Contents lists available at ScienceDirect
Journal of Computational and Applied Mathematics
journal homepage: www.elsevier.com/locate/cam

Fast Taylor polynomial evaluation for the computation of the matrix cosine

Jorge Sastre [a,1], Javier Ibáñez [b,1], Pedro Alonso-Jordá [c,1,*], Jesús Peinado [c,1], Emilio Defez [d,1]

[a] *Instituto de Telecomunicaciones y Aplicaciones Multimedia, Spain*
[b] *Instituto de Instrumentación para Imagen Molecular, Spain*
[c] *Department of Information Systems and Computation, Spain*
[d] *Instituto de Matemática Multidisciplinar, Spain*

ARTICLE INFO

Article history:
Received 14 May 2018
Received in revised form 24 December 2018

Keywords:
Matrix
Matrix trigonometric functions
Matrix cosine
Taylor
Fast matrix polynomial evaluation
GPU computing

ABSTRACT

In this work we introduce a new method to compute the matrix cosine. It is based on recent new matrix polynomial evaluation methods for the Taylor approximation and a mixed forward and backward error analysis. The matrix polynomial evaluation methods allow to evaluate the Taylor polynomial approximation of the matrix cosine function more efficiently than using Paterson–Stockmeyer method. A sequential Matlab implementation of the new algorithm is provided, giving better efficiency and accuracy than state-of-the-art algorithms. Moreover, we provide an implementation in Matlab that can use NVIDIA GPUs easily and efficiently.

1. Introduction

The exact solution of many engineering processes that are described by second order differential equations is given in terms of the trigonometric matrix functions sine and/or cosine. This is the case, for instance, of the wave problem. The most popular state-of-the-art algorithms used to calculate these matrix functions are based on polynomial and rational approximations with scaling and recovering techniques [1–5]. Paterson–Stockmeyer's method [6] is the most used to compute the matrix polynomials that appear in these approximations in order to reduce computational costs. Recently, a new family of methods for the evaluation of general matrix polynomials has been proposed [7], allowing to reduce the number of matrix products needed to evaluate a polynomial with respect to the Paterson–Stockmeyer's method. In this work, we present competitive algorithms for the computation of the matrix cosine based on the evaluation of Taylor approximations using those methods. Sequential and NVIDIA GPU based Matlab implementations of the new algorithms are given. The basic computational kernel of algorithms based on Taylor approximations is matrix multiplication. This kernel can be executed very rapidly on accelerator devices like GPUs (Graphic Processing Units). In this paper we have exploited this fact, together with our previous experience on this subject [8], to build a Matlab script plus a `mex` file capable of executing the new algorithm very efficiently.

The next section presents a scaling and squaring Taylor algorithm for computing the matrix cosine based on the methods described in [7]. Section 3 describes a forward and backward error analysis for computing the Taylor approximation using our algorithm. Finally, in Section 4, we show some numerical results of the Matlab sequential and GPU implementations from both the performance and accuracy points of view. Finally, Section 5 gives some conclusions.

[*] Corresponding author.
E-mail address: palonso@upv.es (P. Alonso-Jordá).
[1] All authors belong to Universitat Politècnica de València.

https://doi.org/10.1016/j.cam.2018.12.041
0377-0427/© 2018 Elsevier B.V. All rights reserved.

Algorithm 1 Given a matrix $A \in \mathbb{C}^{n\times n}$ and a maximum order $m_M \in \mathbb{N}$, this algorithm computes $C = \cos(A)$ by a Taylor approximation of order $2m_k \leq 2m_M$, where m_k are optimal degrees of the Taylor polynomial, i.e. the maximum degrees of the Taylor polynomial which can be evaluated for a certain number of matrix products.

```
1: $B = A^2$
2: SCALING PHASE: Choose $m_k \leqslant m_M$ and an integer scaling parameter $s$ for the Taylor approximation with scaling.
3: $B = B/4^s$
4: Compute $C = P_{m_k}(B)$
5: for $i = 1 : s$ do
6:     $C = 2C^2 - I$
7: end for
```

2. Taylor algorithm for computing the matrix cosine

The matrix cosine can be defined for all $A \in \mathbb{C}^{n\times n}$ by the series

$$\cos(A) = \sum_{i=0}^{\infty} \frac{(-1)^i A^{2i}}{(2i)!}. \tag{1}$$

Let

$$T_{2m}(A) = \sum_{i=0}^{m} p_i A^{2i} = \sum_{i=0}^{m} p_i B^i \equiv P_m(B), \tag{2}$$

be the Taylor approximation of order $2m$ of $\cos(A)$, where $p_i = \frac{(-1)^i}{(2i)!}$ and $B = A^2$. Algorithm 1 shows a general algorithm for computing the matrix cosine based on Taylor series. Since (2) is accurate near the origin, for computing $\cos(A)$ from the Taylor approximation it is often necessary to scale matrix B and recover $\cos(A)$ from the Taylor approximation of the cosine of the scaled matrix. Scaling matrix B by an integer $s > 0$ consists of computing $B := 4^{-s}B$ (Step 3). Once the cosine of the scaled matrix has been computed, $\cos(A)$ can be computed (Steps 5–7) by using repeatedly the double angle formula $\cos(2X) = 2\cos^2(X) - I$ [9, p. 288].

Matrix A can be preprocessed to reduce its norm as described in [10, Alg. 1.2] and this procedure will not be discussed in this paper. Step 4 was traditionally performed by using the Paterson–Stockmeyer's method [6], however, in this paper we use a more efficient method for evaluating $C = P_{m_k}(B)$ based on [7]. This method depends on the value of m_k selected in Step 2 of Algorithm 1 (from now on we will use m instead of m_k for simplicity). Below we analyze each case.

For $m = 1, 2$ and 4, similarly to (10) from [11], the Taylor polynomials $P_m(B)$ can be computed by using the following expressions:

$$\begin{aligned} P_1(B) &= -B/2 + I, \\ P_2(B) &= (B^2/12 - B)/2 + I, \\ P_4(B) &= (((B^2/56 - B)/30 + I)B^2/12 - B)/2 + I. \end{aligned} \tag{3}$$

Following [7, Ex. 3.1], $P_8(B)$ can be evaluated by using the following formulae:

$$\begin{aligned} y_{02}(B) &= B^2(c_1 B^2 + c_2 B), \\ P_8(B) &= (y_{02}(B) + c_3 B^2 + c_4 B)(y_{02}(B) + c_5 B^2) \\ &\quad + c_6 y_{02}(B) + B^2/24 - B/2 + I, \end{aligned} \tag{4}$$

with a cost of 3 matrix products. With that cost the maximum approximation order available with Paterson–Stockmeyer is $m = 6$. The coefficients c_i for IEEE double precision arithmetic are given in Table 1, see [7, Table 4].

In the following we show that the most efficient methods proposed in [7] to evaluate the Taylor polynomial for $m > 8$ are not accurate enough for the matrix cosine approximation. Therefore, other possibilities are proposed for increasing accuracy, in exchange for a higher cost. Despite this higher cost, the proposed matrix polynomial evaluation methods are more efficient than Paterson–Stockmeyer method.

Following [12, Sec. 3.2], and similarly to [7, Ex. 5.1], with a cost of 4 matrix products it is possible to obtain a Taylor based approximation $P_{16}(B)$ of the matrix cosine of order $m = 15$, with several real solutions for the coefficients involved. However, for all the real solutions rounded in IEEE double precision arithmetic, the stability check proposed in [7, Ex. 3.1] gives errors of order $10^{-14} > u$ or greater, where u is the unit roundoff in IEEE double precision arithmetic, i.e. $u = 2^{-53} \approx 1.11^{-16}$. We have checked that these evaluation formulae provided reduced accuracy results in numerical tests.

Table 1
Coefficients for computing the matrix exponential Taylor approximation.

	$m=8$	$m=12$	$m=15$
c_1	$2.186201576339059 \times 10^{-7}$	$1.269542268337734 \times 10^{-12}$	$6.140022498994532 \times 10^{-17}$
c_2	$-2.623441891606870 \times 10^{-5}$	$-3.503936660612145 \times 10^{-10}$	$-2.670909787062621 \times 10^{-14}$
c_3	$6.257028774393310 \times 10^{-3}$	$1.135275478038335 \times 10^{-7}$	$1.438284920333222 \times 10^{-11}$
c_4	$-4.923675742167775 \times 10^{-1}$	$-2.027712316612395 \times 10^{-5}$	$-1.050202496489896 \times 10^{-8}$
c_5	$1.441694411274536 \times 10^{-4}$	$1.647243380001247 \times 10^{-3}$	$4.215975785860907 \times 10^{-6}$
c_6	$5.023570505224926 \times 10^{1}$	$-6.469859264308602 \times 10^{-1}$	$-1.238347173261210 \times 10^{-3}$
c_7		$-4.008589447357360 \times 10^{-5}$	$-3.234597615453410 \times 10^{-9}$
c_8		$9.187724869020796 \times 10^{-3}$	$9.292820886910254 \times 10^{-7}$
c_9		$-1.432942184841715 \times 10^{-2}$	$2.466381973203188 \times 10^{-1}$
c_{10}		$4.555439797286385 \times 10^{-3}$	$-9.369018510939971 \times 10^{-10}$

Then, for a cost of 4 matrix products we will use (34) and (35) from [7] to evaluate $P_{12}(B)$ by using the following formulae:

$$\begin{aligned} y_{02}(B) &= B^3(c_1B^3 + c_2B^2 + c_3B), \\ P_{12}(B) &= (y_{02}(B) + c_4B^3 + c_5B^2 + c_6B)(y_{02}(B) + c_7B^3 + c_8B^2) \\ &\quad c_9y_{02} + c_{10}B^3 + B^2/24 - B/2 + I, \end{aligned} \tag{5}$$

where the coefficients c_i are given in Table 1. From the different real solutions for the coefficients, we selected those ones giving the lowest maximum error in the stability test, similarly to [7, Ex 3.1], giving errors lower than $1.31 \times 10^{-16} = O(u)$. Eq. (5) provides a lower order than 15, but behaves in a stable manner being, in turn, more efficient than Paterson–Stockmeyer method, since with a cost of 4 matrix products the maximum approximation order available with Paterson–Stockmeyer is $m = 9$.

The highest order m used in [5] for $P_m(B)$ is $m = 16$, available with 6 matrix products using Paterson–Stockmeyer method. Using (34) and (35) from [7] it is possible to evaluate $P_{16}(B)$ with 5 matrix products and several possibilities of real coefficients. The stability check proposed in [7, Ex. 3.1] gives a maximum error of $1.03 \times 10^{-15} > u$, and we checked that numerical results were not accurate enough. The stability can be improved using expression (52) from [7], with $s = 3$ and $p = 3$, giving the following formulae for $m = 15$:

$$\begin{aligned} y_{02}(B) &= B^3(c_1B^3 + c_2B^2 + c_3B), \\ P_{15}(B) &= -((y_{02}(B) + c_4B^3 + c_5B^2 + c_6B)(y_{02}(B) + c_7B^3 + c_8B^2) + c_9y_{02} \\ &\quad + c_{10}B^3 + B^2/368800 - B/40320 + I/720)B^3 + B^2/24 - B/2 + I. \end{aligned} \tag{6}$$

The stability check for the coefficients c_i given in Table 1, selected among all the possible real solutions of the coefficients, gives a maximum error of order u, in an exchange for a lower order $m = 15 < 16$. The minus sign at the beginning of expression $P_{15}(B)$ allows to obtain real solutions for all the coefficients involved, as suggested in [7, p. 237]. With a cost of 5 matrix products the maximum approximation order available with Paterson–Stockmeyer is $m = 12$.

All coefficients c_i that appear in expressions (4)–(6) were computed with the MATLAB R2018a Symbolic Math Toolbox, using 200 decimal digit arithmetic. Table 1 shows these values in IEEE double precision arithmetic.

3. Error analysis

In [2] an absolute forward error analysis of the Taylor approximation for the matrix cosine was developed. In [5] a combination of a relative forward and backward error analysis was developed for the same function. In this section we present a unified study of the error analysis for the computation of that matrix function, selecting the analysis among the three types of analysis giving the most efficient option for each degree m of the cosine Taylor approximation. The following theorem is used in this study:

Theorem 1 *([2]). Let $h_l(x) = \sum_{i\geq l} p_i x^i$ be a power series with radius of convergence w, $\tilde{h}_l(x) = \sum_{i\geq l} |p_i| x^i$, $B \in \mathbb{C}^{n\times n}$ with $\rho(B) < w$, $l \in \mathbb{N}$ and $t \in \mathbb{N}$ with $1 \leqslant t \leqslant l$. If t_0 is the multiple of t such that $l \leqslant t_0 \leqslant l + t - 1$ and*

$$\beta_t = \max\{d_j^{1/j} : j = t, l, l+1, \dots, t_0 - 1, t_0 + 1, t_0 + 2, \dots, l + t - 1\},$$

where d_j is an upper bound for $\|B^j\|$, $d_j \geqslant \|B^j\|$, then

$$\|h_l(B)\| \leqslant \tilde{h}_l(\beta_t).$$

If we apply Theorem 1 for $t = l$, then $\|h_l(B)\| \leqslant \tilde{h}_l(\beta_l)$, where

$$\beta_l = \max\{d_j^{1/j} : j = l,\ l+1,\ \dots,\ 2l-1\}. \tag{7}$$

In [13, Sec. 4.1] the authors approximated $\beta_{min} = \min\{\beta_t^{(l)}, 1 \le t \le m+1\}$ by

$$\beta_{min} \approx \max\{d_{l+1}^{1/(l+1)}, d_{l+2}^{1/(l+2)}\}, \tag{8}$$

corresponding to the two first terms of (7).

3.1. Absolute and relative forward errors

Let $A \in \mathbb{C}^{n\times n}$ and $B = A^2$. Using (1) the absolute forward error in exact arithmetic for the Taylor approximation (2) of $\cos(A)$, denoted by $E_f^{(1)}$, is,

$$E_f^{(1)} = \|\cos(A) - P_m(B)\| = \left\| \sum_{i\ge m+1} f_{m,i}^{(1)} B^i \right\|, \tag{9}$$

where $f_{m,i}^{(1)} = (-1)^i/(2i)!$. This error analysis is used in [2, Sec. 4].

If $\|B\| = \|A^2\| < \mathrm{acosh}^2(2) \approx 1.7343$, then $\cos^{-1}(A)$ exists [5, Proposition 1] and it follows that the relative forward error to compute $\cos(A)$ in exact arithmetic, denoted by $E_f^{(2)}$, is [5, Sec. 2.1]

$$\begin{aligned} E_f^{(2)} &= \left\|\cos^{-1}(A)\,(\cos(A) - P_m(B))\right\| \\ &= \left\|I - P_m(B)\cos^{-1}(A)\right\| = \left\| \sum_{i\ge m+1} f_{m,i}^{(2)} B^i \right\|. \end{aligned} \tag{10}$$

If we define $g_{m+1}^{(k)}(x) = \sum_{i\geqslant m+1} f_{m,i}^{(k)} x^i$ and $\tilde{g}_{m+1}^{(k)}(x) = \sum_{i\ge m+1} \left|f_{m,i}^{(k)}\right| x^i$, $k = 1{,}2$, and we apply Theorem 1, then

$$E_f^{(k)} = \left\|g_{m+1}^{(k)}(B)\right\| \le \tilde{g}_{m+1}^{(k)}(\beta_t), \tag{11}$$

for every t, $1 \le t \le m+1$.

3.2. Relative backward error

The backward error ΔA of approximating $\cos(A)$ by Taylor approximation $T_{2m}(A)$ verifies

$$T_{2m}(A) = \cos(A + \Delta A).$$

From [5, Sec. 2.2] the backward error ΔA can be expressed by

$$\Delta A = \sum_{i\ge m} b_{m,i} A^{2i+1},$$

where coefficients $b_{m,i}$ can be computed symbolically, see (8)–(11) of [5, Sec. 2.2] for details. Then, the relative backward error (E_b) in exact arithmetic of approximating $\cos(A)$ by $T_{2m}(A)$ can be computed as

$$E_b = \frac{\|\Delta A\|}{\|A\|} = \frac{\left\|\sum_{i\ge m} b_{m,i} A^{2i+1}\right\|}{\|A\|} \le \left\|\sum_{i\ge m} b_{m,i} A^{2i}\right\| = \left\|\sum_{i\ge m} b_{m,i} B^{i}\right\|.$$

If we define $h_m(x) = \sum_{i\geqslant m} b_{m,i} x^i$ and $\tilde{h}_m(x) = \sum_{i\ge m} \left|b_{m,i}\right| x^i$, and we apply Theorem 1, then

$$E_b = \|h_m(B)\| \le \tilde{h}_m(\beta_t), \tag{12}$$

for every t, $1 \le t \le m$. In [5] was used an error analysis that is a combination of the relative forward and backward error analysis in exact arithmetic.

3.3. Computation of taylor order m and the scaling parameter s

Let $\Theta_{f_k}(m)$, $k = 1{,}2$, be

$$\Theta_{f_k}(m) = \max\left\{\theta \ge 0 : \ \tilde{g}_{m+1}^{(k)}(\theta) = \sum_{i\ge m+1} \left|f_{m,i}^{(k)}\right| \theta^i \le u \right\}, \tag{13}$$

and let $\Theta_b(m)$ be

$$\Theta_b(m) = \max\left\{\theta \ge 0 : \ \tilde{h}_m(\theta) = \sum_{i\ge m} \left|b_{m,i}^{(k)}\right| \theta^i \le u \right\}, \tag{14}$$

Table 2
Values of $\Theta_{f_1}(m)$, $\Theta_{f_2}(m)$, $\Theta_b(m)$, and $\Theta(m)$ for $m = 8, 12, 15$.

	$m = 8$	$m = 12$	$m = 15$
Θ_{f_1}	0.96	6.59	16.45
Θ_{f_2}	0.91	–	–
Θ_b	0.94	6.75	9.91
Θ	0.9625107544271462	6.752349007371135	16.45123831556254

where $u = 2^{-53}$ is the unit roundoff in double precision floating-point arithmetic. We have used MATLAB Symbolic Math Toolbox to compute $\Theta_{f_k}(m)$, $k = 1,2$, and $\Theta_b(m)$ for $m = 8, 12, 15$ in 250-decimal digit arithmetic, considering enough terms to obtain all the Θ values for each m with enough significant digits, and obtaining the coefficients symbolically. Note that $\Theta_b(15)$ needs more than 1500 terms to obtain three significant digits, similarly to what happens with $\Theta_b(16)$ and $\Theta_b(20)$ in [5, Sec. 2.2]. Then, a numerical zero-finder is invoked to determine the highest values $\Theta_{f_k}(m)$, $k = 1,2$, and $\Theta_b(m)$, such that

$$\tilde{g}^{(k)}_{m+1}(\Theta_{f_k}(m)) = \sum_{i\ge m+1} \left|f^{(k)}_{m,i}\right| \Theta^i_{f_k}(m) \le u,$$

and

$$\tilde{h}_m(\Theta_b(m)) = \sum_{i\ge m} \left|e^{(k)}_{m,i}\right| \Theta^i_b(m) \le u,$$

holds. The values of $\Theta_{f_k}(m)$, $k = 1,2$, and $\Theta_b(m)$ for $m = 8, 12, 15$ are depicted in Table 2. For $m = \{1, 2, 4\}$ [5, Sec. 2.2] shows that $\Theta_{f_2}(m) > \Theta_b(m)$, and comparing tables [2, Table 2] and [5, Table 1] one gets that $\Theta_{f_1}(m) \gtrapprox \Theta_{f_2}(m)$. This is a normal behavior since, for those values of m, it follows that $\cos(\sqrt{\Theta_{f_1}(m)}) \approx 1$ and, in that case, the forward absolute error bound (9) and the forward relative error bound (10) are approximately equal. Analogously to [5, Sec. 2], and to minimize the computational cost, we are selecting $\Theta(m) = \max\left\{\Theta_{f_1}(m), \Theta_{f_2}(m), \Theta_b(m)\right\}$ for each $m_k = \{1, 2, 4, 8, 12, 15\}$, $k = 1, 2, \ldots, 6$, i.e. $\Theta_{f_1}(m)$ for $m = \{1, 2, 4, 8, 15\}$ and $\Theta_b(m)$ for $m = 12$. Then, considering (11) and taking into account the values of Θ_m from Table 2, it follows that, for $m = \{1, 2, 4, 8, 15\}$, if $\beta_t \le \Theta_{f_1}(m)$ the absolute forward error is lower than or equal to the unit roundoff:

$$E^{(1)}_f \le \tilde{g}^{(1)}_m(\beta_t) \le \tilde{g}^{(1)}_m(\Theta_{f_1}(m)) \le u,$$

and using (8) one gets

$$\beta^{(m)}_{m+1} \approx \max\{b^{1/(m+1)}_{m+1}, b^{1/(m+2)}_{m+2}\}, \tag{15}$$

where the superscript (m) stands for the Taylor approximation order used (see (15) from [5]). For $m = 12$ and considering (12), if $\beta^{(m)}_{min} \le \Theta_b(m)$, then relative backward error is lower than or equal to the unit roundoff:

$$E_b \le \tilde{h}_m(\beta_t) \le \tilde{h}_m(\Theta_b(m)) \le u,$$

and using (8) one gets

$$\beta^{(m)}_{min} \approx \max\{b^{1/(m)}_m, b^{1/(m+1)}_{m+1}\}, \tag{16}$$

where the superscript (m) also stands for the Taylor approximation order used (see (16) from [5]).

We provide here a scaling algorithm without norm estimations of matrix powers and another one with norm estimations of matrix powers similar to those algorithms developed in [5, Sec. 2.4]: If there exists a value $m \le 15$ such that $\beta^{(m_k)}_{min} \le \Theta(m)$, then one of the above conditions is verified and, in this case, we choose the lower order m_k verifying it with a scaling parameter $s = 0$. Otherwise, we choose the Taylor approximation of order 12 or 15 providing the lower cost, with

$$s = \max\left\{0, \left\lceil \frac{1}{2}\log\left(\frac{\beta^{m_k}_{min}}{\Theta(m_k)}\right)\right\rceil\right\}, \quad m_k = 12 \text{ or } 15.$$

Note that $\Theta_8 < \Theta_{12}/4$ and $\Theta_{12} > \Theta_{15}/4$, and then only $m_k = 12$ and 15 are efficient orders for scaling.

The algorithm without estimation of norms of matrix powers uses bounds of matrix powers based on products of matrix powers previously computed and is analogous to Algorithm 2 from [5]: For $m_k \le 8$, only B and B^2 are available and then, using Theorem 1 and (15), we take

$$\beta^{(m_k)}_{min} = \beta_2 = \max\left\{\left(\|B^2\|^{\frac{m_k}{2}}\|B\|\right)^{\frac{1}{m_k+1}}, \left(\|B^2\|^{\frac{m_k+2}{2}}\right)^{\frac{1}{m_k+2}}\right\}$$
$$= \left(\|B^2\|^{\frac{m_k}{2}}\|B\|\right)^{\frac{1}{m_k+1}}.$$

For $m_k = 12$ and 15, B^3 is also available and then, using Theorem 1, we take $\beta_{min}^{(m_k)} = \min\{\beta_2, \beta_3\}$. Functions `ms_selectNoNormEst` and `beta_NoNormEst` from http://personales.upv.es/jorsasma/software/cosmpol.m are MATLAB implementations of the scaling algorithm with no estimation of norms of matrix powers, and for the computation of β_2 and β_3, respectively.

The algorithm with estimation of norms of matrix powers uses the estimation of two matrix powers taking into account the simplifications (15) and (16), where values d_k are computed approximately by the block 1-norm estimation algorithm of [14], and is also analogous to that on [5]: It reduces the number of estimations combining estimations of the values d_k based on products of norms of matrix powers previously computed or estimated, see function `ms_selectNormEst` from `cosmpol.m`.

4. Numerical experiments

In this section, we compare the new MATLAB function developed in this paper, `cosmpol`, with another two functions:

- `cosm`: Code based on the Padé rational approximation for the matrix cosine [3]. The MATLAB function `cosm` has an argument which allows us to compute $\cos(A)$ by means of just Padé approximants, or also using the real Schur decomposition and the complex Schur decomposition. In these tests we did not use the Schur decomposition since, in the tests carried out in [5] it was shown that, using the Schur decomposition in Taylor algorithms from [5] provides higher efficiency than Padé method from [3] with the Schur decomposition with similar accuracy. The MATLAB code can be found in: http://github.com/sdrelton/cosm_sinm.
- `cosmtay`: Code based on the Taylor series for the matrix cosine [5]. This code allows us to use or not norm estimation. In this paper, we did not use norm estimation. The MATLAB code of this algorithm can be found in: http://personales.upv.es/jorsasma/software/cosmtay.m.
- `cosmpol`: This is the new code presented in this paper for computing the matrix cosine. This code is also able to use or not norm estimation but, as with `cosmtay`, we have not used norm estimation here.

4.1. Numerical tests

To test and compare the accuracy of the three algorithms we define the following tests:

- Test 1: One hundred diagonalizable 128×128 real matrices with a 1-norm varying from 2.32 to 220.04. These matrices have the form $A = V^T DV$, where D is diagonal with real and complex eigenvalues, and V is an orthogonal matrix obtained as $V = H/16$, being H a Hadamard matrix, i.e. a square matrix whose entries are either $+1$ or -1 and whose rows are mutually orthogonal, being $H^{-1} = H^T$, where H^T is the matrix H transposed.
- Test 2: One hundred non diagonalizable 128×128 real matrices whose 1-norms vary from 6.5 to 249.5. These matrices have the form $A = V^T JV$, where J is a Jordan matrix with real and complex eigenvalues. The algebraic multiplicity of the eigenvalues varies between 1 and 3. Matrix V is an orthogonal matrix obtained as $V = H/16$, being H a Hadamard matrix.
- Test 3: Fifteen matrices with dimensions lower than or equal to 128 from the Eigtool MATLAB package [15] and forty four 128×128 real matrices from the function `matrix` of the Matrix Computation Toolbox [16]. Those matrices whose condition number cannot be calculated have dropped from the test. In addition, we have scaled some matrices of this test so that their 1-norm is lower than or equal to 1024 and their matrix cosine can be calculated with the compared functions.

The *"exact"* matrix cosine is computed as $\cos(A) = V^T \cos(D)V$, for matrices of Test 1, and $\cos(A) = V^T \cos(J)V$, for matrices of Test 2 (see [9, p. 10]), by using the MATLAB's Symbolic Math Toolbox with 256 decimal digit arithmetic in all the computations. Following [4, Sec. 4.1], for the other matrices we used MATLAB symbolic versions of a scaled Padé rational approximation from [3] and a scaled Taylor Paterson–Stockmeyer approximation [5, p. 67], both with 4096 decimal digit arithmetic and several orders m and/or scaling parameters s higher than the ones used by `cosm` and `cosmtay`, respectively, checking that their relative difference was small enough. The algorithm accuracy was tested by computing the relative error

$$\mathrm{E} = \frac{\| \cos(A) - \tilde{Y} \|_1}{\|\cos(A)\|_1},$$

where $\tilde{Y}$ is the computed solution and $\cos(A)$ is the exact solution.

To compute the condition number of a matrix cosine function we have used the MATLAB function `funm_condest1`, which estimates the condition number for the matrix 1-norm.

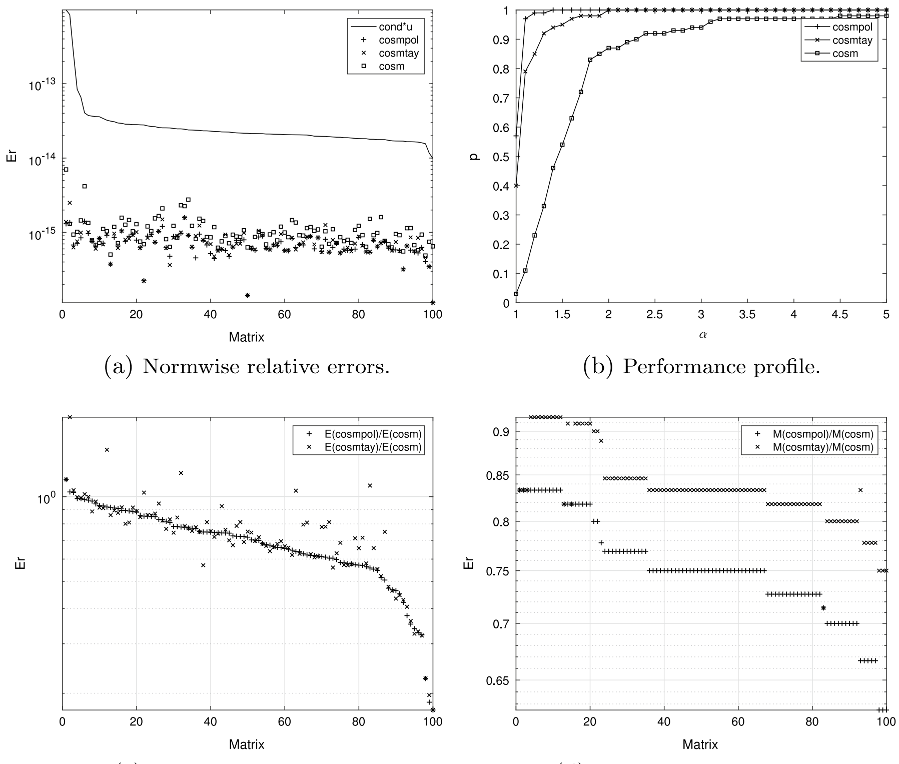
Table 3 shows the computational costs. In this table, the computational cost of each algorithm has been calculated by counting the number of matrix products (M) of each code, since the cost of the rest of operations is negligible compared to matrix products for big enough matrices. The cost of the resolution of linear systems that appears in the code based on Padé approximations has been calculated as 4/3 products because, from a computational point of view, the cost of that operation

Table 3
Matrix products (M) for the three tests using MATLAB functions `cosmpol`, `cosmtay`, and `cosm`. The values shown in columns `cosmtay` and `cosm` are the percentage of extra products carried out by these routines w.r.t. `cosmpol`.

	M(`cosmpol`)	M(`cosmtay`)	M(`cosm`)
Test 1	854	11.00%	32.20%
Test 2	871	10.67%	31.57%
Test 3	511	9.20%	31.70%

Table 4
Relative error comparison.

	Test 1	Test 2	Test 3
E(`cosmpol`) < E(`cosm`)	97%	97%	71.27%
E(`cosmpol`) < E(`cosmtay`)	60%	55%	50.85%



(a) Normwise relative errors. (b) Performance profile.
(c) Ratio of relative errors. (d) Ratio of matrix products.

Fig. 1. Experimental results for Test 1.

compared to the cost of a matrix product is approximately equal to 4/3 (see Table C.1 from [9, p. 336]). According to the figures in this table, `cosmpol` is clearly faster than the other two routines.

To compare the relative errors we can see Table 4. This table shows the percentage of cases in which the relative errors of `cosmpol` are lower than the relative errors of MATLAB codes `cosm` and `cosmtay`.

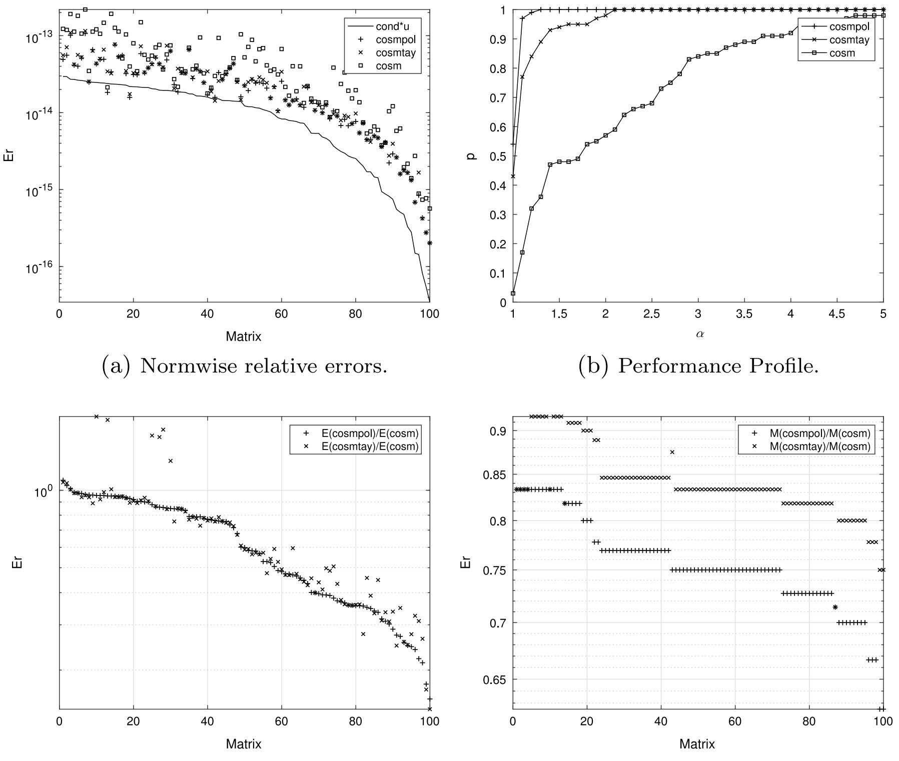
We have plotted in Figs. 1, 2, and 3 the Normwise relative errors (a), the Performance Profiles (b), and the ratio of relative errors (c) to show if these ratios are significant:

$$E(\texttt{cosmpol})/E(\texttt{cosm}), \qquad E(\texttt{cosmtay})/E(\texttt{cosm}),$$

and the ratio of the matrix products (d):

$$M(\texttt{cosmpol})/M(\texttt{cosm}), \qquad M(\texttt{cosmtay})/M(\texttt{cosm}),$$

for the three tests, respectively. In the performance profile, the α coordinate varies between 1 and 5 in steps equal to 0.1, and the p coordinate is the probability that the considered algorithm has a relative error lower than or equal to α-times the smallest error over all methods. The ratios of relative errors are presented in decreasing order with respect to



(a) Normwise relative errors. (b) Performance Profile.

(c) Ratio of relative errors. (d) Ratio of matrix products.

Fig. 2. Experimental results for Test 2.

E(cosmpol)/E(cosm). The solid lines in Figs. 1(a), 2(a) and 3(a) represent function $k_{\cos}u$, where $k_{\cos}$ is the condition number of the matrix cosine function [9, Chap. 3]. Value $u = 2^{-53}$ represents the unit roundoff in double precision floating-point arithmetic.

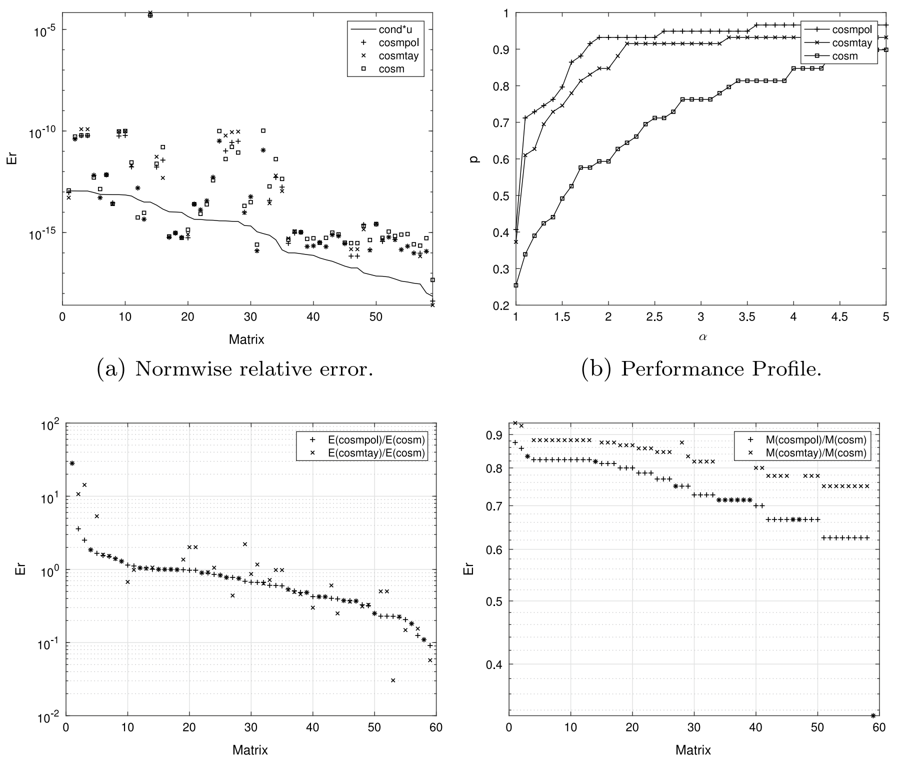
In the light of the results shown by the tables and figures we can make the following analysis:

- Regarding numerical stability we present figures showing the normwise relative error: Fig. 1(a) shows that all the functions behave in a numerical stable way in Test 1. Fig. 2(a) shows that in Test 2 Taylor functions are more stable numerically than the Padé function `cosm`. Fig. 3(a) shows that all the three functions have a similar numerical stability in Test 3. Only in one matrix of this test all the three functions present certain numerical instability, with a relative error more than 10^8 higher than the solid line (see Fig. 3(a)).
- The functions based on polynomial approximations are more accurate than the one based on Padé approximants, being the new function `cosmpol` slightly more accurate than our former `cosmtay` function. The performance profile (Figs. 1(b), 2(b), and 3(b)) shows that the graph of `cosmpol` is above the graphs of the other two functions demonstrating that, in general, it is the most accurate. This is also shown by Table 4 where we see that function `cosmpol` has a lower relative error, between 71.27%—97% of the matrices than `cosm`, and between 50.85%—60% of the matrices, when it is compared with `cosmtay`.
- Regarding the computational costs, Table 3 shows that function `cosmpol` has a lower computational cost than the other two functions. This is also confirmed by Figs. 1(d), 2(d), and 3(d), which show that the ratios of matrix products computed for functions `cosmpol` and `cosm`, i.e. M(cosmpol)/M(cosm) are lower than 1 for all the test matrices, and lower in almost all cases than the ratio of `cosmtay` and `cosm`, i.e. M(cosmtay)/M(cosm).

4.2. The accelerated algorithm

We have implemented an "accelerated" version of Algorithm 1 that can use one *NVIDIA* GPU. The accelerated version of the algorithm has been developed with the aim of being efficient and easy to use, for which we implemented a MATLAB `mex` file.

We use languages CUDA and C++ to implement the `mex` file. This code is to accelerate those parts of the original Matlab function that have a high computational cost, i.e. matrix multiplications. In this work we have taken the `mex` file developed in [8] to modify and adapt it to `cosmpol` using the new method for selecting the degree m and scaling parameter



(a) Normwise relative error. (b) Performance Profile.

(c) Ratio of relative errors. (d) Ratio of matrix products.

Fig. 3. Experimental results for Test 3.

s from Section 3, corresponding to Step 2 of Algorithm 1, and the new methods for evaluating the Taylor matrix polynomial approximations for the matrix cosine from Section 2, corresponding to Step 4 of Algorithm 1. The `mex` function is unique but allows to perform the different operations required by the algorithm. This way data (matrices) are kept in the device (GPU) memory between consecutive calls to the `mex` function. The GPU is mainly in charge of executing matrix multiplications but also performs low cost operations, e.g. calculation of the 1-norm of a matrix, to avoid transmitting data between CPU and GPU only to perform these operations. Other low cost operations are carried out in the host CPU. The Matlab `mex` function, called `call_gpu`, executes different operations (`init`, `power`, `scale`, ...) depending on the arguments with which it is called [8]. The only operation that has been changed in this paper is `eval`, which evaluates a matrix polynomial. Now, this operation implements Eqs. (3)–(6) using coefficients of Table 1.

With this implementation of Algorithm 1, the Matlab script can be executed in both CPU or GPU. Table 5 shows the execution time (in s) obtained in both devices for randomly generated matrices. To obtain the CPU time we used two processors with 12 cores each (model Intel Xeon CPU E5-2697 v2 @2.70 GHz). Thus, the matrix multiplication used by Matlab exploits the 24 cores available in our host. The GPU time was obtained on an NVIDIA Tesla K20Xm, a high performance device that features 2688 CUDA cores. We observe that our new algorithm `cosmpol` is faster than `cosmtay` in both devices, a reduction in time due to the saving in matrix products. The reduction in time from the CPU to the GPU is not so high in `cosmpol` than in `cosmtay` but is still important since the algorithm is also supported on matrix multiplication, a very optimized operation included into the library CUBLAS [17] for NVIDIA GPUs.

5. Conclusions

In this paper we have introduced a new method to compute the matrix cosine function. This method is based on the Taylor approximation of the cosine function using matrix polynomial evaluation methods from [7] and an improved version of the scaling algorithm from [5]. From the different real solutions of the coefficients of the formulae from [7], the coefficients selected in this paper give the lowest maximum error in the stability check from [7], providing a maximum order of approximation $m_M = 15$, i.e. a maximum order of approximation of the cosine Taylor series equal to 30, and excellent accuracy results in numerical tests.

A MATLAB implementation, (`cosmpol`) based on that method has been developed and compared with other state-of-the-art algorithms based on Taylor (`cosmtay`) approximations using Paterson–Stockmeyer method for evaluating the Taylor matrix polynomial approximations, and Padé (`cosm`) approximants. Numerical experiments show that, in general, `cosmpol`

Table 5
Execution time (s) of the algorithm in CPU and the accelerated version in GPU.

	`cosmtay`		`cosmpol`	
n	CPU	GPU	CPU	GPU
1000	0.21	0.19	0.17	0.16
1500	0.54	0.37	0.52	0.31
2000	1.05	0.56	0.77	0.48
2500	1.98	0.93	1.83	0.81
3000	3.36	1.40	3.26	1.23
3500	5.12	1.97	4.71	1.78
4000	7.19	2.69	5.83	2.46
4500	8.30	3.61	8.07	3.32
5000	10.86	4.77	10.13	4.36
5500	15.55	6.13	15.02	5.62
6000	26.01	7.91	21.83	7.33

has a computational cost in terms of matrix products lower than the other functions `cosmtay` and `cosm`; also, `cosmpol` has a higher accuracy in the majority of tests than the other codes with a similar numerical stability.

Finally, we note that all of the above discussion for the fast computation of the matrix cosine is applicable to the computation of the matrix sine since $\sin(A) = \cos(A - \pi/2I)$.

Acknowledgments

This work has been partially supported by Spanish Ministerio de Economía y Competitividad and European Regional Development Fund (ERDF) grants TIN2014-59294-P, and TIN2017-89314-P.

References

[1] E. Defez, J. Sastre, J.J. Ibáñez, P.A. Ruiz, Computing matrix functions arising in engineering models with orthogonal matrix polynomials, Math. Comput. Model. 57 (7–8) (2013) 1738–1743.
[2] J. Sastre, J. Ibáñez, P. Ruiz, E. Defez, Efficient computation of the matrix cosine, Appl. Math. Comput. 219 (2013) 7575–7585.
[3] A.H. Al-Mohy, N.J. Higham, S.D. Relton, New algorithms for computing the matrix sine and cosine separately or simultaneously, SIAM J. Sci. Comput. 37 (1) (2015) A456–A487.
[4] P. Alonso, J. Ibáñez, J. Sastre, J. Peinado, E. Defez, Efficient and accurate algorithms for computing matrix trigonometric functions, J. Comput. Appl. Math. 309 (2017) 325–332.
[5] J. Sastre, J. Ibáñez, P. Alonso, J. Peinado, E. Defez, Two algorithms for computing the matrix cosine functions, Appl. Math. Comput. 312 (2017) 66–77.
[6] M.S. Paterson, L.J. Stockmeyer, On the number of nonscalar multiplications necessary to evaluate polynomials, SIAM J. Comput. 2 (1) (1973) 60–66.
[7] J. Sastre, Efficient evaluation of matrix polynomials, Linear Algebra Appl. 539 (2018) 229–250.
[8] P. Alonso, J. Peinado, J. Ibáñez, J. Sastre, E. Defez, Computing matrix trigonometric functions with GPUs through Matlab, J. Supercomput. (2018) Online.
[9] N.J. Higham, Functions of Matrices: Theory and Computation, SIAM, Philadelphia, PA, USA, 2008.
[10] G.I. Hargreaves, N.J. Higham, Efficient algorithms for the matrix cosine and sine, Numer. Algorithms 40 (2005) 383–400.
[11] J. Sastre, J.J. Ibáñez, E. Defez, P.A. Ruiz, Accurate matrix exponential computation to solve coupled differential models in engineering, Math. Comput. Model. 54 (2011) 1835–1840.
[12] J. Sastre, J.J. Ibáñez, E. Defez, Boosting the computation of the matrix exponential, Appl. Math. Comput. 340 (2019) 206–220.
[13] P. Ruiz, J. Sastre, J. Ibáñez, E. Defez, High perfomance computing of the matrix exponential, J. Comput. Appl. Math. 291 (2016) 370–379.
[14] N.J. Higham, FORTRAN codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation, ACM Trans. Math. Software 14 (4) (1988) 381–396.
[15] T.G. Wright, Eigtool, version 2.1, 2009, URL web.comlab.ox.ac.uk/pseudospectra/eigtool.
[16] N.J. Higham, The Test Matrix Toolbox for MATLAB, Report No. 237, Manchester, England, 1993.
[17] NVIDIA, CUDA toolkit. cuBLAS library, v9.2.148 ed., 2018. docs.nvidia.com/cuda/cublas. (Last accessed July, 2018).