbb{R}^d$ and a continuously varying inner product on the tangent spaces. [Absil, Mahony, and Sepulchre 2008]

A d-dimensional Riemannian manifold $\mathcal{M}$

Notation.

- Geodesic $\gamma(\cdot; p\,,\, q)$
- Tangent space $\mathcal{T}_p\mathcal{M}$
- inner product $(\cdot\,,\,\cdot)_p$
- Logarithmic map $\log_p q = \dot{\gamma}(0; p, q)$
- Exponential map $\exp_p X = \gamma_{p,X}(1)$ where $\gamma_{p,X}(0) = p$ and $\dot{\gamma}_{p,X}(0) = X$
- Parallel transport $\mathrm{P}_{q\leftarrow p} Y$ "move" tangent vectors from $\mathcal{T}_p\mathcal{M}$ to $\mathcal{T}_q\mathcal{M}$



Example I: The Sphere $\mathbb{S}^d \subset \mathbb{R}^{d+1}$

The set of unit vectors or the Sphere

$$\mathbb{S}^d := \left\{ p \in \mathbb{R}^{d+1} \middle| \|p|_2 = 1 \right\}$$

is a Riemannian manifold. A tangent space if of the form

$$T_p\mathbb{S}^d := \left\{ X \in \mathbb{R}^{d+1} \middle| \langle X, p \rangle = 0 \right\}$$

The exponential map is given by "following great arcs" from p in direction X we get

$$\exp_p X = \cos(\|X\|_p)p + \sin(\|X\|_p)\frac{X}{\|X\|_p},$$

But the inverse $\log_p q$ is only locally defined, for example if $p = -q$ are opposite points, there are infinitely many tangent vectors such that $\exp_p X = q$.

Example II: Stiefel & Grassmann

The Stiefel manifold consists of all orthonormal bases (ONB) for k-dimensional subspaces of $\mathbb{R}^n$

$$\mathrm{St}(n,k) := \left\{ p \in \mathbb{R}^{n\times k} \;\middle|\; p^\mathsf{T} p = I_k \right\},$$

For one k-dimensional subspace, there are several ONBs.
Construction: Rotate one ONB such that no vector leaves the subspace
If we are only interested in the subspace, we obtain the Grassmann manifold

$$\mathrm{Gr}(n,k) := \left\{ \mathrm{span}(p) \;\middle|\; p \in \mathbb{R}^{n\times k}, p^\mathsf{T} p = I_k \right\},$$

$\Rightarrow$ All ONBs $p \in \mathrm{St}(n,k)$ of one subspace are the same point $q \in \mathrm{Gr}(n,k)$.
Formally we obtain sets of equivalence classes or a quotient structure

$$\mathrm{Gr}(n,k) = \mathrm{St}(n,k) / \mathrm{O}(k),$$

Implementing a Riemannian manifold

`ManifoldsBase.jl` introduces a manifold type with its field $\mathbb{F} \in \{\mathbb{R}, \mathbb{C}, \mathbb{H}\}$ as parameter to provide an interface for implementing functions like

- `inner(M, p, X, Y)` for the Riemannian metric $(X, Y)_p$
- `exp(M, p, X)` and `log(M, p, q)`,
- more general: `retract(M, p, X, m)`, where `m` is a retraction method
- similarly: `parallel_transport(M, p, X, q)` and `vector_transport_to(M, p, X, q, m)`

for your manifold, which is a subtype of `Manifold{𝔽}`.

😊 mutating version `exp!(M, q, p, X)` works in place in `q`

➔ basis for generic algorithms working on any `Manifold` and generic functions like `norm(M,p,X)`, `geodesic(M, p, X)` and `shortest_geodesic(M, p, q)`

🔗 juliamanifolds.github.io/ManifoldsBase.jl/

Manifolds.jl: A Library of manifolds in Julia

`Manifolds.jl` is based on the `ManifoldsBase.jl` interface.

[Axen, Baran, RB, and Rzecki 2021]

Features.

- different metrics
- Lie groups
- Build manifolds using
  - Product manifold $\mathcal{M}_1 \times \mathcal{M}_2$
  - Power manifold $\mathcal{M}^{n \times m}$
  - Tangent bundle
- Embedded manifolds
- perform statistics
- well-documented, including formulae and references
- well-tested, $>98\,\%$ code cov.

Manifolds. For example

- (unit) Sphere
- Circle & Torus
- Fixed Rank Matrices
- (Generalized) Stiefel & Grassmann
- Hyperbolic space
- Rotations & SO(n)
- Symmetric positive definite matrices
- Symplectic & Symplectic Stiefel
- ...

juliamanifolds.github.io/Manifolds.jl/

JuliaCon 2020 youtu.be/md-FnDGCh9M

Manopt.jl – Internal Structure

`Manopt.jl` is implemented depending only on `ManifoldsBase.jl`.

A solver for an optimization problem consists of three ingredients

- a `Problem P` that specifies static properties
  - the manifold $\mathcal{M}$
  - a cost function $F\colon \mathcal{M} \to \mathbb{R}$
  - (maybe) a gradient $\operatorname{grad} F\colon \mathcal{M} \to T\mathcal{M}$
  - (maybe) a Hessian $\operatorname{Hess} F$
  - ...
- some `Options O` containing dynamic data
  - the current iterate p_i
  - a `StoppingCriterion`
  - any parameter required during an iteration
- implementation of
  1. `initialize_solver!(P, O)` to initialise a solver run
  2. `step_solver!(P, O, i)` to perform the ith step

Running a solver & high level Interfaces

Running a solver consists of

1. generating a `Problem P`
2. generating some `Options O`
3. calling `solve(P,O)`

These steps are usually provided by a high level interface like

Example. For a gradient descent algorithm on a Riemannian manifold one can use

```
gradient_descent(M, F, gradF, p0)
```

which performs

1. create
   - `PG = GradientProblem(M, F, gradF)`
   - `OG = GradientOptions(p0, gradF(M, po))`
2. runs the algorithm by calling `solve(PG,OG)`
3. returns the resulting last iterate (calling `get_solver_result(OG)`)

Stopping Criteria

The `Options` usually include a `StoppingCriterion sc`. This is accessed via `stop_solver!(P,O,i)` at every iteration `i`

A `StoppingCriterion sc` should

- be a functor `sc(P,O,i)` returning true/false
- implement `get_reason(sc)` returning a string with the reason when `true` was returned

😊 Combine stopping criteria using `sc1 | sc2` or `sc1 & sc2`

Examples.

- `StopAfterIteration(N)` - stop after `N` iterations.
- `StopAfterIteration(N) | StopWhenGradientLess(1e-8)`

... or when the gradient is small

Within a step: Stepsize & Linesearch

In many algorithms, after determining a direction "to walk into", e. g.

$$X = -\operatorname{grad} F(p)$$

there is a `Stepsize s` left to determine, which is modelled (again) as a functor `sk = s(p,o,i)`. It can be e. g.

- a `ConstantStepsize(c)`
- an `ArmijoLinesearch(M)`
- a `NonmonotoneLinesearch(M)`

usually the `Options O` then also have a `AbstractRetractionMethod` for example as `O.retraction_method` to perform the actual (gradient) step as

```
retract!(P.N, O.x, O.x, sk * X, O.retraction_method)
```

and the curve `t -> retract(P.M, o.x, t*X)` is also used for line search methods.

Printing debug output

Approach. `D = DebugOptions(O,dA)` where `dA` is a `DebugAction`.

$\Rightarrow$ These options "act like" the original `Options O` But

- in the beginning (overwriting `initialize_step!(P, D)`)
- after each step (overwriting `solver_step!(P, D)`)
- in the end (when the `StoppingCriterion` returns true)

High level interface. Every solver has a `debug=` keyword using `DebugActions`, Strings and Symbols, e. g.

```
debug=[:Iteration, DebugCost(), (:Change, "change: %1.9f\n"), :Stop]
```

prints

- the iteration number and the cost $F(p_k)$ (in default format, also `:Cost`),
- the change $d_{\mathcal{M}}(p_{k-1}, p_k)$ in a specific format
- a line break (after each iteration)
- the reason the algorithm stopped at the end

Recording values

Approach. Analogously `R = RecordOptions(O,rA)` where `rA` is a `RecordAction`.
⇒ These options (also) "act like" the original `Options O` but records

- in the beginning (overwriting `initialize_step!(P, R)`)
- after each step (overwriting `solver_step!(P, R)`)
- in the end (when the `StoppingCriterion` returns true)

High level interface. use the keyword `record=` for example

```
record=[:Iteration, :Cost, :Iterate]
```

and set `return_options=true` ⇒ final state (options `finO`) are returned.

- `get_record(finO)` yields a vector of `(i, cost, point)` tuples
- long form: `get_record(finO, :Iteration, [:Iteration, :Cost, :Iterate])`
- `get_record(finO, :Iteration, :Cost)` yields the vector of recorded `:Cost`s

Manopt.jl – Available Solvers

Currently the following solvers are available

- Gradient Descent
  CG, Stochastic, Momentum, Alternating, Average, Nesterov, ...
- Quasi-Newton
  (L-)BFGS, DFP, Broyden, SR1, ...
- Nelder-Mead, Particle Swarm
- Subgradient Method
- Trust Regions
- Chambolle-Pock (PDHG)
- Douglas-Rachford
- Cyclic Proximal Point

The Manopt Family.

manoptjl.org [RB 2022]

manopt.org [Boumal, Mishra, Absil, and Sepulchre 2014]

pymanopt.org [Townsend, Koep, and Weichwald 2016]

Example Problem: The Riemannian center of mass

The mean of N data points $x_1, \dots, x_N \in \mathbb{R}^n$ is

$$x^* = \frac{1}{N}\sum_{i=1}^{N} x_i \quad \Leftrightarrow \quad x^* = \operatorname*{arg\,min}_{x\in\mathbb{R}^m} \frac{1}{2N}\sum_{i=1}^{N} \|x - x_i\|_2^2$$

$\Rightarrow$ the minimizer of sum of squared distances

For $p_1, \dots, p_N \in \mathcal{M}$: Riemannian center(s) of mass are [Karcher 1977]

$$\operatorname*{arg\,min}_{p\in\mathcal{M}} \frac{1}{2N}\sum_{i=1}^{N} d_{\mathcal{M}}^2(p, p_i),$$

- (in general) neither closed form nor unique
- For $F(p) = \frac{1}{2} d_{\mathcal{M}}^2(p, p_i)$ the gradient is given by $\operatorname{grad} F(p) = -\log_p p_i$

$\Rightarrow$ use gradient descent

Example Codes: The Riemannian center of mass

```
using Manopt, Manifolds, LinearAlgebra
M = Sphere(2)
N = 100

# generate N unit vectors
pts = [normalize(randn(3)) for _ in 1:N]

# define cost and gradient
F(M, p) = sum(pi -> distance(M, pi, p)^2 / 2N, pts)
grad_F(M, p) = sum(pi -> grad_distance(M, pi, p)/N, pts)

# compute a center of mass in place of m
m = copy(M, pts[1])
gradient_descent!(M, F, grad_F, m)

# Alternatively: Use a set of proximal maps and cyclic proximal point
proxes = Function[(M,λ,q) -> prox_distance(M,λ/N,p,q,1) for p in pts]
cyclic_proximal_point(M, F, proxes, pts[1])
```

Summary

`Manopt.jl` is a Julia package that provides

- a framework for optimization algorithms on manifolds
- a library of optimization algorithms within this framework

and includes generic step size / line search functions, debug & record.

Also included. cost functions, gradients, differentials and proximal maps.

...as well as several tutorials at manoptjl.org

Soon. Constrained optimisation algorithms on manifolds,

- Augmented Lagrangian Method
- Exact Penalty Method
- Frank-Wolfe

References

Absil, P.-A., R. Mahony, and R. Sepulchre (2008). *Optimization Algorithms on Matrix Manifolds*. Princeton University Press. DOI: 10.1515/9781400830244.

Axen, S. D., M. Baran, RB, and K. Rzecki (2021). *Manifolds.jl: An Extensible Julia Framework for Data Analysis on Manifolds*. arXiv: 2106.08777.

RB (2022). "Manopt.jl: Optimization on Manifolds in Julia". In: *Journal of Open Source Software* 7.70, p. 3866. DOI: 10.21105/joss.03866.

Boumal, N., B. Mishra, P.-A. Absil, and R. Sepulchre (2014). "Manopt, a Matlab toolbox for optimization on manifolds". In: *The Journal of Machine Learning Research* 15, pp. 1455–1459. URL: https://www.jmlr.org/papers/v15/boumal14a.html.

Karcher, H. (1977). "Riemannian center of mass and mollifier smoothing". In: *Communications on Pure and Applied Mathematics* 30.5, pp. 509–541. DOI: 10.1002/cpa.3160300502.

Townsend, J., N. Koep, and S. Weichwald (2016). "Pymanopt: A Python Toolbox for Optimization on Manifolds using Automatic Differentiation". In: *Journal of Machine Learning Research* 17.137, pp. 1–5. URL: http://jmlr.org/papers/v17/16-177.html.

ronnybergmann.net/talks/2022-JuliaCon-Manoptjl-extended.pdf