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

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

October 27, 2016

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
Karcher mean (with respect to the affine invariant metric \cite{PFA06, JVV12}) of a set of symmetric positive definite (SPD) matrices can be written as an optimization problem over the manifold of SPD matrices \cite{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 $L^2$ space \cite{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 \cite{CESV13, WDAM13, HGZ16}. We refer to \cite{AMS08, Hua13} for more applications.

Many effective and efficient optimization methods on Riemannian manifolds have been proposed and analyzed. In 2007, Absil et al. \cite{ABG07} exploited second order information and developed a trust region Newton method. In 2012, Ring and Wirth \cite{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. \cite{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 \cite{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 \cite{RW12} and only needs the weak Wolfe condition in the line search. Again in the same year, Huang et al. \cite{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 \cite{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 a particular direction, which results in computational benefits in some cases. In 2016, Huang et al. \cite{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 \cite{Abr07} developed by Abrudan implements a conjugate gradient algorithm \cite{AEK09} and a steepest descent algorithm \cite{AEK08} only for the unitary matrix constraint. A more recent Matlab package \cite{WY12} gives a Barzilai-Borwein method for manifolds with orthogonality constraints. The R package GrassmannOptim \cite{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 \cite{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 4). 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 compiled 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.
on the Stiefel manifold. A benchmark is given in Section 4. Conclusion and future work are in Section 5.

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 da-
ta. 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 \url{http://www.cplusplus.com/reference/map/map/}.
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 \textit{AddToTempData} is used to add a temporary data, and the member functions \textit{ObtainReadTempData} and \textit{ObtainWriteTempData} are used to obtain stored temporary data. An example is given in Secton 3.1.

The classes \textit{SharedSpace} and \textit{LinearOPE} are derived classes of \textit{SmartSpace}. \textit{SharedSpace} is only used as space for temporary data. One can either store an arbitrary length double array or an \textit{Element} object, which is a point or a tangent vector on the manifold, to an object of \textit{SharedSpace}. Allowing to attach an \textit{Element} object eases implementations in some cases, see Section 3.1 for an example. \textit{LinearOPE} defines a linear operator on a tangent space and typically is a matrix.

After the class \textit{Element} is defined, a point on any manifold and a tangent vector of any manifold can be defined as a derived class. For example, \textit{StieVariable}, \textit{GrassVariable}, \textit{SPDVariable} are classes for a point on the Stiefel manifold, Grassmann manifold, and the manifold of symmetric positive definite (SPD) matrices, respectively, and \textit{StiefelVector}, \textit{GrassVector}, \textit{SPDVector} are classes for a tangent vector of those three manifolds, respectively. For more manifolds, we refer to the user manual [HAA16].

ROPTLIB defines \textit{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 \textit{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 \textit{ConstructEmpty} is declared in \textit{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 \textit{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 \( \eta_x \in T_x \mathcal{M} \). The function \textit{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.
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 $B_k$ 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 $T_k \circ B_k \circ T_k^{-1}$, where $T_k$ is a vector transport from $T_{x_k} \mathcal{M}$ the tangent space at $x_k$ to $T_{x_{k+1}} \mathcal{M}$ 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 $\text{ObtainIntr}$ ($\text{ObtainExtr}$) 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 $\text{Problem}$. 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 EucGradoGrad, 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>
<thead>
<tr>
<th>Algorithm</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Riemannian trust-region Newton (RTRNewton)</td>
<td>[ABG07]</td>
</tr>
<tr>
<td>Riemannian trust-region symmetric rank-one update (RTRSR1)</td>
<td>[HAG15]</td>
</tr>
<tr>
<td>Limited-memory RTRSR1 (LRTRSR1)</td>
<td>[HAG15]</td>
</tr>
<tr>
<td>Riemannian trust-region steepest descent (RTRSD)</td>
<td>[AMS08]</td>
</tr>
<tr>
<td>Riemannian line-search Newton (RNewton)</td>
<td>[AMS08]</td>
</tr>
<tr>
<td>Riemannian Broyden family (RBroydenFamily)</td>
<td>[HGA15]</td>
</tr>
<tr>
<td>Riemannian BFGS (RWBFBS and RFBS)</td>
<td>[RW12]</td>
</tr>
<tr>
<td>Riemannian BFGS (RWBFBS and RFBS)</td>
<td>[HGA15]</td>
</tr>
<tr>
<td>Subgradient Riemannian (L)BFGS ((L)RBFBSLPSUB)</td>
<td>[AHY16]</td>
</tr>
<tr>
<td>Limited-memory RBFBS (LRBFBS)</td>
<td>[HGA15]</td>
</tr>
<tr>
<td>Riemannian conjugate gradients (RCG)</td>
<td>[NW06]</td>
</tr>
<tr>
<td>Riemannian steepest descent (RSD)</td>
<td>[AMS08]</td>
</tr>
<tr>
<td>Riemannian gradient sampling (RGS)</td>
<td>[Hua13]</td>
</tr>
<tr>
<td></td>
<td>[SH16]</td>
</tr>
</tbody>
</table>
Figure 4: The class hierarchy of problem-related classes in ROPTLIB. We refer to the documentation in the code for the detailed explanations of the functions.
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:

$$x_{k+1} = R_{x_k}(-\alpha_k H_k \nabla f(x_k)),$$

where $\alpha_k$ is a step size and $H_k$ is an inverse Hessian approximation, or a trust-region iteration:

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

where $\eta_{x_k} = \arg \min_{s \in T_{x_k} \mathcal{M}} \|s\| \leq \delta \langle \nabla f(x_k), s \rangle + \frac{1}{2} \langle s, B_k s \rangle$, and $B_k$ is a Hessian approximation. This class QuasiNewton defines the commonly-used update formulas for $H_k$ and $B_k$ and also defines their actions $H_k \xi_k$ and $B_k \xi_k$ for any $\xi_k \in T_{x_k} \mathcal{M}$. 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 Riemannian optimization algorithms. Note that the subgradient-based algorithms, RGS, RBFGLPSUB and LRBFGGLPSUB 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 that solves optimization problems on Riemannian manifolds using ROPTLIB. The problem is to minimize the Brockett cost func-
tion [AMS08, Section 4.8] on the Stiefel manifold $\text{St}(p, n) = \{ X \in \mathbb{R}^{n \times p} | X^T X = I_p \}$

$$\min_{X \in \text{St}(p, n)} \text{trace}(X^T B X D)$$  \hspace{1cm} (3.1) \{\text{prob1}\}

where $B \in \mathbb{R}^{n \times n}$, $B = B^T$, $D = \text{diag}(\mu_1, \mu_2, \ldots, \mu_p)$ and $\mu_1 \geq \mu_2 \geq \cdots \geq \mu_p$. It is known that the columns of a global minimizer, $X^* e_i$, 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) is given in Listing 1. Note that they 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.

Listing 1:

```cpp
1 // File: StieBrockett.h
2 #ifndef STIEBROCKETT_H
3 #define STIEBROCKETT_H
4
5 #include "Stiefel.h"
6 #include "StieVariable.h"
7 #include "StieVector.h"
8 #include "Problem.h"
9 #include "SharedSpace.h"
10 #include "def.h"
11 #include "MyMatrix.h"
12
13 /*Define the namespace*/
14 namespace ROPTLIB{
15     class StieBrockett : public Problem{
16         public:
17             StieBrockett(double *inB, double *inD, integer inn, integer inp);
18             virtual ~StieBrockett();
19             virtual double f(Variable *x) const;
20             virtual void EurGrad(Variable *x, Vector *egf) const;
21             virtual void EurHessianEta(Variable *x, Vector *etax, Vector *exix) const;
22             double *B;
23             double *D;
24             integer n;
25             integer p;
26         }
27     }; /*end of ROPTLIB namespace*/
28 #endif // end of STIEBROCKETT_H
29
30 // File: StieBrockett.cpp
31 #include "StieBrockett.h"
32
33 /*Define the namespace*/
34 namespace ROPTLIB{
35     StieBrockett::StieBrockett(double *inB, double *inD, integer inn, integer inp) {
36         B = inB;
37         D = inD;
38         n = inn;
39         p = inp;
40     }
41     StieBrockett::~StieBrockett(void)
42 }
```
double StieBrockett::f(Variable *x) const
{
    const double *xxM = x->ObtainReadData();
    Vector *BxD = x->ConstructEmpty();
    SharedSpace *Temp = new SharedSpace(BxD);
    double *temp = BxD->ObtainWriteEntireData();
    double result = 0;

    Matrix MB(B, n, n), MxxM(xxM, n, p), Mtemp(temp, n, p);
    Matrix::DGEMM(1, MB, false, MxxM, false, 0, Mtemp);

    for (integer i = 0; i < p; i++)
    {
        // temp(i * n : i * n + N - 1) <- D[i] * temp(i * n : i * n + N - 1),
        // details: http://www.netlib.org/lapack/explore-html/d4/dd0/ddscal_8f.html
        dscal_(const_cast<integer *>(&n), &D[i], temp + i * n, &GLOBAL::IONE);
    }

    integer length = n * p;
    result = ddot_(&length, temp, &GLOBAL::IONE, const_cast<double *>(xxM), &GLOBAL::IONE);

    if (UseGrad)
    {
        x->AddToTempData("BxD", Temp);
    }
    else
    {
        delete Temp;
    }
    return result;
};
void StieBrockett::EucGrad(Variable *x, Vector *egf) const
{
    const SharedSpace *Temp = x->ObtainReadTempData("BxD");
    Vector *BxD = Temp->GetSharedElement();
    Domain->ScaleTimesVector(x, 2.0, BxD, egf);
};
void StieBrockett::EucHessianEta(Variable *x, Vector *etax, Vector *exix) const
{
    const double *etaxTV = etax->ObtainReadData();
    double *exixTV = exix->ObtainWriteEntireData();

    char *transn = const_cast<char *>("n");
    integer N = n, P = p, inc = 1, Length = N * P;
    double one = 1, zero = 0, two = 2;
    // exixTV <- B * etaxTV, details: http://www.netlib.org/lapack/explore-html/d7/d2b/dgemm_8f.html
    dgemm_(transn, transn, &N, &P, &N, &one, B, &N, const_cast<double *>(etaxTV),
           &N, &zero, exixTV, &N);

    for (integer i = 0; i < p; i++)
    {
        // exixTV(i * n : i * n + N - 1) <- D[i] * exixTV(i * n : i * n + N - 1),
        // details: http://www.netlib.org/lapack/explore-html/d4/dd0/ddscal_8f.html
        dscal_(&N, &D[i], exixTV + i * n, &inc);
    }
}
The code defines a problem class `StieBrockett` which inherits the class `problem`. The function evaluation, Euclidean gradient and the action of Euclidean Hessian are overridden as shown in Listing 1 from lines 46 to 100.

As discussed in Section 2.1, one can use one of the following three functions to obtain a double pointer to the data:

1. `virtual const double *ObtainReadData(void) const;`
2. `virtual double *ObtainWriteEntireData(void);`
3. `virtual double *ObtainWritePartialData(void).`

Listing 1 gives an example to show how to use the functions properly. In the function evaluation, the data in `x` do not need to be modified. Therefore, we access its data by function `ObtainReadData` for efficiency. In the action of the Hessian `EuclHessianEta`, the resulting vector must be stored in `exix`. Since the original data in `exix` is not important, we use the function `ObtainWriteEntireData`.

The Brockett cost function in Problem 3.1 is \( \text{trace}(X^T B XD) \) and its Euclidean gradient is \( 2B XD \). Since \( B XD \) must be computed in the cost function evaluation, it can be attached to `X` and reused in the gradient evaluation. Line 49 of Listing 1 creates an empty vector `BxD` such that its size is the same size as `x`. Line 50 creates a `SharedSpace` object `Temp` that points to `BxD`. In Line 67, if the Riemannian algorithm is not gradient free, then the `SharedSpace` object `Temp` is attached to the current iterate `x`. Otherwise, `Temp` must be deleted to avoid memory leakage. Note that the general rule is that one must delete all objects that are created by `new` or `ConstructEmpty` unless the objects are attached as temporary data to other objects. The temporary data is automatically deleted when the main data is modified or deleted. For example, `Temp` is always on `x` as long as `x` is not changed. The temporary data is used in the gradient evaluation. Line 79 of Listing 1 gets the `SharedSpace` object `Temp` and Line 80 obtains the temporary data `BxD` from `Temp`.

A test file for Problem 3.1 is given in Listing 2, which is available in `/ROPTLIB/test/TestSimpleExample.cpp`. 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 40 to 45 in Listing 2 define a Stiefel manifold and a random point on the manifold. Lines 48 and 51 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 58. 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 61 is executed.

---

1 // File: TestSimpleExample.cpp
2 #ifndef TESTSIMPLEEXAMPLE_CPP
3 #define TESTSIMPLEEXAMPLE_CPP
4 #endif
5 #include "StieBrockett.h"
6 #include "StieVector.h"
7 #include "StieVariable.h"

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.
#include "Stiefel.h"
#include "RTRNewton.h"
#include "def.h"

using namespace ROPTLIB;

#ifdef TESTSIMPLEEXAMPLE

int main(void)
{
  // choose a random seed
  unsigned tt = (unsigned)time(NULL);
  init_genrand(tt);

  // size of the Stiefel manifold
  integer n = 12, p = 8;

  // 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); }

  // Obtain an initial iterate
  StieVariable StieX(n, p);
  StieX.RandInManifold();

  // Define the Stiefel manifold
  Stiefel Domain(n, p);

  // Define the Brockett problem
  StieBrockett Prob(B, D, n, p);

  // Set the domain of the problem to be the Stiefel manifold
  Prob.SetDomain(&Domain);

  // output the parameters of the manifold of domain
  Domain.CheckParams();

  // 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 = RTRNewtonSolverGetXopt();
  Prob.CheckGradHessian(xopt);

  // output the minimizer to the screen
  std::cout << "Minimizer is:"
  delete[] B;

  return 0;
}

}
ROPTLIB allows users to output parameters of the domain manifold and the solver. For example, the commands in Line 54 and Line 60 are used to output the parameters of **Domain** and **RTRNewtonSolver** respectively. The resulting parameters are given in Figures 6 and 7. The meanings of the names of the parameters can be found in the user manual of ROPTLIB [HAA16].

**GENERAL PARAMETERS:**
name : Stiefel, IsIntrApproach: 1
IntrinsicDim : 60, ExtrinsicDim : 96
HasHHR : 0, UpdBetaAlone : 0
HasLockCon : 0
Stiefel PARAMETERS:
n : 12, p : 8
metric : EUCLIDEAN, retraction : QF
VecTran : PARALLELIZATION

**Figure 6: Parameters of a Stiefel manifold**

**GENERAL PARAMETERS:**
Stop_Criterion: GRAD_F_0[YES], Tolerance : 1e-006[YES]
Max_Iteration : 500[YES], Min_Iteration : 0[YES]
OutputGap : 1[YES], DEBUG : FINALRESULT[YES]
Diffx : 1e-006[YES], NumExtraGF : 3[YES]
TRUST_REGION_TYPE METHODS PARAMETERS:
initial_Delta : 1[YES], Acceptance_Rho: 0.1[YES]
Shrinked_tau : 0.25[YES], Magnified_tau : 2[YES]
minimum_Delta : 2.22045e-016[YES], maximum_Delta : 1000[YES]
Min_Inner_Iter: 0[YES], Max_Inner_Iter: 1000[YES]
theta : 1[YES], kappa : 0.1[YES]
useRand : 0[YES]

**Figure 7: Parameters of the RTRNewton solver.**

ROPTLIB provides a function, shown in Lines 64 and 66 of Listing 2, to test whether the gradient and the action of Hessian given by users are correct. The idea is based on Taylor’s theorem. More specifically, let \( \tilde{f}_x(\eta_x) \) be \( f(R_x(\eta_x)) \). If \( f \in C^2 \), then using Taylor’s theorem yields

\[
\tilde{f}_x(\eta_x) = \hat{f}_x(0_x) + \langle \nabla \hat{f}_x(0_x), \eta_x \rangle + \frac{1}{2} \langle \text{Hess} \hat{f}_x(0_x)[\eta_x], \eta_x \rangle + o(\|\eta_x\|^2)
\]

\[
= f(x) + \langle \nabla f(x), \eta_x \rangle + \frac{1}{2} \langle \text{Hess} f(x)[\eta_x], \eta_x \rangle + o(\|\eta_x\|^2).
\]

If the retraction \( R \) is a second-order retraction or \( x \) is a stationary point of \( f \), then \( \text{Hess} \hat{f}_x(0_x) = \text{Hess} f(x) \) by [AMS08, Propositions 5.5.5 and 5.5.6]. It follows that

\[
f(y) = f(x) + \langle \nabla f(x), \eta_x \rangle + \frac{1}{2} \langle \text{Hess} f(x)(\eta_x), \eta_x \rangle + o(\|\eta_x\|^2),
\]

where \( y = R_x(\eta_x) \). The function in this package computes

\[
(f(y) - f(x))/\langle \nabla f(x), \eta_x \rangle \tag{3.2}
\]
and
\[
(f(y) - f(x) - \langle \nabla f(x), \eta_x \rangle)/(0.5(\text{Hess } f(R_x(\eta_x))[\eta_x], \eta_x))
\] (3.3) \{UM:e2\}

for $\eta_x = \alpha \xi$ such that $||\xi|| = 1$, $\alpha$ decreases from 100 to $100 \times 2^{-35}$. Suppose there exists an interval of $\alpha$ such that numerical errors do not take effect and the values of $\alpha$ are sufficiently small such that the higher order term is negligible. If (3.2) is approximately 1 in the interval, then \( \nabla f \) is probably correct. Likewise, if (3.3) is approximately 1 in the interval, then \( \text{Hess } f \) is probably correct.

### 3.2 In the Matlab Environment

After setting up the C++ Mex environment in Matlab, one can use command “MyMex DriverMex-Prob” to generate a binary file. The binary can be called from Matlab by inputting function handles and parameter structures. We have wrapped this function by a script `/ROPTLIB/ Matlab/ DriverOPT.m`. DriverOPT.m is used to check correctness of the input parameters and reshape the data from C++ solvers.

The script DriverOPT can be called by Listing 3,

```matlab
[FinalIterate, fv, gfv, gfgf0, iter, nf, ng, nR, nV, nVp, nH, ComTime, funs, grads, times] = DriverOPT(fhandle, gfhandle, Hesshandle, SolverParams, ManiParams, HasHHR, initialIterate)
```

where “initialIterate” and “finalIterate” are structures that contain initial and final iterates respectively; “fv” is the final cost function value; “gfv” is the norm of the final gradient; “gfgf0” is the norm of the final gradient over the norm of the initial gradient; “iter”, “nf”, “ng”, “nR”, “nV/nVp”, “nH” denote the number of iterations, the number of function evaluations, the number of gradient evaluations, the number of retraction evaluations, the number of vector transport (expensive/cheap), and the number of action of the Hessian evaluations respectively; “ComTime” denotes the total computational time; “funs”, “grads”, and “times” are arrays that store the function values, norm of gradients and the accumulated computational time at each iteration. “fhandle”, “gfhandle”, and “Hesshandle” are function handles of cost function, its Euclidean gradient and its action of Euclidean Hessian; “SolverParams” and “ManiParams” are structures that specify parameters of solver and manifold respectively; and “HasHHR” indicates whether the locking condition [HGA15, (2.8)] is satisfied by using the approach in [HGA15, Section 4.1].

An example for Brockett cost function (3.1) is given in Listing 4 and the code can be found in `/ROPTLIB/ Matlab/ ForMatlab/testSimpleExample.m`. First, the cost function, Euclidean gradient and action of Euclidean Hessian are given from Line 25 to Line 36. Their function handles are extracted in the codes from Line 5 to Line 7. The Riemannian optimization algorithm and the domain manifold are specified from Line 9 to Line 17. The Matlab interface is also able to output all the related parameters (see Lines 11 and 16) and to check the correctness of the gradient and action of the Hessian (see Line 12). Iterates and tangent vectors are stored as structures with field “main”, as shown in Line 20 and Line 36. In case of storing a temporary data to save computations, users can put the temporary data on an iterate with a different field. Similar to what has been done in Section 3.1, one can use the result $BXD$ from the function evaluation to reduce computation in the gradient evaluation. This can be seen from definitions of $f(x, B, D)$ and $gf(x, B, D)$ in Line 27.

---

4The 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.
and Line 32. All fields of the parameters in each solver and manifold can be found in Appendices B and C of the user manual [HAA16].

Listing 4:

```matlab
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

function [output, x] = f(x, B, D)
    x.BUD = B * x.main * D;
    output = x.main(:)' * x.BUD(:);
end

function [output, x] = gf(x, B, D)
    output.main = 2 * x.BUD;
end

function [output, x] = Hess(x, eta, B, D)
    output.main = 2 * B * eta.main * D;
end
```

### 3.3 In the Julia Environment

In Julia, a shared library of ROPTLIB needs to be generated first (see the user manual in [HAA16] for details). ROPTLIB then can be added to Julia by running `ROPTLIB/Julia/BeginROPTLIB.jl`. The interface in Julia is given by

```matlab
[FinalIterate, fv, gfv, gfgf0, iter, nf, ng, nR, nV, nVp, nH, ComTime, funs, grads, times] = DriverJuliaOPT(Handles, SolverParams, ManiParams, HasHHR, initialIterate);
```

where the notation is the same as those in Listing 3 except that “Handles” is a composite type containing all the function names (see an example in Lines 8 to 10 in Listing 5).

Listing 5 shows an example to optimize the Brockett cost function (3.1). The code is available in `/ROPTLIB/Julia/JTestSimpleExample.jl`. The manifold is specified from Lines 1 to 7. Note that the name and size of a manifold are defined to be an array. The reason is to make the code compatible for a product of manifolds. See `/ROPTLIB/Julia/JTestProductExample.jl` for an example
on a product of manifolds and related information. The names of the cost function, gradient and action of Hessian are given from Lines 9 to 11. The functions are defined later from Lines 28 to 41. The solver-related parameter is defined from Lines 14 to 17. Unlike ManiParams and FunHandles, an object Sparams of SolverParams has been defined in BeginROPTLIB.jl. Therefore, users do not need to create an object of type SolverParams but need only modify Sparams. The default values of Sparams can be found in Appendix B of the user manual [HAA16].

The Julia interface also supports sharing data across functions. As shown in Line 32 of Listing 5, the temporary data is stored in the object outTmp. In the gradient evaluation, the temporary data is given in the object inTmp and can be used to avoid some redundant computations.

Note that size information about all data is not explicitly stored. Therefore, it is required to reshape the data, as shown in Lines 30, 37, and 43. This has little impact on the efficiency of ROPTLIB since the data in memory do not change when reshaped.

define

Listing 5:

```julia
# 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;
Sparams = ManiParams(IsCheckParams, length(ManArr), pointer(ManArr), pointer(numofmani),
pointer(params), 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

# Define function handles. The function names must be assigned to the "FunHandles" struct.
See lines 17-21

function func(x, inTmp)
    x = reshape(x, n, p) # All the input argument is a vector. One has to reshape it to have a proper size
    outTmp = B * x * D
    fx = vecdot(x, outTmp)
    return (fx, outTmp) # The temporary data "outTmp" will replace the "inTmp"
end

function gfunc(x, inTmp) # The inTmp is the temporary data computed in "func".
inTmp = reshape(inTmp, n, p) # All the input argument is a vector. One has to reshape it to have a proper size
gf = 2.0::Float64 * inTmp
    return (gf, []) # If one does not want to change the temporary data, then let the outTmp be an empty array.
end

{code4}
```
function hfunc(x, inTmp, eta)  # All the input argument is a vector. One has to reshape it
to have a proper size
    eta = reshape(eta, n, p)  # All the input argument is a vector. One has to reshape it
to have a proper size
    result = 2.0::Float64 * B * eta * D
    return (result, [])  # If one does not want to change the temporary data, then let
the outTmp be an empty array.
end

# 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 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

\[ f : \text{St}(p, n) \rightarrow \mathbb{R} : X \mapsto f(X) = -\sum_{i=1}^{N} \| \text{diag}(X^T C_i X) \|^2, \]

where \( \text{St}(p, n) = \{ X \in \mathbb{R}^{n \times p} \mid X^T X = I \} \) denotes the Stiefel manifold, \( C_1, \ldots, C_N \) are given symmetric matrices, \( \text{diag}(M) \) denotes the vector formed by the diagonal entries of \( M \), and \( \| \text{diag}(M) \|^2 \) 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 \( \text{http://www.math.fsu.edu/~whuang2/papers/ROPTLIB.htm} \). 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.

An average computational time and the corresponding average number of function evaluations
of 20 random runs for multiple values of \( p, n \), and \( N \) are given in Table 2. For small size problems,
ROPTLIB is faster than Manopt and Pymanopt by a factor of 100. As \( n \) and \( p \) grow, the factor
gradually reduces to approximately 1 for large-scale problem. In order to understand the phe-
nomenon, 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 computa-
tional 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 100. 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 2: An average computational time and the corresponding average number of function evaluations of 20 random runs given by ROPTLIB, Manopt, and Pymanopt. Multiple values of \(p\), \(n\), and \(N\) are used. \(t\) and \(nf\) denote computational time (second) and the number of function evaluations.

<table>
<thead>
<tr>
<th>(p, n, N)</th>
<th>2, 4, 128</th>
<th>8, 16, 128</th>
<th>32, 64, 32</th>
<th>128, 256, 8</th>
<th>512, 1024, 2</th>
<th>1024, 2048, 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROPTLIB</td>
<td>(t)</td>
<td>0.001</td>
<td>0.004</td>
<td>0.016</td>
<td>0.183</td>
<td>4.059</td>
</tr>
<tr>
<td></td>
<td>(nf)</td>
<td>41</td>
<td>43</td>
<td>44</td>
<td>46</td>
<td>48</td>
</tr>
<tr>
<td>Manopt</td>
<td>(t)</td>
<td>0.091</td>
<td>0.271</td>
<td>0.353</td>
<td>0.847</td>
<td>6.818</td>
</tr>
<tr>
<td></td>
<td>(nf)</td>
<td>41</td>
<td>43</td>
<td>44</td>
<td>46</td>
<td>48</td>
</tr>
<tr>
<td>Pymanopt</td>
<td>(t)</td>
<td>0.164</td>
<td>0.405</td>
<td>0.490</td>
<td>0.894</td>
<td>7.951</td>
</tr>
<tr>
<td></td>
<td>(nf)</td>
<td>41</td>
<td>43</td>
<td>44</td>
<td>46</td>
<td>48</td>
</tr>
</tbody>
</table>

These experiments verify that the computational time given in Manopt and Pymanopt is meaningful only if the computational time of an algorithm is dominated by executing high-efficiency libraries. ROPTLIB avoids this issue to some extent and is an efficient library for small, moderate, and large problems.

5 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


