J. Plasma Phys. (2016), *vol.* 82, 435820501 © Cambridge University Press 2016
doi:10.1017/S0022377816000696

Computational methods for plasma fluid models

G. Fuhr[1],[†], **P. Beyer**[1] **and S. Benkadda**[1]

[1]Aix Marseille Univ, CNRS, PIIM, Faculte de Saint Jerome, C631, 13397 Marseille Cedex 20, France

(Received 8 July 2015; revised 22 July 2016; accepted 22 July 2016)

Challenges in plasma physics are wide. Investigation and advances are made in experiments but at the same time, to understand and to reach the experimental limits, accurate numerical simulations are required from systems of nonlinear equations. The numerical challenges of solving the associated fluid equations are discussed in this paper. Using the framework of the finite difference discretization, the most widely used methods for the problems linked to the diffusion or advection operators are presented.

1. Introduction

Magnetized plasmas are complex systems governed by a wide range of instabilities, geometrical effects and wave interactions. This leads to an extremely large range of important spatial and temporal scales governing the evolution of the system (see Wesson 1997; Freidberg 2007). In a magnetically confined fusion plasma, the phenomena can involve short lengths and time scales of the order of a millimetre and tens of microseconds, respectively, or, at the opposite time scales of the order of some seconds and length scales of the order of the tokamak minor or major radii, i.e. metres. As a consequence, a variety of models exists and cover a large range of physics, from the plasma core to the edge and/or the scrape off layer. As a consequence, simulation models such as gyro-kinetic models or magneto-hydrodynamical (MHD) models (see Wesson 1997), have to be restricted to a subrange of time scales and spatial interactions.

Fluid modelling in plasma physics is based on the integration of distribution functions for electrons and ions. The derivation of associated moments, plus self-consistent dynamics of the electric and magnetic fields, leads to evolution equations for macroscopic quantities like electronic density n_e, electron and ion temperature (T_e, T_i), mass velocity ($v_\parallel$), electrostatic (ϕ) and electromagnetic (ψ) potentials, etc... (see Braginskii 1965).

Moreover, due to the variety of instabilities present, models are heterogeneous. The consequence, for associated simulations codes, is that physical assumptions cannot be used to extract global methods for numerical implementations. However, dealing with the associated mathematical aspects and properties of the corresponding equations allows for a simple classification between two broad families of operators: linear ($L(\cdot)$)

† Email address for correspondence: guillaume.fuhr@univ-amu.fr

and nonlinear ($NL(\cdot)$) operators:

$$\frac{\partial}{\partial t}\begin{bmatrix}\Omega\\ p_e\\ \psi\\ v_\parallel\\ \ldots\end{bmatrix}=L\left(\begin{bmatrix}\phi\\ p_e\\ \psi\\ v_\parallel\\ \ldots\end{bmatrix}\right)+NL\left(\begin{bmatrix}\phi\\ p_e\\ \psi\\ v_\parallel\\ \ldots\end{bmatrix}\right) \tag{1.1}$$

$$\Omega=\nabla_\perp^2\phi, \tag{1.2}$$

where fluid moments are used to derive the dynamics in (1.1). The vorticity field Ω is defined in (1.2). Linear terms are composed of a mix of first and second derivatives and can be seen as a combination of advection and diffusion processes. These processes are associated with resistivity or assumed viscosity of the plasma for the diffusion process. Concerning linear advection, an example is the centrifugal force acting on particles moving around a curved magnetic field. Nonlinear terms correspond typically to a nonlinear advection by a velocity drift (represented by the operator $\boldsymbol{u_D}\boldsymbol{\cdot}\nabla$) or effects linked to the non-uniformity of the magnetic field which leads to the tearing instabilities (operator $\boldsymbol{B}\boldsymbol{\cdot}\nabla$) (see Biskamp 1993),

$$\left.\begin{aligned}\boldsymbol{B}\boldsymbol{\cdot}\nabla f&=\left(\nabla\times\psi\frac{\boldsymbol{B}_0}{B_0}\right)\boldsymbol{\cdot}\nabla f\simeq(\partial_x\psi\,\partial_y f-\partial_y\psi\,\partial_x f)\\ \boldsymbol{u_D}\boldsymbol{\cdot}\nabla f&=\left(\nabla\times\phi\frac{\boldsymbol{B}_0}{B_0}\right)\boldsymbol{\cdot}\nabla f\simeq(\partial_x\phi\partial_y f-\partial_y\phi\partial_x f).\end{aligned}\right\} \tag{1.3}$$

The general form of the linear operator is

$$L(f)=\frac{\partial}{\partial x_i}[A(\boldsymbol{x})f]+\frac{\partial^2}{\partial x_i\partial x_j}[D(\boldsymbol{x})f], \tag{1.4}$$

where $\boldsymbol{x}=\{x_0,x_1,\ldots\}$ represents the position vector, $A(\boldsymbol{x})$ the advection coefficient and $D(\boldsymbol{x})$ the diffusion coefficient. In computational sciences, linear operators are divided into three classes, i.e. hyperbolic, parabolic and elliptic operators. The classification is determined by the nature of the eigenvalues of the associated operator (see Lomax, Pulliam & Zingg 2001). The canonical form of an advection process corresponds to a hyperbolic equation, diffusion corresponds to a parabolic equation and elliptic operators correspond to Poisson-type equations. As a consequence, each kind of equation has to be treated with appropriate methods to reproduce accurately the associated mathematical and physical properties. For example, since information propagates with a finite velocity in an advection process but with an infinite velocity in a diffusion process, the methods used in both cases can be different.

The choice of the discretization to advance the evolution of the considered quantities is not unique. Typical possible discretizations are finite difference (FD) approaches, finite elements or spectral methods of Fourier series (see Canuto *et al.* 1988; Ferszinger 2002). Only the finite difference approach is discussed here. In general in fusion plasma simulations, a combination of finite differences in the radial direction and Fourier modes in the poloidal and toroidal directions is used (see Jardin 2011). Fourier mode representation gives a higher precision in the numerical schemes, however such a representation can be used only when the geometry is periodic in the considered direction. Another approach concerns the field aligned coordinate systems (see Scott 1997), in which case generalized coordinates are used with the property that the grid is aligned to the equilibrium component of the magnetic field.

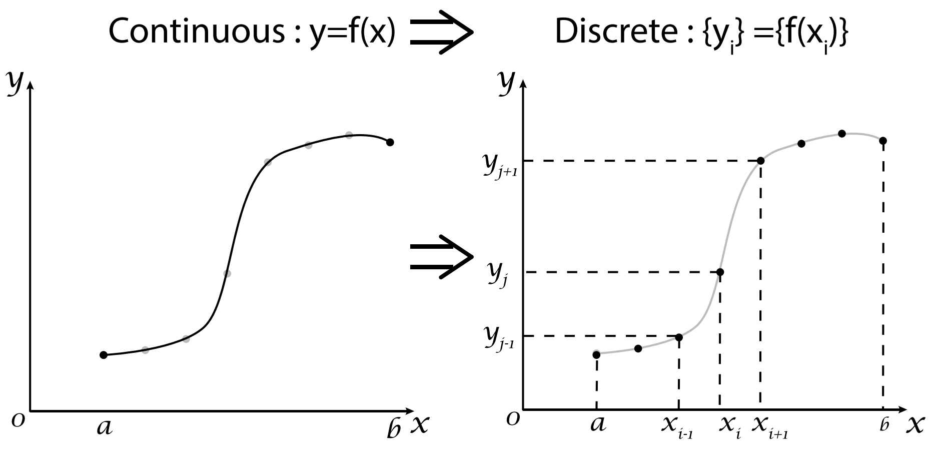

FIGURE 1. Discretization of a function $f(x)$.

The methods described in the following for the resolution of partial differential equations (PDE) focus on the FD approach but they can be used also with these other discretizations. The presented properties and limitations remain the same in the framework of these other representations.

An overview of the possible approaches concerning grid decomposition in plasma physics can be found in D'haeseleer (1991). Many numerical algorithms exist and it may be difficult to decide which one to adopt. The ultimate goal being to obtain the desired accuracy with least effort, or the maximum accuracy with the available resources. In the following sections, typical schemes are detailed, focusing on respective advantages and disadvantages. Section 2 presents the finite difference method, § 3 focuses on the main families of time schemes. These methods are applied to the typical problems of diffusion in § 4 and of advection in § 5. In a last part, the resolution of nonlinear terms is detailed in § 6.

2. Spatial discretization: the finite difference method

Let us consider a general PDE,

$$\partial_t \boldsymbol{u}(\boldsymbol{x}, t) = F(\boldsymbol{x}, t, \boldsymbol{u}, D\boldsymbol{u}, D^2\boldsymbol{u}), \tag{2.1}$$

where $\boldsymbol{u}(\boldsymbol{x}, t)$ is the unknown function, D and D^2 are first- and second-order derivatives in space, respectively. The FD approach consists of evaluating the function $\boldsymbol{u}(\boldsymbol{x}, t)$ on a discretized domain, as represented in figure 1. The derivatives in the differential equations are replaced by polynomial interpolations. This results in a large algebraic system of equations that has to be solved instead of the differential equation.

The discretization is made for illustration on a one-dimensional Cartesian grid. Derivation of operators and equations in a generalized system of coordinates appropriate for plasma physics can be found in D'haeseleer (1991). Equation (2.1) becomes

$$\partial_t u(x, t) = F(x, t, u, \partial_x u, \partial_x^2 u). \tag{2.2}$$

Unless stated otherwise, the unknown function $u(x, t)$ is always assumed to be smooth, meaning that it can be differentiated several times and that each derivative is a well-defined bounded function over an interval containing a particular point of interest x.

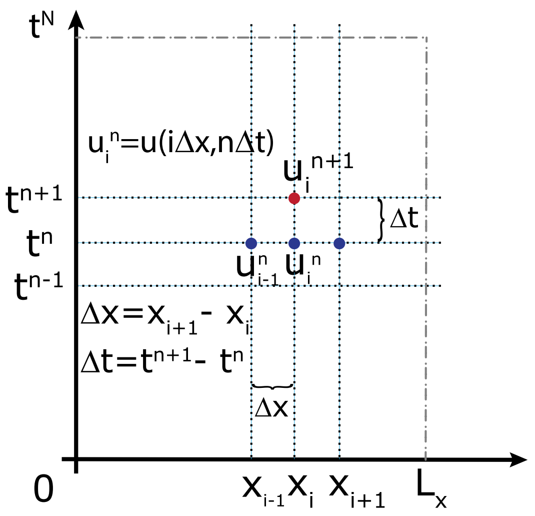

FIGURE 2. Discrete grid used for the function $u(x, t)$.

The cell size or space step is defined by $\Delta x = L_x/N = (b-a)/N$ where N is the total number of cells in the mesh, with a and b the left and right border and L_x the corresponding length. The coordinates of the grid points are then defined by $x_i = a + \mathrm{i}\Delta x$. The solution is computed at each time step Δt which corresponds to a series of discrete times $t^n = n\Delta t$, $(i, n) \in \mathbb{N}$.

$$u(x, t) \rightarrow u(x_i, t^n) \rightarrow u_i^n. \tag{2.3}$$

Here, u_i^n corresponds to the value of $u(x, t)$ at the position $x = a + \mathrm{i}\Delta x$ and at time $t = n\Delta t$. To simplify the notation, the superscript n is omitted when there is no confusion, e.g. u_i denotes u_i^n.

A finite difference scheme is typically obtained by approximating the derivatives in the partial differential equation using a Taylor expansion up to some given order. The minimal order is given by the order of the considered derivative. Generally, the order of a FD scheme, is linked to the rate at which the discretization converges to the exact solution when the step size Δx decreases. The error is proportional to $(\Delta x)^m$, where m is the exponent of the leading truncation error term. As the values of the unknown function are known only at the grid points, Taylor expansions at different grid points are linearly combined to eliminate all derivatives up to the needed order

$$\begin{aligned} u(x+\Delta x, t) &= u_{i+1}^n \\ &= u(x, t) + \Delta x \frac{\partial}{\partial x} u(x, t) + \frac{\Delta x^2}{2!} \frac{\partial^2}{\partial x^2} u(x, t) + \frac{\Delta x^3}{3!} \frac{\partial^3}{\partial x^3} u(x, t) \\ &\quad + \cdots + \frac{\Delta x^p}{p!} \frac{\partial^p}{\partial x^p} u(x, t), \end{aligned} \tag{2.4}$$

$$\begin{aligned} u(x-\Delta x, t) &= u_{i-1}^n \\ &= u(x, t) - \Delta x \frac{\partial}{\partial x} u(x, t) + \frac{\Delta x^2}{2!} \frac{\partial^2}{\partial x^2} u(x, t) - \frac{\Delta x^3}{3!} \frac{\partial^3}{\partial x^3} u(x, t) \\ &\quad + \cdots + \frac{(-\Delta x)^p}{p!} \frac{\partial^p}{\partial x^p} u(x, t). \end{aligned} \tag{2.5}$$

Using (2.4) and (2.5) the finite difference formulation of the first-order derivative can be expressed as

$$(2.4) \Rightarrow \frac{\partial}{\partial x}u(x,t) \simeq \frac{u_{i+1}^n - u_i^n}{\Delta x} + O(\Delta x), \tag{2.6}$$

$$(2.5) \Rightarrow \frac{\partial}{\partial x}u(x,t) \simeq \frac{u_i^n - u_{i-1}^n}{\Delta x} + O(\Delta x). \tag{2.7}$$

Equations (2.6) and (2.7) are called forward (FWD), and backward differences (BWD), respectively. These two equations are only first-order approximations (the order of a scheme is linked roughly with the associated error in the numerical representation) which are not sufficient in computational science. Using a combination of (2.4) and (2.5), second-order approximations can be obtained for the first-order derivative, using a central difference (CD) scheme (2.8), and for the second-order derivative, using again a central finite difference scheme (2.9).

$$(2.6) - (2.7) \Rightarrow \frac{\partial}{\partial x}u(x,t) \simeq \frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x} + O(\Delta x^2), \tag{2.8}$$

$$(2.6) + (2.7) \Rightarrow \frac{\partial^2}{\partial x^2}u(x,t) \simeq \frac{u_{i+1}^n + u_{i-1}^n - 2u_i^n}{\Delta x^2} + O(\Delta x^2). \tag{2.9}$$

The importance of the scheme order is illustrated in figure 3. FD have been used to compute $\mathrm{d}\exp(x)/\mathrm{d}x$ at $x = 0$ using first-, second-, third- and fourth-order approximations for different step sizes. As the step size decreases, the decrease of the associated error is faster for the higher-order schemes and the numerical solution obtained is more accurate for these schemes. It can be remarked that for very small step size, the error increases, this is not due to the error of the scheme order but due to the finite number of digits used for the representation of decimal numbers.

This representation is not unique, finite difference approximations can be deduced for any derivatives of u based on a given set of points using the method of undetermined coefficients. As an example, to construct a one-sided FD approximation of $\partial u/\partial x$ based on x_i, x_{i+1} and x_{i+2}, let $D_2u(x)$ be a second-order approximation of $\partial u/\partial x$,

$$D_2u(x) = au(x) + bu(x+\Delta x) + cu(x+2\Delta x). \tag{2.10}$$

Using Taylor expansion of $u(x+\Delta x)$ and $u(x+2\Delta x)$,

$$\begin{aligned} D_2u(x) &= au(x) + bu(x+\Delta x) + cu(x+2\Delta x) && (2.11)\\ &= (a+b+c)u(x) + (b+2c)\Delta x\partial_x u(x) + \tfrac{1}{2}(b+4c)\Delta x^2\partial_x^2 u(x) && (2.12)\\ &\quad + \tfrac{1}{6}(b+8c)\Delta x^3\partial_x^3 u(x) + \cdots && (2.13) \end{aligned}$$

This expansion represents $\partial_x u(x)$ if the following relations are satisfied

$$\begin{cases} a+b+c=0 \\ b+2c=+\dfrac{1}{\Delta x} \\ b+4c=0 \end{cases} \Rightarrow (a,b,c) = (3/2\Delta x, -2/\Delta x, 1/2\Delta x) \tag{2.14a}$$

$$\partial_x u_i = \frac{-3u_i + 4u_{i+1} - u_{i+2}}{2\Delta x}. \tag{2.14b}$$

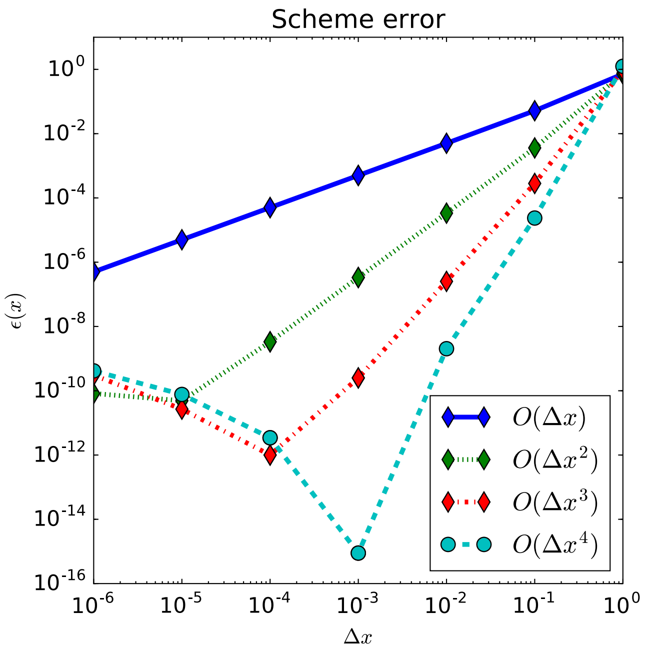

FIGURE 3. Relative error in the calculation of $\mathrm{d}\exp(x)/\mathrm{d}x$ as a function of the step size using different-order approximations.

The applications of the undetermined coefficients method are not limited to the determination of discrete expressions of derivatives, finding discretizations which have a fast convergence rate for simulations of fast flows (see Horton & Estes 1980) and obtention of an appropriate scheme for the Jacobian operator, which is the subject of § 6, are other possible applications. The basic principle of FD discretization can be used for spatial discretization but can also be employed to introduce resolution of the time advancing part of the equations.

3. Schemes for advancing in time

Let us now consider a time-dependent differential equation,

$$\partial_t u(t) = F(t, u(t)). \tag{3.1}$$

The idea behind any time scheme is based on a numerical quadrature of the associated time integral based on fractional steps (see Rosen 1967) for (3.1):

$$\begin{aligned} u(t^{n+1}) - u(t^n) &= \int_{t^n}^{t^{n+1}} \mathrm{d}t F(t, u(t)) \\ &\simeq \Delta t \left[\alpha_{-1} F(t^{n+1}, u^{n+1}) + \alpha_0 F(t^n, u^n) + \sum_{m=1}^{M} \alpha_m \delta u_m \right], \end{aligned} \tag{3.2}$$

with

$$\left.\begin{aligned}\delta u_0 &= F(t^n, u^n)\\ \delta u_1 &= F(t^n+\lambda_1\Delta t, u^n+\mu_{10}\delta u_0)\\ &\cdots\\ \delta u_m &= F\left(t^n+\lambda_m\Delta t, u^n+\sum_{p=0}^{M-1}\mu_{mp}\delta u_p\right),\end{aligned}\right\} \tag{3.3}$$

α_i corresponds to the weight of each expansion coefficient δu_i.

Time advance schemes can be classified depending on the value of α_{-1}. If α_{-1} is non-zero, the method is called implicit. At the opposite, if α_{-1} is zero, the method is called explicit. In an explicit method, all terms are evaluated at previous times. As an example, the explicit or forward Euler scheme (EE), which corresponds to $\alpha_0=1$ and $\alpha_{i\neq 0}=0$ is expressed by,

$$\left.\begin{aligned}\frac{u^{n+1}(x)-u^n(x)}{\Delta t} &= F(x, t^n, u^n(x))\\ u^{n+1}(x) &= u^n(x)+\Delta t F(x, t^n, u^n(x)).\end{aligned}\right\} \tag{3.4}$$

And the implicit version, called implicit or backward Euler (IE), which corresponds to $\alpha_{-1}=1$ and $\alpha_{i\neq -1}=0$, by

$$\left.\begin{aligned}\frac{u^{n+1}(x)-u^n(x)}{\Delta t} &= F(x, t^{n+1}, u^{n+1}(x))\\ u^{n+1}(x)-\Delta t F(x, t^{n+1}, u^{n+1}(x)) &= u^n(x).\end{aligned}\right\} \tag{3.5}$$

Moreover, if F is linear in u, then $F(x, t^{n+1}, u^n(x)) = F_{lin}(x, t^{n+1})u^n(x)$ and the previous relation can be rewritten as,

$$[1-\Delta t F_{lin}(x, t^{n+1})]u^{n+1}(x) = u^n(x). \tag{3.6}$$

Both schemes, implicit and explicit Euler, are only first-order accurate and as a consequence present the same limitations as any first-order scheme. Mainly, the error is proportional to the time step Δt and as a consequence, to increase the accuracy by a factor of 10, the time step must be divided by 10 and so the number of iterations is increased by the same amount. Combinations of explicit and implicit schemes can also be used, these schemes are called semi-implicit.

$$\left.\begin{aligned}\frac{u^{n+1}(x)-u^n(x)}{\Delta t} &= \theta F(x, t^{n+1}, u^{n+1}(x)) + (1-\theta)F(x, t^n, u^n(x))\\ u^{n+1}(x)-\theta\Delta t F(x, t^{n+1}, u^{n+1}(x)) &= u^n(x)+(1-\theta)\Delta t F(x, t^n, u^n(x))\\ [1-\theta\Delta t F_{lin}(x, t^{n+1})]u^{n+1}(x) &= [1+(1-\theta)\Delta t F(x, t^n)]u^n(x)\end{aligned}\right\} \tag{3.7}$$

with $0<\theta<1$. In the case $\theta=1/2$, the method is called Crank–Nicholson (CN) and compared to Euler schemes, this scheme is second-order accurate (the error is reduced quadratically when the step size is decreased).

Each scheme, explicit, implicit and semi-implicit has advantages and disadvantages. In general, implicit schemes are more stable than explicit schemes, where stability can be seen as the property of the solution to remain finite at each time step (this concept is detailed in § 4). However, implicit schemes are more difficult to implement, in particular for nonlinear operators. They also require considerably more calculations than explicit methods and they finally have a poor parallel scalability compared to purely explicit schemes.

The time scheme must be chosen wisely depending on the studied problem as it can affect the physical process described by the equations. To illustrate this, the three schemes presented before (EE, IE and CN) are used to solve the equation describing the one-dimensional (1-D) harmonic oscillator, assuming a unity mass in arbitrary units,

$$\partial_t^2 x(t) + \omega^2 x(t) = 0. \tag{3.8}$$

The analytic solution is given by,

$$\left.\begin{aligned} x(t) &= A_0 \cos(\omega t + \varphi) \\ v(t) &= -\frac{A_0}{\omega}\sin(\omega t + \varphi). \end{aligned}\right\} \tag{3.9}$$

Since no damping effects are considered, the total energy of the system, i.e. the sum of kinetic and potential energies, must be conserved during the motion. The total energy is expressed by,

$$\begin{aligned} E_{tot}(t) &= \frac{v^2(t)}{2} + \frac{\omega^2 x^2(t)}{2}, \\ &= \text{const.} \end{aligned} \tag{3.10}$$

Since the harmonic oscillator is a second-order equation in time, (3.8) should be split into a system of two equations of first order,

$$\left.\begin{aligned} \partial_t x(t) &= v(t) \\ \partial_t v(t) &= -\omega^2 x(t) \end{aligned}\right\} \tag{3.11}$$

and the associated linear systems are,

(i) Explicit Euler (EE)

$$\begin{pmatrix} x^{n+1} \\ v^{n+1} \end{pmatrix} = \begin{bmatrix} 1 & \Delta t \\ -\omega^2 \Delta t & 1 \end{bmatrix} \begin{pmatrix} x^n \\ v^n \end{pmatrix}. \tag{3.12}$$

(ii) Implicit Euler (IE)

$$\begin{bmatrix} 1 & -\Delta t \\ \Delta t \omega^2 & 1 \end{bmatrix} \begin{pmatrix} x^{n+1} \\ v^{n+1} \end{pmatrix} = \begin{pmatrix} x^n \\ v^n \end{pmatrix}. \tag{3.13}$$

(iii) Crank–Nicholson (CN)

$$\begin{bmatrix} 1 & -\Delta t/2 \\ \Delta t \omega^2/2 & 1 \end{bmatrix} \begin{pmatrix} x^{n+1} \\ v^{n+1} \end{pmatrix} = \begin{bmatrix} 1 & \Delta t/2 \\ -\omega^2 \Delta t/2 & 1 \end{bmatrix} \begin{pmatrix} x^n \\ v^n \end{pmatrix}. \tag{3.14}$$

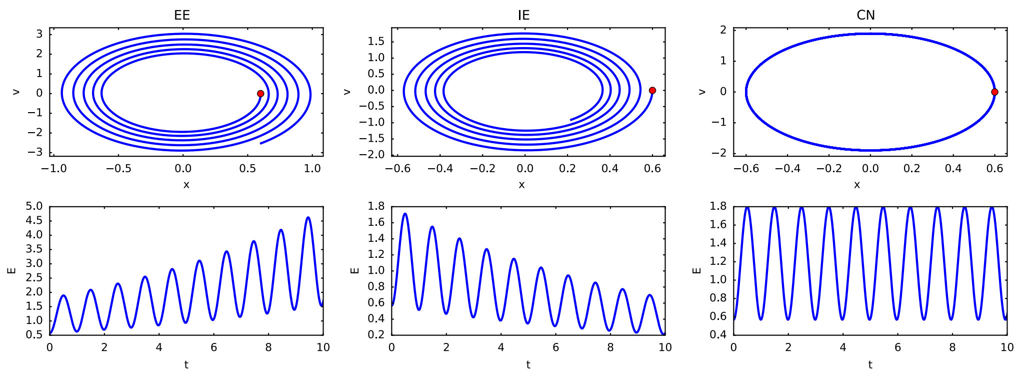

FIGURE 4. Oscillator motion in x–v diagram (phase space) and associated time behaviour of the total energy. The red point corresponds to the initial position velocity.

The numerical solutions for the motion of the harmonic oscillator obtained with the three different time schemes are presented in figure 4. The energy conservation property is violated by the explicit and the implicit schemes. Only the CN scheme verifies the energy conservation property of the system. The error in energy conservation of the three schemes can be derived easily starting from (3.10). Posing

$$E_{tot}^{(n)} = \frac{(v^n)^2}{2} + \omega^2 \frac{(x^n)^2}{2}, \tag{3.15}$$

the evolution of $E_{tot}^{(t)}$ from t to $t+1$ should be expressed as a function of $E_{tot}^{(t)}$

$$E_{tot}^{(n+1)} = \frac{(v^{n+1})^2}{2} + \omega^2 \frac{(x^{n+1})^2}{2}. \tag{3.16}$$

Starting from the three linear systems (3.12)–(3.14), and from the calculation of the quantities x and v at $t+\Delta t$, the following relations are derived:

(i) EE

$$\left.\begin{aligned} x^{n+1} &= x^n + \Delta t v^n \\ v^{n+1} &= v^n - \Delta t \omega^2 x^n \\ \Rightarrow E_{tot}^{(n+1)} &= \left(1 + \Delta t^2 \omega^2\right) E_{tot}^{(n)} > E_{tot}^{(n)}. \end{aligned}\right\} \tag{3.17}$$

(ii) IE

$$\left.\begin{aligned} x^{n+1} &= \frac{1}{1+\omega^2 \Delta t^2}(x^n + \Delta t v^n) \\ v^{n+1} &= \frac{1}{1+\omega^2 \Delta t^2}(v^n - \omega^2 \Delta t v^n) \\ \Rightarrow E_{tot}^{(n+1)} &= \frac{1}{(1+\Delta t^2 \omega^2)} E_{tot}^{(n)} < E_{tot}^{(n)}. \end{aligned}\right\} \tag{3.18}$$

(iii) CN

$$\left.\begin{aligned} x^{n+1} &= \frac{1}{1+\omega^2\Delta t^2/4}([1-\omega^2\Delta t^2/4]x^n+\Delta t v^n) \\ v^{n+1} &= \frac{1}{1+\omega^2\Delta t^2/4}([1-\omega^2\Delta t^2/4]v^n-\omega^2\Delta t v^n) \\ &\Rightarrow E_{tot}^{(n+1)} = E_{tot}^{(n)}. \end{aligned}\right\} \tag{3.19}$$

The violation of the energy conservation for IE and EE schemes appears via a second-order term in Δt. Analytical calculation shows that the observed energy growth is not linked to an inappropriate value of the time step but directly to the numerical properties of the scheme. The implicit scheme however introduces artificial damping from the truncations which leads to a decrease of the energy of the system. Advantages of this numerical dissipation is an increased stability compared to explicit schemes. The associated consequences are described in § 4. The properties of the described system allows to express it as an Hamiltonian operator:

$$\left.\begin{aligned} H(v,x) &= \frac{v^2}{2}+\frac{\omega^2}{2}x^2, \\ \frac{\partial v}{\partial t} &= \frac{\partial H}{\partial x} \\ \frac{\partial x}{\partial t} &= -\frac{\partial H}{\partial v}. \end{aligned}\right\} \tag{3.20}$$

The energy conservation observed with the CN scheme is related to a property of this integrator which is that to be a symplectic integrator, this class of integrators preserves the properties of the associated Hamiltonian (e.g. energy conservation or geometrical properties). In Hairer (2006), more details and possible schemes for such systems are presented.

3.1. *Typical time advance schemes used*

Euler schemes are generally not satisfying for the numerical resolution of PDEs. Even if their implementation is simple and fast, these schemes are only of first-order accuracy. Higher-order schemes have been developed (see Durran 2010) and can be divided in two families:

(i) schemes which use information at steps $t, t-\Delta t, \ldots$ to calculate the solution at time $t+\Delta t$. The main schemes used are Adams–Moulton (AM) and Adams–Bashforth (AB) schemes. AM are implicit linear multistep methods and AB is explicit. A variant of AB used in plasma physic is the time scheme developed by Karniadakis, Israeli & Orszag (1991). Compared to the original AB schemes, this one has a better accuracy and an increased stability region. The third-order explicit version of the scheme is given here

$$\frac{\gamma_0 u^{n+1} - \sum_{p=0}^{2}\alpha_p u^{n-p}}{\Delta t} = \sum_{p=0}^{2}\beta_p F(t^n - p\Delta t, u^{n-p}, \ldots) \tag{3.21}$$

$$\alpha_p = 3, -3/2, 1/3 \tag{3.22}$$

$$\beta_p = 3, -3, 1, \quad \gamma_0 = 11/6. \tag{3.23a,b}$$

In the case where one of the operators present in F corresponds to a diffusion operator with a coefficient ν ($\nu\nabla^2$), an implicit version exists. The implicit version consist mainly in an implicit treatment of the diffusion operator, the advantage of the implicit diffusion is an increased stability region for the scheme due to damping of associated high wavenumbers:

$$\frac{\hat{u} - \sum_{p=0}^{2} \alpha_p u^{n-p}}{\Delta t} = \sum_{p=0}^{2} \beta_p F(t^n - p\Delta t, u^{n-p}, \ldots) \tag{3.24}$$

$$\frac{\gamma_0 u^{n+1} - \hat{u}}{\Delta t} = \nu \nabla^2 u^{n+1} \tag{3.25}$$

$$\alpha_p = 3, -3/2, 1/3 \tag{3.26}$$

$$\beta_p = 3, -3, 1, \quad \gamma_0 = 11/6. \tag{3.27a,b}$$

(ii) schemes where the right-hand side term is approximated at intermediate times between t and $t + \Delta t$. This principle is used in the elaboration of the Runge–Kutta (RK) schemes. RK schemes are not unique at each order, the choice depends on the desired properties for the system (see Shu & Osher 1988; Butcher 2008). As computer performances have increased during the last years, fourth-order RK schemes are actually used widely,

$$u^{n+1} = u^n + \Delta t \sum_{p=0}^{4} \beta_p k_p \tag{3.28}$$

$$k_p = F\left(t^n + c_p \Delta t, u^n + \Delta t \sum_{q=0}^{4} \alpha_{nq} k_q\right). \tag{3.29}$$

4. Numerical modelling of diffusion

The diffusion equation with a constant diffusion coefficient D (also called the heat equation) is the simplest parabolic PDE. The time evolution of the positive definite field $u(x, t)$ is given by,

$$\partial_t u(x, t) = D \partial_x^2 u(x, t). \tag{4.1}$$

Using a second-order scheme in space, the right-hand side of (4.1) is discretized using a central expression,

$$D\partial_x^2 u(x, t) \simeq D \frac{u_{i+1} + u_{i-1} - 2u_i}{\Delta x^2}. \tag{4.2}$$

4.1. *Stability analysis and Courant–Friedrichs–Lewy (CFL) condition*

Numerical simulations have been realized with IE and EE schemes and spatial CD, respectively. The obtained profiles for $u(x, t)$ at different times are plotted in figure 5. Two runs with two different time steps are performed in both cases. Obviously, the implicit Euler scheme produces nearly identical results with both time steps whereas the results obtained with the explicit Euler scheme are strongly affected by the choice of the time step. Here, for the larger time step, the solution $u(x, t)$ shows an oscillating

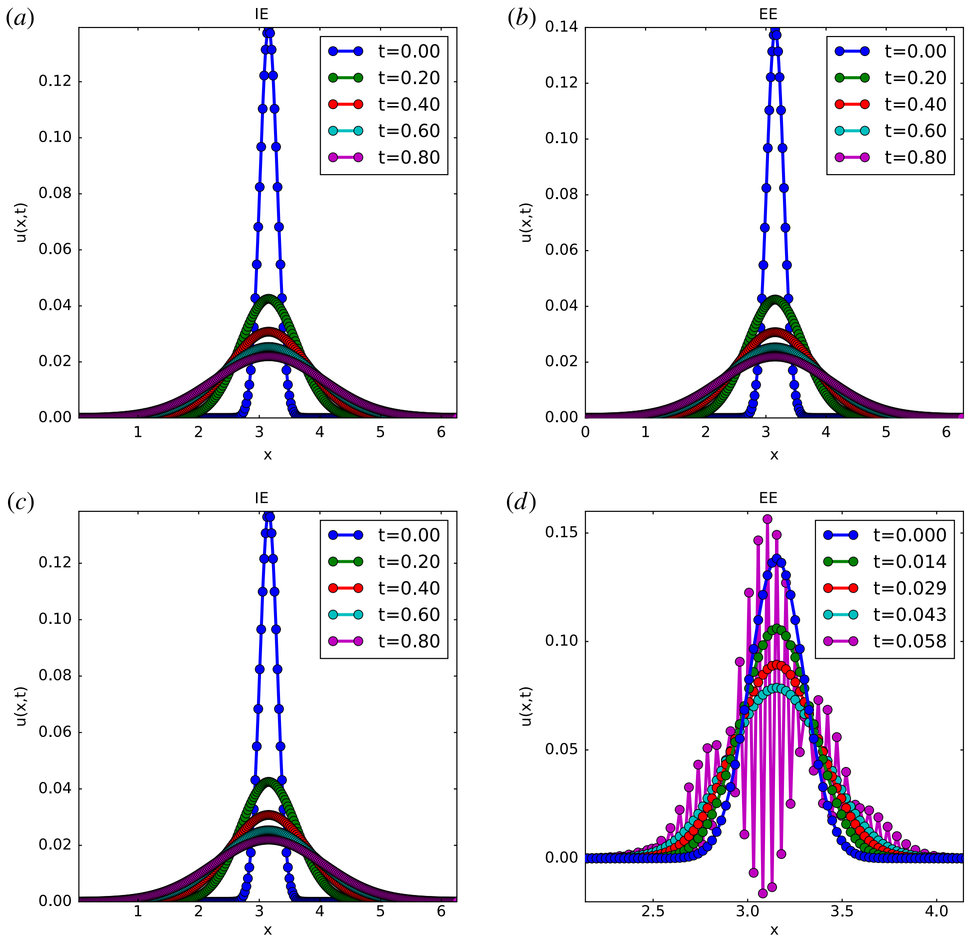

FIGURE 5. Solution obtained for the diffusion equation with $N_x = 256, L_x = 2\pi$. $\Delta t = 5e\text{–}4$ for cases (*a*) and (*b*) and $\Delta t = 8e\text{–}4$ for cases (*c*) and (*d*).

behaviour with an exponential growth. This behaviour can be explained through the corresponding stability analysis.

The stability property of a time advancing scheme can be studied through the Von Neumann stability analysis technique (see Durran 2010). Note that it is very important to realize that the stability of a scheme is different from its accuracy. To be useful, a numerical scheme must be both consistent and stable.

Consistency means that, the (continuous) solution of the PDE must satisfy the discretized version of the equation – using the respective time scheme – with approximation errors that vanish when Δx and Δt approach zero. Consistency guarantees that the scheme truly approximates the equation intended to be solved (and not something else).

Stability means that the scheme does not amplify errors. Obviously, this is very important, since truncation errors are impossible to avoid in any numerical calculation. In fact, even in the ideal case of infinite precision, some errors (e.g. discretization errors) remain. Clearly, if errors are amplified, they will rapidly dominate any computation (making it useless). In the simplest case of linear constant coefficient schemes, a complete stability analysis is possible as the numerical algorithm equations can be solved exactly by separation of variables. It follows that any solution of these equations can be written as a superposition of Fourier modes. Since these Fourier modes are uncoupled, the evolution of each mode can be considered individually. As an example, considering (4.1), the field $u(x, t)$ can be expressed as a sum of complex wave functions. For an infinite space, the initial condition can be written as,

$$u(x, t=0) = u_0(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \tilde{u}(k) \mathrm{e}^{\mathrm{i}kx}\, \mathrm{d}k. \tag{4.3}$$

Let us assume moreover, that the initial condition, for simplicity, can be expressed using only one Fourier mode,

$$u(x, t=0) \rightarrow u_j^0 = U_0 \mathrm{e}^{\mathrm{i}kj\Delta x}. \tag{4.4}$$

After a single application of the differencing scheme, the mode's amplitude and phase are modified by a factor $G(k)$,

$$\mathrm{e}^{\mathrm{i}kj\Delta x} \rightarrow G(k) \mathrm{e}^{\mathrm{i}kj\Delta x}, \tag{4.5}$$

and after n iterations, the field at time $(t_n = n\Delta t)$ has the form

$$u_j^n = G^n(k) u_j^0, \tag{4.6}$$

where $G(k)$ corresponds to the complex amplification factor of the scheme. A scheme is stable only if an initial error is not amplified, which corresponds to $|G(k)| \leqslant 1$. More precisely, if $|G(k)| = 1$, the scheme is classified as neutral and if $|G(k)| < 1$, the scheme is damping. From this analysis, it is found that the EE scheme has an amplification factor expressed by,

$$G(k) = 1 - 4\frac{D\Delta t}{\Delta x^2} \sin^2\left(\frac{k\Delta x}{2}\right). \tag{4.7}$$

From the above condition, the scheme is stable only if

$$\frac{D\Delta t}{\Delta x^2} \leqslant \frac{1}{2}. \tag{4.8}$$

In contrast, for the IE scheme, the amplification factor is

$$G(k) = \frac{1}{1 + 4\dfrac{D\Delta t}{\Delta x^2} \sin^2\left(\dfrac{k\Delta x}{2}\right)}. \tag{4.9}$$

Since, $|G(k)| \leqslant 1\, \forall \Delta t, \Delta x > 0$, this scheme is unconditionally stable.

The value of $|G(k)|$ for the explicit Euler is mainly governed by the number $n_c = D\Delta t/\Delta x^2$. For the results presented above, the first value of the time step corresponds to $n_c = 0.42$ and the second value to $n_c = 0.68$. As expected, the second case is unstable, $n_c = 0.68 > 1/2$. The number n_c is called in the literature the Courant number. The stability condition is associated with the so called Courant–Friedrichs–Lewy condition (see Courant, Friedrichs & Lewy 1928). However this is true mainly only for the single stage linear methods such as the Euler scheme.

In general, the CFL condition is necessary but not sufficient to ensure stability (see Schneider, Kolomenskiy & Deriaz 2013).

A physical interpretation of the unconditional stability of the implicit Euler scheme is linked to the fact that this method avoids growth of small scales through an artificial damping of these scales.

The treatment of diffusion is a challenge in plasma physics computations. The first issue with respect to the diffusion process, which is not detailed here, is the strong anisotropy of diffusion between the directions parallel and perpendicular to the magnetic field. Here, splitting between explicit and implicit operators should be implemented to treat correctly both directions with an acceptable time step (see Crouseilles, Kuhn & Latu 2015).

The constraint is not on the diffusion coefficient (which is typically quite small (see Freidberg 2007)) but the correct treatment of small scales. The presence of small scales requires a fine grid with small Δx. In a 1-D simulation, decreasing Δx e.g. by a factor of 4 for example, implies that Δt needs to be divided by 16. As a consequence, for the same simulation, the simulation time will be $16 \times 4 = 64$ times longer. Increasing grid resolution has an important impact on the simulation time, which is even worst in 3-D; an increased resolution by a factor 4 in each direction leads to a computational time $16 \times 4^3 = 1024$ times higher. A consequence of this is that simulations cannot be done anymore on simple workstations but must use parallel computers.

The simple diffusion equation is used as a starting point for an illustration of parallel implementations based on a purely serial code. Possible parallel implementations can be divided into two families depending on the hardware used for the simulations. Both implementation and how to develop them from a serial code are presented in the following sections.

4.2. *Numerical implementation*

4.2.1. *Performance measurement*

Before presentation of possible numerical implementations applied to the resolution of parabolic equations, it is important to define a way to characterize benefits associated with parallel implementations with respect to the serial one. Performance gain can be both modelled and measured. In this section we will take a another look at parallel computations by giving a simple analytical model that illustrates some of the factors that influence the performance of a parallel program. Consider a program consisting of three parts: an initialization section (time T_i), a computation section (time T_c) and a finalization section (time T_f). The total running time of this program on one processor (called the serial version) is then given as the sum of the times for the three parts:

$$T_s = T_i + T_c + T_f. \tag{4.10}$$

Now, how will this running time evolve when the program is run in parallel? Assuming that the program is run on P processors and that T_i and T_f are almost independent of the number of processors, the parallel time T_p ideally becomes, in particular if the influence of the communication cost between processors is neglected:

$$T_p(P) = T_i + T_c/P + T_f. \tag{4.11}$$

The performance gain is measured by the speed-up $S(P)$ which describes how much faster a problem runs. The actual running time does not appear and the speed-up $S(P)$ is normalized such that

$$S(P) = T_s/T_p(P). \tag{4.12}$$

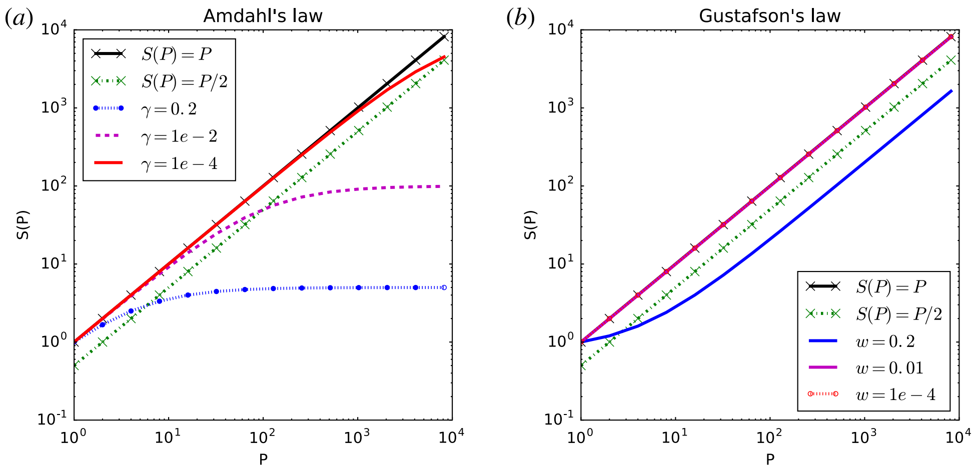

FIGURE 6. Speed-up as function of number of processors, using Amdahl's (*a*) and Gustafson's (*b*) laws.

Defining γ as,

$$\gamma=\frac{T_i+T_f}{T_s}, \tag{4.13}$$

$S(P)$ is simplified as,

$$S(P)=1/(\gamma+(1-\gamma)/P). \tag{4.14}$$

Equation (4.14) is well known as Amdahl's law (see Amdahl 1967) and says that, in absence of unparallelized parts ($\gamma=0$), speed-up scales as the number of processors used for the simulation. This case is defined as an ideal speed-up. However, as can be seen in figure 6, the serial part is a highly limiting factor in the scalability of a code. For example, with $\gamma=0.5$ (which means that 50 % of the code cannot be run in parallel), using 4 processors gives a speed-up of 1.6, far away from the $S=4$ desired. This constraint leads to a strong limitation on the parallel scalability of a code. As observed in figure 6, with $\gamma=1\,\%$, passing from 100 to 1000 processors gives only a negligible gain in running time. And as consequence, Amdahl's law appears as a strong limitation on performance increases of codes with an increasing number of processors used. This conclusion has been softened by Gustafson (see Gustafson 1988) who stated that if the serial part cannot be reduced, it is possible to solve a larger problem efficiently in the same amount of time. As the problem size grows, the work required for the parallel part of the problem frequently grows much faster compared to that required for the serial part, then as the problem size grows, the serial fraction decreases and the speed-up improves. In Gustafson's law, the speed-up follows the rule:

$$S(P)=w+(1-w)*P, \tag{4.15}$$

with w being the part which benefits from resources improvement (as illustrated in figure 6*b*). These performance measurements and associated conclusions will be used in the following section, illustrating 2 different parallel implementations to solve a 1-D diffusion equation.

L_x	70π
N_x	1024
D	0.1
Δt	$1e$–4
T_{max}	500
$u_0(x)$	$0.1\sin\left(\frac{\pi}{L_x}x\right)$
$S(x)$	$0.1\exp(-10*(x-0.3*L_x)^2)$

TABLE 1. Reference numerical set-up for the resolution of (4.16).

4.2.2. *Serial implementation*

Starting point is the resolution of the diffusion equation with a constant source term. For this purpose, a second-order central derivative for the Laplacian operator is used,

$$\left.\begin{aligned} \partial_t u(x,t) &= F(u(x,t)) \\ &= D\partial_x^2 u(x,t) + S(x) \\ \text{initial condition: } & u(x,0) = u_0(x), \\ \text{boundary conditions: } & \partial_x u(t,0) = 0, \quad u(t,L_x) = 0, \end{aligned}\right\} \tag{4.16}$$

where $D\partial_x^2 u(x,t)$ is discretized by

$$D\partial_x^2 u(x,t) \Rightarrow D\frac{u_{i+1}+u_{i-1}-2u_i}{\Delta x^2}. \tag{4.17}$$

The time evolution is resolved by a fourth-order Runge–Kutta scheme (RK-4),

$$\left.\begin{aligned} k_1 &= F(u^n) \\ k_2 &= F\left(u^n + \frac{\Delta t}{2}k_1\right) \\ k_3 &= F\left(u^n + \frac{\Delta t}{2}k_2\right) \\ k_4 &= F\left(u^n + \Delta t k_3\right) \\ u^{n+1} &= u^n + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4), \end{aligned}\right\} \tag{4.18}$$

to yield the discrete equation. For the simulations, the reference numerical set-up used in § 4.2 is indicated in table 1.

A serial implementation of a such model (serial implementation means a version of the numerical code which can be executed only on 1 core) is illustrated in figure 7. The field $u(x,t)$ is modelled through an array in which each cell contains the value of u at a given time t for all spatial grid points. In the case sketched in figure 7, the spatial grid contains 16 cells numbered from 0 to 15. The right-hand side of (4.16) is represented by a function F which is applied to every element of the array u using a RK-4 algorithm (4.18). RK-4 needs 4 intermediate arrays which are not represented on figure 7 for simplicity.

The corresponding implementation of the time loop is presented in listing 1.

Listing 1: RK4 time scheme for serial implementation of diffusion equation. Definitions of used functions are given in § C.1.

```
/*time loop using RK4 scheme*/
for(i=0;i<tmax;i+=1)
{
  calclaplacien(pr_P0,K1,dx*dx, D,fieldSize);              /* K1 = D*
      laplacian(P0) */
  addField(pr_fsrc,K1,fieldSize);                         /* K1 += S(x)
      */
  copyandaddmultField(pr_P0, K1, F1, dt/2, fieldSize);   /* F1 = P0 +
      0.5dt*K1 */
  Generate_BC(F1);

  calclaplacien(F1,K2,dx*dx,D,fieldSize);                  /* K2 = nabla
      (chi nabla)F1 */
  addField(pr_fsrc,K2,fieldSize);                         /* K2 += S(x)
      */
  copyandaddmultField(pr_P0, K2, F2, dt/2, fieldSize);   /* F2 = P0 +
      0.5dt*K2 */
  Generate_BC(F2);

  calclaplacien(F2,K3,dx*dx,D,fieldSize);                  /* K3 = nabla
      (chi nabla)F2 */
  addField(pr_fsrc,K3,fieldSize);                         /* K3 += S(x)
      */
  copyandaddmultField(pr_P0, K3, F3, dt, fieldSize);     /* F3 = P0 + dt
      *K3 */
  Generate_BC(F3);

  calclaplacien(F3,K4,dx*dx,D,fieldSize);                  /* K4 = nabla
      (chi nabla)F3 */
  addField(pr_fsrc,K4,fieldSize);                         /* K4 += S(x)
      */

  copyandaddField(K2,K3,pr_P1,fieldSize);                 /* P1 = K2+K3
      */
  multCnst(2.,pr_P1,fieldSize);                           /* P1 *= 2 */
  addField(K4,pr_P1,fieldSize);                           /* P1 += K4 */
  addField(K1,pr_P1,fieldSize);                           /* P1 += K1 */
  multCnst(dt/6.,pr_P1,fieldSize);                        /* P1 *= dt/6
      */
  addField(pr_P1,pr_P0,fieldSize);                        /* P0 = P0 + P1
         <=> P0 <- P0+dt/6*(K1+K2K+2K3+K4) */
  Generate_BC(pr_P0);
}
```

Figure 8 shows on the left the included source and on the right the initial and final profiles obtained at $t = T_{max}$.

4.2.3. *Message passing interface (MPI) implementation*

The previous code can be greatly accelerated not through low-level optimizations but by taking advantage of modern computers architecture. For that purpose, the development of a parallel version using an MPI library (see Message Passing Interface Forum 2015) is explained. MPI can be seen as a way to execute a software as a set of processes that have only local memory but are able to communicate with other processes by sending and receiving messages. MPI has been designed mainly to be used with a distributed collection of machines. An MPI program can be divided into 3 parts

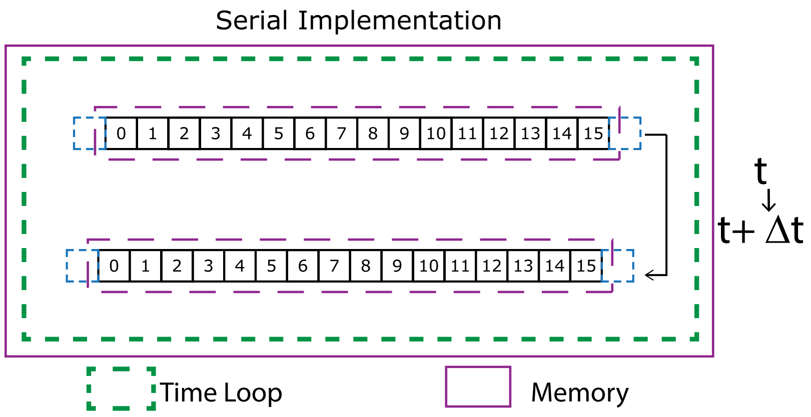

FIGURE 7. Data storage in memory in a serial implementation, the arrow represent the computation of u at $t + \Delta t$ in function of u at t.

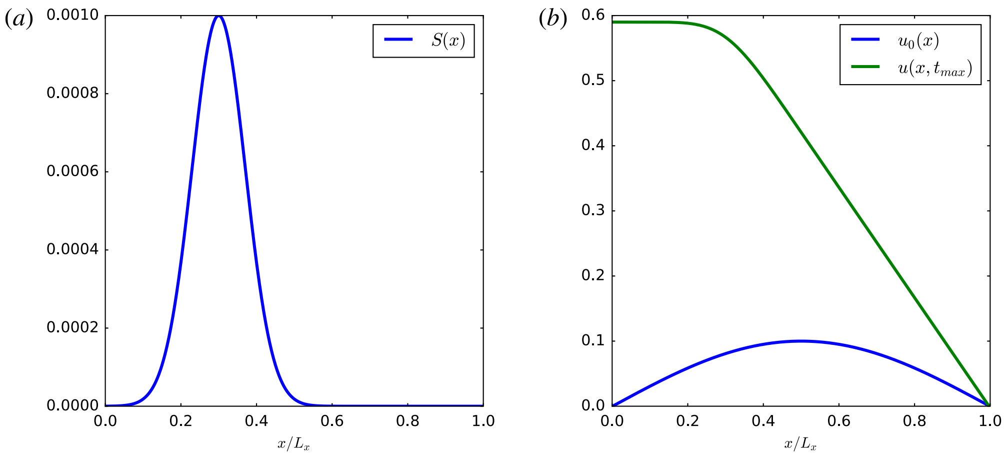

FIGURE 8. Source profile (a) and initial/final field for numerical simulation (b) of (4.16).

(i) initialization through the *MPI_Init* function;
(ii) program core;
(iii) termination of parallel part with the *MPI_Finalize* command.

Since the evaluation of u at time $t^{n+1} = t^n + \Delta t$ depends on its value at time t^n, the time loop cannot be parallelized. The only part which can be parallelized is the calculation of the Laplacian of u. For the following description of the method, the simulation is considered to be run on 4 processes. In the serial algorithm, all the data are stored in the memory of the execution process. This concept cannot be used in an MPI version: if only 1 process can access the data, the other processes are useless. At the opposite, sending all the information to all processes generates an unnecessary memory allocation when the number of CPU (central processing unit) increases. As consequence, for the MPI version, the spatial domain is divided between available processes. Each process can access only its associated memory and has access only to a part of the full domain as presented in figure 9. Since each process knows only a part of the grid, it is necessary to communicate values of 'local' boundaries between processes. These local boundaries, often called ghost points in the literature, correspond to artificial points added during the grid decomposition step. These points are used as buffer for received values from other process. Communications (dashed arrows) are made through calls to MPI functions *MPI_Send* and *MPI_IRecv* (see Message Passing Interface Forum 2015).

With MPI, communications between processes must be made explicitly in the code, indicating which process sends the data and which one should receive them. As a first part, and before the time loop, a 'map' of the processes is generated and each process can know which processes are his left and right neighbours. Generally, this part concerning the creation of the topology for the communications is made with the *MPI_Cart_Create* function but for the purpose of this paper, a simple implementation is made. A structure containing the necessary information is initialized, as presented in listing 2. Using the concept of ghost points previously described, this explains why, on line 34 of listing 2, 2 cells are added to the local size of the corresponding domain.

Listing 2: Initialisation of 1D MPI topology.

```
 1  /* structure containing topology info */
 2  typedef struct map
 3  {
 4      int leftProc;
 5      int rightProc;
 6      int actualProc;
 7      long localSize;
 8      long globalSize;
 9      long posGlobalMin;
10  } struc_map, *pstruc_map;
11
12  /* associated initialization */
13  /*compute mpi grid info with local arrray size */
14  struc_map MPI_Map_generate(const long globalsize)
15  {
16      struc_map mpimap;
17      int rank, ncores;
18      /* MPI function which indicates total number of available process
            */
19      MPI_Comm_size(MPI_COMM_WORLD, &ncores);
20      /* MPI function which indicates number of actual process */
21      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
22
23      mpimap.actualProc = rank;
24      mpimap.leftProc = mpimap.actualProc - 1;
25      mpimap.rightProc = mpimap.actualProc + 1;
26
27      if (mpimap.actualProc == ncores-1)
28      {
29          mpimap.rightProc = -1;
30      }
31
32      mpimap.globalSize = globalsize;
33      /* calculation of size of local domain from size of global domain
            */
34      mpimap.localSize = 2 + (globalsize-2)/ncores;
35      return mpimap;
36  }
```

Once the topology is initialized, the next step is to scatter the data between all processes. This is made with the *MPI_Scatter* function called inside the function

MPI_global2local (see § C.2 for the implementation). Compared to the serial version, each process needs updated values for the local boundaries coming from neighbours. As a consequence, the serial function *Generate_BC* is replaced by a function *MPI_Copy_BC* which manages the communications for each intermediate step of the RK-4 implementation. For process m, the goal is to send and receive at the same time values from processes $m-1$ and $m+1$. To avoid the problem of a deadlock in this function, (a deadlock is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does), non-blocking functions represented by the '_I' in function names are used. This allows each process to wait for data reception from a process and at the same time, send data to another process.

Listing 3: MPI time loop. Code for the used functions are given in § C.2.

```
/*generation of 1D topology */
struc_map mpimap = MPI_Map_generate(sp.Size);

/*data scattering */
MPI_global2local(g_fsrc,pr_fsrc,&mpimap);
MPI_global2local(g_P0,pr_P0,&mpimap);

/*time loop using RK4 scheme*/
/*using MPI implementation */
for(i=0;i<tmax;i+=t_out)
{
  calclaplacien(pr_P0, K1, dx2, mpimap.localSize);
  addField(pr_fsrc,K1,mpimap.localSize);
  copyandaddmultField(pr_P0, K1, F1, dt2, mpimap.localSize);
  MPI_copy_BC(F1,&mpimap);                          /* data transfer
      between process */

  calclaplacien(F1,K2, dx2,mpimap.localSize);
  addField(pr_fsrc,K2,mpimap.localSize);
  copyandaddmultField(pr_P0, K2, F2, dt2, mpimap.localSize);
  MPI_copy_BC(F2,&mpimap);

  calclaplacien(F2,K3, dx2, mpimap.localSize);
  addField(pr_fsrc,K3,mpimap.localSize);
  copyandaddmultField(pr_P0, K3, F3, 2.*dt2, mpimap.localSize);
  MPI_copy_BC(F3,&mpimap);

  calclaplacien(F3,K4, dx2,mpimap.localSize);
  addField(pr_fsrc,K4,mpimap.localSize);

  /* final addition P0+DT/6*(K1+2K2+2K3+K4) */
  copyandaddField(K2,K3,pr_P1,mpimap.localSize);
  multCnst(2.,pr_P1,mpimap.localSize);
  addField(K4,pr_P1,mpimap.localSize);
  addField(K1,pr_P1,mpimap.localSize);
  multCnst(dt6,pr_P1,mpimap.localSize);
  addField(pr_P1,pr_P0,mpimap.localSize);
  MPI_copy_BC(pr_P0,&mpimap);
}
```

With this implementation, the speed-up $S(P)$ is represented in figure 10. All simulations have been performed on a cluster node with 2 processors of 10 cores. On this cluster, speed-up tends to a limit value around 2.5–3 for both resolutions. The measured speed-up tends to reach the ideal one when the resolution increases. Higher resolution generates more operations at each time step, and, since in the presented algorithm the communication cost is independent of the

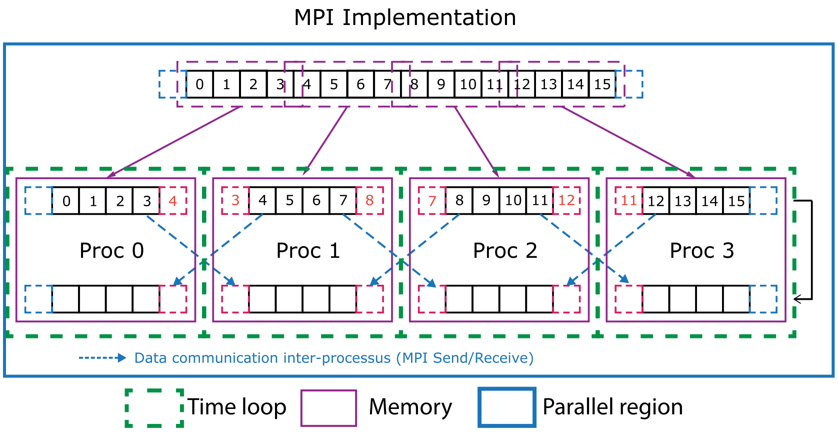

FIGURE 9. MPI implementation sketching computation from time t^n to time t^{n+1}.

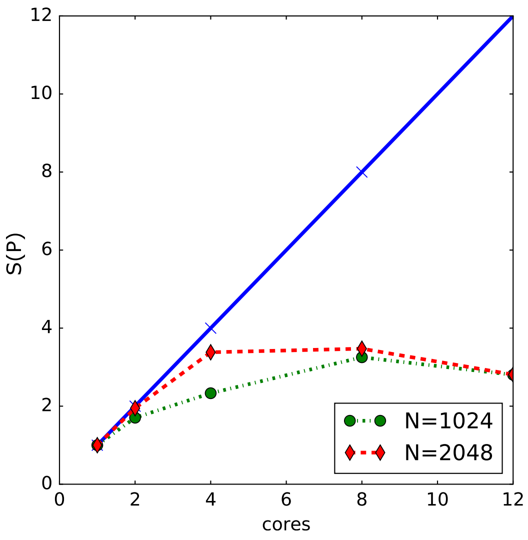

FIGURE 10. Speed-up $S(P)$ obtained using MPI implementation. The solid line corresponds to the ideal case ($S(P) = P$) and dashes lines to the measured values for resolutions of 1024 and 2048 points.

resolution, the obtained speed-up is higher: as expected in Gustafson's law, the ratio communication/computation decreases leading to better performances. In the simulation made to study physical models, the number of operators and considered resolutions are higher than the ones presented here, this leads to a better scaling with a simple domain decomposition method.

MPI was first defined for distributed memory architectures. However, modern clusters are a combination of distributed memory between nodes but each node can be assimilated as a shared memory system. Even if MPI can take advantage of a shared memory environment, another library, OpenMP, is used to present another possible implementation. The choice of OpenMP library is that this library is dedicated to the development of parallel programs on shared memory architectures.

4.2.4. *OpenMP implementation*

OpenMP (see Chapman, Jost & Van der Pas 2007) standard was formulated in 1997 as a set of library routines and tools for writing portable multiprocessing computer programs. Any computer program can be considered as a set of processes (or threads) and OpenMP permits for example to balance the total calculation between the different processes.

OpenMP provides a platform-independent set of compiler pragmas (a pragma is a language construct that specifies how a compiler should process the input source code), directives, function calls and environment variables that explicitly instruct the compiler how and where to use parallelism in the application. The method is well suited for domain (and/or loop) decomposition. The possible OpenMP implementations differ depending on the level of granularity considered in the algorithm. Granularity can be defined as the ratio between the amount of computations executed by a thread and the amount of communications between threads. Fine-grained parallelism means that individual tasks are relatively small in terms of code size and execution time. The data are frequently transferred between processors in amounts of one or a few memory words. The coarse-grained approach is the opposite: data are communicated infrequently, after larger amounts of computation. The finer the granularity, the larger is the potential for parallelism and hence speed-up, but the larger are also the overheads of synchronization and communication. As for MPI implementation, this last point is the critical issue for obtaining an effective speed-up.

The fine-graining technique uses OpenMP directives directly in loops to produce a parallel version of the code at compilation time. The following example shows the corresponding implementation for computation of the Laplacian operator.

Let $u(x, t)$ be discretized on N points in space, the extrema points with index $i = 0$ and respectively $i = N - 1$, are used as 'ghost' points implied by the boundary conditions. The numerical implementation is as follows,

C version:

```
Coef = D/(dx*dx);
for(i=1; i<N-1; i=i+1)
  laplacian[i] = Coef*(un[i+1]+un[i-1]-2*un[i]);
```

Fortran version:

```
Coef = D/(dx*dx);
do i=2,N-1
  laplacian(i) = Coef*(un(i+1)+un(i-1)-2*un(i))
end do
```

To obtain a parallel version of the loop, using OpenMP, only one directive has to be added in front of the loop.

C version:

```
Coef = D/(dx*dx);
#pragma omp parallel for
for(i=1; i<Nx-1; i= i+1)
  laplacian[i] = Coef*(un[i+1]+un[i-1]-2*un[i]);
```

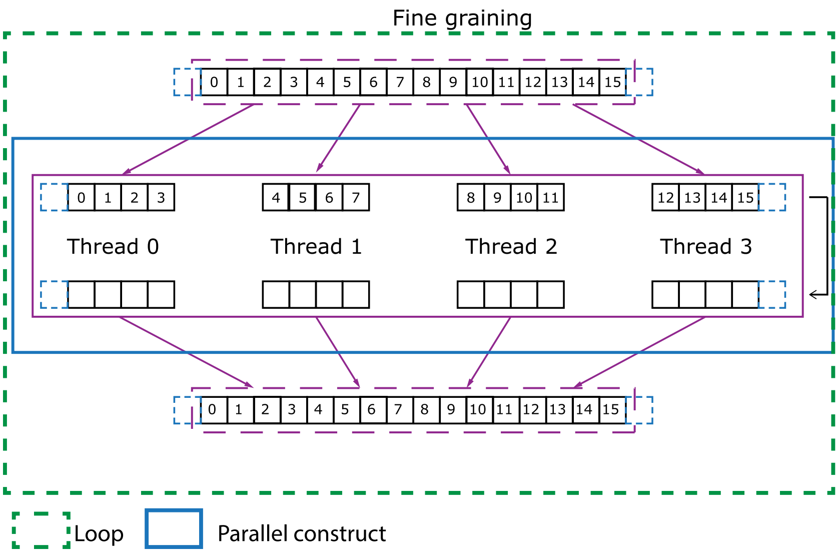

FIGURE 11. Fine-graining method. Arrows indicates scattering and gathering of data between cores through OpenMP directives (4 cores and 16 points are considered here).

Fortran version:

```
Coef = D/(dr*dr);
!$OMP PARALLEL DO
do i=2,Nx-1
  laplacian(i) = Coef*(un(i+1)+un(i-1)-2*un(i))
end do
!$ OMP END PARALLEL DO
```

Fine-graining decomposition is illustrated in figure 11 assuming 4 available threads for the simulation. Compared to the MPI implementation, this time the field is not distributed between processes, each core has access to the full field but loop iterations are divided between cores.

The dashed rectangle represents the do/for loop and the blue rectangle the parallel region. Since each threads runs in parallel, the calculation time is expected to be divided by 4 compared to the serial case. In figure 12, the speed-up $S(P)$ is plotted for an input array of $N = 8096$ elements. The speed-up is close to the theoretical one from 1 and 2 threads, but for more threads, there is a saturation close to a speed-up of 3. Our test case is rather simple and with increasing number of threads, the computational costs linked to the generation and destruction of threads, to the sharing of variables between them and to the associated communications are not negligible anymore. Therefore, this simple example shows that an efficient implementation of an OpenMP structure in a given code cannot be sketched to a simple addition of directives before loops. To be efficient, the directives should be made in the loops which have a large number of calculations per iteration.

OpenMP can also be implemented in a coarse-graining approach. This approach can be seen as a way to simulate an MPI implementation with OpenMP directives. Generally, for parallel numerical tools, domain decomposition is made using MPI techniques, and OpenMP fine-graining techniques are used as a refinement. However, as it has been shown in the previous section, the performances of the

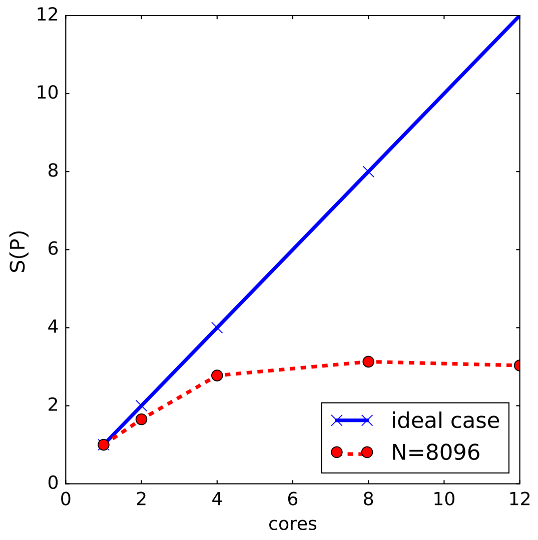

FIGURE 12. Measured speed-up for a fine-graining implementation of OpenMP directives.

fine-graining approach are strongly linked to the number of operations in each loop. Typically, for complicated models, the corresponding code will have many loops with few operations for each. This is due to the choices of the decomposition for each direction for example or the resolutions used in 3-D and the associated simulation time. For this kind of simulation, pure MPI implementation has generally better performance and scaling than an equivalent code mixing MPI domain decomposition and OpenMP fine graining. However, this kind of OpenMP implementation becomes more and more interesting since the generations of cluster nodes or even workstations have between 4 to 16 or even more cores per nodes. Compared to the fine graining where the parallelism is operated at a loop level, coarse graining is more intrusive.

In a coarse-graining (CG) approach, the domain decomposition is operated from the beginning and communications between processes are made explicitly. Understanding CG implementation with OpenMP directives requires a more precise description of the memory environment in an OpenMP process (see Chapman *et al.* 2007).

A program can be divided into smaller sequences of codes which can be managed independently, each one of this sequence is called a thread. OpenMP assumes that in a shared memory environment, all threads can access the same memory for storing and retrieving data. Each thread may have a private, temporary view of the memory (called a thread's view) at the beginning of an OpenMP construct (a parallel region) that it can use instead of the memory to store data during its execution when these data do not need to be seen by other threads.

Data can be 'transferred' between the memory and a thread's view, but can never move between temporary views directly, without going through the memory. Each variable used within a parallel region is either shared or private. Each shared variable reference inside a parallel region refers to the original variable of the same name. For each private variable, a reference to the variable name refers to a variable of the same type and size as the original variable, but private to the thread. It is therefore not accessible by other threads. The elaboration of a communication-like system in such a case is described in the following using again the example of the diffusion equation. As in previous sections, a decomposition on 4 threads with spatial discretization on

16 points is used. As in the MPI version, the parallelization starts in that case from the initialization: values for u_0, u_1, u_2, u_3 are stored on thread 0, values for u_4, u_5, u_6, u_7 are stored on thread 1, . . . and values for u_{12}, u_{13}, u_{14}, u_{15} are stored on thread 3. Since each thread contains its own version of the variable with the same name, we distinguish them using the thread number as a superscript. $A_0^{[0]}$ corresponds to the value of A_0 for the thread 0, $A_0^{[1]}$ corresponds to the value of A_0 for the thread 1 . . . . This concept is illustrated on figure 13.

The starting point is the creation of an equivalent to a 1-D topology. A similar structure as the one used in § 4.2.3 is declared and initialized. Each thread is identified with a unique number starting from 0 and increased by 1 for each thread. The structure itself is similar to the one used for MPI, the difference is the calling function giving unique identity of the thread. This identity is used to create the notion of neighbours for each thread, as in the MPI case.

Listing 4: OpenMP CG topology definition.

```c
typedef struct map
{
    int leftProc;
    int rightProc;
    int actualProc;
    long localSize;
    long globalSize;
    long posGlobalMin; /* offset for the thread private view */
} struc_map, *pstruc_map;

/*1D topology function definition */
/*input parameter globalsize correspond to the full length of the array
     */
struc_map OMP_Map_generate(const long globalsize)
{
    struc_map ompmap={0};
    ompmap.actualProc=omp_get_thread_num(); /* omp_get_thread_num give
        actual thread number */
    ompmap.leftProc = ompmap.actualProc-1;
    /* omp_get_num_threads give the total number of used threads */
    if (ompmap.actualProc==omp_get_num_threads()-1)
    {
        ompmap.rightProc=-1;
    }
    else
    {
        ompmap.rightProc = ompmap.actualProc+1;
    }
    ompmap.localSize=2+(globalsize-2)/omp_get_num_threads();
    if (omp_get_num_threads()>1)
        ompmap.posGlobalMin=1+ompmap.actualProc*(ompmap.localSize-2);
    else
        ompmap.posGlobalMin=0;
    ompmap.globalSize=globalsize;

    return ompmap;
}
```

Once the initialization is completed, data have to be scattered between threads. This part is illustrated in listing 4, only the case of the array corresponding to the source term is represented. For clarity, variable names starting with *g_* correspond to the original variables and names starting with *pr_* are associated with the thread private view.

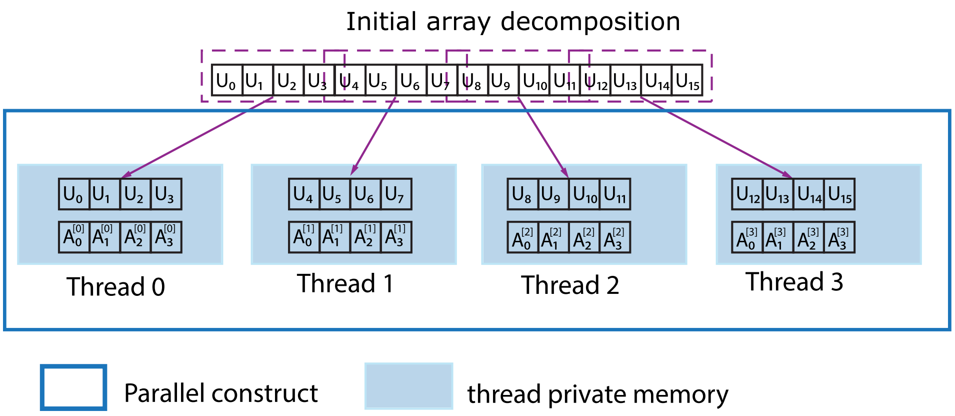
Initial array decomposition
U_0 U_1 U_2 U_3 U_4 U_5 U_6 U_7 U_8 U_9 U_{10} U_{11} U_{12} U_{13} U_{14} U_{15}
$A_0^{[0]}$ $A_1^{[0]}$ $A_2^{[0]}$ $A_3^{[0]}$
Thread 0
Thread 1
Thread 2
Thread 3
Parallel construct
thread private memory

FIGURE 13. Schematic representation of the coarse-graining communication process.

Listing 5: OpenMP CG scattering step (source code for the used functions are given in § C.3.)

```
int main(int argc, char **argc)
{
/*....*/
    double *g_fsrc = NULL;
    g_fsrc = (double *)malloc(MESHSIZE*sizeof(*g_fsrc));

    /*....*/

/* entry of parallel region, variable shared between global view and
    threads views are indicated with the shared keyword */
#pragma omp parallel default(none) private(i,j) shared(g_fsrc, g_P0,sp,
    sh_boundary,sh_lockvar)
    {
        struc_map ompmap = OMP_Map_generate(sp.Size);

        /* variable declared within a parallel region are private to
            this region */
        double *const pr_fsrc=malloc(ompmap.localSize*sizeof(*pr_fsrc))
            ;

        /*copy data from global memory to thread private memory*/
        OMP_global2local(g_fsrc,pr_fsrc,&ompmap);

    /* time loop, not shown in this listing */
     /*.... */

    } /* end of parallel region */
}
```

The most complex part is the simulation of a communication mechanism like send/receive using OpenMP directives. The previous mechanism based on the functions *OMP_global2local* and *OMP_local2global* (function defined in § C.3) cannot be used in the time loop. The associated communication cost is too important and as a consequence the scalability is poor and not efficient at all.

The developed technique is based on two other pragma directives defined in OpenMP (see OpenMP 2015):

(i) atomic: ensure that only one thread at a time accesses a shared variable, and subsequently synchronize updated contents with all other threads view,

(ii) barrier: synchronizes all threads, acts like a breakpoint that all threads must reach before execution of other instructions.

Since each thread knowns only a part of the grid, the values of the local boundaries have to be updated at each iteration due to the stencil used for spatial derivatives. For example, the Laplacian at grid node $i=4$ is based on values of u at position $i=3,4,5$. But, in our example, the value of u_3 is known only from thread 0 and the Laplacian of u_4 is computed by thread 1. As a consequence, in order to send data between threads, a mechanism based on Inputs/Outputs (I/O) on a shared array between all threads is implemented. This mechanism is illustrated in figure 14.

The principle is as follows. Each thread writes to a given position in the array based on its thread number (figure 14*b*). Then, the value is updated for all threads with the atomic directive. The shared array is now up to date and the updated values are synchronized between all threads (figure 14*c*).

As for the initialization, the developed implementation for the time loop is described in listing 6. It can be noticed that in the main kernel, the elaboration of a parallel version leads to similar modifications between the implementations of communication through a MPI library or through the OpenMP directives in a CG approach. All communications are carried out during the computation of the boundaries. In the CG case, the variable called *sh_boundary* represents the array used for data exchanges. This variable is updated in the function *OMP_generate_BC*. To be sure that all threads have updated the necessary cells, a *barrier* directive is incorporated just after. Finally, new values are copied to private variables within the function *OMP_copy_BC*.

Listing 6: OpenMP coarse graining time loop in a simplified version. Full code and code for the used functions are given in § C.3.

```

#ifdef __GNUC__
#pragma omp parallel default(none) private(i,j) shared(g_fsrc, g_P0,sp,
    sh_boundary,sh_lockvar)
#else
#pragma novector
#pragma omp parallel default(none) private(i,j) shared(dt,D,tmax,t_out,
    dt2,dt6, g_fsrc, g_P0,sp,sh_boundary,sh_lockvar)
#endif
    {
        struc_map ompmap = OMP_Map_generate(sp.Size);

        double *const K1=malloc(ompmap.localSize*sizeof(*K1));
        /*same thing for the other variables K2, K3, K4, F1, F2, F3,
            Ftmp, pr_P0, pr_P1, pr_fsrc*/

        double const dx2 = sp.dx*sp.dx;

        /*copy data from global memory to thread private memory*/
        OMP_global2local(g_fsrc,pr_fsrc,&ompmap);
        OMP_global2local(g_P0,pr_P0,&ompmap);

        /* B.C local and globals should be initialized correctly also
            */
        OMP_generate_BC(sh_boundary,pr_P0,&ompmap);
#pragma omp barrier

        OMP_copy_BC(sh_boundary,pr_P0,&ompmap);

        /*time loop using RK4 scheme*/
        /*using coarse graining decomposition */
```

```
          for(i=0;i<tmax;i+=1)
          {

                    /* computation of F1 = P0+dt/2*(D*nabla^2(P0) +S)
                    calclaplacien(pr_P0,K1, dx2, D, ompmap.localSize);
                    addField(pr_fsrc,K1,ompmap.localSize);
                    copyandaddmultField(pr_P0, K1, F1, dt2, ompmap.
                        localSize);

                    /*local update of shared array containing boundary
                        points*/
                    OMP_generate_BC(sh_boundary,F1,&ompmap);
#pragma omp barrier
                    /*new values for boundaries obtained from the shared
                        array */
                    OMP_copy_BC(sh_boundary,F1,&ompmap);

                    /* same principle for K2, K3 and K4... */


                    /* final computation of P1 from intermediate values K1,
                        K2, K3, K4 and P0 */
                    copyandaddField(K2,K3,pr_P1,ompmap.localSize);
                    multCnst(2.,pr_P1,ompmap.localSize);
                    addField(K4,pr_P1,ompmap.localSize);
                    addField(K1,pr_P1,ompmap.localSize);
                    multCnst(dt6,pr_P1,ompmap.localSize);
                    addField(pr_P1,pr_P0,ompmap.localSize);

                    /*update of the boundaries for the next time step */
                    OMP_generate_BC(sh_boundary,pr_P0,&ompmap);
#pragma omp barrier
                    OMP_copy_BC(sh_boundary,pr_P0,&ompmap);

          }

          /* memory unallocation for local arrays */

     } /*OMP block end*/
```

As a last step, each thread reads the needed values in the shared array. The speed-up with this method is plotted on figure 15. Compared to the previous approach, the speed-up obtained is almost optimal, and even superlinear in the present case (speed-up > 1). The reason of the increased speed-up is linked to a reduction of the number of communications and a reduction of the number of shared variables to update each time: in the fine-graining approach, shared variables must be created/destroyed/updated for all threads in each loop where OpenMP directives are used. However in the coarse-graining approach, the shared variables are created/destroyed only one time and the number of variables to share is highly decreased. Another difference is linked to the reduced array size used by each thread: smaller arrays lead to less memory used, allowing optimized and reduced memory transfer between the central memory and the physical processor cache due to the physical memory architecture of modern CPUs. The direct consequence associated with this implementation is an increased efficiency compared to the serial one which has been used as the reference case.

5. Numerical modelling of advection

The other family of operators one frequently deals with in plasma physics are hyperbolic operators. The simplest hyperbolic operator corresponds to an advection

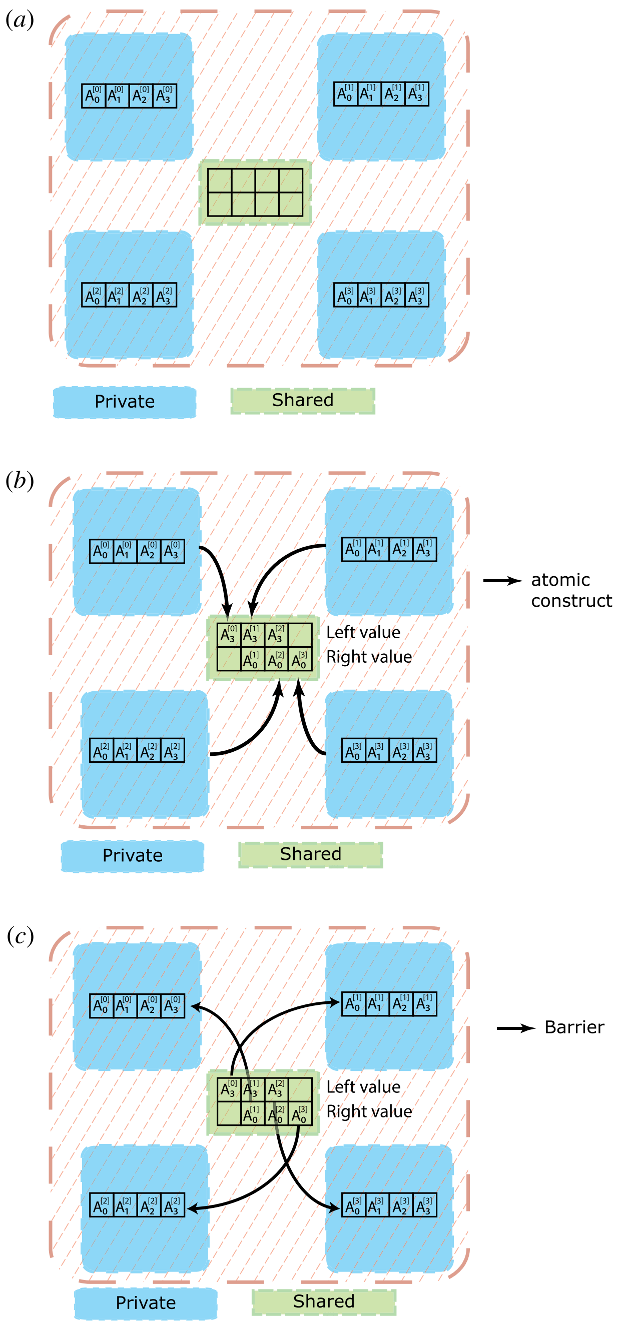
(a)
$A_0^{[0]}$ $A_1^{[0]}$ $A_2^{[0]}$ $A_3^{[0]}$
$A_0^{[1]}$ $A_1^{[1]}$ $A_2^{[1]}$ $A_3^{[1]}$
$A_0^{[2]}$ $A_1^{[2]}$ $A_2^{[2]}$ $A_3^{[2]}$
$A_0^{[3]}$ $A_1^{[3]}$ $A_2^{[3]}$ $A_3^{[3]}$
Private
Shared
(b)
$A_3^{[0]}$ $A_3^{[1]}$ $A_3^{[2]}$
$A_0^{[1]}$ $A_0^{[2]}$ $A_0^{[3]}$
Left value
Right value
atomic construct
Private
Shared
(c)
Left value
Right value
Barrier
Private
Shared

FIGURE 14. Coarse-graining method.

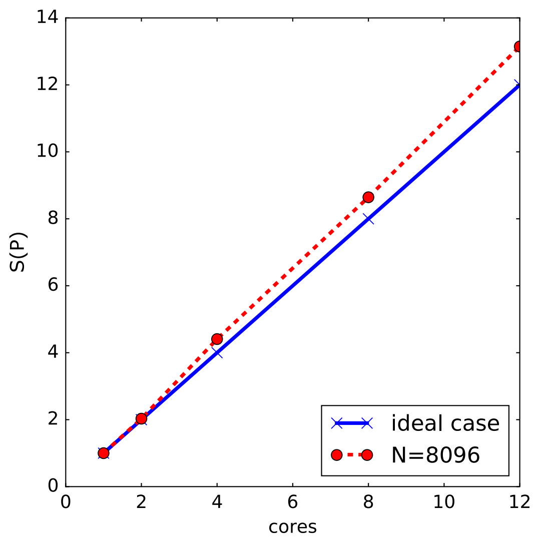

FIGURE 15. Speed-up ($S(P)$) for a coarse-graining implementation of OpenMP directives. Solid line correspond to the ideal case and dashed line to measured values.

L_x	51.2
N_x	512
V	6
Δt	$1e$–2
T_{max}	5
$u_0(x)$	$A_0 \exp\left(-\frac{(x-L_x/2)}{\sigma^2}\right)$
σ	0.5 or 1.5

TABLE 2. Reference numerical set-up for the resolution of (5.1).

process with a constant velocity specified by

$$\left.\begin{aligned} \partial_t u(x, t) + V\partial_x u(x, t) = 0 \\ u(x, 0) = u_0(x). \end{aligned}\right\} \tag{5.1}$$

The solution has the form,

$$u(x, t) = u_0(x - V * t). \tag{5.2}$$

Moreover, to avoid problems linked to boundaries, e.g. wave reflection, a periodic box is considered in the x direction. The reference numerical set-up used for the simulations in this section is given in table 2.

The same discretizations in space and time, as for the modelling of diffusion in the last section, are used here: CD in space and EE or IE in time, resulting in the following discrete equations,

$$\text{EE:}\ \frac{u_i^{n+1} - u_i^n}{\Delta t} = -V\frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x}, \tag{5.3}$$

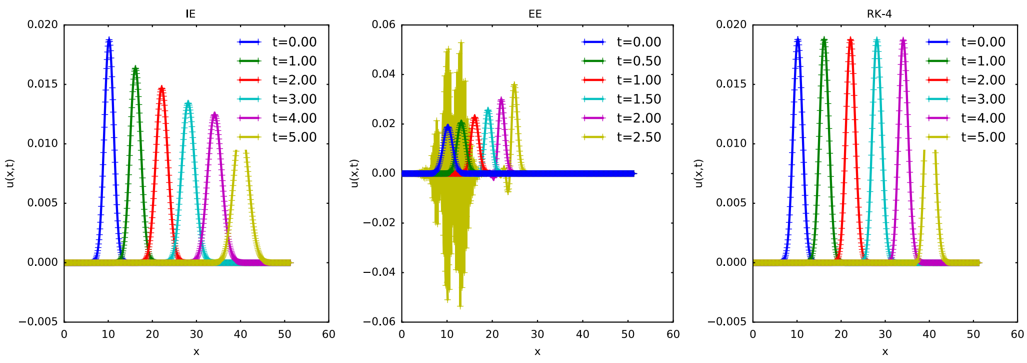

FIGURE 16. Numerical resolution of advection equation using CD in space for $\sigma = 1.5$ with parameters: $N_x = 512$, $L_x = 51.2$ and $\Delta t = 1e\text{–}2$.

$$\text{IE:}\quad \frac{u_i^{n+1} - u_i^n}{\Delta t} = -V\frac{u_{i+1}^{n+1} - u_{i-1}^{n+1}}{2\Delta x}. \tag{5.4}$$

The results from the simulation of advection using EE and IE algorithms are plotted on figure 16. The IE scheme reproduces almost correctly the advection process, except that the Gaussian function is not just advected but also damped. In the case presented here, the IE+CD scheme is stable but, in contrast to the resolution of the diffusion equation, this scheme is not unconditionally stable. The stability is linked to the values of V, Δx and Δt which must verify the inequality:

$$0 \leqslant V\frac{\Delta t}{\Delta x} \leqslant 1. \tag{5.5}$$

The number $\mu = V\Delta t/\Delta x$ corresponds to the Courant number for advection. It has to be noticed that full agreement between the CFL condition and stability analysis is quite rare. As for the diffusion, the CFL condition is necessary but not sufficient to ensure stability (see Ferszinger 2002; Schneider *et al.* 2013). Details on the derivation of the CFL condition for the advection equation can be found for example in Leveque (1992), Ferszinger (2002). However, the EE is unstable even if simulations are performed with a value of $\mu < 0.5$. This unstable evolution is characterized here by an increase of the amplitude of the field (which is inconsistent with a pure advection process) and also, the growth of spurious oscillations at the queue of the profile can be noticed. These oscillations dominate in amplitude and overcome the expected profile. Unlike for diffusion, EE+CD is unconditionally unstable which means that this scheme is unstable for any values of the parameters. This is verified by the same stability analysis using Von Neumann's technique for the derivation of the amplification factor G,

$$G(k) = 1 - \mathrm{i}V\frac{\Delta t}{\Delta x}\sin(k\Delta x) \tag{5.6}$$

$$|G(k)|^2 = 1 + V^2\frac{\Delta t^2}{\Delta x^2}\sin^2(k\Delta x) \geqslant 1. \tag{5.7}$$

The source of the numerical instability is not the order of the spatial differentiation but the combination of spatial and temporal schemes used. Making the assumption

Time scheme	$F(q)$
EE	$F_{EE}(q) = 1 + q$
IE	$F_{IE}(q) = 1/(1-q)$
CN	$F_{CN}(q) = (1 + q/2)/(1 - q/2)$
RK-2	$F_{RK2}(q) = 1 + q + q^2/2$
RK-3	$F_{RK3}(q) = 1 + q + q^2/2 + q^3/6$
RK-4	$F_{RK4}(q) = 1 + q + q^2/2 + q^3/6 + q^4/24$

TABLE 3. Analytic expression for function $F(q)$.

that the finite space derivative is computed with a very-high-order method, e.g. the spectral method, (5.2) becomes

$$\partial_t u(x, t) + \mathrm{i}k_x V u(x, t) = 0, \tag{5.8}$$

and the amplification factor $G(k)$ becomes,

$$G(k) = 1 + \mathrm{i}V\Delta t \tag{5.9}$$

$$|G(k)|^2 = 1 + V^2\Delta t^2 k_x^2 \geqslant 1. \tag{5.10}$$

With an amplification factor always larger than 1 even using a spectral method, the EE scheme cannot be implemented correctly with any CD scheme in space. Time schemes other than Euler methods, such as e.g. Runge–Kutta methods are multistep and so Von Neumann's stability analysis is not well suited. A stability analysis can be made using the notion of absolute stability (see Butcher 2008; Griffiths 2010). Starting from a PDE of the form $\partial_t u(x, t) = \lambda u(x, t)$ with $\lambda \in \mathbb{C}$, λ represents any possible eigenvalue of the associated discrete system, and assuming that the solution has the form

$$u^{n+1} = F(\lambda)u^n, \tag{5.11}$$

the scheme is stable if and only if $|F(\lambda)| \leqslant 1$. As example, considering the EE scheme, F is derived in 2 steps,

$$\partial_t u(t) = \lambda u(t) \tag{5.12}$$

$$\Rightarrow u^{n+1} = (1 + \lambda\Delta t)u^n \tag{5.13}$$

$$\leftrightarrow F(\lambda) = (1 + \lambda\Delta t). \tag{5.14}$$

For the other time schemes presented here, posing $q = \lambda\Delta t$, the function $F(q)$ is given in table 3.

Corresponding stability diagrams of these time schemes are presented in figure 19. From these diagrams, the reasons why RK schemes are so widely used appear more clearly. Compared to Euler schemes, RK algorithms are of higher order and also their stability region increases with the scheme order (e.g. the stability region is wider than the one associated with the CFL condition). The width of the associated stability region is also a huge benefit compared with typical multistep methods for which the stable region shrinks with the scheme order (see Butcher 2008). In the simple case of advection at constant velocity and considering first- or second-order discretizations in space, $F(\lambda)$ values can be calculated literally. With a space discretization on M

mesh points let us start from the general expression of a first- or second-order space derivative,

$$\partial_x u_i = (a u_{i-1} + b u_i + c u_{i+1})/\Delta x. \tag{5.15}$$

Inserting (5.15) in (5.1) and using EE for the time discretization, a linear algebraic system is obtained,

$$U^{n+1} = \left(\boldsymbol{I} - V\frac{\Delta t}{\Delta x}\boldsymbol{M_v}\right) U^n \tag{5.16}$$

with $\boldsymbol{I}$ representing the identity matrix,

$$U^{n+1} = \begin{pmatrix} u_0^{n+1} \\ u_1^{n+1} \\ u_2^{n+1} \\ \vdots \\ u_{M-1}^{n+1} \end{pmatrix}, \quad U^n = \begin{pmatrix} u_0^{n} \\ u_1^{n} \\ u_2^{n} \\ \vdots \\ u_{M-1}^{n} \end{pmatrix}, \tag{5.17a,b}$$

and

$$\boldsymbol{M_v} = \begin{bmatrix} b & c & 0 & & & \cdots & & 0 & a \\ a & b & c & 0 & 0 & \cdots & & 0 & 0 \\ 0 & a & b & c & 0 & \cdots & & 0 & 0 \\ \vdots & 0 & \ddots & \ddots & \ddots & \ddots & & 0 & 0 \\ 0 & 0 & & & & 0 & a & b & c \\ c & 0 & & & & 0 & 0 & a & b \end{bmatrix}. \tag{5.18}$$

The matrix $\boldsymbol{M_v}$ belongs to the class of circulant Toeplitz matrices. A matrix is said to be Toeplitz if it is a diagonal constant matrix, which means that all the elements along a diagonal have the same constant value. Eigenvalues (and eigenfunctions) of such kinds of matrices can be easily calculated analytically (see Smith 1978) and in the tridiagonal case, the eigenvalues of $\boldsymbol{M_v}$ are expressed by (demonstration in appendix B, (B 10)):

$$\left.\begin{aligned} \lambda_s = b + c\exp(-\mathrm{i}ks) + a\exp(\mathrm{i}ks) \\ k = 2\pi/M \\ s \in [0, \ldots, M-1]. \end{aligned}\right\} \tag{5.19}$$

The system (5.16) becomes,

$$\begin{aligned} U^{n+1} &= \left(1 - V\frac{\Delta t}{\Delta x}\lambda_s\right) U^n \\ &= F_{EE}(\lambda_s) U^n. \end{aligned} \tag{5.20}$$

In the case of CD discretization,

$$\left.\begin{aligned} a = -1/2, \quad b = 0, \quad c = 1/2 \\ \rightarrow \lambda_s(k) = \mathrm{i}\sin(sk), \end{aligned}\right\} \tag{5.21}$$

inserting into (5.20),

$$\left.\begin{aligned} F_{EE}(\lambda_s) &= 1 - \mathrm{i}\frac{V\Delta t}{\Delta x}\sin(sk) \\ |F_{EE}(\lambda_s)|^2 &= 1 + \frac{V^2\Delta t^2}{\Delta x^2}\sin^2(sk), \end{aligned}\right\} \tag{5.22}$$

which confirms the unstable characteristic of this scheme in that case, since $|F_{EE}(\lambda_s)| \geqslant 1$. More precisely, for any symmetric discretization, λ_s (and q) is purely imaginary, so in any case, $|F(q)| \geqslant 1$. As illustrated in figure 20, EE and RK-2 schemes are unconditionally unstable for advection using central spatial derivatives.

Since the CD scheme is unconditionally unstable, the next step is to look at other possible spatial discretizations, FWD and BWD discretizations. BWD leads to the following values for a, b and c,

$$\left.\begin{aligned} &a = -1, \quad b = 1, \quad c = 0 \\ &\Rightarrow \lambda_s = 1 - \exp(\mathrm{i}ks) \\ &F_{EE}(\lambda_s) = 1 - \frac{V\Delta t}{\Delta x}(1 - \exp(\mathrm{i}ks)) \\ &|F_{EE}(\lambda_s)| \leqslant 1 \leftrightarrow 0 \leqslant V\Delta t/\Delta x \leqslant 1. \end{aligned}\right\} \tag{5.23}$$

The BWD scheme is stable if $0 \leqslant V\Delta t/\Delta x \leqslant 1$. It has to be noticed that, the coefficient V which represents the advection velocity can be positive or negative, depending on the direction of the flow. As a consequence, if the direction is from right to left, V is negative and therefore BWD becomes unstable.

Assuming $V < 0$, the last possibility is not to use BWD but FWD scheme. The associated coefficients are,

$$\left.\begin{aligned} &a = 0, \quad b = -1, \quad c = 1 \\ &\Rightarrow \lambda_s = -1 + \exp(-\mathrm{i}ks) \\ &F_{EE}(\lambda_s) = 1 - \frac{V\Delta t}{\Delta x}(-1 + \exp(-\mathrm{i}ks)) \\ &|F_{EE}(\lambda_s)| \leqslant 1 \leftrightarrow -1 \leqslant V\Delta t/\Delta x \leqslant 0. \end{aligned}\right\} \tag{5.24}$$

The stability condition is 'symmetric' to the one for $V > 0$ in the BWD case, i.e. for $V < 0$: $-1 \leqslant V\Delta t/\Delta x \leqslant 0$. This class of methods, for which spatial discretization is skewed in the direction from which the flow comes from is called 'upwind' method. The origin of upwind methods can be traced back to the initial work of (see Courant, Isaacson & Rees 1952). The link between stability and the direction of the flow is illustrated in figure 17. When the flow comes from the left ($V > 0$), only BWD are stable. At the opposite, when the flow came from the right ($V < 0$), only FWD are stable, as expected from the stability analysis.

The last property which can be noticed for advection process simulations is that, even if the scheme is stable, the solution is damped. This dissipation phenomenon observed for the EE scheme is directly linked to the truncation error of the method. In fact, the equation solved numerically is, assuming FD in space (see Leveque 1992),

$$\partial_t u(x, t) + V\partial_x u(x, t) - \frac{V}{2}(\Delta x - V\Delta t)\partial_x^2 u(x, t) = 0. \tag{5.25}$$

The dissipative damping generated is directly linked to the discretization steps.

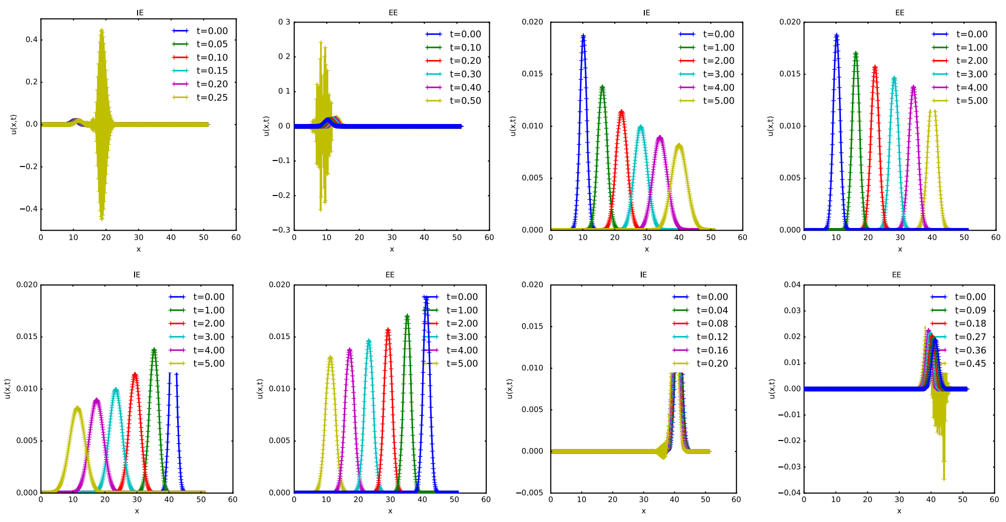

FIGURE 17. Influence on flow direction ($V=-6$ on top and $V=6$ on bottom) on implicit and explicit time schemes (IE and EE) with, forward/backward spatial derivative.

In § 4, the advantage of implicit schemes has been demonstrated for the diffusion operator. In particular, time steps can be larger than the one allowed by the CFL condition. However, in hyperbolic systems, implicit methods require more computational work per time step compared to explicit methods. This is linked to the mathematical properties of the associated matrix. Moreover, due to truncation errors, the solutions obtained with larger time steps are less accurate (see Jardin 2011). The presented schemes are only first order in space, since the unstable character of the CD derivative has been demonstrated. It can be interesting to develop a scheme which is stable and second order in space. This method can be derived using analysis of the truncation error, in the case where CD derivatives are used. The equation which is resolved in fact is the following one,

$$\partial_t u(x,t) + V\partial_x u(x,t) = -D\partial_x^2 u(x,t) \tag{5.26}$$

$$D = \frac{V^2}{2}\Delta t. \tag{5.27}$$

As a consequence, if CD derivatives are implemented, adding an artificial diffusion with a coefficient $D' = (V^2/2)\Delta t$ leads to a cancellation of this anormal diffusion and the scheme becomes stable. This procedure is known as the Lax–Wendroff (LW) differential scheme and permits the resolution of advection with second-order derivatives in space (see Lax & Wendroff 1960) and increased stability in time to order 3. The interest of LW is confirmed in figure 18 where (5.2) is resolved successfully using CD in space. Implementation of CD in space allows to use the same discretization independently of the flow direction. Since the amplitude remains constant in time, the LW scheme, as with RK-4, is non-dissipative, opposite to the Euler schemes.

The non-dissipative property of this scheme is really interesting to obtain an accurate solution. This property is not the only one necessary to obtain a realistic solution, it is also necessary to check if the scheme is dispersive or not. The dispersion

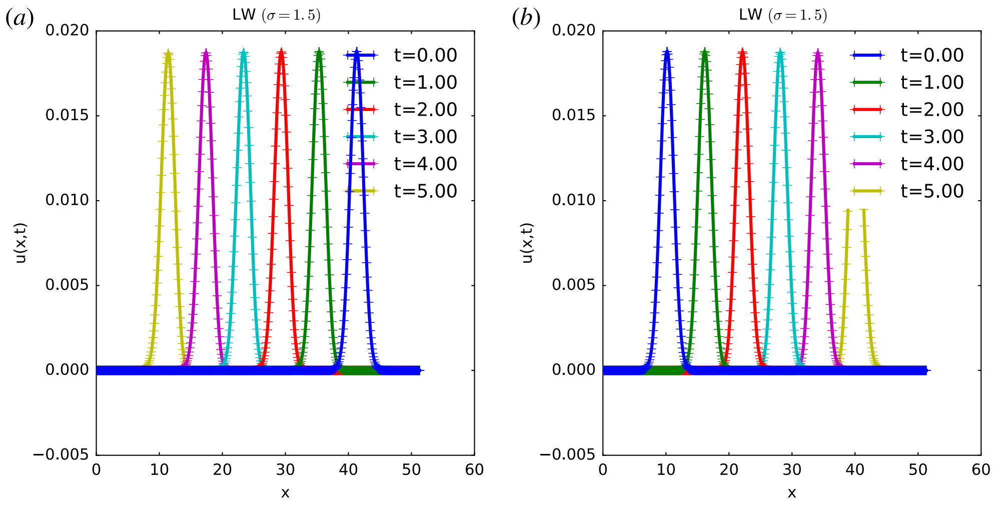

FIGURE 18. Advection using LW scheme with $V = -6$ (*a*) and $V = 6$ (*b*).

error can be seen as the difference between the physical propagation velocity and the numerical propagation velocity. The consequence is that a scheme suffering from dispersion leads to appearance of oscillations in the numerical solution. Both errors, dissipation and dispersion, can be analysed using the amplification factor derived from the stability analysis. The main assumption for the stability (and the accuracy) of a scheme concerns the modulus of the factor G. This modulus gives the diffusion error of the scheme. If $|G| = 1$, the amplitude of the solution is not damped in time due to artificial diffusion. If $|G| \leqslant 1$, the solution is damped but, at the same time, waves can develop at small scales due to numerical errors. Developing schemes with a better control of this damping leads to the existence of more accurate schemes which suppress the oscillations. As mentioned before, an example for the advection equation, the Lax method (see Leveque 2002) adds an artificial diffusion, with a diffusion coefficient derived from the truncation error of the scheme.

Since G is complex, additional information can be obtained using the expression of G in polar form, $G = a + \mathrm{i}b = |G|\mathrm{e}^{\mathrm{i}\theta}$. The argument θ characterizes the phase speed error of the system (see Durran 1999; Hirsch 2007). The dispersion error (ϵ_ϕ) is expressed by the ratio between the physical phase speed of the considered equation and the numerical phase speed, θ. If $\epsilon_\phi > 1$, the numerical propagation is larger than the exact one and the computed solution appears to move faster than the real one. If $\epsilon_\phi < 1$, the numerical solution travels at a lower velocity than the physical one.

The numerical accuracy of a scheme is not only linked to these errors but also, in a more subtle way, to the main assumption concerning Taylor's expansion in space. Finite differences are based on the interpolation of discrete data using polynomials (like Taylor's series) or other simple functions. As a consequence, schemes of order P can advect without error any polynomial of the same order. However, at a position close to a stiff gradient or discontinuity, the interpolation polynomial will not converge and oscillations will appear. Such oscillations, called Gibb's oscillations, will not decay in magnitude when the mesh is refined. As an example, and to explain why more advanced schemes are used to resolve the advection process, the resolution of (5.2) is realized again using the same parameters for V, Δx, Δt presented in

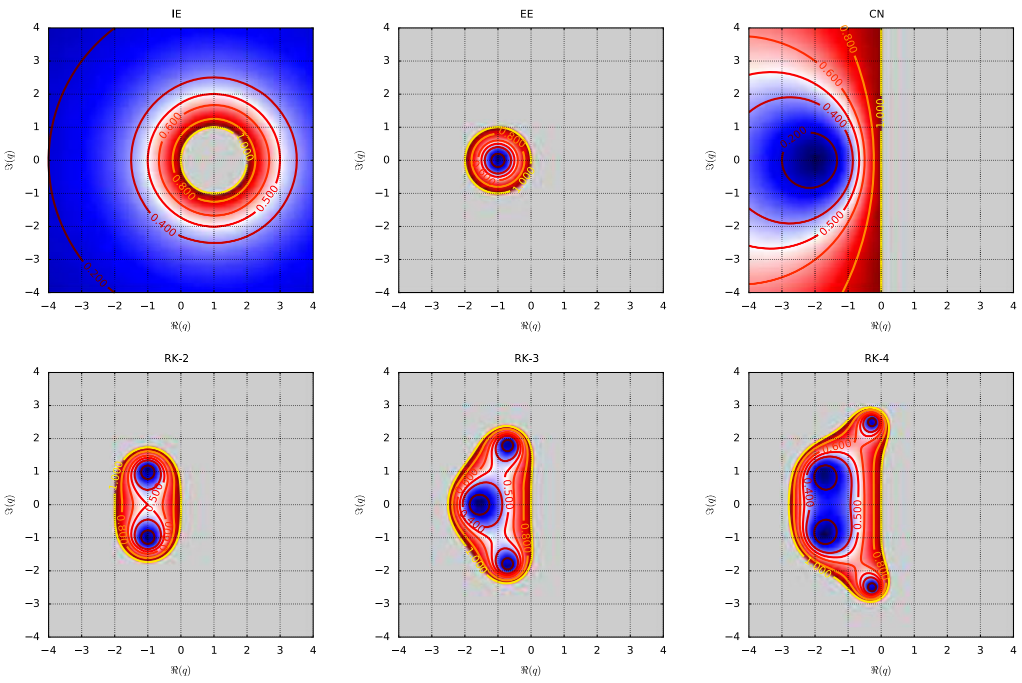

FIGURE 19. Stability diagram for IE, EE, CN, RK-2, RK-3 and RK-4 schemes using the absolute stability criterion. A scheme is stable if $|q| \leqslant 1$ which corresponds to the colour range from blue to red. Grey zones correspond to $|q| > 1$.

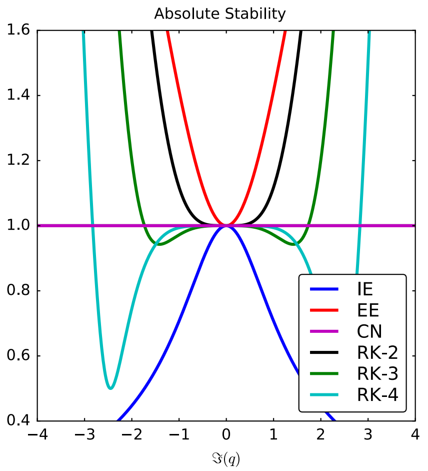

FIGURE 20. Slice of the absolute stability diagrams for IE, EE, CN, RK-2, RK-3 and RK-4 schemes corresponding to values of q purely imaginary.

figure 21. The only change is the width of the Gaussian (σ) used as initial condition. In plasma fluid dynamics, the width of relevant modes are parameter dependent. For example, the width of the eigenfunctions corresponding to resistive ballooning modes

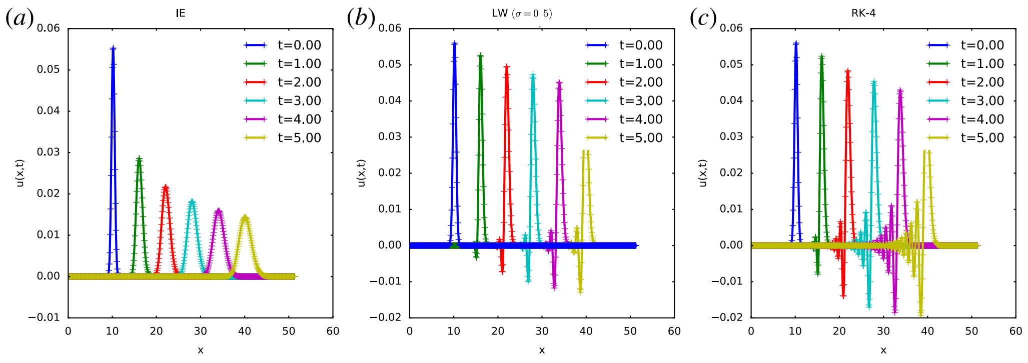

FIGURE 21. Oscillations occurring in the resolution of advection equation ($\sigma = 0.5$) for LW (*b*) and RK-4 (*c*) schemes.

depend on the mode number (see Beyer, Benkadda & Garbet 2000),

$$\left.\begin{array}{c}\text{case } 1 \rightarrow \sigma = 1.5 \\ \text{case } 2 \rightarrow \sigma = 0.5.\end{array}\right\} \tag{5.28}$$

Since the discretization parameters remain the same, a change in the initial profile does not affect the CFL condition. However the RK scheme becomes dissipative and spurious oscillations appear. This behaviour cannot be suppressed switching from central differences for the space derivatives to forward derivatives at the same order. To control and try to avoid this effect, more advanced (higher-order) schemes have been developed. The main hypothesis is that since advection is a conservative process, the algorithm should be formulated to conserve the advected quantity numerically. This formulation is a numerically flux conserving form. Using the Godunov theorem which says that any linear algorithm for solving partial differential equations, with the property of not producing new extrema can be at most first order, there is therefore no hope of finding a linear second-order accurate scheme that does not produce these unwanted oscillations. More advanced nonlinear schemes that combine the higher-order accuracy and the prevention of producing unwanted oscillations can be classified into two categories (see Leveque 1992):

(i) flux-splitting methods (see Boris & Book 1997): artificial viscosity, relaxation schemes or methods based on the application of a limiter to eliminate the oscillations: total variation diminishing (TVD);
(ii) Godunov method (see Godunov 1959): monotonic upwind-centred scheme for conservation laws (MUSCL) scheme, piecewise parabolic method (PPM), essential non-oscillatory (ENO).

In the following, this last method (ENO) is briefly described and an improved version, the weighty essential non-oscillatory (WENO) method is also mentioned (see Shu 1998). As for most advanced schemes for advection, the basic idea is to express the hyperbolic equation as a conservation law

$$\partial_t u(x, t) + \partial_x[f(u(x, t))] = 0. \tag{5.29}$$

As a consequence, the discrete equation can be expressed as,

$$\partial_t u_i = -\frac{\hat{f}_{i+1/2} - \hat{f}_{i-1/2}}{\Delta x}, \tag{5.30}$$

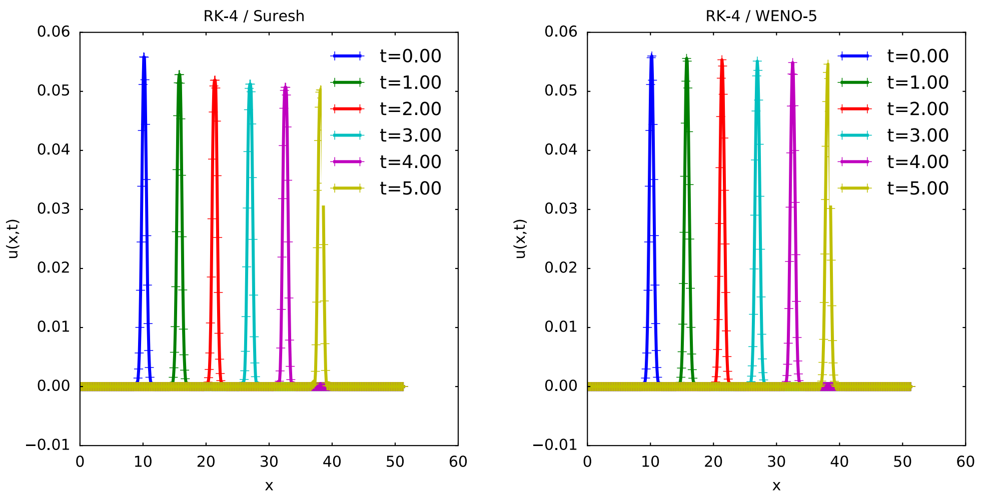

FIGURE 22. Numerical resolution of advection equation for $\sigma = 0.5$ with WENO scheme and parameters: $N_x = 512$, $L_x = 51.2$, and $\Delta t = 1e$–2.

where $\hat{f}$ corresponds to a high-order approximation of the numerical flux. The notation $\mathrm{i} + 1/2$ signifies an estimation of the flux at the interface between two virtual cells of centre i and $i + 1$.

The key idea in ENO schemes to bypass the limitations given by Godunov's theorem is to use an adaptive stencil procedure that results in a non-oscillatory nonlinear interpolation across discontinuities. The weighted essentially non-oscillatory (WENO) scheme is an improvement over the ENO schemes by replacing the stencil selection by a weighted average of the candidate stencils. Smoothness-dependent weights are used such that they approach zero for candidate stencils with discontinuities. Thus, across discontinuities, the WENO schemes behave like the ENO schemes, while in smooth regions of the solution, the weighted average results in a higher-order approximation. An alternative of the WENO scheme has been realized by Suresh and Huyn. This scheme is well adapted for RK time stepping and is accurate and faster than the corresponding fifth-order WENO scheme (see Suresh & Huynh 1997). Figure 22 shows that the higher-order schemes exhibit higher accuracy and the quantity is transported without strong deterioration of the shape. It has to be noticed that only the fifth order is considered here and not the lower-order algorithms (such as third) because these schemes are too dissipative to be considered as accurate. The classical fifth-order WENO has also a dissipative component which is not discussed in detail here. As a consequence, for an accurate resolution of a pure advection problem, an adapted version using an anti-diffusive flux correction technique can be implemented, see Zhengfu & Shu (2005) for details.

6. Nonlinear phenomena: the Poisson bracket

Nonlinear operators in fluid descriptions of plasmas are often expressed by Poisson brackets (Jacobian operator, noted $J(\zeta, \psi)$) between two quantities. The Jacobian is expressed as follows in Cartesian coordinates,

$$J(\zeta, \psi) = \partial_x\zeta(x, y)\partial_y\psi(x, y) - \partial_x\psi(x, y)\partial_y\zeta(x, y). \tag{6.1}$$

The numerical resolution of the Jacobian operator has to be treated carefully. Since nonlinear coupling can drive and dominate energy cascade phenomena and, as a consequence, modifications of the equilibrium or generation of small-scale turbulence may appear, avoiding numerical instabilities or errors (for example truncation errors linked to FD schemes) is important to obtain accurate simulations and results. Compared to linear advection, the computation of the Jacobian is really more expensive. As a consequence using an optimized and accurate method can greatly reduce necessary CPU time for a simulation. Possible algorithms are chosen mainly using geometry properties of the considered system. For fusion plasma simulations, periodicity properties of the tokamak permit to use a Fourier mode decomposition for toroidal and (quite often) poloidal directions. As mentioned previously, (see Canuto *et al.* 1988; Ferszinger 2002), spectral methods are more accurate than any FD scheme. Also, the precision order for calculations of any derivative is higher using Fourier mode decomposition. The accuracy of the calculation of a derivative in real space is determined by the precision order of the considered formula, ($O(\Delta x)$, $O(\Delta x^2)$, ...). In Fourier space, the same accuracy is obtained using only a few modes, it can be considered for example that the calculation with 1 mode corresponds to a precision of $O(\Delta x)$, 2 modes of $O(\Delta x^2)$ and so on.

Using this representation, the resolution of the Jacobian operator in semi spectral space (FD representation in x and spectral representation in y in a 2-D slab geometry) is equivalent to the computation of two discrete convolutions, the associated complexity is of order M^2 with M being the number of modes. However, if the Jacobian is computed in real space, the complexity is of the order M only, which is a benefit with increasing M. Since the fields are stored in a semispectral way, the computation of the discrete Fourier transform (FT) of ζ and ψ must be made first, then calculate the Jacobian in real space, and finally compute an inverse transform (IFT). This method requires more intermediate steps than the direct calculation, but the complexity scales only as $M*(2*\log(M)+1)$. To give an idea of the associated number of operations to perform, if $M=16$, the Fourier method is $\simeq 2.5\times$ faster, if $M=128$, the Fourier method is $\simeq 10\times$ faster. In the 3-D case, using spectral representation with M poloidal and N toroidal modes, the number of operations scales like $N^2 * M^2$ for the direct product and like $N*M*(2*\log(N*M)+1)$ for the Fourier method. As a consequence, the resolution in real space, even with the constraint on the necessary Fourier transforms, is more effective. However, this resolution procedure shadows caveats which can lead to numerical instabilities detailed in the following subsections,

(i) consequences of direct and inverse FT;
(ii) accurate calculation of Jacobian.

6.1. *Fourier transformation*

In computational sciences, the FT corresponds to the discrete Fourier series. As a consequence, the phenomenon of aliasing could appear in the process. Considering a Fourier mode $\mathrm{e}^{\mathrm{i}\boldsymbol{k}_l x_j}$ with $\boldsymbol{k}_l$ the wave vector and x_j the grid point.

$$\boldsymbol{k}_l = 2\pi l/(N*\Delta x), \quad l=-N/2,\ldots,N/2 \tag{6.2}$$
$$x_j = j\Delta x, \quad j=0,1,\ldots,N. \tag{6.3}$$

At the grid point x_j, the following relation holds

$$\mathrm{e}^{\mathrm{i}2\pi(j+l*N)p/N} = \mathrm{e}^{\mathrm{i}2\pi j/N}, \quad p=\ldots,-2,-1,0,1,2,\ldots. \tag{6.4}$$

As a consequence, the modes $k_j = 2\pi j/(N * \Delta x)$ contribute to the discrete FT but, and this is the reason for possible numerical errors, the mode $k_p = 2\pi(j + p * N)/(N * \Delta x)$ also. High-k modes smear out the amplitude of a lower-k mode. This can cause, for example, an increase of the energy of the system. To prevent this aliasing phenomena, several methods have been developed (see Patterson & Orszag 1971), e.g. performing a phase-shifted transform. However, the bottleneck is that the correct transform needs the calculation of eight Fourier transforms each time. The most widely used method is the following one, which can be achieved by enlarging each dimension of the spectral domain by a factor of $(M + 1)/2$ where M is the number of modes. An alternative consist to use $2/3$ of the modes for the Fourier transform, this method is generally called the '2/3 rule'. In general the de-aliasing rule is applied not by an increase of $3/2$ of the number of mode but by an increase to the next power of 2. This is because an optimized algorithm for FT exists when the size is a power of 2, named fast Fourier transform (FFT) method. This algorithm has highly increased performance compared to the approach based on the mathematical definition (see Press *et al.* 2007). In that case, number of necessary operations scales as $N\log(N)$ and not as N^2 for the 1-D case.

6.2. *Accurate Jacobian*

An accurate value of the Jacobian needs a clever differentiating formulation. The Jacobian is mainly a product of derivatives in the x and y directions. Apparently, a second-order FD based on central differences can be implemented,

$$J(\zeta,\psi) = \frac{1}{4\Delta x\Delta y}((\zeta_{i+1,j} - \zeta_{i-1,j}) * (\psi_{i,j+1} - \psi_{i,j-1}) - (\zeta_{i,j+1} - \zeta_{i,j-1}) * (\psi_{i+1,j} - \psi_{i-1,j})). \tag{6.5}$$

As an example, a 2-D mesh is used, and the functions ζ and ψ are defined by,

$$\zeta(x,y) = \mathrm{e}^{-((x-x_0)^2+(y-y_0)^2)} + \mathrm{e}^{-((x+x_0)^2+(y+y_0)^2)} \tag{6.6}$$

$$\psi(x,y) = \partial_x^2\zeta(x,y) + \partial_y^2\zeta(x,y). \tag{6.7}$$

The analytic values of ψ and the associated Jacobian have been analytically calculated and discretized on the same grid to avoid errors associated with any FD scheme (corresponding plots are represented on figure 23).

The Jacobian obtained through a simple product of centre derivatives is plotted in figure 24(*a*,*c*) for two mesh sizes: 64 × 64 and 256 × 256 points, respectively. Comparing to the exact solution (figure 23*c*), the CD method produces inconsistent results due to irrelevant extrema at low resolutions. The amplitudes of these spurious extrema are damped when the resolution is increased but they are still present with a resolution of 256 × 256 points. Using statistical tools, and more precisely the coefficient of determination R^2, a convergence rate to the analytical solution can be expressed as a function of the mesh resolution. The results are plotted on figure 25. It appears that the solution converges to the analytical one when the number of points (and as a consequence the associated precision) increases. However, this convergence requires a large number of points.

Products based on central schemes raise another issue which is more relevant for longer term simulations. Platzmann (see Platzman 1961) has shown that the product of CD leads to the appearance of small eddies which will usually intensify causing computational instability or at least a numerical source of energy in the system. These

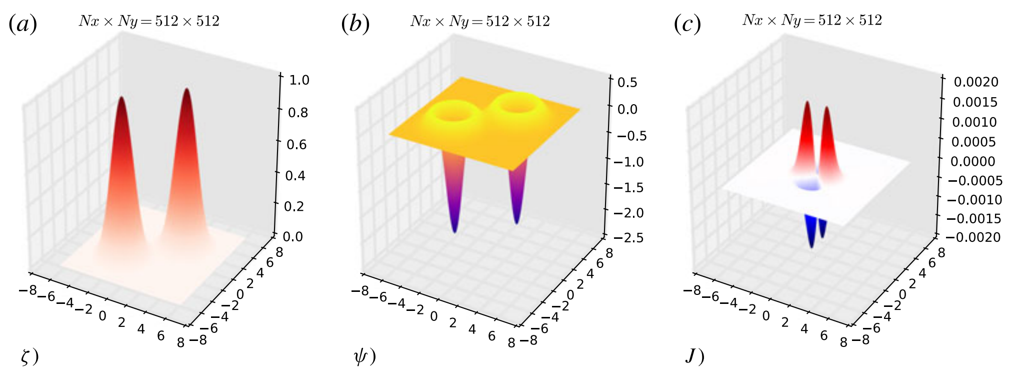

FIGURE 23. Representation of input functions ζ, ψ and calculated Jacobian (J) for a mesh size of 512×512 points.

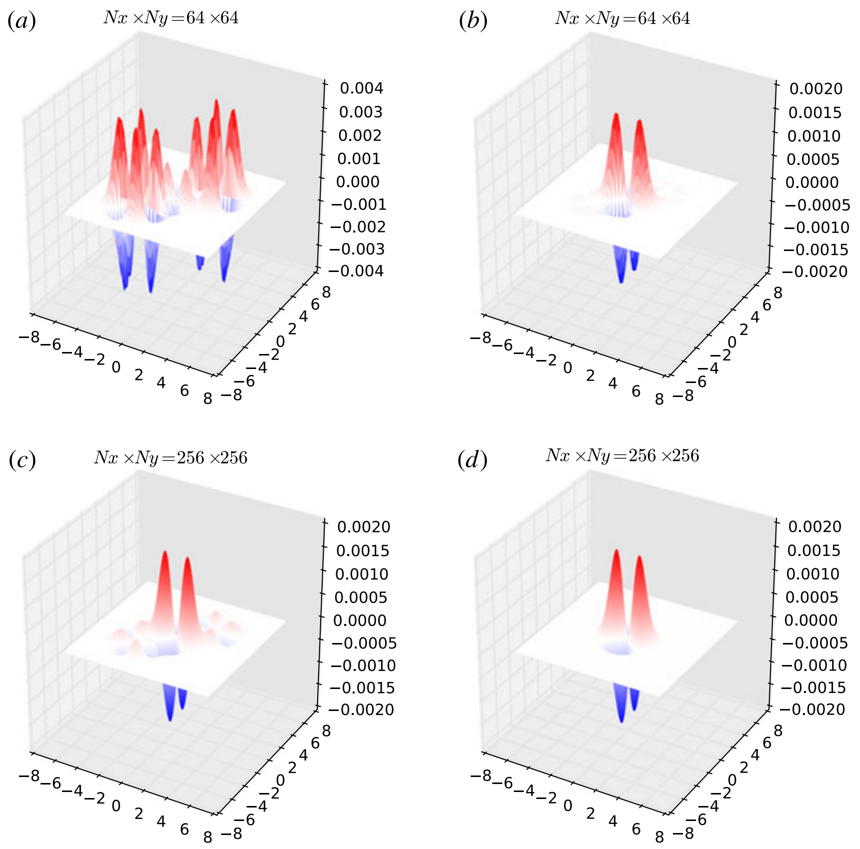

FIGURE 24. Computed Jacobian with CD (*a*,*c*) and Arakawa (*b*,*d*) methods for two grid sizes, 64×64 and 256×256.

eddies, also known as 'noodling' in the literature have been observed in plasma fluid simulations (see Brock & Horton 1982) and they are linked to another aliasing error source which is inherent to FD schemes: a FD scheme is not able to treat correctly

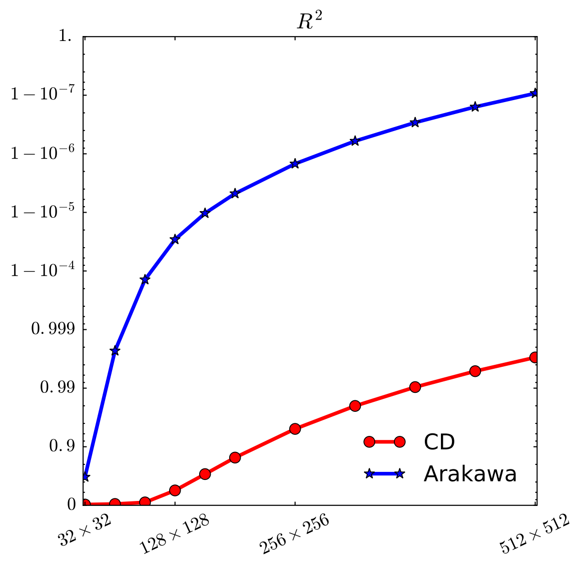

FIGURE 25. R^2 value with respect to exact value of the Jacobian for the 2 schemes, Arakawa and CD.

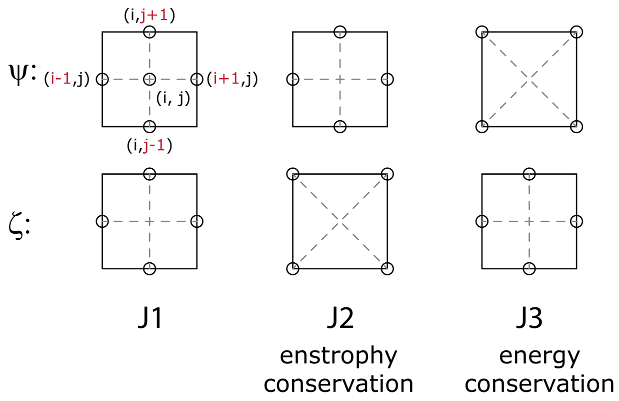

FIGURE 26. Arakawa stencils used to ensure conservation of enstrophy and kinetic energy.

waves with wavenumbers smaller than $\Delta x/2$. A typical consequence is an unexplained growth of the kinetic energy of the system. A more robust scheme for long term simulations which overcomes this computational instability has been developed by Arakawa (see Arakawa 1966). This scheme ensures conservation of energy and enstrophy under the action of the Jacobian. The main idea is to calculate an average between products using central derivatives (called $J1$) and two other alternative formulas which ensure enstrophy conservation ($J2$) and energy conservation $J3$, figure 26.

$$J_1 = \partial_x\zeta\,\partial_y\psi - \partial_y\zeta\,\partial_x\psi \tag{6.8}$$

$$J_2 = -\partial_x(\psi\,\partial_y\zeta) + \partial_y(\psi\,\partial_x\zeta) \tag{6.9}$$

$$J_3 = \partial_x(\zeta\,\partial_y\psi) - \partial_y(\zeta\,\partial_x\psi) \tag{6.10}$$

$$\Rightarrow J(\zeta,\psi) = (J_1+J_2+J_3)/3. \tag{6.11}$$

The procedure can be described as an advanced use of the method of undetermined coefficients for a nine-point stencil for functions ψ and ζ,

$$J_{i,j}(\zeta,\psi)=\sum_{i'=-1}^{1}\sum_{j'=-1}^{1}\sum_{i''=-1}^{1}\sum_{j''=-1}^{1}c_{i,j,i',j',i'',j''}\zeta_{i+i',j+j'}\psi_{i+i'',j+j''}. \tag{6.12}$$

The quantities $a_{i,j,i+i',j+j'}$ and $b_{i,j,i+i'',j+j''}$ which will be used to simplify the final expressions, are defined as,

$$a_{i,j,i+i',j+j'}=\sum_{i''}\sum_{j''}c_{i,j,i',j',i'',j''}\psi_{i+i'',j+j''} \tag{6.13}$$

$$b_{i,j,i+i'',j+j''}=\sum_{i'}\sum_{j'}c_{i,j,i',j',i'',j''}\zeta_{i+i',j+j'}. \tag{6.14}$$

The Jacobian is expressed by,

$$J_{i,j}(\zeta,\psi)=\sum_{i'}\sum_{j'}a_{i,j,i+i',j+j'}\zeta_{i+i',j+j'} \tag{6.15}$$

$$=\sum_{i''}\sum_{j''}b_{i,j,i+i'',j+j''}\psi_{i+i'',j+j''}. \tag{6.16}$$

To verify the properties of the Jacobian and the energy conservation, the subsequent relations between coefficients are used (derivation can be found in Coiffier (2012)),

(i) skew-symmetric property, $J_{i,j}(\zeta,\psi)=-J_{i,j}(\psi,\zeta)$

$$\sum_{i'}\sum_{j'}\sum_{i''}\sum_{j''}c_{i,j,i',j',i'',j''}\zeta_{i+i',j+j'}\psi_{i+i'',j+j''}$$
$$=-\sum_{i'}\sum_{j'}\sum_{i''}\sum_{j''}c_{i,j,i',j',i'',j''}\zeta_{i+i'',j+j''}\psi_{i+i',j+j'}$$
$$\Rightarrow c_{i,j,i',j',i'',j''}=-c_{i,j,i'',j'',i',j'}, \tag{6.17}$$

(ii) conservation of enstrophy $\int \mathrm{dS}\,\zeta J(\zeta,\psi)=0$

$$\sum_{i}\sum_{j}\Delta x\Delta y\zeta_{i,j}J_{i,j}(\zeta,\psi)$$
$$=\sum_{i}\sum_{j}\sum_{i'}\sum_{j'}\Delta x\Delta y a_{i,j,i+i',j+j'}\zeta_{i,j}\zeta_{i+i',j+j'}$$
$$\Rightarrow a_{i+i',j+j',i,j}=-a_{i,j,i+i',j+j'}, \tag{6.18}$$

(iii) conservation of kinetic energy $\int \mathrm{dS}\psi J(\zeta,\psi)=0$

$$\sum_{i}\sum_{j}\Delta x\Delta y\psi_{i,j}J_{i,j}(\zeta,\psi)$$
$$=\sum_{i}\sum_{j}\sum_{i''}\sum_{j''}\Delta x\Delta y b_{i,j,i+i'',j+j''}\psi_{i,j}\psi_{i+i'',j+j''}$$
$$\Rightarrow b_{i+i'',j+j'',i,j}=-b_{i,j,i+i'',j+j''}. \tag{6.19}$$

These three properties lead to the final formula for the discrete Jacobian,

$$J(\zeta,\psi) = \frac{-1}{12\Delta x\Delta y}[(\psi_{i,j-1}+\psi_{i+1,j-1}-\psi_{i,j+1}-\psi_{i+1,j+1}) * \zeta_{i+1,j} \tag{6.20}$$

$$-(\psi_{i-1,j-1}+\psi_{i,j-1}-\psi_{i-1,j+1}-\psi_{i,j+1}) * \zeta_{i-1,j} \tag{6.21}$$

$$+(\psi_{i+1,j}+\psi_{i+1,j+1}-\psi_{i-1,j}-\psi_{i-1,j+1}) * \zeta_{i,j+1} \tag{6.22}$$

$$-(\psi_{i+1,j-1}+\psi_{i+1,j}-\psi_{i-1,j-1}-\psi_{i-1,j}) * \zeta_{i,j-1} \tag{6.23}$$

$$+(\psi_{i+1,j}-\psi_{i,j+1}) * \zeta_{i+1,j+1} \tag{6.24}$$

$$-(\psi_{i,j-1}-\psi_{i-1,j}) * \zeta_{i-1,j-1} \tag{6.25}$$

$$+(\psi_{i,j+1}-\psi_{i-1,j}) * \zeta_{i-1,j+1} \tag{6.26}$$

$$-(\psi_{i+1,j}-\psi_{i,j-1}) * \zeta_{i+1,j-1}]\,. \tag{6.27}$$

This formulation is not the one given in the original paper (see Arakawa 1966) but an optimized one ($\simeq$15 % faster) presented in Kuhn *et al.* (2013). Notice that in Kuhn *et al.* (2013), an optimized implementation of the full algorithm (FT+Jacobian+IFT) is presented and the resulting performance increase is approximately 75 % compared to a basic serial implementation. The results are plotted in figure 24(*b*,*d*). Compared to the CD scheme, the obtained Jacobian is more accurate. This is confirmed in figure 25 where the value of R^2 is plotted. Even with this simple example, the increased accuracy of the Arakawa scheme compared to a simple product of central differences is obvious. The increased accuracy is directly linked to the associated properties and not to the order since the implementation used is also second order as for the product of derivatives. A more detailed comparison of different schemes for the resolution of the Jacobian operator in PDEs has been made in Naulin & Nielsen (2003).

7. Summary and discussion

Plasma fluid computations can be seen as a resolution of systems of partial differential equations. They therefore are subject to the main and principle difficulties linked to the resolution of such equations. The diversity of the physical processes involved in plasma physics leads to models implying a sum of mathematical operators. As a consequence, the associated numerical and mathematical limitations have to be treated carefully.

This paper focused on the main caveats that have to be avoided in plasma fluid simulations and on the algorithms permitting to avoid them. The numerical limitations are linked mainly to the presence of a finite grid and the correct treatment of small scales as well as wave propagation. An accurate treatment of small scales is necessary to study the role of turbulence on plasma dynamics and transport.

The increased performance of available super computers allows for the realization of simulations with higher resolutions for smaller spatial scales. As a consequence, the highest resolution simulations are closer to the experiments. For that purpose, numerical challenges which have not been presented in detail here lie in the matrix algebra and matrix inversion processes associated with the Poisson equation. The reason is linked to the mathematical nature of this equation, more precisely, the resolution of the Poisson equation at one grid point requires to gather data from the complete domain with the methods presented here. As a consequence, the communication due to the parallel resolution has a huge cost. More advanced methods/schemes can be used in that case, like the iterative approach, the conjugate gradient . . . (see Jardin 2010).

Considerable research and improvement over the past years have been accomplished also for the algorithms themselves. For that purpose, and in general, for all algebra linked to matrix inversions, resolution of linear systems, and matrix computations linked to implicit methods, specialized libraries developed should be used, e.g. MUMPS (2015), PasTiX (2015), SuperLU (2015). As cited in § 6, another time consuming computation which can be greatly reduced through the use of optimized algorithms and libraries is the computation of Fourier transforms, in that case a widely used library is the FFTW library (see FFTW 2015). During the development process, with the associated tests of methods, an alternative is to use a global framework for the resolution of PDEs like PETSc (see Balay *et al.* 2016).

Acknowledgements

The authors would like to thank W. Horton for helpful discussions and remarks on the manuscript. This work was granted access to the HPC resources of Aix-Marseille Université financed by the project Equip@Meso (ANR-10-EQPX-29-01) of the program ‘Investissements d’Avenir’ supervised by the Agence Nationale pour la Recherche.

Appendix A. List of abbreviations used

AM	Adams–Moulton
AB	Adams–Bashforth
BWD	backward difference
CD	central difference
CFL	Courant–Friedrichs–Lewy
CG	coarse graining
CN	Crank–Nicholson
EE	explicit Euler
ENO	essentially non-oscillatory scheme
FD	finite difference
FT	Fourier transform
FFT	fast Fourier transform
FWD	forward difference
IE	implicit Euler
IFT	inverse Fourier transform
LW	Lax–Wendroff scheme
MPI	message passing interface
PDE	partial differential equation
RK	Runge–Kutta scheme
WENO	weighted essentially non-oscillatory scheme

Appendix B. Eigenvalues and eigenfunctions of circulant matrices

In this appendix, the expression of eigenvalues for a periodic tridiagonal system is derived based on the derivation of eigenvalues of circulant matrices. Consider a circulant matrix $\boldsymbol{B}$ of size $M \times M \in \mathbb{R}^2$ of the form,

$$\boldsymbol{B} = \begin{bmatrix} c_0 & c_1 & c_2 & & \cdots & c_{M-1} \\ c_{M-1} & c_0 & c_1 & c_2 & \cdots & c_{M-2} \\ & c_{M-1} & c_0 & c_1 & & \\ & & c_{M-1} & c_0 & c_1 & \vdots \\ \ddots & & \ddots & \ddots & \ddots & c_2 \\ & & & & & c_1 \\ c_1 & & & & c_{M-1} & c_0 \end{bmatrix}. \tag{B 1}$$

A given eigenvalue λ and associated eigenvector $\boldsymbol{e}$ with components $\boldsymbol{e}_{(k)}$ verify the relation

$$\boldsymbol{Be} = \lambda \boldsymbol{e}. \tag{B 2}$$

Equation (B 2) should be expressed first for each component $\boldsymbol{e}_{(k)}$:

$$\sum_{p=0}^{M-k-1} c_p \boldsymbol{e}_{(p+k)} + \sum_{p=M-k}^{M-1} c_p \boldsymbol{e}_{(p-(M-k))} = \lambda \boldsymbol{e}_{(k)}, \quad k = 0, 1, \ldots, M-1. \tag{B 3}$$

Assumption is made that the eigenvectors components have the form

$$\boldsymbol{e}_{(p)} = r^p. \tag{B 4}$$

Inserting into (B 3) and simplifying by r^k,

$$\sum_{p=0}^{M-k-1} c_p r^p + r^{-M} \sum_{p=M-k}^{M-1} c_p r^p = \lambda, \quad k = 0, 1, \ldots, M-1. \tag{B 5}$$

Supposing moreover that r is one of the m distinct complex Mth roots of unity,

$$\sum_{p=0}^{M-1} c_p r^p = \lambda. \tag{B 6}$$

The relation (B 6) indicates that, with the assumption made in (B 4), the relation (B 2) is verified and therefore $\boldsymbol{e}$ is an eigenvector with λ as corresponding eigenvalue

$$\boldsymbol{e} = \frac{1}{\sqrt{M}} \begin{pmatrix} 1 \\ r \\ r^2 \\ \vdots \\ r^{M-1} \end{pmatrix}. \tag{B 7}$$

The coefficient $1/\sqrt{M}$ is added to normalize the obtained eigenvectors.

Possible eigenvalues deduced from (B 6) are not unique, more precisely, the relation (B 6) is verified for s different values of λ, $\{\lambda_0, \lambda_1, \ldots, \lambda_{M-1}\}$, each one corresponding to a possible root of unity. Finally the set of possible eigenvalues corresponds to

$$\lambda_s = \sum_{p=0}^{M-1} c_p \exp(-2\mathrm{i}\pi sp/M), \quad s = 0, 1, \ldots, M-1. \tag{B 8}$$

Equation (B 8) can be used to deduce eigenvalues associated with discrete expression of derivatives on a closed domain with periodic boundary conditions, $c_k \neq 0$ if $k \in [0, 1, M-1]$, the following simplified relation is obtained,

$$\lambda_s = c_0 + c_1 \exp(-2\mathrm{i}\pi s/M) + c_{M-1} \exp(-2\mathrm{i}\pi s(M-1)/M) \tag{B 9}$$

$$= c_0 + c_1 \exp(-2\mathrm{i}\pi s/M) + c_{M-1} \exp(2\mathrm{i}\pi s/M). \tag{B 10}$$

Appendix C. Definition of functions used in resolution of diffusion equation

C.1. *Common functions*

```
void calclaplacien(double const*const src, double *const dest,double
    const dx2, double const D,long const size)
{
    long i=0;
    long const sizeloop = size-1;
    double sm = src[0];     /* represent element at index i-1 */
    double sc = src[1];     /* represent element at index i */
    double sp = 0.;         /* represent element at index i+1 */
    double const Didx2 = D/dx2;

    for( i=1;i<sizeloop;i++)
    {
        sp = src[i+1];
        dest[i] = (sm -2.*sc +sp)*Didx2;
        sm = sc;
        sc = sp;
    }
}


/* dest[] = dest[]+ src[] */
void addField(double const*const src, double *const dest, long const
    size)
{
    long i=0;
    for( i=0;i<size;i++)
    {
        dest[i] += src[i];
    }

}

/* calculate values of field at index 0 and Nx-1 corresponding to ghost
     points */
void Generate_BC(double *const dest, const long localSize)
{
    dest[0] = dest[2];
    dest[localSize-1] = -dest[localSize-3];
}

/* dest[] =  val* src2[] */
void multField(double const*const src, double *const dest, long const
    size)
{
    long i=0;
    for( i=0;i<size;i++)
    {
        dest[i]*=src[i];
    }
}
```

```

/* dest[] =  src1[] */
void copyField(double const*const src, double *const dest, long const
    size)
{
    long i=0;
    for( i=0;i<size;i++)
    {
        dest[i]=src[i];
    }
}

/* dest[] =  src1[] + src2[] */
void copyandaddField(double const*const src1,double const*const src2,
    double *const dest, long const size)
{
    long i=0;
    for( i=0;i<size;i++)
    {
        dest[i]=src1[i]+src2[i];
    }
}

/* dest[] =  src1[] + val* src2[] */
void copyandaddmultField(double const*const src1,double const*const
    restrict src2, double *const dest,double const dFac,  long const
    size)
{
    long i=0;
    for( i=0;i<size;i++)
    {
        dest[i]=src1[i]+dFac*src2[i];
    }
}


/* dest[] = val*src[] */
void copyandmultField(double const*const src,double const val, double *
    const dest, long const size)
{
    long i=0;
    for( i=0;i<size;i++)
    {
        dest[i]=src[i]*val;
    }
}
```

C.2. *MPI functions*

```

void MPI_local2global(double const*const src_local, double *const
    dest_global, struc_map const *const mpimap)
{
    int const buffsize = mpimap->localSize-2;

    MPI_Gather(src_local+1, buffsize, MPI_DOUBLE,
        dest_global+1, buffsize, MPI_DOUBLE,
        0, MPI_COMM_WORLD);

}

void MPI_global2local(double const*const src_global, double *const
    dest_local, struc_map const *const mpimap)
```

```
{

    MPI_Scatter(src_global+1, mpimap->localSize-2, MPI_DOUBLE,
        dest_local+1, mpimap->localSize-2, MPI_DOUBLE,
        0, MPI_COMM_WORLD);
}


void MPI_copy_BC(double *const pr_array, struc_map const *const
    pr_mpimap)
{
    double mpitemp, mpitemps;
    int errs = 0;
    int errr = 0;
    int errw = 0;
    MPI_Status   status;
    MPI_Request␣send_request,recv_request;
    const long LGP = 0;                          /* left ghost point index
        */
    const long RGP = pr_mpimap->localSize-1;     /* right ghost point index
         */
    const long LBP = LGP+1;                      /* left boundary point */
    const long RBP = RGP-1;                      /* right boundary point */
    int const tag = 42;

    /* only one core used, no communications*/
    if ((pr_mpimap->leftProc==-1) && (pr_mpimap->rightProc==-1))
    {
        pr_array[LGP]= pr_array[LBP+1];
        pr_array[RGP] = -pr_array[RBP-1];
    } /* core has no left neighbour */
    /* no left neighbor, send/recv only with right neighbor */
    else if (pr_mpimap->leftProc==-1)
    {
        pr_array[LGP] = pr_array[LBP+1];
        mpitemps = pr_array[RBP];
        errr = MPI_Irecv(&mpitemp, 1,MPI_DOUBLE, pr_mpimap->rightProc,
            tag, MPI_COMM_WORLD, &recv_request);
        errs = MPI_Send(&mpitemps, 1,MPI_DOUBLE, pr_mpimap->rightProc,
            tag, MPI_COMM_WORLD);
        errw = MPI_Wait(&recv_request,&status);
        pr_array[RGP] = mpitemp;
    }
    /* no right neighbor, send/recv only with left neighbor */
    else if (pr_mpimap->rightProc==-1)
    {
        mpitemps = pr_array[LBP];
        MPI_Irecv(&mpitemp, 1,MPI_DOUBLE, pr_mpimap->leftProc, tag,
            MPI_COMM_WORLD, &recv_request);
        MPI_Send(&mpitemps, 1,MPI_DOUBLE, pr_mpimap->leftProc, tag,
            MPI_COMM_WORLD);
        errw=MPI_Wait(&recv_request,&status);
        pr_array[LGP] = mpitemp;
        pr_array[RGP] = - pr_array[RBP-1];
    }
    else
    {
        /*send/recv with left neighbor */
        mpitemps = pr_array[LBP];
        MPI_Irecv(&mpitemp,1,MPI_DOUBLE,pr_mpimap->leftProc,tag,
            MPI_COMM_WORLD,&recv_request);
        MPI_Send(&mpitemps,1,MPI_DOUBLE,pr_mpimap->leftProc,tag,
            MPI_COMM_WORLD);
        errw=MPI_Wait(&recv_request,&status);
        pr_array[LGP] = mpitemp;

        /*send/recv with right neighbor */
        mpitemps = pr_array[RBP];
        MPI_Irecv(&mpitemp,1,MPI_DOUBLE,pr_mpimap->rightProc,tag,
            MPI_COMM_WORLD,&recv_request);
        MPI_Send(&mpitemps,1,MPI_DOUBLE,pr_mpimap->rightProc,tag,
            MPI_COMM_WORLD);
        errw=MPI_Wait(&recv_request,&status);
        pr_array[RGP] = mpitemp;
    }
}
```

C.3. *OpenMP functions*

Listing 7: main OpenMP block used for the time loop.

```

#ifdef __GNUC__
#pragma omp parallel default(none) private(i,j) shared(g_fsrc, g_P0,sp,
    sh_boundary,sh_lockvar)
#else
#pragma novector
#pragma omp parallel default(none) private(i,j) shared(dt,D,tmax,t_out,
    dt2,dt6, g_fsrc, g_P0,sp,sh_boundary,sh_lockvar)
#endif
    {
        struc_map ompmap = OMP_Map_generate(sp.Size);

        double *const pr_fsrc=malloc(ompmap.localSize*sizeof(*pr_fsrc))
            ;
        double *const K1=malloc(ompmap.localSize*sizeof(*K1));
        double *const K2=malloc(ompmap.localSize*sizeof(*K2));
        double *const K3=malloc(ompmap.localSize*sizeof(*K3));
        double *const K4=malloc(ompmap.localSize*sizeof(*K4));
        double *const F1=malloc(ompmap.localSize*sizeof(*F1));
        double *const F2=malloc(ompmap.localSize*sizeof(*F2));
        double *const F3=malloc(ompmap.localSize*sizeof(*F3));
        double *const Ftmp=malloc(ompmap.localSize*sizeof(*Ftmp));
        double *const pr_P0=malloc(ompmap.localSize*sizeof(*pr_P0));
        double *const pr_P1=malloc(ompmap.localSize*sizeof(*pr_P1));
        double const dx2 = sp.dx*sp.dx;

        /*copy data from global memory to thread private memory*/
        OMP_global2local(g_fsrc,pr_fsrc,&ompmap);
        OMP_global2local(g_P0,pr_P0,&ompmap);

        /* B.C local and globals should be initialized correctly also
            */
        OMP_generate_BC(sh_boundary,pr_P0,&ompmap);
#pragma omp barrier

        OMP_copy_BC(sh_boundary,pr_P0,&ompmap);

        /*time loop using RK4 scheme*/
        /*using coarse graining decomposition */
        for(i=0;i<tmax;i+=1)
        {

                calclaplacien(pr_P0,K1, dx2, D, ompmap.localSize);
                addField(pr_fsrc,K1,ompmap.localSize);
                copyandaddmultField(pr_P0, K1, F1, dt2, ompmap.localSize);
                OMP_generate_BC(sh_boundary,F1,&ompmap);

#pragma omp barrier
                OMP_copy_BC(sh_boundary,F1,&ompmap);

                calclaplacien(F1,K2, dx2, D, ompmap.localSize);
                addField(pr_fsrc,K2,ompmap.localSize);
                copyandaddmultField(pr_P0, K2, F2, dt2, ompmap.localSize);
                OMP_generate_BC(sh_boundary,F2,&ompmap);

#pragma omp barrier
                OMP_copy_BC(sh_boundary,F2,&ompmap);

                calclaplacien(F2,K3, dx2, D, ompmap.localSize);
                addField(pr_fsrc,K3,ompmap.localSize);
                copyandaddmultField(pr_P0, K3, F3, dt, ompmap.localSize);
                OMP_generate_BC(sh_boundary,F3,&ompmap);

#pragma omp barrier
                OMP_copy_BC(sh_boundary,F3,&ompmap);

                calclaplacien(F3,K4, dx2, D, ompmap.localSize);   /* K4 = nabla(chi nabla)F3 */
                addField(pr_fsrc,K4,ompmap.localSize);

                copyandaddField(K2,K3,pr_P1,ompmap.localSize);
                multCnst(2.,pr_P1,ompmap.localSize);
                addField(K4,pr_P1,ompmap.localSize);
                addField(K1,pr_P1,ompmap.localSize);
                multCnst(dt6,pr_P1,ompmap.localSize);
                addField(pr_P1,pr_P0,ompmap.localSize);
                OMP_generate_BC(sh_boundary,pr_P0,&ompmap);

#pragma omp barrier
                OMP_copy_BC(sh_boundary,pr_P0,&ompmap);

        }

        free( pr_fsrc );
        free( K1 );
        free( K2 );
        free( K3 );
        free( K4 );
        free( F1 );
        free( F2 );
        free( F3 );
        free( Ftmp );
        free( pr_P0 );
        free( pr_P1 );
    } /*OMP block end*
```

```

void OMP_globalBoundary(double *const dest, const long globalSize)
{
    dest[0]=-dest[2];
    dest[globalSize-1]=-dest[globalSize-3];
}
```

```


void OMP_local2global(double const*const src_local, double *const
    dest_global, struc_map const *const ompmap)
{
    long i=0;
    double *const dest_tmp= dest_global+ompmap->posGlobalMin;
    const long loopsize = ompmap->localSize-1;

#pragma ivdep
    for(i=1;i<loopsize;i++)
    {
        dest_tmp[i]=src_local[i];
    }
    if (ompmap->rightProc==-1)
    {
        dest_tmp[ompmap->globalSize-1] = src_local[loopsize];
    }
    if (ompmap->leftProc==-1)
    {
        dest_tmp[0] = src_local[0];
    }

}

void OMP_global2local(double const*const src_global, double *const
    dest_local, struc_map const *const ompmap)
{
    long i=0;
    double const *const src_tmp= src_global+ompmap->posGlobalMin;
    const long loopsize = ompmap->localSize;

#pragma ivdep
    for(i=1;i<loopsize;i++)
    {
        dest_local[i]=src_tmp[i];
    }

}
```

```
void OMP_copy_BC(double const *const sh_src, double *const pr_dest,
    struc_map const *const pr_ompmap)
{
    /*copy data from sh_src to private array pr_dest*/
    const long localPos = 2*pr_ompmap->actualProc;
    pr_dest[0]=sh_src[localPos];
    pr_dest[pr_ompmap->localSize-1]=sh_src[localPos+1];
}
```

```
void OMP_generate_BC(double *const sh_dest, double const*const pr_src,
    struc_map const *const pr_ompmap)
{
    /*Boundaries (ghost points) communicated through shared array
        sh_dest */
    const long localPos = 2*pr_ompmap->actualProc;

    /* only one thread used */
    if ((pr_ompmap->leftProc==-1) && (pr_ompmap->rightProc==-1))
    {
#pragma omp atomic write
        sh_dest[localPos]=pr_src[2];
#pragma omp atomic write
        sh_dest[localPos+1]=-pr_src[pr_ompmap->localSize-3];
    }
                /*current thread corresponds to beginning of the domain => no left neighbour */
    else if (pr_ompmap->leftProc==-1)
    {
#pragma omp atomic write
        sh_dest[localPos]=pr_src[2];
#pragma omp atomic write
        sh_dest[localPos+2]=pr_src[pr_ompmap->localSize-2];
    }
                /*current thread corresponds to end of the domain => no right neighbour */
    else if (pr_ompmap->rightProc==-1)
    {
#pragma omp atomic write
        sh_dest[localPos-1]=pr_src[1];
#pragma omp atomic write
        sh_dest[localPos+1]=-pr_src[pr_ompmap->localSize-2];
    }
    else
    {
#pragma omp atomic write
        sh_dest[localPos-1]=pr_src[1];
#pragma omp atomic write
        sh_dest[localPos+2]=pr_src[pr_ompmap->localSize-2];
    }
}
```

REFERENCES

AMDAHL, G. M. 1967 Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18–20, 1967, Spring Joint Computer Conference, AFIPS '67 (Spring)*, pp. 483–485. ACM.

ARAKAWA, A. 1966 Computational design for long-term numerical integration of the equations of fluid motion: two-dimensional incompressible flow. Part I. *J. Comput. Phys.* **1** (1), 119–143.

BALAY, S., ABHYANKAR, S., ADAMS, M. F., BROWN, J., BRUNE, P., BUSCHELMAN, K., DALCIN, L., EIJKHOUT, V., GROPP, W. D., KAUSHIK, D. *et al.* 2016 PETSc webpage. http://www.mcs.anl.gov/petsc.

BEYER, P., BENKADDA, S. & GARBET, X. 2000 Proper orthogonal decomposition and galerkin projection for a three-dimensional plasma dynamical system. *Phys. Rev.* E **61** (1), 813–823.

BISKAMP, D. 1993 *Nonlinear Magnetohydrodynamics*. Cambridge University Press.

BORIS, J. P. & BOOK, D. L. 1997 Flux-corrected transport. *J. Comput. Phys.* **135** (2), 172–186.

BRAGINSKII, S. I. 1965 Transport processes in a plasma. *Rev. Plasma Phys.* **1**, 205–311.

BROCK, D. & HORTON, W. 1982 Toroidal drift-wave fluctuations driven by ion pressure gradients. *Plasma Phys.* **24** (3), 271.

BUTCHER, J. C. 2008 *Numerical Methods for Ordinary Differential Equations*, 2nd edn. Wiley.

CANUTO, C., HUSSAINI, M., QUARTERONI, A. & ZANG, T. 1988 *Spectral Methods in Fluid Dynamics*, Springer Series in Computational Physics. Springer.

CHAPMAN, B., JOST, G. & VAN DER PAS, R. 2007 *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press.

COIFFIER, J. 2012 *Fundamentals of Numerical Weather Prediction*. Cambridge University Press.

COURANT, R., FRIEDRICHS, K. & LEWY, H. 1928 über die partiellen differenzengleichungen der mathematischen physik. *Math. Ann.* **100** (1), 32–74.

COURANT, R., ISAACSON, E. & REES, M. 1952 On the solution of nonlinear hyperbolic differential equations by finite differences. *Commun. Pure Appl. Maths* **5** (3), 243–255.

CROUSEILLES, N., KUHN, M. & LATU, G. 2015 Comparison of numerical solvers for anisotropic diffusion equations arising in plasma physics. *J. Sci. Comput.* **65** (3), 1091–1128.

D'HAESELEER, W. D. 1991 *Flux Coordinates and Magnetic Field Structure: A Guide to a Fundamental Tool of Plasma Structure*, Springer Series in Computational Physics. Springer.

DURRAN, D. R. 1999 *Numerical Methods for Wave Equations in Geophysical Fluid Dynamics*. Springer.

DURRAN, D. R. 2010 *Numerical Methods for Fluid Dynamics*, 2nd edn. Springer.

FERSZINGER, H. 2002 *Computational Methods for Fluid Dynamics*. Springer.

FFTW 2015 FFTW webpage. http://www.fftw.org/.

FREIDBERG, J. P. 2007 *Plasma Physics and Fusion Energy*. Cambridge University Press.

GODUNOV, S. K. 1959 Finite difference method for numerical computation of discontinuous solutions of the equations of fluid dynamics. *Mat. Sb.* **47**, 271–300.

GRIFFITHS, D. F. 2010 *Numerical Methods for Ordinary Differential Equations*. Springer.

GUSTAFSON, J. L. 1988 Reevaluating Amdahl's law. *Commun. ACM* **31** (5), 532–533.

HAIRER, E. 2006 *Geometric Numerical Integration*. Springer.

HIRSCH, C. 2007 *Numerical Computation of Internal and External Flows, Volume 1, Fundamentals of Computational Fluid Dynamics*, 2nd edn. Butterworth-Heinemann.

HORTON, W. & ESTES, R. D. 1980 Fluid simulation of ion pressure gradient driven drift modes. *Plasma Phys.* **22** (7), 663.

JARDIN, S. 2010 *Computational Methods in Plasma Physics*. CRC Press.

JARDIN, S. 2011 Review of implicit methods for the magnetohydrodynamic description of magnetically confined plasmas. *J. Comput. Phys.* **231**, 822–838.

KARNIADAKIS, G. E., ISRAELI, M. & ORSZAG, S. A. 1991 High-order splitting methods for the incompressible Navier–Stokes equations. *J. Comput. Phys.* **97** (2), 414–443.

KUHN, M., LATU, G., GENAUD, S. & CROUSEILLES, N. 2013 Optimization and parallelization of emerged on shared memory architecture. In *Proceedings of the 2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC '13*, pp. 503–510. IEEE Computer Society.

LAX, P. & WENDROFF, B. 1960 Systems of conservation laws. *Commun. Pure Appl. Maths* **13** (2), 217–237.

LEVEQUE, R. J. 1992 *Numerical Methods for Conservation Laws*, Lectures in Mathematics. Birkhäuser.

LEVEQUE, R. J. 2002 *Finite Volume Methods for Hyperbolic Problems*. Cambridge University Press.

LOMAX, H., PULLIAM, T. H. & ZINGG, D. W. 2001 *Fundamentals of Computational Fluid Dynamics*. Springer.

MESSAGE PASSING INTERFACE FORUM 2015 *MPI: A Message-Passing Interface Standard Version 3.1*. High Performance Computing Center Stuttgart (HLRS).

MUMPS 2015 MUMPS webpage. http://mumps-solver.org/.

NAULIN, V. & NIELSEN, A. H. 2003 Accuracy of spectral and finite difference schemes in 2d advection problems. *SIAM J. Sci. Comput.* **25** (1), 104–126.

OPENMP 2015 OpenMP tutorial from LLNL. https://computing.llnl.gov/tutorials/openMP/.

PASTIX 2015 PasTiX webpage. http://pastix.gforge.inria.fr/.

PATTERSON, G. S. & ORSZAG, S. A. 1971 Spectral calculations of isotropic turbulence: efficient removal of aliasing interactions. *Phys. Fluids* **14** (11), 2538–2541.

PLATZMAN, G. W. 1961 An approximation to the product of discrete functions. *J. Meteorol.* **18** (1), 31–37.

PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T. & FLANNERY, B. P. 2007 *Numerical Recipes: The Art of Scientific Computing*, 3rd edn. Cambridge University Press.

ROSEN, J. S. 1967 The Runge–Kutta equations by quadrature methods. *NASA Tech. Rep.* TR-R-275.

SCHNEIDER, K., KOLOMENSKIY, D. & DERIAZ, E. 2013 *Is the CFL Condition Sufficient? Some Remarks*. pp. 139–146. Birkhäuser Boston.

SCOTT, B. D. 1997 Three-dimensional computation of drift Alfvén turbulence. *Plasma Phys. Control. Fusion* **39** (10), 1635.

SHU, C.-W. 1998 *Essentially Non-Oscillatory and Weighted Essentially Non-Oscillatory Schemes for Hyperbolic Conservation Laws*. pp. 325–432. Springer.

SHU, C.-W. & OSHER, S. 1988 Efficient implementation of essentially non-oscillatory shock-capturing schemes. *J. Comput. Phys.* **77** (2), 439–471.

SMITH, G. D. 1978 *Numerical Solution of Partial Differential Equations*, 2nd edn. Clarendon.

SUPERLU 2015 SuperLU webpage. http://crd-legacy.lbl.gov/∼xiaoye/SuperLU/.

SURESH, A. & HUYNH, H. T. 1997 Accurate monotonicity-preserving schemes with Runge–Kutta time stepping. *J. Comput. Phys.* **136** (1), 83–99.

WESSON, J. 1997 *Tokamaks*, 2nd edn. Clarendon.

ZHENGFU, X. & SHU, C.-W. 2005 Anti-diffusive flux corrections for high order finite difference WENO schemes. *J. Comput. Phys.* **205** (2), 458–485.