The University of Manchester

Computing the Action of the Matrix Exponential, with an Application to Exponential Integrators

Awad H. Al-Mohy and Nicholas J. Higham

March 2010

MIMS EPrint: **2010.30**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: `http://www.manchester.ac.uk/mims/eprints`

And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

COMPUTING THE ACTION OF THE MATRIX EXPONENTIAL, WITH AN APPLICATION TO EXPONENTIAL INTEGRATORS*

AWAD H. AL-MOHY† AND NICHOLAS J. HIGHAM‡

Abstract. A new algorithm is developed for computing $e^{tA}B$, where A is an $n \times n$ matrix and B is $n \times n_0$ with $n_0 \ll n$. The algorithm works for any A, its computational cost is dominated by the formation of products of A with $n \times n_0$ matrices, and the only input parameter is a backward error tolerance. The algorithm can return a single matrix $e^{tA}B$ or a sequence $e^{t_k A}B$ on an equally spaced grid of points t_k. It uses the scaling part of the scaling and squaring method together with a truncated Taylor series approximation to the exponential. It determines the amount of scaling and the Taylor degree using the recent analysis of Al-Mohy and Higham [*SIAM J. Matrix Anal. Appl.* 31 (2009), pp. 970–989], which provides sharp truncation error bounds expressed in terms of the quantities $\|A^k\|^{1/k}$ for a few values of k, where the norms are estimated using a matrix norm estimator. Shifting and balancing are used as preprocessing steps to reduce the cost of the algorithm. Numerical experiments show that the algorithm performs in a numerically stable fashion across a wide range of problems, and analysis of rounding errors and of the conditioning of the problem provides theoretical support. Experimental comparisons with MATLAB codes based on Krylov subspace, Chebyshev polynomial, and Laguerre polynomial methods show the new algorithm to be sometimes much superior in terms of computational cost and accuracy. An important application of the algorithm is to exponential integrators for ordinary differential equations. It is shown that the sums of the form $\sum_{k=0}^{p} \varphi_k(A)u_k$ that arise in exponential integrators, where the φ_k are related to the exponential function, can be expressed in terms of a single exponential of a matrix of dimension $n+p$ built by augmenting A with additional rows and columns, and the algorithm of this paper can therefore be employed.

Key words. matrix exponential, Taylor series, ordinary differential equation, ODE, exponential integrator, φ functions, backward error analysis, condition number, overscaling, Krylov method, Chebyshev polynomial, Laguerre polynomial, MATLAB, `expm`

AMS subject classifications. 15A60, 65F30

1. Introduction.

The most popular method for computing the matrix exponential is the scaling and squaring method. For a matrix $A \in \mathbb{C}^{n\times n}$ it exploits the relation $e^A = (e^{2^{-i}A})^{2^i} \approx (r_m(2^{-i}A))^{2^i}$, where r_m is an $[m/m]$ Padé approximant of the exponential. The parameters m and i can be determined so that truncation errors correspond to a backward error no larger than a specified tolerance (for example, the unit roundoff) [2], [11], [13]. In some applications, notably in the numerical solution of ordinary differential equations (ODEs) and in the approximation of dynamical systems [3, Chap. 4], it is not e^A that is required but the action of e^A on a matrix, e^AB, where $B \in \mathbb{C}^{n\times n_0}$ with $n_0 \ll n$, and often $n_0 = 1$, so that B is a vector. The exponential of a sparse matrix is generally full, so when A is large and sparse it is not practical to form e^A and then multiply it into B. Our first contribution in this work is to derive a new algorithm for computing e^AB without explicitly forming e^A. We adapt the scaling and squaring method by computing $e^AB \approx (T_m(s^{-1}A))^sB$, where T_m is a truncated Taylor series rather than a rational approximation (thus avoiding linear system solves), and by carrying out s multiplications of $n \times n$ by $n \times n_0$ matrices instead of $\log_2 s$ squarings of $n \times n$ matrices. We employ several key ideas:

*Version of October 25, 2010.

†Department of Mathematics, King Khalid University, Abha, Saudi Arabia (aalmohy@hotmail.com, http://www.maths.manchester.ac.uk/~almohy).

‡School of Mathematics, The University of Manchester, Manchester, M13 9PL, UK (higham@ma.man.ac.uk, http://www.ma.man.ac.uk/~higham). The work of this author was supported by Engineering and Physical Sciences Research Council grant EP/E050441/1 (CICADA: Centre for Interdisciplinary Computational and Dynamical Analysis).

(1) Careful choice of the parameters m and s, exploiting estimates of $\|A^p\|^{1/p}$ for several p, in order to keep the backward error suitably bounded while minimizing the computational cost.
(2) Shifting, and optional balancing, to reduce the norm of A.
(3) Premature termination of the truncated Taylor series evaluations.

This basic approach is equivalent to applying a Runge–Kutta or Taylor series method with fixed stepsize to the underlying ODE $y'(t) = Ay(t)$, $y(0) = B$, for which $y(t) = e^{tA}B$, which is the sixth of Moler and Van Loan's "19 dubious ways" [19, sec. 4], [29]. However, in (1)–(3) we are fully exploiting the linear nature of the problem in a way that a general purpose ODE integrator cannot. Moreover, our algorithmic parameters are determined by backward error considerations, whereas standard local error control for an ODE solver is forward error-based. We also adapt our method to compute approximations of $e^{t_k A}B$, for t_k equally spaced on an interval $[t_0, t_q]$, in such a way that the phenomenon of overscaling, which has previously afflicted the scaling and squaring method, is avoided no matter how small the stepsize.

Our second contribution concerns the numerical solution of systems of n nonlinear ODEs by exponential integrators. These methods integrate the linear part of the system exactly and approximate the nonlinear part, making use of a set of φ functions closely related to the exponential, evaluated at an $n \times n$ matrix. We show that these methods can be implemented by evaluating a single exponential of an augmented matrix of order $n+p$, where $p-1$ is the degree of the polynomial used to approximate the nonlinear part of the system, thus avoiding the need to compute any φ functions. In fact, on each integration step the integrator is shown to produce the exact solution of an augmented linear system of ODEs of dimension $n + p$. The replacement of φ functions with the exponential is important because it allows existing methods and software for the exponential to be exploited.

The organization of the paper is as follows. In the next section we derive a theorem that shows how to rewrite linear combinations of φ functions of the form required in exponential integrators in terms of a single exponential of a slightly larger matrix. In Section 3 we derive our algorithm for computing $e^A B$ and discuss preprocessing to increase its efficiency. Analysis of the behavior of the algorithm in floating point arithmetic is given in Section 4, where a condition number for the $e^A B$ problem is derived and the algorithm is shown to be numerically stable for Hermitian A. We extend the algorithm in Section 5 to compute $e^{tA}B$ on an equally spaced grid of t values in such a way that overscaling is avoided. Detailed numerical experiments are given in Section 6, including comparison with two Krylov-based codes and codes based on Chebyshev expansions and Laguerre polynomials. We finish with a discussion in Section 7 that assesses the pros and cons of the new method.

2. Exponential integrators: avoiding the φ functions. Exponential integrators are a class of time integration methods for solving initial value problems written in the form

$$u'(t) = Au(t) + g(t, u(t)), \quad u(t_0) = u_0, \quad t \geq t_0, \tag{2.1}$$

where $u(t) \in \mathbb{C}^n$, $A \in \mathbb{C}^{n\times n}$, and g is a nonlinear function. Spatial semidiscretization of partial differential equations leads to systems in this form. The matrix A usually represents the Jacobian of a certain function or an approximation of it, and it is often large and sparse. The solution of (2.1) satisfies the nonlinear integral equation

$$u(t) = e^{(t-t_0)A}u_0 + \int_{t_0}^{t} e^{(t-\tau)A} g(\tau, u(\tau))\, d\tau. \tag{2.2}$$

By expanding g in a Taylor series about t_0, the solution can be written as [18, Lem. 5.1]

$$u(t) = e^{(t-t_0)A}u_0 + \sum_{k=1}^{\infty} \varphi_k\big((t-t_0)A\big)(t-t_0)^k\, u_k, \tag{2.3}$$

where

$$u_k = \frac{d^{k-1}}{dt^{k-1}} g(t,u(t))\,|_{t=t_0}, \quad \varphi_k(z) = \frac{1}{(k-1)!}\int_0^1 e^{(1-\theta)z}\theta^{k-1}\,d\theta, \quad k \geq 1.$$

By suitably truncating the series in (2.3), we obtain the approximation

$$u(t) \approx \widehat{u}(t) = e^{(t-t_0)A}u_0 + \sum_{k=1}^{p} \varphi_k\big((t-t_0)A\big)(t-t_0)^k\, u_k. \tag{2.4}$$

The functions $\varphi_\ell(z)$ satisfy the recurrence relation

$$\varphi_\ell(z) = z\varphi_{\ell+1}(z) + \frac{1}{\ell!}, \quad \varphi_0(z) = e^z,$$

and have the Taylor expansion

$$\varphi_\ell(z) = \sum_{k=0}^{\infty} \frac{z^k}{(k+\ell)!}. \tag{2.5}$$

A wide class of exponential integrator methods is obtained by employing suitable approximations to the vectors u_k in (2.4), and further methods can be obtained by the use of different approximations to g in (2.2). See Hochbruck and Ostermann [17] for a survey of the state of the art in exponential integrators.

We will show that the right-hand side of (2.4) can be represented in terms of the *single* exponential of an $(n+p)\times(n+p)$ matrix, with no need to explicitly evaluate φ functions. The following theorem is our key result. In fact we will only need the special case of the theorem with $\ell = 0$. We denote by I_n the $n \times n$ identity matrix and use MATLAB subscripting notation.

THEOREM 2.1. *Let $A \in \mathbb{C}^{n\times n}$, $W = [w_1, w_2, \ldots, w_p] \in \mathbb{C}^{n\times p}$, $\tau \in \mathbb{C}$, and*

$$\widetilde{A} = \begin{bmatrix} A & W \\ 0 & J \end{bmatrix} \in \mathbb{C}^{(n+p)\times(n+p)}, \quad J = \begin{bmatrix} 0 & I_{p-1} \\ 0 & 0 \end{bmatrix} \in \mathbb{C}^{p\times p}. \tag{2.6}$$

Then for $X = \varphi_\ell(\tau\widetilde{A})$ with $\ell \geq 0$ we have

$$X(1:n, n+j) = \sum_{k=1}^{j} \tau^k \varphi_{\ell+k}(\tau A)\, w_{j-k+1}, \quad j = 1\colon p. \tag{2.7}$$

Proof. It is easy to show that, for $k \geq 0$,

$$\widetilde{A}^k = \begin{bmatrix} A^k & M_k \\ 0 & J^k \end{bmatrix}, \tag{2.8}$$

where $M_k = A^{k-1}W + M_{k-1}J$ and $M_1 = W$, $M_0 = 0$. For $1 \leq j \leq p$ we have $WJ(:,j) = w_{j-1}$ and $JJ(:,j) = J(:,j-1)$, where we define both right-hand sides to

be zero when $j = 1$. Thus

$$\begin{aligned} M_k(:,j) &= A^{k-1}w_j + (A^{k-2}W + M_{k-2}J)J(:,j) \\ &= A^{k-1}w_j + A^{k-2}w_{j-1} + M_{k-2}J(:,j-1) \\ &= \cdots = \sum_{i=1}^{\min(k,j)} A^{k-i}w_{j-i+1}. \end{aligned}$$

We will write $M_k(:,j) = \sum_{i=1}^{j} A^{k-i}w_{j-i+1}$ on the understanding that when $k < j$ we set to zero the terms in the summation where $i > k$ (i.e., those terms with a negative power of A). From (2.5) and (2.8) we see that the (1,2) block of $X = \varphi_\ell(\tau\widetilde{A})$ is

$$X(1:n, n+1:n+p) = \sum_{k=1}^{\infty} \frac{\tau^k M_k}{(k+\ell)!}.$$

Therefore, the $(n+j)$th column of X is given by

$$\begin{aligned} X(1:n, n+j) &= \sum_{k=1}^{\infty} \frac{\tau^k M_k(:,j)}{(k+\ell)!} = \sum_{k=1}^{\infty} \frac{1}{(k+\ell)!} \left(\sum_{i=1}^{j} \tau^i (\tau A)^{k-i} w_{j-i+1} \right) \\ &= \sum_{i=1}^{j} \tau^i \left(\sum_{k=1}^{\infty} \frac{(\tau A)^{k-i}}{(k+\ell)!} \right) w_{j-i+1} \\ &= \sum_{i=1}^{j} \tau^i \left(\sum_{k=0}^{\infty} \frac{(\tau A)^{k}}{(\ell+k+i)!} \right) w_{j-i+1} = \sum_{i=1}^{j} \tau^i \varphi_{\ell+i}(\tau A) w_{j-i+1}. \qquad \square \end{aligned}$$

With $\tau = 1$, $j = p$, and $\ell = 0$, Theorem 2.1 shows that, for arbitrary vectors w_k, the sum of matrix–vector products $\sum_{k=1}^{p} \varphi_k(A)w_{p-k+1}$ can be obtained from the last column of the exponential of a matrix of dimension $n + p$. A special case of the theorem is worth noting. On taking $\ell = 0$ and $W = [c\ 0] \in \mathbb{C}^{n\times p}$, where $c \in \mathbb{C}^n$, we obtain $X(1\colon n, n+j) = \tau^j \varphi_j(\tau A)c$, which is a relation useful for Krylov methods that was derived by Sidje [25, Thm. 1]. This in turn generalizes the expression

$$\exp\left(\begin{bmatrix} A & c \\ 0 & 0 \end{bmatrix} \right) = \begin{bmatrix} e^A & \varphi_1(A)c \\ 0 & 1 \end{bmatrix}$$

obtained by Saad [22, Prop. 1].

We now use the theorem to obtain an expression for (2.4) involving only the matrix exponential. Let $W(:, p-k+1) = u_k$, $k = 1\colon p$, form the matrix $\widetilde{A}$ in (2.6), and set $\ell = 0$ and $\tau = t - t_0$. Then

$$X = \varphi_0\big((t-t_0)\widetilde{A}\big) = e^{(t-t_0)\widetilde{A}} = \begin{bmatrix} e^{(t-t_0)A} & X_{12} \\ 0 & e^{(t-t_0)J} \end{bmatrix}, \tag{2.9}$$

where the columns of X_{12} are given by (2.7), and, in particular, the last column of X_{12} is

$$X(1:n, n+p) = \sum_{k=1}^{p} \varphi_k\big((t-t_0)A\big)(t-t_0)^k u_k.$$

Hence, by (2.4) and (2.9),

$$\begin{aligned}\widehat{u}(t) &= e^{(t-t_0)A}u_0 + \sum_{k=1}^{p} \varphi_k\big((t-t_0)A\big)(t-t_0)^k\, u_k \\ &= e^{(t-t_0)A}u_0 + X(1:n, n+p) \\ &= \begin{bmatrix} I_n & 0 \end{bmatrix} e^{(t-t_0)\widetilde{A}} \begin{bmatrix} u_0 \\ e_p \end{bmatrix}. \end{aligned} \tag{2.10}$$

Thus we are approximating the nonlinear system (2.1) by a subspace of a slightly larger linear system

$$y'(t) = \widetilde{A}y(t), \quad y(t_0) = \begin{bmatrix} u_0 \\ e_p \end{bmatrix}.$$

To evaluate (2.10) we need to compute the action of the matrix exponential on a vector. We focus on this problem in the rest of the paper.

An important practical matter concerns the scaling of $\widetilde{A}$. If we replace W by ηW we see from (2.6) and (2.7) that the only effect on $X = e^{\widetilde{A}}$ is to replace $X(1:n, n+1:n+p)$ by $\eta X(1:n, n+1:n+p)$. This linear relationship can also be seen using properties of the Fréchet derivative [12, Thm. 4.12]. For methods employing a scaling and squaring strategy a large $\|W\|$ can cause overscaling, resulting in numerical instability. (See Section 5 for a discussion of overscaling.) To avoid overscaling a suitable normalization of W is necessary. In the 1-norm we have

$$\|A\|_1 \le \|\widetilde{A}\|_1 \le \max\big(\|A\|_1, \eta\|W\|_1 + 1\big),$$

since $\|J\|_1 = 1$. We choose $\eta = 2^{-\lceil \log_2(\|W\|_1)\rceil}$, which is defined as a power of 2 to avoid the introduction of rounding errors. The variant of the expression (2.10) that we should evaluate is

$$\widehat{u}(t) = \begin{bmatrix} I_n & 0 \end{bmatrix} \exp\left((t-t_0)\begin{bmatrix} A & \eta W \\ 0 & J \end{bmatrix}\right) \begin{bmatrix} u_0 \\ \eta^{-1}e_p \end{bmatrix}. \tag{2.11}$$

Experiment 10 in Section 6 illustrates the importance of normalizing W.

3. Computing $e^{\boldsymbol{A}}\boldsymbol{B}$. Let r_m be a rational approximation to the exponential function, which we assume to be good near the origin, and let $A \in \mathbb{C}^{n\times n}$ and $B \in \mathbb{C}^{n\times n_0}$ with $n_0 \ll n$. Choose an integer $s \ge 1$ so that $e^{s^{-1}A}$ is well-approximated by $r_m(s^{-1}A)$. Exploiting the relation

$$e^A B = (e^{s^{-1}A})^s B = \underbrace{e^{s^{-1}A}e^{s^{-1}A}\cdots e^{s^{-1}A}}_{s \text{ times}} B, \tag{3.1}$$

the recurrence

$$B_{i+1} = r_m(s^{-1}A)B_i, \quad i = 0\colon s-1, \qquad B_0 = B \tag{3.2}$$

yields the approximation $B_s \approx e^A B$. Since A is possibly large and sparse we wish to assume only the capability to evaluate matrix products with A. Note that throughout the paper, "matrix product" refers to the product of an $n \times n$ matrix with an $n \times n_0$ matrix, and this reduces to a matrix–vector product when $n_0 = 1$.

We choose for r_m a truncated Taylor series

$$T_m(s^{-1}A) = \sum_{j=0}^{m} \frac{(s^{-1}A)^j}{j!}, \tag{3.3}$$

because this allows us to determine an optimal choice of s and m by exploiting the backward error analysis of Higham [11], [13], as refined by Al-Mohy and Higham [2]. Let

$$\Omega_m = \{\, X \in \mathbb{C}^{n\times n} : \rho(e^{-X}T_m(X) - I) < 1 \,\},$$

where ρ is the spectral radius. Then the function

$$h_{m+1}(X) = \log(e^{-X}T_m(X)) \tag{3.4}$$

is defined for $X \in \Omega_m$, where log denotes the principal logarithm [12, Thm. 1.31], and it commutes with X. Hence for $X \in \Omega_m$ we have $T_m(X) = e^{X+h_{m+1}(X)}$. Now choose s so that $s^{-1}A \in \Omega_m$. Then

$$T_m(s^{-1}A)^s = e^{A+sh_{m+1}(s^{-1}A)} =: e^{A+\Delta A}, \tag{3.5}$$

where the matrix $\Delta A = sh_{m+1}(s^{-1}A)$ represents the backward error resulting from the truncation errors in approximating e^A by $T_m(s^{-1}A)^s$. Over Ω_m, the functions h_{m+1} have a power series expansion

$$h_{m+1}(X) = \sum_{k=m+1}^{\infty} c_k X^k.$$

We want to ensure that

$$\frac{\|\Delta A\|}{\|A\|} = \frac{\|h_{m+1}(s^{-1}A)\|}{\|s^{-1}A\|} \le \text{tol},$$

for any matrix norm and a given tolerance, tol. By [2, Thm. 4.2(a)] we have

$$\frac{\|\Delta A\|}{\|A\|} = \frac{\|h_{m+1}(s^{-1}A)\|}{\|s^{-1}A\|} \le \frac{\widetilde{h}_{m+1}(s^{-1}\alpha_p(A))}{s^{-1}\alpha_p(A)}, \tag{3.6}$$

where $\widetilde{h}_{m+1}(x) = \sum_{k=m+1}^{\infty} |c_k|\, x^k$ and

$$\alpha_p(A) = \max\bigl(d_p, d_{p+1}\bigr), \qquad d_p = \|A^p\|^{1/p}, \tag{3.7}$$

with p arbitrary subject to $m+1 \ge p(p-1)$. The reason for working with $\alpha_p(A)$ instead of $\|A\|$ is that $\alpha_p(A) \ll \|A\|$ is possible for nonnormal A, so (3.6) is sharper than the bound $\widetilde{h}_{m+1}(s^{-1}\|A\|)/(s^{-1}\|A\|)$. For example, consider

$$A = \begin{bmatrix} 1 & a \\ 0 & -1 \end{bmatrix}, \quad |a| \gg 1, \qquad \|A^{2k}\|_1 = 1, \quad \|A^{2k+1}\|_1 = 1 + |a|, \tag{3.8}$$

for which $d_j = \|A^j\|_1^{1/j} \ll \|A\|_1$ for $j \ge 2$.

Define

$$\theta_m = \max\{\, \theta : \widetilde{h}_{m+1}(\theta)/\theta \le \text{tol} \,\}. \tag{3.9}$$

Then for any m and p with $m+1 \geq p(p-1)$ we have $\|\Delta A\| \leq \mathrm{tol}\|A\|$ provided that $s \geq 1$ is chosen so that $s^{-1}\alpha_p(A) \leq \theta_m$. For each m, the optimal value of the integer s is given by $s = \max\bigl(\lceil \alpha_p(A)/\theta_m \rceil, 1\bigr)$.

The computational cost of evaluating $B_s \approx e^A B$ by the recurrence (3.2) with $r_m = T_m$ is $C_m(A)$ matrix products, where

$$C_m(A) := sm = m\max\bigl(\lceil \alpha_p(A)/\theta_m \rceil, 1\bigr). \tag{3.10}$$

Here, we are assuming that (3.2) is evaluated by explicit formation of the matrices $A^k B_i$ [12, Alg. 4.3].

Note that this approach, based on the recurrence (3.2), is related to the scaling and squaring method for computing e^A in that it shares the same form of approximation and backward error analysis, but it does not exploit repeated squaring in the final phase. In the case where $s = 2^k$ and $n_0 = 1$, (3.2) employs $2^k m$ matrix–vector products whereas the scaling and squaring method uses k matrix–*matrix* products in the squaring phase.

The sequence $\{C_m(A)\}$ is found to be generally decreasing, though it is not necessarily monotonic. Indeed the sequence $\{\alpha_p(A)\}$ has a generally nonincreasing trend for any A (see [2, sec. 1] and in particular Figure 1.1 therein), and with tol in (3.9) corresponding to single or double precision we find that $\{m/\theta_m\}$ is strictly decreasing. Thus the larger is m, the less the cost. However, a large value of m is generally unsuitable in floating point arithmetic because it leads to the evaluation of $T_m(A)B$ with a large $\|A\|$, and as the analysis in the next section explains, numerical instability may result. Thus we impose a limit $m_{\max}$ on m and obtain the minimizer m_* over all p such that $p(p-1) \leq m_{\max}+1$. For the moment we drop the max in (3.10), whose purpose is simply to cater for nilpotent A with $A^j = 0$ for $j \geq p$. Thus we have

$$C_m(A) = m\lceil \alpha_p(A)/\theta_m \rceil.$$

Note that $d_1 \geq d_k$ in (3.7) for all $k \geq 1$ and so $\alpha_1(A) \geq \alpha_2(A)$. Hence we do not need to consider $p = 1$. Let $p_{\max}$ denote the largest positive integer p such that $p(p-1) \leq m_{\max}+1$. Then the optimal cost is

$$C_{m_*}(A) = \min\bigl\{\, m\lceil \alpha_p(A)/\theta_m \rceil : 2 \leq p \leq p_{\max},\ \ p(p-1)-1 \leq m \leq m_{\max} \,\bigr\}, \tag{3.11}$$

where m_* denotes the smallest value of m at which the minimum is attained. The optimal scaling parameter is $s = C_{m_*}(A)/m_*$, by (3.10). Our experience indicates that $p_{\max} = 8$ and $m_{\max} = 55$ are appropriate choices. The above error and cost analysis are valid for any matrix norm, but it is most convenient to use the 1-norm. As we did in [2], we will use the block 1-norm estimation algorithm of Higham and Tisseur [15] to approximate the quantities $d_p = \|A^p\|_1^{1/p}$ needed to evaluate $\alpha_p(A)$. This algorithm estimates $\|G\|_1$ via about 2 actions of G and 2 actions of G^*, all on matrices of ℓ columns, where the positive integer ℓ is a parameter (typically set to 1 or 2). Therefore computing $\alpha_p(A)$ for $p = 2\colon p_{\max}$, and thus d_p for $p = 2\colon p_{\max}+1$, costs approximately

$$4\ell \sum_{p=2}^{p_{\max}+1} p = 2\ell p_{\max}(p_{\max}+3) \tag{3.12}$$

matrix–vector products. If

$$\|A\|_1 \leq 2\frac{\ell}{n_0}\frac{\theta_{m_{\max}}}{m_{\max}} p_{\max}(p_{\max}+3) \tag{3.13}$$

TABLE 3.1
Selected constants θ_m for $\mathrm{tol} = 2^{-24}$ (*single precision*) *and* $\mathrm{tol} = 2^{-53}$ (*double*).

m	5	10	15	20	25	30	35	40	45	50	55
single	1.3e-1	1.0e0	2.2e0	3.6e0	4.9e0	6.3e0	7.7e0	9.1e0	1.1e1	1.2e1	1.3e1
double	2.4e-3	1.4e-1	6.4e-1	1.4e0	2.4e0	3.5e0	4.7e0	6.0e0	7.2e0	8.5e0	9.9e0

then the cost—namely $n_0 m_{\max}\|A\|_1/\theta_{m_{\max}}$ matrix–vector products—of evaluating B_s with m determined by using $\|A\|_1$ in place of $\alpha_p(A)$ in (3.10) is no larger than the cost (3.12) of computing the $\alpha_p(A)$, and so we should certainly use $\|A\|_1$ in place of the $\alpha_p(A)$. This observation leads to a significant reduction in cost for some matrices. See Experiments 1, 2, and 9 in Section 6 for examples where (3.13) is satisfied. Thus m and s are determined as follows.

CODE FRAGMENT 3.1 ($[m_*, s] = \mathrm{parameters}(A, \mathrm{tol})$). *This code determines m_* and s given A,* tol*, $m_{\max}$, and $p_{\max}$.*

```
if (3.13) is satisfied
   m_* = argmin_{1≤m≤m_max} m⌈‖A‖_1/θ_m⌉
   s = ⌈‖A‖_1/θ_m⌉
else
   Let m_* be the smallest m achieving the minimum in (3.11).
   s = max(C_{m_*}(A)/m_*, 1)
end
```

If we wish to exponentiate the matrix tA for several values of t then, since $\alpha_p(tA) = |t|\alpha_p(A)$, we can precompute the matrix $S \in \mathbb{R}^{(p_{\max}-1)\times m_{\max}}$ given by

$$s_{pm} = \begin{cases} \dfrac{\alpha_p(A)}{\theta_m}, & 2 \le p \le p_{\max},\ \ p(p-1)-1 \le m \le m_{\max}, \\ 0, & \text{otherwise} \end{cases} \tag{3.14}$$

and then for each t obtain $C_{m_*}(tA)$ as the smallest nonzero element in the matrix $\lceil |t|S \rceil \mathrm{diag}(1, 2, \ldots, m_{\max})$, where m_* is the column index of the smallest element. Table 3.1 lists some of the θ_m values corresponding to $u_s = \mathrm{tol} = 2^{-24} \approx 6.0 \times 10^{-8}$ (single precision) and $u_d = \mathrm{tol} = 2^{-53} \approx 1.1 \times 10^{-16}$ (double precision). These values were determined as described in [14, App.].

3.1. Preprocessing and termination criterion. Further reduction of the scaling parameter s can be achieved by choosing an appropriate point about which to expand the Taylor series of the exponential function. For any $\mu \in \mathbb{C}$, both the series $\sum_{k=0}^{\infty} A^k/k!$ and $e^{\mu}\sum_{k=0}^{\infty}(A-\mu I)^k/k!$ yield e^A, but the convergence of the second can be faster if μ is selected so that $\|A - \mu I\| \le \|A\|$. Two different ways to approximate e^A via the matrix $A - \mu I$ are from the expressions

$$e^{\mu}\big[T_m(s^{-1}(A-\mu I))\big]^s, \qquad \big[e^{\mu/s}T_m(s^{-1}(A-\mu I))\big]^s.$$

These two expressions are not equivalent numerically. The first is prone to overflow when A has an eigenvalue with large negative real part [12, sec. 10.7.3], since it explicitly approximates $e^{A-\mu I}$. The second expression avoids this problem and is therefore preferred.

Since we will base our algorithm on the 1-norm, the most natural choice of μ is the one that minimizes $\|A - \mu I\|_1$, for which an explicit expression is given in [12, Thm. 4.21]. However, it is the values $d_p(A) = \|A^p\|_1^{1/p}$ in (3.7), not $\|A\|_1$, that

govern the construction of our approximation, and choosing the shift to minimize $\|A-\mu I\|_1$ does not necessarily produce the smallest values of $d_p(A-\mu I)$. Indeed we have found empirically that the shift that minimizes the Frobenius norm $\|A-\mu I\|_F$, namely $\mu = \mathrm{trace}(A)/n$, leads to smaller values of the d_p for the 1-norm. A partial explanation follows from the observation that if $A = QTQ^*$ is a Schur decomposition then $(A-\mu I)^p = Q(T-\mu I)^p Q^*$. Hence if $A-\mu I$ has zero trace then $T-\mu I$ has diagonal elements with both positive and negative real parts, and this tends to result in cancellation of any large off-diagonal elements when the matrix is powered, as illustrated by (3.8).

Importantly, incorporating shifts does not vitiate the backward error analysis above: if we choose m_* based on the $\alpha_p(A-\mu I)$ values, the same backward error bounds can be shown to hold.

Another way to reduce the norm is by balancing. Balancing is a heuristic that attempts to equalize the norms of the ith row and ith column of A, for each i, by a diagonal similarity transformation, $\widetilde{A} = D^{-1}AD$. The balancing algorithm available in LAPACK and MATLAB uses the 1-norm and also attempts to permute the matrix to block upper triangular form, something that is important for the eigenvalue problem but not relevant to the computation of $e^A B$. With balancing, we compute $e^A B = De^{\widetilde{A}}D^{-1}B$. There is no guarantee that $\|\widetilde{A}\|_1 < \|A\|_1$, or that the $\alpha_p(A)$ values are reduced; we would certainly not use balancing if $\|\widetilde{A}\|_1 > \|A\|_1$. Balancing affects the backward error analysis: the best backward error bound for A involves an extra factor $\kappa(D)$, though in practice this factor is not seen (see Experiment 1 in Section 6). In the context of the eigenvalue problem it is known that balancing can seriously degrade accuracy in special cases [28]. We regard balancing as an option to be used with care and not always to be automatically applied.

The derivation of the θ_m takes no account of the matrix B, so our choice of m is likely to be larger than necessary for some B. We know that our procedure returns $e^{A+\Delta A}B$ with normwise relative backward error $\|\Delta A\|/\|A\| \le \mathrm{tol}$. We now consider truncating the evaluation of $T_m(A)B_i$ in (3.2), and in so doing allow a normwise relative forward error of at most tol to be introduced. With A denoting the scaled and shifted matrix, we will accept $T_k(A)B_i$ for the first k such that

$$\frac{\|A^{k-1}B_i\|}{(k-1)!} + \frac{\|A^k B_i\|}{k!} \le \mathrm{tol}\,\|T_k(A)B_i\|. \tag{3.15}$$

The left-hand side of (3.15) is meant to approximate the norm of the tail of the series, $\sum_{j=k+1}^{m} A^j B_i/j!$. Taking two terms rather than one better captures the behavior of the tail, as illustrated by (3.8); we have found empirically that the use of two terms gives reliable performance.

Our algorithm for computing $e^{tA}B$, where the scalar parameter t is now included for convenience, is summarized as follows. The algorithm is intended for use with $\mathrm{tol} = u_s$ or $\mathrm{tol} = u_d$, for which we know the corresponding θ_m values (see Table 3.1). However, it is straightforward to determine (once and for all) the θ_m corresponding to any other value of tol.

ALGORITHM 3.2 ($F = \mathbf{F}(t, A, B, \mathrm{balance})$). *Given $A \in \mathbb{C}^{n\times n}$, $B \in \mathbb{C}^{n\times n_0}$, $t \in \mathbb{C}$, and a tolerance* tol*, this algorithm produces an approximation $F \approx e^{tA}B$. The logical variable* balance *indicates whether or not to apply balancing.*

1 if balance
2 $\quad \widetilde{A} = D^{-1}AD$
3 $\quad$ if $\|\widetilde{A}\|_1 < \|A\|_1$, $A = \widetilde{A}$, $B = D^{-1}B$, else balance = false, end

```
4  end
5  μ = trace(A)/n
6  A = A − μI
7  if t‖A‖_1 = 0
8     m_* = 0, s = 1   % The case tA = 0.
9  else
10    [m_*, s] = parameters(tA, tol) % Code Fragment 3.1
11 end
12 F = B, η = e^{tμ/s}
13 for i = 1: s
14       c_1 = ‖B‖_∞
15       for j = 1: m_*
16            B = tAB/(sj), c_2 = ‖B‖_∞
17            F = F + B
18            if c_1 + c_2 ≤ tol‖F‖_∞, break, end
19            c_1 = c_2
20       end
21       F = ηF, B = F
22 end
23 if balance, F = DF, end
```

The cost of the algorithm is determined by the number of matrix products; these products occur at line 16 and in the parameters function.

Note that when $n_0 > 1$ we could simply invoke Algorithm 3.2 n_0 times to compute $e^A b_j$, $j = 1\colon n_0$, which may require fewer flops than a single invocation of $e^A B$, since the termination test at line 18 may be satisfied earlier for some b_j than for B as whole. The advantage of working with B is the ability to invoke level 3 BLAS [7] [8], which should lead to faster execution.

4. Rounding error analysis and conditioning. To assess the numerical stability of Algorithm 3.2 in floating point arithmetic we analyze the rounding errors in forming the product $T_m(A)B$, where $T_m(A)$ is the truncated Taylor series (3.3). For simplicity we assume that A does not need scaling (that is, $s = 1$). We then determine the conditioning of the problem and see if our forward error bound reflects the conditioning. We know that $T_m(A) = e^{A+E}$ with $\|E\| \le \mathrm{tol}\|A\|$. The analysis includes two parameters: the backward error in the approximation of the exponential, tol, and the precision of the underlying floating point arithmetic, u. The norm is the 1-norm or the ∞-norm.

LEMMA 4.1. *Let $X = T_m(A)B$ be formed as in Algorithm* 3.2*, but for simplicity ignoring lines* 5*,* 6*, and* 18*, and assume that $s = 1$ in that algorithm with tolerance* tol*. Then the computed $\widehat{X}$ in floating point arithmetic with unit roundoff $u \le$ tol satisfies $\widehat{X} = e^{A+E}B + R$, where $\|E\| \le \mathrm{tol}\|A\|$ and $\|R\| \le \widetilde{\gamma}_{mn}\, T_m(\|A\|)\|B\| \le \widetilde{\gamma}_{mn}\, e^{\|A\|}\|B\|$.*

Proof. Standard error analysis for the evaluation of matrix products [10, sec. 3.5] yields $\|X - \widehat{X}\| \le \widetilde{\gamma}_{mn}\, T_m(\|A\|)\|B\|$. The analysis in Section 3 shows that $X = e^{A+E}B$, with $\|E\| \le \mathrm{tol}\|A\|$. The result follows on using $T_m(\|A\|) \le e^{\|A\|}$. ∎

Lemma 4.1 shows that $\widehat{X}$ satisfies a mixed forward–backward error result where the normwise relative backward error bound is tol and the forward error bound is a multiple of $ue^{\|A\|}\|B\|$. Since $\|A\|$ can exceed 1 the forward error bound is potentially large. To judge whether the forward error bound is acceptable we compare it with a perturbation analysis for the problem.

We derive a perturbation result for the product $X = f(A)B$, where f is an arbitrary matrix function and then specialize it to the exponential. We denote by $L_f(A, \Delta A)$ the Fréchet derivative of f at A in the direction ΔA [12, sec. 3.1]. Let vec denote the operator that stacks the columns of its matrix argument into a long vector. We will use the fact that $\mathrm{vec}(L_f(A, \Delta A)) = K_f(A)\mathrm{vec}(\Delta A)$, with $K_f(A)$ the $n^2 \times n^2$ Kronecker matrix representation of the Fréchet derivative [12, sec. 3.2]. The following lemma makes no assumption about $\|A\|$.

LEMMA 4.2. *Let $X = f(A)B$ and $X + \Delta X = f(A + \Delta A)(B + \Delta B)$ both be defined, where $\|\Delta A\|_F \le \epsilon\|A\|_F$ and $\|\Delta B\|_F \le \epsilon\|B\|_F$. Then, assuming that f is Fréchet differentiable at A,*

$$\frac{\|\Delta X\|_F}{\|X\|_F} \le \epsilon\left(\frac{\|f(A)\|_2\|B\|_F}{\|X\|_F} + \frac{\|(B^T \otimes I)K_f(A)\|_2\|A\|_F}{\|X\|_F}\right) + o(\epsilon), \tag{4.1}$$

and this bound is attainable to within a factor 2 to first order in ϵ.

Proof. We have

$$X + \Delta X = \big(f(A) + L_f(A, \Delta A) + o(\|\Delta A\|_F)\big)(B + \Delta B).$$

Applying the vec operator, and using the fact that $\mathrm{vec}(UV) = (V^T \otimes I)\mathrm{vec}(U)$, gives

$$\mathrm{vec}(\Delta X) = \mathrm{vec}(f(A)\Delta B) + (B^T \otimes I)K_f(A)\mathrm{vec}(\Delta A) + o(\epsilon).$$

Taking the 2-norm and exploiting the relation $\|\mathrm{vec}(X)\|_2 = \|X\|_F$ we have

$$\|\Delta X\|_F \le \epsilon\|f(A)\|_2\|B\|_F + \epsilon\|(B^T \otimes I)K_f(A)\|_2\|A\|_F + o(\epsilon),$$

which is equivalent to (4.1). Since ΔB and ΔA are arbitrary it is clear that $\|\Delta X\|_F$ can attain each of the first two terms in the latter bound with only one of ΔB and ΔA nonzero, and hence the bound is attainable to within a factor 2 to first order. □

In view of the lemma we can regard

$$\kappa_f(A, B) := \frac{\|f(A)\|_2\|B\|_F}{\|X\|_F} + \frac{\|(B^T \otimes I)K_f(A)\|_2\|A\|_F}{\|X\|_F} \tag{4.2}$$

as a condition number for the $f(A)B$ problem. We can weaken this expression to

$$\kappa_f(A, B) \le \frac{\|f(A)\|_F\|B\|_F}{\|X\|_F}(1 + \kappa_f(A)), \tag{4.3}$$

where [12, secs. 3.1, 3.4]

$$\kappa_f(A) := \frac{\|L_f(A)\|_F\|A\|_F}{\|f(A)\|_F}, \qquad \|L_f(A)\|_F := \max_{Z\neq 0}\frac{\|L_f(A, Z)\|_F}{\|Z\|_F} = \|K_f(A)\|_2. \tag{4.4}$$

Applying (4.3) with f the exponential gives

$$\kappa_{\exp}(A, B) \le \frac{\|e^A\|_F\|B\|_F}{\|X\|_F}(1 + \kappa_{\exp}(A)). \tag{4.5}$$

For comparison with Lemma 4.1 we will, for simplicity, replace all norms by the 2-norm, since constant factors are not important. For the condition number in the 2-norm we have [12, Lem. 10.15]

$$\|A\|_2 \le \kappa_{\exp}(A) \le \frac{e^{\|A\|_2}\|A\|_2}{\|e^A\|_2}. \tag{4.6}$$

If $\kappa_{\exp}(A)$ is close to its upper bound in (4.6) then the forward error bound from Lemma 4.1 is smaller than the bound for $\kappa_{\exp}(A,B)$ in (4.5) by a factor $\|A\|_2$. However, there is equality in the lower bound (4.6) for normal A [12, Lem. 10.16], [27]. So for normal A, the normwise relative forward error bound for $\|R\|_2/\|X\|_2$ from Lemma 4.1 exceeds $\kappa_{\exp}(A,B)u$ by about

$$e^{\|A\|_2}/(\|e^A\|_2(1+\|A\|_2)), \tag{4.7}$$

which ranges between $1/(1+\|A\|_2)$ and $e^{2\|A\|_2}/(1+\|A\|_2)$. For Hermitian A, the upper bound is attained when A is negative semidefinite. However, when we apply this analysis to Algorithm 3.2, A refers to the *shifted* matrix $A-(\mathrm{trace}(A)/n)I$, which has extremal eigenvalues of equal magnitude and opposite signs, and so (4.7) always attains its lower bound. Thus for Hermitian matrices our shifting strategy ensures stability.

An example is instructive. Let $A = \mathrm{diag}(-20.5,-1)$ and $B = [1\ 1]^T$. With $x = e^Ab$ and $\widehat{x}$ the computed product from Algorithm 3.2 with $\mathrm{tol} = u_d$, we find that

$$\frac{\|x-\widehat{x}\|_2}{\|x\|_2} = 6.0\times 10^{-16}, \qquad \frac{|x_1-\widehat{x}_1|}{|x_1|} = 2.6\times 10^{-9}, \quad \frac{|x_2-\widehat{x}_2|}{|x_2|} = 6.0\times 10^{-16}.$$

Since $\widetilde{A} := A - \mathrm{trace}(A)/2 = \mathrm{diag}(-9.75, 9.75)$, Algorithm 3.2 takes $s = 1$ and therefore evaluates a truncated Taylor series for the unscaled matrix $\widetilde{A}$. This leads to substantial cancellation in the first component, since the terms in Taylor series for $e^{-9.75}$ grow substantially before they decay, but the second component is computed accurately. While there is a loss of accuracy in the smaller component, in the normwise sense the computation is entirely satisfactory, as the analysis above predicts, and normwise stability is all we can expect of an algorithm designed for general A.

The conclusion from this analysis is that, while it is desirable to keep $\|A\|$ small in order to ensure that Algorithm 3.2 reflects the conditioning of the problem, a large $\|A\|$ does not necessarily imply numerical instability and indeed the algorithm is always stable for Hermitian A.

5. Computing $e^{tA}B$ over a time interval. In practice, it may be required to evaluate $e^{tA}B$ for several values of t belonging to a time interval $[t_0, t_q]$. Suppose that q equally spaced steps are to be taken. Denote the grid points by $t_k = t_0 + kh$, $k = 0\colon q$, where $h = (t_q - t_0)/q$. If $e^{t_0A}B$ is available and Algorithm 3.2, applied to $e^{(t_q-t_0)A}$, selects a scaling parameter s equal to q, then the algorithm automatically generates the required matrices as intermediate quantities, as is clear from (3.1). In general, though, we need an efficient strategy for computing these matrices. The most obvious way to evaluate $B_k = e^{t_kA}B$, $k = 0\colon q$, is directly from the formula, using Algorithm 3.2. However, since the cost of the algorithm is proportional to $\alpha_p(tA) = |t|\alpha_p(A)$, it is more efficient if we reduce each t_k by t_0 by forming $B_0 = e^{t_0A}B$ and then (recall that $\mathbf{F}$ denotes an invocation of Algorithm 3.2)

$$B_k = \mathbf{F}(kh, A, B_0), \qquad k = 1\colon q. \tag{5.1}$$

A further reduction in cost accrues if we obtain each B_k from the preceding one:

$$B_k = \mathbf{F}(h, A, B_{k-1}), \qquad k = 1\colon q. \tag{5.2}$$

In deciding on the best approach we need to consider the effects of rounding errors. We know that the scaling and squaring method for e^A can suffer from overscaling, which occurs when the initial scaling $A \leftarrow 2^{-i}A$ reduces $\|A\|$ by more than is

necessary to achieve the required accuracy and the resulting extra squarings degrade the accuracy due to propagation of rounding errors in the squaring phase [2]. The same danger applies here, but now overscaling can be caused by a too-small step-size h. The danger is illustrated by the example of computing $(1 + x/100)^{100}$ when x is so small that $e^x \approx 1 + x$ is a good enough approximation; the former expression is clearly much more seriously affected by rounding errors than the latter. The gist of the matter can be seen by considering how B_1 and B_2 are computed; both (5.1) and (5.2) compute $B_1 = e^{hA}B_0$, but (5.1) computes $B_2 = e^{2hA}B_0$ while (5.2) uses $B_2 = e^{hA}B_1 = e^{hA}(e^{hA}B_0)$. It may be that $2hA$ needs no scaling (i.e., Algorithm 3.2 chooses $s = 1$ when applied to $2hA$), so that B_1 and B_2 are obtained at exactly the same cost from (5.1) as from (5.2). Although these two ways of obtaining B_2 are equivalent in exact arithmetic, in floating point arithmetic the formula $B_2 = e^{hA}(e^{hA}B_0)$ is more likely to suffer from overscaling.

The following algorithm reduces the chance of overscaling with no cost penalty. It uses the direct formula (5.1) whenever it can do so without increasing the cost (that is, without scaling), but uses (5.2) when (5.1) requires scaling and (5.2) does not.

CODE FRAGMENT 5.1. *This algorithm computes $B_k = e^{t_k A}B$ for $k = 0\colon q$, where $t_k = t_0 + kh$ and $h = (t_q - t_0)/q$, for the case where $q > s_*$, where s_* is the value determined by Algorithm* 3.2 *applied to $(t_q - t_0)A$.*

```
1  s = s_*
2  d = ⌊q/s⌋, j = ⌊q/d⌋, r = q − dj
3  h = (t_q − t_0)/q
4  Z = F(t_0, A, B)     % B_0
5  d̃ = d
6  for i = 1: j + 1
7      if i > j, d̃ = r, end
8      for k = 1: d̃
9          B_{k+(i−1)d} = F(kh, A, Z)
10     end
11     if i ≤ j, Z = B_{id}, end
12 end
```

Note that the same parameter s, namely $s = 1$, is used on each invocation of Algorithm 3.2 on line 9 of Code Fragment 5.1. Some useful computational savings are possible in line 9 by saving and re-using matrix products. We have

$$T_m(kh, A, Z) = \sum_{\ell=0}^{m} \frac{(khA)^\ell Z}{\ell!} = \underbrace{\begin{bmatrix} Z & (hA)Z & \dots & \frac{1}{m!}(hA)^m Z \end{bmatrix}}_{K_m} \begin{bmatrix} 1 \\ k \\ \vdots \\ k^m \end{bmatrix}. \tag{5.3}$$

When invoked at line 9 of Code Fragment 5.1, Algorithm 3.2 generally increases the value of m_* as k increases until m_* reaches its maximal value (not necessarily $m_{\max}$) at $k = \widetilde{d}$. It would be enough to form the matrix K_m for the maximal value of m and reuse it for the smaller values of k. However, this would destroy the computational saving obtained from the stopping test that Algorithm 3.2 uses. Instead, we will build up the required block columns of K_m gradually during the computation by using the stopping test.

With the aid of Code Fragment 5.1 and Algorithm 3.2, we can write the final algorithm. We will denote by $X_{:,j}$ the jth block column of the $n \times n_0(q+1)$ matrix

X partitioned into $n \times n_0$ blocks; if $n_0 = 1$ (so that B is a vector) then $X_{:,j} = X(:,j)$ in the usual notation.

ALGORITHM 5.2. *Given $A \in \mathbb{C}^{n\times n}$, $B \in \mathbb{C}^{n\times n_0}$, and a tolerance* tol*, this algorithm computes a matrix $X \in \mathbb{C}^{n\times n_0(q+1)}$ such that $X_{:,k+1} \approx e^{t_k A}B$, $k = 0\colon q$, where $t_k = t_0 + kh$ and $h = (t_q - t_0)/q$. The logical variable* balance *indicates whether or not to apply balancing.*

```
1  if balance
2    Ã = D^{-1}AD
3    if ‖Ã‖_1 < ‖A‖_1, A = Ã, B = D^{-1}B, else balance = false, end
4  end
5  μ = trace(A)/n
6  [m_*, s] = parameters((t_q − t_0)(A − μI), tol)
7  X_{:,1} = F(t_0, A, B, false)
8  if q ≤ s
9    for k = 1: q
10       X_{:,k+1} = F(h, A, X_{:,k}, false)
11   end
12   if balancing was used, X = DX, end
13   quit
14 end
15 A = A − μI
16 d = ⌊q/s⌋, j = ⌊q/d⌋, r = q − dj, d̃ = d
17 Compute C_{m_*}(dA) from (3.11).
18 Z = X(:, 1)
19 for i = 1: j + 1
20     if i > j, d̃ = r, end
21     K_{:,1} = Z, m̂ = 0
22     for k = 1: d̃
23         F = Z, c_1 = ‖Z‖_∞
24         for p = 1: m_*
25             if p > m̂
26                K_{:,p+1} = hAK_{:,p}/p  % Form K_{:,p+1} if not already formed.
27             end
28             F = F + k^p K_{:,p+1}
29             c_2 = k^p ‖K_{:,p+1}‖_∞
30             if c_1 + c_2 ≤ tol‖F‖_∞, quit, end
31             c_1 = c_2
32         end
33         m̂ = max(m̂, p)
34         X_{:,k+(i−1)d+1} = e^{khμ}F
35     end
36     if i ≤ j, Z = X_{:,id+1}, end
37 end
38 if balance, X = DX, end
```

6. Numerical experiments. We give a variety of numerical experiments to illustrate the efficiency and accuracy of our algorithms. All were carried out in MATLAB R2010a on machines with Core i7 or Core 2 Duo E6850 processors, and errors are computed in the 1-norm, unless stated otherwise.

We compare our codes with four existing codes from the literature. The first two use Krylov techniques along with time-stepping to traverse the interval $[0, t]$. The MATLAB function `expv` of Sidje [24], [25] evaluates $e^{tA}b$ using a Krylov method with a fixed dimension (default $\min(n, 30)$) for the Krylov subspace. The MATLAB function `phipm` of Niesen [20] uses Krylov techniques to compute a sum of the form $\sum_{k=0}^{p} \varphi_k(tA)u_k$. The size of the Krylov subspace is changed dynamically during the integration, and the code automatically recognizes when A is Hermitian and uses the Lanczos process instead of the Arnoldi process. We use both functions with their default parameters unless otherwise stated, except for the convergence tolerance, which varies in our experiments.

The other two codes are from Sheehan, Saad, and Sidje [23]. They use Chebyshev and Laguerre polynomial expansions within a filtered conjugate residual framework. Unlike the Laguerre method, the Chebyshev method requires knowledge of an interval containing the eigenvalues of A. In common with our algorithms, neither of these methods needs to compute inner products (unlike Krylov methods).

We will not attempt to show the benefits of using the $\alpha_p(A)$ in place of $\|A\|_1$, as these have already been demonstrated in [2] for the scaling and squaring algorithm. In all our experiments B is a vector, b, and we set $\ell = 1$ in the block norm estimator.

Experiment 1. The first experiment tests the behavior of Algorithm 3.2 in floating point arithmetic. We use a combination of all four sets of test matrices described in [2, sec. 6], giving 155 matrices in total, with dimensions n up to 50. For each matrix A, and a different randomly generated vector b for each A, we compute $x = e^A b$ in three ways:

- using Algorithm 3.2 with and without balancing, with tol $= u_d$,
- as `expm(A)*b`, where `expm` is the MATLAB function for the matrix exponential, which implements the scaling and squaring algorithm of [11], [13],
- as `expm_new(A)*b`, where `expm_new` implements the improved scaling and squaring algorithm of Al-Mohy and Higham [2].

Figure 6.1 displays the relative errors $\|e^A b - \widehat{x}\|_2/\|e^A b\|_2$, where the "exact" answer is obtained by computing at 100 digit precision with the Symbolic Math Toolbox. The matrices are sorted by decreasing value of the condition number $\kappa_{\exp}(A, b)$ in (4.2), and $\kappa_{\exp}(A, b)u_d$ is shown as a solid line. We compute $\kappa_{\exp}(A, b)$ exactly, using the function `expm_cond` in the Matrix Function Toolbox [9] to obtain the Kronecker matrix representation $K_{\exp}(A)$ of the Fréchet derivative (for larger matrices we can estimate the condition number using an adaptation of the technique of [1, sec. 7]). Figure 6.2 displays the same data as a performance profile, where for a given α the corresponding point on each curve indicates the fraction p of problems on which the method had error at most a factor α times that of the smallest error over all methods in the experiment.

Figure 6.1 reveals that all the algorithms behave in a generally forward stable manner. Figure 6.2 shows that Algorithm 3.2 has the best accuracy, beating both `expm` and the more accurate `expm_new`. Balancing has little effect on the errors, but it can greatly reduce the cost: the quantity "s without balancing divided by s with balancing" had maximum value 1.6×10^4 and minimum value 0.75, with the two values of s differing in 11 cases. The test (3.13) was satisfied in about 77% of the cases.

Experiment 2. In this experiment we take for A the matrix `gallery('lesp',10)`, which is a nonsymmetric tridiagonal matrix with real, negative eigenvalues, and $b_i = i$. We compute $e^{tA}b$ by Algorithm 3.2 for 50 equally spaced $t \in [0, 100]$, with and without balancing, for tol $= u_s$ and tol $= u_d$. The results are shown in Figure 6.3. We see a

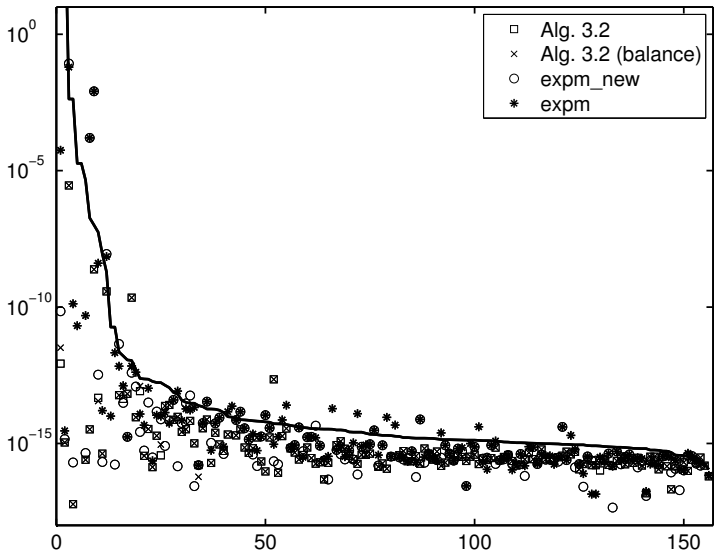

FIG. 6.1. *Experiment* 1*: normwise relative errors in* $e^A b$ *computed by Algorithm* 3.2 *with and without balancing and by first computing* e^A *by* `expm` *or* `expm_new`*. The solid line is* $\kappa_{\exp}(A,b)u_d$*.*

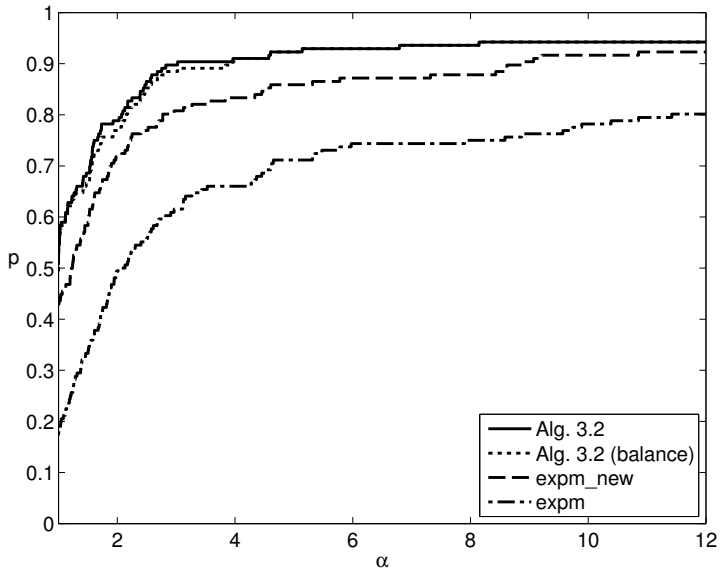

FIG. 6.2. *Same data as in Figure* 6.1 *presented as a performance profile.*

linear growth of the cost of the algorithm with t, as expected. The relative errors are all below the corresponding solid line representing $\kappa_{\exp}(tA,b)u$, which shows that the algorithm is behaving in a forward stable manner. Balancing has no significant effect on the error but leads to a useful reduction in cost. In this test the inequality (3.13) was satisfied in 5% of the cases.

Experiment 3. Now we investigate the effectiveness of Algorithm 5.2 at avoiding overscaling when dense output is requested over an interval. We take for A the

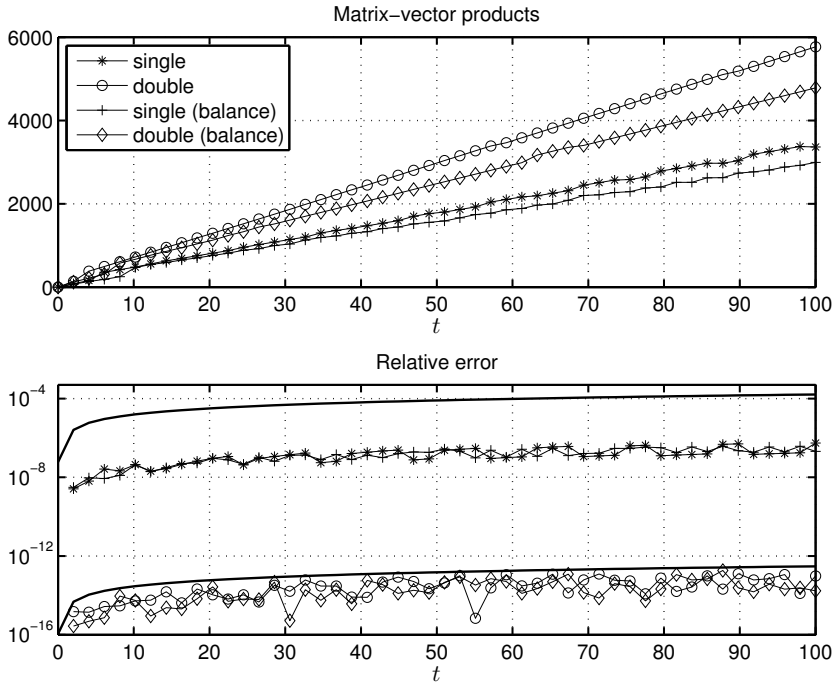

FIG. 6.3. *Experiment* 2*: t versus cost* (*top*) *and accuracy* (*bottom*) *of Algorithm* 3.2 *with and without balancing for $e^{tA}b$. In the bottom plot the solid lines are $\kappa_{\exp}(tA,b)u_s$ and $\kappa_{\exp}(tA,b)u_d$.*

matrix `gallery('frank',3)`, b has equally spaced elements on $[-1,1]$, and $\mathrm{tol} = u_d$; balancing leaves this matrix unchanged. We apply Algorithm 5.2 twice with $t \in [0,10]$ and $q = 200$, once in its given form and again with the "if" test at line 8 forced to be satisfied, thus turning off the logic for avoiding overscaling. The relative errors are shown in Figure 6.4. The improved accuracy provided by our strategy for avoiding overscaling is clear.

Experiment 4. Our next experiment is a variation of one from Trefethen, Weideman, and Schmelzer [26, sec. 3], in which $A \in \mathbb{R}^{9801\times 9801}$ is a multiple of the standard finite difference discretization of the 2D Laplacian, namely the block tridiagonal matrix constructed by `-2500*gallery('poisson',99)` in MATLAB. We compute $e^{\alpha A}b$ for the b specified in [26] with $\alpha = 0.02$ (the problem as in [26]) and $\alpha = 1$. We use four different methods.

1. Algorithm 3.2, with $\mathrm{tol} = u_d$. Since A is symmetric, balancing is not needed.
2. MATLAB functions `expv` and `phipm`, with error tolerance u_d.
3. The MATLAB code in [26, Fig. 4.4], which uses a best rational L_∞ approximation to the exponential of type $(N-1,N)$ with $N = 14$, designed to deliver about double precision accuracy. Unlike the previous two methods, the dominant computational cost is the solution of linear systems with matrices of the form $\alpha I - \beta A$, which is done via the backslash operator.

The results are given in Table 6.1, where "mv" denotes the number of matrix–vector products for the first three methods, "sol" denotes the number of linear system solves for the rational approximation method, and "diff" is the normwise relative difference between the vectors computed by Algorithm 3.2 and the other methods.

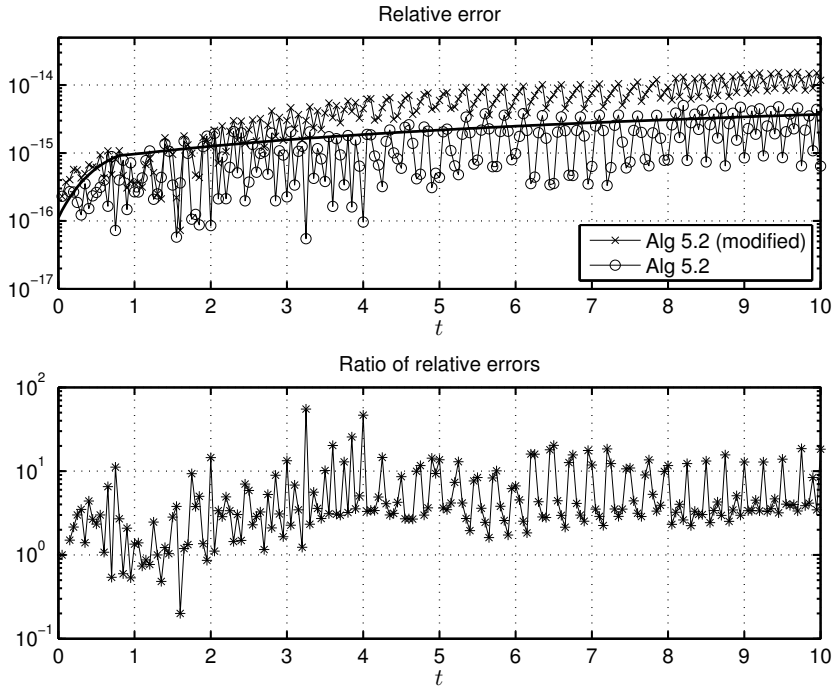

FIG. 6.4. *Experiment* 3*: relative errors* (*top*) *for Algorithm* 5.2 *and for modified version of the algorithm without the logic to avoid overscaling, and ratio of relative errors "modified/original"* (*bottom*)*. The solid line is* $\kappa_{\exp}(tA, b)u_d$.

Here, $\|A\|_1 = 2 \times 10^4$. For $\alpha = 0.02$, Algorithm 3.2 is the fastest of the four methods; it takes $s = 21$. For $\alpha = 1$ a fairly heavy scaling with $s = 1014$ is required and the method is now second slowest. Note that while the rational method is the fastest for $\alpha = 1$ it suffers some loss of accuracy, which can be attributed to the linear systems that it solves having coefficient matrices with condition numbers of order 10^3.

We used the MATLAB `profile` function to profile the M-files in this experiment. We found that Algorithm 3.2 spent 88% of its time doing matrix–vector products. By contrast, `expv` and `phipm` spent 12% and 28% of their time, respectively, on matrix–vector products and most of the rest within the Arnoldi recurrence or in evaluating matrix exponentials via `expm` (1% and 6%, respectively). Note that the vectors in the Arnoldi and Lanczos recurrences are generally full if b is, so the inner products and additions in these recurrences can be of similar cost to a matrix–vector product when A is very sparse, as it is here.

When we modify this experiment to require the values of $e^{\alpha t A}b$ on a grid of 100 equally spaced values of t on $[0, 1]$ and use Algorithm 5.2 in place of Algorithm 3.2 we find that Algorithm 5.2 is the fastest by a substantial margin: the other codes cannot produce intermediate output and so must be called repeatedly to integrate from 0 to t for each grid point t.

Experiment 5. This experiment uses essentially the same tests as Niesen and Wright [21, Experiment 1], which in turn are based on those of Sidje [25]. The three matrices belong to the Harwell-Boeing collection and are obtained from the University of Florida Sparse Matrix Collection [5], [6]. For the first two matrices we compute

TABLE 6.1
Experiment 4: t_{ratio} *denotes time for method divided by time for Algorithm* 3.2.

	Algorithm 3.2		`expv`			`phipm`			rational		
α	t_{ratio}	mv	t_{ratio}	mv	diff	t_{ratio}	mv	diff	t_{ratio}	sol	diff
0.02	1	1010	2.8	403	7.7e-15	1.1	172	3.1e-15	3.8	7	3.3e-14
1.00	1	47702	1.3	8835	4.2e-15	0.2	2504	4.0e-15	0.1	7	1.2e-12

TABLE 6.2
Experiment 5: t_{ratio} *is time for method divided by time for Algorithm* 3.2.

		Algorithm 3.2			`phipm`			`expv`		
	$\|A\|_1$	t_{ratio}	mv	error	t_{ratio}	mv	error	t_{ratio}	mv	error
`orani678`	1.0e5	1	1102	1.0e-14	6.4	651	1.2e-13	22.1	8990	4.0e-14
`bcspwr10`	1.4e2	1	338	4.0e-15	1.1	103	6.6e-15	6.8	341	2.6e-14
`gr_30_30`	3.2e1	1	80	7.2e-9	1.3	40	1.9e-9	3.9	124	9.5e-8

$e^{tA}b$. The matrices and problem details are:

- `orani678`, $n = 2529$, $t = 100$, $b = [1, 1, \ldots, 1]^T$;
- `bcspwr10`, $n = 5300$, $t = 10$, $b = [1, 0, \ldots, 0, 1]^T$.

The third matrix is `gr_30_30`, with $n = 900$, $t = 2$, $b = [1, 1, \ldots, 1]^T$, and we compute $e^{-tA}e^{tA}b$. The tolerance is u_s and for the first two problems we regard the solution computed with Algorithm 3.2 with tol $= u_d$ as the exact solution. Balancing is not applied because MATLAB does not support balancing of matrices stored with the sparse attribute; however, balancing is certainly possible for sparse matrices, and several algorithms are developed by Chen and Demmel [4]. The results are shown in Table 6.2. All three methods deliver the required accuracy, but Algorithm 3.2 proves to be the fastest.

Experiment 6. This example reveals the smoothness of the solution computed by Algorithm 5.2. The matrix A is `-gallery('triw',20,alpha)`, which is upper triangular with constant diagonal -1 and all superdiagonal elements equal to $-\alpha$. We take $b_i = \cos i$ and compute the norms $\|e^{tA}b\|_2$ with tol $= u_d$ for $\alpha = 4$ and $\alpha = 4.1$ by all the methods discussed above for $t = 0\colon 100$. The best rational L_∞ approximation is applicable since A has real, negative eigenvalues. Balancing has no effect on this matrix. Figure 6.5 shows that Algorithm 5.2 is the only method to produce a smooth curve that tracks the growth and decay of the exponential across the whole interval, and indeed each computed norm for Algorithm 5.2 has relative error less than 5×10^{-14}. The accuracy of the other methods is affected by the nonnormality of A, which is responsible for the hump. The rational approximation method solves linear systems with condition numbers up to order 10^{12}, which causes its ultimate loss of accuracy. The Krylov methods are so sensitive to rounding errors that changing α from 4 to 4.1 produces quite different results, and qualitatively so for `phipm`. The problem is very ill-conditioned: $\kappa_{\exp}(tA, b) \geq u_d^{-1}$ for $t \gtrsim 53$.

Experiment 7. Since our approach is in the spirit of the sixth of Moler and Van Loan's "19 dubious ways" [19, sec. 4], it is appropriate to make a comparison with a direct implementation of their "Method 6: single step ODE methods". For the Laplacian matrix and vector b from Experiment 4, we compute $e^{\alpha A}b$, for $\alpha = 1$ and $\alpha = 4$, using Algorithm 3.2 with tol $= u_s$ and also the `ode45` and `ode15s` functions from MATLAB, with absolute and relative error tolerances (used in a mixed absolute/relative error criterion) both set to u_s. The `ode45` function uses an explicit Runge–Kutta (4,5) formula while `ode15s` uses implicit multistep methods. We called both solvers with time interval specified as `[0 alpha/2 alpha]` instead of `[0 alpha]`;

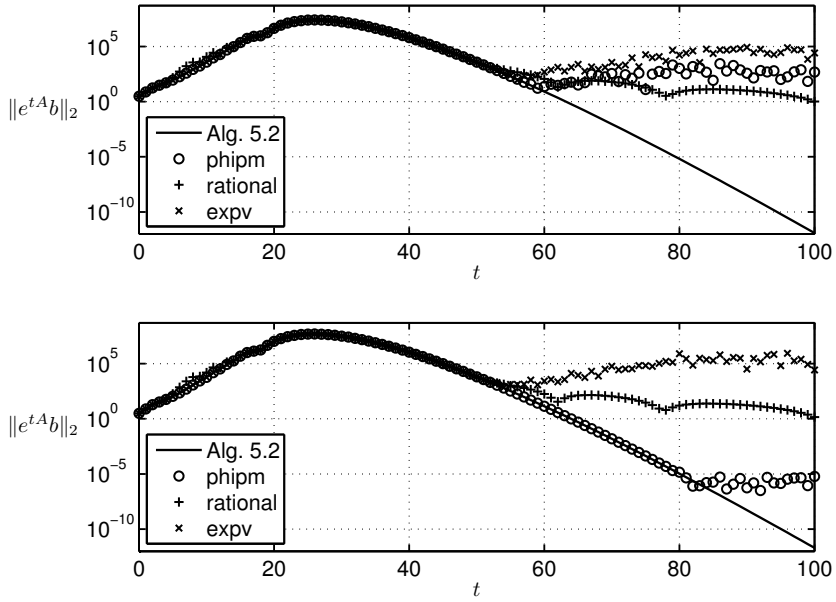

FIG. 6.5. *Experiment* 6*: t versus $\|e^{tA}b\|_2$, with $\alpha = 4$ (top) and $\alpha = 4.1$ (bottom).*

TABLE 6.3
Experiment 7*: "LU" denotes the number of LU factorizations for* `ode15s`*. The subscript on* `poisson` *denotes the value of α.*

	Algorithm 3.2			`ode45`			`ode15s`		
	time	mv	error	time	mv	error	time	mv/LU/sol	error
`orani678`	0.16	702	4.3e-8	3.49	14269	2.7e-8	148	7780/606/7778	1.6e-6
`bcspwr10`	0.022	215	7.2e-7	0.61	3325	9.5e-7	3.51	1890/76/1888	4.8e-5
$\texttt{poisson}_1$	4.30	29255	2.2e-6	9.80	38401	3.6e-7	2.81	402/33/400	8.3e-6
$\texttt{poisson}_4$	16.8	116849	9.0e-6	39.0	154075	2.2e0	3.41	494/40/492	1.4e-1

this stops them returning output at each internal mesh point and so substantially speeds up the integration. The results in Table 6.3 show the superior efficiency of Algorithm 3.2 over `ode45`. The `ode15s` function performs variably, being extremely slow for the `orani678` problem, but faster than Algorithm 3.2 for the `poisson` problem, in this case being helped by the fact that A is highly sparse and structured, so that the linear system solves it requires are relatively inexpensive. However, both ODE solvers fail to produce the desired accuracy for the `poisson` problem with $\alpha = 4$.

Experiment 8. This experiment is taken from Sheehan, Saad, and Sidje [23, sec. 4.2]. The nonsymmetric matrix $A \in \mathbb{R}^{55\times 55}$ arises in modelling a Boeing 767 aircraft. The norm $\|A\|_1 \approx 10^7$ while $\rho(A) \approx 10^3$, this disparity indicating that A is very non-normal. Furthermore, $\|A^k\|_1^{1/k} \ll \|A\|_1$ for $k \geq 2$, and Algorithm 3.2 exploits this property.

We compute $e^{\tau A}b$ for the vector b of ones and three values of τ, with error tolerance u_s. The Krylov subspace dimension for `expv` is set to 15. The Laguerre and Chebyshev codes from [23] need to traverse the interval $[0, 1]$ in several steps: 160 steps for $\tau = 1$, 16 for $\tau = 0.1$, and 1 for $\tau = 0.01$; these values are determined by some analysis of the problem given in [23], which also determines an interval containing the real parts

TABLE 6.4
Experiment 8: t_{ratio} *denotes time for method divided by time for Algorithm* 3.2.

τ	Algorithm 3.2			`expv`			Laguerre			Chebyshev		
	t_{ratio}	mv	error	t_{ratio}	mv	error	t_{ratio}	mv	error	t_{ratio}	mv	error
1.00	1	2314	1.8e-9	3.5	1776	2.4e-9	321.8	7861	6.5e-10	19.5	3471	7.5e-10
0.10	1	475	9.7e-10	1.9	256	1.8e-9	90.7	783	3.4e-11	7.2	368	3.2e-11
0.01	1	231	2.0e-10	0.4	64	2.3e-9	6.4	53	2.6e-11	0.4	26	9.1e-12

TABLE 6.5
Experiment 9: t_{ratio} *denotes time for method divided by time for Algorithm* 3.2.

Algorithm 3.2			`expv`			Laguerre			Chebyshev		
t_{ratio}	mv	error	t_{ratio}	mv	error	t_{ratio}	mv	error	t_{ratio}	mv	error
1	20	1.9e-9	6.2	32	3.9e-16	4.2	39	5.7e-12	1.5	22	2.4e-13

of the eigenvalues that must be given to the Chebyshev code. The results in Table 6.4 show that Algorithm 3.2 is the most efficient method for $\tau = 1$ and $\tau = 0.1$ but is slower than the Krylov and Chebyshev methods for $\tau = 0.01$. We note that of the 231 matrix–vector products that Algorithm 3.2 uses for $\tau = 0.01$, 176 are for norm estimation.

Experiment 9. This experiment is taken from [23, sec. 4.3]. The nonsymmetric matrix A is from a standard 5-point discretization of a 3D diffusion-convection operator and has dimension $250,000$ and norm $\|A\|_1 = 8$. The details are exactly as in [23, sec. 4.3] except that the tolerance is u_s. Table 6.5 shows that Algorithm 3.2 requires the fewest matrix–vector products and is the fastest. The number of matrix–vector products is so small for Algorithm 3.2 because the test (3.13) is satisfied and no matrix norm estimation is required. It is not clear why `expv` and the Laguerre and Chebyshev methods deliver much greater accuracy than requested.

Experiment 10. Our final experiment illustrates the application of Theorem 2.1 in order to compute the exponential integrator approximation $\widehat{u}(t)$ in (2.4) via (2.11). We take for $A \in \mathbb{R}^{400\times 400}$ the symmetric matrix `-gallery('poisson',20)` and random vectors u_k, $k = 0\colon p$, with elements from the normal (0,1) distribution, where $p = 5\colon 5\colon 20$. The matrix $W \in \mathbb{R}^{400\times p}$ has columns $W(:, p-k+1) = u_k$, $k = 1\colon p$. For each p, we compute $\widehat{u}(t)$ at each $t = 1\colon 0.5\colon 10$ by Algorithm 5.2 (with $t_0 = 1$, $t_q = 10$, and $q = 18$) and by `phipm`, which has the ability to compute $\widehat{u}(t)$ via the expression (2.4). For computing errors, we regard as the "exact" solution the vector obtained by using `expm` to compute the exponential on the right-hand side of (2.11). We set the tolerance to u_d for both algorithms. Figure 6.6 plots the relative errors. The total number of matrix–vector products is 1801 for Algorithm 5.2 and 3749 for `phipm`.

The relative error produced by Algorithm 5.2 is of order u_d for all t and p, whereas the relative error for `phipm` deteriorates with increasing t, the more rapidly so for the larger p, suggesting instability in the recurrences that the code uses. The practical implication is that p, which is the degree of the polynomial approximation in an exponential integrator, may be need to be limited for use with `phipm` but is unrestricted for our algorithm. The run time for this experiment is 0.10 seconds for Algorithm 5.2 and 0.60 seconds for `phipm`.

The importance of the normalization by η in (2.11) is seen if we multiply W by 10^6 and repeat the experiment with the default η and then $\eta = 1$. The maximum relative errors for Algorithm 5.2 in the two cases are 2.3×10^{-15} and 3.3×10^{-12}.

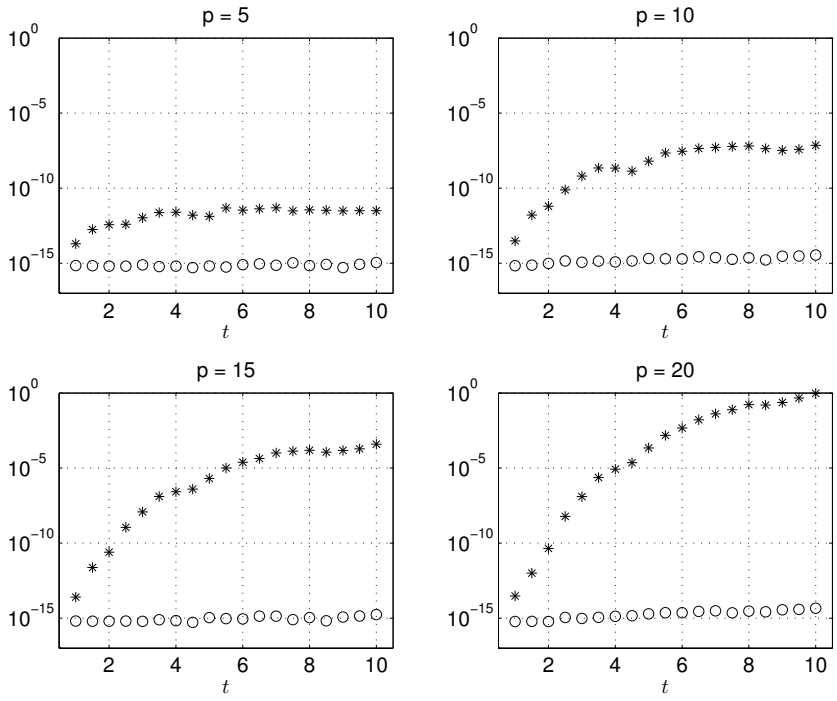

FIG. 6.6. *Experiment* 10*: relative errors of computed* $\widehat{u}(t)$ *in* (2.4) *from Algorithm* 5.2 ($\circ$) *and* `phipm` ($*$) *over the interval* $[1, 10]$ *for* $p = 5\colon 5\colon 20$.

7. Discussion. Our new algorithm, Algorithm 5.2 (and its special case Algorithm 3.2) has a number of attractive features. Suppose, first, that $B \in \mathbb{R}^n$ is a vector. The algorithm spends most of its time computing matrix–vector products, with other computations occupying around 12% of the time. Thus it fully benefits from problems where matrix–vector products are inexpensive, or their evaluation is optimized. The algorithm is essentially direct rather than iterative, because the number of matrix–vector products is known after the initial norm estimation phase and depends only on the values $d_k = \|A^k\|_1^{1/k}$ for a few values of k (sometimes just $k = 1$). No convergence test is needed, except the test for early termination built into the evaluation of the truncated Taylor series. Our experiments demonstrate excellent numerical stability in floating point arithmetic, and this is supported (but not entirely explained) by the analysis in Section 4. The numerical reliability of the algorithm is emphasized by the fact that in Experiment 1 it has a better relative error performance profile than evaluation of $e^A b$ using the best current method for e^A. A particular strength of the algorithm is in the evaluation of e^{tA} at multiple points on the interval of interest, because the scaling used by the algorithm naturally produces intermediate values, and the design of the algorithm ensures that when extremely dense output is required overscaling is avoided.

These benefits contrast with Krylov methods, of which we tested two particular examples. These methods are genuinely iterative, so their cost is difficult to predict and the choice of convergence test will influence the performance and reliability. They also typically require the selection or estimation of the size of the Krylov subspace.

Moreover, they require more storage and the cost of the methods is not necessarily dominated by the matrix–vector products with A; depending on the method and the problem it can be dominated by the computations in the Arnoldi or Lanczos recurrence, and the cost of evaluating exponentials of smaller Hessenberg matrices is potentially significant. Our algorithm computes a (usually) high degree polynomial of A times b, and in exact arithmetic the result will less accurate than the result from a Krylov subspace method using the same number of matrix–vector products, due to the optimality of the Krylov approach. However, Krylov methods do not necessarily choose the same polynomial degree as our algorithm.

The Chebyshev expansion method from [23] requires an interval or region containing the spectrum of A; the Laguerre polynomial method therein does not require this information but is generally less efficient. Both these methods require, in general, a time-stepping strategy for faster convergence of the series expansions and better numerical stability, the choice of which is not straightforward [23].

For the case where B is a matrix with $n_0 > 1$ columns, Algorithm 5.2 is particularly advantageous because the logic of the algorithm is unchanged and the computational effort is now focused on products of $n \times n$ and $n \times n_0$ matrices. An algorithm for $e^A B$ based on block Krylov subspaces is not currently available. However, recent work of Hochbruck and Niehoff [16], motivated by exponential integrators, treats the case where the columns of B are the values of some smooth function on a given grid of points.

The weakness of Algorithm 5.2 is its tendency for the cost to increase with increasing $\|A\|$ (though it is the d_k that actually determine the cost). This weakness is to some extent mitigated by preprocessing, but for symmetric A balancing has no effect and $d_k \approx \|A\|$ (there is equality for the 2-norm, but the algorithm uses the 1-norm). For symmetric semidefinite matrices, error bounds for Krylov and Chebyshev methods suggest that they require on the order of $\|A\|^{1/2}$ matrix–vector products, giving them a theoretical advantage. However, the practical matters of convergence tests, premature termination of series evaluation, and computational overheads can negate this advantage (see Experiment 4).

Algorithm 5.2 requires matrix–vector products with both A and A^* for the norm estimation. However, if A is non-Hermitian and known only implicitly and A^*x cannot be formed, we can revert to the use of $\|A\|_1$ in place of the $\alpha_p(A)$.

In summary, Algorithm 5.2 emerges as the best method for the $e^A B$ problem in our experiments. It has several features that make it attractive for black box use in a wide range of applications: its applicability to any A, its predictable cost after the initial norm estimation phase, its excellent numerical stability, and its ease of implementation.

8. Acknowledgements. We thank the referees for suggestions that led to an improved paper and the authors of [23] for sharing their MATLAB codes with us.

REFERENCES

[1] A. H. Al-Mohy and N. J. Higham. Computing the Fréchet derivative of the matrix exponential, with an application to condition number estimation. *SIAM J. Matrix Anal. Appl.*, 30(4): 1639–1657, 2009.

[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(3):970–989, 2009.

[3] A. C. Antoulas. *Approximation of Large-Scale Dynamical Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2005. ISBN 0-89871-529-6. xxv+479 pp.

[4] T.-Y. Chen and J. W. Demmel. Balancing sparse matrices for computing eigenvalues. *Linear Algebra Appl.*, 309:261–287, 2000.
[5] T. A. Davis. University of Florida sparse matrix collection. `http://www.cise.ufl.edu/research/sparse/matrices/`.
[6] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. Manuscript available at `http://www.cise.ufl.edu/research/sparse/matrices/`, 2010.
[7] J. J. Dongarra, J. J. Du Croz, I. S. Duff, and S. J. Hammarling. A set of Level 3 basic linear algebra subprograms. *ACM Trans. Math. Software*, 16:1–17, 1990.
[8] I. S. Duff, M. A. Heroux, and R. Pozo. An overview of the Sparse Basic Linear Algebra Subprograms: The new standard from the BLAS Technical Forum. *ACM Trans. Math. Software*, 28(2):239–267, 2002.
[9] N. J. Higham. The Matrix Function Toolbox. `http://www.ma.man.ac.uk/~higham/mftoolbox`.
[10] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002. ISBN 0-89871-521-0. xxx+680 pp.
[11] N. J. Higham. The scaling and squaring method for the matrix exponential revisited. *SIAM J. Matrix Anal. Appl.*, 26(4):1179–1193, 2005.
[12] N. J. Higham. *Functions of Matrices: Theory and Computation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008. ISBN 978-0-898716-46-7. xx+425 pp.
[13] N. J. Higham. The scaling and squaring method for the matrix exponential revisited. *SIAM Rev.*, 51(4):747–764, 2009.
[14] N. J. Higham and A. H. Al-Mohy. Computing matrix functions. *Acta Numerica*, 19:159–208, 2010.
[15] 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(4):1185–1201, 2000.
[16] M. Hochbruck and J. Niehoff. Approximation of matrix operators applied to multiple vectors. *Math. Comput. Simulation*, 79:1270–1283, 2008.
[17] M. Hochbruck and A. Ostermann. Exponential integrators. *Acta Numerica*, 19:209–286, 2010.
[18] B. V. Minchev and W. M. Wright. A review of exponential integrators for first order semi-linear problems. Preprint 2/2005, Norwegian University of Science and Technology, Trondheim, Norway, 2005. 44 pp.
[19] 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(1):3–49, 2003.
[20] J. Niesen. `http://www.amsta.leeds.ac.uk/~jitse/software.html`, retrieved on February 17, 2010.
[21] J. Niesen and W. M. Wright. A Krylov subspace algorithm for evaluating the φ-functions appearing in exponential integrators. Technical Report arXiv:0907.4631, 2009. 20 pp.
[22] Y. Saad. Analysis of some Krylov subspace approximations to the matrix exponential operator. *SIAM J. Numer. Anal.*, 29(1):209–228, Feb. 1992.
[23] B. N. Sheehan, Y. Saad, and R. B. Sidje. Computing $\exp(-\tau A)b$ with Laguerre polynomials. *Electron. Trans. Numer. Anal.*, 37(4):147–165, 2010.
[24] R. B. Sidje. Expokit. `http://www.maths.uq.edu.au/expokit`, retrieved October 8, 2009.
[25] R. B. Sidje. Expokit: A software package for computing matrix exponentials. *ACM Trans. Math. Software*, 24(1):130–156, 1998.
[26] L. N. Trefethen, J. A. C. Weideman, and T. Schmelzer. Talbot quadratures and rational approximations. *BIT*, 46(3):653–670, 2006.
[27] C. F. Van Loan. The sensitivity of the matrix exponential. *SIAM J. Numer. Anal.*, 14(6): 971–981, 1977.
[28] D. S. Watkins. A case where balancing is harmful. *Electron. Trans. Numer. Anal.*, 23:1–4, 2006.
[29] D. E. Whitney. More about similarities between Runge–Kutta and matrix exponential methods for evauating transient response. *Proc. IEEE*, 57:2053–2054, 1969.