UNIVERSITÉ DE STRASBOURG
UFR DE MATH-INFO
IRMA

Internship report

To validate the first year of the Master's program

MASTER

in Applied Mathematics

With the topic of

Simulation of 2D Euler equations on different grids

Presented by
M. PAWLUS Jérémy

Under the supervision of
Dr. THOMANN Andrea

Academic Year 2022-2023

Contents

1 Introduction

1.1 Preliminary information

1.1.1 Acknowledgements

I would like to thank my supervisor Andrea Thomann for her guidance and her teaching during this internship. I am also very grateful for her patience and her availability when I needed help, along with bringing me the opportunity to work on something providing answers to my questions about my future career.

1.1.2 Preliminary remarks

About the report This report is about my internship, which takes part in the curriculum of the master of applied mathematics and computational sciences from the University of Strasbourg (Master CSMI). It is a 2-month internship, which began on the 12th of June 2023 and ended on the 4th of August 2023.

About the internship This internship has provided me the opportunity to work on a research project, which is about the implementation and the simulation of the two-dimensional Euler equation on different mesh types. And I decided to apply for it because I intend on working in the academic sector as a researcher in applied mathematics. Such an internship was finally for me an opportunity to work within an environment I have never been in before.

Some important information about the report content Since this report is about the continuation of my previous project (another part of my curriculum) — which dealt with the implementation of the one-and-two-dimensional finite volume method numerical scheme for the Euler equations in Julia —, it contains some unchanged parts from the report of the latter. The almost unchanged content from the report of the previous project include the general and the specific context, the academical and the specific objectives, all the theoretical framework related to the pressure and the numerical fluxes, the introduction of the finite volume method for Cartesian grids and the introduction of the transport problem test case in two dimensions.

1.1.3 Presentation of the research center

The research center: *IRMA* My internship took place within the research center *IRMA* (Institut de Recherche Mathématique Avancée), which is a research unit of the *CNRS* (Centre national de la recherche scientifique) and the University of Strasbourg. It is located in the university campus of Strasbourg and employs more than 110 researchers.

The collaborating research center: *Inria* However, I also collaborated with the research center *Inria* (Institut national de recherche en sciences et technologies du numérique), which is a French public research center in computer science and applied mathematics founded in 1967. Its main goal is to advance in the development of digital sciences and technologies. [2] It employs more than 3,000 researchers and engineers in many research centers and laboratories in France.

My team and my supervisor My internship was supervised by researcher Andrea Thomann, who is working for the TONUS team of *Inria*, whose research interests are about the mathematical modeling of plasma physics simulations.

I also sometimes interacted with reseacher Victor Michel-Dansac, who is working for the TONUS team of *Inria* as well. He was another supervisor alongside Andrea during my previous project.

1.2 Context of the internship

1.2.1 General context

The Euler equations were first introduced in Euler's article entitled *Principes généraux du mouvement des fluides* [1], which was published in 1757.

The Euler equations are a set of partial differential equations often used to deal with hydrodynamic systems, especially when the continuous media hypothesis is met. At very small scales, matter is usually considered as a continuous medium according to the human eye. Such hypothesis therefore allows us to use continuous or differentiable mathematical models to describe the behavior of such systems. For instance, the Euler equations usually serve as the basic mathematical framework for many fields in science and engineering, such as aeronautics, automotive engineering, astrophysics or meteorology.

The Euler equations mathematically represent the motion of inviscid (viscosity-free) fluids. Their foundation relies on the conservation of mass, momentum and energy of the given fluid system translated into equations. The complexity of both the physical data and the mathematical expression of the Euler equations often leads to solutions having complex behaviors, arising for example in turbulences.

The Euler equations can also be appreciated as the generic equations of more specific ones such as the transport equation. This is partly why they are the engine of many researches in applied mathematics.

The Euler equations in two dimensions are the following system of partial differential equations

$$\begin{cases} \frac{\partial \rho}{\partial t} + \frac{\partial(\rho u_x)}{\partial x} + \frac{\partial(\rho u_y)}{\partial y} = 0 \\ \frac{\partial(\rho u)}{\partial t} + \frac{\partial(\rho u_x^2 + p)}{\partial x} + \frac{\partial(\rho u_x u_y)}{\partial y} = 0 \\ \frac{\partial(\rho u)}{\partial t} + \frac{\partial(\rho u_x u_y)}{\partial x} + \frac{\partial(\rho u_y^2 + p)}{\partial y} = 0 \\ \frac{\partial E}{\partial t} + \frac{\partial(u_x(E+p))}{\partial x} + \frac{\partial(u_y(E+p))}{\partial y} = 0 \end{cases} \tag{1.1}$$

where ρ is the mass density of the fluid (in kg/m^3), u_x is the velocity of the fluid in the x-direction (in m/s), u_y is the velocity of the fluid in the y-direction (in m/s), p is the pressure of the fluid (in Pa) and E is the total energy of the fluid (in J/m^3).

The total energy is defined as

$$E = \rho e + \frac{1}{2}\rho(u_x^2 + u_y^2) \tag{1.2}$$

where e is the specific internal energy of the fluid (in J/kg).

Finally, the Euler equations consist of four physical quantities obeying to the laws of conservations, which are ρ, ρu_x (momentum of the fluid in the x-direction, in kgm/s), ρu_y (momentum of the fluid in the y-direction, in kgm/s) and E from (1.1). These four quantities are called the conserved variables.

1.2.2 Specific context

My intership was about implementing and simulating the 2D Euler equations on different grids (Cartesian and triangular meshes) using the finite volume method in Julia.

To begin with, my internship can be appreciated as the continuation and the extension of my project, which was, as a reminder, about the implementation of a finite volume method in Julia for the one-and-two-dimensional Euler equations on **Cartesian grids only**.

My previous project was finished with two notable bugs:

- The first one was about the conversion of the primitive variables to the conserved variables, which was not done correctly. It is indeed a function I am relying on for the setting of the initial conditions of the Gresho vortex test case, for instance. In fact, I had wrongly written the the expression of the total energy E in terms of the pressure.
- The second one was about the missing computations related to y-direction in the update function for the two-dimension numerical scheme. It visually resulted in a more dilated curb of the density of the final solution of the transport problem test case in the y-direction.

My previous project was also supposed to be finished with the computations of errors and convergence rates of the numerical scheme on the transport problem test case (using periodic boundary conditions) on Cartesian grids. However, due to the lack of time, it was undone.

Thus, at the practical level, my internship began with rectifying the dilation-bug and the computation of the errors for both one and two dimensional implementations on Cartesian grids, as explained in the 2.3.1 section.

Then, my internship was about generalizing the code of the Cartesian implementation based on normal vectors, paving the way to triangular mesh implementations: both structured (cuting the square simplexes of the Cartesian grid in half with diagonals of slope $\frac{N_y}{N_x}$ where N_x and N_y designate the number of cells in the x-direction and the y-direction, providing an amount of N_xN_y rectangular cells) and unstructured (using GMSH-generated meshes).

At the end of the day, doing this internship is a way for me to learn the use of consensually-spreading Julia language. More deeply, the goal is to study the Gresho vortex test case for different Mach numbers to check whether triangular mesh implementations provide lower diffusive effects than Cartesian meshes.

Otherwise and personally speaking, this internship serves to answer some questions related to my future career, since I am tempted by the idea of becoming a researcher in applied mathematics (especially in the academic sector). I indeed never worked in a research center or a laboratory before, so this internship is a way to get a taste of it.

1.3 Objectives of the internship

1.3.1 Academical objectives

The master of applied mathematics and computational sciences from the University of Strasbourg (Master CSMI) embeds many compulsory subjects to validate, among which there are two internships subject (in the second semester, and another one in the fourth semester).

The goal of these internships is to make us apply our knowledge and skills in a more professional environment (revolving around engineering and mathematical research), as an addition to the two projects of the master.

It also consists of giving assistance to one or more tutors dealing with a problem containing many mathematical and computational aspects. The academical deliverables of the internships are a final report, a final presentation and a portfolio that sums up all the work.

The report should first bring information about the internship and the context revolving around it, then an analysis of the used data and a theoretical description of the used mathematical and computational models. It should then include the results of the experiments performed during the whole project (results, measurement, uncertainties and possible retroactive effects). The report concludes by providing a summary followed by an outlook on further improvements.

1.3.2 Scientific objectives

The scientific objectives of my internship relying on the Euler equations are:

- **Rectifying the bugs and the undone parts of my previous project:**
  This implies solving the dilation issues that are observed in the plots of the density of the final solution of the transport problem test case and computing the errors and the convergence rates of the numerical scheme on the transport problem test case (using periodic boundary conditions), in order to check the accuracy of the numerical scheme.

- **Generalizing the Cartesian code based on normal vectors:**
  It means adapting the code to the usage of normal vectors in the flux description. In the Cartesian case, the normal vectors are $(1, 0)$ and $(0, 1)$, which are the unit vectors of the x-direction and y-direction axes respectively. When it comes to triangular meshes, the normal vectors are the unit vectors orthogonal to the edges of the triangle and oriented outwards. It involves modifications in the function that computes the Euler flux function, the computation of the numerical fluxes (so the Rusanov and the HLL fluxes, which will be introduced in the 2.2.3) section.

- **Implementation of the finite volume method for unstructured triangular meshes**
  First, unstructured meshes are meshes, whose cells are irregular when it comes to the area, the edge length and its angles. Here, the goal is to implement the finite volume method for such meshes, using the GMSH software to generate the meshes. More concretely, it is about creating a class (mutable struct in the case of Julia) that sparses the information contained in the mesh file (.msh) before relying on the numerical scheme for triangular meshes (which is different from the one for Cartesian meshes).

- **Implementation of a structured triangular mesh**
  Structured triangular meshes are made of right-triangular cells obtained by cutting square simplexes (of a Cartesian mesh) in half by a diagonal of slope $(1, 1)$. The scheme is the same as the one for unstructured triangular meshes.

- **Study of the Gresho vortex test case for different Mach numbers**
  The Gresho vortex test case is a particular solution of the two-dimensional Euler equations. It is used to check the diffusive effects of the numerical scheme, especially through varying the Mach number. The latter is the ratio of the speed of the fluid to the speed of sound. The goal here is to check whether triangular mesh implementations provide, as known, lower diffusive effects than Cartesian meshes. The diffusion is studied through the evolution of the kinetic energy of the fluid. Both quantities will be respectively introduced in sections 3.4.2 and 3.5.1.

- **Computing the errors and the convergence rates of the numerical scheme on the transport problem test case (using periodic boundary conditions) for both Cartesian and triangular meshes**
  This is a way to check the accuracy of the numerical scheme for both Cartesian and triangular

meshes, as intended in my previous project. This part is presented with more details in the 3.3 section.

Overall, the scientific objectives of this project are to implement a finite volume method for the Euler equations in the recently-growing language Julia.

1.3.3 Specific (technical and investigative) objectives

The more specific goals are about implementing a finite volumes scheme in Julia, whose relevancy resides in an ergonomic understanding (similar to Python's), its more low-level-like performance (closer to C/C++ and Rust) and many functionalities including parallelism.

Besides those aspects, the specific objectives of my internship will emphasize on the idea of debugging the code of my previous project and optimizing and making them as efficient (especially when it comes to the durations of computations) as possible (by removing unnecessary loops or trying to use the most efficient functions or operations, etc.).

My code will be implemented using Jupyter notebooks, which are a way to combine many different media like text, images, plots and lines of code.

1.4 Prospects

1.4.1 Estimation of the time needed

The time needed to finish this project may depend on various factors, such as the scope of the project, the complexity of the tasks involved, the resources available, and the pace at which the project is carried out.

- **Debugging the two-dimensional implementation for Cartesian grids and computing the corresponding errors: 1 week versus 3 days in reality.**
  During this phase, I have debugged the two-dimensional implementation and computed the errors for the transport problem test case with periodic boundary conditions. I attribute the quickness of this phase to the fact that I already had a good understanding of the code and the theoretical framework.

- **Support of the normal vectors: 1 week versus 2 days in reality.**
  The generalization of the Cartesian code for any normal vector has been done during this phase: it involved modifications on the Euler flux function, the computation of the eigenvalues, the computation of the numerical fluxes and the numerical scheme itself. Understanding the fact that the generic Euler flux function is the dot product of the normal vector and the Euler flux functions in the x-direction was for me the key to easily process this phase.

- **Triangular meshes implementation: 3 weeks versus 4 weeks.**
  The implementation of the finite volume method for triangular meshes was carried out during this phase. The writing of the neigbors-searching function was the most time-consuming part of this phase and the computation of the time step were tricky to implement.

- **Implementing and simulating many test cases for the 2D Euler equations on all grids: 3 weeks versus 3 weeks.**
  In this stage, I especially worked on the Gresho vortex test case, whose interest resides in studying the diffusive effects of the numerical scheme. Therefore I had to study the test cases for different Mach numbers on differents mesh types and resolutions.

Overall, the project took me 2 months to finish, which is the time I was given to finish it. Some tasks were done simultaneously and overlapping with each other.

2 The Numerical Scheme

Let us here introduce the framework that is used in my internship. Only the two-dimensional formal aspects are presented along this part of the report.

2.1 Mathematical and physical framework

2.1.1 Conservation laws

Let us first remind that the Euler equations are hyperbolic partial differential equations. It means that their solutions tend to have wave properties in a sense that they possess a finite propagation speed of information.

In two-dimensional situations, the conservation law takes the following form [4]:

$$\frac{\partial}{\partial t}q(x,y,t) + \frac{\partial}{\partial x}f(q(x,y,t)) + \frac{\partial}{\partial y}g(q(x,y,t)) = 0 \tag{2.1}$$

where $q : \mathbb{R}^2 \times \mathbb{R}_+ \to \mathbb{R}^m$ is an m-dimensional vector and $f, g : \mathbb{R}^m \to \mathbb{R}^m$ are two flux functions, describing the evolution of q in the x-direction and the y-direction, respectively.

The hyperbolic character of the equation mathematically implies that any real linear combination $\alpha f'(q)+\beta g'(q)$ for $\alpha, \beta \in \mathbb{R}$ of the two Jacobians of the fluxes f and g should be diagonalizable with real eigenvalues.

2.1.2 Writing the Euler equations in two dimensions

Let us introduce the vector $q \in \mathbb{R}^4$ given by

$$q(x,t) = \begin{bmatrix} \rho(x,t) \\ \rho(x,t)u_x(x,t) \\ \rho(x,t)u_y(x,t) \\ E(x,t) \end{bmatrix}.$$

Then the Euler equations in two dimension (as in (1.1)) can be written as $q_t+f(q)_x+g(q)_y = 0$ where

$$f(q) = \begin{bmatrix} \rho u_x \\ \rho u_x^2 + p \\ \rho u_x u_y \\ u_x(E+p) \end{bmatrix} = \begin{bmatrix} q_2 \\ q_2^2/q_1 + p(q) \\ q_2 q_3/q_1 \\ q_2(q_4+p(q))/q_1 \end{bmatrix} \tag{2.2}$$

and

$$g(q) = \begin{bmatrix} \rho u_y \\ \rho u_x u_y \\ \rho u_y^2 + p \\ u_y(E+p) \end{bmatrix} = \begin{bmatrix} q_3 \\ q_2 q_3/q_1 \\ q_3^2/q_1 + p(q) \\ q_3(q_4+p(q))/q_1 \end{bmatrix}. \tag{2.3}$$

Therein, u_x denotes the velocity in the x-direction and u_y the velocity in the y-direction. All the remaining physical quantities are the same as in (1.1).

2.1.3 Other physical quantities related to the Euler equations

Pressure Let us consider a small volume of gas with density ρ, pressure p, and specific internal energy e. According to the ideal gas law, the relationship between these quantities is provided by $pV = mRT$, where V is the volume, m is the mass, R is the specific gas constant, and T is the temperature. [4]

If we then suppose that the gas behaves as an ideal gas and that the specific heat ratio is denoted by γ, the latter is defined as the ratio of the specific heat capacitiy at constant pressure c_p and the same capacity at constant volume c_v.

When it comes to an ideal gas, the specific internal energy e is related to the temperature T by $e = c_v T$. Hence, we can rewrite the ideal gas law as

$$pV = mR\left(\frac{e}{c_v}\right). \tag{2.4}$$

Let us now express the mass m in terms of the density ρ and the volume V using $m = \rho V$. Substituting into (2.4) gives

$$pV = \rho V R\left(\frac{e}{c_v}\right). \tag{2.5}$$

After cancelling out the volume V from both sides, we obtain $p = \rho R\left(\frac{e}{c_v}\right)$.

Since $\gamma = \frac{c_p}{c_v}$, we can express c_p in terms of γ and c_v as $c_p = \gamma c_v$. After substituting this into the modified (2.5), we therefore obtain

$$p = \rho R\left(\frac{e}{\frac{c_p}{\gamma}}\right) = \rho R\left(\frac{\gamma e}{c_p}\right) = (\gamma - 1)\rho e. \tag{2.6}$$

Obtaining the eigenvalues of the Jacobian The Jacobian matrix f' is given by

$$f' = \frac{\partial f(q)}{\partial q} = \begin{bmatrix} \frac{\partial f_1}{\partial q_1} & \frac{\partial f_1}{\partial q_2} & \frac{\partial f_1}{\partial q_3} & \frac{\partial f_1}{\partial q_4} \\ \frac{\partial f_2}{\partial q_1} & \frac{\partial f_2}{\partial q_2} & \frac{\partial f_2}{\partial q_3} & \frac{\partial f_2}{\partial q_4} \\ \frac{\partial f_3}{\partial q_1} & \frac{\partial f_3}{\partial q_2} & \frac{\partial f_3}{\partial q_3} & \frac{\partial f_3}{\partial q_4} \\ \frac{\partial f_4}{\partial q_1} & \frac{\partial f_4}{\partial q_2} & \frac{\partial f_4}{\partial q_3} & \frac{\partial f_4}{\partial q_4} \end{bmatrix} \tag{2.7}$$

where f_i represents the i-th component of the flux function $f(q)$.

For the second Euler function $g(q)$ the process is the same, and we obtain the following Jacobian matrix:

$$g' = \frac{\partial g(q)}{\partial q} = \begin{bmatrix} \frac{\partial g_1}{\partial q_1} & \frac{\partial g_1}{\partial q_2} & \frac{\partial g_1}{\partial q_3} & \frac{\partial g_1}{\partial q_4} \\ \frac{\partial g_2}{\partial q_1} & \frac{\partial g_2}{\partial q_2} & \frac{\partial g_2}{\partial q_3} & \frac{\partial g_2}{\partial q_4} \\ \frac{\partial g_3}{\partial q_1} & \frac{\partial g_3}{\partial q_2} & \frac{\partial g_3}{\partial q_3} & \frac{\partial g_3}{\partial q_4} \\ \frac{\partial g_4}{\partial q_1} & \frac{\partial g_4}{\partial q_2} & \frac{\partial g_4}{\partial q_3} & \frac{\partial g_4}{\partial q_4} \end{bmatrix}. \tag{2.8}$$

Let us now calculate the explicit form of the Jacobian (2.7).

1. For the first row, we first have

$$\frac{\partial f_1}{\partial q_1} = 0, \quad \frac{\partial f_1}{\partial q_2} = 1, \quad \frac{\partial f_1}{\partial q_3} = 0, \quad \frac{\partial f_1}{\partial q_4} = 0.$$

2. For the second row, we then have

$$\frac{\partial f_2}{\partial q_1} = 2\frac{\partial q_2}{\partial q_1}\frac{q_2}{q_1} - \frac{q_2^2}{q_1^2} + \frac{\partial p}{\partial q_1}, \quad \frac{\partial f_2}{\partial q_2} = \frac{2q_2}{q_1} + \frac{\partial p}{\partial q_2}, \quad \frac{\partial f_2}{\partial q_3} = \frac{\partial p}{\partial q_3}, \quad \frac{\partial f_2}{\partial q_4} = \frac{\partial p}{\partial q_4}.$$

3. For the third row, we have

$$\frac{\partial f_3}{\partial q_1} = \frac{q_3}{q_1}\frac{\partial q_2}{\partial q_1} + \frac{q_2}{q_1}\frac{\partial q_3}{\partial q_1} - \frac{q_2 q_3}{q_1^2}, \quad \frac{\partial f_3}{\partial q_2} = \frac{q_3}{q_1}, \quad \frac{\partial f_3}{\partial q_3} = \frac{q_2}{q_1}, \quad \frac{\partial f_3}{\partial q_4} = 0.$$

4. For the fourth row, we finally have

$$\begin{aligned}
\frac{\partial f_4}{\partial q_1} &= \frac{1}{\gamma-1}\frac{1}{q_1^2}\left(\frac{\partial q_2}{\partial q_1}q_1 - q_2\right)(q_2^2 + q_3^2 + 2q_1 p(q)) \\
&\quad + \frac{1}{\gamma-1}\frac{q_2}{q_1}(2q_2\frac{\partial q_2}{\partial q_1} + 2q_3\frac{\partial q_3}{\partial q_1} + 2p + 2q_1\frac{\partial p}{\partial q_1}), \\
\frac{\partial f_4}{\partial q_2} &= \frac{1}{\gamma-1}\frac{1}{q_1}(q_2^2 + q_3^2 + 2q_1 p(q)) \\
&\quad + \frac{1}{\gamma-1}\frac{q_2}{q_1}(2q_2 + 2q_1\frac{\partial p}{\partial q_2}), \\
\frac{\partial f_4}{\partial q_3} &= \frac{1}{\gamma-1}\frac{q_2}{q_1}(2q_3 + 2q_1\frac{\partial p}{\partial q_3}), \\
\frac{\partial f_4}{\partial q_4} &= \frac{1}{\gamma-1}\frac{q_2}{q_1}(2q_1\frac{\partial p}{\partial q_4}).
\end{aligned}$$

.

The process is analogously obtained for the g flux function.

Sound speed and eigenvalues The sound speed a (in m/s) is then defined as

$$a = \sqrt{\frac{\gamma p(q)}{\rho}}, \tag{2.9}$$

where γ is the specific heat ratio.

The eigenvalues of the Jacobian (2.7) of the flux in the x-direction depend on the sound speed a provided in (2.9) and are given by

$$\lambda_1 = u_x - a, \quad \lambda_{2,3} = u_x \text{ (which is a double eigenvalue) and } \quad \lambda_3 = u_x + a.$$

Similarly, the eigenvalues of the Jacobian (2.8) of the flux in the y-direction are given by

$$\lambda_1 = u_y - a, \quad \lambda_{2,3} = u_y \text{ (which is a double eigenvalue) and } \quad \lambda_3 = u_y + a.$$

Expression of the Euler fluxes with respect to any normal vector This paragraph serves as a crucial step to generalize the Cartesian code towards triangular meshes. Indeed, when it comes to Cartesian grids (i.e. rectangular or squared cells), the normal vectors of the interfaces are always the unit vectors of the x-and-y-axes. Moreover, the Euler functions provided in (2.2) and (2.3) are respectively associated to the x-and-y-directions, so are automatically adapted to the Cartesian case.

However, for triangular meshes, on a given triangular cell having a given neighboring cell, the normal vector of the interface is the unit vector pointing from the cell towards the neighboring cell while being orthogonal to the edge shared by both cells.

Hence, if we define $\vec{n} = \begin{bmatrix} n_x \\ n_y \end{bmatrix}$ the normal vector of a given edge, the Euler flux adapted to the direction of $\vec{n}$ can be written as follows:

$$F_{\vec{n}}(q) = F(q) \cdot \vec{n} = \begin{bmatrix} \rho(u_x n_x + u_y n_y) \\ \rho(u_x^2 n_x + u_x u_y n_y) + p n_x \\ \rho(u_x u_y n_x + u_y^2 n_y) + p n_y \\ (E+p)(u_x n_x + u_y n_y) \end{bmatrix} \tag{2.10}$$

where $F(q)$ is a tensor containing the Euler flux functions provided in (2.2) and (2.3), respectively for $\vec{n} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\vec{n} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$.

2.2 Computational framework

2.2.1 Introducing the finite volume method for Cartesian grids

The finite volume method is a numerical scheme used to solve time-dependent two-dimensional systems of conservation laws given by the equation:

$$\frac{\partial q}{\partial t} + \frac{\partial f(q)}{\partial x} + \frac{\partial g(q)}{\partial y} = 0.$$

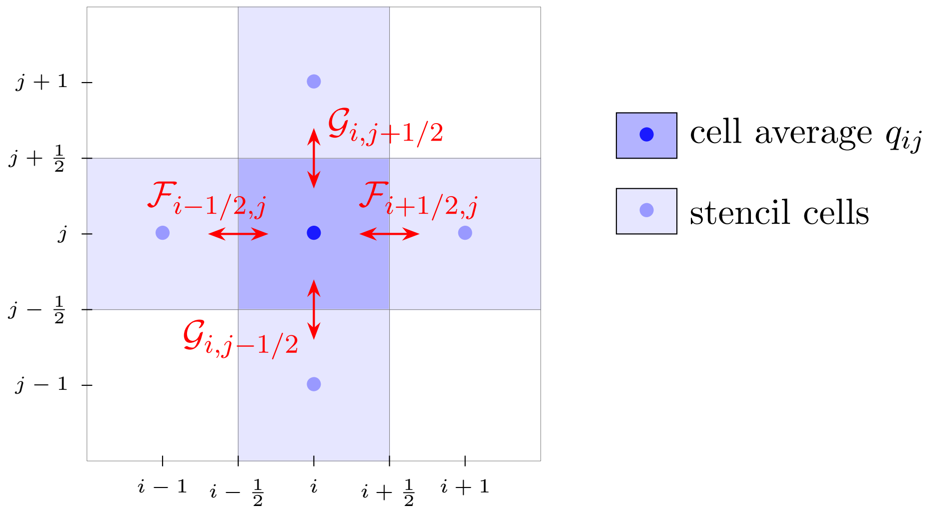

Figure 1: Update of the cell averages on a Cartesian grid in two dimensions.

In this framework, the computational domain is divided into a set $\mathcal{I}$ of discrete volumes or cells. Let us denote by $I_{i,j}$ the Cartesian cell located at the i-th row and the j-th column of the Cartesian grid, having an area of $\Delta x \times \Delta y$. The average value of the solution variable q within a cell $I_{i,j}$ at time level n is denoted as $q_{i,j}^n \approx \frac{1}{\Delta x} \int_{[x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}]} \frac{1}{\Delta y} \int_{[y_{j-\frac{1}{2}}, y_{j+\frac{1}{2}}]} q(x_i, y_j, t^n) dy dx$.

The finite volume scheme updates the cell average values in a single step using the following formula

$$q_{i,j}^{n+1} = q_{i,j}^n + \frac{\Delta t}{\Delta x}\left(\mathcal{F}^{\text{num}}_{i-\frac{1}{2},j} - \mathcal{F}^{\text{num}}_{i+\frac{1}{2},j}\right) + \frac{\Delta t}{\Delta y}\left(\mathcal{G}^{\text{num}}_{i,j-\frac{1}{2}} - \mathcal{G}^{\text{num}}_{i,j+\frac{1}{2}}\right)$$

where Δt is the time step, Δx and Δy are the cell sizes in the x and y directions. Therein, $F^{\text{num}}_{i-\frac{1}{2},j}$ and $F^{\text{num}}_{i+\frac{1}{2},j}$ designate the numerical fluxes (be it the Rusanov or HLL fluxes) across the interface boundaries in the x-direction, and $G^{\text{num}}_{i,j-\frac{1}{2}}$ and $G^{\text{num}}_{i,j+\frac{1}{2}}$ designate the numerical fluxes across the interface boundaries in the y-direction.

The numerical scheme is subject to a time-step restriction $\Delta t \frac{\max_{(i,j)\in[0,N_x]\times[0,N_y]} |\lambda(q_{i,j})|}{\min(\Delta x, \Delta y)} \leq \frac{1}{2}$, where $\lambda(q_{i,j})$ is an eigenvalue of the Jacobian matrix of the flux function f or g, depending on the direction of the interface boundary. In that case, $q_{i,j}$ is the vector of the solution on the cell $I_{i,j}$.

2.2.2 Introducing the finite volume method for triangular meshes

The finite volume method for triangular meshes is different from the one for Cartesian meshes explained above. [5]

The major difference is seen in the computation of the time step Δt, which will be explained later in this section, in paragraph 2.2.2.4 of the section related to the finite volume method for triangular meshes.

Discretization of the computational domain The implementation of the finite volume method for triangular meshes first needs a proper discretization of the computational domain. In that case, the cells are triangles, and the interface boundaries are the edges of any neighboring triangles.

Let us denote by I_i and I_j two neighboring triangular cells, e_{ij} the edge shared by both cells, and $|e_{ij}|$ the length of the latter. The normal vector pointing from I_i to I_j will be denoted as $\vec{n}_{ij}$. Do remember that the normal vector $\vec{n}_{ij}$ is the unit vector orthogonal to the edge e_{ij} and pointing from I_i to I_j.

The computation of the normal vector $\vec{n}_{ij}$ is mathematically explained in the section related to the implementation of the numerical scheme for triangular meshes.

Also, let us call $\mathcal{A}_i$ and $\mathcal{A}_j$ the respective areas of I_i and I_j, and x_i and x_j the respective barycenters of I_i and I_j.

Also, let us designate by $\mathcal{V}_i$ the set of the neighboring cells of the cell I_i.

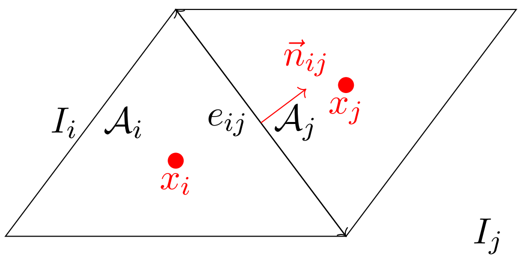

Figure 2: Visualization of the discretization of the computational domain for the finite volume method on triangular meshes.

Reformulating the Euler equations in two dimensions The Euler equations provided in (2.1) can be reformulated as follows:

$$\begin{cases} q_t + \nabla \cdot F(q) = 0 \\ q(x,0) = q_0(x) \end{cases} \tag{2.11}$$

where q_0 corresponds to the state variables vector of the initial condition, q the state variables vector of the solution and $F(q)$ the Euler flux tensor.

Integrating the Euler equations over a cell Let us now integrate the Euler equations (2.11) over a cell I_i of area $\mathcal{A}_i$ and over a duration $[t^n, t^{n+1}]$.

If we average the equation over the cell I_i, we obtain

$$\frac{1}{\mathcal{A}_i}\frac{1}{\Delta t}\int_{I_i}\int_{t^n}^{t^{n+1}} q_t(x,t)dtdx + \frac{1}{\mathcal{A}_i}\frac{1}{\Delta t}\int_{t^n}^{t^{n+1}}\int_{I_i} \nabla \cdot F(q)dxdt = 0. \tag{2.12}$$

The first integral of the left-hand side of the equation (2.12) can be rewritten as

$$\frac{1}{\mathcal{A}_i}\frac{1}{\Delta t}\int_{I_i}\int_{t^n}^{t^{n+1}} q_t(x,t)dtdx = \frac{1}{\mathcal{A}_i}\frac{1}{\Delta t}\int_{I_i} q(x,t^{n+1})dx - \frac{1}{\mathcal{A}_i}\frac{1}{\Delta t}\int_{I_i} q(x,t_n)dx \tag{2.13}$$

Then, if we suppose that the numerical solution is a piecewise constant function, the integral of the state variable vector $q(x,t^n)$ over the cell I_i during the time interval Δt can be formulated as

$$\frac{1}{\mathcal{A}_i}\int_{I_i} q(x,t_n)dx \approx q_i^n.$$

Then, the equation (2.13) can be approximated with

$$\frac{1}{\mathcal{A}_i}\frac{1}{\Delta t}\int_{I_i}\int_{t^n}^{t^{n+1}} q_t(x,t)dtdx \approx \frac{1}{\Delta t}(q_i^{n+1} - q_i^n). \tag{2.14}$$

Furthermore, relying on the **divergence Therorem**, which stipulates that $\int_{I_i} \nabla \cdot F(q)dx = \int_{\partial I_i} F(q) \cdot \vec{n}ds$, we can rewrite the second integral of the left-hand side of the equation (2.12) as

$$\frac{1}{\mathcal{A}_i}\frac{1}{\Delta t}\int_{t^n}^{t^{n+1}}\int_{I_i} \nabla \cdot F(q)dxdt = \frac{1}{\mathcal{A}_i}\sum_{j\in\mathcal{V}_i}\int_{e_{ij}}\frac{1}{\Delta t}\int_{t^n}^{t^{n+1}} F(q)\cdot\vec{n}_{ij}dtds. \tag{2.15}$$

The average value of the flux tensor $F(q)$ can be approximated with the utilization of a numerical flux function $\mathcal{F}$, which is a function of two states q_i and q_j and the normal vector $\vec{n}_{ij}$ of the edge e_{ij} for a first order scheme.

In that discretization framework, let us denote by $F_{i,j}^n$ the numerical flux across the edge e_{ij} at time t^n with

$$\mathcal{F}_{ij}^n = \mathcal{F}(q_i^n, q_j^n; \vec{n}_{ij}) \approx \frac{1}{\Delta t}\int_{t^n}^{t^{n+1}} F(q)\cdot\vec{n}_{ij}dt. \tag{2.16}$$

Now, after substituting (2.16) into the second integral of the left-hand side of the equation (2.12), we obtain:

$$\frac{1}{\mathcal{A}_i}\frac{1}{\Delta t}\int_{t^n}^{t^{n+1}}\int_{I_i} F(q)\cdot\vec{n}dxdt \approx \frac{1}{\mathcal{A}_i}\sum_{j\in\mathcal{V}_i}|e_{ij}|\mathcal{F}_{ij}^n. \tag{2.17}$$

Finally, combining (2.14) and (2.17) and substituting it into the equation (2.12) provides the following expression:

$$\frac{1}{\Delta t}(q_i^{n+1}-q_i^n)+\frac{1}{\mathcal{A}_i}\sum_{j\in\mathcal{V}_i}|e_{ij}|\mathcal{F}_{ij}^n = 0. \tag{2.18}$$

Updating the state variables vector Therefore, the finite volume method for triangular meshes updates the state variables vector q_i on the cell I_i at time t^{n+1} from time t^n with the following formula:

$$q_i^{n+1} = q_i^n - \frac{\Delta t}{\mathcal{A}_i}\sum_{j\in\mathcal{V}_i}|e_{ij}|\mathcal{F}_{ij}^n. \tag{2.19}$$

Adapting the time step The time step Δt is bounded by the following CFL condition:

$$\Delta t \max_{i\in\mathcal{I}}\left(\frac{1}{\mathcal{A}_i}\sum_{j\in\mathcal{V}_i}|e_{ij}|\left(\max_{j\in\mathcal{V}_i}\left(|\lambda(q_i)|,|\lambda(q_j)|\right)\right)\right)\leq\frac{1}{2}, \tag{2.20}$$

where $\mathcal{I}$ is the set of all the cells of the computational domain, and $\lambda(q_i)$ and $\lambda(q_j)$ are any given eigenvalues of the fluxes associated to the state variables vectors q_i and q_j respectively.

In that case, the maximum eigenvalue correspond to the maximal signal speed of the system, which depends on the sound speed a provided in (2.9).

Let us notice that the time step Δt for Cartesian grids is a particular expression of the time step Δt for triangular meshes, where the area $\mathcal{A}_i$ of a cell I_i is equal to $\Delta x\Delta y$ and the length $|e_{ij}|$ of an edge e_{ij} is equal to Δx or Δy depending on the direction of the edge.

2.2.3 Introducing the numerical fluxes in two dimensions

Rusanov flux The Rusanov flux in a two-dimensional domain is a function of two states vectors q_i and q_j and a normal vector $\vec{n}$. Finally, the Rusanov flux $F_{i,j}^{Rusanov}$ across the edge e_{ij} can be defined as follows:

$$\mathcal{F}_{ij}^{\text{Rusanov}} = \frac{1}{2}\left(F_{\vec{n}_{ij}}(q_i)+F_{\vec{n}_{ij}}(q_j)\right)-\frac{1}{2}\max_{\lambda\in\mathcal{Q}}|\lambda|\left(q_j-q_i\right),$$

where $F_{\vec{n}_{ij}}(q_i)$ and $F_{\vec{n}_{ij}}(q_j)$ are the Euler fluxes in the direction of the normal vector $\vec{n}_{ij}$ respectively on the cells I_i and I_j, as defined in (2.10). Therein, $\mathcal{Q}$ is the set of all the eigenvalues of the fluxes associated to the state variables vectors q_i and q_j respectively and given by $\mathcal{Q} = \{Sp(f'(q_i))\cup Sp(g'(q_i))\cup Sp(f'(q_j))\cup Sp(g'(q_j))\}$.

HLL flux Harten, Lax, and van Leer introduced an approximate Riemann solver known as the HLL solver. [3]

The solver is defined as follows:

$$\tilde{q}(x,t) = \begin{cases} q_i & \text{if } \frac{x}{t} \leq S_i, \\ q^{\text{HLL}} & \text{if } S_i \leq \frac{x}{t} \leq S_j, \\ q_j & \text{if } \frac{x}{t} \geq S_j. \end{cases}$$

Therein, q^{HLL} represents a constant state vector, and $S_i = S(I_i, \vec{n}_{ij})$ and $S_j = S(I_j, \vec{n}_{ij})$ are assumed to be known signal speeds associated to the state variable vectors of cells I_i and I_j.

The signal speeds S_i and S_j satisfy the following expressions:

$$S_i = u_i \cdot \vec{n}_{ij} - a_i \text{ and } S_j = u_j \cdot \vec{n}_{ij} + a_j$$

where u_i and u_j are the velocities of the state variables vectors q_i and q_j respectively, and a_i and a_j are the sound speeds of the state variables vectors q_i and q_j respectively, as defined in (2.9).

The structure of the HLL solver consists of three constant states separated by two waves in one dimension: those two waves respectively possess the following signal speed $u - a$ and $u + a$, where u is the velocity of a given state variables vector and a is the sound speed of the same state variables vector. The star region corresponds to an intermediary constant state for waves of wavespeed inbetween those of the two former waves, while all intermediate states between the waves approximated by the Euler flux functions.

For the two-dimensional case, the idea is to expand this logic for each direction, which are associated with the normal vectors of a given triangular cell.

The HLL flux, specifically at the star region, is then provided by

$$F_{ij}^{\text{HLL}} = \frac{S_j F_j - S_i F_i + S_i S_j (q_i - q_j)}{S_j - S_i},$$

where $F_i = F_{\vec{n}}(q_i)$ and $F_j = F_{\vec{n}}(q_j)$ are the Euler fluxes in the direction of the normal vector $\vec{n}$, as defined in (2.10).

The HLL numerical flux is finally given by

$$\mathcal{F}_{ij}^{\text{HLL}} = \begin{cases} F_i & \text{if } S_i > 0, \\ F_{ij}^{\text{HLL}} & \text{if } S_i \leq 0 \leq S_j, . \\ F_j & \text{if } S_j < 0. \end{cases}$$

2.3 Implementation

2.3.1 Debugging and completing the unfinished code for the implementation on 2D Cartesian meshes (before extending it for any normal vectors)

The debugging and the completion of the unfinished code in for the first-order implementation in two dimensions can be decomposed in four steps:

- The first step consisted of debugging the functions related to the state variables vectors. Indeed, in many test cases the relevant state variables to observe are not exactly those contained in the components of the vector q, but rather the density ρ, the velocites u_x and u_y for both respective directions, and the pressure p. Let us call such state variables the primitive variables. All the relations between the conservative variables vector and the primitive variables are provided in (2.4) and (2.5). I already implemented the functions related to the conversion of the conservative variables vector into the primitive variables vector and vice versa in the one-dimensional case. However, the operations inside the function did not provide the proper results. The input was in that case an array containing all the conservative variables for each cell and the output was an array of the same size storing all the primitive variables for each cell. Therefore, I decided to use array-related operations instead of loops to compute the primitive variables vector from the conservative variables vector and vice versa. Furthermore, I had to rectify the location of the density ρ in the expression of the computation of the total energy E from the primitive variables vector. My correction of the functions was successful.

- The second step was about debugging the functions related to the fluxes. My functions were working properly, as the plottings of density of the transport problem test case with periodic boundary conditions confirmed that the HLL flux was less diffusive than the Rusanov flux. In fact, the curb of the density of final solution of the transport problem was thiner for the HLL flux than for the Rusanov flux. This property was observed despite the unrectified bug (during my project) related to higher dilation of the numerical solution in the y-direction than in the x-direction. I still decided to provide one small cosmetic change to the functions related to the fluxes. Instead of outputing multiple variables, for each direction, all the fluxes values (for each neighboring state) and the eigenvalues, I created an array containing all the fluxes values and an array containing all the eigenvalues. This change was not necessary but it was more convenient for the next steps.

- The third step dealt with the debugging of the functions related to the numerical scheme. As said in my previous project report, I accidentally skipped a line in the loop that updates the state variables vector, just before the term updating it for the y-direction. After I solved this bug and used a CFL number of 0.5 for the computation of the time step Δt, the numerical scheme was working properly.

- The fourth step involved the checking of the evolution of the error between the numerical approximation and the exact solution of the Euler equations, for the transport problem with periodic boundary conditions. The initial solution is the exact solution in such case. Actually, numerical schemes tend to lead to the phaenomenon of numerical (or computational) diffusion, which is analogous to the idea of physical diffusion (for quantities like heat). It induces dispersive and flattening effects, visually speaking, and provides erroneous solutions to a certain degree. Studying the density of the final solution of the transport problem test is relevant for the study of computational diffusion, through data visualization or error computations. I created a class in that purpose for two reasons: to make me learn and use the object-oriented programming paradigm in Julia, and to make the routine of the error computation more convenient. In reality, a class in Julia is rather a mutable structure that contains fields and methods. The fields are the variables that are stored in such a structure and the methods are the functions that are associated to the structure. It is mutable in a sense that its fields can be modified after a particular structure has been instantiated (therefore, interesting for post-processing purposes). The mutable structure I created was called `Euler_2D` and was composed of several fields, including information related to the computational domain (the number of cells in the x-direction and in the y-direction, the length of the computational domain in the x-direction and in the y-direction), the initial data (containing both the initial solution q_{ini} and the final solution q_{final}), the used numerical flux (Rusanov or HLL) and the final time. The results were succesful and the convergence rates were close to 1. All of this will be dicussed later in this report, in the 3.3 section.

2.3.2 Implementation of the first order finite volume method in two dimensions for unstructured triangular meshes

The implementation of the finite volume method for the Euler equations in two dimensions for unstructured triangular meshes was significantly different from the one for Cartesian meshes. The major difference is seen for the computation of the time step Δt, which is showed in section 2.2.2.

I relied on the software GMSH to generate the mesh. GMSH is a free and open-source software that is used to generate meshes.

The implementation of the first order scheme in two dimensions for unstructured triangular meshes in Julia is decomposed in five steps

- The first step consisted of creating a mutable structure that reads the mesh file generated by GMSH and parses the coordinates of the nodes and the connectivity matrix. The connectivity matrix is the matrix that contains the indices of the nodes that form each triangular cell (or more widely, every two-dimensional simplexes including rectangles, squares or rombuses) of the mesh. Even though I was expected to achieve this step in a longer time (by Monday 3th July after being introduced to that part on Thursday 22th June), I still managed to finish it in three days. The reason is that I was already familiar with GMSH, as I had one project related it in a project for my C++ course during the first semester. Moreover, thanks to a previous practical work related to partial differential equations course (during the second semester), I had to implement the finite element method in Python for the Poisson equation. And I indeed had to create a Python class that parses the information of a mesh file generated by GMSH. So using the packages `DelimitedFiles` (equivalent of the `os` library in Python) and `LinearAlgebra` (equivalent of the `NumPy` library in Python) of Julia, I translated my Python code into Julia. The information was parsed in many fields, including an array containing the nodes and another one being the connectivity table, and also the areas. The function was working properly and done in three days.

- In the second step I have created a function that computes the normals of the edges of the mesh. It took me one day to achieve it, relying on the fact that the normal vector of an edge is the unit vector orthogonal to the distance vector formed by the two nodes of the edge. I computed the latter vector by subtracting the coordinates of the two nodes of the edge. Since, the orientation of the nodes, as suggested by the connectivity matrix, was clockwise, I just had to reverse the order and the signs of the nodes to obtain the normal vector pointing from a given cell towards a given neighboring cell.

  If we denote, for a given triangular cell I_i, by $M_{i1}(x_1, y_1)$, $M_{i2}(x_2, y_2)$ and $M_{i3}(x_3, y_3)$ the three nodes of such a cell, the three edges of the cell e_{12}, e_{23} and e_{31} can be written as follows:

$$e_{12} = M_2 - M_1 = (x_2 - x_1, y_2 - y_1),$$

$$e_{23} = M_3 - M_2 = (x_3 - x_2, y_3 - y_2),$$

and

$$e_{31} = M_1 - M_3 = (x_1 - x_3, y_1 - y_3).$$

Therefore, using these notations and my understanding of the orientation of the nodes, the normal vectors $\vec{n}_{12}$, $\vec{n}_{23}$ and $\vec{n}_{31}$ of the edges e_{12}, e_{23} and e_{31} respectively can be written as follows:

$$\vec{n}_{12} = (y_1 - y_2, x_2 - x_1),$$

$$\vec{n}_{23} = (y_2 - y_3, x_3 - x_2),$$

and

$$\vec{n}_{31} = (y_3 - y_1, x_1 - x_3).$$

- In the third step, I tackled the computation of the edge lengths, which are indeed needed for the computation of the time step. I also finished it in one day, since the process was about, for each element, computing the distance between every pair (among a total of three) of consecutive nodes of the connectivity matrix. I used the Julia function named `norm` taken from the `LinearAlgebra` package to compute the distance between two nodes. The expression of the edge lengths are provided above.

- During the fourth step, I have written a function searching the neighbors of the elements in the mesh. Such a decision was made in order to pave the way for the implementation of periodic boundary conditions (which are actually needed, as a reminder, for both the transport problem and the Gresho vortex). It was for me the second most challenging step of the implementation. It took me around eight days to find out a proper result. I added four fields in the `Mesh2D` class (mutable struct) that contains the indexes of the nodes belonging to the four possible boundaries (left, right, bottom and top). I used the Julia function named `findall` to find such indexes. For each, I then collected the elements that share two consecutive nodes of the connectivity matrix (because two consecutive nodes of the connectivity matrix form an edge). If one of the indexes of the neighbors equaled to zero, it meant that the element was on the boundary. To implement the periodic boundary conditions, I wrote a procedure that replaces the zero indexes (inside the array containing the neighbours) with the index of the element from the boundary of the other side (for example, if the element is on the left boundary, the zero index is replaced by the index of the element on the right boundary). The function returned the proper results. I provided a final modification to neighbors-searching function after I have learnt from Andrea that it was possible to generate structured triangular meshes with GMSH. Before, I ignored for long that it was possible which is why I created from scratch a `Mesh` mutable structure in another notebook to implement the support for structured triangular mesh (since this variety of mesh was regular, I was able to build the class and the neighbor-searching function by myself). However, I still had the time to learn about how to create structured triangular meshes with GMSH, using the adding of a straight surface and straight lines. I then modified the neighbor-searching function to make it also work for structured triangular meshes. In fact, it was only designed for unstructured triangular meshes. There was still one bug related to getting the neighbors of the last element of the mesh (which was the bottom-right corner, having two null neighbors, since it must have two additional neigbors to set up the periodic boundary conditions). I solved it by writing one more non-returning procedure that replaces the zero indexes (inside the array containing the neighbours) with the index of the element from the boundary of the other side (for example, if the element is on the left boundary, the zero index is replaced by the index of the element on the right boundary). The function then returned the proper results. At least, knowing that GMSH could generate structured triangular meshes made me turn my neighbor-searching function into a more general one, which works for both triangular meshes.

- The fifth step was about the numerical scheme and I considered it as the hardest step of this part. In fact, this part was punctuated with many errors and faillures that sometimes, I really doubted whether I would succeed in it or not. The first problem, was that, even though the execution of the numerical scheme seemed working (taking some times), it provided a final solution that did not seem to differ from the initial solution. Especially for the transport problem test case. I then debugged my code by printing the time step under the advise of Andrea, only showing a dozen of iterations for the transport problem at a final time of 1.0 seconds. I was here performing a simulation on a 980-element unstructured triangular mesh. I consequently asked Andrea if she knew some material related to the utilization of the finite volume method for unstructured triangular meshes. She then brought me a paper from Victor Michel-Dansac that fulfilled my needs. After having succeeded in the implementation of the time step, the results displayed (again, for the transport problem test case with periodic boundary conditions) were intuitively fine but were provided in an overly long time. Using a mesh object associated with a GMSH mesh file containing $1,024$ elements, I took me 52 minutes to plot the density of the final solution of the transport problem test case at a final time equaling 1.0 seconds with the Rusanov flux. I was again pessimistic because of the possibility of delivering an undone work. Andrea however advised to look forward optimizing my code by removing unnecessary loops, for instance. I went through it by resorting to two major sets of modifications. And they made my code faster: on the one hand, I removed loops in favor of using array-based operations, and on the other hand, I wrote another class (mutable struct) named `MeshData` that contains pre-processed data that are used in the numerical scheme. The pre-processed data are the areas of the elements, the edge lengths, the normals of the edges and the neighbors of the elements. Doing so made me decrease the computation-time of a $1,024$-element mesh from 52 minutes to 45 seconds.

2.3.3 Implementation of the first order finite volume method in two dimensions for structured triangular meshes

After discussing with Andrea, who introduced me the Gresho vortex and the notion of the Mach number, she proposed me to compare the results of the test case for different mesh size, different numerical fluxes and different Mach number (M).

During this step, I plotted the mesh used (which will be shown later in the results section) and the initial and final solution of the Gresho vortex test case. The idea was primally to check whether there was at least less diffusion for the implementation in triangular meshes than for the one in Cartesian meshes.

Since she told me that she would not the available for the next week, she advised me to contact Victor to discuss my potential results if some were provided.

Implementing the finite volume method for structured triangular mesh took me two days: it was simpler than the unstructured triangular mesh version because the normal vectors of the edges were always the unit vectors of the x and y axes and the diagonal axis (which always had the same slope). Finding the neighboring elements was also easier because the mesh was structured. In that case, it systematically implements the periodic boundary conditions.

Besides, this personal contribution, I finally added the support for structured triangular meshes in the implementation loading mesh files that are generated by GMSH. I also included the support for periodic boundary conditions.

I still doubted about the possible discrepancies between the implementation for GMSH-generated meshes and the one for my own structured triangular meshes. I then decided to compare the plots of the densities and velocity fields of the final solutions of the Gresho vortex test case for both implementations. The results were similar, which heuristically confirmed that my personal and initial implementation was correct.

3 Results

3.1 Preliminary remarks regarding the plotting of the results

3.1.1 How I plotted the results

I relied on the `Plots` package of Julia to plot the results. The plots are given with respect to the mesh type (Cartesian, my own structured triangular mesh implementation, the structured and the unstructured triangular mesh implementation using files generated with GMSH), the numerical flux used (Rusanov or HLL) and the mesh size.

3.1.2 Plotting the meshes

I used a total of three different mesh formats for the plottings (transport density and Gresho vortex): Cartesian meshes, structured triangular meshes and unstructured triangular meshes, all of them having two sizes, whose number of elements will be provided below.

Cartesian meshes Both Cartesian meshes have exactly and respectively $50 \times 50 = 2,500$ elements and $100 \times 100 = 10,000$ elements.

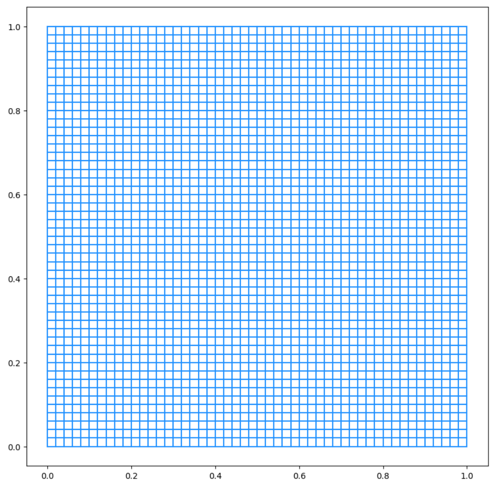

Figure 3: Cartesian mesh with $50 \times 50 = 2,500$ elements

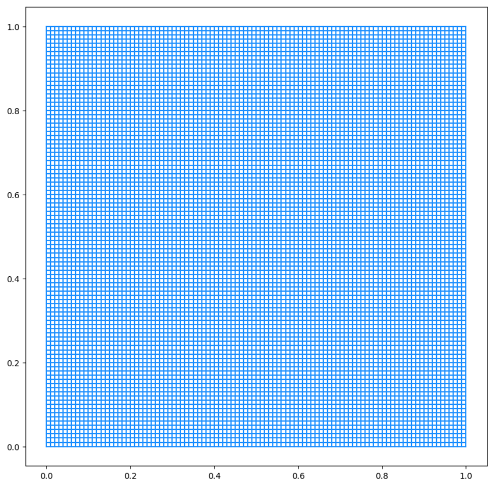

Figure 4: Cartesian mesh with $100 \times 100 = 10,000$ elements

Structured triangular meshes Since the structured triangular meshes are obtained by cuting in half the Cartesian meshes, the number of elements is slightly different from the Cartesian meshes.

Both structured triangular meshes have exactly $36 \times 36 \times 2 = 2,592 \approx 2,500$ elements and $71 \times 71 \times 2 = 10,082 \approx 10,000$ elements.

I resorted to the following relationship to compute the number of elements of the structured triangular meshes: $N_{\text{elements/dimension}} = \lfloor \frac{N_{\text{Cartesian/dimension}}}{\sqrt{2}} \rfloor + 1$.

Therein, $N_{\text{elements/dimension}}$ is the number of elements in one dimension of the structured triangular mesh, $N_{\text{Cartesian/dimension}}$ is the number of elements in one dimension of the Cartesian mesh, and $\lfloor \cdot \rfloor$ is the floor function.

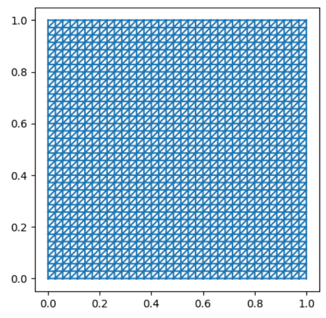
Figure 5: Structured triangular mesh with 2,592 elements

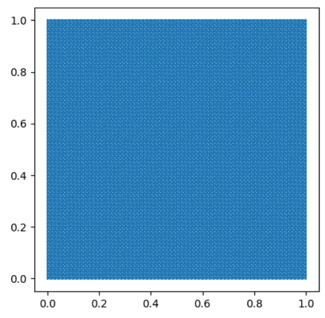
Figure 6: Structured triangular mesh with 10,082 elements

Unstructured triangular meshes Both unstructured triangular meshes have exactly $2,528 \approx 2,500$ elements and $10,756 \approx 10,000$ elements.

I have respectively chosen in GMSH the following values for the fourth coordinate parameters for all points : 0.032 and 0.015. For a given point, the fourth coordinate parameter is the target mesh size around that point. The smaller the value, the finer the mesh around that point.

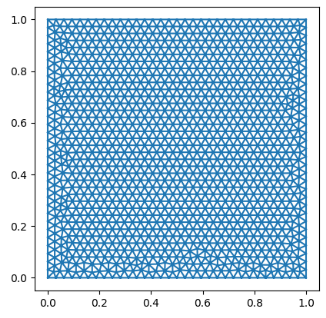
Figure 7: Unstructured triangular mesh with 2,528 elements

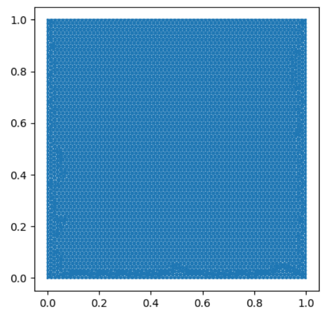
Figure 8: Unstructured triangular mesh with 10,756 elements

3.2 Transport problem test case

3.2.1 Introducing the transport problem test case

The problem of transport of mass density is about the advection of the latter through a given medium over time. Such a transport process can be described mathematically using partial differential equations or conservation laws. In the case of a two-dimensional transport problem, under the hypothesis of the fluid being a perfect gas and constant density and pressure, the Euler equations can turn into the continuity equation, which is given by

$$\frac{\partial \rho}{\partial t} + u \cdot \nabla \rho = 0, \tag{3.1}$$

where ρ is the density or scalar quantity being transported (in kg/m^3), t the time, u the velocity vector (in m/s) and ∇ the gradient operator.

Initial data We consider the problem of the transport of density with the following initial data. The physical quantities include

- The pressure $p = 1$,
- The velocity vector $\mathbf{u} = (1, 1)^T$,
- The density $\rho = 1 + \exp\left(-100\left((x - 0.5)^2 + (y - 0.5)^2\right)\right)$.

The computational domain is in that problem $[0, 1] \times [0, 1]$. The problem includes periodic boundary conditions in both the x-and-y-directions. Actually, periodic boundary conditions in both the x-and-y-directions are used to create a seamless and continuous computational domain by treating the boundaries (the left-right and top-bottom pairs) as if they were connected. It basically means that if a quantity or a variable exits the domain through one boundary, it re-enters the domain from the opposite boundary.

When it comes to numerical fluxes, periodic boundary conditions ensure that the values used to compute the fluxes at one boundary come from the corresponding location on the opposite boundary.

The final time is set to $t_{\text{final}} = 1.0$ because, since the velocity initially set to $(1, 1)^T$ and the periodic boundary conditions are applied, the density is expected to return to its initial position after 1.0 second.

An initial velocity of $(1, 1)^T$ qualitatively means that the density is expected to move diagonally in the computational domain.

The transport problem test case can be seen as a heuristic to check whether the numerical scheme is working properly or not, by taking into account the periodic boundary conditions.

3.2.2 Plottings of density of the final solution with respect to the mesh type

In this section, the mesh type is what varies: a 2,500-element Cartesian mesh, 2,592-element structured triangular mesh (from my personal implementation) and 2,528-element unstructured triangular mesh. The Mach number always equals 0.05 and the numerical flux is the Rusanov flux. The final time is set to $t_{\text{final}} = 1.0$ second.

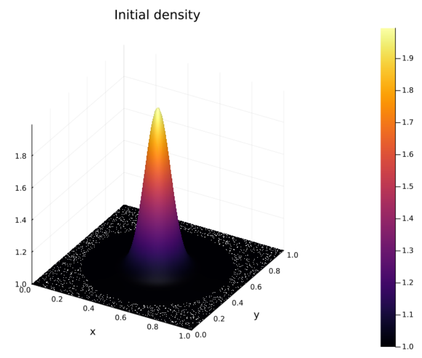

Figure 9: Initial density on a $2,500$-element Cartesian mesh using the Rusanov flux

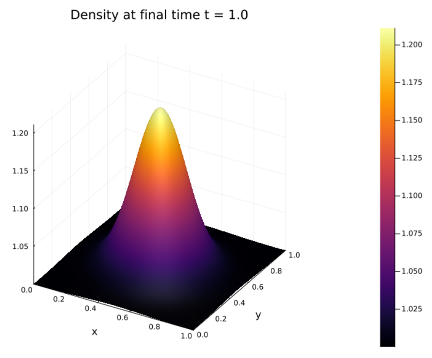

Figure 10: Final density on a $2,500$-element Cartesian mesh using the Rusanov flux

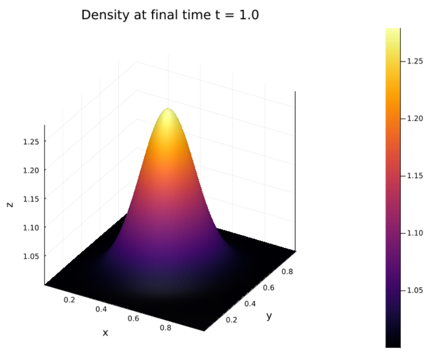

Figure 11: Final density on a $2,592$-element structured triangular mesh (personal implementation) using the Rusanov flux

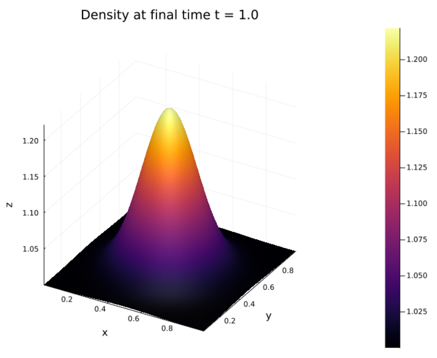

Figure 12: Final density on a $2,528$-element unstructured triangular mesh (GMSH) using the Rusanov flux

Observations and interpretations At first glance, for all mesh types, the density returns to its initial position after 1.0 second and has equal diffusion in both the x-and-y-directions for all the meshes and the numerical fluxes, which means that the numerical scheme is working properly, and that the periodic boundary conditions are properly implemented (for all mesh types).

However, as observed in all the figures above, the distribution of the values of the final densities tends to spread out more than for the initial densities. It is a sign of numerical diffusion, which is inherent to numerical schemes.

Furthermore, when it comes to Cartesian meshes, the maximal value of the final density, using the Rusanov flux, is ≈ 1.21 for the $2,500$-element mesh. Meanwhile, the structured triangular mesh (personal implementation) and the unstructured triangular mesh (GMSH) have a maximal value of the final density of ≈ 1.225 and ≈ 1.25 respectively. It implies that the diffusion is lower for triangular meshes than for Cartesian meshes.

3.2.3 Plottings of density of the final solution with respect to the mesh size

In this section, the mesh size is what varies: a $2,500$-element Cartesian mesh and $10,000$-element Cartesian mesh. The numerical flux utilized is the HLL flux. The final time is set to $t_{\text{final}} = 1.0$ second.

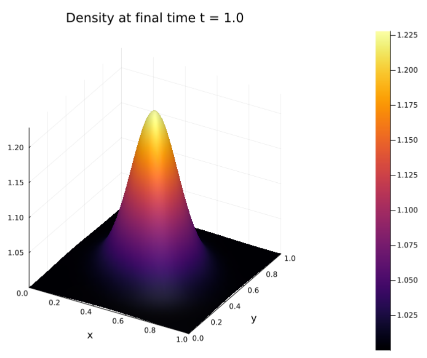

Figure 13: Final density on a $2,500$-element Cartesian mesh using the HLL flux

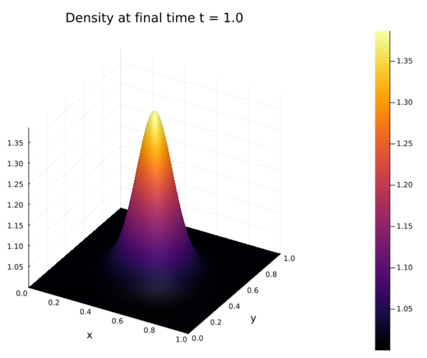

Figure 14: Final density on a $10,000$-element Cartesian mesh using the HLL flux

Observations and interpretations For Cartesian meshes, the maximal value of the final density, using the Rusanov flux, is ≈ 1.225 for the $50 \times 50 = 2,500$-element mesh and ≈ 1.38 for the $100 \times 100 = 10,000$-element mesh. It means that increasing the mesh size leads to lower diffusion. Let us remember that the density of the exact solution (or synonymously, the density of the exact solution) peaks at 2.0.

3.2.4 Plottings of density of the final solution with respect to the numerical flux

In this section, the numerical flux is what varies: either the Rusanov flux or the HLL flux are used to provide the results below. The mesh utilized is $10,786$-element unstructured triangular mesh. The final time is set to $t_{\text{final}} = 1.0$ second.

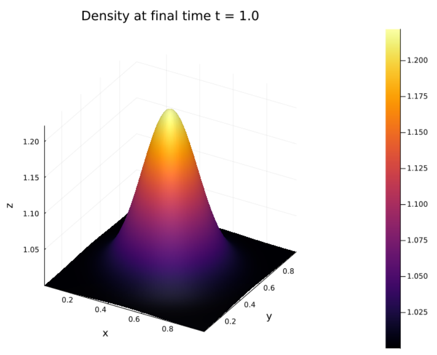

Figure 15: Final density on a $10,786$-element unstructured triangular mesh using the Rusanov flux

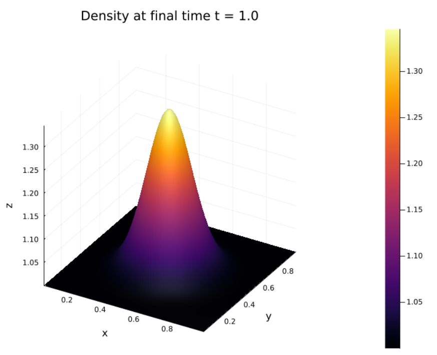

Figure 16: Final density on a $10,786$-element unstructured triangular mesh using the HLL flux

Observations and interpretations Utilizing the HLL flux also provides a lower diffusion than the Rusanov flux. For instance, on a $10,786$-element unstructured triangular mesh (GMSH-generated), the maximal value of the final density is ≈ 1.22 for the Rusanov flux and ≈ 1.35 for the HLL flux.

This observation relates to the fact that the Rusanov depends on extremal signal speeds (which are the extremal eigenvalues of the Jacobian matrix of the flux function), while the HLL flux depends on more intermediary signal speeds. The Rusanov flux is therefore more diffusive than the HLL flux, as seen in the paragraph introducing the Rusanov flux.

3.3 Errors and convergence rates

3.3.1 Interest of studying the convergence rates

The transport problem test case is also a good way to check the convergence rates of the numerical scheme. The errors that I computed were the L^1, the L^2 and the L^∞ errors. I have chosen the mesh sizes in such a way that the step size h was divided by 2 at each time, when it comes to structured meshes. For unstructured meshes, I have manually chosen the fourth coordinate parameters for all points in GMSH in such a way that this pattern was respected as much as possible.

The convergence rates should be approaching 1 for the L^1, the L^2 and the L^∞ errors.

3.3.2 Computing the errors

Mesh-size parameter h On the one hand, for Cartesian meshes, the mesh-size parameter h equals the length of the edge of a square cell (so $h = \Delta x = \Delta y$, where Δx and Δy are the length of the edges of the cells in the x-and-y-directions respectively).

On the other hand, for triangular meshes, the h parameter was rather the maximum edge length within the mesh. Thus, $h = \max_{i\in\mathcal{I}} \left(\max_{j\in\mathcal{V}_i} \left(|e_{ij}|\right)\right)$ where $\mathcal{I}$ is the set of all the cells of the computational domain and $\mathcal{V}_i$ the set of the neighboring cells of the cell I_i.

Error computation The L^1, the L^2 and the L^∞ errors are defined as follows:

$$\begin{aligned}
e_{L^1} &= \sum_{i\in\mathcal{I}} |q^i_{\text{final}} - q^i_{\text{exact}}|\mathcal{A}_i, \\
e_{L^2} &= \sqrt{\sum_{i\in\mathcal{I}} |q^i_{\text{final}} - q^i_{\text{exact}}|^2\mathcal{A}_i}, \\
e_{L^\infty} &= \max_{i\in\mathcal{I}} |q^i_{\text{final}} - q^i_{\text{exact}}|.
\end{aligned} \tag{3.2}$$

where q_{final} is the numerical approximation of the final solution at the final time t_{final}, q_{exact} is the exact final solution at the final time t_{final}, $\mathcal{I}$ is the set of all the cells of the computational domain, $\mathcal{A}_i$ is the area of the cell I_i.

Defining the convergence rate The convergence rate is the rate at which the numerical solution converges on the exact solution as the mesh size decreases. It is defined in the following way

$$r = \frac{\log(e(h_2)/e(h_1))}{\log(h_2/h_1)}, \tag{3.3}$$

where $e(h_1)$ and $e(h_2)$ are errors computed with the mesh sizes h_1 and h_2 respectively.

3.3.3 Results for Cartesian meshes

Parameters Here the parameters are the mesh size, the mesh type and the error type (the L^1, the L^2 and the L^∞ errors). Let us remind that the final time is set to $t_{\text{final}} = 1.0$ second and that the initial solution is the exact solution everytimes. Actually, the initial solution is not time-dependent.

For the errors and the convergence rates provided in the tables below, I have chosen to keep four significant digits.

Mesh Size (h)	L^1 **error**	L^2 **error**	L^∞ **error**
0.0125	1.438	0.9543	3.269
0.00625	1.039	0.7656	2.776
0.003125	0.06881	0.5472	2.098
0.0015625	0.04112	0.03482	1.395

Table 1: Table of errors using the Rusanov flux for the transport problem test case with a Cartesian mesh

Mesh Size (h)	L^1 **convergence rate**	L^2 **convergence rate**	L^∞ **convergence rate**
0.0125			
0.00625	0.4557	0.3177	0.2357
0.003125	0.6077	0.4847	0.4041
0.0015625	0.7426	0.6521	0.5889

Table 2: Table of convergence rates using the Rusanov flux for the transport problem test case with a Cartesian mesh

Mesh Size (h)	L^1 **error**	L^2 **error**	L^∞ **error**
0.0125	0.09989	0.07342	3.269
0.00625	0.06465	0.05137	2.776
0.003125	0.03811	0.03222	2.098
0.0015625	0.02107	0.01853	1.395

Table 3: Table of errors using the HLL flux for the transport problem test case with a Cartesian mesh

Mesh Size (h)	L^1 **convergence rate**	L^2 **convergence rate**	L^∞ **convergence rate**
0.0125			
0.00625	0.6302	0.5152	0.4320
0.003125	0.7600	0.6731	0.6011
0.0015625	0.8552	0.7975	0.7433

Table 4: Table of convergence rates using the HLL flux for the transport problem test case with a Cartesian mesh

3.3.4 Results for structured triangular mesh (my implementation)

Mesh Size (h)	L^1 **error**	L^2 **error**	L^∞ **error**
0.03536	0.1611	0.9543	0.2051
0.01768	0.1248	0.7656	0.1739
0.008838	0.08780	0.5472	0.1338
0.004419	0.05598	0.03482	0.09190

Table 5: Table of errors using the Rusanov flux for the transport problem test case with a structured triangular mesh (my implementation)

Mesh Size (h)	L^1 **convergence rate**	L^2 **convergence rate**	L^∞ **convergence rate**
0.03536			
0.01768	0.3677	0.2483	0.1650
0.008838	0.5076	0.3781	0.3061
0.004419	0.6491	0.5417	0.4765

Table 6: Table of convergence rates using the Rusanov flux for the transport problem test case with a structured triangular mesh (my implementation)

Mesh Size (h)	L^1 **error**	L^2 **error**	L^∞ **error**
0.03536	0.1260	0.1753	0.7543
0.01768	0.08776	0.1334	0.6086
0.008838	0.05534	0.09047	0.4347
0.004419	0.03213	0.05529	0.2758

Table 7: Table of errors using the HLL flux for the transport problem test case with a structured triangular mesh (my implementation)

Mesh Size (h)	L^1 **convergence rate**	L^2 **convergence rate**	L^∞ **convergence rate**
0.03536			
0.01768	0.5220	0.3944	0.3095
0.008838	0.6551	0.5602	0.4856
0.004419	0.7847	0.7104	0.6563

Table 8: Table of convergence rates using the HLL flux for the transport problem test case with a structured triangular mesh (my implementation)

3.3.5 Results for unstructured triangular mesh (GMSH)

Mesh Size (h)	**L^1 error**	**L^2 error**	**L^∞ error**
0.03157	0.1531	0.1987	0.8427
0.01631	0.1152	0.1641	0.7352
0.008042	0.07836	0.1220	0.5772
0.004027	0.04807	0.08028	0.03992

Table 9: Table of errors using the Rusanov flux for the transport problem test case with an unstructured triangular mesh (GMSH)

Mesh Size (h)	**L^1 convergence rate**	**L^2 convergence rate**	**L^∞ convergence rate**
0.03157			
0.01631	0.4308	0.2896	0.2066
0.008042	0.5448	0.4199	0.3421
0.004027	0.7067	0.6049	0.5333

Table 10: Table of convergence rates using the Rusanov flux for the transport problem test case with an unstructured triangular mesh (GMSH)

Mesh Size (h)	**L^1 error**	**L^2 error**	**L^∞ error**
0.03157	0.1209	0.1698	0.7478
0.01631	0.08300	0.1276	0.5984
0.008042	0.05176	0.08556	0.4228
0.004027	0.04807	0.08028	0.2644

Table 11: Table of errors using the HLL flux for the transport problem test case with an unstructured triangular mesh (GMSH)

Mesh Size (h)	**L^1 convergence rate**	**L^2 convergence rate**	**L^∞ convergence rate**
0.03157			
0.01631	0.5696	0.4332	0.3373
0.008042	0.6675	0.5650	0.4914
0.004027	0.8082	0.7353	0.6789

Table 12: Table of convergence rates using the HLL flux for the transport problem test case with an unstructured triangular mesh (GMSH)

3.3.6 Observations and conclusion

When it comes to the utilization of Cartesian meshes, the final convergence rates are 0.7426 (L^1 norm), 0.6521 (L^2 norm), 0.5889 (L^∞ norm) for using Rusanov flux and 0.8552 (L^1 norm), 0.7975 (L^2 norm), 0.7433 (L^∞ norm) for using HLL flux. It confirms the fact that the convergence rates are approaching 1 for the L^1, the L^2 and the L^∞ errors for the less diffusive HLL flux.

Sticking to the HLL flux, the convergence rates for the L^1 norm are the following ones: 0.7600 for the Cartesian mesh (penultimate value), 0.7847 (last value) for the structured triangular mesh (my implementation) and 0.8082 (last value) for the unstructured triangular mesh (GMSH). I decide to compare these values because the mesh sizes are similar for the three meshes: from $h = 0.00625$ to $h = 0.003125$ for the Cartesian, from $h = 0.008838$ to $h = 0.004419$ for the structured triangular mesh (my implementation) and from $h = 0.008042$ to $h = 0.004027$ for the unstructured triangular mesh (GMSH). The convergence rates for the L^1 norm are higher for the structured triangular mesh (my implementation) and for the unstructured triangular mesh (GMSH) than for the Cartesian mesh.

For the Rusanov flux and the unstructured triangular mesh (GMSH), the final convergence rates are 0.7067 (L^1 norm), 0.6049 (L^2 norm), 0.5333 (L^∞ norm). It implies that the convergence rates for the L^1 norm are higher than for the L^2 norm, themselves higher than for the L^∞ norm.

To conclude, the convergence rates are approaching 1 for the L^1, the L^2 and the L^∞ errors: utilizing the HLL flux leads to higher convergence rates than utilizing the Rusanov flux and relying on triangular meshes leads to higher convergence rates than relying on Cartesian meshes. Moreover, the L^1 convergence rates are higher than the L^2 convergence rates, themselves higher than the L^∞ convergence rates.

3.4 Plotting the Gresho vortex test case for different Mach numbers

3.4.1 Introducing the Gresho vortex test case

The Gresho vortex can first be mathematically understood as a steady-state or stationary solution of the Euler equations. It is a system inside which the gradient of the pressure is balanced by the centrifugal force.

It is a good test case to check the diffusivity of numerical solvers based on a parameter that is the Mach number. It will be introduced in the 3.4.2 section.

Again, the periodic boundary conditions are used in both the x-and-y-directions for the Gresho vortex test case.

Initial conditions

We consider the problem of the Gresho vortex with the following initial data. The physical quantities are

- the adiabatic index $\gamma = 1.4$ (ratio of specific heats),
- the initial density $\rho = 1.0$,
- the initial pressure $p_0 = \frac{\rho_0}{\gamma \cdot \mathrm{M}^2}$,
- the coordinates of the center of the domain (which is $[0,1] \times [0,1]$) $(x_0, y_0) = (0.5, 0.5)$,
- the radius of the vortex $r = \sqrt{(x - x_0)^2 + (y - y_0)^2}$,
- the angle $\phi = \arctan(y - y_0, x - x_0)$,
- the velocity in the ϕ-direction

$$u_\phi = \begin{cases} 5.0 \cdot r & \text{if } r < 0.2, \\ 2.0 - 5.0 \cdot r & \text{if } 0.2 \leq r < 0.4, \\ 0.0 & \text{otherwise.} \end{cases}$$

- the pressure $\begin{cases} p = p_0 + 12.5 \cdot r^2 & \text{if } r < 0.2, \\ p = p_0 + 12.5 \cdot r^2 + 4.0 \cdot (1.0 - 5.0 \cdot r - \log(0.2) + \log(r)) & \text{if } 0.2 \leq r < 0.4, \\ p = p_0 - 2.0 + 4.0 \cdot \log(2.0) & \text{otherwise.} \end{cases}$
- the velocity in the x-direction $u_x = -\sin(\phi) \cdot u_\phi$,
- the velocity in the y-direction $u_y = \cos(\phi) \cdot u_\phi$,
- and the density $\rho = 1.0$.

3.4.2 Defining the Mach number

The Mach number is a dimensionless quantity (similarly to the Reynolds number that allows to distinguish between laminar and turbulent flows) that is used to measure the speed of an object moving in a fluid environment, relative to the speed of sound in that fluid. It can be written as follows:

$$\mathrm{M} = \frac{|u|}{a}, \tag{3.4}$$

where u (in m/s) is the velocity of the object and a is the speed of sound (in m/s).

The interest of studying the Gresho vortex for different Mach numbers resides in checking the diffusivity of the numerical scheme. This section serves to check that less diffusive results are obtained for low Mach numbers when it comes to triangular meshes rather than Cartesian meshes.

3.4.3 Plots of the density and the norms of the velocity with respect to the mesh types

In this section, the mesh type is again what varies: a $2,500$-element Cartesian mesh, $2,592$-element structured triangular mesh (from my personal implementation) and $2,528$-element unstructured triangular mesh. The Mach number always equals 0.05 and the numerical flux is the HLL flux. The final time is set to $t_{\text{final}} = 0.1$ second.

Density plots

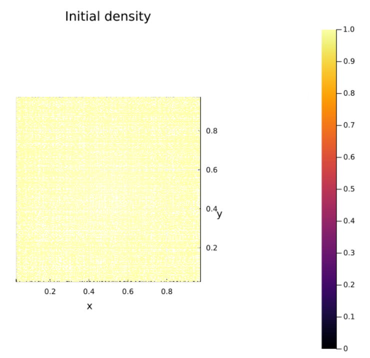

Figure 17: Initial density on a 2, 528-element unstructured triangular mesh using the HLL flux

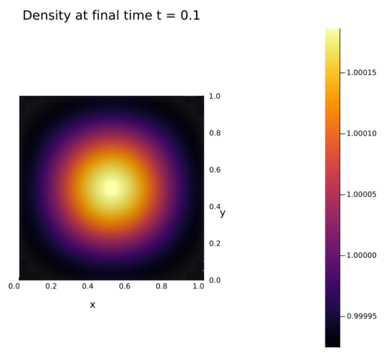

Figure 18: Final density on a 2, 500-element Cartesian mesh using the HLL flux

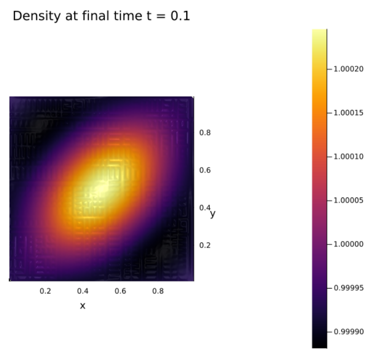

Figure 19: Final density on a 2, 592-element structured triangular mesh (personal implementation) using the HLL flux

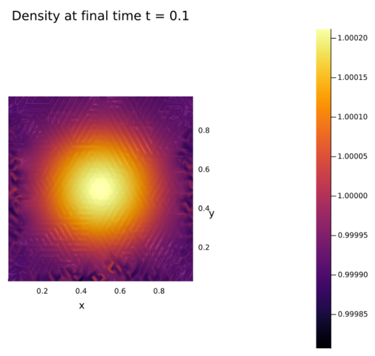

Figure 20: Final density on a 2, 528-element unstructured triangular mesh (GMSH) using the HLL flux

Velocity plots

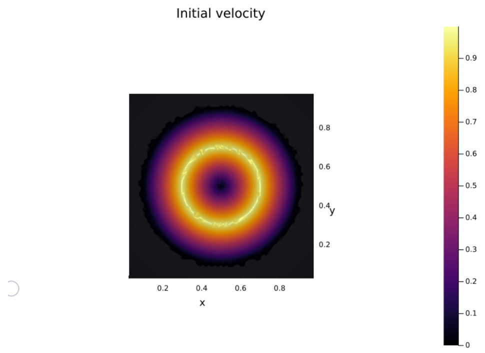

Figure 21: Initial velocity on a 2, 528-element unstructured triangular mesh using the HLL flux

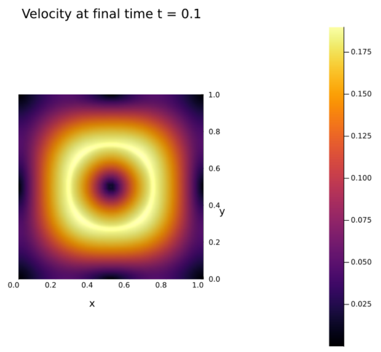

Figure 22: Final velocity on a 2, 500-element Cartesian mesh using the HLL flux

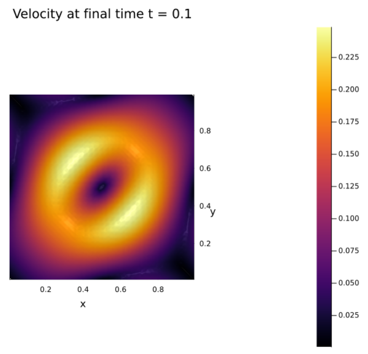

Figure 23: Final velocity on a 2, 592-element structured triangular mesh (personal implementation) using the HLL flux

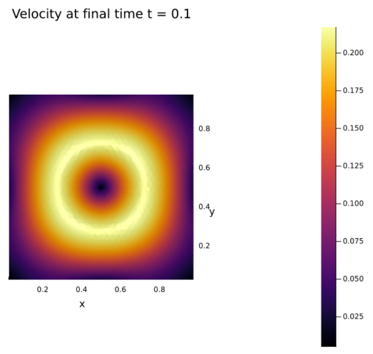

Figure 24: Final velocity on a 2, 528-element unstructured triangular mesh (GMSH) using the HLL flux

Observations and conclusion

All the figures above related to the Gresho vortex test case with a Mach number of 0.05 show that despite being initially constant, the final density deviates from the initial conditions.When it comes for the Cartesian mesh, the extremal values of the final density are ≈ 0.99991 and ≈ 1.00018 whereas for the structured triangular mesh (my implementation) and for the unstructured triangular mesh (GMSH), the extremal values of the final density are ≈ 0.9999 and ≈ 1.00024 then ≈ 0.9998 and ≈ 1.0002 respectively.

Furthermore, such a deviation can also be observed for the final velocity. For instance, the final velocity from the unstructured triangular mesh (2,528 elements) is comprised between 0 and ≈ 0.22, while the initial velocity is comprised between 0 and 1. This is due to the fact that the numerical scheme is diffusive. Let us remind that the Gresho vortex test case is a particular solution of the Euler equations.

For the structured triangular grids, all the plots tend to provide symetric solutions with non-symetric diffusive effect. This property is attributed to the alterning orientation of the triangular elements on the grid.

It is here slightly but insufficiency relevant, regarding asserting that triangular meshes are less diffusive than Cartesian meshes. This is why a further section will be dedicated to the study of this object with the help of the kinetic energy.

3.4.4 Plots of the density and the norms of the velocity with respect to the Mach number

In this section, the Mach number is the varying parameter. The mesh type is a $10,786$-element unstructured triangular mesh (GMSH) and the numerical flux is the Rusanov flux. The final time is set to $t_{\text{final}} = 0.1$ second.

Density plots

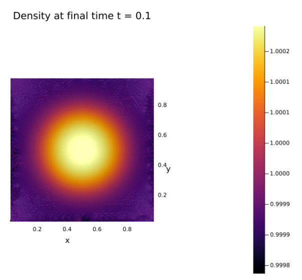

Figure 25: Final density with a Mach number of 0.05 on a $10,756$-element unstructured triangular mesh (GMSH) using the Rusanov flux

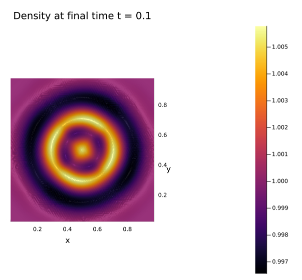

Figure 26: Final density with a Mach number of 0.95 on a $10,756$-element unstructured triangular mesh (GMSH) using the Rusanov flux

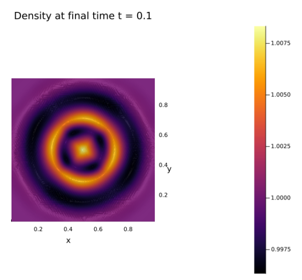

Figure 27: Final density with a Mach number of 1.5 on a $10,756$-element unstructured triangular mesh (GMSH) using the Rusanov flux

Velocity plots

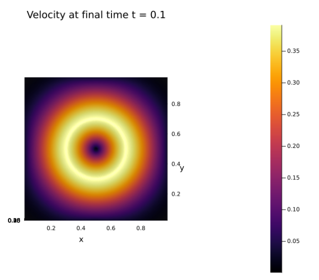

Figure 28: Final velocity with a Mach number of 0.05 on a $10,756$-element unstructured triangular mesh (GMSH) using the Rusanov flux

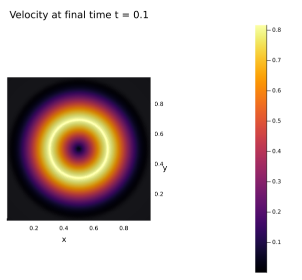

Figure 29: Final velocity with a Mach number of 0.95 on a $10,756$-element unstructured triangular mesh (GMSH) using the Rusanov flux

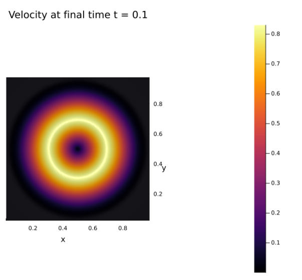

Figure 30: Final velocity with a Mach number of 1.5 on a $10,756$-element unstructured triangular mesh (GMSH) using the Rusanov flux

Observations and conclusion

The figures above show that the final density and the final velocity are more diffusive for lower Mach numbers: in fact the maximum of the final density is ≈ 1.0003 for a Mach number of 0.05, ≈ 1.0075 for a Mach number of 0.95 and ≈ 1.006 for a Mach number of 1.5. In fact, it is the case, because the there are more computations related to the time step Δt (until the final time) to perform for lower Mach numbers. This explained by the fact that Δt is proportionnal to the signal speeds (which tend to be lower for lower Mach numbers because the former are related to the speed of sound).

3.4.5 Plots of the density and the norms of the velocity with respect to the mesh size

In this section, the mesh size is what varies: a 2,592-element structured triangular mesh (from my personal implementation) and 10,082-element structured triangular mesh (from my personal implementation). The Mach number always equals 0.05 and the numerical flux is the HLL flux. The final time is set to $t_{\text{final}} = 0.1$ second.

Density plots

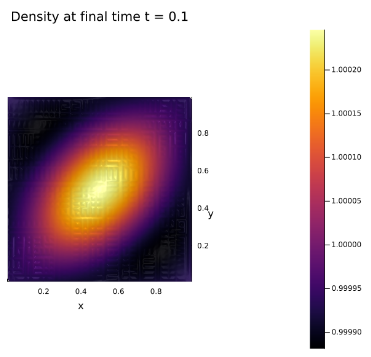

Figure 31: Final density on a 2,528-element structured triangular mesh (GMSH) using the HLL flux

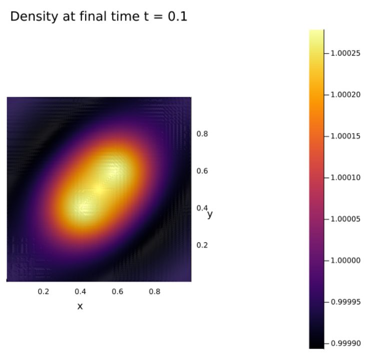

Figure 32: Final density on a 10,082-element structured triangular mesh (GMSH) using the HLL flux

Velocity plots

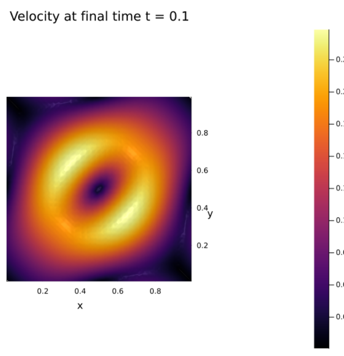

Figure 33: Final velocity on a 2, 528-element structured triangular mesh (GMSH) using the HLL flux

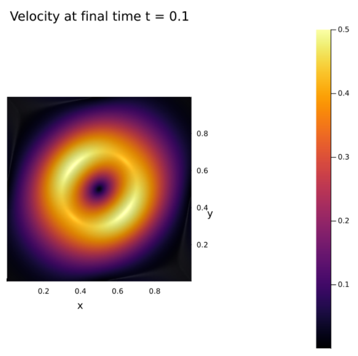

Figure 34: Final velocity on a 10, 082-element structured triangular mesh (GMSH) using the HLL flux

Observations and conclusion

The plots above show that the final density and the final velocity are more diffusive for lower mesh sizes: in fact the maximum of the final velocity is ≈ 0.25 for a 2, 592-element mesh and ≈ 0.5 for a 10, 082-element mesh. This underlines the fact that the numerical scheme is more diffusive for lower mesh sizes. In fact, it is the case, because the there are more computations related to the time step Δt (until the final time) to perform for lower mesh sizes. This explained by the fact that Δt is proportionnal to the signal speeds (which tend to be lower for lower mesh sizes because the former are related to the speed of sound). Again the non-symetric diffusive effect is observed for the structured triangular mesh.

3.4.6 Studying the error of the density of the Gresho vortex test case for different Mach numbers

Studying the error of the density of the Gresho vortex test case for different Mach numbers (0.05, 0.95 and 1.5) is an interesting way to check the accuracy of the numerical scheme. For all mesh types, the error is defined the same way as in the 3.1 section, except that it only encompasses the first component of the state variables vector (therefore, the density).

The final time is again set to be $t_{\text{final}} = 0.1$ second.

Using the initial density as the exact density, the error, for L^1, L^2 and L^∞ norms, should be of order M^2 for all meshes, where M represents the Mach number.

Cartesian mesh

M	$\mathbf{M}^2$	L^1 **error**	L^2 **error**	L^∞ **error**
0.05	0.0025	9.178×10^{-5}	1.077×10^{-4}	0.0002489
0.95	0.9025	0.001396	0.002012	0.006657
1.5	2.25	0.001730	0.002568	0.008530

Table 14: Error of the density for the Gresho vortex test case with a 10, 000-element Cartesian mesh with the Rusanov flux

M	**M^2**	L^1 **error**	L^2 **error**	L^∞ **error**
0.05	0.0025	6.8404×10^{-5}	7.8636×10^{-5}	0.0002720
0.95	0.9025	0.002817	0.003227	0.009799
1.5	2.25	0.004147	0.004483	0.014048

Table 15: Error of the density for the Gresho vortex test case with a $2,500$-element Cartesian mesh with the HLL flux

M	**M^2**	L^1 **error**	L^2 **error**	L^∞ **error**
0.05	0.0025	9.178×10^{-5}	0.0001077	0.0002489
0.95	0.9025	0.001396	0.002012	0.006657
1.5	2.25	0.001730	0.002568	0.008530

Table 16: Error of the density for the Gresho vortex test case with a $10,000$-element Cartesian mesh with the HLL flux

Structured triangular mesh (my implementation)

M	**M^2**	L^1 **error**	L^2 **error**	L^∞ **error**
0.05	0.0025	0.0001	0.0001	0.0003
0.95	0.9025	0.0032	0.0032	0.0135
1.5	2.25	0.0034	0.0034	0.0153

Table 17: Error of the density for the Gresho vortex test case with a $2,592$-element structured triangular mesh with the Rusanov flux

M	**M^2**	L^1 **error**	L^2 **error**	L^∞ **error**
0.05	0.0025	0.000115	0.0001	0.0003
0.95	0.9025	0.0021	0.0021	0.0093
1.5	2.25	0.0022	0.0022	0.0107

Table 18: Error of the density for the Gresho vortex test case with a $10,082$-element structured triangular mesh with the Rusanov flux

M	**M^2**	L^1 **error**	L^2 **error**	L^∞ **error**
0.05	0.0025	6.8404×10^{-5}	7.8636×10^{-5}	0.0001864
0.95	0.9025	0.002293	0.003227	0.009799
1.5	2.25	0.003121	0.004483	0.014048

Table 13: Error of the density for the Gresho vortex test case with a $2,500$-element Cartesian mesh with the Rusanov flux

M	**M**^{2}	L^1 **error**	L^2 **error**	L^∞ **error**
0.05	0.0025	0.0001	0.0001	0.000272
0.95	0.9025	0.002817	0.002817	0.012388
1.5	2.25	0.004147	0.004147	0.018074

Table 19: Error of the density for the Gresho vortex test case with a $2,592$-element structured triangular mesh with the HLL flux

M	**M**^{2}	L^1 **error**	L^2 **error**	L^∞ **error**
0.05	0.0025	0.000113	0.000113	0.000279
0.95	0.9025	0.001741	0.001741	0.008295
1.5	2.25	0.002459	0.002459	0.011436

Table 20: Error of the density for the Gresho vortex test case with a $10,082$-element structured triangular mesh with the HLL flux

Unstructured triangular mesh (GMSH)

M	**M**^{2}	L^1 **error**	L^2 **error**	L^∞ **error**
0.05	0.0025	8.8702×10^{-5}	8.8702×10^{-5}	0.0002045
0.95	0.9025	0.003241	0.003241	0.008251
1.5	2.25	0.003334	0.003334	0.009214

Table 21: Error of the density for the Gresho vortex test case with a $2,528$-element unstructured triangular mesh with the Rusanov flux

M	**M**^{2}	L^1 **error**	L^2 **error**	L^∞ **error**
0.05	0.0025	9.4502×10^{-5}	9.4502×10^{-5}	0.0002406
0.95	0.9025	0.001774	0.001774	0.005733
1.5	2.25	0.001896	0.001896	0.006463

Table 22: Error of the density for the Gresho vortex test case with a $10,756$-element unstructured triangular mesh with the Rusanov flux

M	**M**^{2}	L^1 **error**	L^2 **error**	L^∞ **error**
0.05	0.0025	8.959×10^{-5}	8.959×10^{-5}	0.0002086
0.95	0.9025	0.003218	0.003218	0.01244
1.5	2.25	0.004744	0.004744	0.01993

Table 23: Error of the density for the Gresho vortex test case with a $2,528$-element unstructured triangular mesh with the HLL flux

M	**M^2**	**L^1 error**	**L^2 error**	**L^∞ error**
0.05	0.0025	9.521×10^{-5}	9.521×10^{-5}	0.0002451
0.95	0.9025	0.001691	0.001691	0.007911
1.5	2.25	0.002202	0.002202	0.001228

Table 24: Error of the density for the Gresho vortex test case with a 10,756-element unstructured triangular mesh with the Rusanov flux

As observed in the tables above, the error of the density is of order M^2 for all meshes, where M represents the Mach number. This is a good sign that the numerical scheme is accurate. For example, for a Mach number of 0.05 on a 10,756-element unstructured triangular mesh with the HLL flux, the L^1 error is 9.521×10^{-5} and for a 10,000-element Cartesian mesh with the HLL flux, the L^1 error is 9.178×10^{-5}. The error is of the same order of magnitude for both meshes, which is a good sign that the numerical scheme is accurate. Moreover, for a Mach number of 0.95 on a 10,756-element unstructured triangular mesh with the HLL flux, the L^∞ error is 0.007911 and for a 10,000-element Cartesian mesh with the HLL flux, the L^∞ error is 0.006657. The error is of the same order of magnitude for both meshes, which is a good sign that the numerical scheme is accurate.

3.5 Studying the kinetic energy of the Gresho vortex test case for different Mach numbers

3.5.1 Defining the kinetic energy

The kinetic energy is the energy that an object possesses due to its motion. It is defined as follows:

$$K = \frac{1}{2}\rho(u_x^2 + u_y^2), \tag{3.5}$$

where ρ is the density and u_x and u_y are respectively the x and y components of the velocity.

3.5.2 Motivations for studying the kinetic energy

The kinetic energy is a good way to check more quantitatively whether the numerical scheme is diffusive or not. In fact, the kinetic energy should be conserved over time for the Gresho vortex test case. The more towards zero the kinetic energy evolves over time, the more diffusive the numerical scheme is.

This section is set to show plots of the kinetic energy of the Gresho vortex test case for different Mach numbers (0.05, 0.95 and 1.5) and different mesh sizes, as used above.

The plots were also made with respect to the mesh type (Cartesian, my own structured triangular mesh implementation, the structured and the unstructured triangular mesh implementation using files generated GMSH), the final times ($t_{\text{final}} = 0.1$, $t_{\text{final}} = 0.3$, $t_{\text{final}} = 0.5$ and $t_{\text{final}} = 0.7$ and the numerical flux used, whether it be Rusanov or HLL).

Furthermore, it is also going to embedd the number of iterations needed to reach the final time, for each mesh type, each numerical flux and each Mach number.

3.5.3 Plotting the dissipation of the kinetic energy over time for the Gresho vortex test case for different Mach numbers

In this section, the numerical flux is what varies: either the Rusanov flux or the HLL flux are used to provide the results below. The mesh utilized are a 2,500-element Cartesian mesh, a 2,592-

element structured triangular mesh (personal implementation) and a $2,528$-element unstructured triangular mesh (GMSH-generated). The Mach number is set to be 0.05.

The plots below show the evolution of the kinetic energy ratio defined by $\kappa = \frac{\max K}{\max K_0}$ over time. Therein K is the kinetic energy at a given time and K_0 is the initial kinetic energy.

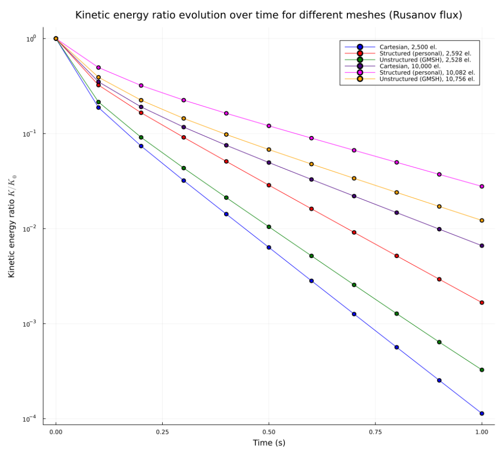

Figure 35: Kinetic energy ratio evolution over time for different meshes using the Rusanov flux

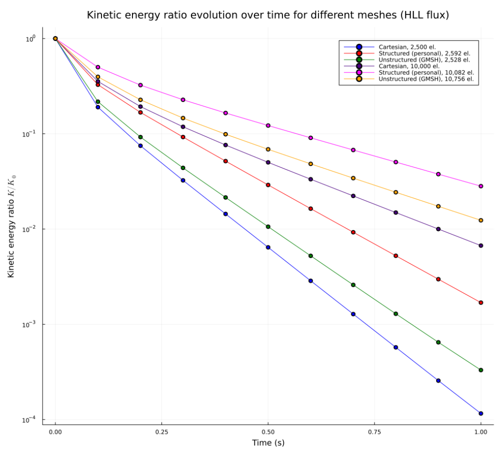

Figure 36: Kinetic energy ratio evolution over time for different meshes using the HLL flux

Observations and interpretations As seen in the figures below for both numerical fluxes, the kinetic energy ratio decreases towards zero over time for all mesh types. This is another sign that the numerical scheme is diffusive. The more towards zero the kinetic energy ratio decreases, the more diffusive the numerical scheme is.

Additionally, the kinetic energy ratio decreases faster for the Cartesian mesh than for the structured and unstructured triangular meshes. It underlines, one more time, the fact that the numerical scheme is more diffusive for the Cartesian mesh than for the structured and unstructured triangular meshes. It also decreases faster for mesh of resolution approaching $\approx 50 \times 50$ elements than for mesh of resolution approaching $\approx 100 \times 100$ elements, since the respective curbs are steeper for the former than for the latter. It provides another signs that the numerical scheme works (less diffusive effects for higher mesh-resolutions).

To get more quantitative results regarding the discrepancies in terms of diffusive effects among different mesh types, the number of iterations needed to reach the final time will be computed in the section below.

3.5.4 Tables containing the number of iterations needed to reach the final time

The interest of this section serves to quantify the accumulation of computational diffusive effects over time.

This part provides a solid quantitative argument that validates the qualitative argument provided in the 3.5.3 section (the fact that the scheme is more diffusive for the Cartesian mesh than for the structured and unstructured triangular meshes).

The final time is set to be $t_{\text{final}} = 0.1$ second. The Mach number are set to be 0.05, 0.95 or 1.5. The numerical flux is either the Rusanov flux or the HLL flux. The mesh sizes are the following: $2,500$ elements for the Cartesian mesh, $2,592$ elements for the structured triangular mesh (my personal implementation) and 2,528 elements for the unstructured triangular mesh (GMSH-generated), 10,000 elements for the Cartesian mesh, 10,082 elements for the structured triangular mesh (my personal implementation) and 10,756 elements for the unstructured triangular mesh (GMSH-generated).

Mesh Type	Mach Number	Iterations (Rusanov)	Iterations (HLL)
Cartesian	0.05	987	987
	0.95	97	100
	1.5	85	89
Structured (pers.)	0.05	2,801	2,801
	0.95	286	294
	1.5	253	262
Unstructured	0.05	5,884	5,884
	0.95	582	599
	1.5	513	533

Table 25: Table of results for all mesh approaching a resolution of $\approx 50 \times 50$ elements

Mesh Type	Mach Number	Iterations (Rusanov)	Iterations (HLL)
Cartesian	0.05	2,004	2,005
	0.95	206	211
	1.5	182	188
Structured (pers.)	0.05	5,694	5,694
	0.95	601	614
	1.5	532	546
Unstructured	0.05	12,604	12,606
	0.95	1,306	1,336
	1.5	1,153	1,185

Table 26: Table of results for all mesh approaching a resolution of $\approx 100 \times 100$ elements

Observations and interpretations As observed, there is a higher number of iterations for the triangular meshes than for the Cartesian mesh. For example, using a Mach number of 0.05, the number of iterations needed to reach the final time is 987 for a 2,500-element Cartesian mesh, 2,801 for a 2,592-element structured triangular mesh and 5,884 for a 2,528-element unstructured triangular mesh. Even though the number of iterations is higher for triangular meshes, the previous section showed that the kinetic energy ratio decreases faster for the Cartesian mesh than for the structured and unstructured triangular meshes. It means that the numerical scheme is more intensively diffusive for the Cartesian mesh than for the structured and unstructured triangular meshes.

3.6 Benchmarking the performance of the implementation for all the different mesh types

3.6.1 Why benchmarking the performance?

The benchmarking of the performance of the implementation for all the different mesh types will focus on both computing the CPU time needed to reach the final time and the **number of**

iterations in terms of flux computations needed to reach the final time.

Benchmarching the performance complements well the computation of the errors and the convergence rates. The former can be appreciated as a way to test the rapidity of the implementation, while the latter can be employed to check the accuracy of the implementation. This participates therefore more as a way to check whether the implementation or the numerical scheme are efficient or not.

In that case, the test case used for that purpose is the transport problem test case with periodic boundary conditions at final time $t_{\text{final}} = 1.0$ second. The results are provided for the Cartesian meshes, the structured triangular meshes and the unstructured triangular meshes. The parameters are the numerical flux used (Rusanov or HLL) and the mesh sizes.

3.6.2 Results for the Cartesian meshes

I personally decided to keep four significant digits for the CPU time and for the number of iterations.

Mesh Size	**CPU time (in s)**	**Number of iterations**
32×32	0.9662	136
64×64	8.049	276
128×128	64.87	555
256×256	525.2	1,114

Table 27: Table of results for Cartesian meshes using the Rusanov flux

Mesh Size	**CPU time (in s)**	**Number of iterations**
32×32	1.118	136
64×64	8.419	276
128×128	72.85	555
256×256	582.0	1,114

Table 28: Table of results for Cartesian meshes using the HLL flux

3.6.3 Results for structured triangular meshes (my personal implementation)

Mesh Size	**CPU time (in s)**	**Number of iterations**
32×32	3.219	272
64×64	25.59	556
128×128	184.5	1,112
256×256	1,420	2,236

Table 29: Table of results for structured triangular meshes (my personal implementation) using the Rusanov flux

Mesh Size	**CPU time (in s)**	**Number of iterations**
32 × 32	2.885	272
64 × 64	23.62	556
128 × 128	188.4	1,112
256 × 256	1,521	2,236

Table 30: Table of results for structured triangular meshes (my personal implementation) using the HLL flux

3.6.4 Results for structured triangular meshes (GMSH-generated)

Mesh Size	**CPU time (in s)**	**Number of iterations**
32 × 32	9.435	985
64 × 64	81.09	2,034
128 × 128	656.5	4,031
256 × 256	6,408	8,065

Table 31: Table of results for structured triangular meshes (GMSH-generated) using the Rusanov flux

Mesh Size	**CPU time (in s)**	**Number of iterations**
32 × 32	9.435	1,076
64 × 64	81.09	2,289
128 × 128	656.5	4,748
256 × 256	6,408	9,883

Table 32: Table of results for structured triangular meshes (GMSH-generated) using the HLL flux

3.6.5 Results for unstructured triangular meshes (GMSH-generated)

Mesh Size	**CPU time (in s)**	**Number of iterations**
32 × 32	8.499	964
64 × 64	82.14	2,185
128 × 128	639.7	3,854
256 × 256	5,214	7,875

Table 33: Table of results for unstructured triangular meshes (GMSH-generated) using the Rusanov flux

Mesh Size	**CPU time (in s)**	**Number of iterations**
32 × 32	10.17	964
64 × 64	97.94	2,185
128 × 128	677.1	3,854
256 × 256	5,646	7,875

Table 34: Table of results for unstructured triangular meshes (GMSH-generated) using the HLL flux

3.6.6 Comparing the results for the different meshes with respect to the numerical fluxes

In this section, I will compare the results for the different meshes with respect to the numerical fluxes. There will be one plot for the CPU time for all meshes using the Rusanov flux and one plot for the CPU time for all meshes using the HLL flux. The same will be done for the number of iterations.

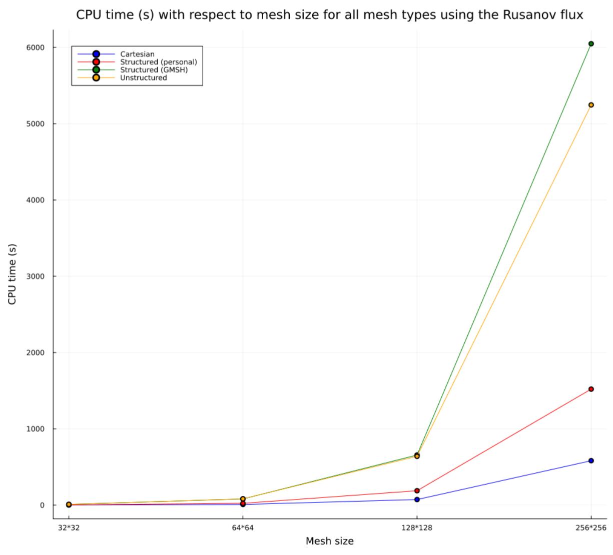

Figure 37: CPU time with respect to the mesh size (total number of elements) for all meshes using the Rusanov flux

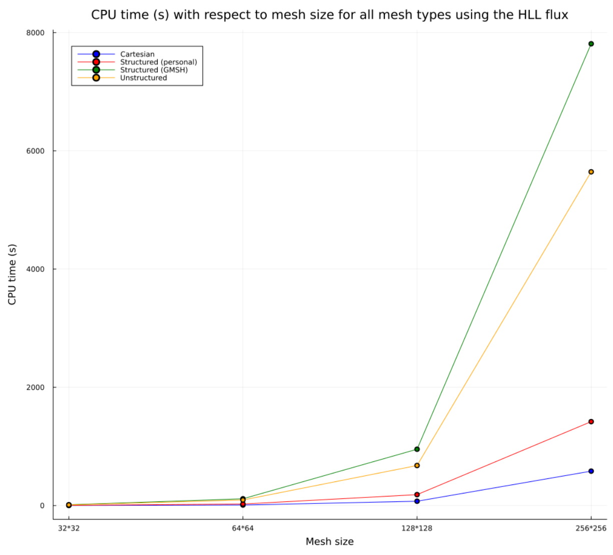

Figure 38: CPU time with respect to the mesh size (total number of elements) for all meshes using the HLL flux

Observations and interpretations As remarked above, for all mesh types, the tables for the CPU time with respect to the mesh size (total number of elements) for my personal unstructured triangular meshes implementation using both numerical fluxes, the variation rate is again approximately 2. This means that the CPU time is proportional to approximately the square of the number of elements.

Still, the variation rate of the number of iterations with respect to the mesh size (total number of elements), relying on the tables for Cartesian meshes using both numerical fluxes, is again approximately 0.5. This means that the number of iterations is proportional to the square root of the number of elements.

However, the variation rate in the table containing the number of iterations with respect to the mesh size (total number of elements) for Cartesian meshes using both numerical fluxes is again approximately $\frac{2}{3}$. This means that the number of iterations is proportional to the square root of the number of elements.

When it comes to Cartesian meshes, let us remember that for each time step, inside my code, there are $N_x(N_y - 1) + N_y(N_x - 1)$ fluxes to compute, where N_x and N_y are the number of cells in the x-and-y-directions respectively. In the case of this benchmarking, the number of cells in the x-and-y-directions are equal.

In the case of this benchmarking for triangular meshes, applying the periodic boundary conditions, there are $3N$ fluxes to compute for each time step, where N is the total number of elements. In fact, each cells possesses three neighbors, relying on the periodic boundary conditions. Thus, there are in fact three interfaces per cell.

Finally, for all meshes, the complexity of the CPU time is $\mathcal{O}(N^2)$. For Cartesian mesh, the complexity of the **number of iterations in terms of flux computations** is $\mathcal{O}(\sqrt{N}) \times \mathcal{O}(N) = \mathcal{O}(N^{3/2})$. Then, for triangular meshes, the complexity of the **number of iterations in terms of flux computations** is $\mathcal{O}(N^{\frac{2}{3}}) \times \mathcal{O}(N) = \mathcal{O}(N^{5/3})$. The initial complexity of the number of iterations in terms of the number of time steps is, for all mesh types, $\mathcal{O}(\sqrt{N})$.

For the figures related to the CPU times for all mesh, the results are as expected: the CPU time increases with the number of elements. The CPU time is higher for the unstructured triangular meshes and the GMSH-generated structured triangular meshes than for the Cartesian meshes and the structured triangular meshes from my personal implementation. For the Cartesian meshes, the CPU peaks at 525.2 seconds for the Rusanov flux and at 582.0 seconds for the HLL flux; for the structured triangular meshes from my personal implementation, the CPU peaks at $2,236$ seconds for both fluxes; for the GMSH-generated structured triangular meshes, the CPU peaks at $8,065$ seconds for the Rusanov flux and at $9,883$ seconds for the HLL flux; for the unstructured triangular meshes, the CPU peaks at $7,875$ seconds for both fluxes. This is due to the fact that the GMSH-generated structured triangular meshes and the unstructured triangular meshes are more complex than the Cartesian meshes and the structured triangular meshes from my personal implementation.

4 Summary and outlook

My internship has finally come to an end. I was able to implement a working finite volume method numerical scheme in Julia for simulations related to the two-dimensional Euler equations with Cartesian, structured triangular and unstructured triangular meshes. My scheme fits the theoretical expectations with aspects related the HLL and Rusanov fluxes and the convergence orders approaching 1.

I was also able to find out that my numerical scheme is less diffusive for triangular meshes than for Cartesian meshes, when it comes to studying the final solution of the Gresho vortex with low Mach numbers.

Moreover, this experience was for me a way to deepen my knowledge in aspects related to numerical schemes and fluxes, and to sharpen my overall skills in Julia, more especially in debugging.

This experience was enriching and I feel grateful to have been able to work on this project. I am finally comforted in my choice of pursuing a career in research.

References

[1] Euler, L., *Principes généraux du mouvement des fluides*, 1757.

[2] "Inria & son écosystème". Inria [visited on August 8th 2023]. Available.

[3] Toro, E. F., *Riemann solvers and numerical methods for fluid dynamics : A practical introduction*, 2009.

[4] Leveque, R. J., *Numerical Methods for Conservation Laws*, 2000.

[5] Dansac, V. M., "Finite volume methods", *Finite volume schemes for the Euler equations on unstructured meshes*, PhD Thesis, University of Nantes, 2016.