ROPTLIB: an object-oriented C++ library for optimization on Riemannian manifolds*

Wen Huang [†¶] P.-A. Absil [‡] K. A. Gallivan [§] Paul Hand [†]

May 11, 2018

Abstract

Riemannian optimization is the task of finding an optimum of a real-valued function defined on a Riemannian manifold. Riemannian optimization has been a topic of much interest over the past few years due to many applications including computer vision, signal processing, and numerical linear algebra. The substantial background required to successfully design and apply Riemannian optimization algorithms is a significant impediment for many potential users. Therefore, multiple packages, such as Manopt (in Matlab) and Pymanopt (in Python), have been developed. This paper describes ROPTLIB, a C++ library for Riemannian optimization. Unlike prior packages, ROPTLIB simultaneously achieves the following goals: i) it has user-friendly interfaces in Matlab, Julia and C++; ii) users do not need to implement manifold- and algorithm-related objects; iii) it provides efficient computational time due to its C++ core; iv) it implements state-of-the-art generic Riemannian optimization algorithms, including quasi-Newton algorithms; and v) it is based on object-oriented programming, allowing users to rapidly add new algorithms and manifolds.

Keywords: Riemannian optimization; non-convex optimization; orthogonal constraints; symmetric positive definite matrices; low-rank matrices; Matlab interface; Julia interface;

1 INTRODUCTION

Riemannian optimization concerns optimizing a real-valued function f defined on a Riemannian manifold $\mathcal{M}$:

$$\min_{x \in \mathcal{M}} f(x).$$

Many problems can be formulated into an optimization problem on a manifold. For example, matrix/tensor completion [Van13, Mis14, KM15, CA16] can be written as an optimization problem over a manifold of matrices/tensors with fixed, low rank. As the second example, finding the Karcher mean (with respect to the affine invariant metric [PFA06a, JVV12]) of a set of symmetric

[†]Department of Computational and Applied Mathematics, Rice University, Houston, USA
[‡]Department of Mathematical Engineering, Université catholique de Louvain, Louvain-la-Neuve, Belgium.
[§]Department of Mathematics, Florida State University, Tallahassee FL, USA.
[¶]Corresponding author. E-mail: huwst08@gmail.com.

*This paper presents research results of the Belgian Network DYSCO (Dynamical Systems, Control, and Optimization), funded by the Interuniversity Attraction Poles Programme initiated by the Belgian Science Policy Office. This work was supported by FNRS under grant PDR T.0173.13.

positive definite (SPD) matrices can be written as an optimization problem over the manifold of SPD matrices [JVV12, YHAG16]. As the third example, the registration problem between two shapes using an elastic shape analysis framework can be written as an optimization problem on the unit sphere in the $\mathbb{L}^2$ space [HGSA15]. As the final example, the phase retrieval problem can be written as an optimization problem on the manifold of Hermitian positive definite matrices with fixed rank [CESV13, WDAM15, HGZ17]. We refer to [AMS08, Hua13] for more applications.

Many effective and efficient optimization methods on Riemannian manifolds have been proposed and analyzed. In 2007, Absil et al. [ABG07] exploited second order information and developed a trust region Newton method. In 2012, Ring and Wirth [RW12] generalized two first order methods—the BFGS method and Fletcher-Reeves nonlinear conjugate gradient method—to the Riemannian setting. The generalization and convergence analyses rely on a step size set by the strong Wolfe condition. In 2015, Huang et al. [HAG15] presented a Riemannian trust region symmetric rank one update method, which combines the trust region with the quasi-Newton approach. In the same year, Sato [Sat16] defined a Dai-Yuan-type Riemannian conjugate gradient method. This method relaxes an assumption required in the Fletcher-Reeves nonlinear conjugate gradient method in [RW12] and only needs the weak Wolfe condition in the line search. Again in the same year, Huang et al. [HGA15] proposed a Broyden family of Riemannian quasi-Newton methods, which includes the well-known Broyden-Fletcher-Goldfarb-Shanno (BFGS) method. Unlike the Riemannian RBFGS in [Ring and Wirth 2012], which requires the differentiated retraction along an arbitrary direction, the Riemannian BFGS by Huang et al. only requires the differentiated retraction along an particular direction, which results in computational benefits in some cases. In 2016, Huang et al. [HAG16a] gave a Riemannian BFGS method by further relaxing requirements on the differentiated retraction.

Several packages exist for Riemannian optimization. Some packages are applicable only to problems on specific manifolds using specific algorithms. For example, a Matlab package [Abr07] developed by Abrudan implements a conjugate gradient algorithm [AEK09] and a steepest descent algorithm [AEK08] only for the unitary matrix constraint. A more recent Matlab package [WY12] gives a Barzilai-Borwein method for manifolds with orthogonality constraints. The R package GrassmannOptim [AW13] has a gradient descent method to solve problems defined on the Grassmann manifold.

The generic Riemannian trust-region (GenRTR) package introduced more flexibility by allowing users to define their own manifolds. This package uses Matlab function handles to split functions related to solvers from functions related to a specific problem. Specifically, in problem-related functions, users are asked to define cost-function-related operations, such as function evaluation, Riemannian gradient evaluation and action of the Riemannian Hessian, and manifold-related operations such as retraction, projection, and evaluation of a Riemannian metric. The function handles of those problem-related functions are then passed to a solver that performs the Riemannian trust region method [ABG07]. While GenRTR allows users to treat optimization algorithms as a black box, it requires users to supply technical operations on Riemannian manifolds.

The Matlab toolbox, Manopt [BMAS14], further improves the ease of use of Riemannian optimization by implementing a broad library of Riemannian manifolds. Consequently, it makes Riemannian optimization easily accessible to users without significant background in this field. Unfortunately, some state-of-the-art Riemannian methods are not implemented in Manopt, such as Riemannian quasi-Newton methods[1]. Further computation may be slow because of the Matlab

[1]To the best of our knowledge, Matlab is not an efficient language for Riemannian quasi-Newton methods. Specifically, since Matlab cannot invoke the rank-1 update function *dsyr* in BLAS directly without going

environment.

An auxiliary package to Manopt is the geometric optimization toolbox (GOPT) [HS14]. This package implements a limited memory version of a Riemannian BFGS method and applies it to problems on the manifold of SPD matrices. Note that its corresponding Riemannian BFGS method does not have any convergence analysis results.

Manopt requires the commercial software Matlab which restricts the range of the potential users. The package Pymanopt [TKW16] implements Manopt using the Python language and adds automated differentiation for calculating gradients. The ability of auto-differentiation further increases the ease of use. Note that Pymanopt contains exactly the same Riemannian optimization algorithms and manifolds as those in Manopt. Therefore, it does not include some state-of-the-art Riemannian algorithms. As Python is interpreted, its computational time is slower than C++. Another Python package is Rieoptpack [RHPA15]. This package contains a limited-memory version of Riemannian BFGS method [HGA15], which is not included in Pymanopt.

Even though Manopt and Pymanopt are user-friendly packages and do not require users to have much knowledge about Riemannian manifolds, it is not easy to have efficient implementations using these two packages. To the best of our knowledge, the interpreted languages, Matlab and Python, are often more than 10 times slower than compiled languages, such as C++ and Fortran (see Section 5). To overcome this difficulty, Matlab and Python allows users to invoke high efficient libraries such as BLAS and LAPACK. It follows that it is difficult to obtain meaningful computational time from a Matlab or Python package in the sense that a function using different implementations may have very different computational time. As a result, some researchers resort to complied languages for efficiency.

A package using a complied language for Riemannian optimization is the C++ library for optimization on Riemannian manifolds (LORM) [Ehl13]. This package focuses on global optimization of polynomials on the sphere, the torus and the special orthogonal group. Multiple initial points are generated and a Riemannian algorithm is used for each initial point. Only a Riemannian steepest descent method and a Riemannian nonlinear conjugate gradient method are implemented.

This paper describes ROPTLIB, a C++ library for Riemannian optimization. Unlike prior packages, ROPTLIB simultaneously achieves the following goals:

1. It is used on any environments that support C++, such as desktop, laptop, and HPC. It has been tested on Windows, Ubuntu and MAC platforms.

2. It has user-friendly interfaces in Matlab, Julia and C++. Users are also allowed to add interface to another language. For example, an interface to the language R can be found in [MRHA16].

3. Users do not need to implement manifold- and algorithm-related objects. If new algorithms and manifolds are necessary, the use of object-oriented programming in ROPTLIB allows users to rapidly add one.

4. It provides efficient computational time due to its C++ core. Compared to interpreted languages such as Matlab and Python, the computational savings occur not only in the functions implemented by users but also in the library.

through a C++ or Fortran interface, the implementation of the Hessian approximation update formula would be slow, (see [RHPA15, HAG15, HGA15] for examples of update formulas). In addition, the efficient vector transport [HAG16b] needs functions e.g., *dgeqrf* and *dormqr*, in LAPACK, which cannot be called from Matlab directly either without through C++ and Fortran interfaces.

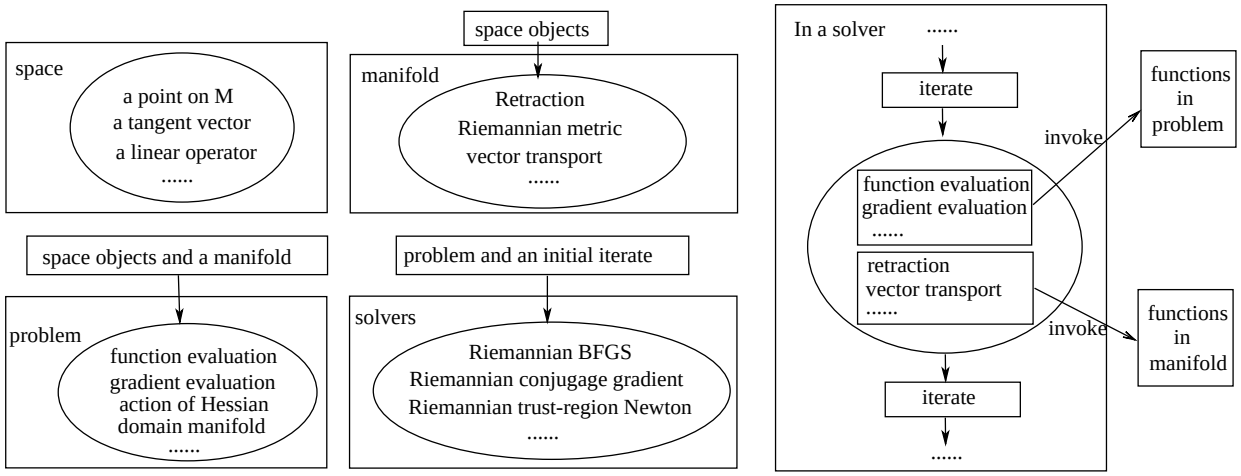

Figure 1: A sketch of the structure of ROPTLIB and the work flow in ROPTLIB.

5. It implements state-of-the-art generic Riemannian optimization algorithms, including quasi-Newton algorithms.

ROPTLIB uses the standard libraries BLAS and LAPACK for efficient linear algebra operations. For examples of using ROPTLIB, see Section 3.

Using object-oriented programming to develop optimization packages is, of course, not new. But as far as we know, most of them are restricted to Euclidean optimization, (see a review of optimization software in [Mit10]). Here, we refer to two excellent review papers [MOHW07] and [PJM12], which describe, respectively, a C++ and a Python Euclidean optimization package. For those unfamiliar with object-oriented programming terminology, we refer to [LLM12].

This paper is organized as follows. In Section 2, we present the structure and the philosophy of ROPTLIB and its main classes. Section 3 gives an example that uses ROPTLIB to solve a problem on the Stiefel manifold. Section 4 demonstrates the importance of ROPTLIB by two applications. A benchmark is given in Section 5. Conclusion and future work are in Section 6.

2 SOFTWARE DESCRIPTION

The idea behind the ROPTLIB software design is to guarantee the ease of use for multiple types of users, including general users that want to solve particular optimization problems over commonly-used manifolds and developers that want to extend ROPTLIB to include new algorithms or manifolds. Therefore, we divided the classes of ROPTLIB into four families: i) space-related classes, ii) manifold-related classes, iii) problem-related classes, and iv) solver-related classes. The four families are separated in the sense that a class in a family is not built on any class in another family. This approach enables maximal code reuse each time a new problem is presented, a new algorithm is developed or a new manifold is added.

Figure 1 sketches the structure and relationship between the four families of classes and the work flow in ROPTLIB. The space-related classes define objects on manifolds, such as a point

on a manifold, a tangent vector on a manifold, and a linear operator on a manifold. It supports the copy-on-write strategy (see Section 2.1), which avoids some unnecessary copy operations. The manifold-related classes define operations on manifolds. The functions in those classes receive objects as input arguments, such as points on a manifold and tangent vectors of a manifold, to perform operations, such as retraction, vector transport, and the evaluation of a Riemannian metric. The problem-related classes define cost-function evaluation, gradient evaluation and the action of the Hessian. The domain of a problem is specified using a pointer to a manifold. The solver-related classes receive a problem-related class and a point on the domain manifold (an initial iterate) to perform a specified Riemannian optimization algorithm. Specifically, as shown in the right of Figure 1, a solver follows the specified algorithmic rule to produce new iterates. During this process, functions in a problem-related class and manifold-related class are invoked when needed. The class hierarchies of the four families are described separately in detail below.

ROPTLIB has prototypes of operations for the four families of classes. The state-of-the-art Riemannian optimization algorithms and many commonly-used manifolds are included in ROPTLIB with user-friendly interfaces. If the problems given by users are defined on manifolds which have been implemented in ROPTLIB, then the users are only required to write problem-related classes defining their own problems. Note that ROPTLIB only needs Euclidean gradient and action of Euclidean Hessian since the manifold-related classes are able to convert them to corresponding Riemannian gradient and action of Riemannian Hessian automatically.

Throughout this paper, a class or a function is written in *this italics font* and an object is written in **this boldface font**.

Polymorphism is particularly important in ROPTLIB, since in an optimization algorithm, it is unknown what problem or manifold is used, and polymorphism allows the solvers to automatically choose the correct problem object and the correct manifold object. For example, in a solver, a user-defined problem class is pointed to by a pointer of the base class, *Problem*. When invoking member functions using the pointer of *Problem* class, the functions defined in the user-defined derived class rather than functions in *Problem* are used. This property of automatically choosing member functions based on the true type of object rather than the type of pointer is polymorphism.

2.1 Space Classes

ROPTLIB provides its own space-related classes since the container classes in the standard C++ libraries do not support the two features that are crucial to the efficiency of ROPTLIB: copy-on-write and storing temporary data.

Copy-on-write is important to save computational time especially when handling large-scale problems. If data stored in memory is used in multiple tasks and the data only need be modified occasionally, then one does not have to create multiple copies of the data for each of the tasks. A copy is created only if the data in a task is required to be modified. For example, suppose A is a 1000-by-1000 matrix. When the matrix A is assigned to a matrix B, it is not necessary to create a new copy for the 1000-by-1000 matrix immediately. One can simply assign the address of the matrix A to B. A new copy is created only when one of the matrices is modified.

The class hierarchy of space-related classes is given in Figure 2. The class *SmartSpace* is the pure virtual class that defines the most basic behavior of copy-on-write. The pointer *Space* points to the memory of the data and *sharedtimes* gives the number of objects using this memory. Three member functions *ObtainReadData*, *ObtainWriteEntireData*, and *ObtainWritePartialData* define three different ways to handle the data in the memory. *ObtainReadData* returns a constant pointer

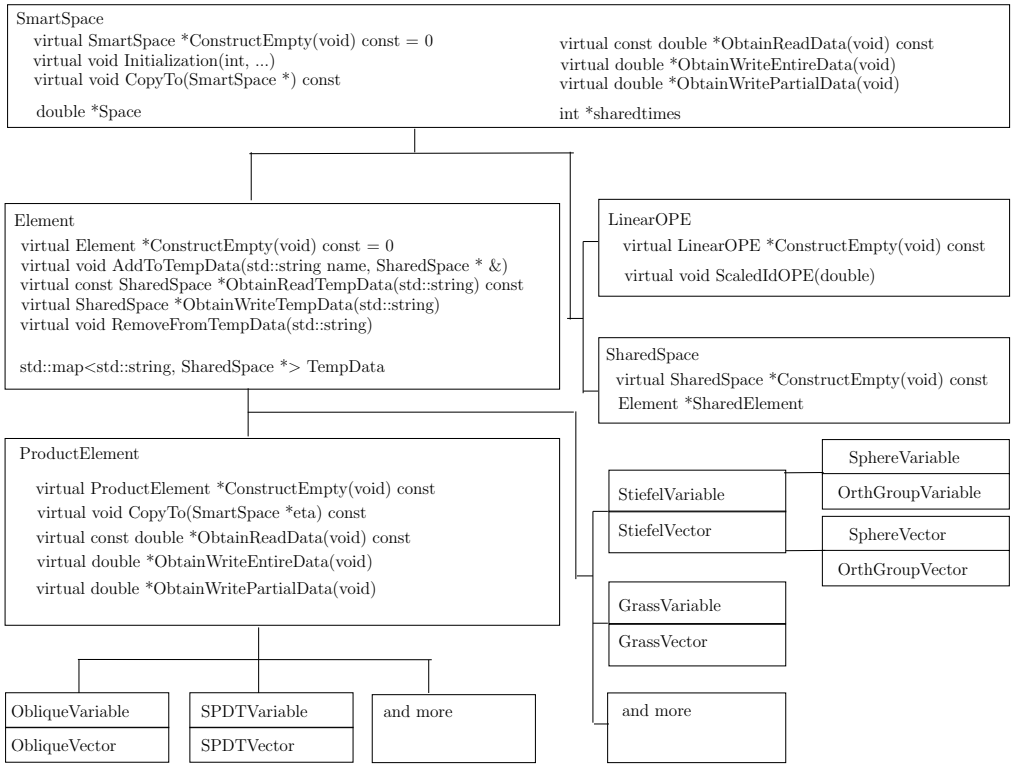

Figure 2: The class hierarchy of space-related classes in ROPTLIB. Note that *Variable* and *Vector* are defined to be *Element*.

and users are not allowed to modify the data. This is the fastest way to access the data but users have the most limited authority. Whereas, the memory functions *ObtainWriteEntireData* and *ObtainWritePartialData* are allowed to access the data and modify them. *ObtainWriteEntireData* may not preserve the old data in the memory and this function is used when users want to completely overwrite the data. *ObtainWritePartialData* guarantees that the memory has the old data. This is the most inefficient approach but it preserves the old data information and is used if users only partially modify the data.

The ability of storing temporary data in a point or a tangent vector of a manifold is important for the efficiency of ROPTLIB in the sense that it avoids redundant computations. For instance, in many problems, the computational cost of the gradient evaluation can be significantly reduced if the cost function evaluation has been done. Since the cost function value and gradient are related to the current iterate, one can attach temporary data onto the current iterate in the function evaluation and reuse this data in the gradient evaluation. See an example in e.g., `/ROPTLIB/Matlab/ForMatlab/testSimpleExample.m`. The *Element* class defines the functionalities of a point and a tangent vector of a manifold. Besides using copy-on-write, an object of this class also has the ability to include temporary data. To this end, ROPTLIB uses an object **TempData** of a map[2], whose key is *string* type and value is *SharedSpace* type which are introduced in the next paragraph. Therefore, string can be used to attach or withdraw specific temporary data. Specifically, the member function *AddToTempData* is used to add a temporary data, and the member functions *ObtainReadTempData* and *ObtainWriteTempData* are used to obtain stored temporary data. An example is given in Secton 3.

The classes *SharedSpace* and *LinearOPE* are derived classes of *SmartSpace*. *SharedSpace* is only used as space for temporary data. One can either store an arbitrary length double array or an *Element* object, which is a point or a tangent vector on the manifold, to an object of *SharedSpace*. Allowing to attach an *Element* object eases implementations in some cases, see Section 3 for an example. *LinearOPE* defines a linear operator on a tangent space and typically is a matrix.

After the class *Element* is defined, a point on any manifold and a tangent vector of any manifold can be defined as a derived class. For example, *StieVariable*, *GrassVariable*, *SPDVariable* are classes for a point on the Stiefel manifold, Grassmann manifold, and the manifold of symmetric positive definite (SPD) matrices, respectively, and *StiefelVector*, *GrassVector*, *SPDVector* are classes for a tangent vector of those three manifolds, respectively. For more manifolds, we refer to the user manual [HAGH16].

ROPTLIB defines *ProductElement* class, which can be used as a point on a product manifold or a tangent vector of a product manifold. This class re-implements a few member functions of *Element* to guarantee that the space of elements are consecutive. The motivation of this approach is to utilize the principle of locality and improve efficiency. Some products of manifolds are implemented in ROPTLIB, such as the Oblique manifold, (or equivalently the product of unit spheres) and the SPD tensors (or equivalently the product of SPD manifolds).

Note that the function *ConstructEmpty* is declared in *SmartSpace* and reimplemented in all the derived classes. This function makes use of the polymorphism and gives a virtual constructor for all the space-related classes.

[2]It is a container class, see details in `http://www.cplusplus.com/reference/map/map/`.

2.2 Manifold Classes

A manifold-related class defines all operations that are only related to a Riemannian manifold, such as Riemannian metric, retrection, and vector transport. ROPTLIB has supplied many commonly-used manifolds. In addition, a user-friendly interface is also provided for advanced users in case they would like to define their own manifolds. The base class of all the manifold-related classes includes the prototypes of all the necessary operations on a manifold. Figure 3 shows the hierarchy of the manifold-related classes and some important prototypes of functions in the base class *Manifold*. The functions are all virtual since the ability to automatically choose the manifold class in a solver requires treating manifold-related classes polymorphically. The functions can be classified into three groups. The first group functions, which are in the solid box, must be overridden in a derived class of a specific manifold in general. Note that for the non-pure virtual functions we have provided default implementations which are operations for the Euclidean manifold. If an operation on a manifold is the same as the operation on the Euclidean space, then the function that defines such a operation does not need be overridden. Otherwise, the function must be overridden.

The functions in the second and the third groups do not need be overridden generally. The second group functions, in the dotted box, define a vector transport satisfying the locking condition, (see [HGA15] for the definition and the use of the locking condition), which is done by invoking the vector transport related functions in the first group. The third group functions, in the dashed box, use the properties of operations on a manifold to verify whether the first group functions, which may be overridden by users, are correct or not. For example, a retraction on a manifold satisfies

$$\frac{d}{dt}R_x(t\eta_x)|_{t=0} = \eta,$$

where $x \in \mathcal{M}$ and $\eta_x \in \mathrm{T}_x\,\mathcal{M}$. The function *CheckRetraction* compares η and $(R_x(\delta\eta_x) - R_x(0\eta_x))/\delta$ for a small value of δ. We refer to the documentation in the code in [HAGH16] for details.

ROPTLIB is the first package that emphasizes the efficiency of quasi-Newton on Riemannian manifolds. Unlike Euclidean quasi-Newton methods, Riemannian quasi-Newton methods usually have extra costs on the implementations of vector transports. Specifically, suppose $\mathcal{B}_k$ is a Hessian approximation at the iterate x_k. In order to obtain a Hessian approximation at the next iterate x_{k+1}, one has to compute the composition $\mathcal{T}_k \circ \mathcal{B}_k \circ \mathcal{T}_k^{-1}$, where $\mathcal{T}_k$ is a vector transport from $\mathrm{T}_{x_k}\,\mathcal{M}$ the tangent space at x_k to $\mathrm{T}_{x_{k+1}}\,\mathcal{M}$ the tangent space at x_{k+1}. This composition involves matrix multiplications in general and often dominates the cost of the entire algorithm. A recent result shows that a vector transport is essentially an identity, which is the cheapest one can expect, if a particular approach is used to represent a tangent vector (see [HAG15, Section 2.2] or [HAG16b, Section 3] for details). ROPTLIB follows the ideas in the papers and gives an efficient implementation. Specifically, the function *ObtainIntr* (*ObtainExtr*) is the prototype that converts a tangent vector from the ordinary (resp. efficient) representation to the efficient (resp. ordinary) representation. To the best of our knowledge, this has not been done by any existing Riemannian optimization packages. The improvements on vector transport benefit all algorithms that involve vector transport, including Riemannian conjugate gradient methods and limited-memory Riemannian quasi-Newton methods.

2.3 Problem Classes

A Riemannian optimization problem can be characterized by a cost function evaluation, a gradient evaluation, an action of a Hessian evaluation (if Newton method is used) and the domain of the

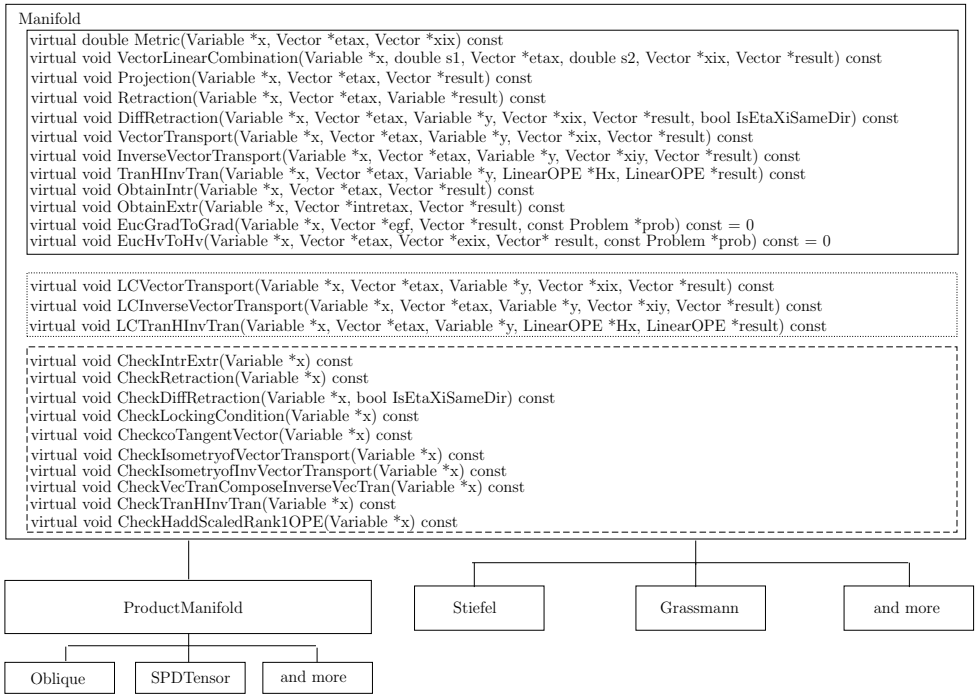

Figure 3: The class hierarchy of manifold-related classes in ROPTLIB. We refer to the documentations in the code for the detailed explanations of the functions.

cost function. The base class *Problem* includes the prototype of the above objects and another function which is used to check if the overridden functions are correct.

In order to define a problem in ROPTLIB, one needs to give a derived class of the base class *Problem*. Figure 4 shows the hierarchy of problem-related classes as well as some prototypes of the base class *Problem*. We once again define the prototypes in *Problem* as virtual functions since polymorphism is necessary for the same reason as manifold-related and space-related classes. The cost function f is declared as a pure virtual function, which must be overridden in derived classes. The function *Grad* calls the function *RieGrad* and may or may not represent the gradient obtained using the efficient representation based on given parameters. Users are able to define a gradient evaluation by overriding the function *RieGrad*, which is the Riemannian gradient. Overriding the function *RieGrad* requires users to have a background in Riemannian manifolds. Therefore, we also provide another approach, which is to override the function *EucGrad*. In this case, the function *EucGradtoGrad*, which has been implemented in *Manifold*, automatically converts the resulting Euclidean gradient to the Riemannian gradient. The implementation for action of Hessian is similar to the gradient evaluation, i.e., one of functions *RieHessianEta* and *EucHessianEta* must be overridden if second order information is necessary for the used Riemannian algorithm. The function *CheckGradHessian* is used to test whether the function evaluation, gradient evaluation, and the action of the Hessian, which are overridden by users, are correct in the sense that the Taylor's expansion

$$f_x(R_x(\eta_x)) = f(x) + \langle \operatorname{grad} f(x), \eta_x \rangle + \frac{1}{2}\langle \operatorname{Hess} f(x)[\eta_x], \eta_x \rangle + O(\|\eta_x\|^3).$$

hold, where either the retraction R is a second-order retraction or x is a critical point of f, see details in the user manual [HAGH16]. The pointer *Domain* specifies the domain of the cost function f.

The *mexProblem* class defines a problem for the Matlab interface. Specifically, the member functions of *mexProblem* call the function handles given by Matlab. The member variables **mxf**, **mxgf**, and **mxHess** are Matlab function handles of a cost function evaluation, gradient evaluation, and action of Hessian. Since ROPTLIB and Matlab use different data structures (*Element* for ROPTLIB and *mxArray* for Matlab), we give functions, *ObtainMxArrayFromElement* and *ObtainElementFromMxArray*, to convert from one data structure to the other. It follows that the work flow is, in gradient evaluation for example, i) convert an iterate from *Element* to *mxArray*, ii) call the Matlab function handle **mxgf**, and iii) convert the obtained gradient from *mxArray* to *Element* and return.

Similarly, the *juliaProblem* class defines a problem for the Julia interface. It uses the same work flow as in *mexProblem*. Since it is straightforward to convert the data structures between ROPTLIB and Julia, unlike in *mexProblem* we do not implement such functions in *juliaProblem*.

2.4 Solver Classes

The state-of-the-art Riemannian optimization algorithms listed in Table 1 are included in ROPTLIB. We design the hierarchy of solver-related classes based on their similarities and differences. The details are shown in Figure 5.

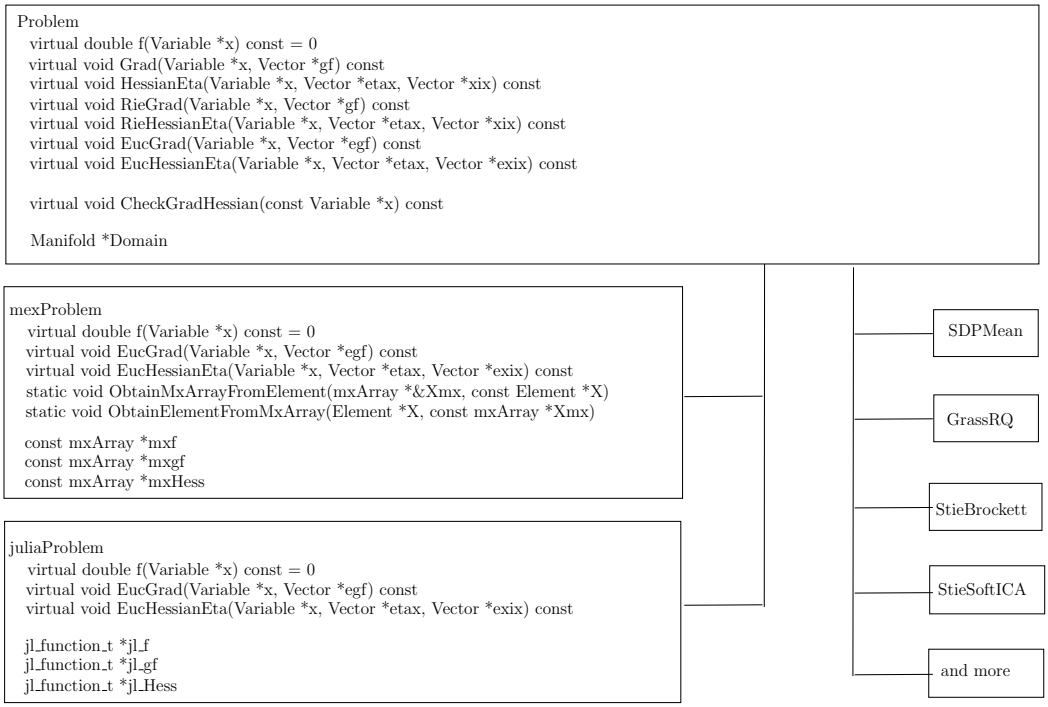

Figure 4: The class hierarchy of problem-related classes in ROPTLIB. We refer to the documentations in the code for the detailed explanations of the functions.

Table 1: Riemannian algorithms in ROPTLIB

Riemannian trust-region Newton (RTRNewton)	[ABG07]
Riemannian trust-region symmetric rank-one update (RTRSR1)	[HAG15]
Limited-memory RTRSR1 (LRTRSR1)	[HAG15]
Riemannian trust-region steepest descent (RTRSD)	[AMS08]
Riemannian line-search Newton (RNewton)	[AMS08]
Riemannian Broyden family (RBroydenFamily)	[HGA15]
Riemannian BFGS (RWRBFGS and RBFGS)	[RW12] [HGA15]
Subgradient Riemannian (L)BFGS ((L)RBFGSLPSub)	[HHY16]
Limited-memory RBFGS (LRBFGS)	[HGA15]
Riemannian conjugate gradients (RCG)	[NW06] [AMS08] [SI13]
Riemannian steepest descent (RSD)	[AMS08]
Riemannian gradient sampling (RGS)	[Hua13] [HU16]

The base class *Solvers* extracts the common points of all the Riemannian methods. We categorize its members into four groups. The functions in the first group define the general behaviors of initializations of all algorithms. The functions in the second group are used during iterations, such as checking whether a stopping criterion is satisfied and printing iteration information. The functions in the third group get optimization results, and the member variables in the fourth group are needed for all the Riemannian methods.

The *QuasiNewton* class defines the updates for Hessian approximations, or equivalently preconditioners, and their actions. Specifically, a gradient-based iterative algorithm has a line-search iteration:

$$x_{k+1} = R_{x_k}(-\alpha_k \mathcal{H}_k \operatorname{grad} f(x_k)),$$

where α_k is a step size and $\mathcal{H}_k$ is an inverse Hessian approximation, or a trust-region iteration:

$$x_{k+1} = R_{x_k}(\eta_{x_k}),$$

where $\eta_{x_k} = \arg\min_{s \in \mathrm{T}_{x_k}\mathcal{M}}$ and $_{\|s\| \le \delta} \langle \operatorname{grad} f(x_k), s\rangle + \frac{1}{2}\langle s, \mathcal{B}_k s\rangle$, and $\mathcal{B}_k$ is a Hessian approximation. This class *QuasiNewton* defines the commonly-used update formulas for $\mathcal{H}_k$ and $\mathcal{B}_k$ and also defines their actions $\mathcal{H}_k\xi_k$ and $\mathcal{B}_k\xi_k$ for any $\xi_k \in \mathrm{T}_{x_k}\mathcal{M}$. Therefore, all the derived classes of *QuasiNewton* have the ability to choose any of the preconditioners implemented.

Since all the iterative optimization methods can be categorized into either line-search-based or trust-region-based methods, we define two classes derived from *QuasiNewton*. One is *SolversLS* and the other is *SolversTR*. The former defines the base class for all the line-search-based algorithms. Specifically, the functions in this class define the general procedure of line-search-based iterations and the commonly-used algorithms for finding a step size. For smooth cost functions, the sophisticated line search algorithms predicated on polynomial interpolation are included (see [DS83, Algorithms A6.3.1 and A6.3.1mod] and [NW06, Algorithm 3.5] for details). For Lipschitz continuous functions, a state-of-the-art line search algorithm [You16, Algorithm 1] is also contained. The latter class, *SolversTR*, is the base class for all trust-region-based methods. Therein, besides defining the general procedure for trust-region-based iterations, a function to approximately solve a local quadratic model is also defined (see [AMS08, Algorithm 11] for details). Due to object-oriented programming, all the derived classes are able to use the functions in the base classes,

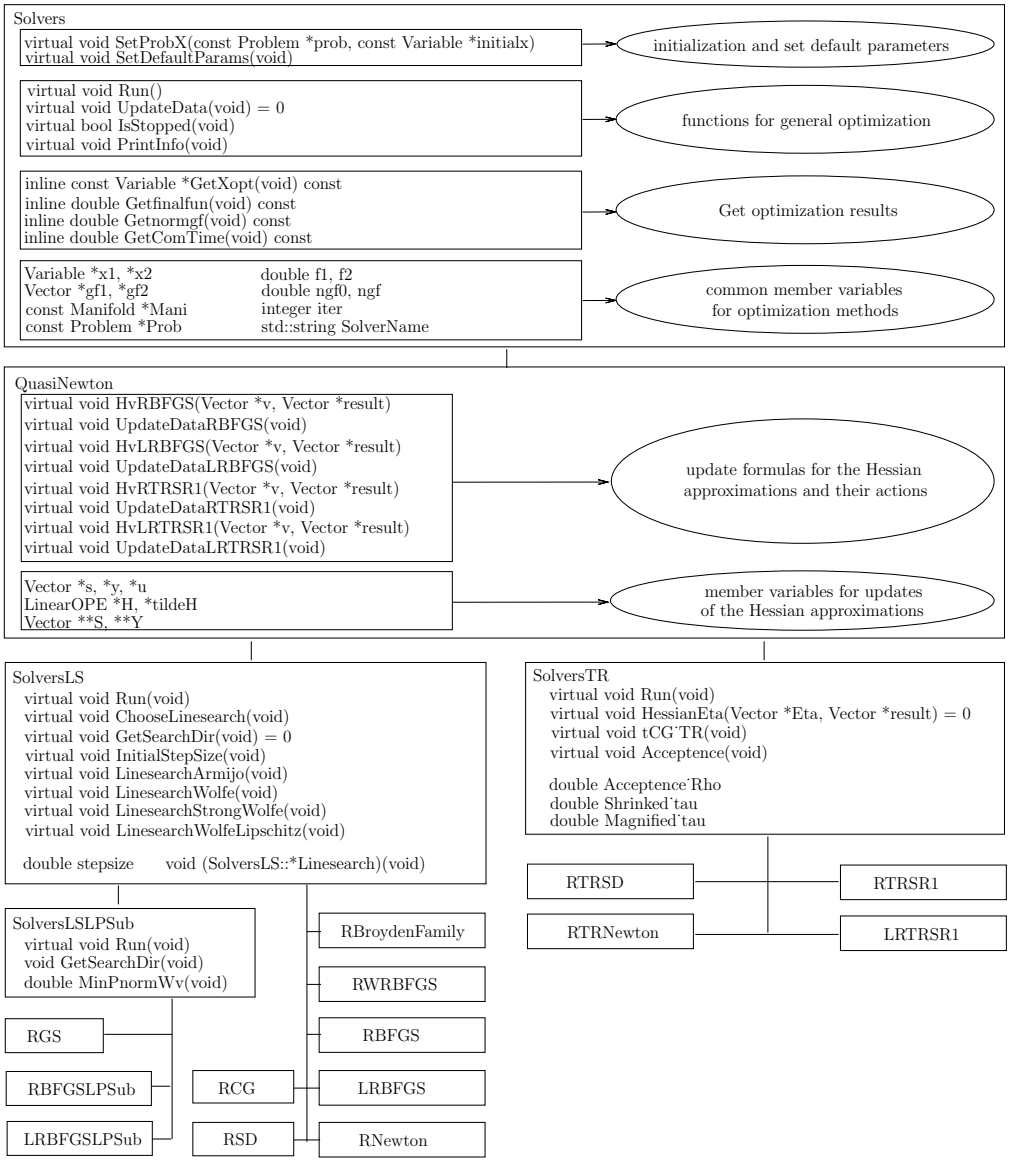

Figure 5: The class hierarchy of solver-related classes in ROPTLIB. We refer to the documentations in the code for the detailed explanations of the functions.

which increases the extendability and reusability of the codes and allows users/us to define new algorithms easily.

By combining a Hessian approximation update formula in class *QuasiNewton* and a line-search strategy or a trust-region strategy, we define state-of-the-art Riemanian optimization algorithms. Note that the subgradient-based algorithms, RGS, RBFGSLPSub and LRBFGSLPSub for nonsmooth optimization, need subgradients or need to approximate subgradients. Therefore, they have common behaviors that do not exist in *SolversLS*. We extract those common points and define a class *SolversLSLPSub*, which is a derived class of *SolversLS*. Class *SolversLSLPSub* redefines the function *GetSearchDir* since it is different from optimization for smooth cost functions in the sense that the search direction requires the estimation of a subgradient by computing the shortest vector in the convex hull of a few given vectors.

3 An Example

To illustrate some of the concepts, we present an example using ROPTLIB. The problem is to minimize the Brockett cost function [AMS08, Section 4.8] on the Stiefel manifold $\mathrm{St}(p,n) = \{X \in \mathbb{R}^{n\times p} | X^TX = I_p\}$

$$\min_{X\in\mathrm{St}(p,n)} \mathrm{trace}(X^TBXD) \tag{3.1}$$

where $B \in \mathbb{R}^{n\times n}$, $B = B^T$, $D = \mathrm{diag}(\mu_1, \mu_2, \ldots, \mu_p)$ and $\mu_1 > \mu_2 > \ldots > \mu_p$. It is known that X^* is a global minimizer if and only if its columns are eigenvectors of B for the p smallest eigenvalues, λ_i, ordered so that $\lambda_1 \leq \cdots \leq \lambda_p$ [AMS08, Section 4.8].

Instructions about compiling the code can be found in the user manual [HAGH16].

Listings 1, 2, and 3 give examples for the Brockett cost function 3.1 in C++, Matlab, and Julia environments respectively. [3] The basic steps to set up the problem and run a Riemannian optimization algorithm in all the environments are the same:

1. Define Problem 3.1 by implementing the cost function, the gradient, and action of the Hessian; (An object of a derived class of the base class *Problem*)

2. generate a Stiefel manifold, which is the domain of Problem 3.1; (An object of a derived class of the base class *Manifold*)

3. generate a point on the Stiefel manifold as an initial iterate; (An object of a derived class of the base class *Element*) and

4. choose an algorithm, specify its parameters and run the algorithm with inputs being the problem and the initial iterate. (An object of a Riemannian optimization algorithm and execute the function *Run*.)

More details of the commands in the Listings can be found in their corresponding comments.

[3] Listing 1, Listing 2 and Listing 3 are available in "/ROPTLIB/test/TestSimpleExample.cpp+", "/ROPTLIB/-Matlab/ForMatlab/testSimpleExample.m", and "/ROPTLIB/Julia/JTestSimpleExample.jl" respectively. The code in the file may not be exactly the same as that in the Listings. The code in the file tests more parameters and runs more/different algorithms. However, the differences are minor and should not cause confusion.

Listing 1:

```
// File: TestSimpleExample.cpp
#ifndef TESTSIMPLEEXAMPLE_CPP
#define TESTSIMPLEEXAMPLE_CPP
#include "StieBrockett.h"
#include "StieVector.h"
#include "StieVariable.h"
#include "Stiefel.h"
#include "RTRNewton.h"
#include "def.h"
using namespace ROPTLIB;
#ifdef TESTSIMPLEEXAMPLE

int main(void)
{
        init_genrand((unsigned) time(NULL));   // choose a random seed
        integer n = 12, p = 8;                 // size of the Stiefel manifold
        // Generate the matrices in the Brockett problem.
        double *B = new double[n * n + p];
        double *D = B + n * n;
        for (integer i = 0; i < n; i++)
        {
                for (integer j = i; j < n; j++)
                {
                        B[i + j * n] = genrand_gaussian();
                        B[j + i * n] = B[i + j * n];
                }
        }
        for (integer i = 0; i < p; i++)
                D[i] = static_cast<double> (i + 1);
        StieBrockett Prob(B, D, n, p);         // Define the Brockett problem
        Stiefel Domain(n, p);                  // Define the Stiefel manifold
        Prob.SetDomain(&Domain);               // Set the domain
        Domain.CheckParams();   // output the parameters of the manifold of domain
        StieVariable StieX(n, p);              // Obtain an initial iterate
        StieX.RandInManifold();
        // test RTRNewton
        std::cout << "*********Check RTRNewton**********" << std::endl;
        RTRNewton RTRNewtonsolver(&Prob, &StieX);
        RTRNewtonsolver.DEBUG = FINALRESULT;
        RTRNewtonsolver.CheckParams();
        RTRNewtonsolver.Run();
        // Check gradient and Hessian
        Prob.CheckGradHessian(&StieX);
        const Variable *xopt = RTRNewtonsolver.GetXopt();
        Prob.CheckGradHessian(xopt);
        // output the minimizer to the screen.
        xopt->Print("Minimizer is:");
        delete[] B;
        return 0;
}
#endif
#endif
```

Listing 2:

```
function [FinalX, fv, gfv, gfgf0, iter, nf, ng, nR, nV, nVp, nH, ComTime, funs, grads, times
    ] = testBrockett()
  n = 5; p = 2;                      % size of the Stiefel manifold
  B = randn(n, n); B = B + B';       % data matrix
  D = sparse(diag(p : -1 : 1));      % data matrix
  fhandle = @(x)f(x, B, D);          % cost function handle
  gfhandle = @(x)gf(x, B, D);        % gradient
  Hesshandle = @(x, eta)Hess(x, eta, B, D); % Hessian

```

```
    ManiParams.name = 'Stiefel';      % Domain is the Stiefel manifold
    ManiParams.n = n;                 % assign size to manifold parameter
    ManiParams.p = p;                 % assign size to manifold parameter
    ManiParams.IsCheckParams = 1;     % output all the parameters of this manifold

    initialX.main = orth(randn(n, p));% initial iterate

    SolverParams.method = 'RSD';      % Use RSD solver
    SolverParams.LineSearch_LS = 0;   % Back tracking for Armijo condition
    SolverParams.IsCheckParams = 1;   % output all the parameters of this solver
    SolverParams.IsCheckGradHess = 1; % Check the correctness of grad and Hess

    HasHHR = 0;                       % locking condition is not guaranteed.
    % call the driver
    [FinalX, fv, gfv, gfgf0, iter, nf, ng, nR, nV, nVp, nH, ComTime, funs, grads, times] =
        DriverOPT(fhandle, gfhandle, Hesshandle, SolverParams, ManiParams, HasHHR, initialX);
end
```

Listing 3:

```
# set function handles
fname = "func"; gfname = "gfunc"; hfname = "hfunc";
isstopped = ""; LinesearchInput = ""; # no given linesearch algorithm and stopping criterion
    .
Handles = FunHandles(pointer(fname), pointer(gfname), pointer(hfname), pointer(isstopped),
    pointer(LinesearchInput))

# set domain manifold to be the Stiefel manifold St(3, 5).
mani1 = "Stiefel";ManArr = [pointer(mani1)] # Domain is the Stiefel manifold
UseDefaultArr = [-1] # -1 means that the default value in C++ is used.
numofmani = [1] # The power of this manifold is 1.
ns = [5]; ps = [3]; # Size of the Stiefel manifold is 5 by 3.
IsCheckParams = 1;
Mparams = ManiParams(IsCheckParams, length(ManArr), pointer(ManArr), pointer(numofmani),
    pointer(paramsets), pointer(UseDefaultArr), pointer(ns), pointer(ps))

# Initial iterate and problem
n = ns[1];p = ps[1];
B = randn(n, n); B = B + B'; # data matrix
D = sparse(diagm(linspace(p, 1, p))) # data matrix
initialX = qr(randn(ns[1], ps[1]))[1] # initial iterate

# A default solver-related parameter has been defined, we only need to modify it.
method = "RTRNewton" # set a solver by modifying  the default one
Sparams.IsCheckParams = 1
Sparams.name = pointer(method)
Sparams.LineSearch_LS = 1 # Backing tracking for Armijo condition
Sparams.IsCheckGradHess = 1 # Check the correctness of grad and Hess

HasHHR = 0 # The locking condition is not guaranteed.

# Call the solver and get results. See the user manual for details about the outputs.
(FinalIterate, fv, gfv, gfgf0, iter, nf, ng, nR, nV, nVp, nH, ComTime, funs, grads, times) =
     DriverJuliaOPT(Handles, Sparams, Mparams, HasHHR, initialX)
```

4 Applications

In this section, two important applications—dictionary learning for symmetric positive definite matrices and the matrix completion problem—are used to show the performance of ROPTLIB. To see the importance of the two applications, we refer to [HSHL12, LWZZ13, SBMP14, CS15] for the dictionary learning problem and [Van13, Mis14] for the matrix completion problem. Note that a

Riemannian optimization algorithm, Riemannian nonlinear conjugate gradient method, has been used for solving these two applications [Van13, CS15].

Note that ROPTLIB has been used to solve optimization problems in many other applications such as the phase retrieval problem [HGZ17], optimization problems in elastic shape analysis [HGSA15, YHGA15], finding an geometric mean of symmetric positive definite matrices [YHAG17], role model extraction [MHB+16], and multi-input multi-output waveform optimization for synthetic aperture sonar [MHGM16].

The codes for the applications are lengthy, therefore not included in this paper. They can be found on ROPTLIB's webpage [HAGH16].

All the experiments are performed in Matlab R2016b on a 64 bit Windows platform with 3.4 GHz CPU (Intel(R) Core (TM) i7-6700). The code can be found at [HAGH17].

4.1 Dictionary Learning for Symmetric Positive Definite Matrices

It is pointed out that the dictionary learning problem usually comes with the sparse coding problem. For symmetric positive definite (SPD) matrices, the optimization problem in dictionary learning for positive definite matrices with sparse coding (DLSC) given by [CS15] is defined to be

$$\min_{\mathbf{B}\in\mathbb{S}_d^n, R\in\mathbb{R}_+^{n\times N}} \frac{1}{2}\sum_{j=1}^{N}\left(\left\|\log\left(X_j^{-1/2}(\mathbf{B}r_j)X_j^{-1/2}\right)\right\|_F^2 + \lambda_R\sum_{i=0}^{n} r_{ij}\right) + \lambda_{\mathbf{B}}\operatorname{trace}(\mathbf{B}), \tag{4.1}$$

where $\mathbf{B}$ is a dictionary, $R = (r_{ij}) = \begin{bmatrix} r_1 & \dots & r_N \end{bmatrix}$ is a sparse code, $\mathbb{S}_d$ denotes the manifold of d-by-d SPD matrices, $\mathbb{S}_d^n$ denotes the product of n manifolds of $\mathbb{S}_d$, $\mathbb{R}_+^{n\times N}$ denotes the set of n by N matrices with entries nonnegative, λ_R and $\lambda_{\mathbf{B}}$ are positive constants, $\operatorname{trace}(\mathbf{B}) = \sum_{i=1}^{n}\operatorname{trace}(B_i)$, and B_i is i-th slice of the tensor $\mathbf{B}$.

Alternating descent method, which alternates between solving the dictionary learning and sparse coding subproblems, is a popular method for solving DLSC problems and has been used [CS15]. In this section, we focus on the dictionary learning problem since it is defined on a manifold $\mathbb{S}_d^n$.

The cost function and gradient of the dictionary learning problem have been given in [CS15]. We give them here for completeness. The cost function is

$$f : \mathbb{S}_d^n \to \mathbb{R} : \mathbf{B} \mapsto \frac{1}{2}\sum_{j=1}^{N}\left\|\log\left(X_j^{-1/2}(\mathbf{B}r_j)X_j^{-1/2}\right)\right\|_F^2 + \lambda_{\mathbf{B}}\operatorname{trace}(\mathbf{B}), \tag{4.2}$$

which is from (4.1) by fixing the sparse codes R, and the Euclidean gradient is $\nabla_{\mathbf{B}} f(\mathbf{B}) = \nabla_{B_1} f(\mathbf{B}) \times \ldots \times \nabla_{B_n} f(\mathbf{B})$, where

$$\nabla_{B_i} f(\mathbf{B}) = \sum_{j=1}^{N} r_{ij} X_j^{-1/2}\log\left(X_j^{-1/2}(\mathbf{B}r_j)X_j^{-1/2}\right)X_j^{1/2}(\mathbf{B}r_j)^{-1} + \lambda_{\mathbf{B}} I. \tag{4.3}$$

LRBFGS and RCG are used to test the performance of ROPTLIB for this problem. The parameters in all the tested algorithms are the default choices in ROPTLIB. Note that the default stopping criterion requires the norm of the gradient to reduce by a factor of at least 10^{-6}. Parameters $\lambda_{\mathbf{B}}$ is set to 10^{-3}. Synthetic data are used and generated as follows. All the slices of the tensor $\mathbf{B} \in \mathbb{S}_d^n$ are given by $B_i = W_i^T W_i, i = 1, \ldots, n$, where $W_i \in \mathbb{R}^{(10d)\times d}$ and entries of W_i are drawn

Table 2: Notation for reporting the experimental results.

iter	number of iterations
nf	number of function evaluations
ng	number of gradient evaluations
nR	number of retraction evaluations
nV	number of vector transport
nH	number of action of Hessian
gf/gf0	$\|\operatorname{grad} f(x_k)\|/\|\operatorname{grad} f(x_0)\|$
t	average wall time (seconds)
err	relative error $\|x_k - x_*\|_F/\|x_*\|_F$

from the standard normal distribution. The number of active atoms in the dictionary $\mathbf{B}$ are the same for all the training data X_j and is denoted by κ. Every training datum X_j is generated by a linear combination of κ atoms randomly chosen from the dictionary and the coefficients of the atoms are drawn from the uniform distribution on $[0, 1]$. The matrix R in the experiments is the true sparse code, i.e., R satisfies $\mathbf{B}r_i = X_i$ for all i.

The notation used in the later tables is given in Table 2.

Initial iterates are important for the performance of the algorithms. To obtain an initial iterate for the dictionary learning problem, we first denote $\mathfrak{X}$ and $\mathfrak{B}$ by the matrix $\begin{bmatrix}\operatorname{vec}(X_1) & \dots & \operatorname{vec}(X_n)\end{bmatrix}$ and $\begin{bmatrix}\operatorname{vec}(B_1) & \dots & \operatorname{vec}(V_n)\end{bmatrix}$ respectively, where $\operatorname{vec}(M)$ denotes the vector by stacking the columns of M. The dictionary learning problem attempts to find a dictionary $\mathbf{B}$ such that $\mathfrak{X} = \mathfrak{B}R$. Therefore, the proposed initial iterate is given by $\mathfrak{B} = \mathfrak{X}(R^\dagger)_+$, where $\dagger$ denotes the pseudo-inverse operator and $(M)_+$ denotes the matrix given by nonnegative entries of M.

Tables 3 and 4 report the comparisons of LRBFGS and RCG with various sizes of d and n for the dictionary learning problem. In particular, Table 3 presents an average of 50 random runs of LRBFGS and RCG with $N = 100$, $n = 20$, $\kappa = 6$ and various d, and Table 4 presents an average of 50 random runs of LRBFGS and RCG for the dictionary learning subproblem with $N = 500$, $d = 4$, $\kappa = 0.3n$ and various n. Both LRBFGS and RCG methods work very well for this problem and are able to find the true dictionary up to the bias caused by the penalty $\lambda_{\mathbf{B}} \operatorname{trace}(\mathbf{B})$. The number of operations, i.e., function evaluations, gradient evaluations, etc, required by LRBFGS is less than those required by RCG and LRBFGS needs less computational time.

4.2 The Matrix Completion Problem

Among several available frameworks to address the low-rank matrix completion problem, we consider the one proposed in [Van13]:

$$\min_X f(X) := \frac{1}{2}\|P_\Omega(X) - P_\Omega(A)\|_F^2,$$
$$\text{subject to } X \in \mathcal{M}_k := \{X \in \mathbb{R}^{m\times n} : \operatorname{rank}(X) = k\},$$

where

$$P_\Omega : \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n} : X_{i,j} \mapsto \begin{cases} X_{i,j}, & \text{if } (i,j) \in \Omega; \\ 0, & \text{if } (i,j) \notin \Omega, \end{cases}$$

Table 3: An average of 50 random runs of (i) LRBFGS and (ii) RCG for the dictionary learning subproblem with $N = 100$, $n = 20$, $\kappa = 6$ and various d. The subscript $-k$ indicates a scale of 10^{-k}.

	$d = 4$		$d = 8$		$d = 12$		$d = 16$		$d = 20$	
	(i)	(ii)	(i)	(ii)	(i)	(ii)	(i)	(ii)	(i)	(ii)
iter	41	51	41	52	44	52	45	55	46	56
nf	44	75	44	76	46	77	49	80	49	82
ng	42	52	42	53	45	53	46	56	47	57
nR	43	74	43	75	45	76	48	79	48	81
nV	314	103	315	105	339	105	351	110	353	112
gf/gf0	7.88_{-7}	7.81_{-7}	7.70_{-7}	7.56_{-7}	7.37_{-7}	7.63_{-7}	7.80_{-7}	7.39_{-7}	7.65_{-7}	7.53_{-7}
t	2.62_{-2}	3.82_{-2}	5.51_{-2}	8.63_{-2}	9.68_{-2}	1.50_{-1}	1.53_{-1}	2.37_{-1}	2.34_{-1}	3.66_{-1}
err	1.58_{-2}	1.58_{-2}	2.88_{-2}	2.88_{-2}	4.51_{-2}	4.51_{-2}	5.73_{-2}	5.73_{-2}	6.87_{-2}	6.87_{-2}

Table 4: An average of 50 random runs of (i) LRBFGS and (ii) RCG for the dictionary learning subproblem with $N = 500$, $d = 4$, $\kappa = 0.3n$ and various n.

	$n = 10$		$n = 20$		$n = 30$		$n = 40$		$n = 50$	
	(i)	(ii)	(i)	(ii)	(i)	(ii)	(i)	(ii)	(i)	(ii)
iter	19	35	34	46	43	59	51	64	62	76
nf	21	44	37	65	47	87	54	98	67	125
ng	20	36	35	47	44	60	52	65	63	77
nR	20	43	36	64	46	86	53	97	66	124
nV	138	69	260	92	332	118	394	128	487	153
gf/gf0	6.26_{-7}	6.35_{-7}	7.09_{-7}	7.58_{-7}	7.61_{-7}	7.63_{-7}	7.97_{-7}	7.79_{-7}	7.63_{-7}	7.89_{-7}
t	4.96_{-2}	1.00_{-1}	9.08_{-2}	1.47_{-1}	1.15_{-1}	1.97_{-1}	1.37_{-1}	2.28_{-1}	1.77_{-1}	2.89_{-1}
err	9.81_{-4}	9.81_{-4}	2.06_{-3}	2.06_{-3}	3.35_{-3}	3.35_{-3}	5.09_{-3}	5.08_{-3}	7.19_{-3}	7.19_{-3}

Table 5: An average of 50 random runs of (i) LRBFGS, (ii) LRTRSR1, (iii) RCG, (iv) RNewton, and (v) RTRNewton methods.

	$m = 100, n = 200, k = 10$					$m = 1000, n = 2000, r = 10$				
	(i)	(ii)	(iii)	(iv)	(v)	(i)	(ii)	(iii)	(iv)	(v)
iter	34	46	40	12	13	57	78	67	19	18
nf	37	47	53	14	14	61	79	99	23	19
ng	35	47	41	13	14	58	79	68	20	19
nR	36	46	52	13	13	60	78	98	22	18
nV	257	46	81	0	0	445	78	135	0	0
nH	0	147	0	64	58	0	241	0	108	94
gf/gf0	6.72_{-7}	6.81_{-7}	7.38_{-7}	8.72_{-8}	4.71_{-8}	7.59_{-7}	7.61_{-7}	7.59_{-7}	8.33_{-8}	1.32_{-7}
t	2.84_{-2}	4.28_{-2}	3.24_{-2}	7.68_{-2}	7.17_{-2}	4.25_{-1}	6.16_{-1}	5.27_{-1}	1.34	1.17
f	1.59_{-8}	1.31_{-8}	1.29_{-8}	7.53_{-10}	5.53_{-10}	1.49_{-6}	1.81_{-6}	1.21_{-6}	6.02_{-8}	1.23_{-7}
err	3.12_{-6}	2.72_{-6}	2.55_{-6}	2.56_{-7}	1.35_{-7}	1.63_{-5}	2.01_{-5}	1.51_{-5}	1.24_{-6}	1.73_{-6}

Ω is a given index set, and $P_\Omega(A)$ is a given sparse matrix. Its Euclidean gradient is

$$\operatorname{grad} f(X) = P_\Omega(X - A),$$

and the action of the Euclidean Hessian along direction η is

$$\operatorname{Hess} f(x)[\eta] = P_\Omega(\eta).$$

Since it is well-known that $\mathcal{M}_k$ is a manifold, ROPTLIB can be used to solve this problem.

The matrix A is generated by GH^T, where $G \in \mathbb{R}^{m\times k}$, $H \in \mathbb{R}^{n\times k}$, and the entries of G and H are drawn from the standard normal distribution. The number of entries in Ω is fixed to be $\tau = 3(m + n - k)k$. Note that $(m + n - k)k$ is the dimension of the manifold $\mathcal{M}_k$. The set Ω is given by the first τ entries of the random permutation vector with size mn. The initial iterate is generated by $X = UDV^T$, where U, D and V are the k dominated singular vectors and values of $P_\Omega(A)$.

The LRBFGS, LRTRSR1, RCG, RNewton, and RTRNewton solvers are tested. The default parameters in ROPTLIB are used. Table 5 reports an average of 50 random runs of the five algorithms. All the algorithms always managed to reduce the norm of the gradient by a factor at least 10^{-6}, which is the default stopping criterion. In addition, all the algorithms are also able to find the true solution in the sense that the relative error is always less than 10^{-4}. The LRBFGS is the best one among them in terms of computational time.

5 Benchmark

We use a joint diagonalization problem on the Stiefel manifold to show a benchmark of efficiency for ROPTLIB, Pymanopt, and Manopt. The joint diagonalization problem considers minimizing an objective function defined as

$$f : \operatorname{St}(p, n) \to \mathbb{R} : X \mapsto f(X) = -\sum_{i=1}^{N} \| \operatorname{diag}(X^T C_i X)\|^2,$$

where $\operatorname{St}(p, n) = \{X \in \mathbb{R}^{n\times p} \mid X^T X = I\}$ denotes the Stiefel manifold, $C_1, \ldots, C_N$ are given symmetric matrices, $\operatorname{diag}(M)$ denotes the vector formed by the diagonal entries of M, and $\| \operatorname{diag}(M)\|^2$

thus denotes the sum of the squared diagonal entries of M. This problem has applications in independent component analysis for blind source separation [TCA09].

All the experiments are performed on a Windows 7 platform with 3.40GHz CPU (Intel(R) Core(TM) i7-6700). The code is available at [HAGH17]. The cost function evaluation and gradient evaluation in ROPTLIB are written in C++, i.e., not using Matlab or Julia interface. To illustrate the performance across the three libraries, we choose the Riemannian steepest descent (RSD) with the backtracking line search algorithm. The number of iteration is fixed to be 30. The Pymanopt implementation was approved by the Pymanopt authors.

An average computational time and the corresponding average number of function evaluations of 50 random runs for multiple values of p, n, and N are given in Table 6. For small size problems, ROPTLIB is faster than Manopt and Pymanopt by a factor of 20 or more. As n and p grow, the factor gradually reduces to approximately 1 for large-scale problem. In order to understand the phenomenon, we point out that i) interpreted languages (Matlab and Python) are much slower than compiled languages (C++) by constant factors, $O(1)$, ii) all three libraries—ROPTLIB, Manopt, and Pymanopt—invoke highly-optimized libraries, i.e., BLAS and LAPACK, and iii) the computational complexity taken in highly-optimized libraries has higher order than a constant factor, i.e., $O(n^2p)$. When the n and p are small, the differences of efficiency between interpreted language and compiled languages is the reason that ROPTLIB is faster than Manopt and Pymanopt by a factor of 20. When n and p get large, the computational time in BLAS and LAPACK starts to dominate the algorithms. Therefore, the factor reduces to approximately 1.

For small-size problems, the improvement in efficiency is crucial if one needs to solve a large number of such problems. For example, interpolation methods on symmetric positive definite (SPD) matrices [AG15], segmentation [PFA06b, BVSF07], and clustering [JHS$^+$13] need to compute many the Karcher means on the manifold of SPD matrices. For large-size problems, ROPTLIB still outperforms Manopt and Pymanopt. It is pointed out here that one source of the differences comes from the use of copy-on-write strategy. Python does not support copy-on-write, which causes pymanopt to have the slowest performance. Matlab supports copy-on-write. However, the copy-on-write in Matlab is different from ROPTLIB. For example, if data in memory is shared by two variables and one wants to overwrite the entire data in one of the variables, Matlab duplicates the data while ROPTLIB only creates memory without copying the data.

Table 6: An average computational time and the corresponding average number of function evaluations of 50 random runs given by ROPTLIB, Manopt, and Pymanopt with gradient provided and Pymanopt using auto-differentiation. Multiple values of p, n, and N are used. t and nf denote computational time (second) and the number of function evaluations.

p, n, N		2, 4, 128	8, 16, 128	32, 64, 32	128, 256, 8	512, 1024, 2	1024, 2048, 2
ROPTLIB	t	0.001	0.004	0.016	0.174	2.989	19.78
	nf	41	42	44	46	47	49
Manopt	t	0.021	0.036	0.078	0.462	4.449	26.27
	nf	41	42	44	46	47	49
Pymanopt(grad)	t	0.014	0.025	0.080	0.438	5.506	36.07
	nf	41	42	44	46	47	49
Pymanopt(auto)	t	0.028	0.047	0.120	0.638	7.554	48.05
	nf	41	42	44	46	47	49

These experiments confirm that Manopt and Pymanopt are competitive (in the sense that the ratio of computation time is between 0.5 and 2) with ROPTLIB only when the computation time is dominated by high-efficiency libraries, such as BLAS and LAPACK.

6 Conclusion and Future Work

In this paper, we described a C++ Riemannian manifold optimization library (ROPTLIB), which makes use of object-oriented programming to ensure the resuability, extensibility, maintainability and understandability of the code. The interfaces for Matlab and Julia are given, which broadens the potential users of ROPTLIB. The experiments shows that ROPTLIB is faster than two state-of-the-art Riemannian optimization packages, Manopt and Pymanopt, by a factor of 10 for small size problems and by a factor of 2 for large size problems. Therefore, it is an efficient library for various scale problems. In the future, more manifolds and new Riemannian algorithms will be added to ensure ROPTLIB remains state-of-the-art and applicable for most applications. We may also integrate C++ new features and pursue parallel or multi-threaded implementations if they further improve the efficiency of ROPTLIB.

Note that ROPTLIB has been developed under appropriate version control and software testing. The released versions with their user manuals and update logs are available at [HAGH16] and the beta version under development is available from github. In order to guarantee the quality, (i) ROPTLIB includes test files for checking if functionalities are working properly and (ii) software such as visual leak detector is used to make sure there is no memory leakage.

References

[ABG07] P.-A. Absil, C. G. Baker, and K. A. Gallivan. Trust-region methods on Riemannian manifolds. *Foundations of Computational Mathematics*, 7(3):303–330, 2007.

[Abr07] T. E. Abrudan. Matlab codes for optimization under unitary matrix constraint, 2007.

[AEK08] T. Abrudan, J. Eriksson, and V. Koivunen. Steepest descent algorithms for optimization under unitary matrix constraint. *IEEE Transaction on Signal Processing*, 56(3):1134–1147, Mar. 2008.

[AEK09] T. Abrudan, J. Eriksson, and V. Koivunen. Conjugate gradient algorithm for optimization under unitary matrix constraint. *Signal Processing (Elsevier)*, 89(9):1704–1714, Sep. 2009.

[AG15] P.-A. Absil and P.-Y. Gousenbourger. Differentiable piecewise-Bezier surfaces on Riemannian manifolds. Technical report, ICTEAM Institute, Universite Catholiqué de Louvain, Louvain-La-Neuve, Belgium, 2015.

[AMS08] P.-A. Absil, R. Mahony, and R. Sepulchre. *Optimization algorithms on matrix manifolds*. Princeton University Press, Princeton, NJ, 2008.

[AW13] K. P. Adragni and S. Wu. Grassmann manifold optimization, 2013.

[BMAS14] N. Boumal, B. Mishra, P.-A. Absil, and R. Sepulchre. Manopt, a Matlab toolbox for optimization on manifolds. *Journal of Machine Learning Research*, 15:1455–1459, 2014.

[BVSF07] Angelos Barmpoutis, Baba C Vemuri, Timothy M Shepherd, and John R Forder. Tensor splines for interpolation and approximation of DT-MRI with applications to segmentation of isolated rat hippocampi. *IEEE transactions on medical imaging*, 26(11):1537–1546, 2007.

[CA16] Léopold Cambier and P.-A Absil. Robust Low-Rank Matrix Completion by Riemannian Optimization. *Siam Journal on Scientific Computing*, 2016. To appear.

[CESV13] E. J. Candès, Y. C. Eldar, T. Strohmer, and V. Voroninski. Phase retrieval via matrix completion. *SIAM Journal on Imaging Sciences*, 6(1):199–225, 2013. arXiv:1109.0573v2.

[CS15] A. Cherian and S. Sra. Riemannian dictionary learning and sparse coding for positive definite matrices. *CoRR*, abs/1507.02772, 2015.

[DS83] J. E. Dennis and R. B. Schnabel. *Numerical methods for unconstrained optimization and nonlinear equations*. Springer, New Jersey, 1983.

[Ehl13] M. Ehler. A C++ library for optimization on Riemannian manifolds, 2013.

[HAG15] W. Huang, P.-A. Absil, and K. A. Gallivan. A Riemannian symmetric rank-one trust-region method. *Mathematical Programming*, 150(2):179–216, February 2015.

[HAG16a] W. Huang, P.-A. Absil, and K. A. Gallivan. A Riemannian BFGS Method for Nonconvex Optimization Problems. *Lecture Notes in Computational Science and Engineering*, 112:627–634, 2016.

[HAG16b] W. Huang, P.-A. Absil, and K. A. Gallivan. Intrinsic representation of tangent vectors and vector transport on matrix manifolds. *Numerische Mathematik*, 136(2):523–543, 2016.

[HAGH16] W. Huang, P.-A. Absil, K. A. Gallivan, and Paul Hand. Riemannian manifold optimization library, 2016.

[HAGH17] W. Huang, P.-A. Absil, K. A. Gallivan, and Paul Hand. Riemannian manifold optimization library for reproducing experiments, 2017.

[HGA15] W. Huang, K. A. Gallivan, and P.-A. Absil. A Broyden Class of Quasi-Newton Methods for Riemannian Optimization. *SIAM Journal on Optimization*, 25(3):1660–1685, 2015.

[HGSA15] W. Huang, K. A. Gallivan, Anuj Srivastava, and P.-A. Absil. Riemannian optimization for registration of curves in elastic shape analysis. *Journal of Mathematical Imaging and Vision*, 54(3):320–343, 2015. DOI:10.1007/s10851-015-0606-8.

[HGZ17] W. Huang, K. A. Gallivan, and X. Zhang. Solving PhaseLift by low rank Riemannian optimization methods for complex semidefinite constraints. *SIAM Journal on Scientific Computing*, 39(5):B840–B859, 2017.

[HHY16] S. Hosseini, Wen Huang, and R. Yousefpour. Line search algorithms for locally Lipschitz functions on Riemannian manifolds. Technical Report INS Preprint No. 1626, Institutfür Numerische Simulation, 2016.

[HS14] S. Hosseini and S. Sra. Geometric optimization toolbox, 2014.

[HSHL12] M. T. Harandi, C. Sanderson, R. Hartley, and B. C. Lovell. Sparse Coding and Dictionary Learning for Symmetric Positive Definite Matrices: A Kernel Approach. *European Conference on Computer Vision*, 2012.

[HU16] S. Hosseini and A. Uschmajew. A Riemannian gradient sampling algorithm for nonsmooth optimization on manifolds. *Institut für Numerische Simulation*, page INS Preprint No. 1607, 2016.

[Hua13] W. Huang. *Optimization algorithms on Riemannian manifolds with applications*. PhD thesis, Florida State University, Department of Mathematics, 2013.

[JHS+13] Sadeep Jayasumana, Richard Hartley, Mathieu Salzmann, Hongdong Li, and Mehrtash Harandi. Kernel methods on the Riemannian manifold of symmetric positive definite matrices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 73–80, 2013.

[JVV12] B. Jeuris, R. Vandebril, and B. Vandereycken. A survey and comparison of contemporary algorithms for computing the matrix geometric mean. *Electronic Transactions on Numerical Analysis*, 39:379–402, 2012.

[KM15] H. Kasai and B. Mishra. Riemannian preconditioning for tensor completion, 2015. arXiv: 1506.02159.

[LLM12] S. B. Lippman, J. Lajoie, and B. E. Moo. *C++ Primer (5th Edition)*. Addison-Wesley Professional, 2012.

[LWZZ13] P. Li, Q. Wang, W. Zuo, and L. Zhang. Log-Euclidean kernels for sparse representation and dictionary learning. *Proceedings of the IEEE International Conference on Computer Vision*, pages 1601–1608, 2013.

[MHB+16] Melissa Marchand, Wen Huang, Arnaud Browet, Paul Van Dooren, and Kyle A. Gallivan. A riemannian optimization approach for role model extraction. In *Proceedings of the 22nd International Symposium on Mathematical Theory of Networks and Systems*, pages 58–64, 2016.

[MHGM16] Melissa Marchand, Wen Huang, Kyle Gallivan, and Bradley Marchand. Multi-input multi-output waveform optimization for synthetic aperture sonar. In *Proc. SPIE*, volume 9823, pages 98231X–98231X–12, 2016.

[Mis14] B. Mishra. *A Riemannian approach to large-scale constrained least-squares with symmetries*. PhD thesis, University of Liege, 2014.

[Mit10] H. Mittelmann. Decision tree for optimization software, 2010. Tech Rep, School of Mathematical and Statistical Sciences, Arizona State University.

[MOHW07] J. C. Meza, R. A. Oliva, P. D. Hough, and P. J. Williams. OPT++: An object-oriented toolkit for nonlinear optimization. *ACM Transactions on Mathematical Software*, 33(2):12–es, 2007.

[MRHA16] S. Martin, A. M. Raim, W. Huang, and K. P. Adragni. ManifoldOptim: An R Interface to the ROPTLIB Library for Riemannian Manifold Optimization. pages 1–25, 2016.

[NW06] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, second edition, 2006.

[PFA06a] X. Pennec, P. Fillard, and N. Ayache. A Riemannian Framework for Tensor Computing. *International Journal of Computer Vision*, 66(5255):41–66, 2006.

[PFA06b] X. Pennec, P. Fillard, and N. Ayache. A Riemannian framework for tensor computing. *International Journal of Computer Vision*, 66(1):41–66, 2006.

[PJM12] R. E. Perez, P. W. Jansen, and J. R. R. A. Martins. PyOpt: A Python-based object-oriented framework for nonlinear constrained optimization. *Structural and Multidisciplinary Optimization*, 45(1):101–118, 2012.

[RHPA15] A. Rathore, W. Huang, and Absil P.-A. Riemannian optimization package, 2015.

[RW12] W. Ring and B. Wirth. Optimization methods on Riemannian manifolds and their application to shape space. *SIAM Journal on Optimization*, 22(2):596–627, January 2012. doi:10.1137/11082885X.

[Sat16] H. Sato. A Dai–Yuan-type Riemannian conjugate gradient method with the weak Wolfe conditions. *Computational Optimization and Applications*, 64(1):101–118, May 2016.

[SBMP14] R. Sivalingam, D. Boley, V. Morellas, and N. Papanikolopoulos. Tensor sparse coding for positive definite matrices. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(3):592–605, 2014.

[SI13] H. Sato and T. Iwai. A Riemannian optimization approach to the matrix singular value decomposition. *SIAM Journal on Optimization*, 23(1):188–212, 2013.

[TCA09] F. J. Theis, T. P. Cason, and P.-A. Absil. Soft dimension reduction for ICA by joint diagonalization on the Stiefel manifold. *Proceedings of the 8th International Conference on Independent Component Analysis and Signal Separation*, 5441:354–361, 2009.

[TKW16] J. Townsend, N. Koep, and S. Weichwald. Pymanopt: A python toolbox for optimization on manifolds using automatic differentiation. *Journal of Machine Learning Research*, 17(137):1–5, 2016.

[Van13] B. Vandereycken. Low-rank matrix completion by Riemannian optimization—extended version. *SIAM Journal on Optimization*, 23(2):1214–1236, 2013.

[WDAM15] I. Waldspurger, A. D´ Aspremont, and S. Mallat. Phase recovery, maxcut and complex semidefinite programming. *Mathematical Programming*, 149(1):47–81, Feb 2015.

[WY12] Z. Wen and W. Yin. Optimization with orthogonality constraints, 2012.

[YHAG16] X. Yuan, W. Huang, P.-A. Absil, and K. A. Gallivan. A Riemannian Limited-Memory BFGS Algorithm for Computing the Matrix Geometric Mean. *Procedia Computer Science*, 80:1–11, 2016.

[YHAG17] Xinru Yuan, Wen Huang, P.-A. Absil, and K. A. Gallivan. A Riemannian quasi-newton method for computing the Karcher mean of symmetric positive definite matrices. Technical Report FSU17-02, Florida State University, 2017. www.math.fsu.edu/ whuang2/papers/RMKMSPDM.htm.

[YHGA15] Y. You, W. Huang, K. A. Gallivan, and P. A. Absil. A Riemannian approach for computing geodesics in elastic shape analysis. In *2015 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 727–731, Dec 2015.

[You16] Rohollah Yousefpour. Combination of steepest descent and bfgs methods for nonconvex nonsmooth optimization. *Numerical Algorithms*, 72(1):57–90, May 2016.