Received: 26 June 2019 | Revised: 30 March 2020 | Accepted: 5 June 2020

DOI: 10.1002/nla.2321

RESEARCH ARTICLE

WILEY

Computing the matrix geometric mean: Riemannian versus Euclidean conditioning, implementation techniques, and a Riemannian BFGS method

Xinru Yuan[1] | Wen Huang[2] | P.-A. Absil[3] | Kyle A. Gallivan[1]

[1]Department of Mathematics, Florida State University, Tallahassee, 32306-4510, Florida, USA

[2]School of Mathematical Sciences, Fujian Provincial Key Laboratory of Mathematical Modeling and High-Performance Scientific Computing, Xiamen University, Fujian, China

[3]Department of Mathematical Engineering, ICTEAM Institute, Université catholique de Louvain, Louvain-la-Neuve, B-1348, Belgium

Correspondence
Wen Huang, School of Mathematical Sciences, Fujian Provincial Key Laboratory of Mathematical Modeling and High-Performance Scientific Computing, Xiamen University, 8 ZengCuoAn west road, SiMing District, Xiamen, Fujian, China.
Email: huwst08@gmail.com

Summary
This paper addresses the problem of computing the Riemannian center of mass of a collection of symmetric positive definite matrices. We show in detail that the condition number of the Riemannian Hessian of the underlying optimization problem is never very ill conditioned in practice, which explains why the Riemannian steepest descent approach has been observed to perform well. We also show theoretically and empirically that this property is not shared by the Euclidean Hessian. We then present a limited-memory Riemannian BFGS method to handle this computational task. We also provide methods to produce efficient numerical representations of geometric objects that are required for Riemannian optimization methods on the manifold of symmetric positive definite matrices. Through empirical results and a computational complexity analysis, we demonstrate the robust behavior of the limited-memory Riemannian BFGS method and the efficiency of our implementation when compared to state-of-the-art algorithms.

KEYWORDS
Karcher mean, limited-memory Riemannian BFGS method, Riemannian center of mass, Riemannian quasi-Newton methods, symmetric positive definite matrices

1 | INTRODUCTION

Symmetric positive definite (SPD) matrices are fundamental objects in various domains. For example, a three-dimensional diffusion tensor, that is, a 3×3 SPD matrix, is commonly used to model the diffusion behavior of the media in diffusion tensor magnetic resonance imaging.[1-3] In addition, representing images and videos with SPD matrices has shown promise for segmentation and recognition in several studies.[4-8] In these and similar applications, it is often of interest to average SPD matrices. Averaging is required, for example, to aggregate several noisy measurements of the same object. It also appears as a subtask in interpolation methods,[9] segmentation,[10,11] and clustering.[12] An efficient implementation of an averaging method is crucial for applications where the mean computation must be done many times. For example, in K-means clustering, one needs to compute the means of each cluster in each iteration.

A natural way to average over a collection of SPD matrices, $\{A_1, \dots, A_K\}$, is to take their arithmetic mean, that is, $G(A_1, \dots, A_K) = (A_1 + \dots + A_K)/K$. However, this is not appropriate in applications where invariance under inversion is required, that is, $G(A_1, \dots, A_K)^{-1} = G(A_1^{-1}, \dots, A_K^{-1})$. In addition, the arithmetic mean may cause a "swelling effect" that should be avoided in diffusion tensor imaging. Swelling is defined as an increase in the matrix determinant after averaging, see the work of Fletcher et al.,[2] for example. An alternative is to generalize the definition of geometric mean from scalars to matrices, which yields $G(A_1, \dots, A_K) = (A_1 \dots A_K)^{1/K}$. However, this generalized geometric mean is not invariant under permutation since matrices are not commutative in general. Ando et al.[13] introduced a list of fundamental properties, referred to as the ALM list, that a matrix "geometric" mean should possess. These properties are known to be important in many applications.[14-16] However, they do not uniquely define a mean for $K \geq 3$. There can be many different definitions of means that satisfy all the properties. The Riemannian center of mass proposed by Karcher,[17] with the affine-invariant metric (reviewed next) as the Riemannian metric, has been recognized as one of the most suitable means for SPD matrices in the sense that it satisfies all properties in the ALM list.[14,15] We will use the widespread name "Karcher mean" to refer to this mean.

1.1 | Karcher mean

Let $\mathcal{S}_{++}^n$ be the manifold of $n \times n$ SPD matrices. Since $\mathcal{S}_{++}^n$ is an open submanifold of the vector space of $n \times n$ symmetric matrices, its tangent space at point X, denoted by $\mathrm{T}_X\mathcal{S}_{++}^n$, can be identified as the set of $n \times n$ symmetric matrices. The manifold $\mathcal{S}_{++}^n$ becomes a Riemannian manifold when endowed with the affine-invariant metric, see the work of Pennec et al.,[11] given by

$$g_X(\xi_X, \eta_X) = \mathrm{trace}(\xi_X X^{-1} \eta_X X^{-1}), \tag{1}$$

where $\xi_X, \eta_X \in \mathrm{T}_X\mathcal{S}_{++}^n$.

The Karcher mean of $\{A_1, \dots, A_K\}$, also called the Riemannian center of mass, is the minimizer of the sum of squared distances

$$\mu = \arg\min_{X \in \mathcal{S}_{++}^n} F(X), \quad \text{with } F : \mathcal{S}_{++}^n \to \mathbb{R},\ X \mapsto \frac{1}{2K} \sum_{i=1}^{K} \delta^2(X, A_i), \tag{2}$$

where $\delta(p, q) = \| \log(p^{-1/2} q p^{-1/2}) \|_{\mathrm{F}}$ is the geodesic distance associated with Riemannian metric (1). It is proved by Karcher[17] that function F has a unique minimizer. Hence a point $\mu \in \mathcal{S}_{++}^n$ is a Karcher mean if it is a stationary point of F, i.e., grad $F(\mu) = 0$, where grad F denotes the Riemannian gradient of F under metric (1). However, a closed-form solution for problem (2) is unknown, in general, and for this reason, the Karcher mean is usually computed by iterative methods.

1.2 | Related work

Various methods have been used to compute the Karcher mean of SPD matrices. Most of them resort to the framework of Riemannian optimization (see, e.g., the work of Absil et al.,[18]) since problem (2) requires optimizing a function on a manifold. In particular, Jeuris et al.[19] present a survey of several optimization algorithms, including Riemannian versions of steepest descent, conjugate gradient, BFGS, and trust-region Newton methods. The authors conclude that the first-order methods, steepest descent and conjugate gradient, are the preferred choices for problem (2) in terms of time efficiency. The benefit of fast convergence of Newton's method and BFGS is nullified by their high computational costs per iteration, especially as the size of the matrices increases. It is also empirically observed[19] that the Riemannian metric yields much faster convergence for their tested algorithms compared with the induced Euclidean metric, which is given by $g_X(\eta_X, \xi_X) = \mathrm{trace}(\xi_X \eta_X)$.

A Riemannian version of the Barzilai-Borwein method (RBB) has been considered by Iannazzo et al.[20] Several stepsize selection rules have been investigated for the Riemannian steepest descent (RSD) method. A constant stepsize strategy is proposed by Rentmeesters et al.[21] and a convergence analysis is given. An adaptive stepsize selection rule based on the explicit expression of the Riemannian Hessian of the cost function F is studied in Algorithm 2 of the paper by Rentmeesters et al.,[22] which is actually the optimal stepsize for strongly convex function in Euclidean space, see Theorem 2.1.14 by Nesterov.[23] That is, the stepsize is chosen as $\alpha_k = 2/(M_k + L_k)$, where M_k and L_k are the lower and upper bounds on the eigenvalues of the Riemannian Hessian of F, respectively. A version of Newton's method for the Karcher mean computation is also provided in the work by Rentmeesters.[22] A Richardson-like iteration is derived and evaluated empirically by Bini et al.,[24] and is available in the Matrix Means Toolbox.[25] It is seen in Section 3.2 that the Richardson-like iteration is a steepest descent method with stepsize $\alpha_k = 1/L_k$. A majorization minimization algorithm has been given by Zhang.[26] This algorithm is easy to use in the sense that it is a parameter-free algorithm. However, it is usually not as efficient as other methods, as shown in Section 5.

1.3 | Contributions

First, by providing lower and upper bounds on the condition number of the Riemannian and Euclidean Hessians of the cost function (2), we give a theoretical explanation for the above-mentioned behavior of the Riemannian and Euclidean steepest descent algorithms for SPD Karcher mean computation. Then we provide a detailed description of a limited-memory Riemannian BFGS (LRBFGS) method for this mean computation problem. Riemannian optimization methods such as LRBFGS involve manipulation of geometric objects on manifolds, such as tangent vectors, evaluation of a Riemannian metric, retraction, and vector transport. We present detailed methods to produce efficient numerical representations of those objects on the $\mathcal{S}_{++}^n$ manifold. In fact, there are several alternatives to choose from for geometric objects on $\mathcal{S}_{++}^n$. We offer theoretical and empirical suggestions on how to choose between those alternatives for LRBFGS based on computational complexity analysis and numerical experiments. Our numerical experiments indicate that as a result, and in spite of the favorable bound on the Riemannian Hessian that ensures RSD to be an efficient method, the obtained LRBFGS method outperforms state-of-the-art methods on various instances of the problem. We also show that RBB is a special case of LRBFGS.

Another contribution of our work is to provide a C++ toolbox[*] for the SPD Karcher mean computation, which includes LRBFGS, RBFGS, RBB, and RSD. The toolbox relies on ROPTLIB, an object-oriented C++ library for optimization on Riemannian manifolds.[27] To the best of our knowledge, there is no other publicly available C++ toolbox for the SPD Karcher mean computation. Our previous work[28] provides a MATLAB implementation[†] for this problem. The Matrix Means Toolbox[25] developed by Bini et al.[24] is also written in MATLAB. As an interpreted language, MATLAB's execution efficiency is lower than compiled languages, such as C++. In addition, the timing measurements in MATLAB can be skewed by MATLAB's overhead, especially for small-size problems. As a result, we resort to C++ for efficiency and reliable timing.

Finally, we test the performance of LRBFGS on problems of various sizes and conditioning, and compare with the state-of-the-art methods mentioned above. The size of a problem is characterized by the number of matrices as well as the dimension of each matrix, and the conditioning of the problem is characterized by the condition numbers of matrices. It is shown empirically that LRBFGS is appropriate for large-size problems or ill-conditioned problems. Especially when one has little knowledge of the conditioning of a problem, LRBFGS becomes the method of choice since it is robust to problem conditioning and parameter setting. The numerical results also illustrate the speedup of using C++ vs. MATLAB, especially for small-size problems. It is observed that the C++ implementation is faster than MATLAB by a factor of 10 or more with the factor gradually reducing as the size of the problem increases.

The paper is organized as follows. Section 2 studies the conditioning of the objective function (2) under the Riemannian metric and the Euclidean metric. Section 3 presents the implementation techniques for $\mathcal{S}_{++}^n$ and computational complexity analysis. Detailed descriptions of the SPD Karcher mean computation methods considered (namely RL, RSD-QR, RBB, and LRBFGS) are given in Section 4. Numerical experiments are reported in Section 5. Conclusions are drawn in Section 6.

A preliminary version of some of the results presented in this paper can be found in the conference paper by Yuan et al.[28]

2 | CONDITIONING OF THE OBJECTIVE FUNCTION

The convergence speed of optimization methods depends on the conditioning of the Hessian of the cost function at the minimizer. Large values of condition number lead to slow convergence of optimization algorithms, especially for steepest descent methods. The choice of the metric has an important influence on the difficulty of an optimization problem via influencing the conditioning of the Hessian of the cost function. A good choice of metric may reduce the condition number of the Hessian.

[*] http://www.math.fsu.edu/~whuang2/papers/RMKMSPDM.htm
[†] http://www.math.fsu.edu/~whuang2/papers/ARLBACMGM.htm

Rentmeesters[22, inequality (3.29)] gives bounds on the eigenvalues of the Riemannian Hessian of the squared distance function $f_A(X) = \frac{1}{2}\delta^2(X, A)$ given $A \in \mathcal{S}^n_{++}$. On this basis, the bounds on the eigenvalues of the Riemannian Hessian of F can be obtained trivially. We summarize the results from Theorem 1 in the work of Rentmeesters[22] and, for completeness, we give the proof omitted by Rentmeesters.[22]

Theorem 1. *Let F be the objective function defined in problem (2) and $X \in \mathcal{S}^n_{++}$. Then the eigenvalues of the Riemannian Hessian of F at X are bounded by*

$$1 \leq \frac{\text{Hess } F(X)[\Delta X, \Delta X]}{\|\Delta X\|^2} \leq 1 + \frac{\log(\max_i \kappa_i)}{2}, \tag{3}$$

where κ_i denotes the condition number of matrix $X^{-1/2}A_iX^{-1/2}$ (or equivalently $L_x^{-1}A_iL_x^{-T}$ with $X = L_xL_x^T$ being the Cholesky decomposition of X).

Proof. The proof is a simple generalization from inequality (3.29) in the work of Rentmeesters,[22] which gives bounds on the eigenvalues of the Riemannian Hessian of the function $f_A(X)$ as

$$1 \leq \frac{\text{Hess } f_A(X)[\Delta X, \Delta X]}{\|\Delta X\|^2} \leq \frac{\log \kappa}{2} \coth\left(\frac{\log \kappa}{2}\right), \tag{4}$$

where κ is condition number of $X^{-1/2}AX^{-1/2}$. Notice that the objective function $F(X) = \frac{1}{K}\sum_{i=1}^K f_{A_i}(X)$. Thus, we have

$$1 \leq \frac{\text{Hess } F(X)[\Delta X, \Delta X]}{\|\Delta X\|^2} \leq \frac{1}{K}\sum_{i=1}^{K} \frac{\log \kappa_i}{2} \coth\left(\frac{\log \kappa_i}{2}\right). \tag{5}$$

Since $x\coth(x)$ is strictly increasing and bounded by $1 + x$ on $[0,\infty]$, the right hand side of inequality (5) is thus bounded by $1 + \log(\max_i \kappa_i)/2$. ∎

Theorem 1 implies that we cannot expect a very ill-conditioned Riemannian Hessian in practice. However, this is not the case when the Euclidean metric is used. In Theorem 2, we derive bounds on the condition number of the Euclidean Hessian of F at the minimizer. We need the following lemma before deriving the bounds.

Lemma 1. *Let $A \in \mathcal{S}^n_{++}$ be a symmetric positive definite matrix with eigenvalues satisfying $0 < \lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n$, and $\eta \in \mathbb{R}^{n\times n}$ be an $n \times n$ real symmetric matrix. Then, we have*

$$\max_{\eta=\eta^T} \frac{\text{tr}(A\eta A\eta)}{\|\eta\|_F^2} = \lambda_n^2, \text{ and } \min_{\eta=\eta^T} \frac{\text{tr}(A\eta A\eta)}{\|\eta\|_F^2} = \lambda_1^2. \tag{6}$$

Proof. Let $A = Q\Sigma Q^T$ be the eigenvalues decomposition of A, where $QQ^T = I$ and $\Sigma = \text{diag}(\lambda_1, \dots, \lambda_n)$. Then, we have

$$\frac{\text{tr}(A\eta A\eta)}{\|\eta\|_F^2} = \frac{\text{tr}(Q\Sigma Q^T\eta^T Q\Sigma Q^T\eta)}{\|Q^T\eta Q\|_F^2} = \frac{\text{tr}(\Sigma\tilde{\eta}^T\Sigma\tilde{\eta})}{\|\tilde{\eta}\|_F^2}, \tag{7}$$

where $\tilde{\eta} = Q^T\eta Q$. Notice that we can rewrite the trace term on the right hand of Equation (7) as $\text{tr}(\Sigma\tilde{\eta}^T\Sigma\tilde{\eta}) = \text{vec}(\tilde{\eta})^T(\Sigma \otimes \Sigma)\text{vec}(\tilde{\eta})$. From the min-max property of Rayleigh quotient, we have

$$\max_{\tilde{\eta}\in\mathbb{R}^{n\times n}} \frac{\text{vec}(\tilde{\eta})^T(\Sigma \otimes \Sigma)\text{vec}(\tilde{\eta})}{\text{vec}(\tilde{\eta})^T\text{vec}(\tilde{\eta})} = \lambda_n^2. \tag{8}$$

Using the fact that the Rayleigh quotient reaches its maximum value at $\tilde{\eta} = e_ne_n^T$ with $e_n = (0, \dots, 0, 1)^T$, which is symmetric, completes the proof.

Similarly, we have

$$\min_{\tilde{\eta}=\tilde{\eta}^T} \frac{\text{vec}(\tilde{\eta})^T(\Sigma \otimes \Sigma)\text{vec}(\tilde{\eta})}{\text{vec}(\tilde{\eta})^T\text{vec}(\tilde{\eta})} = \lambda_1^2. \tag{9}$$

∎

For notational simplicity, we use super script “E” and “R” to differentiate the Euclidean metric and the Riemannian metric.

Theorem 2. *Let $f : S^n_{++} \to \mathbb{R}$ be twice continuously differentiable and μ be a stationary point for f. Assume the largest and smallest eigenvalues of the Riemannian Hessian of f at μ are $\Lambda_{\max}$ and $\Lambda_{\min}$ respectively, that is,*

$$\Lambda_{\min} \leq \frac{\langle \mathrm{Hess}^{\mathrm{R}} f(\mu)[\eta], \eta \rangle^{\mathrm{R}}}{\langle \eta, \eta \rangle^{\mathrm{R}}} \leq \Lambda_{\max}. \tag{10}$$

Then the condition number of the Euclidean Hessian of f at μ, denoted by $\kappa(H^{\mathrm{E}})$, is bounded by

$$\frac{1}{\kappa(H^{\mathrm{R}})}\kappa^2(\mu) \leq \kappa(H^{\mathrm{E}}) \leq \kappa(H^{\mathrm{R}})\kappa^2(\mu), \tag{11}$$

where $\kappa(\mu)$ is the condition number of μ, and $\kappa(H^{\mathrm{R}}) = \Lambda_{\max}/\Lambda_{\min}$ is the condition number of the Riemannian Hessian of f at μ

Proof. Recall that the condition number of the Euclidean Hessian of f at μ can be expressed as

$$\kappa(H^{\mathrm{E}}) = \max_{\eta=\eta^T} \frac{\langle \mathrm{Hess}^{\mathrm{E}} f(\mu)[\eta], \eta \rangle^{\mathrm{E}}}{\langle \eta, \eta \rangle^{\mathrm{E}}} / \min_{\eta=\eta^T} \frac{\langle \mathrm{Hess}^{\mathrm{E}} f(\mu)[\eta], \eta \rangle^{\mathrm{E}}}{\langle \eta, \eta \rangle^{\mathrm{E}}}, \tag{12}$$

and that of the Riemannian Hessian can be written in a similar way.

When μ is a critical point of F, $\langle \mathrm{Hess}\ F(\mu)[\eta], \eta \rangle$ is independent of the metric, see the work of Absil et al.[18, section 5.5] for details. Therefore, we have

$$\langle \mathrm{Hess}^{\mathrm{R}} f(\mu)[\eta], \eta \rangle^{\mathrm{R}} = \langle \mathrm{Hess}^{\mathrm{E}} f(\mu)[\eta], \eta \rangle^{\mathrm{E}}. \tag{13}$$

It follows that

$$\frac{\langle \mathrm{Hess}^{\mathrm{E}} f(\mu)[\eta], \eta \rangle^{\mathrm{E}}}{\langle \eta, \eta \rangle^{\mathrm{E}}} = \frac{\langle \mathrm{Hess}^{\mathrm{R}} f(\mu)[\eta], \eta \rangle^{\mathrm{R}}}{\langle \eta, \eta \rangle^{\mathrm{R}}} \cdot \frac{\langle \eta, \eta \rangle^{\mathrm{R}}}{\langle \eta, \eta \rangle^{\mathrm{E}}}. \tag{14}$$

By assumption, we have

$$\Lambda_{\min} \leq \frac{\langle \mathrm{Hess}^{\mathrm{R}} f(\mu)[\eta], \eta \rangle^{\mathrm{R}}}{\langle \eta, \eta \rangle^{\mathrm{R}}} \leq \Lambda_{\max}. \tag{15}$$

Assume that the eigenvalues of μ are $0 < \lambda_1 \leq \ldots \leq \lambda_n$, and then the eigenvalues of μ^{-1} are $0 < 1/\lambda_n \leq \ldots \leq 1/\lambda_1$. From Lemma 1, we have

$$\max_{\eta=\eta^T} \frac{\langle \eta, \eta \rangle^{\mathrm{R}}}{\langle \eta, \eta \rangle^{\mathrm{E}}} = \frac{1}{\lambda_1^2} \text{ and } \min_{\eta=\eta^T} \frac{\langle \eta, \eta \rangle^{\mathrm{R}}}{\langle \eta, \eta \rangle^{\mathrm{E}}} = \frac{1}{\lambda_n^2}. \tag{16}$$

Combining inequality (15) and Equation (16) gives

$$\frac{\Lambda_{\min}}{\lambda_1^2} \leq \max_{\eta=\eta^T} \frac{\langle \mathrm{Hess}^{\mathrm{E}} f(\mu)[\eta], \eta \rangle^{\mathrm{E}}}{\langle \eta, \eta \rangle^{\mathrm{E}}} \leq \frac{\Lambda_{\max}}{\lambda_1^2}, \tag{17}$$

and

$$\frac{\Lambda_{\min}}{\lambda_n^2} \leq \min_{\eta=\eta^T} \frac{\langle \mathrm{Hess}^{\mathrm{E}} f(\mu)[\eta], \eta \rangle^{\mathrm{E}}}{\langle \eta, \eta \rangle^{\mathrm{E}}} \leq \frac{\Lambda_{\max}}{\lambda_n^2}. \tag{18}$$

Dividing inequality (17) by (18) gives us the lower and upper bounds on the condition number of the Euclidean Hessian of f at stationary point μ:

$$\frac{\Lambda_{\min}}{\Lambda_{\max}} \frac{\lambda_n^2}{\lambda_1^2} \leq \kappa(H^{\mathrm{E}}) \leq \frac{\Lambda_{\max}}{\Lambda_{\min}} \frac{\lambda_n^2}{\lambda_1^2}. \tag{19}$$

That is,

$$\frac{1}{\kappa(H^{\mathrm{R}})}\kappa^2(\mu) \leq \kappa(H^{\mathrm{E}}) \leq \kappa(H^{\mathrm{R}})\kappa^2(\mu). \tag{20}$$

■

For the cost function F in (2), it is seen from Theorem 2 that the condition number of the Euclidean Hessian at the minimizer (stationary point) is bounded below by the square of the condition number of the minimizer scaled by the reciprocal of the condition number of the Riemannian Hessian of F. Hence when the minimizer is ill-conditioned, the Euclidean Hessian of F at the minimizer is ill-conditioned as well, which will slow down the optimization methods. Our numerical experiments in Section 5.5 demonstrate this expectation.

3 | Implementation for the $\mathcal{S}_{++}^n$ manifold

This section is devoted to the implementation details of the required objects for Riemannian optimization methods on the SPD Karcher mean computation problem. Manifold-related objects include tangent vectors, the Riemannian metric, isometric vector transport, and retraction. Problem-related objects include the cost function and Riemannian gradient evaluations. As an extension of our previous work,[28] we also provide a floating point operation (flop) count[‡] for most operations.

3.1 | Representations of a tangent vector and the Riemannian metric

The $\mathcal{S}_{++}^n$ manifold can be viewed as a submanifold of $\mathbb{R}^{n\times n}$, and its tangent space at X is the set of symmetric matrices, i.e., $\mathrm{T}_X\mathcal{S}_{++}^n = \{S \in \mathbb{R}^{n\times n} | S = S^T\}$. The dimension of $\mathcal{S}_{++}^n$ is $d = n(n+1)/2$. Thus, a tangent vector η_X in $\mathrm{T}_X\mathcal{S}_{++}^n$ can be represented either by an n^2-dimensional vector in Euclidean space $\mathcal{E}$, or a d-dimensional vector of coordinates in a given basis B_X of $\mathrm{T}_X\mathcal{S}_{++}^n$. The n^2-dimensional representation is called the extrinsic approach, and the d-dimensional one is called the intrinsic approach. For simplicity, we use w to denote the dimension of the embedding space, that is, $w = n^2$.

The computational benefits of using intrinsic representation are addressed in the work of Huang et al.:[30,31] (i) Working in d-dimension reduces the computational complexity of linear operations on the tangent space. (ii) There exists an isometric vector transport, called vector transport by parallelization, whose intrinsic implementation is simply the identity. (iii) The Riemannian metric can be reduced to the Euclidean metric. However, the intrinsic representation requires a basis of tangent space, and in order to obtain the computational benefits mentioned above, it must be orthonormal. Hence, if a manifold admits a smooth field of orthonormal tangent space bases with acceptable computational complexity, the intrinsic representation often leads to a very efficient implementation. This property holds for $\mathcal{S}_{++}^n$ as shown next.

The orthonormal basis B_X of $\mathrm{T}_X\mathcal{S}_{++}^n$ that we select is given by

$$\{Le_ie_i^TL^T : i = 1, \ldots, n\} \cup \left\{ \frac{1}{\sqrt{2}}L(e_ie_j^T + e_je_i^T)L^T,\ i<j,\ i = 1, \ldots, n,\ j = 1, \ldots, n \right\}, \tag{21}$$

where $X = LL^T$ denotes the Cholesky decomposition, and $\{e_1, \ldots, e_n\}$ is the standard basis of n-dimensional Euclidean space. Another choice is to use the matrix square root $X^{1/2}$ instead of Cholesky decomposition of X, whose computation costs more.[32] It is easy to verify the orthonormality of B_X under the Riemannian metric (1), that is, $B_X^\flat B_X = I_{d\times d}$ for all $X \in \mathcal{S}_{++}^n$. (The notation $a^\flat$ denotes the function $a^\flat : \mathrm{T}_X\mathcal{M} \to \mathbb{R} : v \mapsto g_X(a, v)$, where g stands for the Riemannian metric (1).) We assume throughout the paper that B_X stands for our selected orthonormal basis of $\mathrm{T}_X\mathcal{S}_{++}^n$ defined in (21).

Let η_X be a tangent vector in $\mathrm{T}_X\mathcal{S}_{++}^n$ and v_X be its intrinsic representation. We define function $E2D_X : \eta_X \mapsto v_X = B_X^\flat \eta_X$ that maps the extrinsic representation to the intrinsic representation. Using the orthonormal basis defined in (21), the intrinsic representation of η_X is obtained by taking the diagonal elements of $L^{-1}\eta_X L^{-T}$, and its upper triangular elements row-wise and multiplied by $\sqrt{2}$. A detailed description of function $E2D$ is given in Algorithm 1. The number of flops for each step is given on the right of the algorithm.

[‡]See the work of Golub et al.[29, section 1.2.4]

Since B_X forms an orthonormal basis of $\mathrm{T}_X\mathcal{S}_{++}^n$, the Riemannian metric (1) reduces to the Euclidean metric under the intrinsic representation, that is,

$$\tilde{g}_X(v_X, u_X) := g_X(\eta_X, \xi_X) = g_X(B_X v_X, B_X u_X) = v_X^T u_X, \tag{22}$$

where $\eta_X = B_X v_X$, $\xi_X = B_X u_X \in \mathrm{T}_X\mathcal{S}_{++}^n$. The evaluation of (22) requires $2d$ flops, which is cheaper than the evaluation of (1).

For the intrinsic approach, retractions (see Section 3.2) require mapping the intrinsic representation back to the extrinsic representation, which may require extra computation. Let function $D2E_X : v_X \mapsto \eta_X = B_X v_X$ denote this mapping. In practice, the function $D2E_X$ using basis (21) is described in Algorithm 2.

Algorithm 1. Compute $E2D_X(\eta_X)$

```
Require: $X = LL^T \in \mathcal{S}_{++}^n$, $\eta_x \in \mathrm{T}_X\mathcal{S}_{++}^n$.
1: Compute $Y = L^{-1}\eta_X$ by solving linear system $LY = \eta_X$;        $\triangleright$ # $n^3$
2: $Y \leftarrow Y^T$ (i.e., $Y = \eta_X L^{-T}$) ;
3: Compute $Z = L^{-1}\eta_X L^{-T}$ by solving linear system $LZ = Y$;     $\triangleright$ # $n^3$
4: return $v_X = (z_{11}, \dots, z_{nn}, \sqrt{2}z_{12}, \dots, \sqrt{2}z_{1n}, \sqrt{2}z_{23}, \dots, \sqrt{2}z_{2n}, \dots, \sqrt{2}z_{(n-1)n})^T$;     $\triangleright$ # $d$
```

Algorithm 2. Compute $D2E_X(v_X)$

```
Require: $X = LL^T \in \mathcal{S}_{++}^n$, $v_x \in \mathbb{R}^{n(n+1)/2}$.
1: $\eta_{ii} = v_X(i)$ for $i = 1, \dots, n$;        $\triangleright$ # $n$
2: $k = n + 1$;
3: for $i = 1, \dots, n$ do        $\triangleright$ # $n^2 - n(n+1)/2$
4:     for $j = i + 1, \dots, n$ do
5:         $\eta_{ij} = v_X(k)$ and $\eta_{ji} = v_X(k)$;
6:         $k = k + 1$;
7:     end for
8: end for
9: return $L\eta L^T$;        $\triangleright$ # $2n^3$
```

3.2 | Retraction and vector transport

The concepts of retraction and vector transport can be found in the work of Absil et al.[18] A retraction is a smooth mapping R from the tangent bundle $\mathrm{T}\mathcal{M}$ onto $\mathcal{M}$ such that (i) $R(0_x) = x$ for all $x \in \mathcal{M}$ (where 0_x denotes the origin of $\mathrm{T}_x\mathcal{M}$) and (ii) $\frac{d}{dt}R(t\xi_x)|_{t=0} = \xi_x$ for all $\xi_x \in \mathrm{T}_x\mathcal{M}$. A vector transport $\mathcal{T} : \mathrm{T}\mathcal{M} \oplus \mathrm{T}\mathcal{M} \to \mathrm{T}\mathcal{M}$, $(\eta_x, \xi_x) \mapsto \mathcal{T}_{\eta_x}\xi_x$ with associated retraction R is a smooth mapping such that, for all (x, η_x) in the domain of R and ξ_x, $\zeta_x \in \mathrm{T}_x\mathcal{M}$, it holds that (i) $\mathcal{T}_{\eta_x}\xi_x \in \mathrm{T}_{R(\eta_x)}\mathcal{M}$, (ii) $\mathcal{T}_{0_x}\xi_x = \xi_x$, (iii) $\mathcal{T}_{\eta_x}$ is a linear map. Some methods, such as RBFGS in the work of Huang et al.(33, algorithm 1) and LRBFGS, require the vector transport to be isometric, that is, $g_{R(\eta_x)}(\mathcal{T}_{S_{\eta_x}}\xi_x, \mathcal{T}_{S_{\eta_x}}\zeta_x) = g_x(\xi_x, \zeta_x)$. Throughout the paper, we use the notation $\mathcal{T}_S$ for isometric vector transport.

The choice of retraction and vector transport is a key step in the design of efficient Riemannian optimization algorithms. The exponential mapping is a natural choice for retraction. When $\mathcal{S}_{++}^n$ is endowed with the Riemannian metric (1), the exponential mapping is given by, see the work of Fletcher et al.,[34]

$$Exp_X(\eta_X) = X^{1/2}\exp(X^{-1/2}\eta_X X^{-1/2})X^{1/2}, \tag{23}$$

for all $X \in \mathcal{S}_{++}^n$ and $\eta_X \in \mathrm{T}_X\mathcal{S}_{++}^n$. In practice, the exponential mapping (23) is expensive to compute. The exponential of matrix M is computed as $\exp(M) = U\exp(\Sigma)U^T$, with $M = U\Sigma U^T$ being the eigenvalue decomposition. Obtaining Σ and U by Golub-Reinsch algorithm requires $12n^3$ flops, see figure 8.6.1 in the work of Golub et al.[29] Hence the evaluation of

$\exp(M)$ requires $14n^3$ flops in total. More importantly, when computing the matrix exponential $\exp(M)$, eigenvalues of large magnitude can lead to numerical difficulties. Jeuris et al.[19] proposed a retraction

$$R_X(\eta_X) = X + \eta_X + \frac{1}{2}\eta_X X^{-1}\eta_X, \tag{24}$$

which is a second order approximation to the exponential mapping (23). Retraction (24) is cheaper to compute and requires $3n^3 + o(n^3)$ flops, and tends to avoid numerical overflow. An important property of retraction (24) is stated in Proposition 1.

Proposition 1. *Retraction $R_X(\eta)$ defined in (24) remains symmetric positive definite for all $X \in \mathcal{S}_{++}^n$ and $\eta \in \mathrm{T}_X\mathcal{S}_{++}^n$*

Proof. For all $v \neq 0, X \in \mathcal{S}_{++}^n$, and $\eta \in \mathrm{T}_X\mathcal{S}_{++}^n$, we have

$$\begin{aligned} v^T R_X(\eta)v &= \frac{1}{2}v^T(X + 2\eta + \eta X^{-1}\eta)v + \frac{1}{2}v^T Xv \\ &= \frac{1}{2}v^T(X^{1/2} + \eta X^{-1/2})(X^{1/2} + \eta X^{-1/2})^T v + \frac{1}{2}v^T Xv > 0. \end{aligned} \tag{25}$$

■

Another retraction that can be computed efficiently is the first order approximation to (23), that is,

$$R_X(\eta_X) = X + \eta_X. \tag{26}$$

In fact, retraction (26) is the exponential mapping when $\mathcal{S}_{++}^n$ is endowed with the Euclidean metric. However, the result of retraction (26) is not guaranteed to be positive definite. Therefore one must be careful when using this Euclidean retraction. One remedy is to reduce the stepsize when necessary. The Richardson-like iteration[24] is a steepest descent method using Euclidean retraction (26).

Parallel translation is a particular instance of vector transport. The parallel translation on $\mathcal{S}_{++}^n$ is given by, see the work of Fletcher et al.,[34]

$$\mathcal{T}_{p_{\xi_X}}(\eta_X) = X^{1/2}\exp\left(\frac{X^{-1/2}\xi_X X^{-1/2}}{2}\right)X^{-1/2}\eta_X X^{-1/2}\exp\left(\frac{X^{-1/2}\xi_X X^{-1/2}}{2}\right)X^{1/2}. \tag{27}$$

The computation of parallel translation involves the matrix exponential, which is computationally expensive. Note, however, that if parallel translation is used together with the exponential mapping (23), the most expensive exponential computation can be shared by rewriting (27) and (23) as shown in Algorithm 4. Even so, the matrix exponential computation is still required. We thus resort to another vector transport.

Recently, Huang et al.(30, Section 2.3.1) proposed a novel way to construct an isometric vector transport, called vector transport by parallelization. It is defined by

$$\mathcal{T}_S = B_Y B_X^\flat, \tag{28}$$

where B_X and B_Y are orthonormal bases of $\mathrm{T}_X\mathcal{S}_{++}^n$ and $\mathrm{T}_Y\mathcal{S}_{++}^n$ defined in (21) respectively. Let $v_X = B_X^\flat\eta_X$ be the intrinsic representation of η_X. Then, the intrinsic approach of (28), denoted by $\mathcal{T}_S^d$, is given by

$$\mathcal{T}_S^d v_X = B_Y^\flat\mathcal{T}_S\eta_X = B_Y^\flat B_Y B_X^\flat B_X v_X = v_X \tag{29}$$

That is, the intrinsic representation of vector transport by parallelization is simply the identity, which is the cheapest vector transport one can expect.

Another possible choice for the vector transport is the identity mapping: $\mathcal{T}_{id_{\xi_X}}(\eta_X) = \eta_X$. However, vector transport $\mathcal{T}_{id}$ is not applicable to the LRBFGS (33, algorithm 2) since it is not isometric under Riemannian metric (1).

Given a retraction, Huang et al. (33, section 2) provide a method to construct an isometric vector transport such that the pair satisfies the locking condition

$$\mathcal{T}_{S_{\xi_X}}\xi_X = \beta\mathcal{T}_{R_{\xi_X}}\xi_X,$$

where $\beta = \frac{\|\xi_X\|}{\|\mathcal{T}_{R_{\xi_X}}\xi_X\|}$ and $\mathcal{T}_R$ denotes the differentiated retraction. Such vector transport, denoted by $\mathcal{T}_L$, is given by

$$\mathcal{T}_{L_{\xi_X}}\eta_X = B_Y\left(I - \frac{2v_2v_2^T}{v_2^Tv_2}\right)\left(I - \frac{2v_1v_1^T}{v_1^Tv_1}\right)B_1^\flat\eta_X, \tag{30}$$

where $v_1 = B_X^\flat\xi_X - z$, $v_2 = z - \beta B_Y^\flat\mathcal{T}_{R_{\xi_X}}\xi_X$, $\beta = \|\xi_X\|/\|\mathcal{T}_{R_{\xi_X}}\xi_X\|$. z can be any tangent vector satisfying $\|z\| = \|B_1^\flat\xi_X\| = \|\beta B_2^\flat\mathcal{T}_{R_{\xi_X}}\xi_X\|$. $z = -B_1^\flat\xi_X$ and $z = -\beta B_2^\flat\xi_X$ are natural choices. The intrinsic representation of (30) is given by

$$\mathcal{T}_L^d v_X = B_Y^\flat B_Y\left(I - \frac{2v_2v_2^T}{v_2^Tv_2}\right)\left(I - \frac{2v_1v_1^T}{v_1^Tv_1}\right)B_1^\flat\eta_X \tag{31}$$

$$= \left(I - \frac{2v_2v_2^T}{v_2^Tv_2}\right)\left(I - \frac{2v_1v_1^T}{v_1^Tv_1}\right)v_X. \tag{32}$$

The evaluation of the intrinsic vector transport (32) requires $12d$ flops, that is, $6n^2$, which is higher than the complexity of (29) without imposing the locking condition.

3.3 | Riemannian gradient of the sum of squared distances function

The cost function (2) can be rewritten as

$$f(X) = \frac{1}{2K}\sum_{i=1}^{K}\|\log(A_i^{-1/2}XA_i^{-1/2})\|_F^2 = \frac{1}{2K}\sum_{i=1}^{K}\|\log(L_{A_i}^{-1}XL_{A_i}^{-T})\|_F^2, \tag{33}$$

where $A_i = L_{A_i}L_{A_i}^T$. We use Cholesky decomposition rather than the matrix square root due to computational efficiency. The matrix logarithm is computed in a similar way as the exponential, that is, $\log(M) = U\log(\Sigma)U^T$ with $M = U\Sigma U^T$ being the eigenvalue decomposition. So the evaluation of (33) is $18Kn^3$ flops.

The Riemannian gradient of the cost function F in (2) is given by, see the work of Karcher,[17]

$$\text{grad } F(X) = -\frac{1}{K}\sum_{i=1}^{K}Exp_X^{-1}(A_i), \tag{34}$$

where $Exp_X^{-1}(Y)$ is the log-mapping, that is, the inverse exponential mapping. On S_{++}^n, the log-mapping is computed as

$$Exp_X^{-1}(Y) = X^{1/2}\log(X^{-1/2}YX^{-1/2})X^{1/2} = \log(YX^{-1})X. \tag{35}$$

Note that the computational complexity of the Riemannian gradient is less than that conveyed in formula (35) since the most expensive logarithm computation is already available from the evaluation of the cost function at X. Specifically, each term in (34) is computed as $-Exp_X^{-1}(A_i) = -\log(A_iX^{-1})X = \log(XA_i^{-1})X = L_{A_i}\log(L_{A_i}^{-1}XL_{A_i}^{-T})L_{A_i}^{-1}X$, and the term $\log(L_{A_i}^{-1}XL_{A_i}^{-T})$ is available from the evaluation of the cost function $F(X)$ in (33). Hence the computation of gradient requires $5Kn^3$ flops if $\log(L_{A_i}^{-1}XL_{A_i}^{-T})$ is given.

4 | DESCRIPTION OF THE SPD KARCHER MEAN COMPUTATION METHODS

In this section, we present the algorithms for SPD Karcher mean computation, including a limited-memory Riemannian BFGS (LRBFGS),[33] a Riemannian Barzilai-Borwein (RBB),[20] a RSD with stepsize selection rule proposed by Q. Rentmeesters (RSD-QR),[22] and a Richardson-like iteration (RL).[24] All of the algorithms are retraction-based methods, namely, an iterate x_k on a manifold $\mathcal{M}$ is updated by

$$x_{k+1} = R_{x_k}(\alpha_k \eta_k), \tag{36}$$

where R is a retraction on $\mathcal{M}$, $\eta_k \in \mathrm{T}_{x_k}\mathcal{M}$ is the search direction and $\alpha_k \in \mathbb{R}$ denotes the stepsize.

For the steepest descent method, the search direction in (36) is taken as the negative gradient, that is, $\eta_k = -\mathrm{grad}\, f(x_k)$. RSD-QR for the SPD Karcher mean computation is summarized in Algorithm 3 based on the work of Rentmeesters (22, algorithm 2). The differences between RSD-QR and RL are the choice of stepsize strategy in Step 11 and the retraction in Step 13 of Algorithm 3. For RL, the stepsize is taken as $\alpha_{RL} = 1/\Delta$, where Δ is the upper bound on the eigenvalues of the Hessian of the cost function as computed in Step 10, and the Euclidean retraction (26) is used. The number of flops required per iteration is $22Kn^3 + o(Kn^3)$. For RSD-QR, the chosen stepsize is $\alpha_{QR} = 2/(U + \Delta)$, where $U = 1$ is the lower bound on the eigenvalues of the Hessian of the cost function. It is easy to verify that $1/\Delta \le 2/(1 + \Delta)$, with equality when $\Delta = 1$. Since the eigenvalues of the Hessian of the cost function are bounded by U and Δ, then $\Delta = 1$ implies that all the eigenvalues of the Hessian are exactly 1. So $\alpha_{RL} = \alpha_{QR}$ if and only if the Hessian of the cost function is the identity matrix, and we have $\alpha_{RL} < \alpha_{QR}$ in general. The exponential mapping (23) is used by RSD-QR.[22] In practice, we use retraction (24), since the exponential mapping contains matrix exponential evaluation, and it turns out to be a problem if the eigenvalues of matrices in some intermediate iterations become too large, resulting in numerical overflow. Then each iteration in RSD-QR needs $22Kn^3 + 4/3n^3 + o(Kn^3)$ flops. Even though RSD-QR is slightly more expensive per iteration than RL, it will be seen in our experiments to require fewer iterations to achieve a desired tolerance, to perform very well on small-size problems in terms of time efficiency, and to consistently outperform RL in various situations.

Algorithm 3. RSD for the SPD Karcher mean computation

```
Require: $A_i = L_{A_i} L_{A_i}^T$; tolerance for stopping criterion $\epsilon$; initial iterate $x_0 \in \mathcal{S}_{++}^n$;
 1: $k = 0$;
 2: while $\|\mathrm{grad} f(x_k)\| < \epsilon$ do
 3:     for $i = 1, \dots, K$ do
 4:         Compute $M_i = L_{A_i}^{-1} x_k L_{A_i}^{-T}$;                                ▷ # $2n^3$
 5:         Compute $M_i = U \Sigma U^{-1}$ and set $\lambda = \mathrm{diag}(\Sigma)$;            ▷ # $12n^3$
 6:         Compute the condition number $c_i = \max(\lambda)/\min(\lambda)$;        ▷ # 1
 7:         Compute $K_i = U \log(\Sigma) U^{-1}$;                                  ▷ # $4n^3$
 8:         Compute $G_i = L_{A_i} K_i L_{A_i}^{-1} x_k$;                            ▷ # $4n^3$
 9:     end for
10:     Compute the upper bound on the eigenvalues of the Hessian of the cost
        function: $\Delta = \frac{1}{K} \sum_{j=1}^{K} \frac{\log c_j}{2} \coth(\frac{\log c_j}{2})$;        ▷ # $5K$
11:     Compute stepsize $\alpha_k = \alpha(\Delta) := \alpha_{QR} = \frac{2}{U+\Delta}$;
12:     Compute $\mathrm{grad} f(x_k) = \frac{1}{K} \sum_{i=1}^{K} G_i$;                     ▷ # $(K+1)n^2$
13:     Compute $x_{k+1} = R_{x_k}(-\alpha_k \mathrm{grad} f(x_k))$;
14:     $k = k + 1$;
15: end while
```

RBB also belongs to the class of steepest descent methods, combined with a stepsize that makes implicit use of second order information of the cost function, see the work of di Serafino et al. and Iannazzo et al.[20,35] for details. The two most frequently used versions of the BB stepsize are

$$\alpha_{k+1}^{\mathrm{BB1}} = \frac{g(s_k, s_k)}{g(s_k, y_k)}, \tag{37}$$

$$\alpha_{k+1}^{\mathrm{BB2}} = \frac{g(s_k, y_k)}{g(y_k, y_k)}, \tag{38}$$

where $s_k = \mathcal{T}_{\alpha_k\eta_k}(\alpha_k\eta_k)$, $y_k = \mathrm{grad}\, f(x_{k+1}) - \mathcal{T}_{\alpha_k\eta_k}(\mathrm{grad}\, f(x_k))$, and $\mathrm{g}(s_k, y_k) > 0$. An adaptive BB stepsize selection rule given by Frassoldati,[36] denoted by $\mathrm{ABB}_{\min}$, is defined as

$$\alpha_{k+1}^{\mathrm{ABB}_{\min}} = \begin{cases} \min\left\{ \alpha_j^{\mathrm{BB2}} : j = \max(1, k - m_a), \dots, k \right\}, & \text{if } \frac{\alpha_{k+1}^{\mathrm{BB2}}}{\alpha_{k+1}^{\mathrm{BB1}}} < \tau \\ \alpha_{k+1}^{\mathrm{BB1}}, & \text{otherwise} \end{cases}, \tag{39}$$

where m_a is a nonnegative integer and $\tau \in (0, 1)$.

In the work of Iannazzo,[20] RBB based on (37) has been applied to the SPD Karcher mean computation. We summarize that implementation in Algorithm 4, which uses an extrinsic representation of a tangent vector, exponential mapping (23), parallel translation (27) and α^{BB2} in (38). Algorithm 5 states our implementation of RBB using the intrinsic representation approach, retraction (24) and vector transport by parallelization (29). Algorithms 4 and 5 (and the subsequent algorithms) contain a nonmonotone line search procedure that is activated when the norm of the gradient reaches a prescribed relative value. As in the work of Iannazzo,[20] we have observed that the nonmonotone mechanism does not speed up the algorithm. However, the nonmonotone mechanism turns out to be a convenient fix to the inexact arithmetic issues that arise in the backtracking condition (e.g., Step 7 of Algorithm 4) when $|f(\tilde{x}_k) - f(x_k)|/|f(x_k)|$ gets close to the machine epsilon. We present the number of flops for each step on the right-hand side of the algorithms, except problem-related operations, i.e., function, gradient evaluations and line search procedure. Note that Step 6 and Step 11 in Algorithm 4 share the common term $\exp(\alpha_k x_k^{-1}\eta_k)$, which dominates the computational time and is only computed once. Having $w = n^2$ and $d = n(n+1)/2$, the number of flops per iteration for Algorithm 4 and Algorithm 5 are $97n^3/3 + o(n^3)$ and $22n^3/3 + o(n^3)$ respectively. The number of flops required by Algorithm 5 is smaller than that of Algorithm 4, and the computational efficiency mainly comes from the choice of retraction and the fact that the Riemannian metric reduces to the Euclidean metric when the intrinsic representation of a tangent vector is used.

Algorithm 4. RBB for the SPD Karcher mean computation using extrinsic representation[20]

```
Require: backtracking reduction factor $\varrho \in (0,1)$; Armijo parameter $\delta \in (0,1)$; tolerance for stopping criterion $\epsilon$; initial
    iterate $x_0 \in \mathcal{S}_{++}^n$; the first stepsize $\alpha_0^{BB}$; an integer $\varpi$;
 1: $k = 0$;
 2: Compute $f(x_k)$, $\mathrm{grad} f(x_k)$, and let $S_f = \emptyset$;
 3: while $\|\mathrm{grad} f(x_k)\| < \epsilon$ do
 4:     Set stepsize $\alpha_k = \alpha_k^{BB}$;
 5:     Set $\eta_k = -\mathrm{grad} f(x_k)$;                                                  $\triangleright$ # $w$
 6:     Compute $\tilde{x}_k = x_k \exp(\alpha_k x_k^{-1}\eta_k)$;                               $\triangleright$ # $55n^3/3$
 7:     If
            $f(\tilde{x}_k) \leq \max\left(f(x_k) \cup S_f\right) + \delta\alpha_k \mathrm{g}(\mathrm{grad} f(x_k), \eta_k)$,
        then set $x_{k+1} = \tilde{x}_k$ and go to Step 9;
 8:     Set $\alpha_k = \varrho\alpha_k$ and go to Step 6;
 9:     Compute $\mathrm{grad} f(x_{k+1})$;
10:     Set $S_f \leftarrow S_f \cup \{f(x_{k+1})\}$; If $|S_f| < \varpi$, then remove the oldest entry in$\sim S_f$;
11:     Compute $s_k = \alpha_k\eta_k \exp(\alpha_k x_k^{-1}\eta_k)$, $y_k = \mathrm{grad} f(x_{k+1}) + \eta_k \exp(\alpha_k x_k^{-1}\eta_k)$;   $\triangleright$ # $2n^3 + n^2$
12:     Compute $\alpha_{k+1}^{BB} = \mathrm{g}(s_k, y_k)/\mathrm{g}(y_k, y_k)$;                  $\triangleright$ # $12n^3$
13:     Set $\alpha_{k+1}^{BB} = \min\{\alpha_{max}, \max\{\epsilon, \alpha_{k+1}^{BB}\}\}$ if $\mathrm{g}(s_k, y_k) < 0$; otherwise, $\alpha_{k+1}^{BB} = \alpha_{max}$;
14:     $k = k + 1$;
15: end while
```

Our previous work[28] tailors the LRBFGS[33, Algorithm 2] to the SPD Karcher mean computation problem. The limited-memory BFGS method is based on the BFGS method which stores and transports the inverse Hessian approximation as a dense matrix. Specifically, the search direction in RBFGS is $\eta_k = -\mathcal{B}_k^{-1}\mathrm{grad}\, f(x_k)$, where $\mathcal{B}_k$ is a linear operator that approximates the action of the Hessian on $\mathrm{T}_{x_k}\mathcal{M}$. $\mathcal{B}_k$ requires a rank-two update at each iteration, see the work by Huang et al. (33, algorithm 1) for the update formula. Unlike BFGS, the limited-memory version of BFGS stores only some relatively small number of vectors that represent the approximation implicitly. Therefore LRBFGS is appropriate for large-size problems, due to its benefit in reducing storage requirements and computation time per iteration.

Algorithm 5. RBB for the SPD Karcher mean computation using intrinsic representation and vector transport by parallelization

```
Require: backtracking reduction factor $\varrho \in (0,1)$; Armijo parameter $\delta \in (0,1)$; tolerance for stopping criterion $\epsilon$; initial iterate $x_0 \in \mathcal{M}$; the first stepsize $\alpha_0^{BB}$; an integer $\varpi$;
 1: $k = 0$; and let $S_f = \emptyset$;
 2: Compute $\mathrm{grad} f(x_k)$;
 3: Compute $x_k = L_k L_k^T$;                                              ▷ $\#n^3/3$
 4: Compute $\mathrm{gf}_k^d = E2D_{x_k}(\mathrm{grad} f(x_k))$ by Algorithm 1;      ▷ $\#\ 2n^3 + d$
 5: while $\|\mathrm{gf}_k^d\| < \epsilon$ do
 6:     Set stepsize $\alpha_k = \alpha_k^{BB}$;
 7:     Set $\eta_k = -\mathrm{gf}_k^d$;                                       ▷ $\#d$
 8:     Compute $\eta_k^w = D2E_{x_k}(\eta_k)$ by Algorithm 2;                  ▷ $\#2n^3 + n(n+1)/2$
 9:     Compute $\tilde{x}_k = R_{x_k}(\alpha_k \eta_k^w)$ using (24) ;         ▷ $\#\ 3n^3 + 3n^2$
10:     If
            $$f(\tilde{x}_k) \le \max\left(f(x_k) \cup S_f\right) + \delta \alpha_k \eta_k^T \mathrm{gf}_k^d,$$
        then set $x_{k+1} = \tilde{x}_k$ and go to Step 12;
11:     Set $\alpha_k = \varrho \alpha_k$ and go to Step 9;
12:     Compute $\mathrm{grad} f(x_{k+1})$;
13:     Set $S_f \leftarrow S_f \cup \{f(x_{k+1})\}$; if $|S_f| < \varpi$, then remove the oldest entry in $S_f$;
14:     Compute $x_{k+1} = L_{k+1} L_{k+1}^T$;                                  ▷ $\#n^3/3$
15:     Compute $\mathrm{gf}_k^d = E2D_{x_k}(\mathrm{grad} f(x_k))$ by Algorithm 1;   ▷ $\#\ 2n^3 + d$
16:     Compute $s_k = \alpha_k \eta_k, y_k = \mathrm{gf}_{k+1}^d - \mathrm{gf}_k^d$;        ▷ $\#2d$
17:     Compute $\alpha_{k+1}^{BB} = s_k^T y_k / y_k^T y_k$;                     ▷ $\#\ 4d$
18:     Set $\alpha_{k+1}^{BB} = \min\{\alpha_{\max}, \max\{\epsilon, \alpha_{k+1}^{BB}\}\}$ if $g(s_k, y_k) > 0$; otherwise, $\alpha_{k+1}^{BB} = \alpha_{\max}$;
19:     $k = k + 1$;
20: end while
```

As a continuation of our work,[28] we provide a specific LRBFGS for the SPD Karcher mean computation in Algorithms 6 and 7, so that readers are able to implement the methods easily. In fact, those two algorithms are ready to solve any optimization problem on $\mathcal{S}_{++}^n$ as long as the readers provide a cost function and its Riemannian gradient. Algorithm 6 uses the extrinsic representation, and Algorithm 7 uses the intrinsic representation. The number of flops for each step is given on the right-hand side of the algorithms. For simplicity of notation, we use λ_m, λ_r, and λ_t to denote the flops in the metric, retraction, and vector transport evaluations respectively, and use superscripts, w and d, to denote the extrinsic and intrinsic representations respectively. The numbers of flops per iteration for Algorithm 6 and Algorithm 7, respectively, are

$$\#^w = 2(l+2)\lambda_m^w + 4lw + \lambda_r^w + 4w + 2(l+1)\lambda_t^w, \tag{40}$$

$$\#^d = 2(l+2)\lambda_m^d + 4ld + \lambda_r^d + 4d + (13n^3/3 + 2d). \tag{41}$$

Notice that m is the upper limit of the limited-memory size l. Also notice that there is no λ_t^d term in Equation (41) since the vector transport by parallelization is used, which is the identity. The last term $(13n^3/3 + 2d)$ in (41) comes from the evaluation of functions $E2D$ and $D2E$ given in Algorithms 1 and 2. For the metric evaluation, we have $\lambda_m^w = 6n^3 + o(n^3)$ and $\lambda_m^d = n^2 + o(n^2)$ for different representations. Simplifying and rearranging (40) and (41), we have

$$\#^w = 12ln^3 + 24n^3 + \lambda_r^w + 2(l+1)\lambda_t^w + o(ln^3) + o(n^3), \tag{42}$$

$$\#^d = 4ln^2 + 13n^3/3 + \lambda_r^d + o(ln^2) + o(n^3). \tag{43}$$

Algorithm 6. LRBFGS for problems on $\mathcal{S}_{++}^n$ manifold using extrinsic representation and general vector transport

```
Require: backtracking reduction factor $\varrho \in (0,1)$; Armijo parameter $\delta \in (0,1)$; tolerance for stopping criterion $\epsilon$; initial iterate $x_0 \in \mathcal{M}$; an integer $m > 0$; an integer $\varpi$;
1: $k = 0, \gamma_0 = 1, l = 0$; and let $S_f = \emptyset$;
2: Compute $\mathrm{grad} f(x_k)$;
3: while $\|\mathrm{grad} f(x_k)\| > \epsilon$ do
4:     $\mathcal{H}_k^0 = \gamma_k \mathrm{id}$. Obtain $\eta_k \in \mathrm{T}_{x_k}\mathcal{M}$ by the following algorithm, Step 5 to Step 15:
5:     $q \leftarrow \mathrm{grad} f(x_k)$;
6:     for $i = k-1, k-2, \dots, k-l$ do        $\triangleright$ # $l(\lambda_m^w + 2w)$
7:         $\xi_i \leftarrow \rho_i g(s_i^{(k)}, q)$;
8:         $q \leftarrow q - \xi_i y_i^{(k)}$;
9:     end for
10:    $r \leftarrow \mathcal{H}_k^0 q$;        $\triangleright$ # $w$
11:    for $i = k-l, k-l+1, \dots, k-1$ do        $\triangleright$ # $l(\lambda_m^w + 2w)$
12:        $\omega \leftarrow \rho_i g(y_i^{(k)}, r)$;
13:        $r \leftarrow r + s_i^{(k)}(\xi_i - \omega)$;
14:    end for
15:    Set $\eta_k = -r$, $\alpha_k = 1$;        $\triangleright$ # $w$
16:    Compute $\tilde{x}_k = R_{x_k}(\alpha_k \eta_k)$;        $\triangleright$ # $\lambda_r^w$
17:    If
           $$f(\tilde{x}_k) \le \max\left(f(x_k) \cup S_f\right) + \delta \alpha_k g(\mathrm{grad} f(x_k), \eta_k),$$
       then set $x_{k+1} = \tilde{x}_k$ and go to Step 19;
18:    Set $\alpha_k = \varrho \alpha_k$ and go to Step 16;
19:    Compute $\mathrm{grad} f(x_{k+1})$;
20:    Set $S_f \leftarrow S_f \cup \{f(x_{k+1})\}$; if $|S_f| < \varpi$, then remove the oldest entry in~$S_f$;
21:    Define $s_k^{(k+1)} = \mathcal{T}_{\alpha_k\eta_k} \alpha_k \eta_k$ and $y_k^{(k+1)} = \mathrm{grad} f(x_{k+1}) - \mathcal{T}_{\alpha_k\eta_k} \mathrm{grad} f(x_k)$; §        $\triangleright$ # $2\lambda_t^w + 2w$
22:    Compute $a = g(y_k^{(k+1)}, s_k^{(k+1)})$ and $b = \|s_k^{(k+1)}\|^2$;        $\triangleright$ # $2\lambda_m^w$
23:    if $\frac{a}{b} \ge 10^{-4}\|\mathrm{grad} f(x_k)\|$ then        $\triangleright$ # $\lambda_m^w$
24:        Compute $c = \|y_k^{(k+1)}\|^2$ and define $\rho_k = 1/a$ and $\gamma_{k+1} = a/c$;        $\triangleright$ # $\lambda_m^w$
25:        Add $s_k^{(k+1)}$, $y_k^{(k+1)}$ and $\rho_k$ into storage and if $l \ge m$, then discard vector pair $\{s_{k-l}^{(k)}, y_{k-l}^{(k)}\}$ and scalar $\rho_{k-l}$ from storage, else $l \leftarrow l+1$; Transport $s_{k-l+1}^{(k)}, s_{k-l+2}^{(k)}, \dots, s_{k-1}^{(k)}$ and $y_{k-l+1}^{(k)}, y_{k-l+2}^{(k)}, \dots, y_{k-1}^{(k)}$ from $\mathrm{T}_{x_k}\mathcal{M}$ to $\mathrm{T}_{x_{k+1}}\mathcal{M}$ by $\mathcal{T}$, then get $s_{k-l+1}^{(k+1)}, s_{k-l+2}^{(k+1)}, \dots, s_{k-1}^{(k+1)}$ and $y_{k-l+1}^{(k+1)}, y_{k-l+2}^{(k+1)}, \dots, y_{k-1}^{(k+1)}$;        $\triangleright$ # $2(l-1)\lambda_t^w$
26:    else
27:        Set $\gamma_{k+1} \leftarrow \gamma_k$, $\{\rho_k, \dots, \rho_{k-l+1}\} \leftarrow \{\rho_{k-1}, \dots, \rho_{k-l}\}$, $\{s_k^{(k+1)}, \dots, s_{k-l+1}^{(k+1)}\} \leftarrow \{\mathcal{T}_{\alpha_k\eta_k} s_{k-1}^{(k)}, \dots, \mathcal{T}_{\alpha_k\eta_k} s_{k-l}^{(k)}\}$ and $\{y_k^{(k+1)}, \dots, y_{k-l+1}^{(k+1)}\} \leftarrow \{\mathcal{T}_{\alpha_k\eta_k} y_{k-1}^{(k)}, \dots, \mathcal{T}_{\alpha_k\eta_k} y_{k-l}^{(k)}\}$;        $\triangleright$ # $2l\lambda_t^w$
28:    end if
29:    $k = k+1$;
30: end while
```

From (42) and (43), the computational benefit of the intrinsic representation is substantial. The limited-memory size l imposes a much heavier burden on Algorithm 6 where the extrinsic representation is used. In our implementation of Algorithm 7, we suggest retraction (24), which needs $3n^3 + o(n^3)$ flops. Hence the overall flops required by Algorithm 7 is $\#^d = 4ln^2 + 22n^3/3 + o(ln^2) + o(n^3)$. Notice that if the locking condition is imposed on Algorithm 7, extra $12(l+1)n^2$ flops are needed. For Algorithm 6, any choice of retraction and vector transport would yield a larger flop compared to Algorithm 7. Notice that the identity vector transport using extrinsic representation is not applicable since it is not isometric under metric (1).¶

¶If the locking condition is imposed, then $y_k^{(k+1)} = \mathrm{grad}\, f(x_{k+1})/\beta_k - \mathcal{T}_{\alpha_k\eta_k} \mathrm{grad}\, f(x_k)$, where $\beta_k = \|\alpha_k\eta_k\| / \|\mathcal{T}_{R_{\alpha_k\eta_k}} \alpha_k\eta_k\|$.

Algorithm 7. LRBFGS for problems on $\mathcal{S}_{++}^n$ manifold using intrinsic representation and vector transport by parallelization

```
Require: backtracking reduction factor $\varrho \in (0,1)$; Armijo parameter $\delta \in (0,1)$; tolerance for stopping criterion $\epsilon$; initial
    iterate $x_0 \in \mathcal{M}$; an integer $m > 0$; an integer $\varpi$;
1: $k = 0, \gamma_0 = 1, l = 0$; $\vartheta = false$; and let $S_f = \emptyset$;
2: Compute $\mathrm{grad} f(x_k)$;
3: Compute $x_k = L_k L_k^T$;                                              $\triangleright \# n^3/3$
4: Compute $\mathrm{gf}_k^d = E2D_{x_k}(\mathrm{grad} f(x_k))$ by Algorithm 1;      $\triangleright \#\ 2n^3 + d$
5: while $\|\mathrm{gf}_k^d\| > \epsilon$ do
6:     Obtain $\eta_k \in \mathbb{R}^d$, intrinsic representation of a tangent vector $\eta^w \in \mathrm{T}_{x_k}\mathcal{M}$, by the following algorithm, Step 7 to
    Step 17:
7:     $q \leftarrow \mathrm{gf}_k^d$;
8:     for $i = k-1, k-2, \ldots, k-l$ do                                  $\triangleright \#\ l(\lambda_m^d + 2d)$
9:         $\xi_i \leftarrow \rho_i q^T s_i$;
10:        $q \leftarrow q - \xi_i y_i$;
11:    end for
12:    $r \leftarrow \gamma_k q$;                                                $\triangleright \#\ d$
13:    for $i = k-l, k-l+1, \ldots, k-1$ do                                $\triangleright \#\ l(\lambda_m^d + 2d)$
14:        $\omega \leftarrow \rho_i r^T y_i$;
15:        $r \leftarrow r + s_i(\xi_i - \omega)$;
16:    end for
17:    set $\eta_k = -r$, $\alpha_k = 1$;                                      $\triangleright \#\ d$
18:    Compute $\eta_k^w = D2E_{x_k}(\eta_k)$ by Algorithm 2;               $\triangleright \#\ 2n^3 + n(n+1)/2$
19:    Compute $\tilde{x}_k = R_{x_k}(\alpha_k \eta_k^w)$;                    $\triangleright \#\ \lambda_r^d$
20:    If
               $$f(\tilde{x}_k) \leq \max\left(f(x_k) \cup S_f\right) + \delta \alpha_k \eta_k^T \mathrm{gf}_k^d,$$
       then set $x_{k+1} = \tilde{x}_k$ and go to Step 22;
21:    Set $\alpha_k = \varrho \alpha_k$ and go to Step 19;
22:    Compute $\mathrm{grad} f(x_{k+1})$;
23:    Set $S_f \leftarrow S_f \cup \{f(x_{k+1})\}$; if $|S_f| > \varpi$, then remove the oldest entry in $S_f$;
24:    Compute $x_{k+1} = L_{k+1} L_{k+1}^T$;                                 $\triangleright \# n^3/3$
25:    Compute $\mathrm{gf}_k^d = E2D_{x_k}(\mathrm{grad} f(x_k))$ by Algorithm 1;  $\triangleright \#\ 2n^3 + d$
26:    Define $s_k = \alpha_k \eta_k$ and $y_k = \mathrm{gf}_{k+1}^d - \mathrm{gf}_k^d$;#       $\triangleright \#\ 2d$
27:    Compute $a = y_k^T s_k$ and $b = \|s_k\|_2^2$;                        $\triangleright \#\ 2\lambda_m^d$
28:    if $\frac{a}{b} \geq 10^{-4} \|\mathrm{gf}_k^d\|_2$ then                    $\triangleright \#\ \lambda_m^d$
29:        Compute $c = \|y_k^{(k+1)}\|_2^2$ and define $\rho_k = 1/a$ and $\gamma_{k+1} = a/c$;   $\triangleright \#\ \lambda_m^d$
30:        Add $s_k, y_k$ and $\rho_k$ into storage and if $l \geq m$, then discard vector pair
           $\{s_{k-l}, y_{k-l}\}$ and scalar $\rho_{k-l}$ from storage, else $l \leftarrow l + 1$;
31:    else
32:        Set $\gamma_{k+1} \leftarrow \gamma_k$, $\{\rho_k, \ldots, \rho_{k-l+1}\} \leftarrow \{\rho_{k-1}, \ldots, \rho_{k-l}\}$, $\{s_k, \ldots, s_{k-l+1}\} \leftarrow$
           $\{s_{k-1}, \ldots, s_{k-l}\}$ and $\{y_k, \ldots, y_{k-l+1}\} \leftarrow \{y_{k-1}, \ldots, y_{k-l}\}$
33:    end if
34:    $k = k + 1$;
35: end while
```

However, our complexity analysis above focuses on manifold- and algorithm-related operations, the problem-related operations—function, gradient evaluations and line search procedure—are not considered. From the discussion in

Section 3.3,[4] the evaluation of the cost function requires $18Kn^3$ flops, and the computation of the gradient requires extra $5Kn^3$ flops given the function evaluation. The line search procedure may take a few steps to terminate, and each step requires one cost function evaluation. In the ideal case where the initial stepsize satisfies the Armijo condition in Step 20 in Algorithm 7, that is, the cost function is evaluated only once, the flops required by problem-related operations is $23Kn^3$. As n gets larger, the proportion of computational time spent on function and gradient evaluations is

$$\frac{23Kn^3}{23Kn^3 + 4ln^2 + 22n^3/3} \approx \frac{23K}{23K + 22/3} \geq \frac{23 \cdot 3}{23 \cdot 3 + 22/3} \approx 90.39\%. \tag{44}$$

Inequality (44) implies that the problem-related operations dominate the computation time for matrices with high dimension, which is consistent with our empirical observations in experiments that 70% to 90% of the computational time is from function and gradient evaluations. If the line search procedure requires more steps to terminate, then the problem-related operations would result in a larger proportion of total computational time. Therefore, it is crucial to have a good initial stepsize.

Finally, note that LRBFGS with zero memory size, i.e., $m = 0$, is equivalent to RBB. This is easy to verify by setting $m = 0$ in Algorithm 6 and 7. Just as there are different versions of the BB stepsize, alternatives are available for the initial scalling γ_{k+1} in step 24 of Algorithm 6 and step 29 of Algorithm 7, such as BB1 (37), BB2 (38), and $\mathrm{ABB}_{\min}$ (39). In particular, we use BB2 (38) as the default.

Another optimization method which does not use the framework (36) is the majorization minimization algorithm given by Zhang.[26] The majorization minimization algorithm uses an iterative procedure:

$$X_{k+1} = T(X_k),$$

where $T(Y) = \arg\min_{X \in S_{++}^n} G(X, Y)$ and the majorization function $G(X,Y)$ satisfies $G(X,Y) \geq F(X)$ and $G(Y,Y) = G(Y)$. The resulting algorithm is:

$$X_{k+1} = f_2(X_k)^{1/2}(f_2(X_k)^{1/2} f_1(X_k) f_2(X_k)^{1/2})^{-1/2} f_2(X_k)^{1/2},$$

where $f_1(X) = \sum_{i=1}^{K} A_i^{-1/2} g_1(A_i^{-1/2} X A_i^{-1/2}) A_i^{-1/2}$, $g_1(x) = (\sqrt{\ln^2 x + 1} + \ln x)x^{-1}$ and $f_2(X) = \sum_{i=1}^{K} A_i^{1/2} g_2(A_i^{-1/2} X A_i^{-1/2}) A_i^{1/2}$, $g_2(x) = (\sqrt{\ln^2 x + 1} - \ln x)x$. For efficiency, we replace the square root of an SPD matrix by its Cholesky factor when applicable.

5 | EXPERIMENTS

As customary in the literature (see, e.g., the work of Iannazzo et al.[20] and the references therein), we compare algorithms on synthetic data chosen to reveal their properties. We compare the performance of LRBFGS described in Algorithm 7 and existing state-of-the-art methods, including the RBB[20] (using implementation in Algorithm 5), the RSD method with stepsize selection rule (RSD-QR)(22, section 3.6]) (using implementation in Algorithm 3), the Richardson-like iteration (RL)[24] (Algorithm 3 with retraction and step size modified), the majorization minimization (MAJ) algorithm[26] and the Riemannian BFGS method (RBFGS).[33,37] Algorithm 4 and Algorithm 6 are not compared here since they consistently take similar number of iterations and more computational time when compared to Algorithm 5 and Algorithm 7, respectively, see Figure 1 for a representative example.

All experiments are performed on a 64bit Ubuntu 18.04 platform with 3.5GHz CPU (Intel Core i7-7800X). Experiments in Sections 5.2, 5.3, 5.4 and 5.5 are carried out using C++, compiled with gcc-4.7.x. Section 5.6 presents a comparison of computation time between MATLAB and C++ implementations. All MATLAB experiments are performed using MATLAB R2018b 64-bit (glnxa64). In particular, we use the MATLAB implementation of RL in Bini et al.'s Matrix Means Toolbox.[25]

[4]If retraction (24) and isometric vector transport (32) that satisfy the locking condition are used, s_k and y_k are computed as follows: compute $z^w = \mathcal{T}_{R_{\alpha_k \eta_k^w}}(\alpha_k \eta_k^w)$, where $\mathcal{T}_{R_\xi} \eta = \eta + (\eta X^{-1} \xi + \xi X^{-1} \eta)/2$; obtain the intrinsic representation z of z^w by Algorithm 1; compute $\beta = \alpha_k^2 \eta_k^T \eta_k / z^T z$, $v_1 = 2\alpha_k \eta_k$, $v_2 = -\alpha_k \eta_k - \beta z$; Define $s_k = (I - 2v_2 v_2^T / v_2^T v_2)(I - 2v_1 v_1^T / v_1^T v_1)(\alpha_k \eta_k)$, $y_k = \mathrm{gf}_{k+1}^d / \beta - (I - 2v_2 v_2^T / v_2^T v_2)(I - 2v_1 v_1^T / v_1^T v_1)\mathrm{gf}_k^d$.

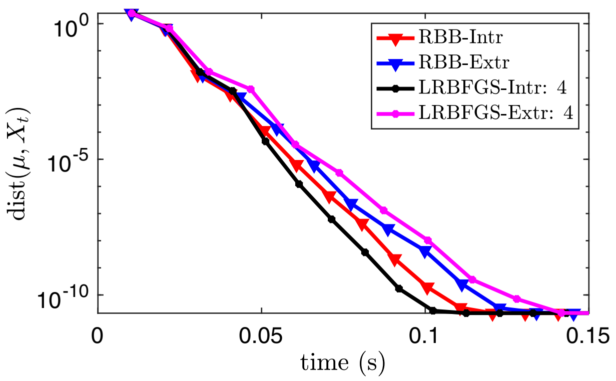


FIGURE 1 An average of 10 random results. Comparison of Riemannian version of the Barzilai-Borwein method and limited-memory Riemannian BFGS (LRBFGS) using extrinsic representation (Algorithm 4 and Algorithm 6) with those using intrinsic representation (Algorithm 5 and Algorithm 7). $k = 30$ and $n = 60$. $m = 4$ in LRBFGS. The μ is the true solution up to numerical errors. See details about parameters and experiment design in later sections

Regarding the parameter setting, we set Armijo parameter $\delta = 10^{-4}$, backtracking reduction factor $\varrho = 0.5$ for well-conditioned data sets and $\varrho = 0.25$ for ill-conditioned ones, maximum stepsize $\alpha_{\max} = 100$, and minimum stepsize $\alpha_{\min}$ is machine epsilon. We mention here that it is found from our experiments (not shown in this paper) that LRBFGS is much less sensitive to ϱ than RBB for ill-conditioned problems. LRBFGS behaves similarly well over a range of values of ϱ, but for RBB, we must set $\varrho = 0.25$ to achieve satisfactory performance. The initial stepsize in the first iteration is chosen by the strategy in the work of Rentmeesters,[22] i.e., $\alpha_0 = 2/(1 + L)$, where L is the upper bound at the initial iterate defined in inequality (3). For LRBFGS, we use different memory sizes m as specified in the legends of the figures, and impose the locking condition for ill-conditioned matrices. Specifically, we impose the locking condition on LRBFGS and RBFGS in the plots of Figure 3 and in the right plots of Figure 4. As we have shown in Section 4, imposing the locking condition requires extra complexity. For well-conditioned data sets, the problem is easy to handle, and the locking condition is not necessary. While in the ill-conditioned case, imposing the locking condition can reduce the number of iterations. The extra time caused by the locking condition is smaller than the time saved by a reduction in the number of function and gradient evaluations. The benefit of the locking condition is also demonstrated in the work of Huang et al.[37] The parameter ϖ in the nonmonotonic line search is set to be 10. Unless otherwise specified, our choice of the initial iterate is the arithmetic-harmonic mean[38] of data matrices. For the purpose of comparing the algorithms, we let them run until the specified maximal time. The only early termination criterion is when the step size α_k gets smaller than $\alpha_{\min}$ (which we set to the machine epsilon). Early termination sometimes happen very close to critical points of f when the difference between the left-hand and right-hand sides of the backtracking stopping criterion (see, e.g., Step 7 of Algorithm 4) gets so small that numerical errors yield an incorrect `false` value for the criterion. Note that, in practical use, since all the tested algorithms compute the gradient of the objective function F (see (2)), they can resort to a stopping criterion that ensures a sufficient decrease of the norm of the gradient, such as $\|\mathrm{grad} f(x_k)\| / \|\mathrm{grad} f(x_0)\| \leq 10^{-10}$.

For simplicity of notation, throughout this section we denote the number, dimension, and condition number of the matrices by K, n, and κ respectively. For each choice of (K,n) and the range of conditioning desired, a single experiment comprises generating 50 different sets of K random $n \times n$ matrices with appropriate condition numbers, and running all 5 algorithms on each set with identical parameters. The result of the experiment is the distance to the true Karcher mean averaged over the 50 sets as a function of iteration and time. To obtain sufficiently stable timing results, an average time is taken of several runs for a total runtime of at least 1 minute.

5.1 | Experiment design

In exact arithmetic, we can choose a desired (true) Karcher mean μ, and construct data matrices A_i's such that their Karcher mean is μ, that is, $\sum_{i=1}^{K} Exp_{\mu}^{-1}(A_i) = 0$ holds. The benefits of this scheme are: (i) We can control the conditioning of μ and the A_i's, and observe the influence of the conditioning on the performance of algorithms. (ii) Since μ is known, we can monitor the distance δ between μ and the iterates produced by various algorithms, thereby removing the need to consider the effects of termination criteria.

Given a Karcher mean μ, the A_i's are constructed as follows: (i) Generate W_i in Algorithm 8, with n being the size of matrix, f the order of magnitude of the condition number, and p an integer less than n, (ii) Compute $\eta_i = Exp_{\mu}^{-1}(W_i)$. (iii)

Enforce the condition $\sum_{i=1}^{K} \eta_i = 0$ on η_i's. (iv) Compute $A_i = Exp_\mu(\eta_i)$. For more details, see the work of Yuan et al. (28, section 5.1).

Algorithm 8. Generate W_i. Here the Matlab language is used, but suppose the exact arithmetic is applied

```
1: + [O, ~] = qr(randn(n)); +
2: + D = diag([rand(1, p) + 1, (rand(1, n - p) + 1) * 10 ̂(-f)]); +
3: + Wi = O * D * O'; Wi = Wi / norm(Wi, 2). +
```

In practice, exact arithmetic is not available. The true Karcher mean μ is only known up to the numerical errors. The influence of the numerical errors is investigated in Section 5.2.

5.2 | Numerical errors in computations of A_i and optimization algorithms

We first investigate the impact of numerical errors made in the construction of the data matrices A_i, $i = 1, \ldots, K$. To this end, we use single precision arithmetic and double precision arithmetic for creating the problem and using optimization algorithms. We use three combinations:

1. single precision arithmetic for computing A_i, double precision arithmetic for optimization algorithms;
2. double precision arithmetic for computing A_i, single precision arithmetic for optimization algorithms;
3. single precision arithmetic for computing A_i, single precision arithmetic for optimization algorithms.

All six algorithms are tested. The parameter K and n are chosen to be 100 and 5, respectively. Since the numerical errors from computations using double precision arithmetic is significantly smaller than that using single precision arithmetic, we neglect the numerical errors from double precision arithmetic in this section.

The experimental results are shown in Figure 2. In the combination 1, the optimization algorithms find the exact Karcher mean of A_i. Therefore, the noise floors gives the distance from the true Karcher mean to μ. In the combination 2, μ is the exact Karcher mean. Therefore, the noise floors shows the numerical errors of the optimization algorithms. It can be seen from Figure 2 that these two kinds of numerical errors are comparable. When both kinds of numerical errors are involved, as shown on the bottom of Figure 2, the noise floors remain approximately the same. Therefore, we conclude that the numerical errors of the true Karcher mean μ created by constructions of A_i is comparable to the noise floor of the optimization algorithms. Therefore, in the remaining experiments, we use the same precision arithmetic, the double precision arithmetic, for constructing A_i and solving the optimization problem.

5.3 | Comparison of performance between different algorithms

As shown in Figure 2, RL iteration, RSD-QR, and MAJ are noticeably slower than the RBB, LRBFGS and RBFGS methods in terms of iterations and computational time, and RBB, LRBFGS, and RBFGS performs similarly in term of number of iterations. This phenomena have been observed for a wide range of problems. The proposed methods, LRBFGS and RBFGS, show advantage compared to RBB when the condition numbers of A_i are large, see Figure 3. When the data matrices are well-conditioned, RBB is one of the competitive methods, see left column in Figure 3. As the data matrices become ill-conditioned, for example, in the right column in Figure 3, the RBB method slows down compared to LRBFGS methods. This is reasonable since, in principle, the condition number of the Hessian increases as the condition numbers of the data matrices increase, and the RBB search direction (i.e., along the negative gradient) is oblivious to second-order information. RBFGS is always comparable to LRBFGS only in terms of iterations. When the dimension of the domain increases, the cost on the Hessian approximation update dominates the computations, and causes that RBFGS is slower than LRBFGS in terms of computational time.

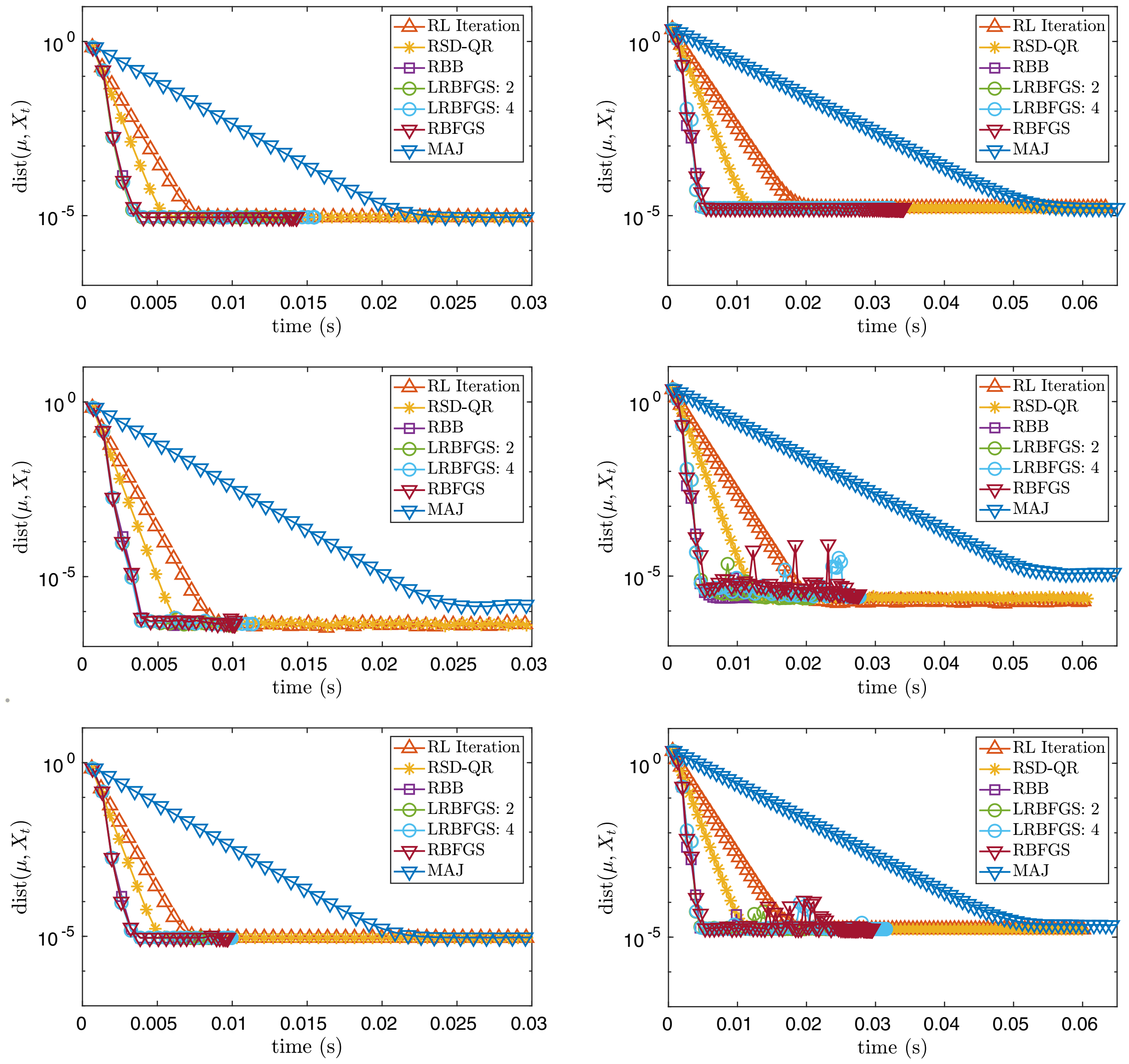
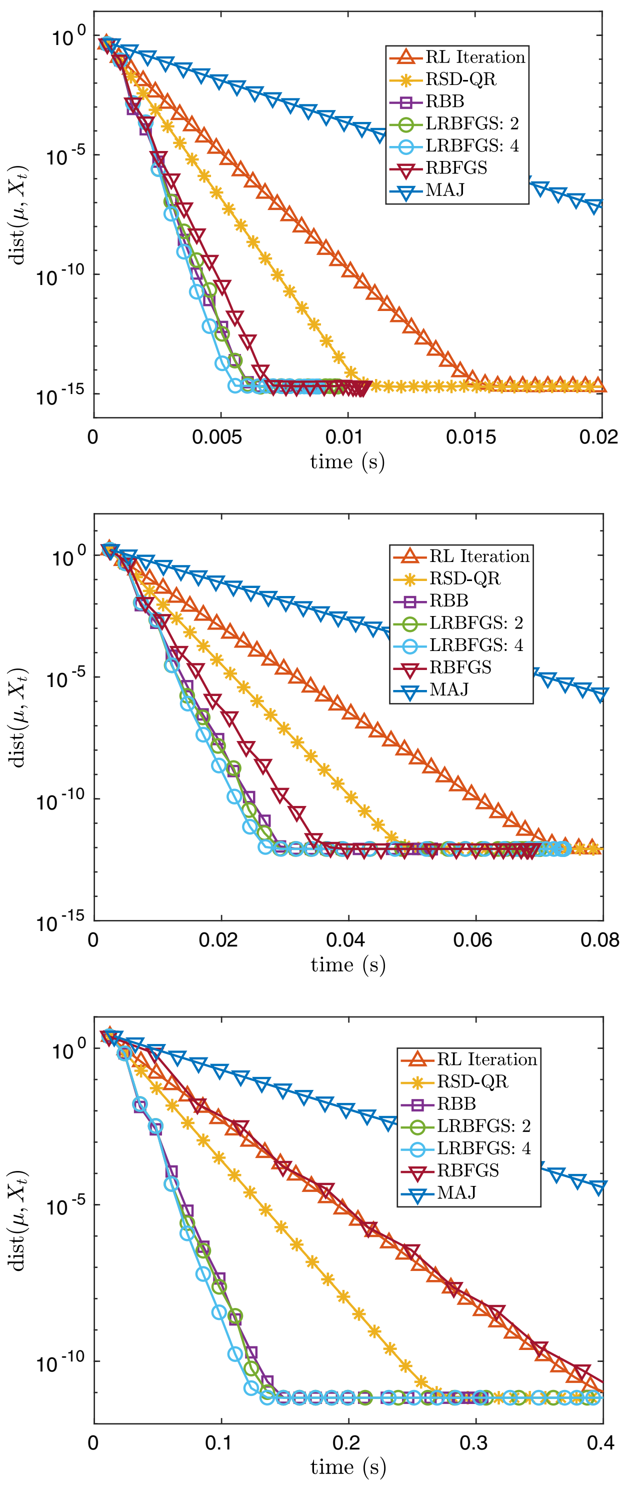


FIGURE 2 Top row: single precision arithmetic for computing A_i, double precision arithmetic for optimization algorithms; Middle row: double precision arithmetic for computing A_i, single precision arithmetic for optimization algorithms; Bottom: single precision arithmetic for computing A_i, single precision arithmetic for optimization algorithms; Left column: $5 \leq \kappa(A_i) \leq 100$; Right column: $100 \leq \kappa(A_i) \leq 5000$

5.4 | Initial iterate

We compare the performances of the algorithms using two different initial iterates: the arithmetic-harmonic mean and the Cheap mean.[39] The Cheap mean is known to be a good approximation of the Karcher mean, but, is not cheap to compute. We use Bini et al.'s Matrix Means Toolbox[25] for the computation of the Cheap mean. We consider 30 badly conditioned 30×30 matrices ($10^5 \leq \kappa(A_i) \leq 10^9$). The results are presented in Figure 4. Notice that the time required to compute the initial iterate is included in the plots. The x-axis of the right plot does not start from 0, which shows that the computation of the Cheap mean is time demanding. When the initial iterate is close enough to the true solution, as shown in the right plot in Figure 4, we observe a faster convergence in the first a few steps for all algorithms. In both cases, LRBFGS with $m = 4$ outperforms the other algorithms in terms of computation time per unit of accuracy required.



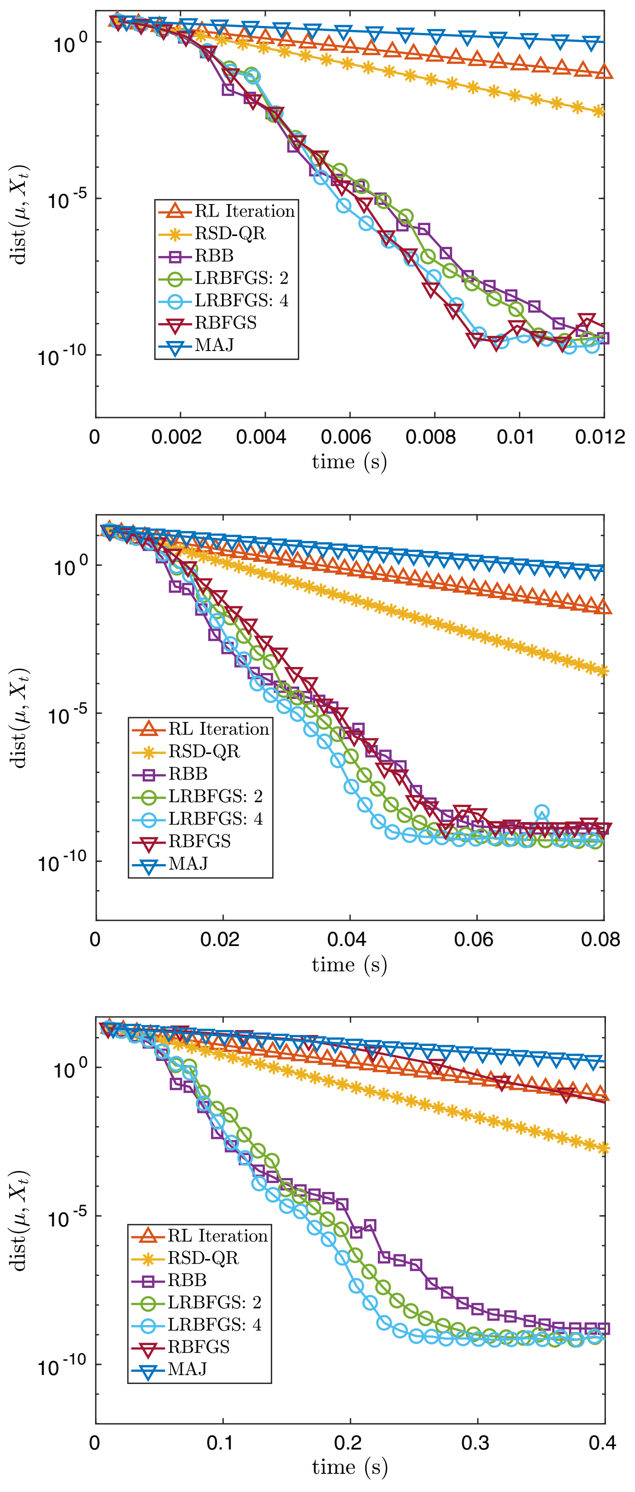


FIGURE 3 Top left: $K = 100$, $n = 3$, $5 \le \kappa(A_i) \le 60$; Top right: $K = 100$, $n = 3$, $10^5 \le \kappa(A_i) \le 10^9$; Middle left: $K = 30$, $n = 30$, $10 \le \kappa(A_i) \le 60$; Middle right: $K = 30$, $n = 30$, $10^5 \le \kappa(A_i) \le 10^9$; Bottom left: $K = 30$, $n = 60$, $10 \le \kappa(A_i) \le 60$; Bottom right: $K = 30$, $n = 60$, $10^5 \le \kappa(A_i) \le 10^9$

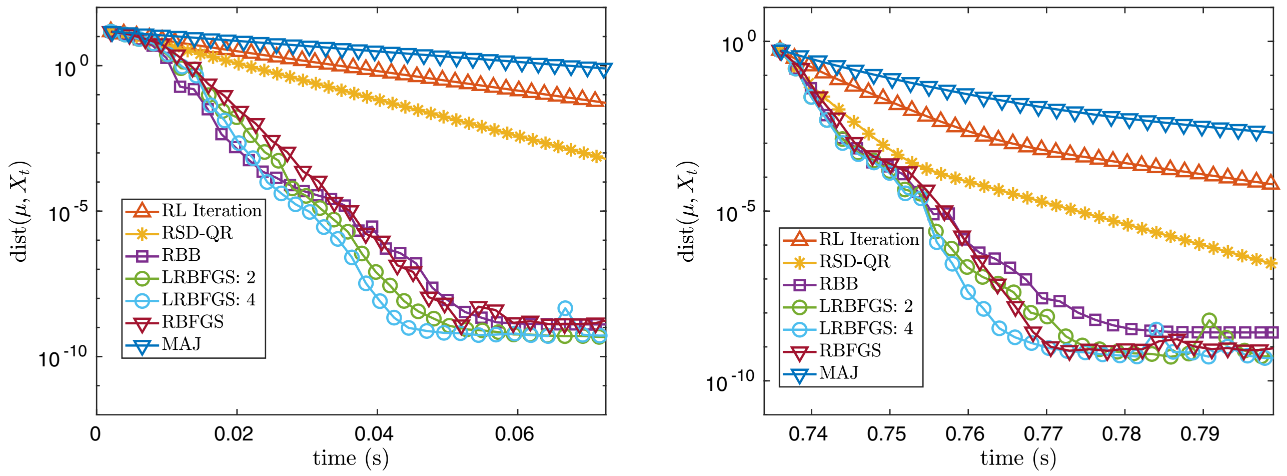


FIGURE 4 Comparison of different algorithms using different initial iterates with $K = 30$, $n = 30$, and $10^5 \leq \kappa(A_i) \leq 10^9$. Top: using the Arithmetic-Harmonic mean as initial iterate; Bottom: using the Cheap mean as initial iterate

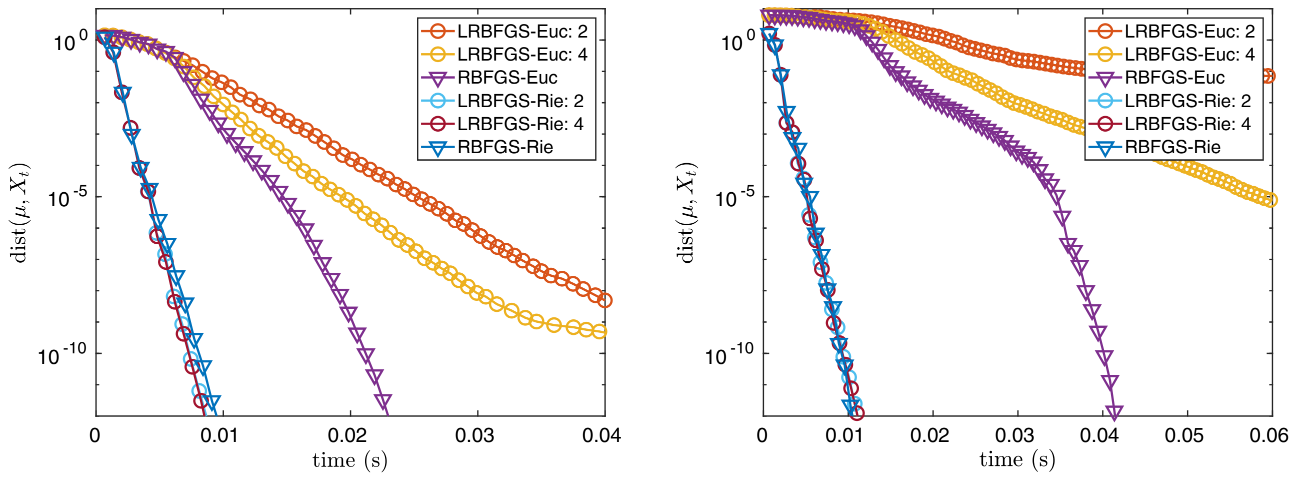


FIGURE 5 Left: $\kappa(\mu) \approx 5$, $50 \leq \kappa(A_i) \leq 2 * 10^3$; Right: $\kappa(\mu) \approx 15$, $50 \leq \kappa(A_i) \leq 2 * 10^4$

5.5 | Comparison of the Riemannian metric and Euclidean metric

In this section we compare two different metrics, the Euclidean metric and Riemannian metric, for LRBFGS and RBFGS. Figure 5 shows the results from representative experiments for $(K,n) = (100,5)$. The condition number of the true Karcher mean of each set of matrices, that is, $\mu(A_1, \ldots, A_K)$, is approximately 5 in the left figure and approximately 15 in the right figure. In the legends, "Euc" refers to the Euclidean metric and "Rie" refers to the Riemannian metric. We observe that the condition number of the true Karcher mean μ changes the convergence speed of the Euclidean metric. When the Riemannian metric is used, LRBFGS and RBFGS behave similarly just as the well-conditioned case in Sections 5.2 and 5.3. However, in the case of the Euclidean metric, LRBFGS with $m = 4$ is faster than $m = 2$. For RBFGS, the influence of the metric becomes less significant compared to simpler methods.

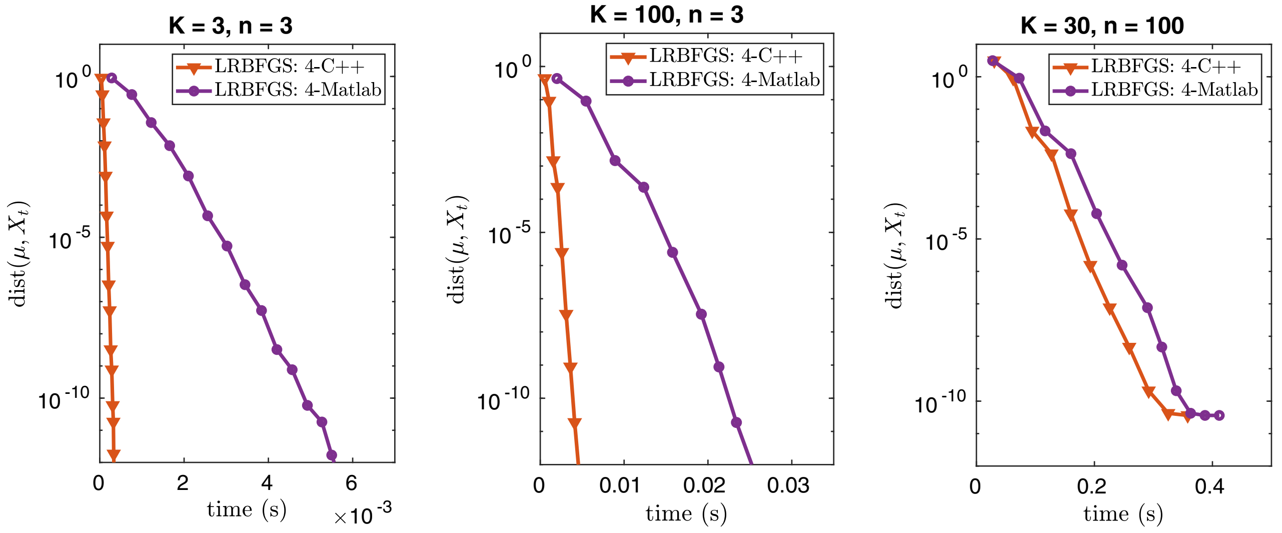


FIGURE 6 Compare the limited-memory Riemannian BFGS (LRBFGS) method using Matlab implementation and C++ implementation with different values of k and n

5.6 | Comparison of C++ and MATLAB implementations

Finally, we compare the time efficiency of the algorithms implemented by C++ and MATLAB for the SPD Karcher mean computation. The results are reported in Figure 6. For small-size problems, the C++ implementation is faster than that of MATLAB by a factor of 10 or more with the factor gradually reducing as n or K gets larger. This phenomenon can be explained by the fact that when n or K is small, the difference of efficiency between C++ and MATLAB implementations is mainly due to the difference between compiled languages and interpreted languages. When n or K increases, the BLAS and LAPACK calls start to dominate the computation time, which leads to a decrease in the factor. Such phenomenon has been observed in the work of Huang et al.[27]

6 | CONCLUSION

In this paper, we have considered computing the Karcher mean of SPD matrices using the Riemannian optimization methods. We have also discussed efficient forms of Riemannian optimization methods. There are several alternatives from which to choose the representation of a tangent vector, retraction and vector transport. We have provided complexity-based recommendations for those alternatives.

Our numerical experiments provide empirical guidelines to choose between various methods and two metrics. It is observed that RSD-QR systematically outperforms RL and RL systematically outperforms MAJ. For most problems, RBB and LRBFGS are observed to behave similarly and to outperform the other methods. Here, we recommend using LRBFGS as the default method for the SPD Karcher mean computation mainly for three reasons: (i) When the data matrices are well-conditioned, LRBFGS and RBB are competitive, with a slight advantage for LRBFGS on some test sets. (ii) As the data matrices get ill-conditioned, LRBFGS outperforms RBB.

We also present empirical illustration of the speed up of C++ implementation compared with MATLAB implementation. Notice that it is demonstrated theoretically and empirically that for large-size problems, the dominant computation time (70%-90%) is in the problem-related operations, that is, function and gradient evaluations.

ACKNOWLEDGEMENTS
This work was supported by (i) “the Fonds de la Recherche Scientifique – NRS and the Fonds Wetenschappelijk Onderzoek – Vlaanderen under EOS Project no 30468160", (ii) “Communauté française de Belgique - Actions de Recherche

Concertées" (contract ARC 14/19-060), (iii) "the Fundamental Research Funds for the Central Universities: number 20720190060". This work does not have any conflicts of interest.

ORCID

Wen Huang https://orcid.org/0000-0001-8324-2416

REFERENCES

1. Cheng G, Salehian H, Vemuri B. Efficient recursive algorithms for computing the mean diffusion tensor and applications to DTI segmentation. Comput Vis ECCV. 2012;2012:390–401.
2. Fletcher PT, Joshi S. Riemannian geometry for the statistical analysis of diffusion tensor data. Signal Process. 2007;87(2):250–262.
3. Rathi Y, Tannenbaum A, Michailovich O. *Segmenting images on the tensor manifold*. Paper presented at: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition; 2007:1-8.
4. Arandjelovic O, Shakhnarovich G, Fisher J, Cipolla R, Darrell T. *Face recognition with image sets using manifold density divergence*. Paper presented at: Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2005. CVPR 2005; vol. 1, 2005:581-588; IEEE.
5. Huang Z, Wang R, Shan S, Chen X. Face recognition on large-scale video in the wild with hybrid Euclidean-and-Riemannian metric learning. Pattern Recogn. 2015;48(10):3113–3124.
6. Lu J, Wang G, Moulin P. *Image set classification using holistic multiple order statistics features and localized multi-kernel metric learning*. Paper presented at: Proceedings of the IEEE International Conference on Computer Vision; 2013:329-336.
7. Shakhnarovich G, Fisher JW, Darrell T. *Face recognition from long-term observations*. Paper presented at: Proceedings of the European Conference on Computer Vision; 2002:851-865; Springer.
8. Tuzel O, Porikli F, Meer P. *Region covariance: a fast descriptor for detection and classification*. Paper presented at: Proceedings of European Conference on Computer Vision; 2006:589-600; Springer.
9. Absil PA, Gousenbourger PY, Striewski P, Wirth B. Differentiable piecewise-Bezier surfaces on Riemannian manifolds. SIAM J Imag Sci. 2016;9(4):1788–1828.
10. Barmpoutis A, Vemuri BC, Shepherd TM, Forder JR. Tensor splines for interpolation and approximation of DT-MRI with applications to segmentation of isolated rat hippocampi. IEEE Trans Med Imag. 2007;26(11):1537–1546.
11. Pennec X, Fillard P, Ayache N. A Riemannian framework for tensor computing. Int J Comput Vis. 2006;66(1):41–66.
12. Jayasumana S, Hartley R, Salzmann M, Li H, Harandi M. *Kernel methods on the Riemannian manifold of symmetric positive definite matrices*. Paper presented at: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition; 2013:73-80.
13. Ando T, Li CK, Mathias R. Geometric means. Linear Algebra Appl. 2004;385:305–334.
14. Bhatia R, Karandikar RL. Monotonicity of the matrix geometric mean. Mathematische Annalen. 2012;353(4):1453–1467.
15. Lawson J, Lim Y. Monotonic properties of the least squares mean. Mathematische Annalen. 2011;351(2):267–279.
16. Moakher M. On the averaging of symmetric positive-definite tensors. J Elasticity. 2006;82(3):273–296.
17. Karcher H. Riemannian center of mass and mollifier smoothing. Commun Pure Appl Math. 1977;30(5):509–541.
18. Absil PA, Mahony R, Sepulchre R. Optimization algorithms on matrix manifolds. Princeton, NJ: Princeton University Press, 2008.
19. Jeuris B, Vandebril R, Vandereycken B. A survey and comparison of contemporary algorithms for computing the matrix geometric mean. Electron Trans Numer Anal. 2012;39:379–402.
20. Iannazzo B, Porcelli M. The Riemannian Barzilai Borwein method with nonmonotone line search and the matrix geometric mean computation. IMA J Numer Anal. 2018;38(1):495–517.
21. Rentmeesters Q, Absil PA. *Algorithm comparison for Karcher mean computation of rotation matrices and diffusion tensors*. Paper presented at: Proceedings of the 19th European Signal Processing Conference; 2011:2229-2233.
22. Rentmeesters Q. Algorithms for data fitting on some common homogeneous spaces [PhD thesis]. Universite catholiqué de Louvain; 2013.
23. Nesterov Y. Introductory lectures on convex programming Volume I: Basic course. Lecture notes. Berlin, Germany: 1998. https://www.springer.com/gp/book/9781402075537.
24. Bini DA, Iannazzo B. Computing the Karcher mean of symmetric positive definite matrices. Linear Algebra Appl. 2013;438(4):1700–1710.
25. Bini D, Iannazzo B. Matrix means toolbox. http://bezout.dm.unipi.it/software/mmtoolbox/.
26. Zhang T. A majorization-minimization algorithm for computing the Karcher mean of positive definite matrices. SIAM J Matrix Anal Appl. 2017;38(2):387–400.
27. Huang W, Absil P, Gallivan K, Hand P. ROPTLIB: an object-oriented C++ library for optimization on Riemannian manifolds. Accepted for publication in ACM Transactions on Mathematical Software; 2018.
28. Yuan X, Huang W, Absil PA, Gallivan KA. A Riemannian limited-memory BFGS algorithm for computing the matrix geometric mean. Proc Comput Sci. 2016;80:2147–2157.
29. Golub GH, Van Loan CF. Matrix computations. Vol 3. Baltimore and England: JHU Press, 2012.
30. Huang W, Absil PA, Gallivan KA. A Riemannian symmetric rank-one trust-region method. Math Progr. 2015;150(2):179–216.
31. Huang W, Absil PA, Gallivan KA. Intrinsic representation of tangent vectors and vector transports on matrix manifolds. Numerische Mathematik. 2017 Jun;136(2):523–543.
32. Higham NJ. Functions of matrices: Theory and computation. Philadelphia, PA: SIAM, 2008.

33. Huang W, Gallivan KA, Absil PA. A Broyden class of quasi-Newton methods for Riemannian optimization. SIAM J Optim. 2015;25(3):1660–1685.
34. Fletcher PT, Lu C, Pizer SM, Joshi S. Principal geodesic analysis on symmetric spaces: statistics of diffusion tensors. Comput Vis Math Methods Med Biomed Image Anal. 2004;3117:87–98.
35. di Serafino D, Ruggierob V, Toraldoc G, Zannid L. On the steplength selection in gradient methods for unconstrained optimization; 2017; vol.318, doi:10.1016/j.amc.2017.07.037.
36. Frassoldati G, Zanni L, Zanghirati G. New adaptive stepsize selections in gradient methods. J Ind Manag Optim. 2008;4(2):299.
37. Huang W, Absil PA, Gallivan KA. A Riemannian BFGS method without differentiated retraction for nonconvex optimization problems. SIAM J Optim. 2018;28(1):470–495.
38. Jeuris B, and Vandebril R. Geometric mean algorithms based on harmonic and arithmetic iterations. In: Nielsen F, and Barbaresco F, editors. Proceedings of the 1st International Conference on Geometric Science of Information GSI 2013, Springer Berlin, Heidelberg/Germany: 2013. p. 785–793.
39. Bini DA, Iannazzo B. A note on computing matrix geometric means. Adv Comput Math. 2011;35(2-4):175–192.

How to cite this article: Yuan X, Huang W, Absil P-A, Gallivan KA. Computing the matrix geometric mean: Riemannian versus Euclidean conditioning, implementation techniques, and a Riemannian BFGS method. *Numer Linear Algebra Appl*. 2020;e2321. https://doi.org/10.1002/nla.2321