A Riemannian quasi-Newton method for computing the Karcher mean of symmetric positive definite matrices

Xinru Yuan * Wen Huang †§ P.-A. Absil ‡ K. A. Gallivan *

Abstract

This paper tackles the problem of computing the Karcher mean of a collection of symmetric positive-definite matrices. We present a concrete limited-memory Riemannian BFGS method to handle this computational task. We also provide methods to produce efficient numerical representations of geometric objects on the manifold of symmetric positive-definite matrices that are required for Riemannian optimization algorithms. Through empirical results and computational complexity analysis, we demonstrate the robust behavior of the limited-memory Riemannian BFGS method and the efficiency of our implementation when comparing to state-of-the-art algorithms.

1 Introduction

Symmetric positive-definite (SPD) matrices have become fundamental objects in various domains. For example, a 3D diffusion tensor, i.e., a 3×3 SPD matrix, is commonly used to model the diffusion behavior of the media in diffusion tensor magnetic resonance imaging (DT-MRI) [9, 10, 30]. In addition, representing images and videos with SPD matrices has shown promise for segmentation and recognition in several works, such as [26, 33, 4, 34, 19]. In these and similar applications, it is often of interest to average SPD matrices. Averaging is required, e.g., to aggregate several noisy measurements of the same object. It also appears as subtasks in interpolation methods [1], segmentation [29, 5], and clustering [21]. An efficient implementation of averaging methods is crucial for applications where the mean computation is needed to 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, $\{A_1, \ldots, A_K\}$, is to take their arithmetic mean, i.e., $G(A_1, \ldots, A_K) = (A_1 + \cdots + A_K)/K$. However, it is not adequate in applications where the invariance under inversion is required, i.e., $G(A_1, \ldots, A_K)^{-1} = G(A_1^{-1}, \ldots, A_K^{-1})$. 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 [10] for example. Another approach is to generalize the definition of geometric mean from scalars to matrices, which yields $G(A_1, \ldots, A_K) = (A_1 \ldots A_K)^{1/K}$. However, this generalized geometric mean is not invariant under permutation since matrices are not commutative in general. Therefore there is a need of defining an appropriate geometric mean for SPD matrices. Ando et al. [3] introduced a list of fundamental properties, denoted by ALM list, that a matrix "geometric" mean should

*Department of Mathematics, Florida State University, Tallahassee FL 32306-4510, USA
†Department of Computational and Applied Mathematics, Rice University, Houston, USA
‡Department of Mathematical Engineering, Université catholique de Louvain, Louvain-la-Neuve, Belgium.
§Corresponding author. E-mail: huwst08@gmail.com.

possess, such as invariance under permutation, monotonicity, congruence invariance, and invariance under inversion, to name a few. These properties are known to be important in numerous applications, e.g. [6, 25, 27]. 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 Karcher mean proposed in [24] 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, which has recently been shown in [6].

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 [29], given by

$$g_X(\xi_X, \eta_X) = \mathrm{trace}(\xi_X X^{-1} \eta_X X^{-1}). \tag{1.1}$$

The Karcher mean of $\{A_1, \ldots, A_K\}$—also termed as the Riemannian center of mass—is the minimizer of the sum of squared distance

$$\mu = \underset{X \in \mathcal{S}_{++}^n}{\arg\min} F(X), \quad \text{with } F : \mathcal{S}_{++}^{\mathrm{n}} \to \mathbb{R},\ X \mapsto \frac{1}{2K} \sum_{i=1}^{K} \delta^2(X, A_i), \tag{1.2}$$

where $\delta(p, q) = \| \log(p^{-1/2} q p^{-1/2}) \|_{\mathrm{F}}$ is the geodesic distance associated with Riemannian metric (1.1). It is proved in [24] 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., $\operatorname{grad} F(\mu) = 0$, where $\operatorname{grad} F$ denotes the Riemannian gradient of F under metric (1.1). However, a closed-form solution for problem (1.2) 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 employed to compute the Karcher mean of SPD matrices. Most of them resort to the framework of Riemannian optimization (see, e.g., [2]), since problem (1.2) requires optimizing a function on a manifold. In particular, [23] presents a survey of several optimization algorithms, including Riemannian versions of steepest descent, conjugate gradient, BFGS, and trust-region Newton's methods. The authors conclude that the first order methods, steepest descent and conjugate gradient, are the preferred choices for problem (1.2) 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, which is especially as the size of the matrices increases. Our previous work [35] explains the good performance of the first order methods, which is because that the condition number of the Hessian of cost function F in (1.2) admits an upper bound that behaves like the logarithm of the largest condition number of data points. More specifically, the bounds for the Hessian of cost function (1.2) at X, see [32], is given by

$$1 \leq \frac{\operatorname{Hess} F(X)[\Delta X, \Delta X]}{\|\Delta X\|^2} \leq \frac{1}{K} \sum_{i=1}^{K} \frac{\log \kappa_i}{2} \coth(\frac{\log \kappa_i}{2}) \tag{1.3}$$

$$\leq 1 + \frac{\log(\max_i \kappa_i)}{2}, \tag{1.4}$$

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 $X = L_xL_x^T$). Inequality (1.4) implies that we cannot expect a very ill-conditioned Hessian in practice because of the logarithm. We also exploit a limited-memory Riemannian BFGS (LRBFGS) method to tackle the SPD Karcher mean computation task in [35]. A Riemannian version of the Barzilai-Borwein method (RBB) has been considered in [20]. Several stepsize selection rules have been investigated for the Riemannian steepest descent (RSD) method. A constant stepsize strategy is proposed in [32] and a convergence analysis is given by imposing a condition on the stepsize. An adaptive stepsize selection rule based on the explicit expression of the Hessian of cost function F is studied in [31], which is actually the optimal stepsize for strongly convex function in Euclidean space, see [28, Theorem 2.1.14]. 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 bound on the Hessian of F, respectively, given by (1.3). [31] also provides a Newton's method for the mean computation. A Richardson-like iteration is derived and evaluated empirically in [7], and is available in the Matrix Means Toolbox[1]. We will see later in Section 2.2 that the Richardson-like iteration is a steepest descent method combined with stepsize $\alpha_k = 1/L_k$.

1.3 Contributions

This paper builds on our earlier work [35], and provides a detailed description of a limited-memory Riemannian BFGS (LRBFGS) method for the SPD Karcher mean computation. Riemannian optimization methods such as LRBFGS involve manipulation of geometric objects on manifolds, such as tangent vector, retraction, vector transport and evaluation of Riemannian metric. We present detailed methods to produce efficient numerical representations of those objects on the $\mathcal{S}_{++}^{\mathrm{n}}$ manifold. In fact, there are several alternatives to choose from for the geometric objects on $\mathcal{S}_{++}^{\mathrm{n}}$. We offer theoretical and empirical suggestions on how to choose between those alternatives for LRBFGS based on computational complexity analysis and numerical experiments. We also show that LRBFGS and RBB are closely related to one another.

Another contribution of our work is to provide a C++ toolbox for the SPD Karcher mean computation, which includes LRBFGS, RBFGS, RBB, and RSD methods. The toolbox, available at `http://www.math.fsu.edu/~whuang2/papers/RMKMSPDM.htm`, relies on ROPTLIB, an object-oriented C++ library for optimization on Riemannian manifolds [17]. To the best of our knowledge, there are no other publicly available C++ toolbox for the SPD Karcher mean computation. Our previous work [35] provides a MATLAB implementation[2] for this problem. The Matrix Means Toolbox[1] developed by Bini et al. in [7] 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 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 one, and the conditioning of the problem is characterized by the condition number 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

[1] `http://bezout.dm.unipi.it/software/mmtoolbox/`

[2] `http://www.math.fsu.edu/~whuang2/papers/ARLBACMGM.htm`

robust to problem conditioning and parameter setting. The numerical results also illustrate the speedup of using C++ vs. MATLAB. Especially for small-size problems, C++ implementation is faster than MATLAB by a factor of 100 or more. The factor gradually reduces as the size of the problem gets larger.

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

2 Implementation for the $\mathcal{S}_{++}^{\mathrm{n}}$ 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 vector, Riemannian metric, isometric vector transport, and retraction. Problem-related objects include cost function and Riemannian gradient evaluation. As an extension of our previous work [35], we also provide a flop[3] count for most operations.

2.1 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 manifold $\mathcal{S}_{++}^{\mathrm{n}}$ 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, i.e., $w = n^2$.

The computational benefits of using intrinsic representation are addressed in [15, 14]: (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 computationally complexity, the intrinsic representation often leads to a very efficient implementation. This property holds for manifold $\mathcal{S}_{++}^{n}$ as shown next.

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

$$\{Le_ie_i^TL^T : i = 1,\ldots,n\}\bigcup\{\frac{1}{\sqrt{2}}L(e_ie_j^T + e_je_i^T)L^T,\ i<j,\ i=1,\ldots,n,\ j=1,\ldots,n\}, \qquad (2.1)$$

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, which however costs more [13]. It is easy to verify the orthonormality of B_X under Riemannian metric (1.1), i.e., $B_X^\flat B_X = I_{d\times d}$ for all X. (The notation $a^\flat$ denotes the

[3] A flop is a floating point operation [12, Section 1.2.4].

function $a^\flat : \mathrm{T}_X \mathcal{M} \to \mathbb{R} : v \mapsto g_X(a, v)$, where g stands for the affine-invariant metric (1.1).) We assume throughout the paper that B_X stands for our selected orthonormal basis of $\mathrm{T}_X \mathcal{S}^n_{++}$ defined in (2.1).

Let η_X be a tangent vector in $\mathrm{T}_X \mathcal{S}^{\mathrm{n}}_{++}$ 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 (2.1), 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.

Since B_X forms an orthonormal basis of $\mathrm{T}_X \mathcal{S}^{\mathrm{n}}_{++}$, then the Riemannian metric (1.1) reduces to the Euclidean metric under the intrinsic representation, i.e.,

$$\tilde{g}_X(v_X, u_X) := g_X(\eta_X, \xi_X) = g_X(B_X v_X, B_X u_X) = v_X^T u_X, \tag{2.2}$$

where $\eta_X = B_X v_X$ and $\xi_X = B_X u_X \in \mathrm{T}_X \mathcal{S}^{\mathrm{n}}_{++}$. The evaluation of (2.2) requires $2d$ flops, which is cheaper than the evaluation of (1.1).

For the intrinsic approach, retractions (see Section 2.2) require mapping the intrinsic representation back to the extrinsic representation, which may require extra work. Let function $D2E_X : v_X \mapsto \eta_X = B_X v_X$ denote this mapping. In practice, the function $D2E_X$ using basis (2.1) is described in Algorithm 2.

Algorithm 1 Compute $E2D_X(\eta_X)$

Input: $X = LL^T \in \mathcal{S}^{\mathrm{n}}_{++}$, $\eta_x \in \mathrm{T}_X \mathcal{S}^{\mathrm{n}}_{++}$.

1: Compute $Y = L^{-1}\eta_X$ by solving linear system $LY = \eta_X$; ▷ # 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 $LZ = Y$; ▷ # n^3
4: return $v_X = (z_{11}, z_{22}, \ldots, z_{nn}, \sqrt{2}z_{12}, \ldots, \sqrt{2}z_{1n}, \sqrt{2}z_{23}, \ldots, \sqrt{2}z_{2n}, \ldots, \sqrt{2}z_{(n-1)n})^T$; ▷ # d

Algorithm 2 Compute $D2E_X(v_X)$

Input: $X = LL^T \in \mathcal{S}^{\mathrm{n}}_{++}$, $v_x \in \mathbb{R}^{n(n+1)/2}$.

1: **for** $i = 1, \ldots, n$ **do** ▷ # n
2: $\quad \eta_{ii} = v_X(i)$;
3: **end for**
4: $k = n + 1$;
5: **for** $i = 1, \ldots, n$ **do** ▷ # $n^2 - n(n+1)/2$
6: $\quad$ **for** $j = i + 1, \ldots, n$ **do**
7: $\quad\quad \eta_{ij} = v_X(k)$ and $\eta_{ji} = v_X(k)$;
8: $\quad\quad k = k + 1$;
9: $\quad$ **end for**
10: **end for**
11: return $L\eta L^T$; ▷ # $2n^3$

2.2 Retraction and vector transport

The concepts of retraction and vector transport can be found in [2]. 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 all $\xi_x,\ \zeta_x \in \mathrm{T}_x\,\mathcal{M}$, 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 method in [18, Algorithm 1] and LRBFGS requires the vector transport, denoted by $\mathcal{T}_S$, to be isometric, i.e., $g_{R(\eta_x)}(\mathcal{T}_{S_{\eta_x}}\xi_x, \mathcal{T}_{S_{\eta_x}}\zeta_x) = g_x(\xi_x, \zeta_x)$. 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 affine-invariant Riemannian metric (1.1), the exponential mapping is given by, see [11],

$$\mathrm{Exp}_X(\eta_X) = X^{1/2}\exp(X^{-1/2}\eta_X X^{-1/2})X^{1/2}, \tag{2.3}$$

for all $X \in \mathcal{S}^n_{++}$ and $\eta_X \in \mathrm{T}_X\,\mathcal{S}^n_{++}$. In practice, the exponential mapping (2.3) 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 [12, Figure 8.6.1]. Hence the evaluation of $\exp(M)$ requires $16n^3$ flops in total. More importantly, when computing the matrix exponential $\exp(M)$, eigenvalues of large magnitude can lead to numerical difficulties, such as overflow. Jeuris et al. [23] proposed a retraction

$$R_X(\eta_X) = X + \eta_X + \frac{1}{2}\eta_X X^{-1}\eta_X, \tag{2.4}$$

which is a second order approximation to exponential mapping (2.3). Retraction (2.4) is cheaper to compute and requires $3n^3 + o(n^3)$ flops. Retraction (2.4) also tends to avoid numerical overflow. An important property of retraction (2.4) is stated in Proposition 2.1. Another retraction that can be computed efficiently is the first order approximation to (2.3), i.e.,

$$R_X(\eta_X) = X + \eta_X. \tag{2.5}$$

In fact, retraction (2.5) is the exponential mapping when $\mathcal{S}^{\mathrm{n}}_{++}$ is endowed with the Euclidean metric. However, the result of retraction (2.5) is not guaranteed to be positive definite. Therefore one has to be careful when using this Euclidean retraction. One remedy is to reduce the stepsize when necessary. Richardson-like iteration in [7] is a steepest descent method using Euclidean retraction (2.5).

Proposition 2.1. *Retraction $R_X(\eta)$ defined in* (2.4) *remains symmetric positive-definite for all $X \in \mathcal{S}^n_{++}$ and $\eta \in \mathrm{T}_X\,\mathcal{S}^{\mathrm{n}}_{++}$.*

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

$$\begin{aligned} X + \eta + \frac{1}{2}\eta X^{-1}\xi &= \frac{1}{2}(X + 2\eta + \eta X^{-1}\eta) + \frac{1}{2}X \\ &= \frac{1}{2}(X^{1/2} + \eta X^{-1/2})(X^{1/2} + \eta X^{-1/2})^T + \frac{1}{2}X \end{aligned} \tag{2.6}$$

Then, for any $v \neq 0$,

$$\begin{aligned} v^T R_X(\eta) v &= \frac{1}{2} v^T (X^{1/2} + \eta X^{-1/2})(X^{1/2} + \eta X^{-1/2})^T v + \frac{1}{2} v^T X v \\ &= \frac{1}{2} \{(X^{1/2} + \eta X^{-1/2})^T v\}^T \{(X^{1/2} + \eta X^{-1/2})^T v\} + \frac{1}{2} v^T X v \\ &> 0 \end{aligned} \tag{2.7}$$

□

Parallel translation is a particular instance of vector transport. The parallel translation on $\mathcal{S}_{++}^n$ is given by, see [11],

$$\mathcal{T}_{p_{\xi_X}}(\eta_X) = X^{1/2} \exp(\frac{X^{-1/2} \xi_X X^{-1/2}}{2}) X^{-1/2} \eta_X X^{-1/2} \exp(\frac{X^{-1/2} \xi_X X^{-1/2}}{2}) X^{1/2}. \tag{2.8}$$

The computation of parallel translation involves matrix exponential, which is computationally expensive. We want to point out that if parallel translation is used together with exponential mapping (2.3), the most expensive exponential computation can be shared by rewriting (2.8) and (2.3) as shown in Algorithm 4. Even so, the matrix exponential computation is still required. We will thus resort to another vector transport.

Recently, Huang et al. [14, Section 2.3.1] proposed a novel way to construct an isometric vector transport, called vector transport by parallelization. It is defined by

$$\mathcal{T}_S = B_Y B_X^\flat, \tag{2.9}$$

where B_X and B_Y are orthonormal bases of $\mathrm{T}_X \mathcal{S}_{++}^{\mathrm{n}}$ and $\mathrm{T}_Y \mathcal{S}_{++}^{\mathrm{n}}$ defined in (2.1) respectively. Let $v_X = B_X^\flat \eta_X$ be the intrinsic representation of η_X. Then, the intrinsic approach of (2.9), denoted by $\mathcal{T}_S^d$, is given by

$$\begin{aligned} \mathcal{T}_S^d v_X &= B_Y^\flat \mathcal{T}_S \eta_X \\ &= B_Y^\flat B_Y B_X^\flat B_X v_X \\ &= v_X \end{aligned} \tag{2.10}$$

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 method since it is not isometric under Riemannian metric (1.1).

Specifying a retraction, Huang et al. [18, Section 2] provides a method to construct an isometric vector transport such that the pair satisfies the locking condition[4], denoted by $\mathcal{T}_L$, which is given by

$$\mathcal{T}_{L_{\xi_X}} \eta_X = B_Y (I - \frac{2 v_2 v_2^T}{v_2^T v_2})(I - \frac{2 v_1 v_1^T}{v_1^T v_1}) B_1^\flat \eta_X, \tag{2.11}$$

[4]see [18, Section 2 Equation (2.8)] for the definition of locking condition

where $v_1 = B_X^\flat \xi_X - w$, $v_2 = w - \beta B_Y^\flat \mathcal{T}_{R_{\xi_X}} \xi_X$, $\beta = \|\xi_X\|/\|\mathcal{T}_{R_{\xi_X}} \xi_X\|$, and $\mathcal{T}_R$ denotes the differentiated retraction. w can be any vector satisfying $\|w\| = \|B_1^\flat \xi_X\| = \|\beta B_2^\flat \mathcal{T}_{R_{\xi_X}} \xi_X\|$. $w = -B_1^\flat \xi_X$ and $w = -\beta B_2^\flat \xi_X$ are natural choices. The intrinsic representation of (2.11) is given by

$$\mathcal{T}_L^d v_X = B_Y^\flat B_Y (I - \frac{2v_2 v_2^T}{v_2^T v_2})(I - \frac{2v_1 v_1^T}{v_1^T v_1}) B_1^\flat \eta_X \tag{2.12}$$

$$= (I - \frac{2v_2 v_2^T}{v_2^T v_2})(I - \frac{2v_1 v_1^T}{v_1^T v_1}) v_X. \tag{2.13}$$

The evaluation of intrinsic vector transport (2.13) requires $12d$ flops, i.e., $6n^2$.

2.3 Riemannian gradient of the sum of squared distances function

The cost function (1.2) can be rewritten as

$$f(X) = \frac{1}{2K} \sum_{i=1}^{K} \| \log(A_i^{-1/2} X A_i^{-1/2}) \|_F^2 \tag{2.14}$$

$$= \frac{1}{2K} \sum_{i=1}^{K} \| \log(L_{A_i}^{-1} X L_{A_i}^{-T}) \|_F^2 \tag{2.15}$$

where $A_i = L_{A_i} L_{A_i}^T$. We use Cholesky factorization rather than the matrix square root due to computational efficiency. The matrix logarithm is computed in a similar way as exponential, i.e., $\log(M) = U \log(\Sigma) U^T$ with $M = U \Sigma U^T$ being the eigenvalue decomposition. Hence the number of flops required by function evaluation (2.15) is $18Kn^3$.

The Riemannian gradient of cost function F in (1.2) is given by, see [24],

$$\operatorname{grad} F(X) = -\frac{1}{K} \sum_{i=1}^{K} \operatorname{Exp}_X^{-1}(A_i), \tag{2.16}$$

where $\operatorname{Exp}_X^{-1}(Y)$ is the log-mapping, i.e., the inverse exponential mapping. On $\mathcal{S}_{++}^n$, the log-mapping is computed as

$$\operatorname{Exp}_X^{-1}(Y) = X^{1/2} \log(X^{-1/2} Y X^{-1/2}) X^{1/2} = \log(Y X^{-1}) X. \tag{2.17}$$

Note that the computational complexity of the Riemannian gradient is less than that conveyed in formula (2.17) since the most expensive logarithm computation is already available from the evaluation of the cost function at X. Specifically, each term in (2.16) is computed as $-\operatorname{Exp}_X^{-1}(A_i) = -\log(A_i X^{-1}) X = \log(X A_i^{-1}) X = L_{A_i} \log(L_{A_i}^{-1} X L_{A_i}^{-T}) L_{A_i}^{-1} X$, and the term $\log(L_{A_i}^{-1} X L_{A_i}^{-T})$ is available from the evaluation of the cost function $F(X)$ in (2.15). Hence the computation of gradient requires extra $5Kn^3$ flops if $\log(L_{A_i}^{-1} X L_{A_i}^{-T})$ is given.

3 Description of the SPD Karcher mean computation methods

In this section, we present detailed descriptions of the considered algorithms for SPD Karcher mean computation, including a limited-memory Riemannian RBFGS (LRBFGS) [18], Riemannian

Barzilai-Borwein (RBB) [20], Riemannian steepest descent with adaptive stepsize selection rule proposed by Quentin Rentmeesters (RSD-QR) [31], and Richardson-like iteration (RL) [7]. All of these algorithms are retraction-based methods, namely, the iterate x_k on the manifold $\mathcal{M}$ is updated by

$$x_{k+1} = R_{x_k}(\alpha_k \eta_k), \tag{3.1}$$

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 (3.1) is taken as the negative gradient, i.e., $\eta_k = -\operatorname{grad} f(x_k)$. RSD for the SPD Karcher mean computation is summarized in Algorithm 3 based on [31, Algorithm 2]. The difference between RSD-QR and RL is the choice of stepsize strategy in Step 11 and retraction in Step 13 in Algorithm 3. For RL, the stepsize is taken as $\alpha_{RL} = 1/\Delta$, where Δ is the upper bound on the Hessian of the cost function as computed in Step 10, and the Euclidean retraction (2.5) 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 Hessian of the cost function. It is easy to verify that $1/\Delta \leq 2/(1 + \Delta)$, and the equal sign holds when $\Delta = 1$. Since the eigenvalues of the Hessian of the cost function is 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 (2.3) is used by RSD-QR in [31]. In practice, we use retraction (2.4), 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 requires fewer iterations to achieve the desired tolerance. We will see later in our experiments that RSD-QR performs very well on small-size problems in terms of time efficiency, and it consistently outperforms RL in various situations.

Algorithm 3 RSD for the SPD Karcher mean computation

Input: $A_i = L_{A_i} L_{A_i}^T$; tolerance for stopping criteria ϵ; initial iterate $x_0 \in \mathcal{S}_{++}^{\mathrm{n}}$;

1: $k = 0$;
2: **while** $\|\operatorname{grad} f(x_k)\| > \epsilon$ **do**
3: **for** $i = 1, \ldots, K$ **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 $\lambda = \operatorname{diag}(\Sigma)$; ▷ # $12n^3$
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 Hessian $\Delta = \frac{1}{K} \sum_{j=1}^{K} \frac{\log c_j}{2} \coth(\frac{\log c_j}{2})$; ▷ # $5K$
11: Compute stepsize $\alpha_k = \alpha(\Delta)$;
12: Compute $\operatorname{grad} f(x_k) = \frac{1}{K} \sum_{i=1}^{K} G_i$; ▷ # $(K+1)n^2$
13: Compute $x_{k+1} = R_{x_k}(-\alpha_k \operatorname{grad} f(x_k))$;
14: $k = k + 1$;
15: **end while**

The 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 [20] for details. The BB stepsize is taken as

$$\alpha_{k+1}^{BB} = \frac{g(s_k, s_k)}{g(s_k, y_k)}, \tag{3.2}$$

where $s_k = \mathcal{T}_{\alpha_k \eta_k}(\alpha_k \eta_k)$, $y_k = \operatorname{grad} f(x_{k+1}) - \mathcal{T}_{\alpha_k \eta_k}(\operatorname{grad} f(x_k))$, and $g(s_k, y_k) > 0$. Another form of the BB stepsize is

$$\alpha_{k+1}^{BB} = \frac{g(s_k, y_k)}{g(y_k, y_k)}. \tag{3.3}$$

In [20], the RBB method has been applied to the SPD Karcher mean computation. We summarize their implementation in Algorithm 4, where they use extrinsic representation of a tangent vector, exponential mapping (2.3) and parallel translation (2.8). Algorithm 5 states our implementation of RBB using the intrinsic representation technique, retraction (2.4) and vector transport by parallelization (2.10). 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 7 and Step 11 in Algorithm 4 share the common term $\exp(\alpha_k x_k^{-1} \eta_k)$, which dominates the computational time and needs to be computed only once. Having $w = n^2$ and $d = n(n+1)/2$, the known number of flops per iteration for Algorithm 4 and Algorithm 5 are $103n^3/3 + o(n^3)$ 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 Riemannian metric reduces to the Euclidean metric using the intrinsic representation of tangent vector.

Our previous work [35] tailors the LRBFGS method in [18, 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 $\eta_k = -\mathcal{B}_k^{-1} \operatorname{grad} f(x_k)$, 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 [18, Algorithm 1] for the update formula. Unlike BFGS, the limited-memory version of BFGS stores only some 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.

As a continuation of our work in [35], we provide concrete LRBFGS methods for the SPD Karcher mean computation in Algorithm 6 and 7, so that readers are able to implement the methods conveniently. In fact, those two algorithms are ready to solve any optimization problems on the manifold of SPD matrices 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 metric, retraction, and vector transport evaluation respectively, and use superscript—w and d—to denote the extrinsic and intrinsic representations respectively. By summing them up, the number of flops per iteration for Algorithm 6 is

$$\#^w = 2(l+2)\lambda_m^w + 4lw + \lambda_r^w + 4w + 2(l+1)\lambda_t^w, \tag{3.8}$$

and the number of flops for Algorithm 7 is

$$\#^d = 2(l+2)\lambda_m^d + 4ld + \lambda_r^d + 4d + (13n^3/3 + 2d) \tag{3.9}$$

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

Input: backtracking reduction factor $\varrho \in (0,1)$; Armijo parameter $\delta \in (0,1)$; tolerance for stopping criteria ϵ; accuracy at which the linesearch procedure is skipped; maximum stepsize α_{max}, minimum stepsize α_{min}; initial iterate $x_0 \in \mathcal{S}^{\mathrm{n}}_{++}$; the first stepsize α_0^{BB};

1: $k = 0$;
2: Compute $f(x_k)$, $\operatorname{grad} f(x_k)$;
3: **while** $\|\operatorname{grad} f(x_k)\| > \epsilon$ **do**
4: Set stepsize $\alpha_k = \alpha_k^{BB}$;
5: Set $\eta_k = -\operatorname{grad} f(x_k)$; ▷ # w
6: If

$$\|\operatorname{grad}\operatorname{grad} f(x_k)\| / \|\operatorname{grad} f(x_0)\| < accuracy \tag{3.4}$$

then set $x_{k+1} = x_k \exp(\alpha_k x_k^{-1}\eta_k)$ and go to Step 10;
7: Compute $\tilde{x}_k = x_k \exp(\alpha_k x_k^{-1}\eta_k)$; ▷ # $61n^3/3$
8: If

$$f(\tilde{x}_k) \leq f(x_k) + \delta\alpha_k g(\operatorname{grad} f(x_k), \eta_k), \tag{3.5}$$

then set $x_{k+1} = \tilde{x}_k$ and go to Step 10;
9: Set $\alpha_k = \varrho\alpha_k$ and go to Step 7;
10: Compute $\operatorname{grad} f(x_{k+1})$;
11: Compute $s_k = \alpha_k\eta_k \exp(\alpha_k x_k^{-1}\eta_k)$, $y_k = \operatorname{grad} f(x_{k+1}) + \eta_k \exp(\alpha_k x_k^{-1}\eta_k)$; ▷ # $2n^3 + n^2$
12: Compute $\alpha_{k+1}^{BB} = g(s_k, s_k)/g(s_k, y_k)$; ▷ # $12n^3$
13: If $g(s_k, y_k) > 0$, set $\alpha_{k+1}^{BB} = \min\{\alpha_{max}, \max\{\epsilon, \alpha_{k+1}^{BB}\}\}$; otherwise, set $\alpha_{k+1}^{BB} = \alpha_{max}$;
14: $k = k + 1$;
15: **end while**

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

Input: backtracking reduction factor $\varrho \in (0,1)$; Armijo parameter $\delta \in (0,1)$; tolerance for stopping criteria ϵ; accuracy at which the linesearch procedure is skipped; maximum stepsize α_{max}, minimum stepsize α_{min}; initial iterate $x_0 \in \mathcal{M}$; the first stepsize α_0^{BB};

1: $k = 0$;
2: Compute $\operatorname{grad} f(x_k)$;
3: Compute $x_k = L_k L_k^T$, which is the required input of Algorithm 1 in the next step; ▷ $\#n^3/3$
4: Compute the intrinsic representation gf_k^d of $\operatorname{grad} f(x_k)$ 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: If

$$\|\operatorname{grad} \mathrm{gf}_k^d\| / \|\mathrm{gf}_0^d\| < accuracy \tag{3.6}$$

then set $x_{k+1} = R_{x_k}(\alpha_k \eta_k^w)$ using (2.4) and go to Step 13;
10: Compute $\tilde{x}_k = R_{x_k}(\alpha_k \eta_k^w)$ using (2.4) ; ▷ $\#\ 3n^3 + 3n^2$
11: If

$$f(\tilde{x}_k) \leq f(x_k) + \delta \alpha_k \eta_k^T \mathrm{gf}_k^d, \tag{3.7}$$

then set $x_{k+1} = \tilde{x}_k$ and go to Step 13;
12: Set $\alpha_k = \varrho \alpha_k$ and go to Step 10;
13: Compute $\operatorname{grad} f(x_{k+1})$;
14: Compute $x_{k+1} = L_{k+1} L_{k+1}^T$, which is the input of Algorithm 1 in the next step; ▷ $\#n^3/3$
15: Compute the intrinsic representation gf_{k+1}^d of $\operatorname{grad} f(x_{k+1})$ 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 $\alpha_{k+1}^{BB} = s_k^T s_k / s_k^T y_k$; ▷ $\#\ 4d$
18: If $s_k^T y_k > 0$, set $\alpha_{k+1}^{BB} = \min\{\alpha_{max}, \max\{\epsilon, \alpha_{k+1}^{BB}\}\}$; otherwise, set $\alpha_{k+1}^{BB} = \alpha_{max}$;
19: $k = k + 1$;
20: **end while**

Notice that there is no λ_t^d term in equation (3.9) since the vector transport by parallelization is used, which has identity implementation. The last term $(13n^3/3 + 2d)$ in (3.9) comes from the evaluation of functions $E2D$ and $D2E$ given in Algorithm 1 and 2. For the metric evaluation using different representations, we have $\lambda_i^w = 6n^3 + o(n^3)$ and $\lambda_i^d = n^2 + o(n^2)$. Simplifying and rearranging (3.8) and (3.9), we have

$$\#^w = 12ln^3 + 24n^3 + \lambda_r^w + 2(l+1)\lambda_t^w + o(ln^3) + o(n^3), \tag{3.10}$$

and

$$\#^d = 4ln^2 + 13n^3/3 + \lambda_r^d + o(ln^2) + o(n^3). \tag{3.11}$$

From (3.10) and (3.11), the computational benefit of the intrinsic representation is substantial. The limited-memory size l imposes much heavier burden on Algorithm 6 where the extrinsic representation is used. In our implementation of Algorithm 7, we suggest retraction (2.4), 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 with Algorithm 7. Notice that the identity vector transport using extrinsic representation is not applicable since it is not isometric under metric (1.1).

However, our complexity analysis above focuses on manifold- and algorithm-related operations, the problem-related operations—function, gradient evaluations and line search—are not considered. From the discussion in Section 2.3, the evaluation of cost function requires $18Kn^3$ flops, and the computation of gradient requires extra $5Kn^3$ flops based on 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 21 in Algorithm 7, i.e., 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

$$\frac{23Kn^3}{23Kn^3 + 4ln^2 + 22n^3/3} \approx \frac{23K}{23K + 22/3} \tag{3.12}$$

$$\geq \frac{23 \cdot 3}{23 \cdot 3 + 22/3} \approx 90.39\% \tag{3.13}$$

Inequality (3.13) comes from the fact that $K \geq 3$ and (3.12) is an increasing function in terms of K. It is shown that the problem-related operations dominate the computation time for high dimensional matrix, which is consistent with our empirical observations in experiments that $70\% - 90\%$ timing comes 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.

We end this section by a simple remark. The LRBFGS with zero memory size, i.e., $m = 0$, is equivalent with the RBB using stepsize (3.3). This is easy to verify by setting $m = 0$ in Algorithm 6 and 7. Our numerical experiments on the SPD Karcher mean computation show that the two versions of the BB stepsize, (3.2) and (3.3), lead to similar results.

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

[6]If retraction (2.4) and isometric vector transport (2.13) that satisfy the locking condition are used, s_k and y_k

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

Input: backtracking reduction factor $\varrho \in (0,1)$; Armijo parameter $\delta \in (0,1)$; tolerance for stopping criteria ϵ; accuracy at which the linesearch procedure is skipped;3 maximum stepsize α_{max}, minimum stepsize α_{min}; initial iterate $x_0 \in \mathcal{M}$; an integer $m > 0$; the first stepsize γ_0;

$k = 0$, $l = 0$.
Compute $\operatorname{grad} f(x_k)$;
while $\|\operatorname{grad} f(x_k)\| > \epsilon$ **do**
  $\mathcal{H}_k^0 = \gamma_k \operatorname{id}$. Obtain $\eta_k \in \mathrm{T}_{x_k}\mathcal{M}$ by the following algorithm, Step 5 to Step 15:
  $q \leftarrow \operatorname{grad} f(x_k)$
  for $i = k-1, k-2, \ldots, k-l$ **do** ▷ # $l(\lambda_m^w + 2w)$
    $\xi_i \leftarrow \rho_i g(s_i^{(k)}, q)$;
    $q \leftarrow q - \xi_i y_i^{(k)}$;
  end for
  $r \leftarrow \mathcal{H}_k^0 q$; ▷ # w
  for $i = k-l, k-l+1, \ldots, k-1$ **do** ▷ # $l(\lambda_m^w + 2w)$
    $\omega \leftarrow \rho_i g(y_i^{(k)}, r)$;
    $r \leftarrow r + s_i^{(k)}(\xi_i - \omega)$;
  end for
  Set $\eta_k = -r$, $\alpha_k = 1$; ▷ # w
  If

$$\|\operatorname{grad} f(x_k)\| / \|\operatorname{grad} f(x_0)\| < accuracy \tag{3.14}$$

  then set $x_{k+1} = R_{x_k}(\alpha_k \eta_k)$ and go to Step 20; ▷ # λ_r^w
  Compute $\tilde{x}_k = R_{x_k}(\alpha_k \eta_k)$; ▷ # λ_r^w
  If

$$f(\tilde{x}_k) \leq f(x_k) + \delta \alpha_k g(\operatorname{grad} f(x_k), \eta_k),$$

  then set $x_{k+1} = \tilde{x}_k$ and go to Step 20;
  Set $\alpha_k = \varrho \alpha_k$ and go to Step 17;
  Compute $\operatorname{grad} f(x_{k+1})$;
  Define $s_k^{(k+1)} = \mathcal{T}_{\alpha_k \eta_k} \alpha_k \eta_k$ and $y_k^{(k+1)} = \operatorname{grad} f(x_{k+1}) - \mathcal{T}_{\alpha_k \eta_k} \operatorname{grad} f(x_k)$; [5] ▷ # $2\lambda_t^w + 2w$
  Compute $a = g(y_k^{(k+1)}, s_k^{(k+1)})$ and $b = \|s_k^{(k+1)}\|^2$; ▷ # $2\lambda_m^w$
  if $\frac{a}{b} \geq 10^{-4} \|\operatorname{grad} f(x_k)\|$ **then** ▷ # λ_m^w
    Compute $c = \|y_k^{(k+1)}\|^2$ and define $\rho_k = 1/a$ and $\gamma_{k+1} = a/c$; ▷ # λ_m^w
    Add $s_k^{(k+1)}$, $y_k^{(k+1)}$ and ρ_k into storage and if $l \geq m$, then discard vector pair $\{s_{k-l}^{(k)}, y_{k-l}^{(k)}\}$ and scalar ρ_{k-l} from storage, else $l \leftarrow l+1$; Transport $s_{k-l+1}^{(k)}, s_{k-l+2}^{(k)}, \ldots, s_{k-1}^{(k)}$ and $y_{k-l+1}^{(k)}, y_{k-l+2}^{(k)}, \ldots, y_{k-1}^{(k)}$ from $\mathrm{T}_{x_k}\mathcal{M}$ to $\mathrm{T}_{x_{k+1}}\mathcal{M}$ by $\mathcal{T}$, then get $s_{k-l+1}^{(k+1)}, s_{k-l+2}^{(k+1)}, \ldots, s_{k-1}^{(k+1)}$ and $y_{k-l+1}^{(k+1)}, y_{k-l+2}^{(k+1)}, \ldots, y_{k-1}^{(k+1)}$; ▷ # $2(l-1)\lambda_t^w$
  else
    Set $\gamma_{k+1} \leftarrow \gamma_k$, $\{\rho_k, \ldots, \rho_{k-l+1}\} \leftarrow \{\rho_{k-1}, \ldots, \rho_{k-l}\}$, $\{s_k^{(k+1)}, \ldots, s_{k-l+1}^{(k+1)}\} \leftarrow \{\mathcal{T}_{\alpha_k \eta_k} s_{k-1}^{(k)}, \ldots, \mathcal{T}_{\alpha_k \eta_k} s_{k-l}^{(k)}\}$ and $\{y_k^{(k+1)}, \ldots, y_{k-l+1}^{(k+1)}\} \leftarrow \{\mathcal{T}_{\alpha_k \eta_k} y_{k-1}^{(k)}, \ldots, \mathcal{T}_{\alpha_k \eta_k} y_{k-l}^{(k)}\}$; ▷ # $2l\lambda_t^w$
  end if
  $k = k+1$;
end while

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

Input: backtracking reduction factor $\varrho \in (0,1)$; Armijo parameter $\delta \in (0,1)$; tolerance for stopping criteria ϵ; accuracy at which the linesearch procedure is skipped; maximum stepsize α_{max}, minimum stepsize α_{min}; initial iterate $x_0 \in \mathcal{M}$; an integer $m > 0$; the first stepsize γ_0;

1: $k = 0$, $l = 0$.
2: Compute $\operatorname{grad} f(x_k)$;
3: Compute $x_k = L_k L_k^T$, which is the input of Algorithm 1 in the next step; ▷ $\#n^3/3$
4: Compute the intrinsic representation gf_k^d of $\operatorname{grad} f(x_k)$ by Algorithm 1; ▷ $\#\ 2n^3 + d$
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** ▷ $\#\ l(\lambda_m^d + 2d)$
9: $\xi_i \leftarrow \rho_i q^T s_i$;
10: $q \leftarrow q - \xi_i y_i$;
11: **end for**
12: $r \leftarrow \gamma_k q$; ▷ $\#\ d$
13: **for** $i = k-l, k-l+1, \ldots, k-1$ **do** ▷ $\#\ l(\lambda_m^d + 2d)$
14: $\omega \leftarrow \rho_i r^T y_i$;
15: $r \leftarrow r + s_i(\xi_i - \omega)$;
16: **end for**
17: set $\eta_k = -r$, $\alpha_k = 1$; ▷ $\#\ d$
18: Compute $\eta_k^w = D2E_{x_k}(\eta_k)$ by Algorithm 2; ▷ $\#\ 2n^3 + n(n+1)/2$
19: If

$$\|\operatorname{grad} \mathrm{gf}_k^d\| / \|\mathrm{gf}_0^d\| < accuracy \tag{3.15}$$

then set $x_{k+1} = R_{x_k}(\alpha_k \eta_k^w)$ and go to Step 23; ▷ $\#\ \lambda_r^d$
20: Compute $\tilde{x}_k = R_{x_k}(\alpha_k \eta_k^w)$; ▷ $\#\ \lambda_r^d$
21: If

$$f(\tilde{x}_k) \le f(x_k) + \delta \alpha_k \eta_k^T \mathrm{gf}_k^d,$$

then set $x_{k+1} = \tilde{x}_k$ and go to Step 23;
22: Set $\alpha_k = \varrho \alpha_k$ and go to Step 20;
23: Compute $\operatorname{grad} f(x_{k+1})$;
24: Compute $x_{k+1} = L_{k+1} L_{k+1}^T$, which is the input of Algorithm 1 in the next step; ▷ $\#n^3/3$
25: Compute the intrinsic representation gf_{k+1}^d of $\operatorname{grad} f(x_{k+1})$ by Algorithm 1; ▷ $\#\ 2n^3 + d$
26: Define $s_k = \alpha_k \eta_k$ and $y_k = \mathrm{gf}_{k+1}^d - \mathrm{gf}_k^d$; [6] ▷ $\#\ 2d$
27: Compute $a = y_k^T s_k$ and $b = \|s_k\|_2^2$; ▷ $\#\ 2\lambda_m^d$
28: **if** $\frac{a}{b} \ge 10^{-4} \|\mathrm{gf}_k^d\|_2$ **then** ▷ $\#\ \lambda_m^d$
29: Compute $c = \|y_k^{(k+1)}\|_2^2$ and define $\rho_k = 1/a$ and $\gamma_{k+1} = a/c$; ▷ $\#\ \lambda_m^d$
30: Add s_k, y_k and ρ_k into storage and if $l \ge m$, then discard vector pair $\{s_{k-l}, y_{k-l}\}$ and scalar ρ_{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**

4 Experiments

In this section, we compare the performance of the LRBFGS method described in Algorithm 7 and existing state-of-the-art methods, including the Riemannian Barzilai-Borwein method (RBB) provided in [20] (using implementation in Algorithm 5), the Riemannian steepest descent method with stepsize selection rule proposed by Rentmeesters et al. (RSD-QR) in [31, Section 3.6], the Richardson-like iteration (RL) of [7], and the Riemannian BFGS method (RBFGS) presented in [16, 18].

All the experiments are performed on the Florida State University HPC system using Quad-Core AMD Opteron(tm) Processor 2356 2.3GHz. Experiments in Section 4.2 are carried out using C++, compiled with gcc-4.7.x. Section 4.3 presents a comparison of computation time between MATLAB and C++ implementations. All the MATLAB experiments are performed using MATLAB R2015b (8.6.0.267246) 64-bit (glnxa64). In particular, we use the MATLAB implementation of the RL iteration in Bini et al.'s Matrix Means Toolbox[1].

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 α_{min} is the machine epsilon. The initial stepsize in the first iteration is chosen as the strategy in [31], i.e., $\alpha_0 = 2/(1 + L)$, where L is the upper bound at the initial iterate defined in inequality (1.3). For LRBFGS, we use different memory sizes m as specified in the legends of figures, and impose the locking condition for ill-conditioned matrices. Specifically, we impose the locking condition on LRBFGS in the bottom plots of Figure 1, 2, 3, 4, and Figure 5. As we have shown in Section 3, 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 much 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 [16]. In order to achieve sufficient accuracy, we skip the line search procedure when the iterate is close enough to the minimizer by setting $accuracy = 10^{-5}$ in Algorithm 5 and 7. Unless otherwise specified, our choice of the initial iterate is the Arithmetic-Harmonic mean [22] of data points. We run the algorithms until they reach their highest accuracy.

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 5 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 5 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.

4.1 Experiment design

The experiments are generated in the same way as our previous work [35]. When defining each set of experiments, we choose a desired (true) Karcher mean μ, and construct data matrices A_i's such that their Karcher mean is exactly μ, i.e., $\sum_{i=1}^{K} \operatorname{Exp}_{\mu}^{-1}(A_i) = 0$ holds. The benefits of this scheme

are computed as following: 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 $s_k = (I - 2v_2 v_2^T/v_2^T v_2)(I - 2v_1 v_1^T/v_1^T v_1)(\alpha_k \eta_k)$, $y_k = \mathrm{gf}_{k+1}^d/\beta - (I - 2v_2 v_2^T/v_2^T v_2)(I - 2v_1 v_1^T/v_1^T v_1)\mathrm{gf}_k^d$.

are: (i) We can control the conditioning of μ and A_i's, and observe the influence of the conditioning on the performance of algorithms. (ii) Since the true Karcher mean 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 Matlab, with n the size of matrix, f the order of magnitude of the condition number, and p some number less than n,

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

(ii) Compute $\eta_i = \mathrm{Exp}_\mu^{-1}(W_i)$. (iii) Enforce the condition $\sum_{i=1}^K \eta_i = 0$ on η_i's. Specifically, we test on data sets with $K = 3,\ 30,\ 100$. In the case of $K = 3$, we enforce $\eta_3 = -\eta_1 - \eta_2$. When $K = 30$ or $K = 100$, let $k_i = 5(k-1)+i$ for $1 \le k \le K/5$ and $1 \le i \le 5$. We enforce $\eta_{k_4} = -\eta_{k_1} - 0.5\eta_{k_3}$ and $\eta_{k_5} = -\eta_{k_2} - 0.5\eta_{k_3}$, which gives $\sum_{i=1}^5 \eta_{k_i} = 0$, and thus $\sum_{k=1}^{K/5}\sum_{i=1}^5 \eta_{k_i} = 0$. (iv) Compute $A_i = \mathrm{Exp}_\mu(\eta_i)$.

Note that instead of producing η_i directly, we produce W_i first and obtain η_i from the log-mapping, since this procedure gives us greater control over the conditioning of data points. As discussed in [35, Section 2], we can take the Karcher mean μ as the identity matrix in numerical experiments, so we "translate" the data set $\{\mu, A_1, \ldots, A_K\}$ to $\{I, L^{-1}A_1L^{-T}, \ldots, L^{-1}A_KL^{-T}\}$ using an isometry, where $\mu = LL^T$ as the final step.

4.2 Comparison of performance between different algorithms using C++

We now compare the performances of all 5 algorithms on various data sets by examining performance results from representative experiments for different choices of (K, n, κ).

Figure 1 displays the performance results of different algorithms running on small-size problems, taking $K = 3$ and $n = 3$. Both well-conditioned ($1 \le \kappa(A_i) \le 20$) and ill-conditioned ($10^5 \le \kappa(A_i) \le 10^{10}$) data sets are tested. For LRBFGS, we take $m = 2$ and $m = 4$. In the well-conditioned case, it is seen that all 5 algorithms are comparable in terms of time efficiency even though they require different numbers of iterations. For ill-conditioned matrices, RL iteration and RSD-QR require significantly more iterations, but they are still efficient in terms of timing due to the cheap cost per iteration. RBB and LRBFGS require similar number of iterations, but LRBFGS with $m > 0$ takes longer time. We discussed in Section 3 that the overall complexities for Algorithm 7 without and with locking condition are $23Kn^3 + 22n^3/3 + 4ln^2 + o(ln^2) + o(n^3)$ and $23Kn^3 + 22n^3/3 + 16ln^2 + o(ln^2) + o(n^3)$ respectively. Therefore when the size of the problem is small, and the locking condition is imposed, the impact of memory size l is visible. But this is not the case when the size of the problem gets larger, as shown in Figure 2 and Figure 3. Note that m is the upper limit of the limited-memory size l.

Figure 2 and Figure 3 report the results of tests conducted on data sets with large K ($K = 100,\ n = 3$) and large n ($K = 30,\ n = 100$) respectively. Note that when $n = 100$, the dimension of manifold $\mathcal{S}_{++}^n$ is $d = n(n+1)/2 = 5050$. In each case, both well-conditioned and ill-conditioned data sets are tested. For well-conditioned matrices, we observe that LRBFGS and RBB perform similarly, with a slightly advantage for LRBFGS. The advantage of LRBFGS becomes larger as the matrices become ill-conditioned. We want to point out that RBFGS is inefficient for large n as shown in Figure 3.

As the last test in this section, we compare the performances of considered algorithms using two different initial iterates: the Arithmetic-Harmonic mean and the Cheap mean [8]. The Cheap mean is known to be a good approximation of the Karcher mean, which is however not cheap to compute. We resort to Bini et al.'s Matrix Means Toolbox[1] for the computation of Cheap mean. We consider 30 30×30 badly conditioned matrices ($10^6 \leq \kappa(A_i) \leq 10^{10}$). The results are presented in Figure 4. Notice that we include the time required to compute the initial iterate. The x-axis of the middle-right plot did not start from 0, which shows that the Cheap mean is expensive to compute. We observe that the choice of initial iterate is crucial to RBFGS, and it impacts the other algorithms in the early steps. When the initial iterate is close enough to the true solution, as shown in the second row in Figure 4, we observe a faster convergence in the first a few steps. In both cases, LRBFGS outperforms the other algorithms in terms of computation time and number of iterations per unit of accuracy required. We also investigate the robustness of LRBFGS and RBB to parameter setting, say the back tracking reduction factor ϱ. The plots in Figure 5 displays the results for varying ϱ. For LRBFGS, we take memory size $m = 8$. It is shown that the performance of LRBFGS is consistent for different values of ϱ, while RBB is sensitive to the choice of ϱ. We therefore conclude that LRBFGS appears to be the method of choice for computing the SPD Karcher mean, especially when working with ill-conditioned matrices.

4.3 Comparison between C++ and MATLAB implementations

In the end, we compare the time efficiency of the considered algorithms on the SPD Karcher mean computation using C++ and MATLAB. The results are reported in Figure 6. The first column indicates that C++ and MATLAB implementations are identical in terms of iterations. The second column displays the log-log plots of computation time vs. averaged distance between each iterate and the exact Karcher mean, from which we could tell the order of magnitude of computational time easily. For small-size problem, C++ implementation is faster than MATLAB by a factor of 100 or more. Especially for LRBFGS and RBB, which are implemented as a user friendly library. The MATLAB library machinery dominates the computation time, which is slower than C++ library by a factor of 1000. The factor gradually reduces as n or K gets larger. This phenomenon can be explained by the fact that when n and K are small, the difference of efficiency between C++ and Matlab is mainly due to the difference between compiled languages and interpreted languages. When n or K gets larger, the BLAS and LAPACK calls start to dominate the computation time, which leads to d decrease in the factor. In addition, we could also observe that the overhead of MATLAB library machinery becomes invisible for large-size problems.

5 Conclusion

In this paper, we present a concrete LRBFGS method for the SPD Karcher mean computation, and provide efficient implementation techniques. There are several alternatives to choose from for the representation of the tangent vector, retraction and vector transport. We provide theoretical suggestions on how to choose between the alternatives by complexity analysis.

Our numerical experiments provide empirical guideline to choose between various methods. We observe that RSD-QR and RL perform very well on small-size and well-conditioned problems, and RSD-QR is systematically faster than RL. LRBFGS is the winner on large-size or ill-conditioned problems in terms of number of iterations and time efficiency. It exhibits a robust behavior with

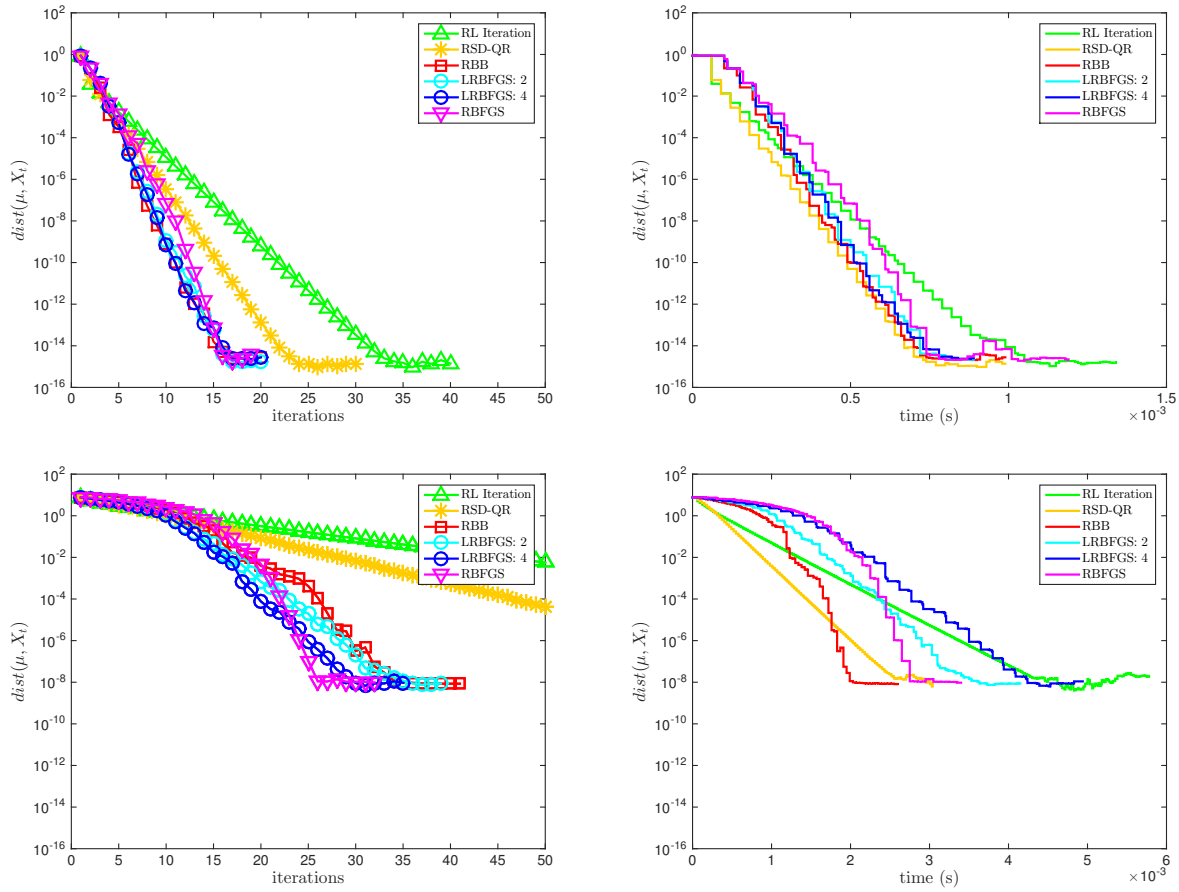

Figure 1: Evolution of averaged distance between current iterate and the exact Karcher mean with respect to time and iterations with $K = 3$, $n = 3$. Top: $1 \leq \kappa(A_i) \leq 20$; Bottom: $10^5 \leq \kappa(A_i) \leq 10^{10}$

respect to the conditioning of the problem and parameter setting. We also present empirical illustration of the speedup of C++ implementation compared with MATLAB implementation. Notice that it is demonstrated theoretically and empirically that for the large-size problems, the dominate computation time (70% - 90%) is on the problem-related operations, i.e., function and gradient evaluations. We believe that our implementations of manifold- and algorithm-related objects have reached the limit of efficiency.

A ALM list of properties for SPD matrix geometric means

P1 Consistency with scalars. If $A_1, \ldots, A_K$ commute then $G(A_1, \ldots, A_K) = (A_1 \cdots A_K)^{1/K}$.

P2 Joint homogeneity. $G(\alpha_1 A_1, \ldots, \alpha_K A_K) = (\alpha_1 \cdots \alpha_K)^{1/K} G(A_1, \ldots, A_K)$.

P3 Permutation invariance. For any permutation $\pi(A_1, \ldots, A_K)$ of $(A_1, \ldots, A_K)$, $G(A_1, \ldots, A_K) = G(\pi(A_1, \ldots, A_K))$.

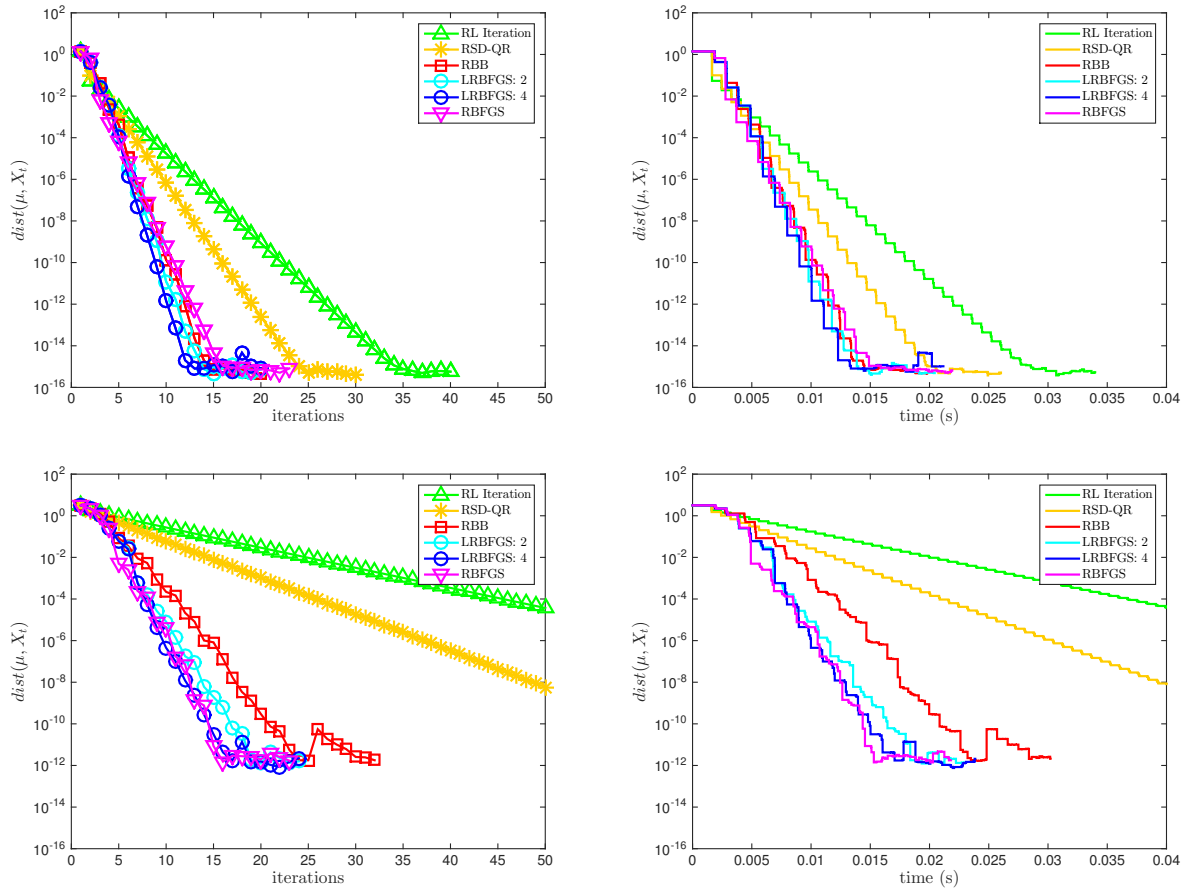

Figure 2: Evolution of averaged distance between current iterate and the exact Karcher mean with respect to time and iterations with $K = 100$ and $n = 3$; Top: $1 \leq \kappa(A_i) \leq 200$; Bottom: $10^3 \leq \kappa(A_i) \leq 10^7$

P4 Monotonicity. If $A_i \geq B_i$ for all i, then $G(A_1, \ldots, A_K) \geq G(B_1, \ldots, B_K)$ in the positive semidefinite ordering.

P5 Continuity from above. If $\{A_1^{(n)}\}, \ldots, \{A_K^{(n)}\}$ are monotonic decreasing sequences (in the positive semidefinite ordering) converging to $A_1, \ldots, A_K$, respectively, then $G(A_1^{(n)}, \ldots, A_K^{(n)})$ converges to $G(A_1, \ldots, A_K)$.

P6 Congruence invariance. $G(S^T A_1 S, \ldots, S^T A_K S) = S^T G(A_1, \ldots, A_K) S$ for any invertible S.

P7 Joint concavity. $G(\lambda A_1 + (1-\lambda)B_1, \ldots, \lambda A_K + (1-\lambda)A_K) \geq \lambda G(A_1, \ldots, A_K) + (1-\lambda) G(B_1, \ldots, B_K)$.

P8 Invariance under inversion. $G(A_1, \ldots, A_K)^{-1} = G(A_1^{-1}, \ldots, A_K^{-1})$.

P9 Determinant identity. $\det G(A_1, \ldots, A_K) = (\det A_1 \cdots \det A_K)^{1/K}$.

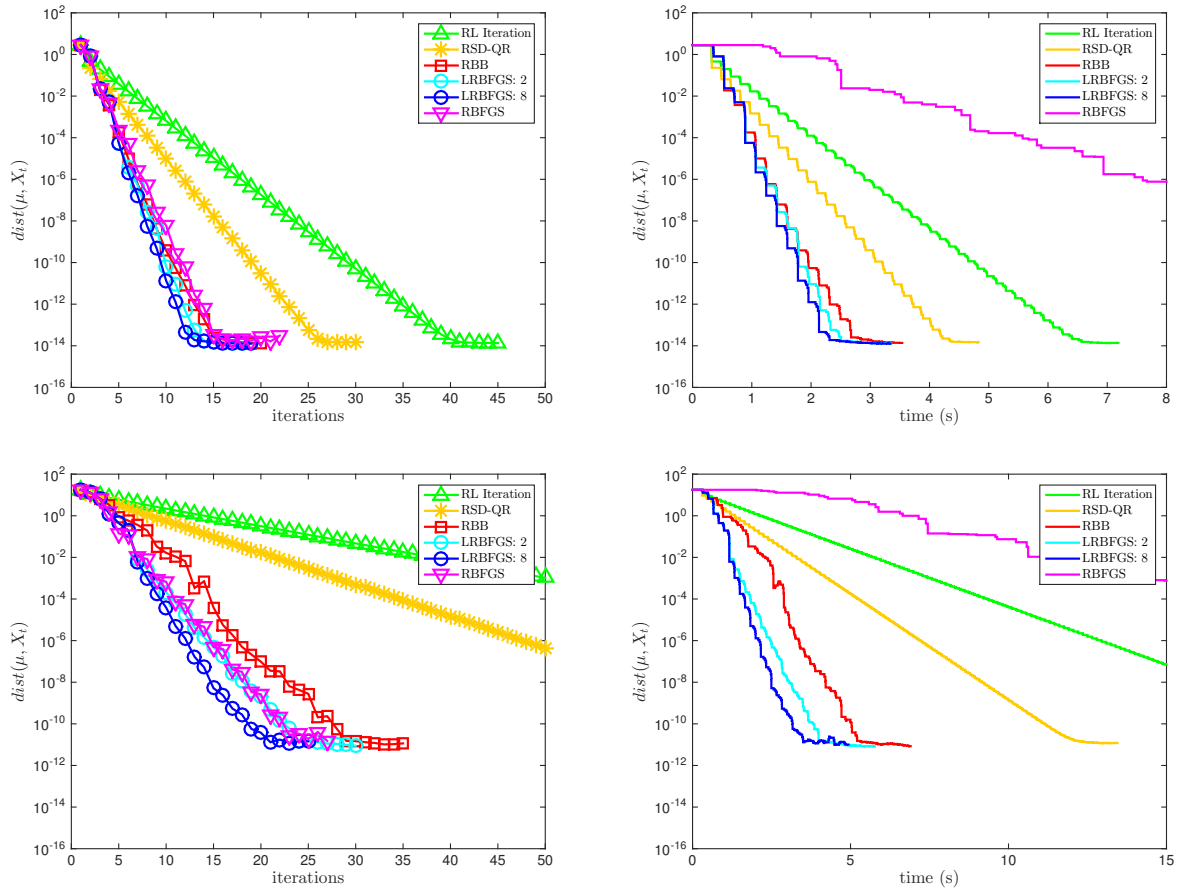

Figure 3: Evolution of averaged distance between current iterate and the exact Karcher mean with respect to time and iterations with $K = 30$ and $n = 100$; Top: $1 \leq \kappa(A_i) \leq 20$; Bottom: $10^4 \leq \kappa(A_i) \leq 10^7$

References

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

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

[3] T. Ando and R. Li, C.-K.and Mathias. Geometric means. *Linear Algebra and its Applications*, 385:305–334, 2004.

[4] Ognjen Arandjelovic, Gregory Shakhnarovich, John Fisher, Roberto Cipolla, and Trevor Darrell. Face recognition with image sets using manifold density divergence. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 581–588. IEEE, 2005.

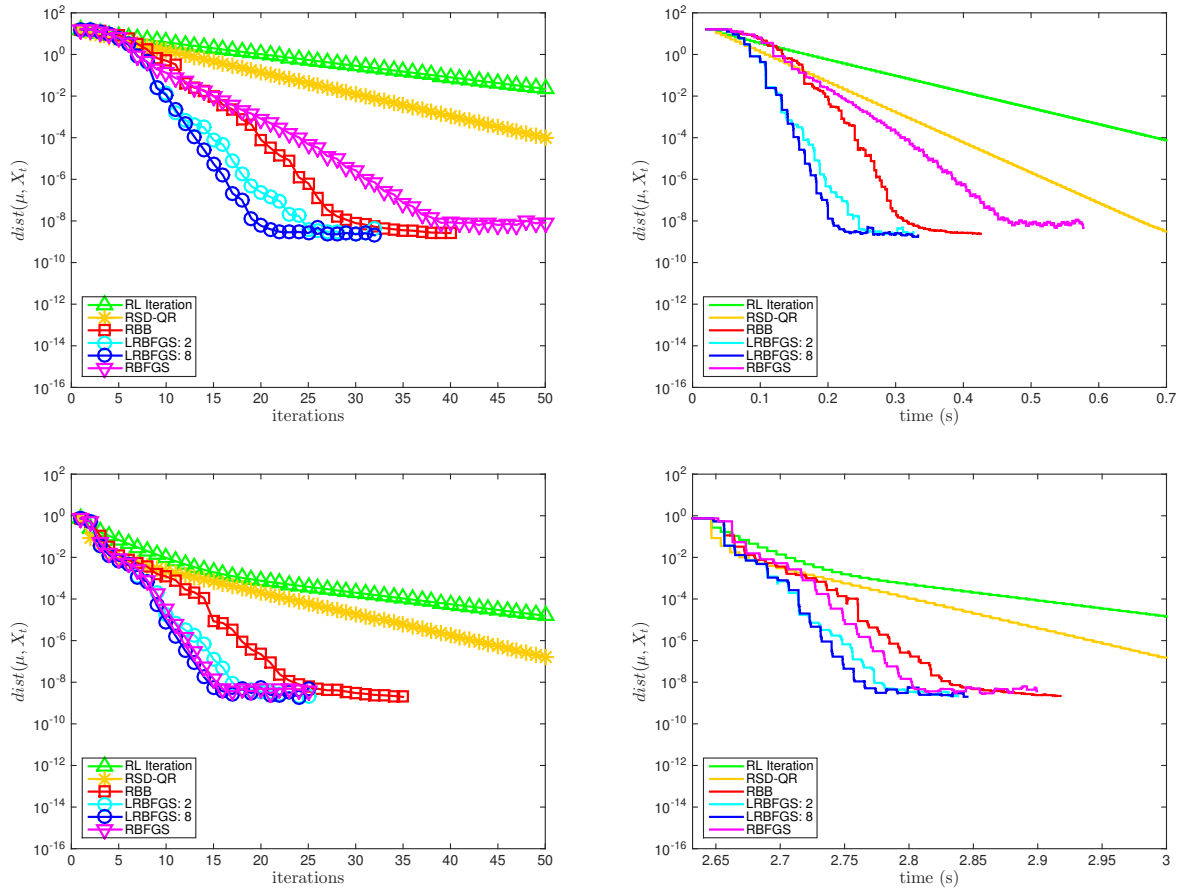

Figure 4: Comparison of different algorithms using different initial iterates with $K = 30$, $n = 30$, and $10^6 \leq \kappa(A_i) \leq 10^9$. Top: using the Arithmetic-Harmonic mean as initial iterate; Bottom: using the Cheap mean as initial iterate.

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

[6] Rajendra Bhatia and Rajeeva L Karandikar. Monotonicity of the matrix geometric mean. *Mathematische Annalen*, 353(4):1453–1467, 2012.

[7] D. A. Bini and B. Iannazzo. Computing the Karcher mean of symmetric positive definite matrices. *Linear Algebra and its Applications*, 438(4):1700–1710, 2013.

[8] Dario Andrea Bini and Bruno Iannazzo. A note on computing matrix geometric means. *Advances in Computational Mathematics*, 35(2-4):175–192, 2011.

[9] Guang Cheng, Hesamoddin Salehian, and Baba Vemuri. Efficient recursive algorithms for computing the mean diffusion tensor and applications to DTI segmentation. *Computer Vision–ECCV 2012*, pages 390–401, 2012.

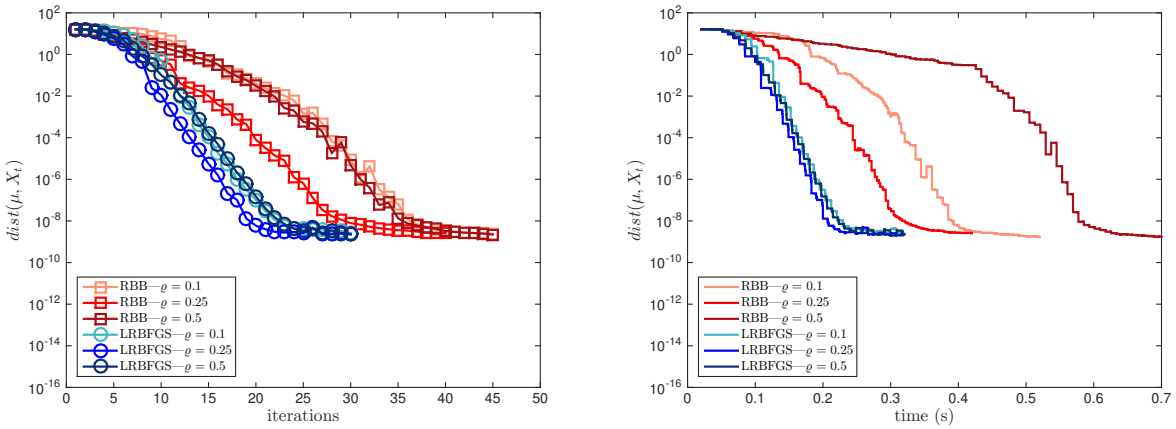

Figure 5: Comparison of RBB and LRBFGS with $m = 8$ using different back tracking reduction factor ϱ with $K = 30$, $n = 30$, and $10^6 \leq \kappa(A_i) \leq 10^9$.

[10] P. T. Fletcher and S. Joshi. Riemannian geometry for the statistical analysis of diffusion tensor data. *Signal Processing*, 87(2):250–262, 2007.

[11] P. T. Fletcher, C. Lu, S. M. Pizer, and S. Joshi. Principal geodesic analysis on symmetric spaces: statistics of diffusion tensors. *Computer Vision and Mathematical Methods in Medical and Biomedical Image Analysis*, 3117:87–98, 2004.

[12] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[13] Nicholas J Higham. *Functions of matrices: theory and computation*. SIAM, 2008.

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

[15] Wen Huang, P-A Absil, and Kyle A Gallivan. Intrinsic representation of tangent vectors and vector transports on matrix manifolds. *Numerische Mathematik*, pages 1–21, 2016.

[16] Wen Huang, P-A Absil, and Kyle A Gallivan. A Riemannian BFGS method for nonconvex optimization problems. In *Numerical Mathematics and Advanced Applications ENUMATH 2015*, pages 627–634. Springer, 2016.

[17] Wen Huang, PA Absil, KA Gallivan, and Paul Hand. ROPTLIB: an object-oriented C++ library for optimization on Riemannian manifolds. Technical report, Technical Report FSU16-14, Florida State University, 2016.

[18] Wen Huang, Kyle A Gallivan, and P-A Absil. A Broyden class of quasi-Newton methods for Riemannian optimization. *SIAM Journal on Optimization*, 25(3):1660–1685, 2015.

[19] Zhiwu Huang, Ruiping Wang, Shiguang Shan, and Xilin Chen. Face recognition on large-scale video in the wild with hybrid Euclidean-and-Riemannian metric learning. *Pattern Recognition*, 48(10):3113–3124, 2015.

[20] Bruno Iannazzo and Margherita Porcelli. The Riemannian Barzilai-Borwein method with nonmonotone line-search and the Karcher mean computation. *Optimization online, December*, 2015.

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

[22] B. Jeuris and R. Vandebril. Geometric mean algorithms based on harmonic and arithmetic iterations. In F. Nielsen and F. Barbaresco, editors, *Geometric Science of Information: First International Conference, GSI 2013, Paris, France, August 28-30, 2013. Proceedings*, pages 785–793. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

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

[24] H. Karcher. Riemannian center of mass and mollifier smoothing. *Communications on Pure and Applied Mathematics*, 1977.

[25] J. Lawson and Y. Lim. Monotonic properties of the least squares mean. *Mathematische Annalen*, 351(2):267–279, 2011.

[26] Jiwen Lu, Gang Wang, and Pierre Moulin. Image set classification using holistic multiple order statistics features and localized multi-kernel metric learning. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 329–336, 2013.

[27] M. Moakher. On the averaging of symmetric positive-definite tensors. *Journal of Elasticity*, 82(3):273–296, 2006.

[28] Yu Nesterov. Introductory lectures on convex programming volume I: basic course. *Lecture notes*, 1998.

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

[30] Y. Rathi, A. Tannenbaum, and O. Michailovich. Segmenting images on the tensor manifold. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, June 2007.

[31] Q. Rentmeesters. *Algorithms for data fitting on some common homogeneous spaces.* PhD thesis, Universite catholiqué de Louvain, 2013.

[32] Q. Rentmeesters and P.-A. Absil. Algorithm comparison for Karcher mean computation of rotation matrices and diffusion tensors. In *19th European Signal Processing Conference*, pages 2229–2233, Aug 2011.

[33] Gregory Shakhnarovich, John W Fisher, and Trevor Darrell. Face recognition from long-term observations. In *European Conference on Computer Vision*, pages 851–865. Springer, 2002.

[34] Oncel Tuzel, Fatih Porikli, and Peter Meer. Region covariance: A fast descriptor for detection and classification. In *European conference on computer vision*, pages 589–600. Springer, 2006.

[35] Xinru Yuan, Wen Huang, P-A Absil, and Kyle A Gallivan. A Riemannian limited-memory BFGS algorithm for computing the matrix geometric mean. *Procedia Computer Science*, 80:2147–2157, 2016.

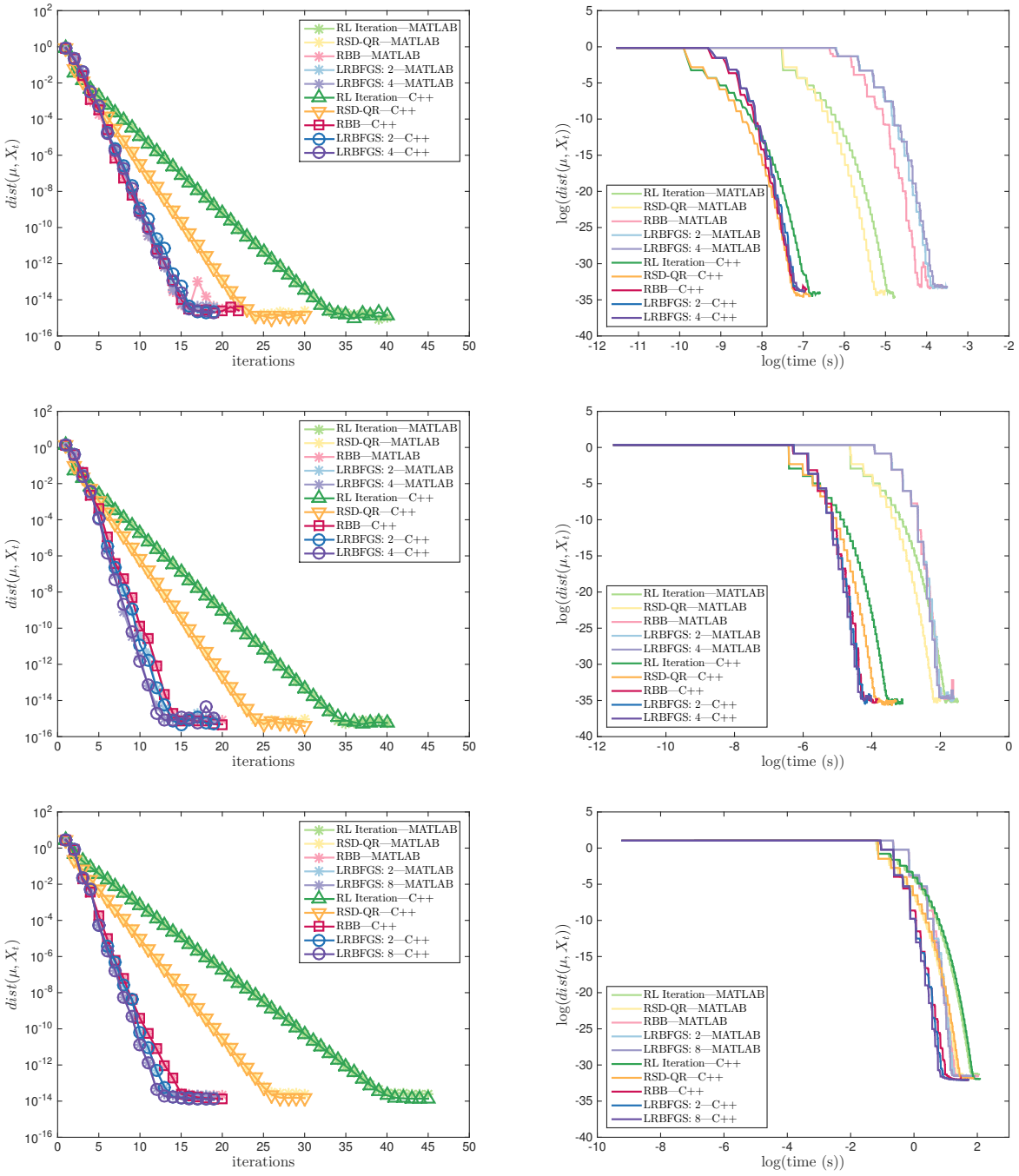

Figure 6: Comparison between C++ and MATLAB implementations with different choices of (K, n, κ). Top row: $K = 3$, $n = 3$, and $1 \leq \kappa(A_i) \leq 20$; Middle row: $K = 100$, $n = 3$, and $1 \leq \kappa(A_i) \leq 20$; Bottom: $K = 30$, $n = 100$, and $1 \leq \kappa(A_i) \leq 20$.