An Arbitrary Precision Scaling and Squaring Algorithm for the Matrix Exponential

Fasi, Massimiliano and Higham, Nicholas J.

2018

MIMS EPrint: **2018.36**

ISSN 1749-9097

AN ARBITRARY PRECISION SCALING AND SQUARING ALGORITHM FOR THE MATRIX EXPONENTIAL[*]

MASSIMILIANO FASI[†] AND NICHOLAS J. HIGHAM[‡]

Abstract. The most popular algorithms for computing the matrix exponential are those based on the scaling and squaring technique. For optimal efficiency these are usually tuned to a particular precision of floating-point arithmetic. We design a new scaling and squaring algorithm that takes the unit roundoff of the arithmetic as input and chooses the algorithmic parameters in order to keep the forward error in the underlying Padé approximation below the unit roundoff. To do so, we derive an explicit expression for all the coefficients in an error expansion for Padé approximants to the exponential and use it to obtain a new bound for the truncation error. We also derive a new technique for selecting the internal parameters used by the algorithm, which at each step decides whether to scale or to increase the degree of the approximant. The algorithm can employ diagonal Padé approximants or Taylor approximants and can be used with a Schur decomposition or in transformation-free form. Our numerical experiments show that the new algorithm performs in a forward stable way for a wide range of precisions and that the most accurate of our implementations, the Taylor-based transformation-free variant, is superior to existing alternatives.

Key words. multiprecision arithmetic, matrix exponential, matrix function, scaling and squaring method, Padé approximation, Taylor approximation, forward error analysis, MATLAB, `expm`

AMS subject classifications. 15A16, 65F60

1. Introduction.

The exponential of a matrix has been the subject of much research in the 150 years or so since Laguerre first defined it [24], thanks to its many applications and in particular its central role in the solution of differential equations. Several equivalent definitions of this matrix function exist [17, Table 10.1], of which perhaps the most well known is the representation via its Taylor series expansion: the exponential of $A \in \mathbb{C}^{n\times n}$ is the matrix

$$\mathrm{e}^A = \sum_{k=0}^{\infty} \frac{A^k}{k!}. \tag{1.1}$$

Since the analogous series expansion of e^z, for $z \in \mathbb{C}$, has an infinite radius of convergence, the power series (1.1) is convergent for any $A \in \mathbb{C}^{n\times n}$ [17, Thm. 4.6], and truncating it to the first few terms gives a crude algorithm for approximating e^A. This method is known to be unsatisfactory—so much so that Moler and Van Loan [29], [30] take it as a lower bound on the performance of any algorithm for computing the matrix exponential.

The most popular method for computing the exponential of a matrix is the scaling and squaring algorithm paired with Padé approximation. This technique, originally proposed by Lawson [25], and further developed and analyzed by various authors over the past half-century, proves remarkably reliable in finite precision arithmetic, but its numerical stability is not fully understood. The method owes its name to the identity

$$\mathrm{e}^A = \left(\mathrm{e}^{2^{-s}A}\right)^{2^s}, \tag{1.2}$$

[*] Version of November 24, 2018. **Funding**: This work was supported by MathWorks, Engineering and Physical Sciences Research Council grant EP/P020720/1, and the Royal Society. 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)

and relies on the approximation

$$\mathrm{e}^A \approx r_{km}(2^{-2}A)^{2^s}, \tag{1.3}$$

where $r_{km}(z)$ is the $[k/m]$ Padé approximant to e^z at 0 and the nonnegative integers k, m, and s are chosen so that $r_{km}(2^{-s}A)$ achieves a prescribed accuracy while minimizing the computational cost of the algorithm. In practice, diagonal approximants $r_m := r_{mm}$ are the most common choice, as symmetries in the coefficients of the numerator and denominator enable an efficient evaluation of $r_m(A)$.

In recent years there has been a sharp rise of interest in multiprecision computation, and the number of programming languages that support arbitrary precision floating-point arithmetic, either natively or through dedicated libraries, is growing. In many cases, a wide range of arbitrary precision linear algebra kernels is available. Numerical routines for the evaluation of matrix functions are also sometimes provided, as we now explain.

The computer algebra systems Maple [26] and Mathematica [27] offer functions that can evaluate in arbitrary precision real matrix powers, the matrix logarithm, the matrix exponential, and a function that computes $f(A)$ given a scalar function f and a square matrix A. The open source computer algebra system Sage [35] supports arbitrary precision floating-point arithmetic, but does not implement any algorithms for the evaluation of matrix functions.

Turning to software focused on floating-point arithmetic, the mpmath library [22] for Python provides functions for evaluating in arbitrary precision a wide range of matrix functions, including real powers, exponential, logarithm, sine, and cosine. MATLAB does not support arbitrary precision floating-point arithmetic natively, but arbitrary precision floating-point data types are provided by the Symbolic Math Toolbox [37] and the Multiprecision Computing Toolbox [31]. Both toolboxes implement algorithms for the matrix square root, the exponential, the logarithm, and general matrix functions, and the Multiprecision Computing Toolbox also includes the hyperbolic and trigonometric sine and cosine of a matrix. Finally, the Julia language [6] supports multiprecision floating-point numbers by means of the built-in data type `BigFloat`, which provides only a few basic linear algebra kernels for arbitrary precision computation, and the `ArbFloats` package, a wrapper to the C library Arb [21] for arbitrary-precision ball arithmetic, which is capable of computing the matrix square root and exponential.

The algorithms underlying the functions described above are not publicly available, to our knowledge. Nor are details of the implementations (albeit embodied in the source code of the open source packages), which in some cases may involve symbolic arithmetic.

The MATLAB function `expm` is a careful implementation of the algorithm of Al-Mohy and Higham [2], which relies on diagonal Padé approximants and exploits precomputed constants θ_m that specify how small the 1-norm of certain powers of a matrix A must be in order for $r_m(A)$ to provide an accurate approximation to e^A in IEEE double precision arithmetic. These constants are obtained by combining a floating-point backward error analysis with a mix of symbolic and high precision computations, and, at the price of a computationally expensive algorithm design stage, provide a very efficient algorithm. For arbitrary precision computations, however, a new approach is required, since this procedure, despite being in principle repeatable for any given precision, is impractical to carry out when the accuracy at which the function should be evaluated is known only at runtime and should hence be treated as an input parameter to the algorithm.

The only published algorithm that we are aware of for computing the matrix exponential in arbitrary precision is that of Caliari and Zivcovich [7], which employs a scaling and squaring algorithm based upon Taylor approximation. It includes a new shifting technique less prone to overflow than the classic approach in [17, sect. 10.7.3] and a novel way to compute at runtime a bound on the backward error of the truncated Taylor series. The underlying backward error analysis relies on an explicit series expansion for the backward error of truncated Taylor series approximants [36] that does not readily extend to general Padé approximants, and the technique used to bound the error relies on a conjecture on the decay rate of the terms of this series expansion.

The goal of this work is to develop an algorithm for evaluating the exponential of a matrix in arbitrary precision floating-point arithmetic that can be used with diagonal Padé approximants or Taylor approximants and is fully rigorous. We wish to avoid symbolic computation and we are particularly interested in precisions higher than double. This work complements that for the matrix logarithm in [12].

The broad need for arbitrary precision matrix functions is clear from their inclusion in the software mentioned above. The need to compute the matrix exponential to high precision is needed, for example, in order to compute accurate solutions to the burnup equations in nuclear engineering [34]. Our particular interest in the matrix exponential stems not only from its many applications but also from algorithm development. Estimating the forward error of algorithms for matrix functions requires a reference solution computed in higher precision, and an arbitrary precision algorithm for the matrix exponential can be used not only for the exponential itself, but also for trigonometric [1], [4], [17, Ch. 12], [19] and hyperbolic [8] matrix functions, as these can be expressed in terms of the exponential. Furthermore, such an algorithm allows us to estimate the backward error of algorithms for evaluating matrix functions defined implicitly by equations involving the exponential, such as the logarithm [3], [12], the Lambert W function [13], and inverse trigonometric and hyperbolic functions [5].

After giving some notation and background material in the next section, we derive in section 3 a new bound on the forward error of Padé approximants to the matrix exponential. We also make a conjecture that, if true, would lead to a more cheaply computable error bound. In section 4 we develop a novel algorithm for evaluating the exponential of a matrix in arbitrary precision. In section 5 we test experimentally several versions of this algorithm and compare their performance with that of existing algorithms. In section 6 we summarize our findings and discuss future lines of research.

2. Notation and background.

We denote by $\mathbb{R}_+ = \{x \in \mathbb{R} : x \geqslant 0\}$ the set of nonnegative real numbers, by $\mathbb{N}$ the set of nonnegative integers, and by $\|\cdot\|$ any consistent matrix norm. The spectrum of a square matrix $A \in \mathbb{C}^{n\times n}$ is denoted by $\sigma(A)$, its spectral radius by $\rho(A) = \max\{|\lambda| : \lambda \in \sigma(A)\}$, and the unit roundoff of floating-point arithmetic by u. Given $f : \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$ and $A \in \mathbb{C}^{n\times n}$, we measure the sensitivity of $f(A)$ by means of the relative condition number

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

which is given explicitly by [17, Thm. 3.1]

$$\kappa_f(A) = \frac{\|D_f(A)\|\|A\|}{\|f(A)\|}, \tag{2.1}$$

where $D_f : \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$ is the Fréchet derivative of f at A, which is the unique linear functional that, for all $E \in \mathbb{C}^{n\times n}$, satisfies $f(A+E) = f(A)+D_f(A)[E]+o(\|E\|)$.

3. Padé approximation of the matrix exponential. Let f be a complex function analytic at 0, and let $k, m \in \mathbb{N}$. The rational function $r_{km}(z) = p_{km}(z)/q_{km}(z)$ is the $[k/m]$ Padé approximant of f at 0 if $p_{km}(z)$ and $q_{km}(z)$ are polynomials of degree at most k and m, respectively, the denominator is normalized so that $q_{km}(0) = 1$, and $f(z) - r_{km}(z) = O(z^{k+m+1})$.

The numerator and denominator of the $[k/m]$ Padé approximant to the exponential at 0 are [14, Thm. 5.9.1]

$$\begin{aligned} p_{km}(z) &= \sum_{j=0}^{k} \binom{k}{j} \frac{(k+m-j)!}{(k+m)!} z^j =: \sum_{j=0}^{k} \beta_j^{[k/m]} z^j, \\ q_{km}(z) &= \sum_{j=0}^{m} (-1)^j \binom{m}{j} \frac{(k+m-j)!}{(k+m)!} z^j =: \sum_{j=0}^{k} \delta_j^{[k/m]} z^j. \end{aligned} \tag{3.1}$$

In our algorithm, we will approximate e^A by means of the rational matrix function $r_{km}(A) = q_{km}(A)^{-1} p_{km}(A)$, which we evaluate by first computing $P = p_{km}(A)$ and $Q = q_{km}(A)$ and then solving a multiple right-hand side linear system in order to obtain $X := Q^{-1}P$. The computational efficiency of this method depends entirely on the evaluation scheme chosen to compute P and Q. In the literature, the customary choice is the the Paterson–Stockmeyer method [33], which we now briefly recall.

Let us rewrite the polynomial $p(X) = \sum_{i=0}^{k} \alpha_i X^i$ as

$$p(X) = \sum_{i=0}^{\mu} B_i(X) X^{\nu i}, \tag{3.2}$$

where $\nu \leqslant k$ is a positive integer, $\mu = \lfloor k/\nu \rfloor$, and

$$B_i(X) = \begin{cases} \alpha_{\nu i+\nu-1} X^{\nu-1} + \cdots + \alpha_{\nu i+1} X + \alpha_{\nu i} I, & i = 0, \ldots, \mu - 1, \\ \alpha_k X^{k-\nu\mu} + \cdots + \alpha_{\nu\mu+1} X + \alpha_{\nu\mu} I, & i = \mu. \end{cases}$$

If we use Horner's method with (3.2), then the number of matrix multiplications required to evaluate $p(X)$ is

$$C_\nu^p(k) = \nu + \mu - 1 - \eta(\nu, k), \qquad \eta(x, y) = \begin{cases} 1, & \text{if } x \text{ divides } y, \\ 0, & \text{otherwise,} \end{cases} \tag{3.3}$$

which is approximately minimized by taking either $\nu = \lfloor \sqrt{k} \rfloor$ or $\nu = \lceil \sqrt{k} \rceil$. Therefore evaluating $r_{km}(X)$ requires, in general,

$$C_\nu^r(k) = \nu_k + \nu_m + \left\lfloor \frac{k}{\nu_k} \right\rfloor + \left\lfloor \frac{m}{\nu_m} \right\rfloor - 2 - \eta(\nu_k, k) - \eta(\nu_m, m) \tag{3.4}$$

matrix multiplications, where ν_ℓ denotes $\sqrt{\ell}$ rounded to the nearest integer. This cost can be considerably reduced for diagonal Padé approximants (for which $k = m$) by exploiting the identity $p_m(X) = q_m(-X)$, where $p_m := p_{mm}$ and $q_m := q_{mm}$. By rewriting the numerator as

$$p_m(A) = \sum_{i=0}^{m} \beta_i A^i = \sum_{i=0}^{\lfloor m/2 \rfloor} \beta_{2i} A^{2i} + A \sum_{i=0}^{\lceil m/2-1 \rceil} \beta_{2i+1} A^{2i} =: U_e + AV =: U_e + U_o,$$

where U_o and U_e are the sums of the monomials with even and odd powers, respectively, we obtain that $q_m(A) = U_e - U_o$. By using ν stages of the Paterson–Stockmeyer method on A^2, computing U_e and U_o requires one matrix product to form A^2 and $\nu - 1$ matrix multiplications to compute the first ν powers of A^2; evaluating the polynomials U_e and V require $\lfloor \lfloor m/2 \rfloor / \nu \rfloor - \eta(\nu, \lfloor m/2 \rfloor)$ and $\lfloor \lfloor (m-1)/2 \rfloor / \nu \rfloor - \eta(\nu, \lfloor (m-1)/2 \rfloor)$ matrix multiplications, respectively; and computing U_o requires one additional multiplication by A. Therefore evaluating both $p_m(A)$ and $q_m(A)$ requires

$$C_\nu^e(m) = \nu + 1 + \left\lfloor \frac{\lfloor m/2 \rfloor}{\nu} \right\rfloor + \left\lfloor \frac{\lfloor (m-1)/2 \rfloor}{\nu} \right\rfloor - \eta(\nu, \lfloor m/2 \rfloor) - \eta(\nu, \lfloor (m-1)/2 \rfloor)$$

matrix multiplications, and it can be shown that $C_\nu^e(m)$ is approximately minimized by taking either $\nu = \lfloor \sqrt{m - 1/2} \rfloor$ or $\nu = \lceil \sqrt{m - 1/2} \rceil$. For m between 1 and 21, we have that $\min \left\{ C_{\lfloor \sqrt{m} \rfloor}(m), C_{\lceil \sqrt{m} \rceil}(m) \right\} = \pi_m$, where the π_m are tabulated in [17, Table 10.3].

In principle, when designing an algorithm based on Padé approximation, one could use approximants of any order but, for any given cost, it is worth considering only the approximant that will deliver the most accurate result. By definition, this will be that of highest order, thus if evaluating the approximant of order m requires $C(m)$ matrix multiplications, an algorithm will typically examine only approximants of *optimal* order

$$m' = \operatorname*{argmax}_{m \in \mathbb{N}} \{ C(m) = \zeta \},$$

for some $\zeta \in \mathbb{N}$. For truncated Taylor series and diagonal Padé approximants to the exponential, the sequences of optimal orders are [11, eqs. (2.7) and (4.6)]

$$a_i = \left\lfloor \frac{(i+2)^2}{4} \right\rfloor, \qquad i \in \mathbb{N}, \tag{3.5}$$

and

$$\begin{aligned} & b_0 = 1, \quad b_1 = 2, \\ & b_i = 2 \left\lceil \frac{i-1}{4} \right\rceil \left(i - 3 \left\lfloor \frac{i-1}{4} \right\rfloor \right) + 1, \qquad i \in \mathbb{N} \backslash \{0, 1\}, \end{aligned} \tag{3.6}$$

respectively. Note that, for diagonal Padé approximants to the exponential, all optimal orders but b_1 are odd.

For a thorough discussion of the effect of rounding errors on the evaluation of matrix polynomials using the scheme (3.2) see [17, sect. 4.2].

3.1. Forward error. In this section we present a new upper bound on the norm of the forward error of $r_{km}(A)$ as an approximation to e^A. In section 4, this bound will play a central role in the design of a scaling and squaring algorithm for computing the matrix exponential in arbitrary precision. The leading term of the truncation error of the $[k/m]$ Padé approximant is known [14, Thm. 5.9.1], since

$$\mathrm{e}^z - r_{km}(z) = c_{km}^1 z^{k+m+1} + O(z^{k+m+2}), \tag{3.7}$$

where

$$c_{km}^1 = (-1)^m \frac{k!m!}{(k+m)!(k+m+1)!}. \tag{3.8}$$

We begin by obtaining all the terms in the series expansion of $q_{km}(z)\mathrm{e}^z - p_{km}(z)$, which is closely related the truncation error in (3.7).

Lemma 3.1. *Let $r_{km}(z) = p_{km}(z)/q_{km}(z)$ be the $[k/m]$ Padé approximant to* e^z *at* 0*. Then for all $z \in \mathbb{C}$,*

$$q_{km}(z)\mathrm{e}^z - p_{km}(z) = \sum_{i=1}^{\infty} c_{k,m}^{i} z^{k+m+i}, \tag{3.9}$$

where

$$c_{k,m}^{i} = \frac{(-1)^m k!}{(k+m)!} \frac{(m+i-1)!}{(i-1)!(k+m+i)!}. \tag{3.10}$$

Proof. By equating the coefficients of z^{k+m+i} on the left- and right-hand sides of (3.9), we obtain that $c_{k,m}^{i}$ is the sum, for j from 0 to m, of the jth coefficient of $q_{km}(z)$ multiplied by the $(k+m+i-j)$th coefficient of the series expansion of e^z:

$$\begin{aligned} c_{k,m}^{i} &= \sum_{j=0}^{m} \delta_j^{[k/m]} \frac{1}{(k+m+i-j)!} - 0 \\ &= \frac{1}{(k+m)!} \sum_{j=0}^{m} (-1)^j \binom{m}{j} \frac{(k+m-j)!}{(k+m+i-j)!}. \end{aligned}$$

We prove (3.10) by induction on m. For $m = 1$ we have

$$\begin{aligned} c_{k,m}^{i} &= \frac{1}{(k+1)!} \left(\binom{1}{0} \frac{(k+1)!}{(k+1+i)!} - \binom{1}{1} \frac{k!}{(k+i)!} \right) \\ &= \frac{k!}{(k+1)!(k+1+i)!} (k+1-k-1-i) = \frac{(-1)^1 k! i!}{(k+1)!(k+i)!(i-1)!}. \end{aligned}$$

By exploiting the identity $\binom{a+1}{b} = \binom{a}{b} + \binom{a}{b-1}$, for the inductive step we have

$$\begin{aligned} c_{k,m+1}^{i} &= \frac{1}{(k+m+1)!} \sum_{j=0}^{m+1} (-1)^j \binom{m+1}{j} \frac{(k+m+1-j)!}{(k+m+1+i-j)!} \\ &= \frac{1}{(k+m+1)!} \left(\frac{(k+m+1)!}{(k+m+1+i)!} + \sum_{j=1}^{m} (-1)^j \binom{m}{j} \frac{(k+m+1-j)!}{(k+m+1+i-j)!} \right. \\ &\quad \left. + \sum_{j=1}^{m} (-1)^j \binom{m}{j-1} \frac{(k+m+1-j)!}{(k+m+1+i-j)!} + (-1)^{m+1} \frac{k!}{(k+i)!} \right) \\ &= \frac{1}{(k+m+1)!} \sum_{j=0}^{m} (-1)^j \binom{m}{j} \left(\frac{((k+1)+m-j)!}{((k+1)+m+i-j)!} - \frac{(k+m-j)!}{(k+m+i-j)!} \right) \\ &= \frac{1}{(k+m+1)!} \left(\frac{(-1)^m (k+1)!(m+i-1)!}{(i-1)!(m+k+i+1)!} - \frac{(-1)^m k!(m+i-1)!}{(i-1)!(m+k+i)!} \right) \\ &= \frac{(-1)^m k!(m+i-1)!}{(k+m+1)!(i-1)!(m+k+i+1)!} (k+1-m-k-i-1) \\ &= \frac{(-1)^{m+1} k!((m+1)+i-1)!}{(k+(m+1))!(i-1)!((m+1)+k+i)!}. \end{aligned}$$

□

This result can be exploited to bound the truncation error of $r_{km}(A)$. We will use the result in [2, Thm. 4.2(a)]: if $f(x) = \sum_{i=\ell}^{\infty} c_i x^i$ and c_i has the same sign for all $i \geqslant \ell$, then for any X such that

$$\alpha_d(X) := \min\{\|X^d\|^{1/d}, \|X^{d+1}\|^{1/(d+1)}\}, \quad d(d-1) \leqslant \ell, \tag{3.11}$$

is less than the radius of convergence of the series, we have

$$\|f(X)\| \leqslant \sum_{i=\ell}^{\infty} |c_i| \alpha_d(X)^i = \left| \sum_{i=\ell}^{\infty} c_i \alpha_d(X)^i \right| = |f(\alpha_d(X))|\,. \tag{3.12}$$

An alternative definition of $\alpha_d(X)$ has been recently proposed for the computation of the wave-kernel matrix functions [32]. This more refined strategy requires the computation of $\|A^{d_i}\|_1$ for all d_i, d_j such that $\gcd(d_i, d_j) = 1$ and $d_i d_j - d_i - d_j < k+m$. The cost of finding all such pairs is difficult to determine, but it must be at least $O\big((k+m)\log(k+m)\big)$ operations, as there are at least $O\big((k+m)^2\big)$ pairs to test and Euclid's algorithm for finding the greatest common divisor of two integers $a, b \in \mathbb{N}$ such that $a < b$ requires $2\log_2 a + 1$ operations in the worst case. Moreover, even if all the pairs to be tested were known, the cost of evaluating the bound would increase with k and m, and as both can be potentially large when resorting to high precision, we prefer to use the cheaper bound given by (3.11). However, all the results in this section can be modified by replacing $\alpha_d(X)$ with

$$\alpha^{[k/m]}(X) := \min_{\substack{\gcd(a,b) \\ ab-a-b<k+m}} \max\left\{\|X^a\|^{1/a}, \|X^b\|^{1/b}\right\}.$$

For the truncation error of the $[k/m]$ Padé approximant to the matrix exponential, we would like to obtain a bound of the form

$$\|\mathrm{e}^X - r_{km}(X)\| \leqslant |\mathrm{e}^{\alpha_d(X)} - r_{km}(\alpha_d(X))|, \tag{3.13}$$

which would be true if all the nonzero terms in the series expansion at 0 of $\mathrm{e}^z - r_{km}(z)$ had the same sign. By Lemma 3.1, this would be true if $q_{km}(z)^{-1}$ had a power series expansion with all coefficients of the same sign. This applies to Taylor approximants $t_k := r_{k0}$, since $q_{km}(z)^{-1} = 1$, and by (3.12) we can derive the bound

$$\|\mathrm{e}^x - t_k(x)\| = \left\| \sum_{i=k+1}^{\infty} \frac{1}{i!} x^i \right\| \leqslant \left\| \sum_{i=k+1}^{\infty} \frac{\alpha_d(X)^i}{i!} \right\| = \left| \mathrm{e}^{\alpha_d(X)} - t_k(\alpha_d(X)) \right|.$$

For all other even values of m that we have checked, this is not the case. For example, the series expansion for the reciprocal of the denominator of the $[2/2]$ Padé approximant is

$$\frac{1}{q_{22}(z)} = 1 + \frac{z}{2} + \frac{z^2}{6} + \frac{z^3}{24} + \frac{z^4}{144} - \frac{z^6}{1728} + O\left(z^7\right).$$

However, for the algorithm we are designing we are interested only in bounding the forward error of diagonal approximants of optimal degree, and from (3.6) we know that most optimal degrees are, in fact, odd. The evidence suggests that, in this case, the coefficients of the series expansion are indeed one-signed, and we make a conjecture.

CONJECTURE 3.2. *For odd m, all the coefficients of the series expansion of $q_{km}(z)^{-1}$ at 0 are positive.*

If this conjecture were true, then we could use the bound (3.13) for all diagonal approximants of degree b_i in (3.6) for $i \in \mathbb{N} \backslash \{1\}$, but since we do not have a proof of the conjecture we will bound the truncation error of r_{km} for any k and m. We will use the next result, which combines Lemma 3.1 and (3.12).

COROLLARY 3.3 (bound on the truncation error of Padé approximants). *Let $r_{km}(z) = p_{km}(z)/q_{km}(z)$ be the $[k/m]$ Padé approximant to e^z at 0. Then for $X \in \mathbb{C}^{n\times n}$ and any positive integer d such that $d(d-1) \leqslant k+m+1$,*

$$\left\|\mathrm{e}^X - r_{km}(X)\right\| \leqslant \left\|q_{km}(X)^{-1}\right\| \left|q_{km}(\alpha_d(X))\mathrm{e}^{\alpha_d(X)} - p_{km}(\alpha_d(X))\right| \tag{3.14}$$

Proof. By (3.12), we have

$$\begin{aligned}\left\|\mathrm{e}^X - r_{km}(X)\right\| &= \left\|q_{km}(X)^{-1}\big(q_{km}(X)\mathrm{e}^X - p_{km}(X)\big)\right\| \\ &\leqslant \left\|q_{km}(X)^{-1}\right\| \left\|q_{km}(X)\mathrm{e}^X - p_{km}(X)\right\| \\ &\leqslant \left\|q_{km}(X)^{-1}\right\| \left|q_{km}(\alpha_d(X))\mathrm{e}^{\alpha_d(X)} - p_{km}(\alpha_d(X))\right|,\end{aligned}$$

where for the last inequality we used the fact that the coefficients of the series expansion of $q_{km}(z)\mathrm{e}^z - p_{km}(z)$ all have the same sign, by Lemma 3.1. ∎

Since the norm of $q_{km}^{-1}(X)$ does not depend on $\alpha_d(X)$, the bound in (3.14) is nondecreasing in $\alpha_d(X)$ and therefore is minimized by choosing for d the value

$$d^\star = \underset{1\leqslant d\leqslant d^{[k/m]}}{\operatorname{argmin}} \alpha_d(X), \tag{3.15}$$

where

$$d^{[k/m]} = \max\{d \in \mathbb{N} : d(d-1) \leqslant k+m+1\} = \left\lfloor \frac{1+\sqrt{5+4(k+m+1)}}{2} \right\rfloor. \tag{3.16}$$

Depending on the size of k and m, this choice might require the estimation of $\alpha_d(X)$ for too many values of d, and thus be unpractical. On the other hand, it has been observed [2] that the sequence $(\alpha_d(X))_{d\in\mathbb{N}}$ is typically roughly decreasing, so it is reasonable to use the considerably cheaper approximation $\alpha_{d^{[k/m]}}(X)$. In our algorithm, we adopt an intermediate approach that has the same cost as the computation of $\alpha_{d^{[k/m]}}(X)$, but improves on it by reusing previously computed quantities. We discuss this in detail in section 4.

4. A multiprecision scaling and squaring algorithm. In this section we develop a novel scaling and squaring method for computing the matrix exponential in arbitrary precision floating-point arithmetic. Our algorithm differs from traditional scaling and squaring approaches, such as those of Al-Mohy and Higham [2] and Caliari and Zivcovich [7], in several respects. First, it relies on a bound on the forward error rather than on the backward error of the Padé approximants to the matrix exponential and avoids the use of any precomputed precision-dependent constants by evaluating at runtime the bound (3.14) for some choice of d. Moreover, unlike scaling and squaring algorithms for double precision based on diagonal Padé approximants [2], [16], [17, Alg. 10.20], [18], which use approximants of order at most 13 and a nonzero scaling

Algorithm 4.1: Scaling and squaring algorithm for the matrix exponential.

Given $A \in \mathbb{C}^{n\times n}$, this algorithm computes an approximation to e^A in floating-point arithmetic with unit roundoff u using a scaling and squaring method based upon Padé approximants. The pseudocode of evalBoundDiag and evalPadeDiag is given in Fragment 4.3, that of evalBoundTayl and evalPadeTayl in Fragment 4.5, and that of recompDiags in Fragment 4.2.

```
$\mathcal{A}_0 \leftarrow I$
if use_taylor then
    evalBound ← evalBoundTayl
    evalPade ← evalPadeTayl
    $\mathcal{A}_1 \leftarrow A$
else
    evalBound ← evalBoundDiag
    evalPade ← evalPadeDiag
    $\mathcal{A}_1 \leftarrow A^2$
$s \leftarrow 0$
$i \leftarrow 0$
$\gamma \leftarrow [-\infty, -\infty, \dots]$
$\alpha_{\min} \leftarrow \infty$
$\delta_{old} \leftarrow \infty$
$[\delta, \psi, \kappa_A] \leftarrow$ evalBound($A, \mathtt{m}_i, s$)
while $\delta \geqslant u\psi$ and $s < s_{\max}$ and $i \leqslant N$ do
    if $\kappa_A \geqslant \zeta(u)$ or $\delta_{old} < \delta^2$ then
        $s \leftarrow s + 1$
    else
        $i \leftarrow i + 1$
    $\delta_{old} \leftarrow \delta$
    $[\delta, \psi, \kappa_A] \leftarrow$ evalBound($A, \mathtt{m}_i, s$)
$Y \leftarrow$ evalPade($A, m, s$)
if isQuasiUpperTriangular($A$) then
    $Y \leftarrow$ recompDiags($2^{-s}A, Y$)
for $t \leftarrow 1$ to $s$ do
    $Y \leftarrow Y^2$
    if isQuasiUpperTriangular($A$) then
        $Y \leftarrow$ recompDiags($2^{-s+t}A, Y$)
return $X$
```

parameter only if the approximant of highest degree is expected not to deliver either a truncation error smaller than u or an accurate evaluation of $r_{13}(A)$, our algorithm blends the two phases together and tries to determine both parameters at the same time.

Our arbitrary precision scaling and squaring algorithm for the computation of e^A is given in Algorithm 4.1. Besides the matrix $A \in \mathbb{C}^{n\times n}$, the algorithm accepts several additional input arguments.

- The arbitrary precision floating point parameter $u \in \mathbb{R}_+$ specifies the unit roundoff of the working precision of the algorithm.

- The Boolean parameter `use_taylor` specifies the kind of Padé approximants the algorithm will use: truncated Taylor series if set to **true**, diagonal Padé approximants otherwise.
- The parameter $u_{bnd} \in \mathbb{R}_+$ specifies the unit roundoff of the precision used to evaluate $\|q_{m_i}(2^{-s}A)^{-1}\|_1$ in (4.1) below. The value of u_{bnd} is ignored if `use_taylor` is set to **true**.
- The vector $\mathtt{m} \in \mathbb{N}^N$, sorted in ascending order, specifies what orders of Padé approximants the algorithm can consider. The algorithm will select i between 1 and N, and then evaluate either the truncated Taylor series of order $\mathtt{m}_i$ or the $[\mathtt{m}_i/\mathtt{m}_i]$ Padé approximant, depending on the value of `use_taylor`.
- The nonnegative integer $s_{\max}$ specifies the maximum number of binary powerings the algorithm is allowed to compute during the squaring stage, or, equivalently, the maximum number of times the matrix can be multiplied by $\frac{1}{2}$ during the initial scaling stage.
- The function parameter $\zeta : \mathbb{R}_+ \to \mathbb{R}_+$ specifies a precision-dependent value that is used to predict whether the evaluation of $q_{\mathtt{m}_i}(A)$ will be accurate or not. This parameter is not used when `use_taylor` is set to **true**.

We now discuss the outline of Algorithm 4.1. The variables $\mathcal{A}$, γ, and $\alpha_{\min}$ are assumed to be available within the following code fragments (that is, their scope is global). The Boolean variable `use_taylor` chooses between the auxiliary functions in Fragments 4.3 and 4.5, tailored to the case of diagonal Padé approximants and truncated Taylor series, respectively. Finally, we use the notation $[x, x, \dots]$, to denote a vector whose elements are all initialized to x and whose length is unimportant. We assume that very few of its entries will take a value different from the default, and thus that such a vector can be stored in a memory-efficient way.

The algorithm starts by determining a suitable order and a scaling parameter s for the scaling and squaring method. To this end, on line 10–11 it sets s and i to 0 and then increments them, trying to find a choice for which the right-hand side of (3.14), for $X = 2^{-s}A$, $k = \mathtt{m}_i$, and $m = k$ or $m = 0$, is smaller than $u\psi(2^{-s}A)$, where $\psi(X)$ approximates $\left\|\mathrm{e}^X\right\|_1$. As long as this condition is not satisfied, two heuristics are used to decide which parameter it is more convenient to change. One approach is aimed at keeping the evaluation of the Padé approximant as accurate as possible, by taking into account the conditioning of $q_{\mathtt{m}_i}$; being specific to diagonal approximants this is discussed in section 4.1. On the other hand, we noticed that when $\alpha_d(2^{-s}A) \gg 1$, the bound (3.14) can sometimes decrease exceedingly slowly as m increases, leading to the use of an approximant of degree much larger than needed, which in turn causes loss of accuracy and unnecessary computation. We found that monitoring the rate at which our bound on the truncation error of the approximant decreases provides an effective strategy to prevent this from happening. In particular, we increment s when the bound on the truncation error does not decrease at least quadratically, that is, when $\delta_{old} < \delta^2$, where δ_{old} and δ are the values of the error bound at the previous and current iteration of the while loop on line 16 of Algorithm 4.1, respectively.

As soon as a combination of scaling parameter and Padé approximant order is found, the algorithm computes Y by evaluating the Padé approximant (diagonal or Taylor) of order $\mathtt{m}_i$ at $2^{-s}A$, and finally computes $\mathrm{e}^A \approx Y^{2^s}$, by applying s steps of binary powering to Y. If A is upper quasi-triangular, in order to improve the accuracy of the final result, the function RECOMPDIAGS in Fragment 4.2 is used to recompute the diagonal and first upperdiagonal of the intermediate matrices from the elements of A, as recommended by Al-Mohy and Higham [2].

Fragment 4.2: Recomputation of the diagonals.

```
function RECOMPDIAGS($A \in \mathbb{C}^{n\times n}, X \in \mathbb{C}^{n\times n}$)
▻ Compute main diagonal and first upper-diagonal of $X \approx \mathrm{e}^A$ from $A$.
for $i = 1$ to $n$ do
    if $i = n-1$ or $i \leqslant n-2$ and $a_{i+2,i+1} = 0$ then
        if $a_{i+1,i} = 0$ then
            Recompute $x_{i,i}, x_{i,i+1}, x_{i+1,i+1}$ using [17, Eq. (10.42)].
        else
            Recompute $x_{i,i}, x_{i,i+1}, x_{i+1,i}, x_{i+1,i+1}$ using [2, Eq. (2.2)].
        $i \leftarrow i+1$
    else
        $x_{i,i} \leftarrow \mathrm{e}^{a_{i,i}}$
```

In the next two sections, we discuss how the functions EVALBOUND and EVALPADE can be implemented efficiently for diagonal Padé approximants and truncated Taylor series.

4.1. Diagonal Padé approximants. When `use_taylor` is set to **false**, that is, when diagonal Padé approximants are being considered, the condition that needs to be tested on lines 15 and 22 of Algorithm 4.1 is, for some d such that $d(d-1) < 2\,\mathtt{m}_i+1$,

$$\left\|q_{\mathtt{m}_i}(2^{-s}A)^{-1}\right\|_1 \left|q_{\mathtt{m}_i}(\alpha_d(2^{-s}A))\mathrm{e}^{\alpha_d(2^{-s}A)} - p_{\mathtt{m}_i}(\alpha_d(2^{-s}A))\right| < u\,\psi(2^{-s}A), \tag{4.1}$$

As discussed in section 3, the choice of $\alpha_d(X)$ that would guarantee the best bound is $\alpha_{d^\star}^{[\mathtt{m}_i/\mathtt{m}_i]}(X)$, for $d^\star$ in (3.15), but this value can become impractical to compute, even for Padé approximants of relatively low degree. Taking $\alpha_{d^{[\mathtt{m}_i/\mathtt{m}_i]}}(X)$, where $d^{[\mathtt{m}_i/\mathtt{m}_i]}$ is defined in (3.16), on the other hand, is appealing because this estimate requires the evaluation of $\|A^d\|_1^{1/d}$ for at most two values of d independently of the value of $\mathtt{m}_i$, and is often not far from the best choice, since the sequence $\big(\|A^d\|^{1/d}\big)_{d\in\mathbb{N}}$ is typically roughly decreasing [2].

However, it is sometimes possible to obtain a better bound at almost no extra cost, by reusing quantities computed during previous steps of the algorithm. Observe that, since $\|(2^{-s}A)^d\|^{1/d} = 2^{-s}\|A^d\|^{1/d}$ and thus $\alpha_d(2^{-s}A) = 2^{-s}\alpha_d(A)$, it is enough to estimate the norm of powers of A and then scale their value as required. Moreover, since the algorithm considers the approximants in nondecreasing order of cost, the value of $d^{[\mathtt{m}_i/\mathtt{m}_i]}$ is nondecreasing in i. Therefore, in (4.1) we can replace $\alpha_d(2^{-2}A)$ by $2^{-s}\alpha_{\min}$, where $\alpha_{\min}$ is a variable that keeps track of the smallest value of $\alpha_{d^{[\mathtt{m}_i/\mathtt{m}_i]}}(X)$ computed so far, and is updated only when a new value $\alpha_{d^{[\mathtt{m}_j/\mathtt{m}_j]}}(X) < \alpha_{\min}$ is found for some $j > i$.

Since only the order of magnitude of $\alpha_{\min}$ is actually needed, we estimate $\|A^d\|_1$ by running in precision u_{bnd} the 1-norm estimation algorithm of Higham and Tisseur [20]. This method repeatedly computes the action of A on a tall and skinny matrix without explicitly forming any powers of A, and thus requires only $O(n^2)$ flops. In the pseudocode, the 1-norm estimation is performed by the function NORMEST1, whose only input is a function that computes the product AX given the matrix $X \in \mathbb{C}^{n\times t}$. In order to keep the notation as succinct as possible, anonymous functions are specified using a lambda notation, and $\lambda x.f(x)$ denotes a function that replaces all the occurrences of x in the body of f with the value of its input argument.

By storing the values of $\|A^d\|_1$ in the global array γ, the 1-norm of each power of A is estimated at most once. Further computational savings can be achieved by computing some carefully chosen powers of A within the algorithm and using them to evaluate of the action of powers of A on a vector, as we will discuss later.

As long as the bound (4.1) is not satisfied, the algorithm can decide to either increment the scaling factor or increase the order of the Padé approximant, since either choice will reduce the truncation error of the approximation. Both options, however, may have an adverse effect on the numerical behavior of the algorithm, since taking a Padé approximant of higher degree may significantly increase the conditioning of the coefficient of the linear system to be solved, thus jeopardizing the accurate evaluation of the approximant, whereas increasing s will increase the number of matrix multiplications that will occur during the squaring phase of the algorithm, which is the most sensitive to rounding errors, as shown by [17, Thm. 10.21].

We solve this dilemma by means of a heuristic that prevents the 1-norm condition number of $q_{k_i m_i}(2^{-s}A)$ from getting too large. In particular, if our estimate $\kappa_A = \text{NORMEST1}(\lambda x.q_m(2^{-s}A)^{-1}x)\ \text{NORMEST1}(\lambda x.q_m(2^{-s}A)x)$ is larger than a constant $\zeta(u)$ that depends on the unit roundoff u, we update the scaling parameter and leave the order of the Padé approximant unchanged. Otherwise, we increment i and take an approximant of higher order chosen according to the elements in $\mathtt{m}$. In practice, we set $\zeta(u) := u^{-1/8}$. For IEEE double precision, this choice gives $\zeta(u) = 2^{53/8} \approx 98.70$, which agrees (within a factor of 1.4) with the largest condition number allowed by Al-Mohy and Higham [2, Table 3.1] for double precision.

Within our algorithm, we can exploit the evaluation scheme discussed in section 3 to reduce the computational cost of the evaluation not only of $r_{\mathtt{m}_i}(2^{-s}A)$, but also of the term $\|q_{\mathtt{m}_i}(2^{-2}A)\|_1$ appearing in the bound (4.1).

Since Algorithm 4.1 considers Padé approximants of increasing cost, for diagonal Padé approximants we have that $\mathtt{m}_i < \mathtt{m}_j$ for $i < j$. Hence, whenever in Algorithm 4.1 the bound (4.1) is evaluated for the approximant of order $\mathtt{m}_i$, we are guaranteed that on line 23 $r_{\mathtt{m}_j}(2^{-s}A)$ will be evaluated for some $j \geqslant i$. Since numerator and denominator of the approximant are evaluated by means of the Paterson–Stockmeyer method, we know that at least the first $\nu = \lceil\sqrt{\mathtt{m}_i}\rceil$ powers of $2^{-s}A$ will be needed, and since scaling a matrix requires only $O(n^2)$ flops, it is worth it to compute immediately the first ν powers of A, and subsequently use them to speed up the estimation of $\|A^d\|_1^{1/d}$, $\|q_{\mathtt{m}_i}^{-1}(2^{-s}A)\|_1$, and $\|\mathrm{e}^A\|_1$.

Fragment 4.3 shows how the bound (4.1) can be evaluated efficiently for diagonal Padé approximants. In order to estimate $\|q_m^{-1}(2^{-s}A)\|_1$, the algorithm computes the matrices U_e and U_o using the Paterson–Stockmeyer method given in Fragment 4.4. This implementation stores the powers of A^2 in the global array $\mathcal{A}$, which is updated only when it does not already contain the first $\lceil\sqrt{m}\rceil$ powers of A^2. Since the number of matrices stored in $\mathcal{A}$ changes with m, we introduce a function length($\mathcal{A}$), that returns the number of positive powers stored in $\mathcal{A}$. In other words, if length($\mathcal{A}$) $= \ell$, then $\mathcal{A}$ contains $\ell + 1$ matrices from $\mathcal{A}_0 = I$ to $\mathcal{A}_\ell = A^{2\ell}$.

Note that although it makes sense, from a performance point of view, to compute U_e and U_o in lower precision, the elements of $\mathcal{A}$ must be computed at precision u in order to be reused to evaluate numerator and denominator of $r_m(2^{-s}A)$. These lower precision approximation of U_e and U_o can be used to compute a cheap approximation $\psi(\mathrm{e}^{2^{-s}A})$ to $\|\mathrm{e}^{2^{-s}A}\|_1$ needed in (4.1), since $\mathrm{e}^{2^{-s}A} \approx q_m(X)^{-1}p_m(X) = 2(U_e - U_o)^{-1}U_o + I$ can be evaluated by means of only one multiple right-hand side system solve at precision u_{bnd}.

Fragment 4.3: Auxiliary functions for diagonal Padé approximants r_m.

```
function evalBoundDiag($A \in \mathbb{C}^{n\times n}, m \in \mathbb{N}, s \in \mathbb{N}$)
    ▷ Check (4.1) for $r_m$ and estimate $\kappa_1(q_m(A))$.
    $\alpha_{\min} \leftarrow$ optAlphaDiag($A, m, \alpha_{\min}$)
    $[U_e, U_o] \leftarrow$ evalPadeDiagAux($A, m, s, u_{bnd}$)
    Set working precision to $u_{bnd}$.
    $[L, U] \leftarrow$ lu($U_e - U_o$)
    $\eta \leftarrow$ normest1($\lambda x.(U^{-1}(L^{-1}x))$)
    Set working precision to $u$.
    $\delta \leftarrow \eta\ |q_m(2^{-s}\alpha_{\min})\mathrm{e}^{2^{-s}\alpha_{\min}} - p_m(2^{-s}\alpha_{\min})|$
    $\psi \leftarrow$ normest1($\lambda x.(U^{-1}(L^{-1}(2U_o x)) + x)$)
    $\kappa_A \leftarrow \eta$ normest1($\lambda x.(U_e - U_o)x$)
    return $\delta,\ \psi,\ \kappa_A$
```

```
function evalPadeDiag($A \in \mathbb{C}^{n\times n}, m \in \mathbb{N}, s \in \mathbb{N}$)
    ▷ Evaluate $r_m(2^{-s}A)$.
    $[U_e, U_o] \leftarrow$ evalPadeDiagAux($A, m, s, u$)
    $[L, U] \leftarrow$ lu($U_e - U_o$)
    return $U^{-1}(L^{-1}(2U_o + I))$
```

```
function evalPadeDiagAux($A \in \mathbb{C}^{n\times n}, m \in \mathbb{N}, s \in \mathbb{N}, u \in \mathbb{R}_+$)
    ▷ Evaluate components of $p_{mm}(A)$ and $q_{mm}(A)$.
    Set working precision to $u$.
    $\beta_e \leftarrow \left[\frac{m!(2m-2i)!}{(2m)!(m-2i)!\ (2i)!}\right]_{i=0}^{\lfloor m/2\rfloor}$    $\beta_o \leftarrow \left[\frac{m!(2m-2i-1)!}{(2m)!(m-2i-1)!\ (2i+1)!}\right]_{i=0}^{\lceil m/2-1\rceil}$
    return evalPolyPS($2s, \beta_e, \lceil\sqrt{m}\rceil$), $(2^{-s}A)$ evalPolyPS($2s, \beta_o, \lceil\sqrt{m}\rceil$)
```

```
function optAlphaDiag($A \in \mathbb{C}^{n\times n}, m \in \mathbb{N}, \alpha_{\min} \in \mathbb{R}_+$)
    ▷ Compute $\alpha_{\min}$.
    $d \leftarrow \left\lfloor \frac{1+\sqrt{5+8m}}{2} \right\rfloor$
    if $\gamma_d = -\infty$ then
        $\gamma_d \leftarrow$ normest1($\lambda x.$evalPowVecDiag($d, x$))$^{1/d}$
    if $\gamma_{d+1} = -\infty$ then
        $\gamma_{d+1} \leftarrow$ normest1($\lambda x.$evalPowVecDiag($d+1, x$))$^{1/(d+1)}$
    return $\min\{\max\{\gamma_d, \gamma_{d+1}\}, \alpha_{\min}\}$
```

```
function evalPowVecDiag($d \in \mathbb{N}, X \in \mathbb{C}^{n\times t}$)
    ▷ Compute $A^d X$ using elements in $\mathcal{A}$.
    $\ell \leftarrow \mathrm{length}(\mathcal{A})$
    while $d > 1$ and $\ell > 1$ do
        for $i \leftarrow 1$ to $\lfloor d/(2\ell)\rfloor$ do
            $X \leftarrow \mathcal{A}_\ell X$
        $d \leftarrow d \mod 2\ell$
        $\ell \leftarrow \min\{\ell - 1, \lfloor d/2\rfloor + 1\}$
    if $d = 1$ then
        $X \leftarrow AX$
    return $X$
```

Fragment 4.4: Modified Paterson–Stockmeyer algorithm.

```
function EVALPOLYPS($s \in \mathbb{N}, \beta \in \mathbb{C}^t, \nu \in \mathbb{N}$)
    ▷ Evaluate $\sum_{\ell=0}^{t} \beta_\ell (2^{-s}A)^\ell$ using elements of $\mathcal{A}$.
$\ell \leftarrow \mathrm{length}(\mathcal{A})$
$\mu \leftarrow \lfloor t/\nu \rfloor$
for $i \leftarrow \ell + 1$ to $\nu$ do
    $\mathcal{A}_i \leftarrow \mathcal{A}_{i-1}\,\mathcal{A}_1$
$Y \leftarrow \sum_{j=0}^{m-\nu\mu} \beta_{\nu\mu+j} 2^{-sj} \mathcal{A}_j$
for $i \leftarrow \mu - 1$ down to $0$ do
    $Y \leftarrow Y 2^{-s\nu} \mathcal{A}_s + \sum_{j=0}^{s-1} \beta_{\nu i+j} 2^{-sj} \mathcal{A}_j$
return $Y$
```

In addition, the elements of $\mathcal{A}$ can be used to reduce the computational cost of estimating the 1-norm of powers of A. In order to estimate $\|A^d\|_1$, NORMEST1 computes repeatedly $Y := A^d X$, where $X \in \mathbb{C}^{n\times t}$, with $t \ll n$. If the matrix multiplications are performed from right to left, evaluating Y requires $2dtn^2$ flops, but if some of the powers of A are available, the factor d can be reduced to as little as $\log_2 d$. We illustrate the strategy to perform this cost reduction in the function EVALPOWVECDIAG, which evaluates $A^d X$ using the powers of A^2 stored in $\mathcal{A}$. We use the two variables $\widetilde{d}$ and ℓ, initialized to d and length($\mathcal{A}$), respectively, to keep track of the state of the computation. The function repeatedly multiplies X by $\mathcal{A}_\ell = A^{2\ell}$ for $t = \lfloor \widetilde{d}/(2\ell) \rfloor$ times, that is, until $\widetilde{d}$ becomes smaller than 2ℓ. At this point, the algorithm updates $\widetilde{d}$ and ℓ, setting the former to the number of matrix multiplications left to perform, $\widetilde{d} - t$, and the latter to the largest integer smaller than the new value of $\widetilde{d}$. Since $\mathcal{A}$ contains powers of A^2 rather than A, an additional multiplication by A is necessary for odd d.

This algorithm requires $\big(\min(C_{\lfloor\sqrt{\mathtt{m}_i}\rfloor}(\mathtt{m}_i), C_{\lceil\sqrt{\mathtt{m}_i}\rceil}(\mathtt{m}_i)) + s\big)n^3$ flops in precision u, for evaluating $r_{\mathtt{m}_i}(A)$ and performing the final squaring phase, and $\big(2\sqrt{\mathtt{m}_i} + \frac{2}{3}\big)in^3$ flops in precision u_{bnd} for evaluating and factorizing $q_{\mathtt{m}_i}(2^{-s}A)$, in order to check whether the bound (4.1) is satisfied.

4.2. Taylor approximants. Truncated Taylor series are appealing Padé approximants to use in conjunction with bound (3.14), as the property that $q_{k0}(x) = 1$ enables us to eliminate the computation of $q_{\mathtt{m}_i}(2^{-s}A)$, the most expensive term to evaluate in (4.1), and thus obtain substantial computational savings. Even though for truncated Taylor series there is no need to evaluate the approximant when evaluating the bound, the function EVALBOUNDTAYL updates the array $\mathcal{A}$, which in this case stores powers of A rather than A^2. In fact, these powers can be used to reduce the cost of estimating $\|A^d\|_1$ as well as $\|e^A\|_1$. The elements of $\mathcal{A}$ are estimated by means of NORMEST1, and the action of the powers of A on a vector is computed by means of the function EVALPOWVECTAYL in Fragment 4.5, which uses the elements in $\mathcal{A}$ analogously to EVALPOWVECDIAG.

For ψ in the bound (4.1) one can use a lower bound on the 1-norm of e^A. The inequality $\|e^A\|_1 \geqslant e^{-\|A\|_1}$ [17, Thm. 10.10] can be exploited at no extra cost, but being typically not very sharp can potentially lead to unnecessary computation. Van

Fragment 4.5: Auxiliary functions for truncated Taylor series t_m.

```
function evalBoundTayl($A \in \mathbb{C}^{n\times n}, m \in \mathbb{N}, s \in \mathbb{N}$)
    ▷ Check (4.1) for $t_m$.
    for $i \leftarrow \text{length}(\mathcal{A}) + 1$ to $\lceil \sqrt{m} \rceil$ do
        $\mathcal{A}_i \leftarrow \mathcal{A}_{i-1}\, \mathcal{A}_1$
    $\alpha_{\min} \leftarrow$ optAlphaTayl($A, m, \alpha_{\min}$)
    $\delta \leftarrow |e^{2^{-s}\alpha_{\min}} - p_m(2^{-s}\alpha_{\min})|$
    $\psi \leftarrow$ estimateNormExp($s$)
    return $\delta$, $\psi$, 1

function evalPadeTayl($A \in \mathbb{C}^{n\times n}, m \in \mathbb{N}, s \in \mathbb{N}$)
    ▷ Evaluate $t_m(2^{-s}A)$.
    $\beta \leftarrow \left[\frac{1}{i!}\right]_{i=0}^{m}$
    return evalPolyPS($s, \beta, \sqrt{m}$)

function optAlphaTayl($A \in \mathbb{C}^{n\times n}, m \in \mathbb{N}, \alpha_{\min} \in \mathbb{R}_+$)
    ▷ Compute $\alpha_{\min}$.
    $d \leftarrow \left\lfloor \frac{1+\sqrt{5+4m}}{2} \right\rfloor$
    if $\gamma_d = -\infty$ then
        $\gamma_d \leftarrow$ normest1($\lambda x.$evalPowVecTayl$(d, x))^{1/d}$
    if $\gamma_{d+1} = -\infty$ then
        $\gamma_{d+1} \leftarrow$ normest1($\lambda x.$evalPowVecTayl$(d+1, x))^{1/(d+1)}$
    return $\min\{\max\{\gamma_d, \gamma_{d+1}\}, \alpha_{\min}\}$

function evalPowVecTayl($d \in \mathbb{N}, X \in \mathbb{C}^{n\times t}$)
    ▷ Compute $A^d X$ using elements in $\mathcal{A}$.
    $\ell \leftarrow \text{length}(\mathcal{A})$
    while $d > 0$ do
        for $i \leftarrow 1$ to $\lfloor p/\ell \rfloor$ do
            $X \leftarrow \mathcal{A}_\ell X$
        $d \leftarrow d \mod \ell$
        $\ell \leftarrow \min\{\ell - 1, d\}$
    return $X$

function estimateNormExp($s \in \mathbb{N}$)
    ▷ Estimate $\|e^A\|_1$ using elements in $\mathcal{A}$.
    $Z \leftarrow \sum_{i=0}^{\text{length}(\mathcal{A})} \frac{\mathcal{A}_i}{2^{si} i!}$
    return normest1($\lambda x.Zx$)
```

Loan [38] suggests the bound

$$\|e^A\|_1 \geqslant e^{\lambda^*}, \qquad \lambda^* = \max_{\lambda \in \sigma(A)} \operatorname{Re} \lambda,$$

which is typically tighter than the previous bound, and always so when $\lambda^* > 0$. Estimating λ^*, however, requires either the eigendecomposition of A, or the solution of a family of shifted linear systems [28], and both solutions might be unpractical in that they require $O(n^3)$ flops for dense matrices. A practical estimate that can be computed with only $O(n^2)$ extra cost is provided by the function ESTIMATENORMEXP in Fragment 4.5, which relies on the approximation

$$\mathrm{e}^{2^{-s}A} \approx \sum_{i=0}^{\ell} \frac{(2^{-s}A)^i}{i!} = \sum_{i=0}^{\ell} \frac{2^{-si}\mathcal{A}_i}{i!} =: \xi_\ell^s, \qquad \ell = \mathrm{length}(\mathcal{A}).$$

If only the elements of $\mathcal{A}$ already computed on line 2-3 of EVALBOUNDTAYL are used, computing this estimate requires only $2\ell n^2$ flops. Additional savings can be gained by noting that $\xi_{\ell+1}^s$ can be obtained from ξ_ℓ^s with only one matrix scaling and one matrix sum, and the full cost of $2\ell n^2$ flops need not be paid as long as s does not change.

Overall, this algorithm requires $\left(2\sqrt{\mathtt{m}_i} + s\right)n^3$ floating-point operations.

4.3. Schur–Padé variants. If A is normal ($A^*A = AA^*$) and a multiprecision implementation of the QR algorithm is available, diagonalization in higher precision is another approach for computing e^A. More generally, a (real) Schur decomposition can be used: $A = QTQ^*$, where $T, Q \in \mathbb{C}^{n\times n}$ are, respectively, upper triangular and unitary if A has complex entries and upper quasi-triangular and orthogonal if A has real entries. Then $\mathrm{e}^A = Q\mathrm{e}^T Q^*$. In our experiments, we consider a Schur–Padé approach that computes the Schur decomposition of the input matrix and exploits Algorithm 4.1 to compute the exponential of its triangular factor.

Overall, this algorithm requires $\left(28 + \left(\min\{C_{\lfloor\sqrt{\mathtt{m}_i}\rfloor}(\mathtt{m}_i), C_{\lceil\sqrt{\mathtt{m}_i}\rceil}(\mathtt{m}_i)\} + s\right)/3\right)n^3$ and $\left(28 + (2\sqrt{\mathtt{m}_i} + s)/3\right)n^3$ flops, for diagonal Padé approximants and truncated Taylor series, respectively.

5. Numerical experiments. We now test the algorithm derived in section 4 and compare it with two existing codes for computing the matrix exponential in arbitrary precision floating point arithmetic. We consider two test sets: $\mathcal{H}$, which contains 35 Hermitian matrices, and $\mathcal{N}$, which consists of 97 non-Hermitian matrices. These matrices, of size ranging between 2 and 1000, are taken from the literature of the matrix exponential, from a collection of benchmark problems for the burnup equations [23], [39], and from the MATLAB `gallery` function. The experiments were performed using the 64-bit (glxna64) version of MATLAB 9.4 (2018a) on a machine equipped with an Intel I5-3570 processor running at 3.40GHz and with 8Gb of RAM. The code uses the Multiprecision Computing Toolbox (version 4.4.7.12739) [31], which provides the class `mp` to represent arbitrary precision floating-point numbers and overloads all the MATLAB functions we need in our implementations. We note that this toolbox allows the user to specify the number of *decimal* digits of working precision, but not the number of bits in the fraction of its binary representation, thus, in this section, whenever we refer to d (decimal) digits of precision, we mean that the working precision is set using the command `mp.Digits(d)`. The MATLAB code that runs the tests in this section is available on GitHub.[1]

In our experiments, we compare the following codes.

- `expm`, the built-in function `expm` of MATLAB, which implements the algorithm of Al-Mohy and Higham [2], and is intended for double precision only.
- `exp_mct`, the `expm` function provided by the Multiprecision Computing Toolbox.

[1] https://github.com/mfasi/mpexpm.

- `exp_otf`, the algorithm by Caliari and Zivcovich [7], a shifted scaling and squaring method based on truncated Taylor series. The matrix is shifted by $\mathrm{trace}(A)/n$, and (1.2) is replaced by

$$\mathrm{e}^A = \left(\mathrm{e}^{(2^s+2^t)^{-1}A}\right)^{2^s+2^t}, \quad s \in \mathbb{N}, \quad t \in \mathbb{N} \cup \{-\infty\}.$$

  The order of the approximant is chosen by estimating at runtime a bound on the backward error of the approximant in exact arithmetic.
- `exp_d`, an implementation of Algorithm 4.1 with `use_taylor` = **false**.
- `exp_t`, an implementation of Algorithm 4.1 with `use_taylor` = **true**.
- `exp_sp_d`, an implementation of the Schur–Padé approach discussed in section 4.3 using Algorithm 4.1, with `use_taylor` = **false**, for the triangular Schur factor.
- `exp_sp_t`, an implementation of the Schur–Padé approach discussed in section 4.3 using Algorithm 4.1, with `use_taylor` = **true**, for the triangular Schur factor.

In our implementations of Algorithm 4.1, we set $u_{bnd} = 2^{-53}$, $\epsilon = u$, $s_{\max} = 100$, and the entries of `m` to the elements of (3.5) smaller than 1000 and those of (3.6) smaller than 400, for truncated Taylor series and diagonal Padé approximant, respectively.

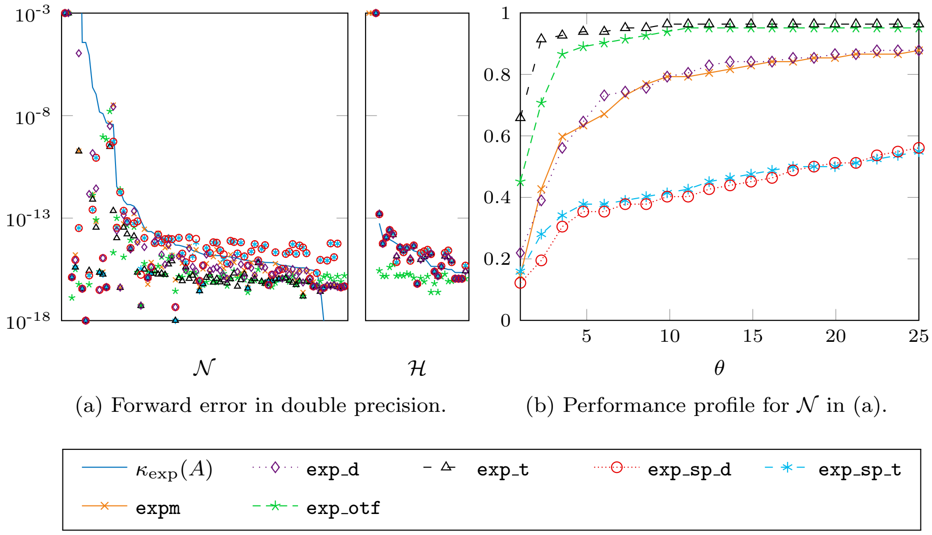
We do not include the function `expm` provided by the Symbolic Math Toolbox in our tests since, for precision higher than double, it appears to be extending the precision internally, using a number of extra digits that increases with the working precision. As a result, the forward error of the algorithm is typically several orders of magnitude smaller than machine epsilon, when computing with 20 or more digits, but tends to be larger than the unit roundoff u, for $u \approx 10^{-20}$ or larger. We note that the accuracy drops for matrices that are nondiagonalizable, singular, or have ill-conditioned eigenvectors. Moreover, the Symbolic Math Toolbox implementation is rather slow on moderately large matrices ($n \gtrsim 50$, say), which makes this code unsuitable for extended testing.

In our tests, we assess the accuracy of the solution $\widetilde{X}$ computed by an algorithm running with d digits of precision by means of the relative forward error $\|X - \widetilde{X}\|_1/\|X\|_1$, where X is a reference solution computed using `exp_mct` with $2d$ significant digits. Since the magnitude of forward errors depends not only on the working precision but also on the conditioning of the problem, in our plots we compare the forward error of the algorithms with $\kappa_{\exp}(A)u$, where $\kappa_{\exp}(A)$ is the 1-norm condition number [17, Ch. 4] of the matrix exponential of A (see (2.1)). We estimate it in double precision using the `funm_condest1` function provided by the Matrix Function Toolbox [15] on `expm`.

When plotting the forward error, we choose the limits of the y-axis in order to show the area where most of the data points lie, and move the outliers to the the closest edge of the box containing the plot. In several cases, we present our experimental results with the aid of performance profiles [10], and adopt the technique of Dingle and Higham [9] to rescale values smaller than u.

5.1. Comparison with `expm` in double precision. In this experiment, we compare the performance of `exp_d`, `exp_t`, `exp_sp_d`, and `exp_sp_t` running in IEEE double precision, with that of `expm` and `exp_otf`. The purpose of this experiment is to verify that the new algorithms are broadly competitive with `expm`; since `expm` is optimized for double precision, we do not expect them to be as efficient.

Figure 5.1a compares the forward error of the algorithms on the matrices in our test sets, sorted by decreasing condition number. The performance profile in



(a) Forward error in double precision.

(b) Performance profile for $\mathcal{N}$ in (a).

(c) Legend for the data in (a) and (b).

Figure 5.1: Left: forward error of `expm` and `exp_sp_d`, `exp_sp_t`, `exp_d`, and `exp_t` running in IEEE double precision arithmetic on the matrices in $\mathcal{N}$ and $\mathcal{H}$, ordered by decreasing value of $\kappa_{\exp}(A)$. Right: performance profile for the matrices in $\mathcal{N}$.

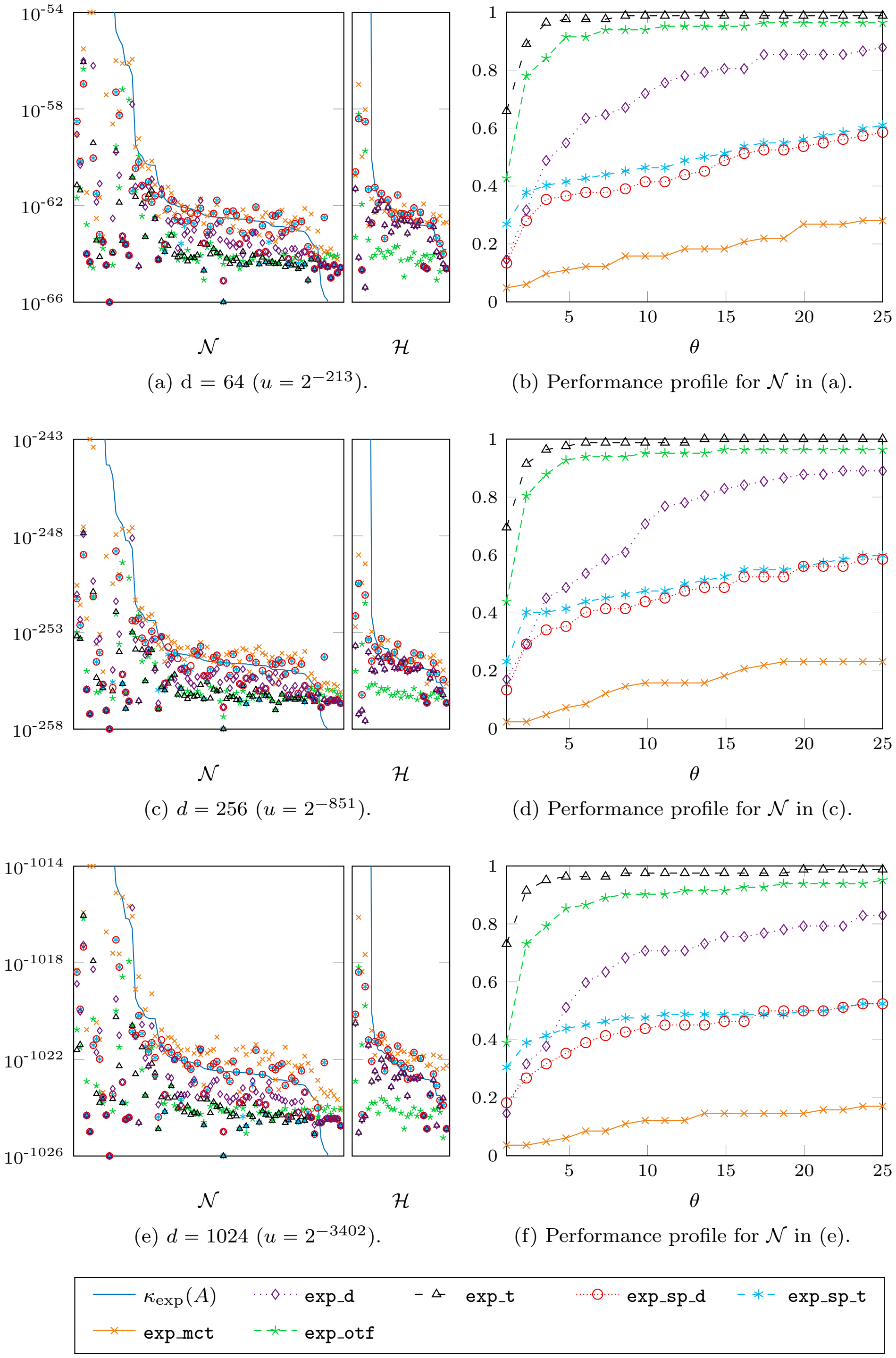
Figure 5.1b presents the results for $\mathcal{N}$ on a by-algorithm rather than by-matrix basis: for a given method, the height of the line at $\theta = \theta_0$ represents the fraction of matrices in the data set for which the relative forward error is within a factor θ_0 from the error of the algorithm that delivers the most accurate result for that matrix.

For Hermitian matrices, we do not provide a performance profile, as the error plot clearly shows that while `exp_otf`, `exp_t`, and `exp_d` all provide forward error well below $\kappa_{\exp}(A)u$, `exp_otf` is consistently the most accurate on that data set. The performance of `expm` is the same as that of `exp_sp_d` because both implementations reduce to the evaluation of the scalar exponential at the eigenvalues of A when A is Hermitian.

For the matrices in $\mathcal{N}$, the errors of `expm`, `exp_otf`, `exp_d`, and `exp_t` are approximately bounded by $\kappa_{\exp}(A)u$, with the algorithms based on truncated Taylor series being overall more accurate than those based on diagonal Padé approximants. The algorithms based on the Schur decomposition of A tend to give somewhat larger errors, with the result that the performance profile curves are the least favorable.

5.2. Behavior in higher precision. Now we investigate the accuracy of our algorithm in higher precision and compare it with `exp_mct`, the built-in function of the Multiprecision Computing Toolbox, and `exp_otf`. The left column of Figure 5.2 compares the quantity $\kappa_{\exp}(A)u$ with the forward errors of `exp_mct`, `exp_otf`, `exp_sp_d`, `exp_d`, `exp_sp_t`, and `exp_t` running with about 64, 256, and 1024 decimal significant digits on the matrices in our test sets sorted by decreasing condition number $\kappa_{\exp}(A)$. The right-hand column presents the data for the matrices in $\mathcal{N}$ by means of performance profiles.

The results show that, as for double precision, transformation-free algorithms



(a) d = 64 ($u = 2^{-213}$).

(b) Performance profile for $\mathcal{N}$ in (a).

(c) $d = 256$ ($u = 2^{-851}$).

(d) Performance profile for $\mathcal{N}$ in (c).

(e) $d = 1024$ ($u = 2^{-3402}$).

(f) Performance profile for $\mathcal{N}$ in (e).

(g) Legend for the data in (a)–(f).

Figure 5.2: Left: forward error of the methods on the matrices in the test sets. Right: Corresponding performance profiles for the matrices in $\mathcal{N}$.

Table 5.1: Execution time breakdown of `exp_t` and `exp_d`, run in quadruple precision on three matrices of increasing size. The table reports, for each algorithm, the number of squarings (M_{sqr}), the number of matrix multiplications needed to evaluate the Padé approximant (M_{eval}), the total execution time in seconds (T_{tot}), and the percentage of time spent evaluating the scalar bound (T_{bnd}), evaluating the Padé approximant(T_{eval}), and performing the final squaring step (T_{sqr}).

		`exp_t`						`exp_d`					
	n	M_{sqr}	M_{eval}	T_{bnd}	T_{eval}	T_{sqr}	T_{tot}	M_{sqr}	M_{eval}	T_{bnd}	T_{eval}	T_{sqr}	T_{tot}
`A`	10	5	13	20%	39%	41%	0.1	7	8	60%	12%	28%	0.2
	20	6	13	13%	31%	56%	0.1	8	8	41%	14%	45%	0.2
	50	7	15	7%	26%	67%	0.3	9	9	37%	17%	46%	0.6
	100	8	15	4%	29%	67%	0.8	10	9	35%	23%	42%	1.6
	200	9	15	4%	38%	58%	2.7	11	9	42%	29%	29%	6.4
	500	10	16	3%	45%	51%	27.5	12	9	39%	33%	28%	59.2
	1000	11	16	3%	45%	52%	197.1	12	10	33%	38%	29%	365.3
`B`	10	1	11	24%	61%	15%	0.0	1	9	69%	24%	8%	0.1
	20	2	12	14%	58%	28%	0.1	2	9	54%	27%	18%	0.1
	50	3	13	8%	44%	48%	0.2	3	10	45%	28%	27%	0.4
	100	4	13	5%	34%	61%	0.4	5	9	41%	28%	31%	1.1
	200	5	13	4%	33%	63%	1.3	6	9	46%	31%	24%	4.4
	500	6	14	3%	30%	67%	11.2	6	10	39%	39%	22%	37.6
	1000	7	14	2%	24%	74%	69.0	7	10	35%	39%	26%	238.7
`C`	10	2	11	35%	64%	1%	0.0	1	10	76%	24%	0%	0.1
	20	2	12	23%	76%	1%	0.1	2	9	69%	30%	1%	0.1
	50	3	12	16%	75%	9%	0.1	2	10	53%	45%	2%	0.3
	100	3	13	8%	78%	14%	0.5	2	11	40%	56%	3%	1.4
	200	3	14	6%	79%	15%	3.6	3	10	34%	58%	8%	6.5
	500	4	13	5%	73%	21%	52.4	4	9	21%	63%	16%	69.5
	1000	4	14	5%	74%	21%	434.0	5	9	16%	62%	22%	505.2

tend to produce a more accurate approximation of e^A than those based on the Schur decomposition of A. The code `exp_mct` typically delivers the least accurate results, with a forward error usually larger than $\kappa_{\exp}(A)u$.

On the Hermitian matrices in the test set, `exp_otf` is typically the most accurate algorithm, having a surprising ability to produce errors much less than $\kappa_{\exp}(A)u$. On the set of non-Hermitian matrices, `exp_otf` and `exp_t` are the most accurate algorithms, with `exp_t` having the superior performance profile. On this dataset, the least accurate of our versions of Algorithm 4.1 is `exp_sp_t`, closely followed by `exp_sp_d`; the forward error of both, despite being typically smaller than that of `exp_mct`, is often slightly larger than $\kappa_{\exp}(A)u$, especially for the better conditioned of our test matrices. Finally, `exp_d` performs better than Schur-based variants, but is slightly less accurate than `exp_otf` and `exp_t` on most of the matrices in this data set. We remark, however, that its forward error is typically smaller than $\kappa_{\exp}(A)u$ when that of the other two Taylor-based algorithms is.

We note that the tests of `exp_otf` in [7] have precision target 10^{-25} or larger, which is between double and quadruple precision. This experiment shows that `exp_otf` maintains its good accuracy up to much higher precisions.

5.3. Code profiling. In Table 5.1, we compare the execution time of our MATLAB implementations of `exp_t` and `exp_d`, running in quadruple precision ($d = 34$ and unit roundoff $u = 2^{-113}$), on the matrices

```
A = 1e2*triu(gallery(’pei’’,n),1);
B = zeros(n); B(n+1:n+1:n^2) = 1:n-1;  % Upper bidiagonal.
C = expm(gallery(’lotkin’, n));
```

where n ranges between 10 and 1000. For each matrix, we report the overall execution time in seconds of the two implementations (T_{tot}), specifying how much time is spent evaluating the scalar bound (T_{bnd}), evaluating the approximant on the input matrix (T_{eval}), and performing the final squaring stage (T_{sqr}).

For `exp_d`, the evaluation of the forward bound on the truncation error of the Padé approximant is typically the most expensive operation for small matrices, and even when the size of the matrix increases the cost of this operation remains nonnegligible, as the estimation of the quantity $\left\|(q_{km}(A))^{-1}\right\|_1$, which appears in the bound (4.1), has cubic dependence on the size of the input matrix. For `exp_t`, on the other hand, T_{bnd} depends only quadratically on n, and tends to become relatively small for matrices of size larger than 100.

6. Conclusions.

State-of-the-art scaling and squaring algorithms for computing the matrix exponential typically rely on a backward error analysis in order to scale the matrix and then select an appropriate Padé approximant to meet a given accuracy threshold. The result of this analysis is a small set of precision-dependent constants that are hard to compute but can be easily stored for subsequent use. This approach is not viable in multiprecision environments, where the working precision is known only at run time and is an input argument of the algorithm rather than a property of the underlying floating-point number system. For truncated Taylor series, it is possible to estimate on-the-fly a bound on the backward error of the approximation [7], but this technique relies on a conjecture and does not readily generalize to other Padé approximants.

We have developed a new algorithm (Algorithm 4.1) based on Padé approximation for computing the matrix exponential in arbitrary precision. In particular, we derived a new representation for the truncation error of a general $[k/m]$ Padé approximant and showed how it can be used to compute practical error bounds for truncated Taylor series and diagonal Padé approximants. In the first case, the bound is cheap to compute, requiring only $O(n^2)$ flops. For diagonal Padé approximants the new bound requires $O(n^3)$ low-precision flops, but if Conjecture 3.2 turns out to be true then this cost will reduce to $O(n^2)$ flops.

According to our experimental results, Algorithm 4.1 in transformation-free form using truncated Taylor series is the best option for computing the matrix exponential in precisions higher than double. In particular, the algorithm is the most accurate on non-Hermitian matrices. For Hermitian matrices, it is natural to compute e^A via a spectral decomposition (as does the MATLAB function `expm`), but an interesting finding is that on our Hermitian test matrices this approach (to which `exp_sp_d` and `exp_sp_t` effectively reduce) is less accurate than Algorithm 4.1 and the algorithm of [7].

When developing the algorithm and testing it experimentally we focused on precisions higher than double. We believe that a different approach is needed to address the computation of the matrix exponential in low precision arithmetic, such as IEEE half precision. Indeed, due its very limited range this number format is prone to underflow and overflow, which makes accuracy and robustness difficult to achieve. How to handle these challenges will be the subject of future work.

REFERENCES

[1] A. H. Al-Mohy, *A truncated Taylor series algorithm for computing the action of trigonometric and hyperbolic matrix functions*, SIAM J. Sci. Comput., 40 (2018), pp. A1696–A1713, https://doi.org/10.1137/17M1145227.

[2] A. H. Al-Mohy and N. J. Higham, *A new scaling and squaring algorithm for the matrix exponential*, SIAM J. Matrix Anal. Appl., 31 (2009), pp. 970–989, https://doi.org/10.1137/09074721X.

[3] A. H. Al-Mohy and N. J. Higham, *Improved inverse scaling and squaring algorithms for the matrix logarithm*, SIAM J. Sci. Comput., 34 (2012), pp. C153–C169, https://doi.org/10.1137/110852553.

[4] A. H. Al-Mohy, N. J. Higham, and S. D. Relton, *New algorithms for computing the matrix sine and cosine separately or simultaneously*, SIAM J. Sci. Comput., 37 (2015), pp. A456–A487, https://doi.org/10.1137/140973979.

[5] M. Aprahamian and N. J. Higham, *Matrix inverse trigonometric and inverse hyperbolic functions: Theory and algorithms*, SIAM J. Matrix Anal. Appl., 37 (2016), pp. 1453–1477, https://doi.org/10.1137/16M1057577.

[6] 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.

[7] M. Caliari and F. Zivcovich, *On-the-fly backward error estimate for matrix exponential approximation by Taylor algorithm*, J. Comput. Appl. Math., 346 (2019), p. 532548, https://doi.org/10.1016/j.cam.2018.07.042.

[8] E. Defez, J. Sastre, J. Ibáñez, and J. Peinado, *Solving engineering models using hyperbolic matrix functions*, Applied Mathematical Modelling, 40 (2016), pp. 2837–2844, https://doi.org/https://doi.org/10.1016/j.apm.2015.09.050.

[9] N. J. Dingle and N. J. Higham, *Reducing the influence of tiny normwise relative errors on performance profiles*, ACM Trans. Math. Software, 39 (2013), pp. 24:1–24:11, https://doi.org/10.1145/2491491.2491494.

[10] E. D. Dolan and J. J. Moré, *Benchmarking optimization software with performance profiles*, Math. Programming, 91 (2002), pp. 201–213, https://doi.org/10.1007/s101070100263.

[11] M. Fasi, *Optimality of the Paterson–Stockmeyer method for evaluating matrix polynomials and some rational matrix functions*, MIMS EPrint, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, 2018. In preparation.

[12] 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.

[13] M. Fasi, N. J. Higham, and B. Iannazzo, *An algorithm for the matrix Lambert W function*, SIAM J. Matrix Anal. Appl., 36 (2015), pp. 669–685, https://doi.org/10.1137/140997610.

[14] W. Gautschi, *Numerical Analysis*, Birkhäuser Basel, 2011.

[15] N. J. Higham, *The Matrix Function Toolbox*. http://www.maths.manchester.ac.uk/~higham/mftoolbox.

[16] N. J. Higham, *The scaling and squaring method for the matrix exponential revisited*, SIAM J. Matrix Anal. Appl., 26 (2005), pp. 1179–1193, https://doi.org/10.1137/04061101X.

[17] 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.

[18] N. J. Higham, *The scaling and squaring method for the matrix exponential revisited*, SIAM Rev., 51 (2009), pp. 747–764, https://doi.org/10.1137/090768539.

[19] N. J. Higham and P. Kandolf, *Computing the action of trigonometric and hyperbolic matrix functions*, SIAM J. Sci. Comput., 39 (2017), pp. A613–A627, https://doi.org/10.1137/16M1084225.

[20] N. J. Higham and F. Tisseur, *A block algorithm for matrix* 1*-norm estimation, with an application to* 1*-norm pseudospectra*, SIAM J. Matrix Anal. Appl., 21 (2000), pp. 1185–1201, https://doi.org/10.1137/S0895479899356080.

[21] F. Johansson, *Arb: Efficient arbitrary-precision midpoint-radius interval arithmetic*, IEEE Trans. Comput., 66 (2017), p. 12811292, https://doi.org/10.1109/tc.2017.2690633.

[22] F. Johansson et al., *Mpmath: A Python library for arbitrary-precision floating-point arithmetic*. http://mpmath.org.

[23] D. Lago and F. Rahnema, *Development of a set of benchmark problems to verify numerical methods for solving burnup equations*, Annals of Nuclear Energy, 99 (2017), pp. 266–271, https://doi.org/10.1016/j.anucene.2016.09.004.

[24] E. N. Laguerre, *Le calcul des systèmes linéaires, extrait d'une lettre adressé à M. Hermite*, in Oeuvres de Laguerre, C. Hermite, H. Poincaré, and E. Rouché, eds., vol. 1, Gauthier–

Villars, Paris, 1898, pp. 221–267, http://gallica.bnf.fr/ark:/12148/bpt6k90210p/f242.table. The article is dated 1867 and is "Extrait du Journal de l'École Polytechnique, LXII^{e} Cahier".

[25] J. D. Lawson, *Generalized Runge–Kutta processes for stable systems with large lipschitz constants*, SIAM J. Numer. Anal., 4 (1967), pp. 372–380, https://doi.org/10.1137/0704033.

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

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

[28] K. Meerbergen, A. Spence, and D. Roose, *Shift-invert and Cayley transforms for detection of rightmost eigenvalues of nonsymmetric matrices*, BIT, 34 (1994), pp. 409–423, https://doi.org/10.1007/bf01935650.

[29] C. B. Moler and C. F. Van Loan, *Nineteen dubious ways to compute the exponential of a matrix*, SIAM Rev., 20 (1978), pp. 801–836, https://doi.org/10.1137/1020098.

[30] C. B. Moler and C. F. Van Loan, *Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later*, SIAM Rev., 45 (2003), pp. 3–49, https://doi.org/10.1137/S00361445024180.

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

[32] P. Nadukandi and N. J. Higham, *Computing the wave-kernel matrix functions*, MIMS EPrint 2018.4, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, Feb. 2018, http://eprints.maths.manchester.ac.uk/2651/. Revised August 2018. To appear in SIAM J. Sci. Comput.

[33] M. S. Paterson and L. J. Stockmeyer, *On the number of nonscalar multiplications necessary to evaluate polynomials*, SIAM J. Comput., 2 (1973), pp. 60–66, https://doi.org/10.1137/0202007.

[34] M. Pusa, *Rational approximations to the matrix exponential in burnup calculations*, Nuclear Science and Engineering, 169 (2011), p. 155167, https://doi.org/10.13182/nse10-81.

[35] The Sage Developers, *Sage Mathematics Software*. http://www.sagemath.org.

[36] J. Sastre, J. Ibáñez, P. Ruiz, and E. Defez, *Accurate and efficient matrix exponential computation*, Internat. J. Comput. Math., 91 (2013), p. 97112, https://doi.org/10.1080/00207160.2013.791392.

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

[38] C. F. Van Loan, *The sensitivity of the matrix exponential*, SIAM J. Numer. Anal., 14 (1977), pp. 971–981, https://doi.org/10.1137/0714065.

[39] S. Zhao, *Matrix exponential approximation for burnup equation*, M.Sc. thesis, The University of Manchester, 2017.