Computing the Action of Trigonometric and Hyperbolic Matrix Functions

Higham, Nicholas J. and Kandolf, Peter

2016

MIMS EPrint: **2016.40**

Manchester Institute for Mathematical Sciences

School of Mathematics

The University of Manchester

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

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 for example useful in the solution of the Schrödinger equation and finds an application in higher order splitting methods [8].

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

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

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:

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

$$= \begin{bmatrix} \cos(tA)y_0 + A^{-1}\sin(tA)y_0' \\ -A\sin(tA)y_0 + \cos(tA)y_0' \end{bmatrix}.$$

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 the algorithm for the action of the exponential 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 the algorithm of [2], 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]:

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

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

Furthermore, we have

$$(1.3) \qquad e^{iA} = \cos A + i\sin A,$$

which implies that for real A, $\cos A = \operatorname{Re} e^{iA}$ and $\sin A = \operatorname{Im} e^{iA}$. 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 $e^{\beta A}V$ and $e^{-\beta A}V$ simultaneously with $\beta = i$ and $\beta = 1$, using a modification of the algorithm 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 $e^A V$. 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 $\pm iA$ in (1.2), in order to cover all cases we treat βA, where $|\beta| \leq 1$. Consider the matrix

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

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

$$(2.4) \qquad E = \log(e^{-\beta A} r(\beta A)) = \log(I + G),$$

where $\rho(G) < 1$ is assumed for the existence of the logarithm. Our only assumption about r is that $r(X) \to e^X$ as $X \to 0$, which is enough to ensure that $\rho(G) < 1$ for small enough βA.

Exponentiating (2.4), and using the fact that all terms commute (each being a function of A), we obtain

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

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 (2.4), 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

$$(2.5) \qquad \|E\| \leq \sum_{i=\ell}^{\infty} |c_i| \, \|\beta A\|^i =: g(\|\beta A\|),$$

With $\widehat{\theta}$ defined by

$$(2.6) \qquad \widehat{\theta} := \max\{\, \theta : \theta^{-1} g(\theta) \leq \mathrm{tol} \,\},$$

we have the backward error result that $\|\beta A\| \leq \widehat{\theta}$ implies $r(\beta A) = e^{\beta A + E}$, with $\|E\| \leq \mathrm{tol} \|\beta A\|$. 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 $\|\beta A/s\| \leq \widehat{\theta}$, we have

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

The crucial point is that since $g(\|\beta A\|) = g(|\beta| \|A\|) \leq g(\|A\|)$, for all $|\beta| \leq 1$, the parameter s chosen for A can be used for βA. Consequently, the original analysis gives the same bounds for $\pm A$ and $\pm iA$ 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 (2.5) the quantity

$$(2.7) \qquad \alpha_p(X) = \max(d_p, d_{p+1}), \quad d_p = \|X^p\|^{1/p},$$

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 method for computing the action of the matrix exponential we take the truncated Taylor series method of Al-Mohy and Higham [2]. Other algorithms, such as the Leja method presented in [4], can be employed in a similar fashion, though the details will be different.

For the truncated Taylor series method the scaling factor s and the Taylor polynomial degree m_* are determined so that the cost of the algorithm is minimized. We briefly recall some necessary details from [2].

The parameter $\widehat{\theta}$ in (2.6) has a dependence on the polynomial degree $m = \ell - 1$ and the tolerance tol. This results in the cost function

$$(3.8) \qquad C_m(A) = m \max\left\{1, \left\lceil \alpha_p(A)/\widehat{\theta}_m \right\rceil\right\}$$

that measures the number of matrix–vector products, and consequently in [2] the authors derive the optimal choice for m_* as

$$(3.9) \quad C_{m_*}(A) = \min\left\{ m \left\lceil \alpha_p(A)/\widehat{\theta}_m \right\rceil : \ 2 \leq p \leq p_{\max}, \ p(p-1) - 1 \leq m \leq m_{\max} \right\}.$$

Here, $m_{\max}$ is the maximal admissible Taylor polynomial degree and $p_{\max}$ is the maximum value of p such that $p(p-1) \leq m_{\max} + 1$. The parameters m_* and s are determined by the following algorithm, which is [2, Code Fragment 3.1]. The parameters θ_m depend on the tolerance tol and are given in [1, Table 3.1] for IEEE single and double precision arithmetic.

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

This code determines m_* and s given A, tol, $m_{\max}$, and $p_{\max}$. It is assumed that the α_p in (3.9) are estimated using the block 1-norm estimation algorithm of Higham and Tisseur [13] with 2 columns.

1   if $\|A\|_1 \leq 2 \dfrac{\ell}{q} \dfrac{\theta_{m_{\max}}}{m_{\max}} p_{\max}(p_{\max} + 3)$
2       $m_* = \arg\min_{1 \leq m \leq m_{\max}} m \lceil \|A\|_1/\theta_m \rceil$
3       $s = \lceil \|A\|_1/\theta_m \rceil$
4   else
5   Let m_* be the smallest m achieving the minimum in (3.9).
6       $s = \max\{C_{m_*}(A)/m_*, 1\}$
7   end

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

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

that we define precisely below. The exponential computed in line 8 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].

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

For $\tau_1, \ldots, \tau_q \in \mathbb{C}$, $A \in \mathbb{C}^{n \times n}$, $B \in \mathbb{C}^{n \times q}$, 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 $g(B) \approx \left(B + \frac{1}{s}\widetilde{A}BD + \frac{1}{2!s^2}\widetilde{A}^2 BD^2 + \frac{1}{3!s^3}\widetilde{A}^3 BD^3 + \cdots \right) J$, where $\widetilde{A}$ and J are given in the algorithm.

1   $\widetilde{A} = A - \mu I$, where $\mu = \text{trace}(A)/n$
2   $t = \max_k |\tau_k|$
3   if $t\|\widetilde{A}\|_1 = 0$
4       $m_* = 0$, $s = 1$
5   else
6       $[m_*, s] = \text{parameters}(t\widetilde{A}, \text{tol})$ % Algorithm 3.1
7   end
8   $F = B$, $J = e^{\mu D(\tau)/s}$
9   for $k = 1\!:\!s$
10      $c_1 = \|B\|_\infty$
11   for $j = 1\!:\!m_*$
12          $B = \widetilde{A}B(D(\tau)/(sj))$,
13          $c_2 = \|B\|_\infty$
14          $F = F + B$
15   if $c_1 + c_2 \le \text{tol}\|F\|_\infty$, break, end
16          $c_1 = c_2$
17   end
18      $F = FJ$, $B = F$
19 end

Note that for $B = [b_1, b_2]$ and $D(\tau) = \text{diag}(\tau_1, \tau_2)$ we have $g(B) = [g_1, g_2]$ with

$$g_j \approx \left(b_j + \frac{\widetilde{A}}{s}b_j\tau_j + \frac{\widetilde{A}^2}{s^2 2!}b_j\tau_j^2 + \frac{\widetilde{A}^3}{s^3 3!}b_j\tau_j^3 + \cdots \right) e^{\mu\tau_j/s}$$

$$= e^{(A-\mu I)\tau_j/s}e^{\mu\tau_j/s}b_j = e^{A\tau_j/s},$$

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

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

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

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

$$(3.10) \qquad \exp\left(\begin{bmatrix} 0 & t \\ -t & 0 \end{bmatrix}\right) = \begin{bmatrix} \cos t & \sin t \\ -\sin t & \cos t \end{bmatrix}.$$

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*

$$(3.11) \qquad \begin{bmatrix} f_\mathrm{r}, & f_\mathrm{i} \end{bmatrix} = \mathbf{F}\big(D(\mathrm{i}t), A, \begin{bmatrix} b_\mathrm{r}, b_\mathrm{i} \end{bmatrix}\big), \qquad D(\mathrm{i}t) = \begin{bmatrix} 0 & t \\ -t & 0 \end{bmatrix}.$$

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

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

and on collecting terms, applying (3.10) and the addition formulas [12, Thm. 12.1], and recalling that $\widetilde{A} = A - \mu I$, we find that

$$g(B) \approx \begin{bmatrix} \cos\left(\dfrac{t\widetilde{A}}{s}\right) b_\mathrm{r} - \sin\left(\dfrac{t\widetilde{A}}{s}\right) b_\mathrm{i}, & \sin\left(\dfrac{t\widetilde{A}}{s}\right) b_\mathrm{r} + \cos\left(\dfrac{t\widetilde{A}}{s}\right) b_\mathrm{i} \end{bmatrix} \begin{bmatrix} \cos(t\mu/s) & \sin(t\mu/s) \\ -\sin(t\mu/s) & \cos(t\mu/s) \end{bmatrix}$$

$$= \begin{bmatrix} \cos\left(\dfrac{t\widetilde{A}}{s}\right) & -\sin\left(\dfrac{t\widetilde{A}}{s}\right) \\ \sin\left(\dfrac{t\widetilde{A}}{s}\right) & \cos\left(\dfrac{t\widetilde{A}}{s}\right) \end{bmatrix} \begin{bmatrix} b_\mathrm{r} \\ b_\mathrm{i} \end{bmatrix} =: C \begin{bmatrix} b_\mathrm{r} \\ b_\mathrm{i} \end{bmatrix}. \qquad \blacksquare$$

Hence, overall, using (3.10) again,

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

as required. $\qquad\qquad \square$

As a consequence of Lemma 3.1 we can compute, with D defined in (3.11),

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

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

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

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

We compute the matrix exponential J in line 8 in Algorithm 3.2 by making use of (3.10).

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, Sec. 10.7.4] for details of the φ functions). This in return 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 is performed in parallel for all the different values of t and level-3 BLAS routines are 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 4.5 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 sometime required to compute the action of our four functions not for a vector but for 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 behaviour of Algorithm 3.2. All the experiments were carried out in MATLAB R2015a (glnxa64) on a Linux machine and for time measurements only one processor is used.

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 implementation `cosmsinm` from https://github.com/sdrelton/cosm_sinm of the algorithm of [3, Alg. 6.2] for simultaneously computing the matrix sine and cosine.

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. The algorithm can switch between *half, single*, and *double* precision.
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 argument it and $-$it, and (1.2b) is used. The algorithm can switch between *half, single*, and *double* precision.
3. `dense` denotes the use of `cosmsinm` to compute the dense matrices $\cos(tA)$ and $\sin(tA)$ simultaneously before the multiplication with b, This method uses *double* precision only.
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 exponential we use `expmv`. The algorithm can switch between *half, single*, and *double* precision.

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

1. `trighmv` denotes Algorithm 3.2, computing the two functions simultaneously. The algorithm can switch between *half, single*, and *double* precision.
2. `trigh_expmv` denotes the use of `expmv` called twice with $\pm t$ as step arguments. The algorithm can switch between *half, single*, and *double* precision.
3. `expmv_tspan` denotes [2, Alg. 5.2] called with $t_0 = -t$, $q = 1$ and $t_q = t$, as discussed in of Remark 3.3. The algorithm can switch between *half, single*, and *double* precision.
4. `dense` denotes the use of `cosmsinm` to compute simultaneously the dense matrices $\cosh(tA)$ and $\sinh(tA)$ as $\cos(itA)$ and $-i\sin(itA)$, respectively, before the multiplication with b. This method uses *double* precision only.
5. `trigh_block` denotes the use of formula (1.1) with $y_0 = 0$ and $y_0' = b$ where iA is substituted for A. We need one extra matrix–vector product to compute $\sinh(tA)b$. In order to compute the exponential we use `expmv`. The algorithm can switch between *half, single*, and *double* precision.

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, $\|X - \widehat{X}\|_1 / \|X\|_1$. In Example 4.5, $\widehat{X}$ is the computed function and X denotes a reference solution computed with the Multiprecision Computing Toolbox [16] at 100-digit precision. In Examples 4.6 and 4.7 the matrices are too large for multiprecision computations so the reference solution X is taken as that obtained via `cosmsinm`.

Example 4.5 (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
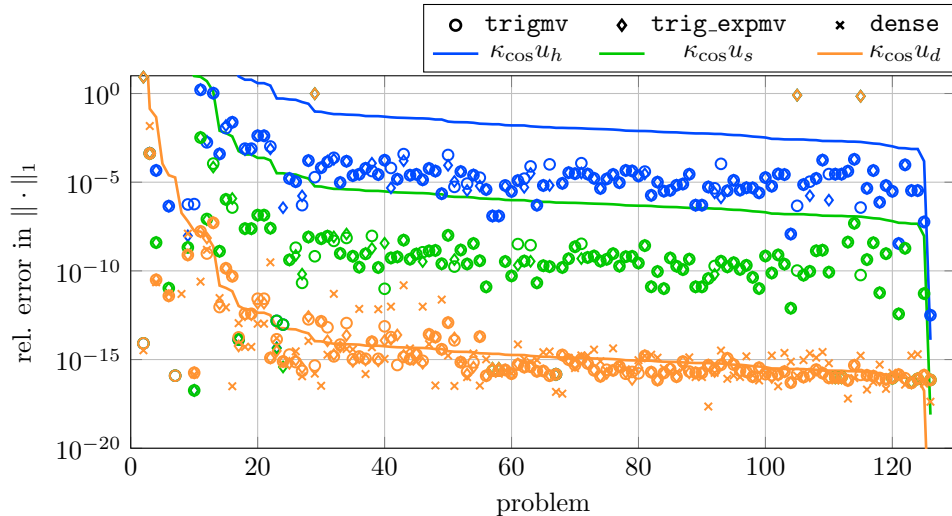
FIG. 4.1. *Relative error in* 1-*norm for computing* $\cos(A)b$ *with three algorithms in* half *(blue)*, single *(green), and* double *(orange) precision. The solid lines correspond to the condition number multiplied by the precision.*
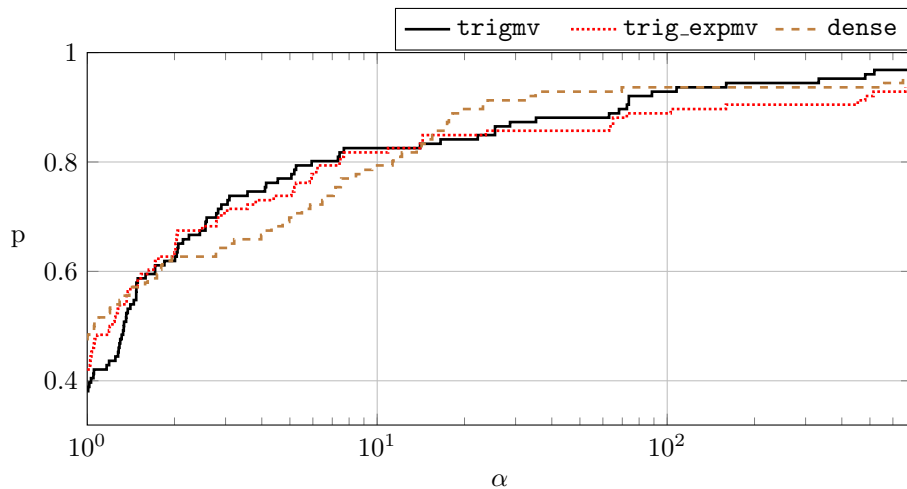

FIG. 4.2. *Same data as in* Figure 4.1 *for the experiment in* double *precision but presented as a performance profile.*

results for sin and sinh are very similar.

As test matrices we use Set 1-3 from [1, Sec. 6], with matrices for which any of the functions overflow removed. 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 different precisions, namely *half*, *single*, and *double*.

The relative errors are shown in Figure 4.1, with the test matrices ordered by decreasing condition number 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, Chap. 3.2].
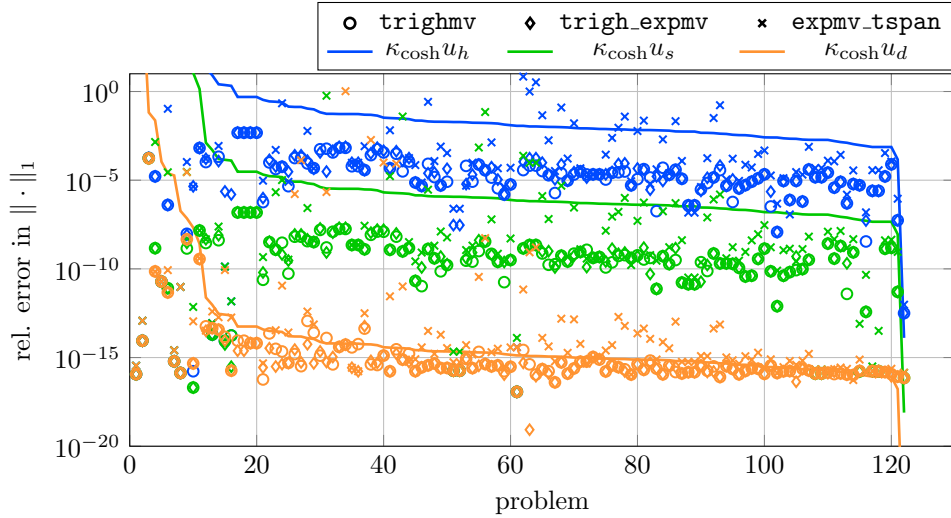
FIG. 4.3. *Relative error in 1-norm for computing* $\cosh(A)b$ *with three algorithms in* half *(blue),* single *(green), and* double *(orange) precision. The solid lines correspond to the condition number multiplied by the precision.*
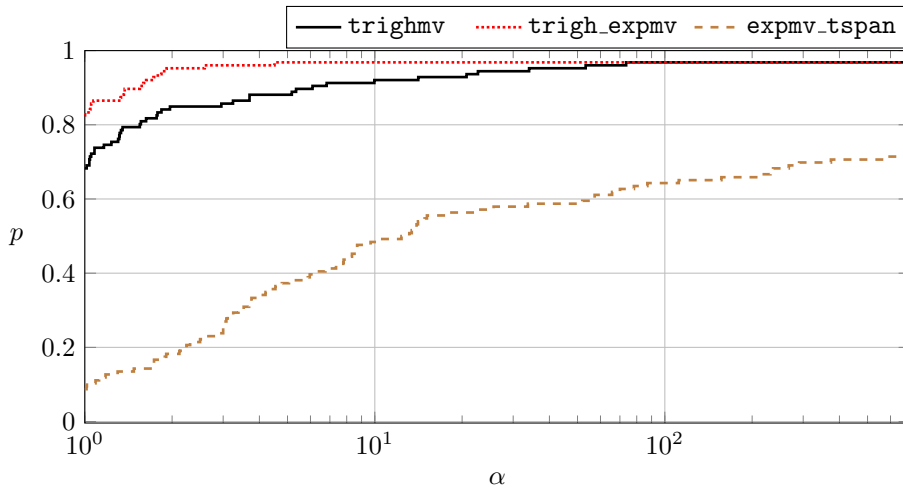

FIG. 4.4. *Same data as in* Figure 4.3 *for the experiment in* double *precision but presented as a performance profile.*

From the error plot in Figure 4.1 one can see that `trigmv` behaves 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 precision. We also show in Figure 4.2 a performance profile for the experiments in double precision. The performance profile is computed with the code from [10, Sec. 22.4] and we employ the idea of [6] to reduce the bias of relative errors significantly less than the precision. The profile suggests that the behavior of the three algorithms is very similar, with no clear winner or loser.

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 then from this compute $b_2 = \mathrm{e}^{2A}b_1$ the result is not always stable, as discussed in Remark 3.3. We see that `trighmv` behaves in a forward stable manner, while `trigh_expmv` has slightly better accuracy than `trighmv`, as made clear by the performance profile in Figure 4.4 (again for double precision).

The behavior of the methods is similar for all three precisions.

Example 4.6 (Behaviour for large matrices). In this numerical experiment we take a closer look at the behaviour of several algorithms for large (sparse) matrices. For the computation of the trigonometric functions we compare `trigmv` against `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 look at `trighmv` in comparison 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 (Algorithm 3.1) step is only performed once.

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 from the MATLAB `gallery` function and 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 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 $[1, \ldots, 1]^{\mathrm{T}}$ for `orani676`, $[1, 0, \ldots, 0, 1]^{\mathrm{T}}$ for `bcspwr10`, the discretization of $256 \cdot x^2(1-x)^2 y^2(1-y)^2$ for `L2`, the discretization of $4096 x^2(1 - x)^2 y^2(1-y)^2 z^2(1-z)^2$ for `S3D`, the discretization of $\exp(-100(x-0.5)^2)$ for `Trans1D`, and $v_i = \cos i$ for all other examples.

Table 4.1 reports the results. The different algorithms are run in *double* precision. 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`.

In Table 4.1a we show the results for the trigonometric functions cos and sin and in Table 4.1b the results for their hyperbolic counterparts. 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 the method `trigmv` always needs the fewest matrix–vector products and with the sole exception of `triw` it is always the fastest method. We can also see that, as expected, the `trig_block` method, 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 obtain $s = 7$, $mv = 618$, and $mvd = 44$ (matrix–vector products performed in the preprocessing

TABLE 4.1

Behaviour of the algorithms for large (sparse) matrices.

	t	trigmv		trig_expmv		trig_block		dense
		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

(a) Results for the computation of cos and sin for large matrices. The tolerance is set to *double*.

	t	trighmv		trigh_expmv		trigh_block		dense
		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

(b) Results for the computation of cosh and sinh for large matrices. The tolerance is set to *double*.

stage, in Algorithm 3.1) for our `trigmv` algorithm. On the other hand, for `trig_block` we get $s = 10$, $mv = 696 \cdot 2 = 1392$, 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 the amount is equal or `trigh_expmv` needs slightly fewer matrix–vector products, `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` matrix is the only case where there is not a speedup of at least a factor of 10. Due to the structure of the `triw` matrix the α_p values, which help to overcome the nonnormality of the matrix, decay very slowly. As a result the estimates of the scaling factor s is a high overestimate and the performance of the algorithms is hindered. 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 4.7 (Schrödinger equation). In this example we solve an evolution equation. We consider the 3D Schrödinger equation with harmonic potential

$$(4.12) \qquad \partial_t u = \frac{\mathrm{i}}{2}\left(\Delta - \frac{1}{2}\left(x^2 + y^2 + z^2\right)\right)u.$$

12

TABLE 4.2

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

	tol $= 2^{-23}$			tol $= 2^{-54}$		
	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 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 $4096x^2(1-x)^2y^2(1-y)^2z^2(1-z)^2$. Therefore, we obtain a discretization matrix iA 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 (4.12) can be interpreted as

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

Table 4.2 reports the results for both single precision and double precision, 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 roughly achieve the same accuracy. We also see that `trigmv` requires significantly fewer matrix–vector products than `trig_expmv` `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 a 1000 times longer for the computation than the designated algorithms.

5. Concluding remarks. We have developed the first algorithms 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 $e^A b$ 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 reducing 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 re-use 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/kandolfp/?

13

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, doi: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, doi: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, doi: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, doi:10.1137/15M1027620.

[5] T. A. DAVIS AND Y. HU, *The University of Florida Sparse Matrix Collection*, ACM Trans. Math. Software, 38 (2011), pp. 1:1–1:25, doi: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), pp. 24:1–24:11, doi:10.1145/2491491.2491494.

[7] E. ESTRADA, D. J. HIGHAM, AND N. HATANO, *Communicability and multipartite structures in complex networks at negative absolute temperatures*, Physical Review E, 78 (2008), p. 026102, doi:10.1103/PhysRevE.78.026102.

[8] E. HANSEN AND A. OSTERMANN, *High order splitting methods for analytic semigroupsexist*, BIT Numerical Mathematics, 49 (2009), pp. 527–542, doi:10.1007/s10543-009-0236-x, http://dx.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, doi:10.1007/s11075-005-8141-0.

[10] D. J. HIGHAM AND N. J. HIGHAM, *MATLAB Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second ed., 2005, doi:10.1137/1.9780898717891.

[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*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008, doi: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, doi:10.1137/S0895479899356080.

[14] M. HOCHBRUCK AND A. OSTERMANN, *Exponential integrators*, Acta Numerica, 19 (2010), pp. 209–286, doi: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 Press, New York, 2012, pp. 37–44, doi:10.1145/2365934.2365942.

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