Numerical Algorithms (2023) 92:2483–2508
https://doi.org/10.1007/s11075-022-01399-4

ORIGINAL PAPER

A μ-mode BLAS approach for multidimensional tensor-structured problems

Marco Caliari[1] · Fabio Cassini[2] · Franco Zivcovich[3]

Received: 27 December 2021 / Accepted: 17 August 2022 / Published online: 4 October 2022
© The Author(s) 2022

Abstract
In this manuscript, we present a common tensor framework which can be used to generalize one-dimensional numerical tasks to *arbitrary* dimension d by means of tensor product formulas. This is useful, for example, in the context of multivariate interpolation, multidimensional function approximation using pseudospectral expansions and solution of stiff differential equations on tensor product domains. The key point to obtain an efficient-to-implement BLAS formulation consists in the suitable usage of the μ-mode product (also known as tensor-matrix product or mode-n product) and related operations, such as the Tucker operator. Their MathWorks MATLAB®/GNU Octave implementations are discussed in the paper, and collected in the package KronPACK. We present numerical results on experiments up to dimension six from different fields of numerical analysis, which show the effectiveness of the approach.

Keywords μ-mode product · Tensor-structured problems · Exponential of Kronecker sum · ADI preconditioners · Multivariate interpolation · Multidimensional spectral transforms

✉ Marco Caliari
marco.caliari@univr.it

Fabio Cassini
fabio.cassini@unitn.it

Franco Zivcovich
franco.zivcovich@sorbonne-universite.fr

[1] Department of Computer Science, University of Verona, Verona, Italy

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

[3] Laboratoire Jacques–Louis Lions, Sorbonne University, Paris, France

1 Introduction

Many one-dimensional tasks in numerical analysis can be generalized to a two-dimensional formulation by means of tensor product formulas. This is the case, for example, in the context of spectral decomposition or interpolation of multivariate functions. Indeed, the one-dimensional formula

$$s_i = \sum_{j=1}^{m} t_j \ell_{ij}, \quad 1 \le i \le n,$$

where the values t_j are linearly combined to obtain the values s_i (i.e., $\boldsymbol{s} = \boldsymbol{L}\boldsymbol{t}$, with $\boldsymbol{s} = (s_i) \in \mathbb{C}^n$, $\boldsymbol{t} = (t_j) \in \mathbb{C}^m$, and $L = (\ell_{ij}) \in \mathbb{C}^{n\times m}$), can be easily extended to the two-dimensional case as

$$s_{i_1 i_2} = \sum_{j_2=1}^{m_2} \sum_{j_1=1}^{m_1} t_{j_1 j_2} \ell^1_{i_1 j_1} \ell^2_{i_2 j_2}, \quad 1 \le i_1 \le n_1, \quad 1 \le i_2 \le n_2. \tag{1}$$

The meaning of the involved scalar quantities depends on the specific example under consideration. In any case, a straightforward implementation of formula (1) requires four nested for-loops, with a resulting computational cost of $\mathcal{O}(n^4)$ (if, for simplicity, we consider $m_1 = m_2 = n_1 = n_2 = n$). On the other hand, formula (1) can be written equivalently in matrix formulation as

$$\boldsymbol{S} = L_1 \boldsymbol{T} L_2^{\mathsf{T}}, \tag{2}$$

where $L_1 = (\ell^1_{i_1 j_1}) \in \mathbb{C}^{n_1\times m_1}$, $L_2 = (\ell^2_{i_2 j_2}) \in \mathbb{C}^{n_2\times m_2}$, $\boldsymbol{T} = (t_{j_1 j_2}) \in \mathbb{C}^{m_1\times m_2}$ and $\boldsymbol{S} = (s_{i_1 i_2}) \in \mathbb{C}^{n_1\times n_2}$. The usage of formula (2) requires two separate matrix-matrix products as floating point operations, each of which can be implemented with three nested for-loops: this approach reduces then the cost of computing the elements of $\boldsymbol{S}$ to $\mathcal{O}(n^3)$. On the other hand, a more efficient way to realize formula (2) is to exploit optimized Basic Linear Algebra Subprograms (BLAS) [1, 2, 3, 4], which are a set of numerical linear algebra routines that perform the just mentioned matrix operations with a level of efficiency close to the theoretical hardware limit. A performance

Table 1 Wall-clock time (in seconds) for the computation of the values $s_{i_1 i_2}$ in formula (1) with increasing size $m_1 = m_2 = n_1 = n_2 = n$ and different approaches, using MathWorks MATLAB® R2019a. The input values are standard normal distributed random numbers

	$n=50$	$n=100$	$n=200$	$n=400$
Nested for-loops	1.8e-2	2.8e-1	4.8e0	8.0e1
Matrix-matrix products (for-loops)	7.8e-4	5.5e-3	4.9e-2	3.9e-1
Matrix-matrix products (BLAS)	2.1e-5	5.6e-5	1.7e-4	1.2e-3

comparison of the three approaches to compute the values $s_{i_1 i_2}$ in MATLAB[1] language, for increasing size of the task, is given in Table 1. As expected, for all the values of n under study, the most efficient way to compute the elements of $\boldsymbol{S}$ is realizing formula (2) through the BLAS approach. Remark that the considerations on the complexity cost and BLAS efficiency are basically language-independent, and apply for other interpreted or compiled languages as well, like PYTHON, JULIA, R, FORTRAN, and C++. For clarity of exposition and simplicity of presentation of the codes, we will use in this manuscript, from now on, MATLAB programming language.

In other contexts, such as numerical solution of (stiff) differential equations on two-dimensional tensor product domains by means of exponential integrators or preconditioned iterative methods, it is required to compute quantities like

$$\mathrm{vec}(\boldsymbol{S}) = (L_2 \otimes L_1)\mathrm{vec}(\boldsymbol{T}), \tag{3}$$

being again L_1, L_2, $\boldsymbol{T}$ and $\boldsymbol{S}$ matrices of suitable size whose meaning depends on the specific example under consideration. Here $\otimes$ denotes the standard Kronecker product of two matrices, while vec represents the vectorization operator, see Appendix for their formal definitions. A straightforward implementation of formula (3) would need to assemble the large-sized matrix $L_2 \otimes L_1$. If, for simplicity, we consider again $m_1 = m_2 = n_1 = n_2 = n$, this approach requires a storage and a computational cost of $\mathcal{O}(n^4)$, which is impractical. However, owing to the properties of the Kronecker product (see Appendix), we can see that formula (3) is equivalent to formula (2). Therefore, all the considerations made for the previous example on the employment of optimized BLAS apply also in this case.

The aim of this work is to provide a common framework for generalizing formula (2) in *arbitrary* dimension d, which will result in an efficient BLAS realization of the underlying task. This is very useful in the context of solving tensor-structured problems which may arise from different scientific and engineering fields. The pursued approach is illustrated in detail in Section 2, in which we present the μ-mode product and some associated operations (the Tucker operator, in particular), both from a theoretical and a practical point of view. These operations are widely known by the tensor algebra community, but their usage is mostly restricted in the context of tensor decompositions (see [5, 6]). Then, we proceed in Section 3 by describing more precisely the one- and two-dimensional formulations of the problems mentioned in this section, as well as their generalization to the d-dimensional case in terms of μ-mode products. We collect in Section 4 the related numerical experiments and we finally draw the conclusions in Section 5.

All the functions and the scripts needed to perform the relevant tensor operations and to reproduce the numerical examples of this manuscript are contained in our MATLAB package KronPACK.[2]

[1] We refer to MATLAB as the common language interpreted by the softwares MathWorks MATLAB® and GNU Octave, for instance

[2] The software is available from Netlib (http://www.netlib.org/numeralgo/) as the `na58` package. A maintained version, freely distributed under the MIT license, is available at https://github.com/caliarim/KronPACK

2 The μ-mode product and its applications

In order to generalize formula (2) to the d-dimensional case, we rely on some concepts from tensor algebra (see [5, 6] for more details). Throughout this section, we assume that $\boldsymbol{T} \in \mathbb{C}^{m_1\times\cdots\times m_d}$ is an order-d tensor whose elements are either denoted by $t_{j_1\ldots j_d}$ or by $\boldsymbol{T}(j_1,\ldots,j_d)$.

Definition 2.1 A *μ-fiber* of $\boldsymbol{T}$ is a vector in $\mathbb{C}^{m_\mu}$ obtained by fixing every index of the tensor but the μth.

A μ-fiber is nothing but a generalization of the concept of rows and columns of a matrix. Indeed, for an order-2 tensor (i.e., a matrix), 1-fibers are the columns, while 2-fibers are the rows. On the other hand, for an order-3 tensor, 1-fibers are the column vectors, 2-fibers are the row vectors while 3-fibers are the so-called "page" or "tube" vectors, which means vectors along the third dimension.

Definition 2.2 The *μ-matricization* of $\boldsymbol{T}$, denoted by $T^{(\mu)} \in \mathbb{C}^{m_\mu\times m_1\cdots m_{\mu-1}m_{\mu+1}\cdots m_d}$, is defined as the matrix whose columns are the μ-fibers of $\boldsymbol{T}$.

Remark that for an order-2 tensor the 1- and 2-matricizations simply correspond to the matrix itself and its transpose. In dimensions higher than two, the μ-matricization requires the concept of generalized transpose of a tensor and its unfolding into a matrix. The first operation is realized in MATLAB by the function `permute`, that we use to interchange μ-fibers with 1-fibers of the tensor $\boldsymbol{T}$. The second operation is performed by the `reshape` function, that we use to unfold the "transposed" tensor into the matrix $T^{(\mu)}$. In MATLAB, the anonymous function which performs the μ-matricization of a tensor `T`, given

```
m = size(T);
d = length(m);
```

can be written as

```
mumat = @(T,mu) reshape(permute(T,[mu,1:mu-1,mu+1:d]),...
          m(mu),prod(m([1:mu-1,mu+1:d])));
```

By means of μ-fibers, it is possible to define the following operation.

Definition 2.3 Let $L \in \mathbb{C}^{n\times m_\mu}$ be a matrix. The *μ-mode product* of $\boldsymbol{T}$ with L, denoted by $\boldsymbol{S} = \boldsymbol{T} \times_\mu L$, is the tensor $\boldsymbol{S} \in \mathbb{C}^{m_1\times\cdots\times m_{\mu-1}\times n\times m_{\mu+1}\times\cdots\times m_d}$ obtained by multiplying the matrix L onto the μ-fibers of $\boldsymbol{T}$.

From this definition, it appears clear that the μ-fiber $\boldsymbol{S}(j_1,\ldots,j_{\mu-1},\cdot,j_{\mu+1},\ldots,j_d)$ of $\boldsymbol{S}$ can be computed as the matrix-vector product of L and the μ-fiber $\boldsymbol{T}(j_1,\ldots,j_{\mu-1},\cdot,j_{\mu+1},\ldots,j_d)$. Therefore, the μ-mode product $\boldsymbol{T} \times_\mu L$ might be performed by calling $m_1\cdots m_{\mu-1}m_{\mu+1}\cdots m_d$ times level 2 BLAS. However, owing to the concept of matricization of a tensor introduced in Definition 2.2, it is possible to perform the same task more efficiently by using a single level 3 BLAS call. Indeed, the μ-mode product of $\boldsymbol{T}$ with L is just the tensor $\boldsymbol{S}$ such that

$$S^{(\mu)} = LT^{(\mu)}. \tag{4}$$

In particular, in the two-dimensional setting, the 1-mode product corresponds to the multiplication $L\boldsymbol{T}$, while the 2-mode product corresponds to $(L\boldsymbol{T}^{\mathsf{T}})^{\mathsf{T}} = \boldsymbol{T}L^{\mathsf{T}}$. In general, we can compute the matrix $S^{(\mu)}$ appearing in formula (4) as `L*mumat(T,mu)`, and in order to recover the tensor $\boldsymbol{S}$ from $S^{(\mu)}$ we need to invert the operations of unfolding and "transposing". This can be done easily with the aid of the MATLAB functions `reshape` and `ipermute`, respectively. All in all, given the value `n = size(L,1)`, the anonymous function that computes the μ-mode product of an order-d tensor $\boldsymbol{T}$ with L by a single matrix-matrix product can be written as

```
mump = @(T,L,mu) ipermute(reshape(L*mumat(T,mu),...
        [n,m([1:mu-1,mu+1:d])]),[mu,1:mu-1,mu+1:d]);
```

Notice that from formula (4) it appears clear that the computational cost of the μ-mode product, in terms of floating point operations, is $\mathcal{O}(nm_1 \cdots m_{\mathrm{d}})$.

One of the main applications of the μ-mode product is the so-called *Tucker operator*, which is implemented in KronPACK in the function `tucker`.

Definition 2.4 Let $L_\mu \in \mathbb{C}^{n_\mu \times m_\mu}$ be matrices, with $\mu = 1,\ldots,d$. The *Tucker operator* of $\boldsymbol{T}$ with $L_1,\ldots,L_d$ is the tensor $\boldsymbol{S} \in \mathbb{C}^{n_1 \times \cdots \times n_d}$ obtained by concatenating d consecutive μ-mode products with matrices L_μ, that is

$$\boldsymbol{S} = \boldsymbol{T} \times_1 L_1 \times_2 \cdots \times_d L_d. \tag{5}$$

We notice that the single element $s_{i_1\ldots i_d}$ of $\boldsymbol{S}$ in formula (5) turns out to be

$$s_{i_1\ldots i_d} = \sum_{j_d=1}^{m_d} \cdots \sum_{j_1=1}^{m_1} t_{j_1\ldots j_d} \prod_{\mu=1}^{d} \ell^{\mu}_{i_\mu j_\mu}, \quad 1 \le i_\mu \le n_\mu, \tag{6}$$

provided that $\ell^{\mu}_{i_\mu j_\mu}$ are the elements of L_μ. Hence, as formula (6) is clearly the generalization of formula (1) to the d-dimensional setting, formula (5) is the sought d-dimensional generalization of formula (2). We also notice that the Tucker operator (5) is invariant with respect to the ordering of the μ-mode products, and that the implicit ordering given by Definition 2.4 is equivalent to performing the sums in formula (6) starting from the innermost.

The Tucker operator is strictly connected with the Kronecker product of matrices applied to a vector.

Lemma 2.1 Let $L_\mu \in \mathbb{C}^{n_\mu \times m_\mu}$ be matrices, with $\mu = 1,\ldots,d$. Then, the elements of $\boldsymbol{S}$ in formula (5) are equivalently given by

$$\operatorname{vec}(\boldsymbol{S}) = (L_d \otimes \cdots \otimes L_1)\operatorname{vec}(\boldsymbol{T}). \tag{7}$$

Proof The μ-mode product satisfies the following property

$$\boldsymbol{S} = \boldsymbol{T} \times_1 L_1 \times_2 \cdots \times_d L_d \iff S^{(\mu)} = L_\mu T^{(\mu)} (L_d \otimes \cdots \otimes L_{\mu+1} \otimes L_{\mu-1} \otimes \cdots \otimes L_1)^{\mathsf{T}},$$

see [6]. Then, with $\mu = 1$ we obtain

$$\boldsymbol{S} = \boldsymbol{T} \times_1 L_1 \times_2 \cdots \times_d L_d \iff S^{(1)} = L_1 T^{(1)} (L_d \otimes \cdots \otimes L_2)^{\mathsf{T}}.$$

By means of the properties of the Kronecker product (see Appendix) we have then

$$S^{(1)} = L_1 T^{(1)} (L_d \otimes \cdots \otimes L_2)^{\mathsf{T}} \iff \operatorname{vec}(S^{(1)}) = (L_d \otimes \cdots \otimes L_1)\operatorname{vec}(T^{(1)})$$

and finally, by definition of vec operator,

$$\operatorname{vec}(S^{(1)}) = (L_d \otimes \cdots \otimes L_1)\operatorname{vec}(T^{(1)}) \iff \operatorname{vec}(\boldsymbol{S}) = (L_d \otimes \cdots \otimes L_1)\operatorname{vec}(\boldsymbol{T}).$$

□

Notice that formula (7) is precisely the d-dimensional generalization of formula (3). Hence, tasks written as in formula (7) can be equivalently stated and computed more efficiently again by formula (5), without assembling the large-sized matrix $L_d \otimes \cdots \otimes L_1$.

We can then summarize as follows: the *element-wise* formulation (6), the *tensor* formulation (5) and the *vector* formulation (7) can all be used to compute the entries of the tensor $\boldsymbol{S}$. However, in light of the considerations for the μ-mode product, only the tensor formulation can be efficiently computed by d calls of level 3 BLAS, with an overall computational cost of $\mathcal{O}(n^{d+1})$ for the case $m_\mu = n_\mu = n$. This is the reason why the relevant functions of our package KronPACK are based on formulation (5).

Remark 1 The implementation of a *single* μ-mode product in the function `mump` of KronPACK involves two explicit permutations of the tensor (except the 1-mode and the d-mode products, which are realized without explicitly permuting, thanks to the design of the function `reshape` in MATLAB). On the other hand, the function `tucker`, which realizes the Tucker operator (5), performs a composition of any pair of consecutive permutations, thus reducing their overall number. In fact, this is important when dealing with large-sized tensors, because the cost of permuting is not negligible due to the underlying alteration of the memory layout. For this reason, several algorithms which further reduce or completely avoid permutations in an efficient way have been developed (see, for instance, [7, 8, 9, 10]). In this context, for instance, it is possible to use the function `pagemtimes` to efficiently realize a "Loops-over-GEMMs" strategy. However, as this function has been recently introduced in MathWorks MATLAB® R2020b and it is still not available in the latest stable GNU Octave release 7.1.0, for compatibility reasons we do not follow this approach.

Notice that the definition of μ-mode product and its realization through the function `mump` can be easily extended to the case in which instead of a matrix L we have a *matrix-free* operator $\mathcal{L}$.

Definition 2.5 Let $\mathcal{L} : \mathbb{C}^{m_\mu} \to \mathbb{C}^n$ be an operator. Then the *μ-mode action* of $\boldsymbol{T}$ with $\mathcal{L}$, still denoted $\boldsymbol{S} = \boldsymbol{T} \times_\mu \mathcal{L}$, is the tensor $\boldsymbol{S} \in \mathbb{C}^{m_1 \times \cdots \times m_{\mu-1} \times n \times m_{\mu+1} \times \cdots \times m_d}$ obtained by the action of the operator $\mathcal{L}$ on the μ-fibers of $\boldsymbol{T}$.

In MATLAB, if the operator $\mathcal{L}$ is represented by the function `Lfun` which operates on columns, we can implement the μ-mode action by

```
mumpfun=@(T,Lfun,mu) ipermute(reshape(Lfun(mumat(T,mu)),...
            [n,m([1:mu-1,mu+1:d])]),[mu,1:mu-1,mu+1:d]);
```

The corresponding generalization of the Tucker operator, denoted again by

$$\boldsymbol{S} = \boldsymbol{T} \times_1 \mathcal{L}_1 \times_2 \cdots \times_d \mathcal{L}_d \tag{8}$$

and implemented in KronPACK in the function `tuckerfun`, follows straightforwardly. Clearly, in this case, some properties of the Tucker operator (5), such as the aforementioned invariance with respect to the ordering of the μ-mode product operations, may not hold anymore for generic operators $\mathcal{L}_\mu$. Generalization (8) is useful in some instances, see Remark 3 and Section 4.2 for an example. We remark that such an extension is not available in some other popular tensor algebra toolboxes, such as *Tensor Toolbox for MATLAB* [11] — which does not have GNU Octave support, too — and *Tensorlab* [12], both of which are more devoted to tensor decomposition and related topics.

The μ-mode product is also useful for computing the action of the Kronecker sum (see Appendix for its definition) of the L_μ matrices to a vector $\boldsymbol{v}$, that is

$$(L_d \oplus \cdots \oplus L_1)\boldsymbol{v} = \operatorname{vec}\left(\sum_{\mu=1}^{d} (\boldsymbol{V} \times_\mu L_\mu)\right), \tag{9}$$

where $\boldsymbol{v} = \operatorname{vec}(\boldsymbol{V})$. In fact, as it can be noticed from formula (4), the identity matrix is the identity element of the μ-mode product. Combining this observation with Lemma 2.1, we easily obtain formula (9). In our package KronPACK, the matrix resulting from the Kronecker sum on the left hand side of equality (9) can be computed as `kronsum(L)`, where `L` is the cell array containing L_μ in `L{mu}`. On the other hand, its action on $\boldsymbol{v}$ can be computed equivalently in tensor formulation, without forming the matrix itself, by `kronsumv(V,L)`.

3 Problems formulation in *d* dimensions

In this section we discuss in more detail the problems that were briefly introduced in Section 1. Their generalization to arbitrary dimension d is addressed thanks to the common framework presented in Section 2.

3.1 Pseudospectral decomposition

Suppose that a function $f : R \to \mathbb{C}$, with $R \subseteq \mathbb{R}$, can be expanded into a series

$$f(x) = \sum_{i=1}^{\infty} f_i \phi_i(x),$$

where f_i are complex scalar coefficients and $\phi_i(x)$ are complex functions orthonormal with respect to the standard $L^2(R)$ inner product, i.e.,

$$\int_R \phi_i(x)\overline{\phi_j(x)}dx = \delta_{ij}, \quad \forall i,j.$$

Then, the spectral coefficients f_i are defined by

$$f_i = \int_R f(x)\overline{\phi_i(x)}dx,$$

and can be approximated by a quadrature formula. Usually, in this context, specific Gaussian quadrature formulas are employed, whose node and weights vary depending on the chosen family of basis functions. If we consider q quadrature nodes ξ^k and weights w^k, we can compute the first m *pseudospectral* coefficients by

$$\hat{f}_i = \sum_{k=1}^{q} f(\xi^k)\overline{\phi_i(\xi^k)}w^k \approx f_i, \quad 1 \le i \le m.$$

By collecting the values $\overline{\phi_i(\xi^k)}$ in position (i,k) of the matrix $\Psi \in \mathbb{C}^{m\times q}$ and the values $f(\xi^k)w^k$ in the vector $\boldsymbol{f_w}$, we can compute the pseudospectral coefficients by means of the single matrix-vector product

$$\hat{\boldsymbol{f}} = \Psi \boldsymbol{f}_{\boldsymbol{w}}.$$

In the two-dimensional case, the coefficients of a pseudospectral expansion in a tensor product basis (see, for instance, [13, Ch. 6.10]) are given by

$$\hat{f}_{i_1 i_2} = \sum_{k_2=1}^{q_2}\sum_{k_1=1}^{q_1} f\left(\xi_1^{k_1}, \xi_2^{k_2}\right)\overline{\phi_{i_1}^1(\xi_1^{k_1})\phi_{i_2}^2(\xi_2^{k_2})}w_1^{k_1}w_2^{k_2},$$

which can be efficiently computed as

$$\hat{\boldsymbol{F}} = \Psi_1 \boldsymbol{F}_{\boldsymbol{W}} \Psi_2^{\mathsf{T}},$$

where $\Psi_\mu \in \mathbb{C}^{m_\mu \times q_\mu}$ has element $\overline{\phi_{i_\mu}^{\mu}(\xi_\mu^{k_\mu})}$ in position (i_μ,k_μ), with $\mu = 1,2$, and $\boldsymbol{F}_{\boldsymbol{W}}$ is the matrix with element $f(\xi_1^{k_1}, \xi_2^{k_2})w_1^{k_1}w_2^{k_2}$ in position (k_1,k_2).

In general, the coefficients of a d-dimensional pseudospectral expansion in a tensor product basis are given by

$$\hat{f}_{i_1\ldots i_d} = \sum_{k_d=1}^{q_d}\cdots\sum_{k_1=1}^{q_1} f(\xi_1^{k_1},\ldots,\xi_d^{k_d})\overline{\phi_{i_1}^1(\xi_1^{k_1})}\cdots\overline{\phi_{i_d}^d(\xi_d^{k_d})}w_1^{k_1}\cdots w_d^{k_d}.$$

In tensor formulation, the coefficients can be computed as (see formulas (5) and (6))

$$\hat{\boldsymbol{F}} = \boldsymbol{F_W} \times_1 \Psi_1 \times_2 \cdots \times_d \Psi_d,$$

where Ψ_μ is the transform matrix with element $\overline{\phi^\mu_{i_\mu}(\xi^{k_\mu}_\mu)}$ in position (i_μ,k_μ), and we collect in the order-d tensors $\hat{\boldsymbol{F}}$ and $\boldsymbol{F_W}$ the values $\hat{f}_{i_1\ldots i_d}$ and $f(\xi^{k_1}_1, \ldots, \xi^{k_d}_d)w^{k_1}_1 \cdots w^{k_d}_d$, respectively. The corresponding pseudospectral approximation of $f(\boldsymbol{x})$ is

$$\hat{f}(\boldsymbol{x}) = \sum_{i_d=1}^{m_d} \cdots \sum_{i_1=1}^{m_1} \hat{f}_{i_1\ldots i_d} \phi^1_{i_1}(x_1) \cdots \phi^d_{i_d}(x_d), \tag{10}$$

where $\boldsymbol{x} = (x_1,\ldots,x_d)$. An application to Hermite–Laguerre–Fourier function decomposition is given in Section 4.2.

3.2 Function approximation

Suppose we are given an approximation of a univariate function $f(x)$ in the form

$$\tilde{f}(x) = \sum_{i=1}^{m} c_i \phi_i(x) \approx f(x), \tag{11}$$

where c_i are scalar coefficients and $\phi_i(x)$ are generic (basis) functions. This is the case, for example, in the context of function interpolation or pseudospectral expansions. We are interested in the evaluation of formula (11) at given points x^ℓ, with $1 \leq \ell \leq n$. This can be easily realized in a single matrix-vector product: indeed, if we collect the coefficients c_i in the vector $\boldsymbol{c} \in \mathbb{C}^m$ and we form the matrix $\Phi \in \mathbb{C}^{n\times m}$ with element $\phi_i(x^\ell)$ in position (ℓ,i), the sought evaluation is given by

$$\tilde{\boldsymbol{f}} = \Phi \boldsymbol{c},$$

being $\tilde{\boldsymbol{f}} \in \mathbb{C}^n$ the vector containing the approximated function at the given set of evaluation points.

The extension of formula (11) to the tensor product bivariate case is straightforward (see, for instance, [14, Ch. XVII]). Indeed, in this case the approximating function is given by

$$\tilde{f}(x_1, x_2) = \sum_{i_2=1}^{m_2} \sum_{i_1=1}^{m_1} c_{i_1 i_2} \phi^1_{i_1}(x_1) \phi^2_{i_2}(x_2) \approx f(x_1, x_2), \tag{12}$$

where $c_{i_1 i_2}$ represent scalar coefficients and $\phi^\mu_{i_\mu}(x_\mu)$ the (univariate) basis function, with $1 \leq i_\mu \leq m_\mu$ and $\mu = 1,2$. Then, given a Cartesian grid of points $(x^{\ell_1}_1, x^{\ell_2}_2)$, with $1 \leq \ell_\mu \leq n_\mu$, the evaluation of approximation (12) can be computed efficiently in matrix formulation by

$$\tilde{\boldsymbol{F}} = \Phi_1 \boldsymbol{C} \Phi_2^{\mathsf{T}}.$$

Here we collected the function evaluations $\tilde{f}(x_1^{\ell_1}, x_2^{\ell_2})$ in the matrix $\tilde{\boldsymbol{F}}$, we formed the matrices $\Phi_\mu \in \mathbb{C}^{n_\mu \times m_\mu}$ of element $\phi_{i_\mu}^{\mu}(x_\mu^{\ell_\mu})$ in position (ℓ_μ, i_μ), and we let $\boldsymbol{C}$ be the matrix of element $c_{i_1 i_2}$ in position (i_1, i_2).

In general, the approximation of a d-variate function f with tensor product basis functions is given by

$$\tilde{f}(\boldsymbol{x}) = \sum_{i_d=1}^{m_d} \cdots \sum_{i_1=1}^{m_1} c_{i_1 \ldots i_d} \phi_{i_1}^{1}(x_1) \cdots \phi_{i_d}^{d}(x_d) \approx f(\boldsymbol{x}), \tag{13}$$

where $c_{i_1 \ldots i_d}$ represent scalar coefficients while $\phi_{i_\mu}^{\mu}(x_\mu)$ the (univariate) basis functions, with $1 \leq i_\mu \leq m_\mu$. Then, given a Cartesian grid of points $(x_1^{\ell_1}, \ldots, x_d^{\ell_d})$, with $1 \leq \ell_\mu \leq n_\mu$, the evaluation of approximation (13) can be expressed in tensor formulation as

$$\tilde{\boldsymbol{F}} = \boldsymbol{C} \times_1 \Phi_1 \times_2 \cdots \times_d \Phi_d, \tag{14}$$

see formulas (5) and (6). Here we denote Φ_μ the matrix with element $\phi_{i_\mu}^{\mu}(x_\mu^{\ell_\mu})$ in position (ℓ_μ, i_μ), and we collect in the order-d tensors $\boldsymbol{C}$ and $\tilde{\boldsymbol{F}}$ the coefficients and the resulting function approximation at the evaluation points, respectively. We present an application to barycentric multivariate interpolation in Section 4.3.

Remark 2 Clearly, formula (14) can be employed to evaluate a pseudospectral approximation (10) at a generic Cartesian grid of points, by properly defining the involved tensor $\boldsymbol{C}$ and matrices Φ_μ. In the context of direct and inverse spectral transforms, for example for the effective numerical solution of differential equations (see [15]), one could be interested in the evaluation of pseudospectral decompositions at the same grid of quadrature points $(\xi_1^{k_1}, \ldots, \xi_d^{k_d})$ used to approximate the spectral coefficients. Under standard hypothesis, this can be done by applying formula (14) with matrices $\Phi_\mu = \Psi_\mu^*$, where the symbol $*$ denotes the conjugate transpose. Without forming explicitly the matrices Φ_μ, the desired evaluation can be computed using the matrices Ψ_μ by means of the KronPACK function `cttucker`.

Remark 3 Several functions which perform the whole one-dimensional procedure of approximating a function and evaluating it on a set of points, given suitable inputs, are available. This is the case, for example in the interpolation context, of the MATLAB built-in functions `spline`, `interp1` (that performs different kinds of one-dimensional interpolations), and `interpft` (which performs a resample of the input values by means of FFT techniques), or of the functions provided by the QIBSH++ library [16] in the approximation context. Yet, it is possible to extend the usage of this kind of functions to the approximation in the d-dimensional tensor setting by means of concatenations of μ-mode actions (see Definition 2.5), yielding the generalization of the Tucker operator (8). In practice, we can perform this task with the KronPACK function `tuckerfun`, see the numerical example in Section 4.2.

3.3 Action of the matrix exponential

Suppose we want to solve the linear Partial Differential Equation (PDE)

$$\begin{cases} \partial_t u(t,x) = \mathcal{A}u(t,x), \quad t>0, \quad x \in \Omega \subset \mathbb{R}, \\ u(0,x) = u_0(x), \end{cases} \tag{15}$$

coupled with suitable boundary conditions, where $\mathcal{A}$ is a linear time-independent spatial (integer or fractional) differential operator, typically stiff. The application of the method of lines to equation (15), by discretizing first the spatial variable, e.g., by finite differences or spectral differentiation, leads to the system of Ordinary Differential Equations (ODEs)

$$\begin{cases} \boldsymbol{u}'(t) = A\boldsymbol{u}(t), \quad t>0, \\ \boldsymbol{u}(0) = \boldsymbol{u}_0, \end{cases} \tag{16}$$

for the unknown vector $\boldsymbol{u}(t)$. Here, $A \in \mathbb{C}^{n\times n}$ is the matrix which approximates the differential operator $\mathcal{A}$ on the grid points x^ℓ, with $1 \le \ell \le n$. The exact solution of system (16) is obviously $\boldsymbol{u}(t) = \exp(tA)\boldsymbol{u}_0$ and, if the size of A allows, it can be effectively computed by Padé or Taylor approximations (see [17, 18]). If the size of A is too large, then one has to rely on algorithms to approximate the action of the matrix exponential $\exp(tA)$ on the vector $\boldsymbol{u}_0$. Examples of such methods are [19, 20, 21, 22].

Suppose now we want to solve instead

$$\begin{cases} \partial_t u(t,x_1,x_2) = \mathcal{A}u(t,x_1,x_2), \quad t>0, \quad (x_1,x_2) \in \Omega \subset \mathbb{R}^2, \\ u(0,x_1,x_2) = u_0(x_1,x_2), \end{cases} \tag{17}$$

coupled again with suitable boundary conditions. If PDE (17) admits a Kronecker structure, such as for some linear Advection–Diffusion–Absorption (ADA) equations on tensor product domains or linear Schrödinger equations with a potential in Kronecker form (see [15] for more details and examples), then the method of lines yields the system of ODEs

$$\begin{cases} \boldsymbol{u}'(t) = \left(I_2 \otimes A_1 + A_2 \otimes I_1\right)\boldsymbol{u}(t), \quad t>0, \\ \boldsymbol{u}(0) = \boldsymbol{u}_0. \end{cases} \tag{18}$$

Here A_μ, with $\mu = 1,2$, represent the one-dimensional stencil matrices corresponding to the discretization of the one-dimensional differential operators that constitute $\mathcal{A}$ on the grid points $x_\mu^{\ell_\mu}$, with $1 \le \ell_\mu \le n_\mu$. Moreover, the notation I_μ stands for identity matrices of size n_μ, and the component $\ell_1 + (\ell_2 - 1)n_1$ of $\boldsymbol{u}$ corresponds to the grid point $(x_1^{\ell_1}, x_2^{\ell_2})$, that is

$$u_{\ell_1+(\ell_2-1)n_1}(t) \approx u(t, x_1^{\ell_1}, x_2^{\ell_2}).$$

This, in turn, is consistent with the linearization of the indexes of the vec operator defined in Appendix.

Clearly, the solution of system (18) is given by

$$\boldsymbol{u}(t) = \exp\left(t(I_2 \otimes A_1 + A_2 \otimes I_1)\right)\boldsymbol{u}_0, \tag{19}$$

which again could be computed by any method to compute the action of the matrix exponential on a vector. Remark that, since the matrices $I_2 \otimes A_1$ and $A_2 \otimes I_1$ commute and using the properties of the Kronecker product (see Appendix), one could write everything in terms of the exponentials of the *small-sized* matrices A_μ. Indeed, we have

$$\begin{aligned}\boldsymbol{u}(t) &= \exp\left(t(I_2 \otimes A_1 + A_2 \otimes I_1)\right)\boldsymbol{u}_0 = \exp(t(I_2 \otimes A_1))\exp(t(A_2 \otimes I_1))\boldsymbol{u}_0\\ &= \left(I_2 \otimes \exp(tA_1)\right)\left(\exp(tA_2) \otimes I_1\right)\boldsymbol{u}_0 = (\exp(tA_2) \otimes \exp(tA_1))\boldsymbol{u}_0.\end{aligned}$$

However, as in general the matrices $\exp(tA_\mu)$ are full, their Kronecker product results in a large and full matrix to be multiplied into $\boldsymbol{u}_0$, which is an extremely inefficient approach. Nevertheless, if we fully exploit the tensor structure of the problem, we can still compute the solution of the system efficiently just in terms of the exponentials $\exp(tA_\mu)$. Indeed, let $\boldsymbol{U}(t)$ be the $n_1 \times n_2$ matrix whose stacked columns form the vector $\boldsymbol{u}(t)$, that is

$$\operatorname{vec}(\boldsymbol{U}(t)) = \boldsymbol{u}(t).$$

Then, using this matrix notation and by means of the properties of the Kronecker product, problem (18) takes the form

$$\begin{cases}\boldsymbol{U}'(t) = A_1\boldsymbol{U}(t) + \boldsymbol{U}(t)A_2^{\mathsf{T}}, & t > 0,\\ \boldsymbol{U}(0) = \boldsymbol{U}_0,\end{cases}$$

and it is well-known (see [23]) that its solution can be computed in matrix formulation as

$$\boldsymbol{U}(t) = \exp(tA_1)\boldsymbol{U}_0\exp(tA_2)^{\mathsf{T}}.$$

In general, the d-dimensional version of solution (19) is

$$\boldsymbol{u}(t) = \exp\left(t\sum_{\mu=1}^{d}\left(I_d \otimes \cdots \otimes I_{\mu+1} \otimes A_\mu \otimes I_{\mu-1} \otimes \cdots \otimes I_1\right)\right)\boldsymbol{u}_0,$$

which can be written in more compact notation as

$$\boldsymbol{u}(t) = \exp\left(t\left(A_d \oplus \cdots \oplus A_1\right)\right)\boldsymbol{u}_0. \tag{20}$$

Here, A_μ are square matrices of size n_μ, and $\boldsymbol{u}_0$ is a vector of length $N = n_1 \cdots n_d$. Then, similarly to the two-dimensional case, we have

$$\boldsymbol{u}(t) = \exp\left(t(A_d \oplus \cdots \oplus A_1)\right)\boldsymbol{u}_0 = (\exp(tA_d) \otimes \cdots \otimes \exp(tA_1))\boldsymbol{u}_0.$$

Finally, using Lemma 2.1, we have

$$\boldsymbol{U}(t) = \boldsymbol{U}_0 \times_1 \exp(tA_1) \times_2 \cdots \times_d \exp(tA_d), \tag{21}$$

where $\boldsymbol{U}(t)$ and $\boldsymbol{U}_0$ are d-dimensional tensors such that $\boldsymbol{u}(t) = \operatorname{vec}(\boldsymbol{U}(t))$ and $\boldsymbol{u}_0 = \operatorname{vec}(\boldsymbol{U}_0)$. Hence, the action of the large-sized matrix exponential appearing in formula (20) can be computed by the Tucker operator (21) which just involves the small-sized matrix exponentials $\exp(tA_\mu)$. For an application in the context of solution of an ADA linear evolutionary equation with spatially variable coefficients, see Section 4.4.

3.4 Preconditioning of linear systems

Suppose we want to solve the semilinear PDE

$$\begin{cases} \partial_t u(t,x) = \mathcal{A}u(t,x) + f(t,u(t,x)), \quad t>0, \quad x \in \Omega \subset \mathbb{R}, \\ u(0,x) = u_0(x), \end{cases} \tag{22}$$

coupled with suitable boundary conditions, where $\mathcal{A}$ is a linear time-independent spatial differential operator and f is a nonlinear function. Using the method of lines, similarly to what led to system (16), we obtain

$$\begin{cases} \boldsymbol{u}'(t) = A\boldsymbol{u}(t) + \boldsymbol{f}(t,\boldsymbol{u}(t)), \quad t>0, \\ \boldsymbol{u}(0) = \boldsymbol{u}_0. \end{cases} \tag{23}$$

A common approach to integrating system (23) in time involves the use of IMplicit EXplicit (IMEX) schemes. For instance, the application of the well-known backward-forward Euler method with constant time step size τ leads to the solution of the linear system

$$M\boldsymbol{u}_{k+1} = \boldsymbol{u}_k + \tau\boldsymbol{f}(t_k,\boldsymbol{u}_k)$$

at every time step, where $M = (I - \tau A) \in \mathbb{C}^{n\times n}$ and I is an identity matrix of suitable size. If the space discretization allows (second order centered finite differences, for example), the system can then be solved by means of the very efficient Thomas algorithm. If, on the other hand, this is not the case, a suitable direct or (preconditioned) iterative method can be employed.

Let us consider now the two-dimensional version of the semilinear PDE (22), i.e.,

$$\begin{cases} \partial_t u(t,x_1,x_2) = \mathcal{A}u(t,x_1,x_2) + f(t,u(t,x_1,x_2)), \quad t>0, \quad (x_1,x_2) \in \Omega \subset \mathbb{R}^2, \\ u(0,x_1,x_2) = u_0(x_1,x_2), \end{cases} \tag{24}$$

again with suitable boundary conditions, $\mathcal{A}$ a linear time-independent spatial differential operator and f a nonlinear function. As for equation (17), if the PDE admits a Kronecker sum structure, the application of the method of lines leads to

$$\begin{cases} \boldsymbol{u}'(t) = (I_2 \otimes A_1 + A_2 \otimes I_1)\boldsymbol{u}(t) + \boldsymbol{f}(t,\boldsymbol{u}(t)), \quad t>0, \\ \boldsymbol{u}(0) = \boldsymbol{u}_0, \end{cases} \tag{25}$$

which can be integrated in time again by means of the backward-forward Euler method. The matrix of the resulting linear system to be solved at every time step is now

$$M = I_2 \otimes M_1 + M_2 \otimes I_1 = I_2 \otimes \left(\frac{1}{2}I_1 - \tau A_1\right) + \left(\frac{1}{2}I_2 - \tau A_2\right) \otimes I_1.$$

If we use an iterative method, we can obtain the action of the matrix M to a vector $\boldsymbol{v}$ as

$$M_1 \boldsymbol{V} + \boldsymbol{V} M_2^{\mathsf{T}} = \boldsymbol{V}_M, \quad \mathrm{vec}(\boldsymbol{V}) = \boldsymbol{v},$$

by observing that

$$M\boldsymbol{v} = \mathrm{vec}(\boldsymbol{V}_M).$$

Moreover, examples of effective preconditioners for this kind of linear systems are the ones of Alternating Direction Implicit (ADI) type (see [24]). In this case, we can use the product of the matrices arising from the discretization of equation (24) after neglecting all the spatial variables but one in the operator $\mathcal{A}$. We obtain then the preconditioner

$$(I_2 - \tau A_2) \otimes (I_1 - \tau A_1) = P_2 \otimes P_1 = P, \tag{26}$$

which is expected to be effective since $P = M + \mathcal{O}(\tau^2)$. In addition, the action of P^{-1} to a vector $\boldsymbol{v}$ can be efficiently obtained as

$$P_1^{-1} \boldsymbol{V} P_2^{-\mathsf{T}} = \boldsymbol{V}_{P^{-1}},$$

by noticing that

$$P^{-1}\boldsymbol{v} = (P_2^{-1} \otimes P_1^{-1})\boldsymbol{v} = \mathrm{vec}(\boldsymbol{V}_{P^{-1}}).$$

Remark 4 Another approach of solution to equation (25) would be to write the equivalent matrix formulation of the problem, i.e.,

$$\begin{cases} \boldsymbol{U}'(t) = A_1 \boldsymbol{U}(t) + \boldsymbol{U}(t) A_2^{\mathsf{T}} + \boldsymbol{F}(t, \boldsymbol{U}(t)), & t > 0, \\ \boldsymbol{U}(0) = \boldsymbol{U}_0, \end{cases}$$

and then apply appropriate algorithms to integrate it numerically, mainly based on the solution of Sylvester equations. This is the approach pursued, for example, in [25].

In general, for a d-dimensional semilinear problem with a Kronecker sum structure, the linear system to be solved at every time step has now matrix

$$M = M_d \oplus \cdots \oplus M_1, \quad M_\mu = \left(\frac{1}{d}I_\mu - \tau A_\mu\right).$$

Again, the action of the matrix M on a vector $\boldsymbol{v}$ can be computed without assembling the matrix (see equivalence (9)). Finally, an effective preconditioner for the linear system is a straightforward generalization of formula (26), i.e.,

$$(I_d - \tau A_d) \otimes \cdots \otimes (I_1 - \tau A_1) = P_d \otimes \cdots \otimes P_1 = P.$$

Similarly to the two-dimensional case, its inverse action to a vector $\boldsymbol{v}$ can be computed efficiently as

$$\boldsymbol{V} \times_1 P_1^{-1} \times_2 \cdots \times_d P_d^{-1} = \boldsymbol{V}_{P^{-1}}, \tag{27}$$

see Lemma 2.1. In our package KronPACK, formula (27) can be realized without explicitly inverting the matrices P_μ by using the function `itucker`. We notice that this is another feature not available in the tensor algebra toolboxes mentioned in Section 2. For an example of application of these techniques, in the context of solution of evolutionary diffusion–reaction equations, see Section 4.5.

We finally notice that there exist also specific techniques to solve linear systems in Kronecker form, usually arising in the discretization of time-independent differential equation, see for instance [26, 27].

4 Numerical experiments

We present in this section some numerical experiments of the proposed μ-mode approach for tensor-structured problems, which make extensively use of the functions contained in our package KronPACK. We remark that, when we employ Cartesian grids of points, they have been produced by the MATLAB command `ndgrid`. If instead one would prefer to use the ordering induced by the `meshgrid` command (which, however, works only up to dimension three), it is enough to interchange the first and the second matrix in the Tucker operator (5). The resulting tensor is then the (2,1,3)-permutation of $\boldsymbol{S}$ in Definition 2.4.

All the numerical experiments have been performed with MathWorks MATLAB® R2019a on an Intel® Core™ i7-8750H CPU with 16 GB of RAM. The degrees of freedom of the problems have been kept at a moderate size, in order to be reproducible with the package KronPACK in a few seconds on a personal laptop.

4.1 Code validation

In this section we validate the `tucker` function of KronPACK, by comparing it to the corresponding functions of the toolboxes mentioned in Section 2, i.e., `ttm` and `tmprod` of Tensor Toolbox for MATLAB and Tensorlab, respectively. We performed several tests on tensors of different orders and sizes and the three functions always produced the same output (up to round-off unit) at comparable computational times. For simplicity of exposition, we report in Fig. 1 just the wall-clock times of the experiments with tensors of order $d = 3$ and $d = 6$. For each selected value of d, we take as tensors and matrices sizes $m_\mu = n_\mu = n$, $\mu = 1,\ldots,d$, for different values of n, in such a way that the number of degrees of freedom n^d ranges from $N_{\min} = 12^6$ to $N_{\max} = 18^6$. The input tensors and matrices have normal distributed random values, and the complete code can be found in the script `code_validation.m`.

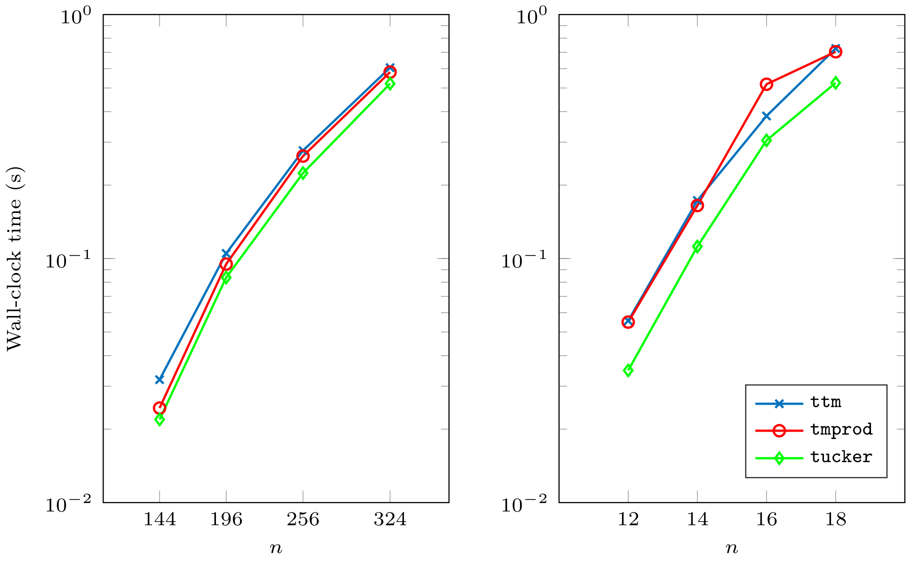

Fig. 1 Wall-clock times for different realizations of the Tucker operator (5) with the functions `ttm`, `tmprod`, and `tucker`. The left plot refers to the case $d = 3$, while the right plot refers to the case $d = 6$. Each test has been repeated several times in order to avoid fluctuations

4.2 Hermite–Laguerre–Fourier function decomposition

We are interested in the approximation, by means of a pseudospectral decomposition, of the trivariate function

$$f(\boldsymbol{x}) = \frac{x_2^2 \sin(20x_1)\sin(10x_2)\exp(-x_1^2 - 2x_2)}{\sin(2\pi x_3) + 2}, \quad \boldsymbol{x} = (x_1, x_2, x_3) \in \Omega,$$

where $\Omega = [-b_1,b_1] \times [0,b_2] \times [a_3,b_3]$. The decays in the first and second directions and the periodicity in the third direction suggest the use of a Hermite–Laguerre–Fourier (HLF) expansion. This mixed transform is useful, for instance, for the solution of differential equations with cylindrical coordinates by spectral methods, see [28]. We then introduce the *normalized and scaled* Hermite functions (orthonormal in $L^2(\mathbb{R})$)

$$\mathcal{H}_{i_1}^{\beta_1}(x_1) = \sqrt{\frac{\beta_1}{\sqrt{\pi}2^{i_1-1}(i_1-1)!}} H_{i_1}(\beta_1 x_1) \mathrm{e}^{-\beta_1^2 x_1^2/2},$$

where H_{i_1} is the (physicist's) Hermite polynomial of degree $i_1 - 1$. We consider the m_1 scaled Gauss–Hermite quadrature points $\{\xi_1^{k_1}\}_{k_1}$ and define $\Psi_1 \in \mathbb{R}^{m_1 \times m_1}$ to be the corresponding transform matrix with element $\mathcal{H}_{i_1}^{\beta_1}(\xi_1^{k_1})$ in position (i_1,k_1). The parameter β_1 is chosen so that the quadrature points are contained in $[-b_1,b_1]$ (see [29]). This is possible by estimating the largest quadrature point for the unscaled functions by $\sqrt{2m_1 + 1}$ (see [30, Ch. 6]) and setting

$$\beta_1 = \frac{\sqrt{2m_1 + 1}}{b_1}.$$

Moreover, we consider the *normalized and scaled* generalized Laguerre functions (orthonormal in $L^2(\mathbb{R}^+)$)

$$\mathcal{L}_{i_2}^{\alpha,\beta_2}(x_2) = \sqrt{\frac{\beta_2(i_2-1)!}{\Gamma(i_2+\alpha)}} L_{i_2}^{\alpha}(\beta_2 x_2)(\beta_2 x_2)^{\alpha/2} \mathrm{e}^{-\beta_2 x_2/2},$$

where $L_{i_2}^{\alpha}$ is the generalized Laguerre polynomial of degree $i_2 - 1$. We define Ψ_2 to be the corresponding transform matrix with element $\mathcal{L}_{i_2}^{\alpha,\beta_2}(\xi_2^{k_2})$ in position (i_2,k_2), where $\{\xi_2^{k_2}\}_{k_2}$ are the m_2 scaled generalized Gauss–Laguerre quadrature points. The parameter β_2 is chosen, similarly to the Hermite case, as

$$\beta_2 = \frac{4m_2 + 2\alpha + 2}{b_2},$$

see [30, Ch. 6] for the asymptotic estimate which holds for $|\alpha| \geq 1/4$ and $\alpha > -1$. Finally, for the Fourier decomposition, we obviously do not construct the transform matrix, but we rely on a Fast Fourier Transform (FFT) implementation provided by the MATLAB function `interpft`, which performs a resample of the given input values by means of FFT techniques. We measure the approximation error, for varying values of n_μ, $\mu = 1,2,3$, by evaluating the pseudospectral decomposition at a Cartesian grid of points $(x_1^{\ell_1}, x_2^{\ell_2}, x_3^{\ell_3})$, with $1 \leq \ell_\mu \leq n_\mu$. In order to do that, we construct the matrices Φ_1 and Φ_2 containing the values of the Hermite and generalized Laguerre functions at the points $\{x_1^{\ell_1}\}_{\ell_1}$ and $\{x_2^{\ell_2}\}_{\ell_2}$, respectively. The relevant code for the approximation of f and its evaluation, by using the KronPACK function `tuckerfun`, can be written as

```
PSIFUN{1} = @(f) PSI{1}*f;
PSIFUN{2} = @(f) PSI{2}*f;
PSIFUN{3} = @(f) f;
Fhat = tuckerfun(FW,PSIFUN);
PHIFUN{1} = @(f) PHI{1}*f;
PHIFUN{2} = @(f) PHI{2}*f;
PHIFUN{3} = @(f) interpft(f,n(3));
Ftilde = tuckerfun(Fhat,PHIFUN);
```

where `FW` is the three-dimensional array containing the values $f(\xi_1^{k_1}, \xi_2^{k_2}, \xi_3^{k_3}) w_1^{k_1} w_2^{k_2}$, where $\{\xi_3^{k_3}\}_{k_3}$ are the m_3 equispaced Fourier quadrature points in $[a_3,b_3)$ and $\{w_\mu^{k_\mu}\}_{k_\mu}$, with $\mu = 1,2$, are the scaled weights of the Gauss–Hermite and generalized Gauss–Laguerre quadrature rules, respectively. The values $\{\xi_\mu^{k_\mu}\}_{k_\mu}$ and $\{w_\mu^{k_\mu}\}_{k_\mu}$, for $\mu = 1,2$, have been computed by the relevant functions available, for instance, in Chebfun [31]. The complete example can be found in the script `example_spectral.m`.

Given a prescribed accuracy, we look for the smallest number of basis functions (m_1,m_2,m_3) that achieve it, and we measure the computational time needed to perform the approximation of f and its evaluation with the HLF method. As a term of comparison, we consider the same experiment with a three-dimensional Fourier spectral approximation (FFF method): in fact, for the size of the computational domain and the exponential decays along the first and second directions of the function f we are considering, it appears reasonable to approximate f by a periodic function in Ω and take advantage of the efficiency of a three-dimensional FFT.

The results with $\alpha=4$, $b_1=4$, $b_2=11$, $b_3=-a_3=1$, and $n_1=n_2=n_3=301$ evaluation points uniformly distributed in Ω are displayed in Fig. 2. As we can observe, the total number of degrees of freedom needed by the HLF approach is always smaller than the corresponding FFF one. In particular, despite the exponential decay along the second direction, the FFF method requires a very large number of Fourier coefficients along that direction in order to reach the most stringent accuracies. In these situations, the HLF method implemented with the μ-mode approach is preferable in terms of computational time to the well-established implementation by the FFT technique of the FFF method.

4.3 Multivariate interpolation

Let us consider the approximation of a function $f(\boldsymbol{x})$ through a five-variate interpolating polynomial in Lagrange form

$$p(\boldsymbol{x}) = \sum_{i_5=1}^{m_5} \cdots \sum_{i_1=1}^{m_1} f_{i_1\ldots i_5} L_{i_1}(x_1)\cdots L_{i_5}(x_5). \tag{28}$$

Here $L_{i_\mu}(x_\mu)$ is the Lagrange polynomial of degree $m_\mu-1$ on a set $\{\xi_\mu^{k_\mu}\}_{k_\mu}$ of m_μ interpolation points written in the second barycentric form, with $\mu=1,\ldots,5$, i.e.,

$$L_{i_\mu}(x_\mu) = \frac{\frac{w_\mu^{i_\mu}}{x_\mu-\xi_\mu^{i_\mu}}}{\sum_{k_\mu}\frac{w_\mu^{k_\mu}}{x_\mu-\xi_\mu^{k_\mu}}}, \quad w_\mu^{i_\mu} = \frac{1}{\prod_{k_\mu\neq i_\mu}(\xi_\mu^{i_\mu}-\xi_\mu^{k_\mu})},$$

while $f_{i_1\ldots i_5}=f(\xi_1^{i_1},\ldots,\xi_5^{i_5})$.

For our numerical example, we consider the five-dimensional Runge function

$$f(x_1,\ldots,x_5) = \frac{1}{1+16\sum_\mu x_\mu^2}$$

in the domain $[-1,1]^5$. We choose as interpolation points a Cartesian grid of Chebyshev nodes

$$\xi_\mu^{k_\mu} = \cos\left(\frac{(2k_\mu-1)\pi}{2m_\mu}\right), \quad k_\mu = 1,\ldots,m_\mu,$$

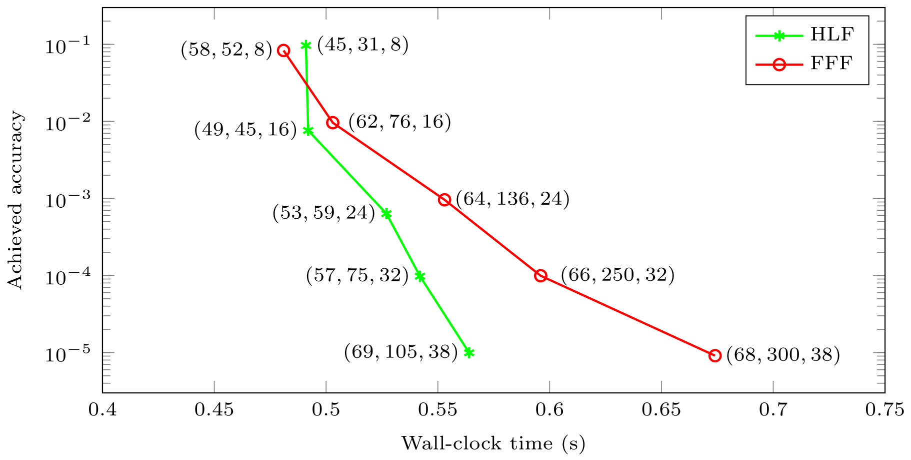

Fig. 2 Achieved accuracies versus wall-clock times (in seconds, averaged over 20 runs) for the Hermite–Laguerre–Fourier (HLF) and the Fourier–Fourier–Fourier (FFF) approaches. The label of the marks in the plot indicates the number of basis functions used in each direction

whose barycentric weights are

$$w_\mu^{k_\mu} = (-1)^{k_\mu+1} \sin\left(\frac{(2k_\mu - 1)\pi}{2m_\mu}\right), \quad k_\mu = 1, \dots, m_\mu.$$

This is the five-dimensional version of one of the examples presented in [32, Sec. 6]. We evaluate the polynomial at a uniformly spaced Cartesian grid of points $(x_1^{\ell_1}, \dots, x_5^{\ell_5})$, with $1 \leq \ell_\mu \leq n_\mu$. Then, approximation (28) at the just mentioned grid can be computed as

$$\boldsymbol{P} = \boldsymbol{F} \times_1 L_1 \times_2 \cdots \times_5 L_5, \tag{29}$$

where we collected the function evaluations at the interpolation points in the tensor $\boldsymbol{F}$ and L_μ contains the element $L_{i_\mu}(x_\mu^{\ell_\mu})$ in position (ℓ_μ, i_μ). If we store the matrices L_μ in a cell L, the corresponding MATLAB command for computing the desired approximation is

```
P = tucker(F,L);
```

The results, for a number of evaluation points fixed to $n_\mu = n = 35$ and varying number of interpolation points $m_\mu = m$, are reported in Fig. 3, and the complete code can be found in the script `example_interpolation.m`.

As expected, the error decreases according to the estimate

$$\|f(\boldsymbol{x}) - p(\boldsymbol{x})\|_\infty \approx K^{-m}, \quad K = \frac{1}{4} + \sqrt{\frac{17}{16}},$$

see [32, 33].

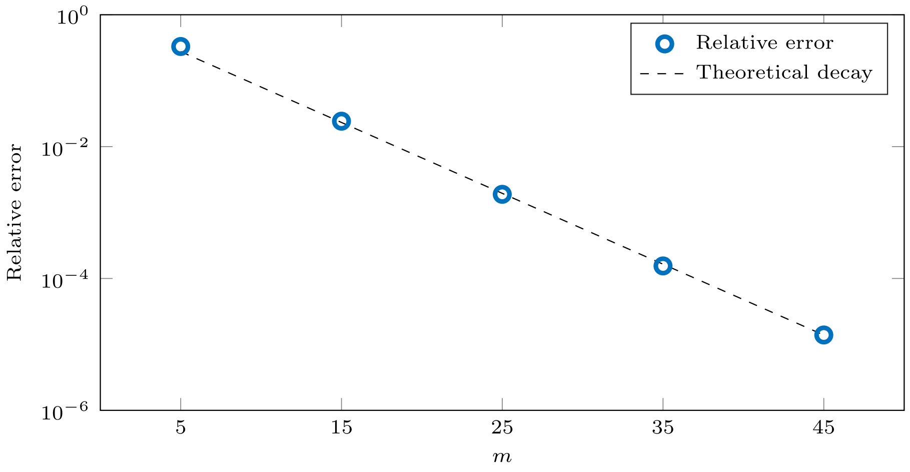

Fig. 3 Results for approximation (29) with an increasing number $m_\mu = m$ of interpolation points. The relative error (blue circles) is computed in maximum norm at the evaluation points. For reference, a dashed line representing the theoretical decay estimate is added

4.4 Linear evolutionary equation

Let us consider the following three-dimensional Advection–Diffusion–Absorption evolutionary equation, written in conservative form, for a concentration $u(t,\boldsymbol{x})$ (see [34])

$$\begin{cases} \partial_t u(t,\boldsymbol{x}) + \sum_{\mu=1}^{3} \beta_\mu \partial_{x_\mu}(x_\mu u(t,\boldsymbol{x})) = \alpha \sum_{\mu=1}^{3} \beta_\mu^2 \partial_{x_\mu}(x_\mu^2 \partial_{x_\mu} u(t,\boldsymbol{x})) - \gamma u(t,\boldsymbol{x}), \\ u(0,\boldsymbol{x}) = u_0(\boldsymbol{x}) = x_1(2-x_1)^2 x_2(2-x_2)^2 x_3(2-x_3)^2, \end{cases} \tag{30}$$

where β_μ, $\mu = 1,2,3$, and $\alpha > 0$ are advection and diffusion coefficients and $\gamma \geq 0$ is a coefficient governing the decay of $u(t,\boldsymbol{x})$. After a space discretization by second order centered finite differences on a Cartesian grid, we end up with a system of ODEs

$$\begin{cases} \boldsymbol{u}'(t) = (A_3 \oplus A_2 \oplus A_1)\boldsymbol{u}(t), \\ \boldsymbol{u}(0) = \boldsymbol{u}_0, \end{cases} \tag{31}$$

where $A_\mu \in \mathbb{R}^{n_\mu \times n_\mu}$ is the one-dimensional discretization of the operator

$$(2\alpha\beta_\mu^2 x_\mu - \beta_\mu x_\mu)\partial_{x_\mu} + \alpha\beta_\mu^2 x_\mu^2 \partial_{x_\mu^2} - \left(\beta_\mu + \frac{\gamma}{3}\right).$$

If we denote by $\boldsymbol{U}_0 = \mathrm{vec}(\boldsymbol{u}_0)$ and $\boldsymbol{U}(t) = \mathrm{vec}(\boldsymbol{u}(t))$ the tensors associated to the vectors $\boldsymbol{u}_0$ and $\boldsymbol{u}(t)$, respectively, then we have

$$\boldsymbol{U}(t) = \boldsymbol{U}_0 \times_1 \exp(tA_1) \times_2 \exp(tA_2) \times_3 \exp(tA_3). \tag{32}$$

We consider equation (30) for $\boldsymbol{x} \in [0,2]^3$, coupled with homogeneous Dirichlet–Neumann conditions ($u(t,\boldsymbol{x}) = 0$ at $x_\mu = 0$ and $\partial_{x_\mu} u(t,\boldsymbol{x}) = 0$ at $x_\mu = 2$, $\mu = 1,2,3$). The coefficients are fixed to

$$\beta_1 = \beta_2 = \beta_3 = \frac{2}{3}, \quad \alpha = \frac{1}{2}, \quad \gamma = \frac{1}{100}.$$

Then, if we compute the needed matrix exponentials by the function `expm` in MATLAB and define

```
E{mu} = expm(tstar*A{mu});
```

the solution $\boldsymbol{U}(t^*)$ at final time $t^* = 0.5$ can be computed as

```
U = tucker(U0,E);
```

since the matrix exponential is the exact solution and thus no substepping strategy is needed. The complete example is reported in the script `example_exponential.m`.

In Table 2 we show the results with a discretization in space of $\boldsymbol{n} = (50,55,60)$ grid points. Since the problem is moderately stiff, we consider for comparison the solution of system (31) by the `ode23` MATLAB function (which implements an explicit adaptive Runge–Kutta method of order (2)3) and by a standard implementation of the explicit Runge–Kutta method of order four (RK4). For the Runge–Kutta methods, we consider both the *tensor* and the *vector* implementations, using the functions `kronsumv` and `kronsum`, respectively (see equivalence (9)). The number of uniform time steps for RK4 has been chosen in order to obtain a comparable error with respect to the result of the variable time step solver `ode23`. As we can see, the tensor formulation (32) implemented using the function `tucker` is much faster than any other considered approach. Indeed, this is due to the fact that formula (32) requires a single time step and calls a level 3 BLAS only three times. For other experiments involving the approximation of the action of the matrix exponential in tensor-structured problems, we invite the reader to check [15].

4.5 Semilinear evolutionary equation

We consider the following three-dimensional semilinear evolutionary equation

$$\begin{cases} \partial_t u(t,\boldsymbol{x}) = \Delta u(t,\boldsymbol{x}) + \dfrac{1}{1 + u(t,\boldsymbol{x})^2} + \Phi(t,\boldsymbol{x}), \\ u(0,\boldsymbol{x}) = u_0(\boldsymbol{x}) = x_1(1-x_1)x_2(1-x_2)x_3(1-x_3), \end{cases} \tag{33}$$

Table 2 Summary of the results for solving the ODEs system (31) with the three described approaches. We report the number of time steps, the wall-clock times in seconds for both the tensor and the vector formulations (when feasible) and the relative error in infinity norm of the final solution with respect to the solution given by the `tucker` approach

	Time steps	Elapsed time vector	Elapsed time tensor	Error
`tucker`	1	–	0.03	–
`ode23`	1496	14.0	11.2	1.0e-4
RK4	1351	9.14	6.33	3.7e-5

for $\boldsymbol{x} \in [0,1]^3$, where the function $\Phi(t,\boldsymbol{x})$ is chosen so that the exact solution is $u(t,\boldsymbol{x}) = \mathrm{e}^t u_0(\boldsymbol{x})$. We complete the equation with homogeneous Dirichlet boundary conditions in all the directions. This is the three-dimensional generalization of the example presented in [35].

We discretize the problem in space by means of second order centered finite differences on a Cartesian grid, with n_μ grid points for the spatial variable x_μ, $\mu = 1,2,3$. Then, the application of the backward-forward Euler method leads to the following marching scheme

$$M\boldsymbol{u}_{k+1} = \boldsymbol{u}_k + \tau \boldsymbol{f}(t_k, \boldsymbol{u}_k), \tag{34}$$

where $\boldsymbol{u}_k \approx u(t_k,\boldsymbol{x})$, τ is the time step size, t_k is the current time and

$$\boldsymbol{f}(t_k, \boldsymbol{u}_k) = \frac{1}{1+\boldsymbol{u}_k^2} + \Phi(t_k, \boldsymbol{x}).$$

The matrix of the linear system (34) is given by

$$M = M_3 \oplus M_2 \oplus M_1, \quad M_\mu = \left(\frac{1}{3} I_\mu - \tau A_\mu\right),$$

where A_μ is the discretization of the partial differential operator $\partial_{x_\mu^2}$ and I_μ is the identity matrix of size n_μ. One could solve the linear system (34) using a direct method, in particular by computing the Cholesky factors of the matrix M once and for all (if the step size τ is constant). Another approach would be to use the Conjugate Gradient (CG) method for the single marching step (34). In MATLAB, the latter can be performed as

```
pcg(M,uk+tau*f(tk,uk),tol,maxit,[],[],uk);
```

or

```
pcg(Mfun,uk+tau*f(tk,uk),tol,maxit,[],[],uk);
```

where `M` is the matrix assembled using `kronsum` (vector approach), while `Mfun` is implemented by means of the function `kronsumv` (tensor approach). As described in Section 3.4, an effective preconditioner for system (34) is the one of ADI-type

$$P_3 \otimes P_2 \otimes P_1, \quad P_\mu = (I_\mu - \tau A_\mu).$$

The action of the inverse of this preconditioner on a vector $\boldsymbol{v}$ can be easily performed in tensor formulation, see formula (27), and the resulting Preconditioned Conjugate Gradient (PCG) method is

```
pcg(Mfun,uk+tau*f(tk,uk),tol,maxit,Pfun,[],uk);
```

where `Pfun` is implemented through the KronPACK function `itucker`. The complete example is reported in the file `example_imex.m`.

In Table 3 we report the results obtained for a space discretization of $\boldsymbol{n} = (40{,}44{,}48)$ grid points. The time step size τ of the marching scheme (34) is 0.01 and the final time of integration is $t^* = 1$. For all the methods, the final relative error in infinity norm with respect to the exact solution is $9.7 \cdot 10^{-3}$. As it is clearly shown, the ADI-type preconditioner is really effective in reducing the number of iterations of the CG method. Moreover, the resulting method is the fastest among all the considered approaches.

5 Conclusions

In this work, we presented how it is possible to state d-dimensional tensor-structured problems by means of composition of one-dimensional rules, in such a way that the resulting μ-mode BLAS formulation can be efficiently implemented on modern computer hardware. The common thread consists in the suitable employment of tensor product operations, with special emphasis on the Tucker operator and its variants. After validating our package KronPACK against other commonly used tensor operation toolboxes, the effectiveness of the μ-mode approach compared to other well-established techniques is shown on several examples from different fields of numerical analysis. More in detail, we employed this approach for a pseudospectral Hermite–Laguerre–Fourier trivariate function decomposition, for the barycentric Lagrange interpolation of a five-variate function and for the numerical solution of three-dimensional stiff linear and semilinear evolutionary differential equations by means of exponential techniques and a (preconditioned) IMEX method, respectively.

Table 3 Summary of the results for solving the semilinear equation (33) by the method of lines and the backward-forward Euler method. The elapsed time is the wall-clock time measured in seconds

	Avg. iterations per time step	Elapsed time
Direct	–	6.7
CG vector	30	3.3
CG tensor	30	2.2
PCG tensor	2	0.5

Appendix

Throughout the manuscript, the symbol $\otimes$ denotes the standard Kronecker product of two matrices. In particular, given $A \in \mathbb{C}^{m\times n}$ and $B \in \mathbb{C}^{p\times q}$, we have

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix} \in \mathbb{C}^{mp\times nq}.$$

Moreover, we define the Kronecker sum of two matrices $A \in \mathbb{C}^{m\times m}$ and $B \in \mathbb{C}^{p\times p}$, denoted by the symbol $\oplus$, as

$$A \oplus B = A \otimes I_B + I_A \otimes B \in \mathbb{C}^{mp\times mp},$$

where I_A and I_B are identity matrices of size m and p, respectively.

We define also the vectorization operator, denoted by vec, which stacks a given tensor $\boldsymbol{T} \in \mathbb{C}^{m_1\times\cdots\times m_d}$ in a vector $\boldsymbol{v} \in \mathbb{C}^{m_1\cdots m_d}$ in such a way that

$$\mathrm{vec}(\boldsymbol{T}) = \boldsymbol{v}, \text{ with } \boldsymbol{v}_j = \boldsymbol{T}(j_1, \ldots, j_d), \quad j = j_1 + \sum_{\mu=2}^{d}(j_\mu - 1)\prod_{k=1}^{\mu-1} m_k,$$

where $1 \le j_\mu \le m_\mu$ and $1 \le \mu \le d$.

The Kronecker product satisfies many properties, see [36] for a comprehensive review. For convenience of the reader, we list here the relevant ones in our context

1. $A \otimes (B_1 + B_2) = A \otimes B_1 + A \otimes B_2$ for every $A \in \mathbb{C}^{m\times n}$ and $B_1, B_2 \in \mathbb{C}^{p\times q}$;
2. $(B_1 + B_2) \otimes A = B_1 \otimes A + B_2 \otimes A$ for every $B_1, B_2 \in \mathbb{C}^{p\times q}$ and $A \in \mathbb{C}^{m\times n}$;
3. $(\lambda A) \otimes B = A \otimes (\lambda B) = \lambda(A \otimes B)$ for every $\lambda \in \mathbb{C}$, $A \in \mathbb{C}^{m\times n}$ and $B \in \mathbb{C}^{p\times q}$;
4. $(A \otimes B) \otimes C = A \otimes (B \otimes C)$ for every $A \in \mathbb{C}^{m\times n}$, $B \in \mathbb{C}^{p\times q}$ and $C \in \mathbb{C}^{r\times s}$;
5. $(A \otimes B)^\mathsf{T} = A^\mathsf{T} \otimes B^\mathsf{T}$ for every $A \in \mathbb{C}^{m\times n}$ and $B \in \mathbb{C}^{p\times q}$;
6. $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$ for every invertible matrix $A \in \mathbb{C}^{m\times m}$ and $B \in \mathbb{C}^{p\times p}$;
7. $(A \otimes B)(D \otimes E) = (AD) \otimes (BE)$ for every $A \in \mathbb{C}^{m\times n}$, $B \in \mathbb{C}^{p\times q}$, $D \in \mathbb{C}^{n\times r}$ and $E \in \mathbb{C}^{q\times s}$;
8. $\mathrm{vec}(ADC) = (C^\mathsf{T} \otimes A)\mathrm{vec}(D)$ for every $A \in \mathbb{C}^{m\times n}$, $D \in \mathbb{C}^{n\times r}$ and $C \in \mathbb{C}^{r\times s}$.

Funding Open access funding provided by Università degli Studi di Verona within the CRUI-CARE Agreement. The authors have received partial support from the Program Ricerca di Base 2019 of the University of Verona entitled "Geometric Evolution of Multi Agent Systems". Franco Zivcovich has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 850941).

Data availability Data sharing not applicable to this article as no datasets were generated or analyzed during the current study.

Declarations

Conflict of interest The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.S.: A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Softw. **16**(1), 1–17 (1990)
2. Intel Corporation: Intel Math Kernel Library. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html (2021). Accessed 27 Dec 2021
3. Xianyi, Z., Qian, W., Yunquan, Z.: Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In: 2012 IEEE 18th International Conference on Parallel and Distributed Systems, pp 684–691 (2012). Accessed 27 Dec 2021
4. NVIDIA Corporation: cuBLAS documentation. https://docs.nvidia.com/cuda/cublas/index.html (2021). Accessed 27 Dec 2021
5. Kolda, T.G.: Multilinear operators for higher-order decompositions. Technical Report SAND2006-2081 Sandia National Laboratories (2006)
6. Kolda, T.G., Bader, B.W.: Tensor decompositions and applications. SIAM Rev. **51**(3), 455–500 (2009)
7. Li, J., Battaglino, C., Perros, I., Sun, J., Vuduc, R.: An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In: SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Association for Computing Machinery, New York (2015)
8. Rogers, D.M.: Efficient primitives for standard tensor linear algebra. In: XSEDE16: Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale. Association for Computing Machinery, New York (2016)
9. Springer, P., Bientinesi, P.: Design of a high-performance GEMM-like tensor–tensor multiplication. ACM Trans. Math. Softw. **44**(3), 1–29 (2018)
10. Matthews, D.A.: High-performance tensor contraction without transposition. SIAM J. Sci. Comput. **40**(1), 1–24 (2018)
11. Bader, B.W., Kolda, T.G., et al.: Tensor Toolbox for MATLAB, Version 3.2.1. https://www.tensortoolbox.org (2021). Accessed 27 Dec 2021
12. Vervilet, N., Debals, O., Sorber, L., Van Barel, M., De Lathauwer, L.: Tensorlab 3.0. https://tensorlab.net. Available online (2016). Accessed 27 Dec 2021
13. Boyd, J.P.: Chebyshev and Fourier Spectral Methods, 2nd edn. DOVER Publications Inc., New York (2000)
14. de Boor, C.: A Practical Guide to Splines, Revised edn. Applied Mathematical Sciences, vol. 27. Springer, New York (2001)
15. Caliari, M., Cassini, F., Einkemmer, L., Ostermann, A., Zivcovich, F.: A μ-mode integrator for solving evolution equations in Kronecker form. J. Comput. Phys. **455**, 110989 (2022)
16. Bertolazzi, E., Falini, A., Mazzia, F.: The object oriented C++ library QIBSH++ for Hermite spline quasi interpolation. arXiv:2208.03260 (2022)
17. Al-Mohy, A.H., Higham, N.J.: A new scaling and squaring algorithm for the matrix exponential. SIAM J. Matrix Anal. Appl. **31**(3), 970–989 (2009)
18. Caliari, M., Zivcovich, F.: On-the-fly backward error estimate for matrix exponential approximation by Taylor algorithm. J. Comput. Appl. Math. **346**, 532–548 (2019)
19. Al-Mohy, A.H., Higham, N.J.: Computing the action of the matrix exponential with an application to exponential integrators. SIAM J. Sci. Comput. **33**(2), 488–511 (2011)

20. Niesen, J., Wright, W.M.: Algorithm 919: A Krylov subspace algorithm for evaluating the ϕ-functions appearing in exponential integrators. ACM Trans. Math. Softw. **38**(3), 1–19 (2012)
21. Gaudreault, S., Rainwater, G., Tokman, M.: KIOPS: A fast adaptive Krylov subspace solver for exponential integrators. J. Comput. Phys. **372**, 236–255 (2018)
22. Caliari, M., Cassini, F., Zivcovich, F.: Approximation of the matrix exponential for matrices with a skinny field of values. BIT Numer. Math. **60**(4), 1113–1131 (2020)
23. Neudecker, H.: A Note on Kronecker matrix products and matrix equation systems. SIAM J. Appl. Math. **17**(3), 603–606 (1969)
24. Arbenz, P., Říha, L.: Batched transpose-free ADI-type preconditioners for a Poisson solver on GPGPUs. J. Parallel Distrib. Comput. **137**, 148–159 (2020)
25. Kirsten, G., Simoncini, V.: A matrix-oriented POD-DEIM algorithm applied to nonlinear differential matrix equations. arXiv:2006.13289 (2020)
26. Palitta, D., Simoncini, V.: Matrix-equation-based strategies for convection–diffusion equations. BIT Numer. Math. **56**(2), 751–776 (2016)
27. Chen, M., Kressner, D.: Recursive blocked algorithms for linear systems with Kronecker product structure. Numer. Algorithms **84**(3), 1199–1216 (2020)
28. Bao, W., Li, H., Shen, J.: A generalized-Laguerre–Fourier–Hermite pseudospectral method for computing the dynamics of rotating Bose–Einstein condensates. SIAM J. Sci. Comput. **31**(5), 3685–3711 (2009)
29. Tang, T.: The Hermite spectral method for Gaussian-type functions. SIAM J. Sci. Comput. **14**(3), 594–606 (1993)
30. Szegö, G.: Orthogonal Polynomials, 4th edn., vol. 23. Colloquium Publications, American Mathematical Society, Providence (1975)
31. Driscoll, T.A., Hale, N., Trefethen, L.N. (eds.): Chebfun Guide. Pafnuty Publications, Oxford (2014)
32. Berrut, J.-P., Trefethen, L.N.: Barycentric Lagrange interpolation. SIAM Rev. **46**(3), 501–517 (2004)
33. Trefethen, L.N.: Multivariate polynomial approximation in the hypercube. Proc. Am. Math. Soc. **145**(11), 4837–4844 (2017)
34. Zoppou, C., Knight, J.H.: Analytical solution of a spatially variable coefficient advection–diffusion equation in up to three dimensions. Appl. Math. Model. **23**(9), 667–685 (1999)
35. Hochbruck, M., Ostermann, A.: Explicit exponential Runge–Kutta methods for semilinear parabolic problems. SIAM J. Numer. Anal. **43**(3), 1069–1090 (2005)
36. Van Loan, C.F.: The ubiquitous Kronecker product. J. Comput. Appl. Math. **123**(1–2), 85–100 (2000)

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.