MASTERARBEIT

zur Erlangung des akademischen Grades
Master of Science (Mathematik)
an der Fakultät für Mathematik und Informatik
der Julius-Maximilians-Universität Würzburg

Adapting an invariant domain preserving numerical scheme for the Euler equations to low Mach numbers

Marius Volpert

18.06.2024

Zusammenfassung

Die vorliegende Arbeit befasst sich mit der Modellierung von Gasströmungen und deren numerische approximative Lösung. Konkret betrachtet werden die Eulergleichungen, welche Gase makroskopisch durch die Beschreibung der Volumendichten der im Raum verteilten Erhaltungsgrößen Masse, Impuls und Energie behandeln. Mathematisch erhält man ein System partieller Differentialgleichungen. Da Erhaltungsgrößen modelliert werden, bezeichnet man diese partiellen Differentialgleichungen als Erhaltungsgleichungen (engl. conservation law). Bei den Eulergleichungen handelt es sich um den Spezialfall der Navier-Stokes Gleichungen ohne Viskosität (innere Reibung) und Wärmeleitung.
Von besonderem Interesse ist der Fall von Strömungen für welche das Gas sich relativ zur Schallgeschwindigkeit langsam bewegt. Das Verhältnis von Strömungs- zu Schallgeschwindigkeit wird Mach-Zahl genannt. Wir untersuchen also den sogenannten "Low Mach" Fall. Hierfür werden die Eulergleichungen zunächst mit einer Referenz-Machzahl reskaliert.

Grundlage der numerischen Lösung bietet ein Verfahren basierend auf einer räumlichen Finite Elemente Diskretisierung der Gleichungen und einer expliziten Zeitevolution. Hierfür wird das Raumgebiet beispielsweise in viereckige Zellen aufgeteilt und die numerische Lösung an den Eckpunkten gespeichert.
Eine wesentliche Eigenschaft numerischer Methoden ist deren Stabilität, welche notwendig für die Konvergenz ist. Zudem sind unphysikalische Oszillationen unerwünscht. Das betrachtete Verfahren erreicht Stabilität durch die sogenannte invariant-domain preservation Eigenschaft durch die Einführung einer geeigneten numerischen Graph-Viskosität. Diese wird in unserem Fall beispielsweise sicherstellen, dass die Massendichte nicht negativ werden kann, sofern die Anfangsdaten des Problems dies erfüllen. Das zunächst vorgestellte Verfahren ist definiert durch eine explizite Aufdatierungsformel, was zu dem Problem einer sehr restriktiven CFL-Bedingung führt. Das bedeutet, dass die Zeitschritte für kleine Machzahlen ebenfalls sehr klein gewählt werden müssen. Die damit einhergehende Ineffizienz wird durch eine Adaption des Verfahrens behoben.

Die Adaption verwendet das IMEX Konzept, bei welchem die zugrundeliegende Gleichung in einen expliziten und einen impliziten Teil gesplittet werden. Der explizite Teil wird mit dem ursprünglichen Verfahren gelöst, für den impliziten Teil wird eine implizite Formulierung derselben verwendet, welche dann durch eine Fixpunkt-Iteration approximativ gelöst wird.
Für die Formulierung des expliziten Teils muss zudem noch ein sogenanntes Riemann-Problem zur Bestimmung der numerischen Parameter gelöst werden.

In einem eigenen Kapitel werden einige Eigenschaften der so definierten Methode untersucht. Unter anderem wird auf die Konsistenz eingegangen, welche aussagt, dass das Verfahren zumindest lokal eine gute Approximation an die tatsächlichen Gleichungen darstellt.

Für die Implementierung der Methode wurde ein auf dem Finite Elemente Framework deal.II basierendes Programm erweitert und angepasst. Dieser C++ Code ist im Anhang zu finden.

Schließlich werden noch diverse numerische Tests durchgeführt, welche das Verhalten des

Verfahrens für bestimmte ausgewählte Probleme demonstrieren.

Danksagung

Ich danke Professor Christian Klingenberg und Matthias Maier für die Begleitung meiner Masterarbeit bezüglich Themenfindung, Unterstützung mit Literatur sowie Feedback und Vorschlägen bei Problemen, die sich gestellt haben.
Desweiteren möchte ich meinen Eltern für ihre Unterstützung meines Studiums danken.

Contents

List of Figures

Chapter 1

Introduction

1.1 Euler equations

The aim of this section is to derive a model for the flow of gasses. Another explanation is given in [5]. A gas consists of many particles (molecules) whose interaction can be understood as elastic collisions in a simplified way. We restrict ourselves to the case of sufficiently dense gasses, for which certain macroscopic thermodynamic state variables and corresponding state equations are reasonable to describe the gas locally.
These state variables are the mass-density ρ, the stream velocity u and the pressure p. Since they represent a locally averaged state, they are functions depending on a time-variable $t \in \mathbb{R}$ and space-variable $x \in \mathbb{R}^3$.

To model the gas dynamics one considers the conserved quantities mass, momentum and energy. We model their spatial distribution as a volume density:

1. The mass density is again ρ
2. The momentum density is given by ρu
3. The energy density is denoted by $\mathcal{E}$

The energy is made up of two parts: The macroscopic kinetic energy with density $\frac{\rho}{2}u^2$ (depending on the frame of reference) and internal energy due to the temperature of the gas. The internal energy is known to be $\frac{f}{2}Nk_BT$, where k_B is the Boltzmann-constant, N the particle number, T the gas temperature and f is the number of degrees of freedom of movement per particle (see [6]). For example, $f = 5$ for nitrogen, oxygen at standard conditions. Using the ideal gas equation $pV = Nk_BT$ for a small volume V which we assume contains gas in a constant state, we find

$$\frac{\frac{f}{2}Nk_BT}{V} = \frac{f}{2}p$$

as the volume density.
This **equation of state** can also be written with the *isentropic exponent* $\gamma = \frac{f+2}{f}$ ($\gamma \approx 1.4$ for air):

$$\mathcal{E} = \mathcal{E}_{\text{int}} + \mathcal{E}_{\text{kin}} = \frac{p}{\gamma - 1} + \frac{\rho}{2}u^2 \tag{1.1}$$

1.1.1 The stress tensor

The interaction of gas particles will also be understood macroscopically with the help of continuum mechanics. Here, for any surface S with a normal vector field ν, a mechanical stress tensor σ models the force the gas on the side of S ν is pointing away from "feels" from the other side of S. The function σ defines for every point in space (and time) a linear map $\mathbb{R}^3 \to \mathbb{R}^3$ and the force is given by

$$F = \int_S \sigma\nu \, d^2x \tag{1.2}$$

In gas dynamics, the stress tensor depends on the state variables. It consists of the pressure term $-p\mathbb{I}$, modeling the isotropic forces the gas particles exert due to their random microscopic movement, and a viscous term modeling a kind of internal friction, which is considered by the Navier-Stokes equations of which the Euler equations are a special case where the viscosity is neglected (i.e. set to 0).
So we will simply use

$$\sigma = -p\mathbb{I} \tag{1.3}$$

(Note that $\mathbb{I}$ refers to the identity matrix)
For a more in depth explanation of the stress tensor see [17].

Now the derivation of the actual Euler equations will be sketched. This amounts to finding a balance law for mass, momentum and energy. To this end, let $\Omega \subset \mathbb{R}^3$ be an open, bounded domain with an outer-facing normal vector field ν.

1.1.2 Mass conservation

The mass contained in Ω is of course $\int_\Omega \rho \, d^3x$. It can only change in time by gas flowing in/out of Ω. For an infinitesimal area element d^2x and time dt, the volume $u \cdot \nu d^2x dt$ of gas flows out of Ω. This corresponds to the mass outflow $\rho u \cdot \nu d^2x dt$. This gives:

$$\frac{d}{dt} \int_\Omega \rho \, d^3x = -\int_{\partial\Omega} \rho u \cdot \nu \, d^2x \tag{1.4}$$

For sufficiently smooth solutions and using the Gauß integral theorem:

$$\int_\Omega \frac{\partial}{\partial t}\rho \, d^3x + \int_\Omega div(\rho u) \, d^3x = 0 \tag{1.5}$$

This has to hold for any Ω, so one further gets the continuum equation:

$$\frac{\partial}{\partial t}\rho + div(\rho u) = 0 \tag{1.6}$$

1.1.3 Momentum conservation

Just like mass, momentum is also transported by the gas. However, the internal interaction as modeled by the stress tensor must now be taken into account as well. It adds the force given in (1.2), since force is the time derivative of momentum (for a given parcel of gas).
The transport contribution reads like in (1.4) if we exchange the mass- with the momentum density.
In total:

$$\frac{d}{dt} \int_\Omega \rho u \, d^3x = -\int_{\partial\Omega} (\rho u(u \cdot \nu) - \sigma\nu) \, d^2x = -\int_{\partial\Omega} (\rho u u^T - \sigma)\nu \, d^2x \tag{1.7}$$

Again using Gauß integral theorem:

$$\frac{d}{dt}\int_\Omega \rho u \, d^3x + \int_\Omega div(\rho u u^T - \sigma)\, d^3x = 0 \tag{1.8}$$

Using 1.3 for smooth solutions we obtain:

$$\frac{\partial}{\partial t}(\rho u) + div(\rho u u^T + p\mathbb{I}) = 0 \tag{1.9}$$

This gives one equation per momentum component:

$$\frac{\partial}{\partial t}(\rho u_i) + div(\rho u_i u) + \frac{\partial}{\partial x_i} p = 0 \quad \text{for} \quad i = 1, 2, 3 \tag{1.10}$$

To see why the transport and stress contributions simply add up, we can derive the equation more stringently as in [5], pp. 4-11. Let $\phi : \mathbb{R}^3 \times \mathbb{R} \times \mathbb{R} \ni (x,t,s) \mapsto \phi(x,t,s) \in \mathbb{R}^3$ denote the trajectory of a particle moving with the fluid starting at position x at time t. I.e. at time s the particle is at position $\phi(x,t,s)$. That means $\phi(x,t,t) = x$ and $\frac{\partial}{\partial s}\phi(x,t,s) = u(\phi(x,t,s),s)$. Define $\phi_{t,s} : x \mapsto \phi_{t,s}(x) = \phi(x,t,s)$ and $\Omega_{t,s} := \phi_{t,s}(\Omega)$ the volume Ω after being advected with the flow. With a given time t, the effect of the stress forces is then simply given by

$$\frac{d}{ds}\int_{\Omega_{t,s}} (\rho u)(x,s)\, d^3x = \int_{\partial\Omega_{t,s}} \sigma\nu \, d^2x = -\int_{\partial\Omega_{t,s}} p\nu\, d^2x \tag{1.11}$$

A change of variables gives

$$\frac{d}{ds}\int_{\Omega_{t,s}} (\rho u)(x,s)\, d^3x = \frac{d}{ds}\int_\Omega (\rho u)(\phi_{t,s}(x),s)|\det(D\phi_{t,s}(x))|\, d^3x$$

The Jacobian should not become singular (the gas is not infinitely compressed) and we assume it to be continuous in time so we can use $\det(D\phi_{t,s}(x)) > 0$ to obtain

$$\frac{d}{ds}\int_\Omega (\rho u)(\phi_{t,s}(x),s)|\det(D\phi_{t,s}(x))|\, d^3x$$
$$= \int_\Omega (\partial_t(\rho u) + u\cdot\nabla_x(\rho u))(\phi_{t,s}(x),s)\det(D\phi_{t,s}(x))\, d^3x + \int_\Omega (\rho u)(\phi_{t,s}(x),s)\frac{d}{ds}\det(D\phi_{t,s}(x))\, d^3x$$

A lengthy but straightforward computation reveals

$$\frac{d}{ds}\det(D\phi_{t,s}(x)) = (div(u))(\phi_{t,s}(x),s)\det(D\phi_{t,s}(x))$$

Using this fact and again using change of variables we get

$$\begin{aligned}
&\frac{d}{ds}\int_{\Omega_{t,s}} (\rho u)(x,s)\, d^3x \\
&= \int_\Omega (\partial_t(\rho u) + u\cdot\nabla_x(\rho u) + \rho u\, div(u))(\phi_{t,s}(x),s)\det(D\phi_{t,s}(x))\, d^3x \\
&= \int_{\Omega_{t,s}} (\partial_t(\rho u) + u\cdot\nabla_x(\rho u) + \rho u\, div(u))(x,s)\, d^3x \\
&= \int_{\Omega_{t,s}} (\partial_t(\rho u) + div(\rho u u^T))(x,s)\, d^3x
\end{aligned}$$

Setting $s = t$ in the above equation and using the Gauß integral theorem gives

$$\frac{d}{ds}\int_{\Omega_{t,s}} (\rho u)(x,s)\bigg|_{s=t} = \int_{\Omega} \partial_t(\rho u)(x,s)\, d^3x + \int_{\partial\Omega} \rho u u^T \nu(x,s)\, d^3x$$

Combining with (1.11) again gives the integral form of momentum conservation:

$$\frac{d}{dt}\int_{\Omega} (\rho u)(x,t)\, d^3x + \int_{\partial\Omega} (\rho u u^T + p\mathbb{I})\nu(x,t)\, d^2x = 0$$

1.1.4 Energy conservation

Again, energy can be transported with the gas flow. Because the time derivative of energy is force times velocity, the contribution due to the pressure interaction is now given by $\int_{\partial\Omega} u \cdot (\sigma\nu)\, d^2x$. One now gets:

$$\frac{d}{dt}\int_{\Omega} \mathcal{E}\, d^3x = -\int_{\partial\Omega} (\mathcal{E}u^T\nu + pu^T\nu)\, d^3x \tag{1.12}$$

With Gauß ...:

$$\frac{d}{dt}\int_{\Omega} \mathcal{E}\, d^3x + \int_{\Omega} div((\mathcal{E}+p)u^T)\, d^3x = 0 \tag{1.13}$$

And finally:

$$\frac{\partial}{\partial t}\mathcal{E} + div((\mathcal{E}+p)u^T) = 0 \tag{1.14}$$

All in all, (1.6), (1.9) and (1.14) give 5 (scalar) equations for 5 (scalar) variables. The equation of state (1.1) explains the pressure terms in (1.9) and (1.14) using only the conservative quantities ρ, ρu and $\mathcal{E}$.

1.1.5 Boundary conditions

Since the Euler equations are time evolution partial differential equations, concrete problems typically define initial conditions and impose certain boundary conditions. Besides Dirichlet boundary conditions and periodic boundaries, free-slip conditions are often used to model (non-moving) walls. Here it must hold that $u \cdot \nu = 0$ on the boundary with ν the boundary normal vector field.

1.2 Scaled Euler equations

As we are interested in problems where the stream velocity tends to be very small (compared to the speed of sound, see section 3.1), it makes sense to rescale the Euler equations by a parameter called the reference Mach number $M_{\text{ref}} > 0$ (typically chosen $\ll 1$), which is understood as the ratio between a reference velocity $||u_{\text{ref}}||$ and a reference speed of sound c_{ref}: $M_{\text{ref}} = ||u_{\text{ref}}||/c_{\text{ref}}$. To this end, define

$$\hat{u} = \frac{u}{M_{\text{ref}}} \;,\; \hat{t} = M_{\text{ref}} t$$

The **equation of state** can now be written

$$\mathcal{E} = \frac{p}{\gamma - 1} + \frac{{M_{\text{ref}}}^2}{2}\rho\hat{u}^2 \tag{1.15}$$

The **mass equation** reads

$$\partial_{\hat{t}}\rho + div(\rho\hat{u}) = \frac{1}{M_{\text{ref}}}\left(\partial_t\rho + div(\rho u)\right) = 0 \tag{1.16}$$

The **momentum equation** reads

$$\partial_{\hat{t}}(\rho\hat{u}) + div(\rho\hat{u}\hat{u}^T + \frac{p}{{M_{\text{ref}}}^2}\mathbb{I}) = \frac{1}{{M_{\text{ref}}}^2}\left(\partial_t(\rho u) + div(\rho u u^T + p\mathbb{I})\right) = 0 \tag{1.17}$$

The **energy equation** becomes

$$\partial_{\hat{t}}\mathcal{E} + div((\mathcal{E}+p)\hat{u}^T) = \frac{1}{M_{\text{ref}}}\left(\partial_t\mathcal{E} + div((\mathcal{E}+p)u^T)\right) = 0 \tag{1.18}$$

From now on, we shall simply write u and t in place of $\hat{u}$ and $\hat{t}$ respectively, and always refer to those rescaled Euler equations unless stated otherwise.

The equations 1.16, 1.17 and 1.18 can be put in the more compact form of a *conservation law* by introducing a vector-valued function of conserved quantities

$$q := \begin{pmatrix} \rho \\ \rho u \\ \mathcal{E} \end{pmatrix}$$

and the *flux function*

$$f(q) := \begin{pmatrix} \rho u^T \\ \rho u u^T + \frac{p}{{M_{\text{ref}}}^2}\mathbb{I} \\ (\mathcal{E}+p)u^T \end{pmatrix} \tag{1.19}$$

The scaled Euler equations are then given by

$$\partial_t q + div(f(q)) = 0 \tag{1.20}$$

1.2.1 Asymptotic limit

Since we are interested in low Mach number flow, a natural question is what happens in the limit $M_{\text{ref}} \searrow 0$. To answer it, we will formally investigate this asymptotic limit. The analysis is similar to the brief one in [16] Suppose we have a family of sufficiently smooth solutions parameterized by M_{ref} in some interval $[0,\epsilon]$ (for simplicity don't include the parameter in the notation) and expanded as

$$0 < \rho(x,t) = \sum_{i=0}^{\infty} {M_{\text{ref}}}^i \rho^{(i)}(x,t)\ ,\ u(x,t) = \sum_{i=0}^{\infty} {M_{\text{ref}}}^i u^{(i)}(x,t)\ ,\ 0 < p(x,t) = \sum_{i=0}^{\infty} {M_{\text{ref}}}^i p^{(i)}(x,t)$$

Assume that all functions and derivatives are sufficiently uniformly bounded such that we can pull derivatives into the sum etc... The momentum equation then reads

$$\begin{aligned} &\partial_t(\rho u) + div(\rho u u^T + \frac{p}{{M_{\text{ref}}}^2}\mathbb{I}) \\ &= \partial_t(\rho^{(0)}u^{(0)}) + \mathcal{O}(M_{\text{ref}}) + div\left(\rho^{(0)}u^{(0)}(u^{(0)})^T + \frac{p^{(0)}}{{M_{\text{ref}}}^2}\mathbb{I} + \frac{p^{(1)}}{M_{\text{ref}}}\mathbb{I} + p^{(2)}\mathbb{I}\right) + \mathcal{O}(M_{\text{ref}}) = 0 \end{aligned} \tag{1.21}$$

From this we get

$$\nabla_x p^{(0)} = \lim_{M_{\text{ref}}\to 0} {M_{\text{ref}}}^2 \left(-\underbrace{\partial_t(\rho^{(0)}u^{(0)})}_{\mathcal{O}(1)} - \underbrace{div\left(\rho^{(0)}u^{(0)}(u^{(0)})^T + \frac{p^{(1)}}{M_{\text{ref}}}\mathbb{I} + p^{(2)}\mathbb{I}\right)}_{\mathcal{O}({M_{\text{ref}}}^{-1})} + \mathcal{O}(M_{\text{ref}}) \right) = 0$$

Inserting $\nabla_x p^{(0)} = 0$ again into (1.21) gives

$$\nabla_x p^{(1)} = 0$$

Which means we are finally left with

$$\lim_{M_{\text{ref}}\to 0} \partial_t(\rho^{(0)}u^{(0)}) + div(\rho^{(0)}u^{(0)}(u^{(0)})^T + p^{(2)}\mathbb{I}) + \mathcal{O}(M_{\text{ref}}) = 0$$

$$\partial_t(\rho^{(0)}u^{(0)}) + div(\rho^{(0)}u^{(0)}(u^{(0)})^T) + \nabla_x p^{(2)} = 0 \tag{1.22}$$

as the limiting equation. For the equation of state we find

$$\mathcal{E} = \underbrace{\frac{p^{(0)}}{\gamma-1}}_{=:\mathcal{E}^{(0)}} + M_{\text{ref}}\underbrace{\frac{p^{(1)}}{\gamma-1}}_{=:\mathcal{E}^{(1)}} + {M_{\text{ref}}}^2 \left(\underbrace{\frac{p^{(2)}}{\gamma-1} + \frac{1}{2}\rho^{(0)}||(u^{(0)})||^2}_{=:\mathcal{E}^{(2)}} \right) + \mathcal{O}({M_{\text{ref}}}^3)$$

to write the energy equation as

$$\partial_t\mathcal{E} + div((\mathcal{E}+p)u) = \partial_t\mathcal{E}^{(0)} + div((\mathcal{E}^{(0)} + p^{(0)})u^{(0)}) + \mathcal{O}(M_{\text{ref}}) = 0$$

Since $p^{(0)}$ and $\mathcal{E}^{(0)} = p^{(0)}/(\gamma-1)$ are constant in space, we obtain in the limit

$$\partial_t\mathcal{E}^{(0)} + (\mathcal{E}^{(0)} + p^{(0)})\, div(u^{(0)}) = 0 \tag{1.23}$$

$$\Rightarrow \partial_t \int_\Omega \mathcal{E}^{(0)}\, d^n x + (\mathcal{E}^{(0)} + p^{(0)}) \int_{\partial\Omega} u^{(0)} \cdot \nu \, d^{n-1}x = 0$$

with Ω being the n-dimensional spatial domain with outward facing normal vector field ν. If the domain is the torus (periodic, no boundary), or if we prescribe no-slip boundary conditions, the boundary term vanishes and we get

$$\partial_t \int_\Omega \mathcal{E}^{(0)}\, d^n x = \partial_t\mathcal{E}^{(0)} \int_\Omega 1\, d^n x = 0$$

$$\Rightarrow \partial_t\mathcal{E}^{(0)} = 0$$

That is, $\mathcal{E}^{(0)}$ and equivalently $p^{(0)}$ are not only constant in space, but also in time. Using this fact again in (1.23) gives $(\mathcal{E}^{(0)} + p^{(0)})\, div(u^{(0)}) = 0$. We required $p > 0$, so it follows that the flow is incompressible in the limit $M_{\text{ref}} \searrow 0$:

$$div(u^{(0)}) = 0 \tag{1.24}$$

Lastly let us look at the limiting mass equation. It is easily found to be

$$\partial_t\rho^{(0)} + div(\rho^{(0)}u^{(0)}) = 0 \tag{1.25}$$

which is just the continuum equation for the limit functions $\rho^{(0)}$, $u^{(0)}$.
In summary for the low Mach limit $\rho^{(0)} = \lim_{M_{\text{ref}} \to 0} \rho$, $u^{(0)} = \lim_{M_{\text{ref}} \to 0} u$ we find the system

$$\begin{aligned} \partial_t \rho^{(0)} + u^{(0)} \cdot \nabla_x \rho^{(0)} &= 0 \\ \partial_t(\rho^{(0)} u^{(0)}) + div(\rho^{(0)} u^{(0)} (u^{(0)})^T) + \nabla_x p^{(2)} &= 0 \\ div(u^{(0)}) &= 0 \end{aligned} \tag{1.26}$$

System (1.26) reads just like the incompressible Euler equations with the pressure $p^{(2)}$. See also [8] for a more in-depth discussion of asymptotic limits.

1.3 Solution concepts

This section is concerned with general conservation laws of the form $\partial_t q + div(f(q)) = 0$ and some basic ideas on how to interpret this equation more generally. Unfortunately, the classical formulation of this partial differential equation often does not have a solution for all times even for smooth initial data, but may still have physical relevance. E.g., for inviscid gas dynamics this is the case with shock waves, where the pressure has a jump discontinuity. To give meaning to the equations in such cases, other solution concepts have to be developed.

1.3.1 Integral form

Going back to the derivation of the Euler equations, we actually obtained an integral equation. Using the Gauß integral theorem, we can write any conservation law in n space dimensions in this form:

$$\int_\Omega q(x,t_1)\, d^n x - \int_\Omega q(x,t_0)\, d^n x + \int_{\partial\Omega} \int_{t_0}^{t_1} f(q(x,t))\nu \, dt \, d^{n-1} x = 0 \tag{1.27}$$

for any open $\Omega \subset \mathbb{R}^n$, for which an outward pointing boundary normal vector field ν is defined. The integral form gives meaning to integrable solutions that can be extended onto $\partial\Omega$ and are integrable on the boundary. The mentioned jump discontinuities don't pose a problem anymore.

1.3.2 Weak form

Using a variational type approach, an equivalent formulation for so called weak solutions is found: We consider only the one-dimensional case of a system of m equations and initial data
$q(x, 0) = q_0(x)$. For all smooth test functions $\phi : \mathbb{R} \times [0, \infty] \to \mathbb{R}^m$ with compact support it has to hold for classical solutions that

$$\int_0^\infty \int_{\mathbb{R}} (\partial_t q(x,t) + \partial_x (f(q(x,t)))\,) \cdot \phi(x,t)\, dx dt = 0$$

Integrating partially, the boundary terms vanish except on $\{t = 0\}$:

$$\int_0^\infty \int_{\mathbb{R}} (q(x,t) \cdot \partial_t \phi + f(q(x,t)) \cdot \partial_x \phi(x,t))\, dx dt + \int_{\mathbb{R}} q_0(x) \cdot \phi(x,0) dx = 0 \tag{1.28}$$

This formulation shifts all the derivatives to the test function and requires less regularity of the solution. See [7], p. 612, where this definition was taken from, for more details.

Unfortunately, while the weak form allows more general solutions, it does not guarantee uniqueness of them.

1.3.3 Vanishing viscosity

A different concept inspired by physics and gas dynamics is that of vanishing viscosity (see [13], pp. 210-211). Here, a diffusion term scaled by a parameter ε is added to the conservation law to work with smoother solutions:

$$\partial_t q^\epsilon + div(f(q^\varepsilon)) = \varepsilon \partial_{xx} q^\varepsilon \tag{1.29}$$

The solution of the original equation is then understood to be the limit (almost everywhere) of q^ε as $\varepsilon \searrow 0$.

1.3.4 Stability under perturbations

Another concept of a necessary condition on a selection method for weak solutions is that the function mapping the initial data to the selected solution at some later time should be continuous (w.r.t. some topologies ...).
For example, an arbitrary small (positive) viscosity will smoothen an unphysical shock wave which could then fan out into a so called rarefaction wave even when the viscosity is ”turned off”, since the unphysical shock is not stable under perturbations.

1.3.5 Entropy functions

Yet another concept inspired by physics is that of entropy. The physical entropy is a non-decreasing quantity whose volume density is a concave function of the conserved state variables. In mathematics instead one typically chooses the entropy density η to be a convex function of q that is advected according to an entropy flux ψ. For smooth solutions the entropy should be conserved, since there are no diffusive effects (only adiabatic changes in physics), so here we require

$$\partial_t(\eta(q)) + \partial_x(\psi(q)) = 0 \tag{1.30}$$

The fact that

$$\partial_t(\eta(q)) + \partial_x(\psi(q)) = D\eta(q)(-\partial_x(f(q))) + D\psi(q)\partial_x q = (D\psi(q) - D\eta(q)Df(q))\partial_x q = 0$$

motivates the constraint $D\psi = D\eta Df$.
Equation (1.30) does not hold (in a weak sense) for all (physically correct) weak solutions though. Following [7], pp. 646-648, we remember the vanishing viscosity approach. We use (1.29) to obtain

$$\partial_t(\eta(q^\varepsilon)) + \partial_x(\psi(q^\varepsilon)) = \varepsilon D\eta(q^\varepsilon)\partial_{xx}q^\varepsilon = \varepsilon\partial_{xx}(\eta(q^\varepsilon)) - (D^2\eta(q^\varepsilon)\partial_x q^\varepsilon)\partial_x q^\varepsilon \le \varepsilon\partial_{xx}(\eta(q^\varepsilon))$$

Let ϕ be a non-negative test function. Then with the help of integration by parts

$$\begin{aligned}
&\int_{\mathbb{R}}\int_{\mathbb{R}} (\eta(q^\varepsilon(x,t))\partial_t\phi(x,t) + \psi(q^\varepsilon(x,t))\partial_x\phi(x,t))\, dxdt \\
&= \int_{\mathbb{R}}\int_{\mathbb{R}} (-\phi(x,t)\partial_t(\eta(q^\varepsilon))(x,t) - \phi(x,t)\partial_x(\psi(q^\varepsilon))(x,t))\, dxdt \\
&\ge \int_{\mathbb{R}}\int_{\mathbb{R}} \phi(x,t)(-\varepsilon\partial_{xx}(\eta(q^\varepsilon))(x,t))\, dxdt \\
&= -\varepsilon\int_{\mathbb{R}}\int_{\mathbb{R}} \eta(q^\varepsilon(x,t))\partial_{xx}\phi(x,t)\, dxdt
\end{aligned}$$

We can take the limit $\varepsilon \to 0$ (under certain conditions) to find the weak form of the entropy inequality:

$$\int_{\mathbb{R}} \int_{\mathbb{R}} \left(\eta(q(x,t))\partial_t \phi(x,t) + \psi(q(x,t))\partial_x \phi(x,t)\right) dxdt \geq 0 \tag{1.31}$$

This weak inequality can now be used to single out physically relevant weak solutions of the conservation law.

Chapter 2

An invariant-domain preserving explicit scheme

This chapter explains the numerical method described in [12].

2.1 Finite element discretization

This section presents a method for numerically solving a conservation law

$$\partial_t q + div f(q) = 0 \tag{2.1}$$

based on a finite element discretization of space.

For a spatial domain Ω, let $\mathcal{T}_h$ be the mesh with a mesh size parameter h, composed of cells (in 2 dimensions we will use quadrilaterals) denoted $\Omega \supset K \in \mathcal{T}_h$ whose combined corners, called nodes, form a set $\{x^i\}_{i\in I}$ with an index set I (typically $I \subset \mathbb{Z}$). For any given $i \in I$ we will denote by I_i the subset of I for which the nodes $\{x^j\}_{j\in I_i}$ are adjacent to x^i in the sense that they share a cell. The finite element approximation space with Lagrangian elements is now spanned by a set of shape functions $\{\varphi_i\}_{i\in I}$, $\varphi_i : \Omega \to [0,1]$ which obey $\varphi_i(x^j) = \delta_{ij}$, i.e. φ_i takes the value 1 on node x^i and zero on all other nodes. We also require the partition of unity property:

$$\sum_{i\in I} \varphi_i \equiv 1 \tag{2.2}$$

In 2 space dimensions, such shape functions can be obtained by transforming the function $[0,1]^2 \ni (x,y) \mapsto (1-x)(1-y)$ onto other quadrilaterals (for bilinear Lagrangian elements). See figure (2.1) for an example of a shape function on a cartesian grid defined as

$$\varphi(x,y) = \begin{cases} (1-x)(1-y) & \text{if } (x,y)^T \in [0,1]^2 \\ (1+x)(1+y) & \text{if } (x,y)^T \in [-1,0]^2 \\ (1-x)(1+y) & \text{if } (x,y)^T \in [0,1]\times[-1,0] \\ (1+x)(1-y) & \text{if } (x,y)^T \in [-1,0]\times[0,1] \\ 0 & \text{else} \end{cases}$$

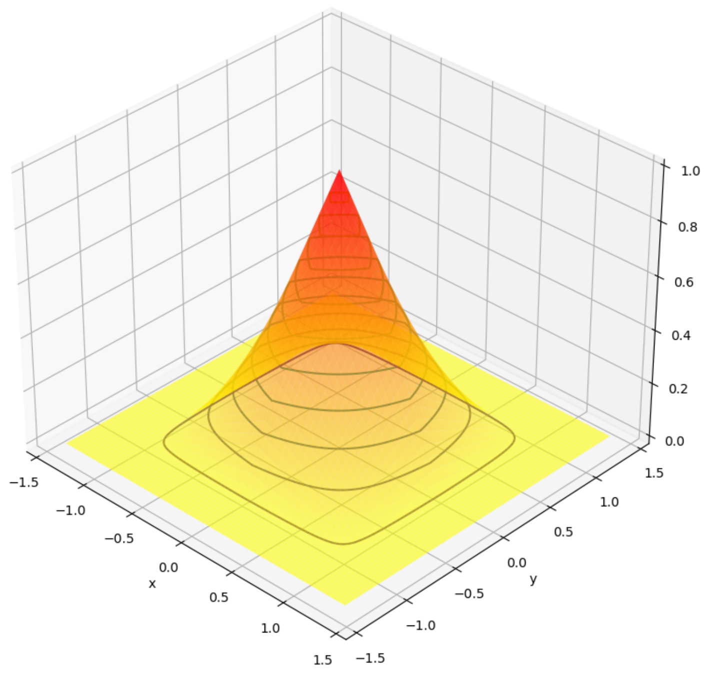


Figure 2.1: Plot of a bilinear shape function at node $(0,0)^T$ with the set $\mathbb{Z}^2$ of grid-points

Time is discretized into timepoints $\{t_n\}_n$.
The numerical approximation to the solution of (2.1) at a time t_n is now given as a linear combination of shape functions

$$q_h^n(x) = \sum_{i \in I} Q_i^n \varphi_i(x) \tag{2.3}$$

where the Q_i^n denote the numerical solution at node x^i and time t_n. Q^n is understood as a function $I \ni i \mapsto Q_i^n$.

2.1.1 Galerkin approach

A Galerkin type discretization of the variational formulation of (2.1) requires

$$\int_\Omega (\partial_t q(x,t_n) + div(f(q(x,t_n))))\varphi_i(x)dx = 0$$

For the time derivative we use the approximation

$$\int_\Omega \partial_t q(x,t_n)\varphi_i(x)dx \approx \int_\Omega \frac{q_h^{n+1}(x^i) - q_h^n(x^i)}{\tau_n}\varphi_i(x)dx = \frac{Q_i^{n+1} - Q_i^n}{\tau_n}\int_\Omega \varphi_i(x)dx$$

with the time step $\tau_n := t_{n+1} - t_n$
It is useful to also linearly approximate the flux in the divergence term

$$f(q(x,t_n)) \approx \sum_{i \in I} f(Q_i^n)\varphi_i(x)$$

With those approximations one obtains

$$\begin{aligned} 0 &= \int_\Omega \varphi_i(x)dx \frac{Q_i^{n+1} - Q_i^n}{\tau_n} + \sum_{j\in I} \int_\Omega div(f(Q_j^n)\varphi_j(x))\varphi_i(x)dx \\ &= \int_\Omega \varphi_i(x)dx \frac{Q_i^{n+1} - Q_i^n}{\tau_n} + \sum_{j\in I} f(Q_j^n) \int_\Omega \nabla\varphi_j(x)\varphi_i(x)dx \end{aligned}$$

Using the definitions

$$m_i := \int_\Omega \varphi_i(x)dx \tag{2.4}$$

$$c_{ij} := \int_\Omega \varphi_i(x)\nabla\varphi_j(x)dx \tag{2.5}$$

the obtained formula can be written more compactly:

$$m_i \frac{Q_i^{n+1} - Q_i^n}{\tau_n} + \sum_{j\in I} f(Q_j^n)c_{ij} = 0 \tag{2.6}$$

Because $c_{ij} = 0$ if $j \notin I_i$, the we only have to sum a few values:

$$m_i \frac{Q_i^{n+1} - Q_i^n}{\tau_n} + \sum_{j\in I_i} f(Q_j^n)c_{ij} = 0 \tag{2.7}$$

As it turns out, this explicit update formula would in general produce an unstable scheme. The solution is to add the right amount of artificial viscosity. This is done by introducing a graph viscosity corresponding to diffusion along edges between nodes. The viscosity coefficients are denoted d_{ij}^n (diffusion between nodes x^i and x^j at time t_n).
They should be symmetric:

$$d_{ij}^n = d_{ji}^n$$

nonnegative:

$$d_{ij}^n \geq 0 \ , \ i \neq j$$

and balanced:

$$\sum_{j\in I_i} d_{ij}^n = 0 \tag{2.8}$$

The correct update formula takes the form

$$m_i \frac{Q_i^{n+1} - Q_i^n}{\tau_n} + \sum_{j\in I_i} f(Q_j^n)c_{ij} = \sum_{j\in I_i} Q_j^n d_{ij}^n \tag{2.9}$$

We now need to determine the graph viscosity from some stability concept. The choice is the property of convex invariant-domain preservation.

2.2 Invariant domain preservation

Definition 2.2.1 (Invariant domain) *Denote the unit sphere in space by S.*
For a left state q_l , a right state q_r and $\nu \in S$, let $q_{q_l,q_r,\nu}$ be the solution to the Riemann problem

$$q_{q_l,q_r,\nu}(x,0) = \begin{cases} q_l & x < 0 \\ q_r & x > 0 \end{cases} \tag{2.10}$$

of the projected conservation law

$$\partial_t q_{q_l,q_r,\nu} + \partial_x(f(q_{q_l,q_r,\nu}) \cdot \nu) = 0$$

A convex set D is called invariant, if and only if

$$\forall (q_l, q_r) \in D \times D \; \forall \nu \in S \; \forall t \in \left(0, \frac{1}{2\lambda_{max}(q_l, q_r, \nu)}\right) : \bar{q}(q_l, q_r, \nu, t) := \int_{-1/2}^{1/2} q_{q_l,q_r,\nu}(s,t)ds \in D \tag{2.11}$$

where $\lambda_{max}(q_l, q_r, \nu)$ denotes the maximum wavespeed in the solution to Riemann problem (2.10), which we understand as the minimal continuous function satisfying

$$\forall t > 0 : \forall x < -\lambda_{max}(q_l, q_r, \nu)t : q_{q_l,q_r,\nu}(x,t) = q_l$$
$$\forall t > 0 : \forall x > \lambda_{max}(q_l, q_r, \nu)t : q_{q_l,q_r,\nu}(x,t) = q_r$$

An example of such an invariant domain is the set of states with positive density and positive pressure for the Euler equations.

2.2.1 The scheme

For nodes with index i in the interior of the numerical domain it holds that

$$c_{ij} = \int_\Omega \varphi_i(x)\nabla\varphi_j(x)dx = -\int_\Omega \varphi_j(x)\nabla\varphi_i(x)dx = -c_{ji} \tag{2.12}$$

since here the function $\varphi_i\varphi_j$ vanishes on the boundary. Also

$$\sum_{j\in I_i} c_{ij} = \int_\Omega \varphi_i(x)\nabla \sum_{j\in I_i} \varphi_j(x)dx = 0 \tag{2.13}$$

since $\nabla \sum_{j\in I_i} \varphi_j = 0$ due to (2.2).
Neglecting boundary conditions and using

$$d_{ii}^n = -\sum_{j\in I_i\setminus\{i\}} d_{ij}^n \tag{2.14}$$

due to balance, the update formula (2.9) can therefore be rewritten

$$
\begin{aligned}
Q_i^{n+1} &= Q_i^n - \frac{\tau_n}{m_i}\sum_{j\in I_i} f(Q_j^n)c_{ij} + \frac{\tau_n}{m_i}\sum_{j\in I_i} Q_j^n d_{ij}^n \\
&= Q_i^n - \frac{\tau_n}{m_i}\sum_{j\in I_i\setminus\{i\}} 2d_{ij}^n f(Q_j^n)\frac{c_{ij}}{||c_{ij}||}\frac{||c_{ij}||}{2d_{ij}^n} + \frac{\tau_n}{m_i}\sum_{j\in I_i\setminus\{i\}} f(Q_i^n)c_{ij} + \frac{\tau_n}{m_i}\sum_{j\in I_i} Q_j^n d_{ij}^n \\
&= Q_i^n - \frac{\tau_n}{m_i}\sum_{j\in I_i\setminus\{i\}} 2d_{ij}^n (f(Q_j^n) - f(Q_i^n))\frac{c_{ij}}{||c_{ij}||}\frac{||c_{ij}||}{2d_{ij}^n} + \frac{\tau_n}{m_i}\left(2Q_i^n d_{ii}^n - Q_i^n d_{ii}^n + \sum_{j\in I_i\setminus\{i\}} Q_j^n d_{ij}^n\right) \\
&= Q_i^n - \frac{\tau_n}{m_i}\sum_{j\in I_i\setminus\{i\}} 2d_{ij}^n \left((f(Q_j^n) - f(Q_i^n))\frac{c_{ij}}{||c_{ij}||}\frac{||c_{ij}||}{2d_{ij}^n} - (Q_i^n + Q_j^n)/2\right) + \frac{\tau_n}{m_i}2Q_i^n d_{ii}^n \\
&= Q_i^n - \sum_{j\in I_i\setminus\{i\}} \frac{2d_{ij}^n\tau_n}{m_i}\left((f(Q_j^n) - f(Q_i^n))\frac{c_{ij}}{||c_{ij}||}\frac{||c_{ij}||}{2d_{ij}^n} - (Q_i^n + Q_j^n)/2\right) - \sum_{j\in I_i\setminus\{i\}} \frac{2d_{ij}^n\tau_n}{m_i} Q_i^n
\end{aligned}
$$

Defining

$$
\bar{Q}_{ij}^{n+1} := (Q_i^n + Q_j^n)/2 - (f(Q_j^n) - f(Q_i^n))\frac{c_{ij}}{||c_{ij}||}\frac{||c_{ij}||}{2d_{ij}^n} \tag{2.15}
$$

yields the formulation

$$
Q_i^{n+1} = \left(1 - \sum_{j\in I_i\setminus\{i\}} \frac{2d_{ij}^n\tau_n}{m_i}\right) Q_i^n + \sum_{j\in I_i\setminus\{i\}} \frac{2d_{ij}^n\tau_n}{m_i}\bar{Q}_{ij}^{n+1} \tag{2.16}
$$

$\bar{Q}_{ij}^{n+1}$ coincides with $\bar{q}(q_l, q_r, \nu, t)$ in definition 2.2.1 for left state $q_l = Q_i^n$, right state $q_r = Q_j^n$, $\nu = \frac{c_{ij}}{||c_{ij}||}$ and $t = \frac{||c_{ij}||}{2d_{ij}^n}$. Therefore, choosing

$$
\frac{||c_{ij}||}{2d_{ij}^n} \leq \frac{1}{2\lambda_{\max}(q_l, q_r, \nu)}
$$

if $i \neq j$ ensures that $\bar{Q}_{ij}^{n+1}$ lies in any invariant domain containing both Q_i^n and Q_j^n. This property motivates the choice

$$
d_{ij}^n = \lambda_{\max}(Q_i^n, Q_j^n, n_{ij})||c_{ij}||\ ,\ i \neq j \tag{2.17}
$$

with the definition $n_{ij} = \frac{c_{ij}}{||c_{ij}||}$ and (2.14).

Of course, this alone does not make the scheme invariant domain preserving. Looking at (2.16), we note that Q_i^{n+1} is obtained as a convex combination of Q_i^n and $\bar{Q}_{ij}^{n+1}$ if

$$
\sum_{j\in I_i\setminus\{i\}} \frac{2d_{ij}^n\tau_n}{m_i} \leq 1
$$

for all j (we use here the fact that the d_{ij}^n are nonnegative for $i \neq j$).
This, along with $\sum_{j\in I_i\setminus\{i\}} d_{ij}^n = -d_{ii}^n$ yields

$$
-2d_{ii}^n\frac{\tau_n}{m_i} \leq 1
$$

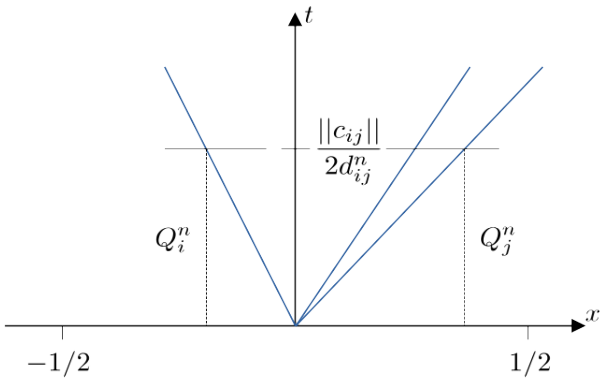


Figure 2.2: Illustrated Riemann problem arising in the derivation of the invariant-domain preserving scheme

Now, if Q_i^n and Q_j^n lie in the same convex invariant domain for all j, so will the updated value Q_i^{n+1}. Since for a given time step size τ_n, that condition has to hold for every i, one obtains a constraint called the *CFL-condition*:

$$\tau_n \leq \inf_i \frac{m_i}{-2d_{ii}^n} \tag{2.18}$$

Conditions (2.17) and (2.18) combined make the method invariant-domain preserving.

2.3 Entropy inequality

In order to further justify the choice of the constraints on the graph viscosity coefficients, let us derive a discrete entropy inequality. For a given convex entropy density η with entropy-flux ψ, the entropy inequality reads

$$\partial_t \eta(q) + div(\psi(q)) \leq 0$$

Applying the convexity of η to (2.16) yields

$$\eta(Q_i^{n+1}) \leq \left(1 - \sum_{j\in I_i\setminus\{i\}} \frac{2d_{ij}^n\tau_n}{m_i}\right)\eta(Q_i^n) + \sum_{j\in I_i\setminus\{i\}} \frac{2d_{ij}^n\tau_n}{m_i}\eta(\bar{Q}_{ij}^{n+1})$$

Recalling that $Q_{ij}^{n+1} = \bar{q}(Q_i^n, Q_j^n, \frac{c_{ij}}{||c_{ij}||}, \frac{||c_{ij}||}{2d_{ij}^n})$ from definition 2.2.1:

$$\eta(\bar{Q}_{ij}^{n+1}) = \eta\left(\bar{q}\left(Q_i^n, Q_j^n, \frac{c_{ij}}{||c_{ij}||}, \frac{||c_{ij}||}{2d_{ij}^n}\right)\right) = \eta\left(\int_{-1/2}^{1/2} q_{Q_i^n, Q_j^n, \frac{c_{ij}}{||c_{ij}||}}\left(s, \frac{||c_{ij}||}{2d_{ij}^n}\right) ds\right)$$

The Jensen inequality then gives

$$\eta(\bar{Q}_{ij}^{n+1}) \leq \int_{-1/2}^{1/2} \eta(q_{Q_i^n, Q_j^n, \frac{c_{ij}}{||c_{ij}||}}(s, \frac{||c_{ij}||}{2d_{ij}^n}))ds$$

Now we can use the integral form of the entropy inequality:

$$\eta(\bar{Q}_{ij}^{n+1}) \leq (\eta(Q_i^n) + \eta(Q_j^n))/2 - (\psi(Q_j^n) - \psi(Q_i^n)) \cdot \frac{c_{ij}}{||c_{ij}||} \frac{||c_{ij}||}{2d_{ij}^n}$$

From those inequalities we obtain

$$\begin{aligned}
&m_i \frac{\eta(Q_i^{n+1} - Q_i^n)}{\tau_n} \leq \sum_{j \in I_i \setminus \{i\}} 2d_{ij}^n (\eta(\bar{Q}_{ij}^{n+1}) - \eta(Q_i^n)) \\
&\leq \sum_{j \in I_i \setminus \{i\}} [d_{ij}^n (\eta(Q_j^n) - \eta(Q_i^n)) - (\psi(Q_j^n) - \psi(Q_i^n)) \cdot c_{ij}] \\
&= \sum_{j \in I} [d_{ij}^n \eta(Q_j^n) - \psi(Q_j^n) \cdot c_{ij}] \\
&= -\int div \left(\sum_{j \in I} \psi(Q_j^n) \varphi_j(x) \right) \varphi_i(x) \, dx + \sum_{j \in I} d_{ij}^n \eta(Q_j^n)
\end{aligned}$$

which gives the discrete entropy inequality

$$\int_\Omega \frac{\eta(Q_i^{n+1}) - \eta(Q - i^n)}{\tau_n} + div \left(\sum_{j \in I} \psi(Q_j^n) \varphi_j(x) \right) \varphi_i(x) \, dx \leq \sum_{j \in I} d_{ij}^n \eta(Q_j^n) \qquad (2.19)$$

with the approximations

$$\int_\Omega \frac{\eta(Q_i^{n+1}) - \eta(Q - i^n)}{\tau_n} \varphi_i(x) \approx \int_\Omega \partial_t q(x, t_n) \varphi_i(x) \, dx$$

and $\psi(q(x, t_n)) \approx \sum_{j \in I} \psi(Q_j^n) \varphi_j(x)$

Chapter 3

IMEX approach to low Mach

3.1 Speed of sound

The CFL-condition (2.18) can be written

$$\tau_n \le \inf_{i\in I} \frac{m_i}{2\sum_{j\in I_i\setminus\{i\}} \lambda_{\max}(Q_i^n, Q_j^n, n_{ij})||c_{ij}||} \le \inf_{i\in I} \frac{m_i}{\min_{j\in I_i\setminus\{i\}} ||c_{ij}||} \left(\min_{j\in I_i\setminus\{i\}} \lambda_{\max}(Q_i^n, Q_j^n, n_{ij})\right)^{-1}$$

We observe that besides the mesh size, the wavespeeds restrict the maximum possible time step size. As the mesh gets refined, neighboring values in the numerical solution approach each other for continuous solutions. Here we can approximate the wavespeeds by linearizing the equations about a state at a given position. In one space dimension, the conservation law $\partial_t q + \partial_x f(q) = 0$ in smooth regions of the solution can be rewritten as $\partial_t q + Df(q)\partial_x q = 0$ and linearized about a constant state $\bar{q}$ it reads:

$$\partial_t q + Df(\bar{q})\partial_x q = 0$$

This partial differential equation is called *hyperbolic* if $Df(\bar{q})$ can be diaganolized as $Df(\bar{q}) = V\Lambda V^{-1}$ and has only real eigenvalues $\lambda_k = \Lambda_{kk}$ (see [13], p.3). This is the case for the (scaled) Euler equations. Defining $v = V^{-1}q$, the linearized conservation law then becomes

$$\partial_t v + \Lambda\partial_x v = 0$$

Since Λ is diagonal, the system decouples into a set of scalar equations $\partial_t v_k + \lambda_k\partial_x v_k = 0$, whose solution is given by $v_k(x,t) = v_k(x - \lambda_k t)$. Therefore the maximum linear wavespeed is the maximum absolute eigenvalue of $Df(\bar{q})$. The 1D linear wavespeeds for the scaled Euler equations

$$\partial_t \begin{pmatrix} \rho \\ \rho u \\ \mathcal{E} \end{pmatrix} + div \begin{pmatrix} \rho u^T \\ \rho u u^T + \frac{p}{{M_{\text{ref}}}^2} \\ (\mathcal{E} + p)u^T \end{pmatrix} = 0 \tag{3.1}$$

are found to be $\lambda_2 = u$, $\lambda_{1,3} = u \pm c$ with the *speed of sound* $c = \frac{1}{M_{\text{ref}}}\sqrt{\gamma\frac{p}{\rho}}$. We notice that for a given state in primitive variables (ρ, u, p), the speed of sound tends to ∞ as $M_{\text{ref}} \searrow 0$. This poses a problem for the explicit scheme, because it forces the time steps to become very small and therefore the number of timesteps necessary to simulate up to some time T blows up.
A relevant indicator here is the (local) **Mach number**

$$Ma = \frac{||u||}{c} \tag{3.2}$$

which kind of compares the CFL-restrictions from the 2-waves (λ_2) to the acoustic waves traveling with the speed of sound relative to the gas.

3.2 Splittings

To solve this issue, one method is to formulate the scheme implicitly, which allows information to travel arbitrarily fast in the numerical solution. Performing an implicit update however is more expensive than an explicit one. Therefore we aim to keep implicit formulations simple. This is achieved by splitting the equations into two parts. There are several ways in which one can do this. An example is the scheme described in [18]. We propose a method along the lines of [3], where the flux is split into:

- An explicit part
$$\partial_t q + div(f^{\text{ex}}(q)) = 0 \tag{3.3}$$
resulting only in slow waves
- An implicit part
$$\partial_t q + div(f^{\text{im}}(q)) = 0 \tag{3.4}$$
to be solved implicitly.

Here we have split the flux $f(q) = f^{\text{ex}}(q) + f^{\text{im}}(q)$. For given $(x,t) \in \mathbb{R}^n \times \mathbb{R}$, the exact solution for a time-step τ is

$$\begin{aligned} q(x, t+\tau) &= q(x,t) + \int_t^{t+\tau} \partial_t q(x,s)\, ds \\ &= q(x,t) - \int_t^{t+\tau} div(f(q(x,s)))\, ds \\ &= q(x,t) - \int_t^{t+\tau} div(f^{\text{ex}}(q(x,s)))\, ds - \int_t^{t+\tau} div(f^{\text{im}}(q(x,s)))\, ds \end{aligned}$$

Let q^{ex} and q^{im} denote the solution to the explicit and the implicit part respectively with the initial data $q^{\text{ex}}(x,t) = q(x,t)$ and $q^{\text{im}}(x,t) = q^{\text{ex}}(x,t+\tau)$. In $[t, t+\tau]$ those functions only differ by an amount in $\mathcal{O}(\tau)$, so we can approximate

$$\int_t^{t+\tau} div(f^{\text{ex}}(q(x,s)))\, ds \approx \int_t^{t+\tau} div(f^{\text{ex}}(q^{\text{ex}}(x,s)))\, ds$$

and

$$\int_t^{t+\tau} div(f^{\text{im}}(q(x,s)))\, ds \approx \int_t^{t+\tau} div(f^{\text{im}}(q^{\text{im}}(x,s)))\, ds$$

Therefore, $q(x, t+\tau)$ can be approximated as

$$q(x,t+\tau) \approx \underbrace{q(x,t) - \int_t^{t+\tau} div(f^{\text{ex}}(q^{\text{ex}}(x,s)))\, ds}_{q^{\text{ex}}(x,t+\tau)} - \int_t^{t+\tau} div(f^{\text{im}}(q^{\text{im}}(x,s)))\, ds = q^{\text{im}}(x,t+\tau)$$

In the numerical method we first solve the explicit part to obtain an approximation to $q^{\text{ex}}(x,t+\tau)$ and then the implicit part with this explicit solution as initial data to finalize the update:

Starting from data Q^n at time t_n, the explicit method produces an intermediate state $\hat{Q}^{n+1}$ from which the implicit scheme gives Q^{n+1}.

$$Q^n \xrightarrow{\text{Explicit time-step } \tau_n} \hat{Q}^{n+1} \xrightarrow{\text{Implicit time-step } \tau_n} Q^{n+1}$$

The time-step τ_n is determined by the CFL-condition for the explicit scheme.

Which splittings may be used? To investigate this question, consider two different splittings of the flux for the scaled Euler equations (1.19):

1. First splitting:

$$f(q) = \underbrace{\begin{pmatrix} \rho u^T \\ \rho u u^T + pI \\ (\mathcal{E} + {M_{\text{ref}}}^2 p) u^T \end{pmatrix}}_{f^{\text{ex}}(q)} + \underbrace{(1 - {M_{\text{ref}}}^2) \begin{pmatrix} 0 \\ \frac{p}{{M_{\text{ref}}}^2} I \\ p u^T \end{pmatrix}}_{f^{\text{im}}(q)} \tag{3.5}$$

2. Second splitting:

$$f(q) = \underbrace{\begin{pmatrix} \rho u^T \\ \rho u u^T \\ \frac{{M_{\text{ref}}}^2}{2} \rho ||u||^2 u^T \end{pmatrix}}_{f^{\text{ex}}(q)} + \underbrace{\begin{pmatrix} 0 \\ \frac{p}{{M_{\text{ref}}}^2} I \\ \frac{\gamma}{\gamma-1} p u^T \end{pmatrix}}_{f^{\text{im}}(q)} \tag{3.6}$$

To analyze the splittings, again look at the one-dimensional linearized equations. It is useful to also rewrite them in primitive variables.

For the first splitting, we obtain

$$\partial_t \begin{pmatrix} \rho \\ u \\ p \end{pmatrix} + \left[\underbrace{\begin{pmatrix} u & \rho & 0 \\ 0 & u & \frac{1}{\rho} \\ 0 & (1 + (\gamma - 1){M_{\text{ref}}}^2) p & u \end{pmatrix}}_{\text{explicit}} + \underbrace{\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & \frac{1 - {M_{\text{ref}}}^2}{{M_{\text{ref}}}^2 \rho} \\ 0 & (\gamma - 1)(1 - {M_{\text{ref}}}^2) p & 0 \end{pmatrix}}_{\text{implicit}} \right] \partial_x \begin{pmatrix} \rho \\ u \\ p \end{pmatrix} = 0 \tag{3.7}$$

The eigenvalues of the explicit part are $\lambda_2 = u$ and $\lambda_{1,3} = u \pm \sqrt{(1 + (\gamma - 1){M_{\text{ref}}}^2) \frac{p}{\rho}}$, so the maximum absolute linear wavespeed is $\lambda_{\max} = |u| + \sqrt{(1 + (\gamma - 1){M_{\text{ref}}}^2) \frac{p}{\rho}}$.

For the second splitting, we obtain

$$\partial_t \begin{pmatrix} \rho \\ u \\ p \end{pmatrix} + \left[\underbrace{\begin{pmatrix} u & \rho & 0 \\ 0 & u & 0 \\ 0 & 0 & 0 \end{pmatrix}}_{\text{explicit}} + \underbrace{\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & \frac{1}{{M_{\text{ref}}}^2 \rho} \\ 0 & \gamma p & u \end{pmatrix}}_{\text{implicit}} \right] \partial_x \begin{pmatrix} \rho \\ u \\ p \end{pmatrix} = 0 \tag{3.8}$$

The eigenvalues of the explicit part clearly are $\lambda_{1,2} = u$ and $\lambda_3 = 0$, so the maximum absolute linear wavespeed is $\lambda_{\max} = |u|$.

Both splittings have explicit parts whose maximum wavespeeds for a given state (ρ, u, p) are of order $\mathcal{O}(1)$ as $M_{\text{ref}} \searrow 0$ and therefore allow reasonable time-step sizes for explicit methods restricted by the CFL-condition. However, only one of those splittings actually works: Splitting an equation introduces an error even if the explicit and implicit equations are solved exactly. This error may blow up over time resulting in an unstable method.

3.2.1 Stability analysis

The following analysis is performed like the one described in [1].
Let the one-dimensional equations be linearized about a state $\bar{w} = (\bar{\rho}, \bar{u}, \bar{p})^T$ in primitive variables:

$$\partial_t w + A(\bar{w})\partial_x w = 0$$

In the following we will assume $\bar{\rho} > 0$ and $\bar{p} > 0$. We analyze the L^2 stability of the splitting on the torus $\mathbb{T}$, i.e. the interval $[-\pi, \pi]$ with a periodic boundary. It is well known that the Fourier-system is an orthonormal basis of $L^2(\mathbb{T})$. Let w_{ex} be a solution of the linearized explicit part

$$\partial_t w^{\text{ex}} + A^{\text{ex}}(\bar{w})\partial_x w^{\text{ex}} = 0 \tag{3.9}$$

and w^{im} a solution of the implicit part

$$\partial_t w^{\text{im}} + A^{\text{im}}(\bar{w})\partial_x w^{\text{im}} = 0 \tag{3.10}$$

expanded as a Fourier series

$$w^{\text{ex}}(x,t) = \sum_{k\in\mathbb{Z}} \tilde{w}_k^{\text{ex}}(t)e^{\text{i}kx}$$

and

$$w^{\text{im}}(x,t) = \sum_{k\in\mathbb{Z}} \tilde{w}_k^{\text{im}}(t)e^{\text{i}kx}$$

with the time-dependent vector-valued Fourier coefficients $\tilde{w}_k^{\text{ex}}(t)$, $\tilde{w}_k^{\text{im}}(t)$ respectively. Inserting those representations into (3.9) respectively (3.10) one obtains

$$\partial_t \tilde{w}_k^{\text{ex}} + \text{i}kA^{\text{ex}}(\bar{w})\tilde{w}_k^{\text{ex}} = 0 \;\forall k \in \mathbb{Z}$$

and

$$\partial_t \tilde{w}_k^{\text{im}} + \text{i}kA^{\text{im}}(\bar{w})\tilde{w}_k^{\text{im}} = 0 \;\forall k \in \mathbb{Z}$$

which are decoupled systems of linear ODE's whose time evolution operators for a time-step τ are known to be given by the matrix exponentials

$$\tilde{w}_k^{\text{ex}}(t+\tau) = \underbrace{\exp(-\text{i}k\tau A^{\text{ex}}(\bar{w}))}_{=:S_k^{\text{ex}}}\tilde{w}_k^{\text{ex}}(t)$$

and

$$\tilde{w}_k^{\text{im}}(t+\tau) = \underbrace{\exp\big(-\text{i}k\tau A^{\text{im}}(\bar{w})\big)}_{=:S_k^{\text{im}}}\tilde{w}_k^{\text{ex}}(t)$$

The splitting now takes some initial data

$$w_0(x) = w(x,0) = \sum_{k\in\mathbb{Z}} \tilde{w}_k(0)e^{\text{i}kx}$$

and applies first the explicit operator followed by the implicit one. The evolution operator of the splitting for a particular Fourier mode, also called the *amplification matrix* is simply the concatenation of the implicit with the explicit operators of that mode:

$$S_k := \exp\big(-\text{i}k\tau A^{\text{im}}(\bar{w})\big)\exp(-\text{i}k\tau A^{\text{ex}}(\bar{w})) \tag{3.11}$$

This produces updated data

$$w^{\text{imex}}(x, n\tau) = \sum_{k\in\mathbb{Z}} (S_k)^n \tilde{w}_k(0) e^{\mathrm{i}kx}$$

To investigate stability, write the L^2 norm of the imex solution using the Fourier expansion:

$$||w^{\text{imex}}(\cdot, n\tau)||_2^2 = \sum_{k\in\mathbb{Z}} ||(S_k)^n \tilde{w}_k(0)||_2^2 \tag{3.12}$$

Stability here means that for every initial data there exists an upper bound on the norm of the solution at all times. This is a reasonable requirement since the exact solution simply consists of a family of waves that do not blow up. A necessary condition for stability is that the spectral radius σ of the evolution operators is at most 1:

$$\forall k \in \mathbb{N} : \sigma(S_k) \leq 1$$

We only need to check for $k \in \mathbb{N}$ (because $\sigma(S_{-k}) = \sigma(S_k)$ and $\sigma(S_0) = 1$).
This condition is necessary since otherwise for some k one could construct initial data of the form $w_0(x) = \bar{w} + \epsilon \operatorname{Re}(e^{\mathrm{i}kx} v)$ for an eigenvector v of S_k to an eigenvalue with modulus greater 1:
According to (3.12), w^{imex} will blow up for the complex initial data $w_0(x) = \bar{w} + \epsilon e^{\mathrm{i}kx} v$. Both real and imaginary part of w_{imex} are again solutions. At least one of them also has to blow up. The initial data $w_0(x) = \bar{w} + \epsilon \operatorname{Im}(e^{\mathrm{i}kx} v) = \bar{w} + \epsilon \operatorname{Re}(e^{\mathrm{i}k(x-\pi/(2k))} v)$ is just a translation of the real part. Due to translational invariance indeed both blow up.

For the first splitting set $\bar{w} = (1.4, 0.5, 1)^T$, $M_{\text{ref}} = 0.2$, $\gamma = 1.4$ and numerically compute the spectral radii of the S_k. Figure (3.1) shows that the necessary condition for linear stability

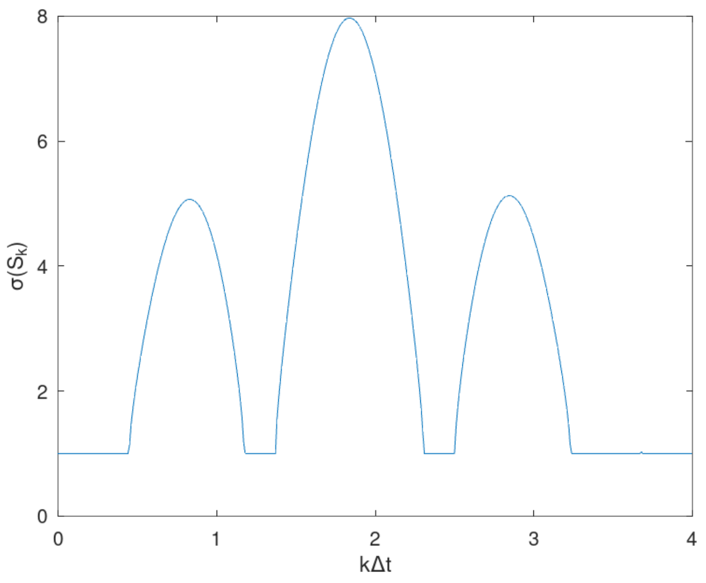


Figure 3.1: Spectral radius of S_k plotted over $k\tau$ for $\bar{w} = (1.4, 0.5, 1)^T$, $M_{\text{ref}} = 0.2$, $\gamma = 1.4$

is violated for the given example data. We should therefore expect for this splitting to result in unstable schemes.

For the second splitting and given $\bar{w} = (\bar{\rho}, \bar{u}, \bar{p})^T$ computing the amplification matrix gives

$$S_k = e^{-\mathrm{i}k\tau\bar{u}} \begin{pmatrix} 1 & 0 & 0 \\ 0 & e^{-\mathrm{i}k\tau\bar{u}/2}(\cos(k\tau a) + \mathrm{i}\frac{\bar{u}}{2a}\sin(k\tau a)) & 0 \\ 0 & 0 & e^{\mathrm{i}k\tau\bar{u}/2}(\cos(k\tau a) - \mathrm{i}\frac{\bar{u}}{2a}\sin(k\tau a)) \end{pmatrix} \tag{3.13}$$

with $a = \sqrt{\left(\frac{\bar{u}}{2}\right)^2 + \gamma\frac{\bar{p}}{{M_{\mathrm{ref}}}^2\bar{\rho}}}$. Since $|\bar{u}| \leq 2a > 0$, we get

$$|\cos(k\tau a) + \mathrm{i}\frac{\bar{u}}{2a}\sin(k\tau a)| \leq 1 \qquad |\cos(k\tau a) - \mathrm{i}\frac{\bar{u}}{2a}\sin(k\tau a)| \leq 1$$

Therefore, the spectral radius is 1 for all S_k. Furthermore, the amplification matrix is diagonal, so the L^2-norm of the IMEX solution is non-increasing:

$$||w^{\mathrm{imex}}(\cdot, n\tau)||_2^2 = \sum_{k\in\mathbb{Z}} \underbrace{||(S_k)^n\tilde{w}_k(0)||_2^2}_{\leq||\tilde{w}_k(0)||_2^2} \leq ||w^{\mathrm{imex}}(\cdot, 0)||_2^2$$

Therefore, the splitting is linearly L^2-stable.

3.3 IMEX adapted invariant-domain preserving method

The following section describes the main scheme, the adaptation of the already derived invariant-domain preserving method to an IMEX method to be used in low Mach computations. The IMEX scheme is built in analogy to that described in [3].

The explicit part is solved directly by solving (2.9) for the explicit flux:

$$m_i\frac{\hat{Q}_i^{n+1} - Q_i^n}{\tau_n} + \sum_{j\in I_i} f^{\mathrm{ex}}(Q_j^n)c_{ij} = \sum_{j\in I_i} Q_j^n d_{ij}^n \tag{3.14}$$

The graph viscosity coefficients are computed from (2.17), where the maximum absolute wavespeeds refer to the explicit problem (3.3). The time-step size τ_n is then chosen to satisfy (2.18). For the implicit part, this equation (with implicit flux) is modified to be implicit and the graph viscosity that was necessary for stability of the explicit scheme can be discarded:

$$m_i\frac{Q_i^{n+1} - \hat{Q}_i^{n+1}}{\tau_n} + \sum_{j\in I_i} f^{\mathrm{im}}(Q_j^{n+1})c_{ij} = 0 \tag{3.15}$$

We use the second splitting of the fluxes (3.6).

To see why writing a scheme in this implicit form makes sense if we want to avoid the CFL-condition, consider the one-dimensional advection equation

$$\partial_t q + a\partial_x q = 0$$

with $a > 0$, discretized on a periodic grid with index set $I = \mathbb{Z}/(2N+1)\mathbb{Z}$ represented with the numbers $0, ..., 2N$ and nodes

$$\forall i \in \{0, 1, ..., 2N\} : x^i = hi \tag{3.16}$$

That means we have $x^{-1} = x^{2N}$, $x^{2N+1} = x^0$ and $Q^n_{-1} = Q^n_{2N}$, $Q^n_{2N+1} = Q^n_0$. Using piecewise linear shape-functions, the finite element coefficients are

$$m_i = h \qquad\qquad c_{ij} = \frac{1}{2}(\delta_{x^{i+1},x^j} - \delta_{x^i,x^{j+1}})$$

From the obvious wavespeed $\lambda_{\max} = a$ the graph viscosity is obtained:

$$d_{ij} = \frac{a}{2}(\delta_{x^{i+1},x^j} + \delta_{x^i,x^{j+1}}) - a\delta_{i,j}$$

The fully explicit method for this equation then becomes

$$\begin{aligned} Q^{n+1}_i &= Q^n_i + \frac{\tau_n}{h}\left(\frac{a}{2}(Q^n_{i-1} + Q^n_{i+1}) - aQ^n_i + \frac{a}{2}(Q^n_{i-1} - Q^n_{i+1})\right) \\ &= Q^n_i + \frac{\tau_n}{h}a\left(Q^n_{i-1} - Q^n_i\right) \\ Q^{n+1}_i &= Q^n_i + \frac{\tau_n}{h}\left(\frac{a}{2}(Q^n_{i-1} + Q^n_{i+1}) - aQ^n_i + \frac{a}{2}(Q^n_{i-1} - Q^n_{i+1})\right) \end{aligned}$$

To analyze the stability, write the scheme in the Fourier basis. That is with $Q^n_i = \sum_{k=-N}^N \tilde{Q}^n_k e^{\mathrm{i}\pi ki/N}$ we obtain

$$\tilde{Q}^{n+1}_k = \left(1 + \frac{\tau_n}{h}a(e^{-\mathrm{i}\pi k/N} - 1)\right)\tilde{Q}^n_k$$

The amplification factor lies in the closed complex unit ball iff $\frac{\tau_n}{h} \leq \frac{1}{a}$, which is precisely the CFL-condition the invariant-domain preservation concept yields.
The implicit method reads

$$Q^{n+1}_i = Q^n_i + \frac{\tau_n}{h}\frac{a}{2}(Q^{n+1}_{i-1} - Q^{n+1}_{i+1})$$

which gives

$$\begin{aligned} &\tilde{Q}^{n+1}_k = \tilde{Q}^n_k + \frac{\tau_n}{h}\frac{a}{2}\left(e^{-\mathrm{i}\pi k/N} - e^{\mathrm{i}\pi k/N}\right)\tilde{Q}^{n+1}_k \\ &\Rightarrow \tilde{Q}^{n+1}_k = \left[1 - \frac{\tau_n}{h}\frac{a}{2}\left(e^{-\mathrm{i}\pi k/N} - e^{\mathrm{i}\pi k/N}\right)\right]^{-1}\tilde{Q}^n_k = \left[1 + \mathrm{i}\frac{\tau_n}{h}a\sin(\pi k/N)\right]^{-1}\tilde{Q}^n_k \end{aligned}$$

Since $|1 + \mathrm{i}\frac{\tau_n}{h}a\sin(\pi k/N)| \geq 1$, the implicit method is (L^2-)stable for arbitrary large timesteps.

Going back to the scheme for the rescaled Euler equations, denote by $[\rho]^n_i$ the density component of the numerical solution at (x^i, t_n) and analogously for other variables. Denote by $[\hat{\rho}]^n_i$ the intermediate states etc. In order to solve the implicit part we first observe that

$$[\rho]^{n+1}_i = [\hat{\rho}]^{n+1}_i \tag{3.17}$$

since the first row in f^{im} (corresponding to the density variable) is zero.
Rewrite the implicit momentum equation as

$$[\rho u]^{n+1}_i - [\hat{\rho u}]^{n+1}_i + \frac{\tau_n}{m_i}\sum_{j\in I_i}\frac{[p]^{n+1}_j}{{M_{\mathrm{ref}}}^2}c_{ij} = 0$$

$$\Rightarrow [\rho u]^{n+1}_i = [\hat{\rho u}]^{n+1}_i - \frac{\tau_n}{m_i}\sum_{j\in I_i}\frac{c_{ij}}{{M_{\mathrm{ref}}}^2}[p]^{n+1}_j$$

The implicit energy equation reads

$$[\mathcal{E}]_i^{n+1} + \frac{\tau_n}{m_i} \sum_{j \in I_i} \frac{\gamma}{\gamma - 1} [p]_j^{n+1} [u]_j^{n+1} \cdot c_{ij} = [\hat{\mathcal{E}}]_i^{n+1}$$

Using the equation of state (1.15) and inserting the momentum equation gives

$$\frac{[p]_i^{n+1}}{\gamma-1} + \frac{{M_{\text{ref}}}^2}{2[\rho]_i^{n+1}} ||[\rho u]_i^{n+1}||^2 + \frac{\tau_n}{m_i} \frac{\gamma}{\gamma-1} \sum_{j \in I_i} \frac{[p]_j^{n+1}}{[\rho]_j^{n+1}} \left([\hat{\rho u}]_j^{n+1} - \frac{\tau_n}{m_j} \sum_{k \in I_j} \frac{c_{jk}}{{M_{\text{ref}}}^2} [p]_k^{n+1} \right) \cdot c_{ij}$$
$$= [\hat{\mathcal{E}}]_i^{n+1}$$

Use $[\rho]_i^{n+1} = [\hat{\rho}]_i^{n+1}$ and rewrite this equation to obtain

$$\begin{aligned} &\frac{[p]_i^{n+1}}{\gamma-1} - \sum_{j \in I_i} \sum_{k \in I_j} \frac{\tau_n^2}{m_i m_j} \frac{\gamma}{\gamma-1} \frac{[p]_j^{n+1}}{[\hat{\rho}]_j^{n+1}} (c_{ij} \cdot c_{jk}) \frac{[p]_k^{n+1}}{{M_{\text{ref}}}^2} \\ &= [\hat{\mathcal{E}}]_i^{n+1} - \frac{{M_{\text{ref}}}^2}{2[\hat{\rho}]_i^{n+1}} ||[\rho u]_i^{n+1}||^2 - \sum_{j \in I_i} \frac{\tau_n}{m_i} \frac{\gamma}{\gamma-1} \frac{[p]_j^{n+1}}{[\hat{\rho}]_j^{n+1}} c_{ij} \cdot [\hat{\rho u}]_j^{n+1} \end{aligned}$$

3.3.1 The Picard iteration

The implicit equations for momentum and energy can now be solved by a picard-iteration: Set $[\rho u]_i^{n+1,0} = [\hat{\rho u}]_i^{n+1}$ and $[p]_i^{n+1,0} = [\hat{p}]_i^{n+1}$. Then iterate

$$\begin{aligned} &\frac{[p]_i^{n+1,r+1}}{\gamma-1} - \sum_{j \in I_i} \sum_{k \in I_j} \frac{\tau_n^2}{m_i m_j} \frac{\gamma}{\gamma-1} \frac{[p]_j^{n+1,r}}{[\hat{\rho}]_j^{n+1}} (c_{ij} \cdot c_{jk}) \frac{[p]_k^{n+1,r+1}}{{M_{\text{ref}}}^2} \\ &= [\hat{\mathcal{E}}]_i^{n+1} - \frac{{M_{\text{ref}}}^2}{2[\hat{\rho}]_i^{n+1}} ||[\rho u]_i^{n+1,r}||^2 - \sum_{j \in I_i} \frac{\tau_n}{m_i} \frac{\gamma}{\gamma-1} \frac{[p]_j^{n+1,r}}{[\hat{\rho}]_j^{n+1}} c_{ij} \cdot [\hat{\rho u}]_j^{n+1} \end{aligned} \tag{3.18}$$

$$[\rho u]_i^{n+1,r+1} = [\hat{\rho u}]_i^{n+1} - \frac{\tau_n}{m_i} \sum_{j \in I_i} \frac{c_{ij}}{{M_{\text{ref}}}^2} [p]_j^{n+1,r+1} \tag{3.19}$$

The solution of the implicit equations is a fix point of the function mapping data $[\rho u]_i^{n+1,r}, [p]_i^{n+1,r}$ to the next iteration $[\rho u]_i^{n+1,r+1}, [p]_i^{n+1,r+1}$. In each iteration one solves (5.2) for $[p]_i^{n+1,r+1}$ and then computes $[\rho u]_i^{n+1,r+1}$. One iterates until reasonable accuracy is obtained at some iteration $\bar{r}$. In an algorithm we then set

$$[u]_i^{n+1} \leftarrow [\rho u]_i^{n+1,\bar{r}} / [\hat{\rho}]_i^{n+1} \tag{3.20}$$

$$[p]_i^{n+1} \leftarrow [p]_i^{n+1,\bar{r}} \tag{3.21}$$

which together with (3.17) defines the new state. Typically two iterations ($\bar{r} = 2$) are sufficient.

On the one-dimensional periodic grid (3.16) with piecewise linear shape functions the Picard iteration gives the following system for the pressure if $r > 0$:

$$\left(1 + \frac{\gamma \tau_n^2}{4{M_{\text{ref}}}^2 h^2} \left(\frac{[p]_{i-1}^{n+1,r}}{[\hat{\rho}]_{i-1}^{n+1}} + \frac{[p]_{i+1}^{n+1,r}}{[\hat{\rho}]_{i+1}^{n+1}} \right)\right) [p]_i^{n+1,r+1} - \frac{\gamma \tau_n^2}{4{M_{\text{ref}}}^2 h^2} \frac{[p]_{i-1}^{n+1,r}}{[\hat{\rho}]_{i-1}^{n+1}} [p]_{i-2}^{n+1,r+1} - \frac{\gamma \tau_n^2}{4{M_{\text{ref}}}^2 h^2} \frac{[p]_{i+1}^{n+1,r}}{[\hat{\rho}]_{i+1}^{n+1}} [p]_{i+2}^{n+1,r+1}$$
$$= (\gamma - 1) \left(\mathcal{E}_i - \frac{{M_{\text{ref}}}^2}{2[\hat{\rho}]_i^{n+1}} \left([\hat{\rho u}]_i^{n+1} - \frac{\tau_n}{2{M_{\text{ref}}}^2 h} ([p]_{i+1}^{n+1,r} - [p]_{i-1}^{n+1,r}) \right)^2 \right) - \frac{\gamma \tau_n}{2h} ([\hat{u}]_{i+1}^{n+1} [p]_{i+1}^{n+1,r} - [\hat{u}]_{i-1}^{n+1} [p]_{i-1}^{n+1,r})$$

With $a_i(p) := \frac{\gamma \tau_n^2}{4{M_{\mathrm{ref}}}^2 h^2} \frac{p_i}{[\hat{\rho}]_i^{n+1}}$ define the mapping $\mathbb{R}^{\mathbb{Z}/(2N+1)\mathbb{Z}} \ni p \mapsto A(p) \in \mathbb{R}^{(2N+1)\times(2N+1)}$:

$$A_{ij} = (1 + a_{i-1} + a_{i+1})\delta_{ij} - a_{i-1}\delta_{x^{i-2},x^j} - a_{i+1}\delta_{x^{i+2},x^j}$$

where the indices i, j run from $0, ..., 2N$. If all p_i are nonnegative, then $A(p)$ is strictly diagonally dominant and the inverse exists with $||A(p)^{-1}||_\infty \le 1$. Define the right hand side

$$b(p) := (\gamma - 1)\left(\mathcal{E}_i - \frac{{M_{\mathrm{ref}}}^2}{2[\hat{\rho}]_i^{n+1}}([\hat{\rho u}]_i^{n+1} - \frac{\tau_n}{2{M_{\mathrm{ref}}}^2 h}(p_{i+1} - p_{i-1}))^2\right) - \frac{\gamma \tau_n}{2h}([\hat{u}]_{i+1}^{n+1} p_{i+1} - [\hat{u}]_{i-1}^{n+1} p_{i-1})$$

The iteration then reads

$$A([p]^{n+1,r})([p]_0^{n+1,r+1}, ..., [p]_{2N}^{n+1,r+1})^T = (b_0([p]^{n+1,r}), ..., b_{2N}([p]^{n+1,r}))^T$$

Assume that the density and pressure at time t_n are nonnegative. Letting the time step (/CFL-number) to zero, we get

$$\lim_{\tau_n \to 0} [\hat{p}]^{n+1} = [p]^n$$

$$\lim_{\tau_n \to 0} A([\hat{p}]^{n+1}) = \mathbb{I}$$

and

$$\lim_{\tau_n \to 0} [p]^{n+1,1} = [p]^n$$

For some $\epsilon \in (0, ||[p]^n||_\infty)$ define $B = \{p \in \mathbb{R}^{\mathbb{Z}/(2N+1)\mathbb{Z}} | ||p - [p]^n||_\infty \le \epsilon\}$. We also have

$$\begin{aligned}\sup_{p \in B} ||\partial_p(A(p)^{-1}b(p))||_\infty &= \sup_{p \in B} ||A(p)^{-1}(\partial_p b(p) - \partial_p A(p) A(p)^{-1} b(p))||_\infty \\ &\le \sup_{p \in B} ||\partial_p b(p) - \partial_p A(p) A(p)^{-1} b(p)||_\infty \overset{\tau_n \to 0}{\to} 0\end{aligned}$$

One can conclude that for a sufficiently small CFL-number, the Picard iteration (for $r > 0$) defines a contraction on B. Therefore, the Banach fix point theorem guarantees the convergence to the unique solution with minimal distance to $[p]^n$.
In practice we try to choose a large CFL number and will perform numerical tests.

3.3.2 The explicit Riemann problem

For the explicit part we need to determine the graph viscosity and thereby CFL-condition by computing the maximum wave speed of the Riemann problem (2.10). Here, the corresponding conservation law reads

$$\partial_t \begin{pmatrix} \rho \\ \rho u \\ \mathcal{E} \end{pmatrix} + \partial_x \begin{pmatrix} \rho(u^T\nu) \\ \rho u(u^T\nu) \\ \frac{{M_{\mathrm{ref}}}^2}{2}\rho||u||^2(u^T\nu) \end{pmatrix} = 0 \tag{3.22}$$

Defining $u^\nu := u^T\nu$ and $u^\perp := u - u^\nu\nu$, we can apply a linear operator L,

$$\begin{pmatrix} \rho \\ \rho u \\ \mathcal{E} \end{pmatrix} \mapsto L\begin{pmatrix} \rho \\ \rho u \\ \mathcal{E} \end{pmatrix} := \begin{pmatrix} \rho \\ \rho u \cdot \nu \\ \rho(u - (u\cdot\nu)\nu) \end{pmatrix}$$

to the conservation law:

$$\partial_t L \begin{pmatrix} \rho \\ \rho u \\ \mathcal{E} \end{pmatrix} + \partial_x L \begin{pmatrix} \rho(u^T\nu) \\ \rho u(u^T\nu) \\ \frac{M_{\mathrm{ref}}{}^2}{2}\rho||u||^2(u^T\nu) \end{pmatrix} = 0$$

$$\Rightarrow \partial_t \begin{pmatrix} \rho \\ \rho u^\nu \\ \rho u^\perp \end{pmatrix} + \partial_x \begin{pmatrix} \rho u^\nu \\ \rho(u^\nu)^2 \\ (\rho u^\perp)u^\nu \end{pmatrix} = 0 \tag{3.23}$$

The (weak) solutions of the linearly transformed equation will agree with the original one in the given variables. Next, extract the first two components of above equation:

$$\partial_t \begin{pmatrix} \rho \\ \rho u^\nu \end{pmatrix} + \partial_x \begin{pmatrix} \rho u^\nu \\ \rho(u^\nu)^2 \end{pmatrix} = 0 \tag{3.24}$$

Equation (3.24) is a closed system, therefore it determines ρ and u^ν in the solution of the Riemann problem to (3.22). Let us now investigate the solution of the Riemann problem

$$(\rho, u^\nu)^T(x,0) = \begin{cases} (\rho_l, u_l^\nu)^T \ , \text{ if } \ x < 0 \\ (\rho_r, u_r^\nu)^T \ , \text{ if } \ x > 0 \end{cases}$$

to (3.24). We will assume nonnegative densities $\rho_l \geq 0$, $\rho_r \geq 0$.

Self similarity

Let q^ε be a solution to $\partial_t q^\varepsilon + \partial_x f(q^\varepsilon) = \varepsilon\partial_{xx}q^\varepsilon$ for some conservation law and take $a > 0$. Assume q^ε sufficiently smooth, then we find that $q^{a,\varepsilon}(x,t) := q^{a\varepsilon}(ax, at)$ is again a solution:

$$\partial_t q^{a,\varepsilon}(x,t) + \partial_x f(q^{a,\varepsilon}(x,t)) = a(\partial_t q^{a\varepsilon}(ax,at) + \partial_x f(q^{a\varepsilon}(ax,at))) = a^2\varepsilon\partial_{xx}q^{a\varepsilon}(ax,at) = \varepsilon\partial_{xx}q^{a,\varepsilon}(x,t)$$

In the case of a Riemann problem, the rescaling does not change the initial data at $t = 0$. If we assume unique solutions and that the vanishing viscosity limit $q, (x,t) \mapsto q(x,t)$ exists, we therefore get

$$q(x,t) = \lim_{\varepsilon\searrow 0} q^{a,\varepsilon}(x,t) = \lim_{\varepsilon\searrow 0} q^{a\varepsilon}(ax,at) = \lim_{\varepsilon\searrow 0} q^{\varepsilon}(ax,at) = q(ax,at)$$

Those solutions are called self similar. One can now represent the solution as a function of only one real variable $\xi = x/t$: $q(x,t) =: q^*(x/t)$. It is constructed as a sequence of waves connecting the left state to the right state. For weak solutions those are *rarefaction waves*, which satisfy the classical/strong form of the partial differential equation, and jump discontinuities.

Rarefactions

For rarefactions connecting a state at some ξ_1 to a state at some $\xi_2 \neq \xi_1$, differentiation rules can be applied to the strong formulation

$$0 = \partial_t q(x,t) + \partial_x f(q(x,t)) = -\frac{x}{t^2}\partial_\xi q^*(x/t) + \frac{1}{t}Df(q^*(x/t))\partial_\xi q^*(x/t)$$

$$\Rightarrow Df(q^*(x/t))\partial_\xi q^*(x/t) = \frac{x}{t}\partial_\xi q^*(x/t) \tag{3.25}$$

We find that the rarefaction wave must move along a curve (parameterized by ξ) whose velocity vector is an eigenvector of $Df(q^*(\xi))$ corresponding to the eigenvalue ξ.

For (3.24) with the notation $q^* = (\rho, \rho u^\nu)^T$, it is

$$Df((\rho, \rho u^\nu)^T) = \begin{pmatrix} 0 & 1 \\ -(u^\nu)^2 & 2u^\nu \end{pmatrix}$$

This matrix only has the eigenvalue u^ν with one-dimensional eigenspace $span\{(1, u^\nu)^T\}$. From this we compute

$$\begin{aligned} & u^\nu \partial_\xi \rho = \partial_\xi(\rho u^\nu) = u^\nu \partial_\xi \rho + \rho \partial_\xi u^\nu \\ & \Rightarrow \rho \partial_\xi u^\nu = 0 \end{aligned}$$

Except for the trivial case of rarefactions only consisting of a constant state, there has to be a nonempty open subset B of the domain $[\xi_1, \xi_2]$ of the rarefaction wave where $\rho \neq 0$ and $\partial_\xi \rho \neq 0$. Here, $\partial_\xi u^\nu = 0$ has to hold. Furthermore $\partial_\xi q$ is not the zero vector and hence

$$\forall \xi \in B : \frac{x}{t} = \xi = u^\nu(\xi) = const.$$

must be satisfied due to (3.25). This condition clearly cannot be met, so we conclude that the solution we seek contains no rarefaction waves (apart from constant states that always satisfy any conservation law ...)

Jump discontinuities

Another form of waves in weak solutions can be represented as jump discontinuities. Here one has a differentiable function $c : [t_1, t_2] \to \mathbb{R}$ into space such that the solution is smooth on both sides of the graph C of c with continuous extension to it. The jump happens across C. Using the integral form of the conservation law (1.27) we can write for some $t^* \in (t_1, t_2)$:

$$\lim_{\delta \to 0} \frac{d}{dt} \int_{c(t^*)-\delta}^{c(t^*)+\delta} q(x,t)\, dx \Bigg|_{t=t^*} = \lim_{\delta \to 0} \left(f(q(c(t^*) - \delta, t^*)) - f(q(c(t^*) + \delta, t^*)) \right) \tag{3.26}$$

Defining the left state $q_\leftarrow := \lim_{\delta \nearrow 0} q(c(t^*) + \delta, t^*)$ and right state $q_\rightarrow := \lim_{\delta \searrow 0} q(c(t^*) + \delta, t^*)$ and the speed of the jump $s := \frac{d}{dt} c(t^*)$ one can show that above expression yields the so called **Rankine-Hugoniot jump condition**

$$s(q_\leftarrow - q_\rightarrow) = f(q_\leftarrow) - f(q_\rightarrow) \tag{3.27}$$

Refer to [4], pp. 168-172, for a derivation from the weak form. Applying (3.27) to (3.24) gives the equation

$$s \begin{pmatrix} \rho_\leftarrow - \rho_\rightarrow \\ \rho_\leftarrow u^\nu_\leftarrow - \rho_\rightarrow u^\nu_\rightarrow \end{pmatrix} = \begin{pmatrix} \rho_\leftarrow u^\nu_\leftarrow - \rho_\rightarrow u^\nu_\rightarrow \\ \rho_\leftarrow (u^\nu_\leftarrow)^2 - \rho_\rightarrow (u^\nu_\rightarrow)^2 \end{pmatrix}$$

from which follows

$$\begin{aligned} & (\rho_\leftarrow u^\nu_\leftarrow - \rho_\rightarrow u^\nu_\rightarrow)^2 = s(\rho_\leftarrow - \rho_\rightarrow)(\rho_\leftarrow u^\nu_\leftarrow - \rho_\rightarrow u^\nu_\rightarrow) = (\rho_\leftarrow - \rho_\rightarrow)(\rho_\leftarrow (u^\nu_\leftarrow)^2 - \rho_\rightarrow (u^\nu_\rightarrow)^2) \\ & \Rightarrow (\rho_\leftarrow \rho_\rightarrow)(u^\nu_\leftarrow - u^\nu_\rightarrow)^2 = 0 \end{aligned}$$

If we need to connect states $0 < \rho_l$, $0 < \rho_r \neq \rho_l$, $u^\nu_l > u^\nu_r$, we now can try a sequence of jumps. The leftmost jump must satisfy $\rho_l \neq \rho_\rightarrow$, since otherwise also $u^\nu_l = u^\nu_\rightarrow$, i.e. there would be no jump. If $\rho_\rightarrow \neq 0$, $u^\nu_l = u^\nu_\rightarrow$ must hold and the speed of the jump has to be $s = u^\nu_l$. If $\rho_\rightarrow = 0$, the speed must again be $s = u^\nu_l$. Therefore the jump of the leftmost jump will be u^ν_l.

Analogously, the rightmost jump would have to move at the speed $s = u^\nu_r < u^\nu_l$, i.e. the speed of the left jump is greater than that of the right. This is a contradiction showing that we cannot in general construct the Riemann solution by sequence of jumps. The reason is that the solution can contain point measures arising from the fact that the density is transported by the flow which moves into a discontinuity (of u^ν). In the full Euler equations the pressure terms cause the formation of intermediate states that contain the mass. We can extend the set of possible solutions by introducing Dirac measures.

Jump-point-measures

Denote the Dirac measure for an open set B at a point x^* like a function:
$\int_B \delta(x-x^*)\,dx = \chi_B(x^*)$. The point-measure is a point of mass $\mu(t)$ moving at speed s having a momentum $\mu(t)s$. It is represented as $\begin{pmatrix} \mu(t) \\ \mu(t)s \end{pmatrix} \delta(x - st)$. The solution shall be the sum of a sequence of point-measures and an integrable function q^I consisting of a sequence of jumps. Combine both concepts to write the solution as a sequence of "jump-point-measures" of magnitude $(\mu(t), \mu(t)s)^T$ and left state $(\rho, \rho u^\nu)^T_\leftarrow$, right state $(\rho, \rho u^\nu)^T_\rightarrow$.
We extend the integral formulation of the conservation law for a jump (3.26) by the contribution of the point measure

$$\lim_{\delta\to 0} \frac{d}{dt}\left[\int_{c(t^*)-\delta}^{c(t^*)+\delta} q^I(x,t)\,dx + \begin{pmatrix} \mu(t) \\ \mu(t)s \end{pmatrix}\right]\Bigg|_{t=t^*} = \lim_{\delta\to 0}\left(f(q(c(t^*)-\delta,t^*)) - f(q(c(t^*)+\delta,t^*))\right)$$

yielding a condition for a jump-point-measure at time t:

$$(q_\leftarrow - q_\rightarrow)s + \begin{pmatrix} \dot\mu(t) \\ \dot\mu(t)s \end{pmatrix} = f(q_\leftarrow) - f(q_\rightarrow)$$

of which the Rankine-Hugoniot condition is a special case with $\mu \equiv 0$. Since the left and right states as well as the wave speed are constant in time in our case (from self similarity), the time derivative $\dot\mu$ must also be constant.
Writing out the components of q and f then produces the following system of equations:

$$\begin{pmatrix} (\rho_\leftarrow - \rho_\rightarrow)s + \dot\mu \\ (\rho_\leftarrow u^\nu_\leftarrow - \rho_\rightarrow u^\nu_\rightarrow + \dot\mu)s \end{pmatrix} = \begin{pmatrix} \rho_\leftarrow u^\nu_\leftarrow - \rho_\rightarrow u^\nu_\rightarrow \\ \rho_\leftarrow (u^\nu_\leftarrow)^2 - \rho_\rightarrow (u^\nu_\rightarrow)^2 \end{pmatrix} \tag{3.28}$$

It has to hold that

$$(\rho_\leftarrow u^\nu_\leftarrow - \rho_\rightarrow u^\nu_\rightarrow - \dot\mu)(\rho_\leftarrow u^\nu_\leftarrow - \rho_\rightarrow u^\nu_\rightarrow + \dot\mu) = (\rho_\leftarrow (u^\nu_\leftarrow)^2 - \rho_\rightarrow (u^\nu_\rightarrow)^2)(\rho_\leftarrow - \rho_\rightarrow) \tag{3.29}$$

from which one obtains

$$\dot\mu^2 = \rho_\leftarrow \rho_\rightarrow (u^\nu_\leftarrow - u^\nu_\rightarrow)^2 \tag{3.30}$$

The density in the differential equations with viscosity won't become negative, so let's require $\mu \geq 0$ allowing us to write

$$\dot\mu = \sqrt{\rho_\leftarrow \rho_\rightarrow (u^\nu_\leftarrow - u^\nu_\rightarrow)^2} \tag{3.31}$$

Distinguish two cases:
$\underline{\rho_\leftarrow = \rho_\rightarrow =: \rho_\leftrightarrow > 0}$:
Here one finds $\dot\mu = \rho_\leftrightarrow (u^\nu_\leftarrow - u^\nu_\rightarrow)$ from the first row in (3.29). To have a jump/point-measure, $u^\nu_\leftarrow \neq u^\nu_\rightarrow$ has to be true. The second row of (3.29) then gives

$$2(u^\nu_\leftarrow - u^\nu_\rightarrow)\rho_\leftrightarrow s = ((u^\nu_\leftarrow)^2 - (u^\nu_\rightarrow)^2)\rho_\leftrightarrow$$

from which the wave speed can be extracted as

$$s = \frac{u^\nu_\leftarrow + u^\nu_\rightarrow}{2} \tag{3.32}$$

$\underline{\rho_\leftarrow \neq \rho_\rightarrow}$:

In the case that the densities are not equal, the wave speed can be computed from the first row in (3.29) as

$$s = \frac{\rho_\leftarrow u^\nu_\leftarrow - \rho_\rightarrow u^\nu_\rightarrow - \dot{\mu}}{\rho_\leftarrow - \rho_\rightarrow} \tag{3.33}$$

The case $u^\nu_l \leq u^\nu_r$

In subdomains where q solves the conservation law in the strong sense, it can be written

$$\begin{aligned} &\partial_t \rho + u^\nu \partial_x \rho + \rho \partial_x u^\nu = 0 \\ &u^\nu \partial_t \rho + \rho \partial_t u^\nu + (u^\nu)^2 \partial_x \rho + 2\rho u^\nu \partial_x u^\nu = 0 \end{aligned}$$

It follows

$$\partial_t u^\nu + u^\nu \partial_x u^\nu = 0$$

which is the scalar *Burgers equation.* Let $x : t \mapsto x(t)$ such that $\dot{x}(t) = u^\nu(x(t), t)$. Then

$$\frac{d}{dt} u^\nu(x(t), t) = \partial_x u^\nu(x(t), t)\dot{x}(t) + \partial_t u^\nu(x(t), t) = \partial_t u^\nu(x(t), t) + u^\nu(x(t), t)\partial_x u^\nu(x(t), t) = 0$$

I.e. u^ν is constant along the curve $t \mapsto (x(t), t)^T$, called a **characteristic curve**. Because their speed is equal to u^ν, those curves are straight lines in our case. We can use this fact to solve the Riemann problem. If we apply a little diffusion to the initial data to smoothen it, we end up with monotonically increasing u^ν and a smooth solution exists for $t > 0$. This solution is found by uniquely tracing back the characteristics to the initial data. Going by the notion of stability under perturbations, the solution to the Burgers equation we choose is

$$(u^\nu)^*(\xi) = \begin{cases} u^\nu_l & \text{, if } \xi \leq u^\nu_l \\ u^\nu_r & \text{, if } \xi \geq u^\nu_r \\ \xi & \text{, else} \end{cases} \tag{3.34}$$

By the continuum equation, the mass is just transported with the flow u^ν, so the density is

$$\rho^*(\xi) = \begin{cases} \rho_l & \text{, if } \xi \leq u^\nu_l \\ \rho_r & \text{, if } \xi \geq u^\nu_r \\ 0 & \text{, else} \end{cases} \tag{3.35}$$

The solution in conservative variables consists of up to 2 jumps. Of course they satisfy the conditions for jump-point-measures we developed (in this case it's just the Rankine-Hugoniot condition).

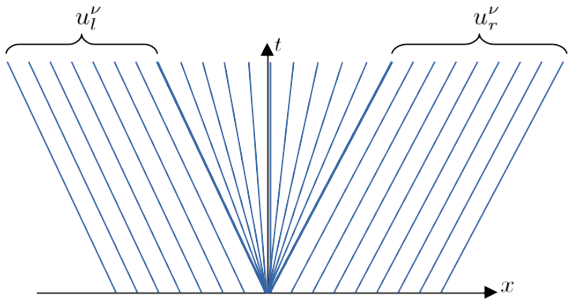


Figure 3.2: Characteristics for Burgers equation (rarefaction of u^ν)

The case $u_l^\nu > u_r^\nu$

In this case the characteristics run into each other, so there will be a jump in u^ν. The mass being transported along the characteristics creates a point-mass. We can construct the solution with the help of (3.31), (3.32) and (3.33) as a jump-point-measure with

$$\dot{\mu} = \sqrt{\rho_l \rho_r}(u_l^\nu - u_r^\nu) \tag{3.36}$$

$$s = \begin{cases} \frac{u_l^\nu + u_r^\nu}{2} & \text{, if } \rho_l = \rho_r \\ \frac{\rho_l u_l^\nu - \rho_r u_r^\nu - \dot{\mu}}{\rho_l - \rho_r} & \text{, else} \end{cases} \tag{3.37}$$

It can easily be verified that s is a convex combination of u_l^ν and u_r^ν.

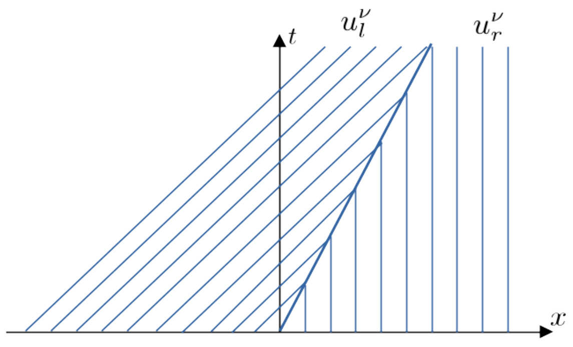


Figure 3.3: Characteristics for Burgers equation (shock of u^ν)

If this solution is smoothened, the information of u^ν and the mass still moves with the characteristics into the (now smoothened) jump-point-measure, renewing it, so we can regard that solution as reasonable.

The variable $\rho u^\perp$

The equation for $u^\perp$ reads (see (3.23))

$$\partial_t \rho u^\perp + \partial_x((\rho u^\perp)u^\nu) = 0$$

which is a continuum equation for $\rho u^\perp$ since $\rho u^\perp$ and u^ν are independent variables. The momentum $\rho u^\perp$ gets transported with the velocity u^ν just like the mass density.
In the case $\underline{u_l^\nu \leq u_r^\nu}$ the solution is therefore found in analogy to the solution for ρ as

$$(\rho u^\perp)^*(\xi) = \begin{cases} \rho_l u_l^\perp & \text{, if } \xi \leq u_l^\nu \\ \rho_r u_r^\perp & \text{, if } \xi \geq u_r^\nu \\ 0 & \text{, else} \end{cases} \tag{3.38}$$

For the case $\underline{u_l^\nu > u_r^\nu}$, there is a single jump in u^ν with velocity given by (3.37) and the mass moving into it. $u^\perp$ will therefore also be constant left and right of the jump and the solution is again constructed as a single jump-point-measure with momentum $m^\perp$. The condition here reads

$$((\rho u^\perp)_l - (\rho u^\perp)_r)s + \dot{m}^\perp = (\rho u^\perp)_l u_l^\nu - (\rho u^\perp)_r u_r^\nu$$

which yields

$$\dot{m}^\perp = (\rho u^\perp)_l u_l^\nu - (\rho u^\perp)_r u_r^\nu + ((\rho u^\perp)_r - (\rho u^\perp)_l)s \tag{3.39}$$

The variable $\mathcal{E}$

The energy equation was

$$\partial_t \mathcal{E} + \partial_x \left(\frac{{M_{\text{ref}}}^2}{2} \rho ||u||^2 u^\nu \right) = 0$$

Where the solution is smooth, one can infer $\partial_t p = 0$, leading to the conclusion that we should construct the solution such that p only varies at points where ρ or ρu experience a jump-point-measure, and possibly at $\{x = 0\}$. Construct now a solution without pressure point-measures, that is: All energy point-measures have the intensity of the kinetic energy $\frac{{M_{\text{ref}}}^2}{2}\dot{\mu} s^2$. We find the possible solutions by looking at the condition

$$(\mathcal{E}_l - \mathcal{E}_r)s + \frac{{M_{\text{ref}}}^2}{2}\dot{\mu} s^2 = \frac{{M_{\text{ref}}}^2}{2}(\rho_l ||u_l||^2 u_l^\nu - \rho_r ||u_r||^2 u_r^\nu)$$

for jump-point-measures of the energy component. It is easy to see that in an interval $\{\xi_1 < \xi < \xi_2\}$ of $\xi = x/t$ where ρ and ρu are constant, so will be $\mathcal{E}$, except for possibly a jump at $\xi = 0$. The density- and momentum components of the solution have already been determined to consist only of certain jumps and point measures for which we know the speed s.
In the case $u_l^\nu \leq u_r^\nu$ one finds that the pressure is constant in time.
In the case $u_l^\nu > u_r^\nu$ one obtains a jump along the known jump-point-measure with

$$p_\rightarrow - p_\leftarrow = \frac{{M_{\text{ref}}}^2}{2}(\gamma - 1)\left(\dot{\mu} s + \frac{\rho_\rightarrow ||u_\rightarrow||^2 (u_\rightarrow^\nu - s) - \rho_\leftarrow ||u_\leftarrow||^2 (u_\leftarrow^\nu - s)}{s} \right)$$

and another possible jump with speed 0.

Summary

Figures (3.4) and (3.5) graphically visualize the complete solution we have constructed in different settings.

Maximum wave speed

The maximum wave speed for the explicit part of the splitting is easily obtained:

$$\lambda_{\max}(q_l, q_r, \nu) = \begin{cases} \max(|u_l^\nu|, |u_r^\nu|) & \text{, if } u_l^\nu \le u_r^\nu \\ \frac{|u_l^\nu + u_r^\nu|}{2} & \text{, if } u_l^\nu > u_r^\nu \wedge \rho_l = \rho_r \\ \left|\frac{\rho_l u_l^\nu - \rho_r u_r^\nu - \sqrt{\rho_l \rho_r}(u_l^\nu - u_r^\nu)}{\rho_l - \rho_r}\right| & \text{, if } u_l^\nu > u_r^\nu \wedge \rho_l \ne \rho_r \end{cases} \tag{3.40}$$

Invariant domains

It is easy to see that $\{(\rho, \rho u, \mathcal{E})^T | \rho \ge 0\}$ is an invariant domain.
Another invariant domain is $\{(\rho, \rho u, \mathcal{E})^T | \rho \ge 0 \wedge p \ge 0\}$.

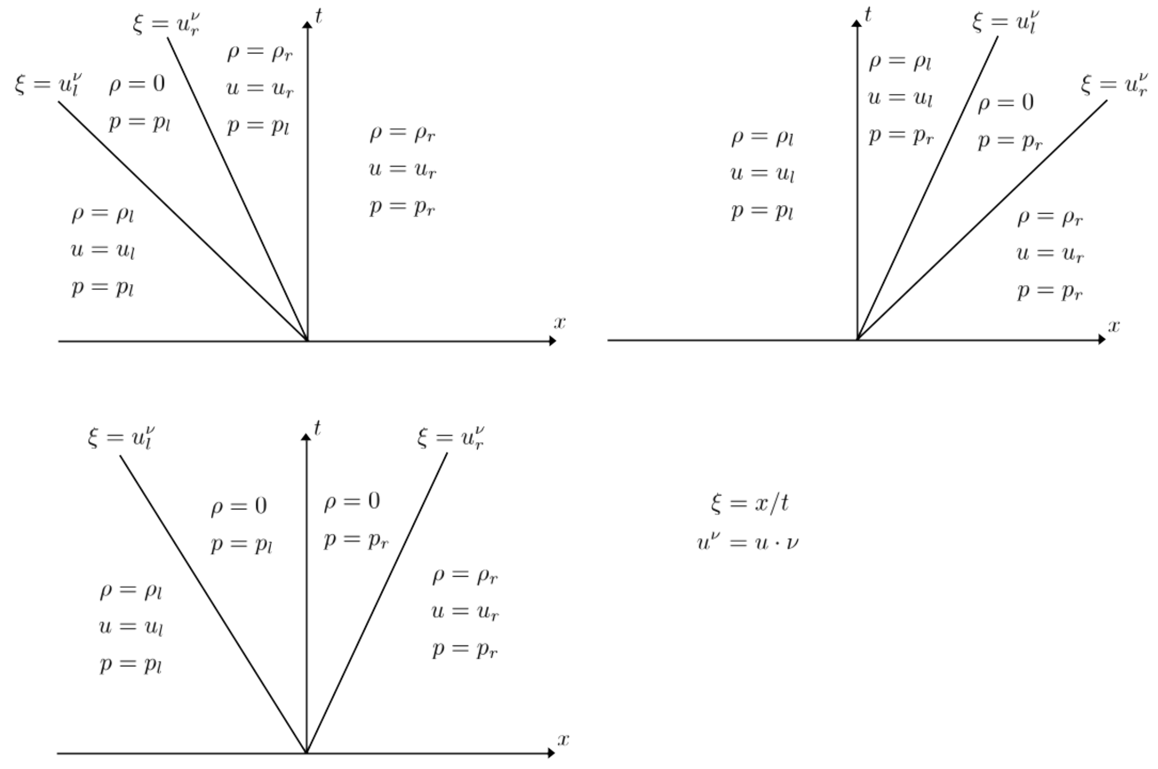
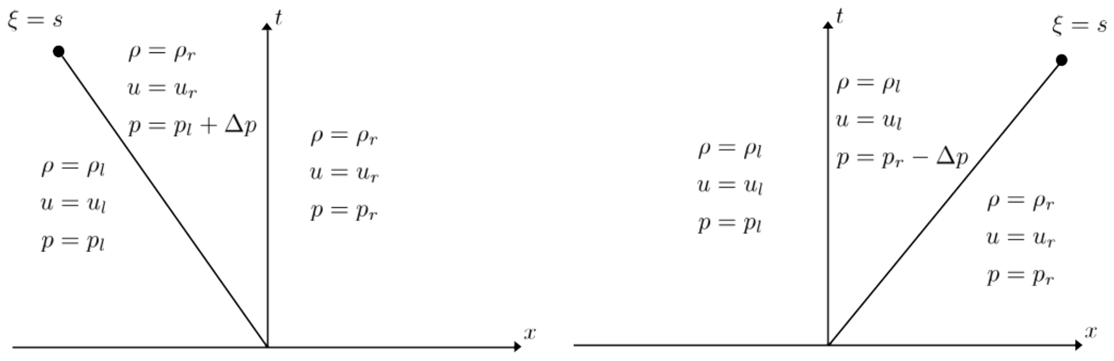


Figure 3.4: Solution of the explicit Riemann problem in the case $u_l^\nu \le u_r^\nu$



Point measure at $\xi = s$ with intensity $\begin{pmatrix} \dot{\mu} \\ \dot{\mu} s \nu + \dot{m}^{\perp} \\ \frac{{M_{\text{ref}}}^2}{2} \dot{\mu} s^2 \end{pmatrix} t$

$$\xi = x/t \qquad \dot{\mu} = \sqrt{\rho_l \rho_r}(u_l^\nu - u_r^\nu)$$

$$u^\nu = u \cdot \nu$$

$$u^\perp = u - u^\nu \nu \qquad s = \begin{cases} \frac{u_l^\nu + u_r^\nu}{2} & \text{, if } \rho_l = \rho_r \\ \frac{\rho_l u_l^\nu - \rho_r u_r^\nu - \dot{\mu}}{\rho_l - \rho_r} & \text{, else} \end{cases}$$

$$\dot{m}^\perp = (\rho u^\perp)_l u_l^\nu - (\rho u^\perp)_r u_r^\nu + ((\rho u^\perp)_r - (\rho u^\perp)_l) s$$

Δp only needs to be determined if $s \neq 0$: $\Delta p = \frac{{M_{\text{ref}}}^2}{2}(\gamma - 1)\left(\dot{\mu} s + \frac{\rho_r ||u_r||^2 (u_r^\nu - s) - \rho_l ||u_l||^2 (u_l^\nu - s)}{s}\right)$

Figure 3.5: Solution of the explicit Riemann problem in the case $u_l^\nu > u_r^\nu$

3.3.3 The complete scheme in pseudo-code

The mesh-generation and the computation of the coefficients m_i and c_{ij} is performed offline. Component/variable extraction functions for density, momentum (density), pressure etc. are assumed to be given. The scheme is then defined by an update function evolving given numerical data Q by one time step with an (explicit) CFL-number CFL and $\bar{r}$ Picard iterations:

```
function UPDATE($Q$,CFL,$\bar{r}$)
    $d \leftarrow 0 \in \mathbb{R}^{|I|\times|I|}$
▷ Compute the $d_{ij}$
    for all $i \in I$ do
        for all $j \in I_i \setminus \{i\}$ do
            $\rho_l \leftarrow \rho(Q_i)$
            $\rho_r \leftarrow \rho(Q_j)$
            $u_l^\nu \leftarrow u(Q_i) \cdot n_{ij}$
            $u_r^\nu \leftarrow u(Q_j) \cdot n_{ij}$
            if $u_l^\nu \leq u_r^\nu$ then
                $\lambda_{\max} \leftarrow \max(|u_l^\nu|, |u_r^\nu|)$
            else if $\rho_l \neq \rho_r$ then
                $\lambda_{\max} \leftarrow |(\rho_l u_l^\nu - \rho_r u_r^\nu - \sqrt{\rho_l \rho_r}(u_l^\nu - u_r^\nu))/(\rho_l - \rho_r)|$
            else
                $\lambda_{\max} \leftarrow |u_l^\nu + u_r^\nu|/2$
            end if
            $d_{ij} \leftarrow \lambda_{\max}||c_{ij}||$
            $d_{ii} \leftarrow d_{ii} - d_{ij}$
        end for
    end for
    $\tau \leftarrow \mathrm{CFL} \cdot \min_{i\in I}(m_i/(-2d_{ii}))$        ▷ Compute the time step size
    $\hat{Q} \leftarrow Q$
▷ Compute the intermediate state (explicit step)
    for all $i \in I$ do
        for all $j \in I_i$ do
            $\hat{Q}_i \leftarrow \hat{Q}_i + \frac{\tau}{m_i}\left(Q_j d_{ij} - f_{\mathrm{ex}}(Q_j)c_{ij}\right)$
        end for
    end for
▷ Perform the implicit step
    $[\rho u] \leftarrow \mathrm{momentum}(\hat{Q})$
    $[p] \leftarrow p(\hat{Q})$
    $A \leftarrow 0 \in \mathbb{R}^{|I|\times|I|}$
    $b \leftarrow 0 \in \mathbb{R}^{|I|}$
    for $r = 1...\bar{r}$ do
▷ Compute the implicit system matrix $A$ and right hand side $b$
        for all $i \in I$ do
            $A_{ii} \leftarrow 1/(\gamma - 1)$
            $b_i \leftarrow \mathcal{E}(\hat{Q}_i) - \frac{{M_{\mathrm{ref}}}^2}{2\rho(\hat{Q}_i)}||[\rho u]_i||^2$
            for all $j \in I_i$ do
                $b_i \leftarrow b_i - \frac{\tau}{m_i}\frac{\gamma}{\gamma-1}\frac{[p]_j}{\rho(\hat{Q}_j)}c_{ij} \cdot \mathrm{momentum}(\hat{Q}_j)$
                for all $k \in I_j$ do
```

```
                    $A_{ik} \leftarrow A_{ik} - \frac{\tau^2}{m_i m_j} \frac{\gamma}{\gamma-1} \frac{[p]_j}{\rho(\hat{Q}_j)} \frac{c_{ij} \cdot c_{jk}}{M_{\text{ref}}{}^2}$
                end for
            end for
        end for
▷ Perform Picard update
        $[p] \leftarrow$ SOLVE($A$,$b$)                                ▷ Set $[p]$ the solution to $A[p] = b$
        for all $i \in I$ do
            $[\rho u]_i \leftarrow \text{momentum}(\hat{Q}_i)$
            for all $j \in I_i$ do
                $[\rho u]_i \leftarrow [\rho u]_i - \frac{\tau}{m_i} \frac{[p]_j}{M_{\text{ref}}{}^2} c_{ij}$
            end for
        end for
    end for
▷ Finally update the solution
    for all $i \in I$ do
        $Q_i \leftarrow \left(\rho(\hat{Q}_i), [\rho u]_i, M_{\text{ref}}{}^2/(2\rho(\hat{Q}_i))||[\rho u]_i||^2 + [p]_i/(\gamma-1)\right)^T$
    end for
    return $(Q, \tau)$
end function
```

Chapter 4

Properties of the scheme

In this chapter some properties of our scheme are investigated for a periodic domain (the torus).

4.1 Conservation

The total mass, momentum and energy of the solution is given through

$$\int_\Omega q_h^n(x)\,dx = \sum_{i\in I} Q_i^n \int_\Omega \varphi_i(x)\,dx = \sum_{i\in I} Q_i^n m_i$$

The explicit step does not change the total conserved quantities:

$$\begin{aligned}
\int_\Omega \hat{q}_h^{n+1}(x)\,dx &= \sum_{i\in I} m_i \hat{Q}_i^{n+1}\\
&= \sum_{i\in I}\left(m_i Q_i^n + \tau_n \sum_{j\in I}\left(Q_j^n d_{ij} - f^{\mathrm{ex}}(Q_j^n)c_{ij}\right)\right)\\
&= \sum_{i\in I} m_i Q_i^n + \tau_n\left(\sum_{i\in I}\sum_{j\in I\setminus\{i\}}(Q_j^n - Q_i^n)d_{ij} - \sum_{j\in I} f^{\mathrm{ex}}(Q_j^n)\sum_{i\in I} c_{ij}\right)\\
&= \int_\Omega q_h^n(x)\,dx + \tau_n \sum_{\substack{i,j\in I\\ i<j}} (Q_j^n - Q_i^n)(d_{ij} - d_{ji}) = \int_\Omega q_h^n(x)\,dx
\end{aligned}$$

Similarly for the implicit step

$$\begin{aligned}
\int_\Omega q_h^{n+1}(x)\,dx &= \sum_{i\in I} m_i Q_i^{n+1}\\
&= \sum_{i\in I}\left(m_i \hat{Q}_i^n - \tau_n \sum_{j\in I} f^{\mathrm{im}}(Q_j^{n+1})c_{ij}\right)\\
&= \sum_{i\in I} m_i \hat{Q}_i^n - \tau_n \sum_{j\in I} f^{\mathrm{im}}(Q_j^{n+1})\sum_{i\in I} c_{ij} = \int_\Omega \hat{q}_h^{n+1}(x)\,dx
\end{aligned}$$

In total we get (by induction):

$$\forall n \in \mathbb{N} : \int_\Omega q_h^n(x)\,dx = \int_\Omega q_h^0(x)\,dx$$

I.e. the scheme conserves the quantities mass, momentum and energy just like the exact solution. This property is important to produce correct shock speeds.

4.2 Consistency

We will formally proof the consistency of the scheme in m space dimensions, i.e. the spatial domain is $\Omega = \mathbb{T}^m$. Let $q : \Omega \times [0,T] \to \mathbb{R}^{m+2}, (x,t) \mapsto q(x,t)$ be a sufficiently smooth (strong) solution of the scaled Euler equations. Due to Ω being compact, q and its derivatives can be assumed to be uniformly bounded where ever required. Consider a sequence of meshes parameterized by the mesh size parameter: $\{h_1, h_2, ...\} \ni h \mapsto \mathcal{T}_h$ such that

$$\sup_{i\in I} \sup_{j\in I_i} ||x^j - x^i|| = \mathcal{O}(h) \tag{4.1}$$

Given the shape functions map into $[0,1]$ this implies

$$\sup_{i\in I} m_i = \mathcal{O}(h^m)$$

We also require

$$\sup_{i\in I} \sup_{j\in I_i} ||c_{ij}|| = \mathcal{O}(h^{m-1}) \tag{4.2}$$

and

$$\sup_{i\in I} \sup_{j\in I_i} \frac{||c_{ij}||}{m_i} = \mathcal{O}(h^{-1}) \tag{4.3}$$

For cartesian grids with m-linear Lagrangian shape functions it can be shown that

$$\forall \nu \in S : \sum_{j\in I_i} \frac{1}{m_i}(x^j - x^i)|c_{ij} \cdot \nu| = 0$$

This equality still holds after linear transformation of the elements. We will only require

$$\forall \nu \in S : \sup_{i\in I} \left\| \sum_{j\in I_i} \frac{1}{m_i}(x^j - x^i)|c_{ij} \cdot \nu| \right\| = \mathcal{O}(h) \tag{4.4}$$

For a mesh with finite elements that can exactly replicate linear functions it holds that

$$\begin{aligned} e_k^T &= \frac{1}{m_i}\int_\Omega \varphi_i(x) e_k\,dx = \frac{1}{m_i}\int_\Omega \varphi_i(x) \nabla \underbrace{\sum_{j\in I} x_k^j \varphi_j(x)}_{=x_k}\,dx = \frac{1}{m_i}\sum_{j\in I} x_k^j \int_\Omega \varphi_i(x)\nabla\varphi_j(x)\,dx \\ &= \frac{1}{m_i}\sum_{j\in I} x_k^j c_{ij} = \frac{1}{m_i}\sum_{j\in I_i}(x^j - x^i)_k c_{ij} \\ &\Rightarrow \frac{1}{m_i}\sum_{j\in I_i} c_{ij}(x^j - x^i)^T = \mathbb{I} \end{aligned}$$

where $\mathbb{I}$ denotes the identity matrix. We require only

$$\sup_{i\in I} \left\| \frac{1}{m_i}\sum_{j\in I_i} c_{ij}(x^j - x^i)^T - \mathbb{I} \right\| = \mathcal{O}(h) \tag{4.5}$$

Explicit step

Set the initial numerical data $Q_i^0 = q(x^i, 0)$. For the maximum wave speed in the explicit part it holds that

$$|\lambda_{\max}(q_l, q_r, \nu) - \lambda_{\max}(q_l, q_l, \nu)| \leq |u_r^\nu - u_l^\nu|$$

and so for the graph viscosity:

$$\begin{aligned} d_{ij}^0 &= \lambda_{\max}(Q_i^0, Q_j^0, n_{ij})||c_{ij}|| \\ &= \lambda_{\max}(Q_i^0, Q_i^0, n_{ij})||c_{ij}|| + \mathcal{O}(|\lambda_{\max}(Q_i^0, Q_j^0, n_{ij}) - \lambda_{\max}(Q_i^0, Q_i^0, n_{ij})|||c_{ij||}) \\ &= |[u]_i^0 \cdot c_{ij}| + \mathcal{O}(h)||c_{ij}|| \end{aligned} \tag{4.6}$$

(Above equation is understood to hold uniformly for sequences of the i, j as $h \to 0$)
Using this we write

$$\begin{aligned} \frac{1}{m_i}\sum j \in I_i Q_j^0 d_{ij}^0 &= \frac{1}{m_i}\sum_{j\in I_i}(Q_i^0 + \underbrace{\partial_x q(x^i,0)(x^j - x^i)}_{\mathcal{O}(h)} + \mathcal{O}(h^2))d_{ij}^0 \\ &= \frac{1}{m_i}Q_i^0 \underbrace{\sum_{j\in I_i} d_{ij}^0}_{=0} + \frac{1}{m_i}\sum_{j\in I_i}\partial_x q(x^i,0)(x^j - x^i)d_{ij}^0 + \mathcal{O}(h) \\ &= \frac{1}{m_i}\partial_x q(x^i,0)\sum_{j\in I_i}(x^j - x^i)|[u]_i^0 \cdot c_{ij}| + \mathcal{O}(h) \overset{(4.4)}{=} \mathcal{O}(h) \end{aligned} \tag{4.7}$$

For the divergence estimate we obtain

$$\begin{aligned} \frac{1}{m_i}\sum_{j\in I_i} f^{\text{ex}}(Q_j^0)c_{ij} &= \frac{1}{m_i}\sum_{j\in I_i}\left(f^{\text{ex}}(Q_i^0) + \sum_{k=1}^m \partial_{x_k} f^{\text{ex}}(q(x^i,0))(x^j - x^i)_k + \mathcal{O}(h^2)\right)c_{ij} \\ &= \sum_{k=1}^m \partial_{x_k} f^{\text{ex}}(q(x^i,0))\underbrace{\frac{1}{m_i}\sum_{j\in I_i} c_{ij}(x^j - x^i)_k}_{=e_k+\mathcal{O}(h)} + \mathcal{O}(h) = div f^{\text{ex}}(q(x^i,0)) + \mathcal{O}(h) \end{aligned} \tag{4.8}$$

In total we get for the explicit update

$$\begin{aligned} \frac{\hat{Q}_i^1 - Q_i^0}{\tau_0} &= \frac{1}{m_i}\left(\sum_{j\in I_i} Q_j^0 d_{ij}^0 - \sum_{j\in I_i} f^{\text{ex}}(Q_j^0)c_{ij}\right) \\ &= \mathcal{O}(h) - div f^{\text{ex}}(q(x^i,0)) \end{aligned} \tag{4.9}$$

Implicit step

Setting $Q_i^* = q(x^i, \tau_0)$, one can approximate the implicit step similar to the explicit step as

$$\begin{aligned} \frac{Q_i^* - \hat{Q}_i^*}{\tau_0} &= -\frac{1}{m_i}\sum_{j\in I_i} f^{\text{im}}(Q_j^*)c_{ij} \\ &= \mathcal{O}(h) - div f^{\text{im}}(q(x^i,\tau_0)) \end{aligned} \tag{4.10}$$

which gives

$$\begin{aligned}\hat{Q}_i^* &= Q_i^* + \tau_0\, div f^{\mathrm{im}}(q(x^i,\tau_0)) + \tau_0\mathcal{O}(h) \\ &= q(x^i,0) - \tau_0\, div f^{\mathrm{ex}}(q(x^i,0)) + \underbrace{\tau_0}_{\mathcal{O}(h)}\left(\underbrace{div(f^{\mathrm{im}}(q(x^i,\tau_0)) - f^{\mathrm{im}}(q(x^i,0)))}_{\mathcal{O}(h)} + \mathcal{O}(h)\right)\end{aligned}$$

This gives

$$||\hat{Q}^* - \hat{Q}^1||_\infty = \mathcal{O}(h^2)$$

Without giving a detailed explanation, assume for sufficiently small τ_n the mapping $\hat{Q}^{n+1} \mapsto Q^{n+1}$ is sufficiently close to identity in some neighborhood of our smooth initial data so that we conclude

$$Q_i^1 = Q_i^* + \mathcal{O}(h^2)$$

which gives

$$\begin{aligned}\frac{Q_i^1 - \hat{Q}_i^1}{\tau_0} &= \underbrace{\frac{\mathcal{O}(h^2)}{\tau_0}}_{\mathcal{O}(h)}\mathcal{O}(h) - \underbrace{div f^{\mathrm{im}}(q(x^i,\tau_0))}_{div f^{\mathrm{im}}(q(x^i,0)) + \mathcal{O}(\tau_0)} \\ &= \mathcal{O}(h) - div f^{\mathrm{im}}(q(x^i,0)) \qquad (4.11)\end{aligned}$$

Combining the explicit and implicit step produces

$$\frac{Q_i^1 - Q_i^0}{\tau_0} = \frac{Q_i^1 - \hat{Q}_i^1}{\tau_0} + \frac{\hat{Q}_i^1 - Q_i^0}{\tau_0} = \mathcal{O}(h) - div f(q(x^i,0)) \qquad (4.12)$$

Finally we arrive at

$$\frac{Q_i^1 - q(x^i,\tau_0)}{\tau_0} = \frac{Q_i^1 - Q_i^0 - \tau_0\partial_t q(x^i,0) + \mathcal{O}(\tau_0^2)}{\tau_0} = \mathcal{O}(h) - \underbrace{\partial_t q(x^i,0) - div f(q(x^i,0))}_{=0} = \mathcal{O}(h) \qquad (4.13)$$

(again to be understood in supremum norm in $i \in I$). This is the consistency (the one step errors are asymptotically smaller than the time step). Of course consistency is given for any time step (not just from $t_0 = 0$ to $t_1 = \tau_0$) due to time invariance.
We observe that the scheme is at least first order accurate. It is in general only first order accurate since we used a first order approximation of the time derivative in the derivation of the method.

4.3 Positivity preservation of the density

A basic truth of gas dynamics is that the density of a gas is always positive unless the initial configuration contained a vacuum state. Certainly the density of a gas cannot be negative. While this is physically obvious, we have to show it for our numerical method:
We proceed assuming the scheme to be well defined. In the last chapter we saw that the set of nonnegative densities $D = \{(\rho, \rho u, \mathcal{E})^T | \rho \geq 0\}$ is an invariant domain for the explicit equation.

Let the data Q^n be given such that every Q_i^n defines a positive density. In particular we have $\forall i \in I : Q_i^n \in D$. The explicit part of the IMEX method can be written in the form (2.16):

$$\hat{Q}_i^{n+1} = \left(1 - \sum_{j\in I_i\setminus\{i\}} \frac{2d_{ij}^n \tau_n}{m_i}\right) Q_i^n + \sum_{j\in I_i\setminus\{i\}} \frac{2d_{ij}^n \tau_n}{m_i} \bar{Q}_{ij}^{n+1}$$

where we recall that $\bar{Q}_{ij}^{n+1}$ lies in any invariant domain that contains both Q_i^n and Q_j^n. In our case this means that it also defines a nonnegative density. The CFL-condition guaranteed that $\sum_{j\in I_i\setminus\{i\}} \frac{2d_{ij}^n\tau_n}{m_i} \in [0,1]$. If we require CFL < 1, we even get $\sum_{j\in I_i\setminus\{i\}} \frac{2d_{ij}^n\tau_n}{m_i} < 1$, meaning that (2.16) then defines a convex combination with a nontrivial component Q_i^n. In our case we therefore get

$$[\hat{\rho}]_i^{n+1} = \underbrace{\left(1 - \sum_{j\in I_i\setminus\{i\}} \frac{2d_{ij}^n \tau_n}{m_i}\right)}_{>0} \underbrace{[\rho]_i^n}_{>0} + \sum_{j\in I_i\setminus\{i\}} \underbrace{\frac{2d_{ij}^n \tau_n}{m_i}}_{\geq 0} \underbrace{[\bar{\rho}]_{ij}^{n+1}}_{\geq 0} > 0 \tag{4.14}$$

I.e. if we choose the CFL number strictly less than 1, the explicit step preserves data with strictly positive densities.
The implicit step is even simpler: $[\rho]_i^{n+1} = [\hat{\rho}]_i^{n+1}$. Hence, by induction, the scheme preserves positive density in the initial data for all future times.

4.4 Exact preservation of constant density and pressure data

One test case for numerical methods for the (scaled) Euler equations are contact discontinuities. Those are jumps in the density with constant pressure and stream velocity. More generally we can investigate the case of initial data with arbitrarily given density but constant pressure and velocity $u(\cdot,0) = \bar{u}, p(\cdot,0) = \bar{p}$. The exact solution of this problem is simply an advection of the density with the flow:

$$\begin{pmatrix} \rho \\ u \\ p \end{pmatrix}(x,t) = \begin{pmatrix} \rho(x - \bar{u}t, 0) \\ \bar{u} \\ \bar{p} \end{pmatrix}$$

We cannot expect the numerical scheme to exactly replicate the density advection, since it contains a lot of numerical viscosity. Preserving a constant state in the whole domain however seems more reasonable. For the explicit step we compute the velocity update using

$$[\hat{\rho}]_i^{n+1} = [\rho]_i^n + \frac{\tau_n}{m_i} \sum_{j\in I_i} \left([\rho]_j^n d_{ij}^n - [\rho]_j^n [u]_j^n \cdot c_{ij}\right) \tag{4.15}$$

$$\begin{aligned} [\hat{\rho u}]_i^{n+1} &= [\rho u]_i^n + \frac{\tau_n}{m_i} \sum_{j\in I_i} \left([\rho u]_j^n d_{ij}^n - [\rho u]_j^n [u]_j^n \cdot c_{ij}\right) \\ &= \bar{u}[\rho]_i^n + \bar{u}\frac{\tau_n}{m_i} \sum_{j\in I_i} \left([\rho]_j^n d_{ij}^n - [\rho]_j^n [u]_j^n \cdot c_{ij}\right) \end{aligned} \tag{4.16}$$

which gives

$$[\hat{u}]_i^{n+1} = \frac{[\hat{\rho u}]_i^{n+1}}{[\hat{\rho}_i^{n+1}]} = \frac{\bar{u}[\hat{\rho}]_i^{n+1}}{[\hat{\rho}]_i^{n+1}} = \bar{u} \tag{4.17}$$

For the pressure we compute

$$\begin{aligned}
[\hat{\mathcal{E}}]_i^{n+1} &= [\mathcal{E}]_i^n + \frac{\tau_n}{m_i}\sum_{j\in I_i}\left([\mathcal{E}]_j^n d_{ij}^n - \frac{{M_{\mathrm{ref}}}^2}{2}[\rho]_j^n \bar{u}^2\bar{u}\cdot c_{ij}\right)\\
&= \frac{{M_{\mathrm{ref}}}^2}{2}[\rho]_i^n||\bar{u}||^2 + \frac{\bar{p}}{\gamma-1} + \frac{\tau_n}{m_i}\sum_{j\in I_i}\left(\frac{{M_{\mathrm{ref}}}^2}{2}[\rho]_j^n||\bar{u}||^2 d_{ij}^n - \frac{{M_{\mathrm{ref}}}^2}{2}[\rho]_j^n||\bar{u}||^2\bar{u}\cdot c_{ij}\right)\\
&= \frac{\bar{p}}{\gamma-1} + \frac{{M_{\mathrm{ref}}}^2}{2}||\bar{u}||^2\left([\rho]_i^n + \frac{\tau_n}{m_i}\sum_{j\in I_i}\left([\rho]_j^n d_{ij}^n - [\rho]_j^n\bar{u}\cdot c_{ij}\right)\right)\\
&= \frac{\bar{p}}{\gamma-1} + \frac{{M_{\mathrm{ref}}}^2}{2}||\bar{u}||^2[\hat{\rho}]_i^{n+1} \qquad (4.18)
\end{aligned}$$

and obtain

$$[\hat{p}]_i^{n+1} = (\gamma-1)\left([\hat{\mathcal{E}}]_i^{n+1} - \frac{{M_{\mathrm{ref}}}^2}{2}[\hat{\rho}]_i^{n+1}||\bar{u}||^2\right) = \bar{p} \qquad (4.19)$$

To show that the implicit update also preserves velocity and pressure, look at the Picard iteration: It is $[p]_i^{n+1,0} = [\hat{p}]_i^{n+1} = \bar{p}$ and $[\rho u]_i^{n+1,0} = [\hat{\rho u}]_i^{n+1}$. Since the density is not changed by the implicit step, define $[u]_i^{n+1,r} = [\rho u]_i^{n+1,r}/[\hat{\rho}]_i^{n+1}$. Assume for some $r \in \mathbb{N}_0$ that $[p]_i^{n+1,r} = \bar{p}$ and $[\rho u]_i^{n+1,r} = [\hat{\rho}]_i^{n+1}\bar{u}$. The Picard update (5.2) then gives

$$\begin{aligned}
&\frac{[p]_i^{n+1,r+1}}{\gamma-1} - \sum_{j\in I_i}\sum_{k\in I_j}\frac{\tau_n^2}{m_i m_j}\frac{\gamma}{\gamma-1}\frac{\bar{p}}{[\hat{\rho}]_j^{n+1}}(c_{ij}\cdot c_{jk})\frac{[p]_k^{n+1,r+1}}{{M_{\mathrm{ref}}}^2}\\
&= \underbrace{[\hat{\mathcal{E}}]_i^{n+1} - \frac{{M_{\mathrm{ref}}}^2}{2}[\hat{\rho}]_i^{n+1}||\bar{u}||^2}_{=\bar{p}/(\gamma-1)} - \frac{\tau_n}{m_i}\frac{\gamma}{\gamma-1}\sum_{j\in I_i}\frac{\bar{p}}{[\hat{\rho}]_j^{n+1}}c_{ij}\cdot\bar{u}[\hat{\rho}]_j^{n+1}\\
&= \frac{\bar{p}}{\gamma-1} - \frac{\tau_n}{m_i}\frac{\gamma}{\gamma-1}\bar{p}\bar{u}\cdot\underbrace{\sum_{j\in I_i}c_{ij}}_{=0}\\
&\Rightarrow [p]_i^{n+1,r+1} = \bar{p} \qquad (4.20)
\end{aligned}$$

Equation (5.3) now gives

$$\begin{aligned}
[\rho u]_i^{n+1,r+1} &= [\hat{\rho u}]_i^{n+1} - \frac{\tau_n}{m_i}\sum_{j\in I_i}\frac{c_{ij}}{{M_{\mathrm{ref}}}^2}[p]_j^{n+1,r+1}\\
&= [\hat{\rho u}]_i^{n+1} - \frac{\tau_n\bar{p}}{{M_{\mathrm{ref}}}^2 m_i}\underbrace{\sum_{j\in I_i}c_{ij}}_{=0} = [\hat{\rho u}]_i^{n+1} \qquad (4.21)
\end{aligned}$$

By induction if follows that

$$\forall r\in\mathbb{N}_0 : [p]_i^{n+1,r} = \bar{p} \;\wedge\; [\rho u]_i^{n+1,r} = [\hat{\rho u}]_i^{n+1} \qquad (4.22)$$

so that in the limit (or after a finite number of iterations) we also get

$$\begin{aligned}
[p]_i^{n+1} &= \bar{p} \qquad (4.23)\\
[u]_i^{n+1} = [\rho u]_i^{n+1}/[\rho]_i^{n+1} &= \underbrace{[\hat{\rho u}]_i^{n+1}}_{=[\hat{\rho}]_i^{n+1}\bar{u}}/[\hat{\rho}]_i^{n+1} = \bar{u} \qquad (4.24)
\end{aligned}$$

Again by induction over n we conclude that the scheme preserves data with a constant velocity and pressure (on a domain without boundary).

Chapter 5

Implementation

This chapter aims to give an overview of the techniques involved in implementing the IMEX-splitted numerical scheme. The implementation of the method is done in C++ based on the code "step-69" that is using the deal.II library.

5.1 deal.II

deal.II [1] ("Differential Equations Analysis Library no. 2", also refer to [2]) is an open source C++ library for finite element computations. It emerged from the Numerical Methods group of the University of Heidelberg in 1998. Some of the features it provides are

- Handling of "triangulations" (composed of quadrilaterals in 2 dimensions)
- Basic manual mesh creation functionality
- Distribution of degrees of freedom (indices)
- Implementation of Lagrangian finite elements
- Provides quadrature points and weights for computing coefficients
- Use of template classes for writing code independent of the (spatial) dimension
- Parallelization for large scale problems (MPI)
- Linear solvers, sparse matrix classes, PETSc- and Trilinos wrapper
- Adaptive mesh refinement using error estimates (implements Kelly-error-estimator based on Laplace equation)
- Parameter input functionality
- Output of .vtu/.pvtu-files for visualization programs

[1] `https://dealii.org/`

5.1.1 Creating a grid

Certain mesh geometries can be manually created in deal.II from a set of functions defined in the GridGenerator class. Some of them are

- subdivided_hyper_cube : A hyper cube subdivided into a number of cells in every dimension
- subdivides_hyper_rectangle
- subdivides_hyper_parallelepiped
- simplex : In 2-D a triangle composed of 4 quadrilaterals
- hyper_cube_with_cylindrical_hole : This mesh can be used to simulate the flow around a cylinder
- subdivided_cylinder
- hyper_ball
- hyper_shell : The (set-theoretic) difference between two hyper-balls (with proper radii)
- and several more ...

All of these functions fill an object of the Triangulation class. To create more complicated meshes, it is possible to merge different meshes with one of the merge_triangulations functions. Finally the created mesh can be refined in a sequence of refinement steps by dividing given cells. The following code produces the grid shown in figure 5.1:

```
Triangulation<2> triangulation;
Triangulation<2> tria1, tria2, tria3, tria4, tria5, tria6;

GridGenerator::hyper_cube_with_cylindrical_hole(tria1, 0.5, 1.);
GridGenerator::subdivided_hyper_rectangle(tria2, {2, 1},
Point<2>(-1., 1.), Point<2>(1., 2.));
GridGenerator::subdivided_hyper_rectangle(tria3, {2, 1},
Point<2>(-1., -1.), Point<2>(1., -2.));
GridGenerator::simplex(tria4,
{Point<2>(1., 1.), Point<2>(3., 1.), Point<2>(1., 2.)});
GridGenerator::simplex(tria5,
{Point<2>(1., -2.), Point<2>(3., -1.), Point<2>(1., -1.)});
GridGenerator::subdivided_hyper_rectangle(tria6, {2, 2},
Point<2>(1., -1.), Point<2>(3., 1.));

GridGenerator::merge_triangulations(
{&tria1, &tria2, &tria3, &tria4, &tria5, &tria6}, triangulation);
triangulation.set_manifold(0, PolarManifold<2>(Point<2>()));
triangulation.refine_global(1);
```

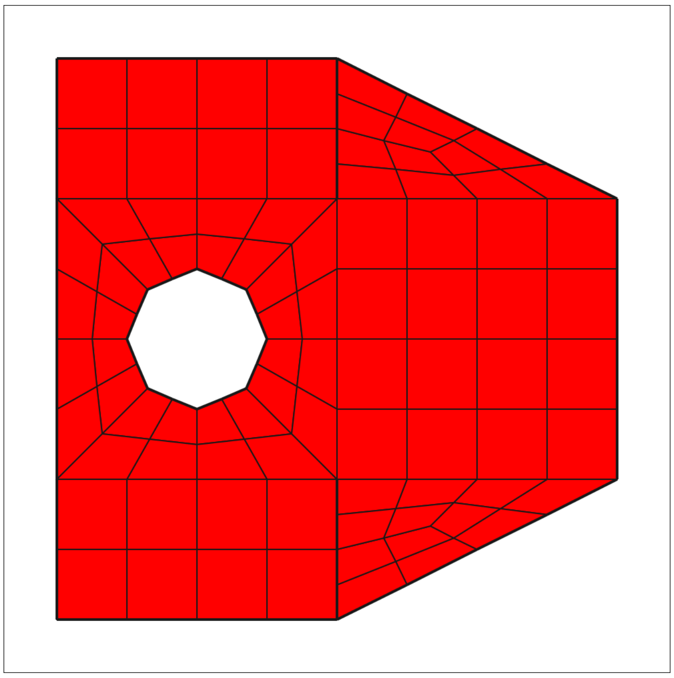


Figure 5.1: Demonstration of a grid created with deal.II

5.1.2 Degrees of freedom

In order to numerically solve a finite element problem, the unknowns must be assigned a range of indices which is necessary to construct the coefficient matrices. In our case this amounts to enumerating the nodes. The deal.II library provides that functionality for given finite elements with the DoFHandler class. Once the grid has been created, the kind of finite elements to be used is specified. For Lagrangian elements, an object of the class FE_Q contains this information. The DoFHandler is initialized with the information about the grid and then distributes the degrees of freedom (indices). If the variable triangulation contains a 2 dimensional grid, the following code gives bilinear elements:

```
FE_Q<2> fe(1);
DoFHandler<2> dof_handler(triangulation);
dof_handler.distribute_dofs(fe);
```

Now that the dof_handler knows the grid, it can be used to iterate over the cells using an iterator range

```
for(&cell : dof_handler.active_cell_iterators())
```

The variable cell then contains relevant information about the current cell. For example, the indices on it can be extracted into a std::vector local_dof_indices by calling

```
cell->get_dof_indices(local_dof_indices);
```

5.1.3 Computing FE coefficients

Computing the finite element coefficients (in our case the m_i and c_{ij}) requires determining an integral. To this end, deal.II provides quadrature points and weights for (exact) numerical integration. First, the type of quadrature formula is chosen. For Gaussian quadrature with $2 \cdot 2 = 4$ quadrature points in 2 dimensions we use the object

```
QGauss<2> quadrature_formula(2);
```

The quadrature points and weights depend on the finite element. The weights are computed as the product of certain base weights and the Jacobian determinant that comes from the mapping of the base element to the concrete cell. It will be denoted JxW. The FEValues class that takes care of this. It also needs to be told what information about the shape functions is required (values, gradients, ...). Given the finite element object fe the code is

```
FEValues<2> fe_values(fe, quadrature_formula,
        update_values | update_gradients | update_JxW_values);
```

The integrals for computing the coefficients are split into a sum over all (relevant) cells. On a given cell (denoted by variable cell), the contribution is computed as a sum over the quadrature points as is sketched in the following code:

```
fe_values.reinit(cell);
// iterate over quadrature points
for(const auto index : fe_values.quadrature_point_indices()){
  for(const auto i : fe_values.dof_indices())
    for(const auto j : fe_values.dof_indices())
      // value of shape function phi_i at given quadrature point
      auto phi_i = fe_values.shape_value(i, index);

      auto phi_j = fe_values.shape_value(j, index);

      // gradient of shape function phi_i at quadrature point
      auto grad_phi_i = fe_values.shape_grad(i, index);

      auto grad_phi_j = fe_values.shape_grad(j, index);
      ...

      auto JxW = fe_values.JxW(index);          // quadrature weight

      auto contribution =
        function(phi_i, phi_j, grad_phi_i, grad_phi_j, ...) * JxW;
      // put contribution into matrix
      ...
}
```

where function is to be replaced by the function we want to integrate.

5.1.4 Sparsity

The matrices arising in FE-problems are often very large, but also very sparse, meaning that most matrix entries are guaranteed to be zero. This fact is exploited by using appropriate matrix classes. deal.II provides a SparseMatrix class using compressed row storage. Here, we need to tell the matrix how the sparsity structure looks like. That is, which entries (given by indices) must be stored because they may be nonzero. This structure is provided by a SparsityPattern object that can be set up in different ways (we will do it manually later):

```
SparsityPattern sparsity_pattern;

fill sparsity_pattern ...

SparseMatrix<double> matrix;
matrix.reinit(sparsity_pattern);
```

deal.II also provides a wrapper to the PETSc library, that implements linear solvers and sparse matrix data structures.

5.2 step-69

The deal.II project comes with a list of tutorial programs enumerated as step-1, step-2, etc. The "step-69" code [2] (from [15]) implements Guermond's invariant domain preserving (fully explicit) scheme for the compressible Euler equations. By modifying this code, a program for the low Mach method is obtained. The step-69 program consists of several classes. They are

- Discretization : Contains the triangulation (grid), the type of finite elements, the quadrature formula and a few more
- OfflineData : Contains among others the boundary normal map (defines a discrete version of a boundary normal vector field), the dof handler, the sparsity pattern, the lumped mass matrix $diag((m_i)_i)$ and the coefficient matrix $(c_{ij})_{ij}$
- ProblemDescription : Contains among others the isentropic exponent γ, the state functions momentum, internal energy, pressure as well as the flux function and the compute_lambda_max function of the Riemann problem
- InitalValues : Defines the initial state
- TimeStepping : Contains among others the CFL-number, the graph viscosity matrix $(d_{ij})_{ij}$ and the update function make_one_step
- SchlierenPostProcessor : A special class for computing a scalar schlieren function used to visualize the flow field
- MainLoop : The main class that instantiates an object for the all of the above classes and defines the run-function for performing the simulation

We can let all of those classes inherit from the ParameterAcceptor class, provided by deal.II, which is used to initialize certain variables (e.g. CFL-number) with values defined in a parameter file. First, the ParameterAcceptor constructor is called with a string corresponding to the relevant subsection in the parameter file. Then, to read in a parameter, the add_parameter method is called and given a variable (reference) and a variable name to register the parameter. Finally, upon running the program, the ParameterAcceptor::initialize function is called in the beginning with the name of the parameter file so it can set the variables with the values defined in it.

[2]https://dealii.org/developer/doxygen/deal.II/step_69.html

5.2.1 Sparsity pattern

In step-69 the sparsity pattern is created manually. The procedure is to first create a DynamicSparsityPattern object. If there are n degrees of freedom relevant for the sparse matrix we can use

```
DynamicSparsityPattern dsp(n,n);
```

One then iterates over the cells to find all index pairs corresponding to all places in the matrix that may be nonzero. Such entries for a given row index dof of the matrix can be added by using the method

```
dsp.add_entries(dof, dof_indices.begin(), dof_indices.end());
```

where dof_indices is a std::vector of column indices of the (to be added) entries in the given row. Once all entries are added to the dynamic sparsity pattern, the object sparsity_pattern of class SparsityPattern can be set up with the information in the dsp by calling

```
sparsity_pattern.copy_from(dsp);
```

5.2.2 Parallelization

MPI

For parallel computations on several machines, deal.II allows the use of the Message Passing Interface (MPI) to exchange information. The step-69 program uses this feature. Not only the computing resources can be distributed to many computing nodes, but also data structures to limit memory requirements per node. Even the grid (triangulation) is distributed. To this end, deal.II provides modules that contain classes for distributed data structures that when constructed are given a MPI_Comm (MPI communicator) object to deal with the information exchange. If we create a distributed triangulation (object of the class parallel::distributed::Triangulation<dim>), we can then, from the dof-handler, get an index range as a vector of locally owned indices and a vector of locally relevant indices. The locally owned indices are the ones directly assigned to the current node, whereas the relevant ones may also include neighboring indices. Those index ranges are important for initializing distributed vectors.

Thread parallelization

The explicit update basically consists of several computationally expensive for-loops for computing the graph viscosity and the actual update. The code can be made faster by invoking the parallel::apply_to_subranges function for running the code on many threads in parallel (on each node). It is given an index range and a function handle. The index range is split into sub-ranges based on a parameter. The code

```
parallel::apply_to_subranges(index_range.begin(), index_range.end(),
          on_subranges, 4096);
```

splits the index_range vector with the grain-size 4096 and calls the function on_subranges with start and end indices for each sub-range.

5.2.3 Output

The numerical solution is written into .vtu files and .pvtu files by using the DataOut<dim>class. First, the underlying geometry is given:

```
DataOut<dim> data_out;
data_out->attach_dof_handler(dof_handler);
```

solution component vectors can be added by calling something like

```
data_out->add_data_vector(component_vector, "component_name");
... other components ...
data_out->build_patches();
```

Without going into details, the output-file is then written with the code:

```
data_out->write_vtu_with_pvtu_record(
"", name + "-solution", cycle, mpi_communicator, 6);
```

The implementation in step-69 is a bit more involved. The output files can be visualized using tools like for example *ParaView*[3].

5.2.4 Boundary treatment

step-69 defines 3 types of boundary conditions:

- do_nothing
- free_slip
- dirichlet

As the name suggests, at do_nothing boundaries, the scheme is unchanged. The behavior of the boundary is determined by the coefficients c_{ij} and m_i (at the boundary). Consider the one-dimensional case of an equidistant grid with do_nothing boundaries. The scheme then becomes similar to one with a first order extrapolation boundary treatment, which can be used if waves should simply run out of the domain when reaching the boundary (although zero order extrapolation is mostly used, see [14]).

The free_slip boundary condition requires $u \cdot \nu = 0$ with the boundary normal vector field ν. In step-69, the free_slip boundary is enforced explicitly after the update is first performed without boundary treatment (do_nothing). To this end, for every index i with $x^i \in \partial\Omega$, we set

$$[\rho u]_i^{n+1} \leftarrow ([\rho u]_i^{n+1} \cdot \nu_i)\nu_i$$

where

$$\nu_i = \frac{\int_{\partial\Omega} \varphi_i(x)\nu(x)dx}{|| \int_{\partial\Omega} \varphi_i(x)\nu(x)dx||}$$

defines the discrete boundary normal vector field.

At a Dirichlet boundary, a solution value is prescribed. The strong enforcement of those boundaries in step-69 simply sets the boundary to those given values after every time-step.

Figure 5.2 demonstrates the output of the step-69 tutorial program (visualized with ParaView) for a 2 dimensional flow around a cylinder. The initial state is $\rho = 1.4$, $u = (3,0)^T$, $p = 1$. The CFL-number is 0.8. Shown is a schlieren image, which visualizes a discrete version of the schlieren data

$$\text{schlieren} = \exp\left(10 \frac{||\nabla\rho|| - \min_{x\in\Omega} ||\nabla\rho(x)||}{\max_{x\in\Omega} ||\nabla\rho(x)|| - \min_{x\in\Omega} ||\nabla\rho(x)||}\right)$$

[3]www.paraview.org

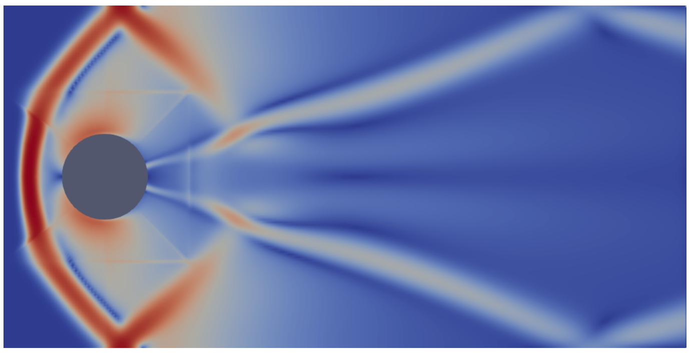


Figure 5.2: Flow over a cylinder computed with deal.II/step-69 and around 37000 gridpoints

5.3 Extension to low Mach

We will now briefly how this code was adapted to the IMEX method. The steps are

- Incorporate M_{ref} parameter (into ProblemDescription and parameter file)
- Modify flux function and maximum wavespeed to the explicit part
- Set up the sparse system matrix, rhs and solution vector data structures for the implicit problem
- Add the implicit step to the end of the make_one_step method in the TimeStepping class
- The output was changed from conservative to primitive variables

The first two points above are pretty straight forward. They require modifying the code in several places but without changing the overall structure.

The sparse matrix class used for the implicit system matrix is PETScWrappers::MPI::SparseMatrix, and the right-hand-side and solution vectors are PETScWrappers::MPI::Vector objects. PETSc is a linear algebra library in C, for which deal.II provides a wrapper. The populate the matrix while looping over all cells to find index pairs and values, the method

```
system_matrix.set(row, col_indices , values);
```

is used, where row is the row index and col_indices is a std::vector of column indices and values the std::vector of matrix entries corresponding to it. To then solve the implicit problem, the iterative BiCGStab method is employed (see [19] for an explanation of the method). To this end, we use the code

```
SolverControl solver_control(system_solution.size(), 1.e-8);
PETScWrappers::PreconditionBlockJacobi preconditioner(system_matrix);
PETScWrappers::SolverBicgstab solver(solver_control);
solver.solve(system_matrix, system_solution, system_rhs, preconditioner);
```

to set up a solver object with certain parameters (like the error tolerance 10^{-8}) and solve the system with a preconditioner.

5.3.1 Boundary treatment

The boundary conditions have to be incorporated into the implicit step:

Dirichlet

For a dirichlet boundary condition $q(x,t) = g(x,t) \,\text{on}\, D \subset \partial\Omega$, the update equation is replaced on every dirichlet boundary node x^i by

$$Q_i^{n+1} = g(x^i, t_n + \tau_n)$$

The strong boundary enforcement after the explicit step already gives $\hat{Q}_i^{n+1} = g(x^i, t_n + \tau_n)$ on that boundary. In the implicit update we now simply keep those values constant by changing the corresponding picard equations to

$$\begin{aligned} [p]_i^{n+1,r+1} &= [p]_i^{n+1,r} \\ [\rho u]_i^{n+1,r+1} &= [\rho u]_i^{n+1,r} \end{aligned} \tag{5.1}$$

for all x^i on a dirichlet boundary.

Free slip

At free slip boundaries, the momentum is projected onto the tangential space of the boundary. This leads to the following equations for the picard iteration:

$$\begin{aligned} &\frac{[p]_i^{n+1,r+1}}{\gamma - 1} - \sum_{j \in I_i} \sum_{k \in I_j} \frac{\tau_n^2}{m_i m_j} \frac{\gamma}{\gamma - 1} \frac{[p]_j^{n+1,r}}{[\hat{\rho}]_j^{n+1}} (c_{ij} \cdot (c_{jk} - c_{jk} \cdot \hat{\nu}_j c_{jk})) \frac{[p]_k^{n+1,r+1}}{{M_{\text{ref}}}^2} \\ &= [\hat{\mathcal{E}}]_i^{n+1} - \frac{{M_{\text{ref}}}^2}{2[\hat{\rho}]_i^{n+1}} ||[\rho u]_i^{n+1,r}||^2 - \sum_{j \in I_i} \frac{\tau_n}{m_i} \frac{\gamma}{\gamma - 1} \frac{[p]_j^{n+1,r}}{[\hat{\rho}]_j^{n+1}} c_{ij} \cdot [\hat{\rho u}]_j^{n+1} \end{aligned} \tag{5.2}$$

$$[\rho u]_i^{n+1,r+1} = [\hat{\rho u}]_i^{n+1} - \frac{\tau_n}{m_i} \sum_{j \in I_i} \frac{c_{ij} - c_{ij} \cdot \hat{\nu}_i c_{ij}}{{M_{\text{ref}}}^2} [p]_j^{n+1,r+1} \tag{5.3}$$

where

$$\hat{\nu}_i = \begin{cases} \nu(x^i)\,, & \text{if } x^i \text{ on free slip boundary} \\ 0\,, & \text{else} \end{cases}$$

Figure 5.3 visualizes the output of the IMEX adapted code for the same problem as in figure 5.2.

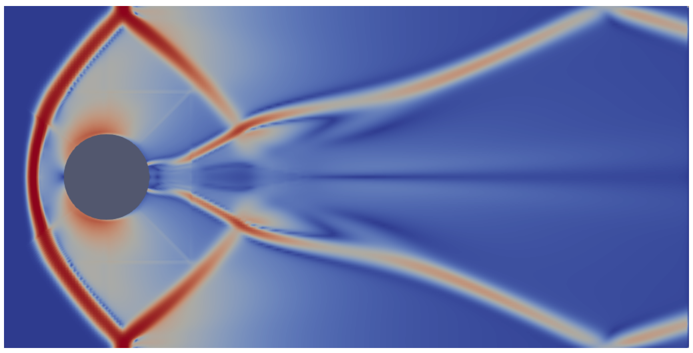


Figure 5.3: Output of the modified code for the original step-69 problem

Chapter 6

Numerical tests

6.1 Riemann problems

6.1.1 Shock tube problem

For the first test compare the IMEX splitted scheme to the original unsplit scheme. The unsplit invariant domain preserving scheme in this case uses an upper bound for the maximum wavespeed adapted from the formulas used in the step-69 program (see also [11]):

$$a = \sqrt{\gamma \frac{p}{\rho}}$$

$$p^*(q_l, q_r, \nu) = p_r \left((a_l + a_r - M_{\text{ref}} \frac{\gamma - 1}{2} (u_r^\nu - u_l^\nu)) / (a_l \frac{p_l}{p_r}^{-(\gamma-1)/(2\gamma)} + a_r) \right)^{2\gamma/(\gamma-1)}$$

$$\lambda_{\text{2-rarefaction}}(q_l, q_r, \nu) =$$

$$\max \left\{ M_{\text{ref}} u_l^\nu - a_l \sqrt{1 + \frac{\gamma+1}{2\gamma} \left(\frac{p^*(q_l, q_r, \nu) - p_l}{p_l} \right)_+} \, , \; M_{\text{ref}} u_r^\nu + a_r \sqrt{1 + \frac{\gamma+1}{2\gamma} \left(\frac{p^*(q_l, q_r, \nu) - p_r}{p_r} \right)_+} \right\}$$

$$\lambda_{\max}(q_l, q_r, \nu) = \frac{1}{M_{\text{ref}}} \min\{\lambda_{\text{2-rarefaction}}(q_l, q_r, \nu), M_{\text{ref}} \max(u_l^\nu, u_r^\nu) + 5 \max(a_l, a_r)\}$$

Pseudocode for the unsplit scheme is given in A.3.
Consider now the one dimensional Riemann problem with initial data

$$\rho_0(x) = \begin{cases} 2 & , \text{ if } x < 0 \\ 1 & , \text{ if } x > 0 \end{cases} \qquad u_0(x) = 0 \qquad p_0(x) = \begin{cases} 2 & , \text{ if } x < 0 \\ 1 & , \text{ if } x > 0 \end{cases} \tag{6.1}$$

for the Euler equations ($M_{\text{ref}} = 1$). For the IMEX scheme in 1D, the code in A.1 was used. The grid is uniform with $\Delta x = h = m_i = 0.02$. The plot in figure 6.1 shows the solutions at the final time $T = 0.5$. The computations were carried out with the parameter CFL $= 0.5$ for both methods. We observe that the unsplit scheme produces a slightly sharper shock wave (on the right). For the rarefaction wave on the left there seem to be only marginal differences. The contact discontinuity (in the middle) is significantly sharper in the IMEX approximation. This is likely due to the fact that the graph viscosity and time step size in the IMEX scheme only depends on the explicit part of the splitting, which deals only with the contact discontinuity. The acoustic waves are treated by the implicit part. The presence of acoustic waves affects the

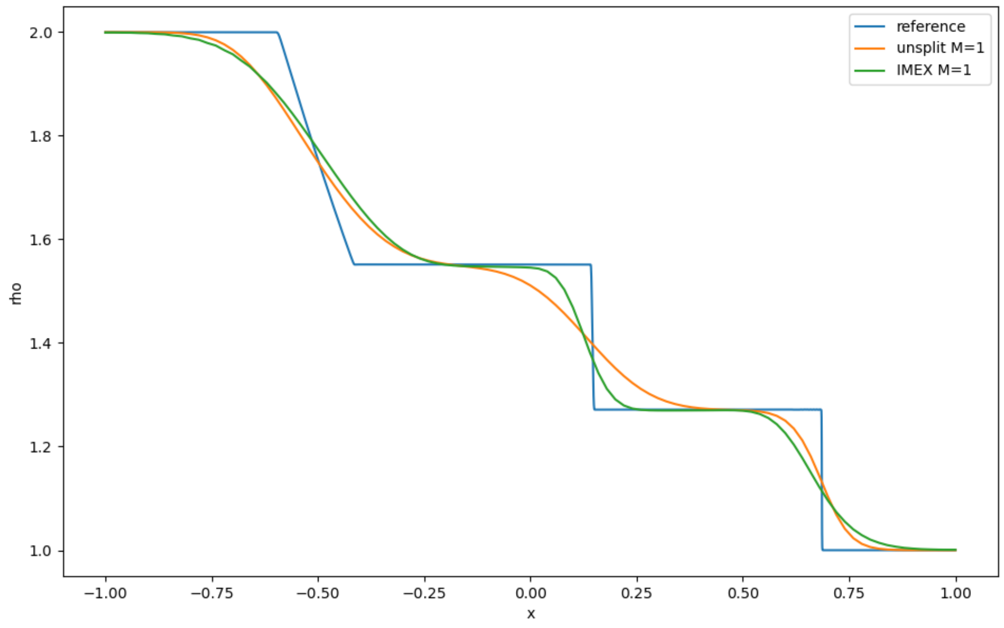


Figure 6.1: Comparison of unsplit scheme and IMEX scheme applied to shock problem

	# Time steps	
$M_{\mathbf{ref}}$	**unsplit**	**IMEX**
0.1	207	17
0.01	1910	17

Table 6.1: 1D pressure disturbance: Comparison of simulation time steps

graph viscosity in the unsplit scheme. Because the graph viscosity is a scalar quantity, this also affects the dissipation of the contact discontinuity.

6.1.2 Pressure disturbance

Consider the initial data

$$\rho_0(x) = 1 \qquad u_0(x) = 1 \qquad p_0(x) = \begin{cases} 1 + {M_{\text{ref}}}^2 & , \text{ if } x < 0 \\ 1 & , \text{ if } x > 0 \end{cases} \tag{6.2}$$

Set the final time of the test to $0.4M_{\text{ref}}$ and the grid parameter $\Delta x = 0.05M_{\text{ref}}$. The IMEX scheme is not well suited to resolve discontinuous or very sharp pressure waves in the case of small M_{ref}. The pressure component of the solution is smeared and spurious oscillations occur. The number of time steps performed is documented in table 6.1. At $M_{\text{ref}} = 0.01$, for every time step of the IMEX scheme, each pressure wave moves by ca. 60 grid points.

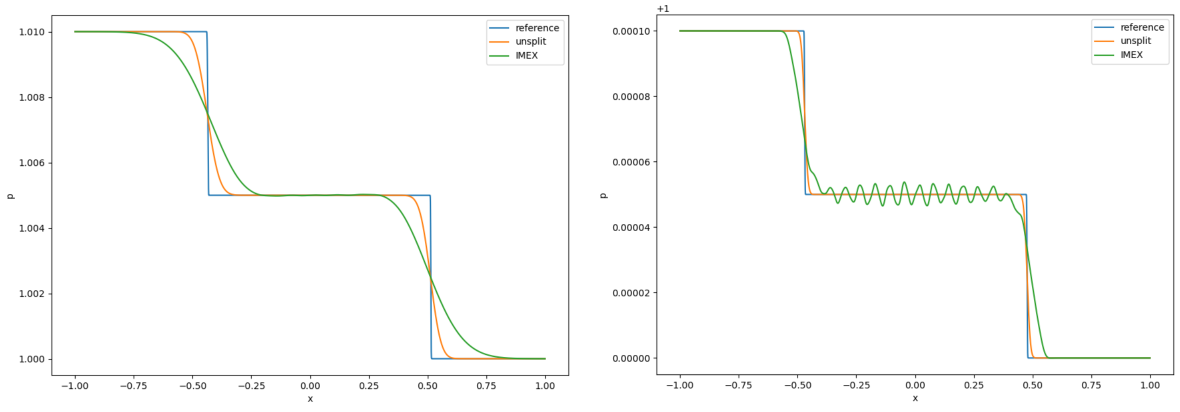


Figure 6.2: Pressure disturbance at $M_{\text{ref}} = 0.1$ (left) and $M_{\text{ref}} = 0.01$ (right)

# Time steps		
$M_{\mathbf{ref}}$	**unsplit**	**IMEX**
1	219	101
0.1	1283	101
0.01	11308	101

Table 6.2: 1D contact discontinuity: Comparison of simulation time steps

6.2 Contact discontinuity

6.2.1 One dimensional

Consider the initial value problem with initial data

$$\rho_0(x) = \begin{cases} 2 & , \text{ if } -1 < x < 0 \\ 1 & , \text{ else} \end{cases} \qquad u_0(x) = 1 \qquad p_0(x) = 1 \tag{6.3}$$

The solution to it are two contact discontinuities traveling with speed 1, regardless of the reference Mach number M_{ref}. Simulating this problem on the domain $[-2, 2]$ with periodic boundaries for different values of M_{ref} again with the unsplit and the IMEX code A.1 at CFL $= 0.5$ and $\Delta x = 0.02$ produces at time $T = 1$ the output visualized in figure 6.3. While the IMEX scheme produces essentially the same solution, the unsplit invariant domain preserving method becomes increasingly dissipative when the reference Mach number is decreased, since the wave speeds and thus also the graph viscosity tend to ∞ as $M_{\text{ref}} \to 0$.
The increasing speed of sound also reduces the time step size of the unsplit method, resulting in a blow up of the number of iterations as the reference Mach number tends to 0. Asymptotically, the number of iterations is inversely proportional to M_{ref}.

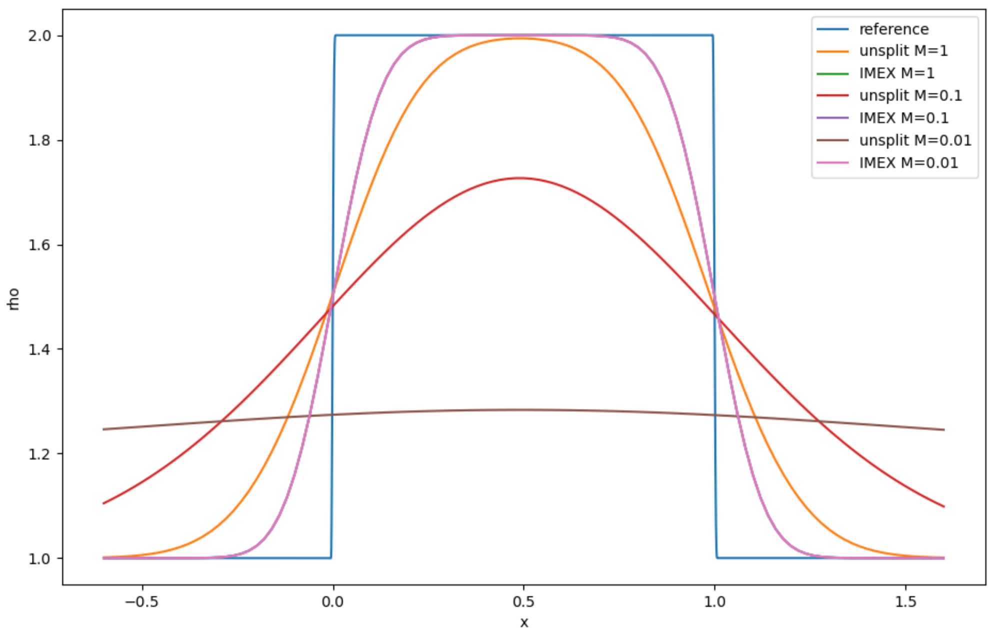


Figure 6.3: Comparison of unsplit scheme and IMEX scheme applied to shock problem

6.2.2 Two dimensional

For the two dimensional test, the initial values were chosen as

$$\rho_0(x) = \begin{cases} 1.54 & , \text{ if } x \in (1,2) \times (0.25, 0.75) \\ 1.4 & , \text{ else} \end{cases} \qquad u_0(x) = (1,0)^T \qquad p_0(x) = 1 \tag{6.4}$$

on the domain $[0,4] \times [0,1]$ with Dirichlet boundary conditions $\rho = 1.4$, $u = (1,0)^T$, $p = 1$. Figure 6.4 shows the visualized output of the IMEX adapted program (A.2) at time $T = 1$. The grid is cartesian with $\Delta x = \Delta y = 1/128$. The reference Mach number was set to $M_{\text{ref}} = 0.01$. The CFL number is 0.5. The pressure and velocity in the numerical solution are indeed constant,

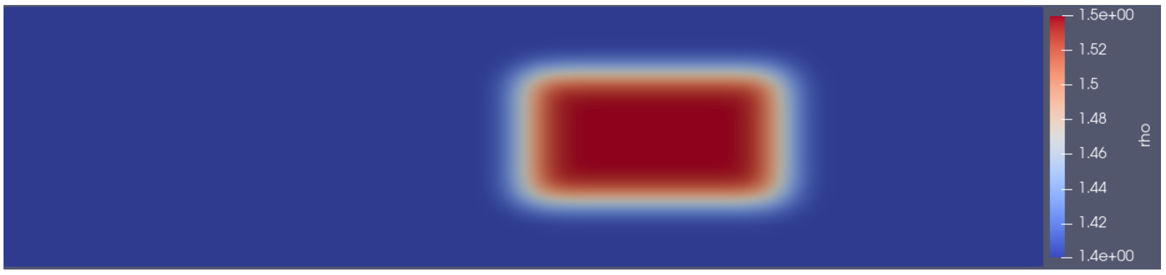


Figure 6.4: Constant velocity and pressure flow in 2D

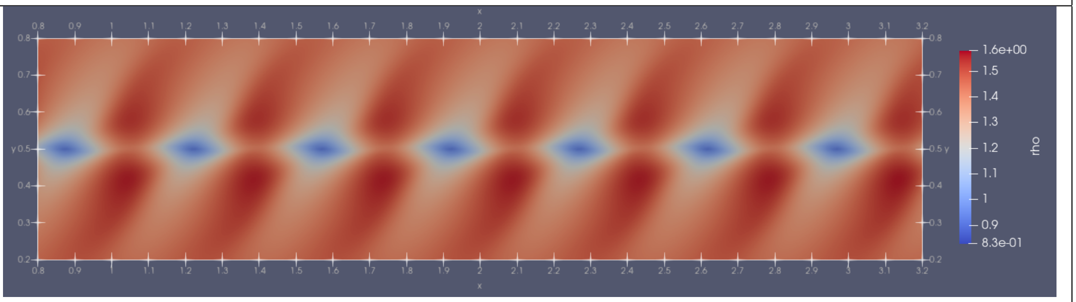
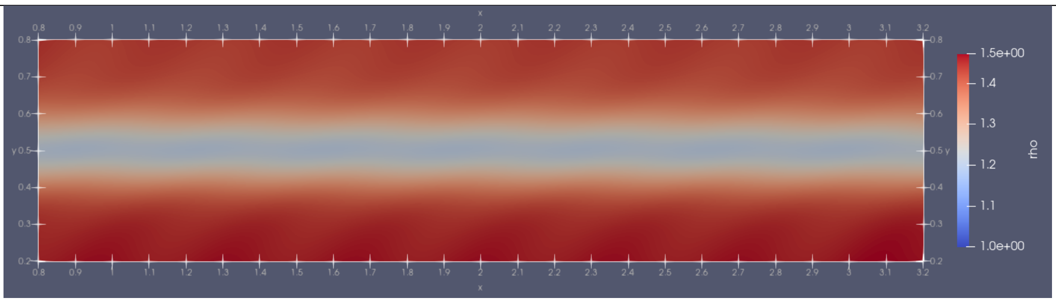
albeit with numerical errors due to finite computational accuracy and error tolerances of the implicit solver. E.g. the variable u_2 varies in the order 10^{-10}.

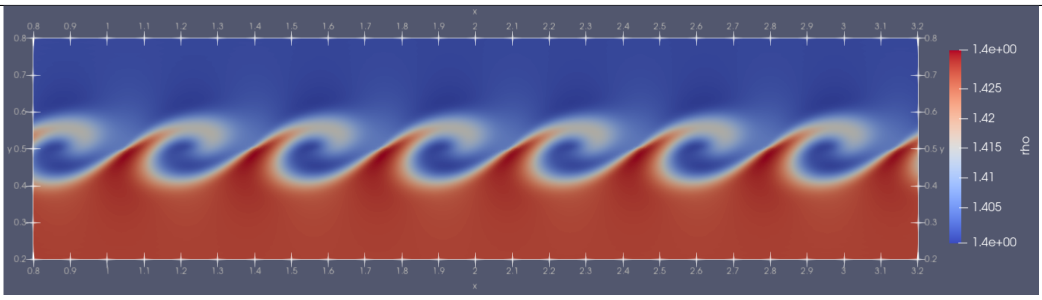
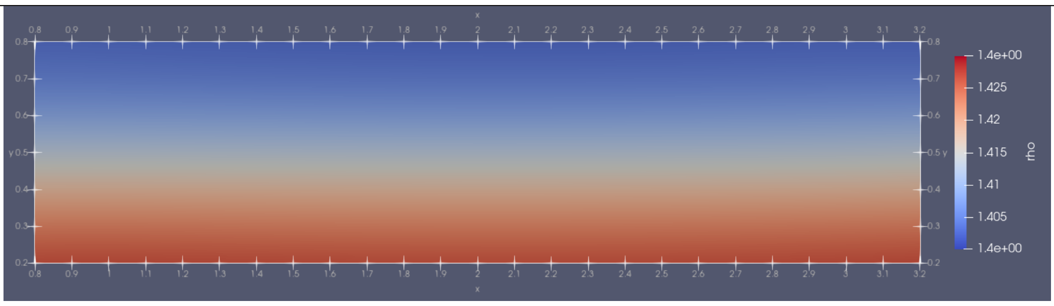
6.3 Kelvin-Helmholtz instability

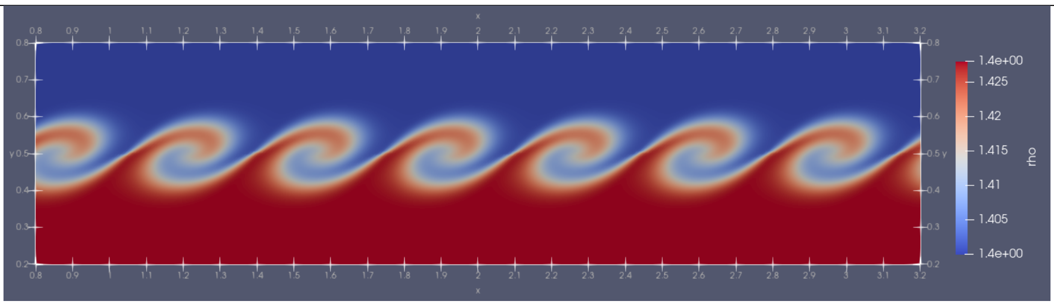
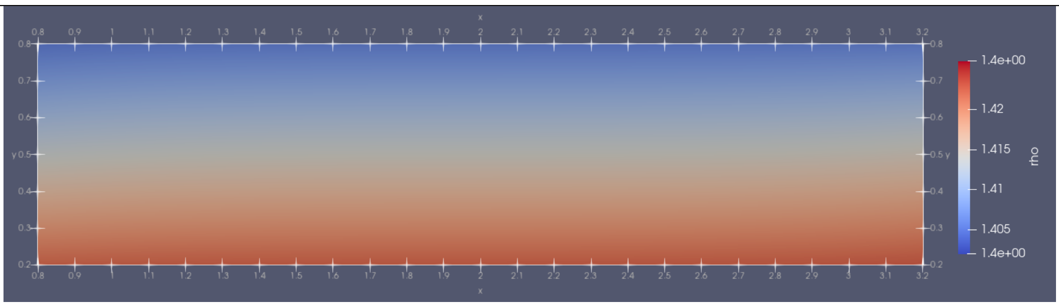
A shear motion of two fluid layers against each other results in an instability called Kelvin-Helmholtz instability. We can try to recreate this from the following initial data containing a small perturbation:

$$(\rho_0(x), u_0(x), p_0(x)) = \begin{cases} (1.428, (-1, 0)^T, 1) & , \text{ if } x_2 < 0.5 + 0.005 \sin(18x_1) \\ (1.4, (1, 0)^T, 1) & , \text{ else} \end{cases} \tag{6.5}$$

Here, the two layers are also distinguished by a 2% difference in density. The simulation domain is $[0, 4] \times [0, 1]$. Dirichlet boundaries matching the initial data are imposed. We compare the original step-69 unsplit scheme with the IMEX scheme for different values of M_{ref}. A cartesian grid with $\Delta x = \Delta y = 1/128$ was used. The CFL-number is 0.75. The following graphics tables present the numerical results at time $t = 0.5$ on the sub-domain $[0.8, 3.2] \times [0.2, 0.8]$.

$M_{\mathbf{ref}} = 1$	
IMEX	
unsplit	

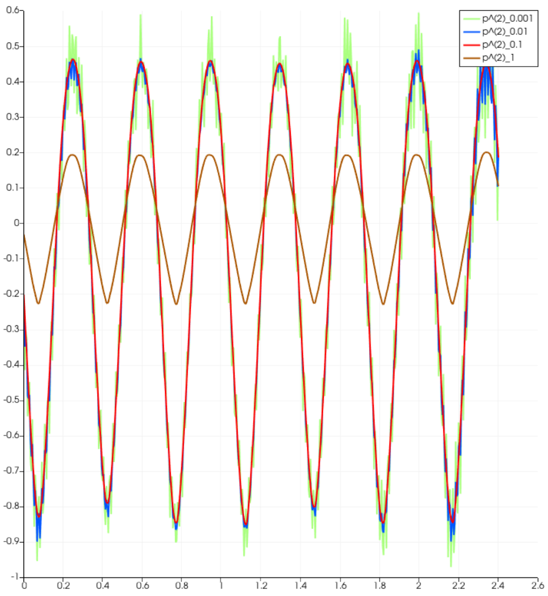
$M_{\mathbf{ref}} = 0.1$	
IMEX	
unsplit	

$M_{\mathbf{ref}} = 0.01$	
IMEX	
unsplit	

$M_{\mathbf{ref}}$	**unsplit**	**IMEX**
1	23.3 s	116 s
0.1	55.9 s	144 s
0.01	432 s	244 s

Table 6.3: Kelvin-Helmholtz simulation: Wall time comparison

We observe that the unsplit scheme introduces a significant amount of dissipation comparable to the case of the contact discontinuity. Since the necessary number of time-steps increases when M_{ref} becomes small, we expect the unsplit scheme to then also take more computational time. A comparison for one sample run each is given in table 6.3 (CPU: AMD Ryzen 5 5625U). The simulations were carried out to final time $T = 1$.



The plot on the left shows the pressure in the Kelvin-Helmholtz IMEX simulations on the slice $[0.8, 3.2] \times \{0.5\}$ for reference Mach numbers down to 0.001. It is $p^{(2)}_{M_{\text{ref}}} := (p-1)/{M_{\text{ref}}}^2$. While the density solution at $M_{\text{ref}} = 0.001$ is quite similar to the one at $M_{\text{ref}} = 0.01$, the pressure contains stronger unphysical oscillations. The unsplit scheme did not resolve any low Mach pressure features just like for the density at the given grid resolution.

6.4 Stationary vortex

A flow is called stationary if the state variables q are constant in time. A stationary vortex centered at the origin is a flow that is rotation invariant, i.e with

$$r(x,t) := ||x|| \qquad e_r(x) = \frac{x}{||x||} \qquad e_\phi(x) = \frac{1}{||x||}(-x_2, x_1)^T$$

in two space dimensions we can write

$$\rho = \rho^\circ(r) \qquad u = u_r^\circ(r)e_r + u_\phi^\circ(r)e_\phi \qquad p = p^\circ(r) \tag{6.6}$$

with the functions

$$r \mapsto \rho^\circ(r) \qquad r \mapsto u^\circ(r) \qquad r \mapsto p^\circ(r)$$

Due to mass conservation all such stationary flows with positive density obey $u_r^\circ = 0$. For any continuous $\rho^\circ > 0$, u_ϕ° with $\rho^\circ(u^\circ)^2 \in \mathcal{O}(r)$ and some $p^{(0)} \in (0, \infty)$ we then obtain a solution to

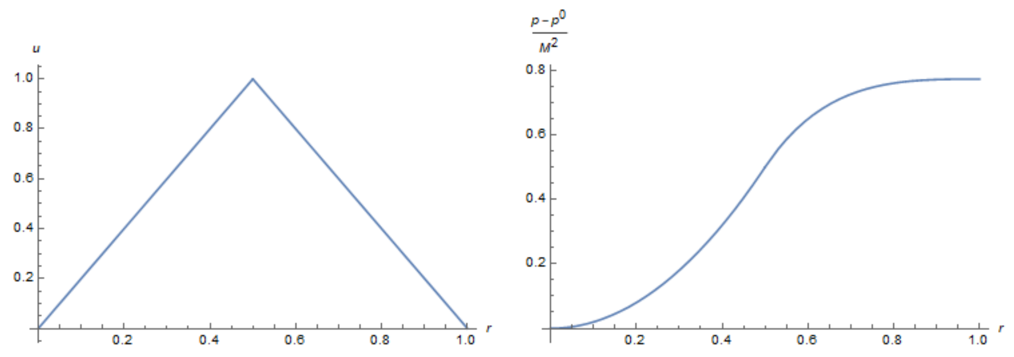


Figure 6.5: Velocity and pressure of the vortex

the scaled Euler equations from

$$p^\circ(r) = p^{(0)} + {M_{\text{ref}}}^2 \int_0^r \frac{1}{s} \rho^\circ(s)(u^\circ(s))^2\, ds \tag{6.7}$$

as can be readily verified by writing the scaled Euler equations in polar coordinates. Equation (6.7) allows us to construct analytical solutions that can be used as test cases. An example similar to the so called Gresho vortex given in [9] is

$$\begin{aligned}
\rho &= 1 \\
u &= \begin{cases} 2re_\phi & \text{, if} \quad r < 0.5 \\ (2-2r)e_\phi & \text{, if} \quad 0.5 \le r < 1 \\ 0 & \text{, if} \quad r \ge 1 \end{cases} \\
p &= \begin{cases} p^{(0)} + {M_{\text{ref}}}^2 2r^2 & \text{, if} \quad r < 0.5 \\ p^{(0)} + {M_{\text{ref}}}^2 \left(4(\ln(2r)+1) + 2r^2 - 8r\right) & \text{, if} \quad 0.5 \le r < 1 \\ p^{(0)} + {M_{\text{ref}}}^2 \left(4(\ln(2)+1) - 6\right) & \text{, if} \quad r \ge 1 \end{cases}
\end{aligned} \tag{6.8}$$

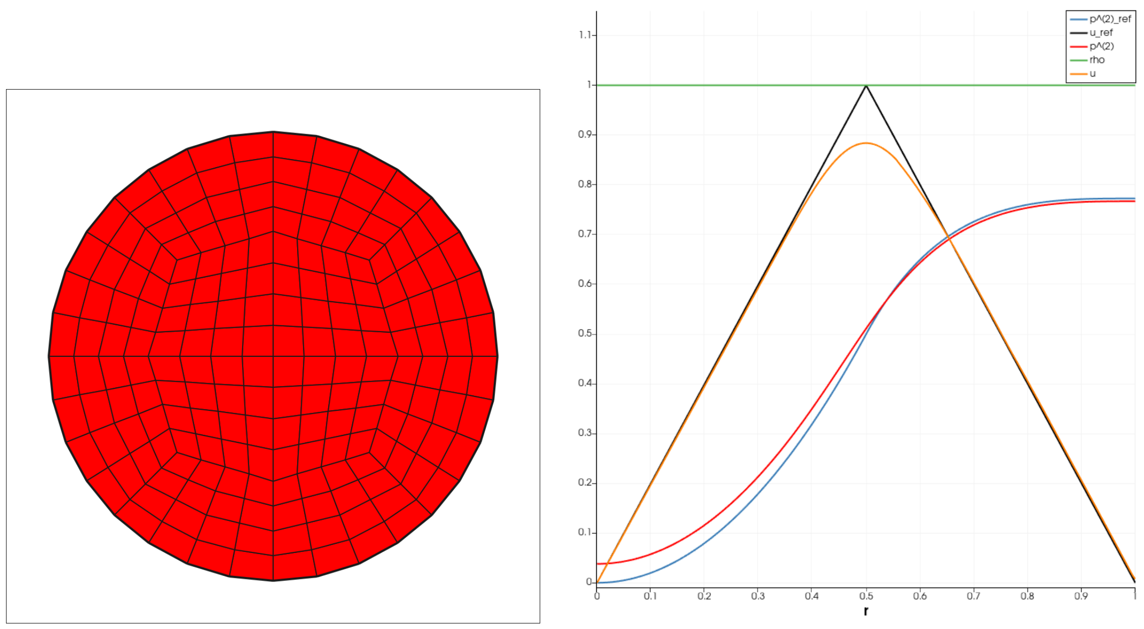


Figure 6.6: Little refined grid for vortex simulation (left) ; Numerical result for most refined grid extracted from the slice $[0,1] \times \{0\}$ (right)

To test the IMEX method with the vortex defined in (6.8), we choose the domain $B_1(0)$, i.e. a ball with radius 1 centered at the origin. We impose free-slip boundary conditions at the boundary $\partial B_1(0)$. We can do so since $u^\circ(1) = 0$. The grid is generated in deal.II from

```
GridGenerator::hyper_ball_balanced(triangulation);
```

and subsequently refined. We simulate up to final time $T = 3$ with $M_{\text{ref}} = 0.04$ (although the simulation also works for much smaller values of M_{ref}) at CFL $= 0.75$. The test is performed on several successively refined grids and the numerical L^1 error at $t_n = T$

$$\begin{aligned} ||q_h^n - q(\cdot, t_n)||_{L^1} &= \int_\Omega || \left(\sum_{i\in I} Q_i^n \varphi_i(x) - q(x, t_n) \right) ||_1 \, dx \\ &\approx \int_\Omega || \sum_{i\in I} (Q_i^n - q(x^i, t_n)) \varphi_i(x) ||_1 \, dx = \sum_{i\in I} ||Q_i^n - q(x^i, t_n)||_1 m_i =: \epsilon_1 \end{aligned} \tag{6.9}$$

is evaluated and summarized in table 6.4. Figure 6.6 shows the velocity and the pressure variation $p^{(2)} = (p-1)/{M_{\text{ref}}}^2$ of the numerical solution at final time T plotted against the exact solution. Each refinement step subdivides every cell into 4 sub-cells. A reasonable choice for the mesh size parameter sequence is then given by $(h_0, h_0/2, h_0/4, ...)$. Fitting the error data in table 6.4 with $\ln(\epsilon_1(h)) = a \ln(h) + b$ gives the experimental order of convergence $a \approx 0.9$. The value seems to increase as the grid refined. We would expect it to approach 1.

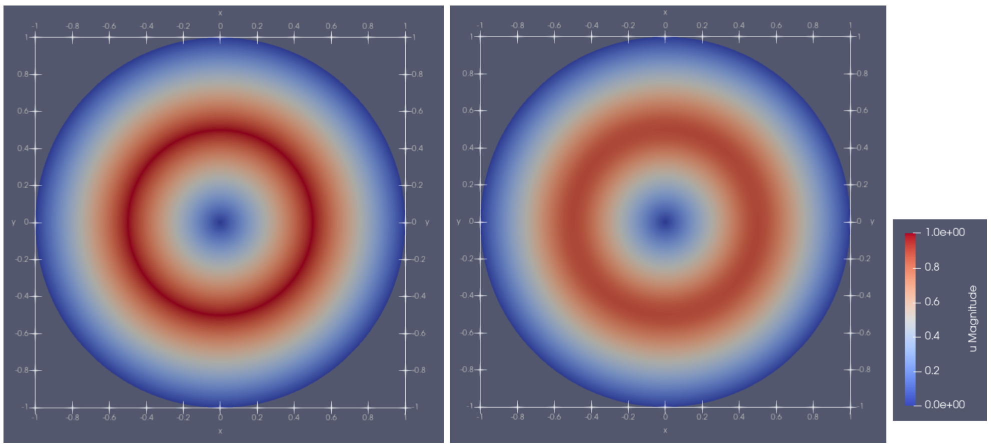


Figure 6.7: Vortex simulated on most refined grid at: $t = 0$ (left) , $t = 3$ (right)

refinement steps	**# grid cells**	ϵ_1	**Total runtime (wall-time)**
4	3072	0.3667	5.27 s
5	12288	0.2044	63.3 s
6	49152	0.1093	342 s
7	196608	0.05705	2250 s

Table 6.4: Error comparison for different grid refinements stages

6.5 2D-flow over a rectangular obstacle

For this test the domain is $([0, 3] \times [0, 1]) \setminus ([1, 1.25] \times [0.25, 0.75])$. The free slip boundary condition is imposed on $\partial([1, 1.25] \times [0.25, 0.75])$. On $\partial([0, 3] \times [0, 1])$ we impose a Dirichlet boundary condition matching the constant initial state

$$\rho_0(x) = 1.4 \qquad u_0(x) = (1, 0)^T \qquad p_0(x) = 1 \tag{6.10}$$

The final time is $T = 0.9$, the CFL-number CFL $= 0.75$. The simulation is done on a cartesian grid with $\Delta x = \Delta y = 1/128$. For the reference Mach number, $M_{\text{ref}} = 0.01$ was chosen. Figure 6.8 shows the flow with a color representation of the velocity magnitude $||u||$ and by visualizing streamlines, which are trajectories of the ordinary differential equation

$$\frac{d}{ds} x_{\text{streamline}}(s) = u(x_{\text{streamline}}(s), t) \tag{6.11}$$

at a given time t (here t=T).

6.6 Flow past a cylinder

Here we take the example problem from step-69 with the domain $[0, 4] \times [0, 2]$ but change the initial conditions to

$$\rho_0(x) = 1.4 \qquad u_0(x) = (0.9, 0.036)^T \qquad p_0(x) = 1 \tag{6.12}$$

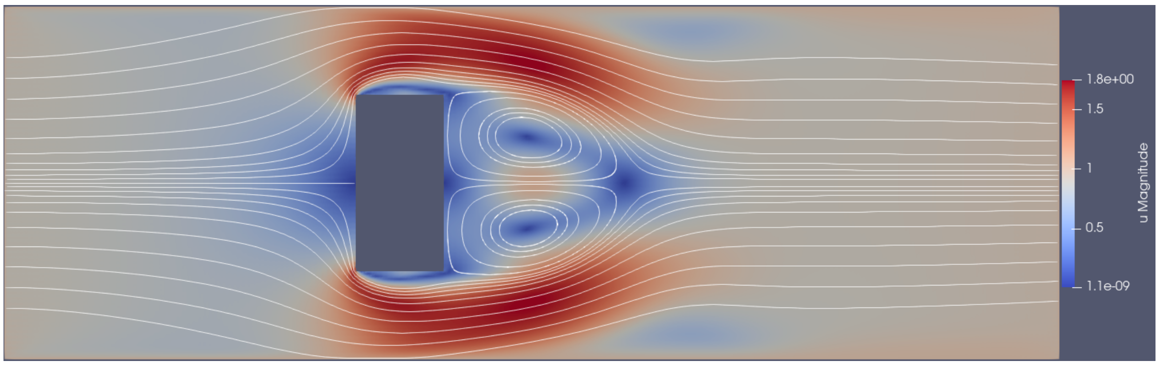


Figure 6.8: Flow over a rectangular obstacle

at $M_{\text{ref}} = 0.5$. At the left boundary, Dirichlet conditions $\rho = 1.4$, $u = (0.9, 0)^T$, $p = 1$ are imposed. The right boundary is a do_nothing boundary. The top and bottom are set as free slip boundaries. The computational domain consists of 147456 cells. The simulation was carried out to final time $T = 10.5$ with CFL $= 0.98$. The flow in the numerical solution contains a vortex street in the wake of the cylinder. To visualize the vortices that move with the flow, we compute the z-component of the vorticity (curl of the stream velocity):

$$v_z = (\nabla \times (u^T, 0)^T)_3 = \partial_{x_1} u_2 - \partial_{x_2} u_1$$

Figure 6.9 shows the vortices alternating between rotating clockwise (blue, negative v_z) and counter-clockwise (red, positive v_z). Streamlines are also plotted. It should be noted that

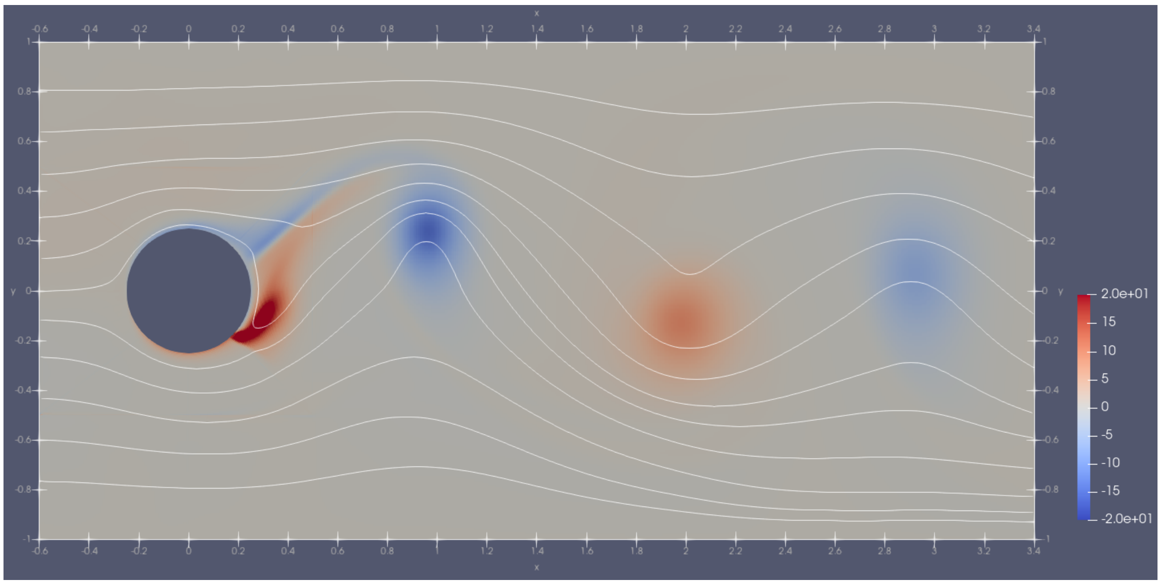


Figure 6.9: Flow past a cylinder with vortex street

the numerical solution on the given (comparatively coarse) grid here still depends a lot on the numerical viscosity.

Chapter 7

Conclusion

To summarize this thesis: A first order invariant domain preserving continuous finite element (in space) method was adapted to the low Mach case by splitting into explicit and implicit parts (IMEX). Deriving the scheme included determining wave speeds of a particular Riemann problem. The obtained scheme was shown under certain conditions to be conservative, consistent and positivity preserving of the mass density. The behavior of contact discontinuities was considered and also numerically tested. The method was compared to the unsplit scheme for the Euler equations, on which it is based. It was found that the dissipation of the density component is significantly reduced for small Mach numbers. For some problems like the Kelvin-Helmholtz instability at low reference Mach numbers, the unsplit scheme did not produce any meaningful data at the given resolution. This improvement comes at the cost of having to solve an implicit problem and the fact that the acoustic waves are found to be less accurate and more oscillatory in the IMEX solution.
Besides this, the method is only first order accurate, which causes it to have a rather large numerical viscosity. A prospect for the future would be to extend the IMEX scheme to second order. Such an extension could be based on an unsplit second order invariant domain preserving method like the one given in [10].

Appendix A

Appendix

A.1 Python code for a one-dimensional periodic grid

Listing A.1: 1D periodic IMEX

```
import numpy as np
from scipy.sparse.linalg import bicgstab

# indices for r: density, u: velocity, p: pressure
#            m: momentum-density, e: energy-density
idr=0
idu=1
idp=2
idm=1
ide=2

# Simulation class for solving the 1 dimensional (scaled) Euler equations
class Simulation:
    # initialize simulation with num-1 grid-cells of size dx and standard values
    def __init__(self, num, dx, r=1., u=1., p=1., gamma=1.4, M=1.):
        self.num = num
        self.gamma = gamma
        self.dx = dx
        self.t = 0
        self.M = M
        self.M2 = M*M

        prim = np.zeros((num+1,3))
        prim[:,idr] = r
        prim[:,idu] = u
        prim[:,idp] = p
        self.cons = self.getConservative(prim)

    # set new constant value data
    def setValues(self, r=1., u=1., p=1.):
        prim = np.zeros((self.num+1,3))
        prim[:,idr] = r
        prim[:,idu] = u
        prim[:,idp] = p
        self.cons = self.getConservative(prim)

    # perform one time step
    def iterate(self, CFL=0.5, picard_iterations=3, dt=None):
        num = self.num
        gamma = self.gamma
        dx = self.dx
        M2 = self.M2
        cons = self.cons

        cons[0,:] = cons[-2,:]
        cons[-1,:] = cons[1,:]
        prim = self.getPrimitive(cons)
        amax = np.max(np.sqrt(gamma*prim[:,idp]/prim[:,idr]))
        u = np.abs(prim[:,idu])
        lambdamax = np.maximum(u[:-1],u[1:])
        indexu = prim[:-1,idu] > prim[1:,idu]
        indexr = np.abs(prim[:-1,idr] - prim[1:,idr]) < 1.e-11
        index = np.logical_and(indexu, np.logical_not(indexr))
        lambdamax[index] = np.abs((prim[:-1,idr]*prim[:-1,idu] -
            prim[1:,idr]*prim[1:,idu] - np.sqrt(prim[:-1,idr]*prim[1:,idr])*
            (prim[:-1,idu]-prim[1:,idu]))[index]/
            (prim[:-1,idr]-prim[1:,idr])[index])
        index = np.logical_and(indexu, indexr)
        lambdamax[index] = 0.5*np.abs(prim[:-1,idu]+prim[1:,idu])[index]
        d = 0.5*lambdamax.reshape((-1,1))
        dii = d[:-1]+d[1:]
        if dt is None:
            dt = CFL*dx/max(np.max(dii), 0.5*amax)
        flux = self.getFluxEx(prim)
        cons[1:-1] += (0.5*(flux[:-2,:]-flux[2:,:]) + cons[:-2,:]*d[:-1] +
            cons[2:,:]*d[1:] - cons[1:-1,:]*dii)*dt/dx

        prim = self.getPrimitive(cons)
        matrix = np.zeros((num-1,num-1))
        b = np.zeros(num-1)
        c = 0.5*dt/dx*gamma*prim[:,idp]/prim[:,idr]
        for i in range(picard_iterations):
            index = np.arange(num-1)
            matrix[index,index] = 1. + 0.5*dt/dx/M2*(c[:-2]+c[2:])
            matrix[index,(index+num-3)%(num-1)] = -0.5*dt/dx/M2*c[:-2]
            matrix[index,(index+2)%(num-1)] = -0.5*dt/dx/M2*c[2:]
            b[index] = (gamma-1)*(cons[1:-1,ide] - 0.5*M2*prim[1:-1,idr]*
                prim[1:-1,idu]**2) + cons[:-2,idm]*c[:-2] - cons[2:,idm]*c[2:]
            prim[1:-1,idp] = bicgstab(matrix,b)[0]
            prim[0,idp] = prim[-2,idp]
            prim[-1,idp] = prim[1,idp]
            prim[1:-1,idu] = (cons[1:-1,idm] + 0.5*dt/dx*
                (prim[:-2,idp]-prim[2:,idp])/M2)/prim[1:-1,idr]
            prim[0,idu] = prim[-2,idu]
            prim[-1,idu] = prim[1,idu]
        self.cons = self.getConservative(prim)
        self.t += dt

    # explicit flux function
    def getFluxEx(self, prim):
        flux = np.zeros((len(prim),3))

        flux[:,idr] = prim[:,idr]*prim[:,idu]
        flux[:,idm] = prim[:,idr]*prim[:,idu]**2
        flux[:,ide] = 0.5*self.M2*prim[:,idr]*prim[:,idu]**3

        return flux

    # primitive variables (r,u,p)
    def getPrimitive(self, cons):
        prim = np.zeros((len(cons),3))

        prim[:,idr] = cons[:,idr]
        prim[:,idu] = cons[:,idm]/cons[:,idr]
        prim[:,idp] = (self.gamma-1)* \
            (cons[:,ide] - 0.5*self.M2*cons[:,idm]*prim[:,idu])

        return prim

    # conservative variables (r,m,e)
    def getConservative(self, prim):
        cons=np.zeros((len(prim),3))

        cons[:,idr] = prim[:,idr]
        cons[:,idm] = prim[:,idr]*prim[:,idu]
        cons[:,ide] = prim[:,idp]/(self.gamma-1) + 0.5*self.M2*prim[:,idr]* \
            (prim[:,idu]**2)

        return cons
```

A.2 IMEX extension of step-69 code

Listing A.2: 2D IMEX

```
#include <deal.II/base/conditional_ostream.h>
#include <deal.II/base/parallel.h>
#include <deal.II/base/parameter_acceptor.h>
#include <deal.II/base/partitioner.h>
#include <deal.II/base/quadrature.h>
#include <deal.II/base/timer.h>
#include <deal.II/base/work_stream.h>

#include <deal.II/distributed/tria.h>

#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_renumbering.h>
#include <deal.II/dofs/dof_tools.h>

#include <deal.II/fe/fe.h>
#include <deal.II/fe/fe_q.h>
#include <deal.II/fe/fe_values.h>
```

```
#include <deal.II/fe/mapping.h>
#include <deal.II/fe/mapping_q.h>

#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/manifold_lib.h>

#include <deal.II/lac/dynamic_sparsity_pattern.h>
#include <deal.II/lac/la_parallel_vector.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/lac/sparse_matrix.templates.h>
#include <deal.II/lac/vector.h>

#include <deal.II/meshworker/scratch_data.h>

#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/vector_tools.h>

#include <deal.II/lac/petsc_vector.h>
#include <deal.II/lac/petsc_sparse_matrix.h>
#include <deal.II/lac/petsc_solver.h>
#include <deal.II/lac/petsc_precondition.h>

#include <boost/archive/binary_iarchive.hpp>
#include <boost/archive/binary_oarchive.hpp>

#include <deal.II/base/std_cxx20/iota_view.h>
#include <boost/range/iterator_range.hpp>

#include <cmath>
#include <fstream>
#include <future>


namespace Step69
{
  using namespace dealii;

  namespace Boundaries
  {
    constexpr types::boundary_id do_nothing = 0;
    constexpr types::boundary_id free_slip  = 1;
    constexpr types::boundary_id dirichlet  = 2;
  }

  template <int dim>
  class Discretization : public ParameterAcceptor
  {
  public:
    Discretization(const MPI_Comm      mpi_communicator,
                   TimerOutput &       computing_timer,
                   const std::string &subsection = "Discretization");

    void setup();

    const MPI_Comm mpi_communicator;

    parallel::distributed::Triangulation<dim> triangulation;

    const MappingQ<dim>   mapping;
    const FE_Q<dim>       finite_element;
    const QGauss<dim>     quadrature;
    const QGauss<dim - 1> face_quadrature;
```

```
private:
  TimerOutput &computing_timer;

  double length;
  double height;
  double disk_position;
  double disk_diameter;

  unsigned int refinement;
};

template <int dim>
class OfflineData : public ParameterAcceptor
{
public:
  using BoundaryNormalMap =
    std::map<types::global_dof_index,
             std::tuple<Tensor<1, dim>, types::boundary_id, Point<dim>>>;

  OfflineData(const MPI_Comm             mpi_communicator,
              TimerOutput &              computing_timer,
              const Discretization<dim> &discretization,
              const std::string &        subsection = "OfflineData");

  void setup();
  void assemble();

  DoFHandler<dim> dof_handler;

  std::shared_ptr<const Utilities::MPI::Partitioner> partitioner;

  unsigned int n_locally_owned;
  unsigned int n_locally_relevant;

  IndexSet locally_owned_system;

  SparsityPattern sparsity_pattern;
  DynamicSparsityPattern sparsity_pattern_system;

  BoundaryNormalMap boundary_normal_map;

  SparseMatrix<double>                  lumped_mass_matrix;
  std::array<SparseMatrix<double>, dim> cij_matrix;
  std::array<SparseMatrix<double>, dim> nij_matrix;
  SparseMatrix<double>                  norm_matrix;

private:
  const MPI_Comm mpi_communicator;
  TimerOutput &  computing_timer;

  SmartPointer<const Discretization<dim>> discretization;
};

template <int dim>
class ProblemDescription : public ParameterAcceptor
{
public:
  static constexpr unsigned int problem_dimension = 2 + dim;

  using state_type = Tensor<1, problem_dimension>;
  using flux_type  = Tensor<1, problem_dimension, Tensor<1, dim>>;
  ProblemDescription(const std::string &subsection="ProblemDescription");

  const static std::array<std::string, problem_dimension> component_names;

  static constexpr double gamma = 7. / 5.;
  static double Mref;
  static double M2;

  static DEAL_II_ALWAYS_INLINE inline Tensor<1, dim>
  momentum(const state_type &U);

  static DEAL_II_ALWAYS_INLINE inline double
  internal_energy(const state_type &U);

  static DEAL_II_ALWAYS_INLINE inline double pressure(const state_type &U);

  static DEAL_II_ALWAYS_INLINE inline double
  speed_of_sound(const state_type &U);

  static DEAL_II_ALWAYS_INLINE inline flux_type flux(const state_type &U);

  static DEAL_II_ALWAYS_INLINE inline double
  compute_lambda_max(const state_type &     U_i,
                     const state_type &     U_j,
                     const Tensor<1, dim> &n_ij);
};

template <int dim>
class InitialValues : public ParameterAcceptor
{
public:
  using state_type = typename ProblemDescription<dim>::state_type;

  InitialValues(const std::string &subsection = "InitialValues");

  std::function<state_type(const Point<dim> &point, double t)> initial_state;

private:
  // We declare a private callback function that will be wired up to the
  // ParameterAcceptor::parse_parameters_call_back signal.
  void parse_parameters_callback();

  Tensor<1, dim> initial_direction;
  Tensor<1, 3>   initial_1d_state;
};

template <int dim>
class TimeStepping : public ParameterAcceptor
{
public:
  static constexpr unsigned int problem_dimension =
    ProblemDescription<dim>::problem_dimension;

  using state_type = typename ProblemDescription<dim>::state_type;
  using flux_type  = typename ProblemDescription<dim>::flux_type;

  using vector_type =
    std::array<LinearAlgebra::distributed::Vector<double>, problem_dimension>;

  TimeStepping(const MPI_Comm                     mpi_communicator,
               TimerOutput &                      computing_timer,
            const OfflineData<dim> &        offline_data,
            const InitialValues<dim> &      initial_values,
            const std::string &             subsection = "TimeStepping");

    void prepare();

    double make_one_step(vector_type &U, double t, unsigned int picard_iterations=2,
            double implicit_tolerance=1.e-8);

  private:
    const MPI_Comm mpi_communicator;
    TimerOutput &  computing_timer;

    SmartPointer<const OfflineData<dim>>   offline_data;
    SmartPointer<const InitialValues<dim>> initial_values;
    SmartPointer<const OfflineData<dim>>   problem_description;

    SparseMatrix<double> dij_matrix;

    PETScWrappers::MPI::SparseMatrix system_matrix;
    PETScWrappers::MPI::Vector system_solution;
    PETScWrappers::MPI::Vector system_rhs;

    vector_type temporary_vector;
    std::array<LinearAlgebra::distributed::Vector<double>, dim> momentum;

    double cfl_update;
  };

  template <int dim>
  class SchlierenPostprocessor : public ParameterAcceptor
  {
  public:
    static constexpr unsigned int problem_dimension =
      ProblemDescription<dim>::problem_dimension;

    using state_type = typename ProblemDescription<dim>::state_type;

    using vector_type =
      std::array<LinearAlgebra::distributed::Vector<double>, problem_dimension>;

    SchlierenPostprocessor(
      const MPI_Comm            mpi_communicator,
      TimerOutput &             computing_timer,
      const OfflineData<dim> &offline_data,
      const std::string &       subsection = "SchlierenPostprocessor");

    void prepare();

    void compute_schlieren(const vector_type &U);

    LinearAlgebra::distributed::Vector<double> schlieren;

  private:
    const MPI_Comm mpi_communicator;
    TimerOutput &  computing_timer;

    SmartPointer<const OfflineData<dim>> offline_data;

    Vector<double> r;

    unsigned int schlieren_index;
  double        schlieren_beta;
};

template <int dim>
class MainLoop : public ParameterAcceptor
{
public:
  using vector_type = typename TimeStepping<dim>::vector_type;

  MainLoop(const MPI_Comm mpi_communnicator);

  void run();

private:
  vector_type interpolate_initial_values(const double t = 0);

  void output(const vector_type &U,
              const std::string &name,
              double             t,
              unsigned int       cycle,
              bool               checkpoint = false);

  const MPI_Comm      mpi_communicator;
  std::ostringstream timer_output;
  TimerOutput         computing_timer;

  ConditionalOStream pcout;

  std::string base_name;
  double      t_final;
  double      output_granularity;

  bool asynchronous_writeback;

  bool resume;

  Discretization<dim>          discretization;
  OfflineData<dim>             offline_data;
  ProblemDescription<dim>      problem_description;
  InitialValues<dim>           initial_values;
  TimeStepping<dim>            time_stepping;
  SchlierenPostprocessor<dim> schlieren_postprocessor;

  vector_type output_vector;

  std::future<void> background_thread_state;
};

template <int dim>
Discretization<dim>::Discretization(const MPI_Comm      mpi_communicator,
                                    TimerOutput &       computing_timer,
                                    const std::string &subsection)
  : ParameterAcceptor(subsection)
  , mpi_communicator(mpi_communicator)
  , triangulation(mpi_communicator)
  , mapping(1)
  , finite_element(1)
  , quadrature(3)
  , face_quadrature(3)
  , computing_timer(computing_timer)
{
  // Parameters specifying grid geometry for flow around cylinder (if used)
  length = 4.;
  add_parameter("length", length, "Length of computational domain");

  height = 2.;
  add_parameter("height", height, "Height of computational domain");

  disk_position = 0.6;
  add_parameter("object position",
                disk_position,
                "x position of immersed disk center point");

  disk_diameter = 0.5;
  add_parameter("object diameter",
                disk_diameter,
                "Diameter of immersed disk");

  refinement = 5;
  add_parameter("refinement",
                refinement,
                "Number of refinement steps of the geometry");
}

template <int dim>
void Discretization<dim>::setup()
{
  TimerOutput::Scope scope(computing_timer, "discretization - setup");

  // Set up cartesian grid
  triangulation.clear();
  GridGenerator::subdivided_hyper_rectangle(
      triangulation, {16,4}, Point<2>(0.,1.), Point<2>(4.,0.));
  for(const auto &cell : triangulation.active_cell_iterators()){
    for(const auto f : cell->face_indices()){
      const auto face = cell->face(f);

      if(face->at_boundary()) face->set_boundary_id(Boundaries::dirichlet);
    }
  }
  triangulation.refine_global(refinement);
}

template <int dim>
OfflineData<dim>::OfflineData(const MPI_Comm             mpi_communicator,
                              TimerOutput &              computing_timer,
                              const Discretization<dim> &discretization,
                              const std::string &        subsection)
  : ParameterAcceptor(subsection)
  , dof_handler(discretization.triangulation)
  , mpi_communicator(mpi_communicator)
  , computing_timer(computing_timer)
  , discretization(&discretization)
{}

template <int dim>
void OfflineData<dim>::setup()
{
  IndexSet locally_owned;
  IndexSet locally_relevant;

  {
    TimerOutput::Scope scope(computing_timer,
                              "offline_data_-_distribute_dofs");

    dof_handler.distribute_dofs(discretization->finite_element);

    locally_owned    = dof_handler.locally_owned_dofs();
    n_locally_owned = locally_owned.n_elements();

    DoFTools::extract_locally_relevant_dofs(dof_handler, locally_relevant);
    n_locally_relevant = locally_relevant.n_elements();

    partitioner =
      std::make_shared<Utilities::MPI::Partitioner>(locally_owned,
                                                    locally_relevant,
                                                    mpi_communicator);
  }

  {
    TimerOutput::Scope scope(
      computing_timer,
      "offline_data_-_create_sparsity_pattern_and_set_up_matrices");

    DynamicSparsityPattern dsp(n_locally_relevant, n_locally_relevant);
    sparsity_pattern_system.reinit(n_locally_relevant, n_locally_relevant);
    SparsityPattern sparsity_system;

    const auto dofs_per_cell = discretization->finite_element.n_dofs_per_cell();
    std::vector<types::global_dof_index> dof_indices(dofs_per_cell);
    std::vector<types::global_dof_index> dof_indices_system(dofs_per_cell);

    for (const auto &cell : dof_handler.active_cell_iterators())
      {
        if (cell->is_artificial())
          continue;

        cell->get_dof_indices(dof_indices);
        cell->get_dof_indices(dof_indices_system);

        std::transform(dof_indices.begin(),
                       dof_indices.end(),
                       dof_indices.begin(),
                       [&](types::global_dof_index index) {
                         return partitioner->global_to_local(index);
                       });

        for(const auto dof : dof_indices)
              dsp.add_entries(dof, dof_indices.begin(), dof_indices.end());
        for(const auto dof : dof_indices_system)
              sparsity_pattern_system.add_entries(dof,
                      dof_indices_system.begin(), dof_indices_system.end());
      }

    sparsity_pattern.copy_from(dsp);
    sparsity_system.copy_from(sparsity_pattern_system);

    locally_owned_system = locally_owned;

    for(const auto i : locally_owned){
      dof_indices_system.clear();
      const auto dof = partitioner->local_to_global(i);
      for(auto jt=sparsity_system.begin(dof);jt != sparsity_system.end(dof);++jt){
        const auto col = jt->column();
        for(auto jt2=sparsity_system.begin(col);jt2 != sparsity_system.end(col);++jt2)
```

```
                dof_indices_system.push_back(jt2->column());
        }
        sparsity_pattern_system.add_entries(dof,
                dof_indices_system.begin(), dof_indices_system.end());
      }
      SparsityTools::distribute_sparsity_pattern(sparsity_pattern_system, locally_owned,
                mpi_communicator, locally_relevant);

      lumped_mass_matrix.reinit(sparsity_pattern);
      norm_matrix.reinit(sparsity_pattern);
      for (auto &matrix : cij_matrix)
        matrix.reinit(sparsity_pattern);
      for (auto &matrix : nij_matrix)
        matrix.reinit(sparsity_pattern);
    }
  }

  namespace
  {

    template <int dim>
    struct CopyData
    {
      bool                                          is_artificial;
      std::vector<types::global_dof_index>          local_dof_indices;
      typename OfflineData<dim>::BoundaryNormalMap  local_boundary_normal_map;
      FullMatrix<double>                            cell_lumped_mass_matrix;
      std::array<FullMatrix<double>, dim>           cell_cij_matrix;
    };

    template <typename IteratorType>
    DEAL_II_ALWAYS_INLINE inline SparseMatrix<double>::value_type
    get_entry(const SparseMatrix<double> &matrix, const IteratorType &it)
    {
      const SparseMatrix<double>::const_iterator matrix_iterator(
        &matrix, it->global_index());
      return matrix_iterator->value();
    }

    template <typename IteratorType>
    DEAL_II_ALWAYS_INLINE inline void
    set_entry(SparseMatrix<double> &            matrix,
              const IteratorType &              it,
              SparseMatrix<double>::value_type  value)
    {
      SparseMatrix<double>::iterator matrix_iterator(&matrix,
                                                     it->global_index());
      matrix_iterator->value() = value;
    }

    template <std::size_t k, typename IteratorType>
    DEAL_II_ALWAYS_INLINE inline Tensor<1, k>
    gather_get_entry(const std::array<SparseMatrix<double>, k> &c_ij,
                     const IteratorType                           it)
    {
      Tensor<1, k> result;
      for (unsigned int j = 0; j < k; ++j)
        result[j] = get_entry(c_ij[j], it);
      return result;
    }

    template <std::size_t k>
  DEAL_II_ALWAYS_INLINE inline Tensor<1, k>
  gather(const std::array<SparseMatrix<double>, k> &n_ij,
         const unsigned int                          i,
         const unsigned int                          j)
  {
    Tensor<1, k> result;
    for (unsigned int l = 0; l < k; ++l)
      result[l] = n_ij[l](i, j);
    return result;
  }

  template <std::size_t k>
  DEAL_II_ALWAYS_INLINE inline Tensor<1, k>
  gather(const std::array<LinearAlgebra::distributed::Vector<double>, k> &U,
         const unsigned int                                                 i)
  {
    Tensor<1, k> result;
    for (unsigned int j = 0; j < k; ++j)
      result[j] = U[j].local_element(i);
    return result;
  }

  template <std::size_t k, int k2>
  DEAL_II_ALWAYS_INLINE inline void
  scatter(std::array<LinearAlgebra::distributed::Vector<double>, k> &U,
          const Tensor<1, k2> &                                     tensor,
          const unsigned int                                        i)
  {
    static_assert(k == k2,
                  "The dimensions of the input arguments must agree");
    for (unsigned int j = 0; j < k; ++j)
      U[j].local_element(i) = tensor[j];
  }
} // namespace

template <int dim>
void OfflineData<dim>::assemble()
{
  lumped_mass_matrix = 0.;
  norm_matrix        = 0.;
  for (auto &matrix : cij_matrix)
    matrix = 0.;
  for (auto &matrix : nij_matrix)
    matrix = 0.;

  unsigned int dofs_per_cell =
    discretization->finite_element.n_dofs_per_cell();
  unsigned int n_q_points = discretization->quadrature.size();

  // What follows is the initialization of the scratch data required by
  // WorkStream

  MeshWorker::ScratchData<dim> scratch_data(
    discretization->mapping,
    discretization->finite_element,
    discretization->quadrature,
    update_values | update_gradients | update_quadrature_points |
      update_JxW_values,
    discretization->face_quadrature,
    update_normal_vectors | update_values | update_JxW_values);

  {
```

```
TimerOutput::Scope scope(
  computing_timer,
  "offline_data - assemble lumped mass matrix, and c_ij");

const auto local_assemble_system = //
  [&](const typename DoFHandler<dim>::cell_iterator &cell,
      MeshWorker::ScratchData<dim> &                 scratch,
      CopyData<dim> &                                copy) {
    copy.is_artificial = cell->is_artificial();
    if (copy.is_artificial)
      return;

    copy.local_boundary_normal_map.clear();
    copy.cell_lumped_mass_matrix.reinit(dofs_per_cell, dofs_per_cell);
    for (auto &matrix : copy.cell_cij_matrix)
      matrix.reinit(dofs_per_cell, dofs_per_cell);

    const auto &fe_values = scratch.reinit(cell);

    copy.local_dof_indices.resize(dofs_per_cell);
    cell->get_dof_indices(copy.local_dof_indices);

    std::transform(copy.local_dof_indices.begin(),
                   copy.local_dof_indices.end(),
                   copy.local_dof_indices.begin(),
                   [&](types::global_dof_index index) {
                     return partitioner->global_to_local(index);
                   });

    // We compute the local contributions for the lumped mass matrix
    // entries $m_i$ and and vectors $c_{ij}$ in the usual fashion:
    for (unsigned int q_point = 0; q_point < n_q_points; ++q_point)
      {
        const auto JxW = fe_values.JxW(q_point);

        for (unsigned int j = 0; j < dofs_per_cell; ++j)
          {
            const auto value_JxW =
              fe_values.shape_value(j, q_point) * JxW;
            const auto grad_JxW = fe_values.shape_grad(j, q_point) * JxW;

            copy.cell_lumped_mass_matrix(j, j) += value_JxW;

            for (unsigned int i = 0; i < dofs_per_cell; ++i)
              {
                const auto value = fe_values.shape_value(i, q_point);
                for (unsigned int d = 0; d < dim; ++d)
                  copy.cell_cij_matrix[d](i, j) += value * grad_JxW[d];

              } /* i */
          }     /* j */
      }         /* q */

    for (const auto f : cell->face_indices())
      {
        const auto face = cell->face(f);
        const auto id   = face->boundary_id();

        if (!face->at_boundary())
          continue;

        const auto &fe_face_values = scratch.reinit(cell, f);
        const unsigned int n_face_q_points =
          fe_face_values.get_quadrature().size();

        for (unsigned int j = 0; j < dofs_per_cell; ++j)
          {
            if (!discretization->finite_element.has_support_on_face(j, f))
              continue;
            Tensor<1, dim> normal;
            if (id == Boundaries::free_slip)
              {
                for (unsigned int q = 0; q < n_face_q_points; ++q)
                  normal += fe_face_values.normal_vector(q) *
                            fe_face_values.shape_value(j, q);
              }

            const auto index = copy.local_dof_indices[j];

            Point<dim> position;
            for (const auto v : cell->vertex_indices())
              if (cell->vertex_dof_index(v, 0) ==
                  partitioner->local_to_global(index))
                {
                  position = cell->vertex(v);
                  break;
                }

            const auto old_id =
              std::get<1>(copy.local_boundary_normal_map[index]);
            copy.local_boundary_normal_map[index] =
              std::make_tuple(normal, std::max(old_id, id), position);
          }
      }
  };

// Last, we provide a copy_local_to_global function as required for
// the WorkStream
const auto copy_local_to_global = [&](const CopyData<dim> &copy) {
  if (copy.is_artificial)
    return;

  for (const auto &it : copy.local_boundary_normal_map)
    {
      std::get<0>(boundary_normal_map[it.first]) +=
        std::get<0>(it.second);
      std::get<1>(boundary_normal_map[it.first]) =
        std::max(std::get<1>(boundary_normal_map[it.first]),
                 std::get<1>(it.second));
      std::get<2>(boundary_normal_map[it.first]) = std::get<2>(it.second);
    }

  lumped_mass_matrix.add(copy.local_dof_indices,
                         copy.cell_lumped_mass_matrix);

  for (int k = 0; k < dim; ++k)
    {
      cij_matrix[k].add(copy.local_dof_indices, copy.cell_cij_matrix[k]);
      nij_matrix[k].add(copy.local_dof_indices, copy.cell_cij_matrix[k]);
    }
};

WorkStream::run(dof_handler.begin_active(),
                    dof_handler.end(),
                    local_assemble_system,
                    copy_local_to_global,
                    scratch_data,
                    CopyData<dim>());
  }

  {
    TimerOutput::Scope scope(computing_timer,
                             "offline_data - compute |c_ij|, and n_ij");

    const std_cxx20::ranges::iota_view<unsigned int, unsigned int> indices(
      0, n_locally_relevant);

    const auto on_subranges = //
      [&](const auto i1, const auto i2) {
        for (const auto row_index :
             std_cxx20::ranges::iota_view<unsigned int, unsigned int>(*i1,
                                                                       *i2))
          {
            // First column-loop: we compute and store the entries of the
            // matrix norm_matrix and write normalized entries into the
            // matrix nij_matrix:
            std::for_each(
              sparsity_pattern.begin(row_index),
              sparsity_pattern.end(row_index),
              [&](const dealii::SparsityPatternIterators::Accessor &jt) {
                const auto   c_ij = gather_get_entry(cij_matrix, &jt);
                const double norm = c_ij.norm();

                set_entry(norm_matrix, &jt, norm);
                for (unsigned int j = 0; j < dim; ++j)
                  set_entry(nij_matrix[j], &jt, c_ij[j] / norm);
              });
          }
      };

    parallel::apply_to_subranges(indices.begin(),
                                 indices.end(),
                                 on_subranges,
                                 4096);

    for (auto &it : boundary_normal_map)
      {
        auto &normal = std::get<0>(it.second);
        normal /= (normal.norm() + std::numeric_limits<double>::epsilon());
      }
  }
}

template <int dim>
double ProblemDescription<dim>::Mref = 1.;

template <int dim>
double ProblemDescription<dim>::M2 = 1.;

template <int dim>
ProblemDescription<dim>::ProblemDescription(const std::string &subsection) :
              ParameterAcceptor(subsection){
  add_parameter("Mref", Mref, "Reference Mach number");
}
```

```
template <int dim>
DEAL_II_ALWAYS_INLINE inline Tensor<1, dim>
ProblemDescription<dim>::momentum(const state_type &U)
{
  Tensor<1, dim> result;
  std::copy_n(&U[1], dim, &result[0]);
  return result;
}

template <int dim>
DEAL_II_ALWAYS_INLINE inline double
ProblemDescription<dim>::internal_energy(const state_type &U)
{
  const double &rho = U[0];
  const auto     m   = momentum(U);
  const double &E   = U[dim + 1];
  return E - 0.5*M2*m.norm_square()/rho;
}

template <int dim>
DEAL_II_ALWAYS_INLINE inline double
ProblemDescription<dim>::pressure(const state_type &U)
{
  return (gamma - 1.) * internal_energy(U);
}

template <int dim>
DEAL_II_ALWAYS_INLINE inline double
ProblemDescription<dim>::speed_of_sound(const state_type &U)
{
  const double &rho = U[0];
  const double  p   = pressure(U);

  return std::sqrt(gamma * p / rho);
}

template <int dim>
DEAL_II_ALWAYS_INLINE inline typename ProblemDescription<dim>::flux_type
ProblemDescription<dim>::flux(const state_type &U)
{
  const double &rho = U[0];
  const auto     m   = momentum(U);

  flux_type result;

  result[0] = m;
  for (unsigned int i = 0; i < dim; ++i)
    {
      result[1 + i] = m * m[i] / rho;
    }
  result[dim + 1] = 0.5 * M2 * m.norm_square() / rho * (m/rho);

  return result;
}

namespace
{
  template <int dim>
  DEAL_II_ALWAYS_INLINE inline std::array<double, 2> riemann_data_from_state(
    const typename ProblemDescription<dim>::state_type U,
    const Tensor<1, dim> &                                n_ij)
  {
      Tensor<1, 3> projected_U;
      projected_U[0] = U[0];
      const auto m   = ProblemDescription<dim>::momentum(U);
      projected_U[1] = n_ij * m;

      const auto perpendicular_m = m - projected_U[1] * n_ij;
      projected_U[2] = U[1 + dim] - 0.5 * perpendicular_m.norm_square() / U[0];

      return {{projected_U[0], projected_U[1] / projected_U[0]}};
    }

    DEAL_II_ALWAYS_INLINE inline double positive_part(const double number)
    {
      return std::max(number, 0.);
    }


    DEAL_II_ALWAYS_INLINE inline double negative_part(const double number)
    {
      return -std::min(number, 0.);
    }
  } // namespace

  template <int dim>
  DEAL_II_ALWAYS_INLINE inline double
  ProblemDescription<dim>::compute_lambda_max(const state_type &    U_i,
                                              const state_type &    U_j,
                                              const Tensor<1, dim> &n_ij)
  {
    const auto riemann_data_i = riemann_data_from_state(U_i, n_ij);
    const auto riemann_data_j = riemann_data_from_state(U_j, n_ij);

    if(riemann_data_i[1] <= riemann_data_j[1])
        return std::max(std::max(std::abs(riemann_data_i[1]),
                std::abs(riemann_data_j[1])), 1.e-10);
    double diff = riemann_data_i[0] - riemann_data_j[0];
    if(std::abs(diff) > 1.e-12){
      auto mdot = std::sqrt(riemann_data_i[0]*riemann_data_j[0])*
                (riemann_data_i[1] - riemann_data_j[1]);
      return std::max(std::abs((riemann_data_i[0]*riemann_data_i[1] -
                riemann_data_j[0]*riemann_data_j[1] - mdot) / diff), 1.e-10);
    }
    return std::max(std::abs((riemann_data_i[1] + riemann_data_j[1])/2.), 1.e-10);
  }

  template <>
  const std::array<std::string, 3> ProblemDescription<1>::component_names{
    {"rho", "u", "p"}};

  template <>
  const std::array<std::string, 4> ProblemDescription<2>::component_names{
    {"rho", "u_1", "u_2", "p"}};

  template <>
  const std::array<std::string, 5> ProblemDescription<3>::component_names{
    {"rho", "u_1", "u_2", "u_3", "p"}};

  template <int dim>
  InitialValues<dim>::InitialValues(const std::string &subsection)
    : ParameterAcceptor(subsection)
  {
    /* We wire up the slot InitialValues<dim>::parse_parameters_callback to
```

```
      the ParameterAcceptor::parse_parameters_call_back signal: */
  ParameterAcceptor::parse_parameters_call_back.connect(
    std::bind(&InitialValues<dim>::parse_parameters_callback, this));

  initial_direction[0] = 1.;
  add_parameter("initial_direction",
                initial_direction,
                "Initial direction of the uniform flow field");

  initial_1d_state[0] = ProblemDescription<dim>::gamma;
  initial_1d_state[1] = 3.;
  initial_1d_state[2] = 1.;
  add_parameter("initial 1d state",
                initial_1d_state,
                "Initial 1d state (rho, u, p) of the uniform flow field");
}

template <int dim>
void InitialValues<dim>::parse_parameters_callback()
{
  AssertThrow(initial_direction.norm() != 0.,
              ExcMessage(
                "Initial direction is set to the zero vector."));
  initial_direction /= initial_direction.norm();

  // Initial state for contact discontinuity simulation
  initial_state = [this](const Point<dim> & point, double /*t*/) {
    const double           rho   = initial_1d_state[0];
    const double           u     = initial_1d_state[1];
    const double           p     = initial_1d_state[2];
    static constexpr double gamma = ProblemDescription<dim>::gamma;
    static double M2 = ProblemDescription<dim>::M2;

    state_type state;

    state[0] = rho;
    for(unsigned int i = 0; i < dim; ++i)
      state[1 + i] = rho*u*initial_direction[i];
    state[dim + 1] = p/(gamma - 1.) + 0.5 * M2*rho*u*u;

    if(point[0]>1 && point[0]<2 && point[1]>0.25 && point[1]<0.75){
      state[0] = 1.1*rho;
      for(unsigned int i = 0; i < dim; ++i)
        state[1 + i] = 1.1*rho*u*initial_direction[i];
      state[dim + 1] = p/(gamma - 1.) + 0.5*M2*1.1*rho*u*u;
    }

    return state;
  };
}

template <int dim>
TimeStepping<dim>::TimeStepping(
  const MPI_Comm                    mpi_communicator,
  TimerOutput &                     computing_timer,
  const OfflineData<dim> &          offline_data,
  const InitialValues<dim> &        initial_values,
  const std::string &               subsection /*= "TimeStepping"*/)
  : ParameterAcceptor(subsection)
  , mpi_communicator(mpi_communicator)
  , computing_timer(computing_timer)
  , offline_data(&offline_data)
  , initial_values(&initial_values)
{
  cfl_update = 0.80;
  add_parameter("cfl_update",
                cfl_update,
                "Relative CFL constant used for update");
}

template <int dim>
void TimeStepping<dim>::prepare()
{
  TimerOutput::Scope scope(computing_timer,
                           "time_stepping - prepare scratch space");

  for (auto &it : temporary_vector)
    it.reinit(offline_data->partitioner);

  for (auto &it : momentum)
    it.reinit(offline_data->partitioner);

  dij_matrix.reinit(offline_data->sparsity_pattern);

  system_matrix.reinit(offline_data->locally_owned_system,
      offline_data->locally_owned_system,
      offline_data->sparsity_pattern_system, mpi_communicator);
  system_solution.reinit(offline_data->locally_owned_system, mpi_communicator);
  system_rhs.reinit(offline_data->locally_owned_system, mpi_communicator);
}

template <int dim>
double TimeStepping<dim>::make_one_step(vector_type &U, double t,
              unsigned int picard_iterations, double implicit_tolerance)
{
  const auto &n_locally_owned    = offline_data->n_locally_owned;
  const auto &n_locally_relevant = offline_data->n_locally_relevant;

  const std_cxx20::ranges::iota_view<unsigned int, unsigned int>
    indices_owned(0, n_locally_owned);
  const std_cxx20::ranges::iota_view<unsigned int, unsigned int>
    indices_relevant(0, n_locally_relevant);

  const auto &sparsity = offline_data->sparsity_pattern;
  const auto &sparsity_system = offline_data->sparsity_pattern_system;

  const auto &lumped_mass_matrix = offline_data->lumped_mass_matrix;
  const auto &norm_matrix        = offline_data->norm_matrix;
  const auto &nij_matrix         = offline_data->nij_matrix;
  const auto &cij_matrix         = offline_data->cij_matrix;

  const auto &boundary_normal_map = offline_data->boundary_normal_map;

  {
    TimerOutput::Scope scope(computing_timer,
                             "time_stepping - 1 compute d_ij");

    const auto on_subranges = //
      [&](const auto i1, const auto i2) {
        for (const auto i :
             std_cxx20::ranges::iota_view<unsigned int, unsigned int>(*i1,
                                                                       *i2))
          {
            const auto U_i = gather(U, i);
          for (auto jt = sparsity.begin(i); jt != sparsity.end(i); ++jt)
            {
              const auto j = jt->column();
              if (j >= i)
                continue;

              const auto U_j = gather(U, j);

              const auto   n_ij = gather_get_entry(nij_matrix, jt);
              const double norm = get_entry(norm_matrix, jt);

              const auto lambda_max =
                ProblemDescription<dim>::compute_lambda_max(U_i, U_j, n_ij);

              double d = norm * lambda_max;

              if (boundary_normal_map.count(i) != 0 &&
                  boundary_normal_map.count(j) != 0)
                {
                  const auto n_ji = gather(nij_matrix, j, i);
                  const auto lambda_max_2 =
                    ProblemDescription<dim>::compute_lambda_max(U_j,
                                                                U_i,
                                                                n_ji);
                  const double norm_2 = norm_matrix(j, i);

                  d = std::max(d, norm_2 * lambda_max_2);
                }

              set_entry(dij_matrix, jt, d);
              dij_matrix(j, i) = d;
            }
        }
    };

  parallel::apply_to_subranges(indices_relevant.begin(),
                               indices_relevant.end(),
                               on_subranges,
                               4096);
}

std::atomic<double> tau_max{std::numeric_limits<double>::infinity()};

{
  TimerOutput::Scope scope(computing_timer,
                           "time_stepping␣-␣2␣compute␣d_ii,␣and␣tau_max");

  const auto on_subranges = //
    [&](const auto i1, const auto i2) {
      double tau_max_on_subrange = std::numeric_limits<double>::infinity();

      for (const auto i :
           std_cxx20::ranges::iota_view<unsigned int, unsigned int>(*i1,
                                                                      *i2))
        {
          double d_sum = 0.;

          for (auto jt = sparsity.begin(i); jt != sparsity.end(i); ++jt)
            {
              const auto j = jt->column();
            if (j == i)
              continue;

            d_sum -= get_entry(dij_matrix, jt);
          }

        dij_matrix.diag_element(i) = d_sum;
        const double mass   = lumped_mass_matrix.diag_element(i);
        const double tau    = cfl_update * mass / (-2. * d_sum);
        tau_max_on_subrange = std::min(tau_max_on_subrange, tau);
      }

    double current_tau_max = tau_max.load();
    while (current_tau_max > tau_max_on_subrange &&
           !tau_max.compare_exchange_weak(current_tau_max,
                                          tau_max_on_subrange))
      ;
  };

  parallel::apply_to_subranges(indices_relevant.begin(),
                               indices_relevant.end(),
                               on_subranges,
                               4096);

  tau_max.store(Utilities::MPI::min(tau_max.load(), mpi_communicator));

  AssertThrow(
    !std::isnan(tau_max.load()) && !std::isinf(tau_max.load()) &&
      tau_max.load() > 0.,
    ExcMessage(
      "I'm sorry, Dave. I'm afraid I can't do that. - We crashed."));
}

{
  TimerOutput::Scope scope(computing_timer,
                           "time_stepping - 3 perform explicit update");

  const auto on_subranges = //
    [&](const auto i1, const auto i2) {
      for (const auto i : boost::make_iterator_range(i1, i2))
        {
          Assert(i < n_locally_owned, ExcInternalError());

          const auto U_i = gather(U, i);

          const auto   f_i = ProblemDescription<dim>::flux(U_i);
          const double m_i = lumped_mass_matrix.diag_element(i);

          auto U_i_new = U_i;

          for (auto jt = sparsity.begin(i); jt != sparsity.end(i); ++jt)
            {
              const auto j = jt->column();

              const auto U_j = gather(U, j);
              const auto f_j = ProblemDescription<dim>::flux(U_j);

              const auto c_ij = gather_get_entry(cij_matrix, jt);
              const auto d_ij = get_entry(dij_matrix, jt);

              for (unsigned int k = 0; k < problem_dimension; ++k)
                {
                    U_i_new[k] +=
                      tau_max / m_i *
                      (-(f_j[k] - f_i[k]) * c_ij + d_ij * (U_j[k] - U_i[k]));
                  }
              }

            scatter(temporary_vector, U_i_new, i);
          }
      };

    parallel::apply_to_subranges(indices_owned.begin(),
                                 indices_owned.end(),
                                 on_subranges,
                                 4096);
  }

  // Step 4: Fix up boundary states.

  {
    TimerOutput::Scope scope(computing_timer,
                             "time_stepping␣-␣4␣fix␣boundary␣states");

    for (auto it : boundary_normal_map)
      {
        const auto i = it.first;

        if (i >= n_locally_owned)
          continue;

        const auto &normal   = std::get<0>(it.second);
        const auto &id       = std::get<1>(it.second);
        const auto &position = std::get<2>(it.second);

        auto U_i = gather(temporary_vector, i);

        // On free slip boundaries we remove the normal component of the
        // momentum:
        if (id == Boundaries::free_slip)
          {
            auto m = ProblemDescription<dim>::momentum(U_i);
            m -= (m * normal) * normal;
            for (unsigned int k = 0; k < dim; ++k)
              U_i[k + 1] = m[k];
          }

        // On Dirichlet boundaries we enforce initial conditions
        // strongly:
        else if (id == Boundaries::dirichlet)
          {
            U_i = initial_values->initial_state(position, t + tau_max);
          }

        scatter(temporary_vector, U_i, i);
      }
  }

  // Implicit step
  {
    TimerOutput::Scope scope(computing_timer, "time_stepping␣-␣implicit␣update");

    constexpr double gamma = ProblemDescription<dim>::gamma;
    const double &M2 = ProblemDescription<dim>::M2;
```

```
const auto on_subranges0 =
  [&](const auto i1, const auto i2){
    for (const auto i : boost::make_iterator_range(i1, i2)){
      const auto U_i = gather(temporary_vector, i);
      for(int k=0;k<dim;k++) momentum[k].local_element(i) = U_i[k+1];
      system_solution[offline_data->partitioner->local_to_global(i)] =
                ProblemDescription<dim>::pressure(U_i);
    }
  };
parallel::apply_to_subranges(indices_owned.begin(), indices_owned.end(),
          on_subranges0, 4096);
system_solution.compress(VectorOperation::insert);

const auto on_subranges =
  [&](const auto i1, const auto i2){
    std::vector<types::global_dof_index> col_indices;
    std::vector<double> values;
    std::unordered_map<types::global_dof_index, unsigned int> mapping;
    double rhs;

    for (const auto i : boost::make_iterator_range(i1, i2)){
      const auto row = offline_data->partitioner->local_to_global(i);
      const auto U_i = gather(temporary_vector, i);
      const auto p_i = system_solution[row];
      const auto ui = gather(momentum,i)/U_i[0];
      const double factor = tau_max / lumped_mass_matrix.diag_element(i);

      auto it = boundary_normal_map.find(i);
      if(it != boundary_normal_map.end() &&
                std::get<1>(it->second) == Boundaries::dirichlet){
        system_matrix.set(row,row,1.);
        system_rhs[row] = p_i;
      }
      else{
        col_indices.clear();
        values.clear();
        unsigned int c=0;
        for(auto ct = sparsity_system.begin(row); ct != sparsity_system.end(row); ++ct){
          const auto col = ct->column();
          col_indices.push_back(col);
          values.push_back(0.);
          mapping[col] = c;
          c++;
        }

        rhs = U_i[dim+1] - 0.5*M2*U_i[0]*ui.norm_square();
        values[mapping[row]] = 1./(gamma - 1.);

        for(auto jt = sparsity.begin(i); jt != sparsity.end(i); ++jt){
          const auto j = jt->column();
          const auto col = offline_data->partitioner->local_to_global(j);
          const auto c_ij = gather_get_entry(cij_matrix, jt);
          const auto U_j = gather(temporary_vector, j);
          const auto p_j = system_solution[col];
          const double factor_j = tau_max / lumped_mass_matrix.diag_element(j);

          rhs -= factor*gamma/(gamma-1.)*p_j/U_j[0]*
                          (c_ij*ProblemDescription<dim>::momentum(U_j));

          it = boundary_normal_map.find(j);
          for(auto kt = sparsity.begin(j); kt != sparsity.end(j); ++kt){
            const auto k = kt->column();
            auto c_jk = gather_get_entry(cij_matrix, kt);

            if(it != boundary_normal_map.end() &&
                       std::get<1>(it->second) == Boundaries::free_slip){
              const auto &normal = std::get<0>(it->second);
              c_jk -= (c_jk*normal)*normal;
            }

            values[mapping[offline_data->partitioner->local_to_global(k)]] -=
                       factor*factor_j*gamma/(gamma-1.)*(c_ij*c_jk)*p_j/U_j[0]/M2;
          }
        }
        system_rhs[row] = rhs;
        system_matrix.set(row, col_indices, values);
      }
    }
  };

for(unsigned int iter=0;iter < picard_iterations;iter++){
  for(auto &it : temporary_vector) it.update_ghost_values();
  for(auto &it : momentum) it.update_ghost_values();
  parallel::apply_to_subranges(indices_owned.begin(), indices_owned.end(),
          on_subranges, 4096);
  system_rhs.compress(VectorOperation::insert);
  system_matrix.compress(VectorOperation::insert);
  SolverControl solver_control(system_solution.size(), implicit_tolerance);
  PETScWrappers::PreconditionBlockJacobi preconditioner(system_matrix);
  PETScWrappers::SolverBicgstab solver(solver_control);
  solver.solve(system_matrix, system_solution, system_rhs, preconditioner);

  const auto on_subranges1 =
    [&](const auto i1, const auto i2){
      for(const auto i : boost::make_iterator_range(i1, i2)){
        auto it = boundary_normal_map.find(i);
        if(it != boundary_normal_map.end() &&
          std::get<1>(it->second) == Boundaries::dirichlet)
          continue;

        double factor = tau_max/lumped_mass_matrix.diag_element(i);
        const auto U_i = gather(temporary_vector,i);
        auto m = ProblemDescription<dim>::momentum(U_i);
        for(auto jt=sparsity.begin(i);jt != sparsity.end(i);++jt){
          const auto j = jt->column();
          auto c_ij = gather_get_entry(cij_matrix, jt);
          if(it != boundary_normal_map.end() &&
                 std::get<1>(it->second) == Boundaries::free_slip){
            const auto &normal = std::get<0>(it->second);
            c_ij -= (c_ij*normal)*normal;
          }
          const auto p_j =
                 system_solution[offline_data->partitioner->local_to_global(j)];
          m -= factor/M2*p_j*c_ij;
        }
        for(unsigned int k=0;k < dim;k++) momentum[k].local_element(i) = m[k];
      }
    };
  parallel::apply_to_subranges(indices_owned.begin(), indices_owned.end(),
                 on_subranges1, 4096);
}
const auto on_subranges2 =
      [&](const auto i1, const auto i2){
        for (const auto i : boost::make_iterator_range(i1, i2)){
          const auto rho_i = gather(temporary_vector,i)[0];
          const auto p_i =
              system_solution[offline_data->partitioner->local_to_global(i)];
          const auto m = gather(momentum,i);
          for(unsigned int k=0;k < dim;k++)
            temporary_vector[k+1].local_element(i) = m[k];
          temporary_vector[dim+1].local_element(i) = p_i/(gamma-1.) +
                      0.5*M2*m.norm_square()/rho_i;
        }
      };
    parallel::apply_to_subranges(indices_owned.begin(), indices_owned.end(),
              on_subranges2, 4096);
    for(auto &it : temporary_vector) it.update_ghost_values();
  }

  // We now update the ghost layer over all MPI ranks

  for (auto &it : temporary_vector)
    it.update_ghost_values();

  U.swap(temporary_vector);

  return tau_max;
}

template <int dim>
SchlierenPostprocessor<dim>::SchlierenPostprocessor(
  const MPI_Comm             mpi_communicator,
  TimerOutput &              computing_timer,
  const OfflineData<dim> &offline_data,
  const std::string &        subsection /*= "SchlierenPostprocessor"*/)
  : ParameterAcceptor(subsection)
  , mpi_communicator(mpi_communicator)
  , computing_timer(computing_timer)
  , offline_data(&offline_data)
{
  schlieren_beta = 10.;
  add_parameter("schlieren_beta",
                schlieren_beta,
                "Beta_factor_used_in_Schlieren-type_postprocessor");

  schlieren_index = 0;
  add_parameter("schlieren_index",
                schlieren_index,
                "Use_the_corresponding_component_of_the_state_vector_for_the_"
                "schlieren_plot");
}

template <int dim>
void SchlierenPostprocessor<dim>::prepare()
{
  TimerOutput::Scope scope(computing_timer,
                           "schlieren_postprocessor_-_prepare_scratch_space");

  r.reinit(offline_data->n_locally_relevant);
  schlieren.reinit(offline_data->partitioner);
}

template <int dim>
void SchlierenPostprocessor<dim>::compute_schlieren(const vector_type &U)
```

```
{
  TimerOutput::Scope scope(
    computing_timer, "schlieren_postprocessor - compute_schlieren_plot");

  const auto &sparsity            = offline_data->sparsity_pattern;
  const auto &lumped_mass_matrix  = offline_data->lumped_mass_matrix;
  const auto &cij_matrix          = offline_data->cij_matrix;
  const auto &boundary_normal_map = offline_data->boundary_normal_map;
  const auto &n_locally_owned     = offline_data->n_locally_owned;

  const auto indices =
    std_cxx20::ranges::iota_view<unsigned int, unsigned int>(0,
                                                   n_locally_owned);

  std::atomic<double> r_i_max{0.};
  std::atomic<double> r_i_min{std::numeric_limits<double>::infinity()};

  {
    const auto on_subranges = //
      [&](const auto i1, const auto i2) {
        double r_i_max_on_subrange = 0.;
        double r_i_min_on_subrange = std::numeric_limits<double>::infinity();

        for (const auto i : boost::make_iterator_range(i1, i2))
          {
            Assert(i < n_locally_owned, ExcInternalError());

            Tensor<1, dim> r_i;

            for (auto jt = sparsity.begin(i); jt != sparsity.end(i); ++jt)
              {
                const auto j = jt->column();

                if (i == j)
                  continue;

                const auto U_js = U[schlieren_index].local_element(j);
                const auto c_ij = gather_get_entry(cij_matrix, jt);
                r_i += c_ij * U_js;
              }

            const auto bnm_it = boundary_normal_map.find(i);
            if (bnm_it != boundary_normal_map.end())
              {
                const auto &normal = std::get<0>(bnm_it->second);
                const auto &id     = std::get<1>(bnm_it->second);

                if (id == Boundaries::free_slip)
                  r_i -= 1. * (r_i * normal) * normal;
                else
                  r_i = 0.;
              }

            const double m_i    = lumped_mass_matrix.diag_element(i);
            r[i]                = r_i.norm() / m_i;
            r_i_max_on_subrange = std::max(r_i_max_on_subrange, r[i]);
            r_i_min_on_subrange = std::min(r_i_min_on_subrange, r[i]);
          }

        double current_r_i_max = r_i_max.load();
        while (current_r_i_max < r_i_max_on_subrange &&
               !r_i_max.compare_exchange_weak(current_r_i_max,
                                                  r_i_max_on_subrange))
          ;

        double current_r_i_min = r_i_min.load();
        while (current_r_i_min > r_i_min_on_subrange &&
               !r_i_min.compare_exchange_weak(current_r_i_min,
                                                  r_i_min_on_subrange))
          ;
      };

    parallel::apply_to_subranges(indices.begin(),
                                 indices.end(),
                                 on_subranges,
                                 4096);
  }

  r_i_max.store(Utilities::MPI::max(r_i_max.load(), mpi_communicator));
  r_i_min.store(Utilities::MPI::min(r_i_min.load(), mpi_communicator));

  {
    const auto on_subranges = //
      [&](const auto i1, const auto i2) {
        for (const auto i : boost::make_iterator_range(i1, i2))
          {
            Assert(i < n_locally_owned, ExcInternalError());

            schlieren.local_element(i) =
              1. - std::exp(-schlieren_beta * (r[i] - r_i_min) /
                            (r_i_max - r_i_min));
          }
      };

    parallel::apply_to_subranges(indices.begin(),
                                 indices.end(),
                                 on_subranges,
                                 4096);
  }

  schlieren.update_ghost_values();
}

template <int dim>
MainLoop<dim>::MainLoop(const MPI_Comm mpi_communicator)
  : ParameterAcceptor("A␣-␣MainLoop")
  , mpi_communicator(mpi_communicator)
  , computing_timer(mpi_communicator,
                    timer_output,
                    TimerOutput::never,
                    TimerOutput::cpu_and_wall_times)
  , pcout(std::cout, Utilities::MPI::this_mpi_process(mpi_communicator) == 0)
  , discretization(mpi_communicator, computing_timer, "B␣-␣Discretization")
  , offline_data(mpi_communicator,
                 computing_timer,
                 discretization,
                 "C␣-␣OfflineData")
  , problem_description()
  , initial_values("D␣-␣InitialValues")
  , time_stepping(mpi_communicator,
                  computing_timer,
                  offline_data,
                  initial_values,
                  "E␣-␣TimeStepping")
  , schlieren_postprocessor(mpi_communicator,
                            computing_timer,
                            offline_data,
                            "F␣-␣SchlierenPostprocessor")
{
  base_name = "test";
  add_parameter("basename", base_name, "Base␣name␣for␣all␣output␣files");

  t_final = 4.;
  add_parameter("final␣time", t_final, "Final␣time");

  output_granularity = 0.02;
  add_parameter("output␣granularity",
                output_granularity,
                "time␣interval␣for␣output");

  asynchronous_writeback = true;
  add_parameter("asynchronous␣writeback",
                asynchronous_writeback,
                "Write␣out␣solution␣in␣a␣background␣thread␣performing␣IO");

  resume = false;
  add_parameter("resume", resume, "Resume␣an␣interrupted␣computation.");
}

namespace
{
  void print_head(ConditionalOStream &pcout,
                  const std::string & header,
                  const std::string & secondary = "")
  {
    const auto header_size   = header.size();
    const auto padded_header = std::string((34 - header_size) / 2, '␣') +
                               header +
                               std::string((35 - header_size) / 2, '␣');

    const auto secondary_size = secondary.size();
    const auto padded_secondary =
      std::string((34 - secondary_size) / 2, '␣') + secondary +
      std::string((35 - secondary_size) / 2, '␣');

    /* clang-format off */
    pcout << std::endl;
    pcout << "␣␣␣␣####################################################" << std::endl;
    pcout << "␣␣␣␣#########␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#########" << std::endl;
    pcout << "␣␣␣␣#########"     <<  padded_header   <<     "#########" << std::endl;
    pcout << "␣␣␣␣#########"     << padded_secondary <<     "#########" << std::endl;
    pcout << "␣␣␣␣#########␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#########" << std::endl;
    pcout << "␣␣␣␣####################################################" << std::endl;
    pcout << std::endl;
    /* clang-format on */
  }
} // namespace

template <int dim>
void MainLoop<dim>::run()
{

  pcout << "Reading␣parameters␣and␣allocating␣objects...␣" << std::flush;

  ParameterAcceptor::initialize("step-69.prm");
  problem_description.M2 = problem_description.Mref * problem_description.Mref;
```

```
pcout << "done" << std::endl;

{
  print_head(pcout, "create␣triangulation");
  discretization.setup();

  pcout << "Number␣of␣active␣cells:␣␣␣␣␣␣␣"
        << discretization.triangulation.n_global_active_cells()
        << std::endl;

  print_head(pcout, "compute␣offline␣data");
  offline_data.setup();
  offline_data.assemble();

  pcout << "Number␣of␣degrees␣of␣freedom:␣"
        << offline_data.dof_handler.n_dofs() << std::endl;

  print_head(pcout, "set␣up␣time␣step");
  time_stepping.prepare();
  schlieren_postprocessor.prepare();
  for(auto &it : output_vector) it.reinit(offline_data.partitioner);
}

double       t            = 0.;
unsigned int output_cycle = 0;

print_head(pcout, "interpolate␣initial␣values");
vector_type U = interpolate_initial_values();

if (resume)
  {
    print_head(pcout, "restore␣interrupted␣computation");

    const unsigned int i =
      discretization.triangulation.locally_owned_subdomain();

    const std::string name = base_name + "-checkpoint-" +
                             Utilities::int_to_string(i, 4) + ".archive";
    std::ifstream file(name, std::ios::binary);

    boost::archive::binary_iarchive ia(file);
    ia >> t >> output_cycle;

    for (auto &it1 : U)
      {
        for (auto &it2 : it1)
          ia >> it2;
        it1.update_ghost_values();
      }
  }

output(U, base_name, t, output_cycle++);

print_head(pcout, "enter␣main␣loop");

for (unsigned int cycle = 0; t < t_final; ++cycle)
  {

    std::ostringstream head;
    std::ostringstream secondary;

    head << "Cycle␣␣" << Utilities::int_to_string(cycle, 6) << "␣␣(" //
            << std::fixed << std::setprecision(1) << t / t_final * 100  //
            << "%)";
        secondary << "at time t = " << std::setprecision(8) << std::fixed << t;

        print_head(pcout, head.str(), secondary.str());

        t += time_stepping.make_one_step(U, t, 2, std::min(1.e-8,0.1*problem_description.M2));

        if (t > output_cycle * output_granularity)
          {
            output(U, base_name, t, output_cycle, true);
            ++output_cycle;
          }
      }

    if (background_thread_state.valid())
      background_thread_state.wait();

    computing_timer.print_summary();
    pcout << timer_output.str() << std::endl;
  }

  template <int dim>
  typename MainLoop<dim>::vector_type
  MainLoop<dim>::interpolate_initial_values(const double t)
  {
    pcout << "MainLoop<dim>::interpolate_initial_values(t = " << t << ")"
          << std::endl;
    TimerOutput::Scope scope(computing_timer,
                             "main_loop - setup scratch space");

    vector_type U;

    for (auto &it : U)
      it.reinit(offline_data.partitioner);

    constexpr auto problem_dimension =
      ProblemDescription<dim>::problem_dimension;

    for (unsigned int i = 0; i < problem_dimension; ++i)
      VectorTools::interpolate(offline_data.dof_handler,
                               ScalarFunctionFromFunctionObject<dim, double>(
                                 [&](const Point<dim> &x) {
                                   return initial_values.initial_state(x, t)[i];
                                 }),
                               U[i]);

    for (auto &it : U)
      it.update_ghost_values();

    return U;
  }

  template <int dim>
  void MainLoop<dim>::output(const typename MainLoop<dim>::vector_type &U,
                             const std::string &                        name,
                             const double                               t,
                             const unsigned int                         cycle,
                             const bool checkpoint)
  {
    pcout << "MainLoop<dim>::output(t = " << t
          << ", checkpoint = " << checkpoint << ")" << std::endl;
```

```
if (background_thread_state.valid())
  {
    TimerOutput::Scope timer(computing_timer, "main_loop␣-␣stalled␣output");
    background_thread_state.wait();
  }

constexpr auto problem_dimension =
  ProblemDescription<dim>::problem_dimension;

// Strore solution in primitive variables in output_vector
for(unsigned int i=0; i < offline_data.n_locally_owned; ++i){
    const auto U_i = gather(U, i);
    output_vector[0].local_element(i) = U_i[0];
    for(unsigned int k=1;k <= dim;k++)
      output_vector[k].local_element(i) = U_i[k]/U_i[0];
    output_vector[dim+1].local_element(i) = ProblemDescription<dim>::pressure(U_i);
}
for(unsigned int i = 0; i < problem_dimension; ++i)
  output_vector[i].update_ghost_values();


schlieren_postprocessor.compute_schlieren(output_vector);

auto data_out = std::make_shared<DataOut<dim>>();

data_out->attach_dof_handler(offline_data.dof_handler);

const auto &component_names = ProblemDescription<dim>::component_names;

for (unsigned int i = 0; i < problem_dimension; ++i)
  data_out->add_data_vector(output_vector[i], component_names[i]);

data_out->add_data_vector(schlieren_postprocessor.schlieren,
                          "schlieren_plot");

data_out->build_patches(discretization.mapping,
                        discretization.finite_element.degree - 1);

const auto output_worker = [this, name, t, cycle, checkpoint, data_out]() {
  if (checkpoint)
    {

      const unsigned int i =
        discretization.triangulation.locally_owned_subdomain();
      std::string filename =
        name + "-checkpoint-" + Utilities::int_to_string(i, 4) + ".archive";

      std::ofstream file(filename, std::ios::binary | std::ios::trunc);

      boost::archive::binary_oarchive oa(file);
      oa << t << cycle;
      for (const auto &it1 : output_vector)
        for (const auto &it2 : it1)
          oa << it2;
    }

  DataOutBase::VtkFlags flags(t,
                              cycle,
                              true,
                              DataOutBase::VtkFlags::best_speed);
  data_out->set_flags(flags);
```

```
      data_out->write_vtu_with_pvtu_record(
        "Output/", name + "-solution", cycle, mpi_communicator, 6);
    };

    if (asynchronous_writeback)
      {
        background_thread_state = std::async(std::launch::async, output_worker);
      }
    else
      {
        output_worker();
      }
  }

} // namespace Step69

int main(int argc, char *argv[])
{
  try
    {
      constexpr int dim = 2;

      using namespace dealii;
      using namespace Step69;

      Utilities::MPI::MPI_InitFinalize mpi_initialization(argc, argv);

      MPI_Comm       mpi_communicator(MPI_COMM_WORLD);
      MainLoop<dim> main_loop(mpi_communicator);

      main_loop.run();
    }
  catch (std::exception &exc)
    {
      std::cerr << std::endl
                << std::endl
                << "----------------------------------------------------"
                << std::endl;
      std::cerr << "Exception on processing: " << std::endl
                << exc.what() << std::endl
                << "Aborting!" << std::endl
                << "----------------------------------------------------"
                << std::endl;
      return 1;
    }
  catch (...)
    {
      std::cerr << std::endl
                << std::endl
                << "----------------------------------------------------"
                << std::endl;
      std::cerr << "Unknown exception!" << std::endl
                << "Aborting!" << std::endl
                << "----------------------------------------------------"
                << std::endl;
      return 1;
    };
}
```

A.3 Pseudocode for unsplit scheme

```
function UPDATE($Q$,CFL,$\bar{r}$)
    $d \leftarrow 0 \in \mathbb{R}^{|I|\times|I|}$
    for all $i \in I$ do
        for all $j \in I_i \setminus \{i\}$ do
            $\rho_l \leftarrow \rho(Q_i)$
            $\rho_r \leftarrow \rho(Q_j)$
            $u_l^\nu \leftarrow u(Q_i) \cdot n_{ij}$
            $u_r^\nu \leftarrow u(Q_j) \cdot n_{ij}$
            $p_l \leftarrow p(Q_i)$
            $p_r \leftarrow p(Q_j)$
            $a_l \leftarrow \sqrt{\gamma p_l/\rho_l}$
            $a_r \leftarrow \sqrt{\gamma p_r/\rho_r}$
            $p^* \leftarrow p_r((a_l + a_r - \frac{\gamma-1}{2} M_{\text{ref}}(u_r^\nu - u_l^\nu))/(a_l(p_l/p_r)^{-\frac{\gamma-1}{2\gamma}} + a_r))^{\frac{2\gamma}{\gamma-1}}$
            $\lambda_- \leftarrow -\min(M_{\text{ref}}u_l^\nu - a_l\sqrt{1 + \frac{\gamma+1}{2\gamma}\max(p^* - p_l, 0)/p_l}, 0)$
            $\lambda_+ \leftarrow \max(M_{\text{ref}}u_r^\nu - a_r\sqrt{1 + \frac{\gamma+1}{2\gamma}\max(p^* - p_r, 0)/p_r}, 0)$
            $\lambda_{\text{exp}} \leftarrow M_{\text{ref}}\max(u_l^\nu, u_r^\nu) + 5\max(a_l, a_r)$
            $\lambda_{\max} \leftarrow \min(\max(\lambda_-, \lambda_+), \lambda_{\text{exp}})/M_{\text{ref}}$
            $d_{ij} \leftarrow \lambda_{\max}||c_{ij}||$
            $d_{ii} \leftarrow d_{ii} - d_{ij}$
        end for
    end for
    $\tau \leftarrow \text{CFL} \cdot \min_{i\in I}(m_i/(-2d_{ii}))$
    $\hat{Q} \leftarrow Q$
    for all $i \in I$ do
        for all $j \in I_i$ do
            $\hat{Q}_i \leftarrow \hat{Q}_i + \frac{\tau}{m_i}(Q_j d_{ij} - f(Q_j)c_{ij})$
        end for
    end for
    return $(\hat{Q}, \tau)$
end function
```

Bibliography

[1] Saul Abarbanel, Pravir Duth, and David Gottlieb. Splitting methods for low mach number euler and navier-stokes equations. *Computers and Fluids*, 1989.

[2] Daniel Arndt, Wolfgang Bangerth, Maximilian Bergbauer, Marco Feder, Marc Fehling, Johannes Heinz, Timo Heister, Luca Heltai, Martin Kronbichler, Matthias Maier, Peter Munch, Jean-Paul Pelteret, Bruno Turcksin, David Wells, and Stefano Zampini. The `deal.II` library, version 9.5. *Journal of Numerical Mathematics*, 31(3):231–246, 2023.

[3] Walter Boscheri, Giacomo Dimarco, Raphaël Loubère, Maurizio Tavelli, and Marie-Hélène Vignal. A second order all mach number imex finite volume solver for the three dimensional euler equations. *Journal of Computational Physics*, 415, 2020.

[4] Alberto Bressan. *Hyperbolic Conservation Laws: An Illustrated Tutorial*, pages 157–245. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[5] Alexandre J. Chorin and Jerrold E. Marsden. *A Mathematical Introduction to Fluid Mechanics. Texts in Applied Mathematics.* Springer, 3 edition, 1993.

[6] Wolfgang Demtröder. *Experimentalphysik 1*, page 208. Springer, 9 edition, 2021.

[7] Lawrence C. Evans. *Partial Differential Equations. Graduate Studies in Mathematics*, volume 19. American Mathematical Society, 2 edition, 2010.

[8] Eduard Feireisl and Antonin Novotny. *Singular Limits in Thermodynamics of Viscous Fluids. Advances in Mathematical Fluid Mechanics*, pages 131–133. Birkhäuser, 2009.

[9] P. M. Gresho and S. T. Chan. On the theory of semi–implicit projection methods for viscous incompressible flow and its implementation via a finite element method that also introduces a nearly consistent mass matrix. part 2: Implementation. *International Journal for Numerical Methods in Fluids*, 11, 1990.

[10] Jean-Luc Guermond, Matthias Maier, Bojan Popov, and Ignacio Tomas. Second-order invariant domain preserving approximation of the compressible navier–stokes equations. *Computer Methods in Applied Mechanics and Engineering*, 375, 2021.

[11] Jean-Luc Guermond and Bojan Popov. Fast estimation of the maximum wave speed in the riemann problem for the euler equations. *Journal of Computational Physics*, 321, 2016.

[12] Jean-Luc Guermond and Bojan Popov. Invariant domains and first-order continous finite element approximation for hyperbolic systems. *SIAM J. Numer. Anal.*, 54(4), 2016.

[13] Randall J. Leveque. *Finite-Volume Methods for Hyperbolic Problems. Cambridge Texts in Applied Mathematics.* Cambridge University Press, 2004.

[14] Randall J. Leveque. *Finite-Volume Methods for Hyperbolic Problems. Cambridge Texts in Applied Mathematics*, pages 130–132. Cambridge University Press, 2004.

[15] Matthias Maier and Ignacio Tomas. step-69. `https://doi.org/10.5281/zenodo.3698223`, 2020.

[16] S. Noelle, G. Bispen, K. R. Arun, M. Lukáčová-Medviďová, and C.-D. Munz. A weakly asymptotic preserving low mach number scheme for the euler equations of gas dynamics. *SIAM Journal on Scientific Computing*, 36(6):989–1024, 2014.

[17] Roger Temam and Alain Miranville. *Mathematical modeling in continuum mechanics*, pages 42–48. Cambridge University Press, 2 edition, 2005.

[18] Andrea Thomann, Gabriella Puppo, and Christian Klingenberg. An all speed second order well-balanced imex relaxation scheme for the euler equations with gravity. *Journal of Computational Physics*, 420, 2020.

[19] H. A. Van der Vorst. Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13(2), 1992.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und die Arbeit keiner anderen Prüfungsbehörde unter Erlangung eines akademischen Grades vorgelegt habe.

Name:	Marius Volpert
Matrikelnummer:	2251948

Würzburg, 18.06.2024	[illegible]
Ort, Datum	Unterschrift