University of Trento
Department of Mathematics

Doctoral Programme in Mathematics
in agreement with the University of Verona

35th cycle

PhD thesis

Exponential integrators:
tensor structured problems and applications

Candidate
Cassini Fabio

Supervisor
Ch.mo Prof. Caliari Marco

April 21, 2023

Abstract

The solution of stiff systems of Ordinary Differential Equations (ODEs), that typically arise after spatial discretization of many important evolutionary Partial Differential Equations (PDEs), constitutes a topic of wide interest in numerical analysis. A prominent way to numerically integrate such systems involves using *exponential integrators*. In general, these kinds of schemes do not require the solution of (non)linear systems but rather the action of the matrix exponential and of some specific exponential-like functions (known in the literature as φ-functions). In this PhD thesis we aim at presenting efficient tensor-based tools to approximate such actions, both from a theoretical and from a practical point of view, when the problem has an underlying Kronecker sum structure. Moreover, we investigate the application of exponential integrators to compute numerical solutions of important equations in various fields, such as plasma physics, mean-field optimal control and computational chemistry. In any case, we provide several numerical examples and we perform extensive simulations, eventually exploiting modern hardware architectures such as multi-core Central Processing Units (CPUs) and Graphic Processing Units (GPUs). The results globally show the effectiveness and the superiority of the different approaches proposed.

Acknowledgments

The person who deserves my deepest and most sincere thanks is Marco Caliari, for the endless amount of time that he spent for me during these years of my PhD. I have the immense fortune to say that these are not formal or forced thanks, and I do not think that I need to spend many more words in this regard. Simply defining him as a supervisor would be way too reductive, and I hope he is proud of me and my path as much as I am pleased to have been his student.

A special thanks also to the numerical analysis group of the University of Innsbruck. In particular, I would like to thank Alexander Ostermann and Lukas Einkemmer, who gently welcomed me as a visiting PhD student. Many thanks also to all the wonderful people that I met there outside the university environment, whose friendship lasts despite the distance.

Furthermore, I would like to thank all the colleagues and friends that spent some time in the office in Ca' Vignal 2, without which the days would have been way more boring. The trips, the lunches, the happy hours, and the dinners without you would not have been the same. A sincere thanks also to the professors of the mathematicians' corridor, with whom I shared many unmissable coffee breaks.

Many thanks to all my longtime nerd friends from Brescia, too, whom I can always count on. The board games, role-playing games, and videogames sessions with you are always unforgettable.

Last but not least, I would like to thank my family, that never stopped supporting and bearing me. Also in this case, I do not think that I need to spend many more words. If I did what I did and I became the person I am now, it is mainly thanks to you.

Contents

Introduction **1**

I Tensor structured problems **5**

1 The μ-mode integrator and exponential-like schemes **7**
1.1 Introduction 7
1.2 The μ-mode integrator for evolution equations in Kronecker form 8
1.3 Application of the μ-mode product to spectral decomposition and reconstruction 12
1.4 Numerical comparison 13
1.4.1 Code validation 14
1.4.2 Pipe flow 16
1.4.3 Schrödinger equation with time-independent potential 17
1.4.4 Schrödinger equation with time-dependent potential 18
1.4.5 Nonlinear Schrödinger/Gross–Pitaevskii equation 20
1.5 Implementation on multi-core CPUs and GPUs 20
1.5.1 Heat equation 22
1.5.2 Schrödinger equation with time-independent potential 22
1.5.3 Schrödinger equation with time-dependent potential 23
1.6 Conclusions 23

2 KronPACK: a μ-mode approach for tensor structured problems **25**
2.1 Introduction 25
2.2 The μ-mode product and its applications 27
2.3 Problems formulation in d dimensions 29
2.3.1 Pseudospectral decomposition 30
2.3.2 Function approximation 31
2.3.3 Action of the matrix exponential 32
2.3.4 Preconditioning of linear systems 34
2.4 Numerical experiments 35
2.4.1 Code validation 35
2.4.2 Hermite–Laguerre–Fourier function decomposition 36
2.4.3 Multivariate interpolation 37
2.4.4 Linear evolutionary equation 39
2.4.5 Semilinear evolutionary equation 40
2.5 Conclusions 41
Appendix 42

3 PHIKS: actions of φ-functions of Kronecker sums 43
3.1 Introduction . . . 43
3.2 Approximation of φ-functions of a Kronecker sum . . . 44
3.2.1 Choice of s, q, and quadrature formula . . . 47
3.3 Numerical experiments . . . 48
3.3.1 Code validation . . . 50
3.3.2 Evolutionary advection–diffusion–reaction equation . . . 50
3.3.3 Allen–Cahn equation . . . 51
3.4 Conclusions . . . 52

4 PHISPLIT: direction splitting of φ-functions for exponential integrators 55
4.1 Introduction . . . 55
4.2 Recall of some exponential integrators of order up to two . . . 57
4.3 Direction splitting of φ-functions . . . 59
4.3.1 Evaluation of small sized matrix φ-functions . . . 60
4.3.2 Practical implementation of the exponential integrators . . . 60
4.4 Numerical experiments . . . 61
4.4.1 2D experiment: linear quadratic control . . . 62
4.4.2 3D experiment: advection–diffusion–reaction . . . 65
4.5 Conclusions . . . 66

II Applications 69

5 Ensign: dynamical low-rank 6D Vlasov simulation 71
5.1 Introduction . . . 71
5.2 Low-rank approximation for the Vlasov–Poisson equations . . . 72
5.3 Matrix formulation of the semi-discrete algorithm . . . 74
5.3.1 Order 1 low-rank projector-splitting algorithm . . . 76
5.3.2 Order 2 low-rank projector-splitting algorithm . . . 76
5.4 Time and space discretization of K, S and L steps . . . 77
5.5 The `Ensign` framework and implementation . . . 78
5.6 Numerical experiments . . . 81
5.6.1 Orders of convergence . . . 82
5.6.2 Linear Landau simulation . . . 82
5.6.3 Two stream instability simulation . . . 83
5.7 Performance results . . . 83
5.7.1 CPU/GPU comparison . . . 84
5.7.2 Varying rank . . . 85
5.8 Conclusions . . . 85

6 Exponential integrators for mean-field selective optimal control problems 91
6.1 Introduction . . . 91
6.2 Mean-field selective optimal control problem . . . 93
6.2.1 First order optimality conditions . . . 93
6.3 Numerical integrators for the semidiscretized equations . . . 94
6.3.1 Forward PDE . . . 95
6.3.2 Backward PDE . . . 96
6.3.3 Matrix functions evaluation . . . 97
6.4 Numerical experiments . . . 98
6.4.1 Control in opinion dynamics: Sznajd model . . . 98
6.4.2 Crowd dynamics: fast exit scenario . . . 98
6.4.3 Mass transfer problem via optimal control . . . 100

6.5 Conclusions 102

7 Efficient exponential integration of inhomogeneous ADR equations 103

7.1 Introduction 103
7.2 Linear stability analysis 104
7.3 Numerical examples 108
7.3.1 One-dimensional linear diffusion equation 108
7.3.2 One-dimensional diffusion–reaction equation 109
7.3.3 Three-dimensional advection–diffusion–reaction equation 112
7.4 Conclusions 113

Bibliography 128

Introduction

The numerical integration of systems of Ordinary Differential Equations (ODEs) is of great interest for many fields of science and engineering. In particular, systems of the form

$$\begin{cases} \boldsymbol{u}'(t) = \boldsymbol{f}(t, \boldsymbol{u}(t)) = M\boldsymbol{u}(t) + \boldsymbol{g}(t, \boldsymbol{u}(t)), \quad t \in [0, T], \\ \boldsymbol{u}(0) = \boldsymbol{u}_0, \end{cases} \tag{1}$$

typically arise after semidiscretization in a spatial domain $\Omega \subseteq \mathbb{R}^d$ of evolutionary Partial Differential Equations (PDEs), which in turn are at the basis of many models for physical and chemical phenomena (see, for instance, References [113, 162]). Here, $\boldsymbol{u}\colon [0, T] \to \mathbb{C}^N$ is the unknown vector, being T the final simulation time and N the total number of degrees of freedom, $M \in \mathbb{C}^{N\times N}$ is a matrix which represents the linear part of the system, and $\boldsymbol{f}\colon [0, T] \times \mathbb{C}^N \to \mathbb{C}^N$ and $\boldsymbol{g}\colon [0, T] \times \mathbb{C}^N \to \mathbb{C}^N$ are generic nonlinear functions. Problems in form (1) can also be seen more generally as abstract ODEs on proper function spaces, see Reference [112] for more details.

We are interested in particular in the case of *stiff* systems of ODEs [105]. These are characterized by the fact that the Jacobian of system (1) has eigenvalues with large negative real parts, and it is well-known that explicit methods perform poorly compared to implicit ones due to the lack of favorable stability properties. A prominent and effective alternative way to numerically integrate in time stiff equations is to employ explicit *exponential integrators*, see Reference [112] for a seminal review. In few words, these schemes are based on the representation of the exact solution of system (1) in terms of the variation-of-constants formula

$$\boldsymbol{u}(t) = \mathrm{e}^{tM}\boldsymbol{u}_0 + \int_0^t \mathrm{e}^{(t-s)M}\boldsymbol{g}(s, \boldsymbol{u}(s))ds. \tag{2}$$

Then, suitable approximations are performed in order to obtain an explicit time marching scheme. For instance, probably the most famous exponential integrator is the exponential version of the (forward) Euler scheme, which is obtained by considering formula (2) in the time interval $[t_n, t_{n+1}]$ of length τ, i.e.,

$$\boldsymbol{u}(t_{n+1}) = \mathrm{e}^{\tau M}\boldsymbol{u}(t_n) + \int_0^\tau \mathrm{e}^{(\tau-s)M}\boldsymbol{g}(t_n + s, \boldsymbol{u}(t_n + s))ds,$$

and by approximating the nonlinear function $\boldsymbol{g}(t_n + s, \boldsymbol{u}(t_n + s))$ in the interval $[0, \tau]$ as $\boldsymbol{g}(t_n, \boldsymbol{u}(t_n))$. Then, by writing $\boldsymbol{u}(t_{n+1}) \approx \boldsymbol{u}_{n+1}$ and $\boldsymbol{u}(t_n) \approx \boldsymbol{u}_n$, after simple calculations we obtain the explicit time marching scheme

$$\boldsymbol{u}_{n+1} = \mathrm{e}^{\tau M}\boldsymbol{u}_n + \tau\varphi_1(\tau M)\boldsymbol{g}(t_n, \boldsymbol{u}_n) = \boldsymbol{u}_n + \tau\varphi_1(\tau M)\boldsymbol{f}(t_n, \boldsymbol{u}_n),$$

which is the just mentioned *exponential Euler* method. In the notation, we employed the exponential-like matrix function

$$\varphi_1(X) = \int_0^1 \mathrm{e}^{(1-\theta)X}d\theta, \quad X \in \mathbb{C}^{N\times N}.$$

The exponential Euler method is an explicit first order A-stable scheme, and hence well-suited for stiff equations (see Reference [111] for a complete and formal analysis of the scheme). More in general,

exponential integrators, as opposed to implicit methods, do not require the solution of (non)linear systems but rather the action of the exponential on a vector and/or of (linear combinations of) the so-called φ-functions, defined for a generic matrix $X \in \mathbb{C}^{N\times N}$ as

$$\varphi_\ell(X) = \int_0^1 \frac{\theta^{\ell-1}}{(\ell-1)!} \mathrm{e}^{(1-\theta)X} d\theta, \quad \ell \geq 1. \tag{3}$$

It is clear that the effective employment *in practice* of this kind of schemes requires *efficient tools* to approximate actions of the matrix exponential and of the just introduced exponential-like functions. The aim of this PhD thesis is precisely to contribute to the numerical analysis area in this direction. More in detail, the contribution can be divided into two parts:

I. Efficiently solving problems with a special characteristic, i.e., that have an underlying Kronecker/tensor structure;

II. Applying exponential integrators to numerically solve important equations in various fields.

Regarding the first one, we developped an integrator of exponential type for differential equations which possess d-dimensional Kronecker sum structure. This means that system (1) is actually given in the form

$$\boldsymbol{u}'(t) = K\boldsymbol{u}(t) + \boldsymbol{g}(t, \boldsymbol{u}(t)), \quad \boldsymbol{u}(0) = \boldsymbol{u}_0, \tag{4}$$

where

$$K = \sum_{\mu=1}^{d} A_{\otimes\mu} \tag{5}$$

and

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

Here, A_μ denotes an arbitrary $n_\mu \times n_\mu$ matrix and I_μ is the identity matrix of size n_μ, with $1 \leq \mu \leq d$. This kind of systems arises, for instance, when semidiscretizing in space evolutionary PDEs defined on domains which are the Cartesian product of d intervals. Typical examples are semilinear diffusion–reaction and Schrödinger equations, with linear operators Δ and $\mathrm{i}\Delta$, respectively. The task has been achieved by exploiting tensor algebra operations and techniques (the μ-mode product and the Tucker operator, in particular), which led to the development of the so-called *μ-mode integrator*. The approach has been extensively tested with many numerical examples, and it also scales very well, in terms of computational time, on Graphic Processing Units (GPUs). A comprehensive explanation with all the details is addressed in Chapter 1.

The d-dimensional Kronecker structure is actually not only present in the context of differential equations of the form (4). Indeed, many tasks in numerical analysis possess a similar tensor product structure. We mention, among the others, multidimensional interpolation, multidimensional function approximation using pseudospectral expansions and preconditioning of linear systems. The μ-mode based techniques mentioned before can still be employed, with suitable modifications, in order to effectively solve such tasks. This idea led to the development of the package KronPACK, which is a collection of MathWorks MATLAB®/GNU Octave functions which perform the needed tensor operations by means of the highly efficient Basic Linear Algebra Subprograms (BLAS). The approach, as well as the package, have been extensively tested on many multidimensional numerical tasks. A thorough explanation can be found in Chapter 2.

Then, as mentioned before, exponential integrators may require not only the action of the matrix exponential, but also of the exponential-like functions (3). In Chapter 3 we present an effective technique to compute actions of φ-functions when the matrix has d-dimensional Kronecker sum structure (5). The proposed approach is based on a suitable quadrature rule in combination with a modified scaling and squaring technique, and it is shown to be more efficient than state-of-the-art techniques to accomplish the same task.

When dealing with exponential integrators up to second order applied to equation (4), it is actually possible to pursue an alternative approach to compute the needed actions of φ-functions, which is more

practical and easier to implement rather than the one explained in Chapter 3. Indeed, in Chapter 4 we present a technique based on a direction splitting of the involved matrix functions, which still lets us exploit the highly efficient level 3 BLAS for the actual computation of the required actions in a μ-mode fashion.

Concerning the second contribution, i.e., the employment of exponential integrators in applications, we present in Chapter 5 an example of usage of these schemes for plasma physics problems, in the context of a fairly recent technique to effectively handle high-dimensional scenarios. In particular, we propose a newly-designed second order projector-splitting dynamical low rank integrator for the full six-dimensional Vlasov–Poisson equations

$$\begin{cases} \partial_t f(t,x,v) + v \cdot \nabla_x f(t,x,v) - E(f)(t,x) \cdot \nabla_v f(t,x,v) = 0, \\ E(f)(t,x) = -\nabla_x \phi(t,x), \\ -\Delta\phi(t,x) = \rho(f)(t,x) + 1, \quad \rho(f)(t,x) = -\displaystyle\int_{\Omega_v} f(t,x,v)\, dv, \end{cases} \tag{6}$$

where $f(t,x,v)$ represents the particle-density function of the species under consideration, $t \in \mathbb{R}_0^+$ is the time variable, $x \in \Omega_x \subset \mathbb{R}^d$ refers to the space variable, $v \in \Omega_v \subset \mathbb{R}^d$ is the velocity variable and $d = 1, 2, 3$. Depending on the specific physical phenomenon under study, system (6) is completed with appropriate boundary and initial conditions. The scheme is based on an exponential integrator in combination with a Fourier spectral discretization, which leads to a numerical method which is free of a Courant–Friedrichs–Lewy (CFL) condition but still fully explicit. The implementation has been performed with the aid of `Ensign`, a software framework (presented in Chapter 5 as well) which facilitates the efficient implementation of dynamical low-rank algorithms on modern multi-core Central Processing Unit (CPU) as well as GPU based systems. In particular, we show numerical results of 6D simulations run on a single suitably equipped workstation, which highlight the significant speedup that can be obtained using GPUs in this context.

A different application is presented in Chapter 6, in which we illustrate the usage of exponential integrators for mean-field optimal control problems. These kinds of models are very popular nowadays among practitioners: indeed, we have mean-field based models for a huge variety of phenomena, like coordinated animal motion, flock herding, opinion formation, pedestrian dynamics and cooperative robots, among the others. In particular, we consider a mean-field optimal control problem with selective action of the control, where the constraint is a continuity equation involving a non-local term and diffusion. To determine the optimal control, we formally derive the first order optimality conditions from the Lagrangian function and obtain a system of coupled PDEs, which are then integrated numerically, in the context of a steepest descent algorithm, with exponential integrators. The simulations performed match satisfactorily with the results already available in the literature, and highlight the effectiveness of the approach.

Finally, in Chapter 7 we present a technique to efficiently integrate in time inhomogeneous evolutionary advection–diffusion–reaction equations, typically arising in computational chemistry, with exponential integrators. The approach is based on the extraction of a constant coefficient diffusion part from the original PDE, whose magnitude is determined by a linear stability analysis of the chosen temporal scheme. The resulting equation can then be numerically solved more efficiently than the initial one, by employing for example Fast Fourier Transform (FFT)-based or tensor μ-mode-based techniques. Also, we present there two new exponential integrators of Lawson type (of first and second order), which appear to have better unconditional stability bounds compared to other well-known exponential integrators.

Overall, the thesis is structured as follows. In Part I, constituted by Chapters 1–4, we collect all the works related to Kronecker/tensor structured problems. The applications of exponential integrators to important differential equations are presented in Chapters 5–7, encompassed in Part II. Moreover, each chapter is self-consistent, so that it is possible to examine each one on its own. Nevertheless, for readability and exposition reasons, the enumeration of the formulas and of the sections is global to the thesis. For similar reasons, we generated a single common bibliography for the whole thesis.

Finally, the complete list of the author's publications and preprints/ongoing works is given in the following.

Publications

1. M. Caliari, F. C. and F. Zivcovich. BAMPHI: Matrix and transpose free action of the combinations of φ-functions from exponential integrators. *J. Comput. Appl. Math.*, 423:114973, 2023.

2. M. Caliari, F. C. and F. Zivcovich. A μ-mode BLAS approach for multidimensional tensor-structured problems. *Numer. Algorithms*, 2022. Published online: 04 October 2022.

3. F. C. and L. Einkemmer. Efficient 6D Vlasov simulation using the dynamical low-rank framework `Ensign`. *Comput. Phys. Commun.*, 280:108489, 2022.

4. M. Caliari, F. C., L. Einkemmer, A. Ostermann and F. Zivcovich. A μ-mode integrator for solving evolution equations in Kronecker form. *J. Comput. Phys.*, 455:110989, 2022.

5. M. Caliari, F. C. and F. Zivcovich. Approximation of the matrix exponential for matrices with a skinny field of values. *BIT Numer. Math.*, 60(4):1113–1131, 2020.

Preprints

1. M. Caliari, F. C. and F. Zivcovich. A μ-mode approach for exponential integrators: actions of φ-functions of Kronecker sums. *arXiv preprint arXiv:2210.07667*, 2022.

2. G. Albi, M. Caliari, E. Calzola and F. C.. Exponential integrators for mean-field selective optimal control problems. *arXiv preprint arXiv:2302.00127*, 2023.

Ongoing works

1. M. Caliari, and F. C.. A μ-mode based direction splitting of φ-functions for exponential integrators. *In preparation*, 2023.

2. M. Caliari, F. C., L. Einkemmer and A. Ostermann. Efficient exponential integration of inhomogeneous evolutionary advection–diffusion–reaction equations. *In preparation*, 2023.

Part I

Tensor structured problems

Chapter 1

The μ-mode integrator and exponential-like schemes

In this chapter, we present 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.

The material of this chapter is taken from Reference [39], i.e., M. Caliari, F. C., L. Einkemmer, A. Ostermann and F. Zivcovich. A μ-mode integrator for solving evolution equations in Kronecker form. *J. Comput. Phys.*, 455:110989, 2022.

1.1 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 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 [112] or splitting methods [138].

Despite the significant advances made in constructing more efficient numerical algorithms, efficiently computing the action of large matrix functions remains a significant challenge. Here, 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 work, 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 1.2 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., [96, 109, 145, 153]). 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., [14]).

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 remaining part of this chapter is structured as follows. In Section 1.2 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 1.3, 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 1.4. The implementation on modern computer architectures, which includes performance results for multi-core CPU and GPU based systems, will be discussed in Section 1.5. Finally, in Section 1.6 we draw some conclusions.

1.2 The μ-mode integrator for evolution equations in Kronecker form

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

$$\begin{aligned} \partial_t u(t,\boldsymbol{x}) &= \Delta u(t,\boldsymbol{x}) = \left(\partial_1^2+\partial_2^2\right) u(t,\boldsymbol{x}), \qquad \boldsymbol{x}\in\Omega\subset\mathbb{R}^2, \quad t\geq 0,\\ u(0,\boldsymbol{x}) &= u_0(\boldsymbol{x}), \end{aligned} \tag{1.1}$$

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

$$u(t,\cdot) = \mathrm{e}^{t\Delta}u_0 = \mathrm{e}^{t\partial_1^2}\mathrm{e}^{t\partial_2^2}u_0 = \mathrm{e}^{t\partial_2^2}\mathrm{e}^{t\partial_1^2}u_0, \tag{1.2}$$

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

Discretizing (1.1) by finite differences on a Cartesian grid with $n_1 \times n_2$ grid points results in the linear differential equation

$$\boldsymbol{u}'(t) = (I_2 \otimes A_1 + A_2 \otimes I_1)\,\boldsymbol{u}(t), \quad \boldsymbol{u}(0) = \boldsymbol{u}_0, \tag{1.3}$$

for the unknown vector $\boldsymbol{u}(t)$. Here, A_1 is a (one-dimensional) stencil matrix for ∂_1^2 on the grid points $x_1^{i_1}$, $1 \le i_1 \le n_1$, and A_2 is a (one-dimensional) stencil matrix for ∂_2^2 on the grid points $x_2^{i_2}$, $1 \le i_2 \le n_2$. 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 (1.3) is given by

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

which is the discrete analog of (1.2).

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 $\boldsymbol{U}(t)$ be the order two tensor of size $n_1 \times n_2$ (in fact, a matrix) whose stacked columns form the vector $\boldsymbol{u}(t)$. The indices of this matrix reflect the structure of the grid. In particular

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

Using this tensor notation, problem (1.3) takes the form

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

and its solution can be expressed as

$$\boldsymbol{U}(t) = \mathrm{e}^{tA_1}\boldsymbol{U}_0\,\mathrm{e}^{tA_2^{\mathsf{T}}}, \tag{1.4}$$

see [147]. From this representation, it is clear that $\boldsymbol{U}(t)$ can be computed as the action of the small matrices e^{tA_1} and e^{tA_2} on the tensor $\boldsymbol{U}_0$. More precisely, the matrices e^{tA_1} and e^{tA_2} act on the first and second indices, respectively. The computation of (1.4) can thus be performed by the simple algorithm

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

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

$$\boldsymbol{u}'(t) = M\boldsymbol{u}(t), \quad \boldsymbol{u}(0) = \boldsymbol{u}_0, \tag{1.5}$$

where

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

and

$$A_{\otimes\mu} = I_d \otimes \cdots \otimes I_{\mu+1} \otimes A_\mu \otimes I_{\mu-1} \otimes \cdots \otimes I_1. \tag{1.6}$$

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 (1.6) 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

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

or the Schrödinger equation with potential in Kronecker form

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

Condition (1.6) is fulfilled also for some problems with non-constant coefficient differential operators, see Section 1.4.2 for an example. We will consider these and other equations later to perform numerical examples.

Equation (1.5) is what we call a linear problem in *Kronecker form*, and its solution is obviously given by

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

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

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

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

$$\begin{aligned}
\boldsymbol{U}^{(0)} &= \boldsymbol{U}_0, \\
\boldsymbol{U}^{(1)}(\cdot, i_2, \ldots, i_d) &= \mathrm{e}^{tA_1}\boldsymbol{U}^{(0)}(\cdot, i_2, \ldots, i_d), && 1 \leq i_\mu \leq n_\mu, \quad 2 \leq \mu \leq d, \\
&\cdots \\
\boldsymbol{U}^{(d)}(i_1, \ldots, i_{d-1}, \cdot) &= \mathrm{e}^{tA_d}\boldsymbol{U}^{(d-1)}(i_1, \ldots, i_{d-1}, \cdot), && 1 \leq i_\mu \leq n_\mu, \quad 1 \leq \mu \leq d-1, \\
\boldsymbol{U}(t) &= \boldsymbol{U}^{(d)}.
\end{aligned} \tag{1.7}$$

We remark that scheme (1.7) 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 (1.7), 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 [61, 62], 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 [89]. Using integrator (1.7) 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., [83, 86, 113]). In this case scheme (1.7) would be used to efficiently solve the flow corresponding to the linear diffusion. We further note that a related approach was pursued by [148] in order to produce schemes that solve two- and three-dimensional biological models.

Implementing integrator (1.7) requires the computation of d small exponentials of sizes $n_1 \times n_1$, $\ldots$, $n_d \times n_d$, respectively. If a marching scheme with *constant* time step size is applied to (1.5), then these matrices can be precomputed once and for all, and their storage cost is negligible compared to that required by the solution $\boldsymbol{U}(t)$. 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 1.4.1). 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 (1.5) also by directly computing the vector $\mathrm{e}^{tM}\boldsymbol{u}_0$. 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, [97, 149]), Taylor series [6], or polynomial interpolation (see, for instance, [40, 44, 45]).

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 (1.5). 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 [168, 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 1.5).

Summarizing, the μ-mode integrator has the following advantages:

- For a heat equation the proposed scheme only requires $\mathcal{O}(N)$ memory operations, compared to an implicit integrator 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 1.4.3 and 1.4.4).

Equation (1.7) gives perhaps the most intuitive picture of the proposed approach. However, we can also formulate this problem in terms of μ-fibers. Indeed, let $\boldsymbol{U} \in \mathbb{C}^{n_1 \times \cdots \times n_d}$ be an order d tensor. A *μ-fiber* of $\boldsymbol{U}$ is a vector in $\mathbb{C}^{n_\mu}$ obtained by fixing every index of the tensor but the μth. In these terms, $\boldsymbol{U}^{(\mu-1)}(i_1, \ldots, i_{\mu-1}, \cdot, i_{\mu+1}, \ldots, i_d)$ is a μ-fiber of the tensor $\boldsymbol{U}^{(\mu-1)}$, and every line in formula (1.7) corresponds to the action of the matrix e^{tA_μ} on the μ-fibers of $\boldsymbol{U}^{(\mu-1)}$. By means of μ-fibers, it is possible to define the following operation.

Definition 1.2.1. *Let $L \in \mathbb{C}^{m \times n_\mu}$ be a matrix. Then the* μ-mode product[1] *of L with $\boldsymbol{U}$, denoted by $\boldsymbol{S} = \boldsymbol{U} \times_\mu L$, is the tensor $\boldsymbol{S} \in \mathbb{C}^{n_1 \times \cdots \times n_{\mu-1} \times m \times n_{\mu+1} \times \cdots \times n_d}$ obtained by multiplying the matrix L onto the μ-fibers of $\boldsymbol{U}$, that is*

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

According to this definition, it is clear that in formula (1.7) we are performing d consecutive μ-mode products with the matrices e^{tA_μ}, $1 \leq \mu \leq d$. We can therefore write scheme (1.7) as follows

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

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 [119]), 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 [120].

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

1.3 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 [21, 179]), 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 $\boldsymbol{x} = (x_1, \ldots, x_d)$ belonging to $\mathbb{R}^d$. We define the d-variate functions $\mathcal{H}_{\mathbf{i}}(\boldsymbol{x})$ as

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

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

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

We recall that Hermite functions satisfy

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

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

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

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

If a d-variate function f can be expanded into a series

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

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

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

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 $W_{\ell_\mu}^{\mu}$, $0 \leq \ell_\mu \leq m_\mu$, and fix to k_μ the number of uni-variate functions $\phi_{i_\mu}^{\mu}(x_\mu)$ to be considered. Then we have

$$\hat{f}_{\mathbf{i}} = \sum_{\boldsymbol{\ell}<\mathbf{m}} f(\boldsymbol{x}_{\boldsymbol{\ell}})\overline{\phi_{\mathbf{i}}}(\boldsymbol{x}_{\boldsymbol{\ell}})w_{\boldsymbol{\ell}}, \qquad \mathbf{i} < \mathbf{k}, \tag{1.8}$$

where $\boldsymbol{x}_{\boldsymbol{\ell}} = (X_{\ell_1}^1, \ldots, X_{\ell_d}^d) \in \mathbb{R}^d$, $w_{\boldsymbol{\ell}} = \prod_{\mu=1}^{d} W_{\ell_\mu}^{\mu}$ 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

$$\hat{f}(\boldsymbol{x}) = \sum_{\mathbf{i}<\mathbf{k}} \hat{f}_{\mathbf{i}}\phi_{\mathbf{i}}(\boldsymbol{x}) \approx f(\boldsymbol{x}) \tag{1.9}$$

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 $\boldsymbol{F_W} \in \mathbb{C}^{m_1\times\cdots\times m_d}$ the tensor with elements $f(\boldsymbol{x_\ell})w_{\boldsymbol{\ell}}$ and by $\hat{\boldsymbol{F}} \in \mathbb{C}^{k_1\times\cdots\times k_d}$ the tensor with elements $\hat{f}_{\mathbf{i}}$. Then, in terms of the Tucker operator, we can write equation (1.8) as follows

$$\hat{\boldsymbol{F}} = \boldsymbol{F_W} \times_1 \Phi_1 \times_2 \cdots \times_d \Phi_d. \tag{1.10}$$

It is then possible to evaluate the function $\hat{f}(\boldsymbol{x})$ in (1.9) at a Cartesian grid $\boldsymbol{y_p} = (Y_{p_1}^1, \ldots, Y_{p_d}^d)$, that is

$$\hat{f}(\boldsymbol{y_p}) = \sum_{\mathbf{i}<\mathbf{k}} \hat{f}_{\mathbf{i}}\phi_{\mathbf{i}}(\boldsymbol{y_p}), \qquad \mathbf{p} < \mathbf{q}, \tag{1.11}$$

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

$$\hat{\hat{\boldsymbol{F}}} = \hat{\boldsymbol{F}} \times_1 \Psi_1 \times_2 \cdots \times_d \Psi_d \tag{1.12}$$

is the tensor formulation of formula (1.11).

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

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

with $\mathbf{m} = \mathbf{k}$, that is, the orthonormality relation among the $\phi_{\mathbf{i}}$ functions is true also at discrete level. This is the case, for instance, when using Gauss–Hermite quadrature nodes for $\phi_{\mathbf{i}}(\boldsymbol{x}) = \mathcal{H}_{\mathbf{i}}(\boldsymbol{x})$. Then, the matrices $\Phi_\mu \in \mathbb{C}^{m_\mu\times m_\mu}$ turn out to be square and formula (1.10) 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 (1.12) 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 the experiments (see Sections 1.4.3 and 1.4.4). Hence, we will use (1.10) and (1.12) for the required spectral transforms.

We also remark that a similar approach was pursued in [106] in the framework of three-dimensional Chebyshev interpolation.

1.4 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 1.4.1, 1.4.2 and 1.4.5, 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 [6] which is based on a Taylor expansion of the exponential;
- `phipm`: a full Krylov method presented in [149];
- `kiops`: a Krylov method based on an incomplete orthogonalization process, described in [97].

Their implementations are publicly available as MATLAB functions. Although the underlying algorithms just 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 pass the explicit matrix to all the MATLAB functions. 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 1.4.1). 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 1.2, 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 [5]. In this context, another method which could be directly used in MATLAB is `exptayotf` from [46]. 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 [172]. We will demonstrate that the MATLAB implementation of the proposed μ-mode integrator outperforms all the iterative schemes by at least a factor of 7.

Concerning the experiments in Sections 1.4.3 and 1.4.4, we compare the 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 [93]. 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 MathWorks MATLAB® R2020b.

1.4.1 Code validation

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

$$\begin{cases} \partial_t u(t,\boldsymbol{x}) = \Delta u(t,\boldsymbol{x}), & \boldsymbol{x} \in [0,2\pi)^3, \quad t \in [0,T], \\ u(0,\boldsymbol{x}) = \cos x_1 + \cos x_2 + \cos x_3, & \end{cases} \tag{1.13}$$

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

$$\boldsymbol{u}'(t) = M\boldsymbol{u}(t), \tag{1.14}$$

where $\boldsymbol{u}$ denotes the vector in which the degrees of freedom are assembled. The exact solution of equation (1.14) is given by the action of the matrix exponential

$$\boldsymbol{u}(t) = \mathrm{e}^{tM}\boldsymbol{u}(0). \tag{1.15}$$

The matrix M has the following Kronecker structure

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

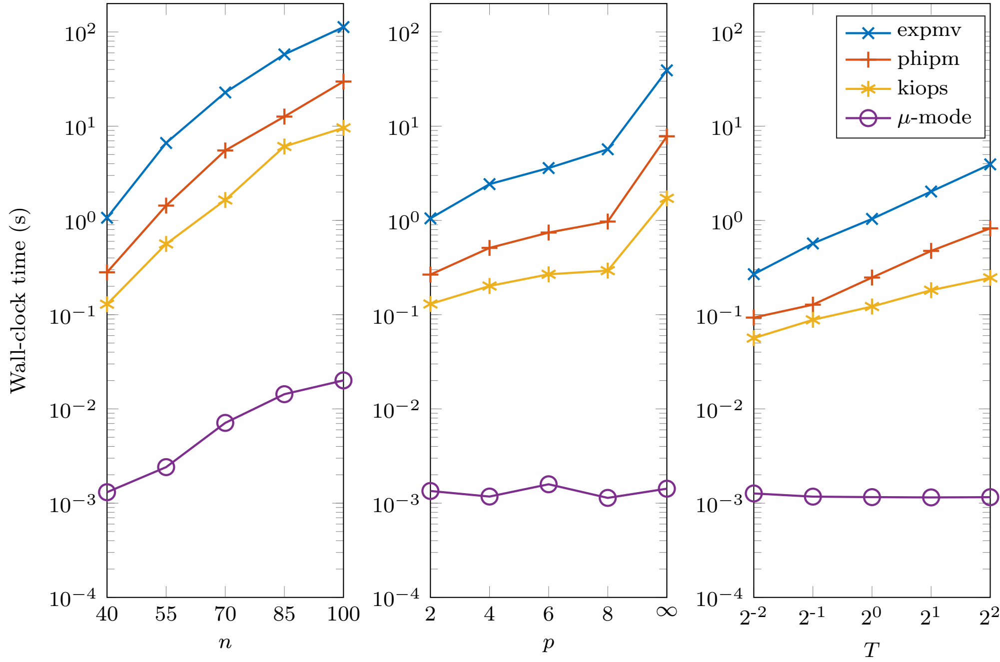

Figure 1.1: The wall-clock time for solving the heat equation (1.13) is shown as a function of n (left), of the order of the finite difference scheme p (middle), and of the final time T (right). Note that $p = \infty$ corresponds to a spectral space discretization.

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 $\boldsymbol{u}(t)$ can be seen as vectorization of the tensor $\boldsymbol{U}(t)$, and we can write (1.15) in tensor form as

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

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

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 TM.

The corresponding results are shown in Figure 1.1. 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 (1.15) 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

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 1.1: Breakdown of wall-clock time (in milliseconds) for the μ-mode integrator for different values of n (cf. left plot of Figure 1.1).

the μ-mode products and not by the exponentiation of the matrices A_μ (see Table 1.1). 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 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.

The second test shows that the iterative schemes have 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.

1.4.2 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 angular variable in the circular cross section, see for example [178]) and a prescribed length-dependent flow velocity. In this case we obtain the following diffusion-advection equation for the concentration c

$$\partial_t c(t,\rho,z) = \alpha\left(\partial_{\rho\rho}c(t,\rho,z) + \frac{1}{\rho}\partial_\rho c(t,\rho,z) + \partial_{zz}c(t,\rho,z)\right) - s(z)\partial_z c(t,\rho,z), \tag{1.16}$$

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 (1.6). 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

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

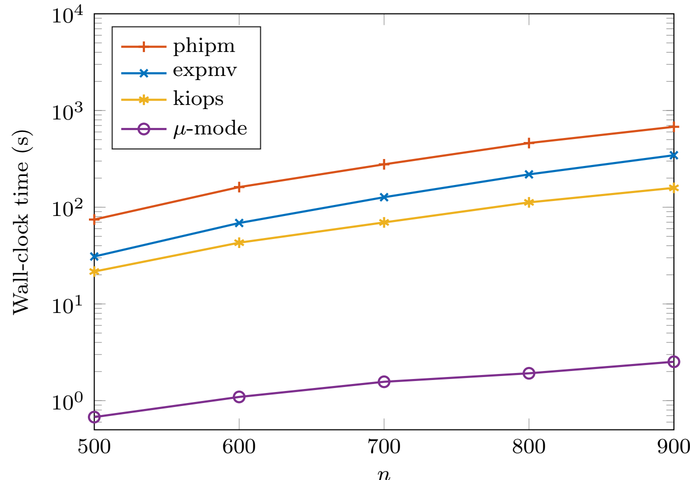

Figure 1.2: Wall-clock time (in seconds) for the integration of (1.16) up to $T = 4$ as a function of n (total number of degrees of freedom $N = n^2$).

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 1.2. 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.

1.4.3 Schrödinger equation with time-independent potential

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

$$\begin{cases} \mathrm{i}\partial_t\psi(t,\boldsymbol{x}) = -\dfrac{1}{2}\Delta\psi(t,\boldsymbol{x}) + V(\boldsymbol{x})\psi(t,\boldsymbol{x}), & \boldsymbol{x}\in\mathbb{R}^3, \quad t\in[0,1] \\ \psi(0,\boldsymbol{x}) = \psi_0(\boldsymbol{x}), \end{cases} \tag{1.17}$$

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

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

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 [115].

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

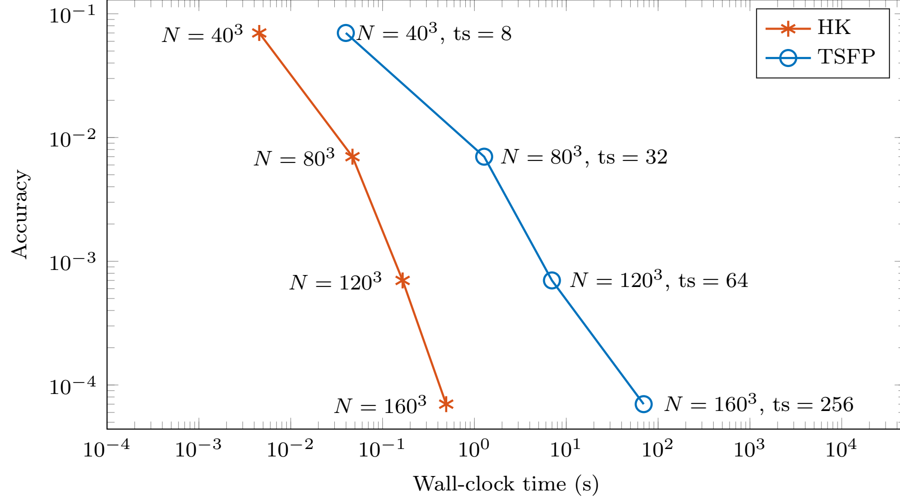

Figure 1.3: Precision diagram for the integration of the Schrödinger equation with a time-independent potential (1.17) up to $T = 1$. The number of degrees of freedom N and the number of time steps (ts) are varied in order to achieve a result which is accurate up to the given tolerance. The reference solution has been computed by the HKP method with $N = 300^3$.

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 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 1.3. 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.

1.4.4 Schrödinger equation with time-dependent potential

Let us now consider the Schrödinger equation

$$\begin{cases} \partial_t \psi(t, \boldsymbol{x}) = H(t, \boldsymbol{x})\psi(t, \boldsymbol{x}), \qquad \boldsymbol{x} \in \mathbb{R}^3, \quad t \in [0, 1] \\ \psi(0, \boldsymbol{x}) = 2^{-\frac{5}{2}}\pi^{-\frac{3}{4}}(x_1 + \mathrm{i}x_2)\exp\left(-x_1^2/4 - x_2^2/4 - x_3^2/4\right), \end{cases} \tag{1.18}$$

where the Hamiltonian is given by

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

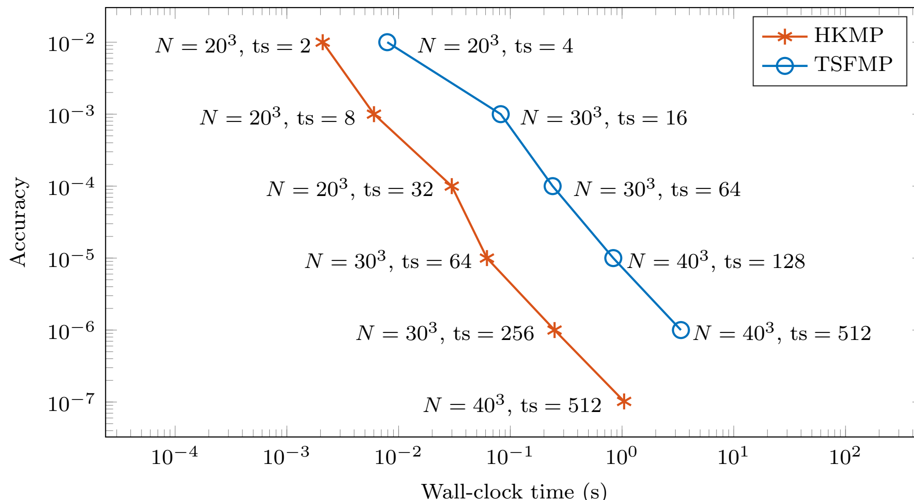

Figure 1.4: Precision diagram for the integration of the Schrödinger equation with a time-dependent potential (1.18) up to $T = 1$. The number of degrees of freedom N and the number of time steps (ts) are varied in order to achieve a result which is accurate up to the given tolerance. The reference solution has been computed by the HKMP method with $N = 100^3$ and ts = 2048.

Note that the potential is now time-dependent, as opposed to the case presented in Section 1.4.3. Such potentials commonly occur in applications, e.g., when studying laser-atom interactions (see, for example, [157]).

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

$$\boldsymbol{u}'(t) = A(t)\boldsymbol{u}(t)$$

be the considered ODE with time-dependent coefficients, and let $\boldsymbol{u}_n$ be the numerical approximation to the solution at time t_n. Then, the exponential midpoint rule provides the numerical solution

$$\boldsymbol{u}_{n+1} = \exp\bigl(\tau_n A(t_n + \tau_n/2)\bigr)\boldsymbol{u}_n \tag{1.19}$$

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 1.4.3, the resulting ODE cannot be integrated exactly in time. For the time discretization, we will then use the order two Magnus integrator (1.19). We call the resulting scheme Hermite Kronecker Magnus Pseudospectral method (HKMP).

The results of the experiments are depicted in Figure 1.4. 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 flow 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 one is.

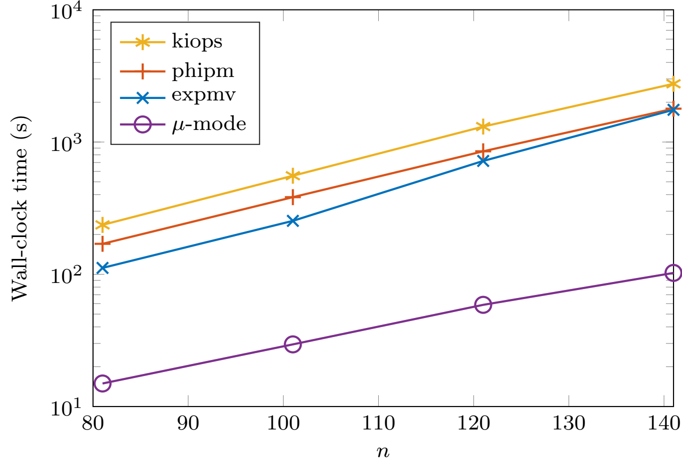

Figure 1.5: Wall-clock time (in seconds) for the integration of (1.20) up to $T = 25$ as a function of n (total number of degrees of freedom $N = n^3$). A constant time step size $\tau = 0.1$ is employed.

1.4.5 Nonlinear Schrödinger/Gross–Pitaevskii equation

In this section we consider the nonlinear Schrödinger equation

$$\partial_t \psi(t, \boldsymbol{x}) = \frac{\mathrm{i}}{2} \Delta \psi(t, \boldsymbol{x}) + \frac{\mathrm{i}}{2} \Big(1 - |\psi(t, \boldsymbol{x})|^2\Big) \psi(t, \boldsymbol{x}), \tag{1.20}$$

which is also known as Gross–Pitaevskii equation. The unknown ψ represents the wave function, $\boldsymbol{x} \in \mathbb{R}^3$, $t \in [0, 25]$, 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 [47] 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 to solve (1.20) numerically, we consider the Time Splitting Finite Difference method proposed in [47]. More specifically, we truncate the unbounded domain to $\boldsymbol{x} \in [-20, 20]^3$ 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 1.4. The nonlinear flow is integrated exactly.

The results of the experiment are presented in Figure 1.5. The μ-mode integrator outperforms `expmv` by approximately a factor of 7. The speedup compared to both `phipm` and `kiops` is even larger.

1.5 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 [16]. In particular, for GPU computing much research effort has been undertaken to obtain efficient implementations (see, e.g., [17, 35, 74, 75, 78, 135, 139, 163, 169, 188]).

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 1.2 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 CPUs 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 the 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, 136]).

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 the implementation (for more details on these routines we refer to [54]). The 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.

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 1.2: Wall-clock time for the heat equation (1.13) 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 milliseconds). 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.

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

1.5.1 Heat equation

We consider the same problem as in Section 1.4.1, Test 1. The wall-clock time for computing the matrix exponentials and a single time step of the proposed algorithm is listed in Table 1.2.

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.

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

$$\begin{cases} \partial_t u(t,\boldsymbol{x}) = \Delta u(t,\boldsymbol{x}), & \boldsymbol{x} \in \left[-\frac{11}{4}, \frac{11}{4}\right]^3, \quad t \in [0,1], \\ u(0,\boldsymbol{x}) = \left(x_1^4 + x_2^4 + x_3^4\right) \exp\left(-x_1^4 - x_2^4 - x_3^4\right), \end{cases} \tag{1.21}$$

with (artificial) Dirichlet boundary conditions, discretized in space as above. From Table 1.3 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 the 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 1.3, 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 the 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.

1.5.2 Schrödinger equation with time-independent potential

We consider the Schrödinger equation with time-independent potential from Section 1.4.3. 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

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 1.3: Wall-clock time for the heat equation (1.21) discretized using second-order centered finite differences with 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 milliseconds).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 1.4: Wall-clock time for the linear Schrödinger equation with time-independent potential (1.17) 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 milliseconds). The speedup is the ratio between the single step performed in CPU and GPU, in double and single precision.

for both the CPU and GPU implementation are listed in Table 1.4. The GPU implementation, for both single and double precision, shows a speedup of approximately 15 compared to the CPU implementation.

1.5.3 Schrödinger equation with time-dependent potential

We consider once again the Schrödinger equation with the time-dependent potential from Section 1.4.4 solved with the HKMP method. The equation is integrated up to $T = 1$ with time step size $\tau = 0.02$. The results are given in Table 1.5. 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 from the CPU to the GPU (for both single and double precision).

1.6 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 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.

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 1.5: Wall-clock time for the Schrödinger equation with time-dependent potential (1.18) 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 milliseconds). 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).

Chapter 2

KronPACK: a μ-mode approach for tensor structured problems

In this chapter, 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, 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.

The material of this chapter is taken from Reference [42], i.e., M. Caliari, F. C., and F. Zivcovich. A μ-mode BLAS approach for multidimensional tensor-structured problems. *Numer. Algorithms*, 2022. Published online: 04 October 2022.

2.1 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

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

where the values t_j are linearly combined to obtain the values s_i (i.e., $\boldsymbol{s} = L\boldsymbol{t}$, 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

$$s_{i_1 i_2} = \sum_{j_2=1}^{m_2} \sum_{j_1=1}^{m_1} t_{j_1 j_2} \ell^1_{i_1 j_1} \ell^2_{i_2 j_2}, \quad 1 \leq i_1 \leq n_1, \quad 1 \leq i_2 \leq n_2. \tag{2.1}$$

The meaning of the involved scalar quantities depends on the specific example under consideration. In any case, a straightforward implementation of formula (2.1) 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 (2.1) can be written equivalently in matrix formulation as

$$\boldsymbol{S} = L_1 \boldsymbol{T} L_2^{\mathsf{T}}, \tag{2.2}$$

	$n=50$	$n=100$	$n=200$	$n=400$
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

Table 2.1: Wall-clock time (in seconds) for the computation of the values $s_{i_1 i_2}$ in formula (2.1) 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.

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 (2.2) 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 (2.2) is to exploit optimized Basic Linear Algebra Subprograms (BLAS) [58, 69, 114, 189], 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 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 2.1. As expected, for all the values of n under study, the most efficient way to compute the elements of $\boldsymbol{S}$ is realizing formula (2.2) 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 work, 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

$$\mathrm{vec}(\boldsymbol{S}) = (L_2 \otimes L_1)\mathrm{vec}(\boldsymbol{T}), \tag{2.3}$$

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 the appendix for their formal definitions. A straightforward implementation of formula (2.3) 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 the appendix), we can see that formula (2.3) is equivalent to formula (2.2). 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 (2.2) 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 2.2, 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 [119, 120]). Then, we proceed in Section 2.3 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 2.4 the related numerical experiments and we finally draw the conclusions in Section 2.5.

All the functions and the scripts needed to perform the relevant tensor operations and to reproduce the numerical examples are contained in the 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`.

2.2 The μ-mode product and its applications

In order to generalize formula (2.2) to the d-dimensional case, we rely on some concepts from tensor algebra (see [119, 120] 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 2.2.1. *A* μ-fiber *of* $\boldsymbol{T}$ *is a vector in* $\mathbb{C}^{m_\mu}$ *obtained by fixing every index of the tensor but the* μ*th.*

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 2.2.2. *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 2.2.3. *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 2.2.2, 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

$$S^{(\mu)} = LT^{(\mu)}. \tag{2.4}$$

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 (2.4) 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 `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 (2.4) it appears clear that the computational cost of the μ-mode product, in terms of floating point operations, is $\mathcal{O}(nm_1 \cdots m_d)$.

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 2.2.4. *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*

$$\boldsymbol{S} = \boldsymbol{T} \times_1 L_1 \times_2 \cdots \times_d L_d. \tag{2.5}$$

We notice that the single element $s_{i_1 \ldots i_d}$ of $\boldsymbol{S}$ in formula (2.5) turns out to be

$$s_{i_1 \ldots i_d} = \sum_{j_d=1}^{m_d} \cdots \sum_{j_1=1}^{m_1} t_{j_1 \ldots j_d} \prod_{\mu=1}^{d} \ell^{\mu}_{i_\mu j_\mu}, \quad 1 \leq i_\mu \leq n_\mu, \tag{2.6}$$

provided that $\ell^{\mu}_{i_\mu j_\mu}$ are the elements of L_μ. Hence, as formula (2.6) is clearly the generalization of formula (2.1) to the d-dimensional setting, formula (2.5) is the sought d-dimensional generalization of formula (2.2). We also notice that the Tucker operator (2.5) is invariant with respect to the ordering of the μ-mode products, and that the implicit ordering given by Definition 2.2.4 is equivalent to performing the sums in formula (2.6) starting from the innermost.

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

Lemma 2.2.1. *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* (2.5) *are equivalently given by*

$$\mathrm{vec}(\boldsymbol{S}) = (L_d \otimes \cdots \otimes L_1)\mathrm{vec}(\boldsymbol{T}). \tag{2.7}$$

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 [120]. 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 the appendix) we have then

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

and finally, by definition of vec operator,

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

□

Notice that formula (2.7) is precisely the d-dimensional generalization of formula (2.3). Hence, tasks written as in formula (2.7) can be equivalently stated and computed more efficiently again by formula (2.5), without assembling the large-sized matrix $L_d \otimes \cdots \otimes L_1$.

We can then summarize as follows: the *element-wise* formulation (2.6), the *tensor* formulation (2.5) and the *vector* formulation (2.7) 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 the package KronPACK are based on formulation (2.5).

Remark 2.2.1. *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* (2.5)*, 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 [129, 137, 164, 174]). 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 2.2.5. *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

$$\boldsymbol{S} = \boldsymbol{T} \times_1 \mathcal{L}_1 \times_2 \cdots \times_d \mathcal{L}_d \tag{2.8}$$

and implemented in KronPACK in the function `tuckerfun`, follows straightforwardly. Clearly, in this case, some properties of the Tucker operator (2.5), 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 (2.8) is useful in some instances, see Remark 2.3.2 and Section 2.4.2 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* [18] — which does not have GNU Octave support, too — and *Tensorlab* [185], 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 the appendix for its definition) of the L_μ matrices to a vector $\boldsymbol{v}$, that is

$$(L_d \oplus \cdots \oplus L_1)\boldsymbol{v} = \operatorname{vec}\left(\sum_{\mu=1}^{d} (\boldsymbol{V} \times_\mu L_\mu)\right), \tag{2.9}$$

where $\boldsymbol{v} = \operatorname{vec}(\boldsymbol{V})$. In fact, as it can be noticed from formula (2.4), the identity matrix is the identity element of the μ-mode product. Combining this observation with Lemma 2.2.1, we easily obtain formula (2.9). In the package KronPACK, the matrix resulting from the Kronecker sum on the left hand side of equality (2.9) 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)`.

2.3 Problems formulation in d dimensions

In this section we discuss in more detail the problems that were briefly introduced in Section 2.1. Their generalization to arbitrary dimension d is addressed thanks to the common framework presented in Section 2.2.

2.3.1 Pseudospectral decomposition

Suppose that a function $f\colon R \to \mathbb{C}$, 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 nodes 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

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

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

$$\hat{\boldsymbol{f}} = \Psi \boldsymbol{f_w}.$$

In the two-dimensional case, the coefficients of a pseudospectral expansion in a tensor product basis (see, for instance, [30, Ch. 6.10]) are given by

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

which can be efficiently computed as

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

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 $\boldsymbol{F_W}$ 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 (2.5) and (2.6))

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

where Ψ_μ is the transform matrix with element $\overline{\phi_{i_\mu}^{\mu}(\xi_\mu^{k_\mu})}$ 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 $f(\xi_1^{k_1}, \ldots, \xi_d^{k_d})w_1^{k_1}\cdots w_d^{k_d}$, respectively. The corresponding pseudospectral approximation of $f(\boldsymbol{x})$ is

$$\hat{f}(\boldsymbol{x}) = \sum_{i_d=1}^{m_d}\cdots\sum_{i_1=1}^{m_1} \hat{f}_{i_1\ldots i_d}\phi_{i_1}^1(x_1)\cdots\phi_{i_d}^d(x_d), \tag{2.10}$$

where $\boldsymbol{x} = (x_1, \ldots, x_d)$. An application to Hermite–Laguerre–Fourier function decomposition is given in Section 2.4.2.

2.3.2 Function approximation

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

$$\tilde{f}(x) = \sum_{i=1}^{m} c_i \phi_i(x) \approx f(x), \tag{2.11}$$

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 (2.11) 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 (2.11) to the tensor product bivariate case is straightforward (see, for instance, [66, Ch. XVII]). Indeed, in this case the approximating function is given by

$$\tilde{f}(x_1, x_2) = \sum_{i_2=1}^{m_2} \sum_{i_1=1}^{m_1} c_{i_1 i_2} \phi^1_{i_1}(x_1) \phi^2_{i_2}(x_2) \approx f(x_1, x_2), \tag{2.12}$$

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 $(x_1^{\ell_1}, x_2^{\ell_2})$, with $1 \leq \ell_\mu \leq n_\mu$, the evaluation of approximation (2.12) 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 Φ_μ of size $n_\mu \times m_\mu$ with element $\phi^\mu_{i_\mu}(x_\mu^{\ell_\mu})$ 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

$$\tilde{f}(\boldsymbol{x}) = \sum_{i_d=1}^{m_d} \cdots \sum_{i_1=1}^{m_1} c_{i_1 \ldots i_d} \phi^1_{i_1}(x_1) \cdots \phi^d_{i_d}(x_d) \approx f(\boldsymbol{x}), \tag{2.13}$$

where $c_{i_1 \ldots i_d}$ represent scalar coefficients while $\phi^\mu_{i_\mu}(x_\mu)$ 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 (2.13) can be expressed in tensor formulation as

$$\tilde{\boldsymbol{F}} = \boldsymbol{C} \times_1 \Phi_1 \times_2 \cdots \times_d \Phi_d, \tag{2.14}$$

see formulas (2.5) and (2.6). Here we denote Φ_μ the matrix with element $\phi^\mu_{i_\mu}(x_\mu^{\ell_\mu})$ 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 2.4.3.

Remark 2.3.1. *Clearly, formula* (2.14) *can be employed to evaluate a pseudospectral approximation* (2.10) *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 [39]), 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* (2.14) *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 2.3.2. *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 [26] 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 2.2.5), yielding the generalization of the Tucker operator* (2.8)*. In practice, we can perform this task with the KronPACK function* `tuckerfun`*, see the numerical example in Section 2.4.2.*

2.3.3 Action of the matrix exponential

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

$$\begin{cases} \partial_t u(t,x) = \mathcal{A}u(t,x), \quad t>0, \quad x\in\Omega\subset\mathbb{R}, \\ \quad u(0,x) = u_0(x), \end{cases} \tag{2.15}$$

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 (2.15), by discretizing first the spatial variable, e.g., by finite differences or spectral differentiation, leads to the system of Ordinary Differential Equations (ODEs)

$$\begin{cases} \boldsymbol{u}'(t) = A\boldsymbol{u}(t), \quad t>0, \\ \boldsymbol{u}(0) = \boldsymbol{u}_0, \end{cases} \tag{2.16}$$

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 (2.16) 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 [5, 46]). 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 [6, 40, 97, 149].

Suppose now we want to solve instead

$$\begin{cases} \partial_t u(t,x_1,x_2) = \mathcal{A}u(t,x_1,x_2), \quad t>0, \quad (x_1,x_2)\in\Omega\subset\mathbb{R}^2, \\ \quad u(0,x_1,x_2) = u_0(x_1,x_2), \end{cases} \tag{2.17}$$

coupled again with suitable boundary conditions. If PDE (2.17) 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 [39] for more details and examples), then the method of lines yields the system of ODEs

$$\begin{cases} \boldsymbol{u}'(t) = (I_2\otimes A_1 + A_2\otimes I_1)\,\boldsymbol{u}(t), \quad t>0, \\ \boldsymbol{u}(0) = \boldsymbol{u}_0. \end{cases} \tag{2.18}$$

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 the appendix.

Clearly, the solution of system (2.18) is given by

$$\boldsymbol{u}(t) = \exp\left(t(I_2 \otimes A_1 + A_2 \otimes I_1)\right)\boldsymbol{u}_0, \tag{2.19}$$

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 the appendix), one could write everything in terms of the exponentials of the *small-sized* matrices A_μ. Indeed, we have

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

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 (2.18) 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 [147]) 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 (2.19) is

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

which can be written in more compact notation as

$$\boldsymbol{u}(t) = \exp\left(t\left(A_d \oplus \cdots \oplus A_1\right)\right)\boldsymbol{u}_0. \tag{2.20}$$

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 2.2.1, we have

$$\boldsymbol{U}(t) = \boldsymbol{U}_0 \times_1 \exp(tA_1) \times_2 \cdots \times_d \exp(tA_d), \tag{2.21}$$

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 (2.20) can be computed by the Tucker operator (2.21) 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 2.4.4.

2.3.4 Preconditioning of linear systems

Suppose we want to solve the semilinear PDE

$$\begin{cases} \partial_t u(t,x) = \mathcal{A}u(t,x) + f(t,u(t,x)), \quad t>0, \quad x \in \Omega \subset \mathbb{R}, \\ u(0,x) = u_0(x), \end{cases} \tag{2.22}$$

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 (2.16), we obtain

$$\begin{cases} \boldsymbol{u}'(t) = A\boldsymbol{u}(t) + \boldsymbol{f}(t,\boldsymbol{u}(t)), \quad t>0, \\ \boldsymbol{u}(0) = \boldsymbol{u}_0. \end{cases} \tag{2.23}$$

A common approach to integrate system (2.23) 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 (2.22), i.e.,

$$\begin{cases} \partial_t u(t,x_1,x_2) = \mathcal{A}u(t,x_1,x_2) + f(t,u(t,x_1,x_2)), \quad t>0, \quad (x_1,x_2) \in \Omega \subset \mathbb{R}^2, \\ u(0,x_1,x_2) = u_0(x_1,x_2), \end{cases} \tag{2.24}$$

again with suitable boundary conditions, $\mathcal{A}$ a linear time-independent spatial differential operator and f a nonlinear function. As for equation (2.17), if the PDE admits a Kronecker sum structure, the application of the method of lines leads to

$$\begin{cases} \boldsymbol{u}'(t) = (I_2 \otimes A_1 + A_2 \otimes I_1)\boldsymbol{u}(t) + \boldsymbol{f}(t,\boldsymbol{u}(t)), \quad t>0, \\ \boldsymbol{u}(0) = \boldsymbol{u}_0, \end{cases} \tag{2.25}$$

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 [13]). In this case, we can use the product of the matrices arising from the discretization of equation (2.24) after neglecting all the spatial variables but one in the operator $\mathcal{A}$. We obtain then the preconditioner

$$(I_2 - \tau A_2) \otimes (I_1 - \tau A_1) = P_2 \otimes P_1 = P, \tag{2.26}$$

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 2.3.3. *Another approach of solution to equation* (2.25) *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 [117].

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 (2.9)). Finally, an effective preconditioner for the linear system is a straightforward generalization of formula (2.26), 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

$$\boldsymbol{V} \times_1 P_1^{-1} \times_2 \cdots \times_d P_d^{-1} = \boldsymbol{V}_{P^{-1}}, \tag{2.27}$$

see Lemma 2.2.1. In the package KronPACK, formula (2.27) 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 2.2. For an example of application of these techniques, in the context of solution of evolutionary diffusion–reaction equations, see Section 2.4.5.

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 [56, 151].

2.4 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 the 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 (2.5). The resulting tensor is then the $(2, 1, 3)$-permutation of $\boldsymbol{S}$ in Definition 2.2.4.

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.

2.4.1 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 2.2, 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 Figure 2.1 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`.

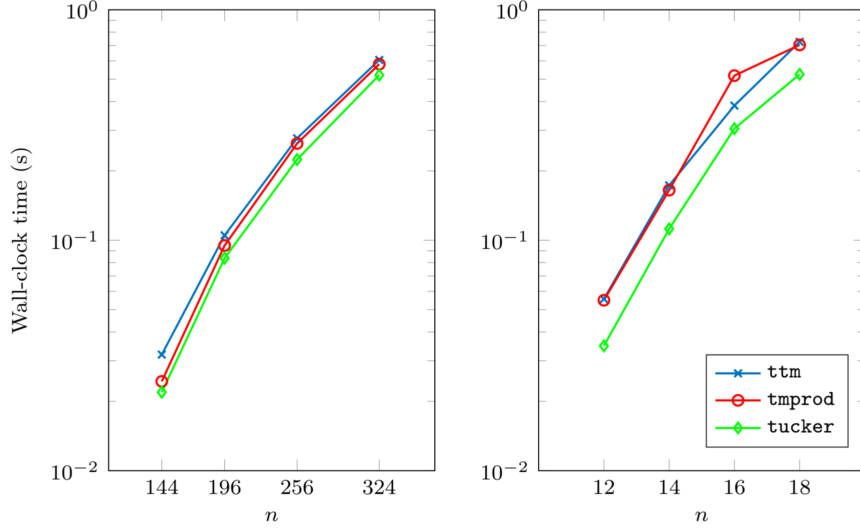

Figure 2.1: Wall-clock times for different realizations of the Tucker operator (2.5) with the functions `ttm`, `tmprod`, and `tucker`. The left plot refers to the case $d = 3$, while the right plot refers to the case $d = 6$. Each test has been repeated several times in order to avoid fluctuations.

2.4.2 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 [20]. We then introduce the *normalized and scaled* Hermite functions (orthonormal in the space $L^2(\mathbb{R})$)

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

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 [177]). This is possible by estimating the largest quadrature point for the unscaled functions by $\sqrt{2m_1 + 1}$ (see [175, 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 the space $L^2(\mathbb{R}^+)$)

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

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 [175, 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 [71]. 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 Figure 2.2. 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.

2.4.3 Multivariate interpolation

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

$$p(\boldsymbol{x}) = \sum_{i_5=1}^{m_5} \cdots \sum_{i_1=1}^{m_1} f_{i_1 \ldots i_5} L_{i_1}(x_1) \cdots L_{i_5}(x_5). \tag{2.28}$$

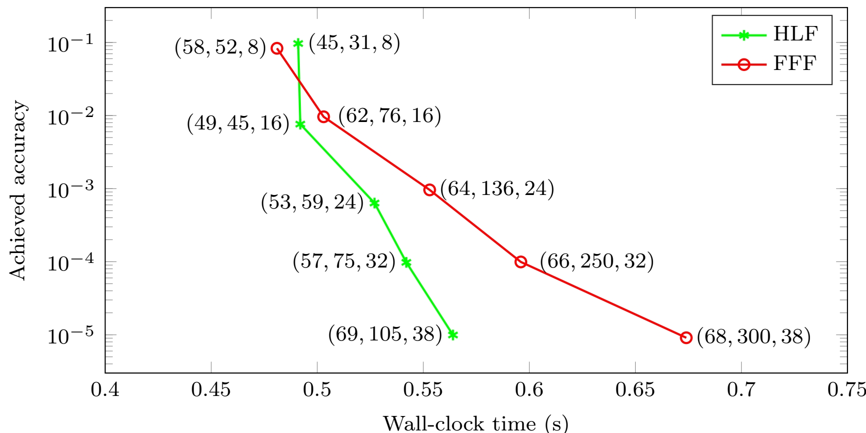

Figure 2.2: Achieved accuracies versus wall-clock times (in seconds, averaged over 20 runs) for the Hermite–Laguerre–Fourier (HLF) and the Fourier–Fourier–Fourier (FFF) approaches. The label of the marks in the plot indicates the number of basis functions used in each direction.

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 this 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

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

whose barycentric weights are

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

This is the five-dimensional version of one of the examples presented in [25, Sec. 6]. We evaluate the polynomial at a uniformly spaced Cartesian grid of points $(x_1^{\ell_1}, \ldots, x_5^{\ell_5})$, with $1 \leq \ell_\mu \leq n_\mu$. Then, approximation (2.28) at the just mentioned grid can be computed as

$$\boldsymbol{P} = \boldsymbol{F} \times_1 L_1 \times_2 \cdots \times_5 L_5, \tag{2.29}$$

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);
```

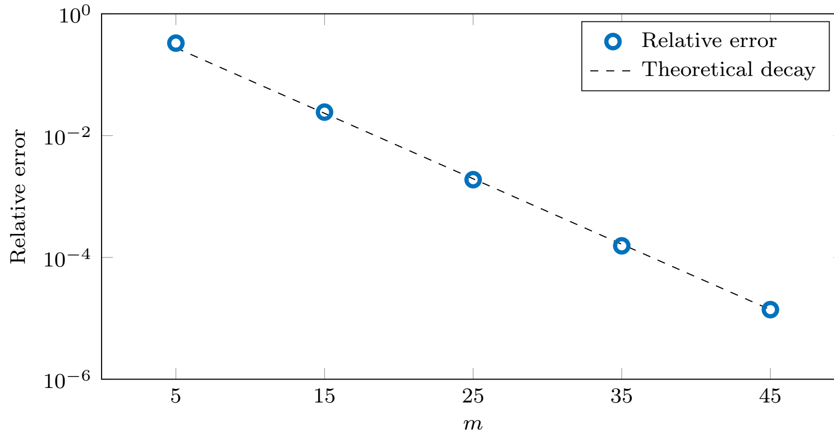

Figure 2.3: Results for approximation (2.29) with an increasing number $m_\mu = m$ of interpolation points. The relative error (blue circles) is computed in maximum norm at the evaluation points. For reference, a dashed line representing the theoretical decay estimate is added.

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 Figure 2.3, 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 [25, 181].

2.4.4 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 [191])

$$\begin{cases} \partial_t u(t,\boldsymbol{x}) + \displaystyle\sum_{\mu=1}^{3} \beta_\mu \partial_{x_\mu}(x_\mu u(t,\boldsymbol{x})) = \alpha \sum_{\mu=1}^{3} \beta_\mu^2 \partial_{x_\mu}(x_\mu^2 \partial_{x_\mu} u(t,\boldsymbol{x})) - \gamma u(t,\boldsymbol{x}), \\ u(0,\boldsymbol{x}) = u_0(\boldsymbol{x}) = x_1(2-x_1)^2 x_2(2-x_2)^2 x_3(2-x_3)^2, \end{cases} \tag{2.30}$$

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

$$\begin{cases} \boldsymbol{u}'(t) = (A_3 \oplus A_2 \oplus A_1)\boldsymbol{u}(t), \\ \boldsymbol{u}(0) = \boldsymbol{u}_0, \end{cases} \tag{2.31}$$

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

$$\boldsymbol{U}(t) = \boldsymbol{U}_0 \times_1 \exp(tA_1) \times_2 \exp(tA_2) \times_3 \exp(tA_3). \tag{2.32}$$

We consider equation (2.30) 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 2.2 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 (2.31) 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 (2.9)). 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 (2.32) implemented using the function `tucker` is much faster than any other considered approach. Indeed, this is due to the fact that formula (2.32) 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 [39].

	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

Table 2.2: Summary of the results for solving the ODEs system (2.31) 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.

2.4.5 Semilinear evolutionary equation

We consider the following three-dimensional semilinear evolutionary equation

$$\begin{cases} \partial_t u(t,\boldsymbol{x}) = \Delta u(t,\boldsymbol{x}) + \dfrac{1}{1+u(t,\boldsymbol{x})^2} + \Phi(t,\boldsymbol{x}), \\ u(0,\boldsymbol{x}) = u_0(\boldsymbol{x}) = x_1(1-x_1)x_2(1-x_2)x_3(1-x_3), \end{cases} \tag{2.33}$$

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 [111].

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

$$M\boldsymbol{u}_{k+1} = \boldsymbol{u}_k + \tau \boldsymbol{f}(t_k, \boldsymbol{u}_k), \tag{2.34}$$

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 (2.34) 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 (2.34) 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 (2.34). 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 2.3.4, an effective preconditioner for system (2.34) 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 (2.27), 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 2.3 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 (2.34) 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.

	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

Table 2.3: Summary of the results for solving the semilinear equation (2.33) by the method of lines and the backward–forward Euler method. The elapsed time is the wall-clock time measured in seconds.

2.5 Conclusions

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 the 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.

Appendix

Throughout the chapter, 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 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 [183] 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 $C \in \mathbb{C}^{r\times s}$.

Chapter 3

PHIKS: actions of φ-functions of Kronecker sums

In this chapter, we present a novel method for computing actions of the so-called φ-functions for a Kronecker sum K of d arbitrary matrices A_μ. It is based on the approximation of the integral representation of the φ-functions by Gaussian quadrature formulas combined with a scaling and squaring technique. The resulting algorithm, which we call PHIKS, evaluates the required actions by means of μ-mode products involving exponentials of the *small sized* matrices A_μ, without using the *large sized* matrix K itself. PHIKS, which profits from the highly efficient level 3 BLAS, is designed to compute different φ-functions applied on the same vector. In addition, due to the underlying scaling and squaring technique, the desired quantities are available simultaneously at suitable time scales. All these features allow the effective usage of PHIKS in the exponential integration context. In particular, we tested our newly designed method on popular exponential Runge–Kutta integrators of stiff order two and three, in comparison with state-of-the-art algorithms for computing actions of φ-functions. Our numerical experiments with discretized semilinear evolutionary 3D advection–diffusion–reaction and Allen–Cahn equations show the superiority of the μ-mode approach of PHIKS.

The material of this chapter is taken from preprint [41], i.e., M. Caliari, F. C., and F. Zivcovich. A μ-mode approach for exponential integrators: actions of φ-functions of Kronecker sums. *arXiv preprint arXiv:2210.07667*, 2022.

3.1 Introduction

We consider a system of Ordinary Differential Equations (ODEs) in *Kronecker form*

$$\begin{cases} \boldsymbol{u}'(t) = K\boldsymbol{u}(t) + \boldsymbol{g}(t,\boldsymbol{u}(t)) = \boldsymbol{f}(t,\boldsymbol{u}(t)), \quad t \in [0,T], \\ \boldsymbol{u}(0) = \boldsymbol{u}_0, \end{cases} \tag{3.1a}$$

where $\boldsymbol{g}\colon [0,T]\times\mathbb{C}^N \to \mathbb{C}^N$ is a nonlinear function of the unknown $\boldsymbol{u}\colon [0,T]\to\mathbb{C}^N$, $N = n_1\cdots n_d$, and $K \in \mathbb{C}^{N\times N}$ is a large sized matrix which can be written as a Kronecker sum, that is

$$K = A_d \oplus A_{d-1} \oplus \cdots \oplus A_1 = \sum_{\mu=1}^{d} A_{\otimes\mu}, \quad A_{\otimes\mu} = I_d \otimes \cdots \otimes I_{\mu+1} \otimes A_\mu \otimes I_{\mu-1} \otimes \cdots \otimes I_1. \tag{3.1b}$$

Here $A_\mu \in \mathbb{C}^{n_\mu\times n_\mu}$ is an arbitrary complex matrix and I_μ is the $n_\mu \times n_\mu$ identity matrix. Such a structure arises for instance when applying the method of lines to some evolutionary PDEs, from different fields of science and engineering, in domains which are the Cartesian product of d intervals.

Typical examples are semilinear diffusion and Schrödinger equations, with linear operators Δ and $\mathrm{i}\Delta$, respectively, discretized with finite differences. More in general, the matrices A_μ can be sparse (as arising from finite differences or lumped mass tensor product finite elements applied to differential operators), dense (as arising from standard tensor product finite elements, spectral differentiations or discretizations of fractional operators), and possibly singular (due to encapsulated boundary conditions, for instance). System (3.1) is typically *stiff*, and a prominent way to numerically integrate it in time is by using explicit exponential integrators [112]. These schemes, as opposed to fully implicit or IMEX integrators, do not require the solution of (non)linear systems but rather the action of the exponential and/or of the so-called φ-functions, defined as

$$\varphi_\ell(X) = \int_0^1 f_\ell(\theta, X) d\theta, \quad f_\ell(\theta, X) = \frac{\theta^{\ell-1}}{(\ell-1)!}\exp((1-\theta)X), \quad \ell \geq 1, \quad X \in \mathbb{C}^{N\times N}. \tag{3.2}$$

The direct approximation of the matrix φ-functions is feasible only when the size of the matrix is not too large. In this case, the most commonly employed techniques are based on Padé approximations [24]. On the other hand, when the size is large (which is the case of interest in this work), it is possible to rely on methods which directly compute the action of the matrix φ-functions on a vector, or even their linear combination at once. Among them, Krylov methods [97, 132, 149] and other polynomial methods [6, 40, 43, 44, 45] do not require the solution of linear systems.

If we consider the matrix exponential case, it is possible to exploit the fact that K has a Kronecker sum structure and compute the action $\exp(K)\boldsymbol{v}$ with a μ-mode approach (see [39, 42]). With this technique, it is possible to efficiently implement exponential schemes which require the action of the matrix exponential only, such as some splitting schemes, exponential Lawson and Magnus integrators. Unfortunately, such an elegant approach does not directly apply to the computation of the action of φ-functions, since they do not enjoy the splitting property of the exponential function.

In this work, we aim at extending the μ-mode approach developed for the matrix exponential to the computation of actions of φ-functions for a matrix K which is the Kronecker sum of d matrices A_μ, without assembling the matrix K itself. In particular, we will derive a quadrature-based algorithm for the computation of actions of φ-functions on the *same* vector

$$\{\exp(\tau K)\boldsymbol{v}, \varphi_1(\tau K)\boldsymbol{v}, \varphi_2(\tau K)\boldsymbol{v}, \ldots, \varphi_p(\tau K)\boldsymbol{v}\} \tag{3.3}$$

at once, where τ is the time step size of the integrator. In addition, as a byproduct of the underlying scaling and modified squaring method [173], the algorithm can output the quantities in formula (3.3) at different time scales.

The remaining part of the chapter is structured as follows. Section 3.2 is devoted to the description of the new algorithm, which we call PHIKS (PHI-functions of Kronecker Sums), for the approximation of actions of φ-functions on the same vector, with an important subsection describing a suitable choice of the scaling parameter and of the quadrature formula. Then, in Section 3.3, we validate the implementation of PHIKS and we apply it to the numerical solution of stiff systems of ODEs with exponential Runge–Kutta integrators of second and third order. Finally, we draw some conclusions in Section 3.4.

3.2 Approximation of φ-functions of a Kronecker sum

In this section, we describe in detail how to approximate the action of single φ-functions on the same vector. A reader not familiar with the μ-mode and tensor formalism is invited to check References [39, 42, 120].

A ν-stage explicit exponential Runge–Kutta integrator [112] with time step size τ is defined by

$$\begin{aligned}
\boldsymbol{u}_{ni} &= \exp(c_i\tau K)\boldsymbol{u}_n + c_i\tau\varphi_1(c_i\tau K)\boldsymbol{g}(t_n,\boldsymbol{u}_n) + \tau\sum_{j=2}^{i-1} a_{ij}(\tau K)\boldsymbol{d}_{nj} \\
&= \boldsymbol{u}_n + c_i\tau\varphi_1(c_i\tau K)\boldsymbol{f}(t_n,\boldsymbol{u}_n) + \tau\sum_{j=2}^{i-1} a_{ij}(\tau K)\boldsymbol{d}_{nj}, \qquad 2 \le i \le \nu, \\
\boldsymbol{u}_{n+1} &= \exp(\tau K)\boldsymbol{u}_n + \tau\varphi_1(\tau K)\boldsymbol{g}(t_n,\boldsymbol{u}_n) + \tau\sum_{i=2}^{\nu} b_i(\tau K)\boldsymbol{d}_{ni} \\
&= \boldsymbol{u}_n + \tau\varphi_1(\tau K)\boldsymbol{f}(t_n,\boldsymbol{u}_n) + \tau\sum_{i=2}^{\nu} b_i(\tau K)\boldsymbol{d}_{ni},
\end{aligned} \tag{3.4a}$$

where

$$\boldsymbol{d}_{ni} = \boldsymbol{g}(t_n + c_i\tau, \boldsymbol{u}_{ni}) - \boldsymbol{g}(t_n,\boldsymbol{u}_n). \tag{3.4b}$$

The matrix functions $a_{ij}(\tau K)$ and $b_i(\tau K)$ are linear combinations of φ-functions. Thus, for each stage $\boldsymbol{u}_{ni}$ and for the final approximation $\boldsymbol{u}_{n+1}$, we need to compute actions of φ-functions on the same vector, and combine them to obtain the required quantities. For example, let us consider the simple stiff second order exponential Runge–Kutta method ETD2RK [59], which has reduced tableau [131]

$$\begin{array}{c|c} 1 & \\ \hline & \varphi_2 \end{array}. \tag{3.5}$$

Here and throughout the chapter, by "reduced tableau" we mean that for each stage and for the final approximation $\boldsymbol{u}_{n+1}$ we write only the coefficients corresponding to the perturbation of the underlying exponential Euler scheme. It can be implemented by computing $\varphi_1(\tau K)\tau(K\boldsymbol{u}_n + \boldsymbol{g}(t_n,\boldsymbol{u}_n))$ and $\varphi_2(\tau K)\tau(\boldsymbol{g}(t_{n+1},\boldsymbol{u}_{n2}) - \boldsymbol{g}(t_n,\boldsymbol{u}_n))$, and then appropriately assembling the results.

We start with the computation of $\varphi_1(K)\boldsymbol{v}$, where, for clarity of exposition, we omit the time step size τ and use a generic vector $\boldsymbol{v}$. As mentioned in the introduction, the idea is to apply a quadrature rule to the integral definition of the φ-functions. In this way, we can fully exploit the possibility to apply the Tucker operator to compute actions of suitable matrix exponentials. Hence, we directly approximate the integral representation

$$\varphi_1(K)\boldsymbol{v} = \int_0^1 \exp((1-\theta)K)\boldsymbol{v}d\theta \tag{3.6}$$

by a quadrature formula. In order to avoid an impractical number of quadrature points, we introduce a scaling strategy. Therefore, the quadrature rule is applied to the computation of $\varphi_1(K/2^s)\boldsymbol{v}$, that is

$$\varphi_1(K/2^s)\boldsymbol{v} = \int_0^1 \exp((1-\theta)K/2^s)\boldsymbol{v}d\theta \approx \sum_{i=1}^{q} w_i \exp((1-\theta_i)K/2^s)\boldsymbol{v},$$

where θ_i and w_i are q quadrature nodes and weights, respectively, and we scale by a power of two in order to employ the favorable scaling and squaring algorithm [173] for matrix φ-functions. The choices of the quadrature formula, of the number q of quadrature nodes, and of the nonnegative integer scaling s will be discussed in detail in Section 3.2.1. Then, the evaluation of the integrand above at each quadrature point $\theta_i \in [0,1]$ can be performed by the Tucker operator

$$\boldsymbol{V} \times_1 \exp((1-\theta_i)A_1/2^s) \times_2 \exp((1-\theta_i)A_2/2^s) \times_3 \cdots \times_d \exp((1-\theta_i)A_d/2^s), \tag{3.7}$$

where $\mathrm{vec}(\boldsymbol{V}) = \boldsymbol{v}$. Finally, in order to recover $\varphi_1(K)\boldsymbol{v}$ from its scaled version, we use the following squaring formula (see again Reference [173])

$$\begin{cases} \varphi_1(K/2^{j-1})\boldsymbol{v} = \dfrac{1}{2}\left(\exp(K/2^j)\varphi_1(K/2^j)\boldsymbol{v} + \varphi_1(K/2^j)\boldsymbol{v}\right), \\ \exp(A_\mu/2^{j-1}) = \exp(A_\mu/2^j)\exp(A_\mu/2^j), \end{cases}$$

which has to be repeated for $j = s, s-1, \ldots, 1$. Notice that, to perform the squaring, *no* full matrix $\exp(K/2^j)$ has to be evaluated in practice. In fact, in order to compute its action on $\varphi_1(K/2^j)\boldsymbol{v}$, which is available as a tensor, it is enough to compute the Tucker operator with the small sized matrices $\exp(A_\mu/2^j)$.

Now, let us consider the approximation of the action of $\varphi_2(K)$, that is

$$\varphi_2(K)\boldsymbol{v} = \int_0^1 \theta \exp((1-\theta)K)\boldsymbol{v} d\theta. \tag{3.8}$$

Comparing integrals (3.6) and (3.8) it appears clear that, if we define a common scaling strategy, we can compute the two approximations at once just by selecting the same quadrature nodes and weights, but different integrand functions

$$\exp((1-\theta)K/2^s)\boldsymbol{v} \quad \text{and} \quad \theta \exp((1-\theta)K/2^s)\boldsymbol{v}.$$

Therefore, the two quadrature formulas can be implemented with common evaluations of the matrices $\exp((1-\theta_i)A_\mu/2^s)$ for each quadrature point θ_i and each μ. Their action on $\boldsymbol{v}$ is computed with a single Tucker operator (3.7), followed by the multiplication by the scalar θ_i needed for the approximation of $\varphi_2(K/2^s)\boldsymbol{v}$. After assembling the quadrature, the steps of the squaring are

$$\begin{cases} \varphi_2(K/2^{j-1})\boldsymbol{v} = \dfrac{1}{4}\left(\exp(K/2^j)\varphi_2(K/2^j)\boldsymbol{v} + \varphi_1(K/2^j)\boldsymbol{v} + \varphi_2(K/2^j)\boldsymbol{v}\right), \\ \varphi_1(K/2^{j-1})\boldsymbol{v} = \dfrac{1}{2}\left(\exp(K/2^j)\varphi_1(K/2^j)\boldsymbol{v} + \varphi_1(K/2^j)\boldsymbol{v}\right), \\ \exp(A_\mu/2^{j-1}) = \exp(A_\mu/2^j)\exp(A_\mu/2^j), \end{cases}$$

to be repeated for $j = s, s-1, \ldots, 1$.

The generalization to the computation of the action of the first p φ-functions on the *same* vector $\boldsymbol{v}$

$$\{\varphi_1(K)\boldsymbol{v}, \varphi_2(K)\boldsymbol{v}, \ldots, \varphi_p(K)\boldsymbol{v}\}$$

is straightforward. First, we compute their approximations at the same scaled matrix by the common quadrature rule, i.e.,

$$\varphi_\ell(K/2^s)\boldsymbol{v} \approx \sum_{i=1}^{q} w_i \frac{\theta_i^{\ell-1}}{(\ell-1)!} \exp((1-\theta_i)K/2^s)\boldsymbol{v}, \quad \ell = 1, \ldots, p. \tag{3.9a}$$

Then, we perform the squaring procedure

$$\begin{cases} \varphi_\ell(K/2^{j-1})\boldsymbol{v} = \dfrac{1}{2^\ell}\Big(\exp(K/2^j)\varphi_\ell(K/2^j)\boldsymbol{v} + \displaystyle\sum_{k=1}^{\ell} \frac{\varphi_k(K/2^j)\boldsymbol{v}}{(\ell-k)!}\Big), \quad \ell = p, p-1, \ldots, 1, \\ \exp(A_\mu/2^{j-1}) = \exp(A_\mu/2^j)\exp(A_\mu/2^j), \end{cases} \tag{3.9b}$$

for $j = s, s-1, \ldots, 1$. We stress that the relevant computations in formulas (3.9) are performed by means of the μ-mode approach, without forming the large sized matrix K. Notice also that, from the squaring formula we obtain at no additional cost also $\varphi_\ell(K/2^{j-1})\boldsymbol{v}$, $j = 2, \ldots, s$, which could be useful for the efficient implementation of exponential integrators that require, for instance, (some of) the quantities

$$\exp(c_j\tau K)\boldsymbol{v},\ \varphi_1(c_j\tau K)\boldsymbol{v},\ \varphi_2(c_j\tau K)\boldsymbol{v},\ \ldots,\ \varphi_p(c_j\tau K)\boldsymbol{v}, \quad c_j = c/2^{j-1},\ c \in \mathbb{C}. \tag{3.10}$$

Remark 3.2.1. *Notice that the quadrature rule in formula* (3.9a) *is equivalent to*

$$\sum_{i=1}^{q} w_i \frac{\theta_i^{\ell-1}}{(\ell-1)!} \exp((1-\theta_i)(K-\sigma I)/2^s)\mathrm{e}^{(1-\theta_i)\sigma/2^s}\boldsymbol{v}, \quad \ell = 1, \ldots, p,$$

where $\sigma \in \mathbb{C}$ is a shift parameter. Given the Kronecker sum structure of K, it is possible to choose σ as the sum of d shifts σ_μ, selected in such a way that $A_\mu - \sigma_\mu I$ has a smaller norm than A_μ (and thus its exponential can be possibly computed in a more efficient way [6, 46]). A common and effective choice for σ_μ is the trace of the matrix A_μ divided by n_μ, which corresponds to its average eigenvalue and minimizes the Frobenius norm of $A_\mu - \sigma_\mu I$.

We now summarize the number of Tucker operators of the whole procedure needed to obtain the quantities (3.3) together with $\hat{s}$ scales (3.10). We recall that, if we assume $n_1 = \ldots = n_d = n$, the computational cost of a single Tucker operator is $\mathcal{O}(n^{d+1})$. For each quadrature point we need to compute one Tucker operator. Then, for each step of the squaring phase, we have p Tucker operators. Finally, we have one Tucker operator for the computation of $\exp(K/2^{j-1})\boldsymbol{v}$ for each $j = 1, \ldots, \hat{s}$. Therefore, the total number of Tucker operators is

$$T(s, \hat{s}, q, p) = q + sp + \hat{s}. \tag{3.11}$$

We remark that for $d \geq 3$ the number T gives an adequate indication of the computational cost of the whole procedure, being the Tucker operator the most expensive operation. On the other hand, for $d < 3$ other tasks such as the computation of the matrix exponential may have a comparable cost (or even higher, in the trivial case $d = 1$).

Remark 3.2.2. *A similar approach can be applied also to the integral formulation of linear combinations of actions of φ-functions on different vectors, i.e., in order to* directly *approximate in a μ-mode fashion quantities of the form*

$$\exp(\tau K)\boldsymbol{v}_0 + \varphi_1(\tau K)\boldsymbol{v}_1 + \varphi_2(\tau K)\boldsymbol{v}_2 + \cdots + \varphi_p(\tau K)\boldsymbol{v}_p.$$

This is useful for high stiff order exponential integrators, see for example References [111, 130]. For clarity and brevity of exposition, the details are not reported here, and an interested reader is invited to check Reference [41].

3.2.1 Choice of s, q, and quadrature formula

The choice of the scaling value s and the number of quadrature points q is based on a suitable expansion of the error of the quadrature formula. After this selection, the algorithm is *direct* and no convergence test or exit criterion is needed. We start writing

$$\varphi_\ell(K) = \int_0^1 f_\ell(\theta, K)d\theta = \sum_{i=1}^{q} w_i f_\ell(\theta_i, K) + R_q(f_\ell(\cdot, K)), \tag{3.12}$$

where $R_q(f_\ell(\cdot, K))$ is the remainder

$$R_q(f_\ell(\cdot, K)) = \frac{1}{2\pi \mathrm{i}} \oint_\Gamma k_q(z) f_\ell(z, K)dz, \tag{3.13}$$

see Section 4.6 of Reference [65]. Here, $\Gamma \subset \mathbb{C}$ is an arbitrary simple closed curve surrounding the interval $[0, 1]$ and k_q is the kernel defined by

$$k_q(z) = \int_0^1 \frac{\pi_q(t)}{\pi_q(z)(z - t)} dt,$$

with $\pi_q(t)$ the monic polynomial of degree q with the quadrature points as roots.

Given a tolerance δ and starting from $s = 0$, we look for the smallest number $q_0 \in [q_{\min}, q_{\max}]$ of quadrature points such that

$$\|R_{q_0}(f_\ell(\cdot, K))\|\|\boldsymbol{v}\| \leq \delta, \quad \ell = 1, \ldots, p.$$

We then repeat the procedure for increasing values of the scaling $s_j \in \{1, 2, \ldots\}$ and look for the corresponding smallest value q_j such that

$$\|R_{q_j}(f_\ell(\cdot, K/2^{s_j}))\|\|\boldsymbol{v}\| \leq \delta \cdot 2^{\ell s_j}, \quad \ell = 1, \ldots, p.$$

Here, the tolerance is amplified by the factor $2^{\ell s_j}$ because we take into account that squaring formula (3.9b) requires s_j divisions by 2^ℓ. We continue until the number $T(s_{\bar{j}+1}, \hat{s}, q_{\bar{j}+1}, p)$ of Tucker operators in formula (3.11) is larger than $T(s_{\bar{j}}, \hat{s}, q_{\bar{j}}, p)$. The obtained values $s = s_{\bar{j}}$ and $q = q_{\bar{j}}$ are then employed in the approximation of actions of φ-functions applied on the vector $\boldsymbol{v}$ through formulas (3.9).

The previous estimates clearly require computable bounds for the remainders with different numbers of quadrature points, integrand functions and scaling parameters. To avoid cumbersome notation, we explain the procedure for $R_q(f_\ell(\cdot, K))$ in formula (3.13). We choose $\Gamma = \Gamma_r$ to be the ellipse with foci in $\{0, 1\}$ and logarithmic capacity (half sum of its semi-axes) $r > 1/4$, that is

$$\Gamma_r = \left\{ z \in \mathbb{C} : z = z(\zeta) = r\mathrm{e}^{\mathrm{i}\zeta} + \frac{1}{2} + \frac{\mathrm{e}^{-\mathrm{i}\zeta}}{16r}, \text{ with } \zeta \in [0, 2\pi) \right\}.$$

Then, we have

$$\begin{aligned}\|R_q(f_\ell(\cdot, K))\| &= \left\| \frac{1}{2\pi\mathrm{i}} \oint_{\Gamma_r} k_q(z) f_\ell(z, K) dz \right\| = \\ &= \frac{1}{2\pi} \left\| \int_0^{2\pi} k_q(z(\zeta)) f_\ell(z(\zeta), K) \left(r\mathrm{e}^{\mathrm{i}\zeta} - \frac{\mathrm{e}^{-\mathrm{i}\zeta}}{16r} \right) d\zeta \right\|.\end{aligned}$$

Finally, by using the fact that the numerical range of K (denoted by $\mathcal{W}(K)$) is a $(1 + \sqrt{2})$-spectral set [63], we estimate in 2-norm

$$\|R_q(f_\ell(\cdot, K))\|_2 \leq \frac{1 + \sqrt{2}}{2\pi} \sup_{w \in \Omega} \left| \int_0^{2\pi} k_q(z(\zeta)) f_\ell(z(\zeta), w) \left(r\mathrm{e}^{\mathrm{i}\zeta} - \frac{\mathrm{e}^{-\mathrm{i}\zeta}}{16r} \right) d\zeta \right|, \tag{3.14}$$

being $\Omega \subset \mathbb{C}$ a smooth, bounded, convex domain which embraces $\mathcal{W}(K)$. In our situation, we can easily find such a domain without assembling the matrix K. Indeed, it is possible to show that

$$\mathcal{W}(K) = \mathcal{W}(A_{\otimes 1}) + \mathcal{W}(A_{\otimes 2}) + \cdots + \mathcal{W}(A_{\otimes d}) = \mathcal{W}(A_1) + \mathcal{W}(A_2) + \cdots + \mathcal{W}(A_d),$$

and $\mathcal{W}(A_\mu)$ can be estimated [40] with a rectangle Ξ_μ obtained by computing the norms of the Hermitian and the skew-Hermitian parts of the small sized matrices A_μ. Thus, the rectangle $\Xi = \Xi_1 + \cdots + \Xi_d$ embraces $\mathcal{W}(K)$ and, thanks to the maximum modulus principle, the supremum in estimate (3.14) is attained at the boundary of Ξ, which we suitably discretize. Moreover, we approximate the integral by the trapezoidal rule.

Concerning the choice of the main quadrature formula (3.12), we use the Gauss–Lobatto–Legendre one. Besides being very accurate, it employs the endpoints of the integration interval $[0, 1]$. This allows on one side to avoid one Tucker operator of type (3.7) (since $\theta_q = 1$), and on the other to avail of the quantities $\exp(A_\mu/2^s)$ (corresponding to $\theta_1 = 0$), which are needed for the squaring procedures. This shrewdness, together with few others, was taken into account in the implementation of the algorithm, leading to a total number of performed Tucker operators slightly smaller than the theoretical value (3.11). Finally, the evaluation of the kernel k_q in estimate (3.14) is obtained by the recurrence relation of the underlying orthogonal polynomials (see Reference [98]).

3.3 Numerical experiments

In this section, we validate the MATLAB[1] implementation of PHIKS and present the effectiveness of the proposed algorithm for the numerical solution of stiff systems of ODEs with exponential Runge–Kutta

[1] The code is available at `https://github.com/caliarim/phiks` and is fully compatible with GNU Octave.

integrators of stiff order two and three. The algorithm employs the function TUCKER contained in the package KronPACK[2] to compute the underlying Tucker operators by means of μ-mode products. In addition, it uses the internal MATLAB function EXPM for the approximation of the needed matrix exponentials. Such a function is based on the double precision scaling and squaring Padé algorithm [5].

Concerning the two-dimensional example described in Section 3.3.3, we compare the efficiency of the μ-mode approach with a technique [144] recently introduced for the computation of φ-functions of matrices that have Kronecker sum structure. The method, which was developed for the two-dimensional case only, is direct, does not require an input tolerance and retrieves the action of a single φ-function of order ℓ by solving ℓ Sylvester equations. This approach has some restrictions on the input matrices (A_1 and $-A_2$ must have disjoint spectra in order to have a unique solution of the Sylvester equation) and it may suffer of ill-conditioning for φ-functions of high order (see again Reference [144]). For our purposes, we combined the relevant parts from the available code of the algorithm[3] in a function that we name SYLVPHI. In addition, in this two-dimensional example and in the three-dimensional one, we compare the μ-mode approach with state-of-the-art algorithms for computing linear combinations of actions of φ-functions for large and sparse general matrices. For convenience of the reader, we briefly describe them in the following.

- PHIPM_SIMUL_IOM[4] is a Krylov subspace solver with incomplete orthogonalization [132] which computes linear combinations of actions of φ-functions at different time scales, by expressing everything in terms of the highest order φ-function and using a recurrence relation.

- KIOPS[5] is another adaptive Krylov subspace solver with incomplete orthogonalization [97]. It computes linear combinations of actions of φ-functions at different time scales by using the augmented matrix technique.

- BAMPHI[6] is a hybrid Krylov-polynomial method [43] for computing linear combinations of actions of φ-functions at different time scales, equipped with a backward error analysis of the underlying polynomial approximation. In contrast to the previous methods, it does not require to store a Krylov subspace.

We used all these methods with an incomplete orthogonalization procedure of length two. Moreover, since their MATLAB implementations output some information that can be effectively used for successive calls, such as an estimate of the appropriate Krylov subspace size, in our numerical experience we obtained overall the best results by adopting the following strategy: for each call of the routine at a certain time step we input the information obtained by the same call at the previous time step. In addition, these three methods, together with PHIKS, require an input tolerance, but their error estimates are substantially different. For this reason, we decided to set the tolerance of each method to a value proportional to both the local error of the used time marching scheme and the 2-norm of the current approximation $\boldsymbol{u}_n$. The proportionality constant has been selected for each method and each integrator as large as possible among the powers of two, in such a way that the final error measured with respect to a reference solution is not affected by the approximation error of the matrix functions. We believe that running the experiments with the tolerances obtained in this way yields a fair comparison among all the methods, ensuring the minimal effort needed to reach the accuracy of the considered time marching schemes. The study of a more sophisticated technique for an effective choice of the tolerances is far beyond the scope of this work.

All the numerical experiments were performed on an Intel® Core™ i7-10750H CPU with six physical cores and 16GB of RAM, using MathWorks MATLAB® R2022a. The errors were measured in infinity norm relatively to either the analytical solution, when available, or to a reference solution computed with the PHIKS routine and a sufficiently large number of time steps.

---

[2]`https://github.com/caliarim/KronPACK`
[3]`https://github.com/jmunoz022/Kronecker_EI`
[4]`https://github.com/drreynolds/Phipm_simul_iom`
[5]`https://gitlab.com/stephane.gaudreault/kiops/-/tree/master/`
[6]`https://github.com/francozivcovich/bamphi`

3.3.1 Code validation

We extensively tested the PHIKS routine and we present here the results regarding the approximation of actions of φ-functions on the same vector up to order $p = 5$. The input matrices $K \in \mathbb{C}^{n^d \times n^d}$ arise from the discretization by standard second order finite differences of the complex operator $(1 + \mathrm{i})/100 \cdot \Delta$ in the spatial domain $[0, 1]^d$, for $d = 3$ and $d = 6$, with homogeneous Dirichlet boundary conditions. The application vector $\boldsymbol{v}$ is the discretization of

$$4096(1 + \mathrm{i}) \prod_{\mu=1}^{d} x_\mu(1 - x_\mu).$$

The number n of discretization points for each spatial direction ranges from 64 to 121 for $d = 3$, and from 8 to 11 for $d = 6$. As a term of comparison we consider the results obtained with KIOPS. Both routines were called with input tolerance set to the double precision unit roundoff value 2^{-53}. We

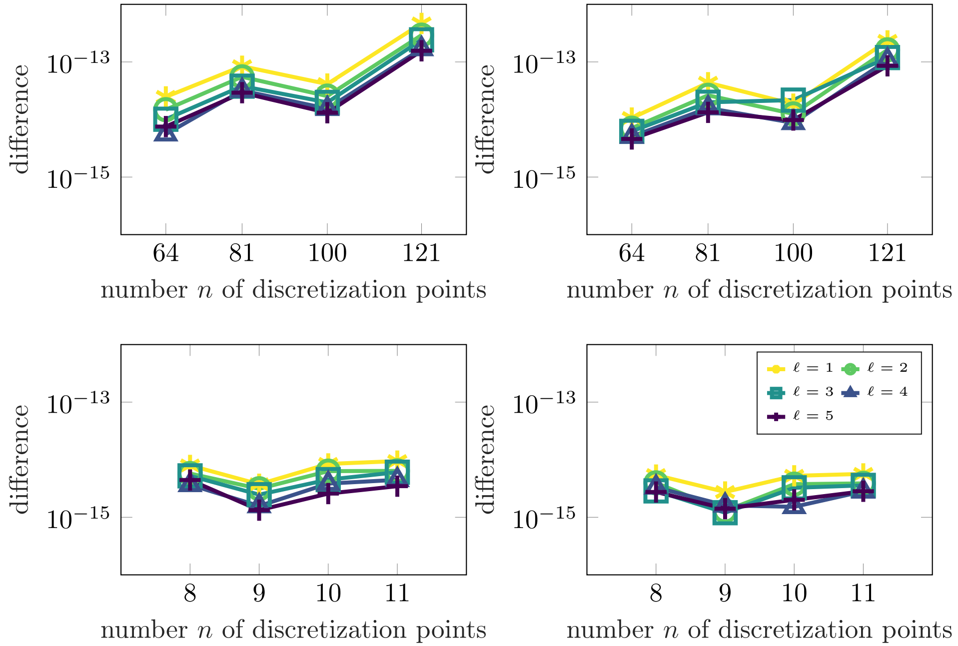

Figure 3.1: Relative difference between KIOPS and PHIKS, measured in infinity norm, for the actions $\varphi_\ell(K/2^j)\boldsymbol{v}$, $\ell = 1, \ldots, p$, in the code validation. The plots refer to $j = 0$ and $d = 3$ (top left), $j = 1$ and $d = 3$ (top right), $j = 0$ and $d = 6$ (bottom left), $j = 1$ and $d = 6$ (bottom right).

report in Figure 3.1 the relative difference in infinity norm between the approaches and, for PHIKS, we collect in Table 3.1 the values of the scaling parameter s, the number of quadrature points q, and the corresponding number of Tucker operators. Overall, we observe an homogeneous behavior of the relative difference between KIOPS and PHIKS for all the values of d, n, and ℓ, and a number of Tucker operators required by the routine PHIKS which increases very slowly with n.

3.3.2 Evolutionary advection–diffusion–reaction equation

In this section we consider the following evolutionary advection–diffusion–reaction (ADR) equation

$$\begin{cases} \partial_t u(t, x_1, x_2, x_3) = \varepsilon \Delta u(t, x_1, x_2, x_3) + \alpha(\partial_{x_1} + \partial_{x_2} + \partial_{x_3})u(t, x_1, x_2, x_3) \\ \qquad\qquad + g(t, x_1, x_2, x_3, u(t, x_1, x_2, x_3)), \\ u_0(x_1, x_2, x_3) = 64x_1(1 - x_1)x_2(1 - x_2)x_3(1 - x_3), \end{cases} \tag{3.15a}$$

	φ-functions on the same vector							
	$d = 3$				$d = 6$			
n	64	81	100	121	8	9	10	11
s	8	8	9	9	3	3	3	4
q	10	12	11	12	11	11	12	10
T	51	53	57	58	27	27	28	31

Table 3.1: Values of the scaling parameter, number of quadrature points, and number of Tucker operators employed by PHIKS in the code validation to compute actions of φ-functions on the same vector.

in the spatial domain $[0, 1]^3$, where the nonlinear function g is defined by

$$g(t, x_1, x_2, x_3, u(t, x_1, x_2, x_3)) = \frac{1}{1 + u(t, x_1, x_2, x_3)^2} + \Psi(t, x_1, x_2, x_3). \tag{3.15b}$$

Here, $\Psi(t, x_1, x_2, x_3)$ is chosen so that the analytical solution is $u(t, x_1, x_2, x_3) = \mathrm{e}^t u_0(x_1, x_2, x_3)$. Finally, the equation is coupled with homogeneous Dirichlet boundary conditions. The diffusion and advection parameters are set to $\varepsilon = 0.5$ and $\alpha = 10$, respectively. After the semidiscretization in space by second order centered finite differences we end up with an ODEs system of type (3.1), with K a matrix with heptadiagonal structure. This is a three-dimensional variation of a standard stiff example [111] for exponential integrators. For the time integration, we consider the ETD2RK scheme (3.5). We test the

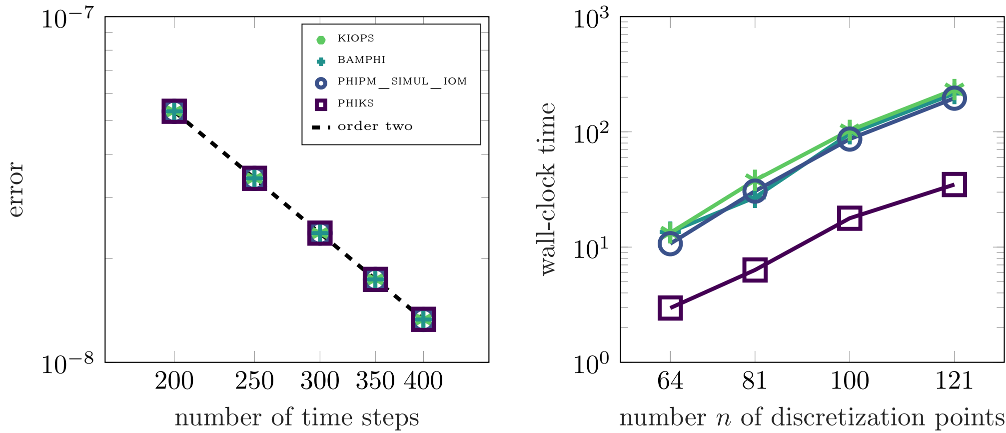

Figure 3.2: Rate of convergence of the ETD2RK scheme for the semidiscretization of ADR equation (3.15) with $n_1 = n_2 = n_3 = n = 20$ discretization points (left) and wall-clock time in seconds for increasing number n of discretization points and 100 time steps, up to final time $T = 0.1$ (right).

correct order of convergence of the scheme and the performance of the routines as the discretization is space becomes finer and finer. The results are collected in Figure 3.2. We notice that PHIKS turns out to be roughly 5.6 times faster than the best of the other methods, PHIPM_SIMUL_IOM, in the largest size scenario (total number of degrees of freedom $N = 121^3$).

3.3.3 Allen–Cahn equation

In this section we examine an example similar to the one reported in Reference [144], which describes the Sylvester approach for the computation of the φ-functions. It is the two-dimensional Allen–Cahn

phase-field model equation [88] for the concentration u

$$\begin{cases} \partial_t u(t,x_1,x_2) = \Delta u(t,x_1,x_2) + \dfrac{1}{\epsilon^2} u(t,x_1,x_2)(1-u^2(t,x_1,x_2)) \\ \qquad\qquad\qquad = \left(\Delta + \dfrac{1}{\epsilon^2}\right) u(t,x_1,x_2) + g(u(t,x_1,x_2)), \\ u(0,x_1,x_2) = u_0(x_1,x_2), \end{cases} \tag{3.16a}$$

in the spatial domain $[0,1]^2$, coupled with homogeneous Neumann boundary conditions. The initial condition is given by

$$u_0(x_1,x_2) = \tanh\left(\frac{\frac{1}{4} + \frac{1}{10}\cos\left(\beta \cdot \operatorname{atan2}\left(x_2 - \frac{1}{2}, x_1 - \frac{1}{2}\right)\right) - \sqrt{\left(x_1 - \frac{1}{2}\right)^2 + \left(x_2 - \frac{1}{2}\right)^2}}{\sqrt{2}\alpha}\right). \tag{3.16b}$$

We set $\epsilon = 0.05$, $\beta = 7$, $\alpha = 0.75$, and we discretize in space with second order centered finite differences, thus obtaining a system in form (3.1) with K a matrix with pentadiagonal structure. We simulate until final time $T = 0.025$. Notice that the linear operator $\Delta + \frac{1}{\epsilon^2}$ guarantees a unique solution for the corresponding Sylvester equation, even with homogeneous Neumann boundary conditions. As time marching scheme, we employ the third order exponential Runge–Kutta integrator with (reduced) tableau (5.9) in Reference [111]

$$\begin{array}{c|cc} c_2 & & \\ c_3 & \gamma c_2 \varphi_{2,2} + \frac{c_3^2}{c_2}\varphi_{2,3} & \\ \hline & \frac{\gamma}{\gamma c_2 + c_3}\varphi_2 & \frac{1}{\gamma c_2 + c_3}\varphi_2 \end{array} \tag{3.17}$$

with $c_3 = 2c_2 = 1/2$ and $\gamma = \frac{(3c_3-2)c_3}{(2-3c_2)c_2} = -4/5$. Its implementation involves the usage only of the φ_1 and φ_2 functions, which do not trigger the ill-conditioning of the Sylvester equation observed in Reference [144]. This integrator requires to compute the following actions (scaled by proper coefficients)

$$\begin{aligned} &\varphi_1(\tau K/2^s)\boldsymbol{f}(t_n,\boldsymbol{u}_n), && s = 0,1,2, \\ &\varphi_2(\tau K/2^s)\boldsymbol{d}_{n2}, && s = 0,1,2, \\ &\varphi_2(\tau K)\boldsymbol{d}_{n3}, && \end{aligned}$$

see formula (3.4). The SYLVPHI routine is then called in total six times: three times to compute the action of the φ_1 function at the different scales of K, twice to compute the action of the φ_2 function for the scales $s = 1$ and $s = 2$ and, finally, once to compute the action of $\varphi_2(\tau K)$ to $(\gamma \boldsymbol{d}_{n2} + \boldsymbol{d}_{n3})/(\gamma c_2 + c_3)$. Therefore, nine Sylvester equations have to be solved. The other four routines have to be called three times, one for each of the above rows. In fact, all of them are natively able to produce the action of single φ-functions simultaneously at different scales of K (see, in particular, Section 3.2 for PHIKS). The results are summarized in Figure 3.3. Also in this two-dimensional example, with numbers of degrees of freedom up to $N = 651^2$, the PHIKS routine turns out to be always the fastest, by a factor of roughly 1.5 with respect to the other techniques.

3.4 Conclusions

In this work, we proposed an efficient μ-mode approach to compute actions of φ-functions for matrices K which are Kronecker sums of any number of arbitrary matrices A_μ. This structure naturally arises when suitably discretizing in space some evolutionary PDEs of great importance in science and engineering, such as advection–diffusion–reaction and Allen–Cahn equations. The corresponding stiff system of ODEs can be effectively solved by exponential integrators, which rely on the efficient approximation of the action of φ-functions. The new method, that we named PHIKS, approximates the integral definition

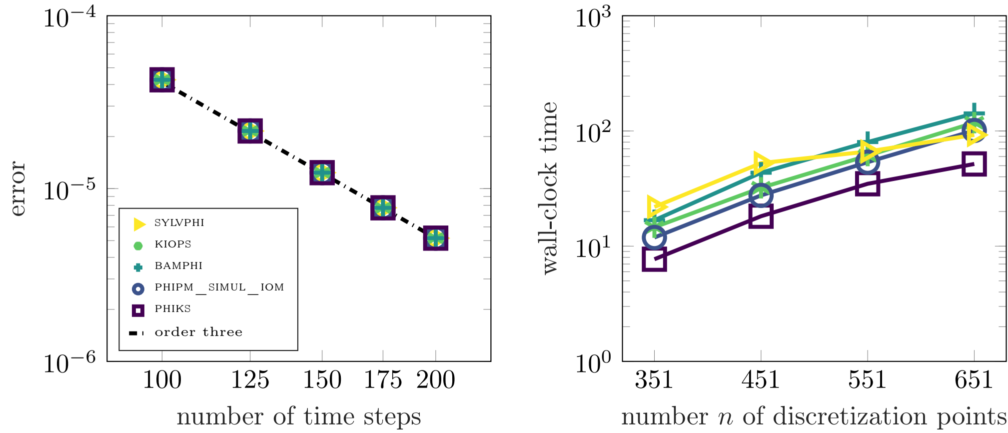

Figure 3.3: Rate of convergence of the exponential Runge–Kutta scheme (3.17) for the semidiscretization of Allen–Cahn equation (3.16) with $n_1 = n_2 = n = 21$ discretization points (left) and wall-clock time in seconds for increasing number n of discretization points and 20 time steps, up to final time $T = 0.025$ (right).

of φ-functions by the Gauss–Lobatto–Legendre quadrature formula, employs the scaling and squaring technique, and computes the required actions in a μ-mode fashion by means of Tucker operators and exponentials of the small sized matrices A_μ, exploiting the efficiency of modern hardware architectures to perform level 3 BLAS operations. We tested the approach on different stiff ODEs systems arising from the discretization of important PDEs in two and three space dimensions, using different exponential integrators (of stiff order two and three). As terms of comparison, we considered another technique for computing actions of φ-functions of Kronecker sums of matrices (based on Sylvester equations, and currently limited to two space dimensions) and more general techniques for computing actions of φ-functions. The proposed method turned out to be always faster than the others, with speed-ups ranging from 1.5 to 5.6, depending on the example under consideration. Interesting future developments are the application of the method to space-fractional diffusion equations [190] and second-order in time partial differential equations [159].

Chapter 4

PHISPLIT: direction splitting of φ-functions for exponential integrators

In this chapter, we present an efficient, practical, and easy-to-implement way to compute actions of φ-functions for matrices with a d-dimensional Kronecker sum structure in the context of exponential integrators up to second order. The method is based on a direction splitting of the involved matrix functions, which lets us exploit the highly efficient level 3 BLAS for the actual computation in a μ-mode fashion of the required actions. The approach has been successfully tested on 2D and 3D problems with various exponential integrators, resulting in a consistent speedup with respect to a state-of-the-art technique for computing actions of φ-functions for Kronecker sums.

The material of this chapter is an ongoing work temporarily named as in Reference [37], i.e., M. Caliari, and F. C.. A μ-mode based direction splitting of φ-functions for exponential integrators. *In preparation*, 2023.

4.1 Introduction

The problem of computing actions of the exponential and exponential-like functions with Kronecker sum structure received a lot of attention in the last years [39, 41, 42, 127, 128, 144]. Indeed, the efficient computation of such quantities allows to effectively employ exponential integrators for the time integration of large stiff systems of Ordinary Differential Equations (ODEs) arising from many problems of science and engineering. More in detail, we suppose to work with the following stiff system of ODEs

$$\begin{cases} \boldsymbol{u}'(t) = K\boldsymbol{u}(t) + \boldsymbol{g}(t,\boldsymbol{u}(t)), \quad t > 0, \\ \boldsymbol{u}(0) = \boldsymbol{u}_0. \end{cases} \tag{4.1}$$

Here $\boldsymbol{g}(t,\boldsymbol{u}(t))$ is a generic nonlinear function of t and of the unknown $\boldsymbol{u}(t) \in \mathbb{C}^N$, with $N = n_1 \cdots n_d$, while $K \in \mathbb{C}^{N\times N}$ is a matrix with Kronecker sum structure, i.e.,

$$K = A_d \oplus A_{d-1} \oplus \cdots \oplus A_1 = \sum_{\mu=1}^{d} A_{\otimes\mu}, \quad A_{\otimes\mu} = I_d \otimes \cdots \otimes I_{\mu+1} \otimes A_\mu \otimes I_{\mu-1} \otimes \cdots \otimes I_1, \tag{4.2}$$

where $A_\mu \in \mathbb{C}^{n_\mu\times n_\mu}$, and I_μ is the identity matrix of size n_μ. Here and throughout the chapter the symbol $\otimes$ denotes the standard Kronecker product of matrices, while $\oplus$ is employed for the Kronecker sum of matrices. Moreover, we refer to system (4.1) as a system in *Kronecker form* or with *Kronecker sum structure*.

This kind of systems naturally arises in many contexts. For example, in the two dimensional case $d = 2$, such a structure appears in constant coefficient matrix Riccati differential equations (see, for

instance, [2, Ch. 3])

$$\begin{cases} \boldsymbol{U}'(t) = A_1\boldsymbol{U}(t) + \boldsymbol{U}(t)A_2^{\mathsf{T}} + C + \boldsymbol{U}(t)B\boldsymbol{U}(t), \\ \boldsymbol{U}(0) = \boldsymbol{U}_0, \end{cases} \tag{4.3}$$

where $A_1 \in \mathbb{C}^{n_1\times n_1}$, $A_2 \in \mathbb{C}^{n_2\times n_2}$, $\boldsymbol{U}(t) \in \mathbb{C}^{n_1\times n_2}$, $B \in \mathbb{C}^{n_2\times n_1}$, and $C \in \mathbb{C}^{n_1\times n_2}$. Using the properties of the Kronecker product [42], we can easily rewrite such a matrix equation as a system of ODEs in Kronecker form (4.1), i.e.,

$$\begin{cases} \boldsymbol{u}'(t) = ((I_2 \otimes A_1) + (A_2 \otimes I_1))\boldsymbol{u}(t) + \operatorname{vec}(C + \boldsymbol{U}(t)B\boldsymbol{U}(t)), \\ \boldsymbol{u}(0) = \operatorname{vec}(\boldsymbol{U}_0), \end{cases} \tag{4.4}$$

where vec is the operator which stacks the columns of the input matrix in a single vector. A remarkable case is the one of Hermitian Riccati differential equations, which are strictly related to linear quadratic optimal control problems (see [2, Ch. 4]).

Systems with Kronecker sum structure often arise also when applying the method of lines to approximate numerically the solution of a Partial Differential Equation (PDE) defined on a tensor product domain. Indeed, after semi-discretization in space of well-known parabolic equations such as Allen–Cahn, Brusselator, Gray–Scott, advection–diffusion–reaction [41, 42] or Schrödinger equations [39], we obtain a large stiff system of ODEs in form (4.1).

Once system (4.1) is given, many techniques can be employed to numerically integrate it in time, and in particular we are interested in the application of exponential integrators [112]. In fact, they are a prominent way to perform the required task since they enjoy favorable stability properties that make them suitable to work in the stiff regime. These kinds of schemes require the computation of the action of the matrix exponential and of exponential-like matrix functions (the so-called φ-functions) on vectors. They are defined, for a generic matrix $X \in \mathbb{C}^{N\times N}$, as

$$\varphi_0(X) = \mathrm{e}^X, \quad \varphi_\ell(X) = \frac{1}{(\ell-1)!}\int_0^1 \mathrm{e}^{(1-\theta)X}\theta^{\ell-1}d\theta, \quad \ell > 0, \tag{4.5a}$$

and their Taylor series expansion is given by

$$\varphi_\ell(X) = \sum_{i=0}^{\infty} \frac{X^i}{(i+\ell)!}, \quad \ell \geq 0. \tag{4.5b}$$

When the size of X allows, it is common in practice to approximate such matrix functions by means of diagonal Padé approximations [5, 24, 173] or via polynomial approximations [126], and then multiply with the vector. On the other hand, when X is large sized, this approach is computationally unfeasible, and many algorithms have been developed to perform directly the action of φ-functions on vectors. We mention, among the others, Krylov-based techniques [97, 132, 149], direct polynomial methods [6, 40, 44, 126], and hybrid techniques [43].

When X is in fact a matrix K with Kronecker sum structure (4.2), it is possible to exploit this information to compute more efficiently the action of the φ-functions on a vector. Indeed, let us consider $\ell = 0$, so that $\varphi_0(K) = \mathrm{e}^K$. Then, it is easy to see [42] that computing

$$\boldsymbol{e} = \mathrm{e}^K\boldsymbol{v} = \mathrm{e}^{A_d\oplus A_{d-1}\oplus\cdots\oplus A_1}\boldsymbol{v} = \mathrm{e}^{A_d} \otimes \mathrm{e}^{A_{d-1}} \otimes \cdots \otimes \mathrm{e}^{A_1}\boldsymbol{v} \tag{4.6}$$

is mathematically equivalent to compute

$$\boldsymbol{E} = \boldsymbol{V} \times_1 \mathrm{e}^{A_1} \times_2 \cdots \times_d \mathrm{e}^{A_d}, \tag{4.7}$$

which we refer to as the *tensor formulation.* Here, $\boldsymbol{E}$ and $\boldsymbol{V}$ are order-d tensors of size $n_1 \times \cdots \times n_d$ that satisfy $\operatorname{vec}(\boldsymbol{E}) = \boldsymbol{e}$ and $\operatorname{vec}(\boldsymbol{V}) = \boldsymbol{v}$, respectively, while vec is the operator which stacks the columns of the input tensor into a suitable single column vector. The symbol $\times_\mu$ denotes the tensor–matrix product along the mode μ, which is also known as μ-mode product, and the computation of consecutive

μ-mode products (as it happens in formula (4.7)) is usually referred to as *Tucker operator*. As extensively explained in Reference [42], the tensor formulation is computationally more attractive, since it can be implemented by exploiting the highly performant level 3 BLAS after computing the *small* sized matrix exponentials e^{A_μ}. This technique led to the so-called μ-mode integrator [39], and has been successfully used to integrate in time semi-discretizations of advection–diffusion–reaction and Schrödinger equations, eventually in combination with a splitting scheme. In particular, a consistent speedup is reported with respect to state-of-the-art techniques to compute the action of the matrix exponential on a vector, as well as a very good scaling when performing GPUs simulations.

When computing actions of φ-function of higher order, i.e., $\varphi_\ell(K)\boldsymbol{v}$ with $\ell > 0$, the last equality in formula (4.6) does not hold anymore. In Reference [41] the authors propose an approach to overcome this difficulty. In particular, they develop a method based on the application of a quadrature rule to the integral definition of the φ-functions (4.5a) in combination with a modified scaling and squaring technique. Then, for each quadrature point and squaring stage, the needed actions of matrix exponentials are performed by a Tucker operator of the form (4.7). In this way, it is possible to compute the required actions of φ-functions at a given tolerance. The technique, which has been named PHIKS, allows to efficiently implement high stiff order exponential integrators, such as exponential Runge–Kutta integrators, and has been shown to be faster than classical state-of-the-art techniques to compute combinations of actions of φ-functions. Moreover, PHIKS turned out to be more efficient than a recently proposed method [144] to compute $\varphi_\ell(K)\boldsymbol{v}$ when K is a matrix of type $I_2 \otimes A_1 + A_2 \otimes I_1$, i.e., with two-dimensional Kronecker sum structure. This approach, currently limited to $d = 2$, is based on the solution of Sylvester equations. Other recent techniques to approximate the action of the so-called Sylvester operator $A_1\boldsymbol{V} + \boldsymbol{V}A_2$ or the Lyapunov operator $A_1\boldsymbol{V} + \boldsymbol{V}A_1$ for the solution of Riccati differential equations by means of exponential Rosenbrock integrators, possibly in the context of low-rank approximation, are presented in References [127, 128].

In this work, we propose an alternative way to approximate $\varphi_\ell(K)\boldsymbol{v}$, with $\ell > 0$ and K a matrix with d-dimensional Kronecker sum structure, in the context of exponential integrators up to second order. The approach, that we call PHISPLIT, is based on a direction splitting of the matrix φ-functions of K, which generates an approximation error compatible with the one of the time marching numerical scheme and evaluates the required actions in a μ-mode fashion by means of a single Tucker operator for each φ-function. In particular, after recalling some exponential integrators in Section 4.2, we describe in Section 4.3 the proposed technique, as well as how to employ it to implement the just mentioned schemes. Then, in Section 4.4 we present some numerical experiments that show the effectiveness of PHISPLIT, and we draw some conclusions in Section 4.5.

4.2 Recall of some exponential integrators of order up to two

When numerically integrating stiff semilinear ODEs in form (4.1) (where the stiff part is represented by the matrix K), a prominent approach is to use exponential integrators [112]. For convenience of the reader, we report here (for simplicity in a constant time step size scenario) a possible derivation of the exponential schemes that will be employed later in the numerical experiments of Section 4.4.

The starting point is the variation-of-constants formula

$$\begin{aligned}\boldsymbol{u}(t_{n+1}) &= \mathrm{e}^{\tau K}\boldsymbol{u}(t_n) + \int_{t_n}^{t_{n+1}} \mathrm{e}^{(t_{n+1}-s)K}\boldsymbol{g}(s,\boldsymbol{u}(s))ds \\ &= \mathrm{e}^{\tau K}\boldsymbol{u}(t_n) + \tau\int_0^1 \mathrm{e}^{(1-\theta)\tau K}\boldsymbol{g}(t_n+\tau\theta,\boldsymbol{u}(t_n+\tau\theta))d\theta\end{aligned} \tag{4.8}$$

which expresses the analytical solution of system (4.1) at time $t_{n+1} = t_n + \tau$, where τ is the time step size. If we approximate the integrand with the rectangle left rule, we get the scheme

$$\boldsymbol{u}_{n+1} = \mathrm{e}^{\tau K}\boldsymbol{u}_n + \tau\mathrm{e}^{\tau K}\boldsymbol{g}(t_n,\boldsymbol{u}_n) = \mathrm{e}^{\tau K}(\boldsymbol{u}_n + \tau\boldsymbol{g}(t_n,\boldsymbol{u}_n)), \tag{4.9}$$

which is known as exponential Lawson–Euler scheme (see [23, Sec. A.1.1]). It is of order one and exact for linear homogeneous problems. The linear part of system (4.1) is solved exactly and thus no restriction

on the time step due to the stiffness is necessary. If instead the trapezoidal quadrature rule is applied to the integral in equation (4.8), we get the approximation

$$\boldsymbol{u}(t_{n+1}) \approx \mathrm{e}^{\tau K}\boldsymbol{u}(t_n) + \frac{\tau}{2}\left(\mathrm{e}^{\tau K}\boldsymbol{g}(t_n, \boldsymbol{u}(t_n)) + \boldsymbol{g}(t_{n+1}, \boldsymbol{u}(t_{n+1}))\right).$$

An explicit time marching scheme is then obtained by creating an intermediate stage $\boldsymbol{u}_{n2}$ which approximates $\boldsymbol{u}(t_{n+1})$ in the right hand side by the exponential Lawson–Euler scheme (4.9). Overall, we get

$$\begin{aligned}\boldsymbol{u}_{n2} &= \mathrm{e}^{\tau K}(\boldsymbol{u}_n + \tau\boldsymbol{g}(t_n, \boldsymbol{u}_n)),\\ \boldsymbol{u}_{n+1} &= \mathrm{e}^{\tau K}\left(\boldsymbol{u}_n + \frac{\tau}{2}\boldsymbol{g}(t_n, \boldsymbol{u}_n)\right) + \frac{\tau}{2}\boldsymbol{g}(t_{n+1}, \boldsymbol{u}_{n2}),\end{aligned} \tag{4.10}$$

which is an exponential Lawson method of order two, also known in literature as Lawson2b (see [23, Sec. A.1.6]).

A different approach to the approximation of the integral in formula (4.8) leads to the so-called exponential Runge–Kutta methods. Indeed, if we approximate only the nonlinear function $\boldsymbol{g}(t_n + \tau\theta, \boldsymbol{u}(t_n + \tau\theta))$ by $\boldsymbol{g}(t_n, \boldsymbol{u}(t_n))$, by using the definition of φ_1 function in equation (4.5a) we get the scheme

$$\boldsymbol{u}_{n+1} = \mathrm{e}^{\tau K}\boldsymbol{u}_n + \tau\varphi_1(\tau K)\boldsymbol{g}(t_n, \boldsymbol{u}_n),$$

which can be rewritten as

$$\boldsymbol{u}_{n+1} = \boldsymbol{u}_n + \tau\varphi_1(\tau K)(K\boldsymbol{u}_n + \boldsymbol{g}(t_n, \boldsymbol{u}_n)) \tag{4.11}$$

and is known as exponential Euler (or exponential Nørsett–Euler, see [23, Sec. A.2.1]). It is a first order scheme, and it is exact for linear problems with constant coefficients. Another possibility is to interpolate $\boldsymbol{g}(t_n + \tau\theta, \boldsymbol{u}(t_n + \tau\theta))$ with a polynomial of degree one in θ at 0 and 1, thus obtaining the approximation

$$\boldsymbol{u}(t_{n+1}) \approx \mathrm{e}^{\tau K}\boldsymbol{u}(t_n) + \tau\int_0^1 \mathrm{e}^{(1-\theta)\tau K}(\theta\boldsymbol{g}(t_{n+1}, \boldsymbol{u}(t_{n+1})) + (1-\theta)\boldsymbol{g}(t_n, \boldsymbol{u}(t_n)))d\theta.$$

By taking a stage $\boldsymbol{u}_{n2}$ which approximates $\boldsymbol{u}(t_{n+1})$ in the right hand side by the exponential Euler scheme and using the definitions of φ_1 and φ_2 functions in formula (4.5a), we obtain the second order exponential Runge–Kutta scheme (also known in literature as ETD2RK, see [23, Sec. A.2.5])

$$\begin{aligned}\boldsymbol{u}_{n2} &= \boldsymbol{u}_n + \tau\varphi_1(\tau K)(K\boldsymbol{u}_n + \boldsymbol{g}(t_n, \boldsymbol{u}_n)),\\ \boldsymbol{u}_{n+1} &= \boldsymbol{u}_{n2} + \tau\varphi_2(\tau K)(\boldsymbol{g}(t_{n+1}, \boldsymbol{u}_{n2}) - \boldsymbol{g}(t_n, \boldsymbol{u}_n)).\end{aligned} \tag{4.12}$$

Finally, we consider the Rosenbrock–Euler method (see [112, Ex. 2.20]), which can be obtained from the application of the exponential Euler scheme to the linearized differential equation (assumed to be autonomous, for simplicity)

$$\boldsymbol{u}'(t) = \left(K + \frac{\partial\boldsymbol{g}}{\partial\boldsymbol{u}}(\boldsymbol{u}_n)\right)\boldsymbol{u}(t) + \boldsymbol{g}(\boldsymbol{u}(t)) - \frac{\partial\boldsymbol{g}}{\partial\boldsymbol{u}}(\boldsymbol{u}_n)\boldsymbol{u}(t).$$

The resulting scheme is

$$\boldsymbol{u}_{n+1} = \boldsymbol{u}_n + \tau\varphi_1(\tau K_n)(K\boldsymbol{u}_n + \boldsymbol{g}(\boldsymbol{u}_n)), \tag{4.13}$$

where K_n is the Jacobian

$$K_n = K + \frac{\partial\boldsymbol{g}}{\partial\boldsymbol{u}}(\boldsymbol{u}_n)$$

evaluated at $\boldsymbol{u}_n$. It is a second order method and, in contrast to all the methods presented above, it requires the evaluation of a different matrix function $\varphi_1(\tau K_n)$ at *each time step*. The extension to non-autonomous systems is straightforward and can be written as

$$\boldsymbol{u}_{n+1} = \boldsymbol{u}_n + \tau\varphi_1(\tau K_n)(K\boldsymbol{u}_n + \boldsymbol{g}(t_n, \boldsymbol{u}_n)) + \tau^2\varphi_2(\tau K_n)\frac{\partial\boldsymbol{g}}{\partial t}(t_n, \boldsymbol{u}_n), \quad K_n = K + \frac{\partial\boldsymbol{g}}{\partial\boldsymbol{u}}(t_n, \boldsymbol{u}_n),$$

see [112, Ex. 2.21]. We invite an interested reader to check the survey paper [112] for more information and details on exponential integrators.

Remark 4.2.1. *We considered here only a selected number of exponential integrators which require the action of φ-functions. Other exponential-type schemes of first or second order could benefit from the μ-mode splitting technique for computing φ-functions of Kronecker sums that we present in this work. We mention, among the others, corrected splitting schemes [86], low-regularity schemes [165], and Magnus integrators for linear time dependent coefficient non-homogeneous equations [100].*

4.3 Direction splitting of φ-functions

As mentioned in the introduction, we suppose that we are dealing with a matrix K with Kronecker sum structure (4.2), and we are interested in computing efficiently $\varphi_\ell(\tau K)\boldsymbol{v}$, $\boldsymbol{v} \in \mathbb{C}^N$, $\tau \in \mathbb{C}$, in the context of exponential integrators. In particular, we know that by employing a scheme of order p, we make a local error $\mathcal{O}(\tau^{p+1})$, being τ the (constant) time step size. Hence, if the integrator requires to compute a quantity of the form $\tau^q\varphi_\ell(\tau K)$, with $q > 0$, it is sufficient to approximate $\varphi_\ell(\tau K)$ with an error $\mathcal{O}(\tau^{p+1-q})$, in order to preserve the order of convergence. For our schemes of interest, i.e, the ones presented in the previous section, we make use of the following result.

Theorem 4.3.1. *Let K be a matrix with d-dimensional Kronecker sum structure* (4.2)*. Then we have*

$$\varphi_\ell(\tau K) = (\ell!)^{d-1}\left(\varphi_\ell(\tau A_d) \otimes \varphi_\ell(\tau A_{d-1}) \otimes \cdots \otimes \varphi_\ell(\tau A_1))\right) + \mathcal{O}(\tau^2). \tag{4.14}$$

Proof. For compactness of presentation, we employ the following notation

$$X_d \otimes X_{d-1} \otimes \cdots \otimes X_1 = \bigotimes_{\mu=d}^{1} X_\mu, \quad X_\mu \in \mathbb{C}^{n_\mu \times n_\mu}.$$

Then, by using the Taylor expansion of the φ_ℓ function (4.5b) and the properties of the Kronecker product (see Reference [183] for a comprehensive review) we obtain

$$\begin{aligned}
(\ell!)^{d-1}\bigotimes_{\mu=d}^{1}\varphi_\ell(\tau A_\mu) &= (\ell!)^{d-1}\bigotimes_{\mu=d}^{1}\left(\frac{I_\mu}{\ell!} + \frac{\tau A_\mu}{(\ell+1)!} + \mathcal{O}(\tau^2)\right) \\
&= (\ell!)^{d-1}\left(\frac{1}{(\ell!)^d}\bigotimes_{\mu=d}^{1} I_\mu + \frac{\tau}{(\ell!)^{d-1}(\ell+1)!}\sum_{\mu=1}^{d} A_{\otimes\mu} + \mathcal{O}(\tau^2)\right) \\
&= \frac{I}{\ell!} + \frac{\tau K}{(\ell+1)!} + \mathcal{O}(\tau^2) \\
&= \varphi_\ell(\tau K) + \mathcal{O}(\tau^2),
\end{aligned}$$

where I is the identity matrix of size $N \times N$. □

Notice that in the case $\ell = 0$, i.e., for the matrix exponential, the direction splitting error of formula (4.14) is null, and when applying to a vector $\boldsymbol{v}$ we recover formula (4.6).

Formula (4.14) allows for an efficient μ-mode based implementation, similarly to the matrix exponential case (4.7). Indeed, given an order d tensor $\boldsymbol{V}$ such that $\boldsymbol{v} = \mathrm{vec}(\boldsymbol{V})$, if we define

$$\boldsymbol{p}_\ell^{(2)} = \varphi_\ell^{(2)}(\tau K)\boldsymbol{v} = \mathrm{vec}\left(((\ell!)^{d-1}\boldsymbol{V}) \times_1 \varphi_\ell(\tau A_1) \times_2 \varphi_\ell(\tau A_2) \times_3 \cdots \times_d \varphi_\ell(\tau A_d)\right)$$

we have

$$\varphi_\ell(\tau K)\boldsymbol{v} = \boldsymbol{p}_\ell^{(2)} + \mathcal{O}(\tau^2). \tag{4.15}$$

We refer to

$$\boldsymbol{P}_\ell^{(2)} = \left((\ell!)^{d-1}\boldsymbol{V}\right) \times_1 \varphi_\ell(\tau A_1) \times_2 \varphi_\ell(\tau A_2) \times_3 \cdots \times_d \varphi_\ell(\tau A_d) \tag{4.16}$$

as the *tensor formulation* of $\boldsymbol{p}_\ell^{(2)}$, so that $\mathrm{vec}(\boldsymbol{P}_\ell^{(2)}) = \boldsymbol{p}_\ell^{(2)}$. This is precisely the formulation that we propose to employ when actions of φ-functions of a matrix with Kronecker sum structure are required

for the above exponential integrators. From now on, we refer to this technique as the PHISPLIT approach. Notice that, after the computation of the *small* sized matrix functions $\varphi_\ell(\tau A_\mu)$, with $\mu = 1, \ldots, d$, a single Tucker operator is required to evaluate approximation (4.16)

Remark 4.3.1. *A similar idea can be employed in order to retrieve higher order approximations for the φ-functions. For example, for the case $d = 2$ we have*

$$\varphi_1(\tau(I_2 \otimes A_1 + A_2 \otimes I_1)) = \varphi_1(\tau A_2) \otimes \varphi_1(\tau A_1) + \frac{1}{12}\left((\mathrm{e}^{\tau A_2} - I_2) \otimes (\mathrm{e}^{\tau A_1} - I_1)\right) + \mathcal{O}(\tau^4)$$

and

$$\begin{aligned}\varphi_2(\tau(I_2 \otimes A_1 + A_2 \otimes I_1)) &= 2(\varphi_2(\tau A_2) \otimes \varphi_2(\tau A_1)) \\ &\quad + 4\left(\left(\varphi_3(2\tau A_2) - \frac{1}{6}I_2\right) \otimes \left(\varphi_3(2\tau A_1) - \frac{1}{6}I_1\right)\right) + \mathcal{O}(\tau^4).\end{aligned}$$

This is useful for the implementation of exponential integrators of order higher than two, or for lower order schemes in order to reduce the magnitude of the direction splitting error. A comprehensive theory for these kinds of approximations is currently under study.

4.3.1 Evaluation of small sized matrix φ-functions

The matrices A_μ have a much smaller size compared to K, and the corresponding matrix φ-functions can be computed without much effort. The most popular technique for general matrices is based on diagonal rational Padé approximations, coupled with a suitable scaling and squaring algorithm (which is in fact the algorithm underlying the MATLAB function `expm` for the matrix exponential). For convenience of the reader, we report here the formula which is usually adopted, that is (see Reference [173])

$$\varphi_\ell(2X) = \frac{1}{2^\ell}\left[\mathrm{e}^X \varphi_\ell(X) + \sum_{j=1}^{\ell} \frac{\varphi_j(X)}{(\ell - j)!}\right], \quad X \in \mathbb{C}^{n \times n}.$$

As we can see, the squaring of $\varphi_\ell(X)$ requires the evaluation of all the $\varphi_j(X)$ functions, for $j < \ell$. These can be computed themselves by a Padé approximation, or by using the recurrence relation

$$\varphi_{j-1}(X) = z\varphi_j(X) + \frac{1}{(j-1)!}, \quad j = 1, \ldots, \ell.$$

Hence, the computation of a single matrix function φ_ℓ makes available also all the matrix functions φ_j, $j < \ell$. A popular MATLAB routine employing this technique is `phipade` [24]. Other more recent techniques that use polynomial Taylor approximations instead of rational Padé ones are available, see for instance Reference [126].

4.3.2 Practical implementation of the exponential integrators

The implementation of the exponential Lawson methods introduced in Section 4.2, which require just actions of matrix exponentials, does not suffer from any direction splitting error, thanks to the equivalence between formulas (4.6) and (4.7). In particular, the tensor formulation of exponential Lawson–Euler is

$$\boldsymbol{U}_{n+1} = (\boldsymbol{U}_n + \tau \boldsymbol{G}(t_n, \boldsymbol{U}_n)) \times_1 \mathrm{e}^{\tau A_1} \times_2 \cdots \times_d \mathrm{e}^{\tau A_d}, \tag{4.17}$$

while the Lawson2b scheme is given by

$$\begin{aligned}\boldsymbol{U}_{n2} &= (\boldsymbol{U}_n + \tau \boldsymbol{G}(t_n, \boldsymbol{U}_n)) \times_1 \mathrm{e}^{\tau A_1} \times_2 \cdots \times_d \mathrm{e}^{\tau A_d}, \\ \boldsymbol{U}_{n+1} &= \left(\boldsymbol{U}_n + \frac{\tau}{2}\boldsymbol{G}(t_n, \boldsymbol{U}_n)\right) \times_1 \mathrm{e}^{\tau A_1} \times_2 \cdots \times_d \mathrm{e}^{\tau A_d} + \frac{\tau}{2}\boldsymbol{G}(t_{n+1}, \boldsymbol{U}_{n2}).\end{aligned} \tag{4.18}$$

Here and in the subsequent formulas we have

$$\mathrm{vec}(\boldsymbol{U}_n) = \boldsymbol{u}_n, \quad \mathrm{vec}(\boldsymbol{U}_{n2}) = \boldsymbol{u}_{n2}, \quad \mathrm{vec}(\boldsymbol{U}_{n+1}) = \boldsymbol{u}_{n+1}$$

and

$$\mathrm{vec}(\boldsymbol{G}(t_n, \boldsymbol{U}_n)) = \boldsymbol{g}(t_n, \boldsymbol{u}_n), \quad \mathrm{vec}(\boldsymbol{G}(t_{n+1}, \boldsymbol{U}_{n2})) = \boldsymbol{g}(t_{n+1}, \boldsymbol{u}_{n2}).$$

The remaining exponential integrators transform as follows. First of all, the action of the matrix K on $\boldsymbol{u}_n$ is computed in tensor form as

$$\sum_{\mu=1}^{d} (\boldsymbol{U}_n \times_\mu A_\mu)$$

without explicitly assembling the matrix K (see Reference [42]). Then, the exponential Euler PHISPLIT method is

$$\boldsymbol{U}_{n+1} = \boldsymbol{U}_n + \tau \left(\sum_{\mu=1}^{d} (\boldsymbol{U}_n \times_\mu A_\mu) + \boldsymbol{G}(t_n, \boldsymbol{U}_n) \right) \times_1 \varphi_1(\tau A_1) \times_2 \cdots \times_d \varphi_1(\tau A_d), \tag{4.19}$$

while the ETD2RK PHISPLIT scheme becomes

$$\begin{aligned} \boldsymbol{U}_{n2} &= \boldsymbol{U}_n + \tau \left(\sum_{\mu=1}^{d} (\boldsymbol{U}_n \times_\mu A_\mu) + \boldsymbol{G}(t_n, \boldsymbol{U}_n) \right) \times_1 \varphi_1(\tau A_1) \times_2 \cdots \times_d \varphi_1(\tau A_d), \\ \boldsymbol{U}_{n+1} &= \boldsymbol{U}_{n2} + \tau \left(\boldsymbol{G}(t_{n+1}, \boldsymbol{U}_{n2}) - \boldsymbol{G}(t_n, \boldsymbol{U}_n) \right) \times_1 \varphi_2(\tau A_1) \times_2 \cdots \times_d \varphi_2(\tau A_d). \end{aligned} \tag{4.20}$$

Finally, concerning the exponential Rosenbrock–Euler method, we assume that the Jacobian K_n can be written as a Kronecker sum, i.e.,

$$K_n = K + \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}}(\boldsymbol{u}_n) = J_d(\boldsymbol{U}_n) \oplus J_{d-1}(\boldsymbol{U}_n) \oplus \cdots \oplus J_1(\boldsymbol{U}_n).$$

Therefore the exponential Rosenbrock–Euler PHISPLIT method is

$$\boldsymbol{U}_{n+1} = \boldsymbol{U}_n + \tau \left(\sum_{\mu=1}^{d} (\boldsymbol{U}_n \times_\mu A_\mu) + \boldsymbol{G}(t_n, \boldsymbol{U}_n) \right) \times_1 \varphi_1(\tau J_1(\boldsymbol{U}_n)) \times_2 \cdots \times_d \varphi_1(\tau J_d(\boldsymbol{U}_n)). \tag{4.21}$$

4.4 Numerical experiments

In this section we present numerical experiments to validate PHISPLIT, the proposed approach in the context of exponential integrators to compute actions of φ-functions when the matrix has a Kronecker sum structure. In particular, we will consider a two-dimensional example from linear quadratic control and a three-dimensional example which models an advection–diffusion–reaction equation. To perform the time marching, we will employ the exponential integrators of Section 4.2 as described in Section 4.3.2 for the PHISPLIT version.

As term of comparison, we will consider the approximation of actions of φ-functions for matrices with Kronecker sum structure using PHIKS[1] [41], for both the 2D and 3D experiments. Also this algorithm operates in tensor formulation using μ-mode products, but it requires an input tolerance, which we take proportional to the local temporal order of the method and to the norm of the current solution. The proportionality constant is chosen so that the error committed by the routine, measured against a reference or analytical solution, does not affect the temporal error.

To compute all the relevant tensor operations, i.e., Tucker operators and μ-mode products, we use the functions contained in the package KronPACK[2]. Moreover, to compute the needed matrix φ-functions, we employ the internal MATLAB function `expm` (for φ_0) and the function `phipade`[3] (for φ_ℓ, $\ell > 0$). In terms of hardware, we run all the experiments employing an Intel® Core™ i7-10750H CPU with six physical cores and 16GB of RAM. As a software, we employ MathWorks MATLAB® R2022a.

[1] https://github.com/caliarim/phiks
[2] https://github.com/caliarim/KronPACK
[3] https://www.math.ntnu.no/num/expint/matlab/

4.4.1 2D experiment: linear quadratic control

We present in this section a classical example from linear quadratic control, see for instance [140, 156]. We are interested in the minimization over the scalar control $v(t) \in \mathbb{R}$ of the functional

$$\mathcal{J}(v) = \frac{1}{2}\int_0^T \left(\alpha s(t)^\mathsf{T} s(t) + v(t)^2\right) dt$$

subject to the constraints

$$\begin{aligned} &w'(t) = Aw(t) + bv(t), \quad w(0) = w_0,\\ &s(t) = cw(t). \end{aligned}$$

Here $w(t) \in \mathbb{R}^{n\times 1}$ is a column vector containing the state variables, $s(t) \in \mathbb{R}$ represents the scalar output, $A \in \mathbb{R}^{n\times n}$ is the system matrix, $b \in \mathbb{R}^{n\times 1}$ is the system column vector, $c \in \mathbb{R}^{1\times n}$ is a row vector, and $\alpha \in \mathbb{R}^+$ is a positive scalar.

Then, the solution of the constrained optimization problem is determined by the optimal control

$$v^*(t) = -b^\mathsf{T}\boldsymbol{U}(t)w(t),$$

where $\boldsymbol{U}(t) \in \mathbb{R}^{n\times n}$ satisfies the symmetric Riccati differential equation

$$\begin{cases} \boldsymbol{U}'(t) = A^\mathsf{T}\boldsymbol{U}(t) + \boldsymbol{U}(t)A + C + \boldsymbol{U}(t)B\boldsymbol{U}(t),\\ \boldsymbol{U}(0) = \boldsymbol{Z}, \end{cases} \tag{4.22}$$

with $C = \alpha c^\mathsf{T} c$ and $B = -bb^\mathsf{T}$. Here $\boldsymbol{Z} \in \mathbb{R}^{n\times n}$ is a matrix containing all zeros entries. Clearly, equation (4.22) is in form (4.3), which in turn can be seen as a problem with two-dimensional Kronecker sum structure and integrated efficiently by means of the techniques described in the Section 4.3. Notice also that the solution of equation (4.22) converges to a steady state determined by the algebraic Riccati equation

$$A^\mathsf{T}\boldsymbol{U}(t) + \boldsymbol{U}(t)A + C + \boldsymbol{U}(t)B\boldsymbol{U}(t) = 0. \tag{4.23}$$

For our numerical experiment, similarly to what previously done in the literature [127, 128, 140, 156], we take $A \in \mathbb{R}^{\hat{n}^2\times\hat{n}^2}$ as the matrix obtained by the discretization with second order centered finite differences of the operator

$$\partial_{xx} + \partial_{yy} - 10x\partial_x - 100y\partial_y \tag{4.24}$$

on the domain $[0, 1]^2$ with homogeneous Dirichlet boundary conditions. Moreover, the components b_k of the vector b are defined as

$$b_k = \begin{cases} 1 \text{ if } 0.1 < x_i \le 0.3,\\ 0 \text{ otherwise,} \end{cases} \qquad k = i + (j-1)\hat{n}, \quad i = 1,\dots,\hat{n}, \quad j = 1,\dots,\hat{n},$$

while for the components c_k of the vector c we take

$$c_k = \begin{cases} 1 \text{ if } 0.7 < x_i \le 0.9,\\ 0 \text{ otherwise,} \end{cases} \qquad k = i + (j-1)\hat{n}, \quad i = 1,\dots,\hat{n}, \quad j = 1,\dots,\hat{n}.$$

Here x_i represents the ith (inner) grid point along the x direction. Finally, we set $\alpha = 100$.

For the temporal integration of equation (4.22) we use the exponential Rosenbrock–Euler method, already employed in [127, 128], and reported in formula (4.13) (see formula (4.21) for the PHISPLIT version). In fact, the Jacobian matrix of system (4.22) has the following Kronecker sum structure

$$K_n = I \otimes (A^\mathsf{T} + \boldsymbol{U}_n B) + (A + B\boldsymbol{U}_n)^\mathsf{T} \otimes I,$$

where I is the identity matrix of size $n \times n$, with $n = \hat{n}^2$. Remark that the exponential Rosenbrock–Euler PHISPLIT method requires to evaluate at *each time step* $\varphi_1(\tau(A^\mathsf{T} + \boldsymbol{U}_n B))$ and $\varphi_1(\tau(A + B\boldsymbol{U}_n)^\mathsf{T})$, to

compute the action $K\boldsymbol{u}_n$ and to perform one Tucker operator. We will employ also the second order exponential Runge–Kutta method ETD2RK, reported in formula (4.12) and presented in PHISPLIT sense in formula (4.20). Although each time step of this integrator requires two actions of matrix functions, and thus two Tucker operators for the PHISPLIT version, plus the action $K\boldsymbol{u}_n$, in a constant time step size implementation the two matrix functions needed $\varphi_1(\tau A^\mathsf{T})$ and $\varphi_2(\tau A^\mathsf{T})$ can be computed once and for all at the beginning.

First of all, we verify the implementation of the involved exponential integrators for a long term simulation, i.e., until reaching the steady state. For this experiment, we employ $\hat{n} = 30$ inner discretization points for the x and y variables. As confirmed by the plot in Figure 4.1, we see that all the methods, both in their PHIKS and PHISPLIT implementation, reach around time 0.15 the solution of equation (4.23), which is obtained with the MATLAB function `icare` from the Control System Toolbox.

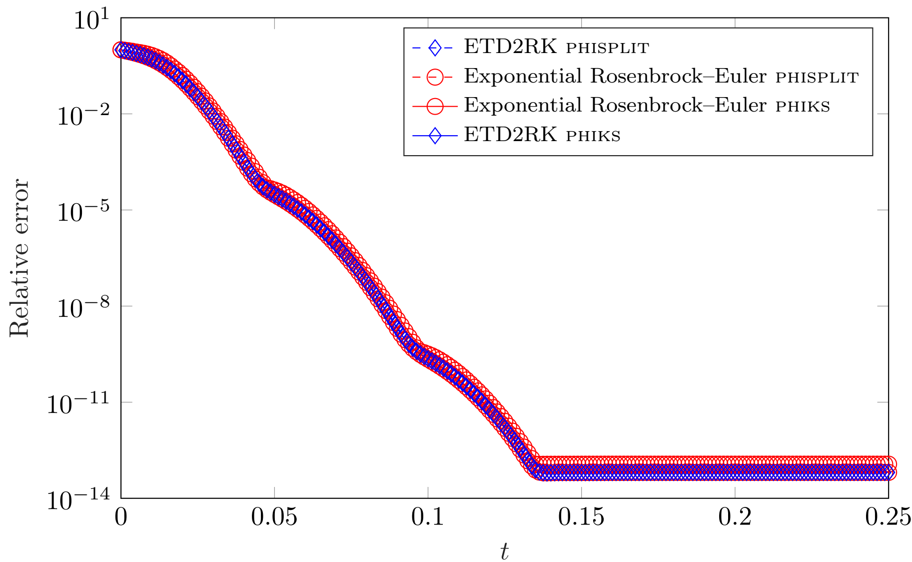

Figure 4.1: Convergence of the exponential Rosenbrock–Euler and of the ETD2RK methods, both in PHIKS and in PHISPLIT variants, to the steady state of Riccati differential equation (4.22). All the integrators have been employed with 200 time steps, and the relative error with respect to the solution of algebraic Riccati equation (4.23) is measured in Frobenius norm.

Then, we compare the performances of the integrators for the solution of equation (4.22) with $\hat{n} = 40$ up to the final time $T = 0.025$. All methods are run with different time step sizes in such a way to reach comparable relative errors with respect to a reference solution. The number of time steps for each method and simulation, together with the numerically observed convergence rate, is reported in Table 4.1. All the methods appear to be of second order, as expected.

Moreover, in Figure 4.2 we report the relative errors and the corresponding wall-clock times of the simulations. Here, we include also the performance of the built-in MATLAB function ODE23. It implements an explicit Runge–Kutta method of order three with variable step size, suggested for not stringent tolerances and for moderately stiff problems. In fact, it turned out to be the fastest routine in the ODE suite to reach accuracies in the same range of the other methods. We notice first of all that the exponential Rosenbrock–Euler method is always faster than ETD2RK in the PHIKS implementation, that is with the action of matrix functions computed at a precision that does not affect the temporal error (see the discussion at the beginning of the section). On the other hand, the two implementations with PHISPLIT are always faster compared with their PHIKS counterparts, although they require a larger number of time steps to reach a comparable accuracy. Moreover, the ETD2RK method turns out to be faster with respect to the exponential Rosenbrock–Euler method. This is mainly due to the fact that the matrix functions in the Runge–Kutta case are computed only once before the time marching. This

Exponential Rosenbrock–Euler PHIKS	# steps	15	30	45	60	75
	order	–	2.10	2.06	2.04	2.03
ETD2RK PHIKS	# steps	10	20	30	40	50
	order	–	2.12	2.07	2.05	2.04
Exponential Rosenbrock–Euler PHISPLIT	# steps	30	65	100	135	170
	order	–	2.05	2.03	2.02	2.02
ETD2RK PHISPLIT	# steps	30	65	100	135	170
	order	–	2.05	2.03	2.02	2.02

Table 4.1: Number of time steps and observed convergence rates for the time integration of the Riccati differential equation (4.22) up to final time $T = 0.025$, with different exponential integrators and $\hat{n} = 40$. The achieved errors and the wall-clock times are displayed in Figure 4.2.

method is in fact at least twice as fast as the other exponential methods, and the fastest one in any case.

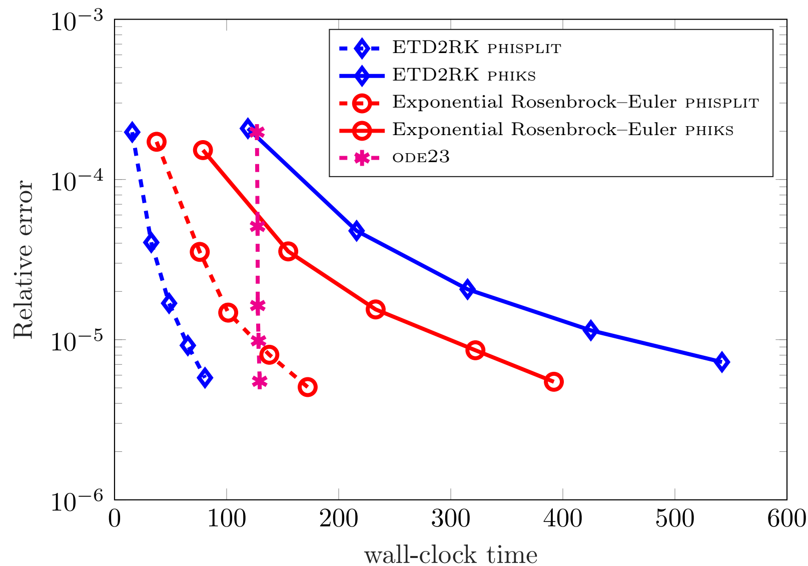

Figure 4.2: Achieved errors and wall-clock times in seconds for the solution of Riccati differential equation (4.22) up to final time $T = 0.025$, with different integrators and $\hat{n} = 40$. The numbers of time steps for each exponential method are reported in Table 4.1. The input tolerances (both absolute and relative) for ODE23 are 3e-3, 4e-4, 2.3e-4, 9.5e-5 and 8.7e-5.

Remark 4.4.1. *The discretization of the operator* (4.24) *has itself a Kronecker sum structure. Hence, it is be possible to write equation* (4.22) *(in vector formulation, for simplicity of exposition) as*

$$\begin{cases} \boldsymbol{u}'(t) = K\boldsymbol{u} + \boldsymbol{g}(\boldsymbol{u}), \\ \boldsymbol{u}(0) = \boldsymbol{z}, \end{cases}$$

where $\boldsymbol{g}$ *and* $\boldsymbol{z}$ *are the vectorization of the nonlinearity and of* $\boldsymbol{Z}$*, respectively, and* K *has the form*

$$K = I \otimes I \otimes I \otimes D_1^{\mathsf{T}} + D_2^{\mathsf{T}} \otimes I \otimes I \otimes I.$$

Here I *is an identity matrix of size* $\hat{n} \times \hat{n}$ *and* $D_1 \in \mathbb{R}^{\hat{n}\times\hat{n}}$ *and* $D_2 \in \mathbb{R}^{\hat{n}\times\hat{n}}$ *the discretizations of the operators* $\partial_{xx} - 10x\partial_x$ *and* $\partial_{yy} - 100y\partial_y$*, respectively. In the context of temporal integration with*

exponential Runge–Kutta schemes, we could then use both the PHIKS *and the* PHISPLIT *approaches with the even smaller sized matrices D_1 and D_2, forming then the approximations at every time steps using Tucker operators with order four tensors. However, as this is just possible because of the specific form of the operator* (4.24)*, we do not pursue this approach here.*

4.4.2 3D experiment: advection–diffusion–reaction

We now consider the semidiscretization in space of the following three dimensional evolutionary advection–diffusion–reaction (ADR) equation (see Reference [41])

$$\begin{cases} \partial_t u(t,x_1,x_2,x_3) = \varepsilon\Delta u(t,x_1,x_2,x_3) + \alpha(\partial_{x_1}+\partial_{x_2}+\partial_{x_3})u(t,x_1,x_2,x_3) \\ \qquad\qquad + g(t,x_1,x_2,x_3,u(t,x_1,x_2,x_3)), \\ u_0(x_1,x_2,x_3) = 64x_1(1-x_1)x_2(1-x_2)x_3(1-x_3). \end{cases} \tag{4.25}$$

The nonlinear function g is given by

$$g(t,x_1,x_2,x_3,u(t,x_1,x_2,x_3)) = \frac{1}{1+u(t,x_1,x_2,x_3)^2} + \Psi(t,x_1,x_2,x_3),$$

where $\Psi(t,x_1,x_2,x_3)$ is such that the analytical solution of the equation is

$$u(t,x_1,x_2,x_3) = \mathrm{e}^t u_0(x_1,x_2,x_3).$$

The problem is solved up to final time $T=1$ in the domain $[0,1]^3$ and completed with homogeneous Dirichlet boundary conditions. The remaining parameters are set to $\varepsilon = 0.75$ and $\alpha = 0.1$. By semidiscretizing in space with second order centered finite differences, we end up with a system of type (4.1) with K in three-dimensional Kronecker sum structure (4.2), where A_μ approximates $\varepsilon\partial_{x_\mu x_\mu} + \alpha\partial_{x_\mu}$, $\mu = 1,2,3$. In particular, we take $n_1 = 80$, $n_2 = 81$ and $n_3 = 82$ internal discretization points for the x_1, x_2 and x_3 variables, respectively. The temporal integration is performed with four methods: the Lawson–Euler scheme (4.9), the Lawson2b scheme (4.10), the Exponential Euler method (4.11) and the ETD2RK method (4.12) (see Section 4.3.2 for their practical implementation and the PHISPLIT versions). In particular, concerning the Lawson schemes, the needed matrix exponentials $\mathrm{e}^{\tau A_\mu}$, $\mu = 1,2,3$, are computed once and for all at the beginning. Then, one and two Tucker operators per time step, for the first order and second order scheme, respectively, are required to form the approximations during the temporal integration. Concerning the PHISPLIT implementation of exponential Euler and ETD2RK, again we compute once and for all the needed matrix function $\varphi_1(\tau A_\mu)$ and $\varphi_2(\tau A_\mu)$ before starting the temporal integration, and we then combine them suitably at each time step. This operation requires a single Tucker operator for the first order scheme and two for the second order one, as for the aforementioned Lawson schemes, plus the action $K\boldsymbol{u}_n$.

The number of time steps for each method, for both the PHISPLIT and PHIKS implementations, is reported in Table 4.2, while the reached relative errors and the wall-clock times are summarized in Figure 4.3. First of all, we notice that all the methods show the expected convergence rate, reported in Table 4.2 as well. In particular, for large time step sizes, the Lawson2b method suffers from an order reduction. This is expected, as in these cases the problem is more stiff, and schemes which employ just the exponential function are affected by this phenomenon (see, for instance, Reference [110]). Then, from Figure 4.3 we observe that the PHISPLIT approach is in any case the most performant among all the methods and techniques considered, with an increasing speedup for more stringent accuracies. More in detail, compared with its PHIKS counterparts, the PHISPLIT implementations are roughly 3.4 and 3.7 times faster for the first order and second order schemes, respectively, even if (in general) they require more time steps to reach a comparable accuracy. Finally, concerning the Lawson schemes, although they require the same number of Tucker operators per time step compared with the PHISPLIT implementations of the exponential Runge–Kutta schemes, they are always the least performant. This is mainly due to the requirement of a large number of time steps to reach the accuracy of the other methods, which is particularly evident for the second order scheme (see bottom of Table 4.2 and of Figure 4.3).

Method						
Lawson–Euler	# steps	800	8800	16800	24800	32800
	order	–	1.00	1.00	1.00	1.00
Exponential Euler PHIKS	# steps	50	450	850	1250	1650
	order	–	1.02	1.00	1.00	1.00
Exponential Euler PHISPLIT	# steps	50	450	850	1250	1650
	order	–	1.03	1.01	1.00	1.00

Method						
Lawson2b	# steps	3000	4500	6000	7500	9000
	order	–	1.79	1.87	1.92	1.94
ETD2RK PHIKS	# steps	20	80	140	200	260
	order	–	1.94	1.97	1.98	1.99
ETD2RK PHISPLIT	# steps	40	140	240	340	440
	order	–	2.10	2.04	2.03	2.02

Table 4.2: Number of time steps and observed convergence rates for the time integration of the semidiscretization of the advection–diffusion–reaction equation (4.25) up to final time $T = 1$, with different exponential integrators. Here we considered $n_1 = 80$, $n_2 = 81$ and $n_3 = 82$ space discretization points. The achieved errors and the wall-clock times are displayed in Figure 4.3.

4.5 Conclusions

In this work, we presented how it is possible to efficiently approximate actions of φ-functions for matrices with d-dimensional Kronecker sum structure using a μ-mode based approach. The technique, that we call PHISPLIT, is suitable when integrating initial valued Ordinary Differential Equations with exponential integrators up to second order. It is based on an inexact direction splitting of the matrix functions involved in the time marching schemes which preserves the order of the method. The effectiveness and superiority of the approach, with respect to another technique to compute actions of φ-functions in Kronecker form, has been shown on a two-dimensional problem from linear quadratic control and on a three-dimensional advection–diffusion–reaction equation, using a variety of exponential integrators. Interesting future developments would be to generalize the approach for higher order integrators and performing GPU simulations with the PHISPLIT technique, possibly in single and/or half precision (which are compatible with the magnitude of the errors of the temporal integration), for different problems from science and engineering. Also, an extension of PHISPLIT to the Mittag–Leffler functions, which appear in exponential time differencing methods for fractional differential equations (see for instance [94, 95]), would be of great interest.

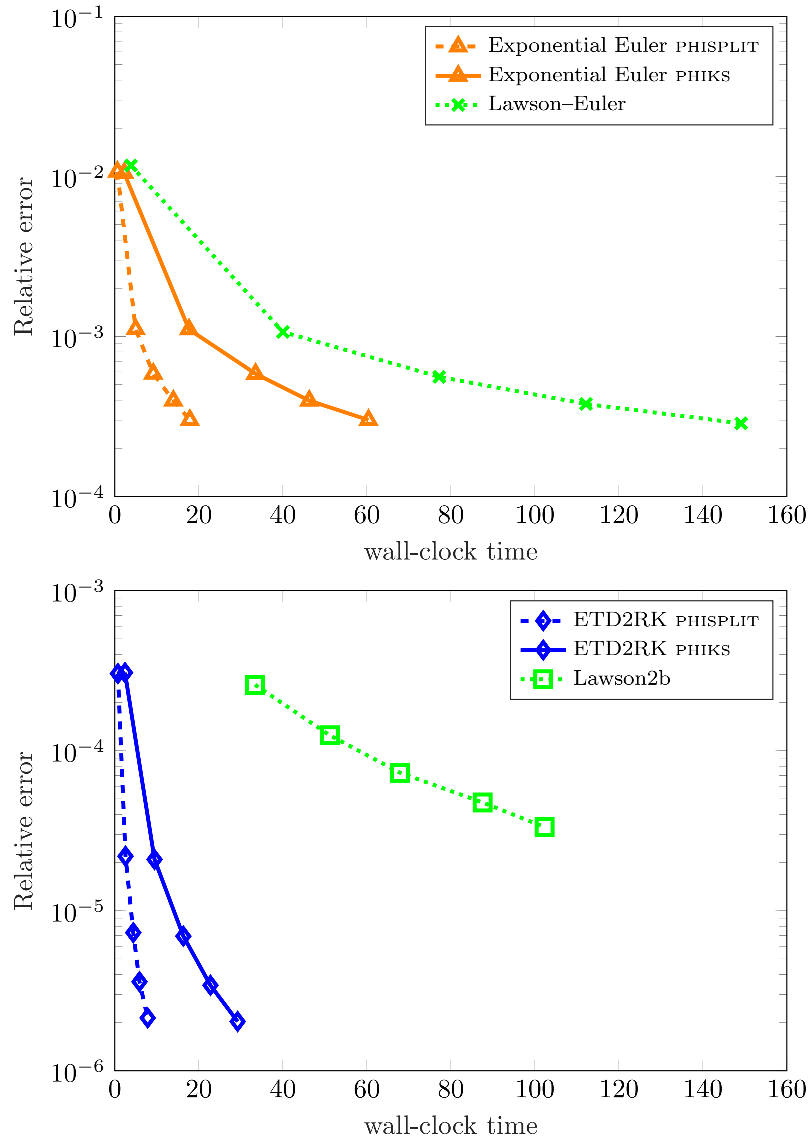

Figure 4.3: Achieved errors and wall-clock times in seconds for the solution of the semidiscretization of the advection–diffusion–reaction equation (4.25) up to final time $T = 1$, with different exponential integrators of order one (top) and order two (bottom). Here we considered $n_1 = 80$, $n_2 = 81$ and $n_3 = 82$ space discretization points. The numbers of time steps for each method are reported in Table 4.2.

Part II

Applications

Chapter 5

Ensign: dynamical low-rank 6D Vlasov simulation

Running kinetic simulations using grid-based methods is extremely expensive due to the up to six-dimensional phase space. Recently, it has been shown that dynamical low-rank algorithms can drastically reduce the required computational effort, while still accurately resolving important physical features such as filamentation and Landau damping. In this chapter, we present a new second order projector-splitting dynamical low-rank algorithm for the full six-dimensional Vlasov–Poisson equations. An exponential integrator based Fourier spectral method is employed to obtain a numerical scheme that is free of a CFL condition but still fully explicit. The resulting method is implemented with the aid of `Ensign`, a software framework which facilitates the efficient implementation of dynamical low-rank algorithms on modern multi-core CPU as well as GPU based systems. Its usage and features are briefly described as well. The presented numerical results demonstrate that 6D simulations can be run on a single workstation and highlight the significant speedup that can be obtained using GPUs.

The material of this chapter is taken from Reference [52], i.e., F. C., and L. Einkemmer. Efficient 6D Vlasov simulation using the dynamical low-rank framework `Ensign`. *Comput. Phys. Commun.*, 280:108489, 2022.

5.1 Introduction

Efficiently solving kinetic equations is important in applications ranging from plasma physics to radiative transport. The main challenge in this context is the up to six-dimensional phase space and the associated unfavorable scaling of computational cost and memory requirements. Assuming n discretization points for each direction of a 6D phase space, the storage cost of a direct discretization scales as $\mathcal{O}(n^6)$. This is usually referred to as the curse of dimensionality. To mitigate this issue, many techniques have been proposed in the literature; we mention, for example, particle methods [48, 184] and sparse grid approximations [102, 122]. However, it is well known that the former misses or underresolves some important physical phenomena (such as Landau damping or regions with low density), while the latter has issues with the Gaussian equilibrium distribution and the low regularity inherent in collisionless kinetic problems. Because of this, direct simulations are routinely conducted on large supercomputers [27, 73]. However, current computational constraints mostly limit this approach to four- and some five-dimensional problems.

More recently, a dynamical low-rank algorithm for solving the Vlasov–Poisson equations has been proposed in [82]. In the context of a 6D phase space, dynamical low-rank integrators approximate the solution by a set of only three-dimensional advection problems. The resulting algorithm has the primary advantage of having storage and computational costs that scale as $\mathcal{O}(rn^3)$ and $\mathcal{O}(r^2n^3)$, respectively, where r is the (usually small) rank of the approximation. This can result in a drastic reduction of both

memory consumption as well as computational effort. The main enabling technology was the introduction of the projector-splitting integrator [134], which allows us to obtain a robust integrator without the need for regularization within the framework of dynamical low-rank approximations [118, 133, 141, 142].

For kinetic equations the dynamical low-rank approach offers a range of advantages, not only strictly related to storage and computational costs. For example, filamented structures in velocity space can be resolved accurately [87]. Moreover, it is known that in the linear regime [82] and for certain fluid [76, 80] and diffusive limits [68, 79] the solution has a low-rank structure, hence it is natural to use a low-rank approximation. Recently, a dynamical algorithm that is conservative from first principle has been constructed [81]. Because of these advances, dynamical low-rank approximations have received significant interest lately, and methods for problems from plasma physics [82, 87, 103, 121], radiation transport [79, 124, 154, 155], and uncertainty quantification for hyperbolic problems [123] have been proposed. These schemes have the potential to enable the 6D simulation of such systems on small clusters or even desktop computers.

While mature software packages exist for solving kinetic problems using both particle methods (e.g., [31, 182]) and methods that directly discretize the phase space on a grid (e.g., [77, 101, 161, 186]), no such software frameworks exist for dynamical low-rank algorithms. However, the need in the latter case is arguably even more critical, as the resulting evolution equations are usually somewhat more complex than the original model.

The purpose of this work is to present six-dimensional simulations of the Vlasov–Poisson equations using the framework `Ensign`[1], which facilitates the easy and efficient implementation of dynamical low-rank algorithms for kinetic equations (both on multi-core CPU and GPU based systems). In particular, we will employ a newly designed second order projector-splitting dynamical low-rank algorithm, which is based on a CFL-free (but still fully explicit) exponential integrator that uses Fourier spectral methods to compute the action of certain matrix functions. We emphasize, however, that in principle the software framework is able to support other dynamical low-rank techniques (such as the unconventional integrator [55]), and is completely flexible with regard to the specific space and time discretizations employed. In fact, almost any third party library suitable for a given problem could be used alongside `Ensign` in an implementation.

The remaining part of this chapter is structured as follows: in Section 5.2 we describe the general structure of a projector-splitting dynamical low-rank integrator for the Vlasov–Poisson equations. In Section 5.3 we then semi-discretize (discrete in space and continuous in time) the equations of motion resulting from the projector-splitting integrator and write them in a matrix formulation. The proposed integrator is described in detail in Section 5.4. In Section 5.5 we provide a big picture overview of the `Ensign` software framework and how it can be used to implement a dynamical low-rank algorithm. We present some numerical results in Section 5.6 and discuss the performance of the dynamical low-rank algorithms on CPUs and GPUs in Section 5.7. Finally, we conclude in Section 5.8.

5.2 Low-rank approximation for the Vlasov–Poisson equations

In this chapter we consider the Vlasov–Poisson equations in dimensionless form given by

$$\begin{cases} \partial_t f(t,x,v) + v \cdot \nabla_x f(t,x,v) - E(f)(t,x) \cdot \nabla_v f(t,x,v) = 0, \\ E(f)(t,x) = -\nabla_x \phi(t,x), \\ -\Delta\phi(t,x) = \rho(f)(t,x) + 1, \quad \rho(f)(t,x) = -\displaystyle\int_{\Omega_v} f(t,x,v)\,dv, \end{cases} \tag{5.1}$$

where $f(t,x,v)$ represents the particle-density function of the species under consideration, $t \in \mathbb{R}_0^+$ is the time variable, $x \in \Omega_x \subset \mathbb{R}^d$ refers to the space variable, $v \in \Omega_v \subset \mathbb{R}^d$ is the velocity variable and $d = 1, 2, 3$. Depending on the physical phenomenon under study, system (5.1) is completed with appropriate boundary and initial conditions.

[1]Publicly available at `https://github.com/leinkemmer/Ensign` under the MIT license.

We now describe how to obtain the dynamical low-rank approximation of the Vlasov–Poisson system (5.1). A reader not familiar with these concepts can find more details, for instance, in [82]. The computational domain is denoted by $\Omega = \Omega_x \times \Omega_v$. Then, instead of directly solving system (5.1), we look for an approximation of the particle-density function $f(t, x, v)$ that, for fixed t, lies in the rank-r manifold

$$\begin{aligned}\mathcal{M} = \Big\{ g(x,v) \in L^2(\Omega) : g(x,v) = \sum_{i,j} X_i(x) S_{ij} V_j(v) \text{ with invertible } S = (S_{ij}) \in \mathbb{R}^{r\times r},\\ X_i \in L^2(\Omega_x), V_j \in L^2(\Omega_v) \text{ with } \langle X_i, X_k \rangle_x = \delta_{ik}, \langle V_j, V_\ell \rangle_v = \delta_{j\ell} \Big\}\end{aligned}$$

with corresponding tangent space

$$\begin{aligned}\mathcal{T}_f\mathcal{M} = \Big\{ \dot g(x,v) \in L^2(\Omega) : \dot g(x,v) = \sum_{i,j} \big(X_i(x) \dot S_{ij} V_j(v) + \dot X_i(x) S_{ij} V_j(v) + X_i(x) S_{ij} \dot V_j(v) \big),\\ \text{with } \dot S \in \mathbb{R}^{r\times r}, \dot X_i \in L^2(\Omega_x), \dot V_j \in L^2(\Omega_v)\\ \text{and } \langle X_i, \dot X_k \rangle_x = 0, \langle V_j, \dot V_\ell \rangle_v = 0 \Big\}.\end{aligned}$$

Here $\langle\cdot,\cdot\rangle_x$ and $\langle\cdot,\cdot\rangle_v$ denote the standard inner products on $L^2(\Omega_x)$ and $L^2(\Omega_v)$, respectively, and we employ Newton's notation for the time derivative. Moreover, all indexes run from 1 to r and, for simplicity of presentation, we drop these limits from the notation.

To obtain the dynamical low-rank approximation of the Vlasov–Poisson system (5.1) we need to compute

$$\partial_t f(t,x,v) = -P(f)(v \cdot \nabla_x f(t,x,v) - E(f) \cdot \nabla_v f(t,x,v)), \tag{5.2}$$

where $P(f)$ is the orthogonal projector onto the tangent space $\mathcal{T}_f\mathcal{M}$. For simplicity of notation, we will keep using the symbol f to denote the low-rank approximation to the particle-density. The orthogonal projection of a generic function g can be written as

$$P(f)g = P_{\overline{V}} g - P_{\overline{V}} P_{\overline{X}} g + P_{\overline{X}} g,$$

where $P_{\overline{X}}$ and $P_{\overline{V}}$ are orthogonal projectors onto the spaces spanned by the functions X_i and V_j, respectively. This formulation suggests a three-term splitting of equation (5.2) with subflows (in the sense of differential equations in the context of splitting schemes) given by

$$\partial_t f_1(t,x,v) = -P_{\overline{V}}(v \cdot \nabla_x f_1(t,x,v) - E(f_1) \cdot \nabla_v f_1(t,x,v)), \tag{5.3}$$

$$\partial_t f_2(t,x,v) = P_{\overline{V}} P_{\overline{X}}(v \cdot \nabla_x f_2(t,x,v) - E(f_2) \cdot \nabla_v f_2(t,x,v)), \tag{5.4}$$

$$\partial_t f_3(t,x,v) = -P_{\overline{X}}(v \cdot \nabla_x f_3(t,x,v) - E(f_3) \cdot \nabla_v f_3(t,x,v)). \tag{5.5}$$

This is the projector-splitting integrator that has been first proposed in [134].

By explicitly applying the projector $P_{\overline{V}}$ on equation (5.3), we can see that solving it is equivalent to

$$\begin{cases} \partial_t K_j(t,x) = -\sum_\ell c^1_{j\ell} \cdot \nabla_x K_\ell(t,x) + \sum_\ell c^2_{j\ell} \cdot E(K)(t,x) K_\ell(t,x), \\ V_j(t,v) = \tilde V_j(v), \end{cases} \tag{5.6}$$

where the approximate particle-density function is written as

$$f_1(t,x,v) = \sum_j K_j(t,x) V_j(t,v), \quad K_j(t,x) = \sum_i X_i(t,x) S_{ij}(t),$$

and

$$c^1_{j\ell} = \int_{\Omega_v} v \tilde V_j(v) \tilde V_\ell(v) dv, \quad c^2_{j\ell} = \int_{\Omega_v} \tilde V_j(v) \nabla_v \tilde V_\ell(v) dv.$$

We refer to [82] for a more detailed derivation and a thorough explanation of the underlying mathematical structure. Equation (5.6) is usually referred to as the *K step* of the low-rank projector-splitting algorithm. Note that the velocity dependent low-rank factors, i.e., V_j, do not change during this step. This means that we can use their values at the beginning of the step, i.e., $\tilde{V}_j$, in all computations of the K step.

By applying both projectors $P_{\overline{V}}$ and $P_{\overline{X}}$, the second subflow (i.e., the one related to equation (5.4)) can be obtained by

$$\begin{cases} \dot{S}_{ij}(t) = \sum_{k,\ell}(c^1_{j\ell} \cdot d^2_{ik} - c^2_{j\ell} \cdot d^1_{ik}[E(S)])S_{k\ell}(t), \\ X_i(t,x) = \breve{X}_i(x), \\ V_j(t,v) = \tilde{V}_j(v), \end{cases} \tag{5.7}$$

where

$$d^1_{ik}[E(S)] = \int_{\Omega_x} \breve{X}_i(x)E(S)\breve{X}_k(x)dx, \quad d^2_{ik} = \int_{\Omega_x} \breve{X}_i(x)\nabla_x\breve{X}_k(x)dx.$$

We refer to equation (5.7) as the *S step* of the low-rank projector-splitting algorithm. Note that neither X_i nor V_j change during this step, i.e., their values are fixed to the ones at the beginning of the step ($\breve{X}_i$ and $\tilde{V}_j$, respectively).

Finally, we can demonstrate that solving equation (5.5) is equivalent to

$$\begin{cases} \partial_t L_i(t,v) = \sum_k d^1_{ik}[E(L)] \cdot \nabla_v L_k(t,v) - \sum_k (d^2_{ik} \cdot v)L_k(t,v), \\ X_i(t,x) = \breve{X}_i(x), \end{cases} \tag{5.8}$$

with the approximate particle-density function written as

$$f_3(t,x,v) = \sum_i X_i(t,x)L_i(t,v), \quad L_i(t,v) = \sum_j S_{ij}(t)V_j(t,v).$$

This step of the low-rank projector-splitting algorithm is referred to as the *L step.* Note that the X_i do not change during this step, i.e., their values are set to $\breve{X}_i$.

Concerning the equations of the electric field, they can be written in terms of the low-rank factors $X_i(t,x)$, $S_{ij}(t)$ and $V_j(t,v)$ as well. Indeed, depending on the need, we can express the charge density $\rho(f)(t,x)$ as

$$\begin{aligned} \rho(K)(t,x) &= -\sum_j K_j(t,x)\rho(\tilde{V}_j(v)), & \rho(\tilde{V}_j(v)) &= \int_{\Omega_v} \tilde{V}_j(v)\,dv, \\ \rho(S)(t,x) &= -\sum_{i,j} \breve{X}_i(x)S_{ij}(t)\rho(\tilde{V}_j(v)), & & \\ \rho(L)(t,x) &= -\sum_i \breve{X}_i(x)\rho(L_i(t,v)), & \rho(L_i(t,v)) &= \int_{\Omega_v} L_i(t,v)\,dv, \end{aligned}$$

where the relevant quantities are defined above.

Finally, the approximate solution to the Vlasov–Poisson equations, i.e., system (5.1), is obtained by combining the partial solutions of the K, S and L steps in a splitting fashion. In the easiest setting, the first order Lie splitting scheme concatenates the three subflows in sequence, see Section 5.3.1. We will also outline a second order scheme based on Strang splitting in Section 5.3.2.

5.3 Matrix formulation of the semi-discrete algorithm

So far we have considered the low-rank approximation in a continuous framework. However, to perform calculations on a computer we have to discretize the equations for the K, S, and L steps. In this

section, we assume that a space discretization has been chosen. The goal is then to collect the degrees of freedom into matrices and vectors and write the resulting equations as operations on those objects. The implementation using the software framework `Ensign`, described later in Section 5.5, is also based on this formulation.

We consider n_{x_k} discretization points for the space variable x_k, $k = 1, \dots, d$, and n_{v_k} discretization points for the velocity variable v_k, $k = 1, \dots, d$. The total numbers of degrees of freedom are denoted as $N_x = n_{x_1} \cdots n_{x_d}$ and $N_v = n_{v_1} \cdots n_{v_d}$ for space and velocity, respectively. Then, for a fixed time t, we define $X = [X_1, \dots, X_r] \in \mathbb{R}^{N_x \times r}$ to be the matrix having as columns the evaluation of the low-rank factors X_i at the chosen spatial grid. Clearly, the resulting matrix entries depend on the discretization performed and on the ordering of the grid points. Similarly, we consider $V = [V_1, \dots, V_r] \in \mathbb{R}^{N_v \times r}$ to be the matrix having as columns the evaluation of the low-rank factors V_j at the velocity grid. The coupling coefficients S_{ij} are collected in the matrix $S \in \mathbb{R}^{r \times r}$. Hence, we can write the evolution equation for the K step (5.6) in matrix formulation as follows

$$\partial_t K(t) = -\sum_{i=1}^{d} \overline{\partial}_{x_i} K(t) C_{1,v_i}^{\mathsf{T}} + \sum_{i=1}^{d} \mathrm{diag}(E_{x_i}(K(t))) K(t) C_{2,v_i}^{\mathsf{T}}, \tag{5.9}$$

where

$$\begin{aligned} K(t) &= X(t)S(t), \qquad\qquad K(t) \in \mathbb{R}^{N_x \times r}, \\ C_{1,v_i} &= \tilde{V}^{\mathsf{T}} \mathrm{diag}(\omega_{1,v_i}^{v_i}) \tilde{V} \in \mathbb{R}^{r \times r}, \\ C_{2,v_i} &= \tilde{V}^{\mathsf{T}} \mathrm{diag}(\omega_{2,v_i}) \overline{\partial}_{v_i} \tilde{V} \in \mathbb{R}^{r \times r}, \end{aligned}$$

and $\omega_{1,v_i}^{v_i}$ and ω_{2,v_i} are suitable quadrature weights. The ith component of the electric field has been denoted by $E_{x_i}(K(t)) \in \mathbb{R}^{N_x}$. In addition, we have used $\overline{\partial}_{x_i}$ to denote the discretization of the spatial derivative operator. While this operator can be represented as a matrix, in many cases it is more efficient to directly compute its application to $K(t)$ (e.g., in a stencil code or by using FFTs). We also note that in order to compute the coefficients C_{1,v_i} and C_{2,v_i} it would (obviously) be very inefficient to form the diagonal matrix. Instead, the framework `Ensign` provides the function `coeff` that takes the matrices as well as a vector of weights as input and computes the corresponding quadrature (see Section 5.5 for more details).

For the evolution equation of the S step (5.7) we obtain

$$\dot{S}(t) = \sum_{i=1}^{d} D_{2,x_i} S(t) C_{1,v_i}^{\mathsf{T}} - \sum_{i=1}^{d} D_{1,x_i}[E(S(t))] S(t) C_{2,v_i}^{\mathsf{T}}, \tag{5.10}$$

where

$$\begin{aligned} D_{1,x_i}[E(S(t))] &= \check{X}^{\mathsf{T}} \mathrm{diag}(\omega_{1,x_i}^{E}) \check{X} \in \mathbb{R}^{r \times r}, \\ D_{2,x_i} &= \check{X}^{\mathsf{T}} \mathrm{diag}(\omega_{2,x_i}) \overline{\partial}_{x_i} \check{X} \in \mathbb{R}^{r \times r}, \end{aligned}$$

and again ω_{1,x_i}^{E} and ω_{2,x_i} are suitable quadrature weights.

Finally, for the evolution equation of the L step (5.8) we have

$$\partial_t L(t) = \sum_{i=1}^{d} \overline{\partial}_{v_i} L(t) D_{1,x_i}^{\mathsf{T}} - \sum_{i=1}^{d} \mathrm{diag}(\overline{v_i}) L(t) D_{2,x_i}^{\mathsf{T}}, \tag{5.11}$$

where

$$L(t) = V(t)S(t)^{\mathsf{T}}, \quad L(t) \in \mathbb{R}^{N_v \times r},$$

and $\overline{v_i} \in \mathbb{R}^{N_v}$ is the vector with the positions of the grid points in velocity space.

In matrix formulation, the equations for the electric field are given by

$$\begin{aligned} (E_{x_1}(f)(t), \dots, E_{x_d}(f)(t)) &= -\overline{\nabla}_x \Phi(f)(t), \quad E_{x_i}(f)(t) \in \mathbb{R}^{N_x}, \quad \Phi(f)(t) \in \mathbb{R}^{N_x}, \\ -\overline{\Delta}\Phi(f)(t) &= P(f)(t) + 1, \qquad P(f)(t) \in \mathbb{R}^{N_x}, \end{aligned}$$

where the discretized charge density $P(f)(t)$ can be computed in terms of the low-rank factors, depending on the need, as

$$P(K)(t) = -K(t)\tilde{P}, \qquad \tilde{P} = \tilde{V}^{\mathsf{T}}\omega_v \in \mathbb{R}^r, \qquad \omega_v \in \mathbb{R}^{N_v}, \tag{5.12}$$

$$P(S)(t) = -\check{X}S(t)\tilde{P},$$

$$P(L)(t) = -\check{X}\overline{P}(L(t)), \qquad \overline{P}(L(t)) = L(t)^{\mathsf{T}}\omega_v \in \mathbb{R}^r. \tag{5.13}$$

Here ω_v is a vector which collects suitable quadrature weights.

Let us also note that the approximation of the particle-density function can be recovered at any moment from the low-rank factors by computing

$$F(t) = X(t)S(t)V(t)^{\mathsf{T}} \in \mathbb{R}^{N_x \times N_v},$$

but clearly this is not needed for the low-rank projector-splitting algorithm, and doing so would be extremely costly.

5.3.1 Order 1 low-rank projector-splitting algorithm

The subflows corresponding to the K step, the S step, and the L step are then combined by a splitting scheme in order to recover an approximation of the particle-density function. For a first order method it is clearly sufficient to consider a Lie–Trotter splitting algorithm. A detailed description of the resulting scheme is given in Algorithm 1.

Algorithm 1: First order dynamical low-rank integrator for the Vlasov–Poisson equations (5.1).

Input: X^0, S^0, V^0 such that $f(0,x,v) \approx \sum_{i,j} X_i^0(x) S_{ij}^0 V_j^0(v)$, time step size τ

Output: X^1, S^3, V^1 such that $f(\tau,x,v) \approx \sum_{i,j} X_i^1(x) S_{ij}^3 V_j^1(v)$

1. Compute C_{1,v_i} and C_{2,v_i}, $i = 1,\ldots,d$, using V^0;
2. Compute K^0 using X^0 and S^0;
3. Compute the electric field E^0 with equation (5.12) using K^0 and V^0;
4. Solve equation (5.9) with initial value K^0 and E^0 up to time τ to obtain K^1;
5. Perform a QR decomposition of K^1 to obtain X^1 and S^1;
6. Compute D_{1,x_i} and D_{2,x_i}, $i = 1,\ldots,d$, using E^0 and X^1;
7. Solve equation (5.10) with initial value S^1 and E^0 up to time τ to obtain S^2;
8. Compute L^0 using V^0 and S^2;
9. Solve equation (5.11) with initial value L^0 and E^0 up to time τ to obtain L^1;
10. Perform a QR decomposition of L^1 to obtain V^1 and S^3;

Remark that the computation of the electric field is performed only once at the beginning of the time step. This is not a restriction in the context of a Lie–Trotter splitting, as fixing the electric field at each time step still results in a first order approximation. In principle, any numerical method can be employed to integrate in time equations (5.9)–(5.11). We will discuss a proposal, based on the choice of a spectral phase space discretization, in Section 5.4. Moreover, after performing the K and the L steps, in order to proceed with the algorithm we need to recover the orthonormal functions X_i, V_j and the coupling coefficients S_{ij}. This can be accomplished, for example, by a QR or an SVD decomposition. Finally, we note that the overall storage and computational costs of the dynamical low-rank algorithm, for $n_{x_i} = n_{v_i} = n$, scale as $\mathcal{O}(rn^d)$ and $\mathcal{O}(r^2n^d)$, respectively. For more details we refer the reader to [82].

5.3.2 Order 2 low-rank projector-splitting algorithm

A straightforward generalization to a second order integrator by employing a Strang splitting procedure instead of a Lie–Trotter one is not possible. Indeed, if we freeze the electric field at the beginning of the

time step, we still end up with a first order algorithm. To overcome this problem, an almost symmetric Strang splitting scheme is proposed in [82]; however, in order to achieve full second order, that algorithm requires several updates of the electric field, which in turn translates into high computational effort. We propose here a slightly different strategy to obtain a second order scheme, listed in detail in Algorithm 2. The underlying idea is that we compute an approximation of the electric field at time $\tau/2$ of (local) second order by means of Algorithm 1. Then, we restart the integration with a classic Strang splitting scheme employing as constant electric field the approximation at the half step. Mathematically, this can still be analyzed as an almost symmetric splitting scheme (see [84, 85]), and as for Algorithm 1 the storage and computational costs scale as $\mathcal{O}(rn^d)$ and $\mathcal{O}(r^2n^d)$, respectively.

Algorithm 2: Second order dynamical low-rank integrator for the Vlasov–Poisson equations (5.1).

Input: X^0, S^0, V^0 such that $f(0,x,v) \approx \sum_{i,j} X_i^0(x) S_{ij}^0 V_j^0(v)$, time step size τ

Output: X^3, S^7, V^1 such that $f(\tau,x,v) \approx \sum_{i,j} X_i^3(x) S_{ij}^7 V_j^1(v)$

1 Perform steps 1–9 of Algorithm 1 with time step size $\tau/2$;
2 Compute the electric field $E^{1/2}$ with equation (5.13) using X^1 and L^1 from step 1;
3 Solve equation (5.9) with initial value K^0 and $E^{1/2}$ up to time $\tau/2$ to obtain K^2;
4 Perform a QR decomposition of K^2 to obtain X^2 and S^3;
5 Compute D_{1,x_i} and D_{2,x_i}, $i = 1, \dots, d$, using $E^{1/2}$ and X^2;
6 Solve equation (5.10) with initial value S^3 and $E^{1/2}$ up to time $\tau/2$ to obtain S^4;
7 Compute L^1 using V^0 and S^4;
8 Solve equation (5.11) with initial value L^1 and $E^{1/2}$ up to time τ to obtain L^2;
9 Perform a QR decomposition of L^2 to obtain V^1 and S^5;
10 Compute C_{1,v_i} and C_{2,v_i}, $i = 1, \dots, d$, using V^1;
11 Solve equation (5.10) with initial value S^5 and $E^{1/2}$ up to time $\tau/2$ to obtain S^6;
12 Compute K^3 using X^2 and S^6;
13 Solve equation (5.9) with initial value K^3 and $E^{1/2}$ up to time $\tau/2$ to obtain K^4;
14 Perform a QR decomposition of K^4 to obtain X^3 and S^7;

5.4 Time and space discretization of K, S and L steps

As already mentioned in Section 5.3, in principle any numerical scheme can be used to integrate in time equations (5.9)–(5.11). However, depending on the selected phase space discretization, some choices can be more adequate than others. In particular, we describe here in detail an exponential integrator based strategy which uses a Fourier spectral discretization for both space and velocity variables. The method converges rapidly in space and velocity (owing to the spectral discretization) and despite being fully explicit does not suffer from a CFL induced step size restriction in time.

Let us consider the K step (5.9). As we are interested in performing 6D simulations, for clarity of exposition we will only consider the case $d = 3$ here. The (simpler) cases $d = 1$ and $d = 2$ can be treated similarly. Then, by performing a further splitting of the K step we obtain the following three equations

$$\partial_t K_1(t) = -\overline{\partial}_{x_1} K_1(t) C_{1,v_1}^{\mathsf{T}}, \tag{5.14}$$

$$\partial_t K_2(t) = -\overline{\partial}_{x_2} K_2(t) C_{1,v_2}^{\mathsf{T}}, \tag{5.15}$$

$$\partial_t K_3(t) = -\overline{\partial}_{x_3} K_3(t) C_{1,v_3}^{\mathsf{T}} + \sum_{i=1}^{3} \operatorname{diag}(E_{x_i}) K_3(t) C_{2,v_i}^{\mathsf{T}}. \tag{5.16}$$

Note that, in principle, the summation term related to the electric field could be put in any of the three equations (5.14)–(5.16), without any substantial change. On the other hand, in a general setting, there

would not be advantages in treating this term in a separate flow, because there is no exact solution in Fourier space and more splitting error would be generated.

Applying then a Fourier transform in the x_i variables (denoted by $\mathcal{F}_{x_i}$) we obtain

$$\begin{aligned}
\partial_t \hat{K}_1(t) &= -D^{\mathcal{F}}_{x_1} \hat{K}_1(t) C^{\mathsf{T}}_{1,v_1}, \\
\partial_t \hat{K}_2(t) &= -D^{\mathcal{F}}_{x_2} \hat{K}_2(t) C^{\mathsf{T}}_{1,v_2}, \\
\partial_t \hat{K}_3(t) &= -D^{\mathcal{F}}_{x_3} \hat{K}_3(t) C^{\mathsf{T}}_{1,v_3} + \sum_{i=1}^{3} \mathcal{F}_{x_3}(\mathrm{diag}(E_{x_i}) K_3(t)) C^{\mathsf{T}}_{2,v_i},
\end{aligned}$$

where $\hat{K}_i(t) = \mathcal{F}_{x_i}(K_i(t))$ and $D^{\mathcal{F}}_{x_i}$ is a diagonal matrix containing the coefficients stemming from the differential operator $\overline{\partial}_{x_i}$ in Fourier space. Then, as the matrices C^{T}_{1,v_i} are symmetric by construction, it is possible to diagonalize them so that $C^{\mathsf{T}}_{1,v_i} = T_{v_i} D_{v_i} T^{\mathsf{T}}_{v_i}$. We note that this operation is computationally cheap as the matrices involved have only size $r \times r$. By performing the substitution $\hat{M}_i(t) = \hat{K}_i(t) T_{v_i}$ we have

$$\partial_t \hat{M}_1(t) = -D^{\mathcal{F}}_{x_1} \hat{M}_1(t) D_{v_1}, \tag{5.17}$$
$$\partial_t \hat{M}_2(t) = -D^{\mathcal{F}}_{x_2} \hat{M}_2(t) D_{v_2}, \tag{5.18}$$
$$\partial_t \hat{M}_3(t) = -D^{\mathcal{F}}_{x_3} \hat{M}_3(t) D_{v_3} + \sum_{i=1}^{3} \mathcal{F}_{x_3}(\mathrm{diag}(E_i) K_3(t)) C^{\mathsf{T}}_{2,v_i} T_{v_3}. \tag{5.19}$$

At this point, equations (5.17) and (5.18) can be solved exactly in time by means of independent pointwise operations, while equation (5.19) can be solved efficiently by means of a first or second order exponential Runge–Kutta method (see [112] for a survey), again just using independent pointwise operations. We choose to use exponential integrators in the time evolution of this step because in this way we remove any CFL-like restriction of the step size coming from the stiffness of the spatial derivative. Moreover, as everything is written in terms of independent pointwise operations, the computation of the single flow can be performed completely in parallel.

Finally, coming back to the original variables $\hat{K}_i$ and performing an inverse Fourier transform, we obtain approximate solutions for the equations involved. Embedding this procedure in a splitting context returns the desired approximation of the evolution equation for the K step. In particular, for the first order method described in Algorithm 1 it is enough to perform a Lie–Trotter splitting, while a (classical) Strang splitting procedure is needed for the second order method presented in Algorithm 2.

Concerning the integration of the evolution equation of the L step (5.11), similar considerations apply. Finally, concerning the S step (5.10), it is an $r \times r$ problem and there is no source of stiffness in it. Hence, we perform its time integration by means of the classical explicit fourth order Runge–Kutta scheme RK4. In principle it would be sufficient to use a first order scheme, in the context of Algorithm 1, and a second order scheme, in the context of Algorithm 2. However, since the S step is cheap, we can use a higher order approximation at negligible additional computational cost and reduce the error term associated with integrating S from the numerical scheme.

5.5 The `Ensign` framework and implementation

As presented in Sections 5.2–5.4, in the context of dynamical low-rank algorithms a function $f(t,x,v)$ is approximated as $f(t,x,v) \approx \sum_{i,j} X_i(t,x) S_{ij}(t) V_j(t,v)$, where the indexes i and j run from 1 to r and r is the chosen approximation rank. The quantities $X_i(t,x)$, $S_{ij}(t)$ and $V_j(t,v)$ constitute the so called low-rank factors and, after discretization in x and v, they can be expressed as matrices, allowing us to write the scheme in matrix formulation. Therefore, independently of the specific case under consideration, the common key point for an efficient implementation of dynamical low-rank algorithms for kinetic equations, both on CPU and GPU based systems, is the fast computation of operations on matrices and vectors (e.g., matrix-matrix and matrix-vector products, certain pointwise operations, etc). For

every modern computer architecture, we have at our disposal heavily optimized routines to perform such operations, usually referred to as BLAS. For example, Intel MKL [114] and OpenBLAS [189] are available for CPU based systems while we have cuBLAS [58] and MAGMA [146] for NVIDIA GPUs. Among their many features, all these libraries are equipped with multithreaded versions of BLAS routines. When possible we use these libraries heavily. However, we note that there are certain operations that we need to perform in the context of a low-rank algorithm that are not part of BLAS.

The main idea behind `Ensign` is to provide the user a collection of structures and functions in order to compute and manipulate *easily* the arising quantities in dynamical low-rank algorithms. Let us note that the goal here is to provide primitives that allow the user to implement dynamical low-rank approximations on a high-level. Thus, we are concerned with relevant data structures for dynamical low-rank algorithms, for computing the coefficients that appear in the approximations (which can have, in general, two or more indices and also spatial dependences), for performing certain operations such as initialization, addition or truncation on low-rank approximations, orthogonalization with respect to arbitrary inner products, and writing such low-rank approximations to disk. This could then be complemented by the user of `Ensign` with libraries that perform spatial and temporal discretization. The framework is written in C++ programming language and uses CUDA internally for the GPU code.

We now illustrate some of its features with the aid of the first order projector-splitting dynamical low-rank algorithm presented in Section 5.3.1, assuming an underlying six-dimensional phase space. To this aim, in Figure 5.1 we translate in source code some lines of the pseudocode of Algorithm 1, and in Figure 5.2 we show how it is possible to initialize the low-rank factors. We can immediately note how the quantities arising from the mathematical equations can be naturally translated into the structures and functions provided by the framework. In particular, in terms of data storage we employ a structure `multi_array` which lets us easily define vectors (`w1v1`, for example) and matrices (`C1v1`, for example) both in CPU and in GPU memory, depending on the need. This structure is also enriched with some user-friendly functions and operators which are useful to perform basic operations between `multi_array`s, such as sum, difference, multiplication with a scalar, and to transfer data from/to CPU/GPU memory (the latter can be simply done by assignment, as commented in Figure 5.2). For convenience of the user, we provide also a structure `lr2`, which contains three 2D `multi_array`s (`X`, `S` and `V`) that reflect the evaluation of the low-rank factors on a discretized grid. In particular, the degrees of freedom are linearized so that each column corresponds to the discretized version of a single low-rank factor. While the specific example of Figure 5.2 is presented in the context of a uniform space discretization, any other kind of discretization (with an arbitrary ordering of the nodes) would work in a straightforward way as well.

We use C++ templates in order to abstract the underlying architecture on which the code is run. Depending on whether a function is called with arguments that reside on the CPU or on the GPU, an efficient implementation suitable for that hardware architecture is selected. Indeed, whether the code runs on the CPU or on the GPU is never explicitly specified in the code example in Figure 5.1, because it is automatically detected from the storage location of the input arguments of the functions. Thus, the algorithmic part of the implementation is completely independent of what computer hardware the code is eventually run on.

Concerning BLAS operations, `Ensign` provides the structure `blas_ops` which contains wrappers for matrix multiplications (e.g., `matmul` and `matmul_transb`) and handles needed to call properly these routines on the GPU. Then, the framework is equipped with a function to compute the C and D integral coefficients, namely `coeff`. Depending on the input matrices and the vector of weights, the function computes the corresponding matrix of coefficients. As an example, given the 1D quadrature weight `multi_array w1v1` and the 2D `multi_array lr_st.V`, the coefficient matrix C_{1,v_1} can be computed using the command in line 8 of the code in Figure 5.1. Also, we can find in the framework the structure `gram_schmidt`, which contains a function to compute the QR decomposition of a matrix with a generic inner product. It is based on a modified Gram–Schmidt algorithm written as much as possible in matrix formulation, so that again the internal computations are automatically performed in parallel by means of calls to appropriate BLAS. Modified Gram-Schmidt is used here because it is easy to parallelize and can operate purely in terms of inner products, even if the associated degrees of freedom are not stored

```
// Declaration and initialization
/* See Figure 2 */
gram_schmidt gs(&blas);
multi_array<double,1> w1v1(stloc::host); // stloc::device if on GPU
multi_array<double,2> C1v1(stloc::host);
/* ... */
// Line 1: Compute C coefficients
coeff(lr_st.V, lr_st.V, w1v1, C1v1, blas);
coeff(lr_st.V, lr_st.V, w1v2, C1v2, blas);
coeff(lr_st.V, lr_st.V, w1v3, C1v3, blas);
coeff(lr_st.V, dV0_v1, w2v1, C2v1, blas);
coeff(lr_st.V, dV0_v2, w2v2, C2v2, blas);
coeff(lr_st.V, dV0_v3, w2v3, C2v3, blas);
// Line 2: Compute K0
tmpX = lr_st.X;
blas.matmul(tmpX, lr_st.S, lr_st.X);
// Line 3: Compute electric field
/* ... */
// Line 4: Solve K step
/* ... */
// Line 5: Perform QR decomposition
gs(lr_st.X, lr_st.S, ip_xx);
/* ... */
// Line 6: Compute D coefficients
coeff(lr_st.X, lr_st.X, wE1x1, D1x1, blas);
coeff(lr_st.X, lr_st.X, wE1x2, D1x2, blas);
coeff(lr_st.X, lr_st.X, wE1x3, D1x3, blas);
coeff(lr_st.X, dX1_x1, w2x1, D2x1, blas);
coeff(lr_st.X, dX1_x2, w2x2, D2x2, blas);
coeff(lr_st.X, dX1_x3, w2x3, D2x3, blas);
// Line 7: Solve S step
/* ... */
// Line 8: Compute L0
tmpV = lr_st.V;
blas.matmul_transb(tmpV, lr_st.S, lr_st.V);
// Line 9: Solve L step
/* ... */
// Line 10: Perform QR decomposition
gs(lr_st.V, lr_st.S, ip_vv);
transpose_inplace(lr_st.S);
```

Figure 5.1: Sketch of a C++ implementation of Algorithm 1 using the `Ensign` framework. To perform computation on the GPU, it is enough to use `d_lr_st` instead of `lr_st`, and to declare the relevant `multi_arrays` with `stloc::device`. The syntax `/*...*/` indicates code not reported for simplicity of exposition.

directly as a vector. The latter is important in the case of hierarchical low-rank approximations such as the those considered in [82]. For example, given the 2D `multi_arrays` `lr_st.X` and `lr_st.S` and the inner product `ip_xx`, the call to perform the QR decomposition of `lr_st.X` is given in line 22 of Figure 5.1.

Finally, we want to emphasize that while we have illustrated the software framework for a simple projector-splitting based dynamical low-rank integrator, every effort has been made in designing `Ensign` to allow also the implementation of other dynamical low-rank integrators, such as the recently proposed unconventional integrator [55] or the conservative dynamical low-rank integrator [81]. Moreover, we

```
typedef ptrdiff_t Index;
array<Index,3> N_xx, N_vv;
array<double,3> lim_xx, lim_vv, h_xx, h_vv;
Index r;
blas_ops blas;
vector<const double*> X0, V0;

// Initialize N_xx, lim_xx, h_xx, N_vv, lim_vv, h_vv and r on CPU
/* ... */

Index N_xx_m = N_xx[0]*N_xx[1]*N_xx[2];
Index N_xx_m = N_vv[0]*N_vv[1]*N_vv[2];

// Define inner products that are used in the algorithm
auto ip_xx = inner_product_from_const_weight(h_xx[0]*h_xx[1]*h_xx[2], N_xx_m);
auto ip_vv = inner_product_from_const_weight(h_vv[0]*h_vv[1]*h_vv[2], N_vv_m);

// Initialize the low-rank structure for the initial value
// (the initial value is given by X0 and V0 and can usually
// be easily determined from the problem)
// To illustrate data movement (see below) we perform the
// initialization on the CPU and then transfer the result
// to the GPU.

lr2<double> lr_st(r,{N_xx_m,N_vv_m});

// Set up X0 and V0
/* ... */

initialize(lr_st, X0, V0, ip_xx, ip_vv, blas);

// Assignment of two lr2 or multi_arrays copies from CPU to GPU
lr2<double> d_lr_st(r,{N_xx_m,N_vv_m},stloc::device); // on GPU
d_lr_st = lr_st;
```

Figure 5.2: Sketch of a C++ implementation for the initialization of low-rank factors using the `Ensign` framework. `N_xx`, `lim_xx` and `h_xx` are arrays which contain the number of discretization points, the left extremes of the space domain and the grid spacing for each direction, respectively (similarly for `N_vv`, `lim_vv` and `h_vv`, which are related to the velocity domain). `r` is the approximation rank, while `lr_st` and `d_lr_st` are structures which contain 2D `multi_arrays` that reside on the CPU and on the GPU, respectively. The syntax `/*...*/` indicates code not reported for simplicity of exposition.

again point out that the user is completely free to choose any space or time discretization appropriate to the problem. The only requirement in terms of space discretization is that the degrees of freedom of the low-rank factors have to be collected in suitable matrices by means of an index linearization. This is certainly possible for all the commonly used space discretization strategies.

5.6 Numerical experiments

In this section we will present some numerical results and validate the implementations of the algorithms described in Section 5.3. The developed code solves the 6D Vlasov–Poisson equations (5.1) and uses the framework `Ensign`. All the experiments in this section have been performed in double precision arithmetic with the aid of a single NVIDIA Tesla A100 card (theoretical peak memory bandwidth of

1555 GB/s and peak floating point processing power for double precision of 9.7 TFlops), equipped with 40 GB of RAM.

5.6.1 Orders of convergence

First of all, we check the implementation of the low-rank projector-splitting algorithms by computing numerically the order of convergence of the methods. For this purpose, we consider a 6D linear Landau problem posed on the domain $\Omega = (0, 4\pi)^3 \times (-6, 6)^3$ with an initial particle-density given by

$$f_0(x_1, x_2, x_3, v_1, v_2, v_3) = \frac{1}{\sqrt{(2\pi)^3}} e^{-(v_1^2+v_2^2+v_3^2)/2}(1+\alpha_1 \cos(\kappa_1 x_1)+\alpha_2 \cos(\kappa_2 x_2)+\alpha_3 \cos(\kappa_3 x_3)). \quad (5.20)$$

The parameters are set to $\alpha_1 = \alpha_2 = \alpha_3 = 10^{-2}$ and $\kappa_1 = \kappa_2 = \kappa_3 = \frac{1}{2}$. We consider the problem with periodic boundary conditions in all directions and integrate it up to final time $T = 1$. Concerning the space discretization, we take 32^3 as number of discretization points for both space and velocity variables. The rank of the solution is fixed to $r = 10$. As a reference solution, we take the result of the second order low-rank projector-splitting algorithm with $m = 2000$ time steps (time step size $\tau = 5 \cdot 10^{-4}$).

We also consider a 6D two stream instability problem defined on the domain $\Omega = (0, 10\pi)^3 \times (-9, 9)^3$ with initial distribution

$$\begin{aligned} f_0(x_1, x_2, x_3, v_1, v_2, v_3) = &\frac{1}{\sqrt{(8\pi)^3}} \left(e^{-(v_1-\overline{v}_1)^2/2} + e^{-(v_1-\tilde{v}_1)^2/2}\right) \\ &\times \left(e^{-(v_2-\overline{v}_2)^2/2} + e^{-(v_2-\tilde{v}_2)^2/2}\right) \\ &\times \left(e^{-(v_3-\overline{v}_3)^2/2} + e^{-(v_3-\tilde{v}_3)^2/2}\right) \\ &\times (1 + \alpha_1 \cos(\kappa_1 x_1) + \alpha_2 \cos(\kappa_2 x_2) + \alpha_3 \cos(\kappa_3 x_3)). \end{aligned} \quad (5.21)$$

In this case the parameters are given by $\alpha_1 = \alpha_2 = \alpha_3 = 10^{-3}$, $\kappa_1 = \kappa_2 = \kappa_3 = \frac{1}{5}$, $\overline{v}_1 = \frac{5}{2}$, $\overline{v}_2 = \overline{v}_3 = 0$, $\tilde{v}_1 = -\frac{5}{2}$, $\tilde{v}_2 = -\frac{9}{4}$ and $\tilde{v}_3 = -2$. As for linear Landau damping, the problem is equipped with periodic boundary conditions in all directions and the rank is fixed to $r = 10$. We perform simulations up to final time $T = \frac{1}{20}$ with 32^3 discretization points for both space and velocity variables. We again consider as a reference solution the results of the second order algorithm with $m = 2000$ time steps (time step size $\tau = 2.5 \cdot 10^{-5}$).

The results for both problems are collected in Figure 5.3. In each of the two cases, we can clearly see that the first and second order algorithms show the expected order of convergence.

5.6.2 Linear Landau simulation

We consider again the 6D linear Landau problem (5.20) with periodic boundary conditions and the same set of parameters. However, now we pick 64^3 discretization points for the space variables and 256^3 for the velocity ones. The approximation rank is fixed to $r = 10$, and we perform two simulations with time step sizes $\tau = 10^{-1}$ and $\tau = 10^{-2}$, respectively. We emphasize that a direct (Eulerian or semi-Lagrangian) Vlasov solver would require a total number of degrees of freedom of approximately $4.4 \cdot 10^{12}$, and at least 70 TB of main memory (RAM) to perform the simulations in double precision arithmetic. This would clearly only be feasible on a supercomputer. The dynamical low-rank simulation, in contrast, runs on a single NVIDIA A100 equipped with 40 GB of memory, and the number of degrees of freedom are $10 \cdot 64^3 + 10 \cdot 256^3 \approx 1.7 \cdot 10^8$. The computational time of execution is approximately six minutes for the simulation with $\tau = 10^{-1}$ and an hour for the one with $\tau = 10^{-2}$.

The results obtained, in terms of electric energy, error in mass and error in total energy, are summarized in Figure 5.4. We can observe that the electric energy shows the expected theoretical exponential rate of decay up to approximately 10^{-6}, with very similar results for both the time step sizes. In this sense, what dominates after time $t = 40$ is the low-rank error, and we basically enter into a stagnation region (see Section 5.7.2 and Figure 5.6 for simulations with higher ranks). Concerning the

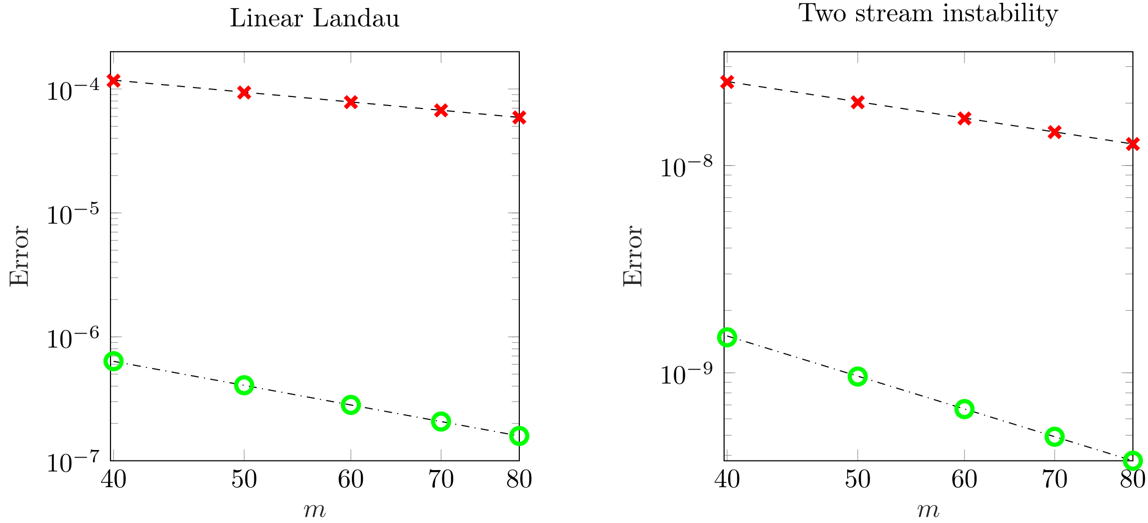

Figure 5.3: Orders of convergence for first order (red crosses) and second order (green circles) low-rank projector-splitting Algorithms 1 and 2. Left plot: linear Landau damping (5.20) with $T = 1$. Right plot: two stream instability problem (5.21) with $T = \frac{1}{20}$. In both cases, the rank is set to $r = 10$. The (relative) error is computed in maximum norm at the final time for a number of time steps equal to $m = 40, 50, 60, 70, 80$, with respect to a reference solution produced with Algorithm 2 and $m = 2000$ time steps. The dashed and dashed-dotted lines are reference lines of slope -1 and -2, respectively.

errors in mass and in total energy (both quantities are conserved by the original equations), even though the proposed low-rank projector-splitting integrator does not preserve a priori any quantity we obtain conservation of mass up to roughly $3 \cdot 10^{-8}$ and up to $7 \cdot 10^{-7}$ or $4 \cdot 10^{-8}$ for the energy, depending on the time step size. In this sense, the simulation with smaller time step size produces better results, meaning that for these quantities we still did not encounter a bound stemming from the low-rank truncation.

5.6.3 Two stream instability simulation

Let us perform now two simulations with the 6D two stream instability problem (5.21) and 128^3 discretization points for both spatial and velocity variables. The set of parameters is the same as for the example presented in Section 5.6.1, but we integrate the problem until time $T = 60$ with the second order Algorithm 2 and time step sizes $\tau = 6 \cdot 10^{-2}$ and $\tau = 10^{-2}$. The rank is fixed to $r = 10$. Again, a direct Vlasov solver would require a total number of degrees of freedom of approximately $4.4 \cdot 10^{12}$, while the dynamical low rank approach has $2 \cdot 10 \cdot 128^3 \approx 4.2 \cdot 10^7$ degrees of freedom. As for the linear Landau problem, we investigate the behavior of the electric energy and the conservation of mass and total energy. The results are summarized in Figure 5.5. In both cases, as expected, the electric energy shows an exponential increase before entering into a saturation phase, and similar considerations apply also for the error in mass and in energy (with slightly better results for the simulation with lower time step size, as expected). This behaviour matches well with what has been previously reported in the literature for this problem.

5.7 Performance results

We now investigate the performance of the low-rank projector-splitting algorithms presented in Section 5.3. This, in particular, should highlight the efficiency of using the software framework `Ensign` and demonstrate that GPUs provide an efficient way to run such simulations.

To perform a comparison with the GPU simulation, we present results on a dual-socket Intel Xeon Gold 6226R CPU based system with 2×16 CPU cores and 192 GB of RAM. For parallelizing the CPU code OpenMP is used and the Intel MKL library is employed for matrix and vector operations. For the GPU performance results we use the NVIDIA card described at the beginning of Section 5.6, and cuBLAS for BLAS operations. All simulations are conducted in double precision arithmetics.

5.7.1 CPU/GPU comparison

We consider here the linear Landau problem (5.20) discretized with 128^3 points in both space and velocity variables. This is done so that the effort of the K and L step can be directly compared. We integrate the problem until final time $T = 60$ with a time step size of $\tau = 10^{-2}$ and rank $r = 10$. We report the timings (in descending order) of a single time step, for the relevant parts of the algorithms, in Table 5.1 and Table 5.2 for the first order and the second order schemes, respectively.

CPU		GPU	
	Wall-clock time (s)		*Wall-clock time (s)*
K step	$2.87 \cdot 10^{0}$	K step	$2.34 \cdot 10^{-2}$
L step	$2.77 \cdot 10^{0}$	L step	$2.34 \cdot 10^{-2}$
D coefficients	$1.17 \cdot 10^{0}$	D coefficients	$1.13 \cdot 10^{-2}$
C coefficients	$1.17 \cdot 10^{0}$	C coefficients	$1.12 \cdot 10^{-2}$
Electric field	$9.76 \cdot 10^{-2}$	QR decomposition K	$5.31 \cdot 10^{-3}$
QR decomposition L	$2.00 \cdot 10^{-2}$	QR decomposition L	$5.26 \cdot 10^{-3}$
QR decomposition K	$1.96 \cdot 10^{-2}$	Electric field	$2.39 \cdot 10^{-3}$
S step	$6.72 \cdot 10^{-5}$	S step	$7.43 \cdot 10^{-4}$
Total	$8.12 \cdot 10^{0}$	Total	$8.30 \cdot 10^{-2}$

Table 5.1: Breakdown of timings for a single step of the first order Algorithm 1, in descending order, for CPU and GPU simulation of the linear Landau problem. The number of discretization points in both space and velocity is 128^3, the final time is $T = 60$, the time step size is $\tau = 10^{-2}$ and the rank is $r = 10$.

CPU		GPU	
	Wall-clock time (s)		*Wall-clock time (s)*
Lie splitting	$8.02 \cdot 10^{0}$	Lie splitting	$8.03 \cdot 10^{-2}$
First K step + QR	$2.83 \cdot 10^{0}$	First K step + QR	$3.19 \cdot 10^{-2}$
Second K step + QR	$2.82 \cdot 10^{0}$	Second K step + QR	$2.99 \cdot 10^{-2}$
L step + QR	$2.82 \cdot 10^{0}$	L step + QR	$2.89 \cdot 10^{-2}$
C coefficients	$1.12 \cdot 10^{0}$	C coefficients	$1.10 \cdot 10^{-2}$
D coefficients	$1.11 \cdot 10^{0}$	D coefficients	$1.09 \cdot 10^{-2}$
Electric field	$1.01 \cdot 10^{-1}$	Electric field	$1.53 \cdot 10^{-3}$
First S step	$5.83 \cdot 10^{-5}$	First S step	$6.81 \cdot 10^{-4}$
Second S step	$5.12 \cdot 10^{-5}$	Second S step	$6.50 \cdot 10^{-4}$
Total	$1.88 \cdot 10^{1}$	Total	$1.96 \cdot 10^{-1}$

Table 5.2: Breakdown of timings for a single step of the second order Algorithm 2, in descending order, for CPU and GPU simulation of the linear Landau problem. The number of discretization points in both space and velocity is 128^3, the final time is $T = 60$, the time step size is $\tau = 10^{-2}$ and the rank is $r = 10$.

As expected, the most costly parts of the algorithms consist of the K and the L steps (with roughly

the same computational time, as the degrees of freedom are equal in space and velocity). The cost of the S step, which involves the solution of a problem of size $r \times r$ is negligible. The remaining major part of the cost lies in the computation of the C and the D coefficients: again, this is expected, as they require a matrix-matrix product of size $N_v \times r$ (and $N_x \times r$, respectively). A single time step of the second order scheme is, as we would expect, approximately twice as costly as the first order scheme.

In both cases, we observe a drastic speedup (up to a factor of 100) between the GPU and the CPU based systems. The main reason for this is that very efficient implementations of matrix-matrix products are available on the GPU (in particular, in cuBLAS). This helps both in computing the coefficients as well as performing the K and L step. In addition, the algorithm needs to compute transcendental functions in order to evaluate the matrix functions in Fourier space. This is also an area where the GPU kernels drastically outperform the corresponding CPU implementation.

5.7.2 Varying rank

We now investigate the performance of the GPU implementation for the second order low-rank projector-splitting algorithm for different ranks. For this purpose, we consider again the 6D linear Landau problem (5.20) with 64^3 discretization points for the spatial variables and 128^3 discretization points for the velocity ones.

We integrate the problem up to $T = 60$ with a time step size of $\tau = 10^{-2}$ and different ranks $r = 5, 10, 15, 20$. The computational times for a single time step of the simulation are reported in Table 5.3. First of all, we observe that the wall-clock time increases roughly in a linear fashion as the rank increases. This scaling is better than the theoretical estimates provided in Section 5.3.2. Then, in

	Wall-clock time (s)
Rank 5	$5.12 \cdot 10^{-2}$
Rank 10	$8.70 \cdot 10^{-2}$
Rank 15	$1.28 \cdot 10^{-1}$
Rank 20	$1.85 \cdot 10^{-1}$

Table 5.3: Timings of a single time step for the linear Landau problem with increasing ranks r. The second order dynamical low-rank Algorithm 2 is employed. The number of discretization points in space is 64^3, in velocity it is 128^3, the final time is $T = 60$ and the time step size is $\tau = 10^{-2}$.

Figure 5.6 we summarize the behavior of the electric energy, of the error in mass and in total energy for all the ranks considered. We can clearly observe that rank 5 is not enough for the 6D problem under investigation. Starting from rank 10, we see a substantial improvement, in particular in terms of decay of electric energy.

Finally, we repeat a similar experiment with the 6D two stream instability problem (5.21). In this case, we consider 64^3 discretization points for both spatial and velocity variables. The problem is then integrated up to $T = 60$ with a time step size of $\tau = 10^{-2}$ and increasing rank $r = 5, 10, 15, 20$. The computational times for a single time step are summarized in Table 5.4, and analogous conclusions as for the linear Landau simulations can be drawn. In terms of electric energy, error in mass and error in total energy, we collect the results in Figure 5.7. We note that for the linear regime rank 5 still gives very good results in that it perfectly predicts both the growth rate of the instability as well as the time of its onset. Starting at saturation the rank 5 solution tends to overestimate the electric energy and thus the rank has to be increased. In order to capture saturation a rank 10 simulation is sufficient. If it is desired to integrate far into the nonlinear regime the rank has to be increased further.

5.8 Conclusions

We have demonstrated, by conducting numerical simulations for two test problems, that six-dimensional Vlasov simulations using a projector-splitting dynamical low-rank algorithm can be efficiently run on GPU

	Wall-clock time (s)
Rank 5	$1.84 \cdot 10^{-2}$
Rank 10	$3.06 \cdot 10^{-2}$
Rank 15	$4.85 \cdot 10^{-2}$
Rank 20	$7.15 \cdot 10^{-2}$

Table 5.4: Timings of a single time step for the two stream instability problem with increasing ranks r. The second order dynamical low-rank Algorithm 2 is employed. The number of discretization points in both space and velocity is 64^3, the final time is $T = 60$ and the time step size is $\tau = 10^{-2}$.

based systems. In particular, we report a drastic speedup compared to the CPU implementation, and we remark that these are the first dynamical low-rank results obtained for the full six-dimensional problem. We also emphasize that results with similar resolution using a direct (Eulerian or semi-Lagrangian) Vlasov solver could only be attained on large-scale supercomputers, while we have conducted the simulations on a single workstation equipped with an NVIDIA A100 card.

Algorithmic efficiency has been achieved by proposing a CFL-free and second order exponential integrator based dynamical low-rank scheme that uses a Fourier spectral phase space discretization. Implementation efficiency has been achieved by basing the implementation on the software framework `Ensign`.

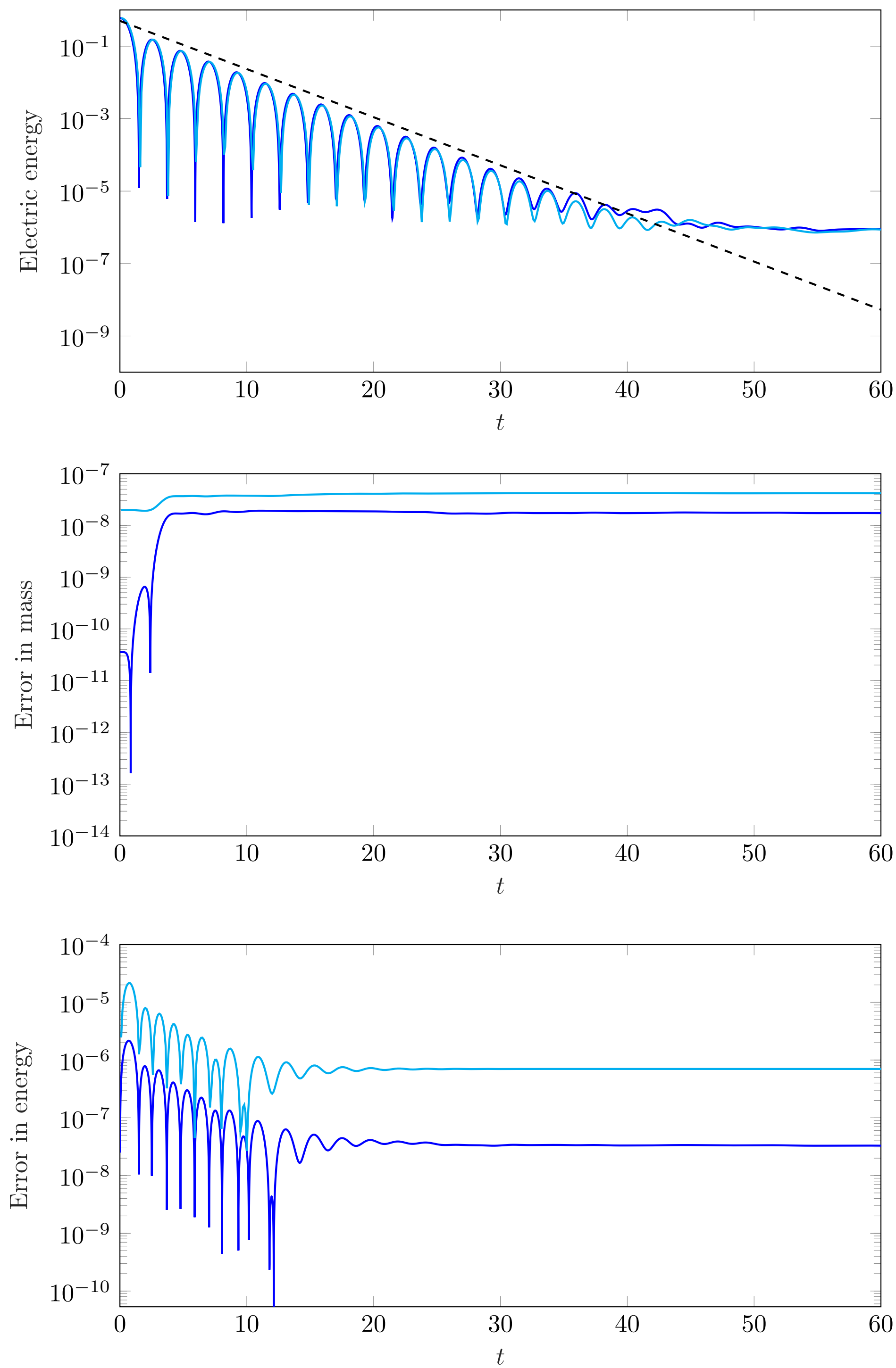

Figure 5.4: Linear Landau simulation with 64^3 space discretization points, 256^3 velocity discretization points, final time $T = 60$, rank $r = 10$ and time step sizes $\tau = 10^{-1}$ (cyan line) and $\tau = 10^{-2}$ (blue line), see Section 5.6.2. The second order low-rank projector-splitting Algorithm 2 is employed. Top plot: behavior of electric energy, with reference decay rate. Center plot: error in mass (relative). Bottom plot: error in total energy (relative).

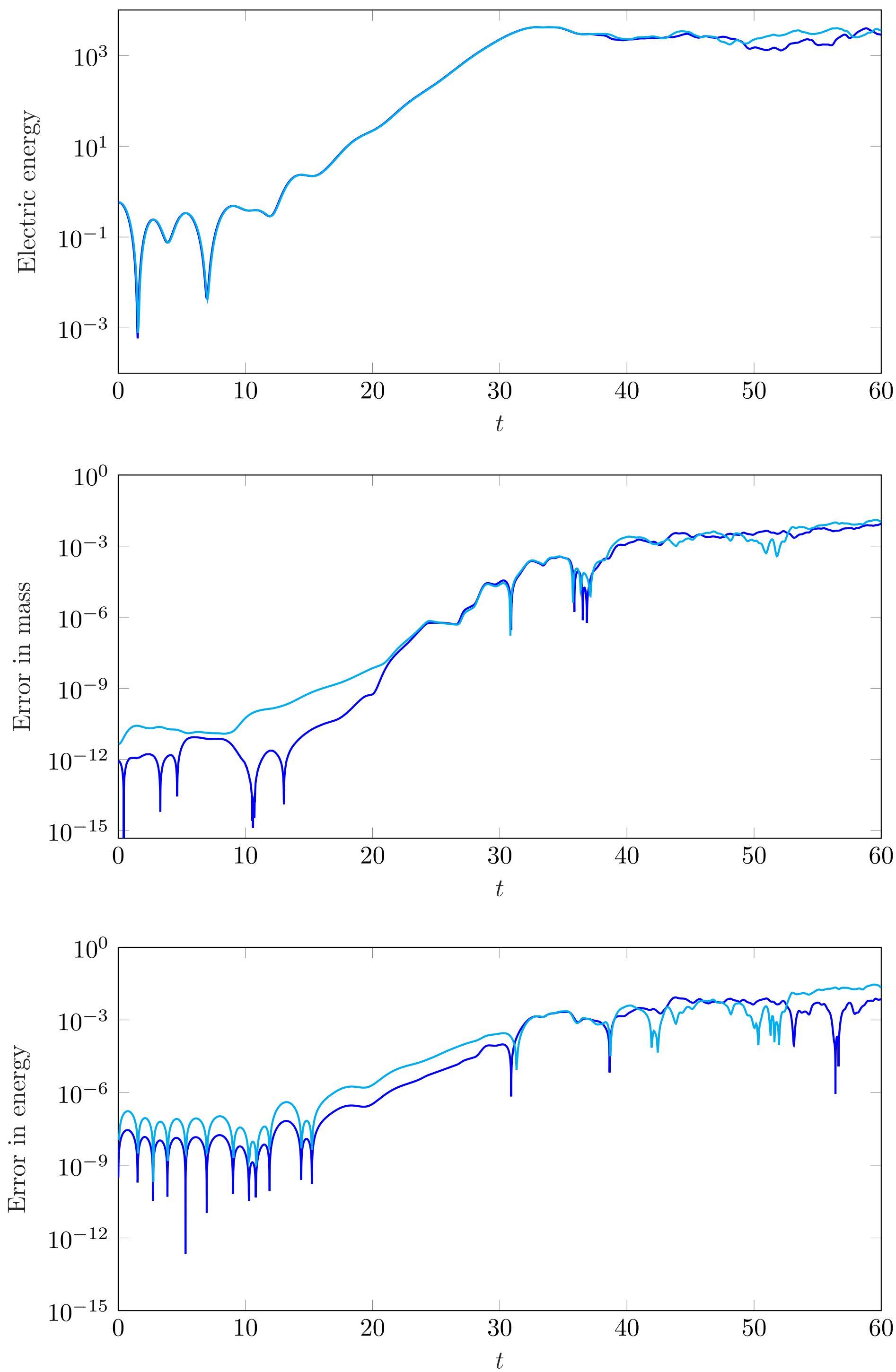

Figure 5.5: Two stream instability simulation with 128^3 discretization points for both space and velocity, final time $T = 60$, rank $r = 10$ and time step sizes $\tau = 6 \cdot 10^{-2}$ (cyan line) and $\tau = 10^{-2}$ (blue line), see Section 5.6.3. The second order low-rank projector-splitting Algorithm 2 is employed. Top plot: behavior of electric energy. Center plot: error in mass (relative). Bottom plot: error in total energy (relative).

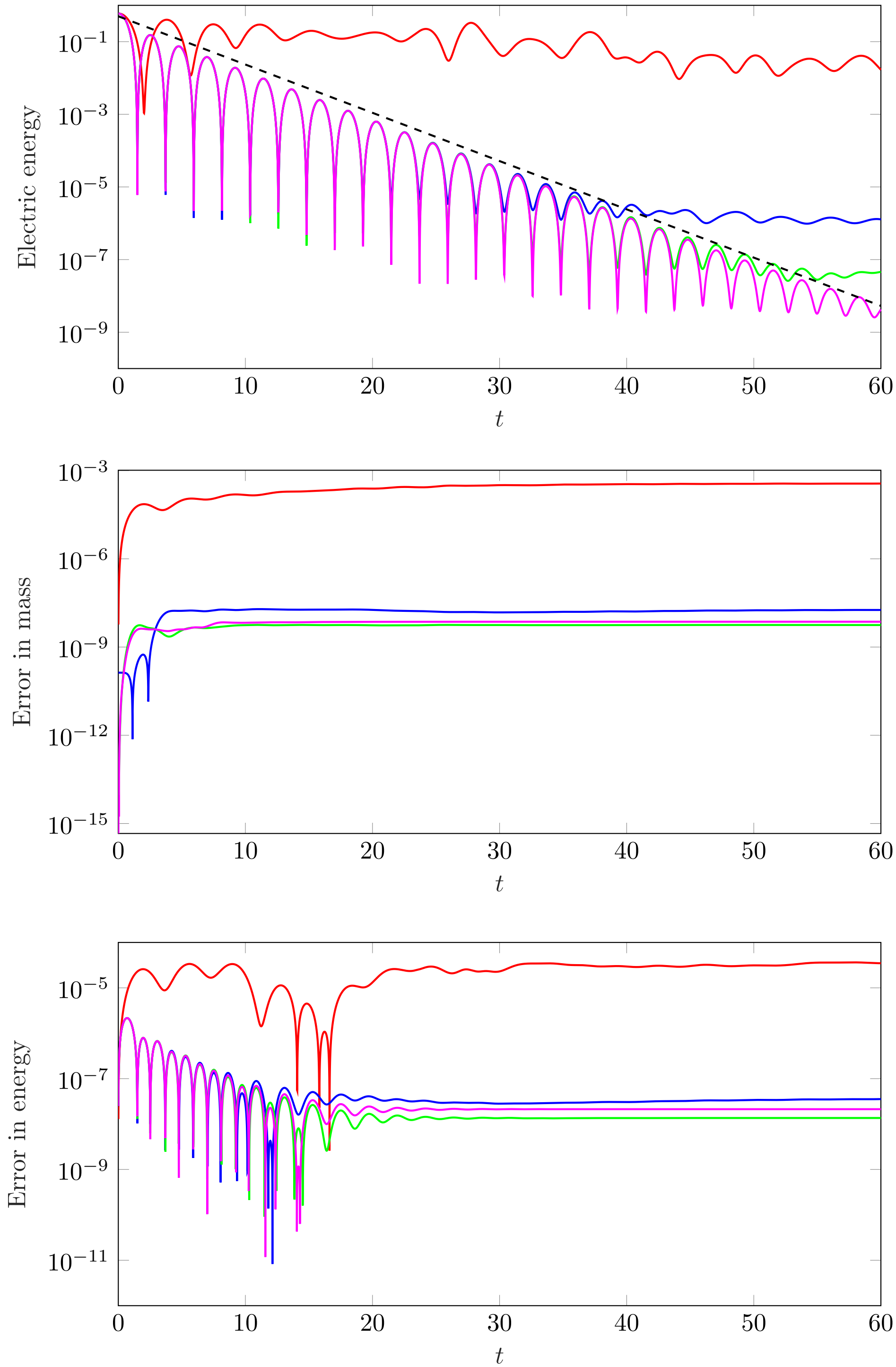

Figure 5.6: Linear Landau simulation with 64^3 space discretization points, 128^3 velocity discretization points, final time $T = 60$, time step size $\tau = 10^{-2}$ and different ranks r, see Section 5.7.2. The second order low-rank projector-splitting Algorithm 2 is employed. The red line corresponds to $r = 5$, the blue one to $r = 10$, the green one to $r = 15$ and the magenta one to $r = 20$. Top plot: behaviour of electric energy, with reference decay rate. Center plot: error in mass (relative). Bottom plot: error in total energy (relative).

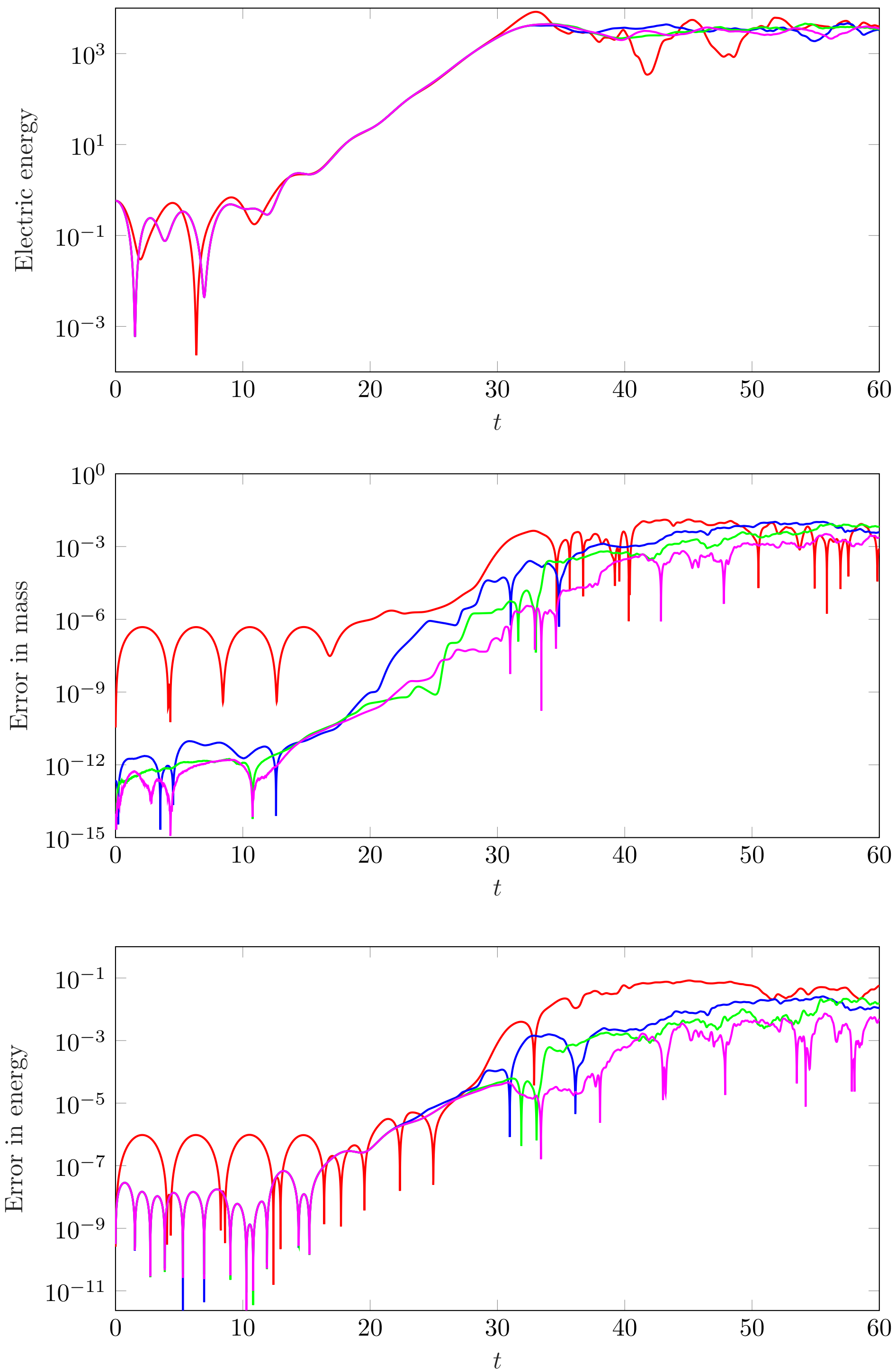

Figure 5.7: Two stream instability simulation with 64^3 discretization points for both space and velocity, final time $T = 60$, time step size $\tau = 10^{-2}$ and different ranks r, see Section 5.7.2. The second order low-rank projector-splitting Algorithm 2 is employed. The red line corresponds to $r = 5$, the blue one to $r = 10$, the green one to $r = 15$ and the magenta one to $r = 20$. Top plot: behaviour of electric energy. Center plot: error in mass (relative). Bottom plot: error in total energy (relative).

Chapter 6

Exponential integrators for mean-field selective optimal control problems

In this chapter we consider mean-field optimal control problems with selective action of the control, where the constraint is a continuity equation involving a non-local term and diffusion. First order optimality conditions are formally derived in a general framework, accounting for boundary conditions. Then, the optimality system is used to construct a reduced gradient method, where we introduce a novel algorithm for the numerical realization of the forward and the backward equations, based on exponential integrators. We illustrate extensive numerical experiments on different control problems for collective motion in the context of opinion formation, pedestrian dynamics and mass transfer.

The material of this chapter is taken from preprint [9], i.e., G. Albi, M. Caliari, E. Calzola and F. C.. Exponential integrators for mean-field selective optimal control problems. *arXiv preprint arXiv:2302.00127*, 2023.

6.1 Introduction

The study of collective motion of interacting agents systems is of paramount importance to understand the formation of coherent global behaviors at various scales, with applications to the study of biological, social, and economic phenomena. In recent years, there has been a surge of literature on the collective behavior of multi-agent systems, covering a wide range of topics such as cell aggregation and motility, coordinated animal motion [64, 70], opinion formation [99, 143, 180], coordinated human behavior [60, 72, 160], and cooperative robots [57, 92, 150, 158]. These fields are vast and constantly evolving, and we refer to the surveys [7, 67, 107] that provide a comprehensive overview of recent developments. Modeling such complex and diverse systems poses a significant challenge, since in general there are no first-principles as, for instance, in classical physics, or statistical mechanics. Nevertheless, the dynamics of the individuals have been successfully described by systems of Ordinary Differential Equations (ODEs) from Newton's laws designing basic interaction rules, such as attraction, repulsion and alignments, or, alternatively, by considering an evolutive game where the dynamics is driven by the simultaneous optimization of costs by N players such as in References [36, 125]. In this context, of paramount importance for several applications is the design of centralized policies able to optimally enforce a desired state of the agents, see for instance References [8, 11, 34, 50].

In this work, we consider a constrained setting, where interacting individuals are influenced by a centralized control with selective action, i.e.,

$$\mathrm{d}x_i = \left(\frac{1}{N} \sum_{j=1}^{N} p(x_i, x_j)(x_j - x_i) + s(t, x_i, \rho^N) u_i \right) \mathrm{d}t + \sigma \mathrm{d}W_i^t, \tag{6.1}$$

with initial data $x^0 = [x_1^0, \dots, x_N^0]$. Here each agent $x_i \in \Omega \subseteq \mathbb{R}^d$, for $i = 1, \dots, N$, accounts for pairwise interactions weighted by the function $p(\cdot,\cdot)$, and for disturbances modelled with a Brownian motion. The action of the control $u = [u_1, \dots, u_N]$ is weighted by a selective function $s(t, x_i, \rho^N)$, with $\rho^N(x)$ the empirical measures associated to the interacting agent system, i.e., $\rho^N(t,x) = N^{-1}\sum_{i=1}^N \delta(x_i(t) - x)$. Then, the optimal control u^* is obtained in the space of admissible controls U, by minimizing the cost functional

$$J(u; x^0) = \mathbb{E}\left[\int_0^T \frac{1}{2N}\sum_{i=1}^N \ell(t, x_i, \rho^N) + \gamma|u_i|^2\right], \tag{6.2}$$

where $\ell(t, x_i, f^N)$ is a running cost to be designed by the controller, with a quadratic penalization of the control for $\gamma \geq 0$.

For a large number of agents, we can write the mean-field optimal control problem corresponding to the finite dimensional optimal control problem (6.1)–(6.2) as follows (see References [90, 91])

$$\min_{u\in U} \frac{1}{2}\int_0^T \int_\Omega \left(\ell(t,x,\rho) + \gamma|u|^2\right)\rho dx dt, \tag{6.3a}$$

where ρ is the density function satisfying the Partial Differential Equation (PDE)

$$\begin{cases} \partial_t \rho + \nabla\cdot\left(\left(\mathcal{P}(\rho) + s(t,x,\rho)u\right)\rho\right) - \dfrac{\sigma^2}{2}\Delta\rho = 0, \\ \rho(0,x) = \rho_0(x). \end{cases} \tag{6.3b}$$

Here the non-local interactions among agents are described by the integral term

$$\mathcal{P}(\rho)(t,x) = \int_\Omega p(x,y)(y-x)\rho(t,y)dy \tag{6.4}$$

and $\rho_0(x)$ is the initial distribution of the agents. Differently from mean-field games [3, 49, 125], in this context the goal is to compute a mean-field optimal strategy capable of driving the population density to a specific target, avoiding the curse of dimensionality induced by the large scale non-linear system of N agents. However, the numerical solution of the PDE-constrained optimization problem (6.3a)–(6.3b) requires careful treatment [29]. To this end, we follow a reduced gradient method, where the first order optimality system is solved iteratively for the realization of the control, as in References [4, 10, 19]. Major challenges arise from the presence of the stiff diffusive and transport operators, and from the stability and storage requirements originated by the choice of the numerical solvers. For these kinds of problems, explicit time marching schemes usually require several time steps due to the lack of favorable stability properties, while implicit ones need possibly expensive solutions of (non)linear systems [12, 104, 108]. A prominent and effective alternative way to numerically integrate stiff equations in time is to employ explicit *exponential integrators*, see Reference [112] for a seminal review. After semidiscretization in space, these schemes require to approximate the action of exponential and of exponential-like matrix functions.

The chapter is structured as follows. In Section 6.2 we present a model of interest which generalizes the one in formulas (6.3), and we derive the formal optimality conditions using the associated Lagrangian function, obtaining a system of coupled PDEs. The first one is forward in time for the density function, while the second is backward in time for the adjoint variable. We numerically couple these equations using the steepest descent algorithm. In Section 6.3 we present the semidiscretization in space of the forward and of the backward PDEs, together with the numerical solution of the arising systems of ODEs using a pair of exponential integrators. For convenience of the reader, we also present there the derivation of the schemes and a brief discussion on common techniques to compute the involved matrix functions. Section 6.4 is devoted to some numerical validations and simulations in opinion formation (Sznajd), pedestrian (see Reference [32]) and mass transfer models. We finally draw some conclusions in Section 6.5.

6.2 Mean-field selective optimal control problem

We consider the mean-field optimal control problem [10, 32, 91] defined by the functional minimization

$$\min_u \mathcal{J}(u;\rho_0), \tag{6.5a}$$

where $\rho = \rho(t,x)$ is a probability density of agents satisfying

$$\begin{cases} \partial_t \rho + \nabla \cdot [(\mathcal{P}(\rho) + s(t,x,\rho)u)\,\rho] - \dfrac{\sigma^2}{2}\Delta\rho = 0, \\ \rho(0,x) = \rho_0(x), \\ \left((\mathcal{P}(\rho) + s(t,x,\rho)u)\,\rho - \dfrac{\sigma^2}{2}\nabla\rho \right) \cdot \vec{n} = \begin{cases} \beta\rho & \text{on } \Gamma_{\mathrm{F}}, \\ 0 & \text{on } \Gamma_{\mathrm{Z}}. \end{cases} \end{cases} \tag{6.5b}$$

and defined for each $(t,x) \in [0,T] \times \Omega$. The evolution of the density is driven by the non-local operator $\mathcal{P}(\rho)(t,x)$, as in equation (6.4), and by the control $u = u(t,x)$ weighted by the selective function $s(t,x,\rho)$. Here, we denoted by Γ_{F} the subset of the boundary in which there is a flux different from zero ($\beta \neq 0$) and by Γ_{Z} the part of $\partial\Omega$ with zero-flux boundary conditions. These two subsets are such that $\Gamma_{\mathrm{F}} \cup \Gamma_{\mathrm{Z}} = \partial\Omega$ and $\Gamma_{\mathrm{F}} \cap \Gamma_{\mathrm{Z}} = \emptyset$, and $\vec{n}$ is the outward normal vector to the boundary with norm equal to one. Finally, the functional in formula (6.5a) is given by

$$\mathcal{J}(u;\rho_0) = \frac{1}{2}\int_0^T \int_\Omega \left(e(t,x,\rho) + \gamma |u|^2 \rho \right) dxdt + \frac{1}{2}\int_\Omega c(T,x,\rho(T,x))dx$$

for a general running cost $e(t,x,\rho)$ and a terminal cost $c(T,x,\rho(T,x))$.

6.2.1 First order optimality conditions

We can derive the first order optimality conditions on a formal level using a Lagrangian approach. For a rigorous treatment we refer to References [10, 33]. We define the Lagrangian function with adjoint variable ψ as

$$\begin{aligned} \mathcal{L}(u,\rho,\psi) = &\frac{1}{2}\int_0^T \int_\Omega \left(e(t,x,\rho) + \gamma |u|^2 \rho \right) dxdt + \frac{1}{2}\int_\Omega c(T,x,\rho(T,x))dx \\ &- \int_0^T \int_\Omega \psi \left(\partial_t \rho + \nabla \cdot [(\mathcal{P}(\rho) + s(t,x,\rho)u)\,\rho] - \frac{\sigma^2}{2}\Delta\rho \right) dxdt. \end{aligned} \tag{6.6}$$

The optimal solution (u^*, ρ^*, ψ^*) can be found by equating to zero the partial Fréchet derivatives of the Lagrangian function, i.e., by solving the following system

$$\begin{cases} D_u \mathcal{L}(u,\rho,\psi) = 0, \\ D_\psi \mathcal{L}(u,\rho,\psi) = 0, \\ D_\rho \mathcal{L}(u,\rho,\psi) = 0. \end{cases} \tag{6.7}$$

Before computing the partial derivatives in system (6.7), we integrate by parts the last term appearing in the Lagrangian function (6.6) and we get

$$\begin{aligned} \mathcal{L}(u,\rho,\psi) = &\frac{1}{2}\int_0^T \int_\Omega \left(e(t,x,\rho) + \gamma |u|^2 \rho \right) dxdt + \frac{1}{2}\int_\Omega c(T,x,\rho(T,x))dx \\ &+ \int_0^T \int_\Omega \rho \left(\partial_t \psi + \frac{\sigma^2}{2}\Delta\psi + (\mathcal{P}(\rho) + s(t,x,\rho)u) \cdot \nabla\psi \right) dxdt \\ &- \int_0^T \int_{\Gamma_{\mathrm{F}}} \rho \left(\frac{\sigma^2}{2}\nabla\psi \cdot \vec{n} + \beta\psi \right) dbdt \\ &- \int_\Omega (\psi(T,x)\rho(T,x) - \psi(0,x)\rho(0,x))dx, \end{aligned}$$

where we used the value of the boundary conditions appearing in equation (6.5b). Performing then the computations of the partial derivatives we obtain the gradient direction for the control variable u

$$D_u\mathcal{L}(u,\rho,\psi) = \gamma u + s(t,x,\rho)\nabla\psi, \tag{6.8}$$

the forward PDE for the density function ρ

$$\begin{cases} \partial_t\rho + \nabla\cdot[(\mathcal{P}(\rho) + s(t,x,\rho)u)\,\rho] - \dfrac{\sigma^2}{2}\Delta\rho = 0, \\ \rho(0,x) = \rho_0(x), \\ \left((\mathcal{P}(\rho) + s(t,x,\rho)u)\,\rho - \dfrac{\sigma^2}{2}\nabla\rho\right)\cdot\vec{n} = \begin{cases} \beta\rho & \text{on } \Gamma_{\mathrm{F}}, \\ 0 & \text{on } \Gamma_{\mathrm{Z}}, \end{cases} \end{cases} \tag{6.9}$$

and the backward PDE for the adjoint variable ψ

$$\begin{cases} -\,\partial_t\psi = \dfrac{\sigma^2}{2}\Delta\psi + (\mathcal{P}(\rho) + (s(t,x,\rho) + \rho D_\rho s(t,x,\rho))u)\cdot\nabla\psi + \\ \qquad + \mathcal{Q}(\rho,\psi) + \dfrac{1}{2}(D_\rho e(t,x,\rho) + \gamma|u|^2), \\ \psi(T,x) = \psi_T(x), \\ \dfrac{\sigma^2}{2}\nabla\psi\cdot\vec{n} = \begin{cases} -\,\beta\psi & \text{on } \Gamma_{\mathrm{F}}, \\ 0 & \text{on } \Gamma_{\mathrm{Z}}, \end{cases} \end{cases} \tag{6.10}$$

where

$$\mathcal{Q}(\rho,\psi)(t,x) = \int_\Omega p(y,x)(x-y)\cdot\nabla\psi(t,y)\rho(t,y)dy$$

and $\psi_T(x) = \frac{1}{2}D_\rho c(T,x,\rho(T,x))$. Now, in order to solve model (6.5), we employ a steepest descent approach (see References [10, 19]). Starting with an initial control u^0, at each iteration ℓ we insert u^ℓ into the forward equation (6.9) and solve it for $\rho = \rho^{\ell+1}$. We then insert u^ℓ and $\rho^{\ell+1}$ into the backward equation (6.10) and solve it for $\psi = \psi^{\ell+1}$. We finally update the control by using the gradient direction (6.8), i.e.,

$$u^{\ell+1} = u^\ell - \lambda^\ell(\gamma u^\ell + s(t,x,\rho^{\ell+1})\nabla\psi^{\ell+1})$$

and get $u^{\ell+1}$. We proceed iterating until $\mathcal{J}(u^{\ell+1})$ has stabilized within a given tolerance. For the numerical solution of equations (6.9) and (6.10) we use the method of lines: in fact, we first discretize in space and then use appropriate exponential integrators for the obtained systems of ODEs.

6.3 Numerical integrators for the semidiscretized equations

In this section, we explain how to solve the forward and the backward PDEs in the steepest descent algorithm. By observing that both are semilinear parabolic equations, the idea is to use numerical schemes tailored for this type of problems. A prominent way is to apply explicit exponential integrators [112] to the systems of ODEs arising from the semidiscretization in space of the PDEs. By construction, these schemes solve exactly linear ODEs systems with constant coefficients, they allow for time steps usually much larger than those required by classical explicit methods (i.e., typically they do not suffer from a CFL restriction), and do not require the solution of (non)linear systems as implicit methods do. On the other hand, this class of integrators requires the computation of the action of exponential-like matrix functions for which different efficient techniques have been developed in recent years.

6.3.1 Forward PDE

For sake of clarity, and since we will present later on one-dimensional numerical examples, we consider $\Omega = [a, b]$ and we rewrite the forward PDE (6.9)

$$\begin{cases} \partial_t \rho(t,x) = \dfrac{\sigma^2}{2}\partial_{xx}\rho(t,x) - \partial_x\left((\mathcal{P}(\rho(t,\cdot))(t,x) + s(t,x,\rho(t,x))u(t,x))\rho(t,x)\right), \\ \rho(0,x) = \rho_0(x), \\ \left((\mathcal{P}(\rho(t,\cdot))(t,x) + s(t,x,\rho(t,x))u(t,x))\rho(t,x) - \dfrac{\sigma^2}{2}\partial_x\rho(t,x)\right)\Big|_a = \beta_a\rho(t,a), \\ \left((\mathcal{P}(\rho(t,\cdot))(t,x) + s(t,x,\rho(t,x))u(t,x))\rho(t,x) - \dfrac{\sigma^2}{2}\partial_x\rho(t,x)\right)\Big|_b = \beta_b\rho(t,b), \end{cases}$$

where $\beta_a, \beta_b \in \mathbb{R}$ can be selected so that it is possible to express both zero and nonzero fluxes. Notice that when we solve this equation we consider $u(t,x)$ a given function. We introduce a semidiscretization in space by finite differences on a grid of points x_i, with $i = 1,\ldots,n$, in such a way that $\boldsymbol{\rho}(t) = [\rho_1(t),\ldots,\rho_n(t)]^\mathsf{T}$ is the unknown vector whose components $\rho_i(t)$ approximate $\rho(t,x_i)$. Now, by denoting D_1 and D_2 the matrices which discretize ∂_x and ∂_{xx} at the grid points, respectively, and P the discretization of the linear integral operator $\mathcal{P}$ by a quadrature formula, the linear part of the right hand side of the equation is discretized by

$$\tilde{A}_\mathrm{F}\boldsymbol{\rho}(t) = \frac{\sigma^2}{2}D_2\boldsymbol{\rho}(t),$$

while the nonlinear part becomes

$$\tilde{\boldsymbol{g}}_\mathrm{F}(t,\boldsymbol{\rho}(t)) = -(D_1P\boldsymbol{\rho}(t))\boldsymbol{\rho}(t) - (P\boldsymbol{\rho}(t))(D_1\boldsymbol{\rho}(t)) - (D_1\boldsymbol{s}(t,\boldsymbol{\rho}(t)))\boldsymbol{u}(t)\boldsymbol{\rho}(t) - \boldsymbol{s}(t,\boldsymbol{\rho}(t))(D_1\boldsymbol{u}(t))\boldsymbol{\rho}(t).$$

Now, we also discretize the boundary conditions with finite differences by using virtual nodes, and we modify accordingly both the linear part $\tilde{A}_\mathrm{F}$ and the nonlinear one $\tilde{\boldsymbol{g}}_\mathrm{F}(t,\boldsymbol{\rho}(t))$. The resulting nonlinear system of ODEs is then

$$\begin{cases} \boldsymbol{\rho}'(t) = A_\mathrm{F}\boldsymbol{\rho}(t) + \boldsymbol{g}_\mathrm{F}(t,\boldsymbol{\rho}(t)), \quad t\in[0,T], \\ \boldsymbol{\rho}(0) = \boldsymbol{\rho}_0. \end{cases} \tag{6.11}$$

Given a time discretization $[t_0,\ldots,t_k,\ldots,t_m]$, with $t_0 = 0$ and $t_m = T$, the exact solution of system (6.11) at time t_{k+1} can be expressed using the variation-of-constants formula, i.e.,

$$\boldsymbol{\rho}(t_{k+1}) = \mathrm{e}^{\tau_{k+1}A_\mathrm{F}}\boldsymbol{\rho}(t_k) + \int_0^{\tau_{k+1}} \mathrm{e}^{(\tau_{k+1}-s)A_\mathrm{F}}\boldsymbol{g}_\mathrm{F}(t_k+s,\boldsymbol{\rho}(t_k+s))ds,$$

where $\tau_{k+1} = t_{k+1} - t_k$, for $k = 0,\ldots,m-1$. In order to obtain an explicit first order numerical scheme, we denote by $\boldsymbol{\rho}_k$ the approximation of $\boldsymbol{\rho}(t_k)$ and approximate the nonlinear function $\boldsymbol{g}_\mathrm{F}(t_k+s,\boldsymbol{\rho}(t_k+s))$ with $\boldsymbol{g}_\mathrm{F}(t_k,\boldsymbol{\rho}_k)$. Hence, we have

$$\begin{aligned} \boldsymbol{\rho}(t_{k+1}) \approx \boldsymbol{\rho}_{k+1} &= \mathrm{e}^{\tau_{k+1}A_\mathrm{F}}\boldsymbol{\rho}_k + \int_0^{\tau_{k+1}} \mathrm{e}^{(\tau_{k+1}-s)A_\mathrm{F}}\boldsymbol{g}_\mathrm{F}(t_k,\boldsymbol{\rho}_k)ds \\ &= \mathrm{e}^{\tau_{k+1}A_\mathrm{F}}\boldsymbol{\rho}_k + \left(\int_0^{\tau_{k+1}} \mathrm{e}^{(\tau_{k+1}-s)A_\mathrm{F}}ds\right)\boldsymbol{g}_\mathrm{F}(t_k,\boldsymbol{\rho}_k) \\ &= \mathrm{e}^{\tau_{k+1}A_\mathrm{F}}\boldsymbol{\rho}_k + \left(\tau_{k+1}\int_0^1 \mathrm{e}^{\tau_{k+1}(1-\theta)A_\mathrm{F}}d\theta\right)\boldsymbol{g}_\mathrm{F}(t_k,\boldsymbol{\rho}_k) \\ &= \mathrm{e}^{\tau_{k+1}A_\mathrm{F}}\boldsymbol{\rho}_k + \tau_{k+1}\varphi_1(\tau_{k+1}A_\mathrm{F})\boldsymbol{g}_\mathrm{F}(t_k,\boldsymbol{\rho}_k). \end{aligned} \tag{6.12}$$

Here we introduced the exponential-like function

$$\varphi_1(X) = \int_0^1 \mathrm{e}^{(1-\theta)X}d\theta,$$

with $X \in \mathbb{C}^{n\times n}$ a generic matrix. This scheme is known as *exponential Euler*, it is a fully explicit method of first (stiff) order and it is A-stable by construction. Its implementation requires at each time step the evaluation of a linear combination of type $\mathrm{e}^{\tau_{k+1}X}\boldsymbol{v}_k + \tau_{k+1}\varphi_1(\tau_{k+1}X)\boldsymbol{w}_k$, where $\boldsymbol{v}_k, \boldsymbol{w}_k \in \mathbb{C}^n$ are suitable vectors, which we will address in Section 6.3.3.

Selective function independent of the density

A remarkable occurrence in the literature is the one in which the selective function does not depend on the density, i.e., $s(t,x,\rho(t,x)) = s(t,x)$ (see Reference [10] for the case $s(t,x) = 1$, which we will also consider in the numerical examples). In this case, some terms in the nonlinear part $\tilde{\boldsymbol{g}}_\mathrm{F}(t,\boldsymbol{\rho}(t))$ can actually be incorporated into the linear one. In fact, we obtain

$$\tilde{A}_\mathrm{F}(t)\boldsymbol{\rho}(t) = \frac{\sigma^2}{2}D_2\boldsymbol{\rho}(t) - (D_1\boldsymbol{s}(t))\boldsymbol{u}(t)\boldsymbol{\rho}(t) - \boldsymbol{s}(t)(D_1\boldsymbol{u}(t))\boldsymbol{\rho}(t) - \boldsymbol{s}(t)\boldsymbol{u}(t)(D_1\boldsymbol{\rho}(t)),$$

while the nonlinear part is now given by

$$\tilde{\boldsymbol{g}}_\mathrm{F}(t,\boldsymbol{\rho}(t)) = -(D_1P\boldsymbol{\rho}(t))\boldsymbol{\rho}(t) - (P\boldsymbol{\rho}(t))(D_1\boldsymbol{\rho}(t)).$$

By modifying accordingly the quantities in order to impose the boundary conditions, we end up with the system of ODEs

$$\begin{cases} \boldsymbol{\rho}'(t) = A_\mathrm{F}(t)\boldsymbol{\rho}(t) + \boldsymbol{g}_\mathrm{F}(t,\boldsymbol{\rho}(t)), \quad t \in [0,T], \\ \boldsymbol{\rho}(0) = \boldsymbol{\rho}_0, \end{cases} \tag{6.13}$$

which is similar to system (6.11), except for the fact that the linear part has time dependent coefficients. Nevertheless, at each t_k we can rewrite equivalently this system as

$$\begin{cases} \boldsymbol{\rho}'(t) = A_\mathrm{F}(t_k)\boldsymbol{\rho}(t) + (A_\mathrm{F}(t) - A_\mathrm{F}(t_k))\boldsymbol{\rho}(t) + \boldsymbol{g}_\mathrm{F}(t,\boldsymbol{\rho}(t)) \\ \qquad\;\; = A_\mathrm{F}(t_k)\boldsymbol{\rho}(t) + \boldsymbol{g}^k_\mathrm{F}(t,\boldsymbol{\rho}(t)), \\ \boldsymbol{\rho}(0) = \boldsymbol{\rho}_0, \end{cases}$$

and apply the exponential Euler method. Thus, we end up with the scheme

$$\begin{aligned} \boldsymbol{\rho}(t_{k+1}) \approx \boldsymbol{\rho}_{k+1} &= \mathrm{e}^{\tau_{k+1}A_\mathrm{F}(t_k)}\boldsymbol{\rho}_k + \tau_{k+1}\varphi_1(\tau_{k+1}A_\mathrm{F}(t_k))\boldsymbol{g}^k_\mathrm{F}(t_k,\boldsymbol{\rho}_k) \\ &= \mathrm{e}^{\tau_{k+1}A_\mathrm{F}(t_k)}\boldsymbol{\rho}_k + \tau_{k+1}\varphi_1(\tau_{k+1}A_\mathrm{F}(t_k))\boldsymbol{g}_\mathrm{F}(t_k,\boldsymbol{\rho}_k), \end{aligned} \tag{6.14}$$

for $k = 0,\ldots,m-1$. As for the general case $s(t,x,\rho(t,x))$, we obtain in this way an explicit method of first order (which we call *exponential Euler–Magnus*) that requires again a linear combination of actions of the matrix exponential and of the matrix φ_1 function.

6.3.2 Backward PDE

We rewrite the backward PDE (6.10) in the one-dimensional case $\Omega = [a,b]$

$$\begin{cases} -\partial_t\psi(t,x) = \dfrac{\sigma^2}{2}\partial_{xx}\psi(t,x) + \mathcal{P}(\rho(t,\cdot))(t,x)\partial_x\psi(t,x) \\ \qquad\qquad + (s(t,x,\rho(t,x)) + \rho(t,x)s_\rho(t,x,\rho(t,x)))u(t,x)\partial_x\psi(t,x) \\ \qquad\qquad + \mathcal{Q}(\rho(t,\cdot),\psi(t,\cdot))(t,x) + \dfrac{1}{2}\left(e_\rho(t,x,\rho(t,x)) + \gamma u^2(t,x)\right), \\ \psi(T,x) = \psi_T(x), \\ \dfrac{\sigma^2}{2}\partial_x\psi(t,x)\big|_a = -\beta_a\psi(t,a), \\ \dfrac{\sigma^2}{2}\partial_x\psi(t,x)\big|_b = -\beta_b\psi(t,b), \end{cases}$$

where $s_\rho(t,x,\rho(t,x)) = D_\rho s(t,x,\rho(t,x))$ and $e_\rho(t,x,\rho(t,x)) = D_\rho e(t,x,\rho(t,x))$. Here we assume that $\rho(t,x)$ and $u(t,x)$ are given functions. By applying a finite difference discretization on the same spatial grid as above and defining Q the discretization of the linear integral operator $\mathcal{Q}$ we obtain the linear part

$$\tilde{A}_{\mathrm{B}}(t)\boldsymbol{\psi}(t) = \frac{\sigma^2}{2}D_2\boldsymbol{\psi}(t)+(P\boldsymbol{\rho}(t))(D_1\boldsymbol{\psi}(t))+(\boldsymbol{s}(t,\boldsymbol{\rho}(t))+\boldsymbol{\rho}(t)\boldsymbol{s}_{\boldsymbol{\rho}}(t,\boldsymbol{\rho}(t)))\boldsymbol{u}(t)(D_1\boldsymbol{\psi}(t))+Q(\boldsymbol{\rho}(t)(D_1\boldsymbol{\psi}(t)))$$

and the source term

$$\tilde{\boldsymbol{g}}_B(t) = \frac{1}{2}\boldsymbol{e}_{\boldsymbol{\rho}}(t,\boldsymbol{\rho}(t)) + \gamma\boldsymbol{u}^2(t).$$

Finally, by taking into consideration boundary conditions, we end up with the inhomogeneous time dependent coefficient linear system of ODEs

$$\begin{cases} -\boldsymbol{\psi}'(t) = A_{\mathrm{B}}(t)\boldsymbol{\psi}(t) + \boldsymbol{g}_{\mathrm{B}}(t), \quad t \in [0,T], \\ \ \ \boldsymbol{\psi}(T) = \boldsymbol{\psi}_T. \end{cases} \tag{6.15}$$

By considering the same time discretization $[t_0,\ldots,t_{k+1},\ldots,t_m]$ as above, system (6.15) has a similar structure to system (6.13). Hence, taking into account that we are marching backward in time, we apply the exponential Euler–Magnus method and we obtain the time marching

$$\boldsymbol{\psi}(t_k) \approx \boldsymbol{\psi}_k = \mathrm{e}^{\tau_{k+1}A_{\mathrm{B}}(t_{k+1})}\boldsymbol{\psi}_{k+1} + \tau_{k+1}\varphi_1(\tau_{k+1}A_{\mathrm{B}}(t_{k+1}))\boldsymbol{g}_{\mathrm{B}}(t_{k+1}), \tag{6.16}$$

for $k = m-1, m-2\ldots,0$.

6.3.3 Matrix functions evaluation

We have introduced two exponential integrators that require, at each time step, the evaluation of

$$\mathrm{e}^{\tau X}\boldsymbol{v} + \tau\varphi_1(\tau X)\boldsymbol{w}, \tag{6.17}$$

where $\tau > 0$, $X \in \mathbb{R}^{n\times n}$, and $\boldsymbol{v}$, $\boldsymbol{w} \in \mathbb{R}^n$. We stress that these quantities depend in general on the current time step, but for simplicity of notation we dropped the subscripts. If we choose a uniform time discretization, i.e., $\tau_k = \tau$ for $k = 0,\ldots,m-1$, in the exponential Euler scheme (6.12) we can compute once and for all the matrices $\mathrm{e}^{\tau A_{\mathrm{F}}}$ and $\varphi_1(\tau A_{\mathrm{F}})$ and then multiply by the corresponding vectors. In this case, for the matrix function approximations the most common techniques are Taylor expansions or Padé rational approximations with scaling and squaring (see, for instance, References [5, 46, 171, 173]). This approach is computationally attractive only for matrices of moderate size, taking into account also that the resulting matrix functions are full even if the original ones were sparse. When employing the exponential Euler–Magnus schemes (6.14) and (6.16), we can still pursue this approach. However, since here the matrices change at each time step, we need to recompute the matrix functions every time accordingly. It is also possible to compute linear combination (6.17) by using a *single* slightly augmented matrix function evaluation. In fact, thanks to [167, Proposition 2.1], we have that the first n rows of

$$\exp\left(\tau\begin{bmatrix} X & \boldsymbol{w} \\ 0\cdots 0 & 0\end{bmatrix}\right)\begin{bmatrix}\boldsymbol{v}\\ 1\end{bmatrix}$$

coincide with vector (6.17). This is an attractive choice in a variable step size scenario, in which both the forward and the backward equations could be solved by a single matrix function evaluation at each time step.

When X is a large sized and sparse matrix, it may be convenient to compute directly vector (6.17) at each time step *without* explicitly computing the matrix exponential. State-of-the-art techniques follow this approach and are based on Krylov methods or direct interpolation polynomial methods (see, for instance, References [6, 43, 97, 132]).

6.4 Numerical experiments

We present in this section numerical examples arising from different choices of parameters and functions in the continuous model (6.5). In particular, we consider numerical experiments for different classes of multi-agent systems: opinion formation, pedestrian dynamics and mass transfer. In all cases, we discretize in space with second order centered finite differences and we employ the trapezoidal rule for the quadrature of the integral operators. All the numerical experiments have been performed on an Intel® Core™ i7-10750H CPU with six physical cores and 16GB of RAM, using MATLAB programming language. As a software, we use MathWorks MATLAB® R2022a. In order to compute the needed actions of exponential and φ_1-function, we employ the `kiops` function[1], which is based on the Krylov method and whose underlying algorithm is thoroughly presented in Reference [97]. This routine requires an input tolerance, which we set sufficiently small in order not to affect the accuracy of the temporal integration.

6.4.1 Control in opinion dynamics: Sznajd model

In this section we consider the Sznajd model for control of opinion dynamics, similarly to References [10, 176]. We set the spatial domain to $\Omega = [-1, 1]$, whose boundaries represent the extremal opinions. The running cost is $e(t, x, \rho) = |x - x_d|^2 \rho(t, x)$ and the selective function $s(t, x, \rho)$ is set to the constant 1 (hence, we use the exponential Euler–Magnus scheme (6.14) for the forward equation). We consider zero-flux boundary conditions everywhere and null terminal cost function $c(T, x, \rho(T, x))$. The interaction function is selected as $p(x, y) = x^2 - 1$, representing a repulsive interaction, and the target point in the running cost is $x_d = -0.5$. Moreover, we set the penalization parameter $\gamma = 0.5$ and the diffusion coefficient $\sigma = \sqrt{0.02}$. The initial density function is of bimodal type

$$\rho_0(x) = C(\rho_+(x + 0.75; 0.05, 0.5) + \rho_+(x - 0.5; 0.15, 1)),$$

where

$$\rho_+(x; a, b) = \max\left\{-\left(\frac{x}{b}\right)^2 + a, 0\right\}$$

and C defined so that $\int_\Omega \rho_0(x)dx = 1$.

First of all, we show that the expected temporal rate of convergence of the exponential integrators is preserved also after a complete solution of the model. In fact, for a semidiscretization in space with $n = 200$ uniform grid points, we solve several times model (6.5) by the steepest descent method described at the end of Section 6.2 and employing an increasing sequence of time steps, ranging from $m = 300$ to $m = 700$. Each time, after the stabilization of the functional $\mathcal{J}$, we measure the error at final time $T = 4$ for $\boldsymbol{\rho}(t)$ and at initial time for $\boldsymbol{\psi}(t)$ with respect to reference solutions. We display in Figure 6.1 the obtained relative errors, which confirm the expected accuracy and rate of convergence.

Then, we show the behavior of the Sznajd model in opinion formation. For this purpose we use a spatial discretization of $n = 1000$ points and $m = 200$ time steps. Notice that we can employ a number of time steps small with respect to the number of discretization points since the exponential integrators applied to this problem do not exhibit any CFL restriction, in contrast to explicit methods. In Figure 6.2 we show the evolution of the density $\rho(t, x)$ and of the control $u(t, x)$. The results have the expected behavior of concentration of the opinions around the target point $x_d = -0.5$ and qualitatively match the analogous simulation available in the literature [10]. Moreover, we show in Figure 6.3 the value of the functional $\mathcal{J}(u^\ell)$ at the successive iterations of the steepest descent method. We observe that the method needs 19 iterations to reach the input tolerance $2 \cdot 10^{-3}$. Finally, the overall computational time of this simulation is about 55 seconds.

6.4.2 Crowd dynamics: fast exit scenario

In this section we consider a model for crowd dynamics taken from Reference [32]. We set the model in the spatial domain $\Omega = [-1, 1]$, whose boundaries represent the exit doors. The non-local interaction

[1] `https://gitlab.com/stephane.gaudreault/kiops/-/tree/master/`

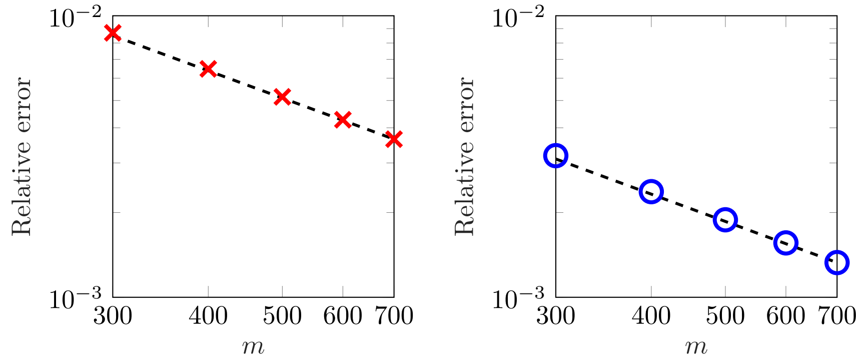

Figure 6.1: Relative errors in infinity norm of $\boldsymbol{\rho}(T)$ (left, $T = 4$) and $\boldsymbol{\psi}(0)$ (right), with respect to a reference solution, for the Sznajd model described in Section 6.4.1 with $n = 200$ spatial discretization points and varying number of time steps m. The reference line of order 1 is also displayed.

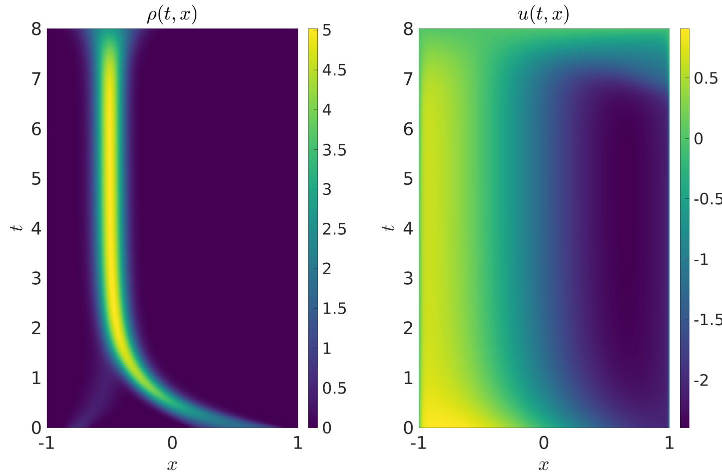

Figure 6.2: Evolution of the density $\rho(t,x)$ (left) and of the control $u(t,x)$ (right) up to final time $T = 8$ for the Sznajd model described in Section 6.4.1 with $n = 1000$ spatial discretization points and $m = 200$ time steps.

kernel $p(x,y)$ is null and the selective function $s(t,x,\rho)$ is $1-\rho$ (hence, we employ the exponential Euler method (6.12) for the forward equation). The diffusion parameter is $\sigma = \sqrt{0.04}$ and the exit intensity flux is $\beta = 10$. The initial density function models the presence of two distinct groups, namely $\rho_0(x) = 0.9\mathrm{e}^{-100(x+0.4)^2} + 0.65\mathrm{e}^{-150x^2}$.

Similarly to the opinion dynamics case, we first show that the expected temporal rate of convergence of the exponential integrators is preserved after a complete solution of the model. To this purpose, we discretize this problem with $n = 200$ spatial discretization points and with different number of time steps, from $m = 300$ to $m = 700$, up to the final time $T = 2$. After the stabilization of the functional $\mathcal{J}$ in the steepest descent algorithm, we measure the error at final time for $\boldsymbol{\rho}(t)$ and at initial time for $\boldsymbol{\psi}(t)$ with respect to reference solutions. We display in Figure 6.4 the obtained relative errors which again confirm the expected accuracy and rate of convergence.

Then, we solve the same model up to the final time $T = 3$ and show its behavior. We discretize this problem with $n = 1000$ spatial discretization points and $m = 250$ time steps. We show the evolution of the density and of the control in Figure 6.5, where we can clearly see the exit of the crowd from the two doors. Moreover, we show in Figure 6.6 the value of the functional $\mathcal{J}(u^\ell)$ at the successive iterations

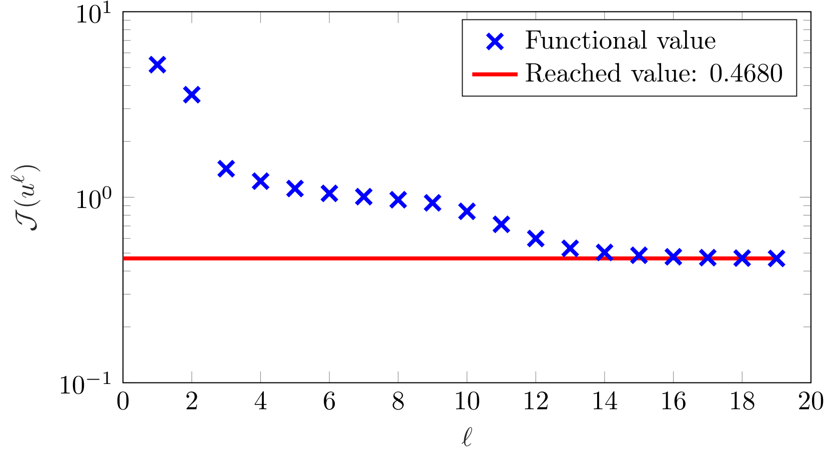

Figure 6.3: Value of the functional $\mathcal{J}(u^\ell)$ at the successive iterations of the steepest descent method for the Sznajd model described in Section 6.4.1 ($n = 1000$ and $m = 200$).

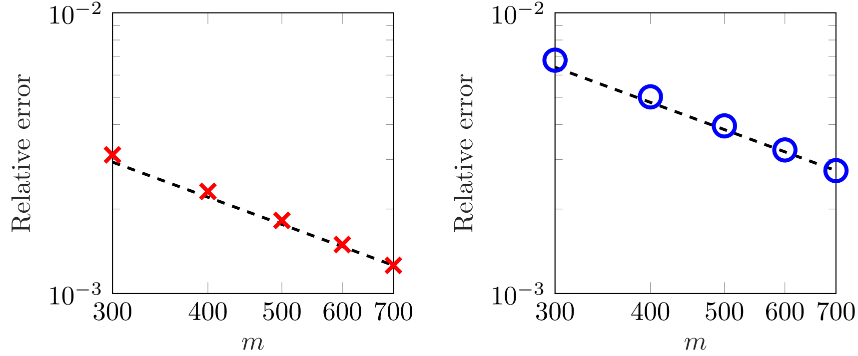

Figure 6.4: Relative errors in infinity norm of $\boldsymbol{\rho}(T)$ (left, $T = 2$) and $\boldsymbol{\psi}(0)$ (right), with respect to a reference solution, for the pedestrian model described in Section 6.4.2 with $n = 200$ spatial discretization points and varying number of time steps m. The reference line of order 1 is also displayed.

of the steepest descent method. We observe that the method needs 14 iterations to reach the input tolerance $2 \cdot 10^{-3}$. Finally, the overall computational time of this simulation is about 45 seconds.

6.4.3 Mass transfer problem via optimal control

In this final example, we present an optimal control approach to a mass transfer problem, see for instance References [22, 170], where the particle density accounts for non-local interactions [28, 51]. Hence, the goal is to move the initial density function in the spatial domain $\Omega = [-1, 1]$

$$\rho_0(x) = C(\mathrm{e}^{-(x-\mu_0)^2/(2\sigma_0^2)}),$$

where $\mu_0 = 0$, $\sigma_0 = 0.1$, and C is defined so that $\int_\Omega \rho_0(x)dx = 1$, to a target one

$$\bar{\rho}(x) = \bar{C}\left(\mathrm{e}^{-(x-\mu_1)^2/(2\sigma_1^2)} + \mathrm{e}^{-(x-\mu_2)^2/(2\sigma_2^2)}\right),$$

where $\mu_1 = 0.5$, $\sigma_1 = 0.1$, $\mu_2 = -0.3$, and $\sigma_2 = 0.15$, and $\bar{C}$ is defined so that $\int_\Omega \bar{\rho}(x)dx = 1$. The boundary conditions are of zero-flux type, the running cost is $e(t, x, \rho) = |\rho - \bar{\rho}|^2$, the interaction kernel

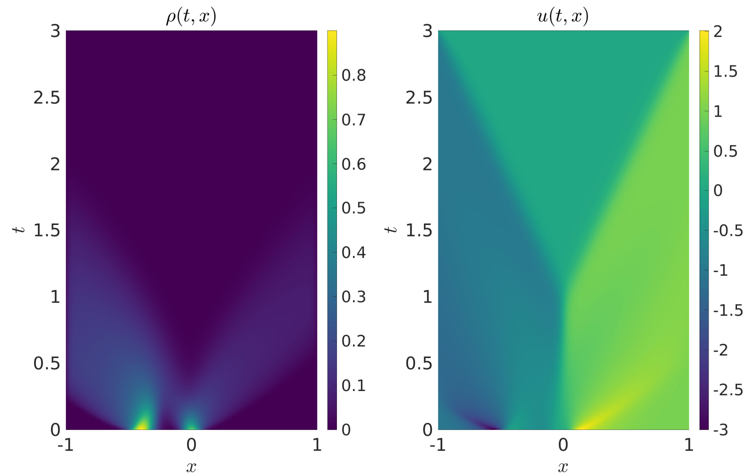

Figure 6.5: Evolution of the density $\rho(t,x)$ (left) and of the control $u(t,x)$ (right) up to final time $T = 3$ for the two-group crowd model described in Section 6.4.2 with $n = 1000$ spatial discretization points and $m = 250$ time steps.

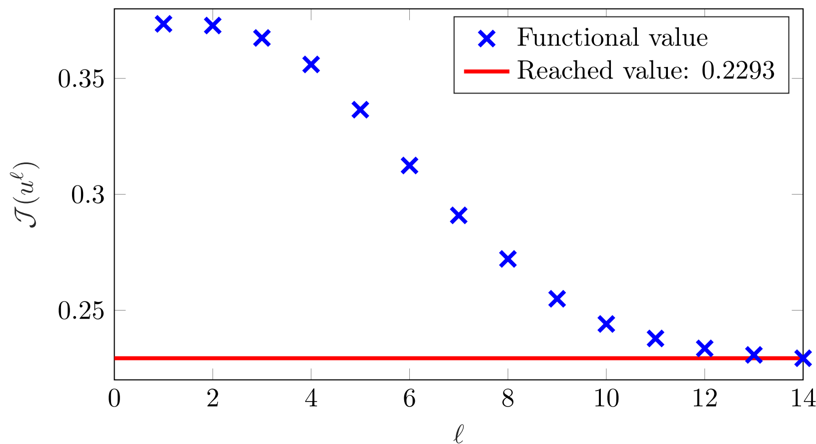

Figure 6.6: Value of the functional $\mathcal{J}(u^\ell)$ at the successive iterations of the steepest descent method for the two-group crowd model described in Section 6.4.2 ($n = 1000$ and $m = 250$).

is of Sznajd type $p(x,y) = (x^2 - 1)/20$, and the selective function is $s(t,x,\rho) = 1$. The penalization parameter is $\gamma = 0.1$ and the diffusion parameter is $\sigma = \sqrt{0.02}$. We discretize the problem with $n = 1000$ spatial grid points and $m = 200$ time steps, and we run the simulation up to the final time $T = 3$. We consider a terminal cost given by $c(T,x,\rho(T,x)) = |\rho(T,x) - \bar{\rho}(x)|^2$, which translates into $\psi_T(x) = \rho(T,x) - \bar{\rho}(x)$. In Figure 6.7 we plot the density functions at the initial and the final time, and we can observe that the initial density is correctly transported to the target one. In addition, in In addition, we show the values of the functional $\mathcal{J}(u^\ell)$ at the successive iterations of the steepest descent method. We observe that the method needs 33 iterations to reach the input tolerance $2 \cdot 10^{-3}$. Finally, in Figure 6.8 we present the evolution of the density and of the control. The overall computational time of this simulation is roughly 75 seconds.

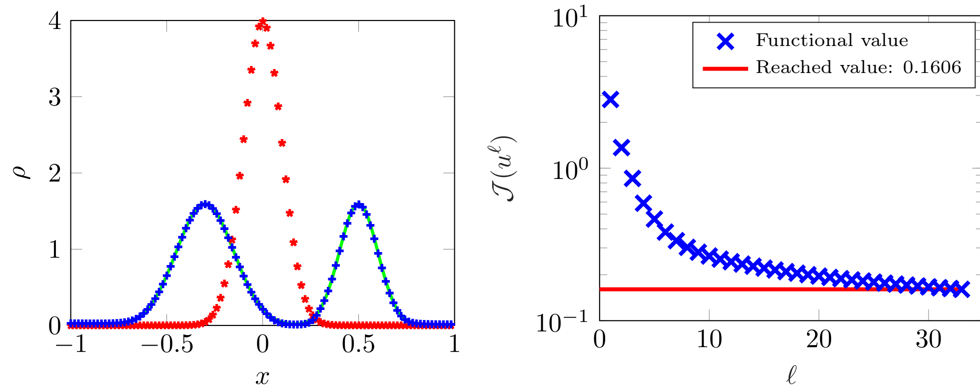

Figure 6.7: Left plot: density functions at initial time (red asterisks) and at final time (blue crosses) for the mass transfer problem described in Section 6.4.3 with $n = 1000$ spatial discretization points and $m = 200$ time steps. For plotting reasons, the target density is reported with a solid line, while the others are displayed each tenth point. Right plot: value of the functional $\mathcal{J}(u^\ell)$ at successive iterations of the steepest descent method.

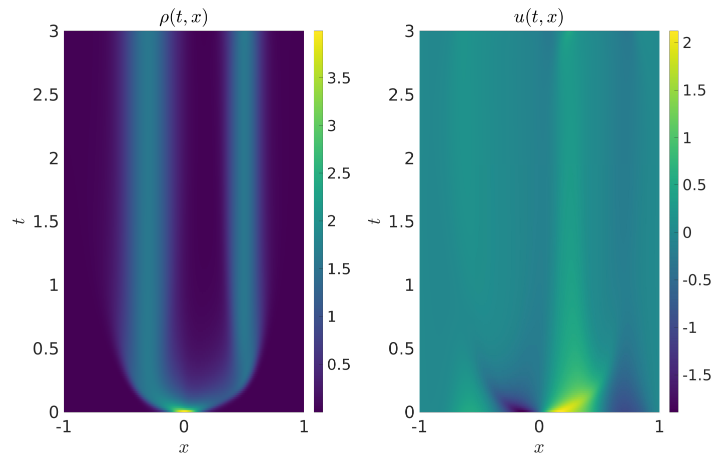

Figure 6.8: Evolution of the density $\rho(t, x)$ (left) and of the control $u(t, x)$ (right) up to final time $T = 3$ for the mass transfer problem described in Section 6.4.3 with $n = 1000$ spatial discretization points and $m = 200$ time steps.

6.5 Conclusions

We presented a mean-field optimal control model where the constraint is represented by a nonlinear PDE with non-local interaction term and diffusion describing the evolution of a continuum of agents. We provide, at a formal level, first order optimality conditions, resulting in a forward-backward coupled system with associated boundary conditions. Thus, a reduced gradient method is derived for the synthesis of the mean-field control, where the primal and adjoint equations are efficiently solved by using exponential integrators. Our proposed approach has been successfully tested on various examples from the literature, including models of opinion formation, pedestrian dynamics and mass transfer in the one-dimensional setting. In future works we plan to exploit the efficiency of exponential integrators to tackle higher dimensional problems (possibly using ad hoc techniques for tensor structured problems [39, 41, 42]) and scenarios where a fine spatial discretization is required to correctly capture the behavior of the controlled dynamics.

Chapter 7

Efficient exponential integration of inhomogeneous ADR equations

In this chapter, we propose a technique to efficiently integrate in time inhomogeneous evolutionary advection–diffusion–reaction equations with exponential integrators. The approach is based on the extraction from the original PDE of a constant coefficient diffusion part, which is determined by a linear stability analysis of the chosen temporal scheme. After semidiscretization in space, the arising system of ODEs can then be numerically integrated efficiently, by employing for example FFT-based or tensor μ-mode-based techniques. Also, we present two new exponential integrators of Lawson type (of first and second order), which appear to have better unconditional stability bounds compared to other well-known exponential integrators. The effectiveness of the approach is highlighted presenting numerical examples with up to three space dimensions.

The material of this chapter is an ongoing work temporarily named as in Reference [38], i.e., M. Caliari, F. C., L. Einkemmer and A. Ostermann. Efficient exponential integration of inhomogeneous evolutionary advection–diffusion–reaction equations. *In preparation*, 2023.

7.1 Introduction

Efficiently solving evolutionary Partial Differential Equations (PDEs) is of great interest for many fields of science and engineering. In particular, many physical and chemical phenomena can be effectively modeled by time-dependent advection–diffusion–reaction equations [113], which we consider in the following conservative form

$$\begin{cases} \partial_t u(t,\boldsymbol{x}) = \nabla\cdot(\lambda(\boldsymbol{x})\nabla u(t,\boldsymbol{x})) + r(t,\boldsymbol{x},u(t,\boldsymbol{x})), \quad t\in[0,T], \quad \boldsymbol{x}\in\Omega\subset\mathbb{R}^d,\\ u(0,\boldsymbol{x}) = u_0(\boldsymbol{x}), \end{cases} \tag{7.1}$$

coupled with suitable boundary conditions.

Many techniques and schemes have been developed in the past years to integrate numerically equations of the form (7.1), see Reference [113] for a comprehensive review. For instance, if the problem is considered with periodic boundary conditions and the diffusion coefficient is constant, i.e., $\lambda(\boldsymbol{x})=\lambda$, Fast Fourier Transform (FFT) based methods are appealing [116, 166]. Or, if problem (7.1) admits a Kronecker sum structure, after space discretization it is possible to employ μ-mode based techniques to efficiently compute the approximate solution [39, 42]. In general, due to the presence of spatial derivatives in the equation, integrators that do not have restrictions stemming from the stiffness should be employed. This is the case, for example, for IMplicit EXplicit (IMEX) schemes [15] or exponential integrators [112]. We will focus in this work on the latter class. For equations in generic form (7.1), the aforementioned FFT or μ-mode based techniques cannot be directly applied, mainly due to the presence of the inhomogeneous term $\lambda(\boldsymbol{x})$.

The idea is then to consider the *equivalent* formulation

$$\partial_t u(t,\boldsymbol{x}) = \Lambda u(t,\boldsymbol{x}) + \underbrace{(\nabla\cdot(\lambda(\boldsymbol{x})\nabla u(t,\boldsymbol{x})) - \Lambda u(t,\boldsymbol{x}) + r(t,\boldsymbol{x},u(t,\boldsymbol{x}))}_{g(t,\boldsymbol{x},u(t,\boldsymbol{x}))}, \tag{7.2}$$

where Λ is an operator which approximates the original advection–diffusion one but can be treated, computationally speaking, in a more efficient manner (see also Reference [53]). Depending on the specific time marching scheme under consideration this means that, for instance, the operators $(\mathcal{I}-\tau\Lambda)^{-1}$ or $\mathrm{e}^{\tau\Lambda}$, being τ the time step size, can be computed in a fast way (using FFT, μ-mode or semi-Lagrangian techniques, for example). The choice of Λ is clearly not uniquely determined. In particular, we will propose our scheme dependent choice, in the context of exponential integrators, based on a linear stability analysis of a simple diffusion equation. This is similar to what has been done in the literature for IMEX schemes [187].

The remaining part of the chapter is structured as follows. In Section 7.2, the main one, we introduce the prototypical equation that we use to analyze different exponential integrators and to determine the corresponding operator Λ. Also, we present there two newly designed schemes of Lawson type, that enjoy favorable unconditional stability properties. In Section 7.3 we validate our implementation and our choice of the approximation operator on one-dimensional examples, as well as we present performance results on a three-dimensional numerical example. Finally, we draw some conclusions in Section 7.4.

7.2 Linear stability analysis

As mentioned in the introduction, we investigate here a model equation to determine the approximation operator Λ in equation (7.2). In particular, we consider the constant coefficient heat equation

$$\partial_t u = \Delta u \tag{7.3}$$

on the domain $\Omega = [-\pi,\pi)^d$ with periodic boundary conditions, and we equivalently write

$$\partial_t u = \lambda\Delta u + (1-\lambda)\Delta u, \tag{7.4}$$

with $\lambda \in [0,1]$. Notice that in this case the approximation operator is simply $\lambda\Delta$. Then, in order to determine the parameter λ we perform a linear stability analysis of the temporal exponential integrator in Fourier space. For convenience of the reader, all the schemes mentioned and studied in this section are collected in the appendix (in a formulation for a generic abstract semilinear ODE).

Let us firstly consider the well-known exponential Euler method, which for equation (7.4) marches in time as follows

$$u^{n+1} = u^n + \tau\varphi_1(\tau\lambda\Delta)\Delta u^n, \tag{7.5}$$

being u^n the approximated solution at time t_n and τ the time step size. Here and throughout the chapter we assume, without loss of generality, that τ is constant. Then we have the following result.

Theorem 7.2.1. *The exponential Euler scheme* (7.5) *is unconditionally stable for* $\lambda \geq \lambda^{\mathrm{ee}} = 1/2$.

Proof. Let us denote $\boldsymbol{k} = (k_1,\ldots,k_d) \in \mathbb{Z}^d$, $k^2 = \sum_\mu k_\mu^2$, and let $\hat{u}_{\boldsymbol{k}}^n$ be the $\boldsymbol{k}$th Fourier mode of u^n. Then, in Fourier space we have

$$\frac{\hat{u}_{\boldsymbol{k}}^{n+1}}{\hat{u}_{\boldsymbol{k}}^n} = 1 - \varphi_1(-\lambda\tau k^2)\tau k^2 = 1 - \frac{1}{\lambda} + \frac{\mathrm{e}^{-\lambda\tau k^2}}{\lambda}.$$

Thus, we have unconditional stability if the growing factor is less or equal than one in absolute value, i.e., if

$$\left|\frac{\hat{u}_{\boldsymbol{k}}^{n+1}}{\hat{u}_{\boldsymbol{k}}^n}\right| \leq 1$$

for all $\boldsymbol{k}$. This implies $1/\lambda \leq 2$ which gives $\lambda \geq 1/2$, as desired. □

Remark 7.2.1. *If instead of considering the semidiscrete (in time) scheme* (7.5) *we work with the fully discrete version, i.e., the Laplacian operator is approximated by a matrix* $A \in \mathbb{R}^{N\times N}$*, being* N *the number of degrees of freedom in space, a similar bound in terms of the largest (in magnitude) eigenvalue can be derived. Moreover, notice that clearly* $\lambda < 1/2$ *is also admissible, accepting a stability constraint.*

Roughly speaking, the result of Theorem 7.2.1 tells us that we need to take at least 1/2 of the magnitude of the original diffusion in order to obtain an unconditionally stable method for equation (7.4). Let us consider now another example of interest, i.e., the exponential Lawson–Euler scheme which marches as

$$u^{n+1} = \mathrm{e}^{\tau\lambda\Delta}(u^n + \tau(1-\lambda)u^n). \tag{7.6}$$

It can alternatively be seen as a Lie splitting where we approximate the second subflow by explicit Euler, that is

$$\breve{u}^{n+1} = \mathrm{e}^{\tau\lambda\Delta}\mathrm{e}^{\tau(1-\lambda)\Delta}\breve{u}^n \approx \mathrm{e}^{\tau\lambda\Delta}(\breve{u}^n + \tau(1-\lambda)\Delta\breve{u}^n).$$

Then we have the following.

Theorem 7.2.2. *The Lawson–Euler scheme* (7.6) *is unconditionally stable for* $\lambda \geq \lambda^{\mathrm{le}} = 0.218$.

Proof. Similarly to the proof of Theorem 7.2.1, in Fourier space we have

$$\frac{\hat{u}_{\boldsymbol{k}}^{n+1}}{\hat{u}_{\boldsymbol{k}}^{n}} = \mathrm{e}^{-\lambda\tau k^2}(1-(1-\lambda)\tau k^2).$$

A straightforward study of the unconditional stability relation for the growing factor gives the following inequality to be satisfied

$$\mathrm{e}^{-\frac{1}{1-\lambda}}\left(1-\frac{1}{\lambda}\right)+1 \geq 0.$$

The result simply follows from numerical approximation of the root of the left hand side. □

This shows that there exist methods which are unconditionally stable for smaller values of $\lambda = \lambda^{\mathrm{ee}} = 1/2$. This is of interest because it is often the case that the accuracy of the method increases as λ decreases (see the numerical examples in Section 7.3). In addition, as mentioned in the introduction, the advantage of using schemes that just employ the exponential function is that in certain situations the exponential can be more efficient to compute than the φ-functions (e.g., exploiting the Kronecker structure or if a semi-Lagrangian scheme is used).

It is possible to improve even more the stability bound, by considering a simple variation of the Lawson–Euler scheme (7.6) applied to equation (7.4), i.e.,

$$u^{n+1} = u^n + \tau\mathrm{e}^{\tau\lambda\Delta}\Delta u^n. \tag{7.7}$$

We call this scheme *stabilized Lawson–Euler*, and it can be easily verified by comparing with a Taylor expansion of the exact solution that this is indeed a first order scheme. Then, we have the following result.

Theorem 7.2.3. *The stabilized Lawson–Euler scheme* (7.7) *applied to equation* (7.4) *is unconditionally stable for* $\lambda \geq \lambda^{\mathrm{sle}} = 1/(2\mathrm{e}) \approx 0.184$.

Proof. In this case, in Fourier space the growing factor is given by

$$\frac{\hat{u}_{\boldsymbol{k}}^{n+1}}{\hat{u}_{\boldsymbol{k}}^{n}} = 1 - \tau k^2 \mathrm{e}^{-\tau\lambda k^2},$$

and the result follows straightforwardly by imposing $|\hat{u}_{\boldsymbol{k}}^{n+1}/\hat{u}_{\boldsymbol{k}}^{n}| \leq 1$ for each $\boldsymbol{k}$. □

Remark 7.2.2. *A similar analysis can be performed also for other classes of schemes. For example, in Reference [187] some IMEX schemes have been analyzed in a fully discretized context. In particular, for the well known backward-forward Euler method*

$$(\mathcal{I}-\tau\lambda\Delta)u^{n+1}=(\mathcal{I}+\tau(1-\lambda)\Delta)u^n \tag{7.8}$$

they obtain the unconditional stability bound $\lambda \geq \lambda^{\mathrm{im1}}=1/2$. *Moreover, they propose the second order method*

$$\begin{aligned}\left(\mathcal{I}-\frac{\tau}{2}\lambda\Delta\right)U&=\left(\mathcal{I}+\frac{\tau}{2}(1-\lambda)\Delta\right)u^n,\\ \left(\mathcal{I}-\frac{\tau}{2}\lambda\Delta\right)u^{n+1}&=\left(\mathcal{I}+\frac{\tau}{2}\lambda\Delta\right)u^n+\tau(1-\lambda)\Delta U,\end{aligned} \tag{7.9}$$

which has the same bound $\lambda \geq \lambda^{\mathrm{im2}}=1/2$.

Let us now consider some examples of second order exponential integrators. In particular, we start with the following class of Runge–Kutta type schemes for equation (7.4), that we name *expRK2phi2*,

$$\begin{aligned}U&=u^n+c_2\tau\varphi_1(c_2\tau\lambda\Delta)\Delta u^n,\\ u^{n+1}&=u^n+\tau\varphi_1(\tau\lambda\Delta)\Delta u^n+\frac{\tau}{c_2}\varphi_2(\tau\lambda\Delta)(1-\lambda)\Delta(U-u^n),\end{aligned} \tag{7.10}$$

where $0<c_2\leq 1$ is a free parameter. Then we have the following result.

Theorem 7.2.4. *The expRK2phi2 scheme* (7.10) *is unconditionally stable for* $\lambda \geq \lambda^{\mathrm{erk2p2}}=1/(1+c_2)$.

Proof. Similarly to what performed for the first order schemes, in Fourier space we have (setting $x=\tau k^2$)

$$\frac{\hat{u}_{\boldsymbol{k}}^{n+1}}{\hat{u}_{\boldsymbol{k}}^{n}}=1-x\varphi_1(-\lambda x)+x^2(1-\lambda)\varphi_2(-\lambda x)\varphi_1(-c_2\lambda x).$$

Then, by imposing $|\hat{u}_{\boldsymbol{k}}^{n+1}/\hat{u}_{\boldsymbol{k}}^{n}|\leq 1$ for each $\boldsymbol{k}$ we have

$$1-\frac{1}{\lambda}+\frac{1-\lambda}{c_2\lambda^2}\leq 1$$

for which we obtain

$$\lambda\geq\frac{1}{1+c_2}$$

as desired. □

The best results, in terms of unconditional stability bound, are obtained by setting the free parameter $c_2=1$, which yields $\lambda\geq 1/2$.

Let us consider now another class of second order Runge–Kutta exponential integrators, which requires just the φ_1 function and for which the stiff order conditions are satisfied in weaker form. For model equation (7.4) we have

$$\begin{aligned}U&=u^n+c_2\tau\varphi_1(c_2\tau\lambda\Delta)\Delta u^n,\\ u^{n+1}&=u^n+\tau\varphi_1(\tau\lambda\Delta)\Delta u^n+\frac{\tau}{2c_2}\varphi_1(\tau\lambda\Delta)(1-\lambda)\Delta(U-u^n).\end{aligned} \tag{7.11}$$

We label this class, dependent on the parameter $0<c_1\leq 1$, as *expRK2phi1*, and we have the following result for the unconditional stability.

Theorem 7.2.5. *The expRK2phi1 scheme* (7.11) *is unconditionally stable for* $\lambda\geq\lambda^{\mathrm{erk2p1}}=1/(1+2c_2)$.

Proof. In Fourier space, the growing factor is in this case given by (setting $x = \tau k^2$)

$$\frac{\hat{u}_{\boldsymbol{k}}^{n+1}}{\hat{u}_{\boldsymbol{k}}^{n}} = 1 - x\varphi_1(-\lambda x) + \frac{x^2}{2}(1-\lambda)\varphi_1(-\lambda x)\varphi_1(-c_2\lambda x)$$

for which we obtain

$$\lambda \geq \frac{1}{1+2c_2}$$

as desired. □

The choice $c_2 = 1$ leads to $\lambda \geq 1/3$, which is better than the bound obtained for the expRK2phi2 class.

Concerning the second order Lawson type schemes, we consider the *Lawson2a* and the *Lawson2b* integrators, which for equation (7.4) march in time as

$$\begin{aligned} U &= \mathrm{e}^{\frac{\tau}{2}\lambda\Delta}u^n + \frac{\tau}{2}\mathrm{e}^{\frac{\tau}{2}\lambda\Delta}(1-\lambda)\Delta u^n, \\ u^{n+1} &= \mathrm{e}^{\tau\lambda\Delta}u^n + \tau\mathrm{e}^{\frac{\tau}{2}\lambda\Delta}(1-\lambda)\Delta U, \end{aligned} \tag{7.12}$$

and

$$\begin{aligned} U &= \mathrm{e}^{\tau\lambda\Delta}u^n + \tau\mathrm{e}^{\tau\lambda\Delta}(1-\lambda)\Delta u^n, \\ u^{n+1} &= \mathrm{e}^{\tau\lambda\Delta}u^n + \frac{\tau}{2}\mathrm{e}^{\tau\lambda\Delta}(1-\lambda)\Delta u^n + \frac{\tau}{2}(1-\lambda)\Delta U, \end{aligned} \tag{7.13}$$

respectively. In terms of unconditional stability, we have the following result.

Theorem 7.2.6. *The Lawson2a scheme* (7.12) *and the Lawson2b scheme* (7.13) *are unconditionally stable for* $\lambda \geq \lambda^{\mathrm{l2a}} = \lambda^{\mathrm{l2b}} = 0.301$.

Proof. The stability relations in Fourier space of the schemes are given by

$$\left|\mathrm{e}^{-\lambda x} - x\mathrm{e}^{-\lambda\frac{x}{2}}(1-\lambda)\left(\mathrm{e}^{-\lambda\frac{x}{2}} - \frac{x(1-\lambda)}{2}\mathrm{e}^{-\lambda\frac{x}{2}}\right)\right| \leq 1$$

and

$$\left|\mathrm{e}^{-\lambda x}\left(1 - \frac{x}{2}(1-\lambda) - \frac{x}{2}(1-\lambda)(1-x(1-\lambda))\right)\right| \leq 1$$

for Lawson2a and Lawson2b, respectively. In both cases, a numerical calculation leads to the restriction $\lambda \geq 0.301$ for unconditional stability. □

Finally, similarly to what we did for the first order Lawson–Euler scheme, we consider a stabilized version for integrators of Lawson type. In particular, letting $0 < \alpha \leq 1$ be a free parameter, we obtain for model equation (7.4) the class

$$\begin{aligned} U &= u^n + \alpha\tau\mathrm{e}^{\alpha\tau\lambda\Delta}\Delta u^n, \\ u^{n+1} &= u^n + \tau\mathrm{e}^{\frac{\tau}{2}\lambda\Delta}\Delta u^n + \frac{\tau}{2\alpha}\mathrm{e}^{\tau\lambda\Delta}(1-\lambda)\Delta(U-u^n), \end{aligned} \tag{7.14}$$

that we name *stabilized Lawson2* and which can easily be seen of second order. Then we obtain the following result.

Theorem 7.2.7. *The stabilized Lawson2 scheme* (7.14) *with* $\alpha = 1/4$ *is unconditionally stable for* $\lambda \geq \lambda^{\mathrm{sl2}} = 0.197$.

Proof. In this case, the stability relation in Fourier space is given by

$$\left|1 - x\mathrm{e}^{-\frac{\lambda x}{2}} + \frac{x^2}{2}(1-\lambda)\mathrm{e}^{-\lambda x}\mathrm{e}^{-\alpha\lambda x}\right| \leq 1.$$

Then, setting $\alpha = 1/4$, a numerical calculation leads to the restriction $\lambda \geq 0.197$ for unconditional stability, as stated. □

Remark 7.2.3. *If for instance we set $\alpha = 1/2$, we numerically obtain the restriction $\lambda \geq 0.266$, which is more stringent than the one stated in Theorem 7.2.7. A more deep investigation, letting α have non-rational values (which is clearly admissible), could lead to an even improved bound with respect to the one in Theorem 7.2.7. In particular, by numerical observations, we claim that the lowest value for α lies in the interval $(1/4, 1/3)$, and that leads to a restriction comparable with the one of the first order stabilized Lawson–Euler scheme. For simplicity and compactness of presentation, we do not investigate further this possibility here.*

A recap table of the methods, in descending order with respect to the stability restriction, is given in Table 7.1.

	order	uncond. stab. lower bound λ^*
IMEX2	2	$\lambda^{\mathrm{im2}} = 1/2$
IMEX1	1	$\lambda^{\mathrm{im1}} = 1/2$
expRK2phi2 ($c_2 = 1$)	2	$\lambda^{\mathrm{erk2p2}} = 1/2$
Exponential Euler	1	$\lambda^{\mathrm{ee}} = 1/2$
expRK2phi1 ($c_2 = 1$)	2	$\lambda^{\mathrm{erk2p1}} = 1/3$
Lawson2a	2	$\lambda^{\mathrm{l2a}} = 0.301$
Lawson2b	2	$\lambda^{\mathrm{l2b}} = 0.301$
Lawson–Euler	1	$\lambda^{\mathrm{le}} = 0.218$
Stabilized Lawson2 ($\alpha = 1/4$)	2	$\lambda^{\mathrm{sl2}} = 0.197$
Stabilized Lawson–Euler	1	$\lambda^{\mathrm{sle}} = 1/(2\mathrm{e}) \approx 0.184$

Table 7.1: Collection of methods, in descending order in terms of unconditional stability. Floating point notation means values obtained by numerical approximations.

The linear stability analysis just presented has been performed on the simple model equation (7.4). However, it gives us a good indication on what to employ as approximation operator in the more general formulation (7.2). In particular, our proposal is to choose

$$\Lambda = \lambda_{\max}\Delta = \lambda \max_{\boldsymbol{x}\in\Omega} \lambda(\boldsymbol{x})\Delta, \tag{7.15}$$

being $\lambda \in [\lambda^*, 1]$, where λ^* is one of the values summarized in Table 7.1, depending on the chosen time marching scheme.

7.3 Numerical examples

We present in this section a numerical validation of the bounds derived in Section 7.2. In particular, we will consider two different one-dimensional equations, one linear and one nonlinear, both with spatially variable diffusion coefficients. Moreover, we present performance results of the proposed technique for a three-dimensional advection–diffusion–reaction equation. All the numerical experiments have been performed on an Intel® Core™ i7-10750H CPU with six physical cores and 16GB of RAM, using MATLAB programming language. As a software, we employ MathWorks MATLAB® R2022a.

7.3.1 One-dimensional linear diffusion equation

We start by considering the following one-dimensional linear diffusion equation with space dependent coefficients

$$\begin{cases} \partial_t u(t,x) = \lambda(x)\partial_{xx} u(t,x), \quad x \in [-\pi,\pi),\ t \in [0,T], \\ u(0,x) = \sin(x), \end{cases} \tag{7.16}$$

completed with periodic boundary conditions. Here $\lambda(x) = 1 + 10\sin^2(x)$ is the space dependent diffusion coefficient.

We rewrite equation (7.16) as

$$\partial_t u(t,x) = \underbrace{\lambda_{\max}\partial_{xx}u(t,x)}_{\Lambda u(t,x)} + \underbrace{(\lambda(x)-\lambda_{\max})\partial_{xx}u(t)}_{g(t,u(t,x))}, \tag{7.17a}$$

where

$$\lambda_{\max} = \lambda \max_x \lambda(x), \quad \lambda \in [0,1]. \tag{7.17b}$$

The structure of the equation allows for an effective discretization in space by means of a Fourier spectral technique. In particular we denote with N the number of Fourier modes. Then, the temporal schemes studied in Section 7.2 and resumed in the appendix can be applied in a straightforward manner, with the computation of derivatives and matrix functions by pointwise operations on Fourier coefficients.

Here, we verify that the theoretical lower bounds for λ found in Section 7.2 also apply to this space dependent coefficients diffusion equation. The actual simulations have been performed with $N = 2^{11}$ and $T = 1/40$, and the achieved errors for different λ and varying number of time steps $m = 2^\ell$, with $\ell = 4, 5, \ldots, 11$, have been measured at the final time T in infinity norm, relatively with respect to the exact solution of equation (7.16). The results are collected in Figure 7.1. First of all, we observe that all the considered exponential methods show the expected order of convergence (in particular also the newly derived schemes of first and second order labeled stabilized Lawson–Euler and stabilized Lawson2, respectively). Then, we also clearly see that as λ decreases some methods fail to be unconditionally stable. In particular, if we compare with the bounds resumed in Table 7.1, we observe that the values found can be applied sharply also to the case of equation (7.16). Indeed, up to $\lambda = 0.5$ all the methods behave nicely. Then, if we further decrease λ, some methods start to blow up when incrementing the number of time steps, in accordance to what presented in Table 7.1. Then, as predicted by the linear analysis, for $\lambda < \lambda^{\text{sle}}$ all the methods fail to be unconditionally stable.

Finally, we notice that each method becomes more and more precise as λ decreases, and we compare in Figure 7.2 the achieved errors of the exponential schemes using their own values λ^*. In this case, we observe that basically all the first order methods behave similarly, while, for the second order methods, the best performant in terms of achieved final error is the stabilized Lawson2 scheme.

7.3.2 One-dimensional diffusion–reaction equation

We now turn our attention to the following one-dimensional nonlinear diffusion–reaction equation

$$\begin{cases} \partial_t u(t,x) = \partial_x(\lambda(x)\partial_x u(t,x)) + r(u), & x \in [-\pi,\pi),\ t \in [0,T], \\ u(0,x) = \sin(x), \end{cases} \tag{7.18}$$

in an inhomogeneous media with periodic boundary conditions, see Reference [152]. Here we select $\lambda(x) = 1 + 10\sin^2(x)$ as space dependent diffusion coefficient, and the nonlinearity is of quadratic type $r(u) = u(1-u)$. Similarly to the previous example, we rewrite equation (7.18) as

$$\partial_t u(t,x) = \underbrace{\lambda_{\max}\partial_{xx}u(t,x)}_{\Lambda u(t,x)} + \underbrace{(\lambda(x)-\lambda_{\max})\partial_{xx}u(t) + \lambda'(x)\partial_x u(t,x) + r(u)}_{g(t,u(t,x))}, \tag{7.19a}$$

where again

$$\lambda_{\max} = \lambda \max_x \lambda(x), \quad \lambda \in [0,1]. \tag{7.19b}$$

We first discretize in space with a Fourier spectral method employing $N = 2^{10}$ modes. Then, as for the previous example, the temporal schemes can be applied straightforwardly with pointwise operations on Fourier coefficients. We simulate until final time $T = 1/10$ with a number of time steps equal to $m = 2^{12}$ for all the methods, different values of λ and we measure the relative errors in infinity norm with respect to a reference solution computed with expRK2phi2 as time integrator (applied to the original equation semidiscretized in space by using spectral differentiation matrices and a sufficiently large number of time steps). We collect the results in Figure 7.3.

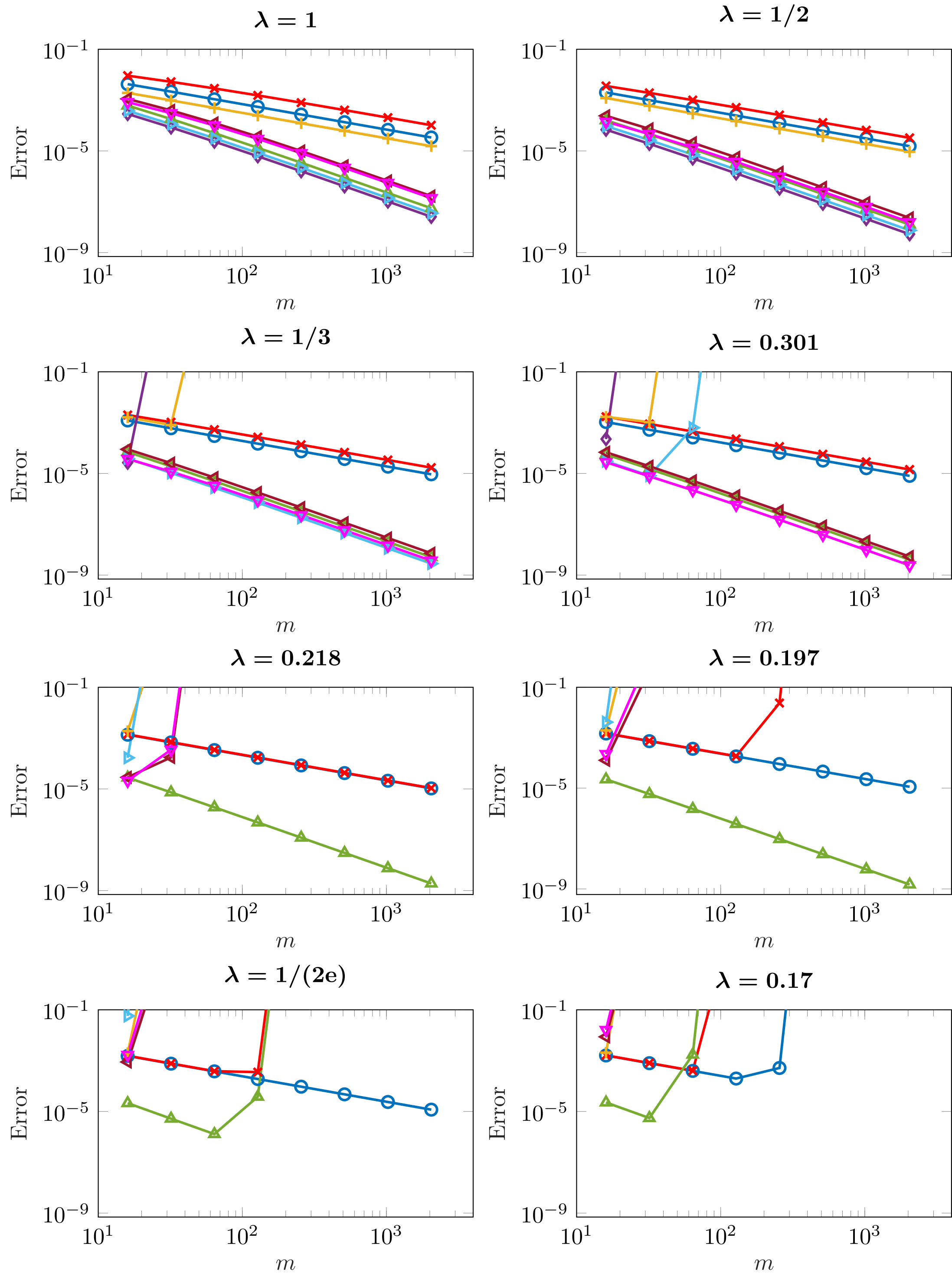

Figure 7.1: Solution of equation (7.16), rewritten as equation (7.17), with different schemes, decreasing λ and varying number of time steps m. The blue $\circ$ line is stabilized Lawson–Euler, the red $\times$ line is Lawson–Euler, the yellow $+$ line is exponential Euler, the purple $\diamond$ line is expRK2phi2 ($c_2 = 1$), the green $\triangle$ line is stabilized Lawson2, the light blue $\triangleright$ line is expRK2phi1 ($c_2 = 1$), the brown $\triangleleft$ line is Lawson2a and the pink $\triangledown$ line is Lawson2b (see also the legend in Figure 7.2).

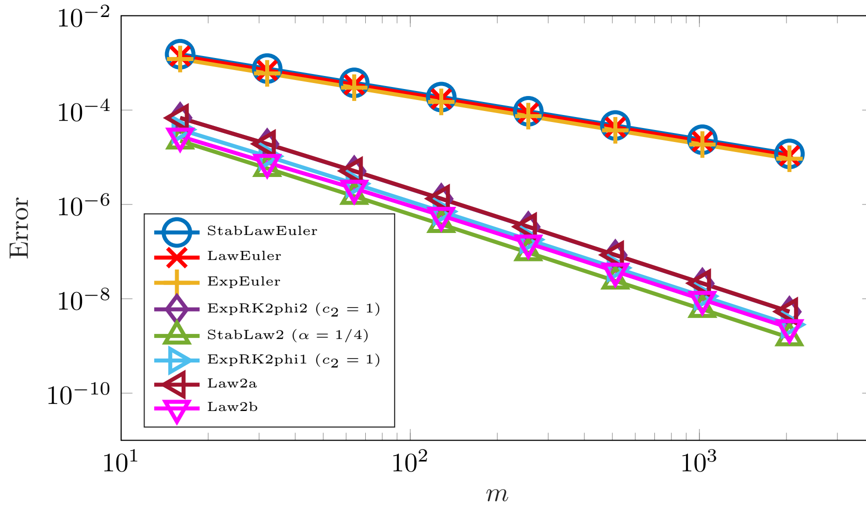

Figure 7.2: Solution of equation (7.16), rewritten as equation (7.17), with different schemes and varying number of time steps m. Each method has been run with its own lower bound λ^* given in Table 7.1.

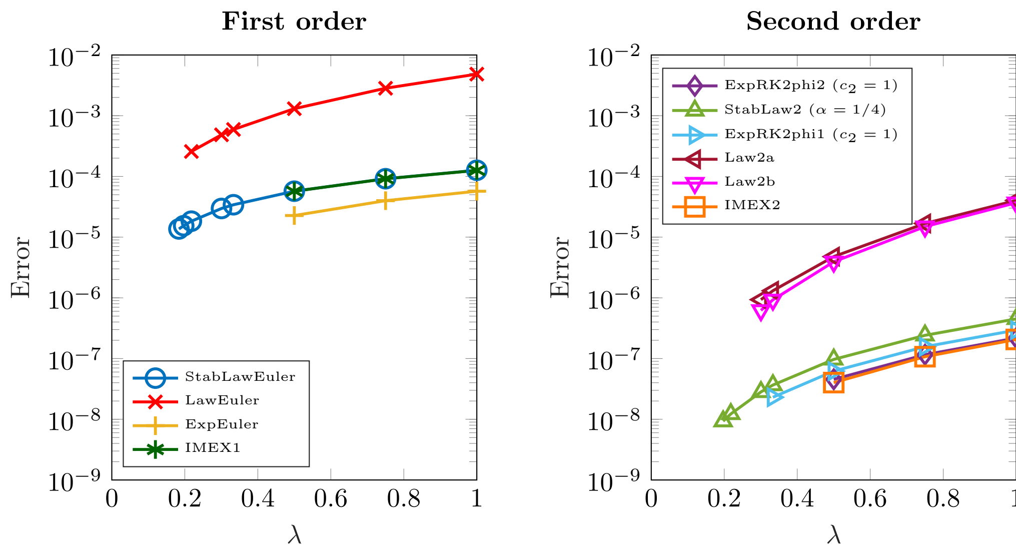

Figure 7.3: Solution of equation (7.18), rewritten as equation (7.19), with different schemes and varying value λ. The number of time steps is fixed to $m = 2^{12}$ for each method. Missing marks means that the error was above the plot threshold 10^{-2}.

Also in this nonlinear case, we observe that the linear analysis predicts very sharply the amount of diffusion that we can consider in the operator Λ while keeping unconditional stability. Moreover, in this example as well, each method becomes more precise as the value λ decreases, with in general a greater gain for second order methods than the first order ones. For completeness, we added also the results obtained with the IMEX schemes mentioned in Remark 7.2.2. Overall, in terms of achieved accuracy, we observe that the best performant methods are the stabilized Lawson–Euler and the stabilized Lawson2 schemes among the methods of order one and of order two, respectively.

7.3.3 Three-dimensional advection–diffusion–reaction equation

We now consider the following three-dimensional advection–diffusion–reaction equation

$$\begin{cases}\partial_t u(t,x_1,x_2,x_3) = \lambda(x_1,x_2,x_3)\Delta u(t,x_1,x_2,x_3) + \beta\nabla u(t,x_1,x_2,x_3) + r(u(t,x_1,x_2,x_3)),\\ u(0,x_1,x_2,x_3) = 2^6 x_1x_2x_3(1-x_1)^2(1-x_2)^2(1-x_3)^2,\end{cases} \tag{7.20}$$

in the spatial domain $\Omega = [0,1]^3$ and $t \in [0,T]$. We equip the problem with homogeneous Dirichlet boundary conditions on the set $\{(x_1,x_2,x_3) \in \partial\Omega : x_1x_2x_3 = 0\}$ and with homogeneous Neumann boundary conditions elsewhere. We set the diffusion coefficient as

$$\lambda(x_1,x_2,x_3) = 0.1\mathrm{e}^{-(x_1-1/2)^2-(x_2-1/2)^2-(x_3-1/2)^2},$$

while we choose the advection parameter as $\beta = -0.01$. Finally, we consider the quadratic nonlinearity $r(u) = u(1-u)$.

Based on the numerical results of the previous sections, we apply here the time marching schemes with the technique presented in Section 7.2, and we rewrite equation (7.20) as

$$\partial_t u(t,x_1,x_2,x_3) = \underbrace{(\Lambda^* + \beta(\partial_{x_1}+\partial_{x_2}+\partial_{x_3}))u(t,x_1,x_2,x_3)}_{\mathcal{M}u(t,x_1,x_2,x_3)} + g(x_1,x_2,x_3,u(t,x_1,x_2,x_3)), \tag{7.21}$$

where Λ^* is defined as in equation (7.15) with

$$\lambda_{\max} = \lambda^*_{\max} = \lambda^* \max_{\boldsymbol{x}\in\Omega} \lambda(\boldsymbol{x})$$

and

$$g(x_1,x_2,x_3,u(t,x_1,x_2,x_3)) = (\lambda(x_1,x_2,x_3) - \lambda^*_{\max})\Delta u(t,x_1,x_2,x_3) + r(u(t,x_1,x_2,x_3)).$$

The employment of different choices of $\lambda_{\max}$ is subject of current study. We discretize in space the equation with standard second order centered finite differences, using $N_{x_1} = N_{x_2} = N_{x_3} = 60$ discretization points for each direction (which leads to a space discretization error of approximately 10^{-4}). By doing so, the linear operator $\mathcal{M}$ in equation (7.21) becomes a matrix with Kronecker sum structure

$$M = M_3 \oplus M_2 \oplus M_1,$$

where $M_\mu \in \mathbb{R}^{N_{x_\mu}\times N_{x_\mu}}$ is the discretization matrix of the operator $\lambda^*_{\max}\partial_{x_\mu x_\mu} + \beta\partial_{x_\mu}$. Here the symbol $\oplus$ denotes the standard Kronecker sum of matrices. Hence, for the exponential integrators that just require the exponential function (i.e., the ones of Lawson type), it is possible to employ μ-mode based techniques in order to efficiently compute the needed actions of the matrix exponential via Tucker operators, see References [39, 42] for more details.

As terms of comparison, we consider here the same time integrators but applied to the original equation (7.20), again spatially discretized with second order centered finite differences and $N_{x_1} = N_{x_2} = N_{x_3} = 60$. In addition, in this formulation, we also present the results obtained with the exponential Euler scheme and with the expRK2phi1 integrator (with $c_2 = 1$). In order to compute the needed actions of exponentials and φ-functions, we employ the `kiops` function[1], whose underlying algorithm is thoroughly presented in Reference [97]. This routine requires an input tolerance, which we set as $\tau^{p+1}/100$, being τ the time step size and p the order of convergence of the time marching scheme. Moreover, we also perform the time integration with the IMEX schemes mentioned in Remark 7.2.2, and reported in the appendix for an abstract semilinear equation. The arising linear systems are solved with an iterative method, namely the biconjugate gradient stabilized method (implemented in the internal MATLAB function `bicgstab`). Also for this routine we set the input tolerance as $\tau^{p+1}/100$, similarly to the previous case.

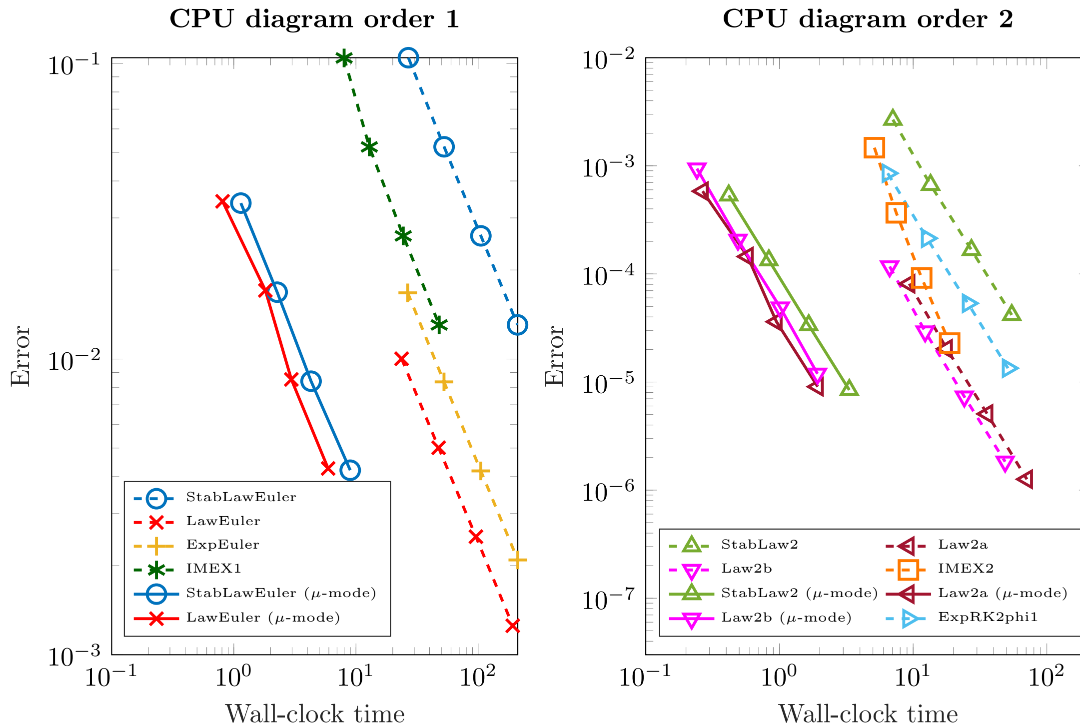

Figure 7.4: Results for the simulations of equation (7.20) (dashed lines), eventually rewritten as equation (7.21) (solid lines), with different integrators and varying number of time steps $m = 2^{\ell_1}$, with $\ell_1 = 9, \ldots, 12$ (left), and $m = 2^{\ell_2}$, with $\ell_2 = 6, \ldots, 9$ (right).

We perform the simulations with a number of time integration steps equal to 2^{ℓ_1}, with $\ell_1 = 9, \ldots, 12$, for the first order methods, while we consider 2^{ℓ_2}, with $\ell_2 = 6, \ldots, 9$, for the second order ones. The final time is in any case set to $T = 1/4$. The results are collected in the CPU diagram of Figure 7.4.

First of all, concerning the schemes of order 1, we observe that in the original formulation (i.e, the dashed lines in the plot) the Lawson–Euler scheme is the one which performs better. Indeed, comparing it with the other exponential methods, it is the one which reaches the lowest error, with all the computational times similar one to each other. The IMEX1 method performs slightly better in term of wall-clock time, but the reached error is higher than both the exponential Euler and the Lawson–Euler scheme. Hence, overall, it is not the preferred method. In any case, we observe that the proposed approach, i.e., solving instead equation (7.21) with μ-mode techniques (solid lines) is effective. Indeed, in this formulation the Lawson–Euler and the stabilized Lawson–Euler methods reach comparable errors, with a slight advantage of the former in terms of computational time.

Similar considerations can be drawn for the second order schemes. Indeed, in the original formulation the Lawson2a and the Lawson2b schemes perform better than the IMEX2 scheme, the expRK2phi1 and the stabilized Lawson2 methods. Then, the μ-mode approach applied to equation (7.21) allows to obtain comparable errors in less wall-clock time, and hence appears to be an effective choice.

7.4 Conclusions

In this chapter, we presented an effective approach to solve inhomogeneous advection–diffusion–reaction equations. It is based on an equivalent rewriting of the equation which allows for the employment of more efficient methods (FFT or μ-mode based, for instance) to integrate it numerically. By working in

[1]https://gitlab.com/stephane.gaudreault/kiops/-/tree/master/

the context of exponential integrators, we also presented two schemes of Lawson type which have better unconditional stability properties compared to methods already available. The conducted numerical examples show the superiority of the approach. A more sophisticated choice of the approximation operator is currently under study.

Appendix

For convenience of the reader, we list here the integrators that have been mentioned and studied throughout the chapter. We suppose here that the equation under study is given in the following abstract form

$$u'(t) = \mathcal{M}u(t) + g(t, u(t)) = F(t, u(t)),$$

being $\mathcal{M}$ a generic linear operator and g a nonlinear function.

Lawson type exponential integrators

The *Lawson–Euler* scheme is given by

$$u^{n+1} = \mathrm{e}^{\tau\mathcal{M}}(u^n + \tau g(t_n, u^n)).$$

The *stabilized Lawson–Euler* scheme is given by

$$u^{n+1} = u^n + \tau \mathrm{e}^{\tau\mathcal{M}} F(t_n, u^n).$$

The *Lawson2a* scheme is given by

$$U = \mathrm{e}^{\frac{\tau}{2}\mathcal{M}}u^n + \frac{\tau}{2}\mathrm{e}^{\frac{\tau}{2}\mathcal{M}}g(t_n, u^n), \quad u^{n+1} = \mathrm{e}^{\tau\mathcal{M}}u^n + \tau\mathrm{e}^{\frac{\tau}{2}\mathcal{M}}g\left(t_n + \frac{\tau}{2}, U\right).$$

The *Lawson2b* scheme is given by

$$U = \mathrm{e}^{\tau\mathcal{M}}u^n + \tau\mathrm{e}^{\tau\mathcal{M}}g(t_n, u^n), \quad u^{n+1} = \mathrm{e}^{\tau\mathcal{M}}u^n + \frac{\tau}{2}\mathrm{e}^{\tau\mathcal{M}}g(t_n, u^n) + \frac{\tau}{2}g(t_n + \tau, U).$$

The *stabilized Lawson2* scheme is given by

$$U = u^n + \alpha\tau\mathrm{e}^{\alpha\tau\mathcal{M}}F(t_n, u^n), \quad u^{n+1} = u^n + \tau\mathrm{e}^{\frac{\tau}{2}\mathcal{M}}F(t_n, u^n) + \frac{\tau}{2\alpha}\mathrm{e}^{\tau\mathcal{M}}(g(t_n + \alpha\tau, U) - g(t_n, u^n)).$$

Classical exponential integrators

The *exponential Euler* scheme is given by

$$u^{n+1} = u^n + \tau\varphi_1(\tau\mathcal{M})F(t_n, u^n).$$

The *exponential Runge–Kutta* scheme of second order involving φ_1 and φ_2 functions is given by

$$\begin{aligned} U &= u^n + c_2\tau\varphi_1(c_2\tau\mathcal{M})F(t_n, u^n), \\ u^{n+1} &= u^n + \tau\varphi_1(\tau\mathcal{M})F(t_n, u^n) + \frac{\tau}{c_2}\varphi_2(\tau\mathcal{M})(g(t_n + c_2\tau, U) - g(t_n, u^n)). \end{aligned}$$

The *exponential Runge–Kutta* scheme of second order involving just the φ_1 function is given by

$$\begin{aligned} U &= u^n + c_2\tau\varphi_1(c_2\tau\mathcal{M})F(t_n, u^n), \\ u^{n+1} &= u^n + \tau\varphi_1(\tau\mathcal{M})F(t_n, u^n) + \frac{\tau}{2c_2}\varphi_1(\tau\mathcal{M})(g(t_n + c_2\tau, U) - g(t_n, u^n)). \end{aligned}$$

IMEX schemes

The *Backward-Forward Euler* scheme is given by

$$(\mathcal{I} - \tau\mathcal{M})u^{n+1} = u^n + \tau g(t_n, u^n).$$

The second order IMEX scheme given in Reference [187] is given by

$$\left(\mathcal{I} - \frac{\tau}{2}\mathcal{M}\right) U = u^n + \frac{\tau}{2} g(t_n, u^n), \quad \left(\mathcal{I} - \frac{\tau}{2}\mathcal{M}\right) u^{n+1} = u^n + \frac{\tau}{2}\mathcal{M}u^n + \tau g\left(t_n + \frac{\tau}{2}, U\right).$$

Bibliography

[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. Cited at page 21.

[2] H. Abou-Kandil, G. Freiling, V. Ionescu, and G. Jank. *Matrix Riccati Equations in Control and Systems Theory.* Springer, Basel, 2003. Cited at page 56.

[3] Y. Achdou and I. Capuzzo-Dolcetta. Mean Field Games: Numerical Methods. *SIAM J. Numer. Anal.*, 48(3):1136–1162, 2010. Cited at page 92.

[4] M. Aduamoah, B. D. Goddard, J. W. Pearson, and J. C. Roden. Pseudospectral methods and iterative solvers for optimization problems from multiscale particle dynamics. *BIT Numer. Math.*, 62:1703–1743, 2022. Cited at page 92.

[5] 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. Cited at pages 14, 32, 49, 56, 97.

[6] 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. Cited at pages 10, 13, 32, 44, 47, 56, 97.

[7] G. Albi, N. Bellomo, L. Fermo, S.-Y. Ha, J. Kim, L. Pareschi, D. Poyato, and J. Soler. Vehicular traffic, crowds, and swarms: From kinetic theory and multiscale methods to applications and research perspectives. *Math. Models Methods Appl. Sci.*, 29(10):1901–2005, 2019. Cited at page 91.

[8] G. Albi, M. Bongini, E. Cristiani, and D. Kalise. Invisible Control of Self-Organizing Agents Leaving Unknown Environments. *SIAM J. Appl. Math.*, 76(4):1683–1710, 2016. Cited at page 91.

[9] G. Albi, M. Caliari, E. Calzola, and F. Cassini. Exponential integrators for mean-field selective optimal control problems. *arXiv preprint arXiv:2302.00127*, 2023. Cited at page 91.

[10] G. Albi, Y.-P. Choi, M. Fornasier, and D. Kalise. Mean Field Control Hierarchy. *Appl. Math. Optim.*, 76:93–135, 2017. Cited at pages 92, 93, 94, 96, 98.

[11] G. Albi, M. Herty, and L. Pareschi. Kinetic description of optimal control problems and applications to opinion consensus. *Commun. Math. Sci.*, 13(6):1407–1429, 2015. Cited at page 91.

[12] G. Albi, M. Herty, and L. Pareschi. Linear multistep methods for optimal control problems and applications to hyperbolic relaxation systems. *Appl. Math. Comput.*, 354:460–477, 2019. Cited at page 92.

[13] P. Arbenz and L. Říha. Batched transpose-free ADI-type preconditioners for a Poisson solver on GPGPUs. *J. Parallel Distrib. Comput.*, 137:148–159, 2020. Cited at page 34.

[14] 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. Cited at page 8.

[15] U. M. Ascher, S. J. Ruuth, and B. T. R. Wetton. Implicit-Explicit Methods for Time-Dependent Partial Differential Equations. *SIAM J. Numer. Anal.*, 32:797–823, 1995. Cited at page 103.

[16] S. Ashby et al. The Opportunities and Challenges of Exascale Computing. Technical report, ASCAC Subcommittee on Exascale Computing, U. S. Department of Energy, 2010. Cited at page 21.

[17] N. Auer, L. Einkemmer, P. Kandolf, and A. Ostermann. Magnus integrators on multicore CPUs and GPUs. *Comput. Phys. Commun.*, 228:115–122, 2018. Cited at page 21.

[18] B. W. Bader, T. G. Kolda, et al. Tensor Toolbox for MATLAB, Version 3.2.1. `https://www.tensortoolbox.org`, April, 2021. Cited at page 29.

[19] R. Bailo, M. Bongini, J. A. Carrillo, and D. Kalise. Optimal consensus control of the Cucker-Smale model. *IFAC-PapersOnLine*, 51(13):1–6, 2018. Cited at pages 92, 94.

[20] W. Bao, H. Li, and J. Shen. 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. Cited at page 36.

[21] 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. Cited at page 12.

[22] J.-D. Benamou and Y. Brenier. A computational fluid mechanics solution to the Monge-Kantorovich mass transfer problem. *Numer. Math.*, 84:375–393, 2000. Cited at page 100.

[23] H. Berland, B. Skaflestad, and W. M. Wright. EXPINT—a MATLAB Package for Exponential Integrators. Technical Report 4, Norwegian University of Science and Technology, 2005. Cited at pages 57, 58.

[24] H. Berland, B. Skaflestad, and W. M. Wright. EXPINT—a MATLAB Package for Exponential Integrators. *ACM Trans. Math. Softw.*, 33(1), 2007. Cited at pages 44, 56, 60.

[25] J.-P. Berrut and L. N. Trefethen. Barycentric Lagrange Interpolation. *SIAM Rev.*, 46(3):501–517, 2004. Cited at pages 38, 39.

[26] E. Bertolazzi, A. Falini, and F. Mazzia. The Object Oriented C++ library QIBSH++ for Hermite spline Quasi Interpolation. *arXiv preprint arXiv:2208.03260*, 2022. Cited at page 32.

[27] J. Bigot, V. Grandgirard, G. Latu, C. Passeron, F. Rozar, and O. Thomine. Scaling gysela code beyond 32K-cores on bluegene/Q. In *ESAIM: Proc.*, volume 43, pages 117–135. EDP Sciences, 2013. Cited at page 71.

[28] M. Bongini and G. Buttazzo. Optimal control problems in transport dynamics. *Math. Models Methods Appl. Sci.*, 27(3):427–451, 2017. Cited at page 100.

[29] A. Borzì and V. Schulz. *Computational Optimization of Systems Governed by Partial Differential Equations*. SIAM, 2011. Cited at page 92.

[30] J. P. Boyd. *Chebyshev and Fourier spectral methods*. DOVER Publications, Inc., New York, second edition, 2000. Cited at page 30.

[31] H. Burau, R. Widera, W. Hönig, G. Juckeland, A. Debus, T. Kluge, U. Schramm, T. E. Cowan, R. Sauerbrey, and M. Bussmann. PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster. *IEEE Trans. Plasma Sci.*, 38(10):2831–2839, 2010. Cited at page 72.

[32] M. Burger, M. Di Francesco, P. A. Markowich, and M.-T. Wolfram. On a mean field game optimal control approach modeling fast exit scenarios in human crowds. In *52nd IEEE Conference on Decision and Control*, 2013. Cited at pages 92, 93, 98.

[33] M. Burger, R. Pinnau, C. Totzeck, and O. Tse. Mean-Field Optimal Control and Optimality Conditions in the Space of Probability Measures. *SIAM J. Control Optim.*, 59(2):977–1006, 2021. Cited at page 93.

[34] M. Burger, R. Pinnau, C. Totzeck, O. Tse, and A. Roth. Instantaneous control of interacting particle systems in the mean-field limit. *J. Comput. Phys.*, 405, 2020. Cited at page 91.

[35] 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. IEEE, 2013. Cited at page 21.

[36] P. E. Caines, M. Huang, and R. P. Malhamé. Large population stochastic dynamic games: closed-loop McKean-Vlasov systems and the Nash certainty equivalence principle. *Commun. Inf. Syst.*, 6(3):221–252, 2006. Cited at page 91.

[37] M. Caliari and F. Cassini. A μ-mode based direction splitting of φ-functions for exponential integrators. *In preparation*, 2023. Cited at page 55.

[38] M. Caliari, F. Cassini, L. Einkemmer, and A. Ostermann. Efficient exponential integration of inhomogeneous evolutionary advection–diffusion–reaction equations. *In preparation*, 2023. Cited at page 103.

[39] M. Caliari, F. Cassini, L. Einkemmer, A. Ostermann, and F. Zivcovich. A μ-mode integrator for solving evolution equations in Kronecker form. *J. Comput. Phys.*, 455:110989, 2022. Cited at pages 7, 31, 32, 40, 44, 55, 56, 57, 102, 103, 112.

[40] M. Caliari, F. Cassini, and F. Zivcovich. Approximation of the matrix exponential for matrices with a skinny field of values. *BIT Numer. Math.*, 60(4):1113–1131, 2020. Cited at pages 10, 32, 44, 48, 56.

[41] M. Caliari, F. Cassini, and F. Zivcovich. A μ-mode approach for exponential integrators: actions of φ-functions of Kronecker sums. *arXiv preprint arXiv:2210.07667*, 2022. Cited at pages 43, 47, 55, 56, 57, 61, 65, 102.

[42] M. Caliari, F. Cassini, and F. Zivcovich. A μ-mode BLAS approach for multidimensional tensor-structured problems. *Numer. Algorithms*, 2022. Published online: 04 October 2022. Cited at pages 25, 44, 55, 56, 57, 61, 102, 103, 112.

[43] M. Caliari, F. Cassini, and F. Zivcovich. BAMPHI: Matrix and transpose free action of the combinations of φ-functions from exponential integrators. *J. Comput. Appl. Math.*, 423:114973, 2023. Cited at pages 44, 49, 56, 97.

[44] 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. Cited at pages 10, 44, 56.

[45] 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. Cited at pages 10, 44.

[46] 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. Cited at pages 14, 32, 47, 97.

[47] 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. Cited at page 20.

[48] E. Camporeale, G. L. Delzanno, B. K. B. Bergen, and J. D. Moulton. On the velocity space discretization for the Vlasov–Poisson system: Comparison between implicit Hermite spectral and Particle-in-Cell methods. *Comput. Phys. Commun.*, 198:47–58, 2016. Cited at page 71.

[49] P. Cannarsa, R. Capuani, and P. Cardaliaguet. Mean field games with state constraints: from mild to pointwise solutions of the PDE system. *Calc. Var. Partial Diff. Equ.*, 60:108, 2021. Cited at page 92.

[50] M. Caponigro, M. Fornasier, B. Piccoli, and E. Trélat. Sparse stabilization and control of alignment models. *Math. Models Methods Appl. Sci.*, 25(3):521–564, 2015. Cited at page 91.

[51] J. A. Carrillo, M. Di Francesco, A. Figalli, T. Laurent, and D. Slepčev. Confinement in nonlocal interaction equations. *Nonlinear Anal. Theory Methods Appl.*, 75(2):550–558, 2012. Cited at page 100.

[52] F. Cassini and L. Einkemmer. Efficient 6D Vlasov simulation using the dynamical low-rank framework `Ensign`. *Comput. Phys. Commun.*, 280:108489, 2022. Cited at page 71.

[53] P. Castillo and Y. Saad. Preconditioning the Matrix Exponential Operator with Applications. *J. Sci. Comput.*, 13(3):275–302, 1998. Cited at page 104.

[54] C. Cecka. Pro Tip: cuBLAS Strided Batched Matrix Multiply. https://developer.nvidia.com/blog/cublas-strided-batched-matrix-multiply, 2017. Cited at page 21.

[55] G. Ceruti and C. Lubich. An unconventional robust integrator for dynamical low-rank approximation. *BIT Numer. Math.*, 62:23–44, 2021. Cited at pages 72, 80.

[56] M. Chen and D. Kressner. Recursive blocked algorithms for linear systems with Kronecker product structure. *Numer. Algorithms*, 84(3):1199–1216, 2020. Cited at page 35.

[57] Y.-P. Choi, D. Kalise, J. Peszek, and A. A. Peters. A Collisionless Singular Cucker–Smale Model with Decentralized Formation Control. *SIAM J. Appl. Dyn. Syst.*, 18(4):1954–1981, 2019. Cited at page 91.

[58] NVIDIA Corporation. cuBLAS documentation. https://docs.nvidia.com/cuda/cublas/index.html, 2021. Cited at pages 26, 79.

[59] S. M. Cox and P. C. Matthews. Exponential Time Differencing for Stiff Systems. *J. Comput. Phys.*, 176(2):430–455, 2002. Cited at page 45.

[60] E. Cristiani, B. Piccoli, and A. Tosin. *Multiscale Modeling of Pedestrian Dynamics*, volume 12 of *MS&A. Model. Simul. Appl.* Springer, 2014. Cited at page 91.

[61] 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. Cited at page 10.

[62] N. Crouseilles, L. Einkemmer, and M. Prugger. An exponential integrator for the drift-kinetic model. *Comput. Phys. Commun.*, 224:144–153, 2018. Cited at page 10.

[63] M. Crouzeix and M. Palencia. The Numerical Range is a $(1+\sqrt{2})$-Spectral Set. *SIAM J. Matrix Anal. Appl.*, 38(2):649–655, 2017. Cited at page 48.

[64] F. Cucker and S. Smale. Emergent Behavior in Flocks. *IEEE Trans. Automat. Control*, 52(5):852–862, 2007. Cited at page 91.

[65] P. J. Davis and P. Rabinowitz. *Methods of numerical integration.* Academic Press, Inc., second edition, 1984. Cited at page 47.

[66] C. de Boor. *A Practical Guide to Splines*, volume 27 of *Applied Mathematical Sciences*. Springer, New York, revised edition, 2001. Cited at page 31.

[67] P. Degond, J.-G. Liu, S. Motsch, and V. Panferov. Hydrodynamic models of self-organized dynamics: Derivation and existence theory. *Methods Appl. Anal.*, 20(2):89–114, 2013. Cited at page 91.

[68] Z. Ding, L. Einkemmer, and Q. Li. Dynamical Low-Rank Integrator for the Linear Boltzmann Equation: Error Analysis in the Diffusion Limit. *SIAM J. Numer. Anal.*, 59(4):2254–2285, 2021. Cited at page 72.

[69] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 1990. Cited at page 26.

[70] M. R. D'Orsogna, Y.-L. Chuang, A. L. Bertozzi, and L. S. Chayes. Self-Propelled Particles with Soft-Core Interactions: Patterns, Stability, and Collapse. *Phys. Rev. Lett.*, 96(10), 2006. Cited at page 91.

[71] T. A. Driscoll, N. Hale, and L. N. Trefethen, editors. *Chebfun Guide*. Pafnuty Publications, Oxford, 2014. Cited at page 37.

[72] J. R. G. Dyer, A. Johansson, D. Helbing, I. D. Couzin, and J. Krause. Leadership, consensus decision making and collective behaviour in humans. *Philos. Trans. R. Soc. Lond., B, Biol. Sci.*, 364(1518):781–789, 2009. Cited at page 91.

[73] L. Einkemmer. High performance computing aspects of a dimension independent semi-Lagrangian discontinuous Galerkin code. *Comput. Phys. Commun.*, 202:326–336, 2016. Cited at page 71.

[74] 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. Cited at page 21.

[75] 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. Cited at page 21.

[76] L. Einkemmer. A Low-Rank algorithm for Weakly Compressible Flow. *SIAM J. Sci. Comput.*, 41(5):A2795–A2814, 2019. Cited at page 72.

[77] L. Einkemmer. A performance comparison of semi-Lagrangian discontinuous Galerkin and spline based Vlasov solvers in four dimensions. *J. Comput. Phys.*, 376:937–951, 2019. Cited at page 72.

[78] L. Einkemmer. Semi-Lagrangian Vlasov simulation on GPUs. *Comput. Phys. Commun.*, 254:107351, 2020. Cited at page 21.

[79] L. Einkemmer, J. Hu, and Y. Wang. An asymptotic-preserving dynamical low-rank method for the multi-scale multi-dimensional linear transport equation. *J. Comput. Phys.*, 439:110353, 2021. Cited at page 72.

[80] L. Einkemmer, J. Hu, and L. Ying. An Efficient Dynamical Low-Rank Algorithm for the Boltzmann-BGK Equation Close to the Compressible Viscous Flow Regime. *SIAM J. Sci. Comput.*, 43(5):B1057–B1080, 2021. Cited at page 72.

[81] L. Einkemmer and I. Joseph. A mass, momentum, and energy conservative dynamical low-rank scheme for the Vlasov equation. *J. Comput. Phys.*, 443:110495, 2021. Cited at pages 72, 80.

[82] L. Einkemmer and C. Lubich. A Low-Rank Projector-Splitting Integrator for the Vlasov–Poisson Equation. *SIAM J. Sci. Comput.*, 40(5):B1330–B1360, 2018. Cited at pages 71, 72, 73, 74, 76, 77, 80.

[83] L. Einkemmer, M. Moccaldi, and A. Ostermann. Efficient boundary corrected Strang splitting. *Appl. Math. Comput.*, 332:76–89, 2018. Cited at page 10.

[84] L. Einkemmer and A. Ostermann. An almost symmetric Strang splitting scheme for nonlinear evolution equations. *Comput. Math. with Appl.*, 67(12):2144–2157, 2014. Cited at page 77.

[85] L. Einkemmer and A. Ostermann. An almost symmetric Strang splitting scheme for the construction of high order composition methods. *J. Comput. Appl. Math.*, 271:307–318, 2014. Cited at page 77.

[86] 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. Cited at pages 10, 59.

[87] L. Einkemmer, A. Ostermann, and C. Piazzola. A low-rank projector-splitting integrator for the Vlasov–Maxwell equations with divergence correction. *J. Comput. Phys.*, 403:109063, 2020. Cited at page 72.

[88] X. Feng and A. Prohl. Numerical analysis of the Allen-Cahn equation and approximation for mean curvature flows. *Numer. Math.*, 94(1):33–65, 2003. Cited at page 52.

[89] F. Filbet and E. Sonnendrücker. Comparison of Eulerian Vlasov solvers. *Comput. Phys. Commun.*, 150(3):247–266, 2003. Cited at page 10.

[90] M. Fornasier, B. Piccoli, and F. Rossi. Mean-field sparse optimal control. *Philos. Trans. R. Soc. Lond., A, Math. Phys. Eng. Sci.*, 372(2028), 2014. Cited at page 92.

[91] M. Fornasier and F. Solombrino. Mean-Field Optimal Control. *ESAIM Control Optim. Calc. Var.*, 20(4):1123–1152, 2014. Cited at pages 92, 93.

[92] G. Freudenthaler and T. Meurer. PDE-based multi-agent formation control using flatness and backstepping: Analysis, design and robot experiments. *Automatica*, 115, 2020. Cited at page 91.

[93] M. Frigo and S. G. Johnson. FFTW: an adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '98)*, pages 1381–1384. IEEE, 1998. Cited at page 14.

[94] R. Garrappa and M. Popolizio. Generalized exponential time differencing methods for fractional order problems. *Comput. Math. with Appl.*, 62(3):876–890, 2011. Cited at page 66.

[95] R. Garrappa and M. Popolizio. Computing the matrix Mittag–Leffler function with applications to fractional calculus. *J. Sci. Comput.*, 77(1):129–153, 2018. Cited at page 66.

[96] 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(1):705830107, 2017. Cited at page 8.

[97] 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. Cited at pages 10, 13, 32, 44, 49, 56, 97, 98, 112.

[98] W. Gautschi and R. S. Varga. Error Bounds for Gaussian Quadrature of Analytic Functions. *SIAM J. Numer. Anal.*, 20(6):1170–1186, 1983. Cited at page 48.

[99] J. Gómez-Serrano, C. Graham, and J.-Y. Le Boudec. The bounded confidence model of opinion dynamics. *Math. Models Methods Appl. Sci.*, 22(2), 2012. Cited at page 91.

[100] C. González, A. Ostermann, and M. Thalhammer. A second-order Magnus-type integrator for nonautonomous parabolic problems. *J. Comput. Appl. Math.*, 189(1–2):142–156, 2006. Cited at page 59.

[101] V. Grandgirard, Y. Sarazin, X. Garbet, G. Dif-Pradalier, P. Ghendrih, N. Crouseilles, G. Latu, E. Sonnendrücker, N. Besse, and P. Bertrand. GYSELA, a full-f global gyrokinetic Semi-Lagrangian code for ITG turbulence simulations. In *AIP Conf. Proc.*, volume 871, pages 100–111. American Institute of Physics, 2006. Cited at page 72.

[102] W. Guo and Y. Cheng. A Sparse Grid Discontinuous Galerkin Method for High-Dimensional Transport Equations and Its Application to Kinetic Simulations. *SIAM J. Sci. Comput.*, 38(6):A3381–A3409, 2016. Cited at page 71.

[103] W. Guo and J.-M. Qiu. A low rank tensor representation of linear transport and nonlinear Vlasov solutions and their associated flow maps. *J. Comput. Phys.*, 458:111089, 2022. Cited at page 72.

[104] W. W. Hager. Runge-Kutta methods in optimal control and the transformed adjoint system. *Numer. Math.*, 87:247–282, 2000. Cited at page 92.

[105] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential Algeraic Problems*, volume 14 of *Springer Series in Computational Mathematics*. Springer, second edition, 1996. Cited at page 1.

[106] B. Hashemi and L. N. Trefethen. Chebfun in Three Dimensions. *SIAM J. Sci. Comput.*, 39(5):C341–C363, 2017. Cited at page 13.

[107] M. Herty and L. Pareschi. Fokker-Planck asymptotics for traffic flow models. *Kinet. Relat. Models*, 3(1):165–179, 2010. Cited at page 91.

[108] M. Herty, L. Pareschi, and S. Steffensen. Implicit-Explicit Runge–Kutta Schemes for Numerical Discretization of Optimal Control Problems. *SIAM J. Numer. Anal.*, 51(4):1875–1899, 2013. Cited at page 92.

[109] 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 International Publishing, 2019. Cited at page 8.

[110] M. Hochbruck, J. Leibold, and A. Ostermann. On the convergence of Lawson methods for semilinear stiff problems. *Numer. Math.*, 145:553–580, 2020. Cited at page 65.

[111] M. Hochbruck and A. Ostermann. Explicit Exponential Runge–Kutta Methods for Semilinear Parabolic Problems. *SIAM J. Numer. Anal.*, 43(3):1069–1090, 2005. Cited at pages 1, 40, 47, 51, 52.

[112] M. Hochbruck and A. Ostermann. Exponential integrators. *Acta Numer.*, 19:209–286, 2010. Cited at pages 1, 7, 44, 45, 56, 57, 58, 78, 92, 94, 103.

[113] W. Hundsdorfer and J. G. Verwer. *Numerical Solution of Time-Dependent Advection-Diffusion-Reaction Equations*, volume 33 of *Springer Series in Computational Mathematics*. Springer, 2003. Cited at pages 1, 10, 103.

[114] Intel Corporation. Intel Math Kernel Library. `https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html`, 2021. Cited at pages 26, 79.

[115] S. Jin, P. Markowich, and C. Sparber. Mathematical and computational methods for semiclassical Schrödinger equations. *Acta Numer.*, 20:121–209, 2011. Cited at page 17.

[116] A. K. Kassam. Solving reaction–diffusion equations 10 times faster. Technical Report 16, Oxford University, 2003. Cited at page 103.

[117] G. Kirsten and V. Simoncini. A matrix-oriented POD-DEIM algorithm applied to nonlinear differential matrix equations. *arXiv preprint arXiv:2006.13289v1*, 2020. Cited at page 35.

[118] O. Koch and C. Lubich. Dynamical Low-Rank Approximation. *SIAM J. Matrix Anal. Appl.*, 29(2):434–454, 2007. Cited at page 72.

[119] T. G. Kolda. Multilinear Operators for Higher-order Decompositions. Technical Report SAND2006-2081, Sandia National Laboratories, 2006. Cited at pages 11, 26, 27.

[120] T. G. Kolda and B. W. Bader. Tensor Decompositions and Applications. *SIAM Rev.*, 51(3):455–500, 2009. Cited at pages 11, 26, 27, 28, 44.

[121] K. Kormann. A Semi-Lagrangian Vlasov Solver in Tensor Train Format. *SIAM J. Sci. Comput.*, 37(4):B613–B632, 2015. Cited at page 72.

[122] K. Kormann and E. Sonnendrücker. Sparse Grids for the Vlasov–Poisson Equation. In *Sparse Grids and Applications - Stuttgart 2014*, volume 109, pages 163–190. Springer International Publishing, 2016. Cited at page 71.

[123] J. Kusch, G. Ceruti, L. Einkemmer, and M. Frank. Dynamical low-rank approximation for Burgers' equation with uncertainty. *Int. J. Uncertain Quantif.*, 12(5), 2022. Cited at page 72.

[124] J. Kusch, L. Einkemmer, and G. Ceruti. On the Stability of Robust Dynamical Low-Rank Approximations for Hyperbolic Problems. *SIAM J. Sci. Comput*, 45(1):A1–A24, 2023. Cited at page 72.

[125] J. M. Lasry and P. L. Lions. Mean field games. *Japanese J. Math.*, 2:229–260, 2007. Cited at pages 91, 92.

[126] D. Li, S. Yang, and J. Lan. Efficient and accurate computation for the φ-functions arising from exponential integrators. *Calcolo*, 59:11, 2022. Cited at pages 56, 60.

[127] D. Li, X. Zhang, and R. Liu. Exponential integrators for large-scale stiff Riccati differential equations. *J. Comput. Appl. Math.*, 389:113360, 2021. Cited at pages 55, 57, 62.

[128] D. Li, Y. Zhang, and X. Zhang. Computing the Lyapunov operator φ-functions, with an application to matrix-valued exponential integrators. *Appl. Numer. Math.*, 182:330–343, 2022. Cited at pages 55, 57, 62.

[129] J. Li, C. Battaglino, I. Perros, J. Sun, and R. Vuduc. 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*, New York, 2015. Association for Computing Machinery. Cited at page 29.

[130] V. T. Luan. Efficient exponential Runge–Kutta methods of high order: construction and implementation. *BIT Numer. Math.*, 61(2):535–560, 2021. Cited at page 47.

[131] V. T. Luan and A. Ostermann. Explicit exponential Runge–Kutta methods of high order for parabolic problems. *J. Comput. Appl. Math.*, 256:168–179, 2014. Cited at page 45.

[132] V. T. Luan, J. A. Pudykiewicz, and D. R. Reynolds. Further development of efficient and accurate time integration schemes for meteorological models. *J. Comput. Phys.*, 376:817–837, 2019. Cited at pages 44, 49, 56, 97.

[133] C. Lubich. *From Quantum to Classical Molecular Dynamics: Reduced Models and Numerical Analysis.* European Mathematical Society, 2008. Cited at page 72.

[134] C. Lubich and I. V. Oseledets. A projector-splitting integrator for dynamical low-rank approximation. *BIT Numer. Math.*, 54:171–188, 2014. Cited at pages 72, 73.

[135] 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. Cited at page 21.

[136] S. Markidis, S. W. D. 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. Cited at page 21.

[137] D. A. Matthews. High-Performance Tensor Contraction without Transposition. *SIAM J. Sci. Comput.*, 40(1):C1–C24, 2018. Cited at page 29.

[138] R. I. McLachlan and G. R. W. Quispel. Splitting methods. *Acta Numer.*, 11:341–434, 2002. Cited at page 7.

[139] M. Mehrenberger, C. Steiner, L. Marradi, N. Crouseilles, E. Sonnendrücker, and B. Afeyan. Vlasov on GPU (VOG project). *ESAIM: Proc.*, 43:37–58, 2013. Cited at page 21.

[140] H. Mena, A. Ostermann, L. M. Pfurtscheller, and C. Piazzola. Numerical low-rank approximation of matrix differential equations. *J. Comput. Appl. Math.*, 340:602–614, 2018. Cited at page 62.

[141] H.-D. Meyer, F. Gatti, and G. A. Worth. *Multidimensional Quantum Dynamics: MCTDH Theory and Applications*. Wiley-VCH, 2009. Cited at page 72.

[142] H.-D. Meyer, U. Manthe, and L. S. Cederbaum. The multi-configurational time-dependent Hartree approach. *Chem. Phys. Lett.*, 165(1):73–78, 1990. Cited at page 72.

[143] S. Motsch and E. Tadmor. Heterophilious Dynamics Enhances Consensus. *SIAM Rev.*, 56(4):577–621, 2014. Cited at page 91.

[144] J. Muñoz-Matute, D. Pardo, and V. M. Calo. Exploiting the Kronecker product structure of φ–functions in exponential integrators. *Int. J. Numer. Methods Eng.*, 123(9):2142–2161, 2022. Cited at pages 49, 51, 52, 55, 57.

[145] T. Namiki. A new FDTD algorithm based on alternating-direction implicit method. *IEEE Trans. Microw. Theory Tech.*, 47(10):2003–2007, 1999. Cited at page 8.

[146] R. Nath, S. Tomov, and J. Dongarra. Accelerating GPU Kernels for Dense Linear Algebra. In *High Performance Computing for Computational Science – VECPAR 2010*, volume 6449, pages 83–92. Springer Berlin Heidelberg, 2011. Cited at page 79.

[147] H. Neudecker. A Note on Kronecker Matrix Products and Matrix Equation Systems. *SIAM J. Appl. Math.*, 17(3):603–606, 1969. Cited at pages 9, 33.

[148] 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. Cited at page 10.

[149] J. Niesen and W. M. Wright. Algorithm 919: A Krylov Subspace Algorithm for Evaluating the ϕ-Functions Appearing in Exponential Integrators. *ACM Trans. Math. Softw.*, 38(3):1–19, 2012. Cited at pages 10, 13, 32, 44, 56.

[150] K.-K. Oh, M.-C. Park, and H.-S. Ahn. A survey of multi-agent formation control. *Automatica*, 53:424–440, 2015. Cited at page 91.

[151] D. Palitta and V. Simoncini. Matrix-equation-based strategies for convection–diffusion equations. *BIT Numer. Math.*, 56(2):751–776, 2016. Cited at page 35.

[152] G. Papanicolau and X. Xin. Reaction–diffusion fronts in periodically layered media. *J. Stat. Phys.*, 63:915–931, 1991. Cited at page 109.

[153] D. W. Peaceman and H. H. Rachford. The Numerical Solution of Parabolic and Elliptic Differential Equations. *J. Soc. Indust. Appl. Math.*, 3(1):28–41, 1955. Cited at page 8.

[154] Z. Peng and R. G. McClarren. A high-order/low-order (HOLO) algorithm for preserving conservation in time-dependent low-rank transport calculations. *J. Comput. Phys.*, 447:110672, 2021. Cited at page 72.

[155] Z. Peng, R. G. McClarren, and M. Frank. A low-rank method for two-dimensional time-dependent radiation transport calculations. *J. Comput. Phys.*, 421:109735, 2020. Cited at page 72.

[156] T. Penzl. *LYAPACK: A MATLAB Toolbox for Large Lyapunov and Riccati Equations, Model Reduction Problems, and Linear–Quadratic Optimal Control Problems. Users' Guide (Version 1.0)*, 2000. Cited at page 62.

[157] 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. Cited at page 19.

[158] A. A. Peters, R. H. Middleton, and O. Mason. Leader tracking in homogeneous vehicle platoons with broadcast delays. *Automatica*, 50(1):64–74, 2014. Cited at page 91.

[159] D. Phan and A. Ostermann. Exponential Integrators for Second-Order in Time Partial Differential Equations. *J. Sci. Comput.*, 93:58, 2022. Cited at page 53.

[160] B. Piccoli and A. Tosin. Pedestrian flows in bounded domains with obstacles. *Contin. Mech. Thermodyn.*, 21(2):85–107, 2009. Cited at page 91.

[161] The SeLaLib project team. SeLaLib library. `https://selalib.github.io/`, 2021. Cited at page 72.

[162] A. Quarteroni and A. Valli. *Numerical Approximation of Partial Differential Equations*, volume 23 of *Computational Mathematics*. Springer, 2008. Cited at page 1.

[163] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, M. Ravishankar, V. Grover, A. Rountev, L.-N. 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. Cited at page 21.

[164] D. M. Rogers. Efficient Primitives for Standard Tensor Linear Algebra. In *XSEDE16: Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale*, New York, 2016. Association for Computing Machinery. Cited at page 29.

[165] F. Rousset and K. Schratz. A General Framework of Low Regularity Integrators. *SIAM J. Numer. Anal.*, 59(3):1735–1768, 2021. Cited at page 59.

[166] V. Rybář. Spectral Methods for Reaction–Diffusion Systems. In *WDS '13 Proceedings of Contributed Papers, Part I*, pages 97–101, 2013. Cited at page 103.

[167] Y. Saad. Analysis of Some Krylov Subspace Approximations to the Matrix Exponential Operator. *SIAM J. Numer. Anal.*, 29(1):209–228, 1992. Cited at page 97.

[168] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Other Titles in Applied Mathematics. SIAM, Second edition, 2003. Cited at page 11.

[169] 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. Cited at page 21.

[170] F. Santambrogio. *Optimal Transport for Applied Mathematicians*, volume 87 of *Progress in Nonlinear Differential Equations and Their Applications*. Birkäuser, 2015. Cited at page 100.

[171] J. Sastre, J. Ibáñez, and E. Defez. Boosting the computation of the matrix exponential. *Appl. Math. Comput.*, 340:206–220, 2019. Cited at page 97.

[172] J. Sastre, Ibáñez. J., and E. Defez. Boosting the computation of the matrix exponential. *Appl. Math. Comput.*, 340:206–220, 2019. Cited at page 14.

[173] B. Skaflestad and W. M. Wright. The scaling and modified squaring method for matrix functions related to the exponential. *Appl. Numer. Math.*, 59(3–4):783–799, 2009. Cited at pages 44, 45, 56, 60, 97.

[174] P. Springer and P. Bientinesi. Design of a High-Performance GEMM-like Tensor–Tensor Multiplication. *ACM Trans. Math. Softw.*, 44(3):1–29, 2018. Cited at page 29.

[175] G. Szegö. *Orthogonal Polynomials*, volume 23 of *Colloquium Publications*. American Mathematical Society, Providence, fourth edition, 1975. Cited at pages 36, 37.

[176] K. Sznajd-Weron and J. Sznajd. Opinion evolution in closed community. *Int. J. of Mod. Phys. C*, 11(6):1157–1165, 2000. Cited at page 98.

[177] T. Tang. The Hermite Spectral Method for Gaussian-Type Functions. *SIAM J. Sci. Comput.*, 14(3):594–606, 1993. Cited at page 36.

[178] G. I. Taylor. Dispersion of soluble matter in solvent flowing slowly through a tube. *Proc. R. Soc. Lond. A.*, 219(1137):186–203, 1953. Cited at page 16.

[179] M. Thalhammer, M. Caliari, and C. Neuhauser. High-order time-splitting Hermite and Fourier spectral methods. *J. Comput. Phys.*, 228(3):822–832, 2009. Cited at page 12.

[180] G. Toscani. Kinetic models of opinion formation. *Commun. Math. Sci.*, 4(3):481–496, 2006. Cited at page 91.

[181] L. N. Trefethen. Multivariate polynomial approximation in the hypercube. *Proc. Am. Math. Soc.*, 145(11):4837–4844, 2017. Cited at page 39.

[182] D. Tskhakaya and R. Schneider. Optimization of PIC codes by improved memory management. *J. Comput. Phys.*, 225(1):829–839, 2007. Cited at page 72.

[183] C. F. Van Loan. The ubiquitous Kronecker product. *J. Comput. Appl. Math.*, 123(1–2):85–100, 2000. Cited at pages 42, 59.

[184] J. P. Verboncoeur. Particle simulation of plasmas: review and advances. *Plasma Phys. Control. Fusion*, 47(5A):A231–A260, 2005. Cited at page 71.

[185] N. Vervilet, O. Debals, L. Sorber, M. Van Barel, and L. De Lathauwer. Tensorlab 3.0. `https://tensorlab.net`, March, 2016. Available online. Cited at page 29.

[186] S. von Alfthan, D. Pokhotelov, Y. Kempf, S. Hoilijoki, I. Honkonen, A. Sandroos, and M. Palmroth. Vlasiator: First global hybrid-Vlasov simulations of Earth's foreshock and magnetosheath. *J. Atmos. Sol. Terr. Phys.*, 120:24–35, 2014. Cited at page 72.

[187] H. Wang, Q. Zhang, S. Wang, and C.-W. Shu. Local discontinuous Galerkin methods with explicit-implicit-null time discretizations for solving nonlinear diffusion problems. *Sci. China Math.*, 63:183–204, 2020. Cited at pages 104, 106, 115.

[188] M. Wiesenberger, L. Einkemmer, M. Held, A. Gutierrez-Milla, X. Sáez, and R. Iakymchuk. Reproducibility, accuracy and performance of the Feltor code and library on parallel computer architectures. *Comput. Phys. Commun.*, 238:145–156, 2019. Cited at page 21.

[189] Z. Xianyi, W. Qian, and Z. Yunquan. Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor. In *2012 IEEE 18th international conference on parallel and distributed systems*, pages 684–691, 2012. Cited at pages 26, 79.

[190] M. Zhao, H. Wang, and A. Cheng. A Fast Finite Difference Method for Three-Dimensional Time-Dependent Space-Fractional Diffusion Equations with Fractional Derivative Boundary Conditions. *J. Sci. Comput.*, 74:1009–1033, 2018. Cited at page 53.

[191] C. Zoppou and J. H. Knight. Analytical solution of a spatially variable coefficient advection–diffusion equation in up to three dimensions. *Appl. Math. Model.*, 23(9):667–685, 1999. Cited at page 39.