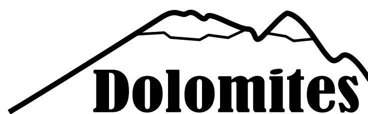
PADOVA UNIVERSITY PRESS


Dolomites Research Notes on Approximation

Volume 12 · 2019 · Pages 28–42

Fast and accurate computation of divided differences for analytic functions, with an application to the exponential function

Franco Zivcovich[a]

Communicated by M. Caliari

Abstract

In this work we analyze in depth the structure of the matrix representing the operator mapping the coefficients of a polynomial $p(x)$ in the monomial basis $\mathcal{M}$ into those in the Newton basis $\mathcal{N}$. A factorization in minimal terms of said matrix has been obtained and as a consequence a factorization of its inverse is available too. As a result, a new high performances routine for the computation of the divided differences of the exponential and the closely related ϕ_l functions has been produced with satisfying results.

1 Introduction

Given a sequence of $n+1$ interpolation points $z=(z_0, z_1, \dots, z_n)$ and the corresponding function values $f(z_0), f(z_1), \dots, f(z_n)$ there exists a unique polynomial $p(x)$ of degree n interpolating $f(x)$ in the Hermite sense at z. There are many polynomial basis under which $p(x)$ can be written, in particular we are going to focus on the monomial and the Newton basis. The *monomial basis* $\mathcal{M}=\{1, x, \dots, x^n\}$ is such that the $n+1$ scalars $a_0, a_1, \dots, a_n$ identify $p(x)$ in $\mathcal{M}$ as

$$p(x)=\sum_{k=0}^{n} a_k x^k$$

or, in vector form, as

$$p(x)=\boldsymbol{m}(x,n)^T \mathbf{a}(n)$$

where $\boldsymbol{m}(x,n)=(1, x, \dots, x^n)^T$ and $\mathbf{a}(n)$ is the vector having on its kth component the coefficient a_k. The *Newton basis* $\mathcal{N}=\{\pi_{0,z}(x), \pi_{1,z}(x), \dots, \pi_{n,z}(x)\}$ where

$$\pi_{0,z}(x)\equiv 1, \quad \pi_{k,z}(x)=\prod_{j=0}^{k-1}(x-z_k)$$

is such that the $n+1$ scalars $d[z_0], d[z_0,z_1], \dots, d[z_0,z_1,\dots,z_n]$, called divided differences, identify $p(x)$ in $\mathcal{N}$ as

$$p(x)=\sum_{k=0}^{n} d[z_0,z_1,\dots,z_k]\pi_{k,z}(x)$$

or, in vector form, as

$$p(x)=\boldsymbol{\pi}(x,n)^T \mathbf{d}(n)$$

where $\boldsymbol{\pi}(x,n)=(\pi_{0,z}(x), \pi_{1,z}(x), \dots, \pi_{n,z}(x))^T$ and $\mathbf{d}(n)$ is the vector having on its kth component the coefficient $d[z_0,z_1,\dots,z_k]$. In this work we are going to analyze in depth the structure of the matrix representing the operator that maps $\mathbf{a}(n)$ into $\mathbf{d}(n)$ and its inverse.

As a result we will come up with several simple algorithms for switching between $\mathbf{a}(n)$ and $\mathbf{d}(n)$ each one of them showing different numerical and analytic properties. In particular one of these algorithms fits especially well the task of computing the table of divided differences of a given function $f(x)$, i.e. the matrix having in the $n-k+1$ entries of its $(k+1)$st column the divided differences

$$d[z_k], d[z_k,z_{k+1}], \dots, d[z_k,z_{k+1},\dots,z_n]$$

of the function $f(x)$. Thanks to this feature together with the properties of the exponential function, we will show how to build a new routine for the fast and accurate computation of the divided differences of the exponential function and of the closely related ϕ_l functions. Each one of the algorithms we will report is meant to work as it is in Matlab.

[a]Department of Mathematics, University of Trento, Italy

2 The change of basis operator

In order to easily define the matrix representing the change of basis operator and its inverse we need to introduce two important kinds of symmetric polynomials: the complete homogeneous symmetric polynomials and the elementary symmetric polynomials. We recall that a symmetric polynomial is a polynomial such that if any of the variables are interchanged the polynomial remains unchanged.

The *complete homogeneous symmetric polynomial* of degree $k-j$ over the variables $z_0, z_1, \ldots, z_j$, that is:

$$h_{k-j}(z_0, z_1, \ldots, z_j) = \sum_{0 \le i_1 \le \ldots \le i_{k-j} \le j} z_{i_1} \cdots z_{i_{k-j}}$$

with $h_{k-j}(z_0, z_1, \ldots, z_j)$ equal to 0 if $k-j<0$ and to 1 if $k-j=0$. In other words, $h_{k-j}(z_0, z_1, \ldots, z_j)$ is the sum of all monomials of total degree $k-j$ in the variables.

The *elementary symmetric polynomial* of degree $k-j$ over the variables $z_0, z_1, \ldots, z_{k-1}$, that is:

$$e_{k-j}(z_0, z_1, \ldots, z_{k-1}) = \sum_{0 \le i_1 < \cdots < i_{k-j} \le k-1} z_{i_1} \cdots z_{i_{k-j}}$$

with $e_{k-j}(z_0, z_1, \ldots, z_{k-1})$ equal to 0 if $k-j<0$ and to 1 if $k-j=0$. In other words, $e_{k-j}(z_0, z_1, \ldots, z_{k-1})$ is the sum of all monomials with $k-j$ factors and of total degree $k-j$ in the variables.

In the literature (see [6], Theorem p. 776) it has been analytically derived that the matrix $\mathbf{H}(z_0, z_1, \ldots, z_n)$ representing the operator that maps $\mathbf{a}(n)$ into $\mathbf{d}(n)$, i.e.:

$$\mathbf{d}(n) = \mathbf{H}(z_0, z_1, \ldots, z_n)\mathbf{a}(n)$$

is defined by means of complete homogeneous symmetric polynomials as the matrix having

$$h_{k-j}(z_0, z_1, \ldots, z_j)$$

in its $(j+1, k+1)$ position. Since the complete homogeneous symmetric polynomials are such that $h_{k-j}(z_0, z_1, \ldots, z_j) = 0$ if $k-j<0$ and $h_{k-j}(z_0, z_1, \ldots, z_j) = 1$ if $k-j=0$ we have that $\mathbf{H}(z_0, z_1, \ldots, z_n)$ is an upper triangular matrix with all ones on the main diagonal independently from the instance of the sequence $z = (z_0, z_1, \ldots, z_n)$. Therefore we know that $\mathbf{H}(z_0, z_1, \ldots, z_n)$ is invertible with inverse $\mathbf{E}(z_0, z_1, \ldots, z_n) := \mathbf{H}(z_0, z_1, \ldots, z_n)^{-1}$. As an example we show in the following the matrix $\mathbf{H}(z_0, z_1, z_2, z_3)$ mapping $\mathbf{a}(3)$ into $\mathbf{d}(3)$:

$$\mathbf{H}(z_0, z_1, z_2, z_3) = \begin{pmatrix} 1 & z_0 & z_0^2 & z_0^3 \\ 0 & 1 & z_0 + z_1 & z_0^2 + z_0 z_1 + z_1^2 \\ 0 & 0 & 1 & z_0 + z_1 + z_2 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Notice that z_3 does not appear in $\mathbf{H}(z_0, z_1, z_2, z_3)$.

The next step is to determine analytically the inverse $\mathbf{E}(z_0, z_1, \ldots, z_n)$. To do so we expand the Newton basis polynomials $\pi_{k,z}(x)$ in their explicit form applying the so called Vieta's formulas expressing the coefficients of a polynomial with respect to the monomial basis as symmetric functions of its roots. Briefly such formulas tell us that the coefficient of the jth power of the kth Newton basis polynomial is such that

$$\mathrm{coef}(j, \pi_{k,z}(x)) = (-1)^{k-j} e_{k-j}(z_0, z_1, \ldots, z_{k-1})$$

and hence we can rewrite the kth Newton basis polynomial $\pi_{k,z}(x)$ in the form

$$\pi_{k,z}(x) = \sum_{j=0}^{k} (-1)^{k-j} e_{k-j}(z_0, z_1, \ldots, z_{k-1}) x^j$$

From here it follows that the matrix $\mathbf{E}(z_0, z_1, \ldots, z_n)$ representing the operator that maps $\mathbf{d}(n)$ into $\mathbf{a}(n)$is defined by means of elementary symmetric polynomials as the matrix having

$$(-1)^{k-j} e_{k-j}(z_0, z_1, \ldots, z_{k-1})$$

in its $(j+1, k+1)$ position. In fact for such $\mathbf{E}(z_0, z_1, \ldots, z_n)$ we have that

$$\boldsymbol{\pi}(x, n)^T = \boldsymbol{m}(x, n)^T \mathbf{E}(z_0, z_1, \ldots, z_n)$$

and hence from

$$p(x) = \boldsymbol{m}(x, n)^T \mathbf{E}(z_0, z_1, \ldots, z_n)\mathbf{d}(n)$$

we can derive by comparison the identity:

$$\mathbf{a}(n) = \mathbf{E}(z_0, z_1, \ldots, z_n)\mathbf{d}(n).$$

As an example we show in the following the matrix $\mathbf{E}(z_0, z_1, z_2, z_3)$ mapping $\mathbf{d}(3)$ into $\mathbf{a}(3)$:

$$\mathbf{E}(z_0, z_1, z_2, z_3) = \begin{pmatrix} 1 & -z_0 & z_0 z_1 & -z_0 z_1 z_2 \\ 0 & 1 & -z_0 - z_1 & z_0 z_1 + z_0 z_2 + z_1 z_2 \\ 0 & 0 & 1 & -z_0 - z_1 - z_2 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Notice that z_3 does not appear in $\mathbf{E}(z_0,z_1,z_2,z_3)$.

We conclude this section by writing two Matlab routines that can be used to build the matrices $\mathbf{H}(z_0,z_1,\ldots,z_n)$ and $\mathbf{E}(z_0,z_1,\ldots,z_n)$ but first we need to give some recurrence properties of the complete homogeneous symmetric polynomials and the elementary symmetric polynomials.

Claim 1. *The complete homogeneous symmetric polynomial over $z_0,z_1,\ldots,z_j$ of degree $k-j$ for any $p\in\{0,1,\ldots,j\}$ can be decomposed as follows:*

$$\begin{aligned} h_{k-j}(z_0,z_1,\ldots,z_j) = & h_{k-j}(z_0,z_1,\ldots,z_{p-1},z_{p+1},\ldots,z_j) \\ & + z_p h_{k-j-1}(z_0,z_1,\ldots,z_j). \end{aligned} \tag{1}$$

Proof. The claim follows if we consider the alternative representation of the complete homogeneous symmetric polynomials

$$h_{k-j}(z_0,z_1,\ldots,z_j) = \sum_{i_0+i_1+\ldots+i_j=k-j} z_0^{i_0} z_1^{i_1}\cdots z_j^{i_j}$$

where $i_0,i_1,\ldots,i_j$ are positive integers. □

Claim 2. *The elementary symmetric polynomial over $z_0,z_1,\ldots,z_{k-1}$ of degree $k-j$ for any $p\in\{0,1,\ldots,k-1\}$ can be decomposed as follows:*

$$\begin{aligned} e_{k-j}(z_0,z_1,\ldots,z_{k-1}) = & e_{k-j}(z_0,z_1,\ldots,z_{p-1},z_{p+1},\ldots,z_{k-1}) \\ & + z_p e_{k-j-1}(z_0,z_1,\ldots,z_{p-1},z_{p+1},\ldots,z_{k-1}). \end{aligned} \tag{2}$$

Proof. Consider the Newton basis polynomial $\pi_{k,z}(x)$. For any $p\in\{0,1,\ldots,k-1\}$ the coefficient of its term of degree j can be computed as

$$\begin{aligned} \operatorname{coef}(j,\pi_{k,z}(x)) &= \operatorname{coef}(j,x(x-z_p)^{-1}\pi_{k,z}(x)) + \operatorname{coef}(j,-z_p(x-z_p)^{-1}\pi_{k,z}(x)) \\ &= \operatorname{coef}(j-1,(x-z_p)^{-1}\pi_{k,z}(x)) - z_p\operatorname{coef}(j,(x-z_p)^{-1}\pi_{k,z}(x)) \end{aligned}$$

The claim follows by applying Vieta's formulas to each side of this identity. □

Thanks to the decomposition (1) with $p=j$ we can express the entry $(j+1,k+1)$ of $\mathbf{H}(z_0,z_1,\ldots,z_n)$ by means of z_j and the entries (j,k) and $(j+1,k)$. This can be exploited to build the following code:

```
 1. function H = build_H( z )
 2. % Return the matrix H mapping a(n) into d(n).
 3. % Input:
 4. % -   z, interpolation sequence
 5. % Output:
 6. % -   H, matrix mapping a(n) into d(n)
 7. n = length( z ) - 1;
 8. H = [ cumprod([ 1 ones(1,n)*z(1) ]); zeros(n,n+1) ];
 9. for j = 1:n
10.  for k = j:n
11.    H( j+1,k+1 ) = H( j,k ) + z( j+1 ) * H( j+1,k );
12.  end
13. end
```

Given the interpolation sequence $z=(z_0,z_1,\ldots,z_n)$ it returns the matrix $\mathbf{H}(z_0,z_1,\ldots,z_n)$.

Thanks to the decomposition (2) with $p=k-1$ we can express the entry $(j+1,k+1)$ of $\mathbf{E}(z_0,z_1,\ldots,z_n)$ by means of z_{k-1} and the entries (j,k) and $(j+1,k)$. This can be exploited to build the following code:

```
 1. function E = build_E( z )
 2. % Return the matrix E mapping d(n) into a(n).
 3. % Input:
 4. % -   z, interpolation sequence
 5. % Output:
 6. % -   E, matrix mapping d(n) into a(n)
 7. n = length( z ) - 1;
 8. E = [ cumprod([ 1,-z(1:n) ]); zeros(n,n+1) ];
 9. for j = 1:n
10.  for k = j:n
11.    E( j+1,k+1 ) = E( j,k ) - z( k ) * E( j+1,k );
12.  end
13. end
```

Given the interpolation sequence $z=(z_0,z_1,\ldots,z_n)$ it returns the matrix $\mathbf{E}(z_0,z_1,\ldots,z_n)$.

3 Factorization of the change of basis operator

Although the two algorithms with which we concluded the past section can be used to switch back and forth between $\mathbf{d}(n)$ and $\mathbf{a}(n)$ it may not be a good idea. In fact much faster algorithms can be created and we will do so in this section after we will have shown some factorization properties of the matrices $\mathbf{E}(z_0, z_1, \ldots, z_n)$ and $\mathbf{H}(z_0, z_1, \ldots, z_n)$.

Consider now again the decomposition of the elementary symmetric polynomials given in equation (2), this time with $p = 0$. We have

$$\begin{aligned} e_{k-j}(z_0, z_1, \ldots, z_{k-1}) &= e_{k-j}(z_1, \ldots, z_{k-1}) + z_0 e_{k-j-1}(z_1, \ldots, z_{k-1}) \\ &= e_{k-j}(0, z_1, \ldots, z_{k-1}) + z_0 e_{k-j-1}(0, z_1, \ldots, z_{k-1}) \end{aligned} \tag{3}$$

where the last equality comes from intrinsic properties of such polynomials. From the very rule that we followed to build the matrix $\mathbf{E}(z_0, z_1, \ldots, z_n)$ the left hand side of (3) equals $(-1)^{k-j}$ times the entry $(j+1, k+1)$ of $\mathbf{E}(z_0, z_1, \ldots, z_n)$. Analogously on the right hand side of (3) we have that $e_{k-j}(0, z_1, \ldots, z_{k-1})$ equals $(-1)^{k-j}$ times the entry $(j+1, k+1)$ of $\mathbf{E}(0, z_1, \ldots, z_n)$ while we have that $e_{k-j-1}(0, z_1, \ldots, z_{k-1})$ equals $(-1)^{k-j-1}$ times the entry $(j+2, k+1)$ of $\mathbf{E}(0, z_1, \ldots, z_n)$.

What we can deduce from this is that for j and k in $\{0, 1, \ldots, n-1\}$ the entry $(j+1, k+1)$ of $\mathbf{E}(z_0, z_1, \ldots, z_n)$ equals the entry $(j+1, k+1)$ of $\mathbf{E}(0, z_1, \ldots, z_n)$ minus z_0 times the entry $(j+2, k+1)$ of $\mathbf{E}(0, z_1, \ldots, z_n)$. In matrix form this can be represented by

$$\mathbf{E}(z_0, z_1, \ldots, z_n) = (\mathbf{I}(n+1) - z_0 \mathbf{J}(n+1))\mathbf{E}(0, z_1, \ldots, z_n) \tag{4}$$

where $\mathbf{I}(n+1)$ and $\mathbf{J}(n+1)$ are respectively the identity matrix and the zero matrix having all ones on its first superdiagonal both of size $n+1$. Bearing in mind how the matrix representing the inverse change of basis operator is built we know that

$$(\mathbf{I}(n+1) - z_0 \mathbf{J}(n+1)) = \mathbf{E}(z_0, 0, \ldots, 0)$$

and hence

$$\mathbf{E}(z_0, z_1, \ldots, z_n) = \mathbf{E}(z_0, 0, \ldots, 0)\mathbf{E}(0, z_1, \ldots, z_n).$$

Again by using the definition of the elementary symmetric polynomials we know that

$$\mathbf{E}(0, z_1, \ldots, z_n) = \begin{pmatrix} 1 & \\ & \mathbf{E}(z_1, \ldots, z_n) \end{pmatrix} \tag{5}$$

and therefore we can iterate the factorization step by step over the submatrices until we get to the full factorization

$$\mathbf{E}(z_0, z_1, \ldots, z_n) = \mathbf{E}(z_0, 0, \ldots, 0)\mathbf{E}(0, z_1, 0, \ldots, 0) \cdots \mathbf{E}(0, \ldots, 0, z_n). \tag{6}$$

There is now the possibility to obtain an even more complete factorization for which the matrix $\mathbf{E}(z_0, z_1, \ldots, z_n)$ is factored into the product of $n(n+1)/2$ matrices. Set the matrix $\mathbf{E}_i(z_p)$ to be the $n+1$ square identity matrix with $-z_p$ in the entry $(i, i+1)$, i.e.

$$\mathbf{E}_i(z_p) = \begin{pmatrix} \mathbf{I}(i) & & & \\ & 1 & -z_p & \\ & & 1 & \\ & & & \mathbf{I}(n-i-1) \end{pmatrix}.$$

If we now left multiply $\mathbf{E}_{i+1}(z_p)$ into $\mathbf{E}_i(z_p)$ it is clear that we are merely copy–pasting the bottom right corner of $\mathbf{E}_{i+1}(z_p)$ into the bottom right corner of $\mathbf{E}_i(z_p)$, since the latter consists of an identity matrix. Following this simple principle we can factor $\mathbf{E}(0, \ldots, z_p, \ldots, 0)$ with $p < n$ into the $n-p$ matrices

$$\mathbf{E}_n(z_p)\mathbf{E}_{n-1}(z_p) \cdots \mathbf{E}_{p+1}(z_p)$$

where $\mathbf{E}(0, \ldots, 0, z_n)$ is already the identity matrix. It is possible hence to rewrite the matrix $\mathbf{E}(z_0, z_1, \ldots, z_n)$ as the product of $n(n+1)/2$ factors

$$\mathbf{E}(z_0, z_1, \ldots, z_n) = \overset{\curvearrowright}{\prod_{k=0}^{n-1}} \Big(\overset{\curvearrowright}{\prod_{j=0}^{n-k-1}} \mathbf{E}_{n-j}(z_k) \Big) \tag{7}$$

where the curved arrow pointing right indicates that the product sign has to be understood as juxtaposing the upcoming matrices on the right side.

The factorizations of $\mathbf{E}(z_0, z_1, \ldots, z_n)$ shown in (6) and (7) offer the possibility to factor the matrix $\mathbf{H}(z_0, z_1, \ldots, z_n)$ too. In fact we know that the matrix $\mathbf{H}(z_0, z_1, \ldots, z_n)$ equals $\mathbf{E}^{-1}(z_0, z_1, \ldots, z_n)$ and hence we know that, analogously to equation (6), $\mathbf{H}(z_0, z_1, \ldots, z_n)$ equals

$$\mathbf{E}(0, \ldots, 0, z_n)^{-1}\mathbf{E}(0, \ldots, 0, z_{n-1}, 0)^{-1} \cdots \mathbf{E}(z_0, 0, \ldots, 0)^{-1}. \tag{8}$$

Following the same reasoning we can rewrite, analogously to equation (7), the matrix $\mathbf{H}(z_0, z_1, \ldots, z_n)$ as the product of $n(n+1)/2$ factors

$$\mathbf{H}(z_0, z_1, \ldots, z_n) = \overset{\curvearrowright}{\prod_{k=0}^{n-1}} \Big(\overset{\curvearrowright}{\prod_{j=0}^{n-k-1}} \mathbf{E}^{-1}_{n-j}(z_k) \Big).$$

As a side note if we define $\mathbf{H}_i(z_p)$ to be the inverse of the matrix $\mathbf{E}_i(z_p)$, it can be easily seen that

$$\mathbf{H}_i(z_p) = \mathbf{E}_i(-z_p) \tag{9}$$

and thus we can rewrite the new factorizaton in a nicer form as

$$\mathbf{H}(z_0,z_1,\ldots,z_n)=\overset{\frown}{\prod_{k=0}^{n-1}}(\overset{\frown}{\prod_{j=0}^{n-k-1}}\mathbf{H}_{n-j}(z_k)). \tag{10}$$

We now list some highly efficient routines for switching between the monomial and Newton's basis coefficients. Such routines will be based on the factorizations given in equations (7) and (10). Part of the efficiency of such algorithm lies in the fact of being in–place, this means that they are algorithms that transform input using no auxiliary data structure. As a consequence at first sight it may be confusing that the vector $\mathbf{d}(n)$ will be given in input under the name `a_n` and viceversa $\mathbf{a}(n)$ will be given in input under the name `d_n`.

We start by listing the code that given $\mathbf{d}(n)$ and the interpolation sequence $z=(z_0,z_1,\ldots,z_n)$ returns the coefficients $\mathbf{a}(n)$, i.e. it performs the mapping of the operator represented by the matrix $\mathbf{E}(z_0,z_1,\ldots,z_n)$.

```
 1. function a_n = Ed_n( z,a_n )
 2. % Compute a(n) given d(n) and z.
 3. % Input:
 4. % -    z, interpolation sequence
 5. % -  a_n, vector d(n)
 6. % Output:
 7. % -  a_n, vector a(n)
 8. n = length( z ) - 1;
 9. for j = n:-1:1
10.   for k = j:n
11.      a_n( k ) =  a_n( k ) - z( j ) * a_n( k+1 );
12.   end
13. end
```

Notice that we can get rid of the inner loop by substituting it with the line

```
10. a_n( k:n ) = a_n( k:n ) - z( k ) * a_n( k+1:n+1 );
```

that is a vectorised and therefore parallelizable version of the code. This is equivalent to implementing the factorization of (6).

We proceed by listing the inverse algorithm that given $\mathbf{a}(n)$ and the interpolation sequence $z=(z_0,z_1,\ldots,z_n)$ returns the coefficients $\mathbf{d}(n)$, i.e. it performs the mapping of the operator represented by the matrix $\mathbf{H}(z_0,z_1,\ldots,z_n)$.

```
 1. function d_n = Ha_n( z,d_n )
 2. % Compute d(n) given a(n) and z.
 3. % Input:
 4. % -    z, interpolation sequence
 5. % -  d_n, vector a(n)
 6. % Output:
 7. % -  d_n, vector d(n)
 8. n = length( z ) - 1;
 9. for j = 0:n-1
10.   for k = n:-1:j+1
11.     d_n( k ) = d_n( k ) + z( j+1 ) * d_n( k+1 );
12.   end
13. end
```

Notice that we can't get rid of the inner loop as we could do with the previous code.

In order to do so we have to reorder somehow the matrix multiplications appearing in equation (10). Thanks to their special structure, the matrices $\mathbf{H}_{i_1}(z_{p_1})$ and $\mathbf{H}_{i_2}(z_{p_2})$ commute if and only if $|i_1-i_2|\neq 1$. Bearing that in mind, consider the matrices in equation (10) written down in their extended form, that is without using the product notation. Considering the matrices from right to left we pick the first one that can commute with its left neighbor, that is $\mathbf{H}_1(z_0)$, and we are going to swap it with its left neighbors until we encounter $\mathbf{H}_2(z_1)$ with which it can't commute. At this point we are going to pick $\mathbf{H}_2(z_1)\mathbf{H}_1(z_0)$ and swap it with its left neighbors until we encounter $\mathbf{H}_3(z_2)$ and so on until the moment in which we obtain

$$\mathbf{H}_n(z_{n-1})\mathbf{H}_{n-1}(z_{n-2})\cdots\mathbf{H}_1(z_0)(\overset{\frown}{\prod_{k=0}^{n-2}}(\overset{\frown}{\prod_{j=0}^{n-k-2}}\mathbf{H}_{n-j}(z_k))). \tag{11}$$

We can iterate the commuting operation on and on over the matrices still nested into the product sign. In this way we obtain the following reordering of the original factorization:

$$\mathbf{H}(z_0,z_1,\ldots,z_n)=\overset{\frown}{\prod_{k=0}^{n-1}}(\overset{\frown}{\prod_{j=0}^{n-k-1}}\mathbf{H}_{n-j}(z_{n-k-j-1})). \tag{12}$$

From this we have available also a reordering for the factorization of the inverse matrix

$$\mathbf{E}(z_0, z_1, \ldots, z_n) = \prod_{k=0}^{\overset{\frown}{n-1}} \Big(\prod_{j=0}^{\overset{\frown}{n-k-1}} \mathbf{E}_{n-j}(z_{n-k-j-1}) \Big). \tag{13}$$

Let us see how the changes in the arrangement of the matrices shown in formula (13) reflect in the code that given $\mathbf{d}(n)$ and the interpolation sequence $z = (z_0, z_1, \ldots, z_n)$ returns $\mathbf{a}(n)$.

```
 1. function a_n = Ed_n_res( z,a_n )
 2. % Compute a(n) given d(n) and z. Resorted factorization.
 3. % Input:
 4. % -    z, interpolation sequence
 5. % -  a_n, vector d(n)
 6. % Output:
 7. % -  a_n, vector d(n)
 8. n = length( z ) - 1;
 9. for k = 0:n-1
10.   for j = n:-1:k+1
11.     a_n( j ) = a_n( j ) - z( j-k ) * a_n( j+1 );
12.   end
13. end
```

Differently from before, this algorithm is not vectorizable hence we cannot get rid of the inner loop nor we can parallelise the calculations.

Let us see instead how the changes in the arrangement of the matrices shown in formula (12) reflect into the code that given $\mathbf{a}(n)$ and the interpolation sequence $z = (z_0, z_1, \ldots, z_n)$ returns $\mathbf{d}(n)$.

```
 1. function d_n = Ha_n_res( z,d_n )
 2. % Compute d(n) given a(n) and z. Resorted factorization.
 3. % Input:
 4. % -    z, interpolation sequence
 5. % -  d_n, vector a(n)
 6. % Output:
 7. % -  d_n, vector d(n)
 8. n = length( z ) - 1;
 9. for k = 0:n-1
10.   for j = n-k:n
11.     d_n( j ) = d_n( j ) + z( j-n+k+1 ) * d_n( j+1 );
12.   end
13. end
```

This algorithm can be successfully vectorised. In fact we can get rid of the inner loop replacing it with the line

```
10. d_n( n-k:n ) = d_n( n-k:n ) + z( 1:k+1 ) .* d_n( n-k+1:n+1 );
```

making it possible to parallelise the calculations.

We conclude this section with a remark reconnecting this work to the literature. At the beginning of Section 2 we mentioned [6, Theorem p. 776], which states that if $\mathbf{V}(z_0, z_1, \ldots, z_n)$ is the Vandermonde matrix over the interpolation sequence then the standard LU-decomposition of its transpose is

$$\mathbf{V}(z_0, z_1, \ldots, z_n)^T = \mathbf{H}(z_0, z_1, \ldots, z_n)^T \mathbf{U}(z_0, z_1, \ldots, z_n)$$

where $\mathbf{U}(z_0, z_1, \ldots, z_n)$ is upper triangular. Since the inverse of $\mathbf{H}(z_0, z_1, \ldots, z_n)^T$ is $\mathbf{E}(z_0, z_1, \ldots, z_n)^T$ we can also write

$$\mathbf{E}(z_0, z_1, \ldots, z_n)^T \mathbf{V}(z_0, z_1, \ldots, z_n)^T = \mathbf{U}(z_0, z_1, \ldots, z_n)$$

and therefore any transpose decomposition of $\mathbf{E}(z_0, z_1, \ldots, z_n)$ can be seen as the concatenation of the Gaussian elimination steps.

4 Computing the divided difference of the exponential and closely related functions

In the recent years in the field of applications the problem of computing functions of matrices has attracted a rising interest. Among such functions it can be highlighted the exponential and more in general the so called ϕ_l functions, that are defined as:

$$\phi_l(x) = \sum_{i=0}^{\infty} \frac{x^i}{(i+l)!} \tag{14}$$

and that include the exponential function since clearly $\phi_0(x)$ coincides with e^x. When it comes to the approximation of functions of matrices it is very often convenient to interpolate at some sequence $z = (z_0, z_1, \ldots, z_n)$ in order to try and exploit spectral properties of the input matrix. Hence a fast and accurate computation of divided differences plays a prominent role in executing

such tasks. In this section we are going to introduce a new algorithm for the efficient computation of the divided differences for the ϕ_l functions at the interpolation sequence $z = (z_0, z_1, \dots, z_n)$, namely

$$d[z_0], d[z_0, z_1], \dots, d[z_0, z_1, \dots, z_n].$$

We start by presenting the state of the art algorithm for the computation of the divided differences of the exponential function at z developed in [10] since our routine will inherit its structure. This algorithm is based on the so called Opitz's theorem, that has been originally introduced in [11]. According to the Opitz theorem, given any $k \in \{0, 1, \dots, n\}$, the divided differences

$$d[z_k], d[z_k, z_{k+1}], \dots, d[z_k, z_{k+1}, \dots, z_n]$$

of a function $f(x)$ are in order in the last $n-k+1$ entries of the $(k+1)$th column of $f(\mathbf{Z}(z_0, z_1, \dots, z_n))$ where

$$\mathbf{Z}(z_0, z_1, \dots, z_n) = \begin{pmatrix} z_0 & & & \\ 1 & z_1 & & \\ & \ddots & \ddots & \\ & & 1 & z_n \end{pmatrix},$$

and hence the wanted divided differences can be derived as

$$\exp(\mathbf{Z}(z_0, z_1, \dots, z_n))e_1$$

where e_1 is the first column of the identity matrix of size $n+1$.

At this point one could think that it is enough to compute this vector by using one of the many routines developed to approximate the action of the matrix exponential. The reality is that such routines are only relatively accurate in norm while we are required to approximate in relatively high precision each one of the divided differences that, for the exponential function, are rapidly decreasing. Furthermore general purpose routines are not designed for this case of interest that obviously presents a pattern that must be exploited.

Returning to the routine developed in [10], a crucial step that must be performed in order to handle accurately arbitrarily large inputs is to apply a scaling technique. It is possible in fact to recover the wanted divided differences by just powering the matrix of the divided differences on a scaled sequence. In practice, for a positive integer s, set

$$\mathbf{F}_{(0)} := \exp(\mathbf{Z}(2^{-s}z_0, 2^{-s}z_1, \dots, 2^{-s}z_n))$$

and set

$$\mathbf{F}_{(i+1)} := \mathbf{R}\mathbf{F}_{(i)}^2\mathbf{R}^{-1}$$

with

$$\mathbf{R} = \begin{pmatrix} 1 & & & \\ & 2 & & \\ & & \ddots & \\ & & & 2^{n-1} \end{pmatrix}$$

then, thanks to the Lemma from ([10], pp.509), we know that for $0 \le i \le s$ the following holds:

$$\mathbf{F}_{(i)} = \exp(\mathbf{Z}(2^{i-s}z_0, 2^{i-s}z_1, \dots, 2^{i-s}z_n))$$

and in particular

$$\mathbf{F}_{(s)} = \exp(\mathbf{Z}(z_0, z_1, \dots, z_m))$$

that is the table of divided differences at z. Since we are not interested in the whole divided differences matrix but just in the first column, the powering part can be done applied to the vector e_1 and hence performing just matrix–vector products. In addition to that, [10] has exploited the work of [12] that investigates the relation between the triangular matrix $\mathbf{T}$ and $f(\mathbf{T})$ when $f(x)$ is an analytic function as the exponential. As a result, $\mathbf{F}_{(0)}$ is computed in an extremely efficient way and the resulting routine is characterized by very high performances.

What we aim to do in this section is to develop a new routine that outperforms the one from [10] by exploiting the tools we built in this work. In particular we will focus on the fast computation of $\mathbf{F}_{(0)}$ while we will maintain the powering part untouched.

One idea would be to adapt the algorithms derived in the previous section to the purpose of approximating the divided differences of a power series. In fact the matrix $\mathbf{H}(z_0, z_1, \dots, z_n)$ has 1 as its last bottom right entry, hence it is assigns the value a_n to $d[z_0, z_1, \dots, z_n]$. This is exact if a_n is the highest nonzero coefficient of $p(x)$ while it merely is an approximation otherwise. To fix this problem we just have to add many enough additional interpolation points $z_{n+1}, z_{n+2}, \dots, z_N$ in order to recover the lost accuracy. Unfortunately, this approach would require computing each column of $\mathbf{F}_{(0)}$ separately with an increased computational complexity due to the additional points. Therefore it may be too slow when it comes to compete with an algorithm as optimized as the one developed in [10].

To tackle this inconvenient we now prove another factorization of the matrix $\mathbf{H}(z_0, z_1, \dots, z_n)$ that we will show to have an interesting property. Consider the following manipulation of formula (4):

$$\begin{aligned} \mathbf{E}(z_0, z_1, \dots, z_n) &= (\mathbf{I}(n+1) - z_0\mathbf{J}(n+1))\mathbf{E}(0, z_1, \dots, z_n) \\ &= \mathbf{E}(0, z_1, \dots, z_n) - z_0\mathbf{J}(n+1)\mathbf{E}(0, z_1, \dots, z_n) \end{aligned}$$

that, together with the trivial identity

$$\mathbf{J}(n+1)\mathbf{E}(0,z_1,\ldots,z_n)=\mathbf{E}(z_1,\ldots,z_n,0)\mathbf{J}(n+1)$$

that is due to the fact that $\mathbf{J}(n+1)$ is the null matrix with ones on its first superdiagonal, gives that $\mathbf{E}(z_0,z_1,\ldots,z_n)$ equals

$$\mathbf{E}(z_1,\ldots,z_n,0)(\mathbf{H}(z_1,\ldots,z_n,0)\mathbf{E}(0,z_1,\ldots,z_n)-z_0\mathbf{J}(n+1)).$$

Let's study now the structure of the matrix $\mathbf{H}(z_1,\ldots,z_n,0)\mathbf{E}(0,z_1,\ldots,z_n)$. To do so we employ formula (11) over $z_1,z_2,\ldots,z_n,0$ instead of $z_0,z_1,\ldots,z_n$, showing that $\mathbf{H}(z_1,\ldots,z_n,0)$ can be rewritten as:

$$\mathbf{H}_n(z_n)\mathbf{H}_{n-1}(z_{n-1})\cdots\mathbf{H}_1(z_1)(\prod_{k=0}^{\overset{\frown}{n-2}}(\prod_{j=0}^{\overset{\frown}{n-k-2}}\mathbf{H}_{n-j}(z_{k+1}))).$$

We know already from the past section that

$$\prod_{k=0}^{\overset{\frown}{n-2}}(\prod_{j=0}^{\overset{\frown}{n-k-2}}\mathbf{H}_{n-j}(z_{k+1}))=\prod_{k=0}^{\overset{\frown}{n-2}}(\prod_{j=0}^{\overset{\frown}{n-k-2}}\mathbf{E}^{-1}_{n-j}(z_{k+1}))=(\prod_{k=0}^{\overset{\frown}{n-2}}(\prod_{j=0}^{\overset{\frown}{n-k-2}}\mathbf{E}_{n-j}(z_{k+1})))^{-1}$$

and hence, by comparison with (7) we obtain

$$(\prod_{k=0}^{\overset{\frown}{n-2}}(\prod_{j=0}^{\overset{\frown}{n-k-2}}\mathbf{E}_{n-j}(z_{k+1})))^{-1}=\mathbf{E}^{-1}(0,z_1,\ldots,z_n),$$

from which we can deduce that

$$\mathbf{H}(z_1,\ldots,z_n,0)\mathbf{E}(0,z_1,\ldots,z_n)=\mathbf{H}_n(z_n)\mathbf{H}_{n-1}(z_{n-1})\cdots\mathbf{H}_1(z_1)$$

which is the $n+1$ sized identity matrix with $z_1,z_2,\ldots,z_n$ on its first superdiagonal. In conclusion we have that the matrix $\mathbf{E}(z_0,z_1,\ldots,z_n)$ equals

$$\mathbf{E}(z_1,z_2,\ldots,z_n,0)\mathbf{H}_n(z_n-z_0)\mathbf{H}_{n-1}(z_{n-1}-z_0)\cdots\mathbf{H}_1(z_1-z_0)$$

that, thanks to (9), we know to be also equal to

$$\mathbf{E}(z_1,z_2,\ldots,z_n,0)\mathbf{E}_n(z_0-z_n)\mathbf{E}_{n-1}(z_0-z_{n-1})\cdots\mathbf{E}_1(z_0-z_1).$$

We can proceed analogously in factorizing the matrix $\mathbf{E}(z_1,z_2,\ldots,z_n,0)$ and then the matrix $\mathbf{E}(z_2,z_3,\ldots,z_n,0,0)$ and so on until we get

$$\mathbf{E}(z_0,z_1,\ldots,z_n)=\prod_{k=0}^{\overset{\frown}{n}}(\prod_{j=0}^{\overset{\frown}{n-1}}\mathbf{E}_{n-j}(z_k-z_{n-j+k})) \tag{15}$$

provided that we set $z_k=0$ for every $k\geq n+1$. Hence

$$\mathbf{H}(z_0,z_1,\ldots,z_n)=\prod_{k=0}^{\overset{\frown}{n}}(\prod_{j=0}^{\overset{\frown}{n-1}}\mathbf{H}_{n-j}(z_k-z_{n-j+k})) \tag{16}$$

provided, again, that $z_k=0$ for every $k\geq n+1$. We now have available a new algorithm for switching between $\mathbf{a}(n)$ and $\mathbf{d}(n)$.

This algorithm enjoys a very useful property for our purposes when we use it to compute $\mathbf{d}(n)$ from $\mathbf{a}(n)$. In fact at each stage $n-k$ of the outer product sign we are computing the divided differences over $z_k,z_{k+1},\ldots,z_{n+k}$ starting from the divided differences at $z_{k+1},z_{k+2},\ldots,z_{n+k+1}$. To make it evident consider as an example the first step we made toward this factorization:

$$\mathbf{E}(z_1,z_2,\ldots,z_n,0)\mathbf{E}_n(z_0-z_n)\mathbf{E}_{n-1}(z_0-z_{n-1})\cdots\mathbf{E}_1(z_0-z_1).$$

when inverted and applied to $\mathbf{a}(n)$ we have

$$\mathbf{H}_1(z_0-z_1)\mathbf{H}_2(z_0-z_2)\cdots\mathbf{H}_n(z_0-z_n)\mathbf{H}(z_1,z_2,\ldots,z_n,0)\mathbf{a}(n)$$

that clearly equals

$$\mathbf{H}_1(z_0-z_1)\mathbf{H}_2(z_0-z_2)\cdots\mathbf{H}_n(z_0-z_n)\mathbf{d}'(n)$$

where $\mathbf{d}'(n)$ is the vector of the divided differences over $z_1,z_2,\ldots,z_n,0$. Hence it is possible to fill up the columns of the matrix $\mathbf{F}_{(0)}$ while computing the divided differences over $z_0,z_1,\ldots,z_N$ from the coefficients $a_0,a_1,\ldots,a_N$, reducing drastically the overall computational time.

For our perticular purpose we picked N to be equal to $n+30$ and the scaling parameter s to be a positive integer such that the interpolation points are not spread too far apart, namely $s^{-1}|z_i-z_j|\leq 3.5$ and $s^{-1}|z_k|\leq 3.5$ for any choice of the positive integers $i,j,k\leq n+1$. This is due to the fact that in the worst case scenario, i.e. $n=0$, we want to approximate accurately $d[s^{-1}z_0]=e^{s^{-1}z_0}$ and, in [1, 4] has been shown that double precision is attainable by means of the Taylor series truncated to degree 30 provided $s^{-1}|z_0|\leq 3.5$.

In order to enforce the condition on the interpolation points and to sensibly simplify the task we apply a preconditioning to the interpolation sequence by computing the divided differences of the exponential function at $z_0-\mu, z_1-\mu, \ldots, z_n-\mu$ with

$$\mu = n^{-1}\sum_{i=0}^{n} z_i.$$

In this way $s^{-1}|z_i - z_j| \leq 3.5$ clearly implies $s^{-1}|z_k - \mu| \leq 3.5$ for any $i, j, k \leq n+1$. Not only, such a shifting of the interpolation sequence may lead to smaller scaling parameters leading to improve the overall efficiency. Then, by exploiting the properties of the exponential function, we recover the divided differences over the original interpolation sequence by multiplying e^{μ} into the divided differences computed over the shifted points.

In the following we list the algorithm we have developed starting by the factorization we just have outlined in combination with the scaling and shifting algorithm.

```
 1. function dd = dd_phi( z,l )
 2. % Compute phi_l(x)'s divided differences.
 3. % Input:
 4. % -    z, interpolation points
 5. % Output:
 6. % -   dd, divided differences
 7. z = [ zeros( l,1 ); z( : ) ]; mu = mean( z ); z = z - mu;
 8. n = length( z ) - 1; N = n + 30;
 9. F = zeros( n+1 );
10. for i = 1:n
11.   F( i+1:n+1,i ) = z( i ) - z( i+1:n+1 );
12. end
13. s = max( ceil( max( max( abs( F ) ) ) / 3.5 ),1 );
14. % Compute F_0
15. dd = [ 1 1./cumprod( (1:N)*s ) ];
16. for j = n:-1:0
17.   for k = N:-1:(n-j+1)
18.     dd( k ) = dd( k ) + z( j+1 ) * dd( k+1 );
19.   end
20.   for k = (n-j):-1:1
21.     dd( k ) = dd( k ) + F( k+j+1,j+1 ) * dd( k+1 );
22.   end
23.   F( j+1,j+1:n+1 ) = dd( 1:n-j+1 );
24. end
25. F( 1:n+2:(n+1)^2 ) = exp( z/s );
26. F = triu( F );
27. % Squaring Part
28. dd = F( 1,: );
29. for k = 1:s-1
30.     dd = dd * F;
31. end
32. dd = exp( mu ) * transp(dd(l+1:n+1));
```

We now explain how we can obtain the divided differences of the generic function $\phi_l(x)$ provided just the two small modifications of lines 7 and 32. From formula (14) and from the representation of the divided differences show in the first part of this work, the approximation of the divided differences of $\phi_l(x)$ over the interpolation sequence $(z_0, z_1, \ldots, z_n)$ equals

$$\mathbf{d}'(n) = \mathbf{H}(z_0, z_1, \ldots, z_n)\mathbf{a}'(n)$$

with $\mathbf{a}'(n)$ being the vector such that the kth entry equals the $(k+l)$th entry of the vector of the coefficients of the exponential function in the monomial basis $\boldsymbol{m}(x, n)$, namely: $a_{k+l} = 1/(k+l-1)!$. As a consequence one can obtain $\mathbf{d}'(n)$ by simply computing the divided differences of the exponential function over the interpolation sequence modified by adding l interpolation points at zero at the beginning $0, \ldots, 0, z_0, z_1, \ldots, z_n$ (executed at line 7) and then discarding the first l (executed at line 32). In fact from (5) we know that

$$\mathbf{H}(0, \ldots, 0, z_0, z_1, \ldots, z_n)$$

is the matrix having an l sized identity matrix in its upper left corner.

As a final note we highlight that with minor modifications of the algorithm in [10] it is possible to obtain the whole divided difference table over an interpolation sequence. In order to apply those minor modification to our routine it is enough to substitute line `13.` with

```
13. s = max( 2^ceil( log2( max( max( abs( F ) ) ) / 3.5 ) ),1 );
```

and lines from `28.` to `32.` with

```
28. for k = 1:log2( s )
29.   F = F^2;
30. end
31. F = exp( mu ) * F( l+1:n+1,l+1:n+1 );
```

and of course demanding the output to be `F` and not `dd`.

5 Numerical experiments

In this section we are going to run some some numerical tests in order to test thoroughly the performances of our new algorithm: `dd_phi` with respect to the most advanced competitors. The competitors are:

- `dd_ts`, this routine is based on the scaling and squaring algorithm applied to Opitz's theorem for triangular matrices developed in [10] and that we outlined in the past section. The Matlab implementation we use is due to the author of [2] that also extended this algorithm to the high performance computation of the ϕ_l functions.
- `exptayotf` from [5], is the only existing routine able to compute the matrix exponential in double precision arithmetic for any given tolerance in a backward stable way. We will use this routine with tolerance set to `realmin` $\approx 2.23\text{e}-308$ in combination with the Opitz's theorem in order to compute the divided differences required for running the tests.

On the other hand we don't compare with the so called Standard Recurrence algorithm that is the algorithm stemming from the recurrence defining the divided difference, i.e.

$$d[z_k, z_{k+1}, \ldots, z_{k+j}] = \frac{d[z_k, z_{k+1}, \ldots, z_{k+j-1}] - d[z_{k+1}, z_{k+2}, \ldots, z_{k+j}]}{z_k - z_{k+j}}$$

where $d[z_k] = \mathrm{e}^{z_k}$. The reason for this is that when it comes to the numerical implementation it is well known that the resulting routine is a very unstable algorithm and prone to huge accuracy loss. Evidence supporting this can be found in Table 3 from [2] where a catastrophic propagation of the error through the steps of the Standard Recurrence algorithm is made evident. In addition to that, we are not going to perform any comparison with the algorithm from [8] based on the computation of divided differences via their representation as a contour integral. In fact although such an algorithm is an excellent tool for computing accurately the divided differences of certain classes of analytic analytic functions, it is not designed expressly for those of the exponential function and therefore a comparison would turn out to be unfair to say the least.

The first test we perform consists in testing the speed performances of the three routines under examination. In order to accurately measure the timing of each computation we run the tests by using just one processor (Matlab's option `-singleCompThread`). Moreover, we run each routine on each sequence 20 times and we register the average time taken to process each input. The set of sequences over which we run the three routines are:

s1. $\gamma \cdot (z_0, z_1, \ldots, z_n)$ where $n \in \{1, 3, 5, \ldots, 99\}$, $\gamma = 8$ and the interpolation points z_i are randomly chosen following a normal distribution of mean 0 and variance 1;

s2. $\gamma \cdot (z_0, z_1, \ldots, z_n)$ where $n \in \{1, 3, 5, \ldots, 99\}$, $\gamma = 8$ and the interpolation points z_i have real and imaginary parts chosen following a normal distribution of mean 0 and variance 1.

We then report the results in two different graphs (see Figure 1) to compare the performances over real and complex inputs. What we can observe is that over both real and complex inputs `dd_phi` performs best by a least one order of magnitude. The surprising data is that the routine `exptayotf` is faster than `dd_ts` in the complex case despite the fact that it is not optimized for this particular task. We repeat this test but we ask instead of the vector of the divided differences the whole divided differences table. The results, reported in Figure 2, show that also in this case `dd_phi` stands alone as the fastest routine.

The next test we are going to perform is meant to establish how accurate is our routine `dd_phi` with respect to `dd_ts` and `exptayotf`. The interpolation sequences that we use to test the accuracy of the three algorithms are:

a1. $\gamma \cdot (z_0, z_1, \ldots, z_n)$ where $n \in \{10, 25, 50, 100\}$, $\gamma \in \{2, 4, \ldots, 512\}$ and the interpolation points z_i are randomly chosen following a normal distribution of mean 0 and variance 1;

a2. $\gamma \cdot (z_0, z_1, \ldots, z_n)$ where $n \in \{10, 25, 50, 100\}$, $\gamma \in \{2, 4, \ldots, 512\}$ and the interpolation points z_i have real and imaginary parts chosen following a normal distribution of mean 0 and variance 1;

a3. $\gamma \cdot (z_0, z_1, \ldots, z_n)$ where $n \in \{10, 25, 50, 100\}$, $\gamma \in \{2, 4, \ldots, 512\}$ and the interpolation points z_i are the n Chebyshev points over the interval $[-1, 1]$;

a4. $\gamma \cdot (z_0, z_1, \ldots, z_n)$ where $n \in \{10, 25, 50, 100\}$, $\gamma \in \{2, 4, \ldots, 512\}$ and the interpolation points z_i are the n Leja points (see [3, 4, 7, 13]) over the interval $[-1, 1]$;

a5. $\gamma \cdot (z_0, z_1, \ldots, z_n)$ where $n \in \{10, 25, 50, 100\}$, $\gamma \in \{2, 4, \ldots, 512\}$ and the interpolation points z_i are the n Leja points over the closed unit disk in the complex plane (see [13, Example 1.3]);

a6. $\gamma \cdot (z_0, z_1, \ldots, z_n)$ where $n \in \{10, 25, 50, 100\}$, $\gamma \in \{2, 4, \ldots, 512\}$ and the interpolation points z_i are coalescing, i.e. $z_i = 2^{-i}$.

The sequences in a1.–a3. are then reordered à la Leja (see [4, Section 3.2] and [13, Formula (1.5)]) so that they can be meaningful from a numerical point of view. In fact such a reordering is shown to lead to higher stability when it comes to Newton interpolation (see [13, Example 4.1]). On the other hand the sequences in a4.–a5. are by construction already ordered à la Leja while we do not reorder those in a6. since they are designed to stress the three routines.

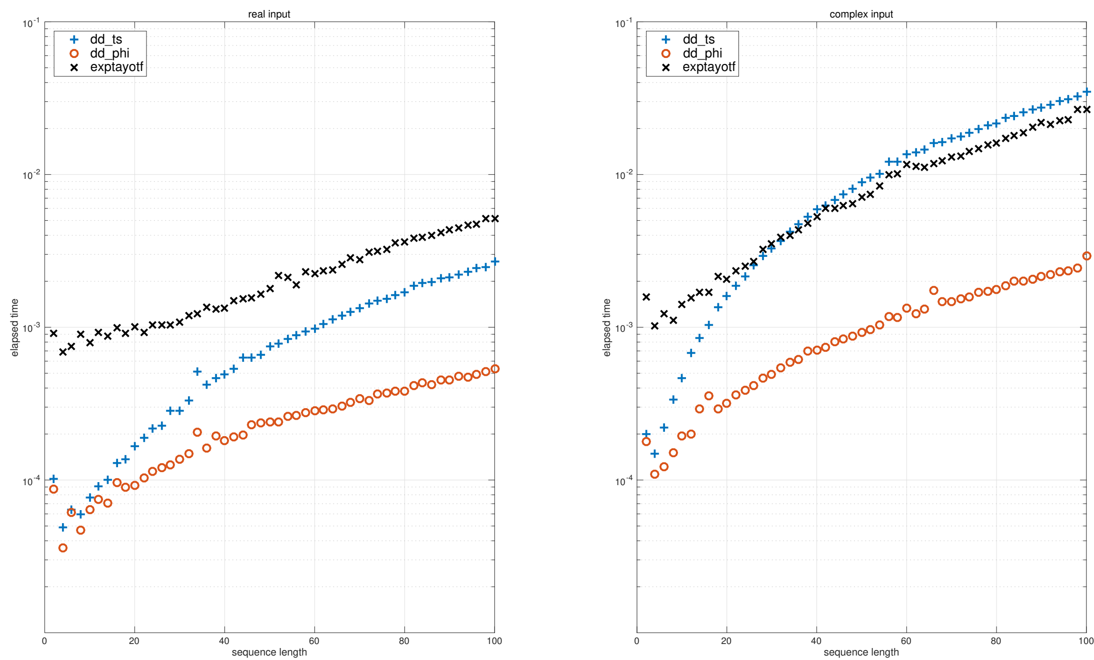


Figure 1: Average elapsed time to compute the divided differences for the sequences described in s1. and s2.

For each sequence $\gamma \cdot (z_0, z_1, \ldots, z_n)$ we compute the mean relative error committed by each routine taking as trusted reference the divided differences computed using `exptayotf` with tolerance set to `realmin` $\approx 2.23\text{e}{-}308$ and the input converted in a 200 digits `vpa` data type from the variable precision arithmetic toolbox of Matlab. We plot in Figures from 3 to 6 the mean relative error committed by each routine as the scale parameter γ increases for the sequences described in a3.–a6., the most significant from a numerical point of view. We deduce from Figures 3 to 6 that while there does exist a clear relation between the choice of the scale parameter γ and the mean relative error it does not exist between the length of a sequence and the mean relative error.

What is more important is that it appears evident that our routine `dd_phi` is broadly as accurate as its main competitors. Now in order to rigorously determine which routine is the most accurate we present the data from Figures 3 to 6 together with the data relative to the families of sequences described in a1. and a2. in Figure 7 as a performance profile where for a given α the corresponding point on each curve indicates the fraction p of sequences on which the method had a mean relative error at most a factor α times a the machine precision `tol`$\approx 2.22\text{e}{-}16$.

It is clearly shown in Figure 7 that the best performance profile is the one held by `dd_phi`, our routine, closely followed by `exptayotf` and `dd_ts`. In fact the data show that `dd_phi`, `exptayotf` and `dd_ts` have respectively the 87.5%, 84.3% and 80.6% probability of committing an error smaller than 50 times `eps`. This numbers increase to 96.3%, 92.6% and 90.7% probability when considering an error smaller than 100 times `eps`. But what is more important is that while `dd_phi` presents a 99% chance of committing an error smaller than about 145 times `eps` while `dd_ts` presents a 99% chance of committing an error smaller than about 354 times `eps` and `exptayotf` presents a 99% chance of committing an error smaller than 499 times `eps`.

6 Conclusions

From the numerical tests we conclude that the new routine `dd_phi` is more stable, accurate and fast than the competitors existing nowadays. This is true both for the computation of the divided differences over an interpolation sequence $z = (z_0, z_1, \ldots, z_n)$ and for the computation of the whole divided difference table.

In conclusion we can state that the thorough analysis of the matrix representing the operator mapping the coefficients of a polynomial $p(x)$ in the monomial basis $\mathcal{M}$ into those in the Newton basis $\mathcal{N}$ we carried on in the first part of this work gave us a powerful theoretical tool for building new algorithms.

The next step is going to be to exploit the structure of the matrices $\mathbf{E}(z_0, z_1, \ldots, z_n)$ and $\mathbf{H}(z_0, z_1, \ldots, z_n)$ in order to tackle difficult problems of interest in the field of numerical polynomial interpolation. Furthermore, as a future work, the plan is to develop brand new routines for the computation of the divided differences of other analytic functions of interest such as the trigonometric or logarithmic functions.

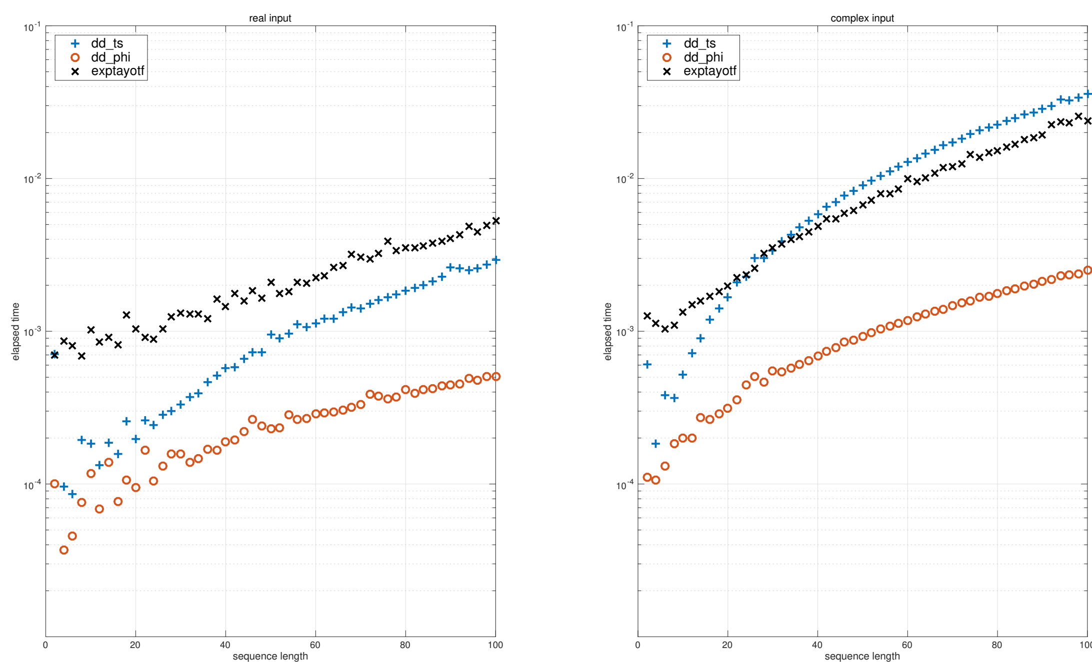


Figure 2: Average elapsed time to compute the divided differences table for the sequences described in s1. and s2.

References

[1] A. H. Al-Mohy, N. J. Higham, Computing the action of the matrix exponential with an application to exponential integrators, SIAM J. Sci. Comput., 33 (2) 488–511, 2011.

[2] M. Caliari, Accurate evaluation of divided differences for polynomial interpolation of exponential propagators, Computing, 80, 2, 189–201, 2007.

[3] M. Caliari, P. Kandolf, A. Ostermann, S. Rainer, The Leja method revisited: backward error analysis for the matrix exponential, SIAM J. Sci. Comput., 38 (3), A1639–A1661, 2016.

[4] M. Caliari, P. Kandolf, F. Zivcovich, Backward error analysis of polynomial approximations for computing the action of the matrix exponential, BIT Num. Math., 58, 4, 907–935, 2018.

[5] M. Caliari, F. Zivcovich, On-the-fly backward error estimate for matrix exponential approximation by Taylor algorithm, Journal of Comp. App. Math., 346, 532–548, 2019.

[6] W. Gander, Change of basis in polynomial interpolation, Linear Algebra Appl., 12, 769–778, 2005.

[7] F. Leja, Sur certaines suites liées aux ensembles plans et leur application à la reprèsentation conforme, Ann. Polon. Math., 4, 8–13, 1957.

[8] M. López–Fernández, S.A. Sauter, Fast and Stable Contour Integration for High Order Divided Differences via Elliptic Functions, Math. Comput., 84, 1291–1315, 2015

[9] A.C. McCurdy, Accurate computation of divided differences, University of California — ERL, 1980.

[10] A.C. McCurdy, K.C. Ng and B.N. Parlett, Accurate computation of divided differences of the exponential function, University of California — Berkeley, CPAM-160, 1983.

[11] G. Opitz, Z. Angew. Math. Mech., 44, Steigungsmatrizen, T52–T54, 1964.

[12] B.N. Parlett, A recurrence among the Elements of Functions of Triangular Matrices, Linear Algebra Appl., 14, 117–121, 1976.

[13] L. Reichel: Newton interpolation at Leja points., BIT, 30, 332–346, 1990.

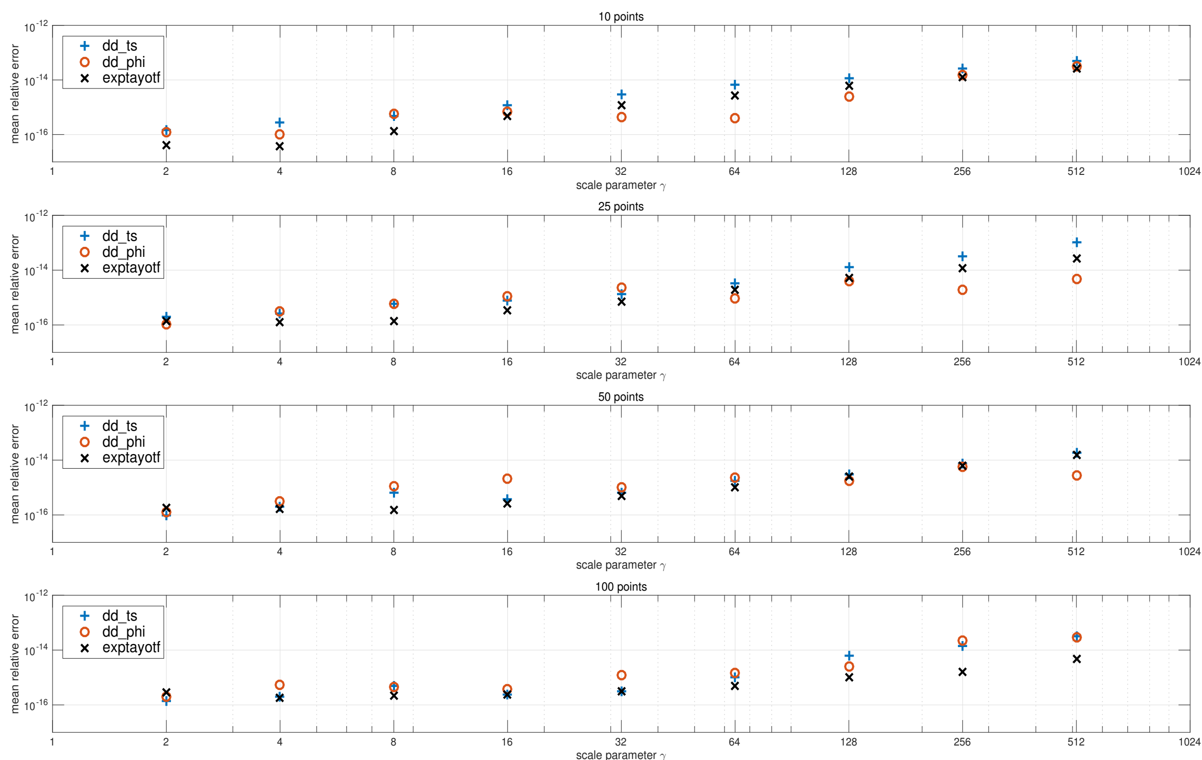


Figure 3: Mean average error committed by the three routines as the scale parameter γ increases over the sequences described in a3., i.e. with the n Chebyshev points over the interval $[-1, 1]$ reordered à la Leja.

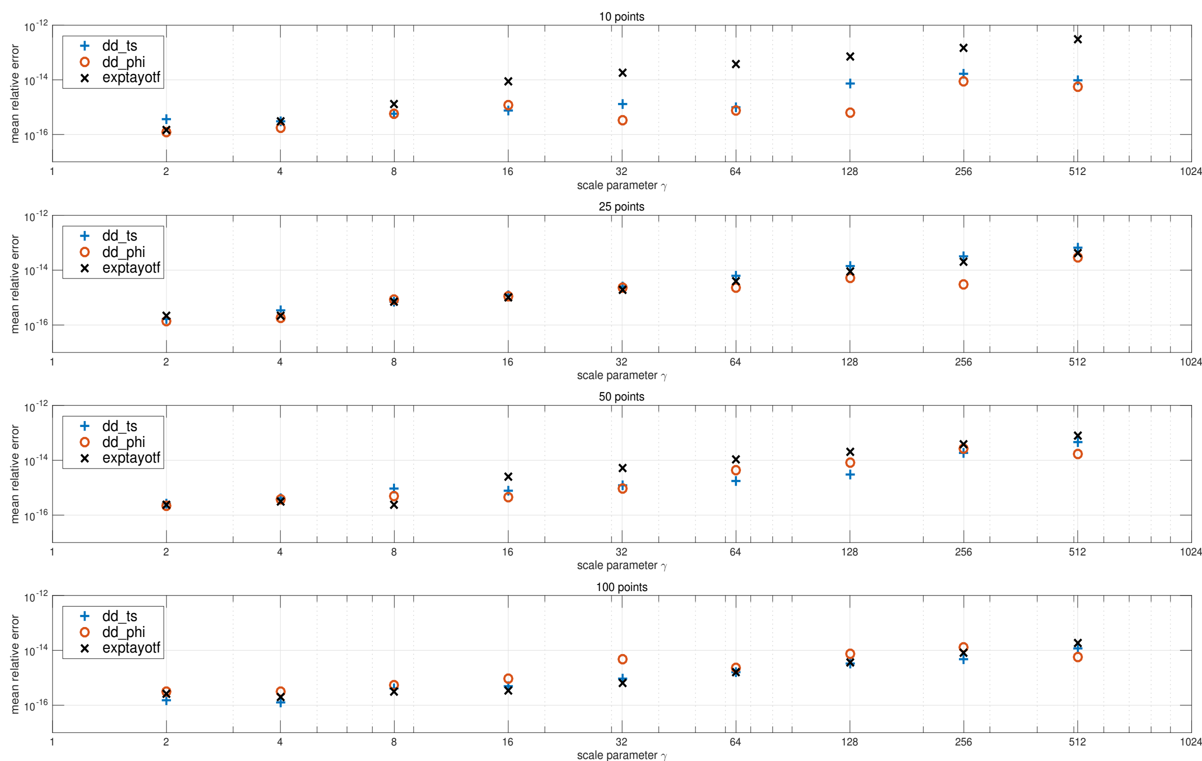


Figure 4: Mean average error committed by the three routines as the scale parameter γ increases over the sequences described in a4., i.e. with the n Leja points over the interval $[-1, 1]$.

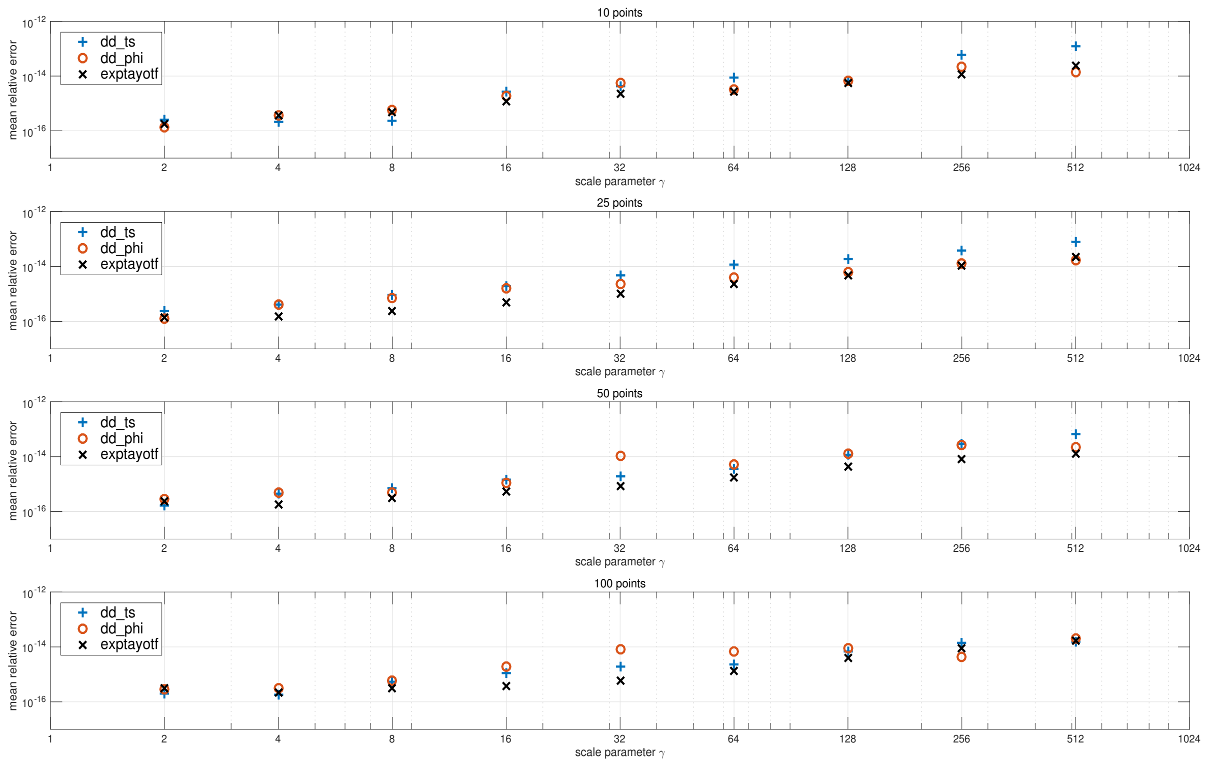


Figure 5: Mean average error committed by the three routines as the scale parameter γ increases over the sequences described in a5., i.e. with the n Leja points over the closed complex unit disk.

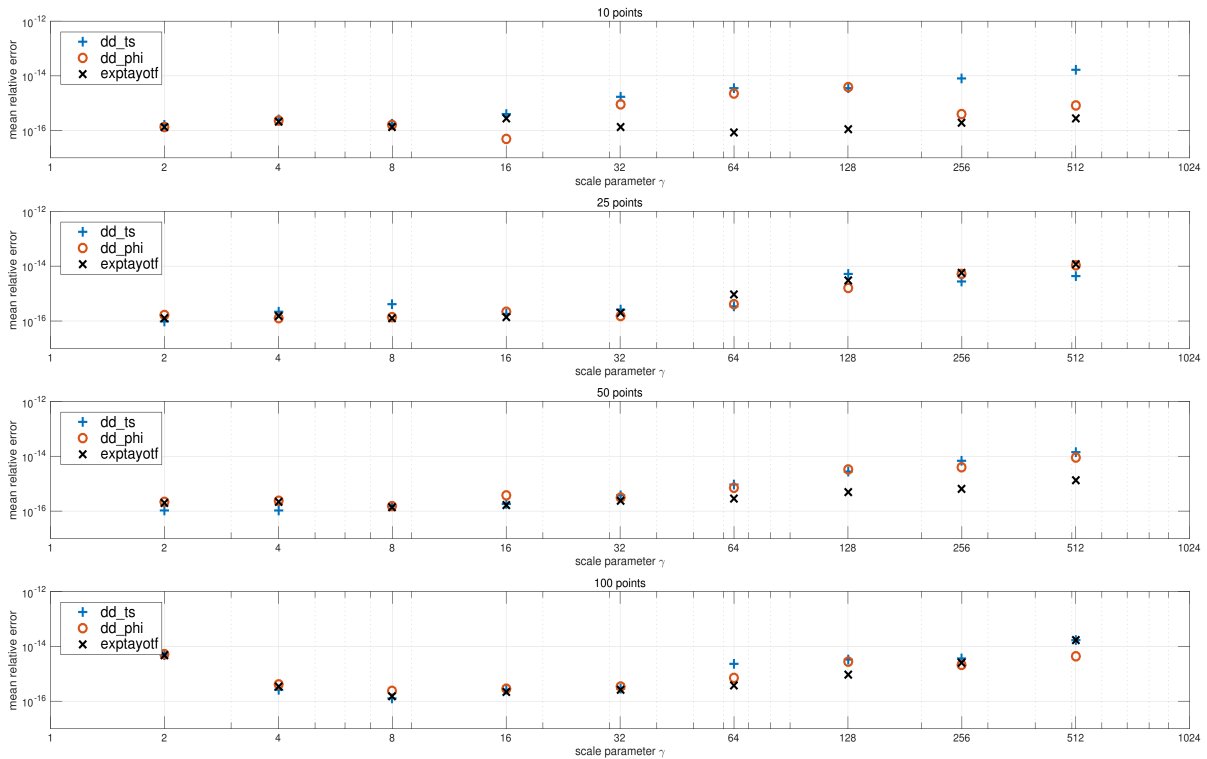


Figure 6: Mean average error committed by the three routines as the scale parameter γ increases over the sequences described in a6., i.e. with coalescing points $z_i = 2^{-i}$.

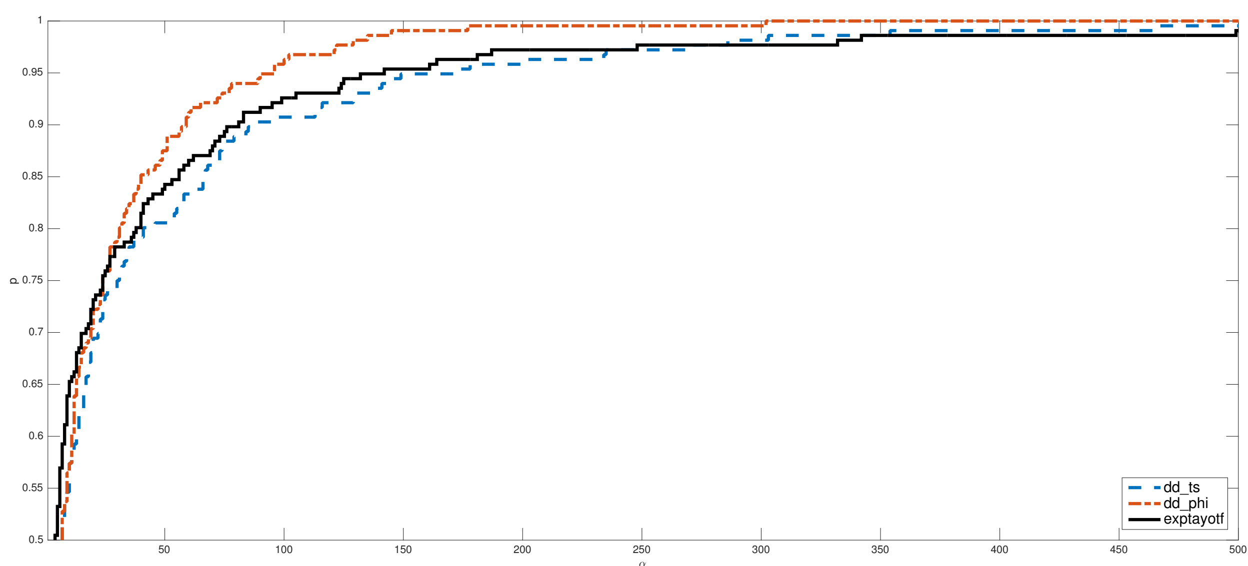


Figure 7: Same data as Figures from 3 to 6 together with data relative to the sequences described in 1. and 2. presented as performance profile.