Document downloaded from:

http://hdl.handle.net/10251/101678

This paper must be cited as:

Defez Candel, E.; Ibáñez González, JJ.; Sastre, J.; Peinado Pinilla, J.; Alonso-Jordá, P. (2018). A new efficient and accurate spline algorithm for the matrix exponential computation. Journal of Computational and Applied Mathematics. 337(1):354-365. doi:10.1016/j.cam.2017.11.029

The final publication is available at

https://doi.org/10.1016/j.cam.2017.11.029

Copyright Elsevier

Additional Information

A new efficient and accurate spline algorithm for the matrix exponential computation[☆]

Emilio Defez[⋆], Javier Ibáñez[♮], Jorge Sastre[†], Jesús Peinado[♮], Pedro Alonso[‡]

⋆ Instituto de Matemática Multidisciplinar.
♮ Instituto de Instrumentación para Imagen Molecular.
† Instituto de Telecomunicaciones y Aplicaciones Multimedia.
‡ Grupo Interdisciplinar de Computación y Comunicaciones.
Universitat Politècnica de València,
Camino de Vera s/n, 46022, Valencia, Spain.
edefez@imm.upv.es, jjibanez@dsic.upv.es, jsastrem@upv.es, {jpeinado, palonso}@dsic.upv.es

Abstract

In this work an accurate and efficient method based on matrix splines for computing matrix exponential is given. An algorithm and a MATLAB implementation have been developed and compared with the state-of-the-art algorithms for computing the matrix exponential. We also developed a parallel implementation for large scale problems. This implementation allowed us to get a much better performance when working with this kind of problems.

Keywords: Matrix exponential, Scaling and squaring method, Taylor series, Matrix splines, NVIDIA, GPGPU, Parallel Computing.

1. Introduction

Matrix exponential computation has received remarkable attention in the last decades due to its usefulness in the solution of systems of linear differential equations. Moreover, in many cases, the resolution of these systems

[☆]This work has been supported by Spanish Ministerio de Economía y Competitividad and the European Regional Development Fund (ERDF) TIN2014-59294-P.

involve large or perturbed matrices [1]. Thus, the use of not only accurate, but also efficient methods becomes absolutely necessary. All the usual methods proposed for matrix exponential computation [2, 3] use the squaring and scaling techniques, and the most accurate and efficient methods are based on Taylor series expansion [4, 5] and on the Padé rational approximation [6].

In this work, we propose an accurate and efficient method to obtain the exponential of a matrix which is based on matrix splines [7]. The algorithm has been implemented in MATLAB and compared with other state-of-the-art existing algorithms for computing the matrix exponential.

In addition, we have also developed a high performance implementation with the aim of addressing large scale problems. This implementation uses a GPU (Graphics Processor Unit) which is, in particular, an NVIDIA GPU. The GPU is used by the MATLAB algorithm through the specific instructions provided by the Parallel MATLAB Toolbox.

All along this paper we will adopt the following notation. We denote by $\mathbb{C}^{n\times n}$ the set of complex matrices of size $n \times n$, and I denotes the identity matrix. Symbol $\mathbb{N}$ denotes the set of positive integers. The matrix norm $\|\cdot\|$ denotes any subordinate matrix norm, in particular $\|\cdot\|_1$ is the 1-norm. The least integer not less than x is denoted by$\lceil x \rceil$, $\lfloor x \rfloor$ denotes the greatest integer not exceeding x, and $\rho(X)$ is the spectral radius of matrix X. Function log is the principal logarithm defined as the complex logarithm whose imaginary part is in $(-\pi, \pi]$.

The paper is organized as follows. Section 2 presents the proposed Taylor-Spline method. Section 3 deals with algorithms for computing the exponential matrix. Section 4 describes the sequential numerical experiments. In Section 5, we show the implementations proposed with the Parallel MATLAB Toolkit and also the experiments results using the GPU. The conclusions are given in last section.

2. Taylor-Spline method

The matrix exponential for a matrix $A \in \mathbb{C}^{n\times n}$ can be defined as

$$e^A = \sum_{i=0}^{\infty} \frac{A^i}{k!},$$

and

$$T_m(A) = \sum_{i=0}^{m} \frac{A^i}{i!}, \tag{1}$$

is the Taylor approximation of order m of e^A. The Taylor approximation $T_m(A)$ can be computed efficiently by using the Paterson-Stockmeyer's method [8][9, p.574][3, 72–74], by using the following expression:

$$\begin{aligned} T_m(A) = & \left(\left(\cdots \left(\frac{A^q}{m!} + \frac{A^{q-1}}{(m-1)!} + \cdots + \frac{A}{(m-q+1)!} + \frac{I}{(m-q)!} \right) \right. \right. \\ & \times A^q + \frac{A^{q-1}}{(m-q-1)!} + \frac{A^{q-2}}{(m-q-2)!} + \cdots + \frac{A}{(m-2q+1)!} + \frac{I}{(m-2q)!} \Bigg) \\ & \times A^q + \frac{A^{q-1}}{(m-2q-1)!} + \frac{A^{q-2}}{(m-2q-2)!} + \cdots + \frac{A}{(m-3q+1)!} + \frac{I}{(m-3q)!} \Bigg) \\ & \cdots \\ & \times A^q + \frac{A^{q-1}}{(q-1)!} + \frac{A^{q-2}}{(q-2)!} + \cdots + A + I. \end{aligned} \tag{2}$$

In [10, p. 6455][3, p. 74] it is shown that, in order to compute more efficiently the Taylor approximation (1) through the Paterson-Stockmeyer's method, the optimal values for the polynomial degree m of (2) must belong to the set:

$$\mathbb{M} = \{2, 4, 6, 9, 12, 16, 20, 25, 30, 36, 42, 49, \ldots\}. \tag{3}$$

Let's denote as $m_1, m_2, m_3, \ldots, m_k, \ldots$ the elements of set $\mathbb{M}$, for $k = 1, 2, \ldots$, then the optimal values of q (2) for a given m_k can be either $q_k = \lceil \sqrt{m_k} \rceil$ or $q_k = \lfloor \sqrt{m_k} \rfloor$. The two values are integer divisors of m_k and the cost of evaluating (2) is the same for the two degrees. Table 1 shows the values for $q_k = \lceil \sqrt{m_k} \rceil$ which are the actual selected.

k	1	2	3	4	5	6	7	8	9	10	11	12	...
m_k	2	4	6	9	12	16	20	25	30	36	42	49	...
q_k	2	2	3	3	4	4	5	5	6	6	7	7	...

Table 1: Values of m_k and q_k for the first 12 elements of $\mathbb{M}$.

Taking into account Table 1, the cost of evaluating $T_{m_k}(A)$ in terms of matrix products, which we denote by Π_{m_k}, is

$$\Pi_{m_k} = k. \tag{4}$$

The problem of applying algorithms based on the Taylor expression (1) is that this approximation is accurate only near the origin, hence the norm of matrix A must be reduced by using techniques based on the scaling an squaring method [11, p. 241]. The idea is to use the identity

$$e^A = \left(e^{A/2^s}\right)^{2^s}, \tag{5}$$

where s is a positive integer, and to apply the following approximation:

$$e^A \cong \left(T_m(A/2^s)\right)^{2^s}.$$

In the next two subsections we derive expressions for the backward and forward errors, respectively, of computing the matrix exponential through the evaluation of a Taylor series like (1).

2.1. Backward error analysis

The backward error, ΔA, of computing e^A by means of $T_m(A)$ verifies

$$e^{A+\Delta A} = T_m(A).$$

If we assume that ΔA and A conmute, and $\rho\left(e^{-A}T_m(A) + I\right) < 1$ (see [4]), then

$$\begin{aligned} &e^A e^{\Delta A} = T_m(A), \\ &e^{\Delta A} = e^{-A}T_m(A), \\ &\Delta A = \log\left(e^{-A}T_m(A)\right) = \log\left(T_m(A)\right) - A. \end{aligned}$$

Developing the above expression in Taylor series, it is obtained

$$\Delta A = \sum_{i=m+1}^{\infty} p_i A^i,$$

and applying now Theorem 1.1 from [6], the relative backward error e_b verifies then that

$$e_b = \frac{\|\Delta A\|}{\|A\|} = \frac{\left\|\sum_{i=m+1}^{\infty} p_i A^i\right\|}{\|A\|} \leq \frac{\sum_{i=m+1}^{\infty} |p_i| \, \|A^i\|}{\|A\|} \leq \sum_{i=m}^{\infty} |p_{i+1}| \, \left\|A^i\right\|$$

$$\leq \sum_{i=m}^{\infty} |p_{i+1}| \left(\left\|A^i\right\|^{1/i}\right)^i \leq \sum_{i=m}^{\infty} |p_{i+1}| \alpha_m^i,$$

where

$$\alpha_m = \max\left\{||A^i||^{1/i} : i \geq m \text{ and } p_{i+1} \neq 0\right\},$$

and, using Theorem 2 from [12], we obtain

$$\alpha_m = \max\left\{||A^i||^{1/i} : \ m \leq i \leq 2m-1 \text{ and } p_{i+1} \neq 0\right\}. \tag{6}$$

Let

$$\Theta_m^{(0)} = \max\left\{\theta \geq 0 : \ \sum_{i=m}^{\infty} |p_{i+1}|\theta^i \leq u\right\},$$

be a threshold value for the backward error used in the algorithms, where $u = 2^{-53}$ is the unit roundoff in the double precision floating-point. In order to compute $\Theta_m^{(0)}$, we can use a symbolic calculation tool like *Mathematica*. For the particular case of the first 9 elements of $\mathbb{M}$, i.e. m_k for $i = 1, \ldots, 9$, the values of $\Theta_{m_k}^{(0)}$ are those shown in the second column of Table 2.

2.2. Forward error analysis

Applying Theorem 1.1 of [6] and Theorem 2 of [12] as it has been previously done, the forward relative error e_f can be computed as follows:

$$e_f = \left\|e^{-A}\left(e^A - T_m(A)\right)\right\| = \left\|I - e^{-A}T_m(A)\right\| = \left\|\sum_{i=m+1}^{\infty} |\bar{p}_i|A^i\right\| \leq$$
$$\leq \sum_{i=m+1}^{\infty} |\bar{p}_i|\,\|A^i\| \leq \sum_{i=m+1}^{\infty} |\bar{p}_i|\left(\|A^i\|^{1/i}\right)^i \leq \sum_{i=m+1}^{\infty} |\bar{p}_i|\bar{\alpha}_m^i,$$

where

$$\bar{\alpha}_m = \max\left\{||A^i||^{1/i} : m+1 \leq \ i \leq 2m \text{ and } \bar{p}_i \neq 0\right\}. \tag{7}$$

The threshold value used now for the forward error is

$$\Theta_m^{(1)} = \max\left\{\theta \geq 0 : \ \sum_{i=m+1}^{\infty} |\bar{p}_i|\theta^i \leq u\right\},$$

and it can also be computed using *Mathematica*. As before, the values of $\Theta_{m_k}^{(1)}$, for $k = 1, \ldots, 9$, are shown in the second column of Table 2.

m_k	$\Theta_{m_k}^{(0)}$	$\Theta_{m_k}^{(1)}$
2	2.675298260329713e-8	8.733457635286420e-6
4	3.397168839977002e-4	1.678018844321752e-3
6	9.065656407595296e-3	1.773082199654024e-2
9	8.957760203223343e-2	1.137689245787824e-1
12	2.996158913811581e-1	3.280542018037261e-1
16	7.802874256626574e-1	7.912740176600239e-1
20	1.438252596804337	1.415070447561532
25	2.428582524442827	2.353642766989427
30	3.539666348743690	3.411877172556770

Table 2: Values of $\Theta_{m_k}^{(0)}$ and $\Theta_{m_k}^{(1)}$.

2.3. Determination of the Taylor order and the scaling factor

In this subsection we show how to properly obtain the Taylor approximation order m (1) and the scaling factor s introduced in (5).

Let Θ_m be defined as $\Theta_m = \max\left\{\Theta_m^{(0)}, \Theta_m^{(1)}\right\}$ ($\Theta_m = \Theta_m^{(1)}$ for $m \geq 16$, or $\Theta_m = \Theta_m^{(0)}$ otherwise). We denote by m_M the maximum Taylor approximation order allowed, and select a value for it such that $m_M \geq 20$. Then there exist two possibilities:

- $\alpha_m \leq \Theta_m$, for some $m \leq m_M$. In this case, $e_f < u$ or $e_b < u$:
  - If $m \geq 16$, $\bar{\alpha}_m \leq \alpha_m \leq \Theta_m = \Theta_m^{(1)}$, hence $e_f < u$.
  - If $m \geq 20$, $\alpha_m \leq \Theta_m = \Theta_m^{(0)}$, hence $e_b < u$.
- $\alpha_{m_M} > \Theta_{m_M} = \Theta_{m_M}^{(0)}$. In this case, we select the first positive integer s that verifies
$$2^{-s}\alpha_{m_M} \leq \Theta_{m_M},$$
that is
$$s = \left\lceil \log_2\left(\frac{\alpha_{m_M}}{\Theta_{m_M}}\right)\right\rceil,$$
and then the relative backward error to compute $e^{2^{-s}A}$ is lower than u, i.e.
$$\frac{\|\Delta 2^{-s}A\|}{\|2^{-s}A\|} \leq u.$$

Instead of using (6) and (7) to compute α_m and $\bar{\alpha}_m$, respectively, we perform the computation of $||A^m||^{1/m}$ only. Since this may adversely affect the accuracy of the resulting factor e^A, we indeed calculate

$$\bar{T}_m(A) = \sum_{i=0}^{m-1} \frac{A^i}{i!} + \Phi,$$

instead of $T_m(A)$ (1), being $\Phi \in \mathbb{C}^{n\times n}$ an unknown matrix that must be previously calculated. To compute matrix Φ we require

$$\bar{T}_m(x) = \sum_{i=0}^{m-1} \frac{A^i}{i!} x^i + \Phi x^m \tag{8}$$

to be the solution, at $x = 1$, of the Cauchy problem

$$\left.\begin{array}{rcl} Y'(x) & = & AY(x) \\ Y(0) & = & I \end{array}\right\}, \ x \in [0,1]. \tag{9}$$

Using (8) and (9) we obtain

$$\begin{aligned} A\sum_{i=0}^{m-2} \frac{A^i}{i!} x^i + m\Phi x^{m-1} &= A\sum_{i=0}^{m-1} \frac{A^i}{i!} x^i + A\Phi x^m, \\ m\Phi x^{m-1} &= A\frac{A^{m-1}}{(m-1)!} x^{m-1} + A\Phi x^m, \end{aligned}$$

and assuming that $x = 1$ in the above expression, matrix Φ is obtained by solving the linear matrix equation:

$$(mI - A)\,\Phi = \frac{A^m}{(m-1)!}. \tag{10}$$

If we apply (2) to $\bar{T}_m(A)$ so that

$$\begin{aligned} \bar{T}_m(A) = \Bigg(\Bigg(\cdots\Bigg(\bar{\Phi} &+ \frac{A^{q-1}}{(m-1)!} + \cdots + \frac{A}{(m-q+1)!} + \frac{I}{(m-q)!}\Bigg) \\ &\times A^q + \frac{A^{q-1}}{(m-q-1)!} + \frac{A^{q-2}}{(m-q-2)!} + \cdots + \frac{A}{(m-2q+1)!} + \frac{I}{(m-2q)!}\Bigg) \\ &\times A^q + \frac{A^{q-1}}{(m-2q-1)!} + \frac{A^{q-2}}{(m-2q-2)!} + \cdots + \frac{A}{(m-3q+1)!} + \frac{I}{(m-3q)!}\Bigg) \\ &\cdots \\ &\times A^q + \frac{A^{q-1}}{(q-1)!} + \frac{A^{q-2}}{(q-2)!} + \cdots + A + I, \end{aligned} \tag{11}$$

we have that $\bar{\Phi}$ is the solution to the equation

$$(mI - A)\,\bar{\Phi} = \frac{A^q}{(m-1)!}. \tag{12}$$

3. Algorithms for computing the matrix exponential

Algorithm 1 computes the matrix exponential based on the method described above for a maximum order of $m_M = 30$. To compute the Taylor approximation we use (11) if the condition number of $mI - A$ is lower than 100, or using (2) otherwise. Of course, the value 100 can be replaced by another one in order to increase the accuracy of the solution of the linear matrix equation (12). The idea is to avoid solving a possible ill-conditioned system.

Algorithm 1 E=expmspl(A)

Given a matrix $A \in \mathbb{C}^{n\times n}$, this algorithm computes $E = e^A$ by a Taylor-spline approximation of order m_k to e^A.

1: $[m_k, s, pA]$ = select_m_s(A) or $[m_k, s, pA]$ = select_m_s_w(A)
2: $B = m_k I - A$
3: **if** cond(B) < 100 **then** ▷ cond(B) returns the condition number of B
4: Compute $E = \bar{T}_{m_M}(pA, m_k, s)$ from (11)
5: **else**
6: Compute $E = T_{m_M}(pA, m_k, s)$ from (2)
7: **end if**
8: **for** $i = 1 : s$ **do**
9: $E \Leftarrow E^2$
10: **end for**

In order to get the optimal values of m and s, we propose the following algorithms:

- Algorithm 2. This algorithm estimates $||A^m||$ from matrices A^q previously computed, using the block 1–norm estimation algorithm of Higham and Tisseur [13]. For instance, in the case $m = 30$, then $q = 6$ (see Table 1) so matrices $A_1 = A$, $A_2 = A^2$, $A_3 = A^3$, $A_4 = A^4$, $A_5 = A^5$, $A_6 = A^6$ have to be computed and then $||A^{30}||$ can be estimated by estimating $||A_6^5||$.

- Algorithm 3. In this algorithm we bound $||A^m||$ from the products of bounds or norms of matrices that have been previously computed. For instance, in the case $m = 16$ we use the bound $\min\{a_3a_9, a_4a_2a_6, a_4^3\}$, where $a_2 = \|A^2\|$, $a_3 = \|A^3\|$, $a_4 = \|A^4\|$, being a_6 and a_9 bounds of $\|A^6\|$ and $\|A^9\|$, respectively.

Algorithm 2 $[m, s, A_1, A_2, A_3, \cdots, A_q] = \text{select\_m\_s}(A)$

Given a matrix $A \in \mathbb{C}^{n\times n}$ and maximum order $m_M = 30$, this function returns the optimal values of m and s, and computes the powers $\left\{A_1 = A, A_2 = A^2, A_3 = A^3, \cdots, A_q = A^q\right\}$.

$s = 0$
$A_1 = A$, $a_1 = \|A_1\|_1$, $A_2 = AA$, $a_1 = \|A_2\|_1$
if $a_1 \le \theta_1$ **then**
  $m = 2$ **return**
end if
$A_2 = A_1A_1$, $a_2 = \|A_2\|_1$, estimate $a_4 = \|A_2\|^2$, $\alpha = \sqrt[4]{a_4}$
if $\alpha \le \theta_2$ **then**
  $m = 4$ **return**
end if
$A_3 = A_2A_1$, estimate $a_6 = \|A_3\|_1^2$, $\alpha = \sqrt[6]{a_6}$
if $\alpha \le \theta_3$ **then**
  $m = 6$ **return**
end if
Estimate $a_9 = \|A_3\|_1^3$, $\alpha = \sqrt[9]{a_9}$
if $\alpha \le \theta_4$ **then**
  $m = 9$ **return**
end if
$A_4 = A_2A_2$, estimate $a_{12} = \|A_3\|_1^4$, $\alpha = \sqrt[12]{a_{12}}$
if $\alpha \le \theta_5$ **then**
  $m = 12$ **return**
end if
Estimate $a_{16} = \|A_4\|_1^4$, $\alpha = \sqrt[16]{a_{16}}$
if $\alpha \le \theta_6$ **then**
  $m = 16$ **return**
end if
$A_5 = A_4A_1$, estimate $a_{20} = \|A_5\|_1^4$, $\alpha = \sqrt[20]{a_{20}}$
if $\alpha \le \theta_7$ **then**
  $m = 20$ **return**
end if
Estimate $a_{25} = \|A_5\|_1^5$, $\alpha = \sqrt[25]{a_{25}}$
if $\alpha \le \theta_8$ **then**
  $m = 25$ **return**
end if
$A_6 = A_5A_1$, estimate $a_{30} = \|A_6\|_1^5$, $\alpha = \sqrt[30]{a_{30}}$
if $\alpha \le \theta_9$ **then**
  $m = 30$ **return**
end if
$s = \left\lceil \log_2\left(\frac{\alpha_{m_9}}{\Theta_{m_9}}\right)\right\rceil$.
$s_{25} = \left\lceil \log_2\left(\frac{\alpha_{m_8}}{\Theta_{m_8}}\right)\right\rceil$.
if $s_{25} \le s$ **then**
  $m = 25$; $s = s_{25}$
else
  $m = 30$
end if
for i=1:q **do**
  $A_i = A_i/2^{si}$
end for

Algorithm 3 $[m, s, A_1, A_2, A_3, \cdots, A_q] = \text{select\_m\_s\_w}(A)$

Given a matrix $A \in \mathbb{C}^{n\times n}$ and maximum order $m_M = 30$, this function returns the optimal values of m and s, and computes the powers $\left\{A_1 = A, A_2 = A^2, A_3 = A^3, \cdots, A_q = A^q\right\}$.

$s = 0$
$A_1 = A$, $a_1 = \|A_1\|_1$, $A_2 = AA$, $a_1 = \|A_2\|_1$
if $a_1 \leq \theta_1$ **then**
  $m = 2$ **return**
end if
$A_2 = A_1A_1$, $a_2 = \|A_2\|_1$, $a_4 = a_2^2$, $\alpha = \sqrt[4]{a_4}$
if $\alpha \leq \theta_2$ **then**
  $m = 4$ **return**
end if
$A_3 = A_2A_1$, $a_6 = \min\left\{a_2^3, a_3^2\right\}$, $\alpha = \sqrt[6]{a_6}$
if $\alpha \leq \theta_3$ **then**
  $m = 6$ **return**
end if
$a_9 = a_3a_6$, $\alpha = \sqrt[9]{a_9}$
if $\alpha \leq \theta_4$ **then**
  $m = 9$ **return**
end if
$A_4 = A_2A_2$, $a_{12} = \min\left\{a_3a_9, a_4a_2a_6, a_4^3\right\}$, $\alpha = \sqrt[12]{a_{12}}$
if $\alpha \leq \theta_5$ **then**
  $m = 12$ **return**
end if
$a_{16} = a_4a_{12}$, $\alpha = \sqrt[16]{a_{16}}$
if $\alpha \leq \theta_6$ **then**
  $m = 16$ **return**
end if
$A_5 = A_4A_1$, $a_{30} = \min\left\{a_5^4, a_5^2a_9a_1, a_5^2a_4^2a_2, a_5a_{12}a_3, a_4a_{16}\right\}$, $\alpha = \sqrt[20]{a_{20}}$
if $\alpha \leq \theta_7$ **then**
  $m = 20$ **return**
end if
Estimate $a_{25} = a_5a_{20}$, $\alpha = \sqrt[25]{a_{25}}$
if $\alpha \leq \theta_8$ **then**
  $m = 25$ **return**
end if
$A_6 = A_5A_1$, $a_{20} = \min\left\{a_6^5, a_6^3a_{12}, a_6^3a_5^2a_2, a_6^2a_{16}a_2, a_6^2a_5^3a_3, a_6^2a_5^2a_4^2, a_6a_{20}a_4, a_5a_{25}\right\}$, $\alpha = \sqrt[30]{a_{30}}$
if $\alpha \leq \theta_9$ **then**
  $m = 30$ **return**
end if
$s = \left\lceil \log_2\left(\frac{\alpha_{m_9}}{\Theta_{m_9}}\right)\right\rceil$.
$s_{25} = \left\lceil \log_2\left(\frac{\alpha_{m_8}}{\Theta_{m_8}}\right)\right\rceil$.
if $s_{25} + 1 < s$ **then**
  $m = 25$; $s = s_{25}$
else
  $m = 30$
end if
for i=1:q **do**
  $A_i = A_i/2^{si}$
end for

4. Numerical experiments

In a preliminar analysis, we tested the performance of Algorithm 2 and Algorithm 3 to get m and s. We found that the errors obtained with each one are very similar but the execution time, however, is greater when Algorithm 2 is used. Hence, in the following, we will only use Algorithm 1 in combination with Algorithm 3 and this algorithm will be denoted as `expmspl`. We compare our algorithm `expmspl` with the following two legacy MATLAB implementations for the computation of matrix exponential:

- `expm_new`: MATLAB implementation based on Padé approximation from [6].
- `exptayns`: MATLAB implementation of exptaynsv3 from [12].

MATLAB[1] implementations were tested on an Intel Core 2 Duo processor at 3.00 GHz with 4 GB main memory. The description of the tests carried out is the following:

- Test 1: This test is composed by a set of fifty 128×128 diagonalizable real random matrices. The 1-norm of these matrices vary between 2.502 and 132.76.
- Test 2: This test is composed by a set of fifty 128×128 non diagonalizable real random matrices with eigenvalues whose algebraic multiplicity vary between 1 and 10. The 1-norm of these matrices vary between 1 and 74.24.
- Test 3: This test is composed by a set of thirty one 128×128 real matrices obtained from the function `matrix` of the Matrix Computation Toolbox [14]. Some matrices were excluded because their matrix exponential can not be computed in double precision due to overflow errors. The 1-norm of these matrices vary between 1 and 129.

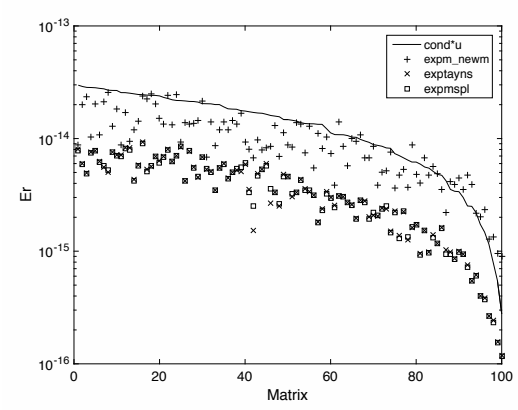
The algorithm accuracy was tested by computing the relative error

$$\mathrm{E} = \frac{\|e^A - \tilde{Y}\|_1}{\|e^A\|_1},$$

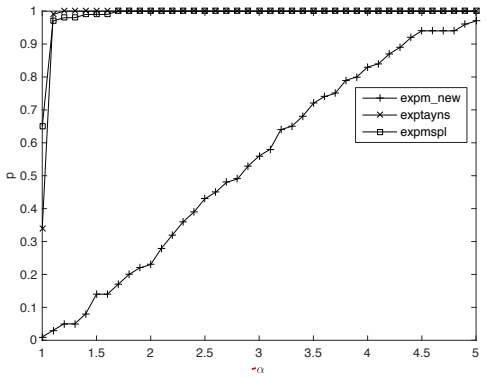
[1]MATLAB version 8.4 (R2014b).

by using the MATLAB Symbolic Math Toolbox, where $\tilde{Y}$ is the computed solution, and e^A is the exact solution (Test 1 and Test 2). When it was not possible to compute analytically, Taylor approximations were used by varying the order and the scaling factor in an iterative process until the norm of the relative difference between the approximations has lower than the unit roundoff in the double precision floating-point u.

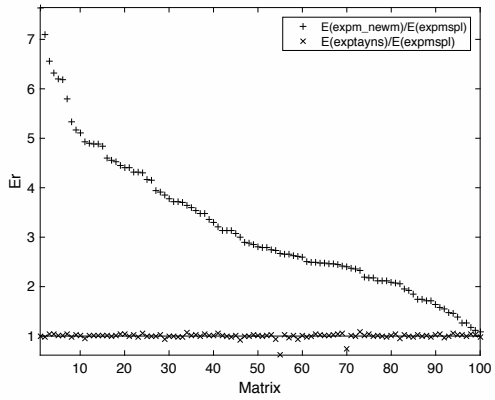
Figures 1, 2 and 3 show the normwise relative errors, the performances, the ratio of relative errors and the ratio of execution times for each test.



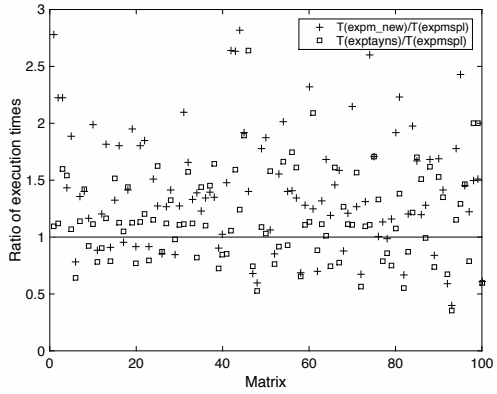
(a) Normwise relative errors.



(b) Perfomances.



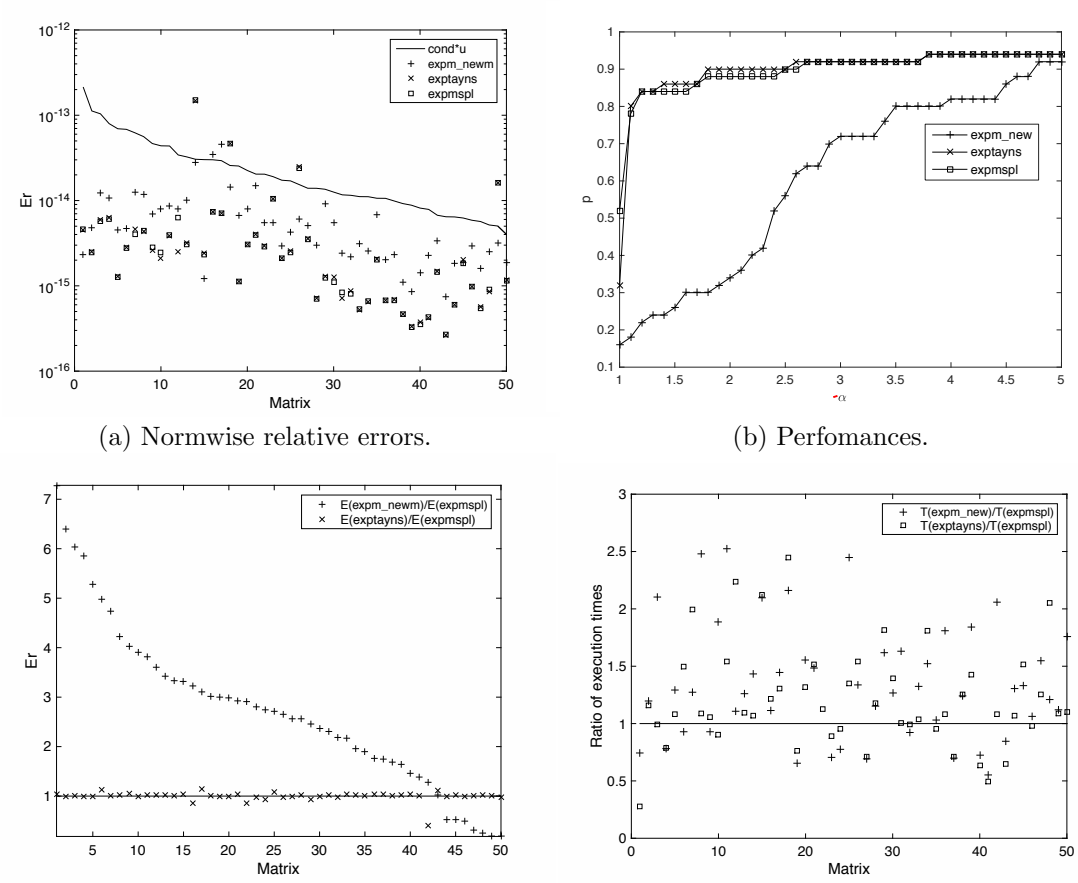
(c) Ratio of relative errors.



(d) Ratio of relative execution times.

Figure 1: Experimental results with Test 1.

The normwise relative errors show the numerical stability of functions `expm_new`, `exptayns`, and `expmspl`. Subfigures 1a, 2a and 3a show the rel-



(a) Normwise relative errors.

(b) Perfomances.

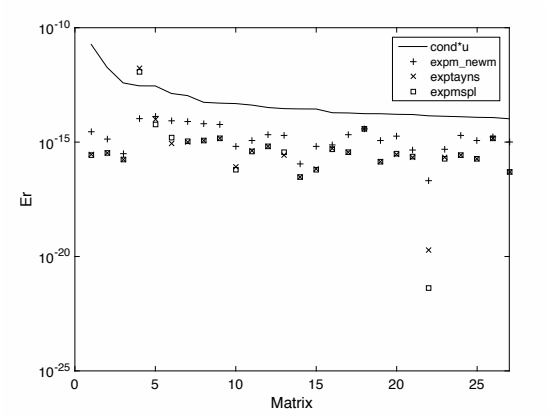
(c) Ratio of relative errors.

(d) Ratio of relative execution times.

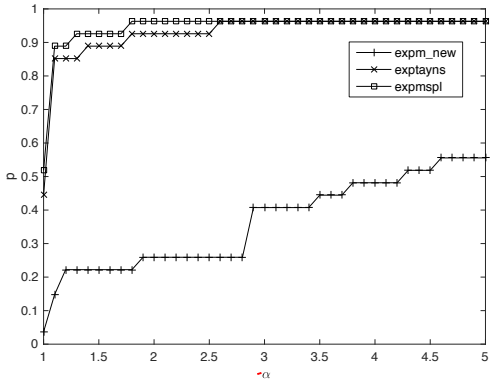
Figure 2: Experimental results with Test 2.

ative errors of all implementations, and a solid line that represents the unit roundoff multiplied by the relative condition number of the exponential function at X [3, p. 55]. The relative condition number was computed using the MATLAB function `funm_condest1` from the Matrix Function Toolbox [3, Appendix D][2]. For a method to perform in a backward and forward stable manner, its error should lie not far above this line on the graph [15, p. 1188]. The normwise subfigures show that all functions performed in a numerically

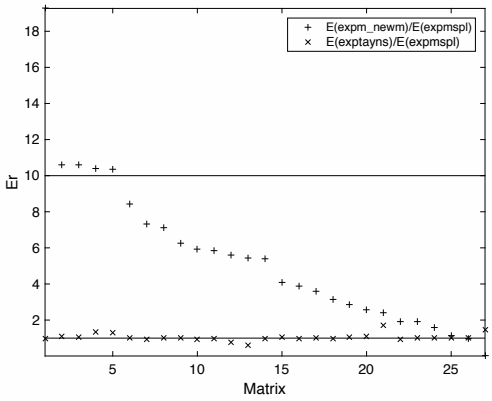
[2] `http://www.maths.manchester.ac.uk/~higham/mftoolbox`



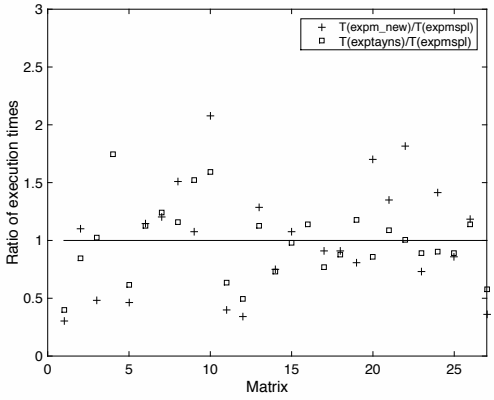
(a) Normwise relative errors.



(b) Perfomances.



(c) Ratio of relative errors.



(d) Ratio of relative execution times.

Figure 3: Experimental results with Test 3.

stable way for all matrices.

Subfigures 1b, 2b and 3b show the performances of the functions [16]. The α value varies between 1 and 5 with a step size equal to 0.1. Value p is the probability that the considered algorithm has a relative error lower than or equal to α-times the smallest error over all the methods.

Subfigures 1c, 2c and 3c show the ratios of errors of `expm_new` and `exptayns` with respect to `expmspl`. Subfigures 1d, 2d and 3d show the ratios of execution times of `expm_new` and `exptayns` with respect to `expmspl`.

According to the results shown in the above figures we can outline the following conclusions:

- The implementations based on Taylor series are more accurate than the implementation based on Padé series (Figures 1b, 1c, 2b, 2c, 3b, and 3c). The implementations based on the Taylor method practically have a similar precision, but the new implementation `expmspl` is slightly more accurate than the implementation `exptayns`, as it can be seen in Tables 3 and 4.
- In general, the implementations based on Taylor series have lower execution times than the implementation based on Padé series (Subfigures 1d, 2d, and 3d). In general, the execution time of `expmspl` is lower than the execution time of `exptayns`.
- Subfigures 1a, 2a and 3a show that the three implementations are numerically stable.

Table 3: Relative error comparative between exptayns(E_2) and `expmspl(`E_3`)` with estimations.

	$E_3 \leq 0.1E_2$	$0.5E2 \leq E_3 < E_2$	$E_2 \leq E_3 < 2E_2$	$2E_2 \leq E_3 < 5E_2$
Test 1	0%	66%	34%	0%
Test 2	0%	62%	36%	2%
Test 3	3.7%	51.86%	44.44%	0%

Table 4: Relative error comparative between exptayns(E_2) and `expmspl(`E_3`)` without estimations.

	$E_3 \leq 0.1E_2$	$0.5E_2 \leq E3 < E_2$	$E_2 \leq E_3 < 2E_2$	$2E_2 \leq E_3 < 5E_2$
Test 1	0%	59%	41%	0%
Test 2	2%	58%	38%	2%
Test 3	7.41%	48.15%	44.44%	0%

5. A high performance solution for the matrix exponential computation

The computational cost of matrix multiplication is $O(n^3)$ flops with a very regular operation pattern whose throughput per data read from memory is

very high. This fact makes matrix multiplication an operation very prone to be improved on manycore processor architectures like are the accelerators we can find attached to computers. For instance, the NVIDIA company delivers, contained in its Software Development Kit, the CUBLAS [17] library, which is an implementation of BLAS routines featuring a very optimized matrix multiplication for NVIDIA GPUs. This way, one of the most important avails of the Taylor series approach for the computation of a matrix exponential is the intensive use of matrix multiplications, an operation that can clearly benefit from the presence of accelerators in the host computer, which is, in turn, a very common situation in current workstations devoted to scientific computing. Thus, a solution that uses GPU can be used to address the computation of matrix exponential of large scale problems and in high performance workstations with a GPU attached. Moreover, also small personal computers or even laptops featuring a GPU can benefit from a solution that uses a GPU to accelerate computations.

We have developed a solution that improves the performance of the computation by using our GPU through the MATLAB Parallel Computing Toolbox (PCT). The MATLAB PCT provides a straightforward way to speed up MATLAB code by driving some computations to an NVIDIA GPU. To this end, the original MATLAB code is slightly modified with the PCT command extensions to MATLAB that allow to upload/download matrices to/from the GPU. The PCT point of view is straightforward, all operations carried out on objects (matrices) uploaded to the GPU are carried out into the GPU. The time needed for data racking between Host and GPU is the downside of this solution, so care must be taken of minimizing the total amount of matrices traveling to the GPU for computation.

The experimental results have been carried out on an NVIDIA K20 (Kepler generation card) [18] (routine `expmsplc_gpu`). The results, in execution time, obtained with the GPU are compared with the MATLAB code (routine `expmsplc`) running on an Intel QuadCore i7-3820 @3.6 GHz CPU, where the GPU is attached (Table 5). For the tests, we used several random matrices (MATLAB `randn` matrices [14]) with sizes $N = 1024, \ldots, 7168$.

A downside of using GPUs to accelerate general purpose computations is the memory limitation, 6GB in the case of our GPU, which means that we can not get the matrix exponential of matrices larger than $N = 7168$. This problem can be solved by using out-of-core algorithms [19], however, a solution based on this type of algorithms can not be implemented in the environment of the PCT, it would need C programming through MEX files [20]

Table 5: Time (in seconds) of the MATLAB routine `expmsplc` using the CPU and the MATLAB implementation based on the PCT using the GPU (`expmsplc_gpu`). The table also shows the speed up.

Size	Speedup	Time CPU	Time GPU
1024	2.43	0.46	0.19
2048	3.58	2.85	0.79
4096	4.43	21.88	4.93
5120	4.27	40.03	9.36
6144	4.31	67.78	15.72
7168	4.34	106.65	24.53

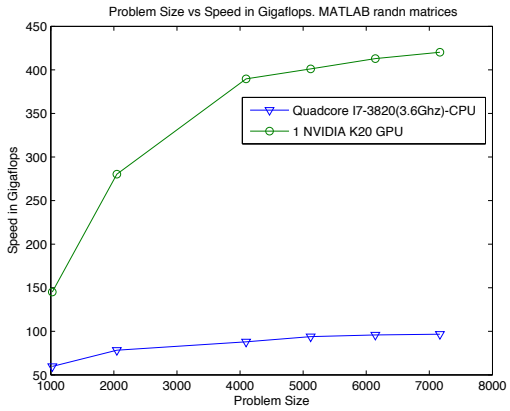


Figure 4: Performance in Gflops of the CPU version (`expmsplc`) and the GPU version (`expmsplc_gpu`) with regard to the problem size.

and this issue falls beyond the scope of this paper. Table 5 shows, in the second column, the speedup of the GPU version with regard to the CPU one. The performance in Gflops of both versions of the routine is shown in Figure 4.

6. Conclusions

In this work an accurate Taylor algorithm has been proposed to compute the exponential matrix. The algorithm uses the scaling technique based on the double angle formula, the Horner and Paterson-Stockmeyer's method for computing the Taylor approximation, and a spline method. A MATLAB

implementation of this (`expmspl`) has been compared with other state of the art MATLAB implementations.

Numerical experiments show that in general the new algorithm has a higher accuracy than `expm_new` and `exptayns` functions in the majority of the tests, with a lower execution time than the implementation based on Padé series (`expm_new`) and similar execution times that the other Taylor implementation (`exptayns`).

With our parallel implementation, we get good results. This implementation is useful when working with large scale problems. We used MATLAB and the PCT environment. The advantages of the PCT are that it is very versatile and it is relatively easy to configure and to use for non experts in parallel engineering. The downside is that probably the benefits are not as good as those that could be achieved with a CUDA+OPENMP+MEX based implementation (due to two K20 cards are used). For example, in [21, 22] the program yields 650 Gigaflops. While here, barely, we reached 450 Gflops. Anyway the GPU implementation (`expmspl_gpu`) is much faster than the implementation `expmspl`.

Acknowledgments

We would like to thank the anonymous referees for helpful comments. A special gratitude we give to Nuria Portillo Poblador (Universitat Politècnica de València) for her valuable collaboration.

References

[1] P. Bader, S. Blanes, M. Seydaoglu, The scaling, splitting, and squaring method for the exponential of perturbed matrices, SIAM Journal on Matrix Analysis and Applications 36 (2) (2015) 594–614.

[2] C. B. Moler, C. V. Loan, Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later*, SIAM Rev. 45 (2003) 3–49.

[3] N. J. Higham, Functions of Matrices: Theory and Computation, SIAM, Philadelphia, PA, USA, 2008.

[4] 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.

[5] J. Sastre, J. Ibáñez, E. Defez, P. Ruiz, New scaling-squaring taylor algorithms for computing the matrix exponential, SIAM Journal on Scientific Computing 37 (1) (2015) A439–A455.

[6] A. H. Al-Mohy, N. J. Higham, A new scaling and squaring algorithm for the matrix exponential, SIAM J. Matrix Anal. Appl. 31 (3) (2009) 970–989.

[7] E. Defez, M. Tung, J. J. Ibáñez, J. Sastre, Approximating and computing nonlinear matrix differential models, Mathematical and Computer Modelling 55 (7).

[8] M. S. Paterson, L. J. Stockmeyer, On the number of nonscalar multiplications necessary to evaluate polynomials, SIAM J. Comput. 2 (1) (1973) 60–66.

[9] G. H. Golub, C. V. Loan, Matrix Computations, 3rd Edition, Johns Hopkins Studies in Mathematical Sciences, The Johns Hopkins University Press, 1996.

[10] J. Sastre, J. J. Ibáñez, E. Defez, P. A. Ruiz, Efficient orthogonal matrix polynomial based method for computing matrix exponential, Appl. Math. Comput. 217 (2011) 6451–6463.

[11] N. J. Higham, Functions of Matrices: Theory and Computation, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008.

[12] P. Ruiz, J. Sastre, J. Ibáñez, E. Defez, High perfomance computing of the matrix exponential, J. Comput. Appl. Math. 291 (2016) 370–379.

[13] N. J. Higham, F. Tisseur, A block algorithm for matrix 1-norm estimation, with an application to 1-norm pseudospectra, SIAM J. Matrix Anal. Appl. 21 (2000) 1185–1201.

[14] N. J. Higham, The Test Matrix Toolbox for MATLAB, Numerical Analysis Report No. 237, Manchester, England (Dec. 1993).

[15] N. J. Higham, The scaling and squaring method for the matrix exponential revisited, SIAM J. Matrix Anal. Appl. 26 (4) (2005) 1179–1193.

[16] E. D. Dolan, J. J. Moré, Benchmarking optimization software with performance profiles, Math. Programming 91 (2002) 201–213.

[17] NVIDIA, CUDA. CUBLAS library (2009).

[18] NVIDIA, NVIDIA Tesla K20-K20X GPU Accelerators - Benchmarks (2012).

[19] E. D'Azevedo, A. Huang, K. Wong, W. Wu, Out-of-core algorithms for dense matrix factorization on gpgpu.

[20] Mathworks, MATLAB MEX Files, http://www.mathworks.com/support/technotes/1600/1605.shtml#intro.

[21] P. Alonso, J. Ibáñez, J. Sastre, J. Peinado, E. Defez, Efficient and accurate algorithms for computing matrix trigonometric functions, J. Comput. Appl. Math. 309 (1) (2017) 325–332.

[22] E. Defez, J. Sastre, J. Ibáñez, J. Peinado, Solving engineering models using hyperbolic matrix functions, Applied Mathematical Modelling 40 (4) (2016) 2837–2844.