Computing the Action of Trigonometric and Hyperbolic Matrix Functions

Higham, Nicholas J. and Kandolf, Peter

2017

MIMS EPrint: **2016.40**

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

SIAM J. SCI. COMPUT.
Vol. 39, No. 2, pp. A613–A627

COMPUTING THE ACTION OF TRIGONOMETRIC AND HYPERBOLIC MATRIX FUNCTIONS*

NICHOLAS J. HIGHAM† AND PETER KANDOLF‡

Abstract. We derive a new algorithm for computing the action $f(A)V$ of the cosine, sine, hyperbolic cosine, and hyperbolic sine of a matrix A on a matrix V, without first computing $f(A)$. The algorithm can compute $\cos(A)V$ and $\sin(A)V$ simultaneously, and likewise for $\cosh(A)V$ and $\sinh(A)V$, and it uses only real arithmetic when A is real. The algorithm exploits an existing algorithm `expmv` of Al-Mohy and Higham for $\mathrm{e}^A V$ and its underlying backward error analysis. Our experiments show that the new algorithm performs in a forward stable manner and is generally significantly faster than alternatives based on multiple invocations of `expmv` through formulas such as $\cos(A)V = (\mathrm{e}^{\mathrm{i}A}V + \mathrm{e}^{-\mathrm{i}A}V)/2$.

Key words. matrix function, action of matrix function, trigonometric function, hyperbolic function, matrix exponential, Taylor series, backward error analysis, exponential integrator, splitting methods

AMS subject classifications. 65F60, 65D05, 65F30

DOI. 10.1137/16M1084225

1. Introduction.

This work is concerned with the computation of $f(A)V$ for trigonometric and hyperbolic functions f, where $A \in \mathbb{C}^{n\times n}$ and $V \in \mathbb{C}^{n\times n_0}$ with $n_0 \ll n$. Specifically, we consider the computation of the actions of the matrix cosine, sine, hyperbolic cosine, and hyperbolic sine functions. Algorithms exist for computing these matrix functions, such as those in [3], [9], but we are not aware of any existing algorithms for computing their actions.

Applications where these actions are required include differential equations (as discussed below) and network analysis [7], [15]. Furthermore, the proposed algorithm can also be utilized to compute the action of the matrix exponential or φ functions at different time steps. This, in return, finds an application in the efficient implementation of exponential integrators [14]. One distinctive feature of the algorithm proposed is that it avoids complex arithmetic for a real matrix. This characteristic can be exploited to use only real arithmetic in the computation of the matrix exponential as well, if the matrix is real but the step argument complex. This is useful for higher order splitting methods [8], or for the solution of the Schrödinger equation, where the problem can be rewritten so that the step argument is complex and the matrix is real (see Example 4.3).

One line of attack is to develop algorithms for $f(A)V$ for each of these four f individually. An algorithm `expmv` of Al-Mohy and Higham [2] for computing the action of the matrix exponential relies on the scaling and powering relation $\mathrm{e}^A b =$

*Submitted to the journal's Methods and Algorithms for Scientific Computing section July 11, 2016; accepted for publication (in revised form) January 30, 2017; published electronically April 20, 2017.

http://www.siam.org/journals/sisc/39-2/M108422.html

Funding: The first author's work was supported by European Research Council Advanced Grant MATFUN (267526) and Engineering and Physical Sciences Research Council grant EP/I01912X/1. The work of the second author was supported by a DOC Fellowship of the Austrian Academy of Science at the Department of Mathematics, University of Innsbruck, Austria.

†School of Mathematics, The University of Manchester, Manchester, M13 9PL, UK (nick.higham@manchester.ac.uk).

‡Institut für Mathematik, Universität Innsbruck, Innsbruck, 6020 Austria (peter.kandolf@uibk.ac.at).

$(\mathrm{e}^{A/s})^s b$, for nonnegative integers s, and uses a Taylor polynomial approximation to $\mathrm{e}^{A/s}$. The trigonometric functions cos and sin do not enjoy the same relation, and while the double- and triple-angle formulas $\cos(2A) = 2\cos^2(A) - I$ and $\sin(3A) = 3\sin(A) - 4\sin^3(A)$ can be successfully used in computing the cosine and sine [3], they do not lend themselves to computing the action of these functions. For this reason our focus will be on exploiting the algorithm of [2] for the action of the matrix exponential. While this approach may not be optimal for each of the four f, we will show that it leads to a numerically reliable algorithm and has the advantage that it allows the use of existing software.

The matrix cosine and sine functions arise in solving the system of second order differential equations

$$\frac{\mathrm{d}^2}{\mathrm{d}t^2} y + A^2 y = 0, \qquad y(0) = y_0, \quad y'(0) = y_0',$$

whose solution is given by

$$y(t) = \cos(tA) y_0 + A^{-1} \sin(tA) y_0'.$$

Note that A^2 is the given matrix, so A may not always be known or easy to obtain. By rewriting this system as a first order system of twice the dimension the solution can alternatively be obtained as the first component of the action of the matrix exponential:

$$\begin{aligned} \begin{bmatrix} y(t) \\ y(t)' \end{bmatrix} &= \exp\left(t \begin{bmatrix} 0 & I \\ -A^2 & 0 \end{bmatrix} \right) \begin{bmatrix} y_0 \\ y_0' \end{bmatrix} = \begin{bmatrix} \cos(tA) & A^{-1}\sin(tA) \\ -A\sin(tA) & \cos(tA) \end{bmatrix} \begin{bmatrix} y_0 \\ y_0' \end{bmatrix} \\ &= \begin{bmatrix} \cos(tA) y_0 + A^{-1}\sin(tA) y_0' \\ -A\sin(tA) y_0 + \cos(tA) y_0' \end{bmatrix}. \end{aligned} \tag{1.1}$$

By setting $y_0 = b$ and $y_0' = 0$, or $y_0 = 0$ and $y_0' = b$, and solving a linear system with A or multiplying by A, respectively, we obtain $\cos(tA)b$ and $\sin(tA)b$. However, as a general purpose algorithm, making use of `expmv` from [2], this approach has several disadvantages. First, each step requires two matrix–vector products with A, when we would hope for one. Second, because the block matrix has zero trace, no shift is applied by `expmv`, so an opportunity is lost to reduce the norms. Third, the coefficient matrix is nonnormal (unless A^2 is orthogonal), which can lead to higher computational cost [2].

We recall that all four of the functions addressed here can be expressed as linear combinations of exponentials [12, chap. 12]:

$$\cosh A = \tfrac{1}{2}(\mathrm{e}^A + \mathrm{e}^{-A}), \qquad \sinh A = \tfrac{1}{2}(\mathrm{e}^A - \mathrm{e}^{-A}), \tag{1.2a}$$

$$\cos A = \tfrac{1}{2}(\mathrm{e}^{\mathrm{i}A} + \mathrm{e}^{-\mathrm{i}A}), \qquad \sin A = \tfrac{-\mathrm{i}}{2}(\mathrm{e}^{\mathrm{i}A} - \mathrm{e}^{-\mathrm{i}A}). \tag{1.2b}$$

Furthermore, we have

$$\mathrm{e}^{\mathrm{i}A} = \cos A + \mathrm{i}\sin A, \tag{1.3}$$

which implies that for real A, $\cos A = \mathrm{Re}\,\mathrm{e}^{\mathrm{i}A}$ and $\sin A = \mathrm{Im}\,\mathrm{e}^{\mathrm{i}A}$. The main idea of this paper is to exploit these formulas to compute $\cos(A)V$, $\sin(A)V$, $\cosh(A)V$, and $\sinh(A)V$ by computing $\mathrm{e}^{\beta A}V$ and $\mathrm{e}^{-\beta A}V$ simultaneously with $\beta = \mathrm{i}$ and $\beta = 1$, using a modification of the algorithm `expmv` of [2].

In section 2 we discuss the backward error of the underlying computation. In section 3 we present the algorithm and the computational aspects. Numerical experiments are given in section 4, and in section 5 we offer some concluding remarks.

2. Backward error analysis. The aim of this section is to bound the backward error for the approximation of $f(A)V$ using truncated Taylor series expansions of the exponential, for the four functions f in (1.2). Here, backward error is with respect to truncation errors in the approximation, and exact computation is assumed.

We will use the analysis of Al-Mohy and Higham [2], with refinements to reflect the presence of two related exponentials in each of the definitions of our four functions.

It suffices to consider the approximation of e^A, since the results apply immediately to $\mathrm{e}^A V$. We consider a general approximation $r(A)$, where r is a rational function, since when r is a truncated Taylor series no simplifications accrue.

Since A appears as $\pm A$ and $\pm \mathrm{i}A$ in (1.2), in order to cover all cases we treat βA, where $|\beta| \le 1$. Consider the matrix

$$G = \mathrm{e}^{-\beta A} r(\beta A) - I.$$

With log denoting the principal matrix logarithm [12, sect. 1.7], let

$$E = \log(\mathrm{e}^{-\beta A} r(\beta A)) = \log(I + G), \tag{2.1}$$

where $\rho(G) < 1$ is assumed for the existence of the logarithm. We assume that r has the property that $r(X) \to \mathrm{e}^X$ as $X \to 0$, which is enough to ensure that $\rho(G) < 1$ for small enough βA.

Exponentiating (2.1), and using the fact that all terms commute (each being a function of A), we obtain

$$r(\beta A) = \mathrm{e}^{\beta A + E},$$

so that E is the backward error matrix for the approximation.

For some positive integer ℓ and some radius of convergence $d > 0$ we have, from (2.1), the convergent power series expansion

$$E = \sum_{i=\ell}^{\infty} c_i (\beta A)^i, \qquad |\beta| \rho(A) < d.$$

We can bound E by taking norms to obtain

$$\|E\| \le \sum_{i=\ell}^{\infty} |c_i| \, \|\beta A\|^i =: g(\|\beta A\|). \tag{2.2}$$

Assuming that $g(\theta) = O(\theta^2)$, the quantity

$$\widehat{\theta} := \max\{\, \theta > 0 : \, \theta^{-1} g(\theta) \le \mathrm{tol} \,\} \tag{2.3}$$

exists and we have the backward error result that $\|\beta A\| \le \widehat{\theta}$ implies $r(\beta A) = \mathrm{e}^{\beta A + E}$, with $\|E\| \le \mathrm{tol} \, \|\beta A\|$. Here tol represents the tolerance specified for the backward error.

In practice, we use scaling to achieve the required bound on $\|\beta A\|$, so our approximation is $r(\beta A/s)^s$ for some nonnegative integer s. With s chosen so that $\|\beta A/s\| \le \widehat{\theta}$, we have

$$r(\beta A/s)^s = \mathrm{e}^{\beta A + sE}, \qquad \frac{\|sE\|}{\|\beta A\|} \le \mathrm{tol}.$$

The crucial point is that since $g(\|\beta A\|) = g(|\beta| \|A\|) \le g(\|A\|)$, for all $|\beta| \le 1$, the parameter s chosen for A can be used for βA. Consequently, the original analysis

gives the same bounds for $\pm A$ and $\pm \mathrm{i}A$ and the same parameters can be used for the computation of all four of these functions. This result does not state that the backward error is the same for each β, but rather the weaker result that each of the backward errors satisfies the same inequality.

In practice, we use in place of $\|\beta A\|$ in (2.2) the quantity $\alpha_p(\beta A)$, where

$$\alpha_p(X) = \max(d_p, d_{p+1}), \quad d_p = \|X^p\|^{1/p}, \tag{2.4}$$

for some p with $\ell \geq p(p-1)$, which gives potentially much sharper bounds, as shown in [1, Thm. 4.2(a)].

Our conclusion is that all four matrix functions appearing in (1.2) can be computed in a backward stable manner with the same parameters. As we will see in the next section, the computations can even be combined to compute the necessary values simultaneously.

3. The basic algorithm. As our core algorithm for computing the action of the matrix exponential, we take the truncated Taylor series algorithm of Al-Mohy and Higham [2]. We recall some details of the algorithm below. Other algorithms, such as the Leja method presented in [4], can be employed in a similar fashion, though the details will be different. The truncated Taylor series algorithm takes

$$r(A) \equiv T_m(A) = \sum_{j=0}^{m} \frac{A^j}{j!}.$$

As suggested in [2], we limit the degree m of the polynomial approximant T_m to $m_{\max} = 55$. In order to allow the algorithm to work for general matrices, with no restriction on the norm, we introduce a scaling factor s and assume that $\mathrm{e}^{s^{-1}A}$ is well approximated by $T_m(s^{-1}A)$. From the functional equation of the exponential we have

$$\mathrm{e}^A V = \left(\mathrm{e}^{s^{-1}A}\right)^s V,$$

and so the recurrence

$$V_{i+1} = T_m(s^{-1}A)V_i, \quad i = 0, \ldots, s-1, \quad V_0 = V,$$

yields the approximation $V_s \approx \mathrm{e}^A V$. For a given m, the function g in (2.2) has $\ell = m + 1$.

The parameter $\widehat{\theta}$ in (2.3), which we now denote by θ_m, depends on the polynomial degree $m = \ell - 1$ and the tolerance tol, and its values are given in [1, Table 3.1] for IEEE single precision arithmetic and double precision arithmetic. The cost function

$$C_m(A) = ms = m \max\{1, \lceil \alpha_p(A)/\theta_m \rceil\} \tag{3.1}$$

measures the number of matrix–vector products, and the optimal degree m_* is chosen in [2] such that

$$C_{m_*}(A) = \min\{m \lceil \alpha_p(A)/\theta_m \rceil : \ 2 \leq p \leq p_{\max}, \ p(p-1) - 1 \leq m \leq m_{\max}\}. \tag{3.2}$$

Here, $m_{\max}$ is the maximal admissible Taylor polynomial degree and $p_{\max}$ is the maximum value of p such that $p(p-1) \leq m_{\max} + 1$, to allow the use of (2.4). Furthermore, $p_{\max} = 8$ is the default choice in the implementation. The parameters m_* and s are determined by Algorithm 3.1, which is [2, Code Fragment 3.1].

Algorithm 3.1 $[m_*, s] = \text{parameters}(A, B, \text{tol})$.

This code determines m_* and s given $A \in \mathbb{C}^{n\times n}$, $B \in \mathbb{C}^{n\times q}$, tol, $m_{\max}$, and $p_{\max}$. It is assumed that the α_p in (3.2) are estimated using the block 1-norm estimation algorithm of Higham and Tisseur [13] with two columns.

```
1  if ‖A‖_1 ≤ (4/q) (θ_{m_max}/m_max) p_max(p_max + 3)
2        m_* = arg min_{1≤m≤m_max} m⌈‖A‖_1/θ_m⌉
3        s = ⌈‖A‖_1/θ_{m_*}⌉
4  else
5        Let m_* be the smallest m achieving the minimum in (3.2).
6        s = max{C_{m_*}(A)/m_*, 1}
7  end
```

A further reduction of the cost can be achieved by choosing an appropriate point μ as the center of the Taylor series expansion. As suggested in [2], the shift is selected such that the Frobenius norm $\|A - \mu I\|_F$ is minimized, that is, $\mu = \text{trace}(A)/n$.

Algorithm 3.2 of [2] computes $\mathrm{e}^A B = [\mathrm{e}^A b_1, \dots, \mathrm{e}^A b_q]$, that is, the action of e^A on several vectors. The following modification of that algorithm essentially computes $[\mathrm{e}^{\tau_1 A} b_1, \dots, \mathrm{e}^{\tau_q A} b_q]$: the actions at different t values. The main difference between our algorithm and [2, Alg. 3.2] is in line 12 of Algorithm 3.2 below, where a scalar "t" has been changed to a (block) diagonal matrix

$$D(\tau) = D(\tau_1, \tau_2, \dots, \tau_q) \in \mathbb{C}^{q\times q}$$

that we define precisely below. The exponential computed in line 8 of Algorithm 3.2 is therefore a matrix exponential.

For simplicity we omit balancing, but it can be applied in the same way as in [2, Alg. 3.2].

Note that for $\widetilde{A} = A - \mu I$, $B = [b_1, b_2]$, and $D(\tau) = \text{diag}(\tau_1, \tau_2)$ we have $g(B) = [g_1, g_2]$ in Algorithm 3.2, with

$$\begin{aligned} g_j &\approx \left(b_j + \frac{\widetilde{A}}{s} b_j \tau_j + \frac{\widetilde{A}^2}{s^2 2!} b_j \tau_j^2 + \frac{\widetilde{A}^3}{s^3 3!} b_j \tau_j^3 + \cdots \right) \mathrm{e}^{\mu\tau_j/s} \\ &= \mathrm{e}^{(A-\mu I)\tau_j/s} \mathrm{e}^{\mu\tau_j/s} b_j = \mathrm{e}^{A\tau_j/s} b_j \end{aligned}$$

for $j = 1, 2$. Therefore, we can compute the four actions of interest by selecting appropriately τ_1, τ_2, and B and carrying out some postprocessing. For given t, A, and b we can compute, with $\mathbf{F}$ as in Algorithm 3.2,

1. an approximation of $\cosh(tA)b$ by

$$B = [b/2, b/2], \quad D(\tau) = \begin{bmatrix} t & 0 \\ 0 & -t \end{bmatrix}, \quad \cosh(tA)b = \mathbf{F}(D(\tau), A, B) \begin{bmatrix} 1 \\ 1 \end{bmatrix};$$

2. an approximation of $\sinh(tA)b$ by

$$B = [b/2, b/2], \quad D(\tau) = \begin{bmatrix} t & 0 \\ 0 & -t \end{bmatrix}, \quad \sinh(tA)b = \mathbf{F}(D(\tau), A, B) \begin{bmatrix} 1 \\ -1 \end{bmatrix};$$

3. an approximation of $\cos(tA)b$ by

$$B = [b/2, b/2], \quad D(\tau) = \begin{bmatrix} \mathrm{i}t & 0 \\ 0 & -\mathrm{i}t \end{bmatrix}, \quad \cos(tA)b = \mathbf{F}(D(\tau), A, B) \begin{bmatrix} 1 \\ 1 \end{bmatrix};$$

Algorithm 3.2 $F = \mathbf{F}(D(\tau), A, B)$.

For $\tau_1, \dots, \tau_q \in \mathbb{C}$, $A \in \mathbb{C}^{n\times n}$, $B \in \mathbb{C}^{n\times q}$, and a tolerance tol, the following algorithm produces a matrix $F = g \circ g \circ \cdots \circ g(B) \in \mathbb{C}^{n\times q}$ (s-fold composition) where $g(B) \approx \big(B + \frac{1}{s}\widetilde{A}BD(\tau) + \frac{1}{2!s^2}\widetilde{A}^2BD(\tau)^2 + \frac{1}{3!s^3}\widetilde{A}^3BD(\tau)^3 + \cdots\big)J$, where $\widetilde{A}$ and J are given in the algorithm.

```
1  Ã = A − μI, where μ = trace(A)/n
2  t = max_k |τ_k|
3  if t‖Ã‖_1 = 0
4        m_* = 0, s = 1
5  else
6        [m_*, s] = parameters(tÃ, B, tol) % Algorithm 3.1
7  end
8  F = B, J = e^{μD(τ)/s}
9  for k = 1: s
10       c_1 = ‖B‖_∞
11       for j = 1: m_*
12             B = ÃB(D(τ)/(sj)),
13             c_2 = ‖B‖_∞
14             F = F + B
15             if c_1 + c_2 ≤ tol‖F‖_∞, break, end
16             c_1 = c_2
17       end
18       F = FJ, B = F
19 end
```

4. an approximation of $\sin(tA)b$ by

$$B = [b/2, b/2], \quad D(\tau) = \begin{bmatrix} \mathrm{i}t & 0 \\ 0 & -\mathrm{i}t \end{bmatrix}, \quad \sin(tA)b = \mathbf{F}(D(\tau), A, B)\begin{bmatrix} -\mathrm{i} \\ \mathrm{i} \end{bmatrix}.$$

Obviously, since they share the same B and $D(\tau)$, we can combine the computation of $\cosh(tA)b$ and $\sinh(tA)b$, and $\cos(tA)b$ and $\sin(tA)b$, respectively, without any additional cost. Furthermore, it is also possible to combine the computation of all four matrix functions by a single call to $\mathbf{F}(D(\tau), A, B)$ with $B = [b, b, b, b]/2$ and $D(\tau) = \mathrm{diag}[t, -t, \mathrm{i}t, -\mathrm{i}t]$.

If A is a real matrix the computation of $\cos(tA)b$ and $\sin(tA)b$ can be performed entirely in real arithmetic, as we now show. We need the formula

$$\exp\left(\begin{bmatrix} 0 & t \\ -t & 0 \end{bmatrix}\right) = \begin{bmatrix} \cos t & \sin t \\ -\sin t & \cos t \end{bmatrix}. \tag{3.3}$$

LEMMA 3.1. *For $A \in \mathbb{R}^{n\times n}$, $b = b_{\mathrm{r}} + \mathrm{i}b_{\mathrm{i}} \in \mathbb{C}^n$, and $t \in \mathbb{R}$, the vector $f = f_{\mathrm{r}} + \mathrm{i}f_{\mathrm{i}} = \mathbf{F}(D(\mathrm{i}t), A, b) \approx \mathrm{e}^{\mathrm{i}tA}b$ can be computed in real arithmetic by*

$$\big[f_{\mathrm{r}}, \quad f_{\mathrm{i}}\big] = \mathbf{F}\big(D(\tau), A, \big[b_{\mathrm{r}}, b_{\mathrm{i}}\big]\big), \quad \textit{where} \quad \tau = \mathrm{i}t, \quad D(\tau) = \begin{bmatrix} 0 & t \\ -t & 0 \end{bmatrix}. \tag{3.4}$$

Furthermore, the resulting vectors f_{r} and f_{i} are approximations of, respectively,

$$f_{\mathrm{r}} = \cos(tA)b_{\mathrm{r}} - \sin(tA)b_{\mathrm{i}}, \qquad f_{\mathrm{i}} = \sin(tA)b_{\mathrm{r}} + \cos(tA)b_{\mathrm{i}}.$$

Proof. With $B = [b_{\mathrm{r}}, b_{\mathrm{i}}]$ we have

$$g(B) \approx \left([\, b_{\mathrm{r}}, b_{\mathrm{i}}\,] + t \left[\frac{\widetilde{A} b_{\mathrm{r}}}{s}, \frac{\widetilde{A} b_{\mathrm{i}}}{s} \right] \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} + t^2 \left[\frac{\widetilde{A}^2 b_{\mathrm{r}}}{s^2 2!}, \frac{\widetilde{A}^2 b_{\mathrm{i}}}{s^2 2!} \right] \right.$$
$$\left. + t^3 \left[\frac{\widetilde{A}^3 b_{\mathrm{r}}}{s^3 3!}, \frac{\widetilde{A}^3 b_{\mathrm{i}}}{s^3 3!} \right] \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} + \cdots \right) \exp\left(\begin{bmatrix} 0 & t\mu/s \\ -t\mu/s & 0 \end{bmatrix} \right),$$

and on collecting terms, applying (3.3) and the addition formulas [12, Thm. 12.1], and recalling that $\widetilde{A} = A - \mu I$, we find that

$$g(B) \approx \left[\cos\left(\tfrac{t\widetilde{A}}{s}\right) b_{\mathrm{r}} - \sin\left(\tfrac{t\widetilde{A}}{s}\right) b_{\mathrm{i}}, \quad \sin\left(\tfrac{t\widetilde{A}}{s}\right) b_{\mathrm{r}} + \cos\left(\tfrac{t\widetilde{A}}{s}\right) b_{\mathrm{i}} \right] \begin{bmatrix} \cos\left(\frac{t\mu}{s}\right) & \sin\left(\frac{t\mu}{s}\right) \\ -\sin\left(\frac{t\mu}{s}\right) & \cos\left(\frac{t\mu}{s}\right) \end{bmatrix}$$
$$= \begin{bmatrix} \cos\left(\frac{tA}{s}\right) & -\sin\left(\frac{tA}{s}\right) \\ \sin\left(\frac{tA}{s}\right) & \cos\left(\frac{tA}{s}\right) \end{bmatrix} \begin{bmatrix} b_{\mathrm{r}} \\ b_{\mathrm{i}} \end{bmatrix} =: C \begin{bmatrix} b_{\mathrm{r}} \\ b_{\mathrm{i}} \end{bmatrix}.$$

Hence, overall, using (3.3) again,

$$\mathbf{F}(D(\mathrm{i}t)A, b) \approx C^s \begin{bmatrix} b_{\mathrm{r}} \\ b_{\mathrm{i}} \end{bmatrix} = \begin{bmatrix} \cos(tA) & -\sin(tA) \\ \sin(tA) & \cos(tA) \end{bmatrix} \begin{bmatrix} b_{\mathrm{r}} \\ b_{\mathrm{i}} \end{bmatrix},$$

as required. ∎

As a consequence of Lemma 3.1 we can compute, with D defined in (3.4),

1. an approximation of $\cos(tA)b$ by

$$B = [b, 0], \quad \tau = \mathrm{i}t, \quad D(\tau) = \begin{bmatrix} 0 & t \\ -t & 0 \end{bmatrix}, \qquad \cos(tA)b = \mathbf{F}(D(\tau), A, B) \begin{bmatrix} 1 \\ 0 \end{bmatrix};$$

2. an approximation of $\sin(tA)b$ by

$$B = [b, 0], \quad \tau = \mathrm{i}t, \quad D(\tau) = \begin{bmatrix} 0 & t \\ -t & 0 \end{bmatrix}, \qquad \sin(tA)b = \mathbf{F}(D(\tau), A, B) \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

We compute the matrix exponential J in line 8 of Algorithm 3.2 by making use of (3.3).

We make three remarks.

Remark 3.2 (other cases.). Algorithm 3.2 can also be used to compute exponentials at different time steps and with the use of [2, Thm. 2.1] it can be used to compute linear combinations of φ functions at different time steps (see, e.g., [12, sect. 10.7.4] for details of the φ functions). This in turn is useful for the implementation of exponential integrators [14]. The internal stages of an exponential integrator often require the evaluation of a φ function at intermediate steps, e.g., $\varphi(c_k tA)b$ for $0 < c_k \leq 1$ and $k \geq 1$. Although the new algorithm can be used in these situations, it might not be optimal for each of the c_k values as the parameters m_* and s are chosen for the largest value of t and might not be optimal for an intermediate point. Nevertheless, the computation can be performed in parallel for all the different values of t and level-3 BLAS routines can be used, which can speed up the process. Furthermore, the algorithm could also be used to generate dense output, in terms of the time step, as is sometimes desired for time integration.

Remark 3.3. We note that in [2, Code Fragment 5.1, Alg. 5.2] the authors also present an algorithm to compute $\mathrm{e}^{t_k A}b$ on equally spaced grid points $t_k = t_0 + hk$ with

$h = (t_q - t_0)/q$. With that code we can compute $\cosh(A)b$ and $\sinh(A)b$ by setting $t_0 = -t$, $t_q = t$, and $q = 1$, so that $b_1 = \mathrm{e}^{t_0 A}b = \mathrm{e}^{-tA}b$, $h = 2t$, and $b_2 = \mathrm{e}^{hA}b_1 = \mathrm{e}^{tA}b$. This is not only slower than our approach, as the code now has to perform a larger time step and compute the necessary steps consecutively and not in parallel, but it can also cause instability. In fact, for some of the matrices of Example 4.1 in section 4 we see a large error if we use [2, Alg. 5.2] as outlined above. Furthermore, as we compute with $\pm\beta$ we can optimize the algorithm by using level-3 BLAS routines and we can avoid complex arithmetic by our direct approach.

Remark 3.4 (block version). As indicated in the introduction, it is sometimes required to compute the action of our four functions not on a vector but on a tall, thin matrix $V \in \mathbb{C}^{n\times n_0}$. It is possible to use Algorithm 3.2 for this task. One simply needs to repeat each τ_k value n_0 times and the matrix V needs to be repeated q times for each of the τ_k values (this corresponds to replacing the vector b by the matrix V in the definition of B). This procedure can be formalized with the help of the Kronecker product $X \otimes Y$. We define the time matrix by $D(\tau) \otimes I_{n_0}$, and the postprocessing matrix $\tilde{P}$ by $P \otimes I_{n_0}$. Furthermore, the matrix B reads as $I_q \otimes V/2$. For $V = [v_1, v_2]$ ($n_0 = 2$) the computation of $\cosh(tA)V$ becomes

$$B = [v_1, v_2, v_1, v_2]/2, \quad \tilde{D}(\tau) = D(\tau) \otimes I_2 = \mathrm{diag}(t, t, -t, -t)$$

and results in

$$\cosh(tA)V = \mathbf{F}(\tilde{D}(\tau), A, B) \begin{bmatrix} I_2 \\ I_2 \end{bmatrix}.$$

4. Numerical experiments. Now we present some numerical experiments that illustrate the behavior of Algorithm 3.2. All of the experiments were carried out in MATLAB R2015a (glnxa64) on a Linux machine and for time measurements only one processor is used. We work with three tolerances in Algorithm 3.2, corresponding to half precision, single precision, and double precision, respectively:

$$u_{\mathrm{half}} = 2^{-11} \approx 4.9 \times 10^{-4},$$
$$u_{\mathrm{single}} = 2^{-23} \approx 6.0 \times 10^{-8},$$
$$u_{\mathrm{double}} = 2^{-53} \approx 1.1 \times 10^{-16}.$$

All computations are in IEEE double precision arithmetic.

We use the implementations of the algorithms of [2] from https://github.com/higham/expmv, which are named `expmv` for [2, Alg. 3.2] and `expmv_tspan` for [2, Alg. 5.2]. We also use the implementations `cosm` and `sinm` from https://github.com/sdrelton/cosm_sinm of the algorithm of [3, Alg. 6.2] for computing the matrix sine and cosine; the default option of using a Schur decomposition is chosen in the first experiment, but no Schur decomposition is used in the second and third experiments. We note that we did not use the function `cosmsinm` for a simultaneous computation as we found it less accurate than `cosm` and `sinm` in Example 4.1.

In order to compute $\cos(tA)b$ and $\sin(tA)b$ we use the following methods.

1. `trigmv` denotes Algorithm 3.2 with real or complex arithmetic (avoiding complex arithmetic when possible), computing the two functions simultaneously.
2. `trig_expmv` denotes the use of `expmv`, in two forms. For a real matrix `expmv` is called with the pure imaginary step argument it, making use of (1.3). For a complex matrix `expmv` is called twice, with step arguments it and $-$it, and (1.2b) is used.

3. `dense` denotes the use of `cosm` and `sinm` to compute the dense matrices $\cos(tA)$ and $\sin(tA)$ before the multiplication with b.
4. `trig_block` denotes the use of formula (1.1) with $y_0 = 0$ and $y_0' = b$. Therefore, we need one extra matrix–vector product to compute $\sin(tA)b$. In order to compute the action of the exponential we use `expmv`.

For the computation of $\cosh(tA)b$ and $\sinh(tA)b$ we use the following methods.

1. `trighmv` denotes Algorithm 3.2, computing the two functions simultaneously.
2. `trigh_expmv` denotes the use of `expmv` called twice with $\pm t$ as step arguments.
3. `expmv_tspan` denotes [2, Alg. 5.2] called with $t_0 = -t$, $q = 1$, and $t_q = t$, as discussed in Remark 3.3.
4. `dense` denotes the use of `cosm` and `sinm` to compute the dense matrices $\cosh(tA)$ and $\sinh(tA)$ as $\cos(\mathrm{i}tA)$ and $-\mathrm{i}\sin(\mathrm{i}tA)$, respectively, before the multiplication with b.
5. `trigh_block` denotes the use of formula (1.1) with $y_0 = 0$ and $y_0' = b$, where $\mathrm{i}A$ is substituted for A. We need one extra matrix–vector product to compute $\sinh(tA)b$. In order to compute the action of the exponential we use `expmv`.

All of the methods except `dense` support tolerances u_{half}, u_{single}, and u_{double}, whereas `dense` is designed to deliver double precision accuracy.

In all cases, when Algorithm 3.1 is called to compute the optimal scaling and truncation degree we use $m_{\max} = 55$ and $p_{\max} = 8$.

We compute relative errors in the 1-norm, $\|x - \widehat{x}\|_1 / \|x\|_1$, where $x = f(A)b$. In Example 4.1, $\widehat{x}$ denotes a reference solution computed with the Multiprecision Computing Toolbox [16] at 100-digit precision. In Examples 4.2 and 4.3 the matrices are too large for multiprecision computations so the reference solution X is taken as that obtained via `cosm` or `sinm`.

Example 4.1 (behavior for existing test sets). In this experiment we compare `trigmv`, `trig_expmv`, and `dense`. We show only the results for cos and cosh, as the results for sin and sinh are very similar.

As test matrices we use Sets 1–3 from [1, sect. 6], with dimensions n up to 50. We remove all matrices from our test sets where any of the considered functions overflow; the overflow also appears for the dense method considered and is due to the result being too large to represent. The elements of the vector b are drawn from the standard normal distribution and are the same for each matrix. We compare the algorithms for tolerances u_{half}, u_{single}, and u_{double}.

The relative errors are shown in Figure 4.1, with the test matrices ordered by decreasing condition number $\kappa_{\cos}$ of the matrix cosine. The estimated condition number is computed by the `funm_condest1` function of the Matrix Function Toolbox [11]. The required Fréchet derivative is computed with the 2×2 block form [12, sect. 3.2].

From the error plot in Figure 4.1 one can see that `trigmv` and `trig_expmv` behave in a forward stable manner, that is, the relative error is always within a modest multiple of the condition number of the problem times the tolerance, and likewise for `dense` except for some mild instability on four problems.

We also show in Figure 4.2 a performance profile for the experiment with tolerance u_{double}. In the performance profile the curve for a given method shows the proportion of problems p for which the error is within a factor α of the smallest error over all methods. In particular, the value at $\alpha = 1$ corresponds to the proportion of problems where the method performs best and for large values of α the performance profile gives an idea of the reliability of the method. The performance profile is computed with the code from [10, sect. 26.4] and we employ the idea of [6] to reduce the bias of

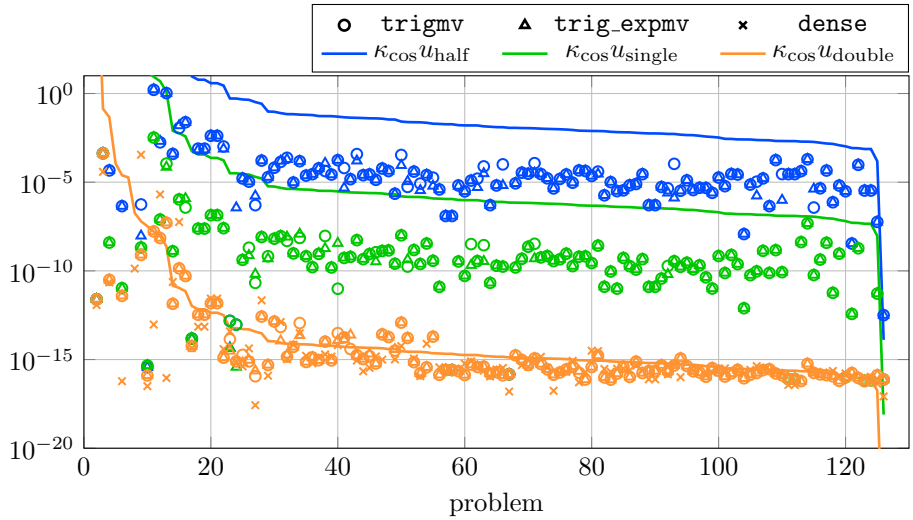

FIG. 4.1. *Relative error in* 1*-norm for computing* $\cos(A)b$ *with three algorithms with tolerances* u_{half} *(blue),* u_{single} *(green), and* u_{double} *(orange). The solid lines are the condition number multiplied by the tolerance.*

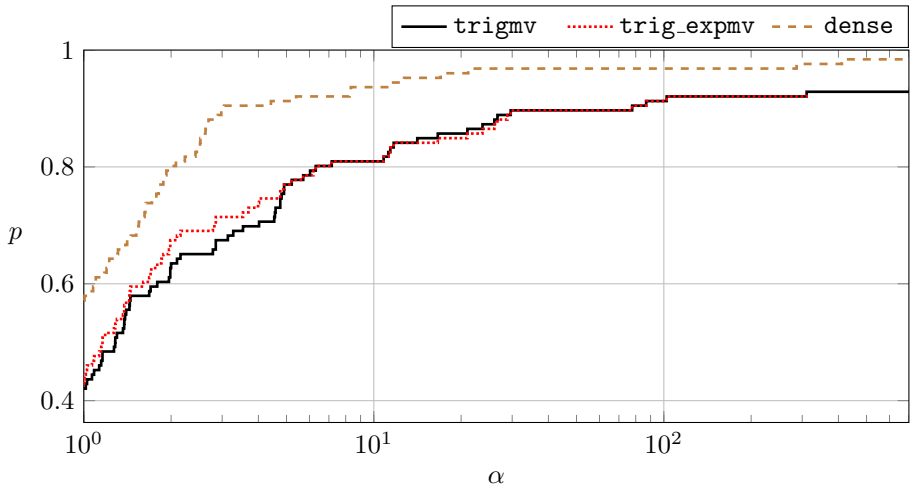

FIG. 4.2. *Same data as in Figure* 4.1 *for* u_{double} *but presented as a performance profile. For each method,* p *is the proportion of problems in which the error is within a factor of* α *of the smallest error over all methods.*

relative errors significantly less than the precision. The performance profile suggests that the overall behavior of `trigmv` and `trig_expmv` is very similar.

For the computation of cosh, shown in Figure 4.3, `expmv_tspan` is clearly not a good choice for the computation. This is related to the implementation of `expmv_tspan`. As the algorithm first computes $b_1 = \mathrm{e}^{-A}b$ and from this computes $b_2 = \mathrm{e}^{2A}b_1$, the result is not always stable, as discussed in Remark 3.3. We see that `trighmv` and `trigh_expmv` behave in a forward stable manner and have about the same accuracy for all three tolerances, as is clear for double precision from the performance profile in Figure 4.4.

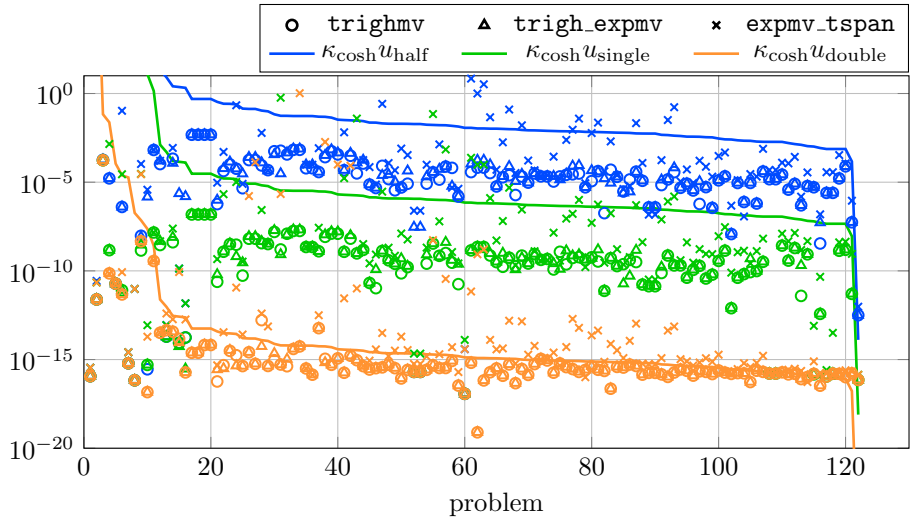

FIG. 4.3. *Relative error in* 1*-norm for computing* $\cosh(A)b$ *with tolerances* u_{half} *(blue),* u_{single} *(green), and* u_{double} *(orange). The solid lines are the condition number multiplied by the tolerance.*

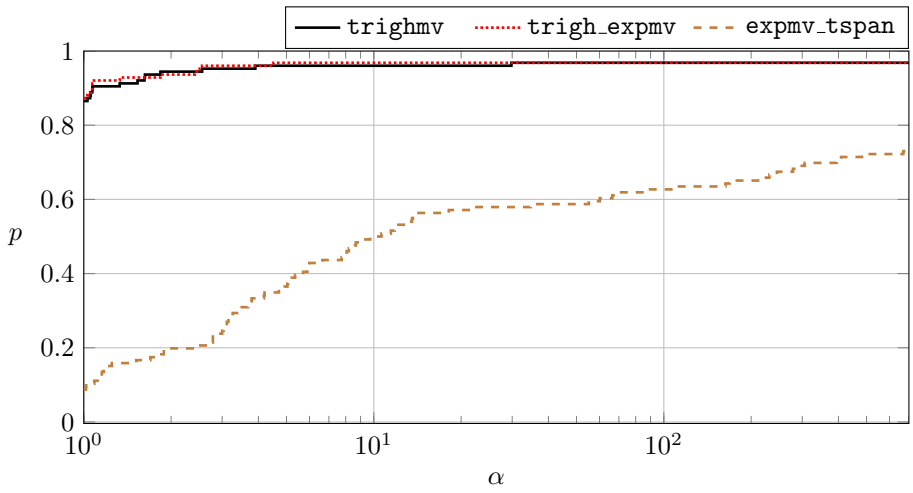

FIG. 4.4. *Same data as in Figure* 4.3 *for* u_{double} *but presented as a performance profile. For each method,* p *is the proportion of problems in which the error is within a factor of* α *of the smallest error over all methods.*

Example 4.2 (behavior for large matrices). In this experiment we take a closer look at the behavior of several algorithms for large (sparse) matrices. For the computation of the trigonometric functions we compare `trigmv` with `trig_block` and `trig_expmv`, which both rely on `expmv`. For a real matrix `trig_expmv` calls `expmv` with a pure imaginary step argument and two calls are made for a complex matrix. For the hyperbolic functions, we compare `trighmv` with `trigh_block` and `trigh_expmv`. This time `trigh_expmv` always calls `expmv` twice and `trigh_block` calls `expmv` with a pure imaginary step argument. When `expmv` is called several times the preprocessing step (Algorithm 3.1) is only performed once.

TABLE 4.1
Behavior of the algorithms for large (sparse) matrices, for tolerance u_{double}.

(a) Results for the computation of cos and sin.

		`trigmv`		`trig_expmv`		`trig_block`		`dense`
	t	mv	Time	mv	Time	mv	Time	Time
`orani676`	100	2200	2.3e-1	4164	3.1e-1	2599	9.5e-1	2.8e2
`bcspwr10`	10	618	4.1e-2	1500	1.2e-1	1392	1.2e-1	2.6e2
`triw`	10	56740	5.7e1	113192	1.1e2	95389	1.2e2	1.9e1
`triu`	40	3936	4.0	7524	8.5	4585	5.2	1.4e1
`L2`	1/4	107528	1.2e1	215320	1.9e1	257803	3.0e1	1.3e3

(b) Results for the computation of cosh and sinh.

		`trighmv`		`trigh_expmv`		`trigh_block`		`dense`
	t	mv	Time	mv	Time	mv	Time	Time
`orani676`	100	2202	2.2e-1	2202	2.7e-1	2619	9.5e-1	2.1e2
`bcspwr10`	10	632	4.1e-2	806	5.4e-2	855	7.4e-2	5.4e2
`triw`	10	56478	5.7e1	57582	1.2e2	94499	1.1e2	1.2e2
`triu`	40	4042	4.1	4031	8.0	4689	5.4	2.9e1
`S3D`	1/2	15962	1.7e1	15934	2.1e1	32135	1.4e1	2.3e4
`Trans1D`	2	13086	2.0e-1	13039	2.4e-1	17551	2.1e-1	6.2

We use the same matrices as in [4, Example 9], namely `orani676` and `bcspwr10`, which are obtained from the University of Florida Sparse Matrix Collection [5]. The matrix `orani676` is a nonsymmetric 2529×2529 matrix with 90158 nonzero entries and `bcspwr10` is a symmetric 5300×5300 matrix with 13571 nonzero entries. The matrix `triw` is `-gallery('triw',2000,4)`, which is a 2000×2000 upper triangular matrix with -1 in the main diagonal and -4 in the upper triangular part. The matrix `triu` is an upper triangular matrix of dimension 2000 with entries uniformly distributed on $[-0.5, 0.5]$. The 9801×9801 matrix `L2` is from a finite difference discretization (second order symmetric differences) of the two-dimensional Laplacian in the unit square. The 27000×27000 complex matrix `S3D` is from a finite difference discretization (second order symmetric differences) of the three-dimensional (3D) Schrödinger equation with harmonic potential in the unit cube. The matrix `Trans1D` is a periodic, symmetric finite difference discretization of the transport equation in the unit square with dimension 1000.

As vector b we use $[1, \dots, 1]^{\mathrm{T}}$ for `orani676`, $[1, 0, \dots, 0, 1]^{\mathrm{T}}$ for `bcspwr10`, the discretization of $256x^2(1-x)^2y^2(1-y)^2$ for `L2`, the discretization of $4096x^2(1-x)^2y^2(1-y)^2z^2(1-z)^2$ for `S3D`, the discretization of $\exp(-100(x-0.5)^2)$ for `Trans1D`, and $v_i = \cos i$ for all other examples.

The results for computing $\cos(tA)b$ and $\sin(tA)b$ are shown in Table 4.1a, and those for $\cosh(tA)b$ and $\sinh(tA)b$ in Table 4.1b. The different algorithms are run with tolerance u_{double}. All the methods behave in a forward stable manner, with one exception, so we omit the errors in the table. The exception is the `trigh_block` method, which has an error about 10^2 times larger than the other methods for `Trans1D`. For the different methods we list the number of real matrix–vector products performed (mv), as well as the overall time in seconds averaged over ten runs. The tables also show the time the dense algorithm needed to compute the reference solution (computing both functions simultaneously).

In Table 4.1a we can see that `trigmv` always needs the fewest matrix–vector products and that with the sole exception of `triw` it is always the fastest method.

TABLE 4.2
Results for the solution of the Schrödinger equation with $N = 30$. We show the number of matrix–vector products performed, the relative error in the 1-norm, and the CPU time.

	tol $= u_{\text{single}}$			tol $= u_{\text{double}}$		
	mv	Rel. err	Time	mv	Rel. err	Time
`trigmv`	11034	1.3e-7	3.9	15846	2.7e-11	5.6
`trig_expmv`	21952	1.3e-7	6.2	31516	2.7e-11	8.8
`trig_block`	15883	5.2e-8	7.1	32023	1.1e-11	1.4e1
`expleja`	11180	8.0e-9	4.3	17348	1.5e-11	6.6
`dense`	-	-	-	-	-	2.2e4

We can also see that, as expected, `trig_block` has higher computational cost than `trigmv`. The increase in matrix–vector products is most pronounced for normal matrices (`bcspwr10` and `L2`). For the matrix `bcspwr10` we find $s = 7$, $mv = 618$, and $mvd = 44$ (matrix–vector products performed in the preprocessing stage, in Algorithm 3.1) for `trigmv`. On the other hand, for `trig_block` we find $s = 10$, $mv = 696 \cdot 2 = 1392$, and $mvd = 328 \cdot 2 = 656$. This means that the preprocessing stage is more expensive as the block matrix is nonnormal and more α_p values need to be computed. We can also see that we need more scaling steps as we miss the opportunity to reduce the norm. In total this sums up to more than twice the number of matrix–vector products.

The results of the experiment for the hyperbolic functions can be seen in Table 4.1b. Again `trighmv` almost always needs fewer matrix–vector products than the other methods where this time `trigh_expmv` is the closest competitor and `trigh_block` has a higher computational effort. Even in the cases where `trigh_expmv` needs the same number of matrix–vector products or slightly fewer, `trighmv` is still clearly faster. This is due to the fact that `trigmv` employs level-3 BLAS.

Comparing the runtime of `trigmv` and `trighmv` with the `dense` algorithms we can see that we potentially save a great deal of computation time. The `triw` and `triu` matrices are the only cases where there is not a speedup of at least a factor of 10. For the `triw` matrix, and to a lesser extent for the `triu` matrix, the α_p values, which help deal with the nonnormality of the matrix, decay very slowly, and this hinders the performance of the algorithms. Nevertheless, in all the other cases we can see a clear speed advantage, most significantly for `bcspwr10` where we have a speedup by a factor 6190.

Example 4.3 (Schrödinger equation). In this example we solve an evolution equation. We consider the 3D Schrödinger equation with harmonic potential

$$\partial_t u = \frac{\mathrm{i}}{2}\left(\Delta - \frac{1}{2}\left(x^2 + y^2 + z^2\right)\right) u. \tag{4.1}$$

We use a finite difference discretization in space with N^3 points on the domain $\Omega = [0,1]^3$ and as initial value we use the discretization of $4096x^2(1-x)^2y^2(1-y)^2z^2(1-z)^2$. We obtain a discretization matrix $\mathrm{i}A$ of size 27000×27000, where A is symmetric with all eigenvalues on the negative real axis. We deliberately keep i separate, and as a result the solution of (4.1) can be interpreted as

$$u(t) = \mathrm{e}^{\mathrm{i}tA}u_0 = \cos(tA)u_0 + \mathrm{i}\sin(tA)u_0.$$

Table 4.2 reports the results for the tolerances u_{single} and u_{double}, for our new algorithm `trigmv`, `trig_expmv`, `trig_block`, and `expleja` (the method from [4] called

in the same fashion as `trig_expmv`). The table shows the number of matrix–vector products performed, the relative error, and the CPU time in seconds. We see that the four methods achieve roughly the same accuracy. We also see that `trigmv` requires significantly fewer matrix–vector products than `trig_expmv` and `trig_block`. On the other hand, even though `expleja` is a close competitor in terms of matrix–vector products performed, the overall CPU time is higher than for `trigmv`. This is due to the fact that `trigmv` is avoiding complex arithmetic and employs level-3 BLAS. Also note that `trigmv` needs less storage than `expleja` as for the latter the matrix needs to be complex. Again we can see that the dense method needs roughly 1000 times longer for the computation than the other algorithms.

5. Concluding remarks. We have developed the first algorithm for computing the actions of the matrix functions $\cos A$, $\sin A$, $\cosh A$, and $\sinh A$. Our new algorithm, Algorithm 3.2, can evaluate the individual actions or the actions of any of the functions simultaneously. The algorithm builds on the framework of the $\mathrm{e}^A b$ algorithm `expmv` of Al-Mohy and Higham [1], inheriting its backward stability with respect to truncation errors, its exclusive use of matrix–vector products (or matrix–matrix products in our modification), and its features for countering the effects of nonnormality. For real A, $\cos(A)b$ and $\sin(A)b$ are computed entirely in real arithmetic. As a result of these features and its careful reuse of information, Algorithm 3.2 is more efficient than alternatives that make multiple calls to `expmv`, as our experiments demonstrate.

Our MATLAB codes are available at https://bitbucket.org/kandolfp/trigmv.

Acknowledgments. The computational results presented have been achieved (in part) using the HPC infrastructure LEO of the University of Innsbruck. We thank Awad H. Al-Mohy for his comments on an early version of the manuscript. We thank the referees for their constructive remarks which helped us to improve the presentation of this paper.

REFERENCES

[1] 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.

[2] A. H. Al-Mohy and N. J. Higham, *Computing the action of the matrix exponential, with an application to exponential integrators*, SIAM J. Sci. Comput., 33 (2011), pp. 488–511, https://doi.org/10.1137/100788860.

[3] 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.

[4] M. Caliari, P. Kandolf, A. Ostermann, and S. Rainer, *The Leja method revisited: Backward error analysis for the matrix exponential*, SIAM J. Sci. Comput., 38 (2016), pp. A1639–A1661, https://doi.org/10.1137/15M1027620.

[5] T. A. Davis and Y. Hu, *The University of Florida Sparse Matrix Collection*, ACM Trans. Math. Software, 38 (2011), 1, https://doi.org/10.1145/2049662.2049663.

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

[7] E. Estrada, D. J. Higham, and N. Hatano, *Communicability and multipartite structures in complex networks at negative absolute temperatures*, Phys. Rev. E, 78 (2008), 026102, https://doi.org/10.1103/PhysRevE.78.026102.

[8] E. Hansen and A. Ostermann, *High order splitting methods for analytic semigroups exist*, BIT, 49 (2009), pp. 527–542, https://doi.org/10.1007/s10543-009-0236-x.

[9] G. I. Hargreaves and N. J. Higham, *Efficient algorithms for the matrix cosine and sine*, Numer. Algorithms, 40 (2005), pp. 383–400, https://doi.org/10.1007/s11075-005-8141-0.

[10] D. J. Higham and N. J. Higham, *MATLAB Guide*, 3rd ed., SIAM, Philadelphia, 2017.

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

[12] N. J. Higham, *Functions of Matrices: Theory and Computation*, SIAM, Philadelphia, 2008, https://doi.org/10.1137/1.9780898717778.

[13] 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.

[14] M. Hochbruck and A. Ostermann, *Exponential integrators*, Acta Numer., 19 (2010), pp. 209–286, https://doi.org/10.1017/S0962492910000048.

[15] J. Kunegis, G. Gröner, and T. Gottron, *Online dating recommender systems: The split-complex number approach*, in Proceedings of the 4th ACM RecSys Workshop on Recommender Systems and the Social Web, RSWeb '12, ACM, New York, 2012, pp. 37–44, https://doi.org/10.1145/2365934.2365942.

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