A μ-mode integrator for solving evolution equations in Kronecker form

Marco Caliari[a], Fabio Cassini[b,*], Lukas Einkemmer[c], Alexander Ostermann[c], Franco Zivcovich[d]

[a]*Department of Computer Science, University of Verona, Italy*
[b]*Department of Mathematics, University of Trento, Italy*
[c]*Department of Mathematics, University of Innsbruck, Austria*
[d]*Laboratoire Jacques-Louis Lions, Sorbonne University, France*

Abstract

In this paper, we propose a μ-mode integrator for computing the solution of stiff evolution equations. The integrator is based on a d-dimensional splitting approach and uses exact (usually precomputed) one-dimensional matrix exponentials. We show that the action of the exponentials, i.e. the corresponding batched matrix-vector products, can be implemented efficiently on modern computer systems. We further explain how μ-mode products can be used to compute spectral transforms efficiently even if no *fast* transform is available. We illustrate the performance of the new integrator by solving, among the others, three-dimensional linear and nonlinear Schrödinger equations, and we show that the μ-mode integrator can significantly outperform numerical methods well-established in the field. We also discuss how to efficiently implement this integrator on both multi-core CPUs and GPUs. Finally, the numerical experiments show that using GPUs results in performance improvements between a factor of 10 and 20, depending on the problem.

Keywords: numerical solution of evolution equations; μ-mode product; dimension splitting; spectral transform; Schrödinger equation; Graphic Processing Unit (GPU)

1. Introduction

Due to the importance of simulation in various fields of science and engineering, devising efficient numerical methods for solving evolutionary partial differential equations has received considerable interest in the literature. For linear problems with time-invariant coefficients, after discretizing in space, the task of solving the partial differential equation is equivalent to computing the action of a matrix exponential to a given initial value. Computing the

*Corresponding author

Email addresses: `marco.caliari@univr.it` (Marco Caliari), `fabio.cassini@unitn.it` (Fabio Cassini), `lukas.einkemmer@uibk.ac.at` (Lukas Einkemmer), `alexander.ostermann@uibk.ac.at` (Alexander Ostermann), `franco.zivcovich@sorbonne-universite.fr` (Franco Zivcovich) action of matrix exponentials is also a crucial ingredient to devise efficient numerical methods for nonlinear partial differential equations; for example, in the context of exponential integrators [28] or splitting methods [35].

Despite the significant advances made in constructing more efficient numerical algorithms, efficiently computing the action of large matrix functions remains a significant challenge. In this paper, we propose a μ-mode integrator that performs this computation for matrices in Kronecker form by computing the action of one-dimensional matrix exponentials only. In d dimensions and with n grid points per dimension the number of arithmetic operations required scales as $\mathcal{O}(n^{d+1})$. Nevertheless, such an approach would not have been viable in the past. With the increasing gap between the amount of floating point operations compared to the amount of memory transactions modern computer systems can perform, however, this is no longer a consequential drawback. In fact, (batched) matrix-matrix multiplications, as are required for this algorithm, can achieve performance close to the theoretical limit of the hardware, and they do not suffer from the irregular memory accesses that plague implementations based on sparse matrix formats. This is particularly true on accelerators, such as Graphic Processing Units (GPUs). Thus, on modern computer hardware, the proposed method is extremely effective. In this paper, we will show that for a range of problems the proposed μ-mode integrator can outperform well-established integrators that are commonly used in the field. We investigate the performances of the method for a two-dimensional pipe flow example. Then, we consider three-dimensional linear Schrödinger equations with time-dependent and time-independent potentials, in combination with Hermite spectral discretization, as well as a cubic nonlinear Schrödinger equation (Gross–Pitaevskii equation) in three space dimensions. In this context, we will also provide a discussion on the implementation of the method for multi-core CPUs and GPUs.

The μ-mode integrator is exact for linear problems in *Kronecker form* (see section 2 for more details). The discretization of many differential operators with constant coefficients fits into this class (e.g., the Laplacian operator Δ and the $\mathrm{i}\Delta$ operator that is commonly needed in quantum mechanics), as well as some more complicated problems (e.g. the Hamiltonian for a particle in a harmonic potential). For nonlinear partial differential equations, the approach can be used to solve the part of the problem that is in Kronecker form: for example, in the framework of a splitting method.

The μ-mode integrator is related to dimension splitting schemes such as alternating direction implicit (ADI) schemes (see, e.g., [24, 27, 37, 41]). However, while the main motivation for the dimension splitting in ADI is to obtain one-dimensional matrix equations, for which efficient solvers such as the Thomas algorithm are known, for the μ-mode integrator the main utility of the dimension splitting is the reduction to one-dimensional problems for which matrix exponentials can be computed efficiently. Because of the exactness property described above, for many problems the μ-mode integrator can be employed with a much larger step size compared to implicit methods such as ADI. This is particularly true for highly oscillatory problems, where both implicit and explicit integrators do suffer from small time steps (see, e.g., [4]).

In the context of spectral decompositions, commonly employed for pseudospectral methods, the structure of the problem also allows us to use μ-mode products to efficiently compute spectral transforms from the space of values to the space of coefficients (and vice versa) even if no d-dimensional *fast* transform is available.

The outline of the paper is as follows. In section 2 we describe the proposed μ-mode integrator and explain in detail what it means for a differential equation to be in Kronecker form. We also discuss for which class of problems the integrator is particularly efficient. We then show, in section 3, how μ-mode products can be used to efficiently compute arbitrary spectral transforms. Numerical results that highlight the efficiency of the approach will be presented in section 4. The implementation on modern computer architectures, which includes performance results for multi-core CPU and GPU based systems, will be discussed in section 5. Finally, in section 6 we draw some conclusions.

2. The μ-mode integrator for differential equations in Kronecker form

As a simple example that introduces the main idea, we consider the two-dimensional heat equation

$$\begin{aligned} \partial_t u(t,\mathbf{x}) &= \Delta u(t,\mathbf{x}) = \left(\partial_1^2 + \partial_2^2\right) u(t,\mathbf{x}), \qquad \mathbf{x} \in \Omega \subset \mathbb{R}^2, \quad t > 0, \\ u(0,\mathbf{x}) &= u_0(\mathbf{x}) \end{aligned} \tag{1}$$

on a rectangle, subject to appropriate boundary conditions (e.g. periodic, homogeneous Dirichlet or homogeneous Neumann). Its analytic solution is given by

$$u(t,\cdot) = \mathrm{e}^{t\Delta} u_0 = \mathrm{e}^{t\partial_1^2}\mathrm{e}^{t\partial_2^2} u_0 = \mathrm{e}^{t\partial_2^2}\mathrm{e}^{t\partial_1^2} u_0, \tag{2}$$

where the last two equalities result from the fact that the partial differential operators ∂_1^2 and ∂_2^2 commute.

Discretizing (1) by finite differences on a Cartesian grid with $n_1 \times n_2$ grid points results in the linear differential equation

$$\mathbf{u}'(t) = \left(I_2 \otimes A_1 + A_2 \otimes I_1\right) \mathbf{u}(t), \quad \mathbf{u}(0) = \mathbf{u}_0 \tag{3}$$

for the unknown vector $\mathbf{u}(t)$. Here, A_1 is a (one-dimensional) stencil matrix for ∂_1^2 on the grid points $x_1^{i_1}$, $1 \leq i_1 \leq n_1$, and A_2 is a (one-dimensional) stencil matrix for ∂_2^2 on the grid points $x_2^{i_2}$, $1 \leq i_2 \leq n_2$. The symbol $\otimes$ denotes the standard Kronecker product between two matrices. Since the matrices $I_2 \otimes A_1$ and $A_2 \otimes I_1$ trivially commute, the solution of (3) is given by

$$\mathbf{u}(t) = \mathrm{e}^{t(I_2\otimes A_1 + A_2 \otimes I_1)}\mathbf{u}_0 = \mathrm{e}^{tI_2\otimes A_1}\mathrm{e}^{tA_2\otimes I_1}\mathbf{u}_0 = \mathrm{e}^{tA_2\otimes I_1}\mathrm{e}^{tI_2\otimes A_1}\mathbf{u}_0,$$

which is the discrete analog of (2).

Using the tensor structure of the problem, the required actions of the large matrices $\mathrm{e}^{tI_2\otimes A_1}$ and $\mathrm{e}^{tA_2\otimes I_1}$ on a vector can easily be reformulated. Let $\mathbf{U}(t)$ be the order two tensor of size $n_1 \times n_2$ (in fact, a matrix) whose stacked columns form the vector $\mathbf{u}(t)$. The indices of this matrix reflect the structure of the grid. In particular

$$\mathbf{U}(t)(i_1,i_2) = u(t, x_1^{i_1}, x_2^{i_2}), \qquad i_1 = 1,\ldots,n_1, \quad i_2 = 1,\ldots,n_2.$$

Using this tensor notation, problem (3) takes the form

$$\mathbf{U}'(t) = A_1\,\mathbf{U}(t) + \mathbf{U}(t)A_2^{\mathsf{T}}, \quad \mathbf{U}(0) = \mathbf{U}_0,$$

and its solution can be expressed as

$$\mathbf{U}(t) = \mathrm{e}^{tA_1}\mathbf{U}_0\,\mathrm{e}^{tA_2^{\mathsf{T}}}, \tag{4}$$

see [38]. From this representation, it is clear that $\mathbf{U}(t)$ can be computed as the action of the small matrices e^{tA_1} and e^{tA_2} on the tensor $\mathbf{U}_0$. More precisely, the matrices e^{tA_1} and e^{tA_2} act on the first and second indices of $\mathbf{U}$, respectively. The computation of (4) can thus be performed by the simple algorithm

$$\begin{aligned}
\mathbf{U}^{(0)} &= \mathbf{U}_0, \\
\mathbf{U}^{(1)}(\cdot, i_2) &= \mathrm{e}^{tA_1}\mathbf{U}^{(0)}(\cdot, i_2), \qquad i_2 = 1, \dots, n_2, \\
\mathbf{U}^{(2)}(i_1, \cdot) &= \mathrm{e}^{tA_2}\mathbf{U}^{(1)}(i_1, \cdot), \qquad i_1 = 1, \dots, n_1, \\
\mathbf{U}(t) &= \mathbf{U}^{(2)}.
\end{aligned}$$

It should be duly noted that the μ-mode integrator is not restricted to the simple example considered until now. Indeed, let us consider the differential equation

$$\mathbf{u}'(t) = M\mathbf{u}(t), \quad \mathbf{u}(0) = \mathbf{u}_0, \tag{5}$$

where

$$M = \sum_{\mu=1}^{d} A_{\otimes\mu}$$

and

$$A_{\otimes\mu} = I_d \otimes \cdots \otimes I_{\mu+1} \otimes A_\mu \otimes I_{\mu-1} \otimes \cdots \otimes I_1. \tag{6}$$

Here, A_μ denotes an *arbitrary* $n_\mu \times n_\mu$ matrix while I_μ is the identity matrix of size n_μ, $1 \le \mu \le d$. The matrix M is also known in the literature as the *Kronecker sum* of the matrices A_μ and is denoted by

$$M = A_d \oplus A_{d-1} \oplus \cdots \oplus A_2 \oplus A_1.$$

Condition (6) holds true for a range of equations with linear and constant coefficient differential operators on tensor product domains. Examples in this class include, after space discretization, the diffusion-advection-absorption equation

$$\partial_t u(t, \mathbf{x}) = \alpha\Delta u(t, \mathbf{x}) + \beta \cdot \nabla u(t, \mathbf{x}) - \gamma u(t, \mathbf{x})$$

or the Schrödinger equation with potential in Kronecker form

$$\mathrm{i}\partial_t \psi(t, \mathbf{x}) = -\frac{1}{2}\Delta\psi(t, \mathbf{x}) + \left(\sum_{\mu=1}^{d} V(t, x_\mu)\right)\psi(t, \mathbf{x}).$$

Condition (6) is fulfilled also for some problems with non-constant coefficient differential operators, see section 4.2 for an example. We will consider these and other equations later in the paper to perform numerical examples.

Equation (5) is what we call a linear problem in *Kronecker form*, and its solution is obviously given by

$$\mathbf{u}(t) = \mathrm{e}^{tA_{\otimes 1}} \cdots \mathrm{e}^{tA_{\otimes d}} \mathbf{u}_0,$$

where the single factors $\mathrm{e}^{tA_{\otimes\mu}}$ mutually commute. Again, the computation of $\mathbf{u}(t)$ just requires the actions of the small matrices e^{tA_μ}. More precisely, consider the order d tensor $\mathbf{U}(t)$ of size $n_1 \times \cdots \times n_d$ that collects the values of a function u on a Cartesian grid, i.e.

$$\mathbf{U}(t)(i_1, \ldots, i_d) = u(t, x_1^{i_1}, \ldots, x_d^{i_d}), \qquad 1 \leq i_\mu \leq n_\mu, \quad 1 \leq \mu \leq d.$$

Then, in the same way as in the two-dimensional heat equation case, the computation of $\mathbf{u}(t)$ can be performed by

$$\begin{aligned} \mathbf{U}^{(0)} &= \mathbf{U}_0, \\ \mathbf{U}^{(1)}(\cdot, i_2, \ldots, i_d) &= \mathrm{e}^{tA_1} \mathbf{U}^{(0)}(\cdot, i_2, \ldots, i_d), \qquad 1 \leq i_\mu \leq n_\mu, \quad 2 \leq \mu \leq d, \\ &\cdots \\ \mathbf{U}^{(d)}(i_1, \ldots, i_{d-1}, \cdot) &= \mathrm{e}^{tA_d} \mathbf{U}^{(d-1)}(i_1, \ldots, i_{d-1}, \cdot), \qquad 1 \leq i_\mu \leq n_\mu, \quad 1 \leq \mu \leq d-1, \\ \mathbf{U}(t) &= \mathbf{U}^{(d)}. \end{aligned} \tag{7}$$

We remark that scheme (7) can also be useful as a building block for solving nonlinear partial differential equations. In this case, an exponential or splitting scheme would be used to separate the linear part, which is treated exactly by the integrator (7), from the nonlinear part which is treated in a different fashion. This is useful for a number of problems. For example, when solving the drift-kinetic equations in plasma physics using an exponential integrator [15, 16], Fourier spectral methods are commonly used. While such FFT based schemes are efficient, it is also well known that they can lead to numerical oscillations [22]. Using integrator (7) would allow us to choose a more appropriate space discretization while still retaining efficiency. Another example are diffusion-reaction equations with nonlinear reaction terms that are treated using splitting methods (see, e.g., [20, 21, 29]). In this case scheme (7) would be used to efficiently solve the subflow corresponding to the linear diffusion. We further note that a related approach was pursued by [39] in order to produce schemes that solve two- and three-dimensional biological models.

Implementing integrator (7) requires the computation of d small exponentials of sizes $n_1 \times n_1$, …, $n_d \times n_d$, respectively. If a marching scheme with *constant* time step is applied to (5), then these matrices can be precomputed once and for all, and their storage cost is negligible compared to that required by the solution $\mathbf{U}(t)$. Otherwise, we need to compute at every time step new matrix exponentials, whose computational cost still represents only a small fraction of the entire algorithm (see section 4.1). Indeed, the main component of the final cost is represented by the computation of matrix-matrix products of size $n_\mu \times n_\mu$

times $n_\mu \times (n_1 \cdots n_{\mu-1} n_{\mu+1} \cdots n_d)$. Thus, the computational complexity of the algorithm is $\mathcal{O}(N \max_\mu n_\mu)$, where $N = n_1 \cdots n_d$ is the total number of degrees of freedom.

Clearly, we can solve equation (5) also by directly computing the vector $\mathrm{e}^{tM}\mathbf{u}_0$. In fact M is an $N \times N$ sparse matrix and, when it is too large for the explicit computation of e^{tM}, the action of the matrix exponential can be approximated by polynomial methods such as Krylov projection (see, for instance, [25, 40]), Taylor series [3], or polynomial interpolation (see, for instance, [9, 10, 11]). All these iterative methods require one matrix-vector product per iteration, which costs $\mathcal{O}(N)$ plus additional vector operations. The number of iterations, however, highly depends on the norm and some properties of the matrix, such as the normality, the condition number, and the stiffness, and it is not easy to predict it. Moreover, for Krylov methods, one has to take into account the storage of a full matrix with N rows and as many columns as the dimension of the Krylov subspace.

Also, an implicit scheme based on a Krylov solver could be applied to integrate equation (5). In particular, if we restrict our attention to the heat equation case and the conjugate gradient method, for example, $\mathcal{O}(\max_\mu n_\mu)$ iterations are needed for the solution (see the convergence analysis in [44, Chap. 6.11]), and each iteration requires a sparse matrix-vector product which is $\mathcal{O}(N)$. Hence, the resulting computational complexity is the same as for the proposed algorithm. However, on modern hardware architectures memory transactions are much more costly than performing floating point operations. A modern CPU or GPU can easily perform many tens of arithmetic operations in the same time it takes to read/write a single number from/to memory (see the discussion in section 5).

Summarizing, our scheme has the following advantages:

- For a heat equation the proposed integrator only requires $\mathcal{O}(N)$ memory operations, compared to an implicit scheme which requires $\mathcal{O}(N \max_\mu n_\mu)$ memory operations. This has huge performance implications on all modern computer architectures. For other classes of PDEs the analysis is more complicated. However, in many situations similar results can be obtained.

- Very efficient implementations of matrix-matrix products that operate close to the limit of the hardware are available. This is not the case for iterative schemes which are based on sparse matrix-vector products.

- The computation of pure matrix exponentials of small matrices is less prone to the problems that affect the approximation of the action of the (large) matrix exponential.

- The proposed integrator is often able to take much larger time step sizes than, for example, an ADI scheme, as it computes the exact result for equations in Kronecker form.

- Conserved quantities of the underlying system, such as mass, are preserved by the integrator.

We will in fact see that the proposed integrator can outperform algorithms with linear computational complexity (see sections 4.3 and 4.4).

Equation (7) gives perhaps the most intuitive picture of the proposed approach. However, we can also formulate this problem in terms of μ-fibers. Indeed, let $\mathbf{U} \in \mathbb{C}^{n_1 \times \cdots \times n_d}$ be an order d tensor. A *μ-fiber* of $\mathbf{U}$ is a vector in $\mathbb{C}^{n_\mu}$ obtained by fixing every index of the tensor but the μth. In these terms, $\mathbf{U}^{(\mu-1)}(i_1, \ldots, i_{\mu-1}, \cdot, i_{\mu+1}, \ldots, i_d)$ is a μ-fiber of the tensor $\mathbf{U}^{(\mu-1)}$, and every line in formula (7) corresponds to the action of the matrix e^{tA_μ} on the μ-fibers of $\mathbf{U}^{(\mu-1)}$. By means of μ-fibers, it is possible to define the following operation.

Definition 2.1. *Let $L \in \mathbb{C}^{m \times n_\mu}$ be a matrix. Then the* μ-mode product[1] *of L with $\mathbf{U}$, denoted by $\mathbf{S} = \mathbf{U} \times_\mu L$, is the tensor $\mathbf{S} \in \mathbb{C}^{n_1 \times \ldots \times n_{\mu-1} \times m \times n_{\mu+1} \times \ldots \times n_d}$ obtained by multiplying the matrix L onto the μ-fibers of $\mathbf{U}$, that is*

$$\mathbf{S}(i_1, \cdots, i_{\mu-1}, i, i_{\mu+1}, \cdots, i_d) = \sum_{j=1}^{n_\mu} L_{ij} \mathbf{U}(i_1, \cdots, i_{\mu-1}, j, i_{\mu+1}, \cdots, i_d), \qquad 1 \leq i \leq m.$$

According to this definition, it is clear that in formula (7) we are performing d consecutive μ-mode products with the matrices e^{tA_μ}, $1 \leq \mu \leq d$. We can therefore write scheme (7) as follows

$$\mathbf{U}(t) = \mathbf{U}_0 \times_1 \mathrm{e}^{tA_1} \times_2 \ldots \times_d \mathrm{e}^{tA_d}.$$

This is the reason why we call the proposed method the μ-mode integrator. Notice that the concatenation of μ-mode products of d matrices with a tensor is also known as the *Tucker operator* (see [31]), and it can be performed using efficient level-3 BLAS operations. For more information on tensor algebra and the μ-mode product we refer the reader to [32].

3. Application of the μ-mode product to spectral decomposition and reconstruction

Problems of quantum mechanics with vanishing boundary conditions are often set in an unbounded spatial domain. In this case, the spectral decomposition in space by Hermite functions is appealing (see [7, 48]), since it allows to treat boundary conditions in a natural way (without imposing artificial periodic boundary conditions as required by Fourier spectral methods, for example).

Consider the multi-index $\mathbf{i} = (i_1, \ldots, i_d) \in \mathbb{N}_0^d$ and the coordinate vector $\mathbf{x} = (x_1, \ldots, x_d)$ belonging to $\mathbb{R}^d$. We define the d-variate functions $\mathcal{H}_\mathbf{i}(\mathbf{x})$ as

$$\mathcal{H}_\mathbf{i}(\mathbf{x}) = \prod_{\mu=1}^{d} H_{i_\mu}(x_\mu) \mathrm{e}^{-x_\mu^2/2},$$

where $\{H_{i_\mu}(x_\mu)\}_{i_\mu}$ is the family of Hermite polynomials ortho*normal* with respect to the weight function $\mathrm{e}^{-x_\mu^2}$ on $\mathbb{R}$, that is

$$\int_{\mathbb{R}^d} \mathcal{H}_\mathbf{i}(\mathbf{x}) \mathcal{H}_\mathbf{j}(\mathbf{x}) d\mathbf{x} = \delta_{\mathbf{ij}}.$$

[1] Also known as mode-n product, n-mode product or mode-α multiplication, depending on the convention.

We recall that Hermite functions satisfy

$$\left(-\frac{1}{2}\sum_{\mu=1}^{d}(\partial_\mu^2 - x_\mu^2)\right)\mathcal{H}_{\mathbf{i}}(\mathbf{x}) = \lambda_{\mathbf{i}}\mathcal{H}_{\mathbf{i}}(\mathbf{x}),$$

where

$$\lambda_{\mathbf{i}} = \sum_{\mu=1}^{d}\left(\frac{1}{2} + i_\mu\right).$$

In general, we can consider a family of functions $\phi_{\mathbf{i}}\colon R_1 \times \cdots \times R_d \to \mathbb{C}$ in tensor form

$$\phi_{\mathbf{i}}(\mathbf{x}) = \prod_{\mu=1}^{d}\phi_{i_\mu}^{\mu}(x_\mu)$$

which are orthonormal on the Cartesian product of intervals $R_1, \ldots, R_d$ of $\mathbb{R}$.

If a function f can be expanded into a series

$$f(\mathbf{x}) = \sum_{\mathbf{i}} f_{\mathbf{i}}\phi_{\mathbf{i}}(\mathbf{x}), \qquad f_{\mathbf{i}} \in \mathbb{C},$$

then its $\mathbf{i}$th coefficient is

$$f_{\mathbf{i}} = \int_{R_1\times\cdots\times R_d} f(\mathbf{x})\overline{\phi_{\mathbf{i}}}(\mathbf{x})d\mathbf{x}.$$

In order to approximate the integral on the right-hand side, we rely on a tensor-product quadrature formula. To do so, we consider for each direction μ a set of m_μ uni-variate quadrature nodes $X_{\ell_\mu}^{\mu}$ and weights $W_{\ell_\mu}^{\mu}$, $0 \le \ell_\mu \le m_\mu$, and fix to k_μ the number of uni-variate functions $\phi_{i_\mu}^{\mu}(x_\mu)$ to be considered. We have then

$$\hat{f}_{\mathbf{i}} = \sum_{\boldsymbol{\ell}<\mathbf{m}} f(\mathbf{x}_{\boldsymbol{\ell}})\overline{\phi_{\mathbf{i}}}(\mathbf{x}_{\boldsymbol{\ell}})w_{\boldsymbol{\ell}}, \qquad \mathbf{i} < \mathbf{k}, \tag{8}$$

where $\mathbf{x}_{\boldsymbol{\ell}} = (X_{\ell_1}^{1}, \cdots, X_{\ell_d}^{d}) \in \mathbb{R}^d$, $w_{\boldsymbol{\ell}} = \prod_{\mu=1}^{d} W_{\ell_\mu}^{\mu}$ and $\mathbf{k}$ is the multi-index which collects the values $\{k_\mu\}_\mu$. We show now how μ-mode products can be employed to compute the coefficients of the spectral decomposition

$$\hat{f}(\mathbf{x}) = \sum_{\mathbf{i}<\mathbf{k}} \hat{f}_{\mathbf{i}}\phi_{\mathbf{i}}(\mathbf{x}) \approx f(\mathbf{x}) \tag{9}$$

of a d-variate function and its evaluation on a Cartesian grid. First of all, for each fixed μ, $1 \le \mu \le d$, we define the matrix $\Phi_\mu \in \mathbb{C}^{k_\mu \times m_\mu}$ with components

$$(\Phi_\mu)_{i\ell} = \overline{\phi_i^{\mu}}(X_\ell^{\mu}),$$

and we denote by $\mathbf{F_W} \in \mathbb{C}^{m_1\times\cdots\times m_d}$ the tensor with elements $f(\mathbf{x}_{\boldsymbol{\ell}})w_{\boldsymbol{\ell}}$ and by $\hat{\mathbf{F}} \in \mathbb{C}^{k_1\times\cdots\times k_d}$ the tensor with elements $\hat{f}_{\mathbf{i}}$. Then, in terms of the Tucker operator, we can write equation (8) as follows

$$\hat{\mathbf{F}} = \mathbf{F_W} \times_1 \Phi_1 \times_2 \cdots \times_d \Phi_d. \tag{10}$$

It is then possible to evaluate the function $\hat{f}(\mathbf{x})$ in (9) at a Cartesian grid $\mathbf{y}_\mathbf{p} = (Y^1_{p_1}, \ldots, Y^d_{p_d})$, that is

$$\hat{f}(\mathbf{y}_\mathbf{p}) = \sum_{\mathbf{i}<\mathbf{k}} \hat{f}_\mathbf{i} \phi_\mathbf{i}(\mathbf{y}_\mathbf{p}), \qquad \mathbf{p} < \mathbf{q}, \tag{11}$$

by the Tucker operator, too. Here the component q_μ of the multi-index $\mathbf{q}$ is the number of uni-variate evaluation points $Y^\mu_{p_\mu}$. Indeed, if we collect the elements $\hat{f}(\mathbf{y}_\mathbf{p})$ in the tensor $\hat{\hat{\mathbf{F}}} \in \mathbb{C}^{q_1 \times \cdots \times q_d}$ and, for fixed μ, we define the matrix $\Psi_\mu \in \mathbb{C}^{q_\mu \times k_\mu}$ with components $(\Psi_\mu)_{pi} = \phi^\mu_i(Y^\mu_p)$, then

$$\hat{\hat{\mathbf{F}}} = \hat{\mathbf{F}} \times_1 \Psi_1 \times_2 \cdots \times_d \Psi_d \tag{12}$$

is the tensor formulation of formula (11).

Now, we restrict our attention to the common case where the quadrature nodes are chosen in such a way that

$$\sum_{\boldsymbol{\ell}<\mathbf{m}} \phi_\mathbf{i}(\mathbf{x}_{\boldsymbol{\ell}}) \overline{\phi_\mathbf{j}}(\mathbf{x}_{\boldsymbol{\ell}}) w_{\boldsymbol{\ell}} = \delta_{\mathbf{ij}}, \qquad \mathbf{i}, \mathbf{j} < \mathbf{k}$$

with $\mathbf{m} = \mathbf{k}$, that is, the orthonormality relation among the $\phi_\mathbf{i}$ functions is true also at the discrete level. This is the case, for instance, when using Gauss–Hermite quadrature nodes for $\phi_\mathbf{i}(\mathbf{x}) = \mathcal{H}_\mathbf{i}(\mathbf{x})$. Then, the matrices $\Phi_\mu \in \mathbb{C}^{m_\mu \times m_\mu}$ turn out to be square and formula (10) is the *spectral transform* from the space of values to the space of coefficients. Moreover, if the evaluation points coincide with the quadrature nodes, then we have $\Psi_\mu = \Phi^*_\mu$, where the symbol $*$ denotes the conjugate transpose of the matrix, and formula (12) is the *inverse spectral transform* from the space of coefficients to the space of values.

As mentioned at the beginning of the section, we will employ the Hermite spectral decomposition in some of our experiments (see sections 4.3 and 4.4). Hence, we will use (10) and (12) for the required spectral transforms.

We also remark that a similar approach was pursued in [26] in the framework of three-dimensional Chebyshev interpolation.

4. Numerical comparison

In this section, we will compare the proposed μ-mode integrator with some widely used techniques to solve partial differential equations. For that purpose a range of PDEs, mainly from quantum mechanics, is considered. Concerning the experiments in sections 4.1, 4.2 and 4.5, we will test the proposed method against the following iterative schemes commonly employed to compute the action of the matrix exponential e^{tM}:

- `expmv`: a polynomial method described in [3] which is based on a Taylor expansion of the exponential;
- `phipm`: a full Krylov method presented in [40];
- `kiops`: a Krylov method based on an incomplete orthogonalization process, described in [25].

The `MATLAB` source code of these methods is publicly available. Although the underlying algorithms of these schemes only require the action of the matrix on a vector, only `kiops` is readily available to do that. Therefore, in order to ensure a fair comparison, we feed the functions with the matrix. Moreover, considering the action of the matrix on a vector (which in our case could be performed entirely in tensor formulation by means of sums of μ-mode products) instead of the matrix itself would not result in a speedup for the schemes (see section 4.1). The tolerance for all the algorithms considered has been set to 2^{-53}, which corresponds to the machine epsilon for double precision computations. As a measure of cost, we consider the computational time (wall-clock time) needed to solve numerically the differential equation under consideration up to a fixed final time. As mentioned in section 2, the μ-mode integrator requires the explicit computation of small matrix exponentials. This is performed using the internal `MATLAB`® function `expm`, which is based on the scaling and squaring rational Padé approximation described in [2]. In this context, another method which could be directly used in `MATLAB` is `exptayotf` from [12]. It is based on a backward stable Taylor approximation for the matrix exponential and is faster than `expm`. Moreover, as it works in single, double and variable precision arithmetic data types, it produces approximations with the desired accuracy. This is not possible for the iterative schemes which approximate the action of e^{tM}, because the `MATLAB`® sparse format is restricted to double precision. Another fast method using a similar technique and suited for double precision is `expmpol` from [46]. We will demonstrate that our `MATLAB` implementation of the proposed μ-mode integrator outperforms all the other schemes by at least a factor of 7.

Concerning the experiments in sections 4.3 and 4.4, we compare our μ-mode based approach with a splitting scheme/FFT based space discretization that is well-established and efficient. In order to perform direct and inverse Fourier transforms, we employ the internal `MATLAB`® functions `fftn` and `ifftn` respectively, which are in turn based on the very efficient FFTW library [23]. Care has been taken to ensure that comparisons conducted in `MATLAB`® give a good indication of the performance that would be obtained in a compiled language. This is possible here as the majority part of the computational time is spent in the FFT routines. For these problems, we will show that the μ-mode integrator can reach a speedup of at least 5.

All the tests in this section have been conducted on an Intel Core i7-5500U CPU with 12GB of RAM using `MATLAB`® `R2020b`.

4.1. Code validation

As an introductory test problem, in order to highlight some qualities of our μ-mode method, we consider the three-dimensional heat equation

$$\begin{cases} \partial_t u(t,\mathbf{x}) = \Delta u(t,\mathbf{x}), \qquad \mathbf{x} \in [0,2\pi)^3, \quad t \in [0,T], \\ u(0,\mathbf{x}) = \cos x_1 + \cos x_2 + \cos x_3 \end{cases} \tag{13}$$

with periodic boundary conditions.

The equation is discretized in space using centered finite differences with n_μ grid points in the μth direction (the total number of degrees of freedom stored in computer memory is hence equal to $N = n_1 n_2 n_3$). By doing so we obtain the following ordinary differential equation (ODE)

$$\mathbf{u}'(t) = M\mathbf{u}(t), \tag{14}$$

where $\mathbf{u}$ denotes the vector in which the degrees of freedom are assembled. The exact solution of equation (14) is given by the action of the matrix exponential

$$\mathbf{u}(t) = \mathrm{e}^{tM}\mathbf{u}(0). \tag{15}$$

The matrix M has the following Kronecker structure

$$M = I_3 \otimes I_2 \otimes A_1 + I_3 \otimes A_2 \otimes I_1 + A_3 \otimes I_2 \otimes I_1,$$

where $A_\mu \in \mathbb{R}^{n_\mu \times n_\mu}$ results from the one-dimensional discretization of the operator ∂_μ^2, and $I_\mu \in \mathbb{R}^{n_\mu \times n_\mu}$ is the identity matrix. The quantity $\mathbf{u}(t)$ can be seen as vectorization of the tensor $\mathbf{U}(t)$, and we can write (15) in tensor form as

$$\mathbf{U}(t) = \mathbf{U}(0) \times_1 \mathrm{e}^{tA_1} \times_2 \mathrm{e}^{tA_2} \times_3 \mathrm{e}^{tA_3},$$

where $\mathbf{U}(t)(i_1, i_2, i_3) = \mathbf{u}(t)_{i_1 + n_1(i_2-1) + n_1 n_2 (i_3-1)}$.

We now present three numerical tests.

Test 1. We consider second order centered finite differences and compute the solution at time $T = 1$ for $n_\mu = n$, $\mu = 1, 2, 3$ with various n. We investigate the wall-clock time as a function of the problem size.

Test 2. We fix the problem size ($n_\mu = 40$, $\mu = 1, 2, 3$) and compute the solution at time $T = 1$ for different orders p of the finite difference scheme. We thereby investigate the wall-clock time as a function of the sparsity pattern of M.

Test 3. We consider second order centered finite differences and fix the problem size ($n_\mu = 40$, $\mu = 1, 2, 3$). We then compute the solution at different final times T. By doing so we investigate the wall-clock time as a function of the norm of M.

The corresponding results are shown in Figure 1. We see that the proposed μ-mode integrator is always the fastest algorithm. The difference in computational time is at least a factor of 60.

Concerning the first test, we measure also the relative error between the analytical solution and the numerical one. As the dimensional splitting performed by the μ-mode integrator is exact, its errors are equal to the ones obtained by computing (15) using the other algorithms. Indeed, for the values of n under consideration, we obtain 2.06e-03, 1.09e-03, 6.71e-04, 4.55e-04 and 3.29e-04 for all the methods. We highlight also that the main cost of the μ-mode integrator is represented by the computation of the μ-mode products and not by the exponentiation of the matrices A_μ (see Table 1). Lastly, notice that the iterative algorithms would not have taken advantage from the computation of the internal matrix-vector products, which constitute their main cost, in tensor formulation (i.e. by means of

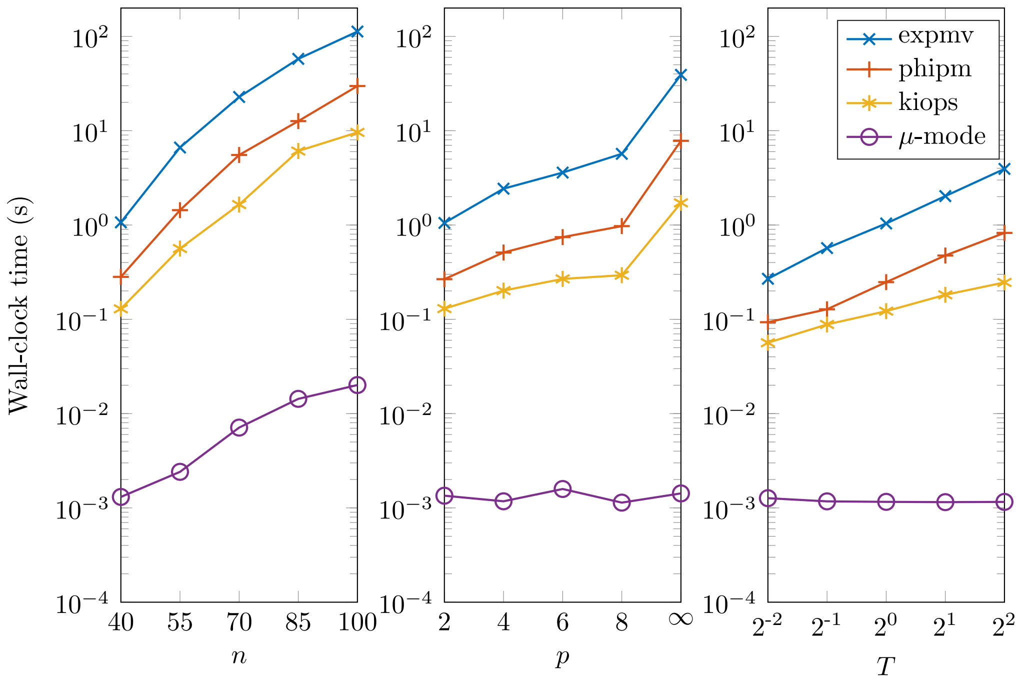

Figure 1: The wall-clock time for solving the heat equation (13) is shown as a function of n (left), of the order of the finite difference scheme p (middle), and of the final time T (right). Note that $p = \infty$ corresponds to a spectral space discretization.

sums of μ-mode products). Indeed, if we measure the wall-clock time for a single action of the matrix on a vector we observe, for the values of n under consideration, a speedup of averagely 1.5 times by using the standard sparse matrix-vector product as opposed to the tensor formulation.

n	40	55	70	85	100
`expm`	0.52	0.71	1.37	3.15	3.54
μ-mode products	0.79	1.71	5.74	10.92	16.89
Total	1.31	2.42	7.11	14.07	20.43

Table 1: Breakdown of wall-clock time (in ms) for the μ-mode integrator for different values of n (cf. left plot of Figure 1).

The second test shows that the iterative schemes see a decrease in performance when decreasing the sparsity of the matrix (i.e. by increasing the order of the method p or by using a spectral approximation). This effect is particularly visible when performing a spectral discretization, which results in full matrices A_μ. On the other hand, the μ-mode integrator is largely unaffected as it computes the exponential of the *full* matrices A_μ, independently of the initial sparsity pattern, by using `expm`.

Similar observations can be made for the third test. While the iterative schemes suffer from increasing computational time as the norm of the matrix increases, for the μ-mode integrator this is not the case. The reason for this is that the scaling and squaring algorithm in `expm` scales very favorably as the norm of the matrix increases.

4.2. Pipe flow

To demonstrate that the μ-mode integrator can be used for some problems with non-constant coefficients, we consider a model for a fluid flowing in a pipe. The main assumptions are that of radial symmetry (i.e. the solution does not depend on the angle variable in the circular cross section, see for example [47]) and a prescribed length-dependent flow velocity. In this case we obtain the following diffusion-advection equation for the concentration c

$$\partial_t c(t,\rho,z) = \alpha\left(\partial_{\rho\rho}c(t,\rho,z) + \frac{1}{\rho}\partial_\rho c(t,\rho,z) + \partial_{zz}c(t,\rho,z)\right) - s(z)\partial_z c(t,\rho,z), \tag{16}$$

where $t \in [0,T]$, $\rho \in [\rho_{\min}, \rho_{\max}]$ and $z \in [0, z_{\max}]$. Here α is the diffusivity and $s(z)$ represents the advection velocity.

After space discretization, which in our case is performed by means of second order centered finite differences with equal number of discretization points n_μ in each direction (i.e. $n_\mu = n$, with $\mu = 1, 2$), the resulting ODE is a linear problem in Kronecker form (6). The system can then be integrated exactly by the μ-mode integrator. For the simulations conducted, we use the following initial and boundary conditions

$$\begin{cases} c(0,\rho,z) = \exp(-8(\rho-\rho_0)^2 - 8(z-z_0)^2), \\ c(t,\rho,0) = 0, \\ \partial_z c(t,\rho,z_{\max}) = 0, \\ \partial_\rho c(t,\rho_{\min},z) = 0, \\ \partial_\rho c(t,\rho_{\max},z) = 0, \end{cases}$$

while the flow velocity is set to

$$s(z) = 2 + \tanh(4(z-5/2)) - \tanh(4(z-5)).$$

The parameters are chosen as $\rho_{\min} = 0.1$, $\rho_{\max} = 5$, $z_{\max} = 8$, $\alpha = 1/90$, $\rho_0 = (\rho_{\min} + \rho_{\max})/2$ and $z_0 = 3/2$. The structure of the problem does not allow an effective use of FFT based methods. The results of the experiment are presented in Figure 2. The μ-mode integrator outperforms all the iterative methods by a consistent factor, with an average speedup of 45 times with respect to `kiops`, the fastest competitor in this simulation.

4.3. Schrödinger equation with time-independent potential

In this section we solve the Schrödinger equation in three space dimensions

$$\begin{cases} \mathrm{i}\partial_t\psi(t,\mathbf{x}) = -\frac{1}{2}\Delta\psi(t,\mathbf{x}) + V(\mathbf{x})\psi(t,\mathbf{x}), \qquad \mathbf{x} \in \mathbb{R}^3, \quad t \in [0,1] \\ \quad \psi(0,\mathbf{x}) = \psi_0(\mathbf{x}) \end{cases} \tag{17}$$

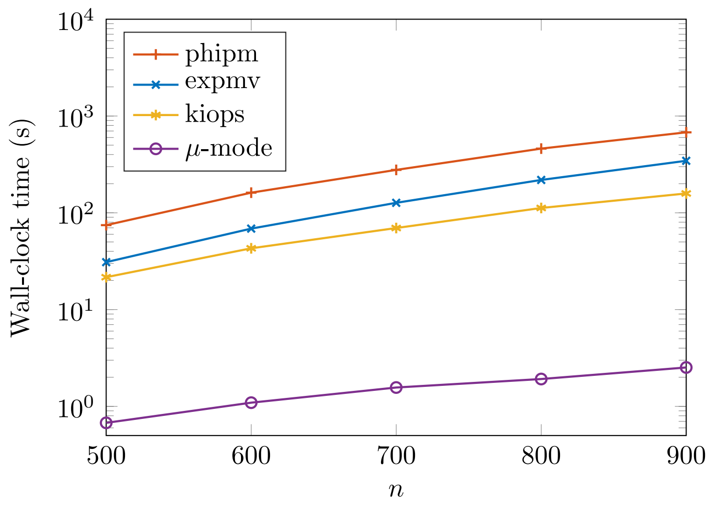

Figure 2: Wall-clock time (in seconds) for the integration of (16) up to $T = 4$ as a function of n (total number of degrees of freedom $N = n^2$).

with a time-independent potential $V(\mathbf{x}) = V_1(x_1) + V_2(x_2) + V_3(x_3)$, where

$$V_1(x_1) = \cos(2\pi x_1), \quad V_2(x_2) = x_2^2/2, \quad V_3(x_3) = x_3^2/2.$$

The initial condition is given by

$$\psi_0(\mathbf{x}) = 2^{-\frac{5}{2}}\pi^{-\frac{3}{4}}(x_1 + \mathrm{i}x_2)\exp\left(-x_1^2/4 - x_2^2/4 - x_3^2/4\right).$$

This equation could be integrated using any of the iterative methods considered in the previous section. However, for reasons of efficiency a time splitting approach is commonly employed. This treats the Laplacian and the potential part of the equations separately. For the former the fast Fourier transform (FFT) can be employed, while an analytic solution is available for the latter. The two partial flows are then combined by means of the Strang splitting scheme. For more details on this Time Splitting Fourier Pseudospectral method (TSFP) we refer the reader to [30].

Another approach is to use a Hermite pseudospectral space discretization. This has the advantage that harmonic potentials are treated exactly, which is desirable in many applications. However, for most of the other potentials, the resulting matrices are full which, for traditional integration schemes, means that using a Hermite pseudospectral discretization is not competitive with respect to TSFP. However, as long as the potential is in Kronecker form, we can employ the μ-mode integrator to perform computations very efficiently. Moreover, the resulting method based on the μ-mode integrator combined with a Hermite pseudospectral space discretization can take arbitrarily large time steps without incurring any time discretization error (as it is exact in time). We call this scheme the Hermite Kronecker Pseudospectral method (HKP).

Before proceeding, let us note that for the TSFP method it is necessary to truncate the unbounded domain. In order to relate the size of the truncated domain to the chosen degrees

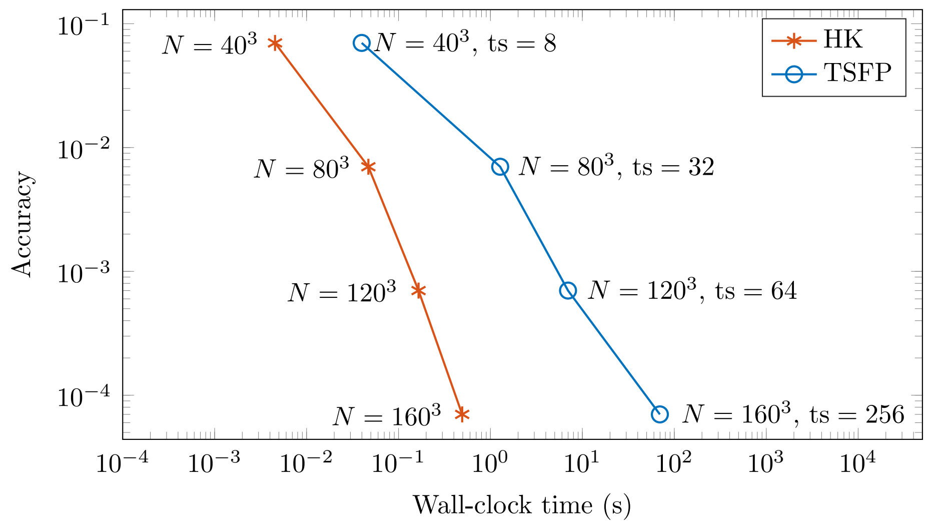

Figure 3: Precision diagram for the integration of the Schrödinger equation with a time-independent potential (17) up to $T = 1$. The number of degrees of freedom N and the number of time steps (ts) are varied in order to achieve a result which is accurate up to the given tolerance. The reference solution has been computed by the HKP method with $N = 300^3$.

of freedom, we considered that, in practice, in the HKP method the domain is implicitly truncated. This truncation is given by the convex hull of the quadrature points necessary to compute the Hermite coefficients corresponding to the initial solution. For any choice of degrees of freedom of the TSFP method, we decided to truncate the unbounded domain to the corresponding convex hull of the quadrature points of the HKP method. In this way, for the same degrees of freedom, the two methods use the same amount of information coming from the same computational domain.

The TSFP and the HKP methods are compared in Figure 3. In both cases, we consider a constant number of space discretization points $n_\mu = n$ for every direction $\mu = 1, 2, 3$ (total number of degrees of freedom $N = n^3$) and integrate the equation until final time $T = 1$ with constant time step size. We see that in terms of wall-clock time the HK method outperforms the TSFP scheme for all levels of accuracy considered here. Also note that the difference in performance increases as we move to more stringent tolerances. The reason for this is that the splitting error forces the TSFP scheme to take relatively small time steps.

4.4. Schrödinger equation with time-dependent potential

Let us now consider the Schrödinger equation

$$\begin{cases} \partial_t \psi(t, \mathbf{x}) = H(t, \mathbf{x})\psi(t, \mathbf{x}), & \mathbf{x} \in \mathbb{R}^3, \quad t \in [0, 1] \\ \psi(0, \mathbf{x}) = 2^{-\frac{5}{2}}\pi^{-\frac{3}{4}}(x_1 + \mathrm{i}x_2)\exp\left(-x_1^2/4 - x_2^2/4 - x_3^2/4\right), \end{cases} \tag{18}$$

where the Hamiltonian is given by

$$H(\mathbf{x}, t) = \frac{\mathrm{i}}{2}\Big(\Delta - x_1^2 - x_2^2 - x_3^2 - 2x_3 \sin^2 t\Big).$$

Note that the potential is now time-dependent, as opposed to the case presented in section 4.3. Such potentials commonly occur in applications, e.g. when studying laser-atom interactions (see, for example, [42]).

Similarly to what we did in the time-independent case, we can use a time splitting approach: the Laplacian part can still be computed efficiently in Fourier space, but now the potential part has no known analytical solution. Hence, for the numerical solution of the latter, we will employ an order two Magnus integrator, also known as the exponential midpoint rule. Let

$$u'(t) = A(t)u(t)$$

be the considered ODE with time-dependent coefficients, and let u_n be the numerical approximation to the solution at time t_n. Then, the exponential midpoint rule provides the numerical solution

$$u_{n+1} = \exp\bigl(\tau_n A(t_n + \tau_n/2)\bigr)u_n \tag{19}$$

at time $t_{n+1} = t_n + \tau_n$, where τ_n denotes the chosen step size. The two partial flows are then combined together by means of the Strang splitting scheme. We call this scheme the Time Splitting Fourier Magnus Pseudospectral method (TSFMP). For the domain truncation needed in this approach, the same reasoning as in the time-independent case applies.

Another technique is to perform a Hermite pseudospectral space discretization. However, as opposed to the case in section 4.3, the resulting ODE cannot be integrated exactly in time. For the time discretization, we will then use the order two Magnus integrator (19). We call the resulting scheme Hermite Kronecker Magnus Pseudospectral method (HKMP).

The results of the experiments are depicted in Figure 4. In both cases, we consider a constant number of space discretization points $n_\mu = n$ for every direction $\mu = 1, 2, 3$ (total number of degrees of freedom $N = n^3$) and solve the equation until final time $T = 1$ with constant time step size. Moreover, concerning the TSFMP method, we integrate the subflow corresponding to the potential part with a single time step. Again, as we observed in the time-independent case, the HKMP method outperforms the TSFMP scheme in any case. Notice in particular that, for the chosen degrees of freedom and time steps, the TSFMP method is not able to reach an accuracy of 1e-07, while the HKMP is.

4.5. Nonlinear Schrödinger/Gross–Pitaevskii equation

In this section we consider the nonlinear Schrödinger equation

$$\partial_t \psi(t, \mathbf{x}) = \frac{\mathrm{i}}{2}\Delta\psi(t, \mathbf{x}) + \frac{\mathrm{i}}{2}\Bigl(1 - |\psi(t, \mathbf{x})|^2\Bigr)\psi(t, \mathbf{x}), \tag{20}$$

which is also known as Gross–Pitaevskii equation. The unknown ψ represents the wave function, $\mathbf{x} \in \mathbb{R}^3$, $t \in [0, 25]$, and the initial condition is constituted by the superimposition of two straight vortices in a background density $|\psi_\infty|^2 = 1$, in order to replicate the classical experiment of vortex reconnection (see [13] and the references therein for more details).

The initial datum and the boundary conditions given by the background density make it quite difficult to use artificial periodic boundary conditions in a truncated domain, unless an expensive mirroring of the domain in the three dimensions is carried out. Therefore, in order

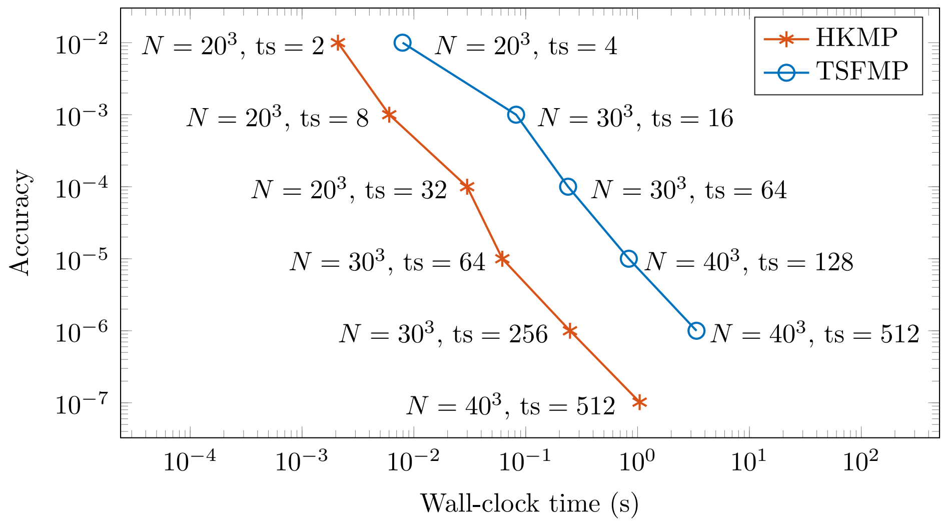

Figure 4: Precision diagram for the integration of the Schrödinger equation with a time-dependent potential (18) up to $T = 1$. The number of degrees of freedom N and the number of time steps (ts) are varied in order to achieve a result which is accurate up to the given tolerance. The reference solution has been computed by the HKMP method with $N = 100^3$ and ts = 2048.

to solve (20) numerically, we consider the Time Splitting Finite Difference method proposed in [13]. More specifically, we truncate the unbounded domain to $\mathbf{x} \in [-20, 20]^3$ and discretize by non-uniform finite differences with homogeneous Neumann boundary conditions. The number n_μ of discretization points is the same in each direction, i.e. $n_\mu = n$, with $\mu = 1, 2, 3$. After a proper transformation of variables in order to recover symmetry, we end up with a system of ODEs of the form

$$\boldsymbol{\psi}'(t) = \frac{\mathrm{i}}{2} M_W \boldsymbol{\psi}(t) + \frac{\mathrm{i}}{2}\Big(1 - W^{-1}|\boldsymbol{\psi}(t)|^2\Big)\boldsymbol{\psi}(t),$$

where M_W is a matrix in Kronecker form and W is a diagonal weight matrix. Then, we employ a Strang splitting scheme for the time integration, in which the linear part is solved either by means of the μ-mode integrator or by using the iterative methods indicated at the beginning of section 4. The nonlinear subflow is integrated exactly.

The results of the experiment are presented in Figure 5. The μ-mode integrator outperforms `expmv` by approximately a factor of 7. The speedup compared to both `phipm` and `kiops` is even larger.

5. Implementation on multi-core CPUs and GPUs

It has increasingly been realized that in order to fully exploit present and future high-performance computing systems we require algorithms that parallelize well and which can be implemented efficiently on accelerators, such as GPUs [5]. In particular, for GPU computing much research effort has been undertaken to obtain efficient implementations (see, e.g., [6, 8, 17, 18, 19, 33, 36, 43, 45, 49]).

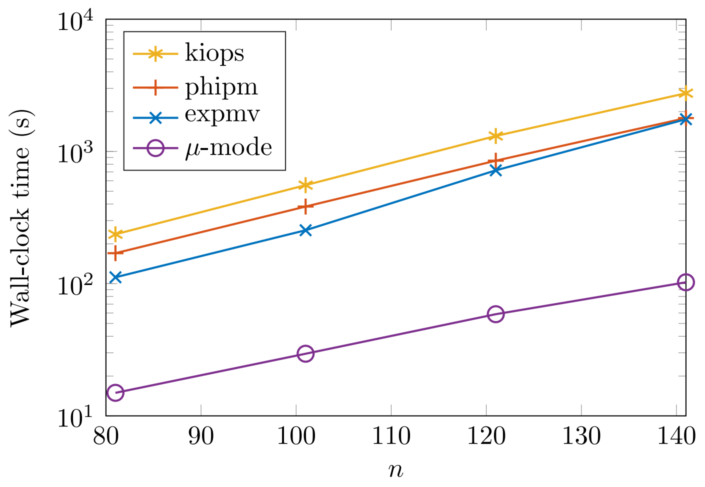

Figure 5: Wall-clock time (in seconds) for the integration of (20) up to $T = 25$ as a function of n (total number of degrees of freedom $N = n^3$). A constant time step size $\tau = 0.1$ is employed.

In this section we will consider an efficient implementation of the proposed μ-mode integrator on multi-core CPUs and GPUs. We note that all modern hardware platforms are much better at performing floating point operations (such as addition and multiplication) than they are at accessing data in memory. This favors algorithms with a high flop/byte ratio; that is, algorithms that perform many floating point operations for every byte that is loaded from or written to memory. The μ-mode product of a square matrix for an array of size $n_1 \times \cdots \times n_{\mu-1} \times n_\mu \times n_{\mu+1} \times \cdots \times n_d$ is computed using a matrix-matrix multiplication of size $n_\mu \times n_\mu$ times $n_\mu \times (n_1 \cdots n_{\mu-1} n_{\mu+1} \cdots n_d)$, see section 2 for more details. For moderate n_μ the relatively small $n_\mu \times n_\mu$ matrix can be kept in cache and thus $\mathcal{O}(n_\mu N)$ arithmetic operations are performed compared to $\mathcal{O}(N)$ memory operations, where $N = n_1 \cdots n_d$ is the total number of degrees of freedom. Thus, the flop/byte ratio of the algorithm is $\mathcal{O}(n_\mu)$, which makes it ideally suited to modern computer hardware. This is particularly true when the μ-mode integrator is compared to an implicit scheme implemented with sparse matrix-vector products. In this case the flop/byte ratio is only $\mathcal{O}(1)$ and modern CPU and GPUs will spend most of their time waiting for data that is fetched from memory.

To make this analysis more precise, we have to compare the flop/byte ratio of the algorithm to that of the hardware. For the benchmarks in this section we will use a multi-core CPU system based on a dual socket Intel Xeon Gold 5118 with 2×12 cores. The system has a peak floating point performance of 1.8 TFlops/s (double precision) and a theoretical peak memory bandwidth of 256 GB/s. Thus, during the time a double precision floating point number is fetched from memory approximately 56 arithmetic operations can be performed. In addition, we will use a NVIDIA V100 GPU with 7.5 TFlop/s double precision performance and 900 GB/s peak memory bandwidth (approximately 67 arithmetic operations can be performed for each number that is fetched from memory). Due to their large floating point performance we expect the algorithm to perform well on GPUs. A feature of the V100 GPU is that it contains so-called tensor cores that can dramatically accelerate half-precision computations (up to 125 Tflops/s). Tensor cores are primarily designed for machine learning tasks, but they can also be exploited for matrix-matrix products (see, e.g., [1, 34]).

For reasonably large n_μ the proposed μ-mode integrator is thus compute bound. However, since very efficient (close to the theoretical peak performance) matrix-matrix routines are available on both of these platforms, one can not be entirely indifferent towards memory operations. There are two basic ways to implement the algorithm. The first is to explicitly form the $n_\mu \times (n_1 \cdots n_{\mu-1} n_{\mu+1} \cdots n_d)$ matrix. This has the advantage that a single matrix-matrix multiplication (gemm) can be used to perform each μ-mode product and that the corresponding operands have the proper sequential memory layout. The disadvantage is that a permute operation has to be performed before each μ-mode product is computed. This is an extremely memory bound operation with strided access for which the floating point unit in the CPU or GPU lies entirely dormant. Thus, while this is clearly the favored approach in a `MATLAB` implementation, it does not achieve optimal performance. The approach we have chosen in this section is to directly perform the μ-mode products on the multi-dimensional array stored in memory (without altering the memory layout in between such operations).

Both Intel MKL and cuBLAS provide appropriate batched gemm routines (`cblas_gemm_batch` for Intel MKL and `cublasGemmStridedBatched` for cuBLAS) that are heavily optimized, and we will make use of those library functions in our implementation (for more details on these routines we refer to [14]). Our code is written in C++ and uses CUDA for the GPU implementation.

Before proceeding, let us briefly discuss how the μ-mode integrator would perform in a distributed memory setting (i.e. when parallelized using MPI). Since, in general, the matrix exponentials are full matrices, each degree of freedom along a coordinate axis couples with each other degree of freedom on that same axis. This data communication pattern is similar to computing a FFT. Thus, we would expect the μ-mode product to scale comparable to FFT on a distributed memory system. This would be worse than a stencil code. However, one should keep in mind that the μ-mode integrator can take much larger time steps. Thus, the overall communication overhead to compute the solution at a specified final time could still be larger for an explicit or an iterative method.

In the remainder of this section we will present benchmark results for our implementations. The speedups are always calculated as ratio between the wall-clock time needed by the CPU and the one needed by the GPU.

5.1. Heat equation

We consider the same problem as in section 4.1, Test 1. The wall-clock time for computing the matrix exponentials and a single time step of the proposed algorithm is listed in Table 2. We consider both a CPU implementation using MKL (double and single precision) and a GPU implementation based on cuBLAS (double, single, and half precision). The GPU implementation outperforms the CPU implementation by a factor of approximately 13. Using half-precision computations on the GPU results in another performance increase by approximately a factor of 2. The relative error with respect to the analytical solution reached by the double precision and single precision, for both CPU and GPU and the values of n under consideration, are 8.22e-05, 3.66e-05, 2.06e-05, 1.47e-05. Results in half precision are not reported as the accuracy of the method is lower than the precision itself.

n	exp	double			single			half
		CPU	GPU	speedup	CPU	GPU	speedup	GPU
200	2.92	38.39	2.66	14.4x	19.48	1.33	14.6x	0.39
300	4.88	136.17	8.90	15.3x	81.65	5.27	15.5x	2.73
400	10.14	310.11	29.88	10.4x	161.97	16.89	9.6x	6.68
500	17.74	711.07	52.86	13.5x	373.36	30.51	12.2x	15.43

Table 2: Wall-clock time for the heat equation (13) discretized using second-order centered finite differences with n^3 degrees of freedom. The time for computing the matrix exponentials (exp) and for one step of the μ-mode integrator are listed (in ms). The speedup is the ratio between the single step performed in CPU and GPU, in double and single precision. The matrix exponential is always computed in double precision.

For a number of simulations conducted we observed a drastic reduction in performance for single precision computations when using Intel MKL. To illustrate this we consider the heat equation

$$\begin{cases} \partial_t u(t,\mathbf{x}) = \Delta u(t,\mathbf{x}), \qquad \mathbf{x} \in \left[-\frac{11}{4}, \frac{11}{4}\right]^3, \quad t \in [0,1], \\ u(0,\mathbf{x}) = \left(x_1^4 + x_2^4 + x_3^4\right) \exp\left(-x_1^4 - x_2^4 - x_3^4\right) \end{cases} \tag{21}$$

with (artificial) Dirichlet boundary conditions, discretized in space as above. From Table 3 we see that the performance of single precision computations with Intel MKL can be worse by a factor of 3.5 compared to double precision, which obviously completely defeats the purpose of doing so. The reason for this performance degradation are so-called denormal numbers, i.e. floating point numbers with leading zeros in the mantissa. Since there is no reliable way to disable denormal numbers on modern x86-64 systems, we avoid them by scaling the initial value in an appropriate way. Since this is a linear problem, the scaling can easily be undone after the computation. The results with the scaling workaround, listed in Table 3, now show the expected behavior (that is, single precision computations are approximately twice as fast as double precision ones). We note that this is *not* an issue with our μ-mode integrator but rather an issue with Intel MKL. The cuBLAS implementation is free from this artifact and thus no normalization is necessary on the GPU.

5.2. Schrödinger equation with time-independent potential

We consider the Schrödinger equation with time-independent potential from section 4.3. The equation is integrated up to $T = 1$ in a single step, as for this problem no error is introduced by the μ-mode integrator. For the space discretization the Hermite pseudospectral discretization is used. The results for both the CPU and GPU implementation are listed in Table 4. The GPU implementation, for both single and double precision, shows a speedup of approximately 15 compared to the CPU implementation.

n	exp	double		single		scaled single	half
		CPU	GPU	CPU	GPU	CPU	GPU
200	2.92	38.80	2.64	92.19	1.34	19.98	0.38
300	6.01	157.41	8.87	385.84	5.22	71.24	2.71
400	13.40	314.96	29.85	1059.78	16.86	154.84	6.67
500	30.19	702.48	52.92	2567.56	30.42	367.34	13.44

Table 3: Wall-clock time for the heat equation (21) discretized using second order centered finite differences with n^3 degrees of freedom. The performance degradation in CPU due to denormal numbers disappears when using the scaling workaround (scaled single). Speedups are not computed in this case.

n	double				single			
	exp	CPU	GPU	speedup	exp	CPU	GPU	speedup
127	5.56	20.89	1.27	16.4x	4.71	13.71	0.64	21.4x
255	8.31	224.13	16.02	13.9x	5.16	134.21	8.11	16.5x
511	50.79	3121.42	219.13	14.2x	28.01	1824.93	119.46	15.2x

Table 4: Wall-clock time for the linear Schrödinger equation with time-independent potential (17) integrated with the HKP method (n^3 degrees of freedom). The time for computing the matrix exponential (exp) and for one step of the μ-mode integrator are listed (in ms). The speedup is the ratio between the single step performed in CPU and GPU, in double and single precision.

5.3. Schrödinger equation with time-dependent potential

We consider once again the Schrödinger equation with the time-dependent potential from section 4.4 solved with the HKMP method. The equation is integrated up to $T = 1$ with time step $\tau = 0.02$. The results are given in Table 5. In this case, the matrix exponential changes as we evolve the system in time. Thus, the performance of computing the matrix exponential has to be considered alongside the μ-mode products. On the CPU this is not an issue as the time required for the matrix exponential is significantly smaller than the time required for the μ-mode products. However, for the GPU implementation and small problem sizes it is necessary to perform the matrix exponential on the GPU as well. To do this we have implemented an algorithm based on a Taylor backward stable approach. Overall, we observe a speedup of approximately 15 by going from the CPU to the GPU (for both single and double precision).

6. Conclusions

We have shown that with the proposed μ-mode integrator we can make use of modern computer hardware to efficiently solve a number of partial differential equations. In particular, we have demonstrated that for Schrödinger equations the approach can outperform

n	double					
	exp (ext)	CPU		GPU		speedup
		exp (int)	μ-mode	exp (int)	μ-mode	
127	0.02	2.56	19.38	0.37	1.05	15.3x
255	0.05	4.52	200.46	0.66	13.79	14.2x
511	0.07	29.71	3043.88	2.38	213.21	14.3x

n	single					
	exp (ext)	CPU		GPU		speedup
		exp (int)	μ-mode	exp (int)	μ-mode	
127	0.01	2.16	12.51	0.25	0.54	18.9x
255	0.03	2.88	100.35	0.34	7.01	13.9x
511	0.05	14.25	1600.86	1.09	108.31	14.8x

Table 5: Wall-clock time for the Schrödinger equation with time-dependent potential (18) integrated with the HKMP method (n^3 degrees of freedom). The time for computing the matrix exponentials and for one step of the μ-mode integrator is listed (in ms). The acronym exp (ext) refers to exponentiation of the time-independent matrices, which are diagonal, while exp (int) refers to the time-dependent ones that have to be computed at each time step. The speedup is the ratio between the single step performed in CPU and GPU, in double precision (top) and single precision (bottom).

well-established integrators in the literature by a significant margin. This was also possible thanks to the usage of the μ-mode product to efficiently compute spectral transforms, which can be beneficial even in applications that are not related to solving partial differential equations. The proposed integrator is particularly efficient on GPUs too, as we have demonstrated, which is a significant asset for running simulation on the current and next generation of supercomputers.

Acknowledgments

The authors would like to thank the Italian Ministry of Instruction, University and Research (MIUR) for partially supporting this research with funds coming from PRIN Project 2017 (No. 2017KKJP4X entitled "Innovative numerical methods for evolutionary partial differential equations and applications").

The authors acknowledge partial support from the Program Ricerca di Base 2019 of the University of Verona entitled "Geometric Evolution of Multi Agent Systems".

FZ has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 850941).

This work is supported by the Austrian Science Fund (FWF) – project id: P32143-N32.

References

[1] A. Abdelfattah, S. Tomov, and J. Dongarra. Fast batched matrix multiplication for small sizes using half-precision arithmetic on GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 111–122. IEEE, 2019.

[2] A.H. Al-Mohy and N.J. Higham. A new scaling and squaring algorithm for the matrix exponential. *SIAM J. Matrix Anal. Appl.*, 31(3):970–989, 2009.

[3] A.H. Al-Mohy and 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.

[4] U.M. Ascher and S. Reich. The midpoint scheme and variants for Hamiltonian systems: advantages and pitfalls. *SIAM J. Sci. Comput.*, 21(3):1045–1065, 1999.

[5] S. Ashby et al. *The opportunities and challenges of exascale computing.* Report of the ASCAC Subcommittee on Exascale Computing, U.S. Department of Energy, 2010.

[6] N. Auer, L. Einkemmer, P. Kandolf, and A. Ostermann. Magnus integrators on multicore CPUs and GPUs. *Comput. Phys. Commun.*, 228:115–122, 2018.

[7] W. Bao and J. Shen. A fourth-order time-splitting Laguerre–Hermite pseudospectral method for Bose–Einstein condensates. *SIAM J. Sci. Comput.*, 26(6):2010–2028, 2005.

[8] M. Bussmann et al. Radiative signatures of the relativistic Kelvin–Helmholtz instability. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2013.

[9] M. Caliari, F. Cassini, and F. Zivcovich. Approximation of the matrix exponential for matrices with skinny fields of values. *BIT Numer. Math.*, 60(4):1113–1131, 2020.

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

[11] M. Caliari, P. Kandolf, and F. Zivcovich. Backward error analysis of polynomial approximations for computing the action of the matrix exponential. *BIT Numer. Math.*, 58(4):907–935, 2018.

[12] M. Caliari and F. Zivcovich. On-the-fly backward error estimate for matrix exponential approximation by Taylor algorithm. *J. Comput. Appl. Math.*, 346:532–548, 2019.

[13] M. Caliari and S. Zuccher. Reliability of the time splitting Fourier method for singular solutions in quantum fluids. *Comput. Phys. Commun.*, 222:46–58, 2018.

[14] C. Cecka. Pro Tip: cuBLAS Strided Batched Matrix Multiply. `https://developer.nvidia.com/blog/cublas-strided-batched-matrix-multiply`, 2017.

[15] N. Crouseilles, L. Einkemmer, and J. Massot. Exponential methods for solving hyperbolic problems with application to collisionless kinetic equations. *J. Comput. Phys.*, 420:109688, 2020.

[16] N. Crouseilles, L. Einkemmer, and M. Prugger. An exponential integrator for the drift-kinetic model. *Comput. Phys. Commun.*, 224:144–153, 2018.

[17] L. Einkemmer. A mixed precision semi-Lagrangian algorithm and its performance on accelerators. In *2016 International Conference on High Performance Computing & Simulation (HPCS)*, pages 74–80. IEEE, 2016.

[18] L. Einkemmer. Evaluation of the Intel Xeon Phi 7120 and NVIDIA K80 as accelerators for two-dimensional panel codes. *PloS ONE*, 12(6):e0178156, 2017.

[19] L. Einkemmer. Semi-Lagrangian Vlasov simulation on GPUs. *Comput. Phys. Commun.*, 254:107351, 2020.

[20] L. Einkemmer, M. Moccaldi, and A. Ostermann. Efficient boundary corrected Strang splitting. *Appl. Math. Comput.*, 332:76–89, 2018.

[21] L. Einkemmer and A. Ostermann. Overcoming order reduction in diffusion-reaction splitting. Part 1: Dirichlet boundary conditions. *SIAM J. Sci. Comput.*, 37(3):A1577–A1592, 2015.

[22] F. Filbet and E. Sonnendrücker. Comparison of Eulerian Vlasov solvers. *Comput. Phys. Commun.*, 150(3):247–266, 2003.

[23] M. Frigo and S.G. Johnson. FFTW: An adaptive software architecture for the FFT. In *1998 IEEE International Conference on Acoustics, Speech and Signal Processing, (ICASSP'98)*, pages 1381–1384. IEEE, 1998.

[24] M. Gasteiger, L. Einkemmer, A. Ostermann, and D. Tskhakaya. Alternating direction implicit type preconditioners for the steady state inhomogeneous Vlasov equation. *J. Plasma Phys.*, 83:705830107, 2017.

[25] S. Gaudreault, G. Rainwater, and M. Tokman. KIOPS: A fast adaptive Krylov subspace solver for exponential integrators. *J. Comput. Phys.*, 372(1):236–255, 2018.

[26] B. Hashemi and L.N. Trefethen. Chebfun in three dimensions. *SIAM Journal on Scientific Computing*, 39(5):C341–C363, 2017.

[27] M. Hochbruck and J. Köhler. On the efficiency of the Peaceman–Rachford ADI-dG method for wave-type problems. In *Numerical Mathematics and Advanced Applications ENUMATH 2017*, pages 135–144. Springer, 2019.

[28] M. Hochbruck and A. Ostermann. Exponential integrators. *Acta Numer.*, 19:209–286, 2010.

[29] W. Hundsdorfer and J.G. Verwer. *Numerical solution of time-dependent advection-diffusion-reaction equations*. Springer, 2013.

[30] S. Jin, P. Markowich, and C. Sparber. Mathematical and computational methods for semiclassical Schrödinger equations. *Acta Numer.*, 20:121–209, 2011.

[31] T.G. Kolda. Multilinear operators for higher-order decompositions. Technical Report SAND2006-2081, Sandia National Laboratories, April 2006.

[32] T.G. Kolda and B.W. Bader. Tensor decompositions and applications. *SIAM Rev.*, 51(3):455–500, 2009.

[33] D.I. Lyakh. An efficient tensor transpose algorithm for multicore CPU, Intel Xeon Phi, and NVidia Tesla GPU. *Comput. Phys. Commun.*, 189:84–91, 2015.

[34] S. Markidis, S.W. Chien, E. Laure, I.B. Peng, and J.S. Vetter. NVIDIA tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531. IEEE, 2018.

[35] R.I. McLachlan and G.R.W. Quispel. Splitting methods. *Acta Numerica*, 11:341–434, 2002.

[36] M. Mehrenberger, C. Steiner, L. Marradi, N. Crouseilles, E. Sonnendrücker, and B. Afeyan. Vlasov on GPU (VOG project). *ESAIM: Proc.*, 43:37–58, 2013.

[37] T. Namiki. A new FDTD algorithm based on alternating-direction implicit method. *IEEE Trans. Microw. Theory Tech.*, 47(10):2003–2007, 1999.

[38] H. Neudecker. A note on Kronecker matrix products and matrix equation systems. *SIAM Journal on Applied Mathematics*, 17(3):603–606, 1969.

[39] Q. Nie, F.Y.M. Wan, Y.-T. Zhang, and X.-F. Liu. Compact integration factor methods in high spatial dimensions. *J. Comput. Phys.*, 227:5238–5255, 2008.

[40] J. Niesen and W.M. Wright. Algorithm 919: A Krylov subspace algorithm for evaluating the φ-functions appearing in exponential integrators. *ACM Trans. Math. Software*, 38(3):1–19, 2012.

[41] D.W. Peaceman and H.H. Rachford Jr. The numerical solution of parabolic and elliptic differential equations. *J. Soc. Ind. Appl.*, 3(1):28–41, 1955.

[42] U. Peskin, R. Kosloff, and N. Moiseyev. The solution of the time dependent Schrödinger equation by the (t, t') method: The use of global polynomial propagators for time dependent Hamiltonians. *J. Chem. Phys.*, 100(12):8849–8855, 1994.

[43] P.S. Rawat, M. Vaidya, A. Sukumaran-Rajam, M. Ravishankar, V. Grover, A. Rountev, L. Pouchet, and P. Sadayappan. Domain-specific optimization and generation of high-performance GPU code for stencil computations. *Proceedings of the IEEE*, 106(11):1902–1920, 2018.

[44] Y. Saad. *Iterative methods for sparse linear systems.* SIAM, 2003.

[45] A. Sandroos, I. Honkonen, S. von Alfthan, and M. Palmroth. Multi-GPU simulations of Vlasov's equation using Vlasiator. *Parallel Comput.*, 39(8):306–318, 2013.

[46] J. Sastre, J. Ibáñez, and E. Defez. Boosting the computation of the matrix exponential. *Applied Mathematics and Computation*, 340:206–220, 2019.

[47] G.I. Taylor. Dispersion of soluble matter in solvent flowing slowly through a tube. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 219(1137):186–203, 1953.

[48] M. Thalhammer, M. Caliari, and C. Neuhauser. High-order time-splitting Hermite and Fourier spectral methods. *J. Comput. Phys.*, 228(3):822–832, 2009.

[49] M. Wiesenberger, L. Einkemmer, M. Held, A. Gutierrez-Milla, X. Saez, and R. Iakymchuk. Reproducibility, accuracy and performance of the Feltor code and library on parallel computer architectures. *Comput. Phys. Commun.*, 238:145–156, 2019.