FUNCTIONAL DIFFERENTIAL EQUATIONS

VOLUME 23
2016, NO 3–4
PP. 11–32

A STABLE ALGORITHM FOR MATRIX EXPONENT CALCULATION

TEDDY LAZEBNIK AND SHLOMO YANETZ *

Abstract. This article proposes a numerical algorithm for the matrix exponent $A(m) = e^M$ calculation, which is stable for every matrix $M \in C^{n\times n}$ and for given number $k \in N$ of significant digits. The algorithm is based on C. Lanczos method of eigenvalues calculation. A theoretical analysis and proof of stability of the algorithm is given.

Key Words. Matrix exponent, numeric algorithms of ordinary differential equations, numerical stability analysis.

AMS(MOS) subject classification. 3404, 34D04.

1. Introduction. Consider a system of n linear homogeneous first order ordinary differential equations with constant coefficients and $k \in N$ the number of significant digits in the calculation.
In this article, we present a high accuracy numerical solution of e^M for any given complex matrix M and k significant digits using a stable algorithm.
A matrix exponent can be formally dened by the convergent power series

$$e^{Mx} = \sum_{n=0}^{\infty} \left(\frac{(Mx)^n}{n!} \right).$$

By this definition, it is clear to see that $e^{Mx} \in C^{nXn}$. The stable computation of this matrix exponent is the main topic of this article. We will assume that we can store all the elements in the main memory of a contemporary computer. A numerical algorithm should be examined by the following attributes: generality, reliability, stability, accuracy, storage requirements, and complexity as follows. Generality: the size of the subspace of options from the

* Bar Ilan University, Department of Mathematics

full option space (the assumptions we add to the problem) Reliability: If the algorithm makes a variation of the solution it alerts about it and returns that the answer is unreliable. Stability: the algorithm will find the right answer for any given input from the sub-space that is defined. Accuracy: the accuracy in k significant digits calculation or floating point calculation.Storage requirements: the standard memory complexity, but the factor is important too.

2. The Known Algorithms. Many numerical algorithms are known that solve the matrix exponent problem [5]. Most of the algorithms are not universal, but good for some class of matrices. Some algorithms are just stable for all matrices except stiff matrices, even of moderate stiffness. Most of the algorithms have a problem with $k \in N$ digits calculations as a result of the rounding errors occurring in the process.
Below are shown the most used methods, known for matrix exponent and their stability weaknesses.

Pade approximation based method:

$$R_{pq}(A) = \left[\sum_{j=0}^{q}\left(\frac{(p+q-j)!p!}{(p+q)!j!\,(q-j)!}\left(-A^{j}\right)\right)\right]^{-1} * \sum_{j=0}^{p}\left(\frac{(p+q-j)!p!}{(p+q)!j!\,(p-j)!}A^{j}\right).$$

<u>Weakness:</u> When $\|A\|$is large then the approximation is poor [5].

General purpose ordinary differential equations ODE solver:
This is not a specific algorithm for computing a matrix exponent. It is based on the idea that calculating the result of the following equation with ODE solve algorithms will produce the right answer. Since the result of this equation defined to be a matrix exponent, it solves the problem (1).

<u>Weakness:</u> This solution does not take into consideration the linear nature of the problem and uses very time-consuming algorithms. For high dimension systems, this method delivers a poor approximation to the solution because the error is related to the dimension of the system.

CayleyHamilton based method:

$$e^{At} = \sum_{j=0}^{n-1} \left(a_j\left(t\right) A^j\right) \text{ when } a_j\left(t\right) = \sum \left(\frac{b_{kj} t^k}{k!}\right)$$

$$\text{and } b_{kj} = \begin{cases} \delta_{kj}, & k < n \\ c_j, & k = n \\ c_0 b_{k-1,n-1}, & k > n,\ j = 0 \\ c_0 b_{k-1,n-1} + b_{k-1,j-1}, & k > n,\ j > 0 \end{cases}$$

<u>Weakness:</u> The coefficients b_{kj} are very sensitive to round off error and its bubble up in the calculation and generate a large error.

Lagrange interpolation based method:

$$e^{At} = \sum_{j=0}^{n-1} \left(e^{\lambda_j t} \prod_{\substack{k=1 \\ k \neq j}}^{n} \left(\frac{A - \lambda_k I}{\lambda_j - \lambda_k}\right)\right) .$$

<u>Weakness:</u> For a matrix with close but not equal eigenvalues, the formula has a big cancellation error as a result of dividing each iteration by $\lambda_j - \lambda_k$. Matrices with eigenvalues with great algebraic multiplicity will produce a big error and the algorithm is unstable in this case [2].

Newton interpolation based method:

$$e^{At} = e^{\lambda_1 t} I + \sum_{j=2}^{n} \left([\lambda_1, \cdots, \lambda_j] \prod_{\substack{k=1 \\ k \neq j}}^{n} (A - \lambda_k I)\right) \text{ when } [\lambda_1, \lambda_2] = \frac{e^{\lambda_1 t} - e^{\lambda_2 t}}{\lambda_1 - \lambda_2}$$

$$\text{and } [\lambda_1, \cdots, \lambda_{k+1}] = \frac{[\lambda_1, \cdots, \lambda_k] - [\lambda_2, \cdots, \lambda_{k+1}]}{\lambda_1 - \lambda_{k+1}} \, (k \geq 2)$$

<u>Weakness:</u> Same as Lagrange interpolation's weakness due to the same reason.

Inverse Laplace transforms based method:

$$e^{At} = \sum_{k=0}^{n-1} \left(L^{-1}\left[\frac{s^{n-k-1}}{c\left(s\right)}\right] A_k\right) \text{ when } A_k = A_{k-1} A - c_{n-k} I, A_0 = I$$

$$\text{and } c(s) = s^n - \sum_{k=0}^{n-1} \left(c_k s^k\right) \text{ when } c_{n-k} = -\frac{tr\left(A_{k-1}A\right)}{k}.$$

Weakness: This is a very time consuming algorithm which is seriously affected by round off errors.

3. Proposed Algorithm Description and Theoretical Analysis.

Algorithm: Given $M \in C^{nXn}$, this algorithm computes e^M

1. Eigs = Lanczos(M)
2. Eigs = controlEigs(Eigs, precision)
3. r = $e^{Eigs(1)}$
4. P = $I^{dim(M)\times dim(M)}$
5. $i = 1$
6. Repeat
   (a) expm = r * P
   (b) $r = \sum_k \sum_b \left(\frac{(-1)^n c_b}{a_k^{n+1}} * \sum_{m=0}^{n} \left(\frac{n!}{(n-m)!} * (-bx)^{n-m}\right)\right)$
   (c) $P = P * (M - I^{dim(M)\times dim(M)} * Eigs_i$
7. Until $(i = dim\,(M))$

when Lanczos is the Lanczos algorithm [6] with the improvement [7] followed by recursive spectral bisection [9].

ControlEigs:

• For given vector of eigenvalues of M and the precession with is considered as "two eigenvalues are close".

1. $i = 1$
2. Repeat
   (a) $j = i$
   (b) Repeat
      i. If $|Eigs_i - Eigs_j| < precession$
         1. $Eigs_i = Eigs_j$
      ii. $j = j + 1$
   (c) Until (j = Length(Eigs))
   (d) If $|Eigs_i| < precession$
      i. $Eigs_i = 0$
   (e) $i = i + 1$
3. Until (i = Length(Eigs))

Definition. We will use the definition of "stable" as "backward stable": for small Δx the result of a function f with the input $x + \Delta x$ has small error $\Delta y, f\,(x + \Delta x) = y + \Delta y$ when $f\,(x) = y$.

THEOREM. *For any given matrix $M \in C^{nXn}$, the proposed numerical algorithm is stable and solves equation (1) with error bounded by $dim\,(M)^3 * \mu\,(M) * \|M^{dim(M)}\|$ when $\mu\,(M)$ is a polynomial of the same size as the matrix.* Note: When working with k significant digit arithmetic. $x, y \in R : |x - y| < 10^{-k} \rightarrow x = y$.

Proof. Consider the following equation:

$$Y^{'} = MY,\ M \in \mathbb{C}^{nXn} \quad .$$

Firstly, according to the proposed algorithm, the eigenvalues of matrix M need to be found. It is possible to find the biggest eigenvalue of M using Lanczos' algorithm [6]. According to Ojalvo and Newman this process is numerally stable [7]. The result of this process is a Tridiagonal matrix T_{mm} which is similar to the original matrix M. The eigenvalues of T_{mm} can be obtained using spectral bisection [9].

In each iteration we will find at least the biggest eigenvalue [7]. For finding all the eigenvalues, we need only to find the biggest eigenvalue and then find the next one in the smaller matrix until there is one dimensional matrix with is the last eigenvalue.

According to Putzer's algorithm [2], the equation: $Y^{'} = MY, M \in C^{nXn}$ has a solution with the form:

$$Y = \sum_{j=0}^{n-1} \left(r_{j+1}\,(x) * P_j\right)$$

where

$$P_0 = I\ ,\ P_j = \prod_{k=1}^{j} (A - \lambda_k I)\ for\ j = 1,\ 2, \ldots,\ n-1$$

and

$$r_1^{'} = \lambda_1 r_1, r_1\,(0) = 1 \& r_j^{'} = \lambda_j r_j + r_{j-1}, r_j\,(0) = 0 for j = 2, 3, , n$$

λ_i for $j = 1, 2, , n$ are the eigenvalues of matrix M.

The solution for $r_1^{'} = \lambda_1 r_1, r_1\,(0) = 1 \& r_j^{'} = \lambda_j r_j + r_{j-1}, r_j\,(0) = 0$ takes the form:

$$r_j = \left(\int \left(e^{-\lambda_j x} r_{j-1}(x)\right) dx - \int \left(r_{j-1}\,(x)\right) dx \Big|_{x=0}\right) * e^{\lambda_j x},\quad r_1 = e^{\lambda_1 x} \quad .$$

Using the form $y = \left(\int \left(\left(e^{\int p(x)dx}\right) * q\,(x)\right) dx + c\right) * e^{-\int p(x)dx}$ when $p\,(x) = -\lambda_i, q\,(x) = r_{i-1}$

$$P_0 = I, P_j = P_{j-1}\,(A - \lambda_j I)\ \text{ for } j = 1, 2, , n-1.$$

If $\lambda_1 = 0$ then $r_i = \frac{(-1)^i}{\lambda_2 \lambda_i}$ for $2 \leq i \leq n$ and $r_1 = 1$. The calculation is stable as a result of multiplicity of numbers but may have a big cancelation error when $\lambda_2 * * \lambda_i \to 0$. Else $\lambda_1 \neq 0$ and then $r_1 = e^{\lambda_1 x}$ with is stable as a result of computing the power of 2 floating numbers. ∎

Claim. *For every $1 \leq i \leq n, r_i$ takes the form $\sum_k (p_k(x) * e^{a_k x})$ where $p_k(x)$ are polynomials and $a_k \in C$ are constants.*

Proof of claim. Induction on i.
For $i = 1$: $r_i = 1 * e^{\lambda_i x}$ with satisfy the form requested form. For r_i assume the form is satisfied, we need to prove that r_{i+1} also satisfies the form. Since, $r_i = \left(\int \left(e^{-\lambda_i x} r_{i-1}(x)\right) dx - \rangle \int (r_{i-1}(x))\, dx|_{x=0}\right) * e^{\lambda_i x}$, it is enough to prove that the calculation of r_i preserves the form. We are looking at $\int (r_{i-1}(x))\, dx|_{x=0} * e^{\lambda_i x}$. $\int (r_{i-1}(x))\, dx|_{x=0}$. It is a complex number for any r_{i-1} that satisfies the form so $\int (r_{i-1}(x))\, dx|_{x=0} * e^{\lambda_i x}$ is equal to $k_i * e^{\lambda_i * x}$ when $C \ni k_i = \int (r_{i-1}(x))\, dx|_{x=0}$ and this satisfy the form. $\int \left(e^{-\lambda_i * x} r_{i-1}(x)\right) dx$ can be presented as $\int \left(e^{-\lambda_i x} \sum_k (p_k(x) * e^{a_k x})\right) dx$ when $b_k = a_k - \lambda_i$. It also can be presented as $\int \left(\sum_k \left(p_k(x) * e^{b_k x}\right)\right) dx$. Integration of sum is equal to the sum of the integrations, so we can write it as $\sum_k \left(\int \left(p_k(x) * e^{b_k x}\right) dx\right)$. It is enough to prove that $\int \left(p_k(x) * e^{b_k x}\right) dx$ satisfies the form, because a sum of sums of elements that satisfy the form, also satisfies the form.
$p_{k(x)}$ can be presented as $p_k(x) = c_n x^n + \ + c_1 x + c_0$ because $p_k(x)$ is a polynomial. Therefore:
$\int \left(p_k(x) * e^{b_k x}\right) dx = \int \left((c_n x^n + \ + c_1 x + c_0) * e^{b_k x}\right) dx = c_n \int \left(x^n e^{b_k x}\right) dx + + c_0 \int \left(e^{b_k x}\right) dx$.
It is enough to prove that $c_n \int \left(x^n e^{b_k x}\right) dx$ satisfies the form, because a sum of sums of elements that satisfy the form, also satisfy the form. $\int \left(c_n x^n e^{b_k x}\right) dx = \int_{-bx}^{\infty} \left(\frac{c_n}{(-b)^n} t^n e^{-t}\right) dt = \left(\frac{-1}{b}\right)^{n+1} c_n e^{bx} * \sum_{m=0}^{n} \left(\frac{n!}{(n-m)!} * (-bx)^{n-m}\right)$.
It is easy to see that $\left(\frac{-1}{b}\right)^{n+1} c_n e^{bx} * \sum_{m=0}^{n} \left(\frac{n!}{(n-m)!} * (-bx)^{n-m}\right)$ satisfies the form. Therefore, $\int \left(e^{-\lambda_i x} \sum_k (p_k(x) * e^{a_k x})\right) dx$ satisfy the form and can be presented as $\sum_k (p_k(x) * e^{a_k x})$.
So,

$$\int \left(e^{-\lambda_i x} \sum_k (p_k(x) * e^{a_k x})\right) dx * e^{\lambda_i x} = \sum_k (p_k(x) * e^{a_k x}) * e^{\lambda_i x}$$
$$=_{b_k = a_k + \lambda_i} \sum_k \left(p_k(x) * e^{b_k x}\right)$$

and satisfies the form as well. The addition of 2 elements that satisfy the form, also satisfy the form.As a result, r_i satisfies the form. ∎

To prove that the calculation of r_i is stable it is enough to prove that the calculation of

$$\int \left(\sum_k (p_k(x) + e^{b_k x}) \right) dx$$

is stable, because r_i is a sum of 2 integrals of this form. Setting $x = 0$ in the function in the form $\sum_k \left(p_k\left(x\right) * e^{b_k x}\right)$ is stable because it is just the sum of the c_{0_k} elements and the sum of numbers is stable. Multiply of $\sum_k \left(p_k\left(x\right) * e^{a_k x}\right)$ by $e^{\lambda_i x}$ is stable as a sum of numbers as well.

Claim. *The calculation of* $\int \left(\sum_k \left(p_k\left(x\right) * e^{a_k x}\right)\right) dx$ *is stable.*

Proof of Claim. The error in calculation of sum of elements is equal to the sum of the errors in the elements it sums. In this case, the error is just a round off error and equals to 10^{-k}. Therefore, the sum is stable if there are less than 10^l to gain precision of $k - l$ significant digits. The calculation of $\frac{n!}{(n-m)!}$ Is equal to the calculation of $n\left(n-1\right)\left(n-m+1\right)$ and therefore stable as result of pure calculation of native numbers. The calculation of $\frac{(-1)^n c_b}{a_k^{n+1}}$ is stable because $\frac{c_b}{a_k^{n+1}}$. If $a_k = 0$, then the process does not calculate the sum and this action never takes place. Elsewhere $\frac{c_b}{a_k^{n+1}}$ can produce a big cancelation error; if $a_k^{n+1} \to 0$ with happens if $a_k \to 0$. If it does, then $e^{a_k} \to 1$ and it can be replaced by 1 to reduce the cancelation error to a round off error which is much smaller and produces a more accurate result. In any other case, $\frac{c_b}{a_k^{n+1}}$ is like multiplying 2 numbers which is stable.In conclusion, the calculation of $\int \left(\sum_k \left(p_k\left(x\right) * e^{a_k x}\right)\right) dx$ is stable. ∎

According to [3], matrix multiplication is stable. So each P_j for $j \in \{2, , n-1\}$ is stable as a sum and multiplication of matrices. Consider a number as a C^{1x1} matrix. Matrix multiplication is stable, so number multiplication is stable, too.

This algorithm converges as a result of the convergence of Putzer's algorithm [2].The difference between both algorithms is in the calculation of r_j, P_j. In the original algorithm, r_j, P_j are calculated using iteration, unlike this algorithm, in which they are calculated using recursion with start condition. The calculation method does not affect the convergence, because the result is the same from both processes. This means that if Putzer's algorithm converges then this algorithm converges too.

So, there is a stable solution for any given M matrix to the equation $Y' = MY, M \in C^{nXn}$. ∎

Note: Assume n is the dimension of the input square matrix.

Analysis of The Algorithm

Firstly, we want to check if the algorithm is completely satisfactory and how it fulfils the parameters of the numerical algorithm.

Generality: The algorithm does not need any other assumption on the matrix so it is the most general algorithm possible for this kind of task.

Reliability: The algorithm is stable and accurate as possible so it is always reliable.

Stability: The algorithm is stable for any given matrix.
The algorithm uses only available software for linear algebraic equations and for finding matrix eigenvalues software.All the methods the algorithm used in the process are available as quality software.

Storage requirements: We will check each step in the algorithm and its storage requirements and for then sum all the requirements together to gain the overall storage requirements of the algorithm.Firstly, the input is a square matrix so it requires n^2 places. For finding the eigenvalues of the matrix using Lanczos algorithm with QR decomposition, the computer needs to store n vectors of n values as v_i and the elements a_i, b_i for each i which satisfies $0 \leq i \leq n-1$. So, n vectors of n values is $2n^2$ elements [8]. Now, we need to store an array of square matricesP_i and array of functions which takes the form:

$$\sum p_i\left(t\right) * e^{a_i t}$$

when $p_i\left(t\right)$ are polynomials presented as arrays of n elements and $a_i \in C$. So, there are n matrices of n^2 elements and n functions presented as 2 arrays with n elements. But, it is possible to compute in each iteration the P_i matrix and multiply it with r_{i+1} and then adding it to the answer matrix. In this case, the computer stores 2 square matrices with storing function in each element. In total, the matrices store n^3 elements. In conclusion, it is easy to see that the storage requirements of the algorithm is $O\left(n^3\right)$.

Complexity: We will check each step in the algorithm and the complexity of its parts; then we will combine all the complexities together to gain the overall complexity of the algorithm. Firstly, we use Lanczos' algorithm to gain the eigenvalues of the matrix. Lanczos' algorithm calculates the T_{mm} matrix in $O\left(n^3\right)$ as a result of multiplicity of a matrix and a vector n times. When

T_{mm} matrix's eigenvalues can be calculated as low as $O(n^2)$ using spectral bisection. Then, we compute P_i and r_i for every $1 \leq i \leq n$. Computing P_i takes the same time as multiplying 2 matrices and subtraction 2 matrices; each iteration with takes $O(n^3)$. There are $(n-1)$ matrices which we need to be computed so the complexity is $O(n^4)$. For computing r_i the algorithm needs only to go over 2 arrays on n elements and change them. Assuming multiplication and addition of 2 numbers is $O(1)$, the computation of r_i will only take $O(n)$ for each r_i and there are n of them so $O(n^2)$.
In conclusion, the worst complexity in the whole process is $O(n^4)$ or the complexity of calculation of the multiplication of 2 matrices n times, and therefore the complexity of the whole algorithm is $O(n^4)$.

Accuracy: We will put an upper bound to the error. This happens as a result of k significant digits calculation and an additional error caused as a result of applying arithmetic functions on input with an error. In each step we assume the worst case scenario to ensure that we get an upper bound of the error in each step separately and of the whole algorithm combined.
First, the input is n^2 elements as a matrix so the error in the input is $n^2 * \varepsilon$ because every element is rounded off to k significant digits. Then, the process of finding the eigenvalues of the matrix is stable and accurate as desired [9]. So, we can assume that in the process produce the n eigenvalues of the input matrix that are produced are as accurate as possible. This means that the error in the eigenvalues vector is $n * \varepsilon$.
After this, we want to check what the error accrues from Putzer's algorithm.We want to go through several error analyses accruing in matrix algebraic actions.

Assumption: There are two $n \times n$ matrices A, B. Each matrix has an error presented as $\Delta A, \Delta B$. The error is also a $n \times n$ matrix too which satisfies

$$\|\Delta A\| \ll \|A\| \ .$$

Therefore, the actual matrices in the calculations are: $A + \Delta A, B + \Delta B$.

Multiplication:

According to [5] the upper bound for two matrices multiple's error is:

$$\|C_{comp} - C\| \leq \mu(n) * \varepsilon \|A\| \|B\| + O\left(\varepsilon^2\right) \ .$$

When $A * B = C$, the norm is an infinite norm, $\mu(n)$ is a polynomial of the size of matrices and ε is the machine precision 10^{-k} in our case. It is

known that $\mu(n)$ has the same size of $O(n^c)$ for moderate C.

<u>Note:</u> For $\varepsilon = 10^{-k} \to \varepsilon^2 = 10^{-2k}$ so $O(\varepsilon^2)$ does not effect on the calculation and can be ignored.
So, the total upper bound error is the sum of all the errors in the worst case scenario is: $AB_{error} \leq \mu(n) * \|A\|\|B\| * \varepsilon$

Then

$$(A + \Delta A) * (B + \Delta B)_{error}$$
$$\leq \varepsilon * \mu(n) * (\|A\|\|B\| + \|A\|\|\Delta B\| + \|\Delta A\|\|B\| + \|\Delta A\|\|\Delta B\|)$$

because

$$\|A\|\|\Delta B\| \leq \|A\|\|B\|, \|\Delta A\|\|B\| \leq \|A\|\|B\|$$

and

$$\|\Delta A\|\|\Delta B\| \leq \|A\|\|B\|.$$

A good upper bound will be:

$$(A + \Delta A) * (B + \Delta B)_{error} \leq 4\mu(n) * \varepsilon \|A\| \|B\|.$$

Addition:

The error happens when adding the follows 2 matrices:

$$(A + \Delta A) + (B + \Delta B) = A + B + \Delta B + \Delta A.$$

This means the error is just the sum of the errors.

Computation of the Error of r_i :
The error of computing r_i increases when i increases as a result of accumulation of errors in each interaction of computingr_i. For $i = 1 : e^{\lambda_i x}$ is the error of computing the power of 2 numbers, but both of them with round off error of epsilon machine, so the error of computing Pow(exp(1), $\lambda_i + \varepsilon$) is

$$(a + \Delta a)^{(b+\Delta b)} = (a + \Delta a)^{floor(b+\Delta b)} * (a + \Delta a)^{(b+\Delta b) - floor(b+\Delta b)}$$
$$\leq (a + \Delta a)^{floor(b+\Delta b)+1} = \sum_{k=0}^{floor(b+\Delta b)+1} \left(() \, a^{b+1-k} \Delta a^k\right).$$

and $\|\Delta A\|\|\Delta B\| \leq \|A\|\|B\|$. A good upper bound will be:

$$(A + \Delta A) * (B + \Delta B)_{error} \leq 4\mu(n) * \varepsilon \|A\| \|B\|.$$

Addition:

The error happens when adding the follows 2 matrices:

$$(A+\Delta A)+(B+\Delta B)=A+B+\Delta B+\Delta A.$$

This means the error is just the sum of the errors.

Computation of the Error of r_i :
The error of computing r_i increases when i increases as a result of accumulation of errors in each interaction of computingr_i. For $i=1: e^{\lambda_i x}$ is the error of computing the power of 2 numbers, but both of them with round off error of epsilon machine, so the error of computing Pow(exp(1), $\lambda_i+\varepsilon$) is

$$\begin{aligned}(a+\Delta a)^{(b+\Delta b)} &= (a+\Delta a)^{floor(b+\Delta b)} * (a+\Delta a)^{(b+\Delta b)-floor(b+\Delta b)} \\ &\leq (a+\Delta a)^{floor(b+\Delta b)+1} = \sum_{k=0}^{floor(b+\Delta b)+1} \left(() \, a^{b+1-k} \Delta a^k\right).\end{aligned}$$

Mark $floor\,(b+\Delta b)$ as B.
Notice $\frac{\Delta a}{a}=\varepsilon$ so $\Delta a=\varepsilon a$ so the equation produces results in:

$$\sum_{k=0}^{B+1}\left(\binom{B+1}{k}\right) * a^{B+1} * \varepsilon^{B+1} = a^{B+1} * 2^{B+1} * \varepsilon^{B+1} = (2\varepsilon a)^{B+1} = (2\Delta a)^{B+1}.$$

Therefore, the error is

$$|\,(2\varepsilon a)^{B+1} - a^b| \leq |\,(2\varepsilon a)^{B+1} - a^{B+1}| = |\,((2\varepsilon-1)\,a)^{B+1}\,|.$$

The calculation of $e^{floor(\lambda_i+\epsilon)}$ is done using recursion and $e^{[(\lambda_i+\epsilon)-floor(\lambda_i+\epsilon)]}$ is done using Taylor series which converse when $|x| \leq 1$ and $|\,(\lambda_i+\epsilon) - floor\,(\lambda_i+\epsilon)\,| \leq 1$ by definition. For

$$i \geq 2 : r_j = \left(\int \left(e^{-\lambda_j x} r_{j-1}\,(x)\right) dx - \rangle \int \left(r_{j-1}\,(x)\right) dx|_{x=0}\right) * e^{\lambda_j x},$$

the error of the computation of $\int \left(r_{j-1}\,(x)\right) dx$ equals to the error of

$$\sum_k \sum_b \left(\frac{(-1)^n\, c_b}{a_k^{n+1}} * \sum_{m=0}^{n}\left(\frac{n!}{(n-m)!} * (-bx)^{n-m}\right)\right),$$

as a result of r_i form.

$\frac{n!}{(n-m)!} \in N$, so it is effected only by round off error. $(-b)^{n-m}$ has the same error as power 2 floating point numbers - $|\left((2\varepsilon-1)\,b\right)^{n-m}|$. Computing a_k^{n+1} has the same error: $|\left((2\varepsilon-1)\,a_k\right)^{n+1}|$ and $c_b * a_k^{-n-1}$ is just the error of multiplying 2 numbers and is: $|\Delta c_b * a_k^{-n-1} + \Delta a_k^{-n-1} * c_b + \Delta c_b \Delta a_k^{-n-1}|$. The error of the sum of elements is just the sum of the errors. It is enough to find the error in one iteration:

$$\begin{aligned}
&\left[\sum_{m=0}^{n}\left(\frac{n!}{(n-m)!} * (-bx)^{n-m}\right)\right]_{error} \\
&\quad \leq n * \left(\frac{n!}{\left(n-\frac{n}{2}\right)!} * |\left((2\varepsilon-1)\,b\right)^{\frac{n}{2}}| + \varepsilon * (-b)^{\frac{n}{2}} + \varepsilon * |\left((2\varepsilon-1)\,b\right)^{\frac{n}{2}}|\right) \\
&\quad = n\left[\left(\lfloor\frac{n}{2}\rfloor! + 1\right)(2\varepsilon-1)^{\frac{n}{2}} + \varepsilon\right](b)^{\frac{n}{2}}.
\end{aligned}$$

This error with the error accrues

$$\left[\frac{(-1)^n c_b}{a_k^{n+1}}\right]_{error} = |\varepsilon * a_k^{-n-1} + \left((2\varepsilon-1)\,a_k\right)^{n+1} * c_b + \varepsilon\left((2\varepsilon-1)\,a_k\right)^{n+1}| =_{a_k>1}.$$

$$|\left((2\varepsilon-1)\,a_k\right)^{n+1} * c_b + \varepsilon\left((2\varepsilon-1)\,a_k\right)^{n+1}| = |\left(c_b+\varepsilon\right)\left((2\varepsilon-1)\,a_k\right)^{n+1}|.$$

So:

$$\begin{aligned}
&\left|(c_b+\varepsilon)((2\varepsilon-1)\ a_k)^{n+1}\right| * \sum_{m=0}^{n}\left(\frac{n!}{(n-m)!} * (-bx)^{n-m}\right) \\
&\quad + \frac{(-1)^n c_b}{a_k^{n+1}} * n\left[\left(\left\lfloor\frac{n}{2}\right\rfloor! + 1\right)(2\varepsilon-1)^{\frac{n}{2}} + \varepsilon\right](b)^{\frac{n}{2}} \\
&\quad + n\left[\left(\left\lfloor\frac{n}{2}\right\rfloor! + 1\right)(2\varepsilon-1)^{\frac{n}{2}} + \varepsilon\right](b)^{\frac{n}{2}} * \left|(c_b+\varepsilon)((2\varepsilon-1)\ a_k)^{n+1}\right|.
\end{aligned}$$

<u>Mark</u> this as "Err". So, the final upper bound to the error is: $K*B*Err$, when K, B are the number of elements in $\sum_k, \sum_b$, respectively.

The error that accrues in computing $\int\left(e^{-\lambda_j x} r_{j-1}(x)\right)dx$ is approximately the same as $\int\left(r_{j-1}(x)\right)dx$ as result of the form of r_j.

Multiplying

$$\left(\int\left(e^{-\lambda_j x} r_{j-1}(x)\right)dx - \rangle \int\left(r_{j-1}(x)\right)dx|_{x=0}\right)$$

with $e^{\lambda_j x}$ adds only a round off error as a result of the form used to present r_i.

Each iteration of the Putzer's algorithm is:

$$(r_i + \Delta r_i)\left[(A + \Delta A) * (B + \Delta B - \Delta\lambda)\right] \ .$$

The error of each iteration of the Putzer's algorithm is:

$$\left[(r + \Delta r)\left[(A + \Delta A) * (B + \Delta B)\right]\right]_{\text{error}} \leq K * B * Err * \|A\| \, \|B\| \ .$$

The error of computing the result of the algorithm is the sum of the error of all the iterations and is bounded by:

$$\left[\sum_{i=1}^{size(M)} (r + \Delta r)\left[(A + \Delta A) * (B + \Delta B)\right]\right]_{error} \leq size\,(M) * K * B * Err * \|P_{size(m)-1}\|.$$

Comparison with other algorithms: In comparison with other algorithms, this algorithm's storage requirements and complexity are similar to other algorithms. The accuracy of the algorithm is not better in practice relative to other algorithms [5]. We want to compare the results of our algorithm with other algorithms. The main objective would be to check the "general" matrix and compare the results from each algorithm, but no such "general" matrix exists. Despite this, we can look at 8 different cases which make the approximation poor and examine them and look at this as a representative sample for the ability of an algorithm to manage "bad" matrices. We will examine 3 matrices for each case in the sizes of $2 \times 2, 3 \times 3, 7 \times 7$. The cases of $2 \times 2, 3 \times 3$ will allow as to see the error in small matrices and the 7×7 will determinate the growth of the error when the size of the matrix grows.

We have the following cases:

1. The difference between the eigenvalues of the matrix is small but not negligible.
2. The eigenvalues are approaching 0.
3. The diameter of the matrix is large.
4. There are eigenvalues with big algebraic multiplicity.
5. The condition number of the matrix is large.
6. There is a single eigenvalue.
7. The eigenvalues are complex with big imaginary part.
8. The eigenvalues are complex with big imaginary part with big algebraic multiplicity.

For the inputted matrices for each case:

1. We will pick a value (a) and an amplitude (ε) and generate eigenvaluesin the spectrum$(a \pm \varepsilon)$.
2. We will generate eigenvalues by the next formula: $1 \leq i \leq n, \lambda_i = \frac{1}{(i+2)^2}$
3. We will pick two real values (a, b) and generate the eigenvalues by the following formula:$1 \leq i \leq n, \lambda_i = a - \frac{(a-b)i}{n}$
4. We will generate randomly values that satisfy:$a_i > \frac{n}{4}, b < \frac{n}{4}, \sum (a_i) + b = n$. For each a_i generate different value with algebraic multiplicity of a_i.
5. We will pick two values (a, b) when $\frac{|a|}{|b|}$ is big. The other eigenvalues we generate randomly and make sure they are not close one to the other (or equal) and in (a, b)
6. We will pick a single value(a).
7. We will pick a different real random valuesa_i and build the fit eigenvalue as $a_i + 10 * a_i^2 i$.
8. We will pick two complex values $(a + bi, c + di)$ and generate the eigenvalues by the following formula:$1 \leq k \leq n, \lambda_k = a - \frac{(a+bi-c-di)k}{n}$. Each case is multiplied by random matrix with small condition number.
   n is the size of the matrix.

Case 1:

$2 \times 2:$

$$A = \begin{pmatrix} 1.998 & 0 \\ 0 & 2.001 \end{pmatrix}, Q = \begin{pmatrix} -1 & 1.2 \\ 1.1 & 1 \end{pmatrix} \rightarrow Q^{-1}AQ = \begin{pmatrix} 1.997 & 0.0016 \\ 0.0014 & 1.993 \end{pmatrix}$$

$3 \times 3:$

$$A = \begin{pmatrix} 3.511 & 0 & 0 \\ 0 & 3.502 & 0 \\ 0 & 0 & 3.491 \end{pmatrix},$$

$$Q = \begin{pmatrix} 1 & 4.1 & 0.5 \\ -3 & 1 & -2 \\ -1.1 & 0.2 & 1 \end{pmatrix} \rightarrow Q^{-1}AQ = \begin{pmatrix} 3.498 & 0.0031 & 0.0044 \\ 0.0023 & 3.5103 & 0.0008 \\ 0.0072 & -0.005 & 3.4957 \end{pmatrix}$$

$7 \times 7:$

$$A = matrix\left([1.0815, 0.9094, 1.0127, 1.0913, 0.9368, 1.0278, 0.9902]\right),$$

$$Q = magic\,(7) \rightarrow Q^{-1}AQ = inv(magic(7)) * A * magic(7)$$

Case 2:

$2 \times 2:$

$$A = \begin{pmatrix} 0.1111 & 0 \\ 0 & 0.0625 \end{pmatrix},$$

$$Q = \begin{pmatrix} -1 & 1.2 \\ 1.1 & 1 \end{pmatrix} \to Q^{-1}AQ = \begin{pmatrix} 0.0834 & -0.0251 \\ -0.0230 & 0.9020 \end{pmatrix}$$

$3 \times 3:$

$$A = \begin{pmatrix} 0.1111 & 0 & 0 \\ 0 & 0.0625 & 0 \\ 0 & 0 & 0.04 \end{pmatrix},$$

$$Q = \begin{pmatrix} 1 & 4.1 & 0.5 \\ -3 & 1 & -2 \\ -1.1 & 0.2 & 1 \end{pmatrix} \to Q^{-1}AQ = \begin{pmatrix} 0.0561 & 0.0138 & 0.0100 \\ 0.0115 & 0.1075 & 0.0050 \\ 0.0154 & 0.0017 & 0.0500 \end{pmatrix}$$

$7 \times 7:$

$$A = matrix\left([0.1111, 0.0625, 0.04, 0.0277, 0.0204, 0.0156, 0.0123]\right),$$

$$Q = magic\,(7) \ \to Q^{-1}AQ = inv(magic(7)) * A * magic(7)$$

Case 3:

$2 \times 2:$

$$A = \begin{pmatrix} 100000 & 0 \\ 0 & -100000 \end{pmatrix},$$

$$Q = \begin{pmatrix} -1 & 1.2 \\ 1.1 & 1 \end{pmatrix} \to Q^{-1}AQ = 10^5 \begin{pmatrix} -0.1379 & -1.0345 \\ -0.9483 & 0.1379 \end{pmatrix}$$

$3 \times 3:$

$$A = \begin{pmatrix} 100000 & 0 & 0 \\ 0 & 5050 & 0 \\ 0 & 0 & 100 \end{pmatrix},$$

$$Q = \begin{pmatrix} 1 & 4.1 & 0.5 \\ -3 & 1 & -2 \\ -1.1 & 0.2 & 1 \end{pmatrix} \to Q^{-1}AQ = 10^3 \begin{pmatrix} 3.2894 & 1.6119 & 2.0257 \\ 1.2391 & 9.6229 & 0.4525 \\ 3.2605 & -0.1315 & 2.2377 \end{pmatrix}$$

$7 \times 7:$

$$A = matrix\left(10^5\left[10, 8.5, 7, 5.5, 4, 2.5, 1\right]\right),$$

$$Q = magic\,(7)\ \rightarrow Q^{-1}AQ = inv(magic(7)) * A * magic(7)$$

Case 4:

$2 \times 2:$

$$A = \begin{pmatrix} 3 & 0 \\ 0 & 3 \end{pmatrix},\ Q = \begin{pmatrix} -1 & 1.2 \\ 1.1 & 1 \end{pmatrix} \rightarrow Q^{-1}AQ = \begin{pmatrix} 3 & 0 \\ 0 & 3 \end{pmatrix}$$

$3 \times 3:$

$$A = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & -1 \end{pmatrix},$$

$$Q = \begin{pmatrix} 1 & 4.1 & 0.5 \\ -3 & 1 & -2 \\ -1.1 & 0.2 & 1 \end{pmatrix} \rightarrow Q^{-1}AQ = \begin{pmatrix} 0.7501 & 0.2273 & 1.1363 \\ 0.0718 & 1.9869 & -0.0653 \\ 1.9108 & -0.3474 & 0.2630 \end{pmatrix}$$

$7 \times 7:$

$$A = matrix\left([3, 3, 3, 3, -2, -2, -2]\right),$$

$$Q = magic\,(7)\ \rightarrow Q^{-1}AQ = inv(magic(7)) * A * magic(7)$$

Case 5:

$2 \times 2:$

$$A = \begin{pmatrix} 1000 & 0 \\ 0 & 1 \end{pmatrix},$$

$$Q = \begin{pmatrix} -1 & 1.2 \\ 1.1 & 1 \end{pmatrix} \rightarrow Q^{-1}AQ = \begin{pmatrix} 431.6034 & -516.7241 \\ -473.6638 & 569.3966 \end{pmatrix}$$

$3 \times 3:$

$$A = \begin{pmatrix} 3500 & 0 & 0 \\ 0 & 855 & 0 \\ 0 & 0 & -20 \end{pmatrix},$$

$$Q = \begin{pmatrix} 1 & 4.1 & 0.5 \\ -3 & 1 & -2 \\ -1.1 & 0.2 & 1 \end{pmatrix} \to Q^{-1}AQ = 10^3 \begin{pmatrix} 0.6517 & 0.7272 & 0.4120 \\ 0.6197 & 3.3062 & 0.2803 \\ 0.6149 & 0.1347 & 0.3771 \end{pmatrix}$$

$7 \times 7:$

$$A = matrix\left([86420, -41, 59783, 3212, 2, -2, -77]\right),$$
$$Q = magic\,(7) \ \to Q^{-1}AQ = inv(magic(7)) * A * magic(7)$$

Case 6:

$2 \times 2:$

$$A = \begin{pmatrix} 1.5 & 0 \\ 0 & 1.5 \end{pmatrix}, \quad Q = \begin{pmatrix} -1 & 1.2 \\ 1.1 & 1 \end{pmatrix} \to Q^{-1}AQ = \begin{pmatrix} 1.5 & 0 \\ 0 & 1.5 \end{pmatrix}$$

$3 \times 3:$

$$A = \begin{pmatrix} 3.5 & 0 & 0 \\ 0 & 3.5 & 0 \\ 0 & 0 & 3.5 \end{pmatrix},$$

$$Q = \begin{pmatrix} 1 & 4.1 & 0.5 \\ -3 & 1 & -2 \\ -1.1 & 0.2 & 1 \end{pmatrix} \to Q^{-1}AQ = \begin{pmatrix} 3.4999 & 0 & 0 \\ 0 & 3.5 & 0 \\ 0 & 0 & 3.5 \end{pmatrix}$$

$7 \times 7:$

$$A = matrix\left([-1.1, -1.1, -1.1, -1.1, -1.1, -1.1, -1.1]\right),$$
$$Q = magic\,(7) \ \to Q^{-1}AQ = inv(magic(7)) * A * magic(7)$$

Case 7:

$2 \times 2:$

$$A = \begin{pmatrix} 3+90i & 0 \\ 0 & 4-160i \end{pmatrix},$$

$$Q = \begin{pmatrix} -1 & 1.2 \\ 1.1 & 1 \end{pmatrix} \rightarrow Q^{-1}AQ = \begin{pmatrix} 3.5689-52.2413i & 0.5172-129.3103i \\ 0.4741-118.5344i & 3.4310-17.7586i \end{pmatrix}$$

$3 \times 3:$

$$A = \begin{pmatrix} 2-40i & 0 & 0 \\ 0 & -1.5+22.5i & 0 \\ 0 & 0 & 1+10i \end{pmatrix},$$

$$Q = \begin{pmatrix} 1 & 4.1 & 0.5 \\ -3 & 1 & -2 \\ -1.1 & 0.2 & 1 \end{pmatrix} \rightarrow$$

$$Q^{-1}AQ = \begin{pmatrix} -0.2451+13.4828i & 0.6852-14.6713i & -0.8402+2.8298i \\ 0.7325-13.8496i & 1.7595-35.5649i & 0.4506-7.3465i \\ -1.5161+6.6010i & 0.6019-7.0255i & -0.0144+14.5821i \end{pmatrix}$$

$7 \times 7:$

$$A = matrix\left([1-10i, 2+40i, 3-90i, 4+160i, -5-250i, -2+40i, -5+250i]\right),$$

$$Q = magic\,(7) \ \rightarrow Q^{-1}AQ = inv(magic(7)) * A * magic(7)$$

Case 8:

$2 \times 2:$

$$A = \begin{pmatrix} 3+i & 0 \\ 0 & 3+i \end{pmatrix}, Q = \begin{pmatrix} -1 & 1.2 \\ 1.1 & 1 \end{pmatrix} \rightarrow Q^{-1}AQ = \begin{pmatrix} 3+i & 0 \\ 0 & 3+i \end{pmatrix}$$

$3 \times 3:$

$$A = \begin{pmatrix} 2-2i & 0 & 0 \\ 0 & 2-2i & 0 \\ 0 & 0 & 2-2i \end{pmatrix},$$

$$Q = \begin{pmatrix} 1 & 4.1 & 0.5 \\ -3 & 1 & -2 \\ -1.1 & 0.2 & 1 \end{pmatrix}$$

$$\rightarrow Q^{-1}AQ = \begin{pmatrix} 0.7501 - 0.7501i & 0.2273 - 0.2273i & 1.1363 - 1.1363i \\ 0.0718 - 0.0718i & 1.9869 - 1.9869i & -0.0653 + 0.0653i \\ 1.9108 - 1.9108i & -0.3474 + 0.3474i & 0.2630 - 0.2630i \end{pmatrix}$$

7×7 :

$$A = matrix\left([3-6i, 3-6i, 3-6i, 3-6i, -2+5i, -2+5i, -2+5i]\right),$$

$$Q = magic\,(7) \;\rightarrow Q^{-1}AQ = inv(magic(7)) * A * magic(7)$$

Compare for each case the results for the most useable algorithms.

2×2	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8
Nave	0.01	1.09	Error	O(10^{-14})	Error	O()	O()	O()
Pade	9.34	3.42	Error	23.19	Error	6.03	35.92	155.98
Newton	5.38	2.38	Error	17.08	Error	2.98	182.74	143.41
Lagrange	0.01	1.09	Error	20.08	Error	4.48	78.46	148.41
Stable	0.07	0.98	Error	$O(10^{-15})$	Error	O()	133.81	O()

3×3	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8
Nave	1.63	0.10	Error	65.59	Error	O()	O()	5.47
Pade	37.37	1.88	Error	9.49	Error	37.01	7.94	8.92
Newton	29.97	1.07	Error	8.47	Error	29.61	45.30	7.13
Lagrange	0.45	0.05	Error	5.27	Error	O()	8.17	5.55
Stable	0.32	0.65	Error	2.37	Error	O()	12.05	13.06

7×7	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8
Nave	1.63	0.10	Error	65.59	Error	O()	O()	72.58
Pade	4.67	1.84	Error	48.61	Error	0.76	168.17	58.79
Newton	2.58	1.02	Error	47.88	Error	1.43	319.22	57.56
Lagrange	1.637	0.10	Error	O()	Error	3.42	193.18	O()
Stable	1.42	0.06	Error	38.19	Error	O()	541.01	112.90

It is easy to see that the Stable algorithm provides a better approximation than other algorithms for the "bad" matrices most of the time.

Therefore, since the approximate time and memory requirements needed by the stable algorithm is similar to the other algorithms together with the better approximation, the stable algorithm can replace the other algorithms.

In cases 3 and 5, all the algorithms return error. The explanation to this phenomena is that according to Lanczos [6] if the eigenvalues satisfy $\sqrt[2]{\frac{\lambda_n}{\lambda_1}} > 100$ then the approximation of the any action of the matrix would be at least 100 % absolute error from the analytical calculation.

Note: When the eigenvalues of the matrix is close but not enough to treat them like an equal it can result in a big cancelation error and produces a poor estimation to the result relative to the estimation that the algorithm provides for other cases.

In addition, we would like to determinate what happens to the error when the problem is high dimensional and the matrices are neither diagonal nor diagonalizable. This test will present what happen to the algorithms when the size of "bad" matrices goes to infinity.

Looking at a matrix of the size 32×32 that is generated using multiplication of a random matrix from the same size with norm bounded by 2 and with full rank denoted 'A ' with $det\,(A) \neq 0$ by a triagonal matrix with real random values denoted 'Q '. When Q satisfies: $\forall \lambda \in \Lambda\,[Q] : Im\,(\lambda) = 0 |\lambda| < 1$

	Nave	Pade	Lagrange	Stable
Case 1	3.1961e+003	235.1477	3.9190e+017	178.2877
Case 2	1.5316e+004	961.0686	5.9541e+018	883.5046
Case 3	4.1077e+003	229.6978	3.1003e+016	248.5912
Case 4	1.1983e+004	597.1864	6.1519e+018	511.5380
Case 5	9.5877e+003	277.7750	6.6513e+019	276.6505
Avg.	8.8382e+003	460.1751	1.5808e+019	419.7144

As presented in the table the most stable algorithm is the "stable" algorithm. The error for all algorithms increases as the size of the matrix increases, but the grown of the error is small compared to the growth of the matrix and therefore the use of the algorithms is still reliable up to some point. Some algorithms like Lagrange's algorithm is more sensitive to the size of the matrix and others like Pade's algorithm is less sensitive and the error increases slowly.

4. Applications. Some of the most common applications of matrix exponentiation in real life originate in engineering. Many types of engineering deal with systems with a number of elements in them. These systems are often described using vectors and matrices. In every model of such system which wishes to describe the system using its differential, matrix exponentiation will be used in the process. For example, consider the following continuoustime system:

$$\begin{cases} \frac{dx}{dt} = Fx\left(t\right) + Gu\left(t\right) \\ y = Hx\left(t\right) + Ju(t) \end{cases} .$$

We wish to transfer it to a discretetime, statespace system. The system takes the form:

$$\begin{cases} x_{k+1} = Ax_k + Bu_k \\ y_k = HX_k + Ju_k \end{cases}$$

when

$$\mathbf{A}=\mathbf{e}^{\mathbf{F}\xi} \; and \; B = \left(\int_o^{\xi} \mathrm{e}^{\mathrm{Ft}} \; dt\right) * G$$

where ξ is the sampling period.
In linear control system theory, matrix exponentiation is used very often.Consider a DiscreteContinuous Linear Time Interval System (DCLTI for short).This kind of system takes the form:

$$\begin{gathered} \dot{X}\left(t\right) = A_c\left(t\right)X\left(t\right) + B_c\left(t\right)u_c\left(t\right) \;\; , \\ t_k \leq t \leq t_{k+1} X\left(t_k\right) = A_d\left(t\right)x\left(t_k - 0\right) + B_d\left(t\right)u_d\left(t_k\right) \;\; , \\ t_k \in \theta \quad \text{where} \quad \theta = \left\{t_k = kh, \; k = 0, \; 1, \; \ldots, \; h = \text{const}\right\}. \end{gathered}$$

5. Conclusions. For research and applications which are used as a part of the bigger complex process, this algorithm is appropriate, because there are time and resources for this heavy process that ensures the stability, in order to be sure that the outcome is reliable.

If some property of the input matrix is known, it is better to use a special algorithm which uses this kind of property from the algorithm list described earlier in this article.

This algorithm and any numerical algorithm is exposed to roundoff and cancellation error caused by using floating point or k significant digits calculations.

Additional research is required for a better understanding of the error accruing in the calculation of the matrix exponent using numerical algorithms and how it is possible to decrease it for this and other known algorithms.

REFERENCES

[1] R. M. Karp, *Reducibility among combinatorial problems*, Complexity of Computer Computations, 1972, pp. 85-103.

[2] Erik Wahln, *Alternative proof of Putzer's algorithm*, 2013, `http://www.ctr.maths.lu.se/media/MATM14/2013vt2013/putzer.pdf`].

[3] James Demmel, Ioana Dumitriu, Olga Holtz, and Robert Kleinberg, *Fast matrix multiplication is stable*, 2006, *https://www.math.washington.edu/ dumitriu/fmm arxiv.pdf*.

[4] Grigory Agranovich, *Observer for Discrete-Continuous LTI with continuous-time measurements*, Ariel University, 2010.

[5] Cleve Moler, Charles Van Loan, *Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later*, Society for Industrial and Applied Mathematics, 2006.

[6] C. Lanczos, An iteration method for the solution of the eigenvalue problem of linear differential and integral operators, *J. Res. Nat'l Bur. Std. 45*, 1950, 225–282.

[7] Ojalvo, I.U. and Newman, M., Vibration modes of large structures by an automatic matrix-reduction method, *AIAA J., 8 (7)*, 1970 1234-1239.

[8] Trefethen, Lloyd N., Bau, David, Numerical linear algebra, *Philadelphia: Society for Industrial and Applied Mathematics* , ISBN 978-0-89871-361-9, 1997.

[9] Stephen T. Barnard, Horst D. Simon, Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems, DOI: 10.1002/cpe.4330060203, 1994.