SIAM J. MATRIX ANAL. APPL.
Vol. 44, No. 1, pp. 156-174

COMPUTING THE SQUARE ROOT OF A LOW-RANK PERTURBATION OF THE SCALED IDENTITY MATRIX*

MASSIMILIANO FASI†, NICHOLAS J. HIGHAM‡, AND XIAOBO LIU‡

Abstract. We consider the problem of computing the square root of a perturbation of the scaled identity matrix, $A = \alpha I_n + UV^*$, where U and V are $n \times k$ matrices with $k \leq n$. This problem arises in various applications, including computer vision and optimization methods for machine learning. We derive a new formula for the pth root of A that involves a weighted sum of powers of the pth root of the $k \times k$ matrix $\alpha I_k + V^*U$. This formula is particularly attractive for the square root, since the sum has just one term when $p = 2$. We also derive a new class of Newton iterations for computing the square root that exploit the low-rank structure. We test these new methods on random matrices and on positive definite matrices arising in applications. Numerical experiments show that the new approaches can yield a much smaller residual than existing alternatives and can be significantly faster when the perturbation UV^* has low rank.

Key words. matrix pth root, matrix square root, low-rank update, matrix iteration, Newton iteration, MATLAB

MSC codes. 15A16, 65F60, 65F99

DOI. 10.1137/22M1471559

1. Introduction.

Any solution of the nonlinear equation $X^p = A$ is a pth root of the square matrix A. This matrix equation arises in many applications [19, sect. 2.14], and various methods for solving it numerically have been proposed in the literature. Particular attention has been devoted to the principal pth root $A^{1/p}$, which for a matrix with no eigenvalues on the closed negative real axis $\mathbb{R}^-$ is the unique pth root whose eigenvalues λ all satisfy $|\arg \lambda| < \pi/p$. For $p = 2$ one obtains the square root, which is the pth root most often needed in applications and most thoroughly investigated in the literature. Throughout this work, "pth root" refers to the principal pth root, and in particular "square root" refers to the principal square root, whose eigenvalues all lie in the open right half-plane.

The state-of-the-art methods for computing the matrix square root are based on the Schur decomposition [6], [8], [18], and can be extended to the computation of the pth root [12], [22], [31]. These methods have excellent numerical stability, in the sense that the computed solution satisfies essentially the same backward error bound as the rounded exact solution.

The Schur decomposition is typically computed using the QR algorithm [34], [35], [36], which is one of the most complex methods in matrix computation [11, sect. 7.5]. Implementing it in a robust and efficient way is a difficult task, so its low prevalence in libraries for matrix computations on custom hardware is not surprising. For ex-

* Received by the editors January 14, 2022; accepted for publication (in revised form) by P. Boito August 18, 2022; published electronically February 27, 2023.
https://doi.org/10.1137/22M1471559
Funding: The work of the first author was supported by the Wenner-Gren Foundations grant UPD2019-0067. The work of the second author was supported by the Engineering and Physical Sciences Research Council grant EP/P020720/1 and the Royal Society.

† Department of Computer Science, Durham University, Upper Mountjoy Campus, Stockton Road, Durham, DH1 3LE, UK (massimiliano.fasi@durham.ac.uk).
‡ Department of Mathematics, University of Manchester, Oxford Road, Manchester, M13 9PL, UK (nick.higham@manchester.ac.uk, xiaobo.liu@manchester.ac.uk).

ample, a nonsymmetric dense eigensolver is not present in the NVIDIA cuSOLVER library.[1] Multiprecision environments often do not supply a routine for computing the Schur decomposition [10, sect. 5]; examples lacking one are the Julia language [5], which currently (version 1.7.1) does not provide it for its BigFloat data type, and the MATLAB Symbolic Math Toolbox [33], where it is not available for the `sym` data type at the time of writing (version 9).

Matrix iterations for computing $A^{1/2}$ are an attractive alternative in these situations. Newton iterations converge quadratically in exact arithmetic and require only matrix multiplication and (in most cases) matrix inversion or the solution of multiple right-hand side linear systems. In deep learning, for example, the Newton–Schulz iteration [19, p. 153], [27] is widely used as an alternative to diagonalization when A is positive semidefinite. Being rich in matrix multiplication, it offers better performance on modern GPUs in a variety of computer vision tasks [32]. Tailored iterations are available for computing the square root of matrices with special structure or properties, including M-matrices, H-matrices, and Hermitian positive definite matrices; these are surveyed in [19, sect. 6.8].

Here we study methods for computing the square root of a matrix $A \in \mathbb{C}^{n\times n}$ of the form

$$A = \alpha I_n + UV^*, \quad \alpha \in \mathbb{C}, \quad U, V \in \mathbb{C}^{n\times k}, \quad k \le n, \quad \Lambda(A) \cap \mathbb{R}^- = \emptyset, \tag{1.1}$$

where I_n is the identity matrix of order n and $\Lambda(A)$ denotes the spectrum of A. The condition $\Lambda(A) \cap \mathbb{R}^- = \emptyset$ implies that A is necessarily nonsingular, and if $k < n$, so that UV^* is rank deficient, it also implies that α lies off $\mathbb{R}^-$ (and in particular is nonzero). The same condition also requires that the $k \times k$ matrix $\alpha I_k + V^*U$ has no eigenvalues on $\mathbb{R}^-$, since the nonzero eigenvalues of BC and CB are the same for any two matrices B and C [19, Thm. 1.32], [20, Thm. 1.3.22].

An explicit expression for a function of a matrix in the form (1.1) is given by Higham in [19, Thm. 1.35] (and also by Harris in [15, Lem. 2] for $\alpha = 0$). The result allows us to evaluate $f(A)$ and only requires that f be defined on the spectrum of A. We recall this definition and give the corresponding theorem.

DEFINITION 1.1 ([19, Def. 1.1]). *Let $A \in \mathbb{C}^{n\times n}$, let $\lambda_1, \ldots, \lambda_m$ be the distinct eigenvalues of A, and let $\zeta_1, \ldots, \zeta_m$ be their respective indices (that is, ζ_i is the order of the largest Jordan block in which λ_i appears). A function f is defined on the spectrum of A if the values $f^{(j)}(\lambda_i)$ exist for $j = 0, \ldots, \zeta_i - 1$ and $i = 1, \ldots, m$.*

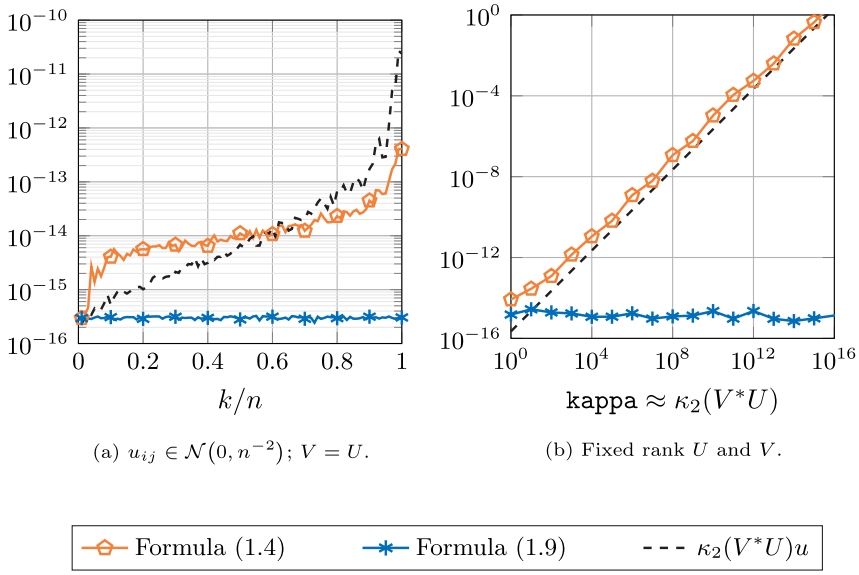
THEOREM 1.2 ([19, Thm. 1.35]). *Let $U, V \in \mathbb{C}^{n\times k}$ with $k \le n$ and assume that V^*U is nonsingular. Let f be defined on the spectrum of $A = \alpha I_n + UV^*$, and if $k = n$ let f be defined at α. Then*

$$f(A) = f(\alpha)I_n + U(V^*U)^{-1}\big(f(\alpha I_k + V^*U) - f(\alpha)I_k\big)V^*. \tag{1.2}$$

The theorem says two things: that $f(A)$, like A, is a perturbation of rank at most k of the identity matrix, and that $f(A)$ can be computed by evaluating f and the inverse at two $k \times k$ matrices. The formula (1.2) is of clear computational interest when $k \ll n$.

Note that if we take $f(x) = x^{-1}$ and write $A + UV^* = A(I_n + A^{-1}UV^*)$, then after a little manipulation we obtain as a special case of (1.2) the Sherman–Morrison–Woodbury formula, which says that if $I_k + V^*A^{-1}U$ is nonsingular then $A + UV^*$ is also nonsingular and

[1] https://docs.nvidia.com/cuda/cusolver/.



(a) $u_{ij} \in \mathcal{N}(0, n^{-2})$; $V = U$.

(b) Fixed rank U and V.

FIG. 1.1. *Relative residual of* (1.4) *and* (1.9) *in the* 2*-norm. The matrix* A *is of the form* (1.1) *for* $n = 100$, $\alpha = 1$, *and* $V = U$. *The elements of* U *are drawn from different distributions in the two panels. Note that the y-axes have a different range.*

$$(A + UV^*)^{-1} = A^{-1} - A^{-1}U(I_k + V^*A^{-1}U)^{-1}V^*A^{-1}. \tag{1.3}$$

Taking for f the square root in (1.2) gives

$$A^{1/2} = \alpha^{1/2}I_n + U(V^*U)^{-1}\big((\alpha I_k + V^*U)^{1/2} - \alpha^{1/2}I_k\big)V^*. \tag{1.4}$$

This formula is valid only if V^*U is nonsingular, yet this condition is not required for $A^{1/2}$ to be defined. This is undesirable, since there is no guarantee that for a rank-k perturbation written as UV^* the matrix V^*U will be nonsingular. Consider the $k = 1$ case with $U = e_i$ and $V = e_j$ for $i \neq j$: formula (1.4) fails since $V^*U = 0$, and there is no alternative way of writing this perturbation. In general, we cannot avoid a singular V^*U given only the assumption that $A^{1/2}$ is well defined, and thus the formula cannot always be applicable.

Another problem with (1.2) is that it may not provide full accuracy when evaluated in floating-point arithmetic if the condition number of V^*U is large. This is illustrated in Figure 1.1, where we compare the accuracy of (1.4) and (1.9) (see below) on matrices of the form (1.1) for $n = 100$ and $\alpha = 1$. The square root of a $k \times k$ matrix is computed by the Schur method using the MATLAB function `sqrtm`. We gauge the accuracy by measuring the 2-norm relative residual of the equation that defines the square root, that is, the quantity

$$\frac{\|\widehat{X}^2 - A\|}{\|A\|}, \tag{1.5}$$

where $\widehat{X}$ is a computed approximation to the square root obtained in MATLAB using binary64 arithmetic with unit roundoff $u_{64} \approx 1.1 \times 10^{-16}$. In order to reduce the magnitude of possible roundoff errors in the computation of the relative residual,

(1.5) is evaluated in binary128 arithmetic by relying on the Multiprecision Computing Toolbox for MATLAB [25].

In Figure 1.1(a), the matrix $U \in \mathbb{C}^{n\times k}$, for k varying between 1 and 100, has entries drawn from the normal distribution with mean 0 and variance n^{-2}, which we denote by $\mathcal{N}(0, n^{-2})$, and we set $V = U$ so that A is Hermitian. In Figure 1.1(b), the parameter k is set to 10, and the matrices U and V are generated with the MATLAB code

```
S = logspace(-log10(kappa), 0, k);
U = orth(randn(n, k));
V = U .* S;
```

This ensures that V^*U has condition number approximately equal to `kappa`, which in our experiment varies between 1 and 10^{16}. The relative residual of the solution computed using (1.4) deteriorates as the rank of UV^* or the condition number of V^*U increase.

Interestingly, the Sherman–Morrison–Woodbury formula (1.3) does not involve $(V^*U)^{-1}$, so for a particular f this term does not necessarily have to appear in a formula for $f(A)$. We now derive a formula for the pth root of a matrix of the form (1.1) that does not have the restriction that V^*U be nonsingular.

THEOREM 1.3. *Let $U, V \in \mathbb{C}^{n\times k}$ with $k \le n$ have full rank and let the matrix $A = \alpha I_n + UV^*$ have no eigenvalues on $\mathbb{R}^-$. Then for any integer $p \ge 1$,*

$$A^{1/p} = \alpha^{1/p} I_n + U \left(\sum_{i=0}^{p-1} \alpha^{i/p} \cdot (\alpha I_k + V^*U)^{(p-i-1)/p} \right)^{-1} V^*. \tag{1.6}$$

Proof. Assume, first, that V^*U is nonsingular. Taking for f the pth root in Theorem 1.2 gives

$$A^{1/p} = \alpha^{1/p} I_n + U(V^*U)^{-1} \left((\alpha I_k + V^*U)^{1/p} - \alpha^{1/p} I_k \right) V^*. \tag{1.7}$$

On the other hand, from the identity $a^p - b^p = (a-b)(\sum_{i=0}^{p-1} a^{p-i-1} b^i)$ we have

$$(\alpha I_k + V^*U)^{1/p} - \alpha^{1/p} I_k = V^*U \left(\sum_{i=0}^{p-1} \alpha^{i/p} \cdot (\alpha I_k + V^*U)^{(p-i-1)/p} \right)^{-1}, \tag{1.8}$$

where the matrix in parentheses is nonsingular because $\alpha I_k + V^*U$ and αI_k have no eigenvalue in common, which means that the left-hand side of (1.8) is nonsingular. Using the identity (1.8) in (1.7) gives (1.6). If V^*U is singular, consider the matrix $A(t) = \alpha I_n + U(t)V^*$, where $U(t) = U + tV$ for $t \in \mathbb{R}$. Then $V^*U(t) = V^*U + tV^*V$ is nonsingular for sufficiently small t (specifically, for any $t > 0$ if V^*U has no negative real eigenvalues and otherwise for $t \in (0, |\lambda|)$, where λ is the algebraically largest negative real eigenvalue of V^*U). By (1.6) we have

$$A(t)^{1/p} = \alpha^{1/p} I_n + (U + tV) \left(\sum_{i=0}^{p-1} \alpha^{i/p} \cdot \left(\alpha I_k + V^*U + tV^*V\right)^{(p-i-1)/p} \right)^{-1} V^*,$$

and taking the limit as $t \to 0$ gives (1.6). ∎

Our main interest is in $p = 2$, for which we have the following corollary.

COROLLARY 1.4. *Let $U, V \in \mathbb{C}^{n\times k}$ with $k \le n$ have full rank and let the matrix $A = \alpha I_n + UV^*$ have no eigenvalues on $\mathbb{R}^-$. Then*

$$A^{1/2} = \alpha^{1/2} I_n + U\big((\alpha I_k + V^*U)^{1/2} + \alpha^{1/2} I_k\big)^{-1} V^*. \tag{1.9}$$

The formula (1.9) in Corollary 1.4 is a significant improvement over (1.4), since it does not contain the factor $(V^*U)^{-1}$. Furthermore, in the experiments of Figure 1.1, formula (1.9) produces relative residuals of order u, unlike (1.4).

In section 2 we describe some applications that motivated this work. In section 3 we derive a Newton iteration that exploits the low-rank structure and provides an alternative to using (1.9); it is a structured variant of the Denman–Beavers iteration. In section 4 we compare the computational cost of several methods applied to the explicitly formed A or to (1.9), and in section 5 we compare the methods in terms of numerical stability and speed on random matrices as well as on positive definite matrices arising in real applications. Concluding remarks are offered in section 6.

We note that similar problems have been addressed in the literature. Bernstein and Van Loan [4] proposed an algorithm for computing $f(X+uv^T)$ for $X \in \mathbb{R}^{n\times n}$ and $u, v \in \mathbb{R}^n$, where f is a rational function defined on the spectra of X and $X + uv^T$. Beckermann, Kressner, and Schweitzer [3] proposed a polynomial Krylov method for approximating $f(X + UV^*)$ for any $X \in \mathbb{C}^{n\times n}$, provided that UV^* has low rank and that f is analytic on some domain containing the spectra of X and $X + UV^*$. The algorithms are given and their convergence analyzed for the case of rank-1 perturbations, but the authors suggest two approaches to apply the proposed algorithms to higher rank. More recently, Beckermann et al. [2] have developed a rational Krylov method to address the same problem. The proposed algorithm requires that the numerical range of X not contain a singularity of f. For the matrix square root, the convergence of the algorithm may be slow when $\alpha \in \mathbb{C}$ in (1.1) is close to the origin in the complex plane. Being Krylov-based, these methods are necessarily iterative, and they aim to approximate the correction $f(X+UV^*)-f(X)$ and compute $f(X+UV^*)$ as an update of $f(X)$, whereas the formula (1.9) is direct, has a predictable cost, and gives an explicit expression for $(\alpha I_n + UV^*)^{1/2}$.

Recently, Shumeli, Drineas, and Avron [30] developed a method to compute the quantity $(X \pm UU^T)^{\pm 1/2}$ for a symmetric positive semidefinite X. Their method is based on the approximate solution of an algebraic Riccati equation, and it allows for either symmetric positive semidefinite or symmetric negative semidefinite perturbations.

2. Applications. The need to compute roots of matrices of the form (1.1) arises in high-order optimization algorithms for machine learning [1], [14] and in machine vision [24].

The Shampoo technique, developed by Gupta, Koren, and Singer [14], is a preconditioned gradient method for second-order optimization. Computationally, the most expensive step of the algorithm is the evaluation of

$$L_t^{-1/2p} G_t R_t^{-1/2q}, \quad t = 1, \dots, \ell,$$

for some positive integers ℓ, p, and q, where

$$L_t = \alpha I_n + \sum_{s=1}^{t} G_s G_s^*, \quad R_t = \alpha I_k + \sum_{s=1}^{t} G_s^* G_s, \quad \alpha > 0, \tag{2.1}$$

and $G_1, \ldots, G_t \in \mathbb{R}^{n\times k}$ are of rank at most r. We note that the matrix $\sum_{s=1}^{t} G_s G_s^*$ can be written as UU^*, where $U = [G_1 \ldots G_t] \in \mathbb{R}^{n\times kt}$, which shows that L_t is of the form (1.1). The original implementation of Shampoo [14] used an SVD-based approach to compute the pth roots of L_t and G_t, but more recently Anil et al. [1] used the Schur–Newton algorithm of Guo and Higham [13] to compute the inverse pth roots. We note that the two algorithms are roughly equivalent for symmetric positive definite matrices such as L_t and R_t, as both the SVD and the Schur decomposition reduce to the eigendecomposition, and the only difference is in the way the pth roots of the eigenvalues are computed: the SVD-based algorithm computes the pth roots of the eigenvalues directly, whereas the Schur–Newton algorithm uses a scalar Newton iteration.

In representations for visual recognition [24, p. 39], the square root of a matrix of the form (1.1) is used in the spectral normalization of bilinear convolutional neural networks. This feature normalization technique runs an input image through a convolutional layer that extracts a set of k feature vectors $x_1, \ldots, x_k \in \mathbb{R}^n$ with nonnegative entries. These features are then aggregated via bilinear pooling, producing the matrix

$$A = \alpha I_n + \frac{1}{k}\sum_{i=1}^{k} x_i x_i^*, \quad \alpha > 0.$$

Since k depends on the size of the input image and of the convolutional filters, whereas n depends on the number of filters, these two numbers can be very different. Even when n and k are of similar magnitude, as often happens in state-of-the-art models [24, p. 52], many of the x_i may in principle be equal or very similar, producing a perturbation of rank much smaller than k. Because of the local nature of the convolutional filters, this is likely to happen when a large portion of the input image is filled by a homogeneous texture, as is the case for images with bursty features such as those considered in [24, Chap. 4]. The low-rank approximation can be obtained efficiently by using, for example, the randomized SVD algorithm recently developed by Nakatsukasa [26].

The need for computing the square root of a matrix of the form (1.1) also comes from numerical considerations in computing the square root of a singular or nearly singular matrix B. Rounding errors in floating-point arithmetic can displace small positive real eigenvalues of B to the negative real axis, where the principal square root is not well defined. In order to avoid potential issues, one can regularize B by adding the term αI for some small positive constant α: if B is factorized into a product of the form UV^* by truncating its SVD, then this diagonal shift produces a matrix of the form (1.1). This technique has been used, for example, to regularize some structured layers of deep neural networks [23]. The same regularization may be of interest when computing the inverse square roots of matrices of the form $\frac{1}{n}X^TX$, where X represents the data matrix and n is the number of samples. Matrices of this form arise in the training of deep neural networks [28], [37].

3. Newton iterations. An obvious approach for computing the square root is to apply any Newton iteration to A in (1.1) directly. For $k \ll n$, a more efficient strategy is to invoke (1.9) in Corollary 1.4 and apply the iteration to the $k \times k$ matrix $\alpha I_k + V^*U$. The standard Newton iteration is known to be numerically unstable [17], [19, sect. 6.4.1], so we focus on two of its numerically stable variants, namely, the Denman–Beavers iteration (DB) and its product form.

The (scaled) DB iteration [9], [19, sect. 6.3] is

$$\begin{aligned} X_{i+1} &= \frac{1}{2}\big(\mu_i X_i + \mu_i{}^{-1} Y_i^{-1}\big), \qquad X_0 = A, \\ Y_{i+1} &= \frac{1}{2}\big(\mu_i Y_i + \mu_i{}^{-1} X_i^{-1}\big), \qquad Y_0 = I, \end{aligned} \tag{3.1}$$

where the positive scaling parameter $\mu_i \in \mathbb{R}$ can be used to accelerate the convergence of the method in its initial steps. The choice $\mu_i = 1$ yields the unscaled DB method, for which X_i and Y_i converge quadratically to $A^{1/2}$ and $A^{-1/2}$, respectively. An effective but possibly expensive technique for choosing the parameter μ_i is determinantal scaling, which we discuss later in this section.

We prove by induction that if A is of the form (1.1) then for $i \geq 0$ the iterates X_i and Y_i can be written in the form

$$X_i = \beta_i I_n + U B_i V^*, \quad \beta_i \in \mathbb{C}, \quad B_i \in \mathbb{C}^{k\times k}, \tag{3.2}$$

$$Y_i = \gamma_i I_n + U C_i V^*, \quad \gamma_i \in \mathbb{C}, \quad C_i \in \mathbb{C}^{k\times k}. \tag{3.3}$$

For $i = 0$, this follows from setting $\beta_0 = \alpha$, $B_0 = I_k$, and $\gamma_0 = 1$, $C_0 = 0$. For the inductive step, by using the Sherman–Morrison–Woodbury formula (1.3) for X_i^{-1} and Y_i^{-1} we obtain

$$\begin{aligned} X_{i+1} &= \frac{1}{2}\big(\mu_i X_i + \mu_i{}^{-1} Y_i^{-1}\big) \\ &= \frac{\mu_i}{2}(\beta_i I_n + U B_i V^*) + \frac{(\mu_i\gamma_i)^{-1}}{2}\big(I_n - U C_i(\gamma_i I_k + V^* U C_i)^{-1} V^*\big) \\ &= \frac{\mu_i\beta_i + (\mu_i\gamma_i)^{-1}}{2} I_n + \frac{1}{2} U\big(\mu_i B_i - (\mu_i\gamma_i)^{-1} C_i(\gamma_i I_k + V^* U C_i)^{-1}\big) V^* \end{aligned}$$

and

$$\begin{aligned} Y_{i+1} &= \frac{1}{2}\big(\mu_i Y_i + \mu_i{}^{-1} X_i^{-1}\big) \\ &= \frac{\mu_i}{2}(\gamma_i I_n + U C_i V^*) + \frac{(\mu_i\beta_i)^{-1}}{2}\big(I_n - U B_i(\beta_i I_k + V^* U B_i)^{-1} V^*\big) \\ &= \frac{\mu_i\gamma_i + (\mu_i\beta_i)^{-1}}{2} I_n + \frac{1}{2} U\big(\mu_i C_i - (\mu_i\beta_i)^{-1} B_i(\beta_i I_k + V^* U B_i)^{-1}\big) V^*, \end{aligned}$$

so that

$$\beta_{i+1} = \frac{\mu_i\beta_i + (\mu_i\gamma_i)^{-1}}{2}, \tag{3.4a}$$

$$B_{i+1} = \frac{1}{2}\big(\mu_i B_i - (\mu_i\gamma_i)^{-1} C_i(\gamma_i I_k + V^* U C_i)^{-1}\big), \tag{3.4b}$$

$$\gamma_{i+1} = \frac{\mu_i\gamma_i + (\mu_i\beta_i)^{-1}}{2}, \tag{3.4c}$$

$$C_{i+1} = \frac{1}{2}\big(\mu_i C_i - (\mu_i\beta_i)^{-1} B_i(\beta_i I_k + V^* U B_i)^{-1}\big). \tag{3.4d}$$

With $W = V^*U \in \mathbb{C}^{k\times k}$ precomputed and stored, each step requires the solution of two $k\times k$ linear systems with k right-hand sides and two $k\times k$ matrix multiplications, for a total cost of $\frac{28}{3}k^3$ floating-point operations (flops).

Note that since $\beta_i \to \alpha^{1/2}$ and $\gamma_i \to \alpha^{-1/2}$, we might be tempted to remove the iterations for β_i and γ_i and replace β_i by $\alpha^{1/2}$ in (3.4d) and γ_i by $\alpha^{-1/2}$ in (3.4b). However, this choice changes the iteration, which is no longer convergent in general.

By introducing the product $M_i = X_iY_i$ and rewriting (3.1), one of the inversions can be traded for a multiplication, giving the product form of the DB iteration [7], [19, sect. 6.3]

$$\begin{aligned} M_{i+1} &= \frac{1}{2}\left(I_n + \frac{{\mu_i}^2 M_i + {\mu_i}^{-2}M_i^{-1}}{2}\right), \qquad M_0 = A,\\ X_{i+1} &= \frac{1}{2}\mu_i X_i\big(I_n + {\mu_i}^{-2}M_i^{-1}\big), \qquad X_0 = A. \end{aligned} \tag{3.5}$$

Here, $X_i \to A^{1/2}$ and $M_i \to I$ as $i \to \infty$.

Now we show that if A has the form (1.1) then for $i \geq 0$ the matrix M_i has the form

$$M_i = \nu_i I_n + UN_iV^*, \quad \nu_i \in \mathbb{C}, \quad N_i \in \mathbb{C}^{k\times k}, \tag{3.6}$$

and X_i has the form (3.2). For $i = 0$, this follows from setting $\nu_0 = \beta_0 = \alpha$ and $N_0 = B_0 = I_k$. For the inductive step, by using the Sherman–Morrison–Woodbury formula (1.3) for M_i^{-1} we obtain for M_{i+1} the expression

$$\begin{aligned} M_{i+1} &= \frac{1}{2}\left(I_n + \frac{{\mu_i}^2 M_i + {\mu_i}^{-2}M_i^{-1}}{2}\right)\\ &= \frac{1}{2}\left(I_n + \frac{{\mu_i}^2(\nu_i I_n + UN_iV^*) + {\mu_i}^{-2}{\nu_i}^{-1}(I_n - UN_i(\nu_i I_k + V^*UN_i)^{-1}V^*)}{2}\right)\\ &= \frac{2 + ({\mu_i}^2\nu_i + {\mu_i}^{-2}{\nu_i}^{-1})}{4}I_n + \frac{1}{4}U\big({\mu_i}^2N_i - ({\mu_i}^2\nu_i)^{-1}N_i(\nu_i I_k + V^*UN_i)^{-1}\big)V^*\\ &= \frac{2 + ({\mu_i}^2\nu_i + {\mu_i}^{-2}{\nu_i}^{-1})}{4}I_n + \frac{1}{4}U\big({\mu_i}^2N_i - S_i\big)V^*, \end{aligned} \tag{3.7}$$

where

$$S_i = ({\mu_i}^2\nu_i)^{-1}N_i(\nu_i I_k + V^*UN_i)^{-1}.$$

Similarly, for X_{i+1} we have

$$\begin{aligned} X_{i+1} &= \frac{\mu_i}{2}X_i\big(I_n + {\mu_i}^{-2}M_i^{-1}\big)\\ &= \frac{\mu_i}{2}(\beta_i I_n + UB_iV^*)\big(I_n + {\mu_i}^{-2}{\nu_i}^{-1}(I_n - UN_i(\nu_i I_k + V^*UN_i)^{-1}V^*)\big)\\ &= \frac{\mu_i}{2}\beta_i(1 + {\mu_i}^{-2}{\nu_i}^{-1})I_n + \frac{1}{2}U\big((\mu_i + {\mu_i}^{-1}{\nu_i}^{-1})B_i - \mu_i\beta_i S_i - \mu_i B_iV^*US_i\big)V^*. \end{aligned} \tag{3.8}$$

From (3.7) and (3.8) we can read off formulas for ν_{i+1}, N_{i+1}, β_{i+1}, and B_{i+1} in terms of ν_i, N_i, β_i, and B_i.

With V^*U computed initially and stored, forming S_i requires one $k \times k$ matrix multiplication and one $k \times k$ linear system solve with k right-hand sides, and computing B_i takes two additional $k \times k$ matrix products. Therefore, each iteration entails three $k \times k$ matrix products and the solution of a $k \times k$ linear system with k right-hand sides, for a total cost of $\frac{26}{3}k^3$ flops.

An effective scaling is determinantal scaling, which is given for the DB iteration by

$$\mu_i = \left| \frac{\sqrt{\det(A)}}{\det(X_i)} \right|^{1/n} = \left| \frac{1}{\det(X_i)\det(Y_i)} \right|^{1/2n}$$

and for the product form of the DB iteration by

$$\mu_i = \left| \frac{1}{\det(M_i)} \right|^{1/2n}.$$

In order to perform this scaling efficiently, however, it is necessary to exploit the structure of the matrices A, X_i, Y_i, and M_i when computing their determinants. We explain how to compute the determinant of X_i in (3.2); those of A in (1.1), of Y_i in (3.3), and of M_i in (3.6) can be computed analogously. By exploiting the identity $\det(I + AB) = \det(I + BA)$, we obtain

$$\begin{aligned} \det(X_i) &= \det(\beta_i I_n + U B_i V^*) \\ &= {\beta_i}^n \det(I_k + {\beta_i}^{-1}(V^*U)B_i) \\ &= {\beta_i}^{n-k} \det(\beta_i I_k + (V^*U)B_i), \end{aligned}$$

where the last expression involves only $k \times k$ matrices. Since β_i can be small, to avoid underflow in forming ${\beta_i}^{n-k}$ for large $n - k$ we should form directly

$$|\det(X_i)|^{1/n} = {\beta_i}^{1-k/n} \, |\det(\beta_i I_k + (V^*U)B_i)|^{1/n},$$

rather than computing $\det(X_i)$ explicitly.

The det term itself is also prone to underflow, so care is needed in its evaluation.

If μ_i becomes an infinity, a NaN, or 0 we set $\mu_i = 1$. As is customary when using scaled iterations [16], we also set $\mu_i = 1$ when the relative difference between μ_{i-1} and μ_i becomes small.

4. Cost comparison of the methods. Now we compare the computational cost of the methods discussed in the previous sections for computing the square root of the matrix A in (1.1).

In Table 4.1 the methods are divided into two categories: Schur-based methods and Newton methods. We report the asymptotic cost of the methods, where we assume that the linear systems are solved using LU factorization. We give the cost in terms of flops and in terms of matrix multiplications, matrix inversions, and multiple-right-hand-side system solves.

For $k \ll n$, the computational cost of computing $A^{1/2}$ is reduced from $O(n^3)$ for the standard (unstructured) methods to $O(n^2)$ for the methods that exploit the low-rank structure, assuming that for the Newton methods the number of iterations does not depend on k or n. Among the Schur-free methods that exploit the low-rank structure, formula (1.9) has the least cost, regardless of what form of the DB iteration

TABLE 4.1
Asymptotic cost of methods for computing $(\alpha I + UV^*)^{1/2}$ for $U, V \in \mathbb{C}^{n\times k}$. The second and third column report the cost of the methods in terms of flops and matrix operations, respectively. Here, D_k, I_k, and M_k denote the solution of a $k \times k$ linear system with k right-hand sides, the inversion of a matrix of order k, and the multiplication of two matrices of order k, respectively. For the iterative methods, N is the total number of iterations performed.

	Method	Total flops	Operations per iteration
Schur-based	Schur method	$28\frac{1}{3}n^3$	–
	Formula (1.9) with Schur method	$2kn^2 + 4k^2n + 29k^3$	–
Newton	DB iteration	$4Nn^3$	$2\,\mathrm{I}_n$
	Product DB iteration	$4Nn^3$	$\mathrm{M}_n + \mathrm{I}_n$
	Structured DB	$2kn^2 + 4k^2n + 9\frac{1}{3}Nk^3$	$2\,\mathrm{M}_k + 2\,\mathrm{D}_k$
	Structured product DB	$2kn^2 + 4k^2n + 8\frac{2}{3}Nk^3$	$3\,\mathrm{M}_k + \mathrm{D}_k$
	Formula (1.9) with DB	$2kn^2 + 4k^2n + 4Nk^3$	$2\,\mathrm{I}_k$
	Formula (1.9) with product DB	$2kn^2 + 4k^2n + 4Nk^3$	$\mathrm{M}_k + \mathrm{I}_k$

is used to compute the $k \times k$ square root. It will be cheaper to evaluate (1.9) using the DB iteration (plain or in product form) as long as convergence is achieved in no more than 7 steps, while the Schur method will be more convenient for matrices that would require 8 or more iterations.

5. Numerical experiments. In this section we evaluate the performance of four methods for computing the square root of matrices of the form (1.1), which are implemented in the following MATLAB codes.

- `schur_full`: an algorithm that first builds the matrix A in (1.1) by computing the outer product and then computes its square root using the MATLAB function `sqrtm`, which implements the Schur method [6], [8], [18]. If $U = V$, the matrix A is normal and its triangular Schur factor is diagonal. In this case, this algorithm reduces to the computation of $Q\Lambda^{1/2}Q^*$, where $A = Q\Lambda Q^*$ is a spectral decomposition of A.
- `schur_k`: an implementation of (1.9), where the square root of the $k \times k$ matrix is computed using `sqrtm` and the $k \times k$ linear system with k right-hand sides is solved using the MATLAB backslash operator.
- `db_prod_k`: an implementation of (1.9), where the square root of the $k \times k$ matrix is computed using the DB iteration in product form (3.5) with determinantal scaling and the $k \times k$ linear system with k right-hand sides is solved using the MATLAB backslash operator.
- `db_prod_struct`: the structured DB iteration in product form discussed in section 3, which iterates on $k \times k$ matrices. The algorithm uses the determinantal scaling in (3.6) and (3.7).

The experiments were run using the 64-bit GNU/Linux version of MATLAB 9.11.0 (R2021b Update 1) on a machine equipped with an AMD Ryzen 7 Pro 5850U running at 1.90 GHz and 32 GiB of RAM. The code we used to produce the results in this section is available on GitHub.[2]

For the DB iterations we use the following stopping criterion: we look at B_i, one of the $k \times k$ matrices on which we iterate, and we return the current approximation when

[2]https://github.com/Xiaobo-Liu/sqrtm-lrpsi.

the 1-norm of the relative change between two successive iterations falls below a given tolerance τ, which for our experiments was set to $10u_{64}$ for binary64 arithmetic and $8u_{32}$ for binary32 precision, where $u_{64} = 2^{-53} \approx 1 \times 10^{-16}$ and $u_{32} = 2^{-24} \approx 6 \times 10^{-8}$ are the unit roundoffs of binary64 and binary32 arithmetic, respectively.

For each computed square root $\widehat{X}$ of A, we compute the 2-norm relative residual in (1.5) and we gauge the quality of $\widehat{X}$ by comparing the relative residual with the quantity $\alpha_2(X)u$, where

$$\alpha_2(X) = \frac{\|X\|^2}{\|A\|}$$

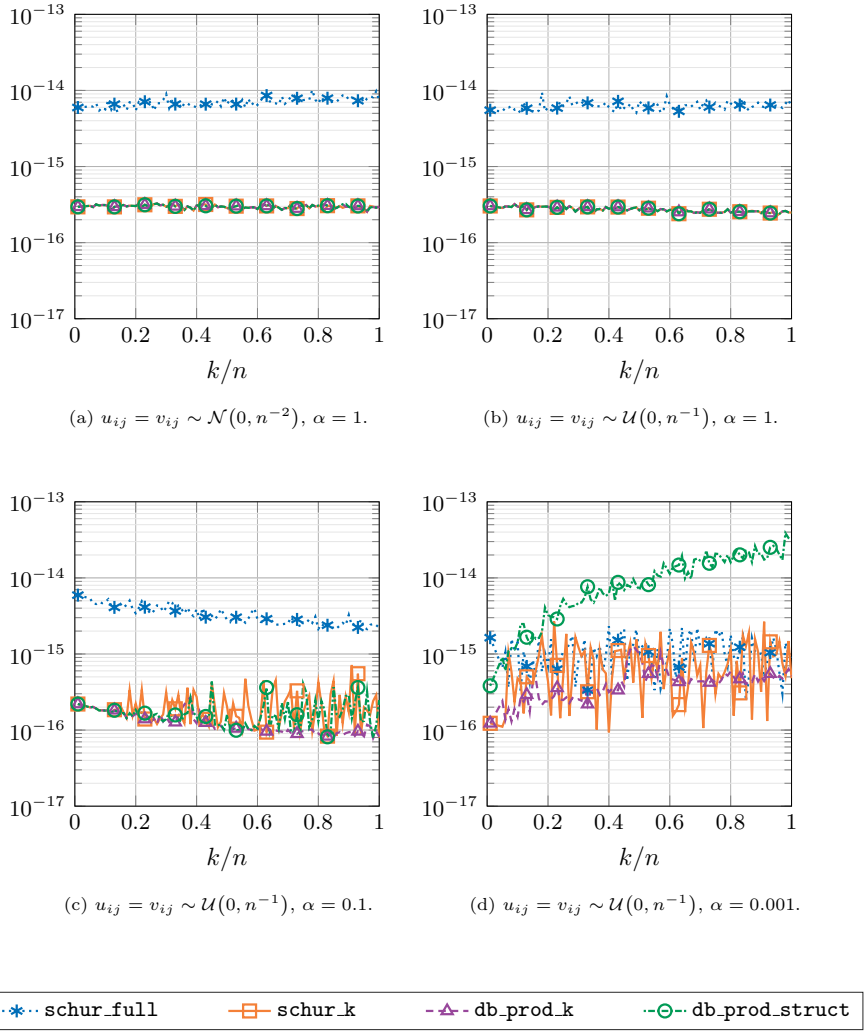
can be regarded as a condition number for the relative residual [6, sect. 4], [18, sect. 5], and u is the unit roundoff of the working precision. We note that $\alpha_2(X) \equiv 1$ if A is normal [6, sect. 4], thus we do not report the value of $\alpha_2(X)$ in such cases.

5.1. Quality. First we compare our four implementations in terms of the quality of the computed solution. In these experiments we consider the matrix A in (1.1) for $U = V$ and $n = 100$. For the dimension k we vary the ratio k/n from 0 to 1 with an increment of 0.05 and replace the zero ratio by 0.01. Figure 5.1 reports the relative residual (1.5). The matrix U has entries drawn from $\mathcal{N}(0, n^{-2})$ in Figure 5.1(a), and from the uniform distribution over the open interval $(0, n^{-1})$, which we denote by $\mathcal{U}(0, n^{-1})$, in Figure 5.1(b). In these two figures $\alpha = 1$ is used. In Figure 5.1(c,d) the matrix U has entries drawn from $\mathcal{U}(0, n^{-1})$ with $\alpha = 0.1$ and $\alpha = 0.001$, respectively.

In Figure 5.1(a,b) the relative residual of `db_prod_struct` is indistinguishable from that of `schur_k` and `db_prod_k`, which exploit the formula (1.9). The relative residual of `schur_full`, on the other hand, is about one and a half orders of magnitude larger in both cases and, in fact, `schur_full` is the only algorithm that produces a relative residual not of the same magnitude as $\alpha_2(X)u$; the reason for this mild instability is not clear. In Figure 5.1(c) the performance of the algorithms does not change much from that in Figure 5.1(b) when α decreases to 0.1. In Figure 5.1(d) we see that if $\alpha = 0.001$ then `db_prod_struct` shows signs of instability as k/n approaches 1, while the other algorithms remain largely stable and `schur_full` has relatively better performance for small α. With the chosen values of α, the matrix A is very well conditioned, as $\kappa_2(A) < 10$. We repeated the experiment with the setting in Figure 5.1(d) but further decreasing α to 10^{-6}. In this case $\kappa_2(A) = O(n)$, and the algorithms behaved as in Figure 5.1(d).

Next we test the algorithms on nonsymmetric matrices. We use the same experimental settings as in previous tests, but we now set $U \neq V$ so that A is nonsymmetric. The results are shown in Figure 5.2. There is no substantial difference between the behavior of the algorithms on symmetric and nonsymmetric matrices, although we note that the quality of the solutions computed by the Schur-based algorithms slightly deteriorates in Figure 5.2 compared with the results in Figure 5.1.

5.2. Timings. In Figure 5.3 we gauge how the execution time of our MATLAB implementations changes as the ratio k/n varies. In this experiment we consider the matrix A in (1.1) for $\alpha = 0.1$, $U = V$, and $n = 1000$, 4000, 7000, and 10000. The results reported here are for $u_{ij} \sim \mathcal{N}(0, n^{-2})$, but we have repeated the experiment with $u_{ij} \sim \mathcal{U}(0, n^{-1})$ and found that the behavior of the methods does not change significantly. As predicted by the analysis of the computational cost in section 4, `schur_full` is the only algorithm whose timings depend only on the order n of the input matrix but not on the rank k of the perturbation. The methods that exploit the structure of A, on the other hand, become slower as the ratio k/n grows. The fastest



(a) $u_{ij} = v_{ij} \sim \mathcal{N}(0, n^{-2})$, $\alpha = 1$.

(b) $u_{ij} = v_{ij} \sim \mathcal{U}(0, n^{-1})$, $\alpha = 1$.

(c) $u_{ij} = v_{ij} \sim \mathcal{U}(0, n^{-1})$, $\alpha = 0.1$.

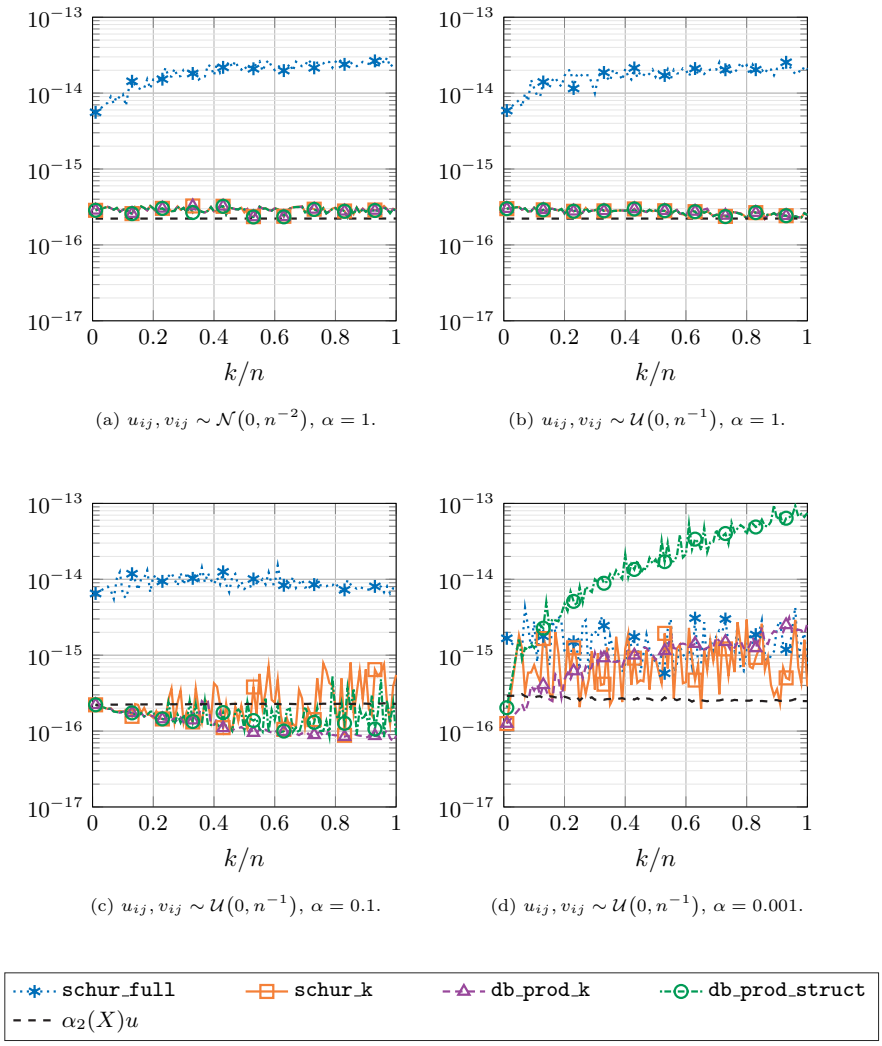
(d) $u_{ij} = v_{ij} \sim \mathcal{U}(0, n^{-1})$, $\alpha = 0.001$.

FIG. 5.1. *Relative residual of algorithms for computing the square root. The matrix A has the form (1.1) for $n = 100$, various choices of α, and $V = U$. The elements of U are drawn from different distributions.*

of the four implementations is `schur_k`, and its execution time never exceeds that of `schur_full` significantly. In this experiment `db_prod_struct` typically requires 7 iterations to converge and is the slowest, and `db_prod_k` typically requires 6 iterations to converge and is just slightly slower than `schur_k`.

We repeated the experiments with $\alpha = 1$ and obtained very similar results, although on this simpler problem `db_prod_k` and `db_prod_struct` were usually faster and required at most 2 and 3 iterations, respectively.

The picture is different for nonsymmetric matrices, as shown by Figure 5.4, which reports the results of the same experiments for the matrix A of the form (1.1) with $n = 1000$, 4000, 7000, and 10000, $\alpha = 0.1$, and $U \neq V$. The behavior of the algorithms does not change significantly in this setting, although `db_prod_k` becomes the fastest



(a) $u_{ij}, v_{ij} \sim \mathcal{N}(0, n^{-2})$, $\alpha = 1$.

(b) $u_{ij}, v_{ij} \sim \mathcal{U}(0, n^{-1})$, $\alpha = 1$.

(c) $u_{ij}, v_{ij} \sim \mathcal{U}(0, n^{-1})$, $\alpha = 0.1$.

(d) $u_{ij}, v_{ij} \sim \mathcal{U}(0, n^{-1})$, $\alpha = 0.001$.

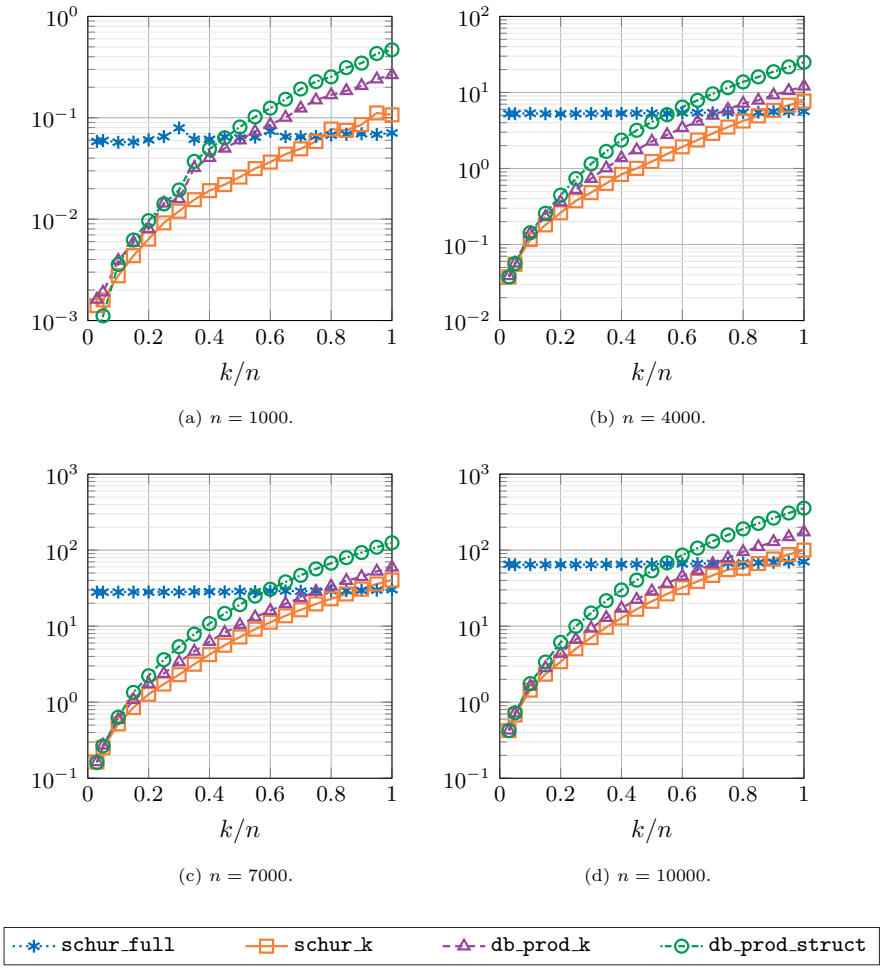
FIG. 5.2. *Relative residual of algorithms for computing the square root. The matrix A has the form* (1.1) *for $n = 100$, various choices of α, and $V \neq U$. The elements of U and V are drawn from different distributions.*

method for all matrix sizes, `schur_k` becomes the slowest by a considerable margin for $n = 1000$, whereas `db_prod_struct` is typically the slowest for larger matrices.

We remark that in both cases, the new algorithms we discuss can be up to two orders of magnitude faster than the traditional approach based on the Schur decomposition, when the ratio k/n is below $1/10$, which can be considered the typical range for low-rank updates.

5.3. Positive definite matrices from applications. Now we compare the structured iterations and the direct algorithms on three test matrices from machine learning applications [1]. These are provided as part of the Lingvo framework for TensorFlow [29] and are available on GitHub.[3] The matrices, which are provided in

[3]https://github.com/tensorflow/lingvo/tree/4c5572b/lingvo/core/testdata.

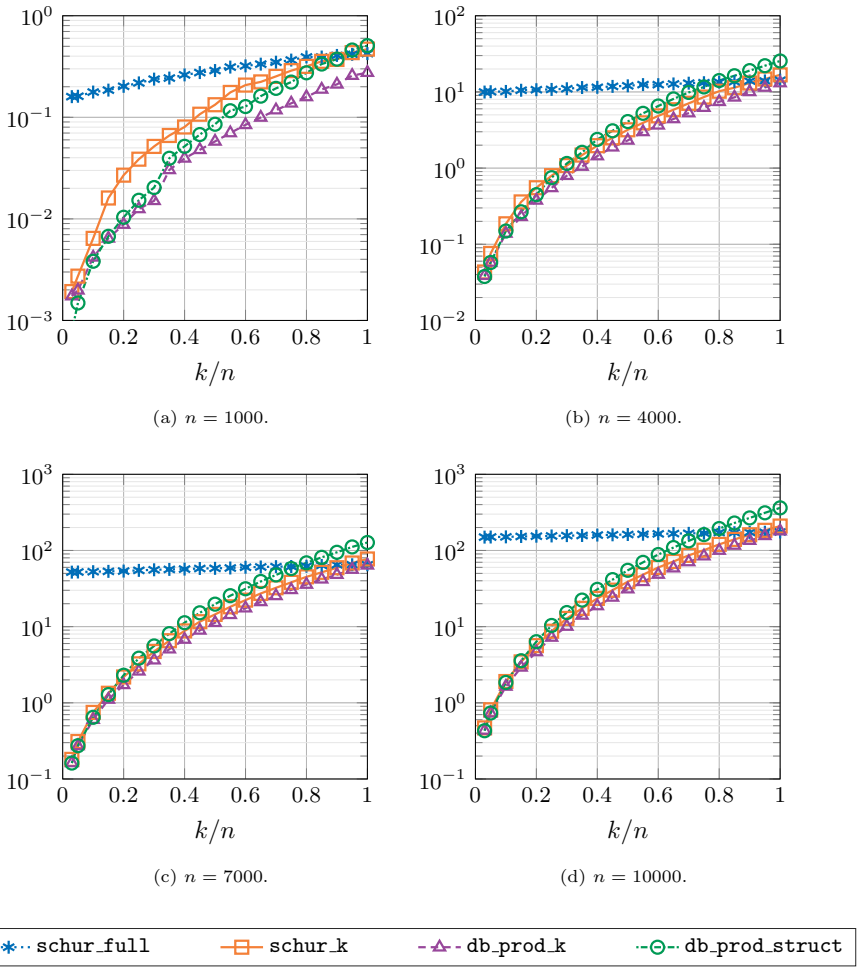


(a) $n = 1000$. (b) $n = 4000$. (c) $n = 7000$. (d) $n = 10000$.

FIG. 5.3. *Execution time (in seconds) of algorithms for computing the square root. The matrix A has the form (1.1) for $\alpha = 0.1$ and $V = U$. The elements of U are drawn from $\mathcal{N}(0, n^{-2})$.*

binary32 format, are formed by accumulating matrix products of the form (2.1) for $\alpha = 0$, and thus they are real and symmetric but are numerically indefinite because of rounding errors in the computation, having small negative real eigenvalues (see Table 5.1). We do not have access to the terms G_s in (2.1) used to generate the test matrices, and for the sake of our experiment we recover them from the test matrices as we now explain.

By the spectral theorem, the symmetric test matrix $B_i \in \mathbb{R}^{n\times n}$ can be decomposed as $Q\Sigma Q^T$, where $Q \in \mathbb{R}^{n\times n}$ is orthogonal and $\Sigma \in \mathbb{R}^{n\times n}$ is diagonal and has diagonal elements sorted in decreasing order. Let us now define the matrix $\widetilde{B}_i = Q_t\Sigma_t Q_t^T$, where $Q_t \in \mathbb{R}^{n\times t}$ collects the first t columns of Q and $\Sigma_t \in \mathbb{R}^{t\times t}$ is the leading principal submatrix minor of Σ of order t. In other words, $\widetilde{B}_i$ approximates B_i by truncating its spectral decomposition to rank t. By taking $G = Q_t{\Sigma_t}^{1/2}$, we can rewrite this approximation as $GG^T = \widetilde{B}_i \approx B_i$, which is implicitly of the form $\sum_{s=1}^{t} G_s G_s^T$ in (2.1). We choose t according to some tolerance $\varepsilon > 0$ such that all eigenvalues of $B_i \in \mathbb{R}^{n\times n}$ not less than ε are retained in $\Sigma_t \in \mathbb{R}^{t\times t}$. In the experiments we consider two different choices of the tolerance: $\varepsilon_1 = 0.1$ and $\varepsilon_2 = n^{3/2}u_{32}$.



(a) $n = 1000$. (b) $n = 4000$.

(c) $n = 7000$. (d) $n = 10000$.

FIG. 5.4. *Execution time (in seconds) of algorithms for computing the square root. The matrix A has the form* (1.1) *for $\alpha = 0.1$ and $V \neq U$. The elements of U and V are drawn from $\mathcal{N}(0, n^{-2})$.*

In Table 5.1 we list some important characteristics of the original test matrices, which we denote by B_1, B_2, and B_3, and our approximations to them, which we denote by $\widetilde{B}_1$, $\widetilde{B}_2$, and $\widetilde{B}_3$, respectively.

We examine the performance of the algorithms for computing the principal square root of $A_i = \alpha I_n + \widetilde{B}_i$ in binary32 arithmetic, where α is a positive real constant chosen so that the smallest eigenvalue of A_i is positive, which implies that A_i is positive definite. Given that practical values of the regularizing scalar α are not mentioned in [1], in the experiments we test three choices: $\alpha = 10^{-6}$, 10^{-3}, and 1.

The results are given in Table 5.2, and Figure 5.5 presents the same data pictorially. The matrices do not appear in the same order in the two panels of Figure 5.5; they are grouped by the value of α in Figure 5.5(a) and by size and rank in Figure 5.5(b).

All the methods except `db_prod_struct` converge for all test matrices with relative residual of the order $\alpha_2(A^{1/2})u_{32}$ in most cases, which indicates good numerical stability. In general, `db_prod_k` gives the solution that has the smallest residual; the other iterative method, `db_prod_struct` computes an unsatisfactory solution for

TABLE 5.1

Characteristics of the test matrices provided as part of the Lingvo framework and our approximations to them. For the test matrices B_i we report the size, n, the smallest and the largest eigenvalues $\lambda_{\min}$ and $\lambda_{\max}$ computed by the MATLAB `eig` *function using binary32 arithmetic, and the numerical rank r as returned by the MATLAB function* `rank`*. For the approximations $\widetilde{B}_i$ we report the order t_i of Σ_t in the truncated spectral decomposition with tolerance ε_i, as discussed in section* 5.3.

	n	$\lambda_{\min}$	$\lambda_{\max}$	r		ε_1	t_1	ε_2	t_2
B_1	1024	-1.3×10^{-2}	7.8×10^{4}	20	$\widetilde{B}_1$	1.0×10^{-1}	82	2.0×10^{-3}	128
B_2	512	-2.4×10^{-4}	5.6×10^{3}	173	$\widetilde{B}_2$	1.0×10^{-1}	221	6.9×10^{-4}	418
B_3	512	-1.9×10^{-4}	1.8×10^{3}	243	$\widetilde{B}_3$	1.0×10^{-1}	177	6.9×10^{-4}	511

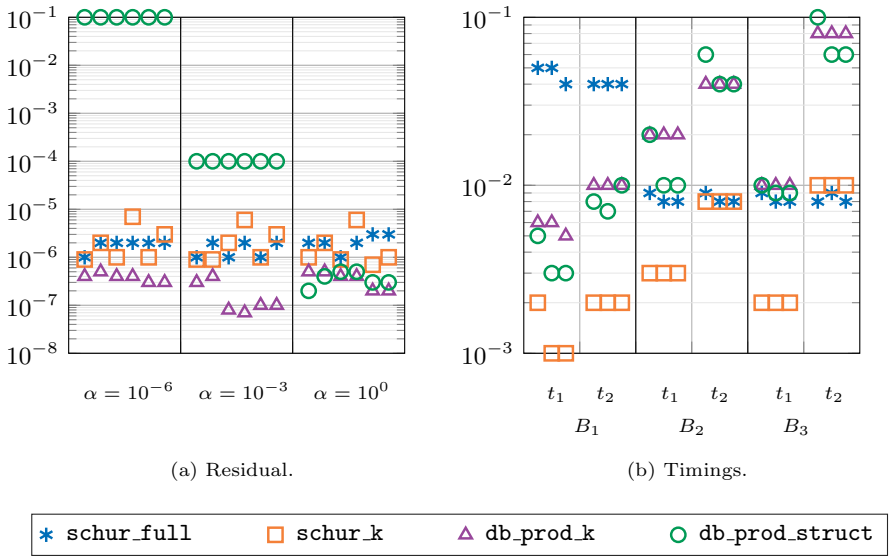
TABLE 5.2

Relative residual and execution time (in seconds) of algorithms for computing the square root. The matrices are those in Table 5.1.

	t	Method	$\alpha = 10^{-6}$		$\alpha = 10^{-3}$		$\alpha = 1$	
			Res	Time	Res	Time	Res	Time
B_1	82	`schur_full`	1×10^{-6}	5×10^{-2}	1×10^{-6}	5×10^{-2}	2×10^{-6}	4×10^{-2}
		`schur_k`	9×10^{-7}	2×10^{-3}	9×10^{-7}	1×10^{-3}	1×10^{-6}	1×10^{-3}
		`db_prod_k`	4×10^{-7}	6×10^{-3}	3×10^{-7}	6×10^{-3}	5×10^{-7}	5×10^{-3}
		`db_prod_struct`	1×10^{-1}	5×10^{-3}	1×10^{-4}	3×10^{-3}	2×10^{-7}	3×10^{-3}
	128	`schur_full`	2×10^{-6}	4×10^{-2}	2×10^{-6}	4×10^{-2}	2×10^{-6}	4×10^{-2}
		`schur_k`	2×10^{-6}	2×10^{-3}	9×10^{-7}	2×10^{-3}	2×10^{-6}	2×10^{-3}
		`db_prod_k`	5×10^{-7}	1×10^{-2}	4×10^{-7}	1×10^{-2}	5×10^{-7}	1×10^{-2}
		`db_prod_struct`	1×10^{-1}	8×10^{-3}	1×10^{-4}	7×10^{-3}	4×10^{-7}	1×10^{-2}
B_2	221	`schur_full`	2×10^{-6}	9×10^{-3}	1×10^{-6}	8×10^{-3}	1×10^{-6}	8×10^{-3}
		`schur_k`	1×10^{-6}	3×10^{-3}	2×10^{-6}	3×10^{-3}	9×10^{-7}	3×10^{-3}
		`db_prod_k`	4×10^{-7}	2×10^{-2}	8×10^{-8}	2×10^{-2}	4×10^{-7}	2×10^{-2}
		`db_prod_struct`	1×10^{-1}	2×10^{-2}	1×10^{-4}	1×10^{-2}	5×10^{-7}	1×10^{-2}
	418	`schur_full`	2×10^{-6}	9×10^{-3}	2×10^{-6}	8×10^{-3}	2×10^{-6}	8×10^{-3}
		`schur_k`	7×10^{-6}	8×10^{-3}	6×10^{-6}	8×10^{-3}	6×10^{-6}	8×10^{-3}
		`db_prod_k`	4×10^{-7}	4×10^{-2}	7×10^{-8}	4×10^{-2}	4×10^{-7}	4×10^{-2}
		`db_prod_struct`	1×10^{-1}	6×10^{-2}	1×10^{-4}	4×10^{-2}	5×10^{-7}	4×10^{-2}
B_3	177	`schur_full`	2×10^{-6}	9×10^{-3}	1×10^{-6}	8×10^{-3}	3×10^{-6}	8×10^{-3}
		`schur_k`	1×10^{-6}	2×10^{-3}	1×10^{-6}	2×10^{-3}	7×10^{-7}	2×10^{-3}
		`db_prod_k`	3×10^{-7}	1×10^{-2}	1×10^{-7}	1×10^{-2}	2×10^{-7}	1×10^{-2}
		`db_prod_struct`	1×10^{-1}	1×10^{-2}	1×10^{-4}	9×10^{-3}	3×10^{-7}	9×10^{-3}
	511	`schur_full`	2×10^{-6}	8×10^{-3}	2×10^{-6}	9×10^{-3}	3×10^{-6}	8×10^{-3}
		`schur_k`	3×10^{-6}	1×10^{-2}	3×10^{-6}	1×10^{-2}	1×10^{-6}	1×10^{-2}
		`db_prod_k`	3×10^{-7}	8×10^{-2}	1×10^{-7}	8×10^{-2}	2×10^{-7}	8×10^{-2}
		`db_prod_struct`	1×10^{-1}	1×10^{-1}	1×10^{-4}	6×10^{-2}	3×10^{-7}	6×10^{-2}

$\alpha = 10^{-6}$ and $\alpha = 10^{-3}$, but for $\alpha = 1$ its performance is on par with that of `db_prod_k`.

In terms of timings, `schur_full` is by far the slowest choice for B_1, but becomes comparable with `db_prod_k` and `db_prod_struct` for B_2 and B_3 when the rank of $\widetilde{B}_i$ is moderate compared with the size. `schur_k` is the fastest method, while its advantage over `db_prod_k` and `db_prod_struct` becomes negligible when $\widetilde{B}_i$ has low rank. The execution time of the two iterative methods `db_prod_k` and `db_prod_struct` is similar on most of the test matrices. Again, we observe that exploiting the structure of A delivers a significant performance improvement when $k \ll n$, in line with that suggested by the cost comparison in Table 4.1.



(a) Residual. (b) Timings.

FIG. 5.5. *Relative residual (left) and execution time in seconds (right) of algorithms for computing the square root. The matrices are those in Table* 5.1*; they are grouped by the value of* α *in the plot on the left, and by matrix in the plot on the right, where the two values of* t *for a matrix* B_i *are separated by a solid gray line.*

We conclude by mentioning that we derived, implemented, and tested the structured version of other variants of the Newton iteration, including the incremental form of Iannazzo [21] and the Newton–Schulz iteration. We found that their performance is similar to that of the structured DB iteration in product form.

6. Concluding remarks. We have investigated numerical methods for computing roots of a matrix $A = \alpha I_n + UV^*$, where U and V have rank $k \leq n$. We derived a new formula for $A^{1/p}$ that has the advantage over the existing formula from Theorem 1.2 of not requiring that V^*U be nonsingular. Focusing on the square root, we have also derived a new structured DB iteration that exploits the low-rank structure of UV^*.

Our numerical experiments confirm that when $k \ll n$, exploiting the structure yields algorithms that are much more efficient than simply applying the Schur method to A. If the Schur decomposition can be computed then using the Schur method to evaluate (1.9) is our preferred method overall. Otherwise, we recommend the use of the DB iteration, either in its structured form or as an unstructured algorithm to compute the $k \times k$ square root appearing in (1.9).

Acknowledgement. We thank the referees for their helpful comments on the manuscript.

REFERENCES

[1] R. ANIL, V. GUPTA, T. KOREN, K. REGAN, AND Y. SINGER, *Scalable Second Order Optimization for Deep Learning*, preprint, [cs.LG], 2021.

[2] B. BECKERMANN, A. CORTINOVIS, D. KRESSNER, AND M. SCHWEITZER, *Low-rank updates of matrix functions* II*: Rational Krylov methods*, SIAM J. Numer. Anal., 59 (2021), pp. 1325–1347, https://epubs.siam.org/doi/10.1137/20M1362553.

[3] B. Beckermann, D. Kressner, and M. Schweitzer, *Low-rank updates of matrix functions*, SIAM J. Matrix Anal. Appl., 39 (2018), pp. 539–565, https://epubs.siam.org/doi/10.1137/17M1140108.
[4] D. S. Bernstein and C. F. Van Loan, *Rational matrix functions and rank-1 updates*, SIAM J. Matrix Anal. Appl., 22 (2000), pp. 145–154, https://doi.org/10.1137/S0895479898333636.
[5] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, *Julia: A fresh approach to numerical computing*, SIAM Rev., 59 (2017), pp. 65–98, https://doi.org/10.1137/141000671.
[6] Å. Björck and S. Hammarling, *A Schur method for the square root of a matrix*, Linear Algebra Appl., 52/53 (1983), pp. 127–140, https://doi.org/10.1016/0024-3795(83)80010-X.
[7] S. H. Cheng, N. J. Higham, C. S. Kenney, and A. J. Laub, *Approximating the logarithm of a matrix to specified accuracy*, SIAM J. Matrix Anal. Appl., 22 (2001), pp. 1112–1125, https://doi.org/10.1137/S0895479899364015.
[8] E. Deadman, N. J. Higham, and R. Ralha, *Blocked Schur algorithms for computing the matrix square root*, in Applied Parallel and Scientific Computing: 11th International Conference, PARA 2012, Helsinki, Finland, Lecture Notes in Comput, Sci. 7782, P. Manninen and P. Öster, eds., Springer, Berlin, 2013, pp. 171–182, https://doi.org/10.1007/978-3-642-36803-5˙12.
[9] E. D. Denman and A. N. Beavers Jr, *The matrix sign function and computations in systems*, Appl. Math. Comput., 2 (1976), pp. 63–94, https://doi.org/10.1016/0096-3003(76)90020-5.
[10] M. Fasi and N. J. Higham, *Multiprecision algorithms for computing the matrix logarithm*, SIAM J. Matrix Anal. Appl., 39 (2018), pp. 472–491, https://doi.org/10.1137/17M1129866.
[11] G. H. Golub and C. F. Van Loan, Matrix Computations. 4th ed., Johns Hopkins University Press, Baltimore, MD, 2013.
[12] F. Greco and B. Iannazzo, *A binary powering Schur algorithm for computing primary matrix roots*, Numer. Algorithms, 55 (2010), pp. 59–78, https://doi.org/10.1007/s11075-009-9357-1.
[13] C.-H. Guo and N. J. Higham, *A Schur–Newton method for the matrix pth root and its inverse*, SIAM J. Matrix Anal. Appl., 28 (2006), pp. 788–804, https://doi.org/10.1137/050643374.
[14] V. Gupta, T. Koren, and Y. Singer, *Shampoo: Preconditioned stochastic tensor optimization*, Proc. Mach. Learn. Res. (PMLR), 80 (2018), pp. 1842–1850, http://proceedings.mlr.press/v80/gupta18a.html.
[15] L. A. Harris, *Computation of functions of certain operator matrices*, Linear Algebra Appl., 194 (1993), pp. 31–34, https://doi.org/10.1016/0024-3795(93)90111-Z.
[16] N. J. Higham, *Computing the polar decomposition—with applications*, SIAM J. Sci. Statist. Comput., 7 (1986), pp. 1160–1174, https://doi.org/10.1137/0907079.
[17] N. J. Higham, *Newton's method for the matrix square root*, Math. Comp., 46 (1986), pp. 537–549, https://doi.org/10.1090/S0025-5718-1986-0829624-5.
[18] N. J. Higham, *Computing real square roots of a real matrix*, Linear Algebra Appl., 88/89 (1987), pp. 405–430, https://doi.org/10.1016/0024-3795(87)90118-2.
[19] N. J. Higham, *Functions of Matrices: Theory and Computation*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008, https://doi.org/10.1137/1.9780898717778.
[20] R. A. Horn and C. R. Johnson, *Matrix Analysis*, 2nd ed., Cambridge University Press, Cambridge, UK, 2013.
[21] B. Iannazzo, *A note on computing the matrix square root*, Calcolo, 40 (2003), pp. 273–283, https://doi.org/10.1007/s10092-003-0079-9.
[22] B. Iannazzo and C. Manasse, *A Schur logarithmic algorithm for fractional powers of matrices*, SIAM J. Matrix Anal. Appl., 34 (2013), pp. 794–813, https://doi.org/10.1137/120877398.
[23] C. Ionescu, O. Vantzos, and C. Sminchisescu, *Matrix backpropagation for deep networks with structured layers*, in Proceedings of the IEEE International Conference on Computer Vision, 2015, pp. 2965–2973, https://openaccess.thecvf.com/content_iccv_2015/papers/Ionescu_Matrix_Backpropagation_for_ICCV_2015_paper.pdf.
[24] T.-Y. Lin, *Higher-Order Representations for Visual Recognition*, Ph.D. thesis, College of Information and Computer Sciences, University of Massachusetts Amherst, Massachusetts, USA, 2020, https://doi.org/10.7275/wsca-m707.
[25] *Multiprecision Computing Toolbox*, Advanpix LLC, http://www.advanpix.com.
[26] Y. Nakatsukasa, *Fast and Stable Randomized Low-Rank Matrix Approximation*, preprint, arXiv:2009.11392 [math.NA], 2020.

[27] G. Schulz, *Iterative Berechung der reziproken Matrix*, Z Angew. Math. Mech., 13 (1933), pp. 57–59, https://doi.org/10.1002/zamm.19330130111.
[28] W. Shao, H. Yu, Z. Zhang, H. Xu, Z. Li, and P. Luo, *BWCP: Probabilistic Learning-to-Prune Channels for ConvNets via Batch Whitening*, preprint, arXiv:2105.06423 [cs.LG], 2021.
[29] J. Shen et al., *Lingvo: A Modular and Scalable Framework for Sequence-to-Sequence Modeling*, preprint, arXiv:1902.08295 [cs.LG], 2019.
[30] S. Shumeli, P. Drineas, and H. Avron, *Low-Rank Updates of Matrix Square Roots*, preprint, arXiv:2201.13156 [math.NA], 2022.
[31] M. I. Smith, *A Schur algorithm for computing matrix pth roots*, SIAM J. Matrix Anal. Appl., 24 (2003), pp. 971–989, https://doi.org/10.1137/S0895479801392697.
[32] Y. Song, N. Sebe, and W. Wang, *Fast Differentiable Matrix Square Root*, preprint, arXiv:2201.08663 [cs.CV], 2022.
[33] *Symbolic Math Toolbox*, The MathWorks, Inc., Natick, MA, USA, http://www.mathworks.co.uk/products/symbolic/.
[34] D. S. Watkins, *Understanding the QR algorithm*, SIAM Rev., 24 (1982), pp. 427–440, https://doi.org/10.1137/1024100.
[35] D. S. Watkins, *The QR algorithm revisited*, SIAM Rev., 50 (2008), pp. 133–145, https://doi.org/10.1137/060659454.
[36] D. S. Watkins, *Francis's algorithm*, Amer. Math. Monthly, 118 (2011), pp. 387–403, https://doi.org/10.4169/amer.math.monthly.118.05.387.
[37] C. Ye, X. Zhou, T. McKinney, Y. Liu, Q. Zhou, and F. Zhdanov, *Exploiting Invariance in Training Deep Neural Networks*, preprint, arXiv:2103.16634v2 [cs.CV], 2021.