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

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

March 14, 2017

**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. The code and a manual can be downloaded from `http://www.math.fsu.edu/~whuang2/Indices/index_ROPTLIB.html`.

**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 [Van12, 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

[†]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.

Karcher mean (with respect to the affine invariant metric [PFA06, JVV12]) of a set of symmetric 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, WDAM13, HGZ16]. 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 [Sat15] 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, 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 environment.

[1]To the best of our knowledge, Matlab is not an efficient language for Riemannian quasi-Newton method-

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: 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. 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

s. Specifically, since Matlab cannot invoke the rank-1 update function *dsyr* in BLAS directly without through 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 interface.

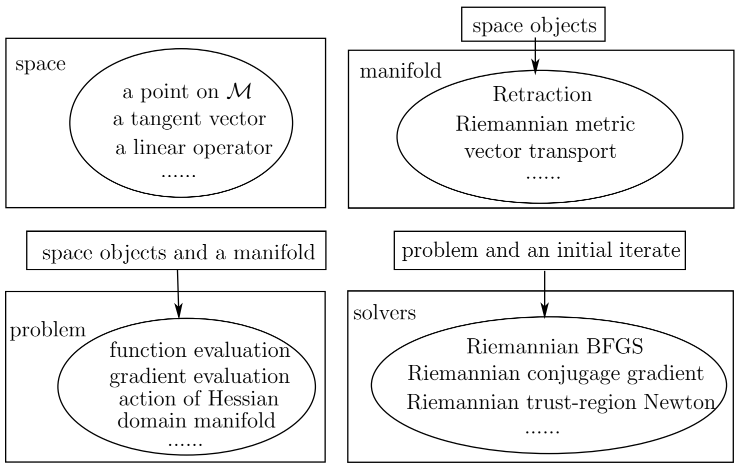


Figure 1: A sketch of the structure of ROPTLIB.

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. 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. 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. Those classes receive objects, 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. 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

The space-related classes support copy-on-write. 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. Copy-on-write is important to save computational time especially when handling large-scaled problems. Therefore, ROPTLIB does not use the standard C++ libraries to manipulate memory since these libraries do not support copy-on-write.

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 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 *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. This functionality is crucial to avoid some 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. 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. Specif-

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

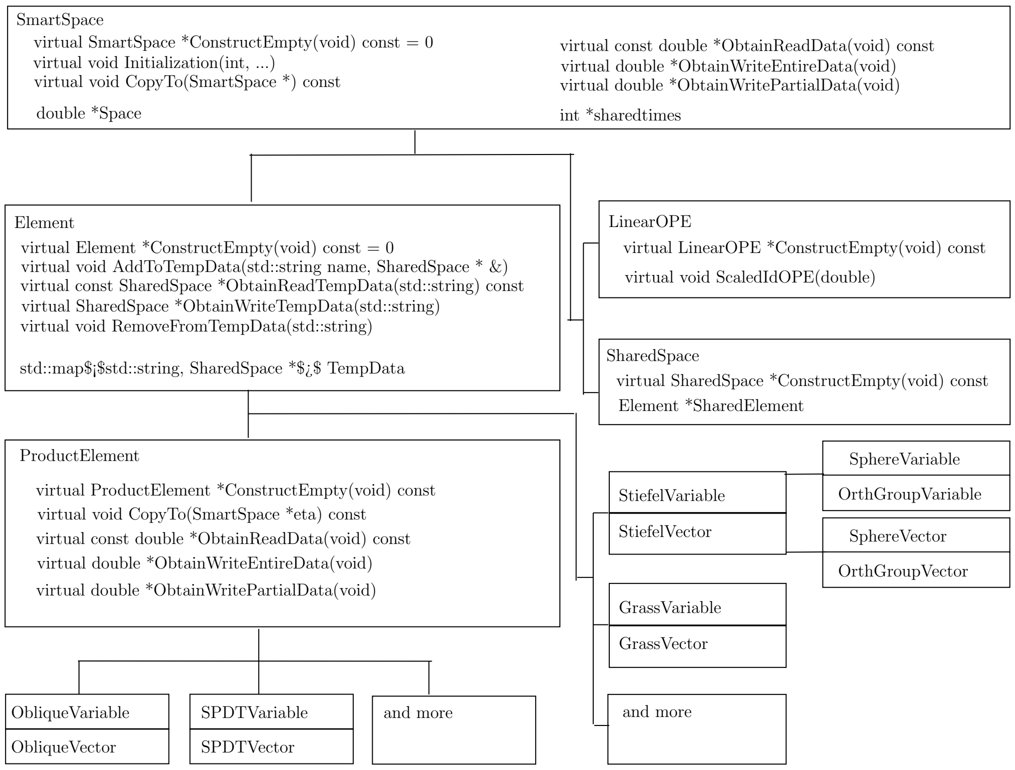

SmartSpace
virtual SmartSpace *ConstructEmpty(void) const = 0
virtual void Initialization(int, ...)
virtual void CopyTo(SmartSpace *) const
double *Space
virtual const double *ObtainReadData(void) const
virtual double *ObtainWriteEntireData(void)
virtual double *ObtainWritePartialData(void)
int *sharedtimes
Element
virtual Element *ConstructEmpty(void) const = 0
virtual void AddToTempData(std::string name, SharedSpace * &)
virtual const SharedSpace *ObtainReadTempData(std::string) const
virtual SharedSpace *ObtainWriteTempData(std::string)
virtual void RemoveFromTempData(std::string)
std::map$¡$std::string, SharedSpace *$¿$ TempData
LinearOPE
virtual LinearOPE *ConstructEmpty(void) const
virtual void ScaledIdOPE(double)
SharedSpace
virtual SharedSpace *ConstructEmpty(void) const
Element *SharedElement
ProductElement
virtual ProductElement *ConstructEmpty(void) const
virtual void CopyTo(SmartSpace *eta) const
virtual const double *ObtainReadData(void) const
virtual double *ObtainWriteEntireData(void)
virtual double *ObtainWritePartialData(void)
StiefelVariable
StiefelVector
SphereVariable
OrthGroupVariable
SphereVector
OrthGroupVector
GrassVariable
GrassVector
and more
ObliqueVariable
ObliqueVector
SPDTVariable
SPDTVector
and more

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

ically, 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 3.1.

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 3.1 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 [HAA16].

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.

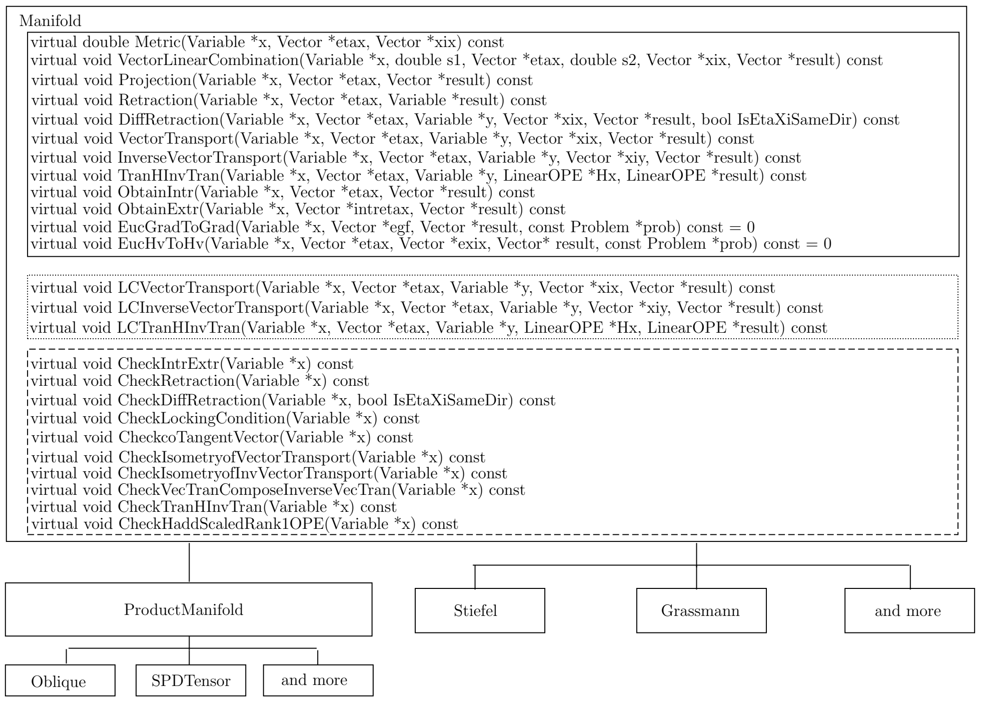
## 2.2 Manifold Classes

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.

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). 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 $\eta$ and $(R_x(\delta\eta_x) - R_x(0\eta_x))/\delta$ for a small value of $\delta$. We refer to the documentation in the code in [HAA16] for details.



```
Manifold
virtual double Metric(Variable *x, Vector *etax, Vector *xix) const
virtual void VectorLinearCombination(Variable *x, double s1, Vector *etax, double s2, Vector *xix, Vector *result) const
virtual void Projection(Variable *x, Vector *etax, Vector *result) const
virtual void Retraction(Variable *x, Vector *etax, Variable *result) const
virtual void DiffRetraction(Variable *x, Vector *etax, Variable *y, Vector *xix, Vector *result, bool IsEtaXiSameDir) const
virtual void VectorTransport(Variable *x, Vector *etax, Variable *y, Vector *xix, Vector *result) const
virtual void InverseVectorTransport(Variable *x, Vector *etax, Variable *y, Vector *xiy, Vector *result) const
virtual void TranHInvTran(Variable *x, Vector *etax, Variable *y, LinearOPE *Hx, LinearOPE *result) const
virtual void ObtainIntr(Variable *x, Vector *etax, Vector *result) const
virtual void ObtainExtr(Variable *x, Vector *intretax, Vector *result) const
virtual void EucGradToGrad(Variable *x, Vector *egf, Vector *result, const Problem *prob) const = 0
virtual void EucHvToHv(Variable *x, Vector *etax, Vector *exix, Vector* result, const Problem *prob) const = 0

virtual void LCVectorTransport(Variable *x, Vector *etax, Variable *y, Vector *xix, Vector *result) const
virtual void LCInverseVectorTransport(Variable *x, Vector *etax, Variable *y, Vector *xiy, Vector *result) const
virtual void LCTranHInvTran(Variable *x, Vector *etax, Variable *y, LinearOPE *Hx, LinearOPE *result) const

virtual void CheckIntrExtr(Variable *x) const
virtual void CheckRetraction(Variable *x) const
virtual void CheckDiffRetraction(Variable *x, bool IsEtaXiSameDir) const
virtual void CheckLockingCondition(Variable *x) const
virtual void CheckcoTangentVector(Variable *x) const
virtual void CheckIsometryofVectorTransport(Variable *x) const
virtual void CheckIsometryofInvVectorTransport(Variable *x) const
virtual void CheckVecTranComposeInverseVecTran(Variable *x) const
virtual void CheckTranHInvTran(Variable *x) const
virtual void CheckHaddScaledRank1OPE(Variable *x) const
```

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.

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

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 Riemannian algorithm used.

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.

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] |

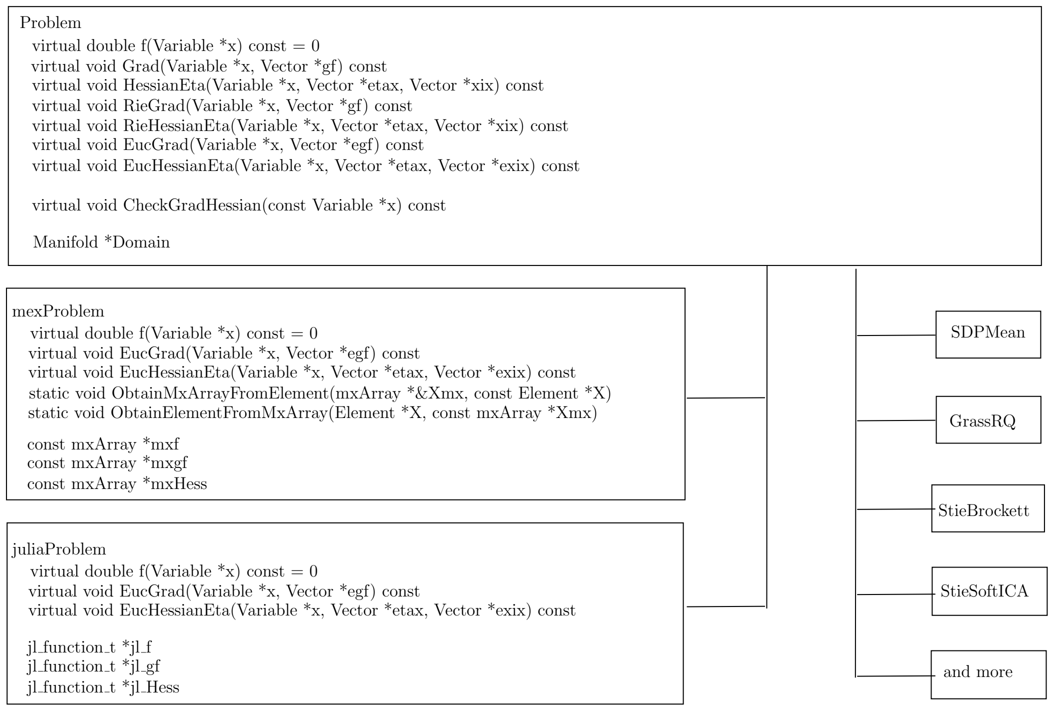

Problem
virtual double f(Variable *x) const = 0
virtual void Grad(Variable *x, Vector *gf) const
virtual void HessianEta(Variable *x, Vector *etax, Vector *xix) const
virtual void RieGrad(Variable *x, Vector *gf) const
virtual void RieHessianEta(Variable *x, Vector *etax, Vector *xix) const
virtual void EucGrad(Variable *x, Vector *egf) const
virtual void EucHessianEta(Variable *x, Vector *etax, Vector *exix) const
virtual void CheckGradHessian(const Variable *x) const
Manifold *Domain
mexProblem
virtual double f(Variable *x) const = 0
virtual void EucGrad(Variable *x, Vector *egf) const
virtual void EucHessianEta(Variable *x, Vector *etax, Vector *exix) const
static void ObtainMxArrayFromElement(mxArray *&Xmx, const Element *X)
static void ObtainElementFromMxArray(Element *X, const mxArray *Xmx)
const mxArray *mxf
const mxArray *mxgf
const mxArray *mxHess
juliaProblem
virtual double f(Variable *x) const = 0
virtual void EucGrad(Variable *x, Vector *egf) const
virtual void EucHessianEta(Variable *x, Vector *etax, Vector *exix) const
jl_function_t *jl_f
jl_function_t *jl_gf
jl_function_t *jl_Hess
SDPMean
GrassRQ
StieBrockett
StieSoftICA
and more

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.

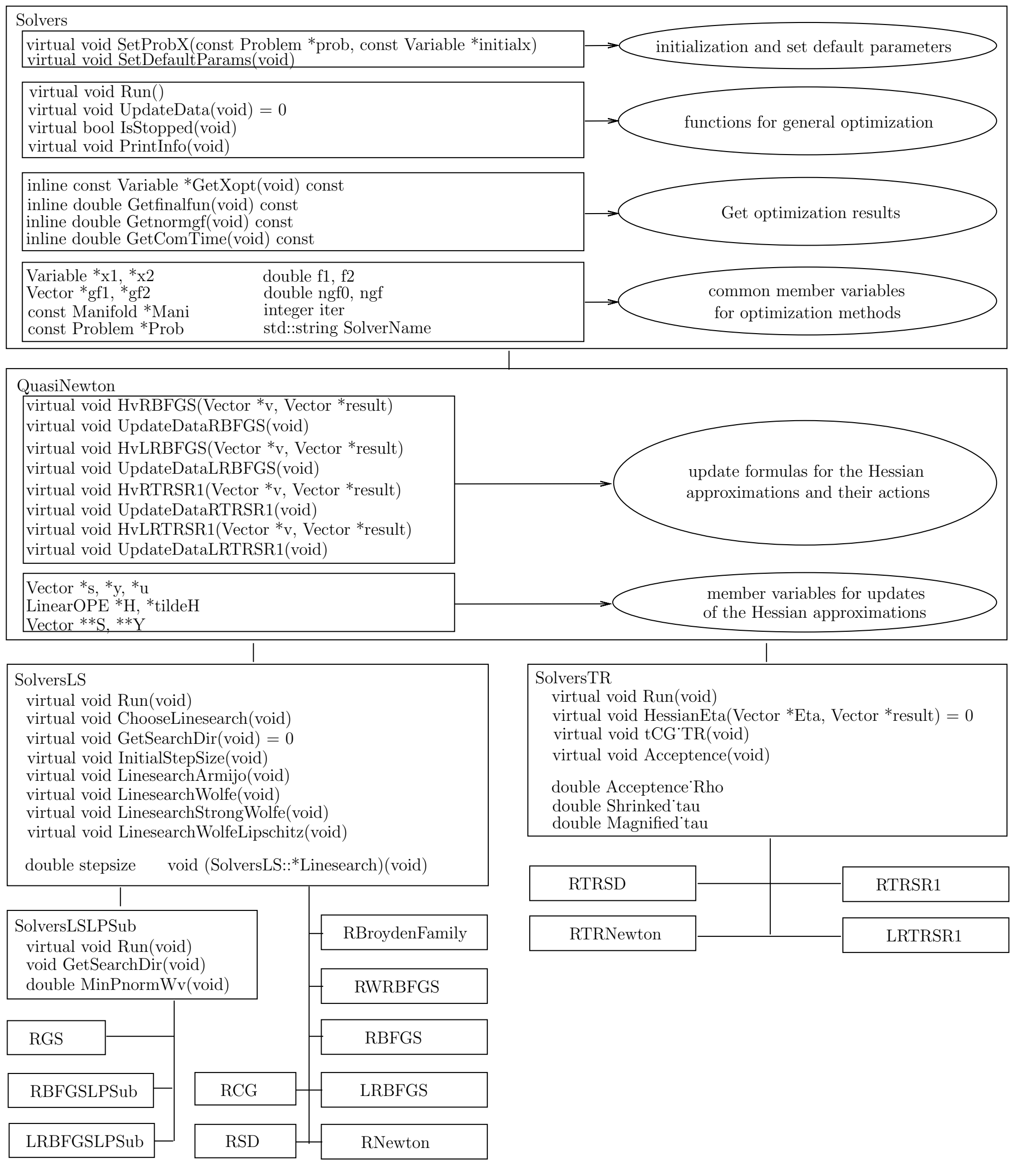

Solvers
virtual void SetProbX(const Problem *prob, const Variable *initialx)
virtual void SetDefaultParams(void)
initialization and set default parameters
virtual void Run()
virtual void UpdateData(void) = 0
virtual bool IsStopped(void)
virtual void PrintInfo(void)
functions for general optimization
inline const Variable *GetXopt(void) const
inline double Getfinalfun(void) const
inline double Getnormgf(void) const
inline double GetComTime(void) const
Get optimization results
Variable *x1, *x2
Vector *gf1, *gf2
const Manifold *Mani
const Problem *Prob
double f1, f2
double ngf0, ngf
integer iter
std::string SolverName
common member variables for optimization methods
QuasiNewton
virtual void HvRBFGS(Vector *v, Vector *result)
virtual void UpdateDataRBFGS(void)
virtual void HvLRBFGS(Vector *v, Vector *result)
virtual void UpdateDataLRBFGS(void)
virtual void HvRTRSR1(Vector *v, Vector *result)
virtual void UpdateDataRTRSR1(void)
virtual void HvLRTRSR1(Vector *v, Vector *result)
virtual void UpdateDataLRTRSR1(void)
update formulas for the Hessian approximations and their actions
Vector *s, *y, *u
LinearOPE *H, *tildeH
Vector **S, **Y
member variables for updates of the Hessian approximations
SolversLS
virtual void Run(void)
virtual void ChooseLinesearch(void)
virtual void GetSearchDir(void) = 0
virtual void InitialStepSize(void)
virtual void LinesearchArmijo(void)
virtual void LinesearchWolfe(void)
virtual void LinesearchStrongWolfe(void)
virtual void LinesearchWolfeLipschitz(void)
double stepsize
void (SolversLS::*Linesearch)(void)
SolversTR
virtual void Run(void)
virtual void HessianEta(Vector *Eta, Vector *result) = 0
virtual void tCG˙TR(void)
virtual void Acceptence(void)
double Acceptence˙Rho
double Shrinked˙tau
double Magnified˙tau
RTRSD
RTRSR1
RTRNewton
LRTRSR1
SolversLSLPSub
virtual void Run(void)
void GetSearchDir(void)
double MinPnormWv(void)
RBroydenFamily
RWRBFGS
RGS
RBFGS
RBFGSLPSub
RCG
LRBFGS
LRBFGSLPSub
RSD
RNewton

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.

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 $\alpha_k$ 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 $\eta_{x_k} = \arg\min_{s \in \mathrm{T}_{x_k}\mathcal{M} \text{ and } \|s\| \leq \delta} \langle \operatorname{grad} f(x_k), s\rangle + \frac{1}{2}\langle s, \mathcal{B}_k s\rangle$, 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 [You15, 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, 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 all 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, $\lambda_i$, 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 [HAA16].

## 3.1 In the C++ Environment

The code that defines the problem (3.1) can be found in the files "StieBrockett.h" and "StieBrockett.cpp" which are in the directory `ROPTLIB/Problems/StieBrockett/` of the source code of ROPTLIB.

A test file for Problem 3.1 is given in Listing 1, which is available in `/ROPTLIB/test/TestSimpleExample.cpp`.[3] In the test file, we show i) how to define a manifold, ii) how to generate an initial iterate, iii) how to construct a problem, and iv) how to run an optimization algorithm. Specifically, Lines 31 to 33 in Listing 1 define a Stiefel manifold and a random point on the manifold. Lines 34 and 35 defines Problem 3.1 by invoking the constructor function of *StieBrockett* and setting the domain to be the *Stiefel* object. Note that the problem class is supposed to be given by users. Therefore, users are responsible for the correctness of invoking the constructor. A solver is created in Line 39. In this case, the RTRNewton algorithm is used for finding a minimizer in **Prob** with the initial iterate **StieX**. The algorithm is run when Line 42 is executed.

Listing 1:

```
1  // File: TestSimpleExample.cpp
2  #ifndef TESTSIMPLEEXAMPLE_CPP
3  #define TESTSIMPLEEXAMPLE_CPP
4  #include "StieBrockett.h"
5  #include "StieVector.h"
6  #include "StieVariable.h"
7  #include "Stiefel.h"
8  #include "RTRNewton.h"
9  #include "def.h"
10 using namespace ROPTLIB;
11 #ifdef TESTSIMPLEEXAMPLE
12 
13 int main(void)
14 {
15         init_genrand((unsigned) time(NULL));   // choose a random seed
16         integer n = 12, p = 8;                 // size of the Stiefel manifold
17         // Generate the matrices in the Brockett problem.
18         double *B = new double[n * n + p];
19         double *D = B + n * n;
20         for (integer i = 0; i < n; i++)
21         {
22                 for (integer j = i; j < n; j++)
23                 {
24                         B[i + j * n] = genrand_gaussian();
25                         B[j + i * n] = B[i + j * n];
26                 }
27         }
28         for (integer i = 0; i < p; i++)
```

[3]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. Therefore, the differences are minor and should not cause confusion.

```
29              D[i] = static_cast<double> (i + 1);
30
31      StieVariable StieX(n, p);              // Obtain an initial iterate
32      StieX.RandInManifold();
33      Stiefel Domain(n, p);                  // Define the Stiefel manifold
34      StieBrockett Prob(B, D, n, p);         // Define the Brockett problem
35      Prob.SetDomain(&Domain);               // Set the domain
36      Domain.CheckParams();   // output the parameters of the manifold of domain
37      // test RTRNewton
38      std::cout << "********Check RTRNewton*********" << std::endl;
39      RTRNewton RTRNewtonsolver(&Prob, &StieX);
40      RTRNewtonsolver.DEBUG = FINALRESULT;
41      RTRNewtonsolver.CheckParams();
42      RTRNewtonsolver.Run();
43      // Check gradient and Hessian
44      Prob.CheckGradHessian(&StieX);
45      const Variable *xopt = RTRNewtonsolver.GetXopt();
46      Prob.CheckGradHessian(xopt);
47      // output the minimizer to the screen.
48      xopt->Print("Minimizer is:");
49      delete[] B;
50      return 0;
51  }
52  #endif
53  #endif
```

ROPTLIB allows users to output parameters of the domain manifold and the solver. For example, the commands in Line 36 and Line 41 are used to output the parameters of **Domain** and **RTRNewtonsolver** respectively. The resulting parameters can be found in the user manual [HAA16].

ROPTLIB provides a function, shown in Lines 44 and 46 of Listing 1, to test whether the gradient and the action of Hessian given by users are correct. We refer to the user manual [HAA16] for details.

### 3.2 In the Matlab and Julia Environments

Listings 2 and 3 give two examples for the Brockett cost function 3.1 in Matlab and Julia environments respectively. The Matlab and Julia codes of the cost function, gradient and action of Hessian are not given here and we refer to `/ROPTLIB/Matlab/ForMatlab/testSimpleExample.m` and `/ROPTLIB/Julia/JTestSimpleExample.jl` for completed versions of the codes. [4] Both Matlab and Julia interfaces support all the functionalities of ROPTLIB, such as output all the related parameters and check the correctness of the gradient and action of the Hessian.

In the Matlab interface, the cost function, Euclidean gradient and action of Euclidean Hessian are passed to ROPTLIB by function handles, see codes from Line 5 to Line 7. In contrast, Julia interface uses the names of the cost function, Euclidean gradient and action of Euclidean Hessian, see Lines 9 to 11. The solver-related parameters and manifold-related parameters are specified by structures in both environments.

All fields of the parameters in each solver and manifold can be found in Appendices B and C of the user manual [HAA16].

[4] 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. Therefore, the differences are minor and should not cause confusion.

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

  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

  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

  HasHHR = 0;                        % locking condition is not guaranteed.
  initialX.main = orth(randn(n, p));% initial iterate

  % 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 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))

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

# 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.

# 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

# 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 [HGZ16], optimization problems in elastic shape analysis [HGSA15, YHGA15], finding an geometric mean of symmetric positive definite matrices [YHAG15], 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 [HAA16].

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 `http://www.math.fsu.edu/~whuang2/papers/ROPTLIB.htm`.

### 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, $R = (r_{ij}) = \begin{bmatrix} r_1 & \ldots & r_N\end{bmatrix}$ 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, $\lambda_R$ 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

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

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) |

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. Parameters $\lambda_{\mathbf{B}}$ is set to 1. 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 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 $\kappa$. Every training datum $X_j$ is generated by a linear combination of $\kappa$ 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 $\begin{bmatrix}\mathrm{vec}(X_1) & \ldots & \mathrm{vec}(X_n)\end{bmatrix}$ and $\begin{bmatrix}\mathrm{vec}(B_1) & \ldots & \mathrm{vec}(V_n)\end{bmatrix}$ respectively, where $\mathrm{vec}(M)$ 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. 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]:

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

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 | 420 | 476 | 360 | 396 | 324 | 355 | 278 | 301 | 276 | 290 |
| nf | 426 | 985 | 365 | 802 | 328 | 709 | 282 | 601 | 280 | 571 |
| ng | 421 | 477 | 361 | 397 | 325 | 356 | 279 | 302 | 277 | 291 |
| nR | 425 | 984 | 364 | 801 | 327 | 708 | 281 | 600 | 279 | 570 |
| nV | 3347 | 952 | 2869 | 792 | 2579 | 710 | 2212 | 602 | 2193 | 580 |
| gf/gf0 | $9.01_{-7}$ | $8.75_{-7}$ | $8.91_{-7}$ | $8.99_{-7}$ | $9.02_{-7}$ | $8.72_{-7}$ | $8.74_{-7}$ | $8.80_{-7}$ | $8.91_{-7}$ | $8.70_{-7}$ |
| t | $2.39_{-1}$ | $4.52_{-1}$ | $4.23_{-1}$ | $8.20_{-1}$ | $6.48_{-1}$ | 1.26 | $8.62_{-1}$ | 1.68 | 1.27 | 2.36 |

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 | 18 | 33 | 35 | 48 | 182 | 176 | 335 | 303 | 389 | 344 |
| nf | 20 | 42 | 37 | 67 | 186 | 333 | 342 | 593 | 397 | 682 |
| ng | 19 | 34 | 36 | 49 | 183 | 177 | 336 | 304 | 390 | 345 |
| nR | 19 | 41 | 36 | 66 | 185 | 332 | 341 | 592 | 396 | 681 |
| nV | 131 | 66 | 264 | 96 | 1444 | 352 | 2670 | 606 | 3103 | 688 |
| gf/gf0 | $6.03_{-7}$ | $7.52_{-7}$ | $7.52_{-7}$ | $7.48_{-7}$ | $8.32_{-7}$ | $8.23_{-7}$ | $8.91_{-7}$ | $8.92_{-7}$ | $9.06_{-7}$ | $8.95_{-7}$ |
| t | $4.65_{-2}$ | $9.44_{-2}$ | $8.92_{-2}$ | $1.50_{-1}$ | $4.59_{-1}$ | $7.10_{-1}$ | $8.69_{-1}$ | 1.32 | 1.04 | 1.53 |

where

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

$\Omega$ 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 $\eta$ 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 $\Omega$ 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 $\Omega$ is given by the first $\tau$ 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, and the LRBFGS is the best one among them in terms of computational time.

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 | 80 | 0 | 0 | 445 | 78 | 134 | 0 | 0 |
| nH | 0 | 0 | 0 | 64 | 58 | 0 | 0 | 0 | 108 | 94 |
| gf/gf0 | $6.72_{-7}$ | $6.81_{-7}$ | $7.38_{-7}$ | $8.72_{-8}$ | $4.71_{-8}$ | $7.59_{-7}$ | $7.61_{-7}$ | $7.71_{-7}$ | $8.33_{-8}$ | $1.32_{-7}$ |
| t | $2.94_{-2}$ | $4.44_{-2}$ | $3.44_{-2}$ | $8.01_{-2}$ | $7.38_{-2}$ | $4.48_{-1}$ | $6.57_{-1}$ | $5.65_{-1}$ | 1.44 | 1.26 |
| f | $1.59_{-8}$ | $1.31_{-8}$ | $1.29_{-8}$ | $7.53_{-10}$ | $5.53_{-10}$ | $1.49_{-6}$ | $1.81_{-6}$ | $1.22_{-6}$ | $6.02_{-8}$ | $1.23_{-7}$ |

## 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

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

where $\mathrm{St}(p, n) = \{X \in \mathbb{R}^{n\times p} \mid X^T X = I\}$ 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 `http://www.math.fsu.edu/~whuang2/papers/ROPTLIB.htm`. 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., $O(n^2 p)$. 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.

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 terms of time efficiency) with ROPTLIB only when the computation time is dominated by high-efficiency libraries.

## 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 and is an efficient library for various sizes 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.

## 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.

[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.

[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.

[HAA16] W. Huang, P.-A. Absil, and Gallivan K. A. Riemannian manifold optimization library, 2016.

[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*, pages 1–8, 2016.

[HAG16b] Wen Huang, P.-A. Absil, and K. A. Gallivan. Intrinsic representation of tangent vectors and vector transport on matrix manifolds. *Numerische Mathematik*, 2016.

[HGA15] Wen 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] Wen 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.

[HGZ16] W. Huang, K. A. Gallivan, and X. Zhang. Solving PhaseLift by low-rank Riemannian optimization methods for complex semidefinite constraints. *UCL-INMA-2015.01.v2*, 2016.

[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 non-smooth 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.

[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, 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 objective-oriented toolkit for nonlinear optimization. *ACM Transactions on Mathematical Software*, 33(2):12–es, 2007.

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

[PFA06] X. Pennec, P. Fillard, and N. Ayache. A Riemannian Framework for Tensor Computing. *International Journal of Computer Vision*, 66(5255):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.

[Sat15] H. Sato. A Dai-Yuan-type Riemannian conjugate gradient method with the weak Wolfe conditions. *Computational Optimization and Applications*, 2015. to appear.

[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.

[Van12] B. Vandereycken. Low-rank matrix completion by Riemannian optimization—extended version. *SIAM Journal on Optimization*, 23(2):1214–1236, 2012.

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

[WDAM13] I. Waldspurger, A. D´ Aspremont, and S. Mallat. Phase recovery, maxcut and complex semidefinite programming. *Mathematical Programming*, December 2013. doi:10.1007/s10107-013-0738-9.

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

[YHAG15] X. Yuan, W. Huang, P.-A. Absil, and K. A. Gallivan. A riemannian limited-memory bfgs algorithm for computing the matrix geometric mean. Technical Report UCL-INMA-2015.12, U.C.Louvain, 2015.

[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.

[YHGA15] Y. You, W. Huang, K. A. Gallivan, and P.-A. Absil. A Riemannian Approach for Computing Geodesics in Elastic Shape Analysis. In *3rd IEEE Global Conference on Signal and Information Processing*, December 2015. to appear.

[You15] R. Yousefpour. Combination of steepest descent and BFGS methods for nonconvex nonsmooth optimization. *Numerical Algorithms*, pages 57–90, 2015.