






Article

Two Taylor Algorithms for Computing the Action of the Matrix Exponential on a Vector

Javier Ibáñez ¹, José M. Alonso ², Pedro Alonso-Jordá ^{3,*}, Emilio Defez ¹ and Jorge Sastre ⁴

¹ Instituto de Matemática Multidisciplinar, Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, Spain; jjibanez@dsic.upv.es (J.I.); edefez@imm.upv.es (E.D.)

² Instituto de Instrumentación para Imagen Molecular, Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, Spain; jmalonso@dsic.upv.es

³ Departamento de Sistemas Informáticos y Computación, Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, Spain; palonso@upv.es

⁴ Instituto de Telecomunicaciones y Aplicaciones Multimedia, Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, Spain; jsastrem@upv.es

* Correspondence: palonso@upv.es

Abstract: The action of the matrix exponential on a vector $e^{At}v$, $A \in \mathbb{C}^{n \times n}$, $v \in \mathbb{C}^n$, appears in problems that arise in mathematics, physics, and engineering, such as the solution of systems of linear ordinary differential equations with constant coefficients. Nowadays, several state-of-the-art approximations are available for estimating this type of action. In this work, two Taylor algorithms are proposed for computing $e^A v$, which make use of the scaling and recovering technique based on a backward or forward error analysis. A battery of highly heterogeneous test matrices has been used in the different experiments performed to compare the numerical and computational properties of these algorithms, implemented in the MATLAB language. In general, both of them improve on those already existing in the literature, in terms of accuracy and response time. Moreover, a high-performance computing version that is able to take advantage of the computational power of a GPU platform has been developed, making it possible to tackle high dimension problems at an execution time significantly reduced.

Keywords: action of the matrix exponential; Taylor series; GPU computing



Citation: Ibáñez, J.; Alonso, J.M.; Alonso-Jordá, P.; Defez, E.; Sastre, J. Two Taylor Algorithms for Computing the Action of the Matrix Exponential on a Vector. *Algorithms* **2022**, *15*, 48. <https://doi.org/10.3390/a15020048>

Academic Editor: Frank Werner

Received: 23 December 2021

Accepted: 26 January 2022

Published: 28 January 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

The study of different matrix functions $f(A)$, where $A \in \mathbb{C}^{n \times n}$, such as the exponential, the trigonometric and hyperbolic functions, the logarithm or the sign, and several families of orthogonal matrix polynomials, among which are those of Hermite, Laguerre, Jacobi, or Chebyshev, form an attractive, wide, and active field of research due to their numerous applications in different areas of science and technology.

In the last years, the matrix exponential e^A has become a constant focus of attention due to its extensive applications—from the classical theory of differential equations for computing the solution of the matrix system $Y'(t) = AY(t)$, given by $Y(t) = e^{At}$, to the graph theory [1–3], even including some recent progress about the numerical solutions of fractional partial differential equations [4,5]—as well as the multiple difficulties involved in its effective computation. They have motivated the development of distinct numerical methods, some of them classic and very well-known, as described in [6], and other more recent and novel using, for example, Bernoulli matrix polynomials [7].

Nevertheless, sometimes it is not the computation of the function $f(A)$ on a square matrix A which is required by applications, but its action on a given vector $v \in \mathbb{C}^n$, i.e., $f(A)v$. Once again, the motivation for this particular problem may come from its applicability in different and varied branches of science and engineering. As an example, the action of the matrix sign on a vector is used in quantum chromodynamics (QCD), see [8] for details. Furthermore, in particular, the action of the matrix exponential operator on a vector

appears in multiple problems arising in areas of mathematics, as in the case of the following first-order matrix differential equation with initial conditions and constant coefficients

$$\left. \begin{aligned} Y'(t) &= AY(t) \\ Y(0) &= v \end{aligned} \right\}'$$

being $A \in \mathbb{C}^{n \times n}$ and $v \in \mathbb{C}^n$, and whose solution is in the form $Y(t) = e^{At}v \in \mathbb{C}^n$ —this kind of problem occurs frequently, for example, in control theory—or in applications involving the numerical solutions of fractional partial differential equations [9]. Moreover, the action of the matrix exponential on a vector is also used in physics and engineering fields such as electromagnetics [10], circuit theory [11], acoustic/elastic wave equations [12], seismic wave propagation [13], chemical engineering [14], robotics [15], and so on.

There exist different methods in the literature to calculate the action of the exponential matrix on a vector (see for example those described in references [16–21]). Additionally, a comparison of recent software can be found in [22]. Among these methods, those based on Krylov subspaces [23]—they reduce the $e^A v$ problem to a corresponding one for a small matrix using a projection technique—and those based on polynomial or Padé approximations, see [16,17,24] and references therein, can be highlighted.

When computing $e^A v$, the approach using Taylor method, in combination with the scaling and squaring technique, consists of determining the order m of the Taylor polynomial $T_m(A)$ and a positive integer s , called the scaling factor, so that $e^A v \approx (T_m(2^{-s}A))^{2^s} v$. Indeed, in this paper, two algorithms that calculate $e^A v$ without explicitly computing e^A have been designed and implemented in MATLAB language. Both of them are based on truncating and computing the Taylor series of the matrix exponential, after having obtained the values of m and s by means of a backward or forward error analysis.

Throughout this work, we will denote the matrix identity of order n as I_n or I . With $\lceil x \rceil$, we will represent the result of rounding x to the nearest integer greater than or equal to x . In the same way, $\lfloor x \rfloor$ will stand for the result of rounding x to the nearest integer less than or equal to x . The matrix norm $\|A\|$ will refer to any subordinate matrix norm and $\|A\|_1$ will be, in particular, the 1-norm. A polynomial of degree m is given by an expression in the form $P_m(x) = a_m x^m + a_{m-1} x^{m-1} + \dots + a_1 x + a_0$, where x is a real or complex variable, and the coefficients $a_j, 0 \leq j \leq m$, are complex numbers with $a_m \neq 0$. Moreover, we can define the matrix polynomial $P_m(A)$, for $A \in \mathbb{C}^{n \times n}$, by means of the formulation $P_m(A) = a_m A^m + a_{m-1} A^{m-1} + \dots + a_1 A + a_0 I$.

This work is organised as follows. First, Section 2 presents two scaling and squaring Taylor algorithms for computing the action of the matrix exponential on a vector. Then, in Section 3, these algorithms are implemented and their numerical and computational properties are compared with those of other state-of-the-art codes by means of different experiments. Next, Section 4 exposes the computational performance of our two codes after their implementation on a GPU-based execution platform. Finally, conclusions are given in the last section.

2. Algorithms for Computing the Action of the Matrix Exponential

Let

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

be the Taylor approximation of order m devoted to the exponential of matrix $A \in \mathbb{C}^{n \times n}$, and let $v \in \mathbb{C}^n$ be a vector. In combination with the scaling and squaring method, the Taylor-based approach is concerned with computing $e^A = (e^{2^{-s}A})^{2^s} \approx (T_m(2^{-s}A))^{2^s}$ [6], where the nonnegative integers m and s are chosen to achieve full machine accuracy at a minimum computational cost.

In our proposal, the values of m and s are calculated such that the absolute forward error for computing $e^{s^{-1}A}v$ is less or equal to $u = 2^{-53}$, the so-called unit roundoff in IEEE double precision arithmetic. The absolute forward error of $e^{s^{-1}A}v$ is bounded as follows:

$$E_{af}(A, v) = \left\| \sum_{k=m+1}^{\infty} \frac{A^k v}{s^k k!} \right\| \approx \left\| \frac{A^{m+1} v}{s^{m+1}(m+1)!} + \frac{A^{m+2} v}{s^{m+2}(m+2)!} \right\|. \tag{2}$$

Hence, if we calculate s such that

$$\left\| \frac{A^{m+1} v}{s^{m+1}(m+1)!} \right\| \leq u,$$

i.e.,

$$s = \left\lceil \sqrt[m+1]{\frac{\|A^{m+1} v\|}{(m+1)!u}} \right\rceil, \tag{3}$$

and we verify that the following inequality is as well satisfied

$$\left\| \frac{A^{m+1} v}{s^{m+1}(m+1)!} + \frac{A^{m+2} v}{s^{m+2}(m+2)!} \right\| \leq u, \tag{4}$$

then, the absolute forward error of computing $e^{s^{-1}A}v$ will be approximately less or equal to u . Once m and s have been calculated, $e^A v$ can be efficiently computed as follows:

$$\begin{aligned} e^A v &= \left(e^{A/s} \right)^s v = \overbrace{e^{A/s} e^{A/s} \dots e^{A/s} e^{A/s}}^{s \text{ times}} v \\ &\cong \overbrace{\sum_{k=0}^m \frac{A^k}{s^k k!} \sum_{k=0}^m \frac{A^k}{s^k k!} \dots \sum_{k=0}^m \frac{A^k}{s^k k!} \sum_{k=0}^m \frac{A^k}{s^k k!}}^{s \text{ times}} v \\ &= \overbrace{\sum_{k=0}^m \frac{A^k}{s^k k!} \sum_{k=0}^m \frac{A^k}{s^k k!} \dots \sum_{k=0}^m \frac{A^k}{s^k k!}}^{s-1 \text{ times}} w_1 \\ &= \overbrace{\sum_{k=0}^m \frac{A^k}{s^k k!} \sum_{k=0}^m \frac{A^k}{s^k k!} \dots \sum_{k=0}^m \frac{A^k}{s^k k!}}^{s-2 \text{ times}} w_2 \\ &= \dots \\ &= \overbrace{\sum_{k=0}^m \frac{A^k}{s^k k!}}^{1 \text{ time}} w_{s-1} = w_s, \end{aligned} \tag{5}$$

where $w_i = \sum_{k=0}^m \frac{A^k w_{i-1}}{s^k k!}$, $i = 1 : s$, with $w_0 = v$.

In this way, Algorithm 1 computes $w = e^A v$, where $A \in \mathbb{C}^{n \times n}$ and $v \in \mathbb{C}^n$, starting with an initial value $m \in \mathbb{N}$, without explicitly working out e^A . First, in lines 1–4, vectors $Av, A^2v, \dots, A^m v, A^{m+1} v, A^{m+2} v$ are computed and stored in the array of vectors V_1, V_2, \dots, V_{m+2} , respectively. Then, lines 5–13 are used to determine the minimum value m and the corresponding value s , calculated in line 7, taking into account expression (3), such that (4) is fulfilled. Next, in lines 15–17, m is set to the maximum value allowed if expression (3) could not be satisfied. Finally, in lines 18–29, $w = e^A v$ is computed according to (5).

Algorithm 1 Given a matrix $A \in \mathbb{C}^{n \times n}$, a vector $v \in \mathbb{C}^n$, and minimum m and maximum M Taylor polynomial orders, $m, M \in \mathbb{N}$, this algorithm computes $w = e^A v \in \mathbb{C}^n$ by (5).

```

1:  $V_1 = Av$ 
2: for  $k = 2 : m + 2$  do
3:    $V_k = AV_{k-1}$ 
4: end for
5:  $f = 0$ 
6: while  $f == 0$  and  $m \leq M$  do
7:    $s = \left\lceil \sqrt[m+1]{\frac{\|V_{m+1}\|}{(m+1)!u}} \right\rceil$ 
8:   if  $\left\| \frac{V_{m+1}}{s^{m+1}(m+1)!} + \frac{V_{m+2}}{s^{m+2}(m+2)!} \right\| \leq u$  then
9:      $f = 1$ 
10:  else
11:     $m = m + 1$ 
12:     $V_{m+2} = AV_{m+1}$ 
13:  end if
14: end while
15: if  $f == 0$  then
16:    $m = M$ 
17: end if
18:  $w = v$ 
19: for  $k = 1 : m$  do
20:    $w = w + V_k / (s^k k!)$ 
21: end for
22:  $A = A/s$ 
23: for  $i = 2 : s$  do
24:    $v = w$ 
25:   for  $k = 1 : m$  do
26:     $v = Av$ 
27:     $w = w + v/k!$ 
28:   end for
29: end for

```

On the contrary, if an absolute backward error analysis for computing $e^{s^{-1}A}v$ were considered, see (10), (11), (13)–(15), and (22) from [25], then

$$\begin{aligned}
 E_{ab}(A, v) &= \left\| \sum_{k \geq 0} \frac{(-1)^k A^{m+1+k} v}{s^{m+1+k} k! m! (m+1+k)} \right\| \\
 &\approx \left\| \frac{A^{m+1} v}{s^{m+1} (m+1)!} - \frac{A^{m+2} v}{s^{m+2} m! (m+2)} \right\|.
 \end{aligned} \tag{6}$$

As in both cases in which the first term of these absolute errors coincides, Algorithm 1 could be easily modified for computing $e^{s^{-1}A}v$ such that $E_{ab} \leq u$. To do this, only the condition expression in line 8, which checks whether convergence is reached, should be replaced by $\left\| \frac{V_{m+1}}{s^{m+1}(m+1)!} - \frac{V_{m+2}}{s^{m+2}(m+2)!} \right\| \leq u$.

An alternative formulation to expression (2) in the absolute backward error estimation would have been to consider only the first of the terms. In this way, starting from a minimum value of m , the value of s is computed from the expression (3), which will already ensure that the error committed will be less or equal to u . With that assumption, together with the objective of reducing the matrix–vector products of the previous algorithm, we designed Algorithm 2, which determines the appropriate values of m and s that satisfy both purposes.

Algorithm 2 Given a matrix $A \in \mathbb{C}^{n \times n}$, a vector $v \in \mathbb{C}^n$, and minimum m and maximum M Taylor polynomial orders, $m, M \in \mathbb{N}$, this algorithm computes $w = e^A v \in \mathbb{C}^n$ by (5).

```

1:  $V_1 = Av$ 
2: for  $k = 2 : m + 1$  do
3:    $V_k = AV_{k-1}$ 
4: end for
5:  $s = \left\lceil \sqrt[m+1]{\frac{\|V_{m+1}\|}{(m+1)!u}} \right\rceil$ 
6:  $p = ms$ 
7:  $f = 0$ 
8: while  $f = 0$  and  $m < M$  do
9:    $m = m + 1$ 
10:   $V_{m+1} = AV_m$ 
11:   $s_1 = \left\lceil \sqrt[m+1]{\frac{\|V_{m+1}\|}{(m+1)!u}} \right\rceil$ 
12:   $p_1 = ms_1$ 
13:  if  $p_1 \leq p$  then
14:     $p = p_1$ 
15:     $s = s_1$ 
16:  else
17:     $m = m - 1$ 
18:     $f = 1$ 
19:  end if
20: end while
21:  $w = v$ 
22: for  $k = 1 : m$  do
23:    $w = w + V_k / (s^k k!)$ 
24: end for
25:  $A = A/s$ 
26: for  $i = 2 : s$  do
27:    $v = w$ 
28:   for  $k = 1 : m$  do
29:     $v = Av$ 
30:     $w = w + v/k!$ 
31:   end for
32: end for

```

For it, in lines 5–20, the number of matrix–vector products p required, obtained as $m \times s$, employing two consecutive values of m and their corresponding values of s , is compared in each iteration of the while loop. The procedure concludes when the number of products associated with a given degree m of the Taylor polynomial is greater than that obtained with the immediately preceding degree.

Table 1 shows the computational and storage costs of Algorithms 1 and 2. The computational cost depends on the parameters m and s and it is specified in terms of the required number of matrix–vector products. The storage costs is expressed as the number of matrices and vectors with which the algorithms work.

Table 1. Computational and storage costs for Algorithms 1 and 2.

	Computational Cost	Matrices to Be Stored	Vectors to Be Stored
Algorithm 1	$ms + 2$	1	$m + 4$
Algorithm 2 ($m < M$)	$ms + 2$	1	$m + 4$
Algorithm 2 ($m = M$)	$ms + 1$	1	$m + 3$

3. Numerical Experiments

In this section, several tests have been carried out to illustrate the accuracy and efficiency of two MATLAB codes based on Algorithms 1 and 2. All these experiments have been executed on Microsoft Windows 10 × 64 PC system with an Intel Core i7 CPU Q720 @1.60 Ghz processor and 6 GB of RAM, using MATLAB R2020b. The following MATLAB codes have been compared among them:

- `expmvtay1`: Implements the Algorithm 1, where absolute backward errors are assumed. The degree m of the Taylor polynomial used in the approximation will vary from 40 to 60. The maximum value allowed for the scaling parameter s , so as not to give rise to an excessively high number of matrix–vector products, will be equal to 45. The code is available at <http://personales.upv.es/joalab/software/expmvtay1.m> (accessed on 22 December 2021).
- `expmvtay2`: Based on Algorithm 2. As mentioned before, it attempts to reduce the number of matrix–vector products required by the code `expmvtay1`. It considers the same limits of m and s as the previous function. The implementation can be downloaded from <http://personales.upv.es/joalab/software/expmvtay2.m> (accessed on 22 December 2021).
- `expmAv`: Code where e^A is firstly expressly computed, by using the function `exptaynsv3` described in [26], and the matrix–vector product $e^A v$ is then carried out. The code `exptaynsv3` is based on a Taylor polynomial approximation to the matrix exponential function in combination with the scaling and squaring technique. The order m of the approximation polynomial will take values no greater than 30.
- `expmv`: This function, implemented by Al-mohy and Higham [17], computes $e^{tA} v$ without explicitly forming e^{tA} . It uses the scaling part of the scaling and squaring method together with a truncated Taylor series approximation to the matrix exponential.
- `expm_newAv`: Code that first explicitly calculates e^A by means of the function `expm_new`, developed by Al-Mohy and Higham [27], and then multiplies it by the vector v to form $e^A v$. The function `expm_new` is based on Padé approximants and it implements an improved scaling and squaring algorithm.
- `explaja`: This code, based on the Leja interpolation method, computes the action of the matrix exponential of $H \times A$ on a vector (or a matrix) v [28]. The result is similar to $e^{H \times A} v$, but the matrix exponential is not explicitly worked out. In our experiments, H will be equal to 1 and default values of the tolerance will be provided.
- `expv`: Implementation of Sidje [23] that calculates $e^{tA} v$ by using Krylov subspace projection techniques with a fixed dimension for the corresponding subspace. It does not compute the matrix exponential in isolation but instead, it calculates directly the action of the exponential operator on the operand vector. The matrix under consideration interacts only via matrix–vector products (matrix-free method).

To evaluate the performance of the codes described above in accuracy and speed, a test battery composed of the following set of matrices has been used. For each matrix A , a distinct vector v with random values in the interval $[-0.5, 0.5]$ has been generated as well. MATLAB Symbolic Math Toolbox with 256 digits of precision was employed in all the computations to provide the “exact” action of the matrix exponential of A on a vector v , thanks to the `vpa` (variable-precision floating-point arithmetic) function:

- (a) Set 1: One hundred diagonalizable 128×128 complex matrices with the form $A = V \times D \times V^T$, where D is a diagonal matrix with complex eigenvalues and V is an orthogonal matrix obtained as $V = H / \sqrt{128}$, being H a Hadamard matrix. The 2-norm of these randomly generated matrices varied from 0.1 to 339.4. The “exact” action of the matrix exponential of A on a vector v was calculated computing first $e^A = V \times e^D \times V^T$ (see [29], p. 10) and then $e^A v$.
- (b) Set 2: One hundred non-diagonalizable 128×128 complex matrices generated as $A = V \times J \times V^T$, where J is a Jordan matrix composed of complex eigenvalues with an algebraic multiplicity randomly varying between 1 and 3. V is an orthogonal matrix

whose randomly obtained elements become progressively larger from one matrix to the next. The 2-norm of these matrices reached values from 3.76 to 323.59. The “exact” action of the matrix exponential on a vector was calculated as for the above set of matrices.

- (c) Set 3: Fifty matrices from the Matrix Computation Toolbox (MCT) [30] and twenty matrices from the Eigtool MATLAB Package (EMP) [31], all of them with a 128×128 size. With the aim of calculating the “exact” action of the matrix exponential on a vector, the “exact” exponential function e^A of each matrix A was initially computed according to the next algorithm consisting of the following three steps, employing the function `vpa` in each of them:
1. Diagonalise the matrix A via the MATLAB function `eig` and obtain the matrices V and D such that $A = V \times D \times V^{-1}$. Then, the matrix E_1 will be computed as $E_1 = V \times e^D \times V^{-1}$.
 2. Calculate the matrix exponential of A by means of the MATLAB function `expm`, i.e., $E_2 = \text{expm}(A)$.
 3. Take into account the “exact” matrix exponential of A only if:

$$\frac{\|E_1 - E_2\|}{\|E_1\|} \leq u.$$

Lastly, the “exact” action was worked out as $E_1 v$, obviously again using the function `vpa`.

Among the seventy-two matrices that initially constitute this third set, only forty-two of them (thirty-five from the MCT and seven from the EMP) could be satisfactorily processed in the numerical tests carried out. The 2-norm of these considered matrices ranged between 1 and 10,716. The reasons for the exclusion of the others are given below:

- The “exact” exponential function for matrices 4, 5, 10, 16, 17, 18, 21, 25, 26, 35, 40, 42, 43, 44, and 49 from the MCT and matrices 4, 6, 7, and 9 from the EMP could not be computed in accordance with the 3-step procedure previously described.
- Matrices 2 and 15 incorporated in the MCT and matrices 1, 3, 5, 10, and 15 belonging to the EMP incurred in a very high relative error by some code, due to the ill-conditioning of these matrices.
- Matrices 8, 11, 13, and 16 from the EMP were repeated, as they were also part of the MCT.

Figures 1–3 show, respectively, the results of the numerical analyses carried out by means of each of the different codes in comparison with the three types of matrices considered. In more detail, these figures depict the normwise relative errors (a), the performance profiles (b), the ratios of normwise relative errors between `expmvtay1` and the rest of the implementations (c), the lowest and highest relative error rates (d), the polynomial orders and the Krylov subspace dimensions (e), the ratios of matrix–vector products between `expmvtay2` and the other codes (f), the response time (g), and the ratio of the execution time between `expmvtay2` and the remaining functions (h).

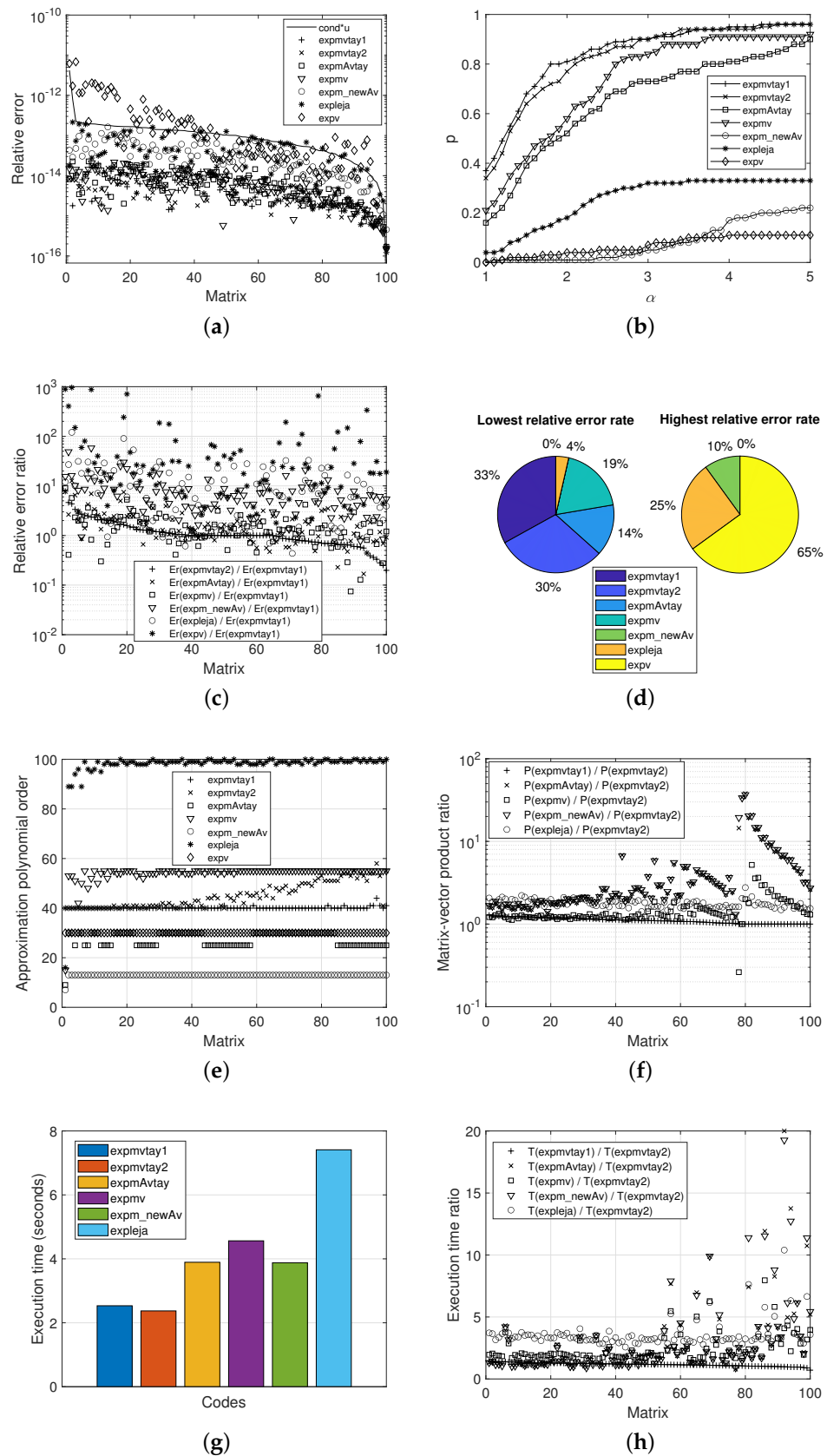


Figure 1. Experimental results for Set 1. (a) Normwise relative errors. (b) Performance profile. (c) Ratio of relative errors. (d) Lowest and highest relative error rate. (e) Polynomial, interpolation and subspace orders. (f) Ratio of matrix–vector products. (g) Execution time. (h) Ratio of execution time.

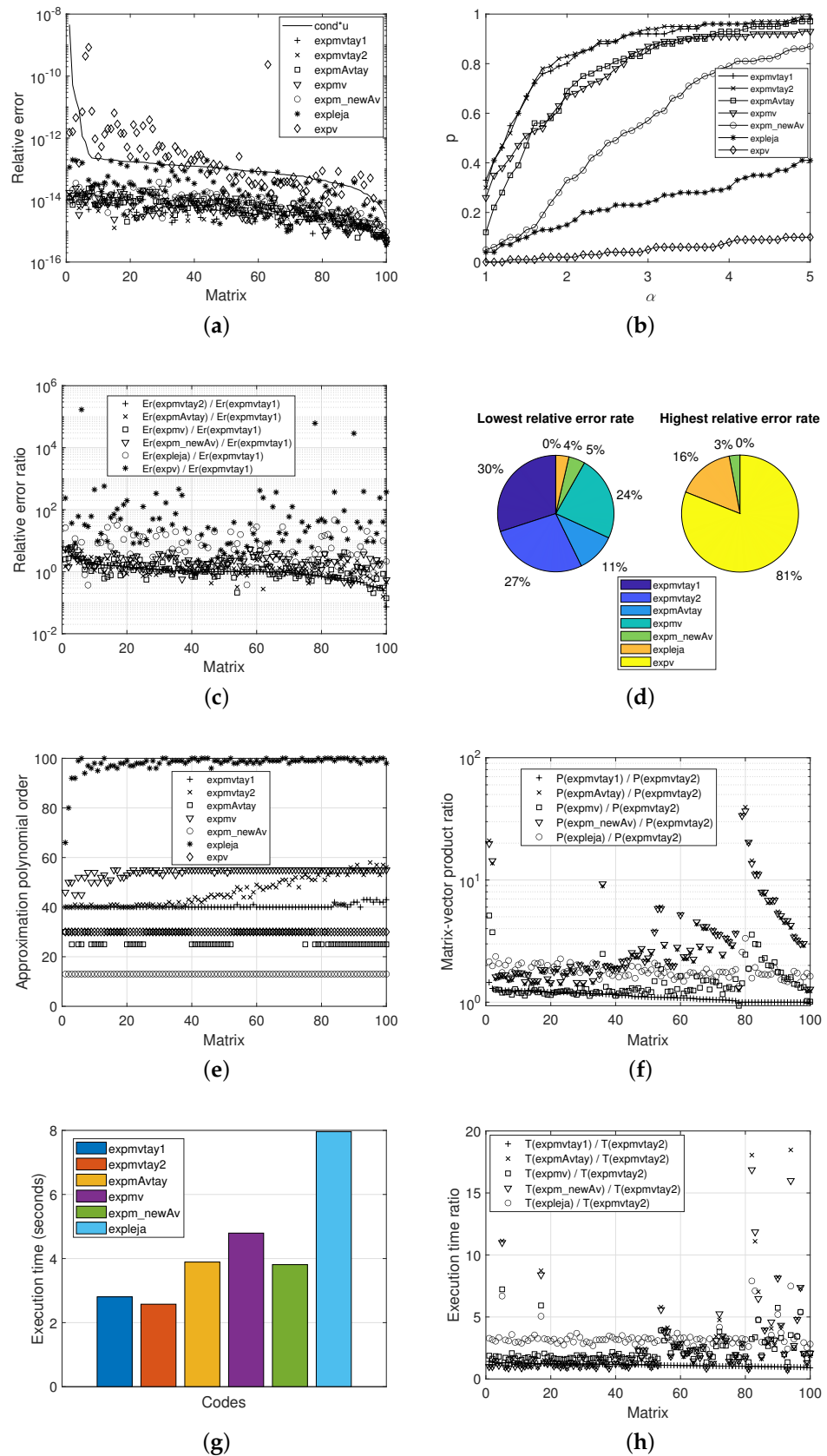


Figure 2. Experimental results for Set 2. (a) Normwise relative errors. (b) Performance profile. (c) Ratio of relative errors. (d) Lowest and highest relative error rate. (e) Polynomial, interpolation and subspace orders. (f) Ratio of matrix–vector products. (g) Execution time. (h) Ratio of execution time.

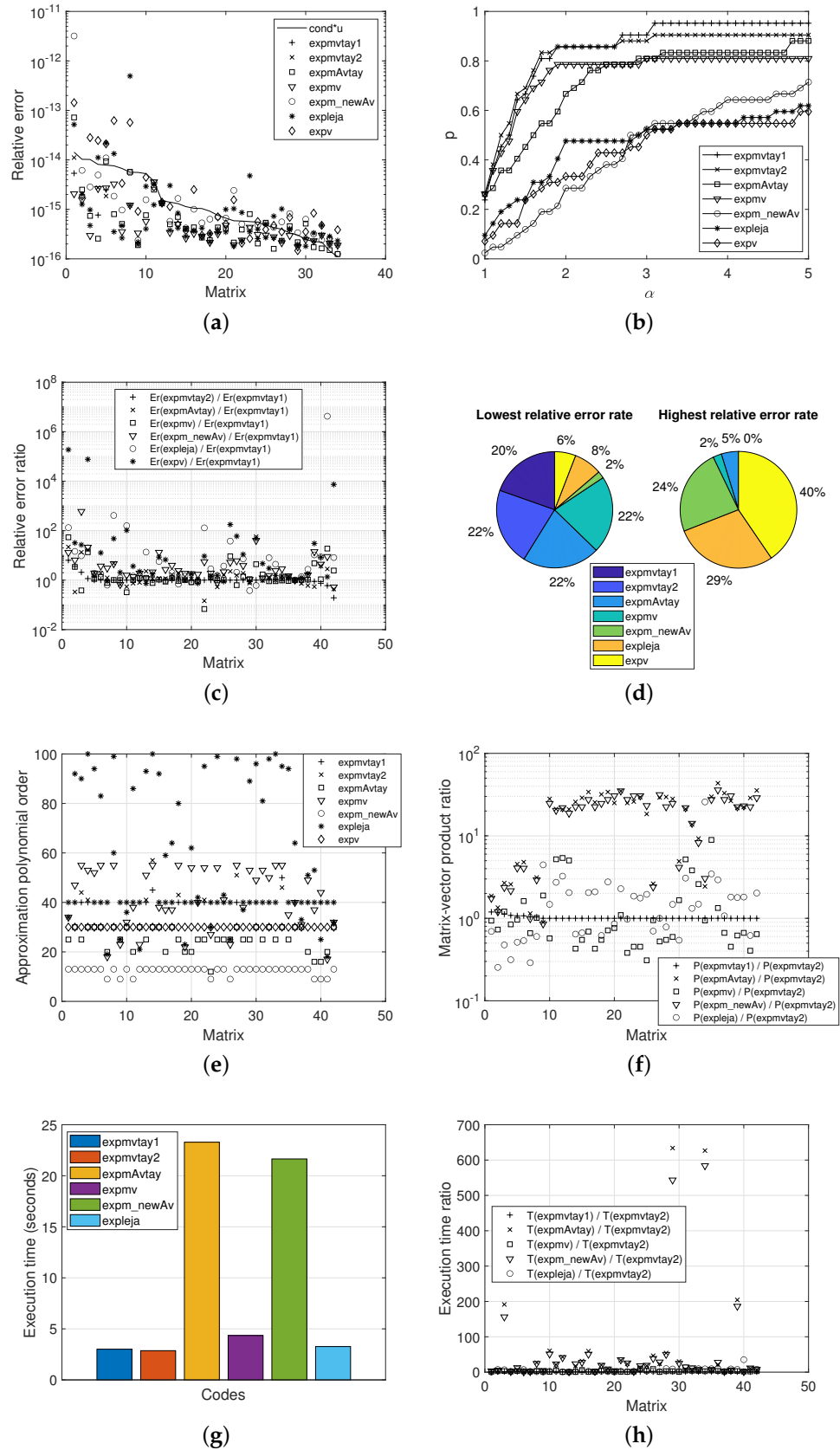


Figure 3. Experimental results for Set 3. (a) Normwise relative errors. (b) Performance profile. (c) Ratio of relative errors. (d) Lowest and highest relative error rate. (e) Polynomial, interpolation and subspace orders. (f) Ratio of matrix–vector products. (g) Execution time. (h) Ratio of execution time.

For each of the methods under evaluation, the normwise relative error $Er(A, v)$ committed in the computation of the action of the exponential function of a matrix A on a vector v was calculated as follows:

$$Er(A, v) = \frac{\|exp(A, v) - \widetilde{exp}(A, v)\|_2}{\|exp(A, v)\|_2},$$

where $exp(A)$ denotes the exact solution and $\widetilde{exp}(A, v)$ represents the approximate one.

Figures 1a–3a present the normwise relative error incurred by each of the seven codes under study. The solid line that appears in them plots the function $k_{exp}u$, where k_{exp} (or *cond*) means the condition number of the matrix exponential function ([29], Chapter 3) for each matrix and u represents the unit roundoff in IEEE double precision arithmetic. The matrices were ordered by decreasing value their condition number. It is well-known that the numerical stability of each method is exposed if its relative errors are positioned not far above this solid line, although it is always preferable that they are below. Consequently, these figures reveal that `expv` is the code that most frequently presents relative errors above the $k_{exp}u$ curve, being therefore the least numerically stable of all the codes analysed for the matrices used in the numerical tests. The rest of the codes can be said to offer high numerical stability.

The percentage of cases in which `expmvtay1` incurred in a normwise relative error less, equal or greater than the other codes is listed in Table 2. As can be appreciated, `expmvtay1` always provided a higher percentage of improvement cases than the rest of its competitors, especially over `expv` followed by `expm_newAv`, `expleja`, and, with very similar rates, by `expmAvtay` and `expmv`. On the other hand, the gain in accuracy by `expmvtay1` over `expmvtay2` is not noticeable, and it can be concluded that the latter will also offer a notable improvement in the reliability of the results compared to the rest of the methods.

Table 2. Normwise relative error comparison, for the three sets, between `expmvtay1` and all other codes.

	Set 1	Set 2	Set 3
$Er(\text{expmvtay1}) < Er(\text{expmvtay2})$	39%	39%	9.52%
$Er(\text{expmvtay1}) > Er(\text{expmvtay2})$	36%	38%	9.52%
$Er(\text{expmvtay1}) = Er(\text{expmvtay2})$	25%	23%	80.95%
$Er(\text{expmvtay1}) < Er(\text{expmAvtay})$	69%	65%	61.90%
$Er(\text{expmvtay1}) > Er(\text{expmAvtay})$	31%	35%	38.10%
$Er(\text{expmvtay1}) = Er(\text{expmAvtay})$	0%	0%	0%
$Er(\text{expmvtay1}) < Er(\text{expmv})$	69%	58%	61.90%
$Er(\text{expmvtay1}) > Er(\text{expmv})$	31%	42%	38.10%
$Er(\text{expmvtay1}) = Er(\text{expmv})$	0%	0%	0%
$Er(\text{expmvtay1}) < Er(\text{expm_newAv})$	97%	89%	90.48%
$Er(\text{expmvtay1}) > Er(\text{expm_newAv})$	3%	11%	9.52%
$Er(\text{expmvtay1}) = Er(\text{expm_newAv})$	0%	0%	0%
$Er(\text{expmvtay1}) < Er(\text{expleja})$	91%	93%	80.95%
$Er(\text{expmvtay1}) > Er(\text{expleja})$	9%	7%	19.05%
$Er(\text{expmvtay1}) = Er(\text{expleja})$	0%	0%	0%
$Er(\text{expmvtay1}) < Er(\text{expv})$	100%	98%	88.10%
$Er(\text{expmvtay1}) > Er(\text{expv})$	0%	2%	11.90%
$Er(\text{expmvtay1}) = Er(\text{expv})$	0%	0%	0%

In a very detailed way, Table 3 collects the values corresponding to the minimum and maximum normwise relative error committed by all the functions for the three sets of matrices employed, as well as the mean, median, and standard deviation. While the minimum relative errors incurred by the codes are very similar, it is easy to observe how the maximum relative error and, consequently, the largest values in the mean, median,

and standard deviation corresponded in general to `expv`, which turned out to be the least reliable code, closely followed by `expleja` and `expm_newAv`. For all these metrics, the other methods, all based on the Taylor approximation, provided better values, analogous to each other.

Table 3. Minimum, maximum, mean, median, and standard deviation values of the relative errors committed by all the codes for Sets 1, 2, and 3, respectively.

	Min.	Max.	Mean	Median	Std. Dev.
<code>expmvtay1</code>	1.23×10^{-16}	1.95×10^{-14}	5.22×10^{-15}	3.75×10^{-15}	4.42×10^{-15}
<code>expmvtay2</code>	1.23×10^{-16}	1.78×10^{-14}	5.46×10^{-15}	4.76×10^{-15}	4.35×10^{-15}
<code>expmAvtay</code>	1.70×10^{-16}	2.27×10^{-14}	7.41×10^{-15}	6.54×10^{-15}	5.46×10^{-15}
<code>expmv</code>	1.52×10^{-16}	2.62×10^{-14}	6.80×10^{-15}	4.98×10^{-15}	5.55×10^{-15}
<code>expm_newAv</code>	4.54×10^{-16}	1.63×10^{-13}	3.51×10^{-14}	2.68×10^{-14}	3.16×10^{-14}
<code>expleja</code>	1.58×10^{-16}	2.15×10^{-13}	4.37×10^{-14}	2.78×10^{-14}	4.69×10^{-14}
<code>expv</code>	1.55×10^{-16}	6.83×10^{-12}	4.09×10^{-13}	5.79×10^{-14}	1.01×10^{-12}
<code>expmvtay1</code>	5.34×10^{-16}	1.72×10^{-14}	5.55×10^{-15}	4.07×10^{-15}	4.40×10^{-15}
<code>expmvtay2</code>	5.34×10^{-16}	3.08×10^{-14}	5.49×10^{-15}	3.87×10^{-15}	4.89×10^{-15}
<code>expmAvtay</code>	5.15×10^{-16}	2.71×10^{-14}	6.45×10^{-15}	5.85×10^{-15}	4.87×10^{-15}
<code>expmv</code>	4.04×10^{-16}	2.45×10^{-14}	6.57×10^{-15}	5.05×10^{-15}	5.41×10^{-15}
<code>expm_newAv</code>	9.63×10^{-16}	3.79×10^{-14}	1.02×10^{-14}	7.91×10^{-15}	8.30×10^{-15}
<code>expleja</code>	3.68×10^{-16}	1.98×10^{-13}	4.22×10^{-14}	2.21×10^{-14}	5.03×10^{-14}
<code>expv</code>	6.13×10^{-16}	8.42×10^{-10}	1.58×10^{-11}	8.64×10^{-14}	9.70×10^{-11}
<code>expmvtay1</code>	1.81×10^{-16}	9.12×10^{-9}	2.17×10^{-10}	3.64×10^{-16}	1.41×10^{-9}
<code>expmvtay2</code>	1.82×10^{-16}	9.12×10^{-9}	2.17×10^{-10}	3.76×10^{-16}	1.41×10^{-9}
<code>expmAvtay</code>	1.25×10^{-16}	4.94×10^{-9}	1.18×10^{-10}	5.04×10^{-16}	7.62×10^{-10}
<code>expmv</code>	1.75×10^{-16}	2.94×10^{-9}	7.01×10^{-11}	3.47×10^{-16}	4.54×10^{-10}
<code>expm_newAv</code>	1.25×10^{-16}	1.44×10^{-8}	3.44×10^{-10}	9.88×10^{-16}	2.23×10^{-9}
<code>expleja</code>	1.80×10^{-16}	1.45×10^{-6}	3.50×10^{-8}	1.05×10^{-15}	2.23×10^{-7}
<code>expv</code>	1.47×10^{-16}	9.43×10^{-7}	2.25×10^{-8}	1.27×10^{-15}	1.46×10^{-7}

Figures 1b–3b, corresponding to the performance profiles, depict the percentage of matrices in each set, expressed in terms of one, for which the error committed by each method in comparison is less than or equal to the smallest relative error incurred by any of them multiplied by α . It is immediately noticeable that `expmvtay1` and `expmvtay2` achieved the highest probability values for the vast totality of the plots, `expmvtay1` showing a slightly highest accuracy than `expmvtay2` in Figure 1b and similar in the other figures. The scores achieved by `expmAvtay` and `expmv` do not differ much from each other, and they are somewhat lower than those provided by the previous codes. Clearly, `expm_newAv`, `expleja`, and `expv` exhibited the poorest results, with a significantly lower accuracy than the other codes.

These accuracy results are also confirmed by the next two types of illustrations. Figures 1c–3c reflect the ratio of the relative errors for any of the methods under study and `expmvtay1`. Values of these ratios are decreasingly ordered and exposed according to the quotient $Er(\text{expmvtay1})/Er(\text{expmvtay2})$. Most of these values are greater than or equal to 1, showing once again the overall superiority of `expmvtay1`, and correspondingly of `expmvtay2`, over the other functions.

As a pie chart, and for each of the sets, Figures 1d–3d show the percentage of matrices in which each method resulted in the lowest or highest relative error. According to the values therein, for Sets 1 and 2, `expmvtay1` and `expmvtay2` gave rise to the lowest errors on a highest percentage of occasions. Notwithstanding, for Set 3, these percentages were almost equally distributed among `expmvtay1`, `expmvtay2`, `expmAvtay`, and `expmv`. If our attention is now turned to the highest relative error rates, `expv` gave place to the worst results in most cases, leading to values equal to 65% and 81% for Sets 1 and 2, respectively.

For Set 3, this percentage dropped to 40%, followed in a 29% by `expleja` and in a 24% by `expm_newAv`.

Table 4 compares the minimum, maximum, mean, and median values of the tuple m and s , i.e., the order of the Taylor approximation polynomial and the value of the scaling parameter used by the first four methods. For the other codes, the parameter m is not comparable as it represents the degree of the Padé approximants to the matrix exponential (`expm_newAv`), the selected degree of interpolation (`expleja`) or the dimension of the Krylov subspace employed (`expmv`). Additionally, s denotes the scaling value (`expm_newAv`) or the scaling steps (`expleja`), but it is not provided for `expv`, because this code does not work with the scaling technique.

Table 4. Minimum, maximum, mean, and median parameters m and s employed for Sets 1, 2, and 3, respectively.

	m				s			
	Min.	Max.	Mean	Median	Min.	Max.	Mean	Median
<code>expmvtay1</code>	40	44	40.13	40.00	1	45	23.65	23.00
<code>expmvtay2</code>	40	58	45.07	44.00	1	43	17.77	20.00
<code>expmAvtay</code>	9	30	27.54	30.00	0	7	5.29	6.00
<code>expmv</code>	15	55	53.94	55.00	1	38	17.03	17.00
<code>expm_newAv</code>	7	13	12.94	13.00	0	9	7.47	8.00
<code>expleja</code>	16	100	97.63	99.00	1	9	4.40	4.50
<code>expv</code>	30	30	30.00	30.00	-	-	-	-
<code>expmvtay1</code>	40	43	40.34	40.00	1	45	25.60	25.50
<code>expmvtay2</code>	40	58	45.85	44.00	1	44	19.08	21.00
<code>expmAvtay</code>	25	30	27.50	27.50	0	7	5.50	6.00
<code>expmv</code>	45	55	54.09	55.00	2	37	18.96	19.00
<code>expm_newAv</code>	13	13	13.00	13.00	2	9	7.59	8.00
<code>expleja</code>	66	100	98.05	99.00	2	11	5.80	5.00
<code>expv</code>	30	30	30.00	30.00	-	-	-	-
<code>expmvtay1</code>	40	60	40.83	40.00	1	45	8.10	1.00
<code>expmvtay2</code>	40	60	41.52	40.00	1	45	7.26	1.00
<code>expmAvtay</code>	12	30	25.45	25.00	0	8	2.07	0.00
<code>expmv</code>	17	55	42.05	42.50	1	189	9.26	2.00
<code>expm_newAv</code>	9	13	12.24	13.00	0	11	2.62	1.00
<code>expleja</code>	18	100	65.48	72.00	1	583	105.44	6.00
<code>expv</code>	30	30	30.00	30.00	-	-	-	-

Regarding the mean values, `expmv` needed the highest orders of approximation polynomials, followed by `expmvtay2`, `expmvtay1`, and `expmAvtay`. The function `expv` was always invoked using the default value of m , which corresponded to 30. Concerning the value of s , also in average terms, `expmAvtay` always required the smallest values, while `expmvtay1`, in the case of matrix Sets 1 and 2, or `expmv`, in the case of Set 3, demanded the highest values. Alternatively, Figures 1e–3e graphically represent the values of m required in the computation of each of the matrices that compose our test battery by the distinct methods.

In addition to the above analysis related to the accuracy of the results provided by all the codes, their computational costs have also been examined from the point of view of the number of matrix–vector products and the execution time invested by each of them. Thus, Table 5 lists the total number of matrix–vector products carried out by the seven codes in the computation of the matrices of our three sets. As can be noted, `expmvtay2` performed the lowest number of products, followed by `expmvtay1`. Then, following an increasing order in the number of operations involved, `expmv`, `expleja`, `expmAvtay`, and `expm_newAv` would be cited, exchanging the position of these last two codes for Set 3. By far, the largest number of products was carried out by `expv`.

Table 5. Matrix–vector products (P) corresponding to the codes under evaluation for Sets 1–3.

	Set 1	Set 2	Set 3
P(expmvtay1)	95,292	104,037	15,240
P(expmvtay2)	83,378	90,958	14,374
P(expmAvtay)	207,195	210,389	64,988
P(expmv)	108,172	116,018	17,985
P(expm_newAv)	210,744	212,619	60,645
P(expleja)	150,838	165,606	40,787
P(expv)	2,490,385	3,663,677	1,278,078

In a more detailed way, Figures 1f–3f show the ratio between the number of matrix–vector products required by expmvtay1, expmAvtay, expmv, expm_newAv, or expleja and that needed by expmvtay2 in the computation of the matrices of the test sets, decreasingly ordered according to the quotient $P(\text{expmvtay1})/P(\text{expmvtay2})$. In order not to distort these figures and to better appreciate the rest of the results, the ratio with respect to expv has not been considered, due to its excessively high number of products demanded. In the case of Sets 1 and 2, this factor reached values greater than or equal to one in the vast majority of the matrices. For Set 3, it took values belonging to the intervals [1.00, 1.19], [0.89, 43.45], [0.31, 8.93], [0.85, 35.97], and [0.25, 25.78], respectively, for expmvtay1, expmAvtay, expmv, expm_newAv, and expleja.

It is convenient to clarify that expmAvtay computes matrix–vector products to obtain the most appropriate values of the polynomial order (m) and the value of the scaling (s), especially with regard to the estimation of the 1-norm of A^p or A^pB operations, where A and B are square matrices and p is the power parameter. In addition, this function works out matrix products not only in the calculation of these mentioned values, but also in the evaluation of the Taylor approximation polynomial by means of the Paterson–Stockmeyer method. Something very similar could be said about expm_newAv, as matrix–vector products will be carried out by the function expm_new in the estimation of the 1-norm of power of matrices, and matrix products will be as well required when calculating the matrix exponential by means of Padé approximation. As a consequence, the computational cost of each matrix product for expmAvtay and expm_newAv was approximated as n matrix–vector products, where n represents the dimension of the square matrices involved.

On the other hand, Table 6 reports the amount of time required by all the codes in comparison to complete its execution. As expected according to the matrix–vector products, expmvtay2 spent the shortest times, closely followed by expmvtay1. Exceedingly time-consuming resulted to be expv, particularly for Sets 1 and 2. The execution time of expv was more moderate in the case of Set 3, where the response times of expmAvtay and expm_newAv were also remarkable due to the explicit computation of the matrix exponential and its subsequent product by the vector. Figures 1g–3g display graphically these same values by means of bar charts. Again, expv times have not been included so as not to distort the graphs.

Table 6. Execution time (T), in seconds, spent by all the codes for Sets 1–3.

	Set 1	Set 2	Set 3
T(expmvtay1)	2.53	2.80	3.01
T(expmvtay2)	2.37	2.57	2.86
T(expmAvtay)	3.89	3.89	23.29
T(expmv)	4.56	4.79	4.34
T(expm_newAv)	3.88	3.81	21.65
T(expleja)	7.41	7.96	3.27
T(expv)	444.14	648.40	72.89

Finally, in Figures 1h–3h, and always following a descending sequence in the quotient $T(\text{expmvtay1})/T(\text{expmvtay2})$, the ratios of the computation times spent by expmvtay1,

expmAvtay, expmv, expm_newAv, and expleja versus expmvtay2 in each matrix computation are plotted. In the specific case of expmv, this ratio took values within the intervals [1.54, 7.96], [1.35, 7.23], and [0.42, 10.21], respectively, for the Sets 1 to 3. As can be easily noticed in the figures, the results corresponding to expmAvtay and expm_newAv were the highest ones for any set.

4. Algorithm Migration to a GPU-Based Computational Platform

For the next experiment, we provide a GPU-CUDA implementation able to be executed from a MATLAB environment. The MATLAB routine receives an argument that points out on which system do we want to execute the algorithm. This way, we can easily select the system and compare execution times and accuracy.

Figure 4 shows execution time (left) and speed up (right) when executing the algorithm on a GPU environment. The system used for this experiment comprises an Intel(R) Core(TM) i9-7960X CPU @ 2.80 GHz with 16 cores that is denoted as “CPU” in the figure. The GPU device is a NVIDIA Quadro RTX 5000 under the CUDA Driver Version 11.2 (3072 cores). Matrices used in the figure are randomly generated with absolute values between 0 and 1, resulting in an accuracy $\frac{\|x_c - x_g\|}{\|x_c\|} \approx 10^{-16}$. The main result we can observe is that behaviour of the speed increase. For a problem size less than $n \approx 2500$, we do not obtain profit using the GPU. For larger problem sizes, we see more and more speed increase as the size increases, achieving more than two times the performance with the GPU than with the CPU, and achieving slightly better results for expmvtay1 than for expmvtay2.

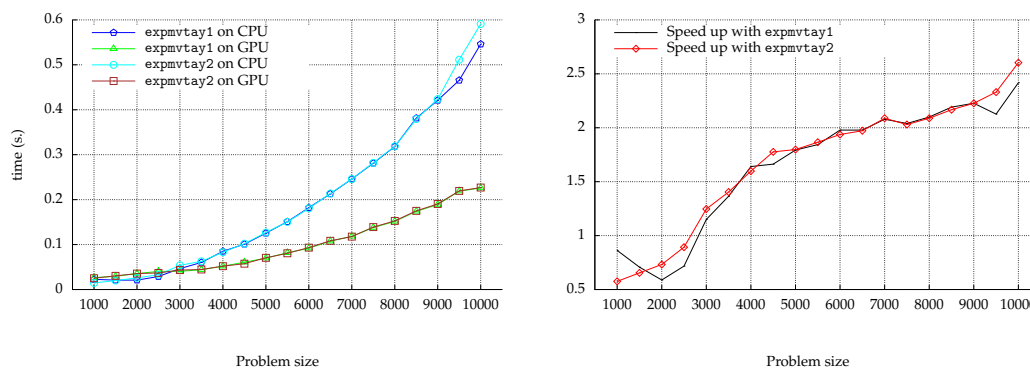


Figure 4. Comparison of time (seconds) and performance (speed up) of expmvtay1 and expmvtay2 being executed on both a CPU and a GPU subsystems.

5. Conclusions

In this paper, two algorithms devoted to the computation of the action of the matrix exponential on a vector have been described. Their numerical and computational performance have been evaluated in several experiments under a testbed composed of distinct state-of-the-art matrices. In general, these algorithms provided a higher accuracy and a lower cost than state-of-the-art algorithms in the literature, with expmvtay1 achieving a slightly higher accuracy in some occasions at a slightly higher cost than expmvtay2.

Both algorithms have been migrated in their implementation to be able to run and take advantage of a computational infrastructure based on GPUs or a traditional computer, such execution being configurable and fully transparent to the user from the MATLAB application itself.

Author Contributions: Conceptualization, J.I., J.M.A. and E.D.; methodology, J.I., J.M.A., P.A.-J., E.D. and J.S.; software, J.I., J.M.A. and P.A.-J.; validation, J.I., J.M.A. and P.A.-J.; formal analysis, J.I., J.M.A., P.A.-J., E.D. and J.S.; investigation, J.I., J.M.A., P.A.-J., E.D. and J.S.; writing—original draft preparation, J.I., J.M.A., P.A.-J., E.D. and J.S.; writing—review and editing, J.M.A. and P.A.-J. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the Vicerrectorado de Investigación de la Universitat Politècnica de València (PAID-11-21).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: Authors wish to thank Marco Caliarì, from the Università di Verona, Italy, for sharing the source code of the function `expLeja` with us.

Conflicts of Interest: The authors declare no conflicts of interest.

References

- Gleich, D.F.; Kloster, K. Sublinear Column-wise Actions of the Matrix Exponential on Social Networks. *Internet Math.* **2015**, *11*, 352–384. [\[CrossRef\]](#)
- De la Cruz Cabrera, O.; Matar, M.; Reichel, L. Analysis of Directed Networks via the Matrix Exponential. *J. Comput. Appl. Math.* **2019**, *355*, 182–192. [\[CrossRef\]](#)
- De la Cruz Cabrera, O.; Matar, M.; Reichel, L. Centrality Measures for Node-weighted Networks via Line Graphs and the Matrix Exponential. *Numer. Algorithms* **2021**, *88*, 583–614. [\[CrossRef\]](#)
- Zhao, Y.L.; Ostermann, A.; Gu, X.M. A low-rank Lie-Trotter splitting approach for nonlinear fractional complex Ginzburg-Landau equations. *J. Comput. Phys.* **2021**, *446*, 110652. [\[CrossRef\]](#)
- Jian, H.Y.; Huang, T.Z.; Gu, X.M.; Zhao, Y.L. Fast compact implicit integration factor method with non-uniform meshes for the two-dimensional nonlinear Riesz space-fractional reaction-diffusion equation. *Appl. Numer. Math.* **2020**, *156*, 346–363. [\[CrossRef\]](#)
- Moler, C.; Van Loan, C. Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later. *SIAM Rev.* **2003**, *45*, 3–49. [\[CrossRef\]](#)
- Defez, E.; Ibáñez, J.; Alonso-Jordá, P.; Alonso, J.; Peinado, J. On Bernoulli matrix polynomials and matrix exponential approximation. *J. Comput. Appl. Math.* **2020**, *404*, 113207. [\[CrossRef\]](#)
- van den Eshof, J.; Frommer, A.; Lippert, T.; Schilling, K.; van der Vorst, H.A. Numerical methods for the QCDd overlap operator. I. Sign-function and error bounds. *Comput. Phys. Commun.* **2002**, *146*, 203–224. [\[CrossRef\]](#)
- Jian, H.Y.; Huang, T.Z.; Ostermann, A.; Gu, X.M.; Zhao, Y.L. Fast IIF-WENO Method on Non-uniform Meshes for Nonlinear Space-Fractional Convection–Diffusion–Reaction Equations. *J. Sci. Comput.* **2021**, *89*, 13. [\[CrossRef\]](#)
- Wang, S.; Peng, Z. Space-time parallel computation for time-domain Maxwell’s equations. In Proceedings of the 2017 International Conference on Electromagnetics in Advanced Applications (ICEAA), Verona, Italy, 11–15 September 2017; pp. 1680–1683.
- Reiman, C.; Das, D.; Rosenbaum, E. Discrete-Time Large-Signal Modeling and Numerical Methods for Flyback Converters. In Proceedings of the 2019 IEEE Power and Energy Conference at Illinois (PECI), Champaign, IL, USA, 28 February–1 March 2019; pp. 1–6.
- Araujo, E.S.; Pestana, R.C. Time evolution of the first-order linear acoustic/elastic wave equation using Lie product formula and Taylor expansion. *Geophys. Prospect.* **2021**, *69*, 70–84. [\[CrossRef\]](#)
- Kole, J. Solving seismic wave propagation in elastic media using the matrix exponential approach. *Wave Motion* **2003**, *38*, 279–293. [\[CrossRef\]](#)
- Falati, M.; Hojjati, G. Integration of chemical stiff ODEs using exponential propagation method. *J. Math. Chem.* **2011**, *49*, 2210–2230. [\[CrossRef\]](#)
- Hammoud, B.; Olivieri, L.; Righetti, L.; Carpentier, J.; Del Prete, A. Fast and accurate multi-body simulation with stiff viscoelastic contacts. *arXiv* **2021**, arXiv:2101.06846.
- Caliari, M.; Kandolf, P.; Zivcovich, F. Backward error analysis of polynomial approximations for computing the action of the matrix exponential. *BIT Numer. Math.* **2018**, *58*, 907–935. [\[CrossRef\]](#)
- Al-Mohy, A.H.; Higham, N.J. Computing the Action of the Matrix Exponential, with an Application to Exponential Integrators. *SIAM J. Sci. Comput.* **2011**, *33*, 488–511. [\[CrossRef\]](#)
- Rostami, M.W.; Xue, F. Robust linear stability analysis and a new method for computing the action of the matrix exponential. *SIAM J. Sci. Comput.* **2018**, *40*, A3344–A3370. [\[CrossRef\]](#)
- Fischer, T.M. On the stability of some algorithms for computing the action of the matrix exponential. *Linear Algebra Its Appl.* **2014**, *443*, 1–20. [\[CrossRef\]](#)
- Fischer, T.M. On the algorithm by Al-Mohy and Higham for computing the action of the matrix exponential: A posteriori roundoff error estimation. *Linear Algebra Its Appl.* **2017**, *531*, 141–168. [\[CrossRef\]](#)

21. Güttel, S.; Kressner, D.; Lund, K. Limited-memory polynomial methods for large-scale matrix functions. *GAMM-Mitteilungen* **2020**, *43*, e202000019.
22. Caliari, M.; Kandolf, P.; Ostermann, A.; Rainer, S. Comparison of software for computing the action of the matrix exponential. *BIT Numer. Math.* **2014**, *54*, 113–128. [[CrossRef](#)]
23. Sidje, R.B. Expokit: A Software Package for Computing Matrix Exponentials. *ACM Trans. Math. Softw. (TOMS)* **1998**, *24*, 130–156. [[CrossRef](#)]
24. Zhu, X.; Li, C.; Gu, C. A new method for computing the matrix exponential operation based on vector valued rational approximations. *J. Comput. Appl. Math.* **2012**, *236*, 2306–2316. [[CrossRef](#)]
25. Sastre, J.; Ibáñez, J.; Ruiz, P.; Defez, E. Accurate and efficient matrix exponential computation. *Int. J. Comput. Math.* **2013**, *91*, 97–112. [[CrossRef](#)]
26. Ruiz, P.; Sastre, J.; Ibáñez, J.; Defez, E. High performance computing of the matrix exponential. *J. Comput. Appl. Math.* **2016**, *291*, 370–379. [[CrossRef](#)]
27. Al-Mohy, A.H.; Higham, N.J. A New Scaling and Squaring Algorithm for the Matrix Exponential. *SIAM J. Matrix Anal. Appl.* **2009**, *31*, 970–989. [[CrossRef](#)]
28. Caliari, M.; Kandolf, P.; Ostermann, A.; Rainer, S. The Leja Method Revisited: Backward Error Analysis for the Matrix Exponential. *SIAM J. Sci. Comput.* **2016**, *38*, A1639–A1661. [[CrossRef](#)]
29. Higham, N.J. *Functions of Matrices: Theory and Computation*; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2008.
30. Higham, N.J. *The Matrix Computation Toolbox*. 2002. Available online: <http://www.ma.man.ac.uk/~higham/mctoolbox> (accessed on 22 December 2021).
31. Wright, T.G. *Eigtool*, Version 2.1. 2019. Available online: <http://www.comlab.ox.ac.uk/pseudospectra/eigtool> (accessed on 22 December 2021).