

Article

A Robust Method for Calculating the Matrix Exponential Based on the Time Finite Element Method

C. S. Jog 

Department of Mechanical Engineering, Indian Institute of Science, Bangalore 560012, India; jogc@iisc.ac.in

Abstract

Schemes that use ordinary-differential-equation solvers for computing the matrix exponential of a constant matrix have generally been considered inefficient compared to other strategies such as scaling and squaring, since they do not exploit the fact that the matrix in question is constant. In this work, we devise a strategy based on the time-finite element that exploits this fact, and thus results in an extremely efficient strategy that requires the solution of a linear system of equations just once at the beginning of the algorithm. Moreover, since the matrix to be inverted is well-conditioned even when the constant matrix is ill-conditioned, the strategy yields very reliable results as demonstrated by means of several challenging examples. The approach outlined here is easily generalized to the case where the matrix is time dependent and where there is a time-dependent forcing function.

Keywords: matrix exponential; differential equation approach; time finite element

MSC: 15A16

1. Introduction

Quite often mathematical models of physical systems require the solution of a system of differential equations of the form $\dot{x}(t) = Ax(t)$ under the initial condition $x(0) = x_0$, where A is a constant $n \times n$ matrix, and $x(t)$ is an $n \times 1$ vector. The analytical solution of this system of equations is $e^{At}x_0$. Using its basic definition, e^{At} can be written as an infinite series; however, it is known that truncating the series to compute the matrix exponential can yield erroneous results, and hence is very unreliable. Alternatively, an exact expression can be derived for the matrix exponential in terms of projections and nilpotent tensors (Equation (40) in Reference [1]). However, finding the projections and nilpotent components of a given matrix is computationally demanding, and thus, although the explicit formula is theoretically interesting, it does not yield an efficient algorithm for numerically computing the matrix exponential. Thus, as we now discuss in greater detail in the following paragraph, finding e^{At} in a computational framework is quite challenging. In this work, we propose a new algorithm that carries out this computation.

Perhaps the most detailed survey of various methods to compute the matrix exponential is the one by Moler and van Loan [2]. In this survey, the authors have discussed various methods such as scaling and squaring, Chebyshev rational approximation, ordinary differential equations approach, polynomial methods, etc. While discussing the differential equations, they write the following:

The scaling and squaring methods of Section 3 and some of the matrix decomposition methods of Section 6 require on the order of 10 to 20 n^3 flops and they obtain higher accuracies than those obtained with 200 n^3 or more flops for the



Academic Editor: Artem A. Lopatin

Received: 23 May 2026

Revised: 4 June 2026

Accepted: 14 June 2026

Published: 3 July 2026

Copyright: © 2026 by the author.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

o.d.e. solvers. This excessive cost is due to the fact that the programs are not taking advantage of the linear, constant coefficient nature of the differential equation. They must repeatedly call for the multiplication of various vectors by the matrix A because, as far as they know, the matrix may have changed since the last multiplication.

It is precisely this constant coefficient nature of the differential equations that we exploit in the proposed approach in this work. Thus, the proposed algorithm needs to solve an $n \times n$ system of equations with n right hand sides just once at the beginning of the program, and during each time step, it needs to multiply this precomputed matrix with the value of the exponential matrix at that time step to obtain the exponential matrix at the next time step.

But more than the efficiency, our emphasis in this work is on the *robustness and generalizability* of the algorithm since so many standard methods are known to fail when A is either ill-conditioned, or when it does not have a linearly independent set of eigenvectors etc. (this is also evident from the title of Reference [2]: ‘Nineteen *dubious* ways ...’). Our method is based on the time finite element method which, similar to the continuum setting, is known to conserve the energy exactly even in the discrete setting for conservative systems [3]. This property is extremely important since this leads to an unconditionally stable algorithm (allowing for arbitrarily large time steps) from an energy perspective. Even if the system is non-conservative, the time finite element method has been demonstrated to yield very reliable results even for chaotic systems such as the Duffing oscillator or the Lorenz system [4]. Thus, the time finite element method is an extremely robust method for the solution of a system of ordinary linear or nonlinear differential equations (in the sense that, for example, it does not fail to compute e^A accurately even when A is ill-conditioned), and this feature will be brought out in Section 3 on a set of challenging examples.

Most of the literature, both classical and recent, seems to prefer the scaling and squaring method [5–8]. There is a class of methods known as exponential integrators [9], but even these algorithms seem to be based on scaling and squaring algorithms for approximating the matrix exponential. Reference [10] compute the matrix exponential approximation using Pade approximations, while References [11,12] use approximations based on Taylor and Euler polynomials, respectively. To the best of our knowledge, there is no algorithm that is based on computing the matrix exponential based on the solution of the governing differential equation, *and which exploits the fact that A is constant to improve the efficiency of the algorithm.*

While solving a dynamical system, one is, ideally speaking, interested in finding the evolution of the solution as a function of time in a given interval, or within a numerical framework, at several discrete time steps in that interval, and in many cases when the matrix A is time dependent and there is, in addition, a forcing function as in Equation (7). The proposed algorithm, since it is based on time-stepping, addresses precisely this need in an efficient manner by using the information from the previous time step. Scaling and squaring algorithms such as the ones implemented as the `expm()` function in MATLAB [13] are extremely robust, efficient, and are highly accurate in computing the exponential of a matrix when A is constant. However, they are not easily generalized to the case given in Equation (7) since the solution may not even be in the form of the exponential of a matrix. We now proceed to describe our algorithm.

2. Formulation

Given a constant matrix A , we are interested in numerically finding a good approximation to e^{At} which can be written as the solution of

$$\dot{Z} = AZ, \quad (1)$$

subject to the initial condition $\mathbf{Z}(0) = \mathbf{I}$, where $\mathbf{Z}$ is an $n \times n$ matrix. We use the time finite element method to carry out the solution of the differential equation in Equation (1). In case we are interested in finding e^{AT} , we partition the interval $[0, T]$ into a number of time finite elements of size $t_\Delta = t_{n+1} - t_n$, where $0 < t_1 < \dots < t_n < t_{n+1} < \dots < T$. Starting from the prescribed initial conditions at $t = 0$, we find the solution at time t_1 , then at time t_2 using the solution obtained at time t_1 , and so on until time T is reached (quite obviously, we simply set $T = 1$ if we are interested in computing e^A). Thus, the key advantage of using the time finite element method is that one obtains not only e^{AT} , but also the entire evolution of the solution for $t \in [0, T]$. As discussed in the Introduction, among the numerous time-stepping schemes on the interval $[t_n, t_{n+1}]$, the time-finite element method is one the most robust strategies that conserves energy even in the discrete setting in the case of conservative systems.

The time finite element is basically a weighted-residual method in time, which uses a piecewise polynomial approximation over each time interval for the variable being computed ($\mathbf{Z}$ in our case), and with the weighing functions being polynomials that are one-order lower than those used for approximating $\mathbf{Z}$ [3]. In this work, we formulate quadratic, cubic and quartic time finite element methods.

Let $N_1(\xi) = \xi(\xi - 1)/2$, $N_2(\xi) = 1 - \xi^2$, $N_3(\xi) = \xi(1 + \xi)/2$, $w_1(\xi) = (1 - \xi)/2$, $w_2(\xi) = (1 + \xi)/2$, $\mathbf{Z}_n := \mathbf{Z}(t_n)$ and $\mathbf{Z}_m = \mathbf{Z}((t_n + t_{n+1})/2)$. With the midnode located at the center of the element, an application of the quadratic time finite element to Equation (1) results in the set of equations

$$t = \frac{(1 - \xi)t_n + (1 + \xi)t_{n+1}}{2}, \quad (2a)$$

$$\mathbf{Z} = N_1(\xi)\mathbf{Z}_n + N_2(\xi)\mathbf{Z}_m + N_3(\xi)\mathbf{Z}_{n+1}, \quad (2b)$$

$$\dot{\mathbf{Z}} = \frac{1}{t_\Delta}[\mathbf{Z}_{n+1} - \mathbf{Z}_n + 2\xi(\mathbf{Z}_{n+1} + \mathbf{Z}_n - 2\mathbf{Z}_m)], \quad (2c)$$

$$\int_{-1}^1 w_i(\xi) (\dot{\mathbf{Z}} - \mathbf{A}\mathbf{Z}) d\xi = \mathbf{0}, \quad i = 1, 2. \quad (2d)$$

Equation (2d) leads to

$$(4\mathbf{I} - 2t_\Delta\mathbf{A})\mathbf{Z}_m - (5\mathbf{I} + t_\Delta\mathbf{A})\mathbf{Z}_n + \mathbf{Z}_{n+1} = \mathbf{0}, \quad (3a)$$

$$(4\mathbf{I} + 2t_\Delta\mathbf{A})\mathbf{Z}_m + \mathbf{Z}_n - (5\mathbf{I} - t_\Delta\mathbf{A})\mathbf{Z}_{n+1} = \mathbf{0}. \quad (3b)$$

Solving the above set of equations we get

$$\mathbf{Z}_m = \frac{(4\mathbf{I} + t_\Delta\mathbf{A})\mathbf{Z}_n + (4\mathbf{I} - t_\Delta\mathbf{A})\mathbf{Z}_{n+1}}{8},$$

which on substituting into any of Equation (3) finally leads to

$$\mathbf{Z}_{n+1} = \mathbf{M}\mathbf{Z}_n, \quad (4)$$

where

$$\mathbf{M} := \left[6(2\mathbf{I} - t_\Delta\mathbf{A}) + t_\Delta^2\mathbf{A}^2 \right]^{-1} \left[6(2\mathbf{I} + t_\Delta\mathbf{A}) + t_\Delta^2\mathbf{A}^2 \right].$$

Similar to the above derivation, an application of the cubic and quartic time finite element methods leads to

$$\mathbf{Z}_{n+1} = \mathbf{M}\mathbf{Z}_n, \quad (5)$$

where

$$\begin{aligned} M &:= \left[120I - 60t_{\Delta}A + 12t_{\Delta}^2A^2 - t_{\Delta}^3A^3 \right]^{-1} \left[120I + 60t_{\Delta}A + 12t_{\Delta}^2A^2 + t_{\Delta}^3A^3 \right] \quad (\text{cubic}), \\ &:= \left[1680I - 840t_{\Delta}A + 180t_{\Delta}^2A^2 - 20t_{\Delta}^3A^3 + t_{\Delta}^4A^4 \right]^{-1} \\ &\quad \left[1680I + 840t_{\Delta}A + 180t_{\Delta}^2A^2 + 20t_{\Delta}^3A^3 + t_{\Delta}^4A^4 \right] \quad (\text{quartic}). \end{aligned}$$

Note that since A is a constant matrix in any of the above methods, and if we use time intervals of equal length, then the matrix to be inverted in the expression for M is a constant matrix. This solution of the $n \times n$ system of equations with n right hand sides to find M is carried out only once at the beginning of the algorithm. Starting with $Z_0 = I$, we march forward in time until the time instant $t = T$ is reached. After each time step calculation (which involves only the multiplication of the pre-calculated M with Z_n), we assign $Z_{n+1} \rightarrow Z_n$, and proceed to the next time step.

Even if A is singular or close to singular, the invertibility of $[6(2I - t_{\Delta}A) + t_{\Delta}^2A^2]$ in the quadratic time finite element method, and the corresponding matrices in the cubic and quartic methods is not affected due to the presence of the identity matrix. Note also that the algorithm is not dependent on any special structure of the matrix A such as diagonalizable, Jordan canonical form, or one having linearly independent eigenvectors, etc., and is thus applicable for any matrix A . Since T can be chosen arbitrarily by the user, the above algorithm actually yields e^{AT} for any choice of T , and, moreover, it yields $[e^{At_n}, e^{At_{n+1}}]$ for every subinterval $[t_n, t_{n+1}]$ that the domain $[0, T]$ is divided into.

Once e^A is computed, other related functions can be computed as

$$\begin{aligned} \cosh A &= \frac{e^A + e^{-A}}{2}, \\ \sinh A &= \frac{e^A - e^{-A}}{2}, \\ \tanh A &= (e^{2A} - I)(e^{2A} + I)^{-1} = (e^{2A} + I)^{-1}(e^{2A} - I). \end{aligned}$$

The proposed method also works for complex matrices A with the only change being that a complex solver is now used to compute M . In fact, we can compute the cosine, sine and tangent of a matrix as

$$\begin{aligned} \cos A &= \operatorname{Re}(e^{iA}), \\ \sin A &= \operatorname{Im}(e^{iA}), \\ i \tan A &= (e^{2iA} - I)(e^{2iA} + I)^{-1} = (e^{2iA} + I)^{-1}(e^{2iA} - I). \end{aligned}$$

The method is easily modified to solve the set of differential equations $\dot{x} = Ax$ with $x(0) = \hat{x}_0$, where x is now a $n \times 1$ vector; in place of Equation (4), we now get

$$x_{n+1} = Mx_n, \quad (6)$$

with $x_0 = \hat{x}_0$, with M calculated only once at $t = 0$ as in the case of Equation (4) or Equation (5), and with the assignment $x_{n+1} \rightarrow x_n$ at the end of each time step.

The method is also even easily extended to the case where the matrix A is a function of time, and, in addition, there is a forcing function, i.e., we now have

$$\dot{x} = A(t)x + f(t), \quad (7)$$

with $x(0) = \hat{x}_0$. As an example, to formulate the quadratic time finite element for this case, let $N_1(\xi) = \xi(\xi - 1)/2$, $N_2(\xi) = 1 - \xi^2$, $N_3(\xi) = \xi(1 + \xi)/2$, $w_1(\xi) = (1 - \xi)/2$, $w_2(\xi) = (1 + \xi)/2$, $x_n := x(t_n)$, $x_m = x((t_n + t_{n+1})/2)$ and $x_{n+1} = x(t_{n+1})$, and let

$$t = \frac{(1 - \xi)t_n + (1 + \xi)t_{n+1}}{2}, \quad (8a)$$

$$x = N_1(\xi)x_n + N_2(\xi)x_m + N_3(\xi)x_{n+1}, \quad (8b)$$

$$\dot{x} = \frac{1}{t_\Delta} [x_{n+1} - x_n + 2\xi(x_{n+1} + x_n - 2x_m)], \quad (8c)$$

$$\int_{-1}^1 w_i(\xi) [\dot{x} - A(t)x - f(t)] d\xi = 0, \quad i = 1, 2, \quad (8d)$$

where the ' t ' in Equation (8d) is given by Equation (8a). Depending on $A(t)$ and $f(t)$, the integrals in Equation (8d) have to be evaluated for w_i , $i = 1, 2$, and, given x_n , one then solves these two equations for (x_m, x_{n+1}) on the interval $[t_n, t_{n+1}]$ with $x_0 = \hat{x}_0$. Of course, this now involves inverting a matrix at every time step (as opposed to the constant A case, where a matrix has to be inverted only once at the beginning of the program), and, as expected, will thus be computationally more expensive compared to the constant A case that we have dealt with in this work. The formulation of the cubic and quartic finite element methods is similar to the above formulation.

There is a dramatic increase in accuracy and a corresponding reduction in the run time (for a given level of accuracy) as one progresses from the quadratic to the cubic, and from the cubic to the quartic time finite element method. The only additional cost is in constructing A^3 and A^4 in the two methods (which is negligible since this is carried out only once at the beginning of the program). One and two additional $n \times n$ arrays, respectively, are required over and above those in the quadratic time finite element method.

3. Numerical Examples

We now illustrate the good performance of the algorithm on examples from the literature where certain algorithms fail, and on some new examples as well. All results are presented using the quartic time finite element method. So far we have not come across an example where an algorithm from the literature works, while the proposed algorithm fails. Since the solution approaches the exact solution as $t_\Delta \rightarrow 0$, one can obtain any desired level of accuracy simply by decreasing t_Δ . The precise value of t_Δ that is to be used of course depends on the matrix A , but extensive numerical experimentation with large or ill-conditioned matrices A indicates that a value of t_Δ of 10^{-3} or 10^{-4} yields extremely accurate results in most cases. Thus, keeping in view that the proposed algorithm is efficient (only one system of equations with multiple right hand sides to be solved at $t = 0$, followed by matrix multiplications at each time step; see the runtime for an example stated below), we have focused more on presenting examples where computing e^A accurately is important, rather than focusing on examples where the size of A is large, although we do present such as example as well. The runs for both the proposed and a Fortran implementation of the scaling and squaring algorithm were carried out on stand-alone I7-6700@3.40 GHz Intel-based PC (Intel Corporation, Santa Clara, CA, USA), with no parallelization of the Fortran codes. To give an idea of the efficiency of the algorithm, the runtime for the 8×8 matrix A with $t_\Delta = 10^{-3}$ is shown by the system as 0.00 s (which means that it is less than 10^{-2} s even on a relatively old desktop, and with no parallelization). Similar to the proposed algorithm, the scaling and squaring algorithm such as the `expm()` function in MATLAB [13] also yields extremely accurate results in negligible time for the small matrix examples, so that we claim no superiority of our algorithm over the `expm()` function in terms of run time. However, as explained in detail in the Introduction, the scaling and

squaring method is not easily generalizable to the case given by Equation (7). The runtimes for the large matrix example with both algorithms are stated in the example itself. In addition to the memory required for storing A and e^A , our algorithm requires five $n \times n$ arrays for the quartic case, while the implementations of the scaling and squaring algorithm that we have come across require six to eight additional $n \times n$ arrays.

Wherever presented, the relative error is calculated using the Frobenium norm as

$$\|E\| = \frac{\|(e^A)_{\text{reference}} - (e^A)_{\text{obtained}}\|_F}{\|(e^A)_{\text{reference}}\|_F},$$

where the reference solution is obtained using Mathematica [14] (which in some cases is the analytical solution).

1. This example shows that using a truncated version of the infinite series expansion of e^A can lead to inaccurate results [2]. The matrix and the result obtained to 8 decimal places using the proposed algorithm with $t_\Delta = 0.01$ are

$$A = \begin{bmatrix} -49 & 24 \\ -64 & 31 \end{bmatrix},$$

$$e^A = \begin{bmatrix} -0.73575876 & 0.55181910 \\ -1.47151760 & 1.10363824 \end{bmatrix},$$

which agrees with the solution in the mentioned reference, and with the error as given in Table 1.

Table 1. Variation of the error norm with time step for Example 1.

	t_Δ	
	0.1	0.01
$\ E\ $	5.73×10^{-12}	4.05×10^{-14}

2. This example is again from [2]. The matrix and the analytical solution are

$$A = \begin{bmatrix} 0 & 6 & 0 & 0 \\ 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

$$e^A = \begin{bmatrix} 1 & 6 & 18 & 36 \\ 0 & 1 & 6 & 18 \\ 0 & 0 & 1 & 6 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

An almost perfect match is found between the solution from the proposed algorithm with $t_\Delta = 0.01$, and the above analytical solution with a deviation only in the 14th or 15th decimal place for some of the entries.

3. If A is nearly defective, then methods based on eigenvectors can yield erroneous results [2]. An example of such a matrix, and the analytical solution for its exponential are given by

$$A = \begin{bmatrix} 1 + \epsilon & 1 \\ 0 & 1 - \epsilon \end{bmatrix},$$

$$e^A = \begin{bmatrix} e^{1+\epsilon} & \frac{e \sinh \epsilon}{\epsilon} \\ 0 & e^{1-\epsilon} \end{bmatrix},$$

For $\epsilon = 10^{-5}$ and $t_\Delta = 0.1$, the error norm is 2.82×10^{-12} showing the high accuracy of the proposed algorithm even with a relatively large t_Δ . Another similar example is presented in Reference [15], where, again, the proposed algorithm yields the correct solution.

4. If A is in the form of a Jordan block, say with the form given below, then we have [2]

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$e^A = e \begin{bmatrix} 1 & 1 & 1/2! & 1/3! \\ 0 & 1 & 1 & 1/2! \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Once again with $t_\Delta = 0.01$, the proposed algorithm yields the above analytical solution almost perfectly with an error norm of around 1.06×10^{-14} .

5. This is a challenging example due to the highly ill-conditioned matrix A , where one of the older versions of the function `expm(.)` in MATLAB [13] yielded an erroneous result (note, however, that the current version does yield the correct result) [16]. With $a = 2 \times 10^{10}$, $b = 4 \times 10^8/6$, $c = 200/3$, $d = 3$, $e = 10^{-8}$, the matrix and the result obtained using the proposed algorithm with $t_\Delta = 0.001$ are as follows:

$$A = \begin{bmatrix} 0 & e & 0 \\ -(a+b) & -d & a \\ c & 0 & -c \end{bmatrix},$$

$$e^A = \begin{bmatrix} 0.44684946828318223 & 1.5404415738395189\text{E-}9 & 0.46281145355877312 \\ -5743067.7794792280 & -1.5283003868680811\text{E-}2 & -4526542.7127836645 \\ 0.44772297784950188 & 1.5427048451959113\text{E-}9 & 0.046348064883764961 \end{bmatrix}.$$

The error norms are as shown in Table 2, and again demonstrate the high accuracy of the proposed algorithm.

Table 2. Variation of the error norm with time step for Example 5.

	t_Δ	
	0.1	0.01
$\ E\ $	7.4×10^{-10}	5.8×10^{-14}

6. Consider A to be a skew-symmetric matrix; in which case, we know that e^A is an orthogonal matrix [17]. As an example,

$$A = \begin{bmatrix} 0 & 10^6 \\ -10^6 & 0 \end{bmatrix},$$

$$e^A = \begin{bmatrix} \cos(10^6) & \sin(10^6) \\ -\sin(10^6) & \cos(10^6) \end{bmatrix}.$$

The error norms with the proposed algorithm are as shown in Table 3.

Table 3. Variation of the error norm with time step for Example 6.

	t_Δ	
	10^{-7}	10^{-8}
$\ E\ $	3.9×10^{-10}	7.0×10^{-9}

Note that a very fine t_Δ is required to capture the very high frequency oscillation of the solution with time. The `expm()` function yields a solution with an error norm of 5.5×10^{-17} , and is thus much more effective than the proposed algorithm both in terms of accuracy and computational time on this particular example. This is because our algorithm is based on time-stepping, and errors accumulate from one time step to another.

7. This is again an example of an extremely ill-conditioned matrix, which makes it very challenging to find its matrix exponential. The matrix and the result that we obtain with $t_\Delta = 0.001$ are

$$A = \begin{bmatrix} 1 & 0 & 0 & 300 & 0 & 0 & 0 & -175 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 30 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 10 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 40 & 0 & -130 & 0 & 0 & 1 \end{bmatrix},$$

$$e^A = \begin{bmatrix} 2.15802578 \times 10^{41} & 0 & -1.67846449 \times 10^{41} & 6.82427671 \times 10^{41} & 5.45500960 \times 10^{41} & 0 & 0 & -3.98082808 \times 10^{41} \\ 0 & 2.71828183 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2.71828183 & 0 & 0 & 0 & 0 & 0 \\ 6.82427671 \times 10^{40} & 0 & -5.30777077 \times 10^{40} & 2.15802578 \times 10^{41} & 1.72502550 \times 10^{41} & 0 & 0 & -1.25884837 \times 10^{41} \\ 0 & 0 & 0 & 0 & 2.71828183 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2.71828183 & 0 & 0 \\ 0 & 27.18281828 & 0 & 0 & 0 & 0 & 2.71828183 & 0 \\ 0 & 0 & 108.73127314 & 0 & -353.37663770 & 0 & 0 & 2.71828183 \end{bmatrix}.$$

Note the dramatic variations in the entries of e^A . The above result agrees very closely with the one obtained using the `expm(.)` function in MATLAB [13]. The function `expmdemo3` in MATLAB [13] which is based on eigenvalues/eigenvectors yields an incorrect result (after giving a warning that the results may be inaccurate due to the ill-conditioning of A). The error norms are as shown in Table 4.

Table 4. Variation of the error norm with time step for Example 7.

	t_Δ	
	0.01	0.001
$\ E\ $	2.8×10^{-6}	9.4×10^{-14}

8. This example shows that the proposed algorithm can compute the cosine or sine of a matrix accurately. Let A be as in Example 2. The analytical solutions are

$$\cos(A) = \begin{bmatrix} 1 & 0 & -18 & 0 \\ 0 & 1 & 0 & -18 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$\sin(A) = \begin{bmatrix} 0 & 6 & 0 & -36 \\ 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

An almost perfect match with the above analytical solutions is obtained with $t_{\Delta} = 10^{-3}$.

9. This example is challenging since the matrix A is both large and singular. Consider a 100×100 matrix A with all entries as 1. Only one eigenvalue of A is nonzero with a value of 100. The diagonal elements of e^A are given by $(e^{100} + 99)/100$ (which is approximately $2.688117141816136 \times 10^{41}$), while the off-diagonal elements are $(e^{100} - 1)/100$. A Fortran implementation of the scaling and squaring algorithm gives a diagonal value of $2.689864309677 \times 10^{41}$ with a run time of 0.04 s, while the proposed algorithm yields diagonal values of $2.6881062463688374 \times 10^{41}$ and $2.6881171418167547 \times 10^{41}$ with run times of 0.33 s ($t_{\Delta} = 10^{-2}$) and 2.5 s ($t_{\Delta} = 10^{-3}$), respectively, showing the extremely high accuracy of the proposed method even for large and singular matrices. The result obtained with the `expm()` function of MATLAB [13] is $2.688117141815979 \times 10^{41}$. Although the run time with the scaling and squaring algorithm may appear to be smaller compared to that of the proposed method, obtaining the solution at all the intermediate 1000 time steps (which is what is generally of interest) as in our algorithm would involve either 1000 runs or matrix multiplications in order to obtain the complete dynamical solution.

4. Conclusions

A robust strategy that simply involves the repeated evaluation of the recursion relations given by Equations (4), Equation (5), or Equation (6) at each time step to find the matrix exponential, or to solve a set of differential equations with constant coefficient, respectively, at different times within a given time interval, has been developed in this work. There are no factors that need to be adjusted either within the algorithm or by the user. The performance of the proposed algorithm has been compared with existing strategies, and has been shown to be robust even on a set of pathological examples. The key advantage of the proposed method over the traditional techniques is that it computes the entire time evolution of the matrix Z in Equation (1), and unlike traditional methods, is easily generalized to the case where the matrix A is a function of time, and where the governing equation involves a forcing function as in Equation (7).

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The author declares no conflicts of interest.

References

1. Jog, C.S. Derivatives of the stretch, rotation and exponential tensors in n -dimensional vector spaces. *J. Elast.* **2006**, *82*, 175–192. [CrossRef]
2. Moler, C.B.; van Loan, C.F. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM Rev.* **2003**, *45*, 3–49. [CrossRef]
3. Betsch, P.; Steinmann, P. Inherently energy-conserving time finite elements for classical mechanics. *J. Comput. Phys.* **2000**, *160*, 88–116. [CrossRef]
4. Jog, C.S.; Agrawal, M.; Nandy, A. The time finite element as a robust general scheme for solving nonlinear dynamic equations including chaotic systems. *Appl. Math. Comput.* **2016**, *279*, 43–61. [CrossRef]
5. Higham, N.J. The scaling and squaring method for the matrix exponential revisited. *SIAM J. Matrix Anal. Appl.* **2005**, *26*, 1179–1193. [CrossRef]
6. 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.
7. Blanes, S.; Kolylov, N.; Seydaoglu, M. Efficient scaling and squaring method for matrix exponential. *SIAM J. Matrix Anal. Appl.* **2025**, *46*, 94–120. [CrossRef]
8. Sastre, J.; Ibanez, J.; Defez, E.; Ruiz, P. New scaling-squaring Taylor algorithms for computing the matrix exponential. *SIAM J. Sci. Comput.* **2015**, *37*, A439–A455. [CrossRef]
9. 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**, *35*, 488–511. [CrossRef]
10. Ward, R.C. Numerical computation of the matrix exponential with accuracy estimate. *SIAM J. Numer. Appl.* **1977**, *14*, 600–610. [CrossRef]
11. Bader, P.; Blanes, S.; Casas, F. Computing the matrix exponential with an optimized Taylor polynomial approximation. *Mathematics* **2019**, *7*, 1174. [CrossRef]
12. Alonso, J.M.; Ibáñez, J.; Defez, E.; Alonso-Jordá, P. Euler polynomials for the matrix exponential approximation. *J. Comput. Appl. Math.* **2023**, *425*, 115074. [CrossRef]
13. MATLAB, version 24.2 (R2024b); The MathWorks Inc.: Natick, MA, USA, 2024. Available online: <https://www.mathworks.com> (accessed on 23 January 2025).
14. Mathematica, version 14.2.1; Wolfram Research Inc.: Champaign, IL, USA, 2014.
15. Matrix Exponentials. Available online: <https://in.mathworks.com/help/matlab/math/matrix-exponentials.html> (accessed on 15 January 2026).
16. A Balancing Act for the Matrix Exponential. Available online: <https://blogs.mathworks.com/cleve/2012/07/23/a-balancing-act-for-the-matrix-exponential/> (accessed on 25 January 2026).
17. Gallier, J.; Xu, D. Computing of exponentials of skew-symmetric matrices and logarithms of orthogonal matrices. *Int. J. Robot. Autom.* **2002**, *17*, 1–11.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.