HAL
open science

CFL-less Discontinuous Galerkin solver

Pierre Gerhard, Philippe Helluy, Victor Michel-Dansac

To cite this version:

Pierre Gerhard, Philippe Helluy, Victor Michel-Dansac. CFL-less Discontinuous Galerkin solver. 2021. hal-03218086v1

HAL Id: hal-03218086

https://hal.science/hal-03218086v1

Preprint submitted on 5 May 2021 (v1), last revised 15 Mar 2022 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

CFL-LESS DISCONTINUOUS GALERKIN SOLVER

PIERRE GERHARD [*,†], PHILIPPE HELLUY [*,†], AND VICTOR MICHEL-DANSAC [†,*]

Abstract. We describe an explicit Discontinuous Galerkin (DG) kinetic scheme for solving systems of conservation laws. The solver is stable for an arbitrary CFL number and has the complexity of an explicit scheme. It can be applied to any hyperbolic system of conservation laws [2, 16]. In this work we assess the performance of the scheme in the particular case of the three-dimensional wave equation and of Maxwell's equations. We measure the benefit of the CFL-less feature and the parallel possibilities of the method.

1. Introduction

The Discontinuous Galerkin (DG) method is an increasingly popular method for solving hyperbolic systems of conservation laws. It is especially useful for numerical simulations in complex geometries, because it allows completely unstructured grids. It is a natural generalization of the finite volume method with high-order approximation. It is well adapted to linear wave models, such as Maxwell's equations or seismic models. It also works for non-linear hyperbolic equations, but requires limiting tools to handle shock waves in a robust and accurate way. The method is also well adapted to parallel software optimizations and hybrid computing. Of course, the method also has some drawbacks.

In this paper, we tackle the delicate point of time integration. Usually, the DG space approximation is coupled with an explicit Runge-Kutta time integration. This kind of time integration imposes quite constraining CFL conditions. Indeed, the time step size is then constrained by the smallest cell size in the mesh, independently of the time variation of the numerical solution. The smallest cell size may be imposed by geometric constraints and not by the required numerical precision. In addition, the possibility to coarsen the mesh to compute low-frequency solutions can be limited by an economic aspect: meshing a complex geometry is time-consuming and it is generally unfeasible, in real-world applications, to generate several meshes with various refinements. In principle, it would be possible to adopt an implicit time integration scheme, but it requires the inversion of a linear system at each time step and is thus very computationally expensive.

In this work, we propose a DG solver that is free from both restrictive CFL stability conditions and expensive linear system inversions. The method is based on an abstract kinetic representation of the system of conservation laws, which is a general method described for example in [19, 16]. The objective of this paper is to apply this general principle to the linear wave equation and to Maxwell's equations.

We use here a minimal set of kinetic velocities to reduce the memory footprint of the method. We recall how to construct an explicit, second-order in time and CFL-less time integration. We explain how to apply general boundary conditions that preserve the second-order time integration. We describe several software optimizations that have been done to parallelize the method. We then apply the method to several academic and physically relevant test cases, to provide a comparison, in terms of accuracy and efficiency, with the usual explicit DG method.

2. Minimal vectorial kinetic representation of conservation laws

In this section, we recall the theoretical framework of the vectorial kinetic representation that our numerical scheme is based on. After introducing notations for hyperbolic systems in section 2.1, we recall the vectorial kinetic representation itself in section 2.2. Section 2.3 explains the over-relaxation needed to use this representation in practice, and section 2.4 analyzes the effect of this over-relaxation on the initial hyperbolic system.

*: IRMA, Université de Strasbourg, CNRS UMR 7501, 7 rue René Descartes, 67084 Strasbourg, France
†: Université de Strasbourg, CNRS, Inria, IRMA, F-67000 Strasbourg, France
Key words and phrases. Discontinuous Galerkin, kinetic approximation, CFL-less.

2.1. **Hyperbolic systems of conservation laws.** In this work, we are interested in the numerical resolution of a hyperbolic system. The vector of unknowns is denoted $\mathbf{u}(\mathbf{x},t) \in \mathbb{R}^m$. It depends on time t and the space variable $\mathbf{x} = (x^1, \dots, x^d) \in \mathbb{R}^d$, where d is the space dimension (in practice, $d = 1, 2$ or 3). We denote by ∂_i the derivative with respect to x^i. The general form of the system is

$$\partial_t \mathbf{u} + \sum_{i=1}^{d} \partial_i \mathbf{q}^i(\mathbf{u}) = 0. \tag{2.1}$$

The flux functions $\mathbf{q}^i : \mathbb{R}^m \to \mathbb{R}^m$ are supposed to be smooth. For $\mathbf{n} = (n_1, \dots, n_d) \in \mathbb{R}^d$, we define the physical flux in the direction $\mathbf{n}$ by

$$\mathbf{q}(\mathbf{u}, \mathbf{n}) = \sum_{i=1}^{d} \mathbf{q}^i(\mathbf{u}) n_i.$$

We assume that the system satisfies the hyperbolicity property, i.e. that the jacobian matrix $\boldsymbol{\nabla}_{\mathbf{u}} \mathbf{q}(\mathbf{u}, \mathbf{n})$ is diagonalizable with real eigenvalues for all $\mathbf{n} \in \mathbb{R}^d$ and $\mathbf{u}$ in the hyperbolicity domain $\mathcal{C}$, which is supposed to be convex.

2.2. **Vectorial kinetic representation.** We now recall how to construct a minimal kinetic representation of (2.1).

For one-dimensional systems, with $d = 1$, this construction reduces to the Jin-Xin relaxation [26].

The general case for $d \geq 1$ is studied in [9, 2, 16]. We only recall here the simplest case, where the number of kinetic velocities is $d + 1$. We consider a set of constant vectors

$$\mathcal{V} = \left\{ \mathbf{v}_k = (v_k^1, \dots, v_k^d)^T \in \mathbb{R}^d, \ k \in \{0, \dots, d\} \right\},$$

called the kinetic velocities. We suppose that the rank of $\mathcal{V}$ is maximal (it must be equal to d). To each kinetic velocity $\mathbf{v}_k$ we associate a kinetic unknown $\mathbf{f}_k(\mathbf{x}, t) \in \mathbb{R}^m$. The kinetic unknowns and the macroscopic data $\mathbf{u}$ are related by

$$\mathbf{u} = \sum_{k=0}^{d} \mathbf{f}_k.$$

We then consider the following system of kinetic equations

$$\partial_t \mathbf{f}_k + \mathbf{v}_k \cdot \boldsymbol{\nabla} \mathbf{f}_k = \frac{1}{\varepsilon} (\mathbf{m}_k(\mathbf{u}) - \mathbf{f}_k), \tag{2.2}$$

where $\mathbf{m}_k(\mathbf{u})$ are the equilibrium kinetic functions and where ε is a small fixed parameter.

This system is similar to the BGK approximation in the kinetic theory of gases. In (2.2), the equilibrium kinetic functions $\mathbf{m}_k(\mathbf{u})$ play the role of the Maxwellian state of the kinetic theory of gases. The main differences are that, in this context, the number of kinetic velocities is finite and the kinetic unknown is a vector instead of a scalar function.

Let us now compute the equilibrium kinetic functions $\mathbf{m}_k(\mathbf{u})$. Summing the kinetic equations (2.2) on k, we obtain

$$\partial_t \mathbf{u} + \sum_{i=1}^{d} \partial_i \left(\sum_{k=0}^{d} v_k^i \cdot \mathbf{f}_k \right) = \frac{1}{\varepsilon} \left(\sum_{k=0}^{d} \mathbf{m}_k(\mathbf{u}) - \mathbf{u} \right). \tag{2.3}$$

Formally, when $\varepsilon \to 0$, in (2.2), we have $\mathbf{f}_k \simeq \mathbf{m}_k(\mathbf{u})$. If we assume that

$$\sum_{k=0}^{d} \mathbf{m}_k(\mathbf{u}) = \mathbf{u},$$

then, when $\varepsilon \to 0$, (2.3) becomes

$$\partial_t \mathbf{u} + \sum_{i=1}^{d} \partial_i \left(\sum_{k=0}^{d} v_k^i \cdot \mathbf{m}_k(\mathbf{u}) \right) = 0.$$

In conclusion, when $\varepsilon \to 0$, the kinetic system (2.2) is equivalent to the initial hyperbolic system (2.1) provided that

$$\sum_{k=0}^{d} \mathbf{m}_k(\mathbf{u}) = \mathbf{u} \quad \text{and} \quad \forall i \in \{1, \dots, d\}, \ \sum_{k=0}^{d} v_k^i \cdot \mathbf{m}_k(\mathbf{u}) = \mathbf{q}^i(\mathbf{u}). \tag{2.4}$$

Note that (2.4) is nothing but a set of $m(d+1)$ linear equations, whose $m(d+1)$ unknowns are the components of the $d+1$ Maxwellian vectors $\mathbf{m}_k(\mathbf{u})$. This linear system can also be written in matrix form

$$\begin{pmatrix}\mathbf{m}_0(\mathbf{u}) & \cdots & \mathbf{m}_d(\mathbf{u})\end{pmatrix}\mathbf{V} = \begin{pmatrix}\mathbf{u} & \mathbf{q}^1(\mathbf{u}) & \cdots & \mathbf{q}^d(\mathbf{u})\end{pmatrix},$$

with

$$\mathbf{V} = \begin{pmatrix} 1 & v_0^1 & & v_0^d \\ 1 & v_1^1 & \cdots & v_1^d \\ \vdots & & & \\ 1 & v_d^1 & \cdots & v_d^d \end{pmatrix} = \begin{pmatrix} 1 & \mathbf{v}_0^T \\ 1 & \mathbf{v}_1^T \\ \vdots & \vdots \\ 1 & \mathbf{v}_d^T \end{pmatrix}.$$

Because $\mathbf{V}$ is invertible, we obtain the Maxwellian state from the macroscopic data $\mathbf{u}$ and the physical flux $\mathbf{q}$ as follows:

$$\begin{pmatrix}\mathbf{m}_0(\mathbf{u}) & \cdots & \mathbf{m}_d(\mathbf{u})\end{pmatrix} = \begin{pmatrix}\mathbf{u} & \mathbf{q}^1(\mathbf{u}) & \cdots & \mathbf{q}^d(\mathbf{u})\end{pmatrix}\mathbf{V}^{-1}. \tag{2.5}$$

Let us remark that once the Maxwellian state is obtained, the physical flux is given by

$$\mathbf{q}(\mathbf{u},\mathbf{n}) = \sum_{k=0}^{d} (\mathbf{v}_k\cdot\mathbf{n})\,\mathbf{m}_k(\mathbf{u}).$$

Now that $\mathbf{m}_k(\mathbf{u})$ is known, we explain how to solve the equation (2.2) in practice.

2.3. **Splitting and over-relaxation.** In (2.2), the BGK source term

$$\frac{1}{\varepsilon}(\mathbf{m}_k(\mathbf{u}) - \mathbf{f}_k)$$

is of theoretical interest, but it is not very useful in practice because it couples in a non-linear way all the kinetic equations.

In practice, the kinetic system (2.2) is replaced by

$$\partial_t \mathbf{f}_k + \mathbf{v}_k\cdot\boldsymbol{\nabla}\mathbf{f}_k = \boldsymbol{\mu}_k,$$

where the source term $\boldsymbol{\mu}_k$ is designed in such a way to obtain $\mathbf{f}_k \simeq \mathbf{m}_k$ but is not a BGK source term. We introduce a time step $\Delta t > 0$ and the Dirac comb:

$$\varphi(t) = \sum_{j\in\mathbb{Z}} \delta(t - j\Delta t).$$

The source term $\boldsymbol{\mu}_k$ is then defined by

$$\boldsymbol{\mu}_k(x,t) = \boldsymbol{\Omega}(\mathbf{u})\varphi(t)\left(\mathbf{m}_k(\mathbf{u}(x,t^-)) - \mathbf{f}_k(x,t^-)\right),$$

where the so-called relaxation matrix $\boldsymbol{\Omega}$ is such that $\boldsymbol{\Omega} = \theta\mathbf{I}$, where $1\le\theta\le 2$ and with $\mathbf{I}$ the identity matrix. From the distribution theory, we see that at time $t = i\Delta t$, $\mathbf{f}_k$ is discontinuous: $\mathbf{f}_k(x,t^+) \neq \mathbf{f}_k(x,t^-)$, and

$$\mathbf{f}_k(x,t^+) = \boldsymbol{\Omega}\mathbf{m}_k(\mathbf{u}(x,t)) + (\mathbf{I}-\boldsymbol{\Omega})\mathbf{f}_k(x,t^-). \tag{2.6}$$

If $\theta = 1$ and the relaxation matrix is $\boldsymbol{\Omega} = \mathbf{I}$, we recover the classical first-order Jin-Xin splitting algorithm, where $\mathbf{m}_k(\mathbf{u}) = \mathbf{f}_k$ at the end of each time step. The *over-relaxation* corresponds to $\boldsymbol{\Omega} = 2\mathbf{I}$. It can be proven that the resulting scheme is a second-order in time approximation of (2.1), up to $\mathcal{O}(\Delta t^2)$. For more details, the reader is referred for instance to [16, 19] and included references.

We see that most of the time, the kinetic variables $\mathbf{f}_k$ satisfy free transport equations at velocity $\mathbf{v}_k$, with relaxation to equilibrium at each time step.

2.4. **Equivalent equation.** In this section, we analyze the effect of the kinetic over-relaxation on the initial variables $\mathbf{u}$ of the hyperbolic system (2.1). In order to get a more compact writing, we set

$$\mathbf{q}^0(\mathbf{u}) = \mathbf{u}, \quad \mathbf{Q}(\mathbf{u}) = \begin{pmatrix}\mathbf{q}^0(\mathbf{u}) & \mathbf{q}^1(\mathbf{u}) & \cdots & \mathbf{q}^d(\mathbf{u})\end{pmatrix},$$

and

$$\mathbf{M}(\mathbf{u}) = \begin{pmatrix}\mathbf{m}_0(\mathbf{u}) & \cdots & \mathbf{m}_d(\mathbf{u})\end{pmatrix}.$$

With these definitions, (2.5) becomes

$$\mathbf{Q}(\mathbf{u}) = \mathbf{M}(\mathbf{u})\mathbf{V}. \tag{2.7}$$

We also define the vector of kinetic variables

$$\mathbf{F} = \begin{pmatrix}\mathbf{f}_0 & \cdots & \mathbf{f}_d\end{pmatrix}. \tag{2.8}$$

Let us introduce the approximate fluxes $\mathbf{Z}$, defined by

$$\mathbf{Z} = \mathbf{F}\mathbf{V}, \quad \text{i.e. } \begin{pmatrix}\mathbf{z}^0 & \mathbf{z}^1 & \cdots & \mathbf{z}^d\end{pmatrix} = \begin{pmatrix}\mathbf{f}_0 & \cdots & \mathbf{f}_d\end{pmatrix}\mathbf{V}. \tag{2.9}$$

Let us emphasize that $\mathbf{z}^0 = \mathbf{u}$. We also introduce the flux errors $\mathbf{Y} = \begin{pmatrix}\mathbf{y}^0 & \cdots & \mathbf{y}^d\end{pmatrix}$, defined by

$$\mathbf{Y} = \mathbf{Z} - \mathbf{Q}(\mathbf{u}), \quad \text{i.e. } \forall i \in \{0,\dots,d\},\ \mathbf{y}^i = \mathbf{z}^i - \mathbf{q}^i(\mathbf{u}). \tag{2.10}$$

Plugging (2.7) and (2.9) into (2.10), we find that

$$\mathbf{Y} = (\mathbf{F} - \mathbf{M}(\mathbf{u}))\mathbf{V}.$$

Let us insist that, with our definition, while $\mathbf{Y}$ has $d+1$ columns, it lives in a $(m \times d)-$dimensional space since

$$\mathbf{y}^0 = 0.$$

With these notations, we can rewrite the transport-relaxation algorithm as follows. We start with a macroscopic field $\mathbf{u}(\mathbf{x},0)$ and flux error field $\mathbf{Y}(\mathbf{x},0)$ at time $t=0$. From this flux error, we compute the approximate fluxes

$$\mathbf{Z} = \mathbf{Y} + \mathbf{Q}(\mathbf{u})$$

and the kinetic variables

$$\mathbf{F} = \mathbf{Z}\mathbf{V}^{-1}.$$

Then, the kinetic variables $\mathbf{F}$ are transported with velocities $\mathcal{V}$ during a time step Δt, as follows:

$$\forall k \in \{0,\dots,d\},\ \mathbf{f}_k(\mathbf{x},\Delta t) = \mathbf{f}_k(\mathbf{x} - \Delta t \mathbf{v}_k, 0). \tag{2.11}$$

This defines the transport operator $\mathcal{T}(\Delta t)$ acting on the initial field $\mathbf{x} \mapsto \mathbf{F}(\mathbf{x},0)$. More precisely, $\mathcal{T}$ is unambiguously defined by (2.8), (2.11), and

$$(\mathcal{T}(\Delta t)\mathbf{F}(\cdot,0))(\mathbf{x}) = \mathbf{F}(\mathbf{x},\Delta t).$$

Finally, we return to the $(\mathbf{u},\mathbf{Y})$ variables by computing

$$\mathbf{u} = \mathbf{F}\cdot\begin{pmatrix}1\\ \vdots \\ 1\end{pmatrix} \quad \text{and} \quad \mathbf{Y} = \mathbf{F}\mathbf{V} - \mathbf{Q}(\mathbf{u}).$$

The transport step can thus be expressed in the following operator form

$$\mathbf{u}(\cdot,\Delta t) = \left\{\mathcal{T}(\Delta t)\left[\left(\mathbf{Y}(\cdot,0) + \mathbf{Q}(\mathbf{u}(\cdot,0)\right)\mathbf{V}^{-1}\right]\right\}\cdot\begin{pmatrix}1\\ \vdots \\ 1\end{pmatrix}$$

$$\mathbf{Y}(\cdot,\Delta t^-) = \left\{\mathcal{T}(\Delta t)\left[\left(\mathbf{Y}(\cdot,0) + \mathbf{Q}(\mathbf{u}(\cdot,0)\right)\mathbf{V}^{-1}\right]\right\}\mathbf{V} - \mathbf{Q}(\mathbf{u}(\cdot,\Delta t)).$$

In this $(\mathbf{u},\mathbf{Y})$ set of variables, the over-relaxation step simply reads

$$\mathbf{Y}(\cdot,\Delta t^+) = -\mathbf{Y}(\cdot,\Delta t^-).$$

The equivalent equation allows to better understand the effect of the relaxation matrix $\mathbf{\Omega}$. Through Taylor expansions in time, it is possible to obtain the partial differential equations satisfied by the pair $(\mathbf{u},\mathbf{Y})$. Details can be found in [19, 21]. Similar works using similar techniques can also be found in [20, 15].

For a simpler exposition, we only give the equivalent equation for a particular set of kinetic velocities and in a two-dimensional context, i.e. we take $d=2$. In this context, we have $m = d+1 = 3$ kinetic velocities. We choose the following kinetic velocities:

$$\mathbf{v}_0 = \lambda\begin{pmatrix}\cos(0)\\ \sin(0)\end{pmatrix}, \quad \mathbf{v}_1 = \lambda\begin{pmatrix}\cos(2\pi/3)\\ \sin(2\pi/3)\end{pmatrix}, \quad \mathbf{v}_2 = \lambda\begin{pmatrix}\cos(4\pi/3)\\ \sin(4\pi/3)\end{pmatrix},$$

where λ is a positive scaling factor. Then it can be proven that $\mathbf{u}$ and $\mathbf{Y}$ satisfy the following PDE up to second-order terms in Δt:

$$\begin{aligned}\partial_t\begin{pmatrix}\mathbf{u}\\ \mathbf{y}_1\\ \mathbf{y}_2\end{pmatrix} &+ \begin{pmatrix}\mathbf{A}^1(\mathbf{u}) & 0 & 0\\ 0 & \frac{\lambda}{2}\mathbf{I} - \mathbf{A}^1(\mathbf{u}) & 0\\ 0 & -\mathbf{A}^2(\mathbf{u}) & -\frac{\lambda}{2}\end{pmatrix}\partial_{x_1}\begin{pmatrix}\mathbf{u}\\ y_1\\ y_2\end{pmatrix}\\ &+ \begin{pmatrix}\mathbf{A}^2(\mathbf{u}) & 0 & 0\\ 0 & 0 & -\frac{\lambda}{2}\mathbf{I} - \mathbf{A}^1(\mathbf{u})\\ & -\frac{\lambda}{2} & -\mathbf{A}^2(\mathbf{u})\end{pmatrix}\partial_{x_2}\begin{pmatrix}\mathbf{u}\\ \mathbf{y}_1\\ \mathbf{y}_2\end{pmatrix} = O(\Delta t^2),\end{aligned} \tag{2.12}$$

where we have set

$$\mathbf{A}^i(\mathbf{u}) = \nabla_{\mathbf{u}}\mathbf{q}^i(\mathbf{u}).$$

We note that this analysis recovers the expected equation on $\mathbf{u}$ in the first line of (2.12). In addition, it also provides a system of PDEs satisfied by $\mathbf{Y}$ in the two next lines. In [21], it is shown that the equivalent equation (2.12) is a hyperbolic system under the condition that the scaling factor λ is large enough. In the literature, this kind of condition is often called a sub-characteristic condition. It was first introduced by Whitham in [30]. It is then encountered in numerous works using the relaxation approach. We refer for instance to [26, 14, 9, 5] and included references.

3. CFL-less second order time integration

To solve the transport part of the transport-relaxation algorithm described in section 2, we rely on the Discontinuous Galerkin (DG) method. It is applied to the kinetic variables $\mathbf{F}$, and we end up solving $d+1$ transport equations with constant velocity. Since this solver is the elementary brick of the whole algorithm, it is crucial to provide an efficient parallel implementation. Such an implementation was introduced in several previous works [4, 7, 16].

We describe in this section how to apply the DG solver to the following generic transport equation with a constant velocity $\mathbf{v}$, of unknown $f : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}$:

$$\partial_t f + \mathbf{v} \cdot \boldsymbol{\nabla} f = 0. \tag{3.1}$$

This equation represents the transport part of the equation (2.2); the treatment of the relaxation part is explained in section 2.3.

In order to avoid restrictive CFL conditions stemming from a few small tetrahedra in the unstructured mesh, we only consider implicit DG solvers. First, we recall in section 3.1 the implicit DG formulation, and we give more details on the computation of the time update on a single tetrahedral element. Let us emphasize that this implicit formulation ensures the unconditional L^2 stability of the resulting scheme. However, even though the scheme as a whole is implicit, it is possible to sweep the mesh in the direction of the velocity vector to enable a quasi-explicit resolution of (3.1). This procedure is explained in section 3.2, where we also discuss the possibility to parallelize this algorithm to increase its computational efficiency. We also give another way to solve the DG system in section 3.3, and the resolution of the relaxation step (2.6) is explained in section 3.4. Finally, the three algorithms are summarized in section 3.5 for the sake of convenience. For more details on our implementation of the Discontinuous Galerkin method, we refer to [4]. We also refer to [10] for technical algorithmic aspects of the task-graph algorithm.

3.1. DG formulation on a three-dimensional unstructured tetrahedral mesh.

For more details on the DG method, we refer (for instance) to [25]. The objective is to solve the transport equation with constant velocity (3.1). We assume that $d = 3$, i.e. we consider three space dimensions.

We consider a space domain $\mathcal{D} \subset \mathbb{R}^3$, meshed with a three-dimensional unstructured mesh $\mathcal{M}$ made of n_t tetrahedral cells with ten nodes (the four nodes of the tetrahedra and the six nodes at the edge midpoints), also known as "T10" elements in the finite element literature. In each cell L, we consider ten polynomial basis functions φ_i^L of degree $p = 2$. For efficiency reasons, the basis functions are Lagrange polynomials based on the ten nodes of the tetrahedra. For the sake of completeness, these basis functions are given on the reference tetrahedron in Appendix A.

In each cell in the mesh $\mathcal{M}$, the unknown function f is approximated with the basis functions, as follows:

$$\forall L \in \mathcal{M},\ \forall \mathbf{x} \in L,\ \forall t \geq 0,\ f(\mathbf{x}, t) \simeq f_L(\mathbf{x}, t) = \sum_{i=0}^{9} f_{L,i}(t)\varphi_i^L(\mathbf{x}). \tag{3.2}$$

Since the basis functions are polynomials of degree $p = 2$ in $\mathbf{x}$, this approximation is also a polynomial of degree $p = 2$, and the order of accuracy in space is expected to be $p+1 = 3$. In the formalism described in this section, the scheme is only first-order accurate in time. In practice, we actually use a second-order implicit Crank-Nicolson time stepping, which is very similar to the forthcoming description, but which we omit for the sake of clarity. Now, the goal is to provide a relevant approximation of the coefficients $f_{L,i}(t)$ of f_L in the polynomial basis. The procedure is fairly standard; however, we recall its main lines in Appendix B for the

sake of completeness. The DG solver for (3.1) then reads, with mesh notations from figure 3.1:

$$\begin{aligned}\sum_{i=0}^{9} \frac{f_{L,i}^n - f_{L,i}^{n-1}}{\Delta t} \left(\int_L \varphi_i^L(\mathbf{x}) \varphi_j^L(\mathbf{x})\, d\mathbf{x} \right) - \sum_{i=0}^{9} f_{L,i}^n \left(\int_L \varphi_i^L(\mathbf{x})\, \mathbf{v} \cdot \boldsymbol{\nabla} \varphi_j^L(\mathbf{x})\, d\mathbf{x} \right) \\ + \sum_{i=0}^{9} f_{L,i}^n \left[\sum_{\alpha=0}^{3} \left(\int_{\partial L_\alpha} \varphi_i^L(\boldsymbol{\eta}) \varphi_j^L(\boldsymbol{\eta})\, d\boldsymbol{\eta} \right) (\mathbf{v} \cdot \mathbf{n}_\alpha)_+ \right] \\ + \sum_{i=0}^{9} f_{R_\alpha,i}^n \left[\sum_{\alpha=0}^{3} \left(\int_{\partial L_\alpha} \varphi_i^{R_\alpha}(\boldsymbol{\eta}) \varphi_j^L(\boldsymbol{\eta})\, d\boldsymbol{\eta} \right) (\mathbf{v} \cdot \mathbf{n}_\alpha)_- \right].\end{aligned} \tag{3.3}$$

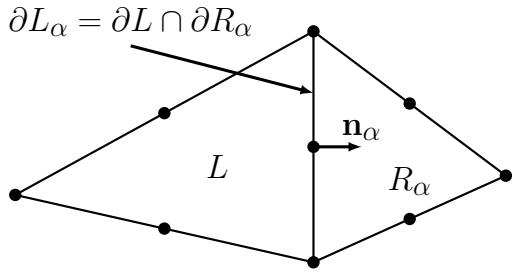


Figure 3.1. Notations for cells, interfaces and normal vectors. Given a cell L, its faces are numbered $\alpha \in \{0, \dots, 3\}$ and denoted by ∂L_α. The neighbor of L through the face ∂L_α is denoted by R_α, and the outer normal vector pointing from L to R_α is denoted by $\mathbf{n}_\alpha$. This figure is in two dimensions for the sake of simplicity, but the actual meshes we are interested in are unstructured three-dimensional tetrahedral meshes.

It turns out that (3.3) can be rewritten under a more compact form. We define two 10×10 matrices, the mass matrix M_L and the volume derivative matrix D_L, as follows:

$$(M_L)_{i,j} = \int_L \varphi_i^L(\mathbf{x}) \varphi_j^L(\mathbf{x})\, d\mathbf{x} \quad \text{and} \quad (D_L)_{i,j} = \int_L \varphi_i^L(\mathbf{x})\, \mathbf{v} \cdot \boldsymbol{\nabla} \varphi_j^L(\mathbf{x})\, d\mathbf{x}. \tag{3.4}$$

With these notations, the scheme (3.3) reads:

$$\begin{aligned}(M_L - \Delta t D_L) \begin{bmatrix} f_{L,0}^n \\ \vdots \\ f_{L,9}^n \end{bmatrix} = M_L \begin{bmatrix} f_{L,0}^{n-1} \\ \vdots \\ f_{L,9}^{n-1} \end{bmatrix} - \sum_{i=0}^{9} f_{L,i}^n \left[\sum_{\alpha=0}^{3} \left(\int_{\partial L_\alpha} \varphi_i^L(\boldsymbol{\eta}) \varphi_j^L(\boldsymbol{\eta})\, d\boldsymbol{\eta} \right) (\mathbf{v} \cdot \mathbf{n}_\alpha)_+ \right] \\ - \sum_{i=0}^{9} f_{R_\alpha,i}^n \left[\sum_{\alpha=0}^{3} \left(\int_{\partial L_\alpha} \varphi_i^{R_\alpha}(\boldsymbol{\eta}) \varphi_j^L(\boldsymbol{\eta})\, d\boldsymbol{\eta} \right) (\mathbf{v} \cdot \mathbf{n}_\alpha)_- \right].\end{aligned} \tag{3.5}$$

In (3.5), we are left with the determination of the two integrals in the right-hand side. To compute them, we need to exploit the mesh connectivity, and develop a procedure allowing us to systematically determine the neighbors of a given tetrahedron. This procedure is detailed in section 3.1.1. Then, once the neighboring tetrahedra and nodes are computed, an algorithm to evaluate the two remaining integrals is given in section 3.1.2.

3.1.1. *Geometry and connectivity.* Here, we detail how to use the mesh connectivity to find the neighbors of a given ten-node tetrahedron L. From now on, we identify the tetrahedron $L \in \mathcal{M}$ with its number $L \in \{0, \dots, n_t - 1\}$, with n_t the number of cells in the mesh $\mathcal{M}$. The DG nodes are numbered locally in the cell and also globally in the mesh. The local numbering is given on figure 3.2.

Because we use a discontinuous approximation, several global DG nodes are found at the same location (for instance at the corners of the tetrahedra or at the middle of the edges). Note that each tetrahedron contains 10 local nodes, and therefore the mesh with n_t tetrahedra contains $10n_t$ global nodes. Therefore, if a given node N has a local number i in a cell L, where $0 \leq i < 10$ and $0 \leq L < n_t$, then its global number i', with $0 \leq i' < 10n_t$, is obtained through the formula

$$i' = 10L + i.$$

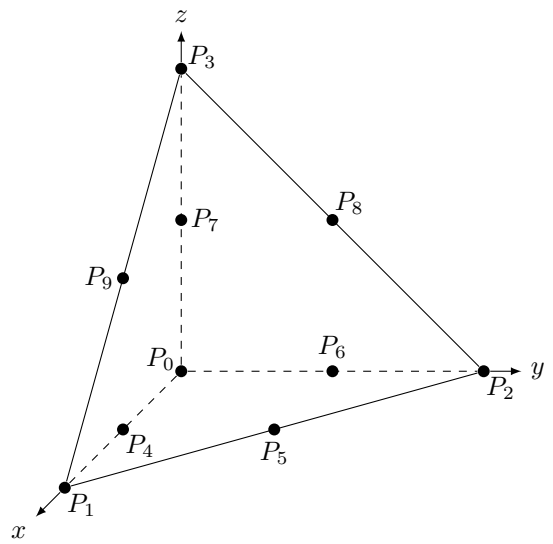


Figure 3.2. Local numbering of the nodes in a cell.

The reverse connectivity is then given by

$$L = \left\lfloor \frac{i'}{10} \right\rfloor \quad \text{and} \quad i = i' \bmod 10.$$

Contrary to the continuous Galerkin method, the connectivity between the local DG nodes and the global DG nodes is a bijection. Let us now denote by $N_{L,i}$ the local nodes of cell L and $N_{i'}$ the global nodes: we get

$$N_{i'} = N_{L,i} \iff i' = 10L + i.$$

We also need a numbering of the nodes inside the cell faces. Consider the local face α in cell L, with $0 \leq \alpha < 4$. We denote by $R_\alpha = \nu(L, \alpha)$ the neighbor cell to L along the face α. We also require a numbering of the nodes inside the local faces. Consider a local face α in cell L, and define k the local node number in the face α, with $0 \leq k < 6$. Then the local node number i in the cell L, with $0 \leq i < 10$, is given by the constant array

$$i = \mathsf{f2c}[\alpha, k], \quad \text{with} \quad \mathsf{f2c} = \begin{pmatrix} 1 & 2 & 3 & 5 & 8 & 9 \\ 0 & 3 & 2 & 7 & 8 & 6 \\ 0 & 1 & 3 & 4 & 9 & 7 \\ 0 & 2 & 1 & 6 & 5 & 4 \end{pmatrix}.$$

This array is obtained by considering the first-order nodes of each face, numbered 0 through 3, and ordering them in a counter-clockwise fashion. Then, we also perform a counter-clockwise ordering of the second-order nodes, numbered 4 through 9.

We leverage the connectivity one last time to recover the global node index in the neighbor along a given face. Let L and R_α be two neighboring cells, connected through local face α in L:

$$R_\alpha = \nu(L, \alpha).$$

We can also find the unique local face β in cell R_α such that

$$L = \nu(R_\alpha, \beta).$$

Now, let k be a local node number in the local face α of cell L. The local node number in cell L is given by

$$i = \mathsf{f2c}(\alpha, k).$$

It corresponds to a global node

$$i' = 10L + i.$$

In the same way, there is a node in cell R_α that is geometrically at the same position than node i'. Suppose that this node is the local node l of face β of cell R_α. The local index j of this node in cell R_α is given by

$$j = \mathsf{f2c}(\beta, l).$$

It corresponds to a global node

$$j' = 10R_\alpha + j.$$

We then set

$$j' = \mathsf{ext_node}(L, \alpha, k).$$

In other words, the $\mathsf{ext_node}$ connectivity array allows us to recover the global nodes that match on a given face. From the above definitions, we also have

$$i' = \mathsf{ext_node}(R_\alpha, \beta, l).$$

The last ingredient we need to compute the scheme (3.5) is the basis functions φ_i^L. Since they are Lagrange polynomials based on the ten nodes of the tetrahedra, they satisfy the interpolation property

$$\varphi_i^L(N_{L,j}) = \delta_{i,j} \ ,$$

where $\delta_{i,j}$ denotes the usual Kronecker delta. This implies that the components of f in the expansion (3.2) are simply the nodal values at the DG nodes

$$f_L(N_{L,i}, t) = f_{L,i}(t).$$

3.1.2. *Application to the DG scheme.* Having introduced all these notations, we can now rewrite the DG scheme (3.5) in cell L under the form of a local 10×10 linear system, where the unknown is the vector of values $f_L^n := (f_{L,i}^n)_{i\in\{0,\dots,9\}}$ at the DG nodes of cell L. This linear system reads

$$\left(M_L - \Delta t D_L + \Delta t F_L^+\right) \begin{bmatrix} f_{L,0}^n \\ \vdots \\ f_{L,9}^n \end{bmatrix} = M_L \begin{bmatrix} f_{L,0}^{n-1} \\ \vdots \\ f_{L,9}^{n-1} \end{bmatrix} - \Delta t \sum_{\alpha=0}^{3} F_{L,\alpha}^- \begin{bmatrix} f^n_{\mathsf{ext_node}(L,\alpha,0)} \\ \vdots \\ f^n_{\mathsf{ext_node}(L,\alpha,5)} \end{bmatrix}, \tag{3.6}$$

with M_L and D_L given by (3.4), and where F_L^+ and $F_{L,\alpha}^-$ are defined as follows:

- F_L^+ is a 10×10 matrix defined by the following assembly algorithm:
  $F_L^+ \leftarrow 0$
  for $\alpha \leftarrow 0$ to 3 **do**
  for $k \leftarrow 0$ to 5 **do**
  $i \leftarrow \mathsf{f2c}(\alpha, k)$
  for $l \leftarrow 0$ to 5 **do**
  $j \leftarrow \mathsf{f2c}(\alpha, l)$
  $\left(F_L^+\right)_{i,j} \leftarrow \left(F_L^+\right)_{i,j} + (\mathbf{v}\cdot\mathbf{n}_\alpha)_+ \displaystyle\int_{\partial L_\alpha} \varphi_i^L(\boldsymbol{\eta})\varphi_j^L(\boldsymbol{\eta})\, d\boldsymbol{\eta}$
- $F_{L,\alpha}^-$ are four 10×6 matrices defined by the following assembly algorithm:
  for $\alpha \leftarrow 0$ to 3 **do**
  $R_\alpha \leftarrow \nu(L, \alpha)$
  for $k \leftarrow 0$ to 5 **do**
  $i \leftarrow \mathsf{f2c}(\alpha, k)$
  for $l \leftarrow 0$ to 5 **do**
  $j \leftarrow \mathsf{f2c}(\alpha, l)$
  $\left(F_{L,\alpha}^-\right)_{i,l} \leftarrow (\mathbf{v}\cdot\mathbf{n}_\alpha)_- \displaystyle\int_{\partial L_\alpha} \varphi_i^{R_\alpha}(\boldsymbol{\eta})\varphi_j^L(\boldsymbol{\eta})\, d\boldsymbol{\eta}$

The main point is that if cell $R_\alpha = \nu(L, \alpha)$ is such that $\mathbf{v}\cdot\mathbf{n} > 0$ on ∂L_α, then the matrix $F_{L,\alpha}^-$ vanishes, and therefore the values of f_R^n in this cell R_α are not needed to compute the values of f_L^n in cell L. In practice, consider the computation of f_L^n in a cell L. For a classical implicit scheme, computing f_L^n requires simultaneously computing f_R^n for each neighbor R of cell L. However, in the scheme (3.5), computing f_L^n only requires the knowledge of f_R^n in cells R upwind from L, i.e. for which $\mathbf{v}\cdot\mathbf{n} > 0$. Reciprocally, to compute f_R^n for such cells R, knowing f_L^n is unnecessary. Therefore, the computation of f^n in a cell is completely decoupled from that of f^n in its neighbor cells. Moreover, since the transport velocity $\mathbf{v}$ is constant, such dependencies can be computed once and for all during the preprocessing phase. This procedure is explained in the next section.

3.2. **Parallel downwind algorithm.** One time step of the implicit DG scheme consists in computing $f^n := (f_L^n)_{L\in\mathcal{M}}$, the distribution function at time t^n, from the distribution function f^{n-1} of the previous time step. From (3.5) it is clear that one has to solve a linear system, as in any implicit scheme. However, because the kinetic velocity $\mathbf{v}$ is constant and because we use the upwind numerical flux, the linear system is triangular. It can thus be solved cell by cell, by simply sweeping the mesh in the direction of the velocity vector. The sweep algorithm relies on the construction of a Directed Acyclic Graph (a DAG) that we define in section 3.2.1. The resolution of the transport equation is explained in section 3.2.2. This mesh sweeping technique is well suited to parallel computing, and we highlight this application in section 3.2.3.

3.2.1. *Reformulation of the mesh as a graph.* Let L and R be two adjacent cells ($\partial L \cap \partial R \neq \emptyset$), with R the neighbor of L through its face α, i.e. $R = \nu(L, \alpha)$. We say that L is *upwind* with respect to R if $\mathbf{v} \cdot \mathbf{n}_\alpha > 0$ on $\partial L \cap \partial R = \partial L_\alpha$. For a cell L, the solution depends only on the values of f in the upwind cells. For a given constant velocity $\mathbf{v}$ we can build a DAG $\mathcal{G}$ to represent the dependencies between the cells. Each vertex of the graph corresponds to a cell of the mesh. Each edge of the graph is associated to a face between two adjacent cells. If the cell L is upwind with respect to R, then the edge associated to $\partial L \cap \partial R$ is oriented from L to R.

We also consider two additional fictitious cells: the "upwind" cell corresponds to the part of the boundary $\partial\mathcal{D}$ where the velocity $\mathbf{v}$ enters the computational domain and the "downwind" cell corresponds to the part of the boundary $\partial\mathcal{D}$ where the velocity $\mathbf{v}$ exits the computational domain.

The dependency graph for a simple two-dimensional mesh and a given constant velocity is represented on figure 3.3.

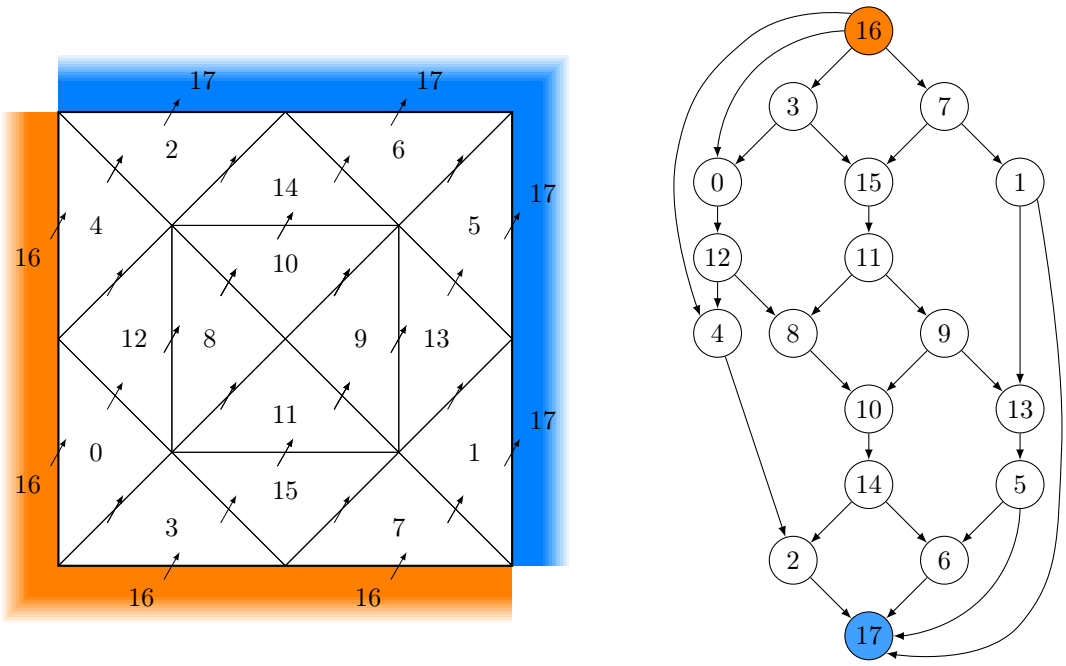


Figure 3.3. Left panel: simple 2D mesh, where the constant velocity $\mathbf{v}$ is represented with an arrow through each edge, and where the upwind and downwind boundary conditions are respectively represented with orange and blue colors. The true cells are numbered from 0 to 15, and the "upwind" and "downwind" cells are respectively numbered 16 and 17. Right panel: dependency graph of the mesh, according to the constant velocity $\mathbf{v}$. The vertices of the graph are the cells of the mesh, and the edges of the graph are the edges of the mesh, The graph edges are directed according to the direction of $\mathbf{v}$ through the corresponding mesh edge.

The construction can be generalized to any unstructured mesh with flat faces. The flatness condition and the fact that the velocity $\mathbf{v}$ is constant ensures that the velocity crosses each face in only one direction. Therefore, the graph does not contain any loop, and it is indeed a DAG. For more details, we refer to [4].

3.2.2. *Solving a transport equation using the graph.* We now state the algorithm to solve one transport equation for a given constant velocity $\mathbf{v}$.

- **Graph ordering**: Compute a topological ordering of the dependency graph $\mathcal{G}$. Let $\mathcal{N}$ be the set of vertices of $\mathcal{G}$. Let $n_{\mathcal{N}}$ be the number of vertices of $\mathcal{G}$, defined by $n_{\mathcal{N}} = \#\mathcal{N}$. Let us recall that $n_{\mathcal{N}} = n_t + 2$, where n_t is the number of cells in the mesh, because of the fictitious "upwind" and "downwind" cells. The ordering is a bijection
$$\sigma : \{1, \dots, n_{\mathcal{N}}\} \to \mathcal{N}$$
such that, if there is an oriented path from A to B, then $\sigma(B) > \sigma(A)$.
- **Linear system preparation**: Perform the assembly, LU decomposition and storage of the local linear system given in (3.6). These computations can also be redone at each time step to save memory at the expense of CPU time; this possibility is explored further in section 6.
- **Transport equation resolution**: At each time step, the following algorithm is applied to each cell in topological order.
  (1) For each upwind face α of L, extract the face values
$$\begin{bmatrix} f^n_{\mathsf{ext_node}(L,\alpha,0)} \\ \vdots \\ f^n_{\mathsf{ext_node}(L,\alpha,5)} \end{bmatrix}.$$
  (2) Solve the local linear system (3.6). Its unknowns correspond to $f^n_{L,j}$ in a given cell L, which are computed from the values $f^n_{R,j}$ in the upwind neighbor cells.

3.2.3. *Parallelization.* As it turns out, it is possible to parallelize large chunks of this transport solver, even though the solver as a whole is not parallelizable. Indeed, because of the dependency graph, we cannot perform all the computations in parallel. For instance, for the mesh of figure 3.3, it is necessary to compute the solution in cells 3 and 7 first. Then, cells 0, 15 and 1 can be computed, etc. Let us emphasize that, in this case, cells 3 and 7 can be computed independently in parallel, like cells 0, 15 and 1, and subsequent downwind cells. Therefore, ordering the graph allows us to subdivide the mesh in several parallel regions, which depend on the transport velocity. In this section, we write the possible parallel optimizations more rigorously.

We assume that the renumbering σ described in section 3.2.2 has been computed. Let us consider a set of cells of the form
$$\mathcal{P} = \{\sigma(k), \sigma(k+1), \dots, \sigma(k+l)\},$$
where k and l are non-negative integers. We shall say that $\mathcal{P}$ is a *parallel region* of size l if there is no edge of the DAG joining two cells inside $\mathcal{P}$. This means that the cells inside $\mathcal{P}$ can be computed completely independently, and in parallel. The algorithm to construct the parallel regions is given below.

$p \leftarrow 0, \quad k_p \leftarrow 0, \quad l_p \leftarrow 0$
while $k_p + l_p < n_{\mathcal{N}}$ **do**
  $\mathcal{P}_p = \emptyset$
  while cell $\sigma(k_p + l_p)$ does not depend on the cells in $\mathcal{P}_p$ **do**
    $\mathcal{P}_p \leftarrow \mathcal{P}_p \cup \{\sigma(k_p + l_p)\}$
    $l_p \leftarrow l_p + 1$
  end while
  $p \leftarrow p + 1, \quad k_p \leftarrow k_{p-1} + l_{p-1}, \quad l_p \leftarrow 0$
end while

Once the n_p parallel regions are constructed, the transport equation resolution from section 3.2.2 is modified as follows.

- **Transport equation resolution**: At each time step, the following algorithm is applied.
  for $p \leftarrow 0$ to $n_p - 1$ **do**
    for each cell L in the parallel region $\mathcal{P}_p$ **do in parallel**
      (1) For each upwind face α of L, extract the face values
$$\begin{bmatrix} f^n_{\mathsf{ext_node}(L,\alpha,0)} \\ \vdots \\ f^n_{\mathsf{ext_node}(L,\alpha,5)} \end{bmatrix}.$$
      (2) Solve the local linear system (3.6).

The parallel efficiency heavily relies on the quality of the topological sort. Indeed, this sorting is generally not unique. A building block of this algorithm is how the graph is visited. The visiting algorithm can be a Depth-First Search (DFS) or a Breadth-First Search (BFS), for instance. In many applications, the DFS is algorithm is preferred, because it is more memory efficient. However, for our application, we have observed a much better parallel efficiency of the topological sorting if a BFS is used instead. See section 6.2 for practical evaluations of the two options.

The parallelization is done by the `rayon`[1] library of the Rust language: once the serial version of the above algorithm is correctly written, the library ensures a correct parallelization. We have also tested other implementations in previous works. For more details on the implementation, we refer for instance to [4], where the algorithm is parallelized with the StarPU runtime, or to [10] where we use a specialized DAG (Direct Acyclic Graph) clustering algorithm.

3.3. **Full matrix assembly.** Another way to practically to apply the implicit DG scheme is to perform a full assembly of the associated linear system. First, arrange all the unknowns $f^n_{L,i}$ in a single vector Φ^n. The implicit DG scheme can be written in the following matrix form

$$\frac{\Phi^n - \Phi^{n-1}}{\Delta t} + \mathbb{K}\Phi^n = S^{n-1},$$

where S^{n-1} is a vector arising from the approximation of the boundary conditions. One time step of the DG algorithm then corresponds to the resolution of the linear system

$$\mathbb{M}\Phi^n = \Phi^{n-1} + \Delta t S^{n-1},$$

with unknown Φ^n.

From what we have seen above, we know that, up to a permutation, the matrix $\mathbb{M}$ is actually a block-triangular matrix and that the size of the diagonal blocks is 10×10. This kind of matrix arises frequently in numerical circuit analysis. Special software libraries have been developed for solving efficiently the associated linear systems. We can mention for instance KLU, from the SuiteSparse library [17]. Accelerated parallel versions of the KLU algorithm also exist [8, 12].

We have thus also implemented a second version of the transport DG solver, where the matrix $\mathbb{M}$ is fully assembled and stored in memory. This obviously induces a higher memory consumption. But the programming is simpler and the task of renumbering the unknowns for discovering the triangular structure of $\mathbb{M}$ is delegated to the KLU library. We compare, in section 3.2.3, the two approaches in terms of memory occupation and efficiency.

Finally, in order to compare our method to the classical DG approach, we have also implemented the standard explicit DG method to directly solve (2.1). The spatial basis functions are the same as above. The spatial approximation is then exactly the same as in [18]. We thus apply an exact quadrature to the integral terms of the DG formulation. In this third implementation, the time integration is performed by the low-storage third-order Runge-Kutta (RK3) scheme from [31]. This choice is motivated by the fact that the RK3 scheme is the lowest order RK scheme that leads to a CFL condition where the limit stability time step Δt is proportional to the smallest cell size h:

$$\Delta t \sim h.$$

For instance, the RK2 scheme would lead to a CFL condition of the form

$$\Delta t \sim h^{3/2}.$$

The resulting approximation is often referred to as the RKDG method. For a discussion on the RKDG method and its variants, we refer, among others, to [13, 25, 22].

3.4. **Relaxation step.** The parallelization of the relaxation step is straightforward. We simply apply for each node the relaxation formula (2.6) to the kinetic data. In the KLU implementation of the scheme, this step is parallelized thanks to a simple `OpenMP` loop. In the Rust implementation, we once again rely on `rayon`. Let us remark that while embarrassingly parallel, the relaxation step induces many cache misses because the organization of the kinetic data into memory cannot be optimal both in the transport and in the relaxation steps.

[1] https://docs.rs/rayon

3.5. **Summary of the algorithms.** In the numerical experiments, we call `D3Q4` the algorithm stemming from the full matrix assembly described in section 3.3, and we denote by `D3Q4P` the parallel version of the downwind algorithm described in section 3.2. The main features of these two algorithms, and of their implementations, are summarized in table 3.1.

Table 3.1. Main differences between the algorithms and the implementation of `D3Q4` and `D3Q4P`.

Method	`D3Q4`	`D3Q4P`
Parallelization	one thread per v_i	deduced from mesh ordering
Memory usage	global DG matrix stored	local DG matrix computed on the fly
Libraries	`SuiteSparse-KLU` (LU-solver) `OpenMP` (parallelization)	`petgraph` (graph ordering) `rayon` (parallelization)
Written in	`C89`	`Rust`

4. CFL condition

The presented method is claimed to be CFL-less, i.e. unconditionally L^2-stable. In order to measure this feature, we have to give a precise definition of the CFL number.

For one-dimensional problems, in the book of Hesthaven [25] on DG nodal methods, the form of the CFL condition is as follows (section 4.8, page 97):

$$\Delta t \leq \beta \frac{1}{\lambda_{\max}} h_{\min}, \tag{4.1}$$

where Δt is the step of the time integrator, $\lambda_{\max}$ is the maximal wave speed, $h_{\min}$ is the minimal distance between two interpolation points and β is the CFL number. In this paper, the maximal wave speed can be either the speed of light c or the kinetic scaling factor λ. From the sub-characteristic condition, we know that $\lambda > \sqrt{3}c$. In this paper, we will consider a CFL condition given by (4.1), with $h_{\min}$ defined by the smallest cell in the mesh, measured by

$$h_{\min} = \min_{L \in \mathcal{M}} \frac{\text{volume}(L)}{\text{surface}(\partial L)} \tag{4.2}$$

The maximum CFL number such that the scheme remains stable depends on the time integration method. For explicit methods such as RK3 or RK4, it is of the order of unity, i.e. $\beta_{\max} \sim 1$.

In higher dimensions, the definition of the CFL number is less straightforward. In several papers (such as [13, 29]), one can find the condition

$$\Delta t \leq \gamma \frac{1}{\lambda_{\max}} \frac{\Delta x}{2r+1},$$

where Δx is the cell size and r the polynomial degree (we only consider the case $r = 2$ in this work). In this context, the CFL number is defined by

$$\gamma := \frac{\lambda_{\max} \Delta t}{\Delta x}(2r+1).$$

This definition is well suited for DG methods based on Gauss-Legendre points. For DG methods based on Legendre-Gauss-Lobatto (LGL) nodes, it appears that the stability condition is less constraining [22], and that a more adapted definition is

$$\theta := \frac{\lambda_{\max} \Delta t}{\Delta x}(r+1).$$

In dimension higher than one, there are very few rigorous results on the stability condition of the DG LGL method on unstructured meshes. What is observed in practice (see [11, 29]) is a more constraining stability condition when the dimension increases. In this paper, we give numerical results for very large values of β. The scheme remains stable at these high CFL numbers. We insist on the fact that our scheme is CFL-free, and that the time step is only restricted by the required accuracy. When the wave velocities do not depend on the solution – which is the case for linear PDEs – the scheme is stable whatever the time step. In this work we have observed that, with our definition (4.1) of the CFL number, the classical explicit RK3DG scheme is stable up to $\beta \simeq 1.85$.

5. Numerical applications

This section is dedicated to the numerical applications of the method described above. In a first step, in section 5.1, we validate our numerical scheme by performing an error analysis on two tests cases, one based on Maxwell's equations and the other on the wave equation. This shows that our method is generic enough to accommodate multiple hyperbolic systems. In a second step, in section 5.2 we illustrate the "CFL-less" aspect of the method and its advantage over an explicit RK3DG method when the mesh presents some cells which are way smaller than the variation scales of the studied solution.

For the sake of simplicity, all the numerical experiments are preformed on the unit cube $\mathcal{D} = [0,1]^3$. The exact solution is denoted by $\mathbf{u}^{\text{inc}}(\mathbf{x}, t)$. To estimate the convergence rates of our scheme, we define, at the final time t_{end}, the discrete L^2 error using the mass matrix M_L from (3.4), as follows:

$$e_r(h_{\min}, \Delta t) = \frac{1}{m} \sum_{k=1}^{m} \sqrt{\sum_{L \in \mathcal{M}} \left[M_L \left(u_{k,L}^n - u_{k,L}^{\text{inc}}(t_{\text{end}}) \right) \right] \left(u_{k,L}^n - u_{k,L}^{\text{inc}}(t_{\text{end}}) \right)}, \tag{5.1}$$

where $h_{\min}$ is defined in (4.2). With this error definition, for a given time step Δt and for two spatial discretizations characterized by $h_{\min}$ and $h'_{min} = h_{\min}/2$, the spatial order of accuracy μ is then computed using the regression:

$$\mu = \frac{\log(e_r(h'_{min}, \Delta t)/e_r(h_{\min}, \Delta t))}{\log(2)}.$$

In a same way, the temporal order of accuracy κ is determined by considering two time steps Δt and $\Delta t' = \Delta t/2$ for a fixed $h_{\min}$:

$$\kappa = \frac{\log(e_r(h_{\min}, \Delta t')/e_r(h_{\min}, \Delta t))}{\log(2)}.$$

Remark. *All our numerical experiments are performed in three dimensions of space and based on unstructured meshes that are composed of first geometrical order tetrahedra (i.e. with straight edges). For the sake of readability and uniformity in this paper, the numerical fields are always depicted in two dimensions of space. The two-dimensional representations are obtained by slicing the physical domain $\mathcal{D}$ with the software* `ParaView` [3]. *Our experiments are performed on the unit cube $\mathcal{D} = [0,1]^3$, which is sliced at $x_3 = 0.5$ for the visualization of the results. An illustration is given in figure 5.1. Additionally, in the context of the spatial error analysis, the series of meshes is specifically built using a "refine by splitting" technique that allows to refine a given unstructured mesh by a factor of two on the whole domain. These mesh generation functionalities are provided by* `Gmsh`, *see* [23].

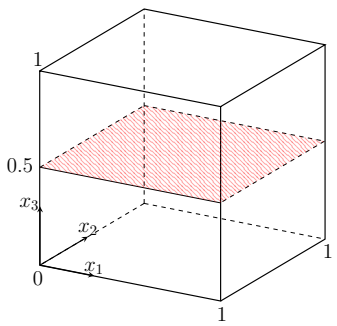


Figure 5.1. Illustration, here on the unit cube $[0,1]^3$, of the setup used to depict the numerical fields. All the simulations are performed in 3D but the data are displayed as a 2D field by using a slice (here in red) through the computational domain.

5.1. **Validation experiments.** In this section, we propose a few experiments to ensure that our method yields the correct approximate solution. First, in section 5.1.1, we discuss the approximation of Maxwell's equations, and provide a study of the order of accuracy, both in time and space. Then, in section 5.1.2, we highlight the generality of our method by considering a numerical approximation of the wave equation.

5.1.1. *Three-dimensional Maxwell's equations.* Let us consider on a spatial domain $\mathcal{D} \subset \mathbb{R}^3$ and for $t > 0$, the following non-dimensional Maxwell's equations

$$\begin{cases} \partial_t \mathbf{E} - \boldsymbol{\nabla} \times \mathbf{H} = 0, \\ \partial_t \mathbf{H} + \boldsymbol{\nabla} \times \mathbf{E} = 0, \end{cases} \tag{5.2}$$

where $\mathbf{E}(\mathbf{x}, t) = (E_1, E_2, E_3)^\intercal \in \mathbb{R}^3$ is the electric field and $\mathbf{H}(\mathbf{x}, t) = (H_1, H_2, H_3)^\intercal \in \mathbb{R}^3$ the magnetic field. For this system, the $m = 6$ conservative variables and the flux in a direction $\mathbf{n} = (n_1, n_2, n_3)^\intercal \in \mathbb{R}^3$ are respectively

$$\mathbf{u}(\mathbf{x}, t) = (E_1, E_2, E_3, H_1, H_2, H_3)^\intercal$$

and

$$\mathbf{q}(\mathbf{u}, \mathbf{n}) = \left(-\mathbf{n} \times \mathbf{H}^T,\ \mathbf{n} \times \mathbf{E}^T\right)^\intercal.$$

One simple exact time-domain solution satisfying (5.2) is a plane wave of the form

$$\mathbf{u}^{\text{inc}}(\mathbf{x}, t) = \begin{pmatrix} 0 \\ 0 \\ f(x_1 - t) \\ 0 \\ -f(x_1 - t) \\ 0 \end{pmatrix}, \tag{5.3}$$

with $f : \mathbb{R} \to \mathbb{R}$ a given function.

For our validation experiments, the test case relies on solving (5.2) on the unit cube $\mathcal{D} = [0, 1]^3$, until the final time $t_{\text{end}} = 1$ and with the following initial and boundary conditions:

$$\begin{cases} \mathbf{u}(\mathbf{x}, 0) = \mathbf{u}^{\text{inc}}(\mathbf{x}, 0), & \forall \mathbf{x} \in \mathcal{D}, \\ \mathbf{u}(\mathbf{x}, t) = \mathbf{u}^{\text{inc}}(\mathbf{x}, t), & \forall \mathbf{x} \in \partial\mathcal{D},\ \forall t \leq t_{\text{end}}. \end{cases}$$

In this case of the non-dimensional Maxwell's equations, we get $\lambda_{\max} = 1$ in the expression (4.1) of the time step Δt.

In order to evaluate the spatial order of the scheme, we choose a low frequency plane wave by setting $f(s) = \cos(2\pi s)$ in (5.3). As an illustration, the third component of the exact solution is displayed in figure 5.2. To ensure that the error component due to time integration vanishes, the time step is set to a very small value $\Delta t = 10^{-4}$. The spatial convergence results are shown in table 5.1a and figure 5.3a. As observed for instance in [24, 6], we found a spatial order of accuracy close to $\mu \simeq p + 1$ when using a DG $p = 2$ approximation, which validates the spatial part of the scheme. The temporal order is evaluated by taking $f(s) = s^2$ in (5.3). Taking such a quadratic function ensures an exact spatial integration of the solution thanks to the DG $p = 2$ approximation, which therefore allows us to isolate the error component due to time integration. The temporal convergence results are shown in table 5.1b and figure 5.3b. We observe that the temporal order is close to 2 which is consistent with the Crank-Nicolson method we are using and thus validates the temporal integration part of the scheme.

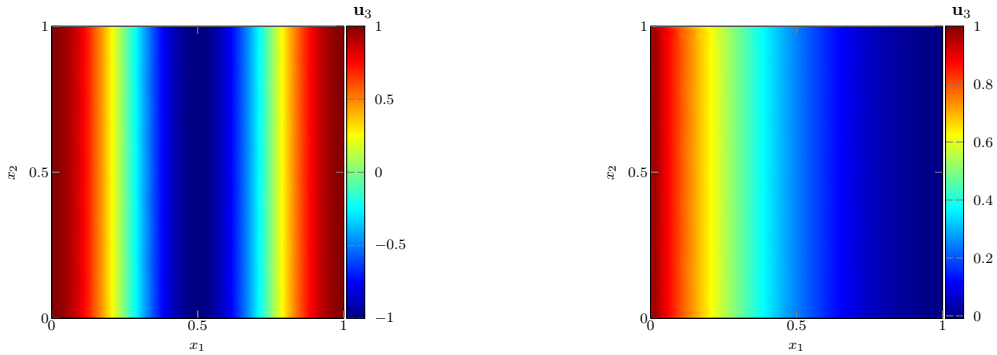
Table 5.1. Maxwell test case from section 5.1.1: study of the order of accuracy. The tables contain the values of the error e_r, as well as the values of the spatial order of accuracy μ with respect to the mesh size $h_{\min}$ (left table) and the temporal order of accuracy κ with respect to the time step Δt (right table).

Size $h_{\min}$	Error e_r	Order μ
$2.290 \cdot 10^{-2}$	$5.820 \cdot 10^{-3}$	—
$1.145 \cdot 10^{-2}$	$8.168 \cdot 10^{-4}$	2.8329
$5.725 \cdot 10^{-3}$	$1.059 \cdot 10^{-4}$	2.9474
$2.863 \cdot 10^{-3}$	$1.307 \cdot 10^{-5}$	3.0182
$1.431 \cdot 10^{-3}$	$1.642 \cdot 10^{-6}$	2.9973

(a) Spatial convergence.

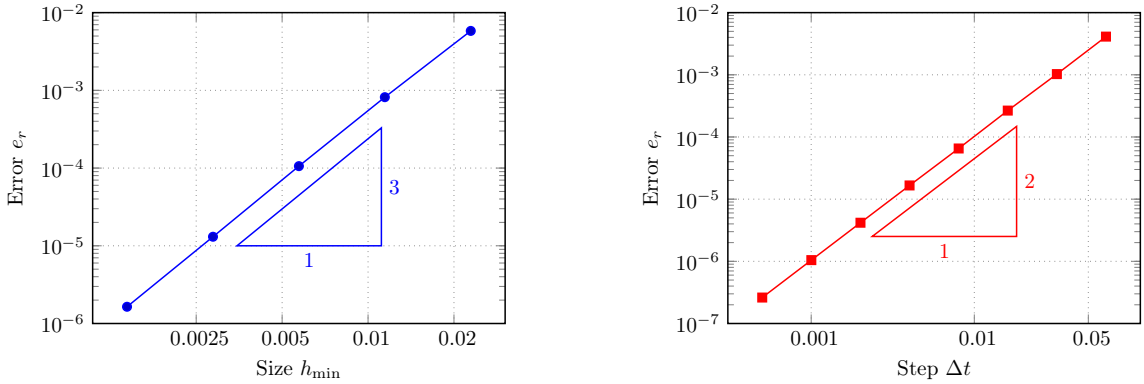
Step Δt	Error e_r	Order κ
$6.415 \cdot 10^{-2}$	$4.114 \cdot 10^{-3}$	—
$3.208 \cdot 10^{-2}$	$1.026 \cdot 10^{-3}$	2.0030
$1.604 \cdot 10^{-2}$	$2.658 \cdot 10^{-4}$	1.9495
$8.019 \cdot 10^{-3}$	$6.531 \cdot 10^{-5}$	2.0247
$4.009 \cdot 10^{-3}$	$1.661 \cdot 10^{-5}$	1.9750
$2.005 \cdot 10^{-3}$	$4.176 \cdot 10^{-6}$	1.9920
$1.002 \cdot 10^{-3}$	$1.046 \cdot 10^{-6}$	1.9980
$5.010 \cdot 10^{-4}$	$2.615 \cdot 10^{-7}$	1.9995

(b) Temporal convergence.



(a) Case 1: $f(s) = cos(2\pi s)$ **(b)** Case 2: $f(s) = s^2$

Figure 5.2. Maxwell test case from section 5.1.1: depiction of the third component of the exact solution $\mathbf{u}_3(\mathbf{x}, t) = E_3(\mathbf{x}, t)$, sliced at $x_3 = 0.5$. In each panel, we have taken a different value for the function $f(s)$ in (5.3). The left and right panels respectively illustrate the exact solutions used in the study of the spatial and temporal orders of accuracy.



(a) Spatial convergence. **(b)** Temporal convergence.

Figure 5.3. Maxwell test case from section 5.1.1: plot in `loglog` scale of the numerical convergence rates. The error e_r is displayed with respect to the mesh size $h_{\min}$ in the left panel, and with respect to the time step Δt in the right panel.

5.1.2. *Three-dimensional wave equation.* To insist on the generic aspect of the present method, we now perform an additional time order validation by considering the propagation, for $t \leq t_{\text{end}}$, of a three-dimensional wave in a homogeneous domain $\mathcal{D} = [0, 1]^3$ in which we impose a given perturbation. Mathematically, the problem reads

$$\begin{cases} \partial_{tt} w(\mathbf{x}, t) - c^2 \Delta w(\mathbf{x}, t) = 0, & \forall (\mathbf{x}, t) \in \mathcal{D},\ \forall t \leq t_{\text{end}}, \\ w(\mathbf{x}, 0) = g(\mathbf{x}, 0), & \forall \mathbf{x} \in \mathcal{D}, \\ w(\mathbf{x}, t) = g(\mathbf{x}, t), & \forall \mathbf{x} \in \partial\mathcal{D},\ \forall t \leq t_{\text{end}}, \end{cases} \tag{5.4}$$

with $w : \mathbb{R}^3 \times \mathbb{R} \to \mathbb{R}$ the unknown function, $g : \mathbb{R}^4 \to \mathbb{R}$ the known perturbation and c the wave velocity. For our applications, we set the wave velocity to $c \equiv 1$. To comply with the formalism described in section 2, the system (5.4) is rewritten into a system of conservation laws using the following definitions of the conserved variables and flux:

$$\mathbf{u}(\mathbf{x}, t) = (\partial_t w,\ \partial_1 w,\ \partial_2 w,\ \partial_3 w)^\intercal$$

and

$$\mathbf{q}(\mathbf{u}, \mathbf{n}) = (-c^2 \mathbf{n} \cdot \boldsymbol{\nabla}_{\mathbf{x}} w,\ -n_1 \partial_t w,\ -n_2 \partial_t w,\ -n_3 \partial_t w)^\intercal.$$

One can compute the exact time-domain solution of (5.4) through Kirchhoff's formula, namely

$$w^{ex}(\mathbf{x},t) = \frac{1}{4\pi}\partial_t\left(t\int_{|\mathbf{y}|=1} g(\mathbf{x}+ct\mathbf{y})ds(\mathbf{y})\right). \tag{5.5}$$

The integral term in (5.5) is numerically evaluated with an accurate Lebedev quadrature [28] of order $p = 27$ which uses 302 points on the unit sphere $\mathbb{S}^2$. Finally, the initial condition g is chosen as a compactly supported function of the form

$$g(\mathbf{x}) = \begin{cases} \left(\dfrac{(1-|\mathbf{x}|^2)}{\varepsilon^2}\right)^k & \text{if } \dfrac{|\mathbf{x}|}{\varepsilon^2} \leq 1, \\ 0 & \text{otherwise,} \end{cases}$$

with $\varepsilon \in \mathbb{R}$ and $k \in \mathbb{N}$. The coefficient ε enables us to control the diameter of the initial condition at $t = 0$. In our simulation, we set $\varepsilon = 0.20$ and $k = 6$ to ensure sufficient regularity on g.

With the aim of studying the time order of the scheme, the simulations are performed on a very fine mesh. To evaluate the behavior of our scheme both before and after the wave front came into contact with the boundaries of the domain, the error (5.1) is evaluated at two different final times, $t_{\text{end}} = 0.3$ and $t_{\text{end}} = 0.5$. In figure 5.4, the numerical values of $u_1 = \partial_t w$ (top panels) and $u_2 = \partial_1 w$ (bottom panels) are depicted. We observe no perturbations due to the boundary, and the numerical solution seems to be in accordance with the exact solution. To quantify this consistency, we report in table 5.2, and we display in figure 5.5, the values of the error e_r and the temporal order κ with respect to the time step and both final times. In each case, we observe that the temporal order remains approximately equal to 2, which further validates the consistency of our approach.

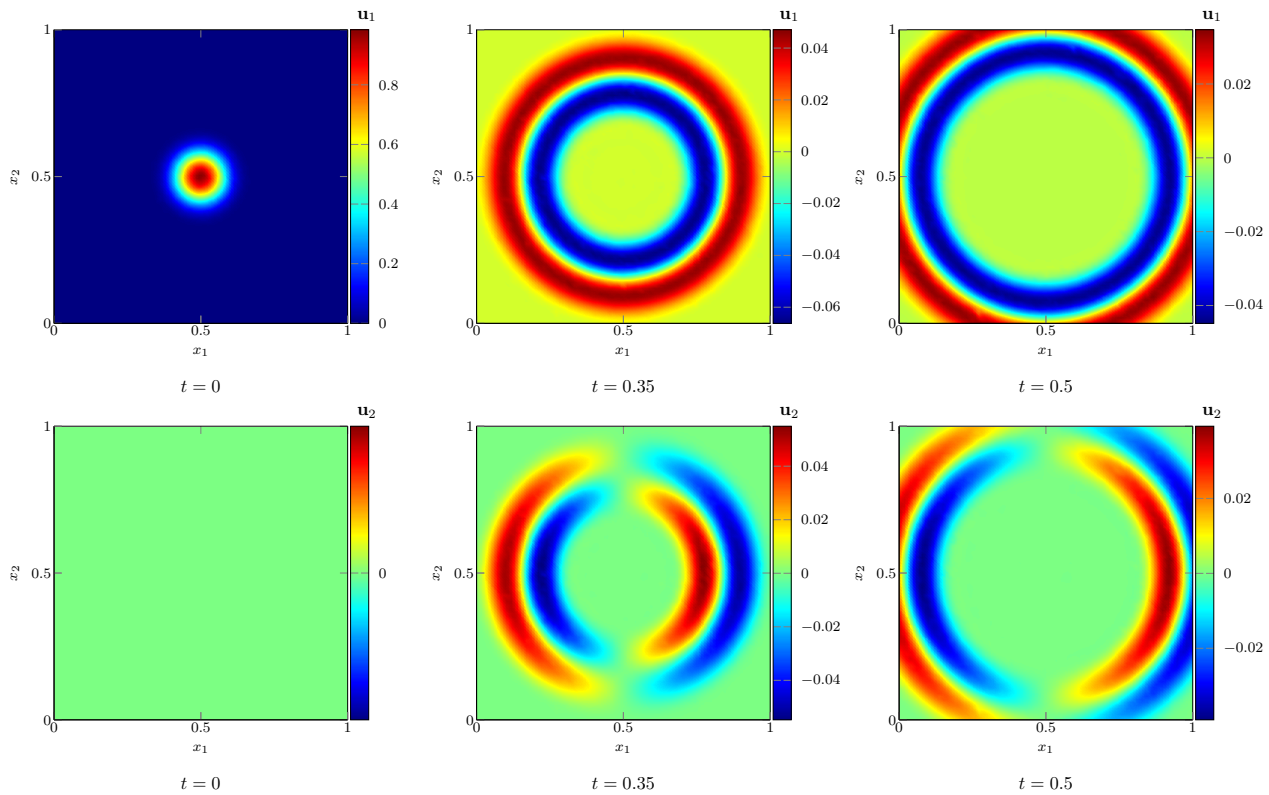


Figure 5.4. Wave equation test case from section 5.1.2: depiction of the first two components of the numerical approximations of $\mathbf{u}_1(\mathbf{x},t) = \partial_t w(\mathbf{x},t)$ (top panels) and $\mathbf{u}_2(\mathbf{x},t) = \partial_1 w(\mathbf{x},t)$ (bottom panels), sliced at $x_3 = 0.5$. From left to right, the solutions are depicted at time $t = 0$, $t = 0.35$ (before the wave hits the boundary) and $t = 0.5$ (after the wave has hit the boundary).

Table 5.2. Wave equation test case from section 5.1.2: study of the temporal order of accuracy. The table contains the values of the error e_r, as well as the values of the temporal order of accuracy κ with respect to the time step Δt. These errors and orders of accuracy are collected at time $t = 0.35$ and $t = 0.5$, respectively before and after the wave has hit the domain boundary.

	$t_{\text{end}} = 0.35$		$t_{\text{end}} = 0.50$	
Step Δt	Error e_r	Order κ	Error e_r	Order κ
$1.343 \cdot 10^{-2}$	$9.091 \cdot 10^{-3}$	—	$1.144 \cdot 10^{-2}$	—
$6.713 \cdot 10^{-3}$	$2.904 \cdot 10^{-3}$	1.6467	$3.520 \cdot 10^{-3}$	1.7012
$3.357 \cdot 10^{-3}$	$7.673 \cdot 10^{-4}$	1.9601	$8.788 \cdot 10^{-4}$	1.9559
$1.678 \cdot 10^{-3}$	$2.026 \cdot 10^{-4}$	1.9813	$2.327 \cdot 10^{-4}$	1.9716

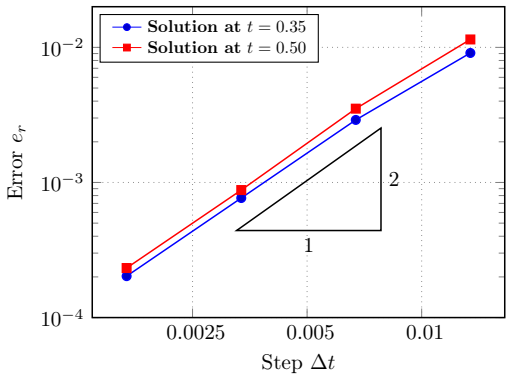


Figure 5.5. Wave equation from section 5.1.1: plot in `loglog` scale of the temporal convergence rates. The error e_r is displayed with respect to the time step Δt, at time $t = 0.35$ and $t = 0.5$.

5.2. **Unconditional stability.** It is commonplace to see appearing, when generating an unstructured mesh composed of tetrahedra, cells whose size can be several orders of magnitude smaller than the largest one. Depending on the problem under consideration and the physical solution, such finely discretized areas may not be necessary for accurate simulations. An example of this behavior is found when the spatial variations take place on scales much larger than the smallest cells of the mesh. A solution would be to modify the mesh to remove these unnecessarily refined areas. However, this is often impossible in an industrial context with regard to the time and the cost that such modifications would impose. With a classical explicit scheme, handling this type of configuration requires, because of the CFL condition, the use of very small time steps. As stated above, the method presented in this study is unconditionally stable and thus offers the possibility to increase the time step according to the nature of the solution and the desired accuracy.

For this numerical experiment, we once again consider the Maxwell test from section 5.1.1 with $f(s) = \cos(2\pi\nu s)$ in (5.3). It represents the illumination of a unit cube by plane wave of frequency ν, characterized by the function $f(s)$. For our simulations, we consider $t_{\text{end}} = 1$, $\nu \in \{1, 2\}$, and two mesh configurations. The first one, which we call $\mathcal{M}_1$, is based on a classical mesh of the unit cube with a refinement $r_c = 16$ imposed on its edges. The second one, called $\mathcal{M}_2$, is based on the geometric inclusion of a small torus in the center of the unit cube. This torus has a large radius $R = 0.1$ and a cross-sectional radius $r = 0.05$ We impose a refinement of $r_c = 8$ on the edges of the cube and a one of $r_t = 128$ on the torus circumference.

Remark. *The presence of the torus is only intended to force the generation of a conformal mesh with very small elements in a delimited area of space. In this test case, the dielectric properties of the cube and of the torus remain absolutely identical (i.e. vacuum). To get a better idea of the employed refinement, representations of the two meshes (sliced at $x_2 = 0.5$) are given in figure 5.6. In this figure, we clearly see the strong variation of the cell size induced by the torus presence.*

To illustrate the interest of our method, we report in tables 5.3 and 5.4 the results obtained with the `D3Q4P` implementation and the one achieved with an explicit `RK3DG` method. The errors between the exact and the numerical solutions are reported with respect to different CFL numbers, i.e. values of β in (4.1). Using the setup described in figure 5.1, we depict for the both meshes in figures 5.7 and 5.8 the third component of the

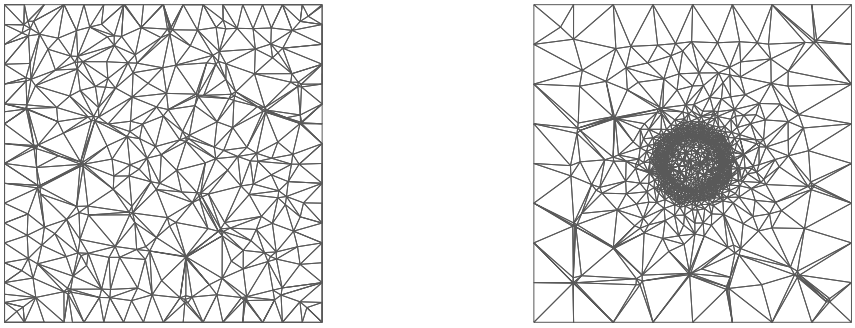


Figure 5.6. Depiction of the two meshes under consideration in section 5.2, sliced at $x_2 = 0.5$. Left panel: mesh $\mathcal{M}_1$, with uniformly spaced unstructured tetrahedra. Right panel: mesh $\mathcal{M}_2$, with uniformly spaced unstructured tetrahedra at the boundary, and with a local refinement at the center of the mesh.

numerical solution. These plots display the spatial variation of E_3 for the two frequencies $\nu \in \{2, 5\}$ and for several values of the CFL number β.

The starting point of this study was first to numerically determine the largest CFL value β for which the `RK3DG` remained stable. On the present test case and as indicated in section 4, the highest value we found was $\beta = 1.85$. After this threshold, the error quickly grows until it explodes for $\beta = 1.87$. Looking at tables 5.3 and 5.4 we first notice that for values of $\beta \leq 1.85$, both methods have converged in time, and their respective errors e_r are of the same order of magnitude. For small time steps, the error is dominated by the spatial accuracy that our scheme is able to achieve on the largest cells of the mesh. For a same given mesh and small time step, the error e_r is also impacted by the spatial variations of the solution. For the high frequency $\nu = 5$, we observe slightly larger error than for the low frequency $\nu = 2$.

For larger values of the CFL number ($\beta > 1.85$) and as claimed in the previous sections, we observe in tables 5.3 and 5.4 that our relaxation scheme remains stable even in the case $\beta = 185$. On figure 5.7, we display the progressive degradation of the solution as the CFL value goes up ($\beta = 1.85$, 37 and 92 from left to right), however the solution never explodes. On the second configuration (i.e. with the locally refined torus), h_{min} is ten times smaller than the one in the first configuration. The `RK3DG` CFL condition directly reflects this variation on the time step. Here, this drastic constraint imposed on the time step is clearly unnecessary to accurately approximate the solution. As shown in 5.4 for both cases $\nu \in \{2, 5\}$, our relaxation scheme is able to produce almost equivalent results with a CFL value $\beta = 18.5$, compared to the value $\beta = 1.85$ needed for the `RK3DG` scheme. On figure 5.8 and for the case $\nu = 2$, we note a barely visible deterioration for $\beta = 185$. For the case $\nu = 5$, the solution still looks very similar for $\beta = 18.5$, but becomes deformed for $\beta = 185$.

Tables 5.3 and 5.4 also contain the CPU times of each simulation for sequential (1 thread) and parallel (24 thread) runs. Since the algorithms, the languages, and the optimizations are different in both implementations, we would like to emphasize the fact that we do not seek to directly compare the execution times of both methods. Indeed, the objective is only to give an order of magnitude of the execution time. Nevertheless, the results in tables 5.3 and 5.4 illustrate the efficiency of the `D3Q4P` method, thereby placing it in the category of explicit methods despite it being implicit and unconditionally stable. Those results confirm thus the unconditional stability of our method, and its interest when one wishes to adapt the simulation time step in function of the nature of the solution, without having to deal with a constraining CFL condition imposed by the underlying mesh.

6. Performance and parallelism

In this section, we evaluate the performance and parallel scalability of the algorithm described in section 3. We seek to compare the `D3Q4` and `D3Q4P` implementations, as described in section 3.5.

To that end, we first compare in section 6.1 the memory footprint and the parallel scalability of the two algorithms. Then, in section 6.2, we discuss two ways to order the mesh graph, and their effects on parallel efficiency.

These comparisons are done on a server with an Intel Xeon E5-2680 v3 processor (24 physical cores, 2.50 GHz). We consider tetrahedral meshes of the unit cube $\mathcal{D} = [0, 1]^3$, represented by a single integer, called the "refinement", which is nothing but the number of tetrahedra on each side of the unit cube. These meshes are once again generated using `gmsh`.

Table 5.3. Results for the mesh $\mathcal{M}_1$ described in section 5.2. For this mesh, we get $h_{\min} \simeq 2.28 \times 10^{-3}$. The numerical test is done with the exact solution from section 5.1.1, with $f(s) = \cos(\nu\pi s)$, $\nu \in \{2, 5\}$. We collect the error e_r and the CPU time with respect to the value of the CFL number, for both values of ν and for the `RK3DG` and `D3Q4P` methods.

			Error e_r		CPU (s)	
Method	CFL β	Δt	$\nu = 2$	$\nu = 5$	1 thread	24 threads
`RK3DG`	0.37	0.00084	0.00032	0.00609	360.01	72.54
`D3Q4P`	0.37	0.00084	0.00046	0.00627	96.27	15.53
`RK3DG`	0.93	0.00211	0.00032	0.00610	146.81	29.19
`D3Q4P`	0.93	0.00211	0.00047	0.00657	38.63	6.52
`RK3DG`	1.85	0.00422	0.00032	0.00609	77.32	16.07
`D3Q4P`	1.85	0.00422	0.00062	0.00891	19.23	3.26
`D3Q4P`	3.70	0.00845	0.00162	0.02397	9.92	1.64
`D3Q4P`	9.25	0.02112	0.00960	0.14851	3.95	0.63
`D3Q4P`	18.50	0.04223	0.03990	0.42444	2.00	0.33
`D3Q4P`	37.00	0.08447	0.14919	0.34411	1.02	0.17
`D3Q4P`	92.50	0.21117	0.25771	0.67218	0.45	0.08
`D3Q4P`	185.00	0.42234	0.45671	0.49513	0.29	0.05

Table 5.4. Results for the mesh $\mathcal{M}_2$ described in section 5.2. For this mesh, we get $h_{\min} \simeq 2.47 \times 10^{-4}$. The numerical test is done with the exact solution from section 5.1.1, with $f(s) = \cos(\nu\pi s)$, $\nu \in \{2, 5\}$. We collect the error e_r and the CPU time with respect to the value of the CFL number, for both values of ν and for the `RK3DG` and `D3Q4P` methods.

			Error e_r		CPU (s)	
Method	CFL β	Δt	$\nu = 2$	$\nu = 5$	1 thread	24 threads
`RK3DG`	0.37	0.00009	0.00070	0.01238	4,607.95	785.28
`D3Q4P`	0.37	0.00009	0.00103	0.01467	1,524.45	234.48
`RK3DG`	0.93	0.00023	0.00070	0.01238	2,189.76	384.79
`D3Q4P`	0.93	0.00023	0.00103	0.01467	613.44	90.84
`RK3DG`	1.85	0.00046	0.00070	0.01238	1,121.96	212.60
`D3Q4P`	1.85	0.00046	0.00103	0.01467	304.41	45.14
`D3Q4P`	3.70	0.00091	0.00103	0.01468	153.09	22.40
`D3Q4P`	9.25	0.00228	0.00104	0.01479	61.60	8.96
`D3Q4P`	18.50	0.00456	0.00115	0.01619	30.76	4.53
`D3Q4P`	37.00	0.00912	0.00210	0.02992	15.34	2.46
`D3Q4P`	92.50	0.02281	0.01107	0.16589	6.17	0.92
`D3Q4P`	185.00	0.04562	0.04509	0.40344	3.10	0.48

6.1. **Scalability and memory footprint.** Recall that, although the `D3Q4` and `D3Q4P` implementations solve the same problem, they have different specificities. For the sake of completeness, we recall them in table 3.1.

We expect the `D3Q4P` implementation, thanks to its superior scalability, to be much faster than the `D3Q4` one on larger meshes and with a larger number of threads. Also, we expect the memory footprint of the `D3Q4P` implementation to be lower than that `D3Q4` one, since it is not necessary to store the global DG matrix.

Remark. *Note that, to improve the CPU time of the serial implementation of `D3Q4P`, it is possible to store the local matrices instead of computing them on the fly. This drastically increases the memory footprint (since $d + 1 = 4$ matrices of size 10×10 have to be stored for each tetrahedron), but also decreases the CPU time of the serial code. However, in the context of the parallel implementation, storing these matrices barely decreases the CPU time, since it also decreases the work done by each thread. Therefore, storing the local matrices in the `D3Q4P` implementation is not useful, since it only increases memory consumption for a small decrease in CPU time.*

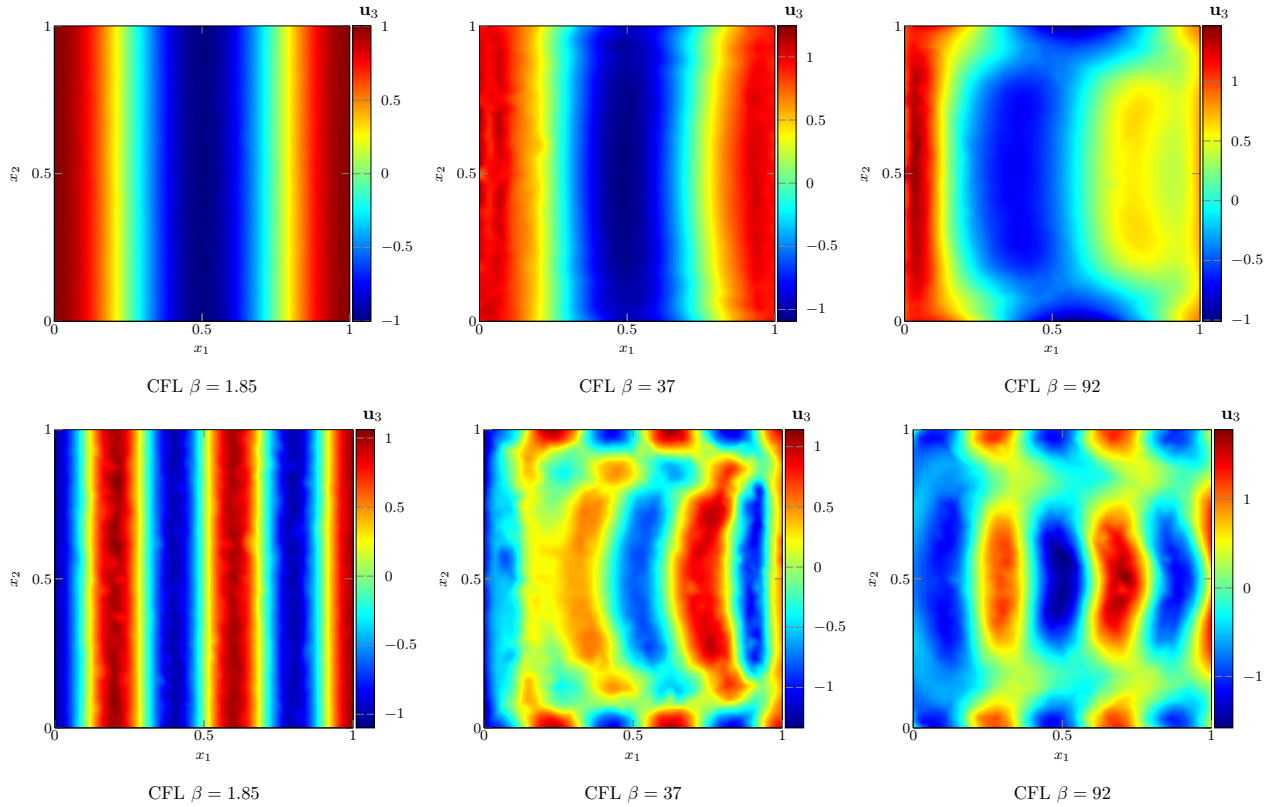


Figure 5.7. Maxwell test case from section 5.2: graphical representation of the third component $\mathbf{u}_3(\mathbf{x}, t) = E_3(\mathbf{x}, t)$ of the approximate solution, sliced at $x_3 = 0.5$, using the first mesh $\mathcal{M}_1$. Top panels: $\nu = 2$ in $f(s)$; bottom panels: $\nu = 5$ in $f(s)$. From left to right, the CFL coefficients are 1.85, 37 and 92.

These expectations are confirmed in table 6.1, where we compare the `D3Q4` and `D3Q4P` methods for refinements levels 8, 16, 32 and 48, on the Maxwell experiment from section 5.1.1 and with 1000 time steps. We limit the `D3Q4P` method to 4 threads for a fair comparison with the `D3Q4` implementation. We remark that the heap memory footprint of the `D3Q4P` implementation is much lower than that of the `D3Q4` method, as expected. The CPU time taken by the serial version of the `D3Q4P` is also lower than that of the `D3Q4` method, which may be explained by more cache misses due to having to recover the global DG matrix in memory. This would also explain the better scalability of the `D3Q4P` method.

Table 6.1. CPU times and memory footprints, for several refinements levels; comparison between the serial and parallel versions of the `D3Q4` and `D3Q4P` codes, with 4 threads.

refinement		`D3Q4` (it/s)			`D3Q4P` (it/s)			heap (MB)	
level	elements	serial	parallel	scalability	serial	parallel	scalability	`D3Q4`	`D3Q4P`
8	1808	24.67	55.55	2.251	72.58	231.9	3.195	21.41	11.85
16	9199	4.87	7.69	1.579	11.34	44.88	3.958	389.6	42.50
32	56967	0.69	1.029	1.491	1.698	5.938	3.497	2483	266.5
48	175138	0.22	0.334	1.518	0.531	1.859	3.501	7554	808.9

To further explore this scalability, we turn to table 6.2, where we compare the serial and parallel versions of the `D3Q4P` implementation, this time not limited to 4 threads but using the full 24 physical threads. We first note that the `D3Q4` implementation consumes about 10 times more heap memory than the `D3Q4P` method for larger meshes. We also remark that, although the parallel code runs much faster than the serial code, the scalability is not perfect on 24 threads. This is possibly due to the fact that tasks have to be distributed to the parallel threads at the start of each parallel region. Therefore, since larger meshes contain larger parallel

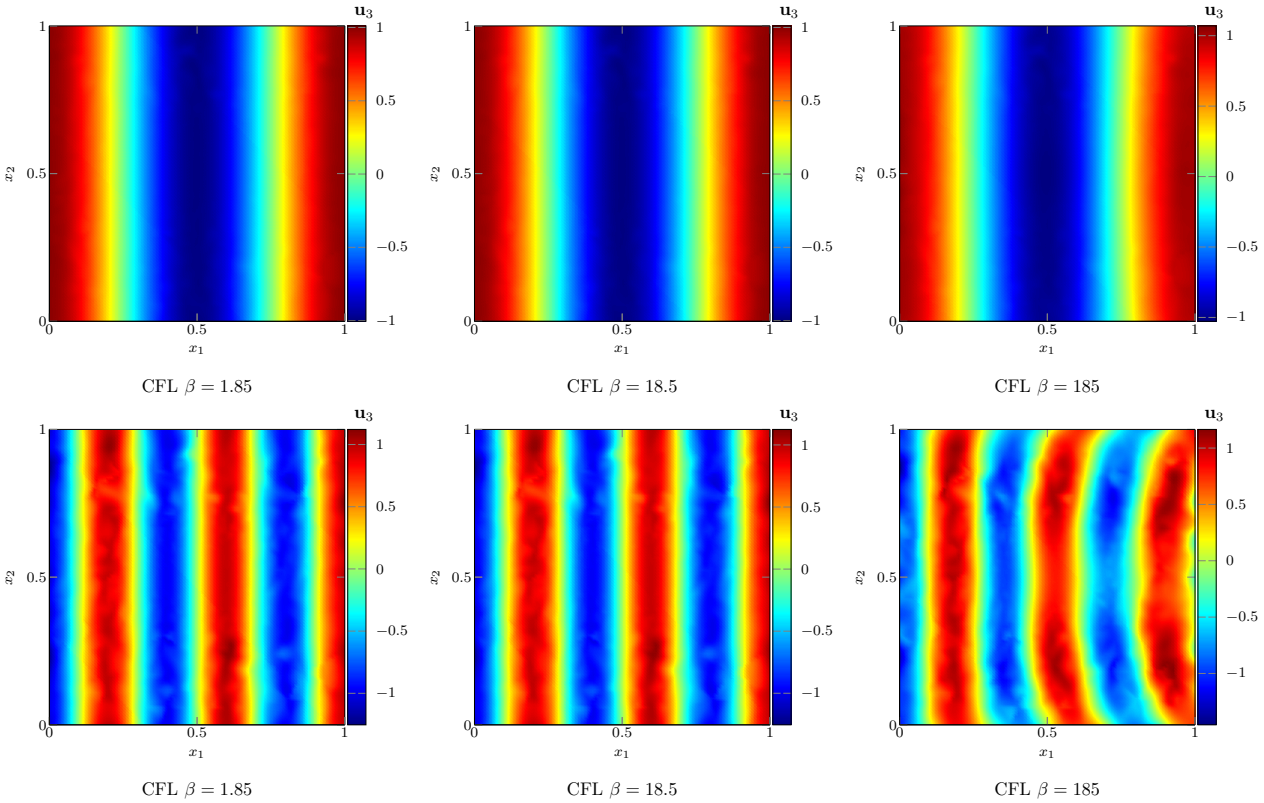


Figure 5.8. Maxwell test case from section 5.2: graphical representation of the third component $\mathbf{u}_3(\mathbf{x}, t) = E_3(\mathbf{x}, t)$ of the approximate solution, sliced at $x_3 = 0.5$, using the second mesh $\mathcal{M}_2$. Top panels: $\nu = 2$ in $f(s)$; bottom panels: $\nu = 5$ in $f(s)$. From left to right, the CFL coefficients are 1.85, 18.5 and 185.

regions, we expected the scalability to increase as the mesh refinement increases. We study the effects of the graph ordering algorithm on the size of the parallel regions in the next section.

Table 6.2. CPU times, for several refinements levels; comparison between the serial and parallel versions of the `D3Q4P` code, with 24 threads.

refinement		it/s		µs/dof/it			
level	elements	serial	parallel	serial	parallel	scalability	heap
8	1808	72.58	346.1	0.425	0.089	4.769	11.85 MB
16	9199	11.34	102.2	0.569	0.063	9.012	42.50 MB
32	56967	1.698	20.19	0.664	0.056	11.89	266.5 MB
48	175138	0.531	7.753	0.718	0.049	14.60	808.9 MB
64	386806	0.236	3.579	0.747	0.049	15.17	1.777 GB
72	544030	0.165	2.531	0.765	0.050	15.34	2.515 GB

6.2. **Graph ordering.** According to section 3.2.3, one has to choose an algorithm to perform the topological sort of the graph $\mathcal{G}$. Two approaches have been discussed: Breadth-First Search (BFS) and Depth-First Search (DFS). Intuitively, the BFS seems more suited to the current problem. In the example of figure 3.3, we display an example of topological sorts based on BFS and DFS in figure 6.1, as well as the parallel regions resulting from these orderings. It is clear that the DFS ordering, below the BFS ordering, is not well suited at all to the current problems, since it results in more, smaller parallel regions. This remark is confirmed in table 6.3, where we display the number of parallel regions, as well as the average and the maximal number of cells in each region, resulting from BFS and DFS orderings of the meshes already deployed in this section. The DFS

ordering is computed with Kosaraju's algorithm from [1], implemented in the `petgraph`[2] library of the Rust language, while the BFS ordering is computed thanks to Kahn algorithm from [27].

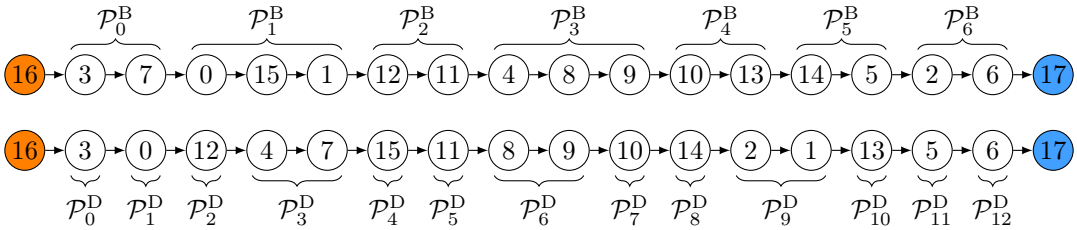


Figure 6.1. Comparison of Breadth-First Search (BFS, above) and Depth-First Search (DFS, below) orderings of the graph from figure 3.3. For each ordering, the regions that can be treated in parallel are displayed; $\mathcal{P}_p^{\mathrm{B}}$ represents the parallel region stemming from the BFS order, while $\mathcal{P}_p^{\mathrm{D}}$ represents the ones obtained with the DFS order. It is apparent that BFS is more suited to the current graph, since it yields more and larger parallel regions.

Table 6.3. Distribution of parallel regions with respect to the topological sorting algorithm used. We present the results using the first kinetic velocity $v_0 = (1, 1, 1)$, but the conclusions are similar with the other three. We have denoted by n_p the number of parallel regions, $n_p^{\max}$ the number of elements in the largest parallel region, and n_p^{avg} the average number of elements in a parallel region.

refinement		BFS order			DFS order		
level	elements	n_p	n_p^{avg}	$n_p^{\max}$	n_p	n_p^{avg}	$n_p^{\max}$
8	1808	73	24	42	1331	1	5
16	9199	152	60	130	7026	1	5
32	56967	299	190	431	43695	1	6
48	175138	444	394	934	135021	1	6
64	386806	592	653	1475	298925	1	8
72	544030	676	804	1939	420828	1	7

7. Conclusion and further work

In this work, we have proposed a new Discontinuous Galerkin (DG) approximation of a hyperbolic system, based on a vectorial kinetic representation. The time stepping is explicit but is not constrained by a CFL stability condition. We have applied this method to Maxwell's equations and to the wave equation.

We have proposed two implementations of the kinetic DG method. The first is based on a classical assembly of the linear system of the DG transport matrix; this linear system is triangular and can be solved by an adequate matrix solver, such as KLU. The second implementation is based on a downwind algorithm that is more memory efficient and has good parallelization opportunities.

In several numerical experiments, we have shown that the method has the expected features: excellent parallel efficiency, low memory footprint like an explicit scheme, and good accuracy. The main application of the method is to treat the case of meshes with small cells to compute low frequency solutions.

There remains several extensions to be explored in order for our approach to be fully useful in practical applications. First it would be interesting to couple it with another classical explicit RKDG method. This would make it possible to couple an explicit scheme, in regions where the cells are large, to the kinetic method, in refined regions. Other points that still deserve some work are related to the treatment of source terms and boundary conditions. Some strategies to deal with these issues are discussed in [16, 19, 21]. Finally, another important improvement is related to parallelism. For the moment, the method is well adapted to shared-memory multiprocessing (generally addressed by OpenMP). The particular structure of the downwind algorithm makes it difficult to extend it efficiently to distributed-memory multiprocessing (generally addressed by MPI). To be able to use MPI, we could for instance adopt a hybrid Implicit-Explicit (IMEX) strategy such as the one given in [18].

[2] https://docs.rs/petgraph

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data structures and algorithms*. Addison-Wesley, Reading, Mass, 1983.

[2] D. Aregba-Driollet and R. Natalini. Discrete Kinetic Schemes for Multidimensional Systems of Conservation Laws. *SIAM J. Numer. Anal.*, 37(6):1973–2004, January 2000.

[3] U. Ayachit. *The ParaView guide : updated for ParaView version 4.3*. Kitware, Clifton Park, New York, 2015.

[4] J. Badwaik, M. Boileau, D. Coulette, E. Franck, Ph. Helluy, C. Klingenberg, L. Mendoza, and H. Oberlin. Task-Based Parallelization of an Implicit Kinetic Scheme. *ESAIM: Proceedings and Surveys*, 63:60–77, 2018.

[5] M. Baudin, C. Berthon, F. Coquel, R. Masson, and Q. H. Tran. A relaxation method for two-phase flow models with hydrodynamic closure law. *Numer. Math.*, 99(3):411–440, nov 2004.

[6] L. Berardocco, M. Kronbichler, and V. Gravemeier. A hybridizable discontinuous Galerkin method for electromagnetics with a view on subsurface applications. *Comput. Methods Appl. Mech. Engrg.*, 366:113071, July 2020.

[7] M. Boileau, B. Bramas, E. Franck, Ph. Helluy, and L. Navoret. Parallel lattice-boltzmann transport solver in complex geometry. working paper or preprint, December 2019.

[8] J. D. Booth, N. D. Ellingwood, H. K. Thornquist, and S. Rajamanickam. Basker: Parallel sparse LU factorization utilizing hierarchical parallelism and data layouts. *Parallel Comput.*, 68:17–31, oct 2017.

[9] F. Bouchut. Construction of BGK Models with a Family of Kinetic Entropies for a Given System of Conservation Laws. *J. Stat. Phys.*, 95(1/2):113–170, 1999.

[10] B. Bramas and A. Ketterlin. Improving parallel executions by increasing task granularity in task-based runtime systems using acyclic DAG clustering. *PeerJ Comput. Sci.*, 6:e247, jan 2020.

[11] N. Castel, G. Cohen, and M. Duruflé. Application of discontinuous Galerkin spectral method on hexahedral elements for aeroacoustic. *J. Comp. Acous.*, 17(02):175–196, jun 2009.

[12] X. Chen, Y. Wang, and H. Yang. NICSLU: An Adaptive Sparse Matrix Solver for Parallel Circuit Simulation. *IEEE Trans. Comput. Aid. D.*, 32(2):261–274, feb 2013.

[13] B. Cockburn and C.-W. Shu. Runge–Kutta discontinuous Galerkin methods for convection-dominated problems. *J. Sci. Comput*, 16(3):173–261, 2001.

[14] F. Coquel and B. Perthame. Relaxation of energy and approximate riemann solvers for general pressure laws in fluid dynamics. *SIAM J. Numer. Anal.*, 35(6):2223–2249, December 1998.

[15] D. Coulette, C. Courtès, E. Franck, and L. Navoret. Vectorial Kinetic Relaxation Model with Central Velocity. Application to Implicit Relaxations Schemes. *Commun. Comput. Phys.*, 27(4):976–1013, jun 2020.

[16] D. Coulette, E. Franck, Ph. Helluy, M. Mehrenberger, and L. Navoret. High-order implicit palindromic discontinuous Galerkin method for kinetic-relaxation approximation. *Comput. & Fluids*, 190:485–502, 2019.

[17] T. A. Davis and E. P. Natarajan. Algorithm 907: Klu, a direct sparse solver for circuit simulation problems. *ACM T. Math. Software*, 37(3):1–17, September 2010.

[18] S. Descombes, S. Lanteri, and L. Moya. Locally Implicit Time Integration Strategies in a Discontinuous Galerkin Method for Maxwell's Equations. *J. Sci. Comput*, 56(1):190–218, 2012.

[19] F. Drui, E. Franck, Ph. Helluy, and L. Navoret. An analysis of over-relaxation in a kinetic approximation of systems of conservation laws. *CR Mécanique*, 347(3):259–269, mar 2019.

[20] F. Dubois. Equivalent partial differential equations of a lattice Boltzmann scheme. *Comput. Math. Appl.*, 55(7):1441–1449, apr 2008.

[21] E. Franck, R. Hélie, Ph. Helluy, and L. Navoret. Equivalent equation of the vectorial kinetic scheme. Ongoing work.

[22] G. Gassner and D. A. Kopriva. A Comparison of the Dispersion and Dissipation Errors of Gauss and Gauss–Lobatto Discontinuous Galerkin Spectral Element Methods. *SIAM J. Sci. Comput.*, 33(5):2560–2579, 2011.

[23] C. Geuzaine and J.-F. Remacle. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *Internat. J. Numer. Methods Engrg.*, 79(11):1309–1331, 2009.

[24] J. Hesthaven and T. Warburton. Discontinuous Galerkin methods for the time-domain Maxwell's equations. *ACES Newsletter*, 19:10–29, 01 2004.

[25] J. S. Hesthaven and T. Warburton. *Nodal Discontinuous Galerkin Methods*. Springer New York, 2008.

[26] S. Jin and Z. P. Xin. The relaxation schemes for systems of conservation laws in arbitrary space dimensions. *Comm. Pure Appl. Math.*, 48(3):235–276, 1995.

[27] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, 1962.

[28] V. I. Lebedev. Quadratures on a sphere. *USSR Comp. Math. Math+*, 16(2):10–24, January 1976.

[29] T. Toulorge and W. Desmet. CFL Conditions for Runge–Kutta discontinuous Galerkin methods on triangular grids. *J. Comput. Phys.*, 230(12):4657–4678, jun 2011.

[30] G. B. Whitham. Linear and nonlinear waves, jun 1999.

[31] J.H Williamson. Low-storage Runge-Kutta schemes. *J. Comput. Phys.*, 35(1):48–56, March 1980.

Appendix A. Lagrange basis on the reference tetrahedron

We consider the reference tetrahedron whose nodes are numbered following figure 3.2. The coordinates of the nodes, as well as the associated Lagrange basis functions, are given in table A.1.

Appendix B. Derivation of the discontinuous Galerkin solver

To determine the coefficients $f_{L,i}(t)$ in the discontinuous Galerkin framework, one multiplies the PDE (3.1) by the basis function φ_j^L for all $\forall j \in \{0, \dots, 9\}$, before integrating the resulting equation on L. In our case,

Table A.1. Point coordinates and Lagrange basis functions on the reference tetrahedron.

point	coordinates	basis function
P_0	$(0,0,0)$	$\varphi_0(x_1,x_2,x_3) = (x_1+x_2+x_3-1)(2x_1+2x_2+2x_3-1)$
P_1	$(1,0,0)$	$\varphi_1(x_1,x_2,x_3) = x_1(2x_1-1)$
P_2	$(0,1,0)$	$\varphi_2(x_1,x_2,x_3) = x_2(2x_2-1)$
P_3	$(0,0,1)$	$\varphi_3(x_1,x_2,x_3) = x_3(2x_3-1)$
P_4	$(\frac{1}{2},0,0)$	$\varphi_4(x_1,x_2,x_3) = -4x_1(x_1+x_2+x_3-1)$
P_5	$(\frac{1}{2},\frac{1}{2},0)$	$\varphi_5(x_1,x_2,x_3) = 4x_1x_2$
P_6	$(0,\frac{1}{2},0)$	$\varphi_6(x_1,x_2,x_3) = -4x_2(x_1+x_2+x_3-1)$
P_7	$(0,0,\frac{1}{2})$	$\varphi_7(x_1,x_2,x_3) = -4x_3(x_1+x_2+x_3-1)$
P_8	$(0,\frac{1}{2},\frac{1}{2})$	$\varphi_8(x_1,x_2,x_3) = 4x_2x_3$
P_9	$(\frac{1}{2},0,\frac{1}{2})$	$\varphi_9(x_1,x_2,x_3) = 4x_3x_1$

we get

$$\forall L \in \mathcal{M},\ \forall j \in \{0,\dots,9\},\ \int_L \partial_t f(\mathbf{x},t)\varphi_j^L(\mathbf{x})\,d\mathbf{x} + \int_L \mathbf{v}\cdot\boldsymbol{\nabla} f(\mathbf{x},t)\varphi_j^L(\mathbf{x})\,d\mathbf{x} = 0.$$

Arguing the divergence theorem yields

$$\int_L \partial_t f(\mathbf{x},t)\varphi_j^L(\mathbf{x})\,d\mathbf{x} - \int_L f(\mathbf{x},t)\,\mathbf{v}\cdot\boldsymbol{\nabla}\varphi_j^L(\mathbf{x})\,d\mathbf{x} + \int_{\partial L} f(\boldsymbol{\eta},t)\,\varphi_j^L(\boldsymbol{\eta})\,\mathbf{v}\cdot\mathbf{n}\,d\boldsymbol{\eta} = 0, \tag{B.1}$$

where $\mathbf{n}$ is the outer normal to ∂L and $\boldsymbol{\eta} \in \partial L$.

We now consider the behavior of each integral in (B.1) under the basis expansion (3.2). For the first integral, we get:

$$\begin{aligned}\int_L \partial_t f(\mathbf{x},t)\varphi_j^L(\mathbf{x})\,d\mathbf{x} &= \int_L \sum_{i=0}^{9} \partial_t f_{L,i}(t)\varphi_i^L(\mathbf{x})\varphi_j^L(\mathbf{x})\,d\mathbf{x}\\ &= \sum_{i=0}^{9} \partial_t f_{L,i}(t)\left(\int_L \varphi_i^L(\mathbf{x})\varphi_j^L(\mathbf{x})\,d\mathbf{x}\right)\\ &\simeq \sum_{i=0}^{9} \frac{f_{L,i}^n - f_{L,i}^{n-1}}{\Delta t}\left(\int_L \varphi_i^L(\mathbf{x})\varphi_j^L(\mathbf{x})\,d\mathbf{x}\right),\end{aligned}$$

where we have set $f_{L,i}^n := f_{L,i}(n\Delta t)$, and where we have approximated the time derivative $\partial_t f_{L,i}(t)$ using a first-order implicit Euler discretization. The second integral becomes:

$$\begin{aligned}\int_L f(\mathbf{x},t)\,\mathbf{v}\cdot\boldsymbol{\nabla}\varphi_j^L(\mathbf{x})\,d\mathbf{x} &= \int_L \sum_{i=0}^{9} f_{L,i}(t)\varphi_i^L(\mathbf{x})\,\mathbf{v}\cdot\boldsymbol{\nabla}\varphi_j^L(\mathbf{x})\,d\mathbf{x}\\ &= \sum_{i=0}^{9} f_{L,i}(t)\left(\int_L \varphi_i^L(\mathbf{x})\,\mathbf{v}\cdot\boldsymbol{\nabla}\varphi_j^L(\mathbf{x})\,d\mathbf{x}\right)\\ &\simeq \sum_{i=0}^{9} f_{L,i}^n\left(\int_L \varphi_i^L(\mathbf{x})\,\mathbf{v}\cdot\boldsymbol{\nabla}\varphi_j^L(\mathbf{x})\,d\mathbf{x}\right),\end{aligned}$$

where we have used the implicit Euler discretization to set $f_{L,i}(t) \simeq f_{L,i}(n\Delta t) = f_{L,i}^n$.

For the third integral, we leverage the fact that the cell L is a tetrahedron with 4 faces. For $\alpha \in \{0,\dots,3\}$, we denote by ∂L_α the α-th face of L, and by $\mathbf{n}_\alpha$ the outer normal vector to ∂L_α, as depicted in figure 3.1. Therefore, the third integral reads:

$$\int_{\partial L} f(\boldsymbol{\eta},t)\,\varphi_j^L(\boldsymbol{\eta})\,\mathbf{v}\cdot\mathbf{n}\,d\boldsymbol{\eta} = \sum_{\alpha=0}^{3}\int_{\partial L_\alpha} f(\boldsymbol{\eta},t)\,\mathbf{v}\cdot\mathbf{n}_\alpha\,\varphi_j^L(\boldsymbol{\eta})\,d\boldsymbol{\eta}. \tag{B.2}$$

We now need to provide an approximation of the flux $f(\boldsymbol{\eta},t)\,\mathbf{v}\cdot\mathbf{n}_\alpha$ through the face ∂L_α. Denoting by R_α the neighboring tetrahedron of L through the face ∂L_α, we choose the following classical upwind discretization:

$$f(\boldsymbol{\eta},t)\,\mathbf{v}\cdot\mathbf{n}_\alpha \simeq f_L(\boldsymbol{\eta},t)\;(\mathbf{v}\cdot\mathbf{n}_\alpha)_+ + f_{R_\alpha}(\boldsymbol{\eta},t)\;(\mathbf{v}\cdot\mathbf{n}_\alpha)_-\,,$$

with $(\mathbf{v}\cdot\mathbf{n}_\alpha)_+ = \max(0,\mathbf{v}\cdot\mathbf{n}_\alpha)$ and $(\mathbf{v}\cdot\mathbf{n}_\alpha)_- = \min(0,\mathbf{v}\cdot\mathbf{n}_\alpha)$. Applying this approximation to (B.2), together with the implicit Euler time discretization, yields the following sequence of approximations:

$$\begin{aligned}
\int_{\partial L} f(\boldsymbol{\eta},t)\,\varphi_j^L(\boldsymbol{\eta})\,\mathbf{v}\cdot\mathbf{n}\,d\boldsymbol{\eta} &\simeq \sum_{\alpha=0}^{3}\int_{\partial L_\alpha}\left[f_L(\boldsymbol{\eta},t)\;(\mathbf{v}\cdot\mathbf{n}_\alpha)_+ + f_{R_\alpha}(\boldsymbol{\eta},t)\;(\mathbf{v}\cdot\mathbf{n}_\alpha)_-\right]\varphi_j^L(\boldsymbol{\eta})\,d\boldsymbol{\eta}\\
&= \sum_{\alpha=0}^{3}\int_{\partial L_\alpha}\Bigg[\left(\sum_{i=0}^{9} f_{L,i}(t)\varphi_i^L(\boldsymbol{\eta})\right)\;(\mathbf{v}\cdot\mathbf{n}_\alpha)_+\\
&\qquad + \left(\sum_{i=0}^{9} f_{R_\alpha,i}(t)\varphi_i^{R_\alpha}(\boldsymbol{\eta})\right)\;(\mathbf{v}\cdot\mathbf{n}_\alpha)_-\Bigg]\varphi_j^L(\boldsymbol{\eta})\,d\boldsymbol{\eta}\\
&\simeq \sum_{i=0}^{9} f_{L,i}^n\left[\sum_{\alpha=0}^{3}\left(\int_{\partial L_\alpha}\varphi_i^L(\boldsymbol{\eta})\varphi_j^L(\boldsymbol{\eta})\,d\boldsymbol{\eta}\right)\;(\mathbf{v}\cdot\mathbf{n}_\alpha)_+\right] +\\
&\quad \sum_{i=0}^{9} f_{R_\alpha,i}^n\left[\sum_{\alpha=0}^{3}\left(\int_{\partial L_\alpha}\varphi_i^{R_\alpha}(\boldsymbol{\eta})\varphi_j^L(\boldsymbol{\eta})\,d\boldsymbol{\eta}\right)\;(\mathbf{v}\cdot\mathbf{n}_\alpha)_-\right].
\end{aligned}$$

Plugging these three integral terms into (B.1) yields the DG solver (3.3).