ON A ZERO-FINDING PROBLEM INVOLVING THE MATRIX EXPONENTIAL

MÁTYÁS A. SUSTIK* AND INDERJIT S. DHILLON*

Abstract. An important step in the solution of a matrix nearness problem that arises in certain machine learning applications is finding the zero of $f(\alpha) = \boldsymbol{z}^T \exp(\log X + \alpha \boldsymbol{z}\boldsymbol{z}^T)\boldsymbol{z} - b$. The matrix valued exponential and logarithm in $f(\alpha)$ arises from the use of the von Neumann matrix divergence $\operatorname{tr}(X \log X - X \log Y - X + Y)$ to measure the nearness between the positive definite matrices X and Y. A key step of an iterative algorithm used to solve the underlying matrix nearness problem requires the zero of $f(\alpha)$ to be repeatedly computed. In this paper we propose zero-finding algorithms that gain their advantage by exploiting the special structure of the objective function. We show how to efficiently compute the derivative of f, thereby allowing the use of Newton-type methods. In numerical experiments we establish the advantage of our algorithms.

Key words. matrix exponential, zero-finder, Newton's method, matrix nearness, von Neumann divergence

AMS subject classifications. 46N10, 49M15, 65F60

1. Introduction.

In certain machine learning applications, for instance as in [15, 18, 19], a matrix nearness problem depends on finding the zero of the function

$$f(\alpha) = \boldsymbol{z}^T e^{\log X + \alpha \boldsymbol{z}\boldsymbol{z}^T} \boldsymbol{z} - b \tag{1.1}$$

where the $n \times n$ symmetric positive definite matrix X, vector $\boldsymbol{z}$ and the scalar $b > 0$ are given parameters; the exponentiation and logarithm used are matrix functions. The zero-finding computation arises during the construction of a positive definite matrix that satisfies linear constraints while minimizing a distance measure called the von Neumann matrix divergence [19]. In these machine learning applications, the constraints are extracted from observations, and the constructed positive definite matrix is used to carry out data analysis tasks such as clustering, classification or nearest neighbor search [9, 16]. In another application, one aims to find the nearest correlation matrix (positive semidefinite matrix with diagonal elements equal to one) to a given initial matrix. In [13], the nearness is measured using the Frobenius norm; however, other measures, such as the von Neumann matrix divergence, are also feasible [10].

The underlying matrix nearness problem can be solved by Bregman's iterative algorithm which consists of matrix updates that depend on finding the zero of $f(\alpha)$. In this paper, we present an efficient zero-finding algorithm that exploits the structure of the function. If the cost of evaluating the derivative is similar to the cost of evaluating the function itself, inverse quadratic interpolation (which needs no derivative computations) is expected to be faster than Newton's method, see [17] and [23][p. 55]. In our problem, the evaluation of $f'(\alpha)$, once $f(\alpha)$ has been computed, costs less than the computation of $f(\alpha)$ alone and therefore the cost of the derivative computations is offset by the faster convergence of Newton's method.

The lack of commutativity of matrix multiplication makes the derivative computation non-trivial. Our algorithm operates on the eigendecomposition of the matrix and arranges the computations of $f(\alpha)$ and $f'(\alpha)$ efficiently. We also take advantage of the not widely used improvement to Newton's method described in [17]. In numerical experiments we compare our algorithm to zero-finders which do not need computation of the derivative.

2. Background and Motivation.

2.1. Matrix Divergence.

To measure the nearness between two matrices, we will use a Bregman matrix divergence: $D_\phi(X, Y) = \phi(X) - \phi(Y) - \operatorname{tr}((\nabla\phi(Y))^T(X - Y))$, where ϕ is a strictly convex, real-valued, differentiable function defined on symmetric matrices, and tr denotes the matrix trace. This matrix divergence is a generalization of the *Bregman vector divergence*, see [5] for

*Dept. of Comp. Sci., The University of Texas, Austin, TX 78712 USA, {sustik|inderjit}@cs.utexas.edu.

details. Examples include $\phi(X) = \|X\|_F^2$, which leads to the well-known squared Frobenius norm $\|X - Y\|_F^2$. In this paper, we use a less well-known divergence. If the positive definite matrix X has eigenvalues $\lambda_1, ..., \lambda_n$, then let $\phi(X) = \sum_i (\lambda_i \log \lambda_i - \lambda_i) = \mathrm{tr}(X \log X - X)$, where $\log X$ is the matrix logarithm. The resulting Bregman divergence is

$$D_{vN}(X, Y) = \mathrm{tr}(X \log X - X \log Y - X + Y), \tag{2.1}$$

which generalizes many properties of squared loss and relative entropy, and we call it the von Neumann divergence. It is also known as quantum relative entropy and is used in quantum information theory [22]. See [10] for further details.

2.2. Bregman's Algorithm. We briefly describe the machine learning problem mentioned in the introduction. Given a positive definite matrix X, we attempt to solve the following for $\overline{X}$:

$$\begin{aligned} &\text{minimize } D_{vN}(\overline{X}, X) \\ &\text{subject to } \mathrm{tr}(\overline{X} A_i) \leq b_i,\ i \in \{1, \ldots, c\}, \text{and } \overline{X} \succ 0. \end{aligned} \tag{2.2}$$

The matrices denoted by A_i and the values b_i describe the c linear constraints, where any of the inequalities may occur with equalities instead. Assuming that the feasible set is non-empty, (2.2) is a convex optimization problem with a unique optimum and may be solved by the iterative method of Bregman projections[1] [5, 8]. The idea is to enforce one constraint at a time, while maintaining dual feasibility [19]; see [8] for a proof of convergence. The single constraint problem leads to the following system of equations to be solved for α:

$$\begin{aligned} \nabla\phi(\overline{X}) &= \nabla\phi(X) + \alpha A, \\ \mathrm{tr}(\overline{X} A) &= b. \end{aligned} \tag{2.3}$$

We have dropped the constraint index i for simplicity. Since $\nabla\phi(X) = \log X$, the second equation in (2.3) can be written as $\mathrm{tr}(e^{\log X + \alpha A} A) - b = 0$. In many applications the constraint matrix has rank one [9,16,18], $A = \boldsymbol{z}\boldsymbol{z}^T$, which leads to the zero-finding problem (1.1) by noting the well known $\mathrm{tr}(XY) = \mathrm{tr}(YX)$ identity.

2.3. Derivative of the Matrix Exponential. The formula for the derivative of the matrix exponential is not as simple as that for the exponential function defined on the reals. The difficulty stems from the non-commutativity of matrix multiplication. We start with some basic properties of the matrix derivative and then review the formula for the derivative of the matrix exponential.

We consider smooth matrix functions of one variable denoted by $M(x) : \mathbb{R} \to \mathbb{R}^{n \times n}$; these can also be thought of as $\mathbb{R} \to \mathbb{R}$ functions arranged in an $n \times n$ matrix. The derivative matrix $M'(x)$ is formed by taking the derivatives of the matrix elements. Our first observation is about the trace of the derivative. By definition:

$$\mathrm{tr}(M(x))' = \mathrm{tr}(M'(x)). \tag{2.4}$$

We turn to multiplication next. The lack of commutativity does not yet indicate any difficulties:

$$(M(x)N(x))' = M'(x)N(x) + M(x)N'(x). \tag{2.5}$$

We are seeking $\mathrm{tr}\left(e^{M(\alpha)} A\right)'$ as the function $f(\alpha)$ defined in (1.1) is of this form with $M(\alpha) = \log X + \alpha \boldsymbol{z}\boldsymbol{z}^T$ and $A = \boldsymbol{z}\boldsymbol{z}^T$. But in order to demonstrate the issues caused by non-commutativity we take a short diversion by looking at the slightly simpler example of $\mathrm{tr}(e^M)'$. From here on, when there is no chance of confusion, we may omit the variable from our formulae.

[1] The name recalls the minimization property of orthogonal projections in Euclidean geometry.

We can express the matrix derivative of the kth power as follows: $(M^k)' = \sum_{i=0}^{k-1} M^i M' M^{k-1-i}$. Note that the summation cannot be collapsed when M and M' do not commute. However, if we take the trace on both sides then the summation can be collapsed since $\operatorname{tr}(AB) = \operatorname{tr}(BA)$ and $\operatorname{tr}(\sum_i A_i) = \sum_i \operatorname{tr}(A_i)$ (the latter also holds for infinite sums when one of the sides converges):

$$\operatorname{tr}\left(M^k\right)' = k \operatorname{tr}\left(M^{k-1}M'\right). \tag{2.6}$$

By (2.6) and the power series expansion of the exponential function $\operatorname{tr}(e^M)'$ we get:

$$\operatorname{tr}\left(e^M\right)' = \operatorname{tr}\left(\sum_{k=0}^{\infty} \frac{M^k}{k!}\right)' = \sum_{k=0}^{\infty} \frac{\operatorname{tr}\left(M^k\right)'}{k!} = \sum_{k=1}^{\infty} \frac{\operatorname{tr}\left(M^{k-1}M'\right)}{(k-1)!} = \operatorname{tr}\left(e^M M'\right).$$

The above argument does not imply that the derivative of e^M equals to $e^M M'$ and it also does not readily extend to $\operatorname{tr}\left(e^{M(\alpha)}A\right)'$. In order to tackle this latter expression, we apply (2.4) and (2.5) to get $\operatorname{tr}(e^{M(\alpha)}A)' = \operatorname{tr}((e^{M(\alpha)})'A)$ and then we use the formula for $(e^M)'$ from [24][p. 15, Theorem 5]:

$$(e^M)' = e^M h(\operatorname{ad}_M)M', \tag{2.7}$$

where the *commutator* operator $\operatorname{ad}_A : \mathbb{R}^{n\times n} \to \mathbb{R}^{n\times n}$ satisfies $\operatorname{ad}_A B = AB - BA$, and

$$h(t) = \begin{cases} \frac{1-e^{-t}}{t}, & t \neq 0 \\ 1 & t = 0. \end{cases} \tag{2.8}$$

The analytical function h can be extended to act on linear operators (transformations) via its Taylor series and by the Jordan canonical form; for a detailed treatment we refer the reader to [14][Chapter 1, Definition 1.2][2]. The extension applied to the operator ad_M maps matrices to matrices and appears on the right hand side of (2.7) operating on M'. The Taylor expansion of $h(t)$ around 0 is:

$$h(t) = 1 - \frac{t}{2!} + \frac{t^2}{3!} - \frac{t^3}{4!} + \ldots = \sum_{i=0}^{+\infty} \frac{(-t)^i}{(i+1)!},$$

so one may write (2.7) in a more verbose way as:

$$(e^M)' = e^M \sum_{i=0}^{+\infty} \frac{1}{(i+1)!}(-\operatorname{ad}_M)^i M'.$$

3. Algorithms.

We propose to solve $f(\alpha) = 0$ using Newton's method and the method described by Jarratt in [17]. The latter zero-finder uses a rational interpolating function of the form

$$y = \frac{x-a}{bx^2 + cx + d} \tag{3.1}$$

fitted to the function and derivative values from *two* previous iterations. For completeness, we outline Jarratt's method in Algorithm 1. When the cost of the interpolation itself is negligible, Jarrat's method needs the same computational work as Newton's method, but it yields faster convergence. Despite this fact, this zero-finder has not gained sufficient attention. The *(asymptotic) efficiency index*[3] in the sense of Ostrowski [23][Chapter 3, Section 11] is $\sqrt{1+\sqrt{3}} \approx 1.653$, if we assume

[2]The space of linear transformations over an n-dimensional vector space can be identified with, and therefore is equivalent to the space of $n \times n$ matrices denoted by M_n. A linear operator, like ad, that acts on M_n can be represented by an $n^2 \times n^2$ matrix, because the underlying linear space, M_n has dimension n^2.

[3]A similar concept is the *order of convergence per function evaluation*.

Algorithm 1: Zero-finding based on P. Jarratt's method, see [17].

Input : Subroutines to evaluate f and f', initial guess α_0.
Output: Sequence of approximation to the solution of $f(\alpha) = 0$.

```
Compute f_0 = f(α_0), f'_0 = f'(α_0).
α_1 = α_0 − f_0/f'_0. (Initial Newton step.)
for i = 2, 3, ... do
    Compute f_{i−1} = f(α_{i−1}) and f'_{i−1} = f'(α_{i−1}).
    Set α_i = α_{i−1} − (α_{i−1} − α_{i−2}) f_{i−1} [f_{i−2}(f_{i−1} − f_{i−2}) − (α_{i−1} − α_{i−2}) f_{i−1} f'_{i−2}] / (2 f_{i−1} f_{i−2}(f_{i−1} − f_{i−2}) − (α_{i−1} − α_{i−2})(f_{i−1}^2 f'_{i−2} + f_{i−2}^2 f'_{i−1})).
end
```

$$\alpha_i = \alpha_{i-1} - \frac{(\alpha_{i-1}-\alpha_{i-2})f_{i-1}[f_{i-2}(f_{i-1}-f_{i-2}) - (\alpha_{i-1}-\alpha_{i-2})f_{i-1}f'_{i-2}]}{2f_{i-1}f_{i-2}(f_{i-1}-f_{i-2}) - (\alpha_{i-1}-\alpha_{i-2})(f_{i-1}^2 f'_{i-2} + f_{i-2}^2 f'_{i-1})}.$$

that the computational cost to evaluate $f(\alpha)$ and $f'(\alpha)$ are the same. The efficiency index for Newton's method under the same assumption is only $\sqrt{2} \approx 1.414$. In comparison, inverse quadratic interpolation, which is the workhorse of Brent's method [6] requires no derivative computations and has asymptotic efficiency index of 1.839. Newton's and Jarratt's method can perform better when the derivative computation costs less than the function evaluation and this is often the case when the objective function is built from exp, sin, cos, see also [17]. In such circumstances, the efficiency index for Newton's and Jarratt's methods may approach the order of convergence, 2 and $1 + \sqrt{3} \approx 2.732$ respectively.

We show how to efficiently carry out and arrange the computations of $f(\alpha)$ and $f'(\alpha)$ in Section 3.1. An additional improvement exploiting the shape of the objective function is discussed in Section 3.2. We end this section by a lemma that establishes that f is strictly monotone, which implies that $f(\alpha) = 0$ has a unique solution. The proof is very similar to Lemma 7 of [1]; the fact that $\boldsymbol{z}\boldsymbol{z}^T$ has rank one allows some simplifications. We also establish convexity.

LEMMA 3.1. *If M is symmetric and $\boldsymbol{z} \neq 0$ then $f(\alpha) + b = \boldsymbol{z}^T e^{M+\alpha\boldsymbol{z}\boldsymbol{z}^T}\boldsymbol{z}$ is strictly monotone increasing and strictly convex.*

Proof. First, we note that it is sufficient to show that the first and second derivatives are positive at any given α_0. Consider the function $\overline{f}(\alpha) = f(\alpha+\alpha_0)$, a shift by α_0. Since $\overline{f}(\alpha) = \boldsymbol{z}^T e^{\overline{M}+\alpha\boldsymbol{z}\boldsymbol{z}^T}\boldsymbol{z} - b$ where $\overline{M} = M + \alpha_0\boldsymbol{z}\boldsymbol{z}^T$ is also a symmetric matrix, we can conclude that it is sufficient to prove that the first and second derivatives are positive at $\alpha = 0$.

Second, we show that we can assume that M is positive definite. Otherwise, pick a β that is large enough so that $\widehat{M} = M + \beta I$ is positive definite. Since $e^{M+\alpha\boldsymbol{z}\boldsymbol{z}^T} = e^{-\beta}e^{\widehat{M}+\alpha\boldsymbol{z}\boldsymbol{z}^T}$, we conclude that the sign of the derivatives is the same for $\widehat{M}$ and M.

In order to establish the claim in the case of a positive definite M and $\alpha = 0$, we inspect the coefficients in the power series expansion of $\boldsymbol{z}^T e^{M+\alpha\boldsymbol{z}\boldsymbol{z}^T}\boldsymbol{z}$ around zero. We note that f is analytical, which can be seen by bounding the terms of the expansion. According to the power series expansion of exp we have:

$$\boldsymbol{z}^T e^{M+\alpha\boldsymbol{z}\boldsymbol{z}^T}\boldsymbol{z} = \boldsymbol{z}^T \sum_{k=0}^{\infty} \frac{(M+\alpha\boldsymbol{z}\boldsymbol{z}^T)^k}{k!}\boldsymbol{z} = \sum_{k=0}^{\infty}\frac{\boldsymbol{z}^T M^k \boldsymbol{z}}{k!} + \alpha\sum_{k=1}^{\infty}\frac{1}{k!}\sum_{i=0}^{k-1}\boldsymbol{z}^T M^i \boldsymbol{z}\boldsymbol{z}^T M^{k-1-i}\boldsymbol{z}$$
$$+ \alpha^2 \sum_{k=2}^{\infty}\frac{1}{k!}\sum_{i+j\le k-2}\boldsymbol{z}^T M^i\boldsymbol{z}\boldsymbol{z}^T M^j \boldsymbol{z}\boldsymbol{z}^T M^{k-2-i-j}\boldsymbol{z} + \dots$$

For a positive definite M and integer i we have $\boldsymbol{z}^T M^i \boldsymbol{z} > 0$, implying that the coefficient of α^l is positive for all $l \geq 0$. □

3.1. Evaluation of f and its derivative. We now show that $f(\alpha)$ can be computed at a cost of $2n^2 + O(n)$ floating point operations (flops), in addition to the flops that are needed to compute the eigendecomposition of a diagonal plus rank-one matrix. This eigendecomposition is expected to dominate the total cost of the function evaluation. In order to compute $f'(\alpha)$ as well, we need $3n^2 + O(n)$ additional flops. We note that n floating point exponentiations (which are significantly more costly than additions and multiplications) are also necessary to get $f(\alpha)$, however the computational cost is still dominated by the $O(n^2)$ additions/multiplications. No additional floating point exponentiations are needed to compute $f'(\alpha)$.

We assume that we maintain the eigendecomposition of each iterate of Bregman's algorithm as is done in the machine learning application, see [19]. We do not count the initial cost of computing this eigendecomposition. In some applications the factors form the input to the whole procedure and the updated factors are the output. Even if the factors have to be produced, or the matrix assembled upon return, these steps need to be carried out only once and the cost is amortized over the iterative steps of Bregman's algorithm.

In the presence of the eigendecomposition $X = V\Lambda V^T$, we can express $f(\alpha)$ as follows:

$$f(\alpha) = \boldsymbol{z}^T e^{\log(V\Lambda V^T)+\alpha \boldsymbol{z}\boldsymbol{z}^T} \boldsymbol{z} - b = \boldsymbol{z}^T V e^{\log\Lambda + \alpha V^T \boldsymbol{z}\boldsymbol{z}^T V} V^T \boldsymbol{z} - b = \boldsymbol{v}^T e^{\log\Lambda + \alpha \boldsymbol{v}\boldsymbol{v}^T} \boldsymbol{v} - b, \quad (3.2)$$

where $\boldsymbol{v} = V^T \boldsymbol{z}$. We begin the evaluation by solving a diagonal plus rank-one eigendecomposition

$$\log\Lambda + \alpha \boldsymbol{v}\boldsymbol{v}^T = U\Theta U^T, \ \Theta = \operatorname{diag}(\theta) \quad (3.3)$$

which can be done in $O(n^2)$ time [12]. Next, we form $\boldsymbol{u} = U^T \boldsymbol{v}$ and get:

$$f(\alpha) = \boldsymbol{v}^T e^{U\Theta U^T} \boldsymbol{v} - b = \boldsymbol{v}^T U e^{\Theta} U^T \boldsymbol{v} - b = \boldsymbol{u}^T e^{\Theta} \boldsymbol{u} - b = (\boldsymbol{u} \circ e^{\theta})^T \boldsymbol{u} - b, \quad (3.4)$$

where $\circ$ denotes the Hadamard product. We move on to the efficient computation of $f'(\alpha)$. The expression in (3.2) can be written in the form $\operatorname{tr}(e^{M(\alpha)}A)$ with $A = \boldsymbol{v}\boldsymbol{v}^T$ and $M(\alpha) = \log\Lambda + \alpha\boldsymbol{v}\boldsymbol{v}^T$. According to (2.4), (2.5) and (2.7) the derivative at α equals:

$$f'(\alpha) = \operatorname{tr}\left((e^{M(\alpha)})' A\right) = \operatorname{tr}\left(e^{\log\Lambda + \alpha \boldsymbol{v}\boldsymbol{v}^T} \cdot \left(h\left(\operatorname{ad}_{\log\Lambda + \alpha\boldsymbol{v}\boldsymbol{v}^T}\right)\boldsymbol{v}\boldsymbol{v}^T\right) \cdot \boldsymbol{v}\boldsymbol{v}^T\right). \quad (3.5)$$

In order to compute the expression $h\left(\operatorname{ad}_{\log\Lambda+\alpha\boldsymbol{v}\boldsymbol{v}^T}\right)\boldsymbol{v}\boldsymbol{v}^T$, we reduce the problem to the diagonal case and then use the spectral decomposition of the operator in question.

LEMMA 3.2. *Let $U \in \mathbb{R}^{n\times n}$ orthogonal and let Θ and B be arbitrary matrices. Then the following holds:*

$$\operatorname{ad}_{U\Theta U^T} B = U \operatorname{ad}_{\Theta}(U^T B U) U^T.$$

Proof. By the definition of the ad operator and $UU^T = I$, the right hand side above may be rewritten as:

$$U(\Theta U^T B U - U^T B U \Theta) U^T = U\Theta U^T B - BU\Theta U^T = \operatorname{ad}_{U\Theta U^T} B. \qquad \square$$

An analytical function can be extended to the operator space using the Jordan canonical form [14] (Chapter 1, Definition 1.2). Lemma 3.3 below generalizes the above result to analytical functions of the operator $\operatorname{ad}_{\Theta}$:

LEMMA 3.3. *Let U, Θ and B be as in Lemma 3.2 and let g be analytical. The following holds:*

$$g(\operatorname{ad}_{U\Theta U^T})B = U g(\operatorname{ad}_{\Theta})(U^T B U) U^T.$$

Proof. Since g is analytical, it is sufficient to show that for any nonnegative integer k:

$$\operatorname{ad}^k_{U\Theta U^T} B = U \operatorname{ad}^k_{\Theta}(U^T B U) U^T.$$

For $k = 0$ the statement is immediate and we proceed by induction on k. Assume that the statement holds for $k \geq 1$, then apply Lemma 3.2 and the definition of ad to finish the proof:

$$\begin{aligned}\operatorname{ad}^k_{U\Theta U^T} B &= \operatorname{ad}_{U\Theta U^T}(\operatorname{ad}^{k-1}_{U\Theta U^T} B) = \operatorname{ad}_{U\Theta U^T}(U \operatorname{ad}^{k-1}_\Theta (U^T B U)U^T) \\ &= U \operatorname{ad}_\Theta (U^T U \operatorname{ad}^{k-1}_\Theta (U^T B U) U^T U) U^T = U \operatorname{ad}^k_\Theta (U^T B U) U^T. \qquad \square\end{aligned}$$

Our next step is to calculate $g(\operatorname{ad}_\Theta)$ using the spectral theorem. By the definition of the adjoint, one can easily show that if X is symmetric, then ad_X is self-adjoint, and so in our case we can use the eigendecomposition of ad_Θ to calculate $g(\operatorname{ad}_\Theta)$. The following argument mimics Lemma 8 of [24, Chapter 1], which gives the eigenvectors of ad_X; here we only need to deal with diagonal matrices. The definition of ad and the elementary calculation

$$\operatorname{ad}_\Theta \boldsymbol{e}_i \boldsymbol{e}_j^T = \Theta \boldsymbol{e}_i \boldsymbol{e}_j^T - \boldsymbol{e}_i \boldsymbol{e}_j^T \Theta = (\theta_i - \theta_j) \boldsymbol{e}_i \boldsymbol{e}_j^T \tag{3.6}$$

shows that the eigenvectors of ad_Θ are the n^2 matrices of the form $\boldsymbol{e}_i \boldsymbol{e}_j^T$ with eigenvalues $\theta_i - \theta_j$ respectively, where $\Theta = \operatorname{diag}(\theta)$.

LEMMA 3.4. *Let $\Theta = \operatorname{diag}(\theta)$ be diagonal, and g analytical. For any B we have:*

$$g(\operatorname{ad}_\Theta) B = \sum_{i,j} g(\theta_i - \theta_j)(\boldsymbol{e}_i^T B \boldsymbol{e}_j) \boldsymbol{e}_i \boldsymbol{e}_j^T. \tag{3.7}$$

Proof. Repeated application of (3.6) establishes that for any nonnegative integer k:

$$\operatorname{ad}^k_\Theta B = \operatorname{ad}^k_\Theta \sum_{ij} (\boldsymbol{e}_i^T B \boldsymbol{e}_j) \boldsymbol{e}_i \boldsymbol{e}_j^T = \sum_{ij} (\theta_i - \theta_j)^k (\boldsymbol{e}_i^T B \boldsymbol{e}_j) \boldsymbol{e}_i \boldsymbol{e}_j^T.$$

The proof is completed by appealing to the analytical property of g. $\square$

Note that the right hand side of (3.7) can be expressed as the Hadamard product of B and the matrix which has its (i, j) element equal to $g(\theta_i - \theta_j)$. According to Lemma 3.3 and equation (3.3), we have for any analytical g,

$$g\left(\operatorname{ad}_{\log \Lambda + \alpha \boldsymbol{v}\boldsymbol{v}^T}\right) \boldsymbol{v}\boldsymbol{v}^T = g(\operatorname{ad}_{U\Theta U^T}) \boldsymbol{v}\boldsymbol{v}^T = U g(\operatorname{ad}_\Theta)(U^T \boldsymbol{v}\boldsymbol{v}^T U) U^T.$$

Recall from Section 3.1 that we introduced $\boldsymbol{u} = U^T \boldsymbol{v}$ and that $\Theta = \operatorname{diag}(\theta)$. Now we define matrix H to have (i, j) element equal to $h(\theta_i - \theta_j)$, where h is as in (2.8) and so finally from (3.5) and Lemma 3.4 we have:

$$f'(\alpha) = \boldsymbol{v}^T e^{U\Theta U^T} U (H \circ \boldsymbol{u}\boldsymbol{u}^T) U^T \boldsymbol{v} = \boldsymbol{v}^T U e^\Theta U^T U (H \circ \boldsymbol{u}\boldsymbol{u}^T) \boldsymbol{u} = (\boldsymbol{u} \circ e^\theta)^T (H \circ \boldsymbol{u}\boldsymbol{u}^T) \boldsymbol{u}. \tag{3.8}$$

An alternative derivation for $f'(\alpha)$ based on the Daleckii–Krein theorem is also possible, see [3][p. 60, p.154].

Note that the computation of the eigenvalues and the vector $\boldsymbol{u}$ is also part of the computations needed to evaluate f at α, see (3.4). Therefore no additional eigendecompositions are necessary to compute the derivative. The direct computation of elements of the matrix H would require n^2 floating point exponentiations. Fortunately, we do not need to compute H explicitly, but instead we may expand the right hand side of (3.8) to get:

$$f'(\alpha) = \sum_{i,j=1}^n u_i^2 u_j^2 e^{\theta_i} h(\theta_i - \theta_j) = 2 \sum_{\substack{1 \le i < j \le n \\ \theta_i \ne \theta_j}} u_i^2 u_j^2 \frac{e^{\theta_i} - e^{\theta_j}}{\theta_i - \theta_j} + 2 \sum_{\substack{1 \le i < j \le n \\ \theta_i = \theta_j}} u_i^2 u_j^2 e^{\theta_i} + \sum_{i=1}^n u_i^4 e^{\theta_i}. \tag{3.9}$$

The above form exploits symmetry and allows the reuse of the e^{θ_i} terms available from the computation of $f(\alpha)$. We need $2.5n^2$ floating point additions, subtractions and multiplications and $0.5n^2$

Algorithm 2: Computations needed to evaluate f and f'.

		$\overset{*}{\pm}$	/	exp
	Input : Matrix X with its $V\Lambda V^T$ eigendecomposition; vector $\boldsymbol{z}$; scalar α.			
	Output: $f(\alpha)$, $f'(\alpha)$, see (1.1).			
1	$\boldsymbol{v} = V^T\boldsymbol{z}$	$2n^2$		
2	Factor $\log\Lambda + \alpha\boldsymbol{v}\boldsymbol{v}^T = U\,\mathrm{diag}(\theta)U^T$	ℓn^2		
3	$\boldsymbol{u} = U^T\boldsymbol{v}$	$2n^2$		
4	$\boldsymbol{x} = \boldsymbol{u} \circ e^{\theta}$	n		n
5	$f(\alpha) = \boldsymbol{x}^T\boldsymbol{u} - b$	$2n$		
6	$f'(\alpha) = \boldsymbol{x}^T(H \circ \boldsymbol{u}\boldsymbol{u}^T)\boldsymbol{u}$ see (3.8), (3.9)	$2.5n^2$	$0.5n^2$	

floating point divisions. The cost of floating point divisions on modern architectures is between 2.5 to 3 times that of floating point addition. It is important to note that exponentiation is a much more expensive operation; its cost is about ten times that of a division. We summarize the computational steps required to compute $f(\alpha)$ and $f'(\alpha)$ in Algorithm 2.

The repeated computation of $f(\alpha)$ takes $(2+\ell)n^2 + O(n)$ floating point operations (flops) where $\ell n^2 + O(n)$ flops are needed for the eigendecomposition of a diagonal plus rank-one matrix[4]. Note that only steps 2 to 6 in Algorithm 2 have to be done repeatedly while finding the zero, so we did not include the matrix-vector multiplication in step 1 in the flop count for computing $f(\alpha)$. When we are computing $f'(\alpha)$, we are reusing intermediate results from the computation of $f(\alpha)$ and therefore we need only about $2.5n^2$ additional floating point additions/multiplications and $0.5n^2$ divisions. We expect the total computational cost to be dominated by the eigendecomposition.

The above discussion of the operation counts did not consider the issue of numerical accuracy. The difference quotient term of $(e^{\theta_i} - e^{\theta_j})/(\theta_i - \theta_j)$ in (3.9) may suffer from catastrophic cancellation when θ_i and θ_j are not well separated. Our solution is to use an alternate formula when $x = (\theta_i - \theta_j)/2$ is sufficiently small:

$$\frac{e^{\theta_i} - e^{\theta_j}}{\theta_i - \theta_j} = e^{\theta_i/2}e^{\theta_j/2}\frac{\sinh(x)}{x} = e^{\theta_i/2}e^{\theta_j/2}\left(1 + \frac{x^2}{3!} + \frac{x^4}{5!} + \frac{x^6}{7!} + R(x)\right).$$

As indicated by the above equation we approximate $\sinh x$ using its Taylor expansion, which converges rapidly for small x. The native floating point instruction computing sinh produces accurate results, but if it were used for all (θ_i, θ_j) pairs, then we would pay a substantial performance penalty[5]. When $|x| \geq 0.1$, we use the original form that appears in (3.9), otherwise we use the above Taylor approximation. Elementary calculations using the Lagrange form of the remainder reveal that $|R(x)|$ is less than the machine epsilon when $|x| < 0.1$. Our implementation uses six floating point multiplications and three additions and no divisions[6] which should be compared to the two subtractions and a division in the original difference quotient formula. We observed no adverse effect on performance.

3.2. Logarithmic prescaling. All the zero-finding algorithms discussed use interpolation to fit simple functions to find the next approximation. Newton's method as well as the secant method use straight lines, the inverse quadratic interpolation method uses the inverse of a quadratic function, as its name suggests, and Jarrat's method uses a function of the form given by (3.1).

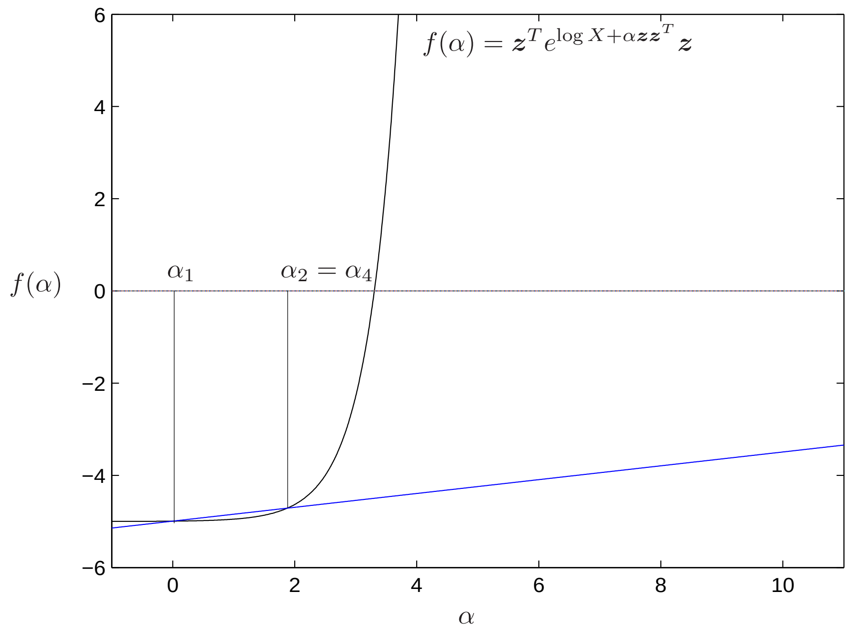
When the graph of the objective function has a known specific shape, it may be advantageous, or even necessary, to fit a different function. We note that convexity of f, established in Lemma 3.1, implies convergence for the secant method, regardless of initial guesses. However, in floating point arithmetic, the presence of overflow, underflow and rounding error may result in lack of convergence. Figure 3.1 depicts the situation where the secant method does not make progress: the function value

[4] We observed the value of ℓ to typically fall between 25 and 50.

[5] We found that the computation of sinh using a floating point instruction is 35 times longer than a multiplication.

[6] Constant divisions are turned into multiplications.

FIG. 3.1. *Overflow and underflow resulting in $\alpha_2 = \alpha_4$ (no progress) for the secant method.*



at α_3 is so large (not shown on the figure) that the computation of α_4 suffers from underflow, resulting in $\alpha_4 = \alpha_2$. This issue affects the other three zero-finding algorithms as well.

A similar problem occurs during the solution of the secular equation used to compute the eigendecomposition of a rank-one update to a diagonal matrix. The solution there is to fit a rational function which has the same asymptotes as the objective function [7, 20]. In our case, better convergence can be attained by fitting parameterized exponential functions. Doing so helps with the overflow/underflow problem depicted in Figure 3.1 and speeds up convergence.

We implement this idea of fitting a nonlinear function using a slightly different approach than what is found in [7, 20]. The main advantage of our solution is that we do not need to derive the (parameterized) fitting function, making it is easier to apply when a function such as (3.1) is used for interpolation. We apply a transformation to the function $f(\alpha)$ that yields a transformed function, $g(\alpha)$, and we use the zero-finders on $g(\alpha)$ in their original form. Our transformation applies a logarithmic prescaling; we introduce:

$$g(\alpha) = \log(f(\alpha) + b) - \log b \tag{3.10}$$

and observe that g is monotone and has the same zero as f. For the Newton-type methods we also need the derivative: $g'(\alpha) = f'(\alpha)/(f(\alpha) + b)$. Note that the additional computations are negligible.

4. Experimental results. We compare Newton's method and Jarrat's method, both of which employ the use of derivatives, to the secant method and inverse quadratic interpolation which are zero-finding algorithms that do not require calculation of the derivative.

We implemented the algorithms in C++ as MATLAB [21] and OCTAVE [11] compatible MEX files which call the Fortran DLAED4 function from LAPACK [2] for the diagonal plus rank-one eigendecompositions. We implemented the correction for accurate eigenvectors according to [12], and also implemented deflation in C++; we utilized fast linear algebra routines from BLAS [4]. In all algorithm versions we accepted an approximation as the zero when the function value was not larger

TABLE 4.1
Running times and number of (rank-one update to a diagonal matrix) eigendecompositions executed by the various algorithms when solving the protein data classification problem using 1000 *constraints. The middle column indicates the relative performance when compared to the secant method applied to f. Function g is defined by* (3.10).

Protein data classification

Applied to	Method	run-time (sec)	ratio of run-time compared to secant on f	number of eigendecomp.
f	secant	7.49	1.00	43,781
	inv. quad. int.	6.82	0.91	39,733
	Newton	5.63	0.75	30,148
	Jarratt	4.73	0.63	24,994
g	secant	6.62	0.88	38,380
	inv. quad. int.	6.20	0.83	35,941
	Newton	4.86	0.65	25,557
	Jarratt	4.49	0.60	23,523

then $n \cdot eps$ for an $n \times n$ matrix. We tested the performance of the algorithms in three sets of experiments. We revisited the protein data experiment (GYRB) from [18, 19]; we carried out a "synthetic" correlation matrix experiment motivated by [13]; and in the third experiment we find the zero of a slightly modified version of (1.1) as a result of the use of the so called "slack variables" in the hand written digits recognition (MNIST) experiment in [19].

We compare running times and the number of eigendecompositions (the most expensive step) executed by the zero-finding methods. We used a computer with an INTEL X3460 CPU running at 2.8GHz utilizing 8MB of cache. We ran the algorithms in single threaded mode (including the BLAS and LAPACK subroutines) with no other programs running.

The first experiment reproduces a result from [18, 19], where the objective is to find a 52×52 kernel matrix for protein data classification. The task is formulated as a matrix nearness problem using the von Neumann matrix divergence, $D_{vN}(X, Y) = \operatorname{tr}(X \log X - X \log Y - X + Y)$, as the nearness measure. We extract 1000 linear inequality constraints from the training data and use Bregman's iterative process starting from the identity matrix; for additional details we refer the reader to [18, 19]. Table 4.1 presents running times of the different zero-finders and the number of eigendecompositions needed. The methods using derivatives are seen to have better performance due to fewer eigendecompositions.

In the second experiment the objective is to find the nearest correlation matrix X to a given positive definite starting matrix Y:

$$\text{minimize } D_{vN}(X, Y), \text{ subject to } X_{ii} = 1, \ i \in \{1, \dots, n\}, \ X \succ 0.$$

We generated Y to be a random symmetric matrix with eigenvalues uniformly distributed in $(0, 1)$. The results in Table 4.2 are averaged from ten runs using 500×500 randomly generated matrices. We observe again that the use of the derivative improves performance when compared to non-derivative based zero-finding methods.

In the third experiment we executed Bregman's algorithm using the MNIST data set consisting of images of handwritten digits encoded as 164-dimensional vectors. For details on this experiment we refer the reader to [19]. The zero-finding problem is a slightly modified version of (1.1) due to the use of slack variables. Here, we only give a short summary. Instead of enforcing the constraints, we penalize deviation from the desired conditions using the relative entropy $KL(\boldsymbol{x}, \boldsymbol{y}) = \sum_i (x_i \log(x_i/y_i) - x_i + y_i)$, the vector divergence from which the von Neumann matrix divergence is generalized:

$$\text{minimize}_{X, \boldsymbol{b}} \ D_{vN}(X, Y) + \gamma KL(\boldsymbol{b}, \boldsymbol{b}_0), \text{ subject to } \ \operatorname{tr}(X A_i) \le \boldsymbol{e}_i^T \boldsymbol{b}, \ i \in \{1, \dots, c\}, \ X \succ 0.$$

TABLE 4.2
Running times and number of (rank-one update to a diagonal matrix) eigendecompositions executed by the various algorithms when solving the correlation matrix problem. The middle column indicates the relative performance when comparing to the secant method applied to f. Function g is defined by (3.10).

Nearest correlation matrix

Applied to	Method	run-time (sec)	ratio of run-time compared to secant on f	number of eigendecomp.
f	secant	201.3	1.00	9,255
	inv. quad. int.	190.1	0.94	8,568
	Newton	172.3	0.86	6,824
	Jarratt	145.5	0.72	5,321
g	secant	182.0	0.90	8,082
	inv. quad. int.	169.9	0.84	7,371
	Newton	141.8	0.70	5,094
	Jarratt	136.6	0.68	4,741

TABLE 4.3
Running times and number of (rank-one update to a diagonal matrix) eigendecompositions executed by the algorithms when solving the MNIST *handwritten digits recognition problem. The algorithms were applied to function g as defined by* (3.10).

MNIST handwritten digits recognition

Method	run-time (sec)	number of rank-one eigendecompositions
secant	281.7	444,385
inv. quad. int.	274.7	432,411
Newton	175.0	241,637
Jarratt	175.0	241,641

The objective function measures the distance from the starting matrix Y as well as the amount by which the constraints are relaxed. The $\gamma > 0$ parameter controls how much "slack" we permit; in essence it is used to find the balance between over- and under-constraining the optimization problem.

The resulting zero-finding problem is a slightly modified version of (1.1):

$$\boldsymbol{z}^T e^{\log X + \alpha \boldsymbol{z}\boldsymbol{z}^T} \boldsymbol{z} + e^{\alpha/\gamma} - b = 0.$$

The derivative computation and other discussions of Section 3 apply after minor modifications.

In Table 4.3 we present the MNIST handwritten digits recognition experiment results for four zero-finding methods. We only show the versions using the logarithmic prescaling, because without that improvement the algorithms greatly suffer from the overflow/underflow problem discussed in Section 3.2, which would force the use of the bisection (or some other, but still inefficient) method for many iterations. Due to the modified objective function, for which the logarithmic prescaling works very well, the number of iterations executed by the zero-finders is quite low (never more than four for Newton and Jarrat's method). The inverse quadratic interpolation provides its first approximation only in the fourth iteration and Jarratt's method in the third. Simply put, the faster convergence has no time to set in for inverse quadratic interpolation and Jarrat's method. As a result, the quadratic interpolation method yields only a slight benefit over the secant method and Jarratt's method does not yield any improvement over Newton's method. Newton's method requires nearly half the number of eigendecompositions when compared to inverse quadratic interpolation, while the running time improvement is 36%.

5. Conclusions. In this paper, we discussed a specific zero-finding problem that arises in certain machine learning applications. We have shown how to efficiently calculate the derivative of the objective function which involves the matrix exponential; a task that is non-trivial due to the lack of commutativity of matrix multiplication. The efficient computation of the derivative and the reuse of computations from the function evaluation allowed us to apply Newton's method and a relatively unknown zero-finder variant due to P. Jarratt. The presented experimental results confirmed our expectation of better performance when compared to zero-finding methods that do not employ the derivative.

6. Acknowledgements. We thank the anonymous reviewers for their thoughtful comments. This research was supported by NSF grants CCF–0431257 and CCF–0728879.

REFERENCES

[1] C.-G. Ambrozie, *Finding positive matrices subject to linear restrictions*, Linear Algebra and its Applications, 426 (2007), pp. 716–728.
[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, third ed., 1999.
[3] R. Bhatia, *Positive Definite Matrices*, Princeton, 2006.
[4] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley, *An updated set of basic linear algebra subprograms (blas)*, ACM Trans. Math. Soft., 28 (2002), pp. 135–151.
[5] L. Bregman, *The relaxation method of finding the common point of convex sets and its application to the solution of problems in convex programming*, USSR Comp. Mathematics and Mathematical Physics, 7 (1967), pp. 200–217.
[6] R. P. Brent, *Algorithms for Minimization without Derivatives*, Prentice-Hall, 1973.
[7] J. R. Bunch, C. P. Nielsen, and D. C. Sorensen, *Rank-one modification of the symmetric eigenproblem*, Numerical Mathematics, 31 (1978), pp. 31–48.
[8] Y. Censor and S. Zenios, *Parallel Optimization*, Oxford University Press, 1997.
[9] J. Davis, B. Kulis, P. Jain, S. Sra, and I. S. Dhillon, *Information-theoretic metric learning*, in Proc. 24th International Conference on Machine Learning, 2007.
[10] I. S. Dhillon and J. Tropp, *Matrix nearness problems using Bregman divergences*, SIAM Journal on Matrix Analysis and Applications, (2007).
[11] J. W. Eaton, *GNU Octave Manual*, Network Theory Limited, 2002.
[12] M. Gu and S. C. Eisenstat, *A stable and efficient algorithm for the rank-one modification of the symmetric eigenproblem*, SIAM Journal on Matrix Analysis and Applications, 15 (1994), pp. 1266–1276.
[13] N. J. Higham, *Computing the nearest correlation matrix*, IMA Journal of Numerical Analysis, 22 (2002), pp. 329–343.
[14] N. J. Higham, *Functions of Matrices: Theory and Computation*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008.
[15] P. Jain, B. Kulis, J. Davis, and I. S. Dhillon, *Metric and kernel learning using a linear transformation*, Journal of Machine Learning Research(JMLR), (2012). to appear.
[16] P. Jain, B. Kulis, and K. Grauman, *Fast image search for learned metrics*, in Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2008.
[17] P. Jarratt, *A rational iteration function for solving equations*, The Computer Journal, 9 (1966), pp. 304–307.
[18] B. Kulis, M. A. Sustik, and I. S. Dhillon, *Learning low-rank kernel matrices*, in Proc. 23rd International Conference on Machine Learning (ICML), 2006.
[19] B. Kulis, M. A. Sustik, and I. S. Dhillon, *Low-rank kernel learning with Bregman matrix divergences*, Journal of Machine Learning Research, 10 (2009), pp. 341–376.
[20] R.-C. Li, *Solving secular equations stably and efficiently*, Tech. Rep. UCB/CSD-94-851, EECS Department, University of California, Berkeley, Dec 1994.
[21] MATLAB, *version 7.12.0.635 (R2011a)*, The MathWorks Inc., Natick, Massachusetts, 2011.
[22] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press, 2000.
[23] A. M. Ostrowski, *Solution of equations in Euclidean and Banach spaces*, Academic Press, 1973.
[24] W. Rossman, *Lie Groups: an Introduction Through Linear Groups*, Oxford University Press, 2002.