CERFACS
CENTRE EUROPÉEN DE RECHERCHE ET DE FORMATION AVANCÉE EN CALCUL SCIENTIFIQUE

INP TOULOUSE ENSEEIHT

Third-year internship report

Study and implementation of implicit time integration methods in a high-order CFD code

Author: Thomas MARCHAL

Supervisors: Marc MONTAGNAC, Laurent MUSCAT, Adèle VEILLEUX

November 14, 2018

Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique
42 avenue Coriolis – 31057 Toulouse Cedex 1 – France
Tél : +33 5 61 19 31 31 – Fax : +33 5 61 19 30 00
http://www.cerfacs.fr – e-mail: secretar@cerfacs.fr

Contents

Nomenclature

f	Flux point index
int	Refer to interface
s	Solution point index
α, α_{SD}	Usual Fourier number and its equivalent in SD
β	Relative numerical error ratio between 2 meshes
$\boldsymbol{A}$, $\boldsymbol{b}$, $\boldsymbol{c}$	The matrix and the two vectors of Butcher's tableau for RK schemes
$\Delta x^i = x^{i+1} - x^i$	Size of cell i
δ	Small scalar in the JFNK context
ϵ	Perturbation for the numerical computation of the Jacobian of the residual
$\epsilon_{Kry,a}$	Absolute decrease tolerance for the norm of the linear residual in the GMRES algorithm
$\epsilon_{Kry,r}$	Relative decrease tolerance, from an initial evaluation, for the norm of the linear residual in GMRES algorithm
$\epsilon_{Newt,a}$	Absolute decrease tolerance for the norm of the nonlinear residual in the Newton algorithm
$\epsilon_{Newt,r}$	Relative decrease tolerance, from an initial evaluation, for the norm of the nonlinear residual in the Newton algorithm
η_{Newt}	Accuracy tolerance for the Newton algorithm
κ	Diffusion coefficient ($m^2.s^{-1}$)
$\underline{\delta}^m$	Unknown of the linear system at iteration m in a Newton algorithm
$\underline{\underline{J}}$	Jacobian of the nonlinear function in Newton's method
$\zeta = [\zeta_f]_{1\leq f\leq p}^T$	Vector containing the Roots of Legendre polynomial P_p
c	Advection speed ($m.s^{-1}$) or chord of the NACA0012 airfoil (m)
CFL, CFL_{SD}	Usual Courant-Friedrichs-Lewy number and its equivalent in SD
col	Variable used to refer to the columns of a matrix
d	Space dimension
h_s^i	s-th polynomial of Lagrange basis inside cell i build at solution point X_s^i
J_{iso}	Jacobian of the isoparametric transformation
$j_{Kry,max}$	Maximum number of Krylov iterations
l_f^i	f-th polynomial of Lagrange basis inside cell i build at solution point X_f^i
$L_2^{abs}(u_{num})$	Absolute numerical error using the 2-norm on vectors
$L_2^{rel}(u_{num})$	Relative numerical error using the 2-norm on vectors
$m_{Newt,max}$	Maximum number of iterations for the Newton algorithm
N_{cells}	Number of cells inside the mesh
n_{cons}	Number of conservative variables
N_{FP}	Number of flux points inside each mesh cell
N_{SP}	Number of solution points inside each mesh cell
p	Order of solution polynomial degree
P_n	Legendre polynomial of degree n
Q	Total number of Runge-Kutta stages
row	Variable used to refer to the rows of a matrix

S	Order of accuracy of the Runge-Kutta method
x^i	Starting abscissa (first abscissa on the left) of cell i
X_f^i	Abscissa of flux point f inside cell i
X_s^i	Abscissa of solution point s inside cell i
X_f	Vector containing the abscissas of all flux points
X_s	Vector containing the abscissas of all solution points
DoF	Degree of Freedom. In 1D-SD: DoF $= (p+1)\,N_{cells}$
i	Cell index
k	Cell indexes for the computing of the Jacobian of the residual
L	Left neighboring to an interface
m	Newton iterate in DIRK schemes
nb	Neighboring cells
q	Current stage of a Runge-Kutta (RK)
R	Right neighboring to an interface
$\bar{F}^i$	Reconstructed flux vector inside cell i at all flux points after the use of a Riemann solver at cell interfaces
$\bar{v}^i$	Reconstructed solution vector inside cell i at all flux points after the average process in 1D-diffusion
$\left(\frac{\partial \underline{R}}{\partial \underline{u}}\right)^n$	Jacobian of the residual taken at instant n
$\left(\frac{\partial R_s^i}{\partial u_j^i}\right)^n$	Matrix elements of $\left(\frac{\partial R^i}{\partial u^i}\right)^n$ taken at instant n where s is for the rows and j for the columns
$\left(\frac{\partial R_s^i}{\partial u_j^{nb}}\right)^n$	Matrix elements of $\left(\frac{\partial R^i}{\partial u^{nb}}\right)^n$ taken at instant n where s is for the rows and j for the columns
$\left(\frac{\partial R^i}{\partial u^i}\right)^n$	Local Jacobian of the residual of cell i with respect to u^i taken at instant n
$\left(\frac{\partial R^i}{\partial u^{nb}}\right)^n$	Local Jacobian of the residual of cell i with respect to u^{nb} taken at instant n
$\mathbf{M}_{Diff}^i$	Diagonal block matrix composed with M_{Diff}^{i-1}, M_{Diff}^i and M_{Diff}^{i+1}
$\underline{G}$	Function of the Newton algorithm applied to SD with implicit time-marching schemes
$\underline{R}$	Residual inside all the cells at all solution points: $\left[R^i\right]_{1\leq i\leq N_{cells}}^T$
$\underline{S}^q$	Sum $\sum_{j=1}^{q-1} a_{qj}\underline{R}^j$ in the DIRK methods.
$\underline{u}$	Solution inside all the cells at all solution points: $\left[u^i\right]_{1\leq i\leq N_{cells}}^T$
$\underline{u}^q$	Solution inside all the cells at all solution points at stage q of the RK scheme
$\underline{v}$	Solution inside all the cells at all flux points: $\left[v^i\right]_{1\leq i\leq N_{cells}}^T$
$\widetilde{\widetilde{u}}^i$	Vector containing the solution vectors u^{i-2}, u^{i-1}, u^i, u^{i+1} and u^{i+2}
$\widetilde{u}^i$	Vector containing the solution vectors u^{i-1}, u^i and u^{i+1}
$\widetilde{v}^i$	Vector containing the solution vectors $v_{N_{FP}}^{i-1}$, v^i, v_1^{i+1}
ξ, η, ζ	Spatial coordinates in the isoparametric domain
A	Average matrix for 1D-diffusion with centred scheme
D^i	Derivative matrix of cell i
E^i	Extrapolation matrix of cell i
F	Flux matrix for 1D-advection
F^i	Flux vector inside cell i at all flux points: $\left[F_f^i\right]_{1\leq f\leq N_{FP}}^T$
F_f^i	Flux vector inside cell i at flux point f
$F_{int}^{L,R}$	Flux at the interface flux point between cell L on its left and cell R on its right
M_{Adv}^i	Compact matrix for the discretization of 1D-advection with SD
M_{Diff}^i	Compact matrix for the discretization of $\left(\frac{\partial u}{\partial x}\right)^i$ with SD

R^i	Residual vector inside cell i at all solution points: $\left[R_s^i\right]_{1\leq s\leq N_{SP}}^T$
R_s^i	Residual vector inside cell i at solution point s
u^i	Solution vector inside cell i at all solution points: $\left[u_s^i\right]_{1\leq s\leq N_{SP}}^T$
u_s^i	Solution vector inside cell i at solution point s
u^{nb}	Solution vector inside the neighbors of cell i at all solution points: $\left[u_s^{nb}\right]_{1\leq s\leq N_{SP}}^T$
u_{ana}	Analytic solution vector for 1D-advection or 1D-diffusion
v^i	Solution vector inside cell i at all flux points: $\left[v_f^i\right]_{1\leq f\leq N_{FP}}^T$
v_f^i	Solution vector inside cell i at flux point f
$v_{int}^{L,R}$	Solution vector at the interface flux point between cell L on its left and cell R on its right
x, y, z	Spatial coordinates in the physical domain

Acknowledgements

I would like to thank all the people that have contributed to the success of my internship and who helped me during the writing of this report.

First of all, I thank profusely my supervisors Marc Montagnac, Laurent Muscat and Adèle Veilleux to have welcomed and trusted me for this internship. I am grateful for their help and all the advises they gave to me. By working with them I have learned a lot about numerical methods such as Newton's algorithm, Runge-Kutta schemes or also methods which deal with the inversion of a matrix for a linear system.

I show gratitude to Jean-François Boussuge for sharing its experience in aerodynamics and high-order methods and also for his advises on this report and on my presentation.

I thank warmly my four co-workmates, Jonathan, Nicolas G., Nicolas U. and later Clovis, who shared with me office I20, for the excellent atmosphere they kept up with special thoughts for Jonathan's daily joke and Nicolas G's drawings. I thank them for all the discussions we had on multiple topics and for their constant availability to help each other. I wish to Jonathan, Nicolas U. and Clovis, a good continuation at CERFACS and also to Nicolas G. for his PhD in Grenoble.

Finally, I really enjoyed the working ambiance at CERFACS during these six months. Thus, I also want to thank all the researchers, PhD students and interns that I rub shoulders with, especially the members of the CERFACS football team with whom I really took pleasure to play.

Abstract

The CERFACS is developing a high-order CFD code called JAGUAR. It solves the three-dimensional Navier-Stokes equations on unstructured hexahedral grids with the spectral differences method for the spatial discretization. However, only explicit time integration was available which was very restrictive in terms of time steps, to ensure stability, especially for viscous dominated flows. Thus, implicit time integration was considered in order to reduce the constraint on the time step and therefore the computation time. The implicit discretization in time leads to a nonlinear system of equations that has to be solved at each time iteration. The resolution of this system involves matrix-vector products with a Jacobian matrix that does not need to be computed explicitly for some kind of nonlinear solvers. One of them is the Jacobian-Free-Newton-Krylov (JFNK) method which was implemented in JAGUAR using the external library PETSc. Consequently, seven implicit time-marching schemes were tested and validated for CFL values higher than one on two classical two-dimensional test cases: the convected vortex and the NACA0012 airfoil.

Résumé

Le CERFACS développe actuellement un code CFD d'ordre élevé appelé JAGUAR. Il résoud les équations de Navier-Stokes en trois dimensions sur des maillages hexaédriques non structurés avec la méthode des différences spectrales pour la discrétisation spatiale. Avant mes travaux de stage, seule une intégration temporelle explicite était disponible. Ainsi, le pas de temps était limité par une condition de stabilité qui devenait très restrictive dans le cas d'écoulements visqueux. Par conséquent, une intégration temporelle implicite a été mise en place dans le but de réduire la contrainte sur le pas de temps et donc sur le temps de calcul. La discrétisation implicite en temps engendre un système d'équations non linéaires qui doit être résolu à chaque itération temporelle. La résolution de ce système fait intervenir des produits matrice-vecteur avec une matrice Jacobienne qui n'a pas besoin d'être calculée explicitement pour certaines techniques de résolution de systèmes linéaires ou non linéaires. L'une d'entre elles est la méthode dite "Jacobian-Free-Newton-Krylov (JFNK)" qui a été implémentée dans JAGUAR en utilisant la librairie externe PETSc. Par conséquent, sept schémas implicites en temps ont été développés et validés pour des valeurs de CFL supérieures à un sur deux cas test bi-dimensionnels: la convection d'un vortex et l'écoulement autour d'un profil d'aile NACA0012.

1 Introduction

All the scientific terms of this part will be detailed throughout the report.

1.1 Place and context of the internship

I did my internship at CERFACS, located at 42 Avenue Gaspard Coriolis on Météo France site in Toulouse.

1.1.1 Presentation of CERFACS

The CERFACS is the acronym for "Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique". It is a research organism working mainly on numerical simulations and modeling. CERFACS employs physicists, mathematicians and computer specialists to investigate issues of climate, high performance computing and aeronautics especially in combustion, acoustics and aerodynamics. It is financed by seven shareholders: Airbus, CNES, EDF, Météo France, ONERA, Safran and Total. CERFACS has also partnerships with CNRS, CEA and INRIA. It is composed of five teams:

- Computational Fluid Dynamics (CFD): it is the team where I worked and which constitutes the biggest number of employees.
- Aviation and environment (AE).
- Climate modeling and Global change (GLOBC).
- Parallel algorithm (ALGO).
- IT management and user support (CSG).

In 2018, there are 65 permanent members and 45 PhD and post-PhD students working at CERFACS. It also offers a lot of training and opportunities to attend conferences. For instance, in the CFD team, each Monday afternoon a PhD student has to present a research article and a useful software in front of the whole team.

1.1.2 Context of the internship

The CFD team at CERFACS is working on a new CFD solver which can make very accurate LES and DNS simulations thanks to a high-order spatial discretization on unstructured meshes. This solver, called JAGUAR (proJect of an Aerodynamic solver using General Unstructured grids And high-order schemes), is a *Fortran* code that uses the spectral differences (SD) method to discretize the 3D Navier-Stokes equations. The SD approach is based on a polynomial representation of the data inside each cell of the mesh. This solver is currently in a validation phase for aerodynamic applications such as the turbo engine blades or the jet noise.

1.1.3 Objective of the internship

The objective of this internship is to extend JAGUAR in order to simulate high Reynolds flows with very fine meshes close to the walls. Currently, an explicit time-marching approach is used and could lead to very small time steps making the number of time iterations excessively high to be feasible in the long term. To do so, implicit time-marching methods for SD have to be developed and implemented inside JAGUAR to reduce the constraint on the time step. These methods have to be validated on academic test cases and some comparisons of the iteration cost between explicit and implicit time-marching schemes have to be made.

1.1.4 Brief summary of my work

Firstly, I made my own 1D model code in order to understand well the SD method before going deeply into JAGUAR which is doing three-dimensional SD for unstructured hexahedral grids and compressible flows. Thus, I developed a *Python* code that solved 1D-advection and 1D-diffusion with SD for both explicit and implicit time-marching schemes. I started by thinking about the numerical method to compute the Jacobian matrix of the residual, rather than directly think on the JFNK method (see section (4.2)), because there was no papers about this method applied to SD. When I tried to implement the numerical computing of these Jacobians for 1D-advection and 1D-diffusion, I have started to think about how to get their analytic expressions to be able to compare them with the numerical computation. Once the numerical computing of Jacobians was validated, it was found to be very expensive in terms of CPU time. Then, I worked on the understanding of the JFNK method, in order to use and implement it with PETSc library in my *Python* code at first and finally in JAGUAR.

1.2 High-order CFD simulations

1.2.1 What is a high-order CFD simulation ?

Mathematically speaking, a numerical method is said to be of order k if the solution error e is proportional to the mesh size h to the power of k ($e \propto h^k$). According to a survey sent in 2007 to the members of the technical committee of the CFD Algorithm Discussion Group (CFDADG) and other researchers outside it, a high-order method is a method of third-order or higher [55]. For them, this is probably because many production codes used in aerospace community are first or second-order accurate.

1.2.2 Why doing high-order CFD ?

In the past two decades, high-order CFD methods have received considerable attention because of their potential in delivering higher accuracy with lower cost than low-order methods. In the review article of the 1st Workshop on High-Order CFD Methods [55], the authors have justified that high-order methods are not expensive compared to low-order ones regarding the CPU times to achieve the same level of accuracy. They also pointed out that high-order methods are needed for engineering purposes especially for vortex-dominated flows such as the flow over a helicopter (accurate resolution of unsteady vortices) or for computational aeroacoustic (CAA) where broadband acoustic waves need to propagate for a long distance without significant numerical dissipation or dispersion errors. Actually, they listed the main reasons why high-order methods are not used in the design process. Among them, the fact that they are more complicated than low-order methods is found or also the high memory requirement if implicit time stepping is employed. However, if these difficulties are overcome, high-order methods are very promising for the future of CFD.

1.2.3 High-order CFD techniques

At first glance, high-order space discretization could mean an increase of the stencil. Nevertheless, for large stencil schemes, the use of structured meshes is preferred to unstructured ones since there is not an unique way to extend the stencil to the neighboring cells for unstructured grids. The issue is that structured meshes are very long to generate for complex geometries. Thus, their use in an industrial context can be complicated since the mesh generation process may take several weeks for very complex industrial problems. Moreover, large stencils would mean the use of many neighboring cells but this is not relevant for High Performance Computing (HPC) because the data exchanges by the processors will highly increased.

Following those remarks, it seems that the way to avoid the use of large stencils is to define high-order techniques with a compact stencil. Thus, these methods have to increase the number of degrees of freedom (DoF) inside each mesh element and some of them follow the same process:

1. Define a high-order representation of the variables inside each mesh element using values at the DoF and a high-order interpolation procedure.
2. At cell boundaries, the reconstructed data are not equal and a Riemann solver is used to take into account these discontinuities.

Most of the methods proposed in the literature based on those ideas can be classified into three main groups:

- The Discontinuous Galerkin (DG) technique is based on the Finite Element (FE) framework. The principle is to look for a polynomial representation of the solution that satisfies a variational form of the governing

system within each element. Even if the technique is quite old (Reed and Hill in 1973 [38]), its extension to the full Navier-Stokes equations is recent and many papers have been published during the last 10 years.

- The Spectral Volume (SV) technique is based on the Finite Volume (FV) framework and it follows the pioneering work of Wang in 2002 [53][56]. It consists in defining element subdivisions on which a classical FV technique is considered. The mean quantity over each volume is necessary to build the high-order representation of data inside the element.
- The Spectral Difference (SD) technique follows the Finite Difference (FD) approach. Kopriva and Kolias published it in 1996 [24] and Liu, Vinokur and Wang [29] published a more general presentation of the technique in 2006. The idea is to define high-order approximation of the quantities but to solve the strong form of the equations, as in FD, inside each mesh cell.

As mentioned above, the interest of all these methods comes from the possibility to manage both the space refinement parameter h and the degree of the polynomial p. Indeed, when classical FV technique is compared with these high-order methods, the main difference lies in the non-universal relation between the mesh element and the number of DoF (one mesh element is not associated with one degree of freedom as in FV).

Note: These high-order methods are typically used with unstructured meshes because their generation process is much easier than for structured grids and takes no longer than a few hours even for complex geometries.

1.2.4 High-order methods used in Europe/US

High-order methods are quite recent compared to low-order ones and scientists from Europe and the United-States (US) have started to create groups to promote and develop these methods. In Europe, the Adaptive High-order Variational Methods for Aerodynamic Applications in Industry (ADIGMA) project [25], supported by a consortium consisting of 22 organizations including main European aircraft manufacturers, major European research establishments and several universities, all with well-proven expertise in CFD. In 2007, several authors of ADIGMA became members of the CFD Algorithm Discussion Group (CFDADG) in the American Institute of Aeronautics and Astronautics Fluid Dynamics Technical Committee (AIAA FDTC). In the first meeting of the CFDADG they tried to find some ways to overcome the difficulties mentioned in paragraph (1.2.1). They decided to focus on the mathematical explanations of high-order methods, error estimates and efficient time marching methods. Z. J. Wang's book, "Adaptive high-order methods for CFD" [54], co-written by a lot of ADIGMA and CFDADG members in 2011, gives the basis of high-order discretizations. The review paper of the 1st Workshop on High-Order CFD Methods [55], is also an excellent reference where comparisons between several high-order methods are done on many test cases.

Table 1.1 summed up some of the methods used by several members of CFDADG or ADIGMA to have an overview of where and which methods are more performed. It appears that DG method is the most popular compare to SV and SD. It also seems that implicit-time marching schemes are more and more used to break stability conditions and reduce time computation. Bassi and Munz use a Matrix-Free (MF) approach for solving nonlinear systems arising from implicit time discretization for DG. As it will be seen in paragraph (1.2.6), the objective of CERFACS is to use the same approach but for SD discretization because it could save time and reduce memory requirements.

Country	**City/Place**	**People**	**Code**	**Numerical method**
Italy	Bergame	F.Bassi	MIGALE	Implicit DG [7][8][20] (MF)
Germany	Stuggart	CD.Munz	Flexi/Fluxo	Implicit DG [9] (MF)
Germany	Cologne (DLR)	R.Hartmann and N.Kroll	PADGE	Implicit DG [22]
Belgium	Cenaero	K.Hillewaert	Argo	Implicit DG [44][45]
Belgium	Brussel	M.Parsani and KVd. Abeele	COOLFluiD	Implicit SD[51][35] and SV[36]
France	Bordeaux (INRIA)	R.Abgrall		DG
France	Chatillon (ONERA)	V.Couaillier	Aghora	Implicit DG [40](MF)
US	Ames	Y.Sun and Z. J. Wang		Implicit SD [47][29][57]

Table 1.1: Overview of high-order methods used in Europe/US

Note: H.Deconinck from the Von Karman Institute, also works on high-order methods with Cenaero. He has also developed in the code COOLFluiD which was, at start, a MPI parallel code with a second-order finite volume approach coupling with implicit time-marching schemes. It also solves the linear system using PETSc library with a GMRES technique.

1.2.5 The choice of the SD method by the CERFACS

At CERFACS, the SD method has been implemented six years ago in a new CFD code called JAGUAR. The choice of this approach was motivated by several reasons: initially the SD technique has been built in order to correct some drawbacks of DG and SV. Firstly, SD seems more efficient in term of CPU usage and less difficult to understand "physically" than the DG technique. Secondly, the SV method suffers from a high sensitivity (in term of stability of the method) with respect to element decomposition which is less present with SD method. Finally, the SD approach is very recent compared to DG then the potential of research is obviously greater.

Note: KVd.Abeele, C.Lacor and Z.J.Wang have shown the connections between SV and SD in 1D in this paper [50]. The conclusion is that SD is equivalent to SV in 1D like FV is equivalent to FD in 1D. This is due to the fact that the SD method is independent of the solution point positions in 1D.

1.2.6 State of the art in implicit time-marching for SD

As mentioned in section (1.2.3), the SD method is quite recent. It is a high-order conservative and efficient method developed by Liu, Vinokur and Wang for conservation laws on unstructured grids in 2006 [29]. Then, Liu, Wang, May and Jameson extended this method for the Euler equations [57] where they implemented limiters for discontinuity capturing and also for the Navier-Stokes equations [49]. They solved these equations using an explicit Runge-Kutta time integration scheme but, as said in paragraph (1.1.3), it suffered from slow convergence, especially for applications with walls, due to viscous terms.

Thus, Sun, Wang and Liu have developed an implicit lower-upper symmetric Gauss-Seidel (LU-SGS) algorithm for the compressible flow computation using SD [48][47]. They used a backward Euler scheme and they compared it to an explicit multi-stage Runge-Kutta scheme. They found that the speed-up went from one to two orders of magnitude when using an implicit time integration scheme which is quite promising for JAGUAR. They also described how they have constructed numerically the Jacobian matrix for the residual based on what was done by Brown and al. [11] and Qin and al. [37].

Then, still for solving the nonlinear algebraic systems arising from backward Euler scheme applied to SD, Van den Abeele, Parsani and Lacor implemented a Newton-Raphson method combined with a generalized minimum residual (GMRES) algorithm and compared it with the LU-SGS method of Sun and al. [51]. They found that, although the LU-SGS algorithm requires less memory than the GMRES algorithm, the latter reaches convergence faster for many cases.

More recently, in 2016, Moreira and al. described the previous Newton-Raphson algorithm coupled with GMRES to solve the two-dimensional Navier-Stokes equations on unstructured grids [31]. For the Jacobian of the residual, they used the same approach as Sun and al. to compute it numerically but they also tried to go through the structure of this Jacobian. Moreover, they did not build their own GMRES solver: they used the one from PETSc library [6].

However, the cost for computing numerically the Jacobian matrix is very expensive for the three-dimensional Navier-Stokes equations discretized with high-order spatial schemes where there are a lot of DoF. Some other techniques are investigated to reduce this cost. Because the Newton-Raphson method only needs the Jacobian matrix in terms of matrix-vector products, one could be tempted to solve the non linear systems by using a Jacobian-free Newton-Krylov (JFNK) method based on what is done for implicit DG. This would avoid to compute the Jacobian matrix because the result of the matrix-vector product between the Jacobian $\boldsymbol{J}(\boldsymbol{u}^m)$ and $\delta\boldsymbol{u}^m = \boldsymbol{u}^{m+1} - \boldsymbol{u}^m$ will be directly computed and put into the GMRES algorithm (see Knoll and Keyes [23]). Therefore, it seems that this approach will be faster and less memory-consuming than the other previous methods as they all compute the matrix numerically and then solve the system.

The PETSc library, already mentioned above, also provides such matrix-free methods and it is widely used in the CFD community. These are the reasons why the CERFACS wants to implement PETSc in code JAGUAR to do implicit time-marching schemes.

2 The Spectral Difference method

2.1 Notations and general principle using a 1D example

2.1.1 Solution points and time evolution process

To introduce the basics of the spectral difference method, let's consider a hyperbolic 1D-equation:

$$\frac{\partial u}{\partial t} + \frac{\partial F}{\partial x} = 0 \tag{2.1}$$

where $u = u(x,t)$ is the vector of conserved variables (also called the solution vector), $F = F(u)$ is the flux vector of u, t is the time and x represents the 1D-space. The size of u corresponds to the number of conserved variables noted n_{cons}. The discretization of the 1D-space is done on a 1D segment composed of N_{cells} cells where Eq. (2.1) is solved.

The principle of the SD method is to assume that vector u varies as a polynomial with a predefined degree p inside each cell of the given mesh. It means that for each cell $i \in [|1, N_{cells}|]$, u must have $p+1$ components (for each conserved variable) which are the values of u at $p+1$ points located inside this cell i. These points are called **solution points** and their number, noted N_{SP}, in 1D is:

$$N_{SP} = p + 1 \tag{2.2}$$

The solution is known at solution points and the numerical values at solution points help to define the polynomial representation of u. As for FD method, the solution is computed at any solution point s inside each cell i:

$$\frac{\partial u_s^i}{\partial t} + \left(\frac{\partial F}{\partial x}\right)_s^i = 0 \tag{2.3}$$

where u_s^i and $\left(\frac{\partial F}{\partial x}\right)_s^i$ are respectively the discrete solution vector and the 1D divergence of the flux taken at solution point s inside cell i. Then, following an explicit time marching process, the evolution of vector u is known once the derivative of the flux F is computed at each solution point.

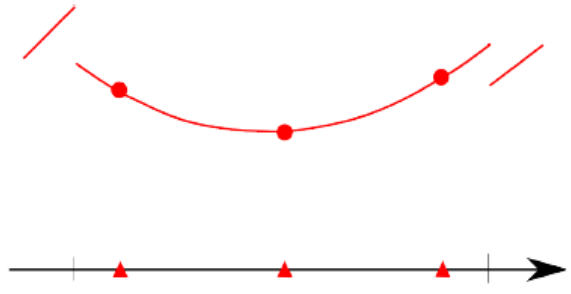
2.1.2 Flux points

With Eq. (2.1), since u is represented by a p-th order polynomial, the derivative of F is also a p-th order polynomial. Thus, the interpolation polynomial of the flux F must be of order $(p+1)$. Therefore, the flux polynomial must be defined on $p+2$ points that are called **flux points**. As for solution points, their total number in one cell, noted N_{FP}, is a function of p and in 1D:

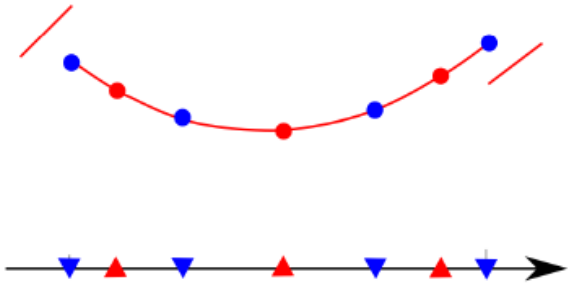
$$N_{FP} = p + 2 \tag{2.4}$$

2.1.3 Example with p = 2

Let's consider the example described in [27] to illustrate the discretization process. Let's say that in any cell i of the mesh, u has to be represented with a second-order polynomial: $p = 2$. The solution points, the values of u at these points and the second-order interpolation polynomial are represented in Figure 2.2a. The polynomial representation is by construction local to each cell: as Figure 2.2a shows, the interpolation polynomial is not the same in the neighbors of cell i. By nature, the SD method does not assume that polynomials are continuous at the interface between two cells: that is why a Riemann solver will be used later for the fluxes at interfaces.



(a) 1^{st} step: solution points (▲), u at solution points (●) and 2^{nd} order polynomial ($-$) inside the cell.

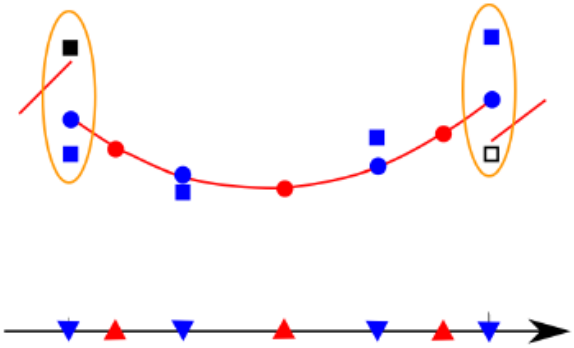


(b) 2^{nd} step: extrapolation of u (●) at flux points (▼).

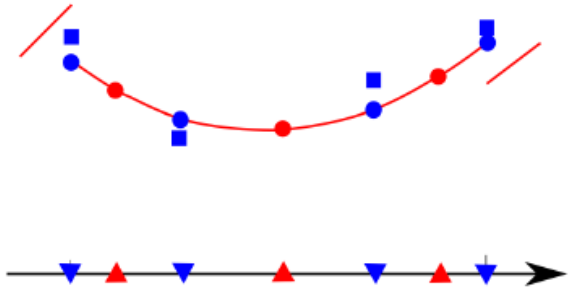
Figure 2.2: 1^{st} and 2^{nd} steps of the SD process in 1D for a hyperbolic equation

Once the second-order interpolation polynomial is built, the vector u is extrapolated at flux points as seen in Figure 2.2b. The flux points are located at the end points of any segment and other flux points are introduced between two consecutive solution points. In the case of $p = 2$, there are $N_{FP} = 4$ flux points: 2 at the cell boundaries and 2 strictly inside the cell.

At this point, the flux vector can be computed at flux points because it is a function of u at these points and its values are represented in Figure 2.3a. At the interface, the flux points are shared by two cells and since values are discontinuous at interfaces, the flux is not defined there without ambiguity. However, the flux at interfaces can be estimated by solving a Riemann problem. Then, by doing it at each cell interfaces it will be defined uniquely in all flux points inside the cell as seen in Figure 2.3b.



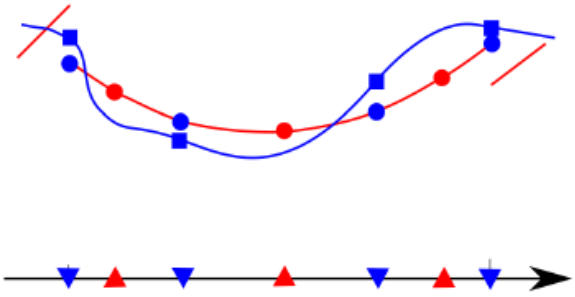
(a) 3^{rd} step: compute F (■) at flux points (▼).



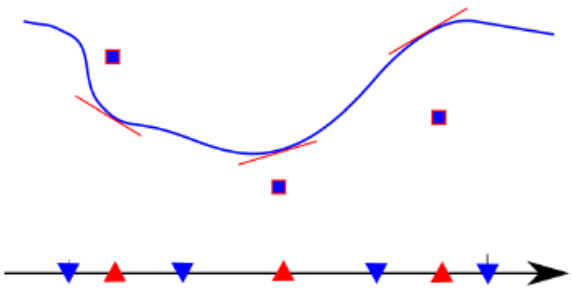
(b) 4^{th} step: unique flux (■) inside the cell after using a Riemann solver at cell interfaces.

Figure 2.3: 3^{rd} and 4^{th} steps of the SD process in 1D for a hyperbolic equation

From the values of F built at flux points, a $(p+1)$-th degree interpolation polynomial can be constructed and is represented in Figure 2.4a for the $p = 2$ case. This polynomial is globally continuous but can only be differentiated strictly inside each cell (not at cell interfaces). Then, it can be differentiated at solution points which are all by construction inside the cell. Therefore, the last step consists in differentiating the polynomial to compute the term $\left(\frac{\partial F}{\partial x}\right)_s^i$ of Eq. (2.3) as seen in Figure 2.4b.



(a) 5^{th} step: building of the flux polynomial ($-$) at flux points (3^{rd} order here).



(b) 6^{th} step: differentiation of the flux polynomial at solution points.

Figure 2.4: 5^{th} and 6^{th} steps of the SD process in 1D for a hyperbolic equation

2.1.4 Position of solution and flux points

Solution points

Inside a given cell i in 1D, the solution points are chosen to be the **Gauss points**. If x^i is the starting abscissa of cell i and $\Delta x^i = x^{i+1} - x^i$ the cell size, then the abscissa of solution point s inside cell i is given by:

$$X_s^i = x^i + \frac{1}{2}\left[1 - \cos\left(\frac{2s-1}{2N_{SP}}\pi\right)\right]\Delta x^i \quad \text{for} \quad 1 \leq s \leq N_{SP} \tag{2.5}$$

Then, for each cell i, a vector containing all the abscissas of solution points in this cell can be built. Repeating this process for all cells gives vector $X_s = \left[\left[X_s^i\right]_{1\leq s\leq N_{SP}}^T\right]_{1\leq i\leq N_{cells}}^T$ of size $N_{cells} \times N_{SP}$, containing the abscissas of all solution points in all cells.

Legendre flux points

In paragraph (2.1.2), it was said that the flux points are "located at the end points of any segment and other flux points are introduced between two consecutive solution points". Thus, still with x^i and x^{i+1} the end points of cell i, the first and the last flux points in this cell, noted respectively $X_{f,1}^i$ and $X_{f,N_{FP}}^i$, are set to:

$$X_{f,1}^i = x^i \quad \text{and} \quad X_{f,N_{FP}}^i = x^{i+1} \tag{2.6}$$

As explained in (2.1.2), in 1D, inside one cell, there are exactly $N_{FP} = p + 2$ flux points, so it remains p flux points to locate inside cell i. For Legendre flux points, the dimensionless abscissas of these p points are defined as the **roots of Legendre polynomial**. This is a serial of polynomial of increasing order, noted P_n with n its degree, defined by the following ordinary differential equation on $x \in [-1, 1]$:

$$\frac{d}{dx}\left(\left(1-x^2\right)P_n'(x)\right) + n(n+1)P_n(x) = 0 \quad \text{where} \quad P_n(1) = 1 \tag{2.7}$$

A lot of explicit expressions of $P_n(x)$ have been found (see [2]). Therefore, the values of its roots are also known for many values of n. For instance, A.N.Lowan, N.Davids and A.Levenson have determined their numerical values for each order n from 1 to 16 [30]. For the Legendre flux points, since p points remain to be defined, the roots of Legendre polynomial of degree p will be used. Once they have been found, they can be gathered in vector $\zeta = [\zeta_f]_{1\leq f\leq p}^T$ and the remaining p flux points can be computed with the following formula:

$$X_f^i = \frac{x^i + x^{i+1}}{2} + \frac{\zeta_f}{2}\Delta x^i \quad \text{for} \quad 2 \leq f \leq N_{FP} - 1 \tag{2.8}$$

As for solution points, repeating this process in all cells gives vector $X_f = \left[\left[X_f^i\right]_{1\leq f\leq N_{FP}}^T\right]_{1\leq i\leq N_{cells}}^T$ of size $N_{cells} \times N_{FP}$, containing the abscissas of all flux points in all cells.

Note: Here, the polynomial order is assumed to be the same inside all the mesh cells. That is why, there is no index i for the components of vector ζ.

Gauss-Lobatto flux points

There is another choice to determine X_f^i inside a cell. The **Gauss-Lobatto points** can be used and in this case, the end points $X_{f,1}^i$ and $X_{f,N_{FP}}^i$ are directly included inside the formula:

$$X_f^i = x^i + \frac{1}{2}\left[1 - \cos\left(\frac{f-1}{N_{FP}-1}\pi\right)\right]\Delta x^i \quad \text{for} \quad 1 \leq f \leq N_{FP} \tag{2.9}$$

2.1.5 Basis of polynomial used

All the interpolation process uses the Lagrange interpolation principle. Using the values of u at N_{SP} solution points inside cell i, a p-degree polynomial can be built using the following Lagrange basis:

$$h_s^i(X) = \prod_{k=1,k\neq s}^{N_{SP}}\left(\frac{X - X_k^i}{X_s^i - X_k^i}\right) \quad \text{for} \quad 1 \leq s \leq N_{SP} \tag{2.10}$$

h_s^i is then the s-th polynomial of Lagrange basis build at solution point X_s^i using all the other solution points X_k^i.

Similarly, using the values of F at N_{FP} flux points, a $(p+1)$-degree polynomial can be built using the following Lagrange basis:

$$l_f^i(X) = \prod_{k=1,k\neq f}^{N_{FP}} \left(\frac{X - X_k^i}{X_f^i - X_k^i} \right) \quad \text{for} \quad 1 \leq f \leq N_{FP} \tag{2.11}$$

Some attention should be put here, X_k^i are now all the other flux points different of X_f^i and not the solution points. Actually, in the SD process, the derivative of expression (2.11) is used when the flux is differentiated at solution points. That is why its analytic formula is recalled in Eq. (2.12):

$$l_f'^i(X) = \frac{\sum_{k=1,k\neq f}^{N_{FP}} \left[\prod_{m=1,m\neq k}^{N_{FP}} \left(X - X_m^i\right) \right]}{\prod_{k=1,k\neq f}^{N_{FP}} \left(X_f^i - X_k^i\right)} \quad \text{for} \quad 1 \leq f \leq N_{FP} \tag{2.12}$$

2.1.6 Global algorithm for a hyperbolic 1D-equation

The unsteady update of the solution vector described in section (2.1) can be summed up in the following steps:

Algorithm 1 General algorithm for solving a hyperbolic 1D-equation with SD

1: Define solution and flux points in each cell of the mesh and initialize u at solution points.
2: **for** iter from 1 to N_{iter} **do**
3: Extrapolate u at flux points. The solution at one flux point f inside cell i is given by:

$$v_f^i = \sum_{s=1}^{N_{SP}} u_s^i h_s^i \left(X_f^i\right) \quad \text{for} \quad 1 \leq f \leq N_{FP} \tag{2.13}$$

4: Compute the internal fluxes in each cell i at each flux point f noted F_f^i with the values of v_f^i.
5: Use a Riemann solver at each interface of each cell to have unique flux values. This step gives $\bar{F}_f^i$
6: Differentiate the flux polynomial at solution points:

$$\left(\frac{\partial \bar{F}}{\partial x}\right)_s^i = \sum_{f=1}^{N_{FP}} \bar{F}_f^i l_f'^i \left(X_s^i\right) \quad \text{for} \quad 1 \leq s \leq N_{SP} \tag{2.14}$$

7: Update the solution using a time integration scheme.
8: **end for**

where iter is the current time iteration, N_{iter} is the total number of time iterations and v_f^i is the solution vector inside cell i at flux point f.

2.1.7 Diffusion scheme

Here, one of the method used in SD to compute diffusive flux, noted F_D, is explained. The main difference between diffusive and convective fluxes is that F_D is also a function of ∇u. Then, the algorithm for computing $F_D(u, \nabla u)$ is different from the one of the convective flux. In the literature, many approaches can be found, see [51][31][27]. One of the method applied to the 1D-diffusion equation ($n_{cons} = 1$) is presented here:

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2} \tag{2.15}$$

where κ stands for the diffusion coefficient and the discretization of the term $\frac{\partial^2 u}{\partial x^2}$ is done by considering a centred scheme. Without going into details, because the process is already detailed in paragraph (7.2.1), the main steps for computing it may be clarified as followed:

Algorithm 2 General algorithm for solving 1D-diffusion with SD using centred scheme

1: Define solution and flux points in each cell of the mesh and initialize u at solution points.
2: **for** iter from 1 to N_{iter} **do**
3: Extrapolate u at flux points. The solution at one flux point f inside cell i is given by:

$$v_f^i = \sum_{s=1}^{N_{SP}} u_s^i h_s^i \left(X_f^i\right) \quad \text{for} \quad 1 \leq f \leq N_{FP} \tag{2.16}$$

4: At cell interfaces, compute the average of left and right values:

$$v_{int}^{L,R} = \frac{v_{N_{FP}}^L + v_1^R}{2} \tag{2.17}$$

5: Compute $\bar{v}_f^i$ by replacing interface values by their average interface values.
6: Differentiate $\bar{v}_f^i$ at solution points to obtain:

$$\left(\frac{\partial u}{\partial x}\right)_s^i = \sum_{f=1}^{N_{FP}} \bar{v}_f^i l_f'^i \left(X_s^i\right) \quad \text{for} \quad 1 \leq s \leq N_{SP} \tag{2.18}$$

7: Extrapolate $\left(\frac{\partial u}{\partial x}\right)$ at flux points to have:

$$\left(\frac{\partial v}{\partial x}\right)_f^i = \sum_{s=1}^{N_{SP}} \left(\frac{\partial u}{\partial x}\right)_s^i h_s^i \left(X_f^i\right) \quad \text{for} \quad 1 \leq f \leq N_{FP} \tag{2.19}$$

8: At cell interfaces, compute the average of left and right values:

$$\left(\frac{\partial v}{\partial x}\right)_{int}^{L,R} = \frac{\left(\frac{\partial v}{\partial x}\right)_{N_{FP}}^L + \left(\frac{\partial v}{\partial x}\right)_1^R}{2} \tag{2.20}$$

9: Compute $\left(\overline{\frac{\partial v}{\partial x}}\right)_f^i$ by replacing interface values by their average interface values.
10: Differentiate $\left(\overline{\frac{\partial v}{\partial x}}\right)_f^i$ at solution points to obtain:

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_s^i = \sum_{f=1}^{N_{FP}} \left(\overline{\frac{\partial v}{\partial x}}\right)_f^i l_f'^i \left(X_s^i\right) \quad \text{for} \quad 1 \leq s \leq N_{SP} \tag{2.21}$$

11: Update the solution using a time integration scheme.
12: **end for**

where $v_{int}^{L,R}$ and $\left(\frac{\partial v}{\partial x}\right)_{int}^{L,R}$ are respectively the solution vector and its derivative at the interface flux point between cell L on its left and cell R on its right.

2.2 Extension to 2D and 3D

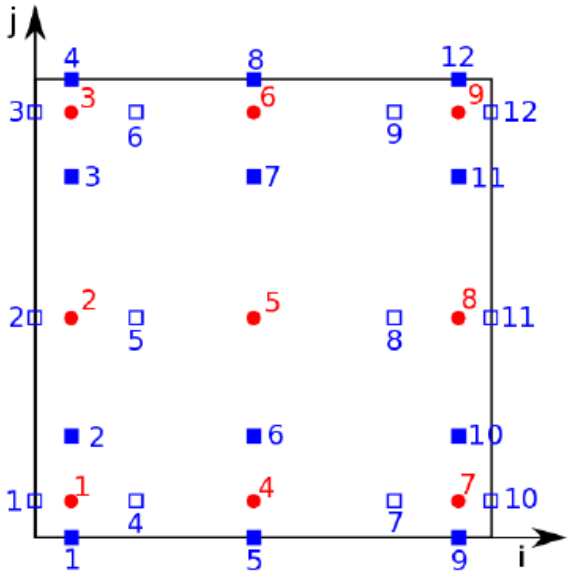
In this section, the extension of the SD method to flows in two and three dimensions is presented. The position of solution and flux points are given on a 2D-example and the isoparametric transformation used to define these positions for any cell is then described. This transformation will be explained for hexahedral cells and not for tetrahedral cells since JAGUAR is currently dealing with hexahedral cells only.

2.2.1 Position of solution and flux points

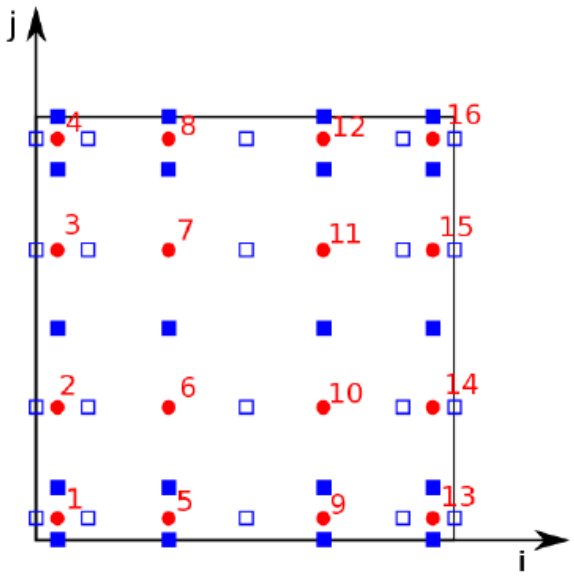
For 2D and 3D, the solution and flux points are put direction per direction by repeating the same 1D process described in section (2.1). Thus, the solution points are still the Gauss points, placed in each direction, and flux points can be either Legendre or Gauss-Lobatto flux points also placed in each direction. Finally, if d stands for the space dimension, inside one cell, the number of solution and flux points are respectively:

$$N_{SP} = (p+1)^d \quad \text{and} \quad N_{FP} = d \times (p+2)(p+1)^{d-1} \tag{2.22}$$

A 2D-example of the positions of solution and flux points for $p = 2$ and $p = 3$ cases is presented in Figure 2.5:



(a) Solution points and flux points in 2D inside one cell for $p = 2$ case.



(b) Solution points and flux points in 2D inside one cell for $p = 3$ case.

Figure 2.5: Positions of solution points (•) and flux points ((■) for i-direction and (□) for j-direction) in 2D inside one cell for $p = 2$ and $p = 3$. Figures from [27].

2.2.2 Isoparametric transformation

For 2D and 3D meshes, cells can have different shapes and different volume sizes. In this case, it can be difficult to place solution and flux points in the same locations inside all cells. For 1D cells it was possible because all cells are always segments. However, for 2D and 3D, an isoparametric transformation is considered to transform all mesh cells from the physical domain (x, y, z) into a standard cubic element $(\xi, \eta, \zeta) \in [0, 1]^3$. Usually, this transformation is written mathematically for a transformation from the isoparametric space to the physical space [27][47]:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \sum_{i=1}^{K} M_i(\xi, \eta, \zeta) \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix} \tag{2.23}$$

where K is the number of points used to define the physical element (for instance $K = 4$ for a tetrahedral element or $K = 8$ for a hexahedral element), (x_i, y_i, z_i) are the Cartesian coordinates of these points and $M_i(\xi, \eta, \zeta)$ are the shape functions. An example of shape function computation in 2D is given here [21].

For the transformation given in (2.23), the Jacobian matrix takes the following form:

$$J_{iso} = \frac{\partial(x, y, z)}{\partial(\xi, \eta, \zeta)} = \begin{pmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} & \frac{\partial x}{\partial \zeta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} & \frac{\partial y}{\partial \zeta} \\ \frac{\partial z}{\partial \xi} & \frac{\partial z}{\partial \eta} & \frac{\partial z}{\partial \zeta} \end{pmatrix} \tag{2.24}$$

Finally, if the transformation is non-singular ($|J_{iso}| \neq 0$), its inverse transformation (from physical space to isoparametric space) also exists and J_{iso}^{-1} is given by:

$$J_{iso}^{-1} = \frac{\partial(\xi, \eta, \zeta)}{\partial(x, y, z)} = \begin{pmatrix} \frac{\partial \xi}{\partial x} & \frac{\partial \xi}{\partial y} & \frac{\partial \xi}{\partial z} \\ \frac{\partial \eta}{\partial x} & \frac{\partial \eta}{\partial y} & \frac{\partial \eta}{\partial z} \\ \frac{\partial \zeta}{\partial x} & \frac{\partial \zeta}{\partial y} & \frac{\partial \zeta}{\partial z} \end{pmatrix} \tag{2.25}$$

2.2.3 Application to unsteady 3D equations written in conservative form

Consider unsteady 3D equations written in conservative form (such as unsteady compressible 3D Navier-Stokes equations):

$$\frac{\partial U}{\partial t} + \frac{\partial F}{\partial x} + \frac{\partial G}{\partial y} + \frac{\partial H}{\partial z} = 0 \tag{2.26}$$

where U is the vector of conserved variables and F, G, H are respectively the total fluxes in directions x, y and z. Eq. (2.26) in the physical domain is transformed into the isoparametric domain to obtain [47]:

$$\frac{\partial \widetilde{U}}{\partial t} + \frac{\partial \widetilde{F}}{\partial x} + \frac{\partial \widetilde{G}}{\partial y} + \frac{\partial \widetilde{H}}{\partial z} = 0 \tag{2.27}$$

with

$$\tilde{U} = |J_{iso}|.U, \quad \begin{bmatrix} \widetilde{F} \\ \widetilde{G} \\ \widetilde{H} \end{bmatrix} = |J_{iso}| J_{iso}^{-1}. \begin{bmatrix} F \\ G \\ H \end{bmatrix} \tag{2.28}$$

At each time step, the space discretization is done in the isoparametric space and then Eq. (2.28) is used to update the solution in the physical space. For more precisions and because the purpose of the internship was not to focus on spatial discretization, the discretization process for the unsteady compressible 3D Navier-Stokes equations is completely described in [47].

2.3 Notion of residual

The discretization process described in sections (2.1) and (2.2) can be gathered in one single term called the residual. In this case, for a given cell i, any conservation law discretized by the SD method can be put into the semi-discrete formulation:

$$\frac{du^i}{dt} = R^i\left(u^i, u^{nb}\right) \tag{2.29}$$

where $R^i\left(u^i, u^{nb}\right)$ is the "residual". In the case of Eq. (2.1) it corresponds to the discretization of the term $-\frac{\partial F}{\partial x}$ inside cell i. It obviously depends on u^i but also on u^{nb} ("nb" for the neighboring cells of cell i) because of the flux interpolation at the interface between cell i and its neighbors. For instance, for the 1D-advection, the neighbors where a change of u will change R^i are only the cells $i-1$ and $i+1$: $R^i\left(u^i, u^{nb}\right) = R^i\left(u^{i-1}, u^i, u^{i+1}\right)$. This will be shown in paragraphs (6.2.1) and (6.2.2). About the size of these vectors, since cell i has N_{SP} solution points, R^i and u^i are both vectors of size $n_{cons} \times N_{SP}$. They have the following structure: $R^i = \left(R_1^i, ..., R_{N_{SP}}^i\right)^T = \left[R_s^i\right]_{1 \leq s \leq N_{SP}}^T$ and $u^i = \left(u_1^i, ..., u_{N_{SP}}^i\right)^T = \left[u_s^i\right]_{1 \leq s \leq N_{SP}}^T$ where R_s^i and u_s^i are both of size n_{cons}.

Once R^i is known inside all cells, Eq. (2.29) can be replaced by the vectorial semi-discrete equation valid in all the spatial domain:

$$\frac{d\underline{u}}{dt} = \underline{R}\left(\underline{u}\right) \tag{2.30}$$

where $\underline{u}$ and $\underline{R}$ are two vectors of size $N_{cells} \times n_{cons} \times N_{SP}$ with the following structure:

$$\underline{u} = \left(u^1, ..., u^{N_{cells}}\right)^T = \left(\left(u_1^1, ..., u_{N_{SP}}^1\right), ..., \left(u_1^{N_{cells}}, ..., u_{N_{SP}}^{N_{cells}}\right)\right)^T \tag{2.31}$$

$$\underline{R} = \left(R^1, ..., R^{N_{cells}}\right)^T = \left(\left(R_1^1, ..., R_{N_{SP}}^1\right), ..., \left(R_1^{N_{cells}}, ..., R_{N_{SP}}^{N_{cells}}\right)\right)^T \tag{2.32}$$

Note: In Eqs. (2.29) and (2.30), the time derivative has become a total derivative since u^i and $\underline{u}$ depend only on time after the spatial discretization by the SD method.

3 Time-marching schemes

High-order spatial methods need to be associated with high-order temporal schemes otherwise the accuracy won in space is then lost with time discretization. One way to increase the time accuracy is to use Runge-Kutta (RK) schemes which was the choice made for JAGUAR.

3.1 Runge-Kutta schemes

3.1.1 Characteristics of a RK scheme

A Runge-Kutta scheme has four main characteristics which are:

1. Its order of accuracy, noted S here.
2. Its number of stages, noted Q here.
3. Its stability: A-stable [14] or L-stable [1]. L-stability is a special case of A-stability which is better for solving nonlinear stiff equations.
4. Its type of integration: explicit or implicit.

Very often, if "nameRK" is the name of the Runge-Kutta method, this method will be noted "nameRK(Q,S)" which allows the reader to directly know the order of accuracy and the number of stages.

3.1.2 General algorithm

The RK integration process is the sum of two tasks:

- Task 1: The Q stage values $\underline{u}^q$ are computed using a "stage equation":

$$\underline{u}^q = \underline{u}^n + \Delta t \sum_{j=1}^{Q} a_{qj} \underline{R}^j \quad \text{for} \quad 1 \leq q \leq Q \tag{3.33}$$

- Task 2: The solution at time t^{n+1} is computed using an "update equation":

$$\underline{u}^{n+1} = \underline{u}^n + \Delta t \sum_{q=1}^{Q} b_q \underline{R}^q \tag{3.34}$$

where $\underline{R}^j = \underline{R}\left(\underline{u}^j\right)$, the matrix $\boldsymbol{A} = [a_{qj}]_{1 \leq q,j \leq Q}$ and the vector $\boldsymbol{b} = [b_q]^T_{1 \leq q \leq Q}$ are given by the RK method. They are typically gathered in a "Butcher's Tableau" (see appendix A). These coefficients are found by solving a system of equations which depends on the order of accuracy and also on dissipation and dispersion properties that are investigated to obtain a given scheme. An example of such system can be seen in [32].

Note: Since $\underline{R}$ does not depend on time, the vector $\boldsymbol{c}$ in the RK method is not used.

3.2 Explicit Runge-Kutta schemes

3.2.1 Consequence on the stage equation

For explicit RK schemes, $a_{qj} = 0$ for $j \geq q$ meaning that the matrix $\boldsymbol{A}$ is a strictly lower triangular matrix. Thus, each stage $\underline{u}^q$ is "explicitly" known thanks to the values of the previous stages (which is not the case for

implicit RK schemes as it will be seen later) and Eq. (3.33) can be recast as:

$$\underline{u}^q = \underline{u}^n + \Delta t \sum_{j=1}^{q-1} a_{qj} \underline{R}^j \quad \text{for} \quad 1 \leq q \leq Q \tag{3.35}$$

Explicit Runge-Kutta schemes are typically named ERK(Q,S). Below are the ERK schemes used in JAGUAR and in my model code:

- In JAGUAR: ERK(2,2), ERK(4,4), ERK(4,4) low-storage, ERK(6,2)LDLD, ERK(6,4)SD.
- In my model code: forward Euler scheme, ERK(2,2) and ERK(4,4).

Butcher's tableau for forward Euler, ERK(2,2) and ERK(4,4) methods can be found in [3]. ERK(4,4) low-storage, ERK(6,2)LDLD and ERK(6,4)SD are low-storage RK schemes. These are special cases of RK methods where their coefficients are such that each $\underline{u}^q$ can be computed with the values of $\underline{R}^{q-1}$ only. That is why they are called low-storage because they do not need to store all the $\underline{R}^j$, $j \in [|1, q-1|]$, vectors. Actually, the solution is advanced directly at each stage and the final stage corresponds to the solution at instant $n+1$: $\underline{u}^Q = \underline{u}^{n+1}$.

Note 1: ERK(6,2)LDLD corresponds to the Explicit Runge-Kutta of second-order and six stages with low-dissipation and low-dispersion designed by Bogey and Bailly for aeroacoustic applications [10]. More precision about low-storage RK can be found in [46] especially if the reader is interested on knowing how Butcher's coefficients can be manipulated to obtain only one coefficient γ_q to update the solution at each stage. It should be mentioned that all the RK processes cannot be changed into low-storage RK processes [13][58].

Note 2: ERK(6,4)SD is an Explicit Runge-Kutta of fourth-order with six stages designed for SD by J.Vanharen, a former CERFACS PhD, where the last two coefficients, γ_5 and γ_6, depend on p. This scheme was presented during the 23rd AIAA CFD conference at Denver in June 7th 2017 [4].

3.2.2 Limitations due to a stability criterion

Similarly to other numerical methods using explicit time integration schemes, the SD method is stable under a CFL condition. This condition depends on the time integration scheme used and also on the value of p such as in DG where the stability condition is [15][19]:

$$CFL = \frac{c\Delta t}{\Delta x} \leq \frac{1}{2p+1} \tag{3.36}$$

For SD, this stability limit is often too restrictive and for high values of p the time step becomes very small closing the doors for an industrial use. This is why implicit methods for SD are currently studied in order to break this stability condition. More precision about CFL number in SD and its stability can be found in appendix B.

3.3 Implicit Runge-Kutta schemes

As it was underlined in (3.2.2) and (1.2.6), implicit time marching schemes have to be developed for SD to increase time steps and implicit Runge-Kutta (IRK) schemes can be considered. These IRK schemes can be split into two categories: fully IRK schemes and Diagonally Implicit Runge-Kutta (DIRK) schemes.

3.3.1 Fully IRK schemes

This kind of RK schemes appears when all the elements of matrix $\boldsymbol{A}$ are non-zero. In this case, to determine each solution at stage q, given by Eq. (3.33), the following nonlinear system has to be solved [18]:

$$\begin{pmatrix} \underline{u}^1 - \underline{u}^n \\ \vdots \\ \underline{u}^Q - \underline{u}^n \end{pmatrix} = \boldsymbol{A} \begin{pmatrix} \underline{R}^1 \\ \vdots \\ \underline{R}^Q \end{pmatrix} \tag{3.37}$$

Because $\underline{u}$ and $\underline{R}$ are of size $n_{cons} \times N$, where $N = N_{cells} \times N_{SP}$, the system given by Eq. (3.37) is of size: $Q \times n_{cons} \times N$. Some examples of such schemes can be found here [18] such as the RadauIIA of order 3 and 5 or Lobatto methods [3]. They can have an order that is greater than their number of stages which is quite interesting. However, because of the size of the system to solve, they are less used than DIRK methods and they were not considered during my internship.

3.3.2 DIRK schemes

General case

DIRK schemes are Implicit Runge-Kutta schemes where $a_{qj} = 0$ for $j > q$, meaning that the matrix $\boldsymbol{A}$ is a lower triangular matrix [18]. In this case, the stage equation becomes:

$$\boxed{\underline{u}^q = \underline{u}^n + \Delta t a_{qq} \underline{R}^q + \Delta t \sum_{j=1}^{q-1} a_{qj} \underline{R}^j \quad \text{for} \quad 1 \leq q \leq Q} \tag{3.38}$$

Eq. (3.38) is a nonlinear and implicit system of equations since $\underline{R}^q$ is a nonlinear operator and is needed to find $\underline{u}^q$. Thus, the main idea is to solve successively the Q stages by considering the nonlinear system of size $n_{cons} \times N$, given by Eq. (3.38), at each stage. Then, for each $\underline{u}^q$ found, $\underline{R}^q$ can be computed and stored. Finally, Eq. (3.34) is used to update $\underline{u}$ at time t^{n+1}.

Note: Each $\underline{R}^q$, for $q \in [|1, Q|]$, has to be stored for the computation of the sum $\underline{S}^q = \sum_{j=1}^{q-1} a_{qj} \underline{R}^j$ that must be done at each stage q because a_{qj} depends on q.

SDIRK schemes

A special case of DIRK schemes is SDIRK schemes for Singly Diagonally Implicit Runge-Kutta. For such schemes, a_{qq} is equal to a constant $\forall q \in [|1, Q|]$. This hypothesis reduces the number of unknowns to determine for building the Runge-Kutta scheme. However, only for special cases, the order achieved by one single stage is limited to one. It means that most of SDIRK(Q,S) methods are of order $S \leq Q$. Actually, Crouzeix [16] determined all the SDIRK(2,3) and SDIRK(3,4) methods which are A-stable. Their Butcher's table can be found in appendix A.

3.3.3 IRK schemes in JAGUAR

During my internship, the following seven IRK schemes were implemented and tested in code JAGUAR: implicit midpoint, SDIRK(2,2), SDIRK(2,3), SDIRK(3,3), DIRK(3,3), SDIRK(3,4) and ILDDRK(3,4). The last one is a low-dispersion and low-dissipation DIRK scheme. All of their Butcher's tableau can be found in appendix A.

3.3.4 Conclusion on implicit schemes

With implicit schemes, time steps will be higher than those used with explicit schemes. However, it is necessary to bear in mind that doing implicit schemes also has a cost since the nonlinear system of Eq. (3.37) for full IRK schemes or the Q nonlinear systems given by Eq. (3.38) for DIRK schemes, have to be solved at each time iteration. Thus, by doing implicit time-marching schemes, less time iterations are done but each iteration is more costly compared to explicit time-marching schemes. Therefore, it is important to compare the iteration cost between explicit and implicit time-marching schemes.

4 Solver for the nonlinear system of equations

The stage equation for DIRK schemes (i.e Eq. (3.38)) is the nonlinear system of equations that has to be solved if DIRK schemes are considered for JAGUAR. Thus, nonlinear solvers have to be use and the choice was made on Newton solvers. That is why, the basic concepts of Newton algorithms are recalled in this chapter.

4.1 Inexact Newton method

4.1.1 Newton methods

Let's consider the general form of a nonlinear system of equations:

$$\underline{F}\left(\underline{X}\right) = \underline{0} \tag{4.39}$$

where $\underline{F}\left(\underline{X}\right)$ is the vector-valued function of nonlinear residuals and $\underline{X}$ is the state vector to be found.
The Newton iteration for Eq. (4.39) derives from a multivariate Taylor expansion about a current point $\underline{X}^m$:

$$\underline{F}\left(\underline{X}^{m+1}\right) = \underline{F}\left(\underline{X}^m\right) + \underline{F}'\left(\underline{X}^m\right)\left(\underline{X}^{m+1} - \underline{X}^m\right) + \text{higher-order terms} \tag{4.40}$$

Setting Eq. (4.40) to zero and neglecting the terms of higher-order curvature leads to iterations over a sequence of linear systems [23]:

$$\underline{\underline{J}}\left(\underline{X}^m\right)\underline{\delta}^m = -\underline{F}\left(\underline{X}^m\right) \quad \text{for} \quad m = 0, 1, ... \tag{4.41}$$

given a $\underline{X}^0$. Here, $\underline{\underline{J}} \equiv \underline{F}'$ is the associated Jacobian matrix of $\underline{F}$, m is the nonlinear iteration index and $\underline{\delta}^m = \left(\underline{X}^{m+1} - \underline{X}^m\right)$ is the unknown of the linear system.
In the general case, Newton iterations are stopped when one of the following inequality is satisfied:

$$\frac{||\underline{F}\left(\underline{X}^m\right)||}{||\underline{F}\left(\underline{X}^0\right)||} \leq \epsilon_{Newt,r} \quad \text{or} \quad ||\underline{F}\left(\underline{X}^m\right)|| \leq \epsilon_{Newt,a} \quad \text{or} \quad \frac{||\left(\underline{X}^{m+1} - \underline{X}^m\right)||}{||\underline{X}^m||} \leq \eta_{Newt} \tag{4.42}$$

where $\epsilon_{Newt,r}$ is the relative decrease tolerance for the norm of $\underline{F}$ from an initial norm evaluation, $\epsilon_{Newt,a}$ is the absolute tolerance for the norm of $\underline{F}$ and η_{Newt} is the convergence tolerance for the difference between two successive Newton iterate. To avoid an infinite loop if none of these criterion is reached, a last stopping criterion is defined to stop the computation when $m \geq m_{Newt,max}$ where $m_{Newt,max}$ is the maximum number of iterations that are allowed for the Newton algorithm.

4.1.2 Inexact Newton

Linear systems defined by Eq. (4.41) can be solved with either a direct or an iterative method. As the other high-order methods, the SD approach has a high number of DoF which entails that Eq. (4.41) is a large linear problem. However, for such problems, direct methods are too slow and iterative methods are preferred. It can be seen as inexact Newton methods [17] because the solution of Eq. (4.41) will be an approximation of the exact solution that could have been found with a direct method.

Among the iterative methods, Krylov methods and mainly the GMRES algorithm will be used.

4.2 The Jacobian-Free-Newton-Krylov method

The Jacobian-Free-Newton-Krylov (JFNK) method is an inexact Newton method that solved Eq. (4.41) using a Krylov method and without computing explicitly $\underline{\underline{J}}(\underline{X}^m)$. In this section, a brief description of the Krylov methods and GMRES is done. Then, the Jacobian-Free approach is presented along with its use in a Newton algorithm.

4.2.1 Krylov methods

Krylov methods are approaches for solving large linear systems introduced as iterative methods in 1971 [39]. They are projection methods for solving $\underline{\underline{A}}.\underline{x} = \underline{b}$ using the Krylov subspace $\widehat{K}_j$ defined as [42]:

$$\widehat{K}_j = \text{span}\left(\underline{r}_0, \underline{\underline{A}}.\underline{r}_0, \underline{\underline{A}}^2.\underline{r}_0, ..., \underline{\underline{A}}^{j-1}.\underline{r}_0\right) \tag{4.43}$$

where $\underline{r}_0 = \underline{b} - \underline{\underline{A}}.\underline{x}_0$ with $\underline{x}_0$ an initial guess.
These methods require only matrix-vector products to carry out the linear iteration (not the individual elements of $\underline{\underline{A}}$). Among the wide variety of Krylov methods, it turns out that GMRES algorithm developed by Saad and Schultz [43] is the best choice for the JFNK method [23].
The Generalized Minimal RESidual (GMRES) method is an Arnoldi-based method. In GMRES, the Arnoldi basis vectors form the trial subspace out of which the solution is constructed. A major beneficial feature of the algorithm is that only one matrix-vector product is required per iteration to create each new trial vector and the iterations are terminated on a by-product estimate of the residual that does not require explicit construction of intermediate residual vectors or solutions. However, GMRES has a residual minimization property in the Euclidean norm which requires the storage of all previous Arnoldi basis vectors. That is why the restart version [43], noted GMRES(m_{Res}) where m_{Res} is some fixed integer, is often used to restart GMRES algorithm at every m_{Res} steps. This resulting pressure on memory has put an increased emphasis on quality preconditioning believing that it is only through effective preconditioning that JFNK is feasible on large-scale problems.

4.2.2 Jacobian-Free approach

The Jacobian-Free approach is the fact to estimate the matrix-vector product between a Jacobian matrix $\underline{\underline{J}}$ times a vector $\underline{v}$ with the following formula [11]:

$$\underline{\underline{J}}(\underline{X}^m)\,\underline{v} \approx \frac{\underline{F}(\underline{X}^m + \delta\underline{v}) - \underline{F}(\underline{X}^m)}{\delta} \tag{4.44}$$

where δ is a scalar. This approach is perfectly adapted for an use with Newton's method because the left-hand side of Eq. (4.41) can be computed using formula (4.44) with $\underline{v} = \underline{\delta}^m$. Then, the resulting vector can be put into GMRES algorithm which will solve the system without forming any Jacobian matrix.

Note 1: In this approximation the error is proportional to δ. Thus, the value of δ is very important and some discussions on the various options for choosing it can be found here [23]. What is usually done is to take $\delta = \frac{\delta_0}{||\underline{v}||_2}$ with δ_0 closed to the square root of the machine precision ($\delta_0 \approx 10^{-7}$).

Note 2: In [23], there is also a demonstration of formula (4.44) in the case of two coupled nonlinear equations.

4.2.3 More precision about JFNK

The primary motivation for developing JFNK methods is the ability to perform a Newton iteration without forming the Jacobian. In the JFNK approach, a Krylov method is used to solve the linear system of equations given by Eq. (4.41). At nonlinear iteration m, an initial linear residual, $\underline{r}_0^m$, is defined, given an initial guess, $\underline{\delta}_0^m = \underline{X}_0^{m+1} - \underline{X}_0^m$, for the Newton correction:

$$\underline{r}_0^m = -\underline{F}(\underline{X}^m) - \underline{\underline{J}}(\underline{X}^m)\,\underline{\delta}_0^m \tag{4.45}$$

Now, let j be the Krylov iteration index. Since the Krylov solution is a Newton correction and since a locally optimal move was just made in the direction of the previous Newton correction, the initial iterate for Krylov iteration, $\underline{\delta}_0^m$, is typically zero. This is asymptotically a reasonable guess in the Newton context, as the converged value for $\underline{\delta}^m$ should approach zero in late Newton iterations [23]. The j-th GMRES iteration minimizes $||\underline{\underline{J}}(\underline{X}^m)\,\underline{\delta}_j^m + \underline{F}(\underline{X}^m)||_2$ within a subspace of small dimension, relative to the number of unknowns, in a least-squares sense. $\underline{\delta}_j^m$ is drawn from the subspace spanned by the Krylov vectors,

$\{\underline{r}_0^m, \underline{\underline{J}}(\underline{X}^m)\,\underline{r}_0^m, \underline{\underline{J}}^2(\underline{X}^m)\,\underline{r}_0^m, ..., \underline{\underline{J}}^{j-1}(\underline{X}^m)\,\underline{r}_0^m\}$, and can be expressed as:

$$\underline{\delta}_j^m = \underline{\delta}_0^m + \sum_{i=0}^{j-1} \beta_i \underline{\underline{J}}^i(\underline{X}^m)\,\underline{r}_0^m \tag{4.46}$$

where the scalars β_i minimize $||\underline{\underline{J}}(\underline{X}^m)\,\underline{\delta}_j^m + \underline{F}(\underline{X}^m)\,||_2$. In Eq. (4.46) all the matrix-vector $\underline{\underline{J}}^i(\underline{X}^m)\,\underline{r}_0^m$ will be computed with Eq. (4.44). Then, the linear residual $\underline{r}_j^m = -\underline{F}(\underline{X}^m) - \underline{\underline{J}}(\underline{X}^m)\,\underline{\delta}_j^m$ is computed. If it satisfies $\frac{||\underline{r}_j^m||}{||\underline{r}_0^m||} \leq \epsilon_{Kry,r}$ or $||\underline{r}_j^m|| \leq \epsilon_{Kry,a}$ the GMRES iterations are stopped and a new Newton iterate can be computed. $\epsilon_{Kry,r}$ and $\epsilon_{Kry,a}$ are respectively the relative decrease tolerance for the norm of the linear residual from an initial linear residual and the absolute tolerance used for the norm of the linear residual. A maximum number of Krylov iterations, noted $j_{Kry,max}$, is also set.

5 Application to SD with implicit time integration

As explained in section (3.3), doing implicit time-marching schemes involves solving nonlinear systems. Then, in chapter (4) the inexact Newton algorithm and the JFNK method were presented as solver for such systems. However, they were introduced on a general form of a nonlinear system and not directly on Eq. (3.38). Thus, in this chapter, the practical application of the inexact Newton method and the JFNK method to Eq. (3.38) are described.

5.1 Solve the stage equation for DIRK schemes with an inexact Newton method by computing explicitly the Jacobian

5.1.1 Nonlinear system to solve

To solve Eq. (3.38) by a Newton method, the following function, of which the zeros are looked for, is introduced [18]:

$$\underline{G}\left(\underline{u}^q\right) = \underline{u}^q - \underline{u}^n - \Delta t a_{qq}\underline{R}^q - \Delta t \sum_{j=1}^{q-1} a_{qj}\underline{R}^j \tag{5.47}$$

Its Jacobian matrix is given by:

$$\underline{\underline{J}} = \left[\underline{\underline{I}} - \Delta t a_{qq}\left(\frac{\partial \underline{R}}{\partial \underline{u}}\right)\right] \tag{5.48}$$

Thus, the update from the current Newton iterate $\underline{u}^{q,m}$ to the new Newton iterate $\underline{u}^{q,m+1}$ is given by:

$$\left[\underline{\underline{I}} - \Delta t a_{qq}\left(\frac{\partial \underline{R}}{\partial \underline{u}}\right)^m\right]\left(\underline{u}^{q,m+1} - \underline{u}^{q,m}\right) = -\underline{G}\left(\underline{u}^{q,m}\right) \tag{5.49}$$

where m is still the nonlinear Newton index, $\underline{\underline{I}}$ is the identity matrix of size $N_{cells} \times n_{cons} \times N_{SP}$ and $\left(\frac{\partial \underline{R}}{\partial \underline{u}}\right)^m$ the Jacobian of the residual taken at iterate m. The algorithm starts by setting $\underline{u}^{1,0} = \underline{u}^n$ for the first stage ($q = 1$) and $\underline{u}^{q,0} = \underline{u}^{q-1}$ for the other stages ($q > 1$) . Therefore, at each Newton iteration, the linear system given by Eq. (5.49) is solved using a matrix inversion method.

Usually, when the computation of $\underline{\underline{J}}$ is very costly, a simplified Newton method is considered. In the context of DIRK schemes, it means that $\underline{\underline{J}}$ is only computed at instant n and not at every iterate m [18]. Consequently, if a simplified Newton method is considered, $\frac{\partial \underline{R}}{\partial \underline{u}}$ is computed only at instant n. This is what is done in SD when implicit time integration is used since the computation of $\frac{\partial \underline{R}}{\partial \underline{u}}$ is quite costly for complex PDE [51][31]. Obviously, this method has a worse convergence behavior compared to a classical Newton method but it is a good compromise in term of computation time. Thus, in my model code for 1D-advection and 1D-diffusion, I have also considered a simplified Newton method. Actually, in the cases of 1D-advection and 1D-diffusion, this was not a huge approximation because $\frac{\partial \underline{R}}{\partial \underline{u}}$ is not time dependent for these equations.

5.1.2 Jacobian of residual and local Jacobians

Eq. (5.49) showed that the Jacobian of the residual $\frac{\partial \underline{R}}{\partial \underline{u}}$ is needed. Let's go back to Eq . (2.29) in one mesh cell with R^i taken at some instant q, noted $\left(R^i\right)^q$. In the case of an implicit time-marching scheme, $\left(R^i\right)^q$ is not

known since $q > n$ for such schemes. Thus, a Taylor expansion can be done and since R^i is a function of u^i and u^{nb}, its variations can be observed with respect to u^i and u^{nb} [47][51]:

$$\left(R^i\right)^q \approx \left(R^i\right)^n + \left(\frac{\partial R^i}{\partial u^i}\right)^n \left(\left(u^i\right)^q - \left(u^i\right)^n\right) + \sum_{nb \neq i} \left(\frac{\partial R^i}{\partial u^{nb}}\right)^n \left(\left(u^{nb}\right)^q - \left(u^{nb}\right)^n\right) \tag{5.50}$$

where $\frac{\partial R^i}{\partial u^i}$ and $\frac{\partial R^i}{\partial u^{nb}}$ can be considered as "local Jacobians matrices" as they correspond to the variations of the cell residual with respect to the solution inside cell i and its surrounding. Their general expressions, because R^i, u^i and u^{nb} are vectors with $n_{cons} \times N_{SP}$ components, are the following square matrices of size $n_{cons} \times N_{SP}$:

$$\frac{\partial R^i}{\partial u^i} = \begin{pmatrix} \frac{\partial R^i_1}{\partial u^i_1} & \cdots & \frac{\partial R^i_1}{\partial u^i_{N_{SP}}} \\ \vdots & \frac{\partial R^i_s}{\partial u^i_j} & \vdots \\ \frac{\partial R^i_{N_{SP}}}{\partial u^i_1} & \cdots & \frac{\partial R^i_{N_{SP}}}{\partial u^i_{N_{SP}}} \end{pmatrix} = \left[\left(\frac{\partial R^i_s}{\partial u^i_j}\right)\right]_{\substack{1 \leq s \leq N_{SP} \\ 1 \leq j \leq N_{SP}}}, \quad \frac{\partial R^i}{\partial u^{nb}} = \begin{pmatrix} \frac{\partial R^i_1}{\partial u^{nb}_1} & \cdots & \frac{\partial R^i_1}{\partial u^{nb}_{N_{SP}}} \\ \vdots & \frac{\partial R^i_s}{\partial u^{nb}_j} & \vdots \\ \frac{\partial R^i_{N_{SP}}}{\partial u^{nb}_1} & \cdots & \frac{\partial R^i_{N_{SP}}}{\partial u^{nb}_{N_{SP}}} \end{pmatrix} = \left[\left(\frac{\partial R^i_s}{\partial u^{nb}_j}\right)\right]_{\substack{1 \leq s \leq N_{SP} \\ 1 \leq j \leq N_{SP}}} \tag{5.51}$$

where the components of these matrices, $\frac{\partial R^i_s}{\partial u^i_j}$ and $\frac{\partial R^i_s}{\partial u^{nb}_j}$, are square matrices of size n_{cons}. Thus, using Eq. (5.50), $\underline{R}^q$ can be expressed as:

$$\underline{R}^q = \begin{pmatrix} \left(R^1\right)^q \\ \vdots \\ \left(R^i\right)^q \\ \vdots \\ \left(R^{N_{cells}}\right)^q \end{pmatrix} \approx \underline{R}^n + \begin{pmatrix} \left(\frac{\partial R^1}{\partial u^1}\right)^n \left(\left(u^1\right)^q - \left(u^1\right)^n\right) + \sum_{nb \neq 1} \left(\frac{\partial R^1}{\partial u^{nb}}\right)^n \left(\left(u^{nb}\right)^q - \left(u^{nb}\right)^n\right) \\ \vdots \\ \left(\frac{\partial R^i}{\partial u^i}\right)^n \left(\left(u^i\right)^q - \left(u^i\right)^n\right) + \sum_{nb \neq i} \left(\frac{\partial R^i}{\partial u^{nb}}\right)^n \left(\left(u^{nb}\right)^q - \left(u^{nb}\right)^n\right) \\ \vdots \\ \left(\frac{\partial R^{N_{cells}}}{\partial u^{N_{cells}}}\right)^n \left(\left(u^{N_{cells}}\right)^q - \left(u^{N_{cells}}\right)^n\right) + \sum_{nb \neq N_{cells}} \left(\frac{\partial R^{N_{cells}}}{\partial u^{nb}}\right)^n \left(\left(u^{nb}\right)^q - \left(u^{nb}\right)^n\right) \end{pmatrix} \tag{5.52}$$

where the vector on the right corresponds to the matrix-vector product $\left(\frac{\partial \underline{R}}{\partial \underline{u}}\right)^n \left(\underline{u}^q - \underline{u}^n\right)$:

$$\left(\frac{\partial \underline{R}}{\partial \underline{u}}\right)^n \left(\underline{u}^q - \underline{u}^n\right) = \begin{pmatrix} \left(\frac{\partial R^1}{\partial u^1}\right)^n & \cdots & \left(\frac{\partial R^1}{\partial u^i}\right)^n & \cdots & \left(\frac{\partial R^1}{\partial u^{N_{cells}}}\right)^n \\ \vdots & & \vdots & & \vdots \\ \left(\frac{\partial R^i}{\partial u^1}\right)^n & \cdots & \left(\frac{\partial R^i}{\partial u^i}\right)^n & \cdots & \left(\frac{\partial R^i}{\partial u^{N_{cells}}}\right)^n \\ \vdots & & \vdots & & \vdots \\ \left(\frac{\partial R^{N_{cells}}}{\partial u^1}\right)^n & \cdots & \left(\frac{\partial R^{N_{cells}}}{\partial u^i}\right)^n & \cdots & \left(\frac{\partial R^{N_{cells}}}{\partial u^{N_{cells}}}\right)^n \end{pmatrix} \begin{pmatrix} \left(u^1\right)^q - \left(u^1\right)^n \\ \vdots \\ \left(u^i\right)^q - \left(u^i\right)^n \\ \vdots \\ \left(u^{N_{cells}}\right)^q - \left(u^{N_{cells}}\right)^n \end{pmatrix} \tag{5.53}$$

Therefore, with Eq. (5.53), it seems that the Jacobian of the residual $\frac{\partial \underline{R}}{\partial \underline{u}}$ is a square matrix of size $N_{cells} \times n_{cons} \times N_{SP}$ and composed of all the local Jacobians of all cells. Usually, this matrix is not dense because a lot of local Jacobians are zero matrices. For instance, in paragraphs (6.2.1) and (6.2.2), it is shown that $\frac{\partial \underline{R}}{\partial \underline{u}}$ is a three-diagonal block matrix for the 1D-advection equation:

$$\frac{\partial \underline{R}}{\partial \underline{u}} = \begin{pmatrix} \ddots & \ddots & \ddots & (0) & \\ (0) & \frac{\partial R^i}{\partial u^{i-1}} & \frac{\partial R^i}{\partial u^i} & \frac{\partial R^i}{\partial u^{i+1}} & (0) \\ & (0) & \ddots & \ddots & \ddots \end{pmatrix} \tag{5.54}$$

It comes from the fact that the residual inside any cell i, R^i, depends only on u^{i-1}, u^i and u^{i+1} for the 1D-advection case. Consequently, for each cell i, there are only three local Jacobians $\frac{\partial R^i}{\partial u^{i-1}}$, $\frac{\partial R^i}{\partial u^i}$ and $\frac{\partial R^i}{\partial u^{i+1}}$. It indicates that the length of the diagonal block in $\frac{\partial \underline{R}}{\partial \underline{u}}$ depends on the number of local Jacobians and so on the number of neighboring cells of cell i where the solution inside this cell has an influence on R^i.

5.1.3 Numerical computation of the Jacobian of the residual

In sections (6.2) and (7.2) analytic expressions for the Jacobian of the residual respectively for 1D-advection and 1D-diffusion will be found. However, it will be very difficult to compute these analytic expressions for more

complex partial differential equations because expressions of R_s^i or R^i will be hard to obtained with the SD discretization. To overcome this problem, one method is to compute them numerically (see [47] and [31]):

$$\frac{\partial R^i}{\partial u^i} \approx \frac{R^i\left(u^i+\epsilon, u^{nb}\right) - R^i\left(u^i, u^{nb}\right)}{\epsilon} \quad , \quad \frac{\partial R^i}{\partial u^{nb}} \approx \frac{R^i\left(u^i, u^{nb}+\epsilon\right) - R^i\left(u^i, u^{nb}\right)}{\epsilon} \tag{5.55}$$

where ϵ is a small parameter. It means that for each cell $k \in \{i, nb\}$, u_j^k, with $j \in [|1, N_{SP}|]$, is altered one by one and the residual inside cell i uniquely is re-computed to do the numerical derivative. Actually, this notation is not really appropriate: it is used to "sum up" the process that computes these Jacobians. However, more precisions have to be made to explain what is really done to construct them numerically. By looking at expressions (5.51) of local Jacobians, it seems that altering u_j^k by an amount of ϵ and recomputing the new residual inside cell i, noted $R^{i,new} = \left(R_1^{i,new}, ..., R_{N_{SP}}^{i,new}\right)^T$, gives one column of the local Jacobians:

$$\begin{pmatrix} \frac{\partial R_1^i}{\partial u_j^k} \\ \vdots \\ \frac{\partial R_s^i}{\partial u_j^k} \\ \vdots \\ \frac{\partial R_{N_{SP}}^i}{\partial u_j^k} \end{pmatrix} \tag{5.56}$$

using the formula:

$$\frac{\partial R_s^i}{\partial u_j^k} = \frac{R_s^{i,new} - R_s^i}{\epsilon} \quad \text{for} \quad 1 \leq s \leq N_{SP} \tag{5.57}$$

taken at a fixed j. Therefore, in practice, local Jacobians relative to cell k are computed column by column, each column corresponding to one alteration of the solution at solution point j inside cell k.

5.1.4 Implementation of the method

This inexact Newton method was implemented in my model code for both 1D-advection and 1D-diffusion. Since analytic expressions for the Jacobian of the residual have been derived for these equations, the user can choose between an analytical computation or a numerical computation. A comparison between these methods will be done in section (6.5) for 1D-advection and in section (7.4) for 1D-diffusion. For the model code, the Newton algorithm was explicitly written by following the explanations of section (4.1). However, for the linear solver, the Numpy library was used with direct methods (LU-numpy or LU-scipy) or iterative (conjugate gradient, GMRES) methods. However, as it will be shown in chapter 6 and 7, the numerical computation of the Jacobian of the residual is very costly, even for 1D equations. Thus, a JFNK approach was considered to avoid the explicit computation of these Jacobians.

5.2 Solve the stage equation for DIRK schemes with a Jacobian-Free-Newton-Krylov method

The idea is to find the zeros of function $\underline{G}$ defined in Eq. (5.47) without computing explicitly the matrix of linear system (5.49). To do so, the external library PETSc will be used.

5.2.1 The use of PETSc library

The JFNK method described in section (4.2) is very promising for solving nonlinear systems such as Eq. (3.38). However, in order to reduce the time needed for the implementation of implicit time-marching schemes in JAGUAR, an external library will be used to solve these systems with a JFNK approach. The library which was chosen is the Portable Extensible Toolkit for Scientific Computation (PETSc) developed by Argonne National Laboratory.

PETSc contains a suite of parallel linear and nonlinear solvers and time integrators that can be used in application codes written in FORTRAN, Python, C and C++. For the implementation of a JFNK method, the SNES

(Scalable Nonlinear Equation Solver) module of PETSc has to be employed because it provides a Newton solver with a Matrix-Free method. One main advantage of PETSc solvers is that almost every parameter of the solver can be set by the user such as the types of nonlinear and linear solvers or stopping criterion. Moreover, almost everything can be monitored like the number of nonlinear and linear iterations done, the residual norms in the Newton algorithm or also the reason why convergence or divergence was reached.

For a Matrix-Free use, this library is very powerful because it only needs a subroutine that explains how the Newton function $\underline{F}$ (or $\underline{G}$ in the SD case) has to be computed given the unknown vector $\underline{X}$. Then, PETSc nonlinear solver solves Eq. (4.39) with the JFNK method described in section (4.2). For instance, let's consider the following system of equations for $n \in \mathbb{N}$, called singular Broyden [28]:

$$F_1(\underline{X}) = ((3 - hX_1)X_1 - 2X_2 + 1)^2 \tag{5.58a}$$

$$F_i(\underline{X}) = ((3 - hX_i)X_i - X_{i-1} - 2X_{i+1} + 1)^2 \quad i = 2, ..., n-1 \tag{5.58b}$$

$$F_n(\underline{X}) = ((3 - hX_n)X_n - X_{n-1} + 1)^2 \tag{5.58c}$$

with $h = 2$ and the initial guess is set to $\underline{X}_i^0 = -1 \ \forall i \in [|1, n|]$. To solve this system with a JFNK method using PETSc, it only needs a subroutine that takes $\underline{X}$ as input and $\underline{F}$ as output which is computed using Eq. (5.58). A structure for the remaining constants, such as h and n in singular Broyden, is also needed if the code is written in FORTRAN. Thus, the same principle has to be applied for the nonlinear function defined by Eq. (5.47).

5.2.2 Function to give to PETSc

In the definition of $\underline{G}$, the unknown is vector $\underline{u}^q$ but the residual $\underline{R}^q = \underline{R}(\underline{u}^q)$ also depends on $\underline{u}^q$ so they are both unknowns. The remaining constants Δt and a_{qq}, vector $\underline{u}^n$ and the sum $\underline{S}^q = \sum_{j=1}^{q-1} a_{qj}\underline{R}^j$ are known and can be considered as parameters of system (3.38) such as h and n of system (5.58). Then, by considering $\underline{X} = \underline{u}^q$, the Newton function to give to PETSc is simply:

$$\boxed{\underline{G}(\underline{X}) = \underline{X} - \underline{u}^n - \Delta t a_{qq}\underline{R}(\underline{X}) - \Delta t \underline{S}^q} \tag{5.59}$$

with $\underline{X}^{1,0} = \underline{u}^n$ for the first stage ($q = 1$) and $\underline{X}^{q,0} = \underline{u}^{q-1}$ for the other stages ($q > 1$). All the parameters listed previously will be put in a FORTRAN structure for the code JAGUAR. The implementation of such function using PETSc library is discussed in the next paragraph. Eq. (5.59) shows that at each Newton iterate $\underline{X}^{q,m}$, the residual $\underline{R}(\underline{X}^{q,m})$ has to be recomputed. Therefore, for the JFNK method it is important that the SD code computes the residual quickly.

5.2.3 Implementation of the JFNK method in code JAGUAR

Here, the algorithm implemented in JAGUAR for an use of the JFNK method with PETSc is described. As mentioned in paragraph (3.3.2), for DIRK schemes, at each time iteration the Q stages are solved successively and Algorithm 3 summed up the process that has to be done at each time iteration. The subroutine "FormFunction" defines how to compute $\underline{G}$ using Eq. (5.59). Thus, the JFNK method using PETSc library was implemented in code JAGUAR following Algorithm 3. The subroutine that updates the solution in time was modified to take into account the use of DIRK schemes. The characteristics of the nonlinear and linear solvers, from the SNES module of PETSc, used for code JAGUAR, will be described in section (8.1).

Algorithm 3 Compute RK stages for SD in case of DIRK schemes using Matrix-Free (MF) approach

Define FormFunction($snes, \underline{X}, \underline{F}, \underline{u}^n, \underline{S}^q, \Delta t, a_{qq}$).
Define SNES context and its parameters.
for q from 1 to Q **do**
    Set $\underline{S}^q = \underline{0}$
    if q = 1 **then**
        Find $\underline{X} = \underline{u}^1$ with PETSc MF solver for FormFunction($snes, \underline{X}, \underline{F}, \underline{u}^n, \underline{0}, \Delta t, a_{11}$).
        Compute $\underline{R}^1$
    else
        for j from 1 to $q-1$ **do**
            $\underline{S}^q = \underline{S}^q + a_{qj}\underline{R}^j$
        end for
        Find $\underline{X} = \underline{u}^q$ with PETSc MF solver for FormFunction($snes, \underline{X}, \underline{F}, \underline{u}^n, \underline{S}^q, \Delta t, a_{qq}$).
        Compute $\underline{R}^q$
    end if
end for
Set $\underline{S} = \underline{0}$
for q from 1 to Q **do**
    $\underline{S} = \underline{S} + b_q\underline{R}^q$
end for
$\underline{u}^{n+1} = \underline{u}^n + \Delta t \times \underline{S}$

6 Model code for 1D-advection

This part is dedicated to solve the 1D-advection equation with SD using explicit and implicit temporal schemes. For a variable $u(x,t)$ transported at a constant speed c, this equation is:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \tag{6.60}$$

There is only one conserved variable: $n_{cons} = 1$ in all this chapter. Eq. (2.29) is a very simple equation to learn how the SD method can be implemented in a code and also for testing the numerical methods explained in the previous parts for implicit time-marching schemes applied to SD.

6.1 Theoretical results

To validate the numerical results, they will be compared to the exact solution of Eq. (6.60). For an initial state given by $u(x,0) = u_0(x)$ and an infinite domain, the analytic solution of Eq. (6.60) is:

$$u_{ana}(x,t) = u_0(x - ct) \qquad \forall (x,t) \in \mathbb{R} \times \mathbb{R}^+ \tag{6.61}$$

Since an infinite domain does not exist in practice, **periodic boundary conditions** are considered meaning that the solution "scan" the domain several times. For instance, for a finite domain $[-L_x, L_x]$, this condition means that:

$$u(-L_x, t) = u(L_x, t) \qquad \forall t > 0 \tag{6.62}$$

Thus, in this case, the solution is repeated every time $c \times t = 2 \times L_x$. In the model code, the absolute numerical error, noted $L_2^{abs}(u_{num})$, and the relative numerical error, noted $L_2^{rel}(u_{num})$, are computed using the classical formulas:

$$L_2^{abs}(u_{num}) = ||u_{num} - u_{ana}||_2 \tag{6.63a}$$

$$L_2^{rel}(u_{num}) = \frac{||u_{num} - u_{ana}||_2}{||u_{ana}||_2} \tag{6.63b}$$

where u_{num} stands for the numerical solution and $||.||_2$ for the 2-norm on vectors.

6.2 Analytical computing for the Jacobian of the residual

In the previous parts, it was said that the analytical computation for the Jacobian of the residual is possible for the 1D-advection equation. This is the purpose of this section where two different approaches are considered for the analytic computation of the Jacobian. The first approach, called "sequential", expresses R_s^i as a function of u_j^k. The second one uses the matrix form of SD in order to have directly R^i as a function of u_j^k. Obviously, both methods give the same results and the analytic Jacobian can be compared with the numerical Jacobian computed using the method described in section (5.1.3).

6.2.1 "Sequential" approach

The objective of this part is to find an analytical expression for the terms $\frac{\partial R_s^i}{\partial u_j^i}$ or $\frac{\partial R_s^i}{\partial u_j^{nb}}$ in the case of the 1D-advection equation. To do so the SD algorithm which computes the residual at solution point s inside cell i is used.

This algorithm starts with the fact that u is known at solution points. Let's note these values u_s^i where s is referring to "solution" points. This is a simplified notation to write $u^i\left(X_s^i, t\right)$ where X_s^i stands for the abscissa of solution point s inside cell i and t for the time. The next step is the interpolation of u at flux points X_f using Lagrange's polynome basis at solution points $h_s\left(X\right)$ where X can be any point inside the cell:

$$v^i = \left[v_f^i\right]_{1\leq f\leq N_{FP}}^T = \left[\sum_s u_s^i h_s^i\left(X_f^i\right)\right]_{1\leq f\leq N_{FP}}^T \tag{6.64}$$

where ^{T} means the transpose operation.

The computation of the flux at flux points, noted F^i, for the 1D-advection case is straightforward:

$$F^i = \left[F_f^i\right]_{1\leq f\leq N_{FP}}^T \overbrace{=}^{1D-Adv} c\times v^i = c\times\left[\sum_s u_s^i h_s^i\left(X_f^i\right)\right]_{1\leq f\leq N_{FP}}^T \tag{6.65}$$

After that, before computing the flux derivatives (the residual) at solution points, a Riemann solver needs to be used at each cell interface so that each interface will have only one value for the flux.

For an interface with on its left a state (u_L, F_L) and on its right a state (u_R, F_R), the flux at the interface is given by (upwind formulation for 1D-advection):

$$F_{int}^{L,R} = \frac{F_R + F_L - |c|\left(u_R - u_L\right)}{2} \tag{6.66}$$

If this formula is applied for a given interface which has cell numbered **i** on its **left** and $\mathbf{i+1}$ on its **right**, Eq. (6.66) becomes:

$$F_{int}^{i,i+1} = \frac{\overbrace{cv_1^{i+1}}^{F_R} + \overbrace{cv_{N_{FP}}^i}^{F_L} - |c|(\overbrace{v_1^{i+1}}^{u_R} - \overbrace{v_{N_{FP}}^i}^{u_L})}{2} \tag{6.67}$$

Some factorizations can be made inside Eq. (6.67):

$$F_{int}^{i,i+1} = \frac{\left(c-|c|\right)v_1^{i+1} + \left(c+|c|\right)v_{N_{FP}}^i}{2} \tag{6.68}$$

Finally, Eq. (6.64) is used to have $F_{int}^{i,i+1}$ expressed as a function of u_s^i and u_s^{i+1}:

$$\boxed{F_{int}^{i,i+1} = \frac{\left(c-|c|\right)\left(\sum_s u_s^{i+1}h_s^{i+1}(X_{f,1}^{i+1})\right) + \left(c+|c|\right)\left(\sum_s u_s^i h_s^i(X_{f,N_{FP}}^i)\right)}{2}} \tag{6.69}$$

At this point, all the flux values needed to compute its derivatives at solution points have been determined. The new flux containing the flux values strictly inside the cell (those which have not change from F^i) and also those on each interface updated after the use of the Riemann solver will be denoted $\bar{F}^i$. The Lagrange's polynomial basis created at flux points $l_f^i\left(X\right)$ inside cell i to differentiate $\bar{F}^i$ is still used:

$$R_s^i = -\left(\frac{\partial \bar{F}}{\partial x}\right)_s^i = -\sum_f \bar{F}_f^i l_f'^i\left(X_s^i\right) = -F_{int}^{i-1,i}l_1'^i\left(X_s^i\right) - F_{int}^{i,i+1}l_{N_{FP}}'^i\left(X_s^i\right) - \sum_{f\subset cell} F_f^i l_f'^i\left(X_s^i\right) \tag{6.70}$$

Here, $\bar{F}^i$ was split into its interface fluxes, $F_{int}^{i-1,i}$ and $F_{int}^{i,i+1}$, and its strictly intern fluxes F_f^i. Thus, because $F_{int}^{i-1,i}$ and $F_{int}^{i,i+1}$ **depend on** u **inside the neighboring cells** $\mathbf{i-1}$ **and** $\mathbf{i+1}$, the residual also depends on u inside cells $i-1$ and $i+1$ in addition to u inside cell i. Consequently, $nb = \{i-1, i+1\}$ for the 1D-advection equation and three local Jacobians have to be computed: $\frac{\partial R^i}{\partial u^{i-1}}$, $\frac{\partial R^i}{\partial u^i}$ and $\frac{\partial R^i}{\partial u^{i+1}}$.

The derivative of R_s^i with respect to u_j^k, where $k\in\{i-1, i, i+1\}$ and $j\in[|1, N_{SP}|]$ (one solution point inside cell k), is given by:

$$\frac{\partial R_s^i}{\partial u_j^k} = -\frac{\partial F_{int}^{i-1,i}}{\partial u_j^k}l_1'^i\left(X_s^i\right) - \frac{\partial F_{int}^{i,i+1}}{\partial u_j^k}l_{N_{FP}}'^i\left(X_s^i\right) - \sum_{f\subset cell}\frac{\partial F_f^i}{\partial u_j^k}l_f'^i\left(X_s^i\right) \tag{6.71}$$

where if $k=i$ (u **inside the cell**):

$$\frac{\partial F_{int}^{i-1,i}}{\partial u_j^i} = \frac{\left(c-|c|\right)h_j^i(X_{f,1}^i)}{2} \tag{6.72a}$$

$$\frac{\partial F_{int}^{i,i+1}}{\partial u_j^i} = \frac{\left(c+|c|\right)h_j^i(X_{f,N_{FP}}^i)}{2} \tag{6.72b}$$

$$\frac{\partial F_f^i}{\partial u_j^i} = c\times h_j^i\left(X_f^i\right) \tag{6.72c}$$

If $k = i - 1$ (u inside the **left** neighbor):

$$\frac{\partial F_{int}^{i-1,i}}{\partial u_j^{i-1}} = \frac{(c + |c|)\, h_j^{i-1}(X_{f,N_{FP}}^{i-1})}{2} \tag{6.73a}$$

$$\frac{\partial F_{int}^{i,i+1}}{\partial u_j^{i-1}} = 0 \tag{6.73b}$$

$$\frac{\partial F_f^i}{\partial u_j^{i-1}} = 0 \tag{6.73c}$$

and if $k = i + 1$ (u inside the **right** neighbor):

$$\frac{\partial F_{int}^{i-1,i}}{\partial u_j^{i+1}} = 0 \tag{6.74a}$$

$$\frac{\partial F_{int}^{i,i+1}}{\partial u_j^{i+1}} = \frac{(c - |c|)\, h_j^{i+1}(X_{f,1}^{i+1})}{2} \tag{6.74b}$$

$$\frac{\partial F_f^i}{\partial u_j^{i+1}} = 0 \tag{6.74c}$$

Therefore, $\frac{\partial R_s^i}{\partial u_j^k}$ can be expressed explicitly in the three different cases where u_j^k could be inside the considered cell or in its left and right neighbors:

$$\frac{\partial R_s^i}{\partial u_j^{i-1}} = -\frac{(c + |c|)\, h_j^{i-1}(X_{f,N_{FP}}^{i-1})}{2} l_1'^i\left(X_s^i\right) \tag{6.75a}$$

$$\frac{\partial R_s^i}{\partial u_j^i} = -\frac{(c - |c|)\, h_j^i\left(X_{f,1}^i\right)}{2} l_1'^i\left(X_s^i\right) - \frac{(c + |c|)\, h_j^i\left(X_{f,N_{FP}}^i\right)}{2} l_{N_{FP}}'^i\left(X_s^i\right) - \sum_{f \subset cell} c \times h_j^i\left(X_f^i\right) l_f'^i\left(X_s^i\right) \tag{6.75b}$$

$$\frac{\partial R_s^i}{\partial u_j^{i+1}} = -\frac{(c - |c|)\, h_j^{i+1}\left(X_{f,1}^{i+1}\right)}{2} l_{N_{FP}}'^i\left(X_s^i\right) \tag{6.75c}$$

Here, Eq. (6.75) do not depend on time but uniquely on space. Therefore, **the analytic Jacobian matrix can be entirely computed before the time loop**. Now, these formulas will be investigated in the cases where the function u is advected towards the right ($c > 0$) or towards the left ($c < 0$).

Advection towards the right ($c > 0$)

In this case, $|c| = +c$, therefore:

$$\frac{\partial R_s^i}{\partial u_j^{i-1}} = -c \times h_j^{i-1}(X_{f,N_{FP}}^{i-1}) l_1'^i\left(X_s^i\right) \tag{6.76a}$$

$$\frac{\partial R_s^i}{\partial u_j^i} = -c \times h_j^i\left(X_{f,N_{FP}}^i\right) l_{N_{FP}}'^i\left(X_s^i\right) - \sum_{f \subset cell} c \times h_j^i\left(X_f^i\right) l_f'^i\left(X_s^i\right) \tag{6.76b}$$

$$\frac{\partial R_s^i}{\partial u_j^{i+1}} = 0 \tag{6.76c}$$

The classical result for the advection towards the right is recovered: for a given cell there is **no information coming from the right**. The Jacobian of the residual is actually a **lower bi-diagonal block matrix**.

Advection towards the left ($c < 0$)

In this case, $|c| = -c$ this time, so:

$$\frac{\partial R_s^i}{\partial u_j^{i-1}} = 0 \tag{6.77a}$$

$$\frac{\partial R_s^i}{\partial u_j^i} = -c \times h_j^i(X_{f,1}^i) l_1'^i\left(X_s^i\right) - \sum_{f \subset cell} c \times h_j^i\left(X_f^i\right) l_f'^i\left(X_s^i\right) \tag{6.77b}$$

$$\frac{\partial R_s^i}{\partial u_j^{i+1}} = -c \times h_j^{i+1}(X_{f,1}^{i+1}) l_{N_{FP}}'^i\left(X_s^i\right) \tag{6.77c}$$

Here, unlike the previous case, there is **no information coming from the left**. Consequently, the Jacobian of the residual is an **upper bi-diagonal block matrix**.

Algorithm

To summarize the previous developments, Algorithm 4 can be used to compute **one local Jacobian** of a cell in the case of the 1D-advection equation. In the model code, it corresponds to a *Python* function called **computeLocJacAnaAdv1D**$(N_{FP}, N_{SP}, X_s, X_f, i, k, c)$. To compute the Jacobian of the residual for all cells, the previous algorithm is applied for all of them resulting in Algorithm 5.

Algorithm 4 Compute analytically $\frac{\partial R^i}{\partial u^k}$ for 1D-advection

Input(s): N_{FP}, N_{SP}, X_s, X_f, i, k, c

if k = i - 1 **then**
  for s from 1 to N_{SP} **do**
    Compute $l_1'^i\left(X_s^i\right)$
    for j from 1 to N_{SP} **do**
      Compute $h_j^{i-1}(X_{f,N_{FP}}^{i-1})$
      Compute $\frac{\partial R_s^i}{\partial u_j^{i-1}} = -\frac{(c+|c|)}{2} h_j^{i-1}(X_{f,N_{FP}}^{i-1}) l_1'^i\left(X_s^i\right)$
    end for
  end for
else if k = i **then**
  for s from 1 to N_{SP} **do**
    Compute $l_1'^i\left(X_s^i\right)$ and $l_{N_{FP}}'^i\left(X_s^i\right)$
    for j from 1 to N_{SP} **do**
      Compute $h_j^i(X_{f,1}^i)$ and $h_j^i(X_{f,N_{FP}}^i)$
      Set $S = 0$
      for f from 2 to $N_{FP} - 1$ **do**
        Compute $h_j^i(X_f^i)$ and $l_f'^i\left(X_s^i\right)$
        $S \leftarrow S + h_j^i(X_f^i) \times l_f'^i\left(X_s^i\right)$
      end for
      Compute $\frac{\partial R_s^i}{\partial u_j^i} = -\frac{(c-|c|)}{2} h_j^i(X_{f,1}^i) l_1'^i\left(X_s^i\right) - \frac{(c+|c|)}{2} h_j^i(X_{f,N_{FP}}^i) l_{N_{FP}}'^i\left(X_s^i\right) - c \times S$
    end for
  end for
else if k = i + 1 **then**
  for s from 1 to N_{SP} **do**
    Compute $l_{N_{FP}}'^i\left(X_s^i\right)$
    for j from 1 to N_{SP} **do**
      Compute $h_j^{i+1}(X_{f,1}^{i+1})$
      Compute $\frac{\partial R_s^i}{\partial u_j^{i+1}} = -\frac{(c-|c|)}{2} h_j^{i+1}(X_{f,1}^{i+1}) l_{N_{FP}}'^i\left(X_s^i\right)$
    end for
  end for
end if

Output(s): $\frac{\partial R^i}{\partial u^k}$

Algorithm 5 Compute analytically $\frac{\partial \underline{R}}{\partial \underline{u}}$ for 1D-advection

Input(s): N_{cells}, N_{FP}, N_{SP}, X_s, X_f, c

Initialization of $\frac{\partial \underline{R}}{\partial \underline{u}}$ as a square matrix of size $N_{cells} \times N_{SP}$.
for i from 1 to N_{cells} **do**
  $\left(\frac{\partial \underline{R}}{\partial \underline{u}}\right)_{\substack{1+(i-1)N_{SP} \leq row \leq iN_{SP} \\ 1+(i-2)N_{SP} \leq col \leq (i-1)N_{SP}}} = \text{computeLocJacAnaAdv1D}(N_{FP}, N_{SP}, X_s, X_f, i, i-1, c)$
  $\left(\frac{\partial \underline{R}}{\partial \underline{u}}\right)_{\substack{1+(i-1)N_{SP} \leq row \leq iN_{SP} \\ 1+(i-1)N_{SP} \leq col \leq iN_{SP}}} = \text{computeLocJacAnaAdv1D}(N_{FP}, N_{SP}, X_s, X_f, i, i, c)$
  $\left(\frac{\partial \underline{R}}{\partial \underline{u}}\right)_{\substack{1+(i-1)N_{SP} \leq row \leq iN_{SP} \\ 1+iN_{SP} \leq col \leq (i+1)N_{SP}}} = \text{computeLocJacAnaAdv1D}(N_{FP}, N_{SP}, X_s, X_f, i, i+1, c)$
end for

Output(s): $\frac{\partial \underline{R}}{\partial \underline{u}}$

where *row* and *col* are respectively the variables for the rows and the columns of matrix $\frac{\partial \underline{\underline{R}}}{\partial \underline{u}}$ here.

As mentioned above, the analytical Jacobian of the residual does not depend on time for the 1D-advection. It means that algorithm 5 has to be done before the time loop.

Note: In Algorithm 5 for $i = 1$ and $i = N_{cells}$, the fictive cells $i = 0$ and $i = N_{cells} + 1$, used for the boundary conditions, are appearing. Actually, what is done in practice, is to define a 1D-space with $N_{cells} + 2$ cells, where cells numbered $i = 1$ and $i = N_{cells} + 2$ are the fictive cells for the boundary conditions and cells from $i = 2$ to $i = N_{cells} + 1$ are the real cells. Thus, in practice, the loop in Algorithm 5 is done from $i = 2$ to $i = N_{cells} + 1$.

6.2.2 With matrix form of the SD procedure

In section (6.2.1) an expression of R_s^i as a function of u_j^k with $k \in \{i-1, i, i+1\}$ and $j \in [|1, N_{SP}|]$ was found. Then, formulas for the elements of the local Jacobians were obtained. However, another approach is to think on R^i as a function of u^{i-1}, u^i and u^{i+1} instead of doing it on R_s^i. Thus, the objective is to express R^i by involving u^{i-1}, u^i and u^{i+1} like it was done in [52]. This method has the advantage to need less calculations and also to involve matrix-product operations which can make the computation faster.

SD process for computing the residual of the 1D-advection equation

The column solution vector of size $N_{SP} = p + 1$ inside cell i, u^i, is reintroduced:

$$u^i = \left[u_s^i\right]_{1 \leq s \leq N_{SP}}^T \tag{6.78}$$

The **first step** is to extrapolate the solution at flux points using the Lagrange polynomial basis built at solution points. Then, vector v^i given by Eq. (6.64), is obtained.

However, this vector is not sufficient since a Riemann solver must be solved on the cell boundaries: information from neighboring cells $i-1$ and $i+1$ are required. Thus, two more vectors are introduced: $\widetilde{u}^i$ a column vector of size $3 \times N_{SP}$ containing the values of the solution at solution points inside cells $i-1$, i and $i+1$ and the column vector $\widetilde{v}^i$ of size $1 + (p+2) + 1 = p + 4$ containing the solution at flux points inside cell i, v^i and also the solution on the border flux points from adjacent cells (last component of v^{i-1} and first component of v^{i+1}) [52]:

$$\widetilde{u}^i = \begin{bmatrix} u^{i-1} \\ u^i \\ u^{i+1} \end{bmatrix} \tag{6.79}$$

$$\widetilde{v}^i = \begin{bmatrix} v_{N_{FP}}^{i-1} \\ v^i \\ v_1^{i+1} \end{bmatrix} \tag{6.80}$$

One important thing to notice is that the Lagrange basis used for $v_{N_{FP}}^{i-1}$ and v_1^{i+1} have to be those **built respectively inside cells $\mathbf{i-1}$** and **$\mathbf{i+1}$**.

Then, by defining $O_{m,n}$ the zero matrix of dimension $m \times n$, the extrapolation process for cell i can be seen as the following matrix-vector product $\widetilde{v}^i = E^i \widetilde{u}^i$ where E^i is the extrapolation matrix of size $(p+4) \times (3N_{SP})$ and the index i is to referred to cell i:

$$E^i = \begin{bmatrix} \left[h_s^{i-1}\left(X_{f,N_{FP}}^{i-1}\right)\right]_{1 \leq s \leq N_{SP}} & O_{1,N_{SP}} & O_{1,N_{SP}} \\ O_{1,N_{SP}} & \left[h_s^i\left(X_{f,1}^i\right)\right]_{1 \leq s \leq N_{SP}} & O_{1,N_{SP}} \\ O_{1,N_{SP}} & \left[h_s^i\left(X_{f,2}^i\right)\right]_{1 \leq s \leq N_{SP}} & O_{1,N_{SP}} \\ \vdots & \vdots & \vdots \\ O_{1,N_{SP}} & \left[h_s^i\left(X_{f,N_{FP}}^i\right)\right]_{1 \leq s \leq N_{SP}} & O_{1,N_{SP}} \\ O_{1,N_{SP}} & O_{1,N_{SP}} & \left[h_s^{i+1}\left(X_{f,1}^{i+1}\right)\right]_{1 \leq s \leq N_{SP}} \end{bmatrix} \tag{6.81}$$

The **second step** consists of computing the flux at flux points. For the 1D-advection, it means that a matrix F such as $c \times v^i = F\widetilde{v}^i$ has to be found. As a consequence, F is of size $N_{FP} \times (p+4)$. Here, the Riemann

problem is solved using the upwind Godunov scheme which can be put in the following form (starting from Eq. (6.66)):

$$F_{int}^{L,R} = \frac{F_R + F_L - |c|\,(u_R - u_L)}{2} = c\left(\frac{1 + sign\,(c)}{2}u_L + \frac{1 - sign\,(c)}{2}u_R\right) \tag{6.82}$$

where $sign\,(c) = c/|c|$ and the form of F is now straightforward with I_p the identity matrix of size p:

$$F = c\begin{bmatrix} \frac{1+sign(c)}{2} & \frac{1-sign(c)}{2} & O_{1,N_{SP}} & 0 & 0 \\ O_{1,N_{SP}} & O_{1,N_{SP}} & I_p & O_{1,N_{SP}} & O_{1,N_{SP}} \\ 0 & 0 & O_{1,N_{SP}} & \frac{1+sign(c)}{2} & \frac{1-sign(c)}{2} \end{bmatrix} \tag{6.83}$$

There is no index i here because **this matrix is the same for all the cells.**

For the **third** and **last step**, the flux polynomial $F\widetilde{v}^i$ is differentiated and its derivative is computed at solution points to have the term $\frac{\partial}{\partial x}\left[c \times u^i\right] = D^i F\widetilde{v}^i\,(t)$ where D^i is the derivative matrix of size $N_{SP} \times N_{FP}$ given by:

$$D^i = \left[l_f'^i\left(X_s^i\right)\right]_{\substack{1 \le s \le N_{SP} \\ 1 \le f \le N_{FP}}} \tag{6.84}$$

Finally, the overall process for computing the residual for 1D-advection inside cell i with matrix form of SD is:

$$R^i = \left[R_s^i\right]_{1 \le s \le N_{SP}}^T = -\underbrace{D^i F E^i}_{M_{Adv}^i}\widetilde{u}^i \tag{6.85}$$

where M_{Adv}^i is the "compact matrix" of the process which is of size $N_{SP} \times 3N_{SP}$.

With Eq. (6.85) and (6.79), the fact that, for 1D-advection, the residual inside cell i depends only on the solution inside cells $i-1$, i and $i+1$ is recovered.

Jacobian of the residual using matrix form of SD

With Eq. (6.85) and (6.79), it seems that all the information about the variations of R^i with respect to u^{i-1} or u^i or u^{i+1} is contained into $-M_{Adv}^i$. Actually, because this matrix is of size $N_{SP} \times 3N_{SP}$ each square submatrix of size N_{SP} corresponds to one of the local Jacobian matrix. Thus, the links between local Jacobians and M_{Adv}^i are the following:

$$\frac{\partial R^i}{\partial u^{i-1}} = -\left(M_{Adv}^i\right)_{\substack{1 \le row \le N_{SP} \\ 1 \le col \le N_{SP}}} \tag{6.86a}$$

$$\frac{\partial R^i}{\partial u^{i}} = -\left(M_{Adv}^i\right)_{\substack{1 \le row \le N_{SP} \\ N_{SP}+1 \le col \le 2N_{SP}}} \tag{6.86b}$$

$$\frac{\partial R^i}{\partial u^{i+1}} = -\left(M_{Adv}^i\right)_{\substack{1 \le row \le N_{SP} \\ 2N_{SP}+1 \le col \le 3N_{SP}}} \tag{6.86c}$$

With Eq. (6.86), the fact that Jacobian matrix of the residual, $\frac{\partial \underline{R}}{\partial \underline{u}}$, is a **tri-diagonal** block matrix is recovered. Each block of size $N_{SP} \times 3N_{SP}$ is exactly the opposite of matrix M_{Adv}^i.

Algorithm

Like in section (6.2.1), the algorithm to compute the Jacobian of the residual using the matrix form approach can be summed up. This algorithm is simpler than the one written in section (6.2.1) especially because it does not involve the computation of local Jacobians one by one with the function "ComputeLocalJacAnaAdv1D".

Algorithm 6 has also to be done before the time loop.

Note: Like in Algorithm 5, in practice, the loop in algorithm 6 is done from $i = 2$ to $i = N_{cells} + 1$.

6.3 Numerical computing for the Jacobian of the residual

Thanks to the study of the analytic case, it was shown that for each cell i, there are three local Jacobians to compute for the 1D-advection equation. Thus, it means that for each cell, the solution has to be altered at $3 \times N_{SP}$ solution points. Therefore, at each time step, $N_{cells} \times 3N_{SP}$ alterations of the solution have to be

Algorithm 6 Compute $\frac{\partial \underline{\underline{R}}}{\partial \underline{u}}$ for 1D-advection using matrix form of SD

Input(s): N_{cells}, N_{FP}, N_{SP}, X_s, X_f, i, c

1: Compute matrix F
2: **for** i from 1 to N_{cells} **do**
3: Compute E^i
4: Compute D^i
5: $\left(\frac{\partial \underline{\underline{R}}}{\partial \underline{u}}\right)_{\substack{1+(i-1)N_{SP}\leq row\leq iN_{SP} \\ 1+(i-2)N_{SP}\leq col\leq (i+1)N_{SP}}} = -D^i F E^i$
6: **end for**

Output(s): $\frac{\partial \underline{\underline{R}}}{\partial \underline{u}}$

done. Fortunately, when the solution is altered at a solution point, not all the process described in section (2.1) has to be done again for the three involved cells $i-1$, i and $i+1$. Actually, because the residual in all cells, $\underline{R}$, is known at instant n, it means that the values of v and the values of the flux at flux points were already computed in all the cells as illustrated on Figure 6.6 for the $p=2$ case:

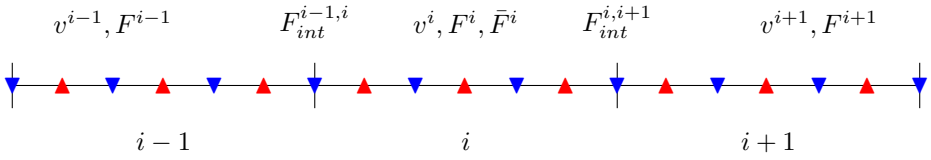


Figure 6.6: Initial situation before altering one solution point inside cell $i-1$ or i or $i+1$. Solution points (▲) and flux points (▼) for $p=2$.

Now, let's say that the solution at the center solution point in cell $i-1$ is altered, as seen in green on Figure 6.7:

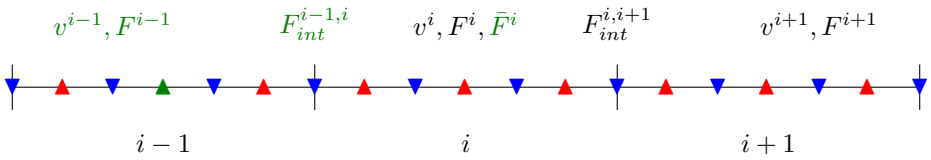


Figure 6.7: Altering the solution at green solution point inside cell $i-1$. The variables in green are those changing with this alteration.

Then, the interpolation process inside cell $i-1$ will give another polynomial of degree p and therefore the values of the solution at flux points v^{i-1} will change so the flux F^{i-1} too. Consequently, $F_{int}^{i-1,i}$ will also change but the values of v^i and F^i, also used to compute it, are already known and there is no need to recompute them again. Moreover, $F_{int}^{i,i+1}$ is also already known since none of the values which are used to compute it have changed. Finally, $\bar{F}^i$ is recomputed and the differentiation of it gives the new residual inside cell i. The same reasoning can be applied if the solution inside cell $i+1$ is altered as seen on Figure 6.8:

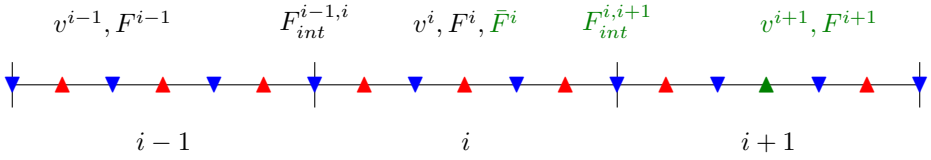


Figure 6.8: Altering the solution at green solution point inside cell $i+1$. The variables in green are those changing with this alteration.

The last case is when the solution inside cell i is altered. In this case, both $F_{int}^{i-1,i}$ and $F_{int}^{i,i+1}$ are changing and to compute them the already known data in cells $i-1$ and $i+1$ are used as seen on Figure 6.9:

The process explained in this section is summed up into two algorithms, 10 and 11 that can be found in appendix D.

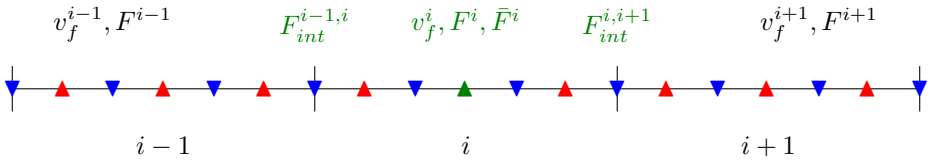


Figure 6.9: Altering the solution at green solution point inside cell i. The variables in green are those changing with this alteration.

6.4 Use of JFNK method with PETSc in the model code

The JFNK method using PETSc library was also implemented in the model code in *Python*. To do so the module "petsc4py.py" was used and the implementation follows algorithm 3. This was done in order to compare the inexact Newton method using the numerical Jacobian and the JFNK method which does not build any Jacobian.

6.5 Comparison between the different methods for computing the Jacobian of the residual

In this section, the inexact Newton method with numerical computing of the Jacobian will be compared to the JFNK method in terms of CPU time. The following test case is considered:

- Physical domain: $[-L_x, L_x]$ with $L_x = 1$ m.
- Advection speed: $c = 1$ m.s^{-1}.
- Physical time simulated: $T_f = 2$ s.
- Periodic boundary conditions.
- $N_{cells} = 216$ and $p = 2$.
- Initial solution: $u_0(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}}$ with $\sigma = 0.1$.
- Time integration schemes: ILDDRK(3,4) for implicit simulations and ERK(4,4) for explicit ones.

It represents the advection of a 1D-Gaussian profile over a domain of total length $L = 2$ m during 2 s at a speed of 1 m.s^{-1}. Consequently the Gaussian profile will do one loop of the domain. The value of p was chosen to have a high-order spatial scheme and also without too many DoF so that computations are not too long. The CPU time needed and the value of $L_2^{rel}(u_{num})$ will be picked for simulations with numerical computation of the Jacobian, the JFNK approach with PETSc but also for the analytic computation and the ERK(4,4) scheme. For the explicit scheme, the CFL_{SD} was set to 0.8 which appeared to ensure stability. For implicit scheme it was set to 10 so that the total number of time iterations was reduced by more than a factor of 10. This value seemed also to avoid too much dissipation and dispersion. Obviously, implicit schemes are doing less time iterations than explicit ones but their cost per iteration is higher. However, it is difficult to compare the cost per iteration for implicit schemes since an increase in the time step increases the cost of the nonlinear solver but reduces the number of time iterations. The results are presented in Table 6.2.

Results analysis: Firstly, only the implicit scheme with the analytical Jacobian is faster than the ERK(4,4) simulation. It was expected because, as shown in section (6.2) , the analytic Jacobian is not time dependent so it is computed one time before the time loop. Secondly, although its error is two times smaller, the JFNK method is slower than the inexact Newton method with numerical computation of the Jacobian which was not something expected. It probably comes from the boundary conditions of the domain. Indeed, as seen on Figure 6.10 when the solution is reaching the boundaries, there are oscillations that are appearing (red circles) but they disappear when the solution is coming back to the center of the domain. These boundary conditions are probably not well implemented. Thus, the JFNK method is doing more and more nonlinear and linear iterations when the solution reaches the boundaries whereas the inexact Newton method written in the model code does not. To remove the influence of the boundaries, the same simulation was done but for $T_f = 0.1$ s instead of 2 s. The results are presented in Table 6.3. The JFNK method is still slower but it is closer than the inexact Newton with numerical computation for this value of T_f. Moreover, the errors are the same for both methods meaning that there is no convergence issues. Finally, dissipation is recovered for implicit schemes since they have an error which is twice the error of the explicit scheme.

Time integration scheme	t_{CPU} (s)	$L_2^{rel}(u_{num})$
ERK(4,4)	104	0.009
ILDDRK(3,4) with analytical computation	27	0.006
ILDDRK(3,4) with numerical computation	225	0.011
ILDDRK(3,4) with JFNK	414	0.006

Table 6.2: Comparison between ERK(4,4) and ILDDRK(3,4) schemes with different approaches for computing the Jacobian of the residual in the case of 1D-advection. $T_f = 2$ s

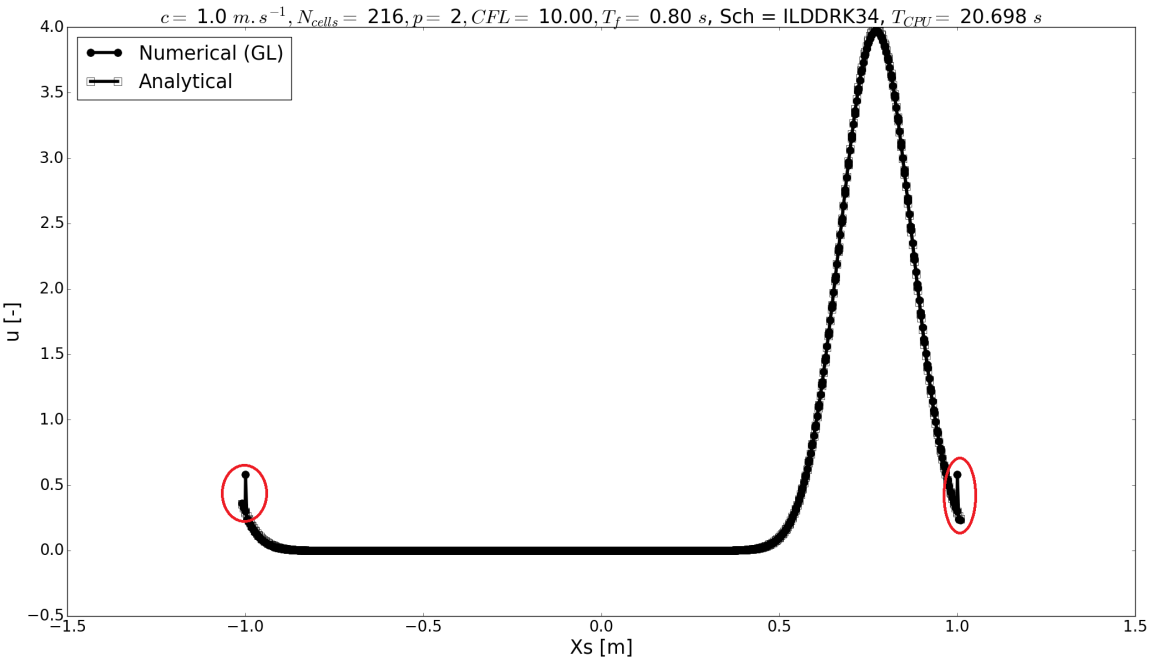


Figure 6.10: Oscillations (red circles) near the boundaries in 1D-advection with the model code.

Time integration scheme	t_{CPU} (s)	$L_2^{rel}(u_{num})$
ERK(4,4)	10	0.00001
ILDDRK(3,4) with analytical computation	2.6	0.0005
ILDDRK(3,4) with numerical computation	19	0.0005
ILDDRK(3,4) with JFNK	26	0.0005

Table 6.3: Comparison between ERK(4,4) and ILDDRK(3,4) schemes with different approaches for computing the Jacobian of the residual in the case of 1D-advection. $T_f = 0.1$ s

7 Model code for 1D-diffusion

One current problem in SD is the treatment of shocks which are strong discontinuities. Knowing that, and because the purpose of the internship was not to study shocks with SD, a diffusion of an initial gaussian profile in an "infinite" domain was done instead of the diffusion inside a solid bar in contact with a constant source of temperature for instance. The use of an infinite domain was also motivated by the observations done in section (6.5) where the boundaries had an influence on the numerical results. Thus, the following Initial Value Problem (IVP) for a variable $u(x,t)$ diffused in a medium of constant diffusivity κ is considered:

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2} \qquad -\infty < x < \infty, t > 0 \tag{7.87a}$$

$$u(x,0) = f(x) \qquad -\infty < x < \infty \tag{7.87b}$$

where $f(x)$ is the initial temperature or concentration profile inside the domain. As for 1D-advection, there is only one conserved variable: $n_{cons} = 1$ in all this chapter.

7.1 Theoretical results

The solution of problem (7.87) is composed of Green's function in 1D noted $G(x,t;\xi)$:

$$u_{ana}(x,t) = \int_{-\infty}^{\infty} G(x,t;\xi) f(\xi)\ d\xi = \int_{-\infty}^{\infty} \frac{e^{-\frac{(x-\xi)^2}{4\kappa t}}}{\sqrt{4\pi\kappa t}} f(\xi)\ d\xi \tag{7.88}$$

In the model code, this analytic solution is computed using the method "integrate" in scipy module.

As for 1D-advection, the process described in section (2.1.7) is used for the spatial discretization of Eq. (7.87). For a given cell i, the semi-discrete formulation of equation (7.87) can still be described by Eq. (2.29). Here, $R^i\left(u^i, u^{nb}\right)$ is the residual corresponding to the discretization of $\kappa\frac{\partial^2 u}{\partial x^2}$ of Eq. (7.87). In section (7.2), it will be seen that the neighbors where a change of u produces a change of R^i, are cells $i-2$, $i-1$, $i+1$ and $i+2$ and not only cells $i-1$ and $i+1$ which is the case of 1D-advection.

One thing has to be noticed for the diffusion term: its discretization is still of order $p+1$ and not $p+2$ as it could be thought. It can be checked numerically if the process described in appendix C is repeated. This comes from the fact that there is no introduction of what can be called "Diffusion Points": flux points are still kept to interpolate ∇u. That is why the method remains of order $p+1$ for diffusive terms.

7.2 Analytical computing for the Jacobian of the residual

In this section, methods for computing the Jacobian of the residual analytically for the 1D-diffusion equation will be derived. As for 1D-advection, the "sequential" approach is used in a first time to find an expression of R_s^i and in a second time the matrix form of the SD procedure is used to find an expression of R^i under the form of matrix-vector products.

7.2.1 "Sequential" approach

The objective of this part is to find an analytic expression for the terms $\frac{\partial R_s^i}{\partial u_j^i}$ and $\frac{\partial R_s^i}{\partial u_j^{nb}}$ in the case of the 1D-diffusion equation. To do so, the SD algorithm starting with the polynomial defined by Eq. (6.64) is still used. Then, before computing the term $\left(\frac{\partial u}{\partial x}\right)_s^i$ an unique value for v at each interface needs to be found. For

an interface which has cell numbered **i** on its **left** and **i + 1** on its **right** an interface value for the solution is defined (centred scheme):

$$v_{int}^{i,i+1} = \frac{v_{N_{FP}}^{i} + v_{1}^{i+1}}{2} \tag{7.89}$$

As for the flux in 1D-advection, a vector $\bar{v}^i$ is obtained and the Lagrange's polynomial basis created at flux points $l_f^i(X)$ can be used to have:

$$\left(\frac{\partial u}{\partial x}\right)_s^i = \sum_f \bar{v}_f^i l_f'^i = v_{int}^{i-1,i} l_1'^i\left(X_s^i\right) + v_{int}^{i,i+1} l_{N_{FP}}'^i\left(X_s^i\right) + \sum_{f\subset cell} v_f^i l_f'^i\left(X_s^i\right) \tag{7.90}$$

Then, the interpolation process is restarted by computing:

$$\left(\frac{\partial v}{\partial x}\right)_f^i = \sum_s \left(\frac{\partial u}{\partial x}\right)_s^i h_s^i\left(X_f^i\right) \tag{7.91}$$

Again, the values of $\left(\frac{\partial v}{\partial x}\right)_f^i$ at cell interfaces have to be computed. With the same notations as for Eq. (7.89) a formula for the gradient at the interface is introduced:

$$\left(\frac{\partial v}{\partial x}\right)_{int}^{i,i+1} = \frac{\left(\frac{\partial v}{\partial x}\right)_{N_{FP}}^{i} + \left(\frac{\partial v}{\partial x}\right)_{1}^{i+1}}{2} \tag{7.92}$$

Then, the new values of $\left(\frac{\partial v}{\partial x}\right)^i$, noted $\left(\overline{\frac{\partial v}{\partial x}}\right)_f^i$, are obtained and finally, the term $\left(\frac{\partial^2 u}{\partial x^2}\right)_s^i$ is given by:

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_s^i = \sum_f \left(\overline{\frac{\partial v}{\partial x}}\right)_f^i l_f'^i\left(X_s^i\right) = \left(\frac{\partial v}{\partial x}\right)_{int}^{i-1,i} l_1'^i\left(X_s^i\right) + \left(\frac{\partial v}{\partial x}\right)_{int}^{i,i+1} l_{N_{FP}}'^i\left(X_s^i\right) + \sum_{f\subset cell}\left(\frac{\partial v}{\partial x}\right)_f^i l_f'^i\left(X_s^i\right) \tag{7.93}$$

The expression of the residual inside cell i at solution point s for the 1D-diffusion process is directly:

$$R_s^i = \kappa \times \left(\frac{\partial^2 u}{\partial x^2}\right)_s^i \tag{7.94}$$

The derivative of Eq. (7.94) with respect to one of the u_j^k (k could be any cell at this point) involves:

$$\begin{aligned}\frac{\partial}{\partial u_j^k}\left[\left(\frac{\partial v}{\partial x}\right)_{int}^{i,i+1}\right] &= \frac{1}{2}\frac{\partial}{\partial u_j^k}\left[\sum_s\left(v_{int}^{i-1,i} l_1'^i\left(X_s^i\right) + v_{int}^{i,i+1} l_{N_{FP}}'^i\left(X_s^i\right) + \sum_{f\subset cell} v_f^i l_f'^i\left(X_s^i\right)\right) h_s^i\left(X_{f,N_{FP}}^i\right)\right] \\ &+ \frac{1}{2}\frac{\partial}{\partial u_j^k}\left[\sum_s\left(v_{int}^{i,i+1} l_1'^{i+1}\left(X_s^{i+1}\right) + v_{int}^{i+1,i+2} l_{N_{flux}}'^{i+1}\left(X_s^{i+1}\right) + \sum_{f\subset cell} v_f^{i+1} l_f'^{i+1}\left(X_s^{i+1}\right)\right) h_s^{i+1}\left(X_{f,1}^{i+1}\right)\right]\end{aligned} \tag{7.95}$$

and also:

$$\frac{\partial}{\partial u_j^k}\left[\left(\frac{\partial v}{\partial x}\right)_f^i\right] = \sum_s\left(\frac{\partial v_{int}^{i-1,i}}{\partial u_j^k} l_1'^i\left(X_s^i\right) + \frac{\partial v_{int}^{i,i+1}}{\partial u_j^k} l_{N_{FP}}'^i\left(X_s^i\right) + \sum_{f\subset cell}\frac{\partial v_f^i}{\partial u_j^k} l_f'^i\left(X_s^i\right)\right) h_s^i\left(X_f^i\right) \tag{7.96}$$

To compute $\frac{\partial}{\partial u_j^k}\left[\left(\frac{\partial v}{\partial x}\right)_{int}^{i-1,i}\right]$ Eq. (7.95) can be used by changing i for $i-1$ and $i+1$ for i. These three terms involved the derivatives of several v_{int} with respect to u_j^k. Using, Eq. (7.89) and (6.64) the general expression for $\frac{\partial v_{int}^{i,i+1}}{\partial u_j^k}$ is found:

$$\frac{\partial v_{int}^{i,i+1}}{\partial u_j^k} = \frac{1}{2}\begin{cases} h_j^i(X_{f,N_{FP}}^i) & \text{if } \; k = i \\ h_j^{i+1}(X_{f,1}^{i+1}) & \text{if } \; k = i+1 \\ 0 & \text{otherwise}\end{cases} \tag{7.97}$$

These three terms also involved $\frac{\partial v_f^i}{\partial u_j^k}$ which has the following expression by using Eq. (6.64):

$$\frac{\partial v_f^i}{\partial u_j^k} = \begin{cases} h_j^i(X_f^i) & \text{if } \; k = i \\ 0 & \text{otherwise}\end{cases} \tag{7.98}$$

Then, because the derivatives $\frac{\partial v_{int}^{i-2,i-1}}{\partial u_j^k}$ (in $\frac{\partial}{\partial u_j^k}\left[\left(\frac{\partial v}{\partial x}\right)_{int}^{i-1,i}\right]$) and $\frac{\partial v_{int}^{i+1,i+2}}{\partial u_j^k}$ (in $\frac{\partial}{\partial u_j^k}\left[\left(\frac{\partial v}{\partial x}\right)_{int}^{i,i+1}\right]$) are involved, there is a dependence on cells $i-2$ and $i+2$ for the residual. Thus, in the case of the 1D-diffusion, nb and k are different from the 1D-advection process:

$$nb = \{i-2, i-1, i+1, i+2\} \tag{7.99a}$$

$$k = \{i-2, i-1, i, i+1, i+2\} \tag{7.99b}$$

Like for 1D-advection, this Jacobian matrix is not time dependent and it is a **fifth-diagonal block** matrix.

Algorithm

To compute $\frac{\partial v_{int}^{i,i+1}}{\partial u_j^k}$ and $\frac{\partial v_f^i}{\partial u_j^k}$ let's create functions "Compute_dvintduj(X_f, j, k, L, R)" and "Compute_dvduj(X_f, j, k, i)". These two previous functions can be used to compute $\frac{\partial}{\partial u_j^k}\left[\left(\frac{\partial v}{\partial x}\right)_{int}^{i,i+1}\right]$ and $\frac{\partial}{\partial u_j^k}\left[\left(\frac{\partial v}{\partial x}\right)_{f}^{i}\right]$ using respectively Eq. (7.95) and (7.96). Let's call these functions "Compute_ddvintdxduj(X_s, X_f, j, k, L, R)" and "Compute_ddvdxduj(X_s, X_f, j, k, i)". With these functions, Algorithm 7 can be built to compute analytically $\frac{\partial R^i}{\partial u^k}$ for 1D-diffusion. In the model code, it corresponds to a *Python* function called

Algorithm 7 Compute analytically $\frac{\partial R^i}{\partial u^k}$ for 1D-diffusion

Input(s): N_{FP}, N_{SP}, X_s, X_f, i, k, κ
for s from 1 to N_{SP} **do**
  Compute $l_1'^i\left(X_s^i\right)$ and $l_{N_{FP}}'^i\left(X_s^i\right)$
  for j from 1 to N_{SP} **do**
    Compute $\frac{\partial}{\partial u_j^k}\left[\left(\frac{\partial v}{\partial x}\right)_{int}^{i-1,i}\right]$ and $\frac{\partial}{\partial u_j^k}\left[\left(\frac{\partial v}{\partial x}\right)_{int}^{i,i+1}\right]$ with Compute_ddvintdxduj($X_s, X_f, j, k, i-1, i$) and Compute_ddvintdxduj($X_s, X_f, j, k, i, i+1$)
    Set $S = 0$
    for f from 2 to $N_{FP}-1$ **do**
      Compute $l_f'^i\left(X_s^i\right)$ and $\frac{\partial}{\partial u_j^k}\left[\left(\frac{\partial v}{\partial x}\right)_{f}^{i}\right]$ with Compute_ddvdxduj(X_s, X_f, j, k, i)
      $S \leftarrow S + l_f'^i\left(X_s^i\right) \times \frac{\partial}{\partial u_j^k}\left[\left(\frac{\partial v}{\partial x}\right)_{f}^{i}\right]$
    end for
  end for
  Compute $\frac{\partial R_s^i}{\partial u_j^k} = \kappa\left[\frac{\partial}{\partial u_j^k}\left[\left(\frac{\partial v}{\partial x}\right)_{int}^{i-1,i}\right] \times l_1'^i\left(X_s^i\right) + \frac{\partial}{\partial u_j^k}\left[\left(\frac{\partial v}{\partial x}\right)_{int}^{i,i+1}\right] \times l_{N_{FP}}'^i\left(X_s^i\right) + S\right]$
end for
Output(s): $\frac{\partial R^i}{\partial u^k}$

computeLocJacAnaDiff1D($N_{FP}, N_{SP}, X_s, X_f, i, k, \kappa$). To compute the Jacobian of the residual for all cells, the previous algorithm is applied for all of them resulting in Algorithm 8.

Algorithm 8 Compute analytically $\frac{\partial \underline{R}}{\partial \underline{u}}$ for 1D-diffusion

Input(s): N_{cells}, N_{FP}, N_{SP}, X_s, X_f, c
Initialization of $\frac{\partial \underline{R}}{\partial \underline{u}}$ as a square matrix of size $N_{cells} \times N_{SP}$.
for i from 1 to N_{cells} **do**
  $\left(\frac{\partial \underline{R}}{\partial \underline{u}}\right)_{\substack{1+(i-1)N_{SP}\leq row\leq iN_{SP} \\ 1+(i-3)N_{SP}\leq col\leq(i-2)N_{SP}}} = \text{computeLocJacAnaDiff1D}(N_{FP}, N_{SP}, X_s, X_f, i, i-2, \kappa)$
  $\left(\frac{\partial \underline{R}}{\partial \underline{u}}\right)_{\substack{1+(i-1)N_{SP}\leq row\leq iN_{SP} \\ 1+(i-2)N_{SP}\leq col\leq(i-1)N_{SP}}} = \text{computeLocJacAnaDiff1D}(N_{FP}, N_{SP}, X_s, X_f, i, i-1, \kappa)$
  $\left(\frac{\partial \underline{R}}{\partial \underline{u}}\right)_{\substack{1+(i-1)N_{SP}\leq row\leq iN_{SP} \\ 1+(i-1)N_{SP}\leq col\leq iN_{SP}}} = \text{computeLocJacAnaDiff1D}(N_{FP}, N_{SP}, X_s, X_f, i, i, \kappa)$
  $\left(\frac{\partial \underline{R}}{\partial \underline{u}}\right)_{\substack{1+(i-1)N_{SP}\leq row\leq iN_{SP} \\ 1+iN_{SP}\leq col\leq(i+1)N_{SP}}} = \text{computeLocJacAnaDiff1D}(N_{FP}, N_{SP}, X_s, X_f, i, i+1, \kappa)$
  $\left(\frac{\partial \underline{R}}{\partial \underline{u}}\right)_{\substack{1+(i-1)N_{SP}\leq row\leq iN_{SP} \\ 1+(i+1)N_{SP}\leq col\leq(i+2)N_{SP}}} = \text{computeLocJacAnaDiff1D}(N_{FP}, N_{SP}, X_s, X_f, i, i+2, \kappa)$
end for
Output(s): $\frac{\partial \underline{R}}{\partial \underline{u}}$

Note: In Algorithm 8 for $i = 1$, $i = 2$, $i = N_{cells}-1$ and $i = N_{cells}$, the fictive cells $i = -1$, $i = 0$, $i = N_{cells}+1$ and $i = N_{cells}+2$, used for the boundary conditions, are appearing. Actually, what is done in practice, is to

define a 1D-space with $N_{cells}+4$ cells, where cells numbered $i=1$, $i=2$, $i=N_{cells}+3$ and $i=N_{cells}+4$ are the fictive cells for the boundary conditions and cells from $i=3$ to $i=N_{cells}+2$ are the real cells. Thus, in practice, the loop in Algorithm 8 is done from $i=3$ to $i=N_{cells}+2$.

7.2.2 With matrix form of the SD procedure

Like for 1D-advection, the SD process can be presented in matrix form for the 1D-diffusion equation. The principle is almost the same than the one presented in paragraph (6.2.2): only two main things are different. Firstly, as seen in sections (2.1.7) and (7.2.1), there is no Riemann solver since there is no advective flux: only the solution values at cell interfaces have to be recomputed to ensure continuity. Secondly, the number of steps is different because a second derivative needs to be computed. It means that the process extrapolation⇒averaging⇒differentiating is done two times. Thus, this process will be described using a matrix approach for SD knowing that the final objective is to express R^i by involving u^{i-2}, u^{i-1}, u^i, u^{i+1} and u^{i+2}.

SD process for computing the residual of the 1D-diffusion equation

The beginning of this process is quite similar to the one of the 1D-advection equation. Actually the **first step** is the same:

$$\widetilde{v^i} = E^i \widetilde{u^i} \tag{7.100}$$

The **second step is different** because the flux at flux points is not computed but only the solution at flux points and the information from neighboring cells $i-1$ and $i+1$ is still needed. For an interface between cell L at its left and cell R at its right, a **centered scheme** is taken:

$$v_{int}^{L,R} = \frac{v_{N_{FP}}^L + v_1^R}{2} \tag{7.101}$$

Thus, the matrix A (for "Average" matrix) of size $N_{FP} \times (p+4)$ such that $v^i = A\widetilde{v^i}$ is introduced:

$$A = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & O_{1,N_{SP}} & 0 & 0 \\ O_{1,N_{SP}} & O_{1,N_{SP}} & I_p & O_{1,N_{SP}} & O_{1,N_{SP}} \\ 0 & 0 & O_{1,N_{SP}} & \frac{1}{2} & \frac{1}{2} \end{bmatrix} \tag{7.102}$$

The **third step is the same** as for the 1D-advection process, the solution at flux points v^i is differentiated:

$$\left(\frac{\partial u}{\partial x}\right)^i = D^i A \widetilde{v^i} = \underbrace{D^i A E^i}_{M_{Diff}^i} \widetilde{u}^i \tag{7.103}$$

where M_{Diff}^i is the compact matrix for the computation of $\left(\frac{\partial u}{\partial x}\right)^i$ with SD.

Now, this process has to be restarted to obtain the term $\left(\frac{\partial^2 u}{\partial x^2}\right)^i$. To do so, $\left(\frac{\partial v}{\partial x}\right)^i$ is needed. The **fourth step** considered the following column vector:

$$\left(\widetilde{\frac{\partial u}{\partial x}}\right)^i = \begin{bmatrix} \left(\frac{\partial u}{\partial x}\right)^{i-1} \\ \left(\frac{\partial u}{\partial x}\right)^{i} \\ \left(\frac{\partial u}{\partial x}\right)^{i+1} \end{bmatrix} = \begin{bmatrix} M_{Diff}^{i-1} \widetilde{u}^{i-1} \\ M_{Diff}^{i} \widetilde{u}^{i} \\ M_{Diff}^{i+1} \widetilde{u}^{i+1} \end{bmatrix} \tag{7.104}$$

Since $\widetilde{u}^{i-1}$, $\widetilde{u}^i$ and $\widetilde{u}^{i+1}$ have some terms in common, a matrix $\boldsymbol{M}_{Diff}^i$ such that $\left(\widetilde{\frac{\partial u}{\partial x}}\right)^i = \boldsymbol{M}_{Diff}^i \widetilde{\widetilde{u}}^i$ has to be found where:

$$\widetilde{\widetilde{u}}^i = \begin{bmatrix} u^{i-2} \\ u^{i-1} \\ u^i \\ u^{i+1} \\ u^{i+2} \end{bmatrix} \tag{7.105}$$

Then, it seems that $\boldsymbol{M}^i_{Diff}$ is a matrix of size $3N_{SP} \times 5N_{SP}$ with this structure:

$$\boldsymbol{M}^i_{Diff} = \begin{bmatrix} M^{i-1}_{Diff} & O_{1,N_{SP}} & O_{1,N_{SP}} \\ O_{1,N_{SP}} & M^i_{Diff} & O_{1,N_{SP}} \\ O_{1,N_{SP}} & O_{1,N_{SP}} & M^{i+1}_{Diff} \end{bmatrix} \tag{7.106}$$

Thus, the fourth step ends with the extrapolation process:

$$\left(\widetilde{\frac{\partial v}{\partial x}}\right)^i = E^i \left(\widetilde{\frac{\partial u}{\partial x}}\right)^i = E^i \boldsymbol{M}^i_{Diff} \widetilde{\widetilde{u}}^i \tag{7.107}$$

The **fifth step** is equivalent to the second step:

$$\left(\frac{\partial v}{\partial x}\right)^i = A \left(\widetilde{\frac{\partial v}{\partial x}}\right)^i = AE^i \boldsymbol{M}^i_{Diff} \widetilde{\widetilde{u}}^i \tag{7.108}$$

Finally, the **sixth step** is equivalent to the third step:

$$\left(\frac{\partial^2 u}{\partial x^2}\right)^i = D^i \left(\frac{\partial v}{\partial x}\right)^i = \underbrace{D^i A E^i}_{M^i_{Diff}} \boldsymbol{M}^i_{Diff} \widetilde{\widetilde{u}}^i \tag{7.109}$$

The overall process for computing the residual for 1D-diffusion inside cell i with matrix form of SD is:

$$R^i = \left[R^i_s\right]^T_{1 \leq s \leq N_{SP}} = \kappa \times M^i_{Diff} \boldsymbol{M}^i_{Diff} \widetilde{\widetilde{u}}^i \tag{7.110}$$

Jacobian of the residual using matrix form of SD

With Eq. (7.110) and (7.105), it seems that all the information about the variations of R^i with respect to one of the u^k, where $k \in \{i-2, i-1, i, i+1, i+2\}$, is contained into $\kappa D^i A E^i \boldsymbol{M}^i_{Diff}$. Actually, because this matrix is of size $N_{SP} \times 5N_{SP}$, each square submatrix of size N_{SP} corresponds to one of the local Jacobian matrix. Thus, the links between local Jacobians and $\kappa D^i A E^i \boldsymbol{M}^i_{Diff}$ are the following:

$$\frac{\partial R^i}{\partial u^{i-2}} = \kappa \left(D^i A E^i \boldsymbol{M}^i_{Diff}\right)_{\substack{1 \leq row \leq N_{SP} \\ 1 \leq col \leq N_{SP}}} \tag{7.111a}$$

$$\frac{\partial R^i}{\partial u^{i-1}} = \kappa \left(D^i A E^i \boldsymbol{M}^i_{Diff})\right)_{\substack{1 \leq row \leq N_{SP} \\ N_{SP}+1 \leq col \leq 2N_{SP}}} \tag{7.111b}$$

$$\frac{\partial R^i}{\partial u^{i}} = \kappa \left(D^i A E^i \boldsymbol{M}^i_{Diff}\right)_{\substack{1 \leq row \leq N_{SP} \\ 2N_{SP}+1 \leq col \leq 3N_{SP}}} \tag{7.111c}$$

$$\frac{\partial R^i}{\partial u^{i+1}} = \kappa \left(D^i A E^i \boldsymbol{M}^i_{Diff}\right)_{\substack{1 \leq row \leq N_{SP} \\ 3N_{SP}+1 \leq col \leq 4N_{SP}}} \tag{7.111d}$$

$$\frac{\partial R^i}{\partial u^{i+2}} = \kappa \left(D^i A E^i \boldsymbol{M}^i_{Diff}\right)_{\substack{1 \leq row \leq N_{SP} \\ 4N_{SP}+1 \leq col \leq 5N_{SP}}} \tag{7.111e}$$

With Eq. (7.111), the fact that Jacobian matrix of the residual for 1D-diffusion, $\frac{\partial \overline{R}}{\partial \overline{u}}$, is a **fifth-diagonal** block matrix is recovered where each block of size $N_{SP} \times 5N_{SP}$ is exactly the matrix $\kappa D^i A E^i \boldsymbol{M}^i_{Diff}$.

Algorithm

The overall process described above can be summed up in Algorithm 9.

Note: Like in Algorithm 8, in practice, the loop in Algorithm 9 is done from $i = 3$ to $i = N_{cells} + 2$.

7.3 Numerical computing for the Jacobian of the residual

As for the advection case, the Jacobian of the residual can be computed numerically. The process is a bit more complicated since u^{nb} is bigger for the 1D-diffusion than for the 1D-advection.

Thanks to the study of the analytic case, it was shown that for each cell i, there are five local Jacobians to compute for the 1D-diffusion equation. Thus, it means that for each cell, the solution has to be altered at

Algorithm 9 Compute $\frac{\partial \underline{\underline{R}}}{\partial \underline{u}}$ for 1D-diffusion using matrix form of SD

Input(s): N_{cells}, N_{FP}, N_{SP}, X_s, X_f, i, κ

1: Compute Average matrix A
2: **for** i from 1 to N_{cells} **do**
3: Compute E^i
4: Compute D^i
5: Compute $\boldsymbol{M}^i_{Diff}$ using M^{i-1}_{Diff}, M^i_{Diff} and M^{i+1}_{Diff}
6: $\left(\frac{\partial \underline{\underline{R}}}{\partial \underline{u}}\right)_{\substack{1+(i-1)N_{SP}\leq row\leq iN_{SP} \\ 1+(i-3)N_{SP}\leq col\leq (i+2)N_{SP}}} = \kappa \times D^i A E^i \boldsymbol{M}^i_{Diff}$
7: **end for**

Output(s): $\frac{\partial \underline{\underline{R}}}{\partial \underline{u}}$

$5 \times N_{SP}$ solution points. Therefore, at each time step, $N_{cells} \times 5N_{SP}$ alterations of the solution must be done. Fortunately, like for 1D-advection, when the solution is altered at a solution point not all the process described in section (2.1.7) has to be done again for the five involved cells $i-2$, $i-1$, i, $i+1$ and $i+2$. Actually, because the residual in all cells, $\underline{R}$, is known at instant n, it means that the values of v and the values of its derivatives at flux points in all the cells have already been computed as seen on Figure 7.11 for the $p = 2$ case.

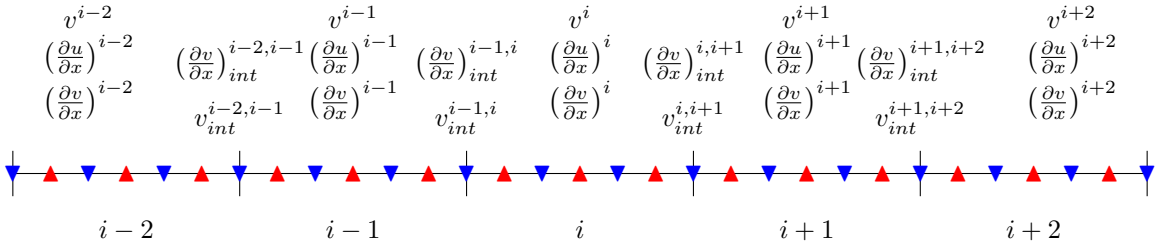


Figure 7.11: Initial situation before altering one solution point inside cell $i-2$ or $i-1$ or i or $i+1$ or $i+2$. Solution points (▲) and flux points (▼).

As for 1D-advection, the "optimized" process to compute the local Jacobians numerically can be written down. Let's say that the solution at the center solution point in cell $i-2$ is altered as seen on Figure 7.12:

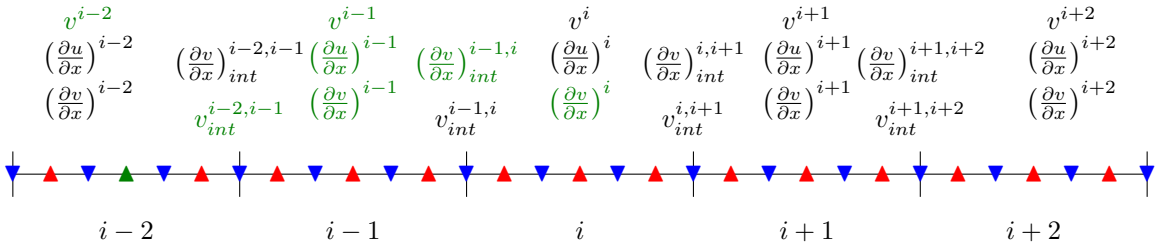


Figure 7.12: Altering the solution at green solution point inside cell $i-2$. The variables in green are those changing with this alteration.

Then, the interpolation process in this cell will give another polynomial of degree p and the values of the solution at flux points v^{i-2} will change so the value $v^{i-2,i-1}_{int}$ will change too. Consequently, v^{i-1} will also change but the former values of v^{i-1}, also used to compute $v^{i-2,i-1}_{int}$, are already known so there is no need to recompute them again: only v^{i-1}_1 will change. Moreover, $v^{i-1,i}_{int}$ is also already known since none of the values which are used to compute it have changed. Thus, the polynomial $\left(\frac{\partial u}{\partial x}\right)^{i-1}$ built using v^{i-1} is different and therefore $\left(\frac{\partial v}{\partial x}\right)^{i-1}$ also changed. Consequently, $\left(\frac{\partial v}{\partial x}\right)^{i-1,i}_{int}$ is changing so $\left(\frac{\partial v}{\partial x}\right)^{i}$ is changing because $\left(\frac{\partial v}{\partial x}\right)^{i}_1$ is different. Finally, $\left(\frac{\partial^2 u}{\partial x^2}\right)^{i}$ is recomputed with the differentiation of $\left(\frac{\partial v}{\partial x}\right)^{i}$ and it gives the new residual inside cell i for the 1D-diffusion.

It has to be noted that obviously the three terms $\left(\frac{\partial u}{\partial x}\right)^{i-2}$, $\left(\frac{\partial v}{\partial x}\right)^{i-2}$ and $\left(\frac{\partial v}{\partial x}\right)^{i-2,i-1}_{int}$ are also changing but, since they are not needed for the computation of $R^{i,new}$, they are not recomputed. This is the reason why they are not in green in Figure 7.12. This process can be summed up as follows by putting in order only the quantities

that need to be recomputed:

$$u^{i-2} \Rightarrow v^{i-2} \Rightarrow v_{int}^{i-2,i-1} \Rightarrow v_1^{i-1} \Rightarrow \left(\frac{\partial u}{\partial x}\right)^{i-1} \Rightarrow \left(\frac{\partial v}{\partial x}\right)^{i-1} \Rightarrow \left(\frac{\partial v}{\partial x}\right)_{int}^{i-1,i} \Rightarrow \left(\frac{\partial v}{\partial x}\right)_1^{i} \Rightarrow \left(\frac{\partial^2 u}{\partial x^2}\right)^{i} \Rightarrow R^{i,new}$$

The same reasoning can be applied if the solution inside cell $i+2$ is altered as seen on Figure 7.13 with the summary of the process just below it:

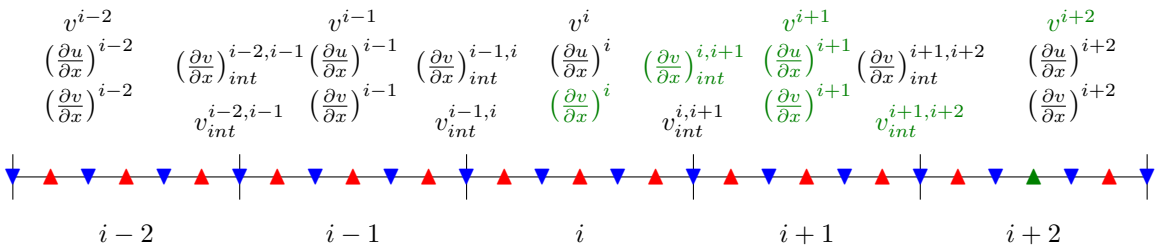


Figure 7.13: Altering the solution at green solution point inside cell $i+2$. The variables in green are those changing with this alteration.

$$u^{i+2} \Rightarrow v^{i+2} \Rightarrow v_{int}^{i+1,i+2} \Rightarrow v_{N_{FP}}^{i+1} \Rightarrow \left(\frac{\partial u}{\partial x}\right)^{i+1} \Rightarrow \left(\frac{\partial v}{\partial x}\right)^{i+1} \Rightarrow \left(\frac{\partial v}{\partial x}\right)_{int}^{i,i+1} \Rightarrow \left(\frac{\partial v}{\partial x}\right)_{N_{FP}}^{i} \Rightarrow \left(\frac{\partial^2 u}{\partial x^2}\right)^{i} \Rightarrow R^{i,new}$$

Another case is when the solution is altered inside the close neighbors of cell i: $i-1$ and $i+1$. Let's take the alteration of the solution inside cell $i-1$ in example as seen in Figure 7.14, still with its summary below:

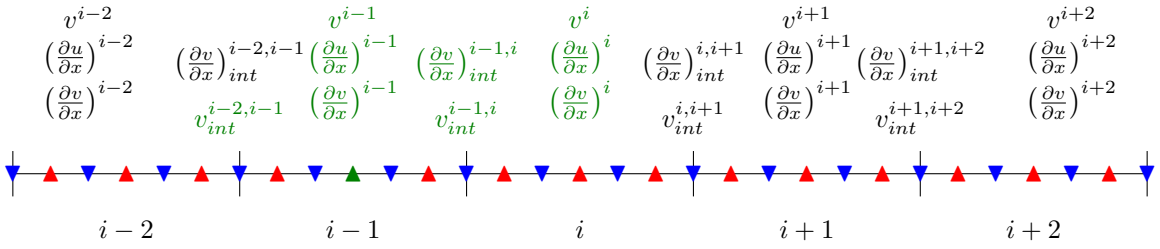


Figure 7.14: Altering the solution at green solution point inside cell $i-1$. The variables in green are those changing with this alteration.

$$u^{i-1} \Rightarrow v^{i-1} \Rightarrow v_{int}^{i-2,i-1}, v_{int}^{i-1,i} \Rightarrow v_1^{i-1}, v_{N_{FP}}^{i-1}, v_1^{i} \Rightarrow \left(\frac{\partial u}{\partial x}\right)^{i-1}, \left(\frac{\partial u}{\partial x}\right)^{i} \Rightarrow \left(\frac{\partial v}{\partial x}\right)^{i-1}, \left(\frac{\partial v}{\partial x}\right)^{i}$$
$$\Rightarrow \left(\frac{\partial v}{\partial x}\right)_{int}^{i-1,i}, \left(\frac{\partial v}{\partial x}\right)_{int}^{i,i+1} \Rightarrow \left(\frac{\partial v}{\partial x}\right)_1^{i}, \left(\frac{\partial v}{\partial x}\right)_{N_{FP}}^{i} \Rightarrow \left(\frac{\partial^2 u}{\partial x^2}\right)^{i} \Rightarrow R^{i,new}$$

Then, the process for cell $i+1$ is straightforward:

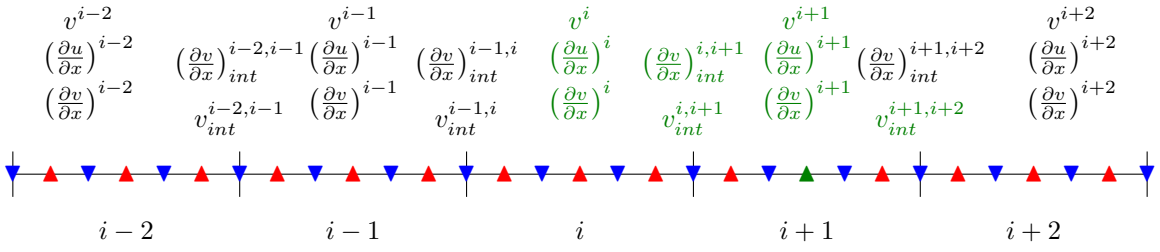


Figure 7.15: Altering the solution at green solution point inside cell $i+1$. The variables in green are those changing with this alteration.

$$u^{i+1} \Rightarrow v^{i+1} \Rightarrow v_{int}^{i,i+1}, v_{int}^{i+1,i+2} \Rightarrow v_{N_{FP}}^{i}, v_1^{i+1}, v_{N_{FP}}^{i+1} \Rightarrow \left(\frac{\partial u}{\partial x}\right)^{i}, \left(\frac{\partial u}{\partial x}\right)^{i+1} \Rightarrow \left(\frac{\partial v}{\partial x}\right)^{i}, \left(\frac{\partial v}{\partial x}\right)^{i+1}$$
$$\Rightarrow \left(\frac{\partial v}{\partial x}\right)_{int}^{i-1,i}, \left(\frac{\partial v}{\partial x}\right)_{int}^{i,i+1} \Rightarrow \left(\frac{\partial v}{\partial x}\right)_{1}^{i}, \left(\frac{\partial v}{\partial x}\right)_{N_{FP}}^{i} \Rightarrow \left(\frac{\partial^2 u}{\partial x^2}\right)^{i} \Rightarrow R^{i,new}$$

The last case is the alteration of the solution inside cell i as presented in Figure 7.16 :

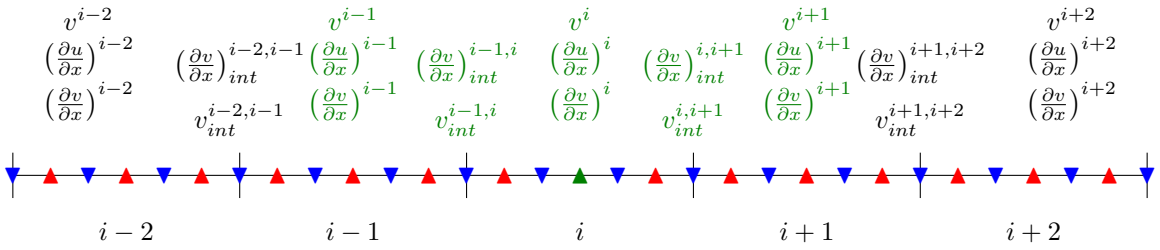


Figure 7.16: Altering the solution at green solution point inside cell i. The variables in green are those changing with this alteration.

$$u^{i} \Rightarrow v^{i} \Rightarrow v_{int}^{i-1,i}, v_{int}^{i,i+1} \Rightarrow v_{N_{FP}}^{i-1}, v_1^{i}, v_{N_{FP}}^{i}, v_1^{i+1} \Rightarrow \left(\frac{\partial u}{\partial x}\right)^{i-1}, \left(\frac{\partial u}{\partial x}\right)^{i}, \left(\frac{\partial u}{\partial x}\right)^{i+1} \Rightarrow \left(\frac{\partial v}{\partial x}\right)^{i-1}, \left(\frac{\partial v}{\partial x}\right)^{i}, \left(\frac{\partial v}{\partial x}\right)^{i+1}$$
$$\Rightarrow \left(\frac{\partial v}{\partial x}\right)_{int}^{i-1,i}, \left(\frac{\partial v}{\partial x}\right)_{int}^{i,i+1} \Rightarrow \left(\frac{\partial v}{\partial x}\right)_{1}^{i}, \left(\frac{\partial v}{\partial x}\right)_{N_{FP}}^{i} \Rightarrow \left(\frac{\partial^2 u}{\partial x^2}\right)^{i} \Rightarrow R^{i,new}$$

The process explained in this section is summed up into two algorithms, 13 and 12, that can be found in appendix E. As seen here, this type of reasoning is not simple and it is only a 1D-case with one variable. Therefore, in 3D multivariable, this process is very fastidious because it needs a certain knowledge of the structure of the Jacobian of the residual which is very complicated. That is why, the JFNK approach is also preferred, not only because it should be faster, but also it does not need a knowledge of the Jacobian since there is no explicit computation of it.

7.4 Comparison between the different methods for computing the Jacobian of the residual

In this section, the inexact Newton method with numerical computing of the Jacobian will be compared to the JFNK method in terms of CPU time. The following test case is considered

- Physical domain: $[-L_x, L_x]$ with $L_x = 0.15$ m .
- Diffusion coefficient: $\kappa = 0.0155$ m^2.s^{-1}.
- Physical time simulated: $T_f = 0.002$ s $<< \frac{L_x^2}{\kappa} = 1.45$ s.
- No boundary conditions.
- $N_{cells} = 216$ and $p = 2$.
- Initial solution: $f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}}$ with $\sigma = 0.01$.
- Time integration schemes: ILDDRK(3,4) for implicit simulations and ERK(4,4) for explicit ones.

It represents the diffusion of a 1D-Gaussian profile over a domain of total length $L = 0.3$ m during a time T_f very small compares to the characteristic time of diffusion in this problem. This was done in order to simulate an "infinite" domain where the boundaries have no influence on the numerical solution. As for 1D-advection, CPU times and $L_2^{rel}(u_{num})$ will be picked for both simulations with numerical and analytical computation of the Jacobian, the JFNK approach with PETSc and also the ERK(4,4) scheme. The time step will be computed with the Fourier number. It was set to $\alpha_{SD} = 0.4$ for ERK(4,4): a value that ensures stability. For ILDDRK(3,4) $\alpha_{SD} = 10$ was chosen: this value seemed to avoid too much dissipation and dispersion. The results are presented in Table 7.4.

Time integration scheme	t_{CPU} (s)	$L_2^{rel}(u_{num})$
ERK(4,4)	172	1.33e-5
ILDDRK(3,4) with analytical computation	22	1.34e-5
ILDDRK(3,4) with numerical computation	125	1.34e-5
ILDDRK(3,4) with JFNK	85	1.32e-5

Table 7.4: Comparison between ERK(4,4) and ILDDRK(3,4) schemes with different approaches for computing the Jacobian of the residual in the case of 1D-diffusion.

Results analysis: Unlike the 1D-advection case, all the implicit computations are faster than the explicit one. That was expected because in the advection case, the time step ratio was of 12.5 (ratio of the CFL_{SD}) whereas in the diffusion case it is 25. Therefore, even less iterations are done. Moreover, the JFNK is well faster than the inexact Newton with numerical computation when there is no influence of the boundaries which is quite promising. About the errors, they are quite the same even with a bigger time step for ILDDRK(3,4). Maybe because the error is linked to the spatial scheme in this case.

These results are quite encouraging for the use of the JFNK method in code JAGUAR where the boundary conditions should be well implemented.

8 Simulations with code JAGUAR

8.1 Characteristics of PETSc solver used in code JAGUAR

In section (5.2), it was said that the JFNK solver from PETSc library will be used to solve the nonlinear system arising from the implicit time discretization. However, although the solver is Jacobian-Free, a lot of Newton and Krylov solvers are available in SNES module of PETSc. In this section, the type of Newton and Krylov solvers and also their characteristics, used with code JAGUAR, will be detailed.

8.1.1 Type of Newton solver

Among the numerous Newton solvers offered in SNES module of PETSc, the one that was selected is the "Newton Line Search" which is the default Newton solver in this module. This solver is used in a Matrix-Free context that has to be specified with the following command line at the execution: *-snes_mf*. Like all the Newton methods, this one has stopping criterions and the user can change their values easily using some command lines. Using the same notations as in paragraph (4.1.1), the defaults values and the command lines to change them are summed up in Table 8.5:

	Default value	Command line
$\epsilon_{Newt,r}$	10^{-8}	*-snes_rtol*
$\epsilon_{Newt,a}$	10^{-50}	*-snes_atol*
η_{Newt}	10^{-8}	*-snes_stol*
$m_{Newt,max}$	50	*-snes_max_it*

Table 8.5: Default values and command lines for the stopping criterions of the Newton Line Search solver of SNES module in PETSc.

8.1.2 Type of Krylov solver

In SNES module, for the Matrix-Free context, the default Krylov solver is GMRES(m_{Res}) and it will be the one used for code JAGUAR. Actually, all the Krylov solvers are in another PETSc module which is called KSP (Krylov SubsPace). When using the JFNK method of SNES module, the KSP module is automatically set and by default it calls the GMRES(m_{Res}) solver. If the user wants to have another Krylov methods, he just has to use the command line *ksp_type* with the name of the method found in PETSc documentation [6]. The defaults values for the stopping criterions and also for some characteristics of the GMRES(m_{Res}) algorithm are summed up in Table 8.6 along with their associated command line:

	Default value	Command line
$\epsilon_{Kry,r}$	10^{-5}	*-ksp_rtol*
$\epsilon_{Kry,a}$	10^{-50}	*-ksp_atol*
$j_{Kry,max}$	10^{4}	*-ksp_max_it*
m_{Res}	30	*-ksp_gmres_restart*

Table 8.6: Default values and command lines for the GMRES(m_{Res}) solver of KSP module in PETSc.

It has to be mentioned that no preconditioners were used to accelerate the convergence. Actually, in KSP module, a lot of preconditioners are already available such as Jacobi, SOR, LU and ILU. However, these preconditioners are not suitable for a Matrix-Free use because they need the Jacobian matrix explicitly. Thus, a Matrix-Free preconditioner has to be built. One idea is to use the FGMRES (for Flexible-GMRES) algorithm

[41], already available in KSP module, instead of GMRES and do preconditioning with GMRES. This will be the purpose of further developments in the future for JAGUAR.

8.2 Euler and Navier-Stokes equations in 2D

As mentioned in paragraph (1.1.2), JAGUAR is solving the 3D Navier-Stokes equations on unstructured hexahedral grids. However, the implicit time-marching schemes have first been tested on 2D test cases. Thus, the flows of interest in this report are assumed to be adequately modeled either by the 2D Euler equations (first test case) or the 2D Navier-Stokes equations (second test case).

The Euler equations describe the most general flow configuration for a non-viscous, non-heat conducting fluid. They are formed by the combination of three conservation laws, namely the conservation of mass, the momentum equations and conservation of energy. These three conservation laws are combined with the equation of state for perfect gases to form a closed system of equations.

The Navier-Stokes equations extend the mathematical model of the Euler equations to account for viscous effects and heat conduction in the fluid. These equations can be put in the form of Eq. (2.26) where the fluxes can be split into a convective (indexed "C") and a diffusive (indexed "D") part:

$$U = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho E \end{pmatrix},\ F = F_C - F_D,\ G = G_C - G_D,\ H = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{8.112}$$

with

$$F_C = \begin{pmatrix} \rho u \\ P + \rho u^2 \\ \rho u v \\ u\left(P + \rho E\right) \end{pmatrix},\ G_C = \begin{pmatrix} \rho v \\ \rho u v \\ P + \rho v^2 \\ v\left(P + \rho E\right) \end{pmatrix} \tag{8.113}$$

$$F_D = \mu \begin{pmatrix} 0 \\ 2u_x - \frac{2}{3}\left(u_x + v_y\right) \\ v_x + u_y \\ u\left(2u_x - \frac{2}{3}\left(u_x + v_y\right)\right) + v\left(v_x + u_y\right) + \frac{C_p}{P_r} T_x \end{pmatrix},\ G_D = \mu \begin{pmatrix} 0 \\ v_x + u_y \\ 2v_y - \frac{2}{3}\left(u_x + v_y\right) \\ v\left(2v_y - \frac{2}{3}\left(u_x + v_y\right)\right) + u\left(v_x + u_y\right) + \frac{C_p}{P_r} T_y \end{pmatrix} \tag{8.114}$$

where ρ is the fluid density, u and v are respectively the x-velocity and y-velocity of the fluid, P is the static pressure, T is the local temperature, μ is the dynamic viscosity, C_p is the specific heat at constant pressure, P_r is the Prandtl number of the fluid and E is the total energy per unit mass. The notations $._x$ and $._y$ mean respectively $\frac{\partial .}{\partial x}$ and $\frac{\partial .}{\partial y}$.

This system is closed by the equation of state for perfect gases which relates static pressure with the conserved variables:

$$P = (\gamma - 1)\left[\rho E - \frac{\rho\left(u^2 + v^2\right)}{2}\right] \tag{8.115}$$

The Euler equations correspond to the special case where $\mu = 0$ meaning that $F_D = (0,0,0,0)^T$ and $G_D = (0,0,0,0)^T$.

8.3 Vortex transported by an uniform flow in 2D

This test case corresponds to problem $C1.6.$ presented in the review paper [55]. It aims at testing a high-order method capability to preserve vorticity in an unsteady inviscid flow.

8.3.1 Governing equations

The governing equations are the unsteady 2D Euler equations with a constant ratio of specific heats $\gamma = 1.4$ and a gas constant $R_{gas} = 287.15$ J.kg^{-1}.K^{-1}.

8.3.2 Flow conditions

The domain is first initialized with an uniform flow of pressure P_∞, temperature T_∞, a given Mach number M_∞ and an initial vortex profile well-chosen which will be described below. In this case, the unperturbed flow has a horizontal speed given by:

$$U_\infty = M_\infty \sqrt{\gamma R_{gas} T_\infty} \tag{8.116}$$

The initial vortex is a 2D Lamb-Oseen inviscid vortex characterized by the following stream function [26]:

$$\psi(x,y) = \Gamma \exp\left(-\frac{(x-x_c)^2 + (y-y_c)^2}{2R_c^2}\right) \tag{8.117}$$

where Γ is the vortex strength, R_c is the vortex radius and (x_c, y_c) are the coordinates of the vortex center. The vortex defined by Eq. (8.117) is very interesting because it is a solution of the 2D Euler equations. The velocity and pressure fields can be obtained and for an uniform flow over x axis they write:

$$u = U_\infty + \frac{\partial \psi}{\partial y} = U_\infty - \Gamma \frac{y-y_c}{R_c^2} \exp\left(-\frac{(x-x_c)^2 + (y-y_c)^2}{2R_c^2}\right) \tag{8.118a}$$

$$v = -\frac{\partial \psi}{\partial x} = \Gamma \frac{x-x_c}{R_c^2} \exp\left(-\frac{(x-x_c)^2 + (y-y_c)^2}{2R_c^2}\right) \tag{8.118b}$$

$$P = P_\infty - \frac{\rho \Gamma^2}{2R_c^2} \exp\left(-\frac{(x-x_c)^2 + (y-y_c)^2}{R_c^2}\right) \tag{8.118c}$$

Thus, theoretically, the vortex is transported by the flow without any physical dissipation. The exact solution for any time t_0 can be obtained by changing the vortex center coordinates as follows:

$$x_c(t_0) = x_c(0) + U_\infty t_0 \quad \text{and} \quad y_c(t_0) = y_c(0) \tag{8.119}$$

That is why, this test case is well suited to study numerical dissipation and dispersion of numerical schemes because if dissipation or/and dispersion occurs, it has to come from numerical dissipation or/and dispersion.

8.3.3 Geometry and mesh

The computational domain is rectangular with $(x,y) \in [0, L_x] \times [0, L_y]$, where $L_x = L_y = 0.1$ m and the mesh is composed of $N_{cells} = 4096$ cells. For all the simulations, the order of interpolation polynomial was set to $p = 4$ because, according to previous simulations of this test case with JAGUAR, this value was associated with very few numerical dissipation and dispersion from the spatial scheme. Consequently, the dissipation regarding several time-marching schemes, especially implicit ones, could be studied. This value of p sets the number of DoF to $4096 \times (4+1)^2 = 102400$.

8.3.4 Boundary conditions

Translational periodic boundary conditions are imposed for the left/right and top/bottom boundaries respectively, to enable the vortex to turn many times in the domain.

8.3.5 Testing conditions

Initially, the vortex is positioned at the center of the computational domain:

$$x_c(0) = 0.05 \text{ m} \quad \text{and} \quad y_c(0) = 0.05 \text{ m} \tag{8.120}$$

with a radius $R_c = 0.005$ m and a strength $\Gamma = 34.728$ m^2.s^{-1}. The uniform flow is initialized with:

$$P_\infty = 10^5 \text{ Pa}, \quad T_\infty = 300 \text{ °K} \quad \text{and} \quad M_\infty = 0.5 \tag{8.121}$$

With this values, Eq. (8.116) gives $U_\infty = 174$ m.s^{-1} for the speed of the unperturbed flow. As explained in paragraph (8.3.2), the vortex should be transported by the flow at speed U_∞. Thus, because there are periodic boundary conditions, the time needed for the vortex to perform one turn of the domain is:

$$T = \frac{L_x}{U_\infty} = 5.75 \times 10^{-4} \text{ s} \tag{8.122}$$

A representation of the initial flow with the vortex can be found in appendix F. Finally, for the JFNK method, the values of the parameters for the nonlinear and linear solver used in all simulations were kept to their default value written in Tables 8.5 and 8.6 except for $\epsilon_{Newt,a}$ and $\epsilon_{Kry,a}$ which were set respectively to 10^{-5} and 10^{-6}. These values were used because they gave approximately the same results as computations with the default values but with a smaller computation time. It should be mentioned that all the computations were done on only one core.

8.3.6 Comparison between time-marching schemes

Simulations over five periods (physical time $T_f = 5{\times}T$) where done with the implicit midpoint, the ILDDRK(3,4) and the ERK(6,2)LDLD schemes for different CFL values. The explicit scheme was set with $CFL = 0.5$ and implicit schemes were used with $CFL = 5$ to see if greater time steps were possible. The implicit midpoint was also tested with $CFL = 0.5$ to compare its numerical dissipation with the case $CFL = 5$. The profile $P\,(x = 0.05, y)$ obtained with these schemes are compared with the analytic solution and the results are presented in Figures 8.17 and 8.18 where a zoom around the pressure peak is done:

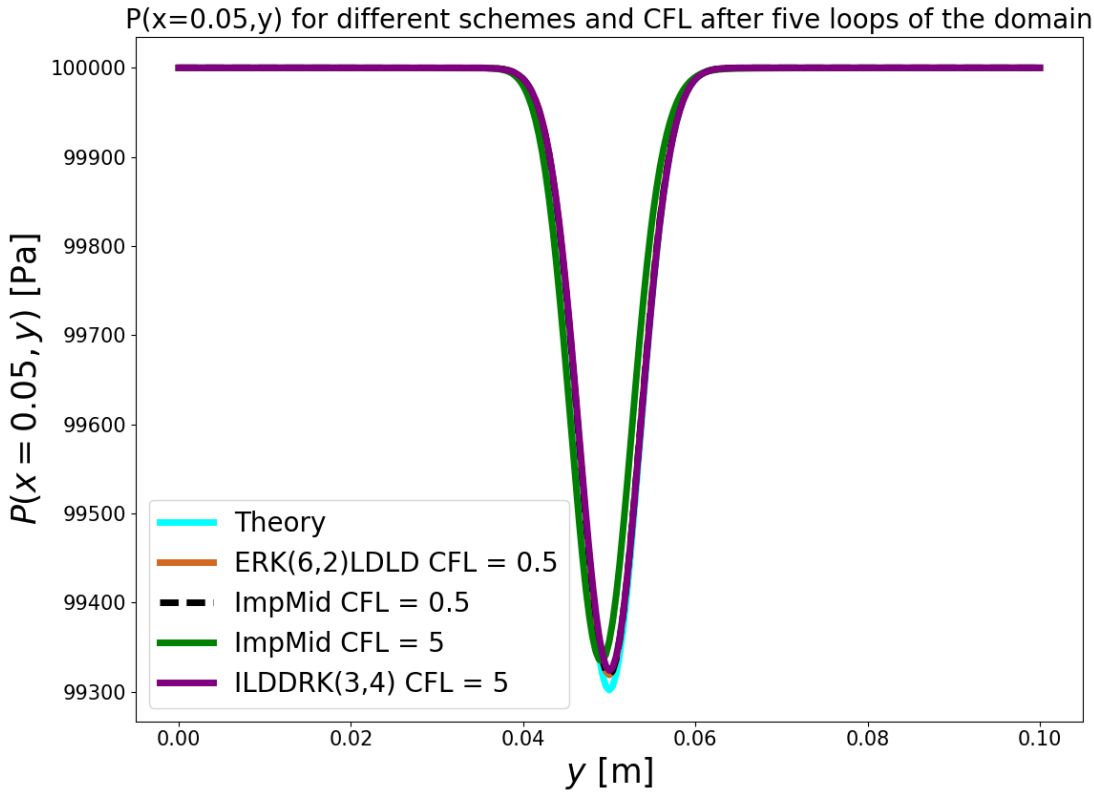


Figure 8.17: Pressure along $x = 0.05\ m$ for implicit midpoint, ILDDRK(3,4) and the ERK(6,2)LDLD schemes.

Results analysis: As expected, all the schemes are dissipating and dispersing. For $CFL = 0.5$ both ERK(6,2)LDLD and implicit midpoint schemes give the same results. However, for $CFL = 5$, the implicit midpoint is dispersive. On the contrary, the ILDDRK(3,4) scheme is few dispersive for $CFL = 5$. Moreover, its dissipation is non-significant compared to the one appearing for the implicit midpoint when $CFL = 0.5$ even though the time step is ten times greater. It is probably due to the low-dissipative and low-dispersive properties of ILDDRK(3,4) scheme. Actually, it seems that the number of stages and also the order of the RK have a big impact on dissipation and dispersion. On Figure 8.19 the results for the seven implicit schemes implemented in JAGUAR are plotted. The SDIRK(2,2) scheme is less dispersive than the implicit midpoint whereas the SDIRK(2,3) and SDIRK(3,3) schemes are few dispersive. The SDIRK(2,3) scheme is the most dissipative. The SDIRK(3,4) is a little better than the SDIRK(3,3) on dissipation showing that the order of the scheme has an impact on dissipation. At the end, the ILDDRK(3,4) scheme seems to be the better one to limit dissipation and dispersion.

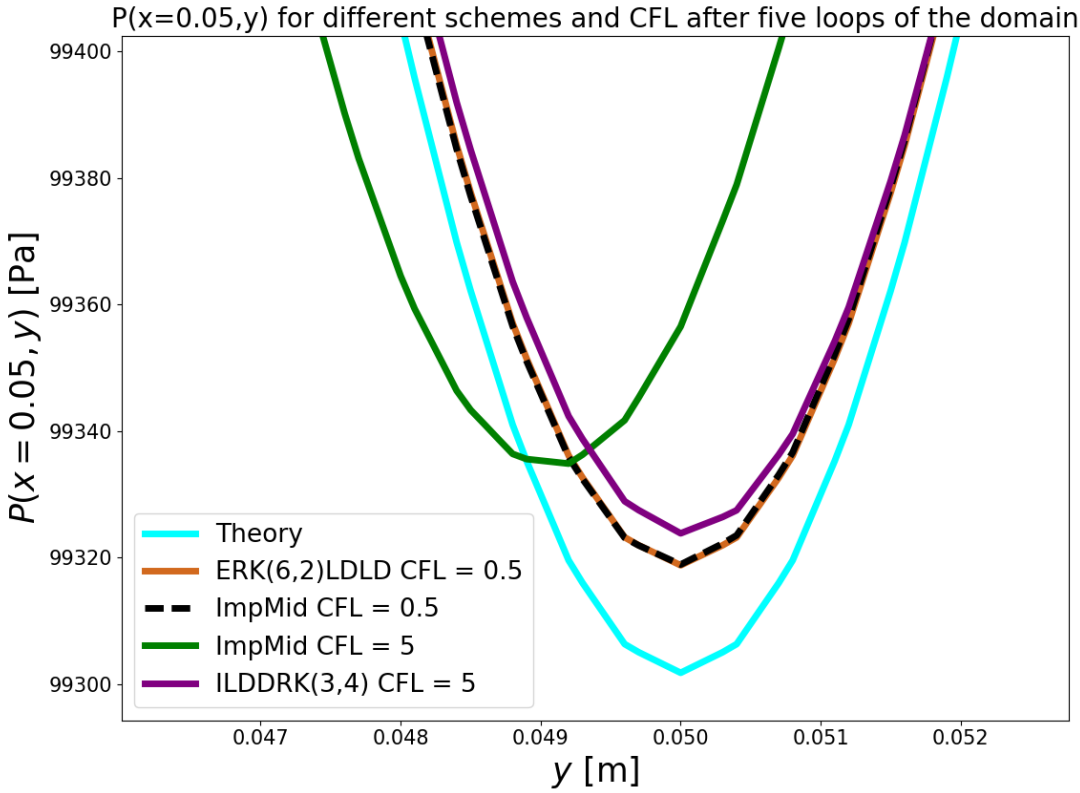


Figure 8.18: Zoom around $y = 0.05\ m$ on pressure profiles along $x = 0.05\ m$.

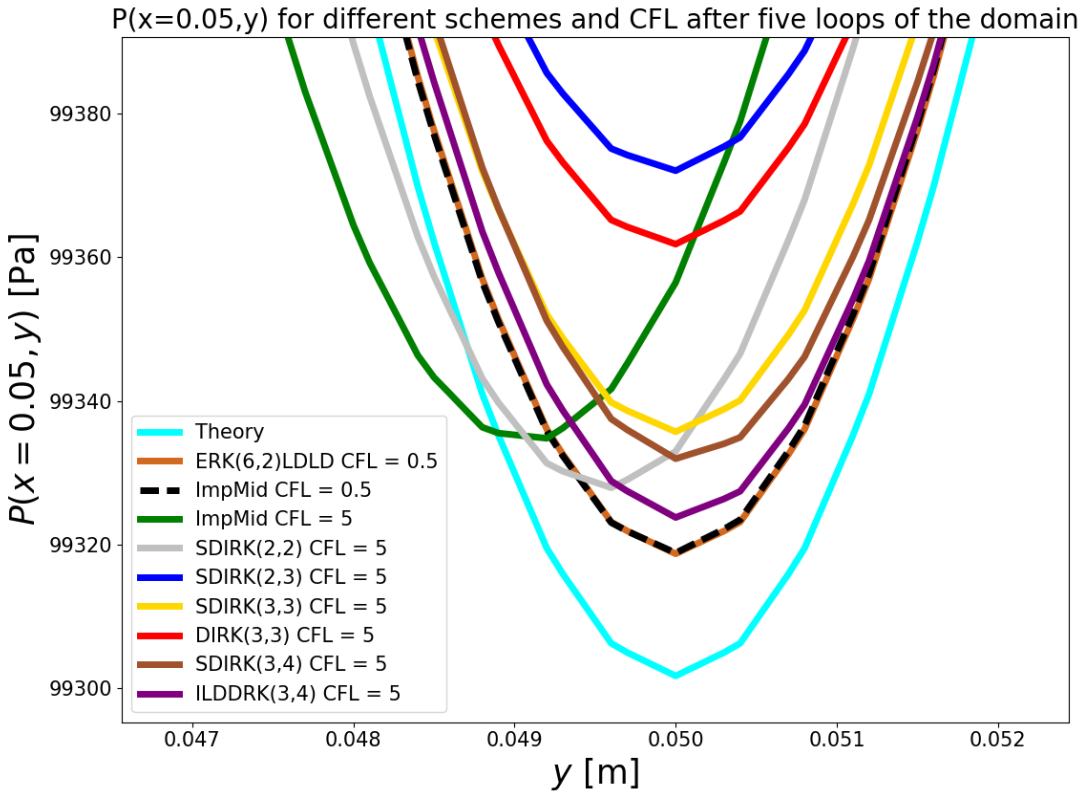


Figure 8.19: Zoom around $y = 0.05\ m$ on pressure profiles along $x = 0.05\ m$ for the seven IRK schemes implemented in JAGUAR.

8.3.7 Performance of implicit schemes

In paragraph (8.3.6), it was shown that JAGUAR was able to handle $CFL > 1$ thanks to the implementation of implicit time-marching schemes. However, it is essential to determine if it is worth it to do implicit time integration rather than an explicit one in terms of CPU time. As said in paragraph (1.2.6), the expected speed up should be between one to two order of magnitude compared to explicit schemes. Thus, several simulations of the 2D transported vortex were performed, at fixed different time steps for the seven implicit schemes implemented in JAGUAR. The computation time and the 2-norm error on pressure field, noted respectively t_{CPU} and $L_2^{abs}(P)$, are picked and they are compared to these values in the case of ERK(6,2)LDLD scheme with a fixed time step set to $\Delta t = 2.8 \times 10^{-7}$ s (corresponding to approximately $CFL = 0.5$). All the simulations were done only during one period T. Moreover, at this time of the internship, the parallelization of the implicit schemes was not done. Therefore simulations over a lot of periods were quite long. Three different time steps were taken for the implicit schemes: 1.12×10^{-6} s ($CFL \approx 2$), 2.24×10^{-6} s ($CFL \approx 4$) and 4.48×10^{-6} s ($CFL \approx 8$). Thus, if the scheme is of order k, the error should increase by a factor of 2^k between each of these time steps. The results are presented in Figures 8.20 and 8.21. For Figure 8.20, all the t_{CPU} values of the seven IRK schemes are divided by $t_{ref} = 378$ s which is the computation time taken by the ERK(6,2)LDLD scheme to perform a one loop simulation. Time steps were also divided by the time step taken for the ERK(6,2)LDLD scheme: $\Delta t_{ref} = 2.8 \times 10^{-7}$ s. All of these values are also presented in Table F.1 to Table F.7 in appendix F.

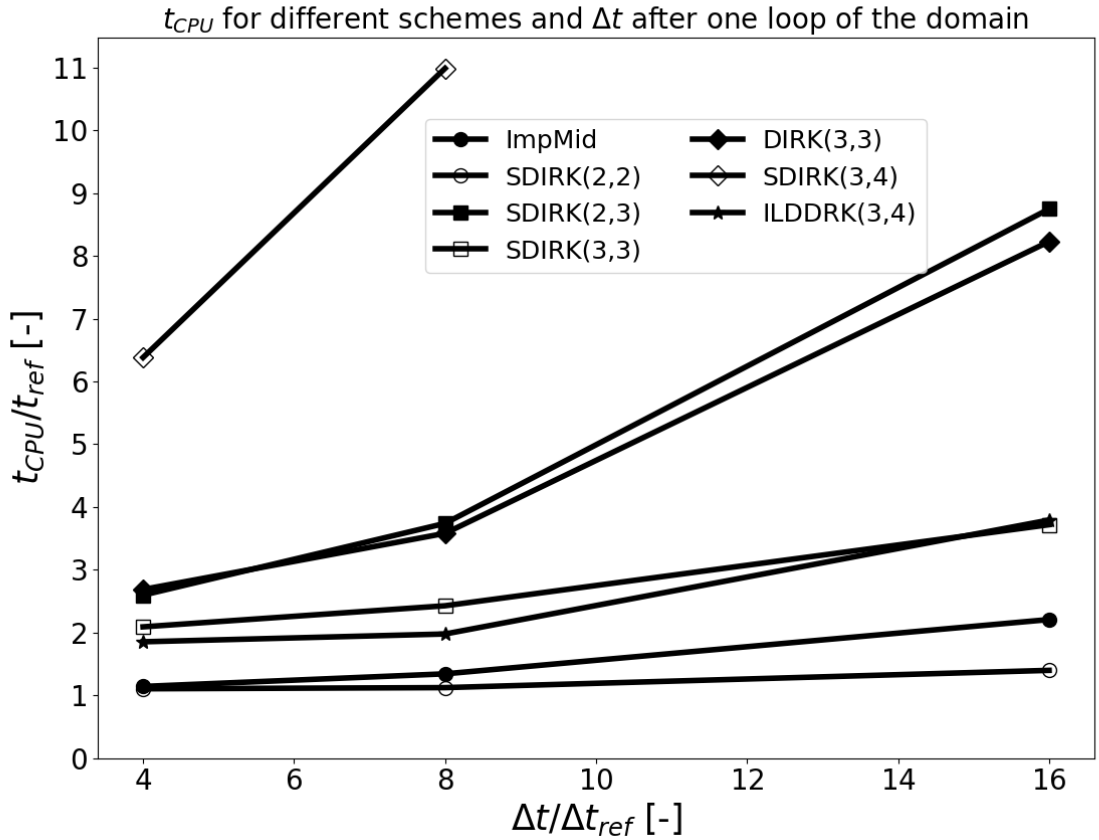


Figure 8.20: Evolution of t_{CPU} versus Δt for the seven IRK schemes implemented in JAGUAR. $t_{ref} = 378$ s and $\Delta t_{ref} = 2.8 \times 10^{-7}$ s.

Results analysis: For the CPU times, all schemes are slower than the explicit one. Moreover, when Δt is higher, t_{CPU} is also higher even though less time iterations are done. Obviously increasing the time step also increases the cost per time iteration as it can be seen from Table F.1 to Table F.7 in appendix F where the number of nonlinear and linear iterations is bigger when Δt is higher. However, it should be counterbalanced by the fact that less time iterations are done which is not the case here. This is a real issue since doing implicit time integration should reduce the computation time compared to explicit integration. The implicit midpoint and the SDIRK(2,2) schemes are the fastest among the implicit schemes but as seen before they are dispersive. The SDIRK(2,2) is even faster than the implicit midpoint although it has two stages. The ILDDRK(3,4) is quite fast despite its three stages: as seen in Table F.7 in appendix F, the number of nonlinear and linear iterations is very small for each stage. Thus, in terms of CPU time, it seems that this scheme is the best one among all the implicit schemes. However, about the errors, all the schemes orders are retrieved but not for

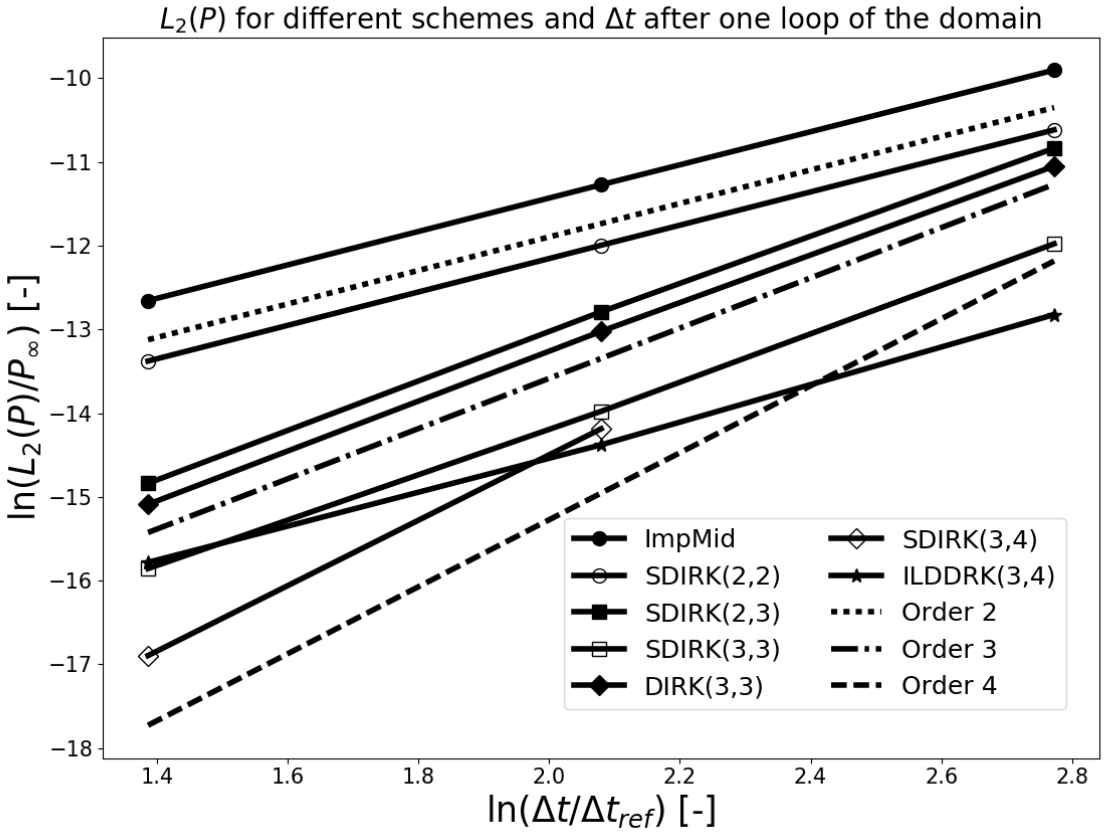


Figure 8.21: Evolution of $L_2^{abs}(P)$ versus Δt for the seven implicit schemes implemented in JAGUAR.

the ILDDRK(3,4) scheme. Actually, according to Figure 8.21, it is of order 2 as the implicit midpoint and the SDIRK(2,2) methods. Moreover, in [33], the authors are also claiming that this ILDDRK(3,4) is of order 2 and they propose other values of the coefficients to have a real ILDDRK(3,4) scheme. Their method was tested in JAGUAR but it failed to converge so no results were obtained to see if this scheme was really of order 4. Actually, when the order is increasing, the convergence is harder (see Table F.6 for SDIRK(3,4) where t_{CPU} is already huge for $\Delta t = 2.24 \times 10^{-6}$ and there was no convergence for $\Delta t = 4.48 \times 10^{-6}$) therefore this might be linked to that observation. Finally, even though the ILDDRK(3,4) seems to be of order 2, it is a good compromise in terms of dissipation, computation time end errors.

The fact that implicit schemes are not faster than explicit ones for the convected vortex is not surprising. Moreover, some improvements are still possible such as the use of preconditioners to accelerate the convergence. Actually, because there is no viscosity here, the time steps can still be sufficient for explicit time-marching schemes without stability issues. That is why, a viscous test case will be considered to see if implicit schemes are still less efficient than explicit ones.

8.4 Flow over a NACA0012 airfoil

At start, the objective was to simulate the NACA0012 test case corresponding to problem $C1.3$. presented in the review paper [55]. This test case aims at testing a high-order method for the computation of external flow with a high-order curved boundary representation. However, the boundary condition on the wing is an adiabatic viscous wall and in JAGUAR only the isothermal viscous wall is implemented. Thus, the results obtained with JAGUAR cannot be compared to the one from [55] but it is still possible to compare the results given by the implicit time integration schemes with the one given by the ERK(6,2)LDLD scheme.

8.4.1 Governing equations

The governing equations are the 2D Navier-Stokes equations still with $\gamma = 1.4$ and a Prandtl number $P_r = 0.72$. A constant dynamic viscosity will be assumed. Its value will be found using the flow conditions that are

considered. The gas constant is still $R_{gas} = 287.15$ J.kg^{-1}.K^{-1}.

8.4.2 Flow conditions

The flow conditions are those of a subsonic viscous flow problem over the NACA0012 airfoil at an angle of attack $\alpha = 1°$ and a Reynolds number, based on the chord length, $R_e = 5000$.

8.4.3 Geometry and mesh

The geometry is a NACA0012 airfoil with the following analytical expression for the airfoil surface:

$$y = \pm 0.6 \left(0.2969\sqrt{x} - 0.1260x - 0.3516x^2 + 0.2843x^3 - 0.1015x^4\right) \tag{8.123}$$

where $x \in [0, 1]$. Thus, the chord length is equal to one. The airfoil defined by Eq. (8.123) has not a zero thickness at the trailing edge since $y(1) = 1.26 \times 10^{-3} \neq 0$.

Note: Various ways exist in the literature to modify this definition such that the trailing edge has a zero thickness. A simple one is to set the coefficient of x^4 to 0.1036 instead of 0.1015.

The mesh is composed of $N_{cells} = 3611$ triangular cells and is extended about 80 chords away from the airfoil. It means that the spatial domain is rectangular with $(x, y) \in [-L_x, L_x] \times [-L_y, L_y]$, where $L_x = L_y = 80$ m. All the simulations were done with $p = 3$ ensuring a fourth-order spatial accuracy. In this case, the number of DoF was: $3611 \times (3 + 1)^2 = 57776$.

8.4.4 Boundary conditions

The airfoil is treated as an isothermal viscous wall since the adiabatic viscous wall was not implemented yet in code JAGUAR.

8.4.5 Testing conditions

The uniform flow is initialized with:

$$P_\infty = 101325 \text{ Pa}, \quad T_\infty = 293.15 \text{ °K} \quad \text{and} \quad M_\infty = 0.5 \tag{8.124}$$

In that case, Eq. (8.116) gives $U_\infty = 172$ m.s^{-1} and the density is given by the ideal gas law:

$$\rho_\infty = \frac{P_\infty}{R_{gas} T_\infty} = 1.2 \text{ kg.m}^{-3} \tag{8.125}$$

Finally, as mentioned in paragraph (8.4.1), the dynamic viscosity can be computed to match with the Reynolds value wanted:

$$\mu = \frac{\rho_\infty U_\infty c}{R_e} = 4.128 \times 10^{-2} \text{ kg.m}^{-1}.\text{s}^{-1} \tag{8.126}$$

where $c = 1$ m is the chord of the airfoil here. About the JFNK method, the values for the stopping criterions were set to $\epsilon_{Newt,a} = \epsilon_{Kry,a} = 10^{-4}$ which appeared to be sufficient for precision. Finally, like for the convected vortex, it should be mentioned that all the calculations were done on only one core.

8.4.6 Results

In section (8.3), the ILDDRK(3,4) scheme was found to be a good compromise in terms of CPU time and precision. Thus, the results given by this scheme will be compared to the ones from ERK(6,2)LDLD scheme for a simulation with a physical time $T_f = 0.01$ s. About the CFL, it was set to 0.5 for the explicit case and to 5 for the implicit one. The values of the drag coefficient C_d, the lift coefficient C_l and the computation time that were obtained are presented in Table 8.7. As it will be shown, the ILDDRK(3,4) was not faster than ERK(6,2)LDLD scheme but it was found that SDIRK(2,2) was. Therefore, results for the SDIRK(2,2) scheme are also presented in Table 8.7.

Results analysis: For the C_d values, all the schemes are giving the same results which is a good point. However, for the C_l values, there are strong differences between explicit and implicit time integration schemes

Scheme	C_d	C_l	t_{CPU} (s)
ERK(6,2)LDLD	7.60×10^{-3}	8.90×10^{-7}	3834
ILDDRK(3,4)	7.60×10^{-3}	1.41×10^{-6}	5808
SDIRK(2,2)	7.60×10^{-3}	1.67×10^{-6}	3498

Table 8.7: Values of C_d, C_l and t_{CPU} for ERK(6,2)LDLD, ILDDRK(3,4) and SDIRK(2,2) that were obtained for the NACA0012 test case when $T_f = 0.01$.

and even in between implicit schemes. It probably comes from dissipation and dispersion already observed in the convected vortex test case. About the computation times, the ILDDRK(3,4) scheme is very costly compared to ERK(6,2)LDLD whereas the SDIRK(2,2) is not. It means that when there are viscous effects and a smaller time step, an implicit scheme could be faster than an explicit one. This is very interesting since implicit time-marching methods aim at reducing the computation time for viscous flows where the time step is very small if explicit time-marching schemes are used. The remaining issue is that the SDIRK(2,2) was found to be very dissipative and dispersive compared to ILDDRK(3,4) or SDIRK(3,3) for instance. Therefore, some efforts have to be made in order to reduce the computation time for the less dissipative and dispersive schemes. One idea is the use of preconditioners for the JFNK solver which will be the topic of further developments in JAGUAR.

9 Conclusion and perspectives

As seen in Table 1.1, high-order spatial discretization methods are more and more coupled with an implicit time discretization. It is due to the fact that such methods suffer from very restrictive stability conditions when they are used with an explicit time integration. For the SD method, if implicit time-marching schemes are used, the expected speed-up is between one to two orders of magnitude compared to explicit ones [47]. Thus, the high-order SD solver on unstructured hexahedral grids, JAGUAR, developed at CERFACS has to use implicit time integration.

Before developing into JAGUAR, some studies were made on the 1D-advection and 1D-diffusion equations discretized by the SD method and advanced in time with implicit time-marching schemes. Algorithms for computing analytically and numerically the Jacobian of the residual were derived for these two equations. It turns out that the numerical computation of the Jacobian matrix was very costly. That is why a Jacobian-Free-Newton-Krylov (JFNK) approach was considered to avoid the explicit computation of the Jacobian. The JFNK solver from PETSc library was used and firstly implemented for 1D-advection and 1D-diffusion in my model code and then in code JAGUAR.

For the 1D-cases, the results were quite encouraging since the JFNK method was faster than the inexact Newton with numerical computation of the Jacobian. Following those observations, seven implicit time integration schemes have been implemented in JAGUAR. At first, they were tested on the isentropic vortex transported by an uniform flow test case. Although all of these schemes could handle $CFL > 1$, they were slower than explicit time-marching schemes. Actually, the increasing cost of a time iteration was not counterbalanced with the fact that less time iterations were done when the time step was higher. Dissipation and dispersion properties were studied: the ILDDRK(3,4) was the less dissipative and dispersive among the seven time-marching schemes. However, this scheme was found to be of order two rather than order four but it is a good compromise in terms of dissipation and dispersion properties and also in computation time. The convected vortex is an inviscid test case: there were no walls and consequently the time step was not very small even for explicit time integration schemes. A viscous test case was studied to see if better performances can be seen when walls are considered. Thus, a NACA0012 test case was simulated with the ERK(6,2)LDLD, the ILDDRK(3,4) and the SDIRK(2,2) schemes. The last one was choosen because it was the fastest among implicit schemes in the convected vortex test case. The three schemes gave the same results for the drag around the airfoil but for the lift they shew huge differences. In terms of CPU time, the ILDDRK(3,4) scheme is still slower than ERK(6,2)LDLD but the SDIRK(2,2) was faster of about ten percent.

Thus, the results were very far to what was expected in terms of speed-up. However, there was no use of preconditioners for solving the linear systems with PETSc: some improvements can be hoped in this way.

Bibliography

[1] L-stability. https://en.wikipedia.org/wiki/L-stability.

[2] Legendre polynomial. https://en.wikipedia.org/wiki/Legendre_polynomials.

[3] List of Runge-Kutta methods. https://en.wikipedia.org/wiki/List_of_Runge%E2%80%93Kutta_methods.

[4] Optimised Runge-Kutta time integration for the Spectral Difference method. https://www.grc.nasa.gov/hiocfd/wp-content/uploads/sites/22/AIAA_Aviation_2017_BALAN.pdf.

[5] R. Alexander. Diagonally implicit Runge-Kutta methods for stiff ODE's. *SIAM Journal on Numerical Analysis*, 14(6):1006–1021, 1977.

[6] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. Gropp, D. Kaushik, et al. Petsc users manual revision 3.8. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2017.

[7] F. Bassi, L. Botti, A. Colombo, A. Crivellini, A. Ghidoni, A. Nigro, and S. Rebay. Time integration in the discontinuous Galerkin code MIGALE-Unsteady problems. In *IDIHOM: Industrialization of High-Order Methods-A Top-Down Approach*, pages 205–230. Springer, 2015.

[8] F. Bassi, L. Botti, A. Colombo, and S. Rebay. Agglomeration based discontinuous Galerkin discretization of the Euler and Navier-Stokes equations. *Computers & Fluids*, 61:77–85, 2012.

[9] P. Birken, G. Gassner, M. Haas, and C.-D. Munz. Preconditioning for modal discontinuous Galerkin methods for unsteady 3D Navier-Stokes equations. *Journal of Computational Physics*, 240:20–35, 2013.

[10] C. Bogey and C. Bailly. A family of low dispersive and low dissipative explicit schemes for flow and noise computations. *Journal of Computational physics*, 194(1):194–214, 2004.

[11] P. N. Brown and Y. Saad. Hybrid Krylov methods for nonlinear systems of equations. *SIAM Journal on Scientific and Statistical Computing*, 11(3):450–481, 1990.

[12] J. C. Butcher. *Numerical methods for ordinary differential equations*. John Wiley & Sons, 2016.

[13] M. H. Carpenter and C. A. Kennedy. Fourth-order 2N-storage Runge-Kutta schemes. 1994.

[14] F. Chipman. A-stable Runge-Kutta processes. *BIT Numerical Mathematics*, 11(4):384–388, 1971.

[15] B. Cockburn and C.-W. Shu. Runge-Kutta discontinuous Galerkin methods for convection-dominated problems. *Journal of scientific computing*, 16(3):173–261, 2001.

[16] M. Crouzeix. *Sur l'approximation des équations différentielles opérationnelles linéaires par des méthodes de Runge-Kutta*. PhD thesis, 1975.

[17] R. S. Dembo, S. C. Eisenstat, and T. Steihaug. Inexact Newton methods. *SIAM Journal on Numerical analysis*, 19(2):400–408, 1982.

[18] M. Duarte, R. Dobbins, and M. Smooke. High order implicit time integration schemes on multiresolution adaptive grids for stiff PDEs. *arXiv preprint arXiv:1604.00355*, 2016.

[19] S. Gérald. *Méthode de Galerkin Discontinue et intégrations explicites-implicites en temps basées sur un découplage des degrés de liberté. Applications au système des équations de Navier-Stokes*. PhD thesis, Université Pierre et Marie Curie-Paris VI, 2013.

[20] A. Ghidoni, A. Colombo, S. Rebay, and F. Bassi. Simulation of the transitional flow in a low pressure gas turbine cascade with a high-order discontinuous Galerkin method. *Journal of Fluids Engineering*, 135(7):071101, 2013.

[21] N. Gindrier. Développement d'une méthode de suivi lagrangien de particules dans un code CFD d'ordre élevé pour la LES. Working note, CERFACS, Toulouse, France, 8 2018.

[22] R. Hartmann, J. Held, T. Leicht, and F. Prill. Discontinuous Galerkin methods for computational aerodynamics—3D adaptive flow simulation with the DLR PADGE code. *Aerospace Science and Technology*, 14(7):512–519, 2010.

[23] D. A. Knoll and D. E. Keyes. Jacobian-free Newton-Krylov methods: a survey of approaches and applications. *Journal of Computational Physics*, 193(2):357–397, 2004.

[24] D. A. Kopriva. A conservative staggered-grid Chebyshev multidomain method for compressible flows. II. A semi-structured method. *Journal of computational physics*, 128(2):475–488, 1996.

[25] N. Kroll. ADIGMA: A european project on the development of adaptive higher order variational methods for aerospace applications. In *47th AIAA Aerospace Sciences Meeting including The New Horizons Forum and Aerospace Exposition*, page 176, 2009.

[26] M. Kuzmin. Spectral Difference method for the Euler equations on unstructured grids. Working note, Université de Toulouse - ISAE - CERFACS, 9 2012.

[27] M. Lemesle. Analysis of several extensions for the Spectral Difference method to handle discontinuity, Ref.: WN/CFD/14/62. September 5, 2014.

[28] W. J. Leong, M. A. Hassan, and M. W. Yusuf. A matrix-free quasi-Newton method for solving large-scale nonlinear systems. *Computers & Mathematics with Applications*, 62(5):2354–2363, 2011.

[29] Y. Liu, M. Vinokur, and Z. Wang. Spectral Difference method for unstructured grids I: basic formulation. *Journal of Computational Physics*, 216(2):780–801, 2006.

[30] A. N. Lowan, N. Davids, and A. Levenson. Table of the zeros of the Legendre polynomials of order 1-16 and the weight coefficients for Gauss' mechanical quadrature formula. *Bulletin of the American Mathematical Society*, 48(10):739–743, 1942.

[31] F. M. Moreira, E. Jourdan, C. Breviglieri, A. R. Aguiar, and J. L. F. Azevedo. Implicit Spectral Difference Method Solutions of Compressible Flows Considering High-Order Meshes. In *46th AIAA Fluid Dynamics Conference*, page 3352, 2016.

[32] A. Najafi-Yazdi and L. Mongeau. A low-dispersion and low-dissipation implicit Runge-Kutta scheme. *Journal of Computational Physics*, 233:315–323, 2013.

[33] F. Nazari, A. Mohammadian, and M. Charron. High-order low-dissipation low-dispersion diagonally implicit Runge-Kutta schemes. *Journal of Computational Physics*, 286:38–48, 2015.

[34] H. Owhadi and L. Zhang. Gamblets for opening the complexity-bottleneck of implicit schemes for hyperbolic and parabolic ODEs/PDEs with rough coefficients. *Journal of Computational Physics*, 347:99–128, 2017.

[35] M. Parsani, G. Ghorbaniasl, and C. Lacor. Validation and application of an high-order Spectral Difference method for flow induced noise simulation. *Journal of Computational Acoustics*, 19(03):241–268, 2011.

[36] M. Parsani, K. Van den Abeele, and C. Lacor. Implicit LU-SGS time integration algorithm for high-order spectral volume method with p-multigrid strategy. In *West-East High-Speed Flow Field Conference, Moscow, Russia*, 2007.

[37] N. Qin and B. E. Richards. Sparse quasi-Newton method for Navier-Stokes solution. In *Proceedings of the Eighth GAMM-Conference on Numerical Methods in Fluid Mechanics*, pages 474–483. Springer, 1990.

[38] W. H. Reed and T. Hill. Triangular mesh methods for the neutron transport equation. Technical report, Los Alamos Scientific Lab., N. Mex.(USA), 1973.

[39] J. K. Reid. On the method of conjugate gradients for the solution of large sparse systems of linear equations. In *Pro. the Oxford conference of institute of mathematics and its applications*, pages 231–254, 1971.

[40] F. Renac, M. de la Llave Plata, E. Martin, J.-B. Chapelier, and V. Couaillier. Aghora: a high-order DG solver for turbulent flow simulations. In *IDIHOM: Industrialization of High-Order Methods-A Top-Down Approach*, pages 315–335. Springer, 2015.

[41] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM Journal on Scientific Computing*, 14(2):461–469, 1993.

[42] Y. Saad. *Iterative methods for sparse linear systems*, volume 82. siam, 2003.

[43] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

[44] P. Schrooyen, L. Arbaoui, J. Cagnone, N. Poletz, and K. Hillewaert. A high-order Extended Discontinuous Galerkin Method to Treat Hydrodynamics Problems.

[45] P. Schrooyen, K. Hillewaert, T. E. Magin, and P. Chatelain. Fully implicit integrated approach for the numerical simulation of aerothermal flows through and around ablative thermal protection systems. In *8th European Symposium on Aerothermodynamics for Space Vehicles*, 2015.

[46] D. Stanescu and W. Habashi. 2N-storage low dissipation and dispersion Runge-Kutta schemes for computational acoustics. *Journal of Computational Physics*, 143(2):674–681, 1998.

[47] Y. Sun, Z. Wang, and Y. Liu. Efficient implicit non-linear LU-SGS approach for viscous flow computation using high-order Spectral Difference method. In *18th AIAA Computational Fluid Dynamics Conference*, page 4322, 2007.

[48] Y. Sun, Z. Wang, Y. Liu, and C. Chen. Efficient implicit LU-SGS algorithm for high-order Spectral Difference method on unstructured hexahedral grids. In *45th AIAA Aerospace Sciences Meeting and Exhibit*, page 313, 2007.

[49] Y. Sun, Z. J. Wang, and Y. Liu. High-order multidomain spectral difference method for the Navier-Stokes equations on unstructured hexahedral grids. *Communications in Computational Physics*, 2(2):310–333, 2007.

[50] K. Van den Abeele, C. Lacor, and Z. J. Wang. On the connection between the spectral volume and the spectral difference method. *Journal of Computational Physics*, 227(2):877–885, 2007.

[51] K. Van den Abeele, M. Parsani, and C. Lacor. An implicit spectral difference Navier-Stokes solver for unstructured hexahedral grids. In *47th AIAA Aerospace Sciences Meeting including The New Horizons Forum and Aerospace Exposition*, page 181, 2009.

[52] J. Vanharen, G. Puigt, X. Vasseur, J.-F. Boussuge, and P. Sagaut. Revisiting the spectral analysis for high-order spectral discontinuous methods. *Journal of Computational Physics*, 337:379–402, 2017.

[53] Z. J. Wang. Spectral (finite) volume method for conservation laws on unstructured grids. basic formulation: Basic formulation. *Journal of Computational Physics*, 178(1):210–251, 2002.

[54] Z. J. Wang. *Adaptive high-order methods in computational fluid dynamics*, volume 2. World Scientific, 2011.

[55] Z. J. Wang, K. Fidkowski, R. Abgrall, F. Bassi, D. Caraeni, A. Cary, H. Deconinck, R. Hartmann, K. Hillewaert, H. T. Huynh, et al. High-order CFD methods: current status and perspective. *International Journal for Numerical Methods in Fluids*, 72(8):811–845, 2013.

[56] Z. J. Wang and Y. Liu. Spectral (finite) volume method for conservation laws on unstructured grids: II. Extension to two-dimensional scalar equation. *Journal of Computational Physics*, 179(2):665–697, 2002.

[57] Z. J. Wang, Y. Liu, G. May, and A. Jameson. Spectral Difference method for unstructured grids II: extension to the Euler equations. *Journal of Scientific Computing*, 32(1):45–71, 2007.

[58] J. Williamson. Low-storage Runge-Kutta schemes. *Journal of Computational Physics*, 35(1):48–56, 1980.

Appendices

Appendix A

Coefficients of implicit Runge-Kutta schemes used in JAGUAR

A convenient way to represent a Runge-Kutta scheme is the coefficient table suggested by Butcher [12]:

$\boldsymbol{c}$	$\boldsymbol{A}$
	$\boldsymbol{b}^T$

Table A.1: General form of a Butcher's tableau

where matrix $\boldsymbol{A}$ and vectors $\boldsymbol{b}$ and $\boldsymbol{c}$ introduced in section (3.1) define the RK method.

Butcher's tableau for implicit midpoint method

The implicit midpoint method is of second-order time accuracy. It is the simplest method in the class of collocation methods known as the Gauss methods [3]. There is only one stage here and its Butcher's tableau is given by Table A.2:

$\frac{1}{2}$	$\frac{1}{2}$
	1

Table A.2: Butcher's tableau of implicit midpoint method

Butcher's tableau for SDIRK(2,2) method

The Singly Diagonally Implicit Runge-Kutta of second-order with two stages is an implicit RK method where the elements in the diagonal of matrix $\boldsymbol{A}$ are all equal to the same value. Here, by denoting $\lambda = \frac{2-\sqrt{2}}{2}$, its Butcher's tableau is given by Table A.3 [18]:

λ	λ	0
$1-\lambda$	$1-2\lambda$	λ
	$\frac{1}{2}$	$\frac{1}{2}$

Table A.3: Butcher's tableau of SDIRK(2,2) method

This scheme is L-stable [18].

Butcher's tableau for SDIRK(2,3) method

The Singly Diagonally Implicit Runge-Kutta of third-order with two stages is an implicit RK method. Its Butcher's tableau is the same as SDIRK(2,2) found in Table A.3 but with $\lambda = \frac{1}{2}\left(1 + \frac{1}{\sqrt{3}}\right)$ [16][5]. This scheme is only A-stable [18].

Butcher's tableau for SDIRK(3,3) method

The Singly Diagonally Implicit Runge-Kutta of third-order with three stages which was implemented in JAGUAR is the one from Butcher's book [12]. With $\lambda = 0.4358665215$, its Butcher's tableau is given by Table A.4:

λ	λ	0	0
$\frac{1+\lambda}{2}$	$\frac{1-\lambda}{2}$	λ	0
1	$\frac{-6\lambda^2+16\lambda-1}{4}$	$\frac{6\lambda^2-20\lambda+5}{4}$	λ
	$\frac{-6\lambda^2+16\lambda-1}{4}$	$\frac{6\lambda^2-20\lambda+5}{4}$	λ

Table A.4: Butcher's tableau of SDIRK(3,3) method

This scheme is L-stable [34].

Butcher's tableau for DIRK(3,3) method

The Diagonally Implicit Runge-Kutta of third-order with three stages which was implemented in JAGUAR is the one from this paper [34]. Its Butcher's tableau is given by Table A.5:

0.0585104413419415	0.0585104413426586	0	0
0.8064574322792799	0.0389225469556698	0.7675348853239251	0
0.2834542075672883	0.1613387070350185	-0.5944302919004032	0.7165457925008468
	0.1008717264855379	0.4574278841698629	0.4417003893445992

Table A.5: Butcher's tableau of DIRK(3,3) method

This scheme is L-stable [34].

Butcher's tableau for SDIRK(3,4) method

The Singly Diagonally Implicit Runge-Kutta of fourth-order with three stages is the one from Crouzeix [16]. Its Butcher's tableau is given by Table A.6 [5]:

$\frac{1+\alpha}{2}$	$\frac{1+\alpha}{2}$	0	0
$\frac{1}{2}$	$\frac{-\alpha}{2}$	$\frac{1+\alpha}{2}$	0
$\frac{1-\alpha}{2}$	$1+\alpha$	$-(1+2\alpha)$	$\frac{1+\alpha}{2}$
	$\frac{1}{6\alpha^2}$	$1-\frac{1}{3\alpha^2}$	$\frac{1}{6\alpha^2}$

Table A.6: Butcher's tableau of SDIRK(3,4) method

where $\alpha = \frac{2}{\sqrt{3}}\cos\left(\frac{\pi}{18}\right)$. This scheme is A-stable [32].

Butcher's tableau for ILDDRK(3,4) method

The Implicit Low-Dispersion and low-Dissipation Runge-Kutta of fourth-order with three stages is a Diagonally Implicit RK (DIRK) schemes designed by A. Najafi-Yazdi and L.Mongeau [32]. Its Butcher's tableau is given by Table A.7:

0.257820901066211	0.377847764031163	0	0
0.434296446908075	0.385232756462588	0.461548399939329	0
0.758519768667167	0.675724855841358	-0.061710969841169	0.241480233100410
	0.750869573741408	-0.362218781852651	0.611349208111243

Table A.7: Butcher's tableau of ILDDRK(3,4) method

This scheme is A-stable [32]. However, it seems that this scheme is actually of order 2 according to our results and also to this paper [33].

Appendix B

Links between dimensionless numbers in SD and FD

CFL numbers

SD and FD do not have the same mean distance between two adjacent DoF. In 1D, with an uniform mesh, it is Δx for FD and $\frac{\Delta x}{p+1}$ for SD. Thus, a new CFL number can be defined for SD with a length scale corresponding to the distance between two adjacent DoF [52]:

$$CFL_{SD} = \frac{c\Delta t}{\frac{\Delta x}{p+1}} = (p+1) \times CFL_{FD} \tag{B.1}$$

where c is the 1D linear advection speed, Δx the distance between two consecutive mesh cells, Δt the time step and p the order of the interpolation polynomial. Eq. (B.1) shows that increasing the spatial order of accuracy, and keeping N_{cells} constant, leads to a smaller time step for a given value of CFL_{SD}. However, even if Δt is decreasing, the scheme could not be stable for a higher p even if it was the case for a lower p [52]. It comes from the fact that spectral properties of schemes depend on the value of p for SD which is not the case for FD where $p = 0$ all the time so the stability limit is always $CFL = 1$. Thus, the stability limit for the time step changes with p. Therefore, for SD, stability limits should be think in terms of time step rather than in terms of CFL condition.

Fourier numbers

The same reasoning can be applied for Fourier numbers. In 1D, still for an uniform mesh, a new Fourier number can be introduced for SD, noted α_{SD}, based on what is done for CFL numbers. Using the same notations of the previous paragraph it is defined by:

$$\alpha_{SD} = \frac{\kappa\Delta t}{\frac{\Delta x^2}{(p+1)^2}} = (p+1)^2 \alpha_{FD} \tag{B.2}$$

where κ is the diffusion coefficient.

Appendix C

Verification of spatial accuracy of the model code

The purpose of this paragraph is to see if the expected spatial order of accuracy is reached by the model code. The following test case is considered:

- Physical domain: $[-L_x, L_x]$ with $L_x = 1$ m.
- Advection speed: $c = 1$ m.s^{-1}.
- Physical time simulated: $T_f = 0.5$ s.
- Periodic boundary conditions.
- Order of interpolation polynomial: $p = 2$
- Initial solution: $u_0(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}}$ with $\sigma = 0.1$.
- Time integration scheme: Explicit Runge-Kutta of order 2 (ERK2).

It represents the advection of a 1D-Gaussian profile over a domain of total length $L = 2$ m during 0.5 s at a speed of 1 m.s^{-1}. Because $p = 2$, the spatial scheme should be of order $p + 1 = 3$. Thus, if a constant time step is kept and two simulations are done with two different meshes, characterized by Δx_1 and $\Delta x_2 = \frac{\Delta x_1}{2}$, the numerical error has to decrease by a factor of $2^{p+1} = 8$ between the two meshes. Therefore, several simulations with different values of N_{cells}, always taken as a power of 2, were tested. The values of $L_2^{rel}(u_{num})$ and the error ratio between 2 consecutive meshes, noted β, were collected. For these simulations, the time step was set to $\Delta t = 1.30 \times 10^{-5}$ s, corresponding to $CFL_{SD} = 0.01$ computed with $p = 2$ and $N_{cells} = 512$ (the finest mesh used). The results are presented in Table C.1:

$\mathbf{N_{cells}}$	16	32	64	128	256	512
$\mathbf{L_2^{rel}(u_{num})}$	4.31e-2	6.25e-3	7.91e-4	1.00e-4	1.26e-5	1.67e-6
β	6.90	7.90	7.91	7.94	7.54	-

Table C.1: Comparison between numerical errors for different meshes when $p = 2$ in order to check that the spatial scheme is of order $p + 1 = 3$

Results analysis: The model code is behaving quite well. The ratio between two consecutive meshes is close to 8 except between meshes with $N_{cells} = 16$ and $N_{cells} = 32$. It is may be because they are very coarsed. This ratio is also decreasing between meshes with $N_{cells} = 256$ and $N_{cells} = 512$, probably due to the mesh convergence that has been reached. Thus, it seems that the model code is giving the expected order of accuracy.

The strength of the SD method is that it is easy to reach high-order only by changing the value of p. Actually what is really important is the number of DoF. For a 1D-SD process, DoF $= N_{cells} \times (p + 1)$, then when p is increased, N_{cells} can be reduced in order to keep DoF approximately constant to have clear comparisons. Normally, even if N_{cells} decreases, the error is smaller when p increases. The previous test case has been done but with different values of p and N_{cells} in order to keep the number of DoF around 650 as in [27]. Moreover, to minimize the error linked to Δt, CFL_{SD} was kept to 0.01 to have a very small time step. The values of t_{CPU} in Table C.2 are not very significant because usually the time step is well bigger but it is still possible to compare them to see which simulation is faster.

$\mathbf{p}$	1	2	3	4
$\mathbf{N_{cells}}$	325	216	162	130
DoF	650	648	648	650
$\mathbf{L_2^{rel}}(\mathbf{u_{num}})$	1.11e-3	2.10e-5	1.11e-6	8.08e-7
$\mathbf{t_{CPU}}$ [s]	1551	2447	3483	5197

Table C.2: Comparison between numerical errors for different meshes and values of p when DoF is kept approximately constant.

Results analysis: As expected, the error is decreasing when p is higher and DoF kept constant. However, it also increases the simulation time, which was also expected. With this example, the strength of the SD method is highlighted: it can easily go to high-order but it has a cost in terms of computation time due to a stability criterion if explicit time-marching schemes are used.

Appendix D

Algorithms for the numerical computation of the Jacobian of the residual for 1D-advection

Algorithm that computes numerically $\frac{\partial R^i}{\partial u^k}$ for the 1D-advection equation

Here the algorithm used to compute numerically the local Jacobians of a cell i in the case of the 1D-advection equation is presented. It follows the explanations from section (6.3).

Algorithm 10 Compute numerically $\frac{\partial R^i}{\partial u^k}$ for 1D-advection

Input(s): X_s, X_f, N_{SP}, N_{FP}, i, k, $\underline{u}$, $\underline{v}$, $\underline{F}$, R^i, c, ϵ

if k = i - 1 **then**
  for j from 1 to N_{SP} **do**
    $u_j^{i-1} \leftarrow u_j^{i-1} + \epsilon$
    Recompute v^{i-1}, F^{i-1} and $F_{int}^{i-1,i}$
    Recompute $\bar{F}^i$ using the new value of $F_{int}^{i-1,i}$ and the already known values F^i and $F_{int}^{i,i+1}$
    Compute $R^{i,new}$
    $\left(\frac{\partial R^i}{\partial u^{i-1}}\right)_{\substack{1 \le row \le N_{SP} \\ col=j}} = \left(\frac{R^{i,new} - R^i}{\epsilon}\right)_{1 \le row \le N_{SP}}$
  end for
else if k = i **then**
  for j from 1 to N_{SP} **do**
    $u_j^{i} \leftarrow u_j^{i} + \epsilon$
    Recompute v^{i}, F^{i}, $F_{int}^{i-1,i}$ and $F_{int}^{i,i+1}$
    Recompute $\bar{F}^i$ using the new values of $F_{int}^{i-1,i}$, F^i and $F_{int}^{i,i+1}$
    Compute $R^{i,new}$
    $\left(\frac{\partial R^i}{\partial u^{i}}\right)_{\substack{1 \le row \le N_{SP} \\ col=j}} = \left(\frac{R^{i,new} - R^i}{\epsilon}\right)_{1 \le row \le N_{SP}}$
  end for
else if k = i + 1 **then**
  for j from 1 to N_{SP} **do**
    $u_j^{i+1} \leftarrow u_j^{i+1} + \epsilon$
    Recompute v^{i+1}, F^{i+1} and $F_{int}^{i,i+1}$
    Recompute $\bar{F}^i$ using the new value of $F_{int}^{i,i+1}$ and the already known values F^i and $F_{int}^{i-1,i}$
    Compute $R^{i,new}$
    $\left(\frac{\partial R^i}{\partial u^{i+1}}\right)_{\substack{1 \le row \le N_{SP} \\ col=j}} = \left(\frac{R^{i,new} - R^i}{\epsilon}\right)_{1 \le row \le N_{SP}}$
  end for
end if

Output(s): $\frac{\partial R^i}{\partial u^k}$

Algorithm 10 is a *Python* function called **computeLocJacNumAdv1D**$\left(X_s, X_f, N_{SP}, N_{FP}, i, k, \underline{u}, \underline{v}, \underline{F}, R^i, c, \epsilon\right)$.

Global algorithm for the 1D-advection equation

Algorithm 11 Compute numerically $\frac{\partial \underline{\underline{R}}}{\partial \underline{u}}$ for 1D-advection

Input(s): N_{cells}, N_{SP}, $\underline{u}$, $\underline{v}$, $\underline{F}$, R^i, c, ϵ

1: Initialization of $\frac{\partial \underline{\underline{R}}}{\partial \underline{u}}$ as a square matrix of size $N_{cells} \times N_{SP}$.

2: **for** i from 1 to N_{cells} **do**

3: $\left(\frac{\partial \underline{\underline{R}}}{\partial \underline{u}}\right)_{\substack{1+(i-1)N_{SP} \leq row \leq iN_{SP} \\ 1+(i-2)N_{SP} \leq col \leq (i-1)N_{SP}}} = \text{computeLocJacNumAdv1D}\big(X_s, X_f, N_{SP}, N_{FP}, i, i-1, \underline{u}, \underline{v}, \underline{F}, R^i, c, \epsilon\big)$

4: $\left(\frac{\partial \underline{\underline{R}}}{\partial \underline{u}}\right)_{\substack{1+(i-1)N_{SP} \leq row \leq iN_{SP} \\ 1+(i-1)N_{SP} \leq col \leq iN_{SP}}} = \text{computeLocJacNumAdv1D}\big(X_s, X_f, N_{SP}, N_{FP}, i, i, \underline{u}, \underline{v}, \underline{F}, R^i, c, \epsilon\big)$

5: $\left(\frac{\partial \underline{\underline{R}}}{\partial \underline{u}}\right)_{\substack{1+(i-1)N_{SP} \leq row \leq iN_{SP} \\ 1+iN_{SP} \leq col \leq (i+1)N_{SP}}} = \text{computeLocJacNumAdv1D}\big(X_s, X_f, N_{SP}, N_{FP}, i, i+1, \underline{u}, \underline{v}, \underline{F}, R^i, c, \epsilon\big)$

6: **end for**

Output(s): $\frac{\partial \underline{\underline{R}}}{\partial \underline{u}}$

Appendix E

Algorithms for the numerical computation of the Jacobian of the residual for 1D-diffusion

Global algorithm for the 1D-diffusion equation

Algorithm 12 Compute numerically $\frac{\partial \underline{\underline{R}}}{\partial \underline{u}}$ for 1D-diffusion

Input(s): N_{cells}, N_{SP}, $\underline{u}$, $\frac{\partial \underline{v}}{\partial x}$, $\underline{F}$, R^i, κ ϵ

1: Initialization of $\frac{\partial \underline{\underline{R}}}{\partial \underline{u}}$ as a square matrix of size $N_{cells} \times N_{SP}$.

2: **for** i from 1 to N_{cells} **do**

3: $\left(\frac{\partial \underline{\underline{R}}}{\partial \underline{u}}\right)_{\substack{1+(i-1)N_{SP} \leq row \leq iN_{SP} \\ 1+(i-3)N_{SP} \leq col \leq (i-2)N_{SP}}} = \text{computeLocJacNumDiff1D}\Big(X_s, X_f, N_{SP}, N_{FP}, i, i-2, \underline{u}, \underline{v}, \frac{\partial \underline{v}}{\partial x}, R^i, \kappa, \epsilon\Big)$

4: $\left(\frac{\partial \underline{\underline{R}}}{\partial \underline{u}}\right)_{\substack{1+(i-1)N_{SP} \leq row \leq iN_{SP} \\ 1+(i-2)N_{SP} \leq col \leq (i-1)N_{SP}}} = \text{computeLocJacNumDiff1D}\Big(X_s, X_f, N_{SP}, N_{FP}, i, i-1, \underline{u}, \underline{v}, \frac{\partial \underline{v}}{\partial x}, R^i, \kappa, \epsilon\Big)$

5: $\left(\frac{\partial \underline{\underline{R}}}{\partial \underline{u}}\right)_{\substack{1+(i-1)N_{SP} \leq row \leq iN_{SP} \\ 1+(i-1)N_{SP} \leq col \leq iN_{SP}}} = \text{computeLocJacNumDiff1D}\Big(X_s, X_f, N_{SP}, N_{FP}, i, i, \underline{u}, \underline{v}, \frac{\partial \underline{v}}{\partial x}, R^i, \kappa, \epsilon\Big)$

6: $\left(\frac{\partial \underline{\underline{R}}}{\partial \underline{u}}\right)_{\substack{1+(i-1)N_{SP} \leq row \leq iN_{SP} \\ 1+iN_{SP} \leq col \leq (i+1)N_{SP}}} = \text{computeLocJacNumDiff1D}\Big(X_s, X_f, N_{SP}, N_{FP}, i, i+1, \underline{u}, \underline{v}, \frac{\partial \underline{v}}{\partial x}, R^i, \kappa, \epsilon\Big)$

7: $\left(\frac{\partial \underline{\underline{R}}}{\partial \underline{u}}\right)_{\substack{1+(i-1)N_{SP} \leq row \leq iN_{SP} \\ 1+(i+1)N_{SP} \leq col \leq (i+2)N_{SP}}} = \text{computeLocJacNumDiff1D}\Big(X_s, X_f, N_{SP}, N_{FP}, i, i+2, \underline{u}, \underline{v}, \frac{\partial \underline{v}}{\partial x}, R^i, \kappa, \epsilon\Big)$

8: **end for**

Output(s): $\frac{\partial \underline{\underline{R}}}{\partial \underline{u}}$

Algorithm that computes numerically $\frac{\partial R^i}{\partial u^k}$ for the 1D-diffusion equation

Here the algorithm used to compute the local Jacobians of a cell i in the case of the 1D-diffusion equation is presented. It follows the explanations from section (7.3).

Algorithm 13 is a *Python* function called **computeLocJacNumDiff1D**$\Big(X_s, X_f, N_{SP}, N_{FP}, i, k, \underline{u}, \underline{v}, \frac{\partial \underline{v}}{\partial x}, R^i, \kappa, \epsilon\Big)$.

Algorithm 13 Compute numerically $\frac{\partial R^i}{\partial u^k}$ for 1D-diffusion

Input(s): N_{SP}, i, k, $\underline{u}$, $\underline{v}$, $\frac{\partial \underline{v}}{\partial x}$, R^i, κ, ϵ

if k = i - 2 **then**
  for j from 1 to N_{SP} **do**
    $u_j^{i-2} \leftarrow u_j^{i-2} + \epsilon$
    Recompute v^{i-2} and $v_{int}^{i-2,i-1}$ using the new value $v_{N_{FP}}^{i-2}$ and the already known value v_1^{i-1}
    Set $v_1^{i-1} = v_{int}^{i-2,i-1}$
    Recompute $\left(\frac{\partial u}{\partial x}\right)^{i-1}$ using the new v^{i-1} and after that $\left(\frac{\partial v}{\partial x}\right)^{i-1}$ using the new $\left(\frac{\partial u}{\partial x}\right)^{i-1}$
    Recompute $\left(\frac{\partial v}{\partial x}\right)_{int}^{i-1,i}$ using the new value of $\left(\frac{\partial v}{\partial x}\right)_{N_{FP}}^{i-1}$ and the already known value of $\left(\frac{\partial v}{\partial x}\right)_1^i$
    Set $\left(\frac{\partial v}{\partial x}\right)_1^i = \left(\frac{\partial v}{\partial x}\right)_{int}^{i-1,i}$
    Compute $R^{i,new}$
    $\left(\frac{\partial R^i}{\partial u^{i-2}}\right)_{\substack{1 \le row \le N_{SP} \\ col=j}} = \left(\frac{R^{i,new} - R^i}{\epsilon}\right)_{1 \le row \le N_{SP}}$
  end for
else if k = i - 1 **then**
  for j from 1 to N_{SP} **do**
    $u_j^{i-1} \leftarrow u_j^{i-1} + \epsilon$
    Recompute v^{i-1}, $v_{int}^{i-2,i-1}$ and $v_{int}^{i-1,i}$
    Set $v_1^{i-1} = v_{int}^{i-2,i-1}$, $v_{N_{FP}}^{i-1} = v_{int}^{i-1,i}$ and $v_1^i = v_{int}^{i-1,i}$
    Recompute $\left(\frac{\partial u}{\partial x}\right)^{i-1}$, $\left(\frac{\partial u}{\partial x}\right)^{i}$, $\left(\frac{\partial v}{\partial x}\right)^{i-1}$ and $\left(\frac{\partial v}{\partial x}\right)^{i}$
    Recompute $\left(\frac{\partial v}{\partial x}\right)_{int}^{i-1,i}$ using the new values of $\left(\frac{\partial v}{\partial x}\right)_{N_{FP}}^{i-1}$ and $\left(\frac{\partial v}{\partial x}\right)_1^i$
    Recompute $\left(\frac{\partial v}{\partial x}\right)_{int}^{i,i+1}$ using the new value of $\left(\frac{\partial v}{\partial x}\right)_{N_{FP}}^{i}$ and the already new value of $\left(\frac{\partial v}{\partial x}\right)_1^{i+1}$
    Set $\left(\frac{\partial v}{\partial x}\right)_1^i = \left(\frac{\partial v}{\partial x}\right)_{int}^{i-1,i}$ and $\left(\frac{\partial v}{\partial x}\right)_{N_{FP}}^i = \left(\frac{\partial v}{\partial x}\right)_{int}^{i,i+1}$
    Compute $R^{i,new}$
    $\left(\frac{\partial R^i}{\partial u^{i-1}}\right)_{\substack{1 \le row \le N_{SP} \\ col=j}} = \left(\frac{R^{i,new} - R^i}{\epsilon}\right)_{1 \le row \le N_{SP}}$
  end for
else if k = i **then**
  for j from 1 to N_{SP} **do**
    $u_j^{i} \leftarrow u_j^{i} + \epsilon$
    Recompute v^{i}, $v_{int}^{i-1,i}$ and $v_{int}^{i,i+1}$
    Set $v_{N_{FP}}^{i-1} = v_{int}^{i-1,i}$, $v_1^i = v_{int}^{i-1,i}$, $v_{N_{FP}}^{i} = v_{int}^{i,i+1}$ and $v_1^{i+1} = v_{int}^{i,i+1}$
    Recompute $\left(\frac{\partial u}{\partial x}\right)^{i-1}$, $\left(\frac{\partial u}{\partial x}\right)^{i}$, $\left(\frac{\partial u}{\partial x}\right)^{i+1}$, $\left(\frac{\partial v}{\partial x}\right)^{i-1}$, $\left(\frac{\partial v}{\partial x}\right)^{i}$ and $\left(\frac{\partial v}{\partial x}\right)^{i+1}$
    Recompute $\left(\frac{\partial v}{\partial x}\right)_{int}^{i-1,i}$ and $\left(\frac{\partial v}{\partial x}\right)_{int}^{i,i+1}$
    Set $\left(\frac{\partial v}{\partial x}\right)_1^i = \left(\frac{\partial v}{\partial x}\right)_{int}^{i-1,i}$ and $\left(\frac{\partial v}{\partial x}\right)_{N_{FP}}^i = \left(\frac{\partial v}{\partial x}\right)_{int}^{i,i+1}$
    Compute $R^{i,new}$
    $\left(\frac{\partial R^i}{\partial u^{i}}\right)_{\substack{1 \le row \le N_{SP} \\ col=j}} = \left(\frac{R^{i,new} - R^i}{\epsilon}\right)_{1 \le row \le N_{SP}}$
  end for
else if k = i + 1 **then**
  for j from 1 to N_{SP} **do**
    $u_j^{i+1} \leftarrow u_j^{i+1} + \epsilon$
    Recompute v^{i+1}, $v_{int}^{i,i+1}$ and $v_{int}^{i+1,i+2}$
    Set $v_1^{i+1} = v_{int}^{i,i+1}$, $v_{N_{FP}}^{i+1} = v_{int}^{i+1,i+2}$ and $v_{N_{FP}}^{i} = v_{int}^{i,i+1}$
    Recompute $\left(\frac{\partial u}{\partial x}\right)^{i+1}$, $\left(\frac{\partial u}{\partial x}\right)^{i}$, $\left(\frac{\partial v}{\partial x}\right)^{i+1}$ and $\left(\frac{\partial v}{\partial x}\right)^{i}$
    Recompute $\left(\frac{\partial v}{\partial x}\right)_{int}^{i,i+1}$ using the new values of $\left(\frac{\partial v}{\partial x}\right)_1^{i+1}$ and $\left(\frac{\partial v}{\partial x}\right)_{N_{FP}}^{i}$
    Recompute $\left(\frac{\partial v}{\partial x}\right)_{int}^{i-1,i}$ using the new value of $\left(\frac{\partial v}{\partial x}\right)_1^{i}$ and the already new value of $\left(\frac{\partial v}{\partial x}\right)_{N_{FP}}^{i-1}$
    Set $\left(\frac{\partial v}{\partial x}\right)_1^i = \left(\frac{\partial v}{\partial x}\right)_{int}^{i-1,i}$ and $\left(\frac{\partial v}{\partial x}\right)_{N_{FP}}^i = \left(\frac{\partial v}{\partial x}\right)_{int}^{i,i+1}$
    Compute $R^{i,new}$
    $\left(\frac{\partial R^i}{\partial u^{i+1}}\right)_{\substack{1 \le row \le N_{SP} \\ col=j}} = \left(\frac{R^{i,new} - R^i}{\epsilon}\right)_{1 \le row \le N_{SP}}$
  end for
else if k = i + 2 **then**
  for j from 1 to N_{SP} **do**
    $u_j^{i+2} \leftarrow u_j^{i+2} + \epsilon$
    Recompute v^{i+2} and $v_{int}^{i+1,i+2}$ using the new value v_1^{i+2} and the already known value $v_{N_{FP}}^{i+1}$
    Set $v_{N_{FP}}^{i+1} = v_{int}^{i+1,i+2}$
    Recompute $\left(\frac{\partial u}{\partial x}\right)^{i+1}$ using the new v^{i+1} and after that $\left(\frac{\partial v}{\partial x}\right)^{i+1}$ using the new $\left(\frac{\partial u}{\partial x}\right)^{i+1}$
    Recompute $\left(\frac{\partial v}{\partial x}\right)_{int}^{i,i+1}$ using the new value of $\left(\frac{\partial v}{\partial x}\right)_1^{i+1}$ and the already known value of $\left(\frac{\partial v}{\partial x}\right)_{N_{FP}}^{i}$
    Set $\left(\frac{\partial v}{\partial x}\right)_{N_{FP}}^i = \left(\frac{\partial v}{\partial x}\right)_{int}^{i,i+1}$
    Compute $R^{i,new}$
    $\left(\frac{\partial R^i}{\partial u^{i+2}}\right)_{\substack{1 \le row \le N_{SP} \\ col=j}} = \left(\frac{R^{i,new} - R^i}{\epsilon}\right)_{1 \le row \le N_{SP}}$
  end for
end if

Output(s): $\frac{\partial R^i}{\partial u^k}$

Appendix F

Vortex transported by an uniform flow

Initial vortex

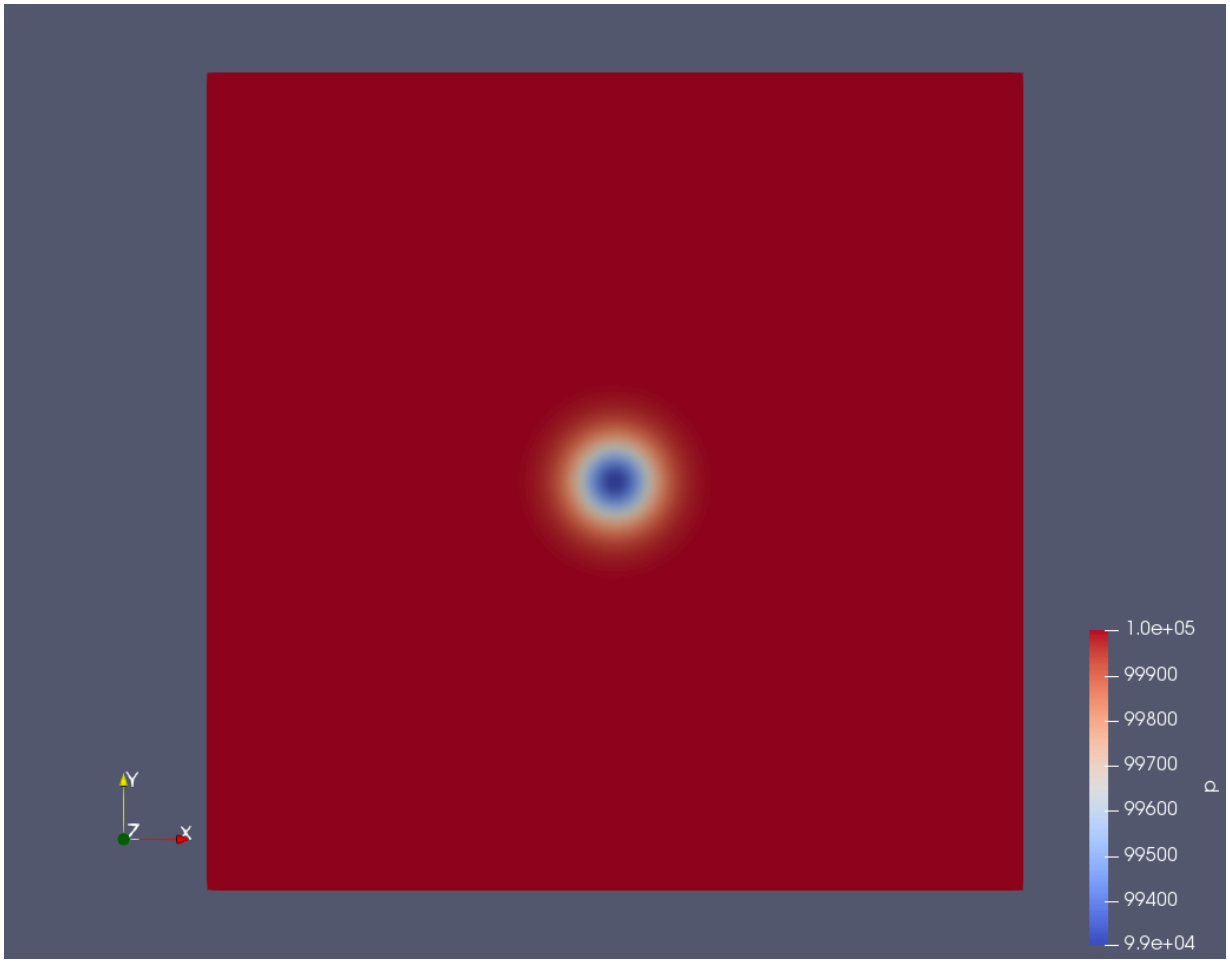


Figure F.1: Initial vortex which will be transported by the flow.

Performance of implicit schemes

Δt [s]	$1.12e^{-6}$	$2.24e^{-6}$	$4.48e^{-6}$	$2.8e^{-7}$
Nonlinear iterations at each RK stage	1	1	1	-
Total linear iterations at each RK stage	15	37	104	-
t_{CPU} [s]	431	507	834	378
$L_2^{abs}(P)$ [Pa]	0.319	1.273	5.008	$6.56e^{-4}$

Table F.1: For implicit midpoint during one loop (0.000575 s), $\epsilon_{Newt,a} = 10^{-5}$ and $\epsilon_{Kry,a} = 10^{-6}$

Δt [s]	$1.12e^{-6}$	$2.24e^{-6}$	$4.48e^{-6}$	$2.8e^{-7}$
Nonlinear iterations at each RK stage	1-1	1-1	1-1	-
Total linear iterations at each RK stage	6-7	15-18	41-46	-
t_{CPU} [s]	417	424	528	378
$L_2^{abs}(P)$ [Pa]	0.155	0.618	2.452	$6.56e^{-4}$

Table F.2: For SDIRK(2,2) during one loop (0.000575 s), $\epsilon_{Newt,a} = 10^{-5}$ and $\epsilon_{Kry,a} = 10^{-6}$

Δt [s]	$1.12e^{-6}$	$2.24e^{-6}$	$4.48e^{-6}$	$2.8e^{-7}$
Nonlinear iterations at each RK stage	1-1	1-1	2-1	-
Total linear iterations at each RK stage	23-20	67-55	269-162	-
t_{CPU} [s]	981	1413	3306	378
$L_2^{abs}(P)$ [Pa]	0.036	0.279	1.969	$6.56e^{-4}$

Table F.3: For SDIRK(2,3) during one loop (0.000575 s), $\epsilon_{Newt,a} = 10^{-5}$ and $\epsilon_{Kry,a} = 10^{-6}$

Δt [s]	$1.12e^{-6}$	$2.24e^{-6}$	$4.48e^{-6}$	$2.8e^{-7}$
Nonlinear iterations at each RK stage	1-1-1	1-1-1	1-1-2	-
Total linear iterations at each RK stage	10-11-12	25-26-29	72-76-117	-
t_{CPU} [s]	790	917	1405	378
$L_2^{abs}(P)$ [Pa]	0.013	0.085	0.629	$6.56e^{-4}$

Table F.4: For SDIRK(3,3) during one loop (0.000575 s), $\epsilon_{Newt,a} = 10^{-5}$ and $\epsilon_{Kry,a} = 10^{-6}$

Δt [s]	$1.12e^{-6}$	$2.24e^{-6}$	$4.48e^{-6}$	$2.8e^{-7}$
Nonlinear iterations at each RK stage	1-1-1	1-1-1	1-2-1	-
Total linear iterations at each RK stage	2-22-18	3-59-50	6-264-142	-
t_{CPU} [s]	1015	1353	3110	378
$L_2^{abs}(P)$ [Pa]	0.028	0.221	1.589	$6.56e^{-4}$

Table F.5: For DIRK(3,3) during one loop (0.000575 s), $\epsilon_{Newt,a} = 10^{-5}$ and $\epsilon_{Kry,a} = 10^{-6}$

Δt [s]	$1.12e^{-6}$	$2.24e^{-6}$	$4.48e^{-6}$	$2.8e^{-7}$
Nonlinear iterations at each RK stage	1-1-1	1-1-1	-	-
Total linear iterations at each RK stage	34-29-26	104-89-80	-	-
t_{CPU} [s]	2412	4153	-	378
$L_2^{abs}(P)$ [Pa]	0.0046	0.069	-	$6.56e^{-4}$

Table F.6: For SDIRK(3,4) during one loop (0.000575 s), $\epsilon_{Newt,a} = 10^{-5}$ and $\epsilon_{Kry,a} = 10^{-6}$

Δt [s]	$1.12e^{-6}$	$2.24e^{-6}$	$4.48e^{-6}$	$2.8e^{-7}$
Nonlinear iterations at each RK stage	1-1-1	1-1-1	1-2-1	-
Total linear iterations at each RK stage	7-11-6	24-35-16	69-36-43	-
t_{CPU} [s]	700	747	1433	378
$L_2^{abs}(P)$ [Pa]	0.014	0.057	0.270	$6.56e^{-4}$

Table F.7: For ILDDRK(3,4) during one loop (0.000575 s), $\epsilon_{Newt,a} = 10^{-5}$ and $\epsilon_{Kry,a} = 10^{-6}$