MANCHESTER
1824

The University of Manchester

Multiprecision Algorithms for Computing the Matrix Logarithm

Fasi, Massimiliano and Higham, Nicholas J.

2017

MIMS EPrint: **2017.16**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

MULTIPRECISION ALGORITHMS FOR COMPUTING THE MATRIX LOGARITHM*

MASSIMILIANO FASI† AND NICHOLAS J. HIGHAM‡

Abstract. Two algorithms are developed for computing the matrix logarithm in floating point arithmetic of any specified precision. The backward error-based approach used in the state of the art inverse scaling and squaring algorithms does not conveniently extend to a multiprecision environment, so instead we choose algorithmic parameters based on a forward error bound. We derive a new forward error bound for Padé approximants that for highly nonnormal matrices can be much smaller than the classical bound of Kenney and Laub. One of our algorithms exploits a Schur decomposition while the other is transformation-free and uses only the computational kernels of matrix multiplication and the solution of multiple right-hand side linear systems. For double precision computations the algorithms are competitive with the state of the art algorithm of Al-Mohy, Higham, and Relton implemented in `logm` in MATLAB. They are intended for computing environments providing multiprecision floating point arithmetic, such as Julia, MATLAB via the Symbolic Math Toolbox or the Multiprecision Computing Toolbox, or Python with the mpmath or SymPy packages. We show experimentally that the algorithms behave in a forward stable manner over a wide range of precisions, unlike existing alternatives.

Key words. multiprecision arithmetic, matrix logarithm, principal logarithm, inverse scaling and squaring method, Fréchet derivative, Padé approximation, Taylor approximation, forward error analysis, MATLAB, logm.

AMS subject classifications. 65F30, 65F60

1. Introduction.

Let $A \in \mathbb{C}^{n\times n}$ be nonsingular with no nonpositive real eigenvalues. Any matrix $X \in \mathbb{C}^{n\times n}$ satisfying the matrix equation

$$X = \mathrm{e}^A \tag{1.1}$$

is a matrix logarithm of A. This equation has infinitely many solutions, but in applications one is typically interested in the principal matrix logarithm, denoted by $\log A$, which is the unique matrix X whose eigenvalues have imaginary part strictly between $-\pi$ and π. This choice is the most natural in that it guarantees that if the matrix is real then so is its logarithm and that if the matrix has positive real spectrum then so does its logarithm.

More generally, the unique matrix satisfying (1.1) having spectrum in the complex strip

$$L_k = \{z \in \mathbb{C} \mid (k-1)\pi < \operatorname{Im} z < (k+1)\pi\}, \qquad k \in \mathbb{Z},$$

is called the kth branch of the matrix logarithm, and is denoted by $\log_k A$. The choice $k = 0$ yields the principal logarithm $\log A$. From a computational viewpoint, being able to approximate $\log A$ is enough to determine the value of $\log_k A$ for any $k \in \mathbb{Z}$, in view of the identity $\log_k A = \log A + 2k\pi\mathrm{i}I$.

*Version of May 11, 2017. **Funding**: This work was supported by MathWorks, European Research Council Advanced Grant MATFUN (267526), and Engineering and Physical Sciences Research Council grant EP/I01912X/1. The opinions and views expressed in this publication are those of the authors, and not necessarily those of the funding bodies.

†School of Mathematics, The University of Manchester, Manchester M13 9PL, UK (`massimiliano.fasi@manchester.ac.uk`)

‡School of Mathematics, The University of Manchester, Manchester, M13 9PL, UK (`nick.higham@manchester.ac.uk`, http://www.maths.manchester.ac.uk/~higham)

The aim of this work is to develop an algorithm for $\log A$ that is valid for floating point arithmetic of any given precision. A new algorithm is needed because the state of the art algorithm of Al-Mohy, Higham, and Relton [3], [4], which is implemented in the MATLAB function `logm`, is designed specifically for IEEE double precision arithmetic. Indeed most available software for the matrix logarithm has this limitation of being precision-specific [31]. Applications of the new algorithm will be in both low and high precision contexts. For example, both the matrix logarithm [34] and low precision computations (perhaps 32-bit single precision, or 16-bit half precision) [15], [25], have been recently used in machine learning, and a combination of the two is potentially of interest.

The need for high precision arises in several contexts. For instance, in order to estimate the forward error of a double precision algorithm for the matrix logarithm a reference solution computed at quadruple or higher precision is usually needed. Estimating the backward error of a double precision algorithm for the matrix exponential also requires the ability to evaluate $\log A$ at high precision. Let $X = \mathrm{e}^A$ and let $\widetilde{X}$ be a solution computed by a double precision algorithm for the matrix exponential. If the spectrum of A lies inside L_k, so that $A = \log_k X$, then the backward error of $\widetilde{X}$ is naturally defined as the matrix ΔA such that

$$\log_k \widetilde{X} = A + \Delta A, \tag{1.2}$$

because then $\widetilde{X} = \mathrm{e}^{A+\Delta A}$ and the normwise relative backward error is $\|\Delta A\|/\|A\| = \|\log \widetilde{X} - A\|/\|A\|$.

A multiprecision algorithm for the matrix logarithm is needed in a variety of languages and libraries that attempt to provide multiprecision implementations of a wide range of functions with both scalar and matrix arguments. The Julia language [8] and Python's SymPy [42], [47] and mpmath [37] libraries currently lack such an algorithm, and we will show that the the algorithms proposed here improve upon those in version 7.1 of the Symbolic Toolbox for MATLAB [46] and version 4.3.2 of the Multiprecision Computing Toolbox [43].

The algorithm of Al-Mohy, Higham, and Relton used by `logm` is the culmination of a line of inverse scaling and squaring algorithms that originates with Kenney and Laub [38] for matrices and goes back to Briggs [11] in the 17th century in the scalar case. In essence, the algorithm performs three steps. Initially, it takes square roots of A, s of them, say, until the spectrum of $A^{1/2^s} - I$ is within the unit disk, which is the largest disk centered at the origin in which the principal branch of $\log(1+x)$ is analytic and its Padé approximants are therefore well defined. Then it selects a Padé approximant $r_{km}(x) := p_k(x)/q_m(x)$ to $\log(1+x)$ of suitable degree $[k/m]$, evaluates the rational matrix function $r_{km}(X) = p_{km}(X)\ q_{km}(X)^{-1}$ at $X = A^{1/2^s} - I$, and finally reverts the square roots to form the approximation $\log A \approx 2^s r_{km}(A^{1/2^s} - I)$. The algorithm is based on a backward error analysis and uses pre-computed constants that specify how small a normwise measure of $A^{1/2^s} - I$ must be in order for a given diagonal Padé approximant r_{mm} to deliver a backward stable evaluation in IEEE double precision arithmetic. These constants require a mix of symbolic and high precision computation and it is not practical to compute them during the execution of the algorithm for different precisions. Therefore in this work we turn to forward error bounds, as were used in earlier work [14], [38].

Kenney and Laub [39] showed that for $\|X\| < 1$ and any subordinate matrix norm,

$$\|\log(I - X) - r_{km}^{-}(X)\| \le |\log(1 - \|X\|) - r_{km}^{-}(\|X\|)|, \tag{1.3}$$

where $r^-_{km}(x)$ is the $[k/m]$ approximant to $\log(1-x)$. In subsequent literature, the equivalent bound

$$\|\log(I+X) - r_{km}(X)\| \le |\log(1-\|X\|) - r_{km}(-\|X\|)| \tag{1.4}$$

has been preferred [29], [30, sec. 11.4]. Both upper bounds can be evaluated at negligible cost, so they provide a way to choose the Padé degrees k and m. We will derive and exploit a new version of the latter bound that it is phrased in terms of the quantities

$$\alpha_p(X) = \max\bigl(\|X^p\|^{1/p}, \|X^{p+1}\|^{1/(p+1)}\bigr), \tag{1.5}$$

for suitable p, instead of $\|X\|$. Since $\alpha_p(X)$ is no larger than $\|X\|$, and can be much smaller when X is highly nonnormal, the new bound leads to a more efficient algorithm.

Since in higher precision the algorithm may need a considerable number of square roots, it can happen that $\log(I+X)$ has very small norm and thus that an absolute error bound is not sufficient to guarantee that the algorithm will deliver a result with small relative error. For this reason, unlike in some previous algorithms we will use a relative error bound containing an inexpensive estimate of $\|\log(I+X)\|$.

It is well known that for $X \ge 0$ and $k+m$ fixed the diagonal Padé approximant ($k=m$) minimizes the error $\|\log(I-X) - r^-_{km}(X)\|$ [39] and the cost in flops of evaluating $r_{km}(X)$ is roughly constant. Therefore diagonal approximants $r_m := r_{mm}$ have been favoured. However, the special case of the Taylor approximant $t_m := r_{m0}$ merits consideration here, as its evaluation requires only matrix multiplications, which in practice are faster than multiple right-hand side solves. Throughout the paper we write f_m to denote either the Padé approximant r_m or the Taylor approximant t_m.

In addition to the matrix logarithm itself, we are also interested in evaluating its Fréchet derivative. Being able to evaluate the Fréchet derivative and its adjoint allows us to estimate the condition number $\kappa_{\log}(A)$ of the matrix logarithm, which in turn gives an estimate of the accuracy of the computed logarithm.

We use the term "multiprecision arithmetic" to mean arithmetic supporting multiple, and usually arbitrary, precisions. These precisions can be lower or higher than the single and double precisions that are supported by the IEEE standard [35] and usually available in hardware. We note that the 2008 revision of the IEEE standard [36] also supports a quadruple precision floating point format and, for storage only, a half precision format.

We begin the paper by summarizing in Section 2 available multiprecision computing environments. In Section 3 we derive a new forward bound for the error of Padé approximation of a class of hypergeometric functions, which yields a bound sharper than (1.3) and (1.4) for highly nonnormal matrices. In Section 4 we describe a new Schur–Padé algorithm for computing the matrix logarithm and its Fréchet derivative at any given precision. Section 5 explores a Schur-free version of the algorithm. Numerical examples are presented in Section 6 and concluding remarks are given in Section 7.

Finally, we introduce some notation. The unit roundoff of the floating point arithmetic is denoted by u. We recall that the Frechét derivative of a matrix function $f : \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$ at $A \in \mathbb{C}^{n\times n}$ is the linear functional $D_f(A) : \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$ that satisfies

$$f(A+E) = f(A) + D_f(A)[E] + o(\|E\|).$$

The relative condition number of the matrix function f at A is defined as

$$\kappa_f(A) = \lim_{\epsilon \to 0} \sup_{\|E\| \le \epsilon \|A\|} \frac{\|f(A+E) - f(A)\|}{\epsilon \|f(A)\|},$$

and is explicitly given by the formula [30, Thm. 3.1]

$$\kappa_f(A) = \frac{\|K_f(A)\| \, \|A\|}{\|f(A)\|}.$$

2. Support for multiple precision arithmetic. A wide range of software supporting multiprecision floating point arithmetic is available. Multiprecision capabilities are a built-in feature of Maple [40] and Mathematica [41] as well as the open-source PARI/GP [44] and Sage [45] computer algebra systems, and are available in MATLAB through the Symbolic Math Toolbox [46] and the Multiprecision Computing Toolbox [43]. The programming language Julia [8] supports multiprecision floating point numbers by means of the built-in data type `BigFloat`, while for other languages third-party libraries are available: mpmath [37] and SymPy [42], [47], for Python, the GNU MP Library [24] and the GNU MPFR Library [20] for C, the BOOST libraries [10] for C++, and the ARPREC library [5] for C++ and Fortran. The GNU MPFR Library is used in Julia, Maple, Sage, and the Multiprecision Computing Toolbox for MATLAB, and interfaces for several programming languages are available[1].

3. Error in the approximation of hypergeometric functions. We recall that the rational function $r_{km} = p_{km}/q_{km}$ is a $[k/m]$ Padé approximant of f if p_{km} and q_{km} are polynomials of degree at most k and m, respectively, $q_{km}(0) = 1$, and $f(x) - r_{km}(x) = O(x^{k+m+1})$. In order to obtain the required error bounds for Padé approximants to the logarithm we consider more generally Padé approximants to the hypergeometric function

$$ {}_2F_1(a, 1, c, x) = \sum_{i=0}^{\infty} \frac{(a)_i}{(c)_i} x^i,$$

where a and c are real numbers, x is complex with $|x| < 1$, and $(a)_i = a(a+1)\cdots(a+i-1)$ is the Pochhammer symbol for the rising factorial. Such Padé approximants have been well studied [6, sec. 2.3].

By combining the analysis of Kenney and Laub [39] with a result of Al-Mohy and Higham [2] we obtain the following stronger version of the error bound [39, Cor. 4]. Recall that α_p is defined in (1.5).

THEOREM 3.1. *Let $X \in \mathbb{C}^{n\times n}$ be such that $\rho(X) < 1$. Let $\widetilde{r}_{km}$ be the $[k/m]$ Padé approximant to ${}_2F_1(a, 1, c, y)$. If $m \le k+1$ and $0 < a < c$ then*

$$\|{}_2F_1(a, 1, c, X) - \widetilde{r}_{km}(X)\| \le |{}_2F_1(a, 1, c, \alpha_p(X)) - \widetilde{r}_{km}(\alpha_p(X))|, \tag{3.1}$$

for any p satisfying $p(p-1) \le k+m+1$.

Proof. Kenney and Laub [39, Thm. 5] show that if $m \le k+1$, $|y| < 1$, and $0 < a < c$ then

$${}_2F_1(a, 1, c, y) - \widetilde{r}_{km}(y) = \frac{q_{km}(1)}{q_{km}(y)} \sum_{i=k+m+1}^{\infty} \frac{(a)_i (i-k-m)_m}{(c)_i (i+a-m)_m} y^i, \tag{3.2}$$

[1] See http://www.mpfr.org.

where q_{km} is the denominator of $\widetilde{r}_{km}$, a polynomial of degree m. By [39, Cor. 1] the zeros of q_{km} are simple and lie on $(1,\infty)$, and since $q_{km}(0) = 1$ it follows that for $|y| < 1$,

$$q_{km}^{-1}(y) = \sum_{j=0}^{\infty} d_j y^j,$$

with $d_i > 0$ for all i. Since $a, c > 0$ and $i > k+m$, the coefficients of the series in (3.2) are positive and so (3.2) can be rewritten as

$$ {}_2F_1(a,1,c,y) - \widetilde{r}_{km}(y) = \sum_{i=k+m+1}^{\infty} \psi_i y^i, \tag{3.3}$$

where $\operatorname{sign}(\psi_i) = \operatorname{sign}(q_{km}(1))$ for all i. Therefore, applying [2, Thm. 4.2(a)] gives

$$\begin{aligned} \|{}_2F_1(a,1,c,X) - \widetilde{r}_{km}(X)\| &\le \sum_{i=k+m+1}^{\infty} |\psi_i| \alpha_p(X)^i \\ &= |{}_2F_1(a,1,c,\alpha_p(X)) - \widetilde{r}_{km}(\alpha_p(X))|, \end{aligned}$$

for $p(p-1) \le m+k+1$. ∎

For the matrix logarithm, we have

$$\frac{\log(1+x)}{x} = {}_2F_1(1,1,2,-x), \tag{3.4}$$

and thus the $[k/m]$ Padé approximant $r_{km}(x)$ to $\log(1+x)$ and the $[k/m]$ Padé approximant $\widetilde{r}_{km}(x)$ to ${}_2F_1(1,1,2,x)$ are related by

$$\frac{r_{km}(x)}{x} = \widetilde{r}_{k-1,m}(-x). \tag{3.5}$$

COROLLARY 3.2. *Let $X \in \mathbb{C}^{n\times n}$ be such that $\rho(X) < 1$ and $\alpha_p(X) < 1$, and let r_{km} be the $[k/m]$ Padé approximant to $\log(1+x)$. Then for $m \le k$, and p such that $p(p-1) \le k+m+1$, we have*

$$\|\log(I+X) - r_{km}(X)\| \le |\log(1-\alpha_p(X)) - r_{km}(-\alpha_p(X))|. \tag{3.6}$$

Proof. From (3.3), (3.4), and (3.5), with $a = 1$, $c = 2$, and $x = -y$ we have

$$-y^{-1}\left(\log(1-y) - r_{km}(-y)\right) = {}_2F_1(1,1,2,y) - \widetilde{r}_{k-1,m}(y) = \sum_{i=k+m}^{\infty} \psi_i y^i,$$

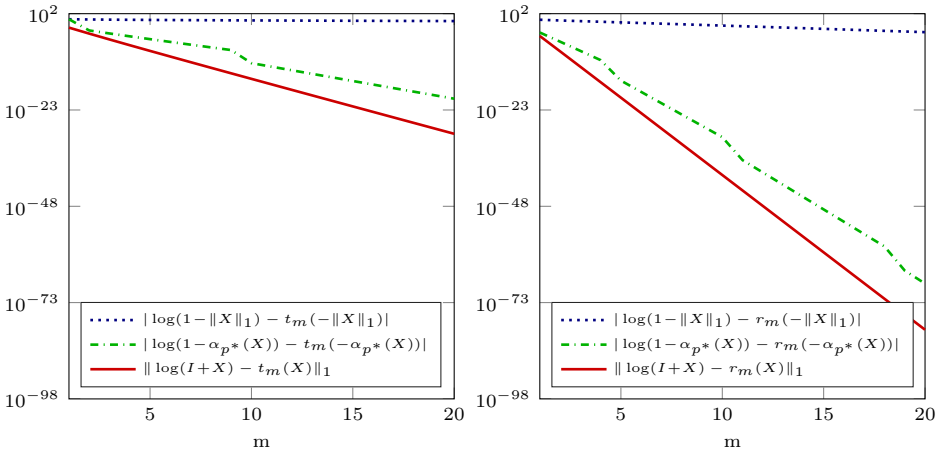
that is,

$$\log(1-y) - r_{km}(-y) = -\sum_{i=k+m}^{\infty} \psi_i y^{i+1}.$$

We know from the proof of Theorem 3.1 that the ψ_i are one-signed, and so we deduce that

$$\begin{aligned} \|\log(I-X) - r_{km}(-X)\| &\le \left| \sum_{i=k+m}^{\infty} \psi_i \alpha_p(X)^{i+1} \right| \\ &= |\log(1-\alpha_p(X)) - r_{km}(-\alpha_p(X))|. \end{aligned}$$

Since $\alpha_p(-X) = \alpha_p(X)$, we obtain the bound (3.6) on replacing X by $-X$. ∎



(a) t_m (b) r_m

Figure 3.1: Comparison of the bounds (1.4) and (3.6) for A in (3.9) and $1 \le m \le 20$, with p^* given by (3.8).

From the condition $p(p-1) \le k+m+1$ of the corollary we see that (3.6) holds for any $p \in \mathcal{I}_{[k/m]}$, where

$$\mathcal{I}_{[k/m]} = \left\{ n \in \mathbb{N} : 1 \le n \le \frac{(1+\sqrt{5+4(k+m)})}{2} \right\}. \tag{3.7}$$

Since the bound (3.6) is decreasing in $\alpha_p(X)$, the smallest bound is obtained for

$$p^* = \arg\min \left\{ \alpha_p(X) : p \in \mathcal{I}_{[k/m]} \right\}. \tag{3.8}$$

In practice we will approximate p^* rather than compute it exactly, as discussed in Section 4.

We compare in Figure 3.1 the bounds (1.4) and (3.6) for the diagonal Padé approximant r_m and the Taylor approximant t_m, for m between 1 and 20, with the fairly nonnormal matrix

$$A = \begin{bmatrix} 0.01 & 0.95 \\ 0 & 0.04 \end{bmatrix}. \tag{3.9}$$

We see that for both t_m and r_m the new bound can be many orders of magnitude smaller than (1.4) and it is a much better estimate of the actual error.

4. Schur–Padé Algorithm. In this section and the next we develop two new algorithms for computing the matrix logarithm in arbitrary precision floating point arithmetic, one using the Schur decomposition and one transformation-free. The algorithms build on the inverse scaling and squaring algorithms in [3], [4], [14], [30, Algs 11.9, 11.10]. They combine features from these algorithms in a novel way that yields algorithms that take the unit roundoff as an input parameter and need no pre-computed constants.

We approximate the Fréchet derivative of the logarithm by the Fréchet derivative of the logarithm's Padé approximant. We will not give an error bound for this approximation because, as noted in [4], it is problematic to obtain error bounds for it that are expressed in terms of the $\alpha_p(A)$. However our main intended use of the Fréchet derivative is for condition number estimation, which does not require accurate derivatives, and the same approximation was found to be perform well at double precision in [4].

Our precision-independent algorithm for the matrix logarithm and its Fréchet derivative is given in Algorithm 4.1. Instructions with an underlined line number are to be executed only when the Fréchet derivative of the matrix logarithm is required.

The algorithm begins by computing the Schur decomposition $A = QTQ^*$. In line 10 it repeatedly takes square roots of T until the spectrum of $T - I$ is within the unit ball centered at 0. Although the requirement $\rho(X) = \rho(T - I) < 1$ in Corollary 3.2 is now satisfied, there is no guarantee that the relative forward error $\|\log T - f_{m_{\max}}(T-I)\|_1 / \|\log T\|_1$, where $m_{\max}$ is the maximum degree of approximant allowed and f_m denotes r_m or t_m, is less than u. This is especially true for Taylor approximation, which could need hundreds of terms to achieve a bound on the forward error smaller than u.

Hence in line 13 the algorithm keeps taking square roots until

$$|\log(1 - \alpha_p(T - I)) - f_{m_{\max}}(-\alpha_p(T - I))| < u\psi(T), \tag{4.1}$$

where $\psi(T)$ is an estimate of the 1-norm of $\log T$ and p is chosen as described below. Approximating $\log T$ by the first term of the Taylor series, $\psi(T) = \|T - I\|_1$, provides an estimate accurate enough for all matrices and levels of precision considered in our numerical experiments.

Note that $\alpha_p(\|X\|_1)$ can be estimated efficiently, without explicitly forming any powers of X using the block 1-norm estimation algorithm of Higham and Tisseur [33], which requires only $O(n^2)$ flops. Since only an estimate of $\|X^p\|^{1/p}$ is computed there is no need to use high precision for this sub-problem, so we carry out this computation in double precision (or single precision if the requested precision is lower than double), in order to exploit the floating point hardware. When the matrix dimension is small and the working precision is not too high the cost of estimating $\alpha_p(T-I)$ can nevertheless be non-negligible. Rather than computing α_p for the optimal value p^* in (3.8), we compute it only for the largest possible p. Some justification for this choice comes from the fact that despite a sometimes considerably nonmonotonic behaviour, the sequence $\{\alpha_p(X)\}_{p\in\mathbb{N}}$ is typically roughly decreasing [2]. We denote the α value corresponding to the diagonal Padé approximant r_m by

$$\widetilde{\alpha}^r_m(X) = \alpha_p(X), \qquad p = \left\lfloor (1 + \sqrt{5 + 8m})/2 \right\rfloor,$$

and that corresponding to the truncated Taylor series t_m by

$$\widetilde{\alpha}^t_m(X) = \alpha_p(X), \qquad p = \left\lfloor (1 + \sqrt{5 + 4m})/2 \right\rfloor.$$

Thus (4.1) is used in the form

$$|\log(1 - \widetilde{\alpha}^f_{m_{\max}}(T - I)) - f_{m_{\max}}(-\widetilde{\alpha}^f_{m_{\max}}(T - I))| < u\psi(T). \tag{4.2}$$

We now discuss the cost of the algorithm, beginning with the case of diagonal Padé approximants. Higham [29] considered several ways of evaluating the rational

Algorithm 4.1 Schur–Padé algorithm for matrix logarithm and Fréchet derivative.

Given $A \in \mathbb{C}^{n\times n}$ with no eigenvalues on $\mathbb{R}^-$ this algorithm computes $X = \log A$, and optionally the Fréchet derivative $Y \approx D_{\log}(A)[E]$, in floating point arithmetic with unit roundoff u using inverse scaling and squaring with Padé approximation. $s_{\max}$ is the maximum allowed number of square roots and $m_{\max}$ the maximum allowed approximant degree. The logical parameter `use_taylor` determines whether a diagonal Padé approximant or a Taylor approximant is to be used. $\psi(X)$ provides an approximation to $\|\log X\|_1$.

Compute the complex Schur decomposition $A = QTQ^*$.
if `use_taylor` **then**
    $f = t$ and $\zeta(m)$ is defined as in (4.6). ▷ Taylor approximant.
else
    $f = r$, $\zeta(m) \leftarrow m - 2$ ▷ Padé approximant.
end if
$E \leftarrow Q^*EQ$
$T_0 \leftarrow T$
$s \leftarrow 0$
while $\max_{1\le i\le n}(|\sqrt{t_{ii}} - 1|) > 1$ **and** $s < s_{\max}$ **do**
    $[T, \overline{T}, E, s] \leftarrow \text{SQRTM}(T, E, s)$
end while
while $|\log(1 - \widetilde{\alpha}^f_{m_{\max}}(\overline{T})) - f_{m_{\max}}(-\widetilde{\alpha}^f_{m_{\max}}(\overline{T}))| \ge u\psi(T)$ **and** $s < s_{\max}$ **do**
    $[T, \overline{T}, E, s] \leftarrow \text{SQRTM}(T, E, s)$
end while
$\widetilde{m} \leftarrow \min\{m \le m_{\max} : |\log(1 - \widetilde{\alpha}^f_m(\overline{T})) - f_m(-\widetilde{\alpha}^f_m(\overline{T}))| < u\psi(T)\}$
while $|\log(1 - \widetilde{\alpha}^f_{\zeta(\widetilde{m})}(\overline{T})/2) - f_{\zeta(\widetilde{m})}(-\widetilde{\alpha}^f_{\zeta(\widetilde{m})}(\overline{T})/2)| < u\psi(T)$ **and** $s < s_{\max}$ **do**
    $[T, \overline{T}, E, s] \leftarrow \text{SQRTM}(T, E, s)$
    $\widetilde{m} \leftarrow \min\{m \le \widetilde{m} : |\log(1 - \widetilde{\alpha}^f_m(\overline{T})) - f_m(-\widetilde{\alpha}^f_m(\overline{T}))| < u\psi(T)\}$
end while
$\text{diag}(T, 1) \leftarrow \text{diag}(T_0^{1/2^s}, 1)$ using [32, eq. (5.6)].
$\text{diag}(T) \leftarrow \text{diag}(T_0)^{1/2^s} - I$ using [1, Alg. 2].
$X \leftarrow 2^s f_{\widetilde{m}}(T)$
$\text{diag}(T) \leftarrow \log(\text{diag}(T_0))$
Update the superdiagonal of T using [30, eq. (11.28)].
$X \leftarrow QXQ^*$
$Y \leftarrow 2^s Q f'_{\widetilde{m}}(T)Q^*$

function SQRTM($T \in \mathbb{C}^{n\times n}, E \in \mathbb{C}^{n\times n}, s \in \mathbb{N}$))
    $T \leftarrow T^{1/2}$ using [9, Alg. 6.3].
    $E \leftarrow X$, where X is the solution of $TX + XT = E$.
    return $T, T - I, E, s + 1$

function $r_m(X)$. The partial fraction form

$$r_m(A) = \sum_{j=1}^{m} \gamma_j^{(m)} (I + \delta_j^{(m)} A)^{-1} A, \tag{4.3}$$

where $\gamma_j^{(m)}$ and $\delta_j^{(m)}$ are the weights and nodes, respectively, of the m-point Gauss–

Legendre quadrature rule on $[0,1]$, was found to provide the best balance between efficiency and numerical stability, and it also has the advantage of allowing parallel evaluation. For a triangular matrix, the evaluation of (4.3) requires $mn^3/3$ flops and the computation of a matrix square root costs $n^3/3$ flops. Thus, when diagonal approximants are used, the algorithm requires $25n^3$ flops for the computation of the Schur decomposition, $\chi_r(s,m) := (s+m)\ n^3/3$ for the inverse scaling and squaring phase, and $3n^3$ flops to recover the solution.

Since $\chi_r(s,m) = \chi_r(s+1,m-1)$, an additional square root will save computational effort only if the degree of the approximant decreases by at least 2, in which case the overall reduction in cost is at least $n^3/3$ flops. In view of the approximation [3]

$$\alpha_p(A^{1/2^{s+1}} - I) \approx \frac{\alpha_p(A^{1/2^s} - I)}{2}, \tag{4.4}$$

an additional square root is taken only if

$$|\log(1 - \widetilde{\alpha}^f_{\zeta(\widetilde{m})}(T-I)/2) - f_{\zeta(\widetilde{m})}(-\widetilde{\alpha}^f_{\zeta(\widetilde{m})}(T-I)/2)| < u\psi(T), \tag{4.5}$$

with $f \equiv r$ and $\zeta(\widetilde{m}) = \widetilde{m} - 2$, where $\widetilde{m}$ is the current degree of the approximant (defined in line 16 of Algorithm 4.1).

Turning to Taylor series approximants, in order to evaluate the truncated Taylor series the algorithm uses the Paterson–Stockmeyer method, which among the four methods for polynomial evaluation considered in [30, Thm. 4.5] is the one that minimizes the number of matrix-matrix multiplications while satisfying a forward error bound of the same form as for the other methods. The computational cost of the square roots and Taylor approximant evaluation is approximately $\chi_t(s,m) = (s + 2\sqrt{m})\ n^3/3$ flops. Simple algebraic manipulations show that

$$\chi_t(s,m) = \chi_t\left(s+1, \left(\sqrt{m} - \frac{1}{2}\right)^2\right),$$

and thus an additional square root is taken only if (4.5) holds with $f \equiv t$ and

$$\zeta(\widetilde{m}) = \left\lceil \left(\sqrt{\widetilde{m}} - \frac{1}{2}\right)^2 \right\rceil - 1, \tag{4.6}$$

which guarantees that $\zeta(\widetilde{m}) < (\sqrt{\widetilde{m}} - \frac{1}{2})^2$. Since the cost function χ_t is monotonic in both arguments, the reduction in the number of flops will be at least

$$\chi_t(s,\widetilde{m}) - \chi_t(s+1,\zeta(\widetilde{m})) = \frac{2n^3}{3}\left(\sqrt{\widetilde{m}} - \left(\left\lceil \left(\sqrt{\widetilde{m}} - \frac{1}{2}\right)^2 \right\rceil - 1\right)^{1/2} - \frac{1}{2}\right), \tag{4.7}$$

which depends only on $\widetilde{m}$. Unlike in the Padé case, we cannot put a useful lower bound on the decrease in flops resulting from an extra square root.

For both types of approximant we can perform a binary search to find the smallest $m^* \in [1,\widetilde{m}]$ such that

$$|\log(1 - \widetilde{\alpha}^f_{m^*}(T-I)) - f_{m^*}(-\widetilde{\alpha}^f_{m^*}(T-I))| < u\psi(T), \tag{4.8}$$

which requires the estimation of $\|X^p\|_1^{1/p}$ for no more than $2\log_2 \widetilde{m} - 1$ values of p.

If the Frechét derivative is needed then each time a square root is taken the algorithm solves an additional Sylvester equation, as detailed in [30, sec. 11.8]. Once the optimal values for s and $\widetilde{m}$ have been found, in order to increase the accuracy the algorithm recomputes the first superdiagonal of T from T_0, making use of the identity [32, eq. (5.6)], and then the main diagonal of $T - I$, by applying [1, Alg. 2] to the diagonal of T_0. The algorithm can be easily adapted to compute the adjoint of the Frechét derivative of A in the direction E, by replacing the increment E by E^* and returning Y^* [4].

Algorithm 4.1 can be extended to compute an estimate of the 1-norm condition number of the matrix logarithm, using the same approach as in [4, Alg. 4.1]. Since in this case the Frechét derivative of the matrix logarithm of A and its adjoint need to be computed in several directions, but neither s nor $\widetilde{m}$ depend on E, the algorithm can be modified to store the matrix T after each square root is taken, and then use it to solve several Sylvester cascades for different matrices E. If η is the number of bits required to store a single entry of the matrix, then this modification increases the memory requirement of the algorithm by about $\eta(s-1)n^2/2$ bits if the upper triangular pattern is exploited.

5. Transformation-free algorithm. Multiprecision computing environments often provide just a few linear algebra kernels. For example, version 7.1 of the Symbolic Math Toolbox [46] does not support the Schur decomposition in its variable precision arithmetic (VPA). In this section we therefore present a version of Algorithm 4.1 that does not require the computation of the Schur decomposition and builds solely on level 3 BLAS operations and multiple right-hand side system solves.

The algorithm, whose pseudocode is given in Algorithm 5.2, builds on the transformation-free algorithms [3, Alg. 5.2], [14, Alg. 7.1], and again makes use of the improved forward error bound (3.6).

The algorithm starts by taking enough square roots to guarantee that the Padé or Taylor approximants will produce a relative forward error below the unit roundoff threshold. Since in this case the matrix is not triangular, to compute square roots the algorithm employs the scaled Denman–Beavers iteration (in product form) [30, eq. (6.29)], whose computational cost depends on the number of iterations and is thus not known a priori. However, it has been observed [30, sec. 11.5.2] that in practice up to ten iterations are typically required for the first few square roots, but just four or five are enough in the later stages. Since the cost of one iteration is $4n^3$ flops, it is customary to consider that the computation of a square root requires $16n^3$ flops. On the other hand, evaluating the diagonal Padé approximant in partial fraction form requires $8mn^3/3$ if the matrix is not upper triangular. The cost of the algorithm is $\chi_r(s,m)$ flops, where

$$\chi_r(s,m) = \left(\frac{8m}{3} + 16s\right) n^3,$$

and it can be readily seen that $\chi_r(s,m) = \chi_r(s+1, m-6)$, and thus an additional square root is taken if (4.5) holds for $f \equiv r$, $T = A$, and $\zeta(\widetilde{m}) = \widetilde{m} - 7$. Using the Patterson–Stockmeyer scheme to evaluate the truncated Taylor expansion, we get the asymptotic cost $\chi_t(s,m) = 4(\sqrt{m} + 4s)n^3$, which satisfies $\chi_t(s,m) = \chi_t(s+1, (\sqrt{m}-4)^2)$. In this case, an additional square root will be worthwhile if (4.5) holds for $f \equiv t$, $T = A$, and

Algorithm 5.2 Transformation-free matrix logarithm with Fréchet derivative

Given $A \in \mathbb{C}^{n\times n}$ with no eigenvalues on $\mathbb{R}^-$ this algorithm computes $X = \log A$, in floating point arithmetic with unit roundoff u using inverse scaling and squaring with Padé approximation. $s_{\max}$ is the maximum allowed number of square roots and $m_{\max}$ the maximum allowed approximant degree. The logical parameter `use_taylor` determines whether a diagonal Padé approximation or a Taylor approximation is to be used. $\psi(X)$ provides an approximation to $\|\log X\|_1$.

Compute the complex Schur decomposition $A = QTQ^*$.
if `use_taylor` **then**
    $f = t$ and $\zeta(m)$ is defined as in (5.1). ▷ Taylor approximant.
else
    $f = r$, $\zeta(m) \leftarrow m - 7$ ▷ Padé approximant.
end if
$s \leftarrow 0$
$\overline{A} \leftarrow A - I$
$Z \leftarrow \overline{A}$
$P \leftarrow I$
while $|\log(1 - \widetilde{\alpha}^f_m(\overline{A})) - f_m(-\widetilde{\alpha}^f_m(\overline{A}))| \geq u\psi(A)$ **and** $s < s_{\max}$ **or** $\|A\|_1 > 1$ **do**
    $[A, \overline{A}, P, s] \leftarrow$ SQRTM_DB(A, P, s)
end while
$\widetilde{m} \leftarrow \min\{m \leq m_{\max} : |\log(1 - \widetilde{\alpha}^f_m(\overline{A})) - f_m(-\widetilde{\alpha}^f_m(\overline{A}))| < u\psi(A)\}$
while $|\log(1 - \widetilde{\alpha}^f_{\zeta(\widetilde{m})}(\overline{A})/2) - f_{\zeta(\widetilde{m})}(-\widetilde{\alpha}^f_{\zeta(\widetilde{m})}(\overline{A})/2)| < u\psi(A)$ **and** $s < s_{\max}$ **do**
    $[A, \overline{A}, P, s] \leftarrow$ SQRTM_DB(A, P, s)
    $\widetilde{m} \leftarrow \min\{m \leq \widetilde{m} : |\log(1 - \widetilde{\alpha}^f_m(\overline{A})) - f_m(-\widetilde{\alpha}^f_m(\overline{A}))| < u\psi(A)\}$
end while
if $s < 2$ **then**
    $Y \leftarrow \overline{A}$
else
    $Y \leftarrow ZP^{-1}$
end if
$X \leftarrow 2^s f_{\widetilde{m}}(Y)$
return X

function SQRTM_DB$(A \in \mathbb{C}^{n\times n}, P \in \mathbb{C}^{n\times n}, s \in \mathbb{N})$
    $A \leftarrow A^{1/2}$ using the iteration in [30, eq. (6.29)].
    if $s > 1$ **then**
        $P \leftarrow P(A + I)$
    end if
    return $A, A - I, P, s$

$$\zeta(\widetilde{m}) = \left\lceil \left(\sqrt{\widetilde{m}} - 4\right)^2 \right\rceil - 1, \tag{5.1}$$

which guarantees a reduction in the number of flops of at least

$$\chi_t(s, \widetilde{m}) - \chi_t(s+1, \zeta(\widetilde{m})) = 4n^3\left(\sqrt{\widetilde{m}} - \left(\left\lceil \left(\sqrt{\widetilde{m}} - 4\right)^2 \right\rceil - 1\right)^{1/2} - 4\right). \tag{5.2}$$

To reduce the chances of numerical cancellation in the computation of $A^{1/2^s} - I$, the matrix form of [1, Alg. 2] is used, as in [3].

Note that Algorithm 5.2 is not suitable for the computation of the Fréchet derivative of the matrix logarithm, nor for the estimation of its condition number. Standard methods for the solution of Sylvester equations [7], [22] start by computing the Schur decomposition of one or both the coefficients of the matrix equation, and are thus not suitable for a framework where a multiprecision implementation of the QR algorithm is not available. The alternative of converting the Sylvester equation to an $n^2 \times n^2$ structured linear system $Ax = b$ and solving by Gaussian elimination has too high a computational cost to be useful in this context.

6. Numerical experiments. In this section we describe numerical tests with the new multiprecision algorithms for the matrix logarithm. All the experiments were performed using the 64-bit version of MATLAB 2017a on a machine equipped with an Intel I5-5287U processor running at 2.90GHz. For the underlying computations the implementations exploit the overloaded methods from the Multiprecision Computing Toolbox (version 4.3.2.12168) [43] to run in different precisions.

We test the following algorithms.

- `logm_mct`: the (overloaded) `logm` function from the Multiprecision Computing Toolbox, which implements a blocked version of the Schur–Parlett algorithm [16]. After computing the complex Schur decomposition, a blocking of the matrix is computed according to [16], [30, sec. 9.3]. For each diagonal block T_{ii} of the triangular Schur factor the algorithm repeatedly takes the square root until the spectrum of $T_{ii} - I$ lies within the unit ball centered at 1. Then it chooses the degree of the (diagonal) Padé approximant as the smallest m so that the bound in (1.3) is less than the unit roundoff. Since this strategy does not optimize the balance between the number of square roots and the Padé degree it tends to choose higher degrees than our algorithm. The off-diagonal blocks are obtained via the block Schur–Parlett recurrence.
- `logm_agm`: an algorithm for the computation of the matrix logarithm based on the arithmetic-geometric mean iteration. In particular, we use the result in [12, Thm. 8.2], which gives the approximation $\log A \approx \log(4/\varepsilon) - (\pi/2)\,\mathrm{AGM}(\varepsilon A)^{-1}$, where $\varepsilon = \sqrt{u}/\|A\|_F$ and $\mathrm{AGM}(A)$ is the arithmetic-geometric mean iteration, which we compute by means of the stable double recursion [12, eqs. (5.2), (5.3)] with stopping criterion $\|P_k - I\|_F \le u\|A\|_F$. We do not implement the optimization in [12, sec. 7], because it is precision dependent and aimed at speed rather than accuracy.
- `logm_pade`: the version of Algorithm 4.1 employing diagonal Padé approximants and relative error bounds, that is, $\psi(X) = \|X - I\|_1$.
- `logm_pade_abs`: the version of Algorithm 4.1 employing diagonal Padé approximants and absolute error bounds, that is, $\psi(X) = 1$.
- `logm_tayl`: the version of Algorithm 4.1 employing truncated Taylor approximants and relative error bounds, that is, $\psi(X) = \|X - I\|_1$.
- `logm_tayl_abs`: the version of Algorithm 4.1 employing truncated Taylor approximants and absolute error bounds, that is, $\psi(X) = 1$.
- `logm_tfree_pade`: the transformation-free Algorithm 5.2 employing diagonal Padé approximants and relative error bounds, that is, $\psi(X) = \|X - I\|_1$.
- `logm_tfree_tayl`: the transformation-free Algorithm 5.2 employing truncated Taylor approximants and relative error bounds, that is, $\psi(X) = \|X - I\|_1$.

- `logm`: the built-in MATLAB function that implements the algorithms for real and complex matrices from [3], [4] and is designed for double precision only.

In the implementations, $m_{\max}$ and $s_{\max}$ are set to 200 and 100, for diagonal Padé approximants, and to 400 and 100, for truncated Taylor series, respectively.

The Gauss–Legendre nodes and weights in (4.3) are computed by means of the `GaussLegendre` method of the `mp` class provided by the Multiprecision Computing Toolbox. This algorithm is based on Newton root-finding [21] and computes the nodes and weights of the quadrature formula of order m in $O(m)$ flops. This is more efficient than the Golub–Welsh algorithm [23], which is based on the computation of the eigensystem of a tridiagonal matrix and cost $O(m^2)$ flops.

We note that version 7.1 of the Symbolic Math Toolbox provides an overloaded version of the MATLAB function `logm`, but for several of our test matrices this function either gives an error or fails to return an answer, so we exclude it from our tests.

We evaluate the forward errors $\|X - \widehat{X}\|_1/\|X\|_1$, where $\widehat{X}$ is a computed solution and $X \approx \log A$ is a reference solution computed with `logm_pade` using $8d$ decimal significant digits, where d is the number of digits used for the computation of $\widehat{X}$.

To gauge the forward stability of the algorithms we plot the quantity $\kappa_{\log}(A)u$, where $\kappa_{\log}(A)$ is the 1-norm condition number of the matrix logarithm of A estimated using `funm_condest1` from the Matrix Function Toolbox [28], with the aid of the Fréchet derivatives in Algorithm 4.1.

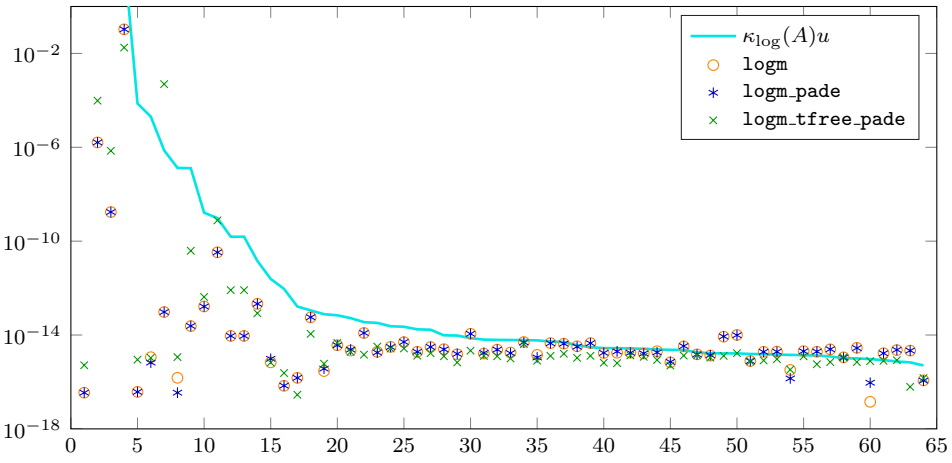
We use a test set of 64 matrices, of sizes ranging from 2×2 to 100×100, including matrices from the literature of the matrix logarithm and from the MATLAB `gallery` function.

6.1. Comparison with `logm` in double precision. Our first experiment compares `logm_pade` and `logm_tfree_pade` running in IEEE double precision ($u = 2^{-53}$) with the built-in MATLAB function `logm`, in order to check that the new algorithms performs well in double precision. Figure 6.1a shows the forward errors sorted by decreasing condition number of the matrix logarithm. Figure 6.1b reports the same data in the form of a performance profile, which we compute with the MATLAB function `perfprof` described in [26, sec. 26.4]. Here, for each method M the height of the line at θ represents the fraction of matrices in the test set for which the relative forward error of M is at most θ times that of the algorithm that delivers the most accurate result. In our performance profiles we use the technique of Dingle and Higham [19] to rescale errors smaller than the unit roundoff in order to avoid abnormally small errors skewing the profiles.

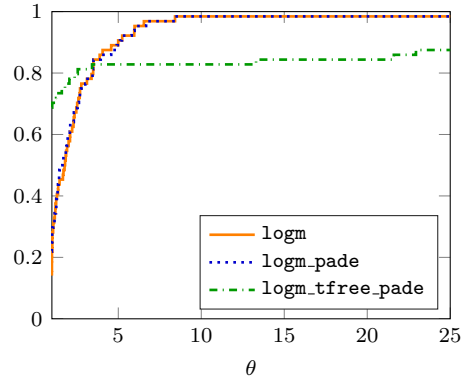
The results show that `logm_pade` and `logm` produce errors bounded approximately by $\kappa_{\log}(A)u$, that is, they behave in a forward stable manner. The same is true of `logm_tfree_pade` except for one matrix, and this algorithm is most often the most accurate, while also being the least reliable, as shown by the performance profile.

Figure 6.1c shows that the computational cost of `logm_pade` can be higher than that of `logm`, but overall is comparable with the state of the art.

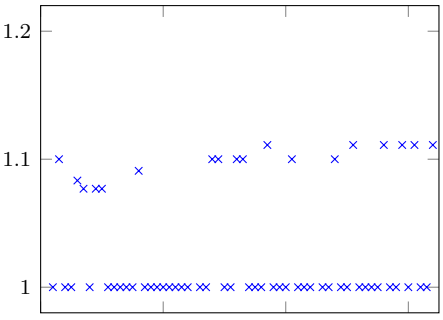
As a further experiment we sought to maximize the ratios between the forward errors of `logm` and `logm_pade`, using the multidirectional search method of Dennis and Torczon [17], implemented in the `mdsmax` function in the Matrix Computation Toolbox [27]. Initializing that method with random 10×10 matrices with no positive real eigenvalues we have not been able to find a matrix for which either ratio of errors exceeds 1.4. This provides further evidence that the two algorithms deliver



(a) Forward errors.



(b) Performance profile for data in (a).



(c) Computational cost.

Figure 6.1: Top: forward errors of `logm_pade`, `logm_tfree_pade`, and `logm` on the test set, all running in IEEE double precision arithmetic. Bottom left: performance profile. Bottom right: on the same test set, the number of square roots and multiple right-hand side linear system solves for `logm_pade` divided by the corresponding number for `logm`.

similar accuracy.

6.2. Relative and absolute error. Now we show the importance of using a relative error bound as opposed to an absolute bound, as was used in earlier algorithms intended for double precision [13], [14], [18], [38]. Figure 6.2 reports how the relative forward error to unit roundoff ratio varies, as the precision increases, for `logm_pade`, `logm_pade_abs`, `logm_tayl` and `logm_tayl_abs`, on three of our test matrices.

The ratio for `logm_pade` and `logm_tayl` is influenced by the conditioning of the problem, which is below 100 for these matrices, but tends to remain stable as the working precision increase. The ratio for the algorithms based on an absolute error bounds, on the other hand, grows exponentially, and we can conclude that `logm_pade_abs` and `logm_tayl_abs` are unstable.

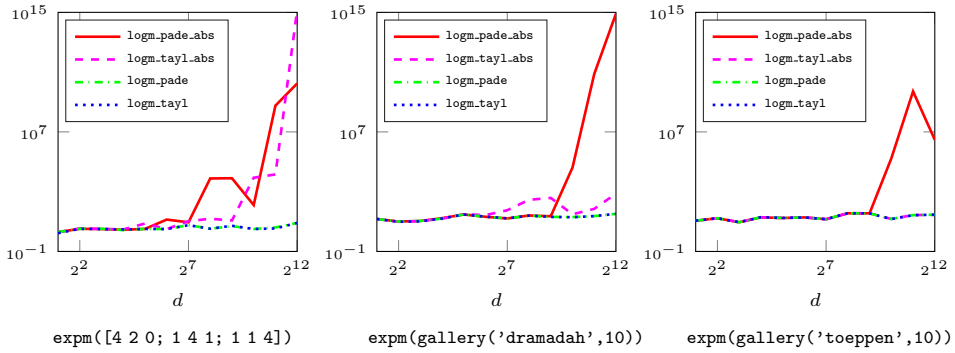


Figure 6.2: Forward error divided by unit roundoff $u = 2^{\lceil \log_2(10^{-d}) \rceil}$, where the number of decimal significant digits d is shown on the x-axis, for three matrices in the test set.

6.3. Experiments at higher precisions. Now we compare the accuracy of Algorithm 4.1, Algorithm 5.2 and several competing methods at four different precisions. Figure 6.3a plots, for the matrices in our test set sorted by decreasing condition number, the relative forward errors of `logm_mct`, `logm_agm`, `logm_tayl`, `logm_tfree_tayl`, `logm_pade`, and `logm_tfree_pade` against $\kappa_{\log}(A)u$. If the forward error of an algorithm falls outside the range reported in the graph, we put the corresponding marker on the nearest edge (top or bottom) of the plot. The right-hand column of Figure 6.3 reports the same data in the form of performance profiles.

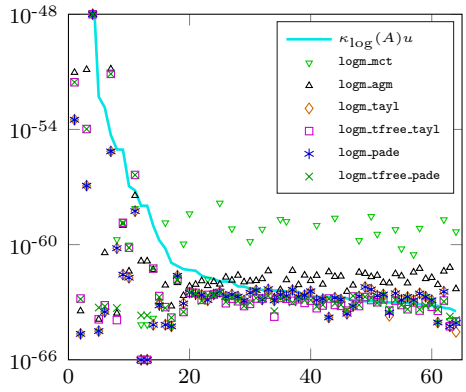
For a working precision of 16 digits (the results for which are not shown here), the six algorithms exhibit a similar behaviour. As illustrated in Figure 6.3c and 6.3e, as the number of significant digits increases, `logm_mct` loses accuracy on almost 40 percent of the matrices, and the accuracy of the solution degrades quickly with respect to $\kappa_{\log}(A)u$. The loss of accuracy of `logm_agm` is not as severe, but it affects the entire dataset and is particularly noticeable for well-conditioned matrices.

The new algorithms show a forward stable behavior, since the forward error remains less than or only slightly larger than $\kappa_{\log}(A)u$ as the working precision increases. On our test set, Algorithm 4.1 is more accurate than Algorithm 5.2, and the performance of `logm_tayl` and `logm_pade` is almost identical whereas `logm_tfree_tayl` is more accurate than `logm_tfree_pade` and often provides the most accurate result on the best-conditioned of the test matrices.

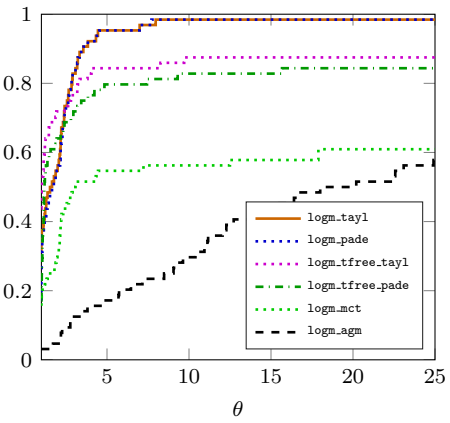
6.4. Code profiling. Table 6.1 compares the execution times of our implementations of `logm_tayl` and `logm_pade`, profiling the four main operations performed by the algorithms: Schur decomposition, square roots, evaluation of the bound to determine the degree of the Padé approximant to be used (which includes computation of the norm estimates used in forming the α_p), and evaluation of the approximant itself. We consider the matrices

```
A = expm(gallery('chebvand', n))
B = expm(gallery('randsvd', n))
C = expm(gallery('chow', n))
```

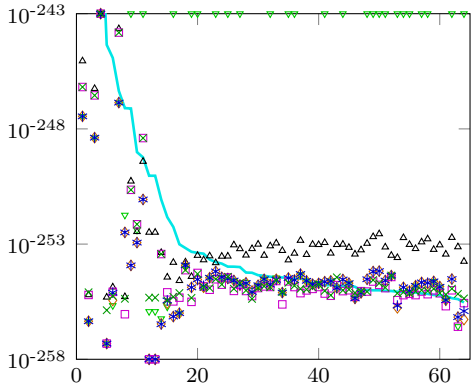
The unit roundoff is 2^{-1701}, which roughly gives 512 decimal significant digits.



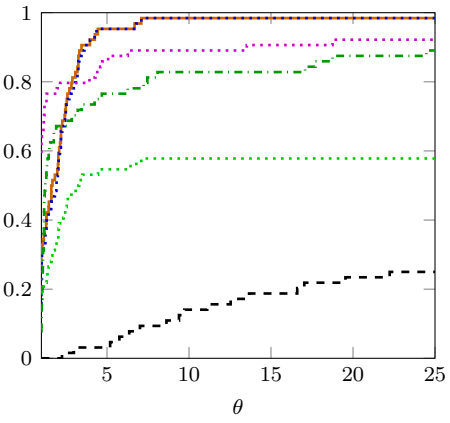
(a) $u = 2^{-213}$ (∼64 digits)



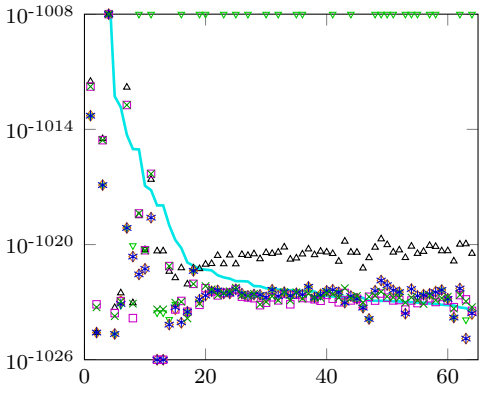
(b) Performance profile for data in (a).



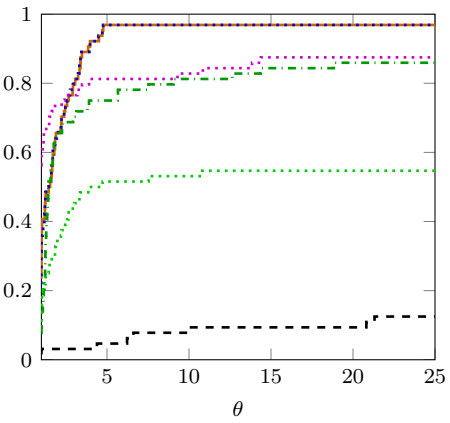
(c) $u = 2^{-851}$ (∼256 digits)



(d) Performance profile for data in (c).



(e) $u = 2^{-426}$ (∼1024 digits)



(f) Performance profile for data in (e).

Figure 6.3: Forward errors for 64, 256, and 1024 digit precisions.

Table 6.1: Execution time breakdown of `logm_tayl` and `logm_pade`, run with $u = 2^{-1701}$ on three matrices of increasing size. The table reports, for each algorithm, the number of square roots (s), the degree of the Padé approximant (m), the total execution time in seconds (T_{tot}), and the percentage of time spent computing the Schur decomposition (T_{sch}), taking the square roots (T_{sqrt}), evaluating the scalar bound (T_{bnd}), and evaluating the Taylor and Padé approximants (T_{eval}).

		`logm_tayl`							`logm_pade`						
	n	s	m	T_{sch}	T_{sqrt}	T_{bnd}	T_{eval}	T_{tot}	s	m	T_{sch}	T_{sqrt}	T_{bnd}	T_{eval}	T_{tot}
`A`	10	41	40	18%	45%	4%	33%	0.4	18	39	9%	47%	11%	32%	0.5
	20	41	40	34%	37%	2%	28%	1.0	18	39	28%	29%	5%	38%	1.2
	50	40	40	53%	32%	0%	15%	9.0	17	40	54%	16%	1%	30%	8.9
	100	41	40	56%	33%	0%	11%	63.0	18	38	59%	15%	0%	25%	59.7
	200	41	40	57%	34%	0%	10%	460.3	20	35	60%	18%	0%	22%	432.1
	500	41	40	41%	47%	0%	12%	5053.2	19	37	46%	24%	0%	31%	4550.7
`B`	10	44	40	20%	48%	5%	27%	0.3	21	40	9%	52%	12%	27%	0.5
	20	45	40	31%	35%	2%	32%	1.1	24	36	25%	31%	15%	29%	1.2
	50	46	40	45%	40%	0%	15%	8.0	24	37	45%	23%	2%	30%	7.9
	100	46	40	48%	40%	0%	12%	56.9	24	38	50%	22%	0%	28%	54.3
	200	47	40	48%	42%	0%	10%	424.3	24	39	51%	23%	0%	26%	400.3
	500	48	40	37%	51%	0%	11%	5300.1	25	39	41%	29%	0%	30%	4837.7
`C`	10	45	40	27%	43%	4%	27%	0.4	24	36	11%	40%	26%	23%	0.6
	20	48	38	37%	34%	1%	28%	1.2	24	37	32%	30%	9%	29%	1.3
	50	48	39	52%	35%	0%	12%	9.5	24	39	53%	20%	1%	26%	9.3
	100	48	40	56%	34%	0%	10%	69.5	26	37	58%	20%	0%	22%	66.1
	200	49	40	55%	36%	0%	9%	510.5	27	37	58%	21%	0%	21%	482.0
	500	54	40	47%	44%	0%	9%	6952.3	31	37	51%	27%	0%	22%	6595.6

In both cases, evaluating the scalar bound (T_{bnd}) is relatively expensive for small matrices, but its impact drops as the size of the matrices increases and it is typically negligible for matrices of size larger than 100. We can see that `logm_tayl` needs approximately twice as many square roots as `logm_pade` on these matrices, and that while the evaluation time (T_{eval}) is larger for `logm_pade` this algorithm is slightly faster in most cases.

7. Conclusions. The state of the art inverse scaling and squaring algorithms for the matrix logarithm and the matrix exponential, implemented in the MATLAB functions `logm` and `expm`, are tuned specifically for double or single precision arithmetic, via the use of pre-computed constants obtained from backward error bounds. This approach does not extend in any convenient way to a multiprecision computing environment. Here we have shown that by using forward error bounds we can obtain algorithms for the matrix logarithm that perform in a forward stable way across a wide range of precisions. The Schur-based algorithms, based on Algorithm 4.1, are competitive with `logm` when run in double precision and are superior to existing algorithms at higher precisions. For computing environments lacking a variable precision Schur decomposition we recommend the transformation-free Algorithm 5.2.

The algorithms rely on three innovations. First, we have derived a new sharper version of the forward error bound of Kenney and Laub [39] that can be much smaller for nonnormal matrices. Second, we have implemented the bound in the form of a relative error bound, as we found that the absolute error bounds used in some previous algorithms yield poor results at high precision due to the need for X in the

approximations to $\log(I + X)$ to have a small norm. Third, we have devised a new strategy for choosing the degree of the approximants and the number of square roots. We investigated both Padé approximants and Taylor approximants and found that there is very little to choose between them in speed or accuracy.

We are currently looking at extending the ideas herein to other matrix functions, including the matrix exponential and real matrix powers.

REFERENCES

[1] Awad H. Al-Mohy. A more accurate Briggs method for the logarithm. *Numer. Algorithms*, 59 (3):393–402, 2012.

[2] Awad H. Al-Mohy and Nicholas J. Higham. A new scaling and squaring algorithm for the matrix exponential. *SIAM J. Matrix Anal. Appl.*, 31(3):970–989, 2009.

[3] Awad H. Al-Mohy and Nicholas J. Higham. Improved inverse scaling and squaring algorithms for the matrix logarithm. *SIAM J. Sci. Comput.*, 34(4):C153–C169, 2012.

[4] Awad H. Al-Mohy, Nicholas J. Higham, and Samuel D. Relton. Computing the Fréchet derivative of the matrix logarithm and estimating the condition number. *SIAM J. Sci. Comput.*, 35(4):C394–C410, 2013.

[5] David H. Bailey, Hida Yozo, Xiaoye S. Li, and Brandon Thompson. ARPREC: An arbitrary precision computation package. Technical report, Lawrence Berkeley National Laboratory, 2002.

[6] George A. Baker, Jr. and Peter Graves-Morris. *Padé Approximants*, volume 59 of *Encyclopedia of Mathematics and Its Applications*. Second edition, Cambridge University Press, Cambridge, UK, 1996. xiv+746 pp.

[7] Richard H. Bartels and George W. Stewart. Algorithm 432: Solution of the matrix equation $AX + XB = C$. *Comm. ACM*, 15(9):820–826, 1972.

[8] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Rev.*, 59(1):65–98, 2017.

[9] Åke Björck and Sven Hammarling. A Schur method for the square root of a matrix. *Linear Algebra Appl.*, 52/53:127–140, 1983.

[10] BOOST C++ libraries. http://www.boost.org.

[11] Henry Briggs. *Arithmetica Logarithmica*. William Jones, London, 1624.

[12] João R. Cardoso and Rui Ralha. Matrix arithmetic-geometric mean and the computation of the logarithm. *SIAM J. Matrix Anal. Appl.*, 37(2):719–743, 2016.

[13] João R. Cardoso and F. Silva Leite. Theoretical and numerical considerations about Padé approximants for the matrix logarithm. *Linear Algebra Appl.*, 330:31–42, 2001.

[14] Sheung Hun Cheng, Nicholas J. Higham, Charles S. Kenney, and Alan J. Laub. Approximating the logarithm of a matrix to specified accuracy. *SIAM J. Matrix Anal. Appl.*, 22(4): 1112–1125, 2001.

[15] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications, 2015. ArXiv preprint 1412.7024v5. 10 pp.

[16] Philip I. Davies and Nicholas J. Higham. A Schur–Parlett algorithm for computing matrix functions. *SIAM J. Matrix Anal. Appl.*, 25(2):464–485, 2003.

[17] John E. Dennis, Jr. and Virginia Torczon. Direct search methods on parallel machines. *SIAM J. Optim.*, 1(4):448–474, 1991.

[18] Luca Dieci, Benedetta Morini, and Alessandra Papini. Computational techniques for real logarithms of matrices. *SIAM J. Matrix Anal. Appl.*, 17(3):570–593, 1996.

[19] Nicholas J. Dingle and Nicholas J. Higham. Reducing the influence of tiny normwise relative errors on performance profiles. *ACM Trans. Math. Software*, 39(4):24:1–24:11, 2013.

[20] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Software*, 33(2):13:1–13:15, 2007.

[21] Andreas Glaser, Xiangtao Liu, and Vladimir Rokhlin. A fast algorithm for the calculation of the roots of special functions. *SIAM J. Sci. Comput.*, 29(4):1420–1438, 2007.

[22] Gene H. Golub, Stephen Nash, and Charles F. Van Loan. A Hessenberg–Schur method for the problem $AX + XB = C$. *IEEE Trans. Automat. Control*, AC-24(6):909–913, 1979.

[23] Gene H. Golub and John H. Welsch. Calculation of gauss quadrature rules. *Mathematics of Computation*, 23(106):221–s10, 1969.

[24] Torbjörn Granlund and the GMP development team. GNU MP: The GNU multiple precision arithmetic library. http://gmplib.org/.

[25] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *JMLR: Workshop and Conference Proceedings*, 2015, pages 1737–1746.

[26] Desmond J. Higham and Nicholas J. Higham. *MATLAB Guide*. Third edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2017. xxvi+476 pp. ISBN 978-1-61197-465-2.

[27] Nicholas J. Higham. The Matrix Computation Toolbox. http://www.maths.manchester.ac.uk/~higham/mctoolbox.

[28] Nicholas J. Higham. The Matrix Function Toolbox. http://www.maths.manchester.ac.uk/~higham/mftoolbox.

[29] Nicholas J. Higham. Evaluating Padé approximants of the matrix logarithm. *SIAM J. Matrix Anal. Appl.*, 22(4):1126–1135, 2001.

[30] Nicholas J. Higham. *Functions of Matrices: Theory and Computation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008. xx+425 pp. ISBN 978-0-898716-46-7.

[31] Nicholas J. Higham and Edvin Deadman. A catalogue of software for matrix functions. Version 2.0. MIMS EPrint 2016.3, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, January 2016. 22 pp. Updated March 2016.

[32] Nicholas J. Higham and Lijing Lin. A Schur–Padé algorithm for fractional powers of a matrix. *SIAM J. Matrix Anal. Appl.*, 32(3):1056–1078, 2011.

[33] Nicholas J. Higham and Françoise Tisseur. A block algorithm for matrix 1-norm estimation, with an application to 1-norm pseudospectra. *SIAM J. Matrix Anal. Appl.*, 21(4):1185–1201, 2000.

[34] Weiming Hu, Haiqiang Zuo, Ou Wu, Yunfei Chen, Zhongfei Zhang, and David Suter. Recognition of adult images, videos, and web page bags. *ACM Trans. Multimedia Comput. Commun. Appl.*, 78:28:1–28:24, 2011.

[35] *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*. Institute of Electrical and Electronics Engineers, New York, 1985. Reprinted in SIGPLAN Notices, 22(2):9–25, 1987.

[36] *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008 (revision of IEEE Std 754-1985)*. IEEE Computer Society, New York, 2008. 58 pp. ISBN 978-0-7381-5752-8.

[37] Fredrik Johansson et al. Mpmath: A Python library for arbitrary-precision floating-point arithmetic. http://mpmath.org.

[38] Charles S. Kenney and Alan J. Laub. Condition estimates for matrix functions. *SIAM J. Matrix Anal. Appl.*, 10(2):191–209, 1989.

[39] Charles S. Kenney and Alan J. Laub. Padé error estimates for the logarithm of a matrix. *Internat. J. Control*, 50(3):707–730, 1989.

[40] Maple. Waterloo Maple Inc., Waterloo, Ontario, Canada. http://www.maplesoft.com.

[41] Mathematica. Wolfram Research, Inc., Champaign, IL, USA. http://www.wolfram.com.

[42] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertik, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. SymPy: Symbolic computing in Python. *PeerJ Computer Science*, 3:e103, 2017.

[43] Multiprecision Computing Toolbox. Advanpix, Tokyo. http://www.advanpix.com.

[44] PARI/GP. http://pari.math.u-bordeaux.fr.

[45] The Sage Developers. Sage Mathematics Software. http://www.sagemath.org.

[46] Symbolic Math Toolbox. The MathWorks, Inc., Natick, MA, USA. http://www.mathworks.co.uk/products/symbolic/.

[47] SymPy Development Team. Sympy: Python library for symbolic mathematics. http://www.sympy.org.