Write your first CFD solver

From theory to implemented CFD solver in less than a weekend

Tom-Robin Teschner

cfd.university

CFD.UNIVERSITY
LEARN THROUGH CODING

Write your first CFD solver

From theory to implemented CFD solver in less than a weekend

by

Tom-Robin Teschner

The cover page shows the launch of a space shuttle, a vehicle that experiences strong shock waves and shock-induced heating as part of the re-entry phase of a mission. Capturing these shock waves has resulted in decades of research on shock wave capturing schemes. This guide will teach you the importance of numerical schemes and how they can make or break a simulation. Image reproduced with permission from WikiImages

CFD.UNIVERSITY
LEARN THROUGH CODING

Contents

1

Getting started

Computational Fluid Dynamics (CFD) is constantly evolving, and new applications are found every day that benefit from using CFD. Initially, engineers were the sole beneficiaries, but CFD has outgrown engineering and is used by non-technical experts each day.

With the democratisation of CFD, more and more users are looking for resources to learn CFD. While there are some good books and online resources that explain the theory, very few attempt to show how to put the theory into code.

When I was a PhD student, I was constantly looking for a place to learn about how CFD can be implemented into code. There was none. This frustration stuck with me, especially during my days as a software developer working on a commercial CFD code.

Eventually, after roaming through books and online resources on CFD, software engineering, and programming, I obtained the knowledge I was after, but it took me years to get to this point. This is why I have started cfd.university, to share with you this knowledge and make that accessible to you in a far shorter time frame.

I believe that if you can set up a simulation and obtain results with CFD, you can solve today's problems. But, if you can *write* your own CFD solver, you can solve tomorrow's problems. We need more people who can write or customise existing CFD solvers to tackle unsolved challenges and make meaningful contributions.

This is where this guide comes in. If you want to take your CFD knowledge to the next level and start writing your own CFD solvers, this guide is for you. Once you start writing your own solvers, you will be well on your way towards CFD mastery.

Even if you have no interest in writing your own solvers in your professional career, writing at least a simple solver will help you build up an intuition for how numerical schemes and algorithms work. With that intuition, you can set up simulations with more confidence.

I really hope this guide and the resources available on cfd.university will help you to become a CFD expert. When you do, I hope you will use that skill set to make

meaningful contributions to the challenges we face in engineering and science.

1.1. Setting up a programming environment to develop CFD codes

Before we start writing code, let's make sure that your PC or laptop is set up in a way that allows you to develop code and visualise the results. I have already written an in-depth article on how to set up a coding environment on Windows, Linux, and macOS, so if you have no idea of how to get started, you can check out the article linked below:

Resource 1.1 Setting up a programming environment to develop CFD codes

All you need is a text editor to follow this guide. You will also need a C++ compiler to translate your written code into an executable. That's it. If you feel comfortable doing that, you can skip the above article.

Next, you want to make sure that you are comfortable with C++. While it is not the most user-friendly programming language to get started with, it is the most suitable language to write CFD solvers. No other language comes even close. Python, Fortran, Matlab, C, C#, and so on are, at best, a distant second choice. But C++ remains king, and you will make your life easier by picking up the basics of C++ programming now.

If you want to understand why C++ is the best choice for CFD development, I have a write-up for this, too. You can find it below:

Resource 1.2 Choosing the right programming language for CFD development

Resource 1.3 Why you should use C++ for CFD development

I will use C++ in this guide, but I have, on purpose, not used any fancy C++ syntax. I want this guide to be accessible. As a result, my C++ program looks more like a sequential C program rather than an object-orientated C++ code. For the simple code that we will develop, any of the more advanced C++ techniques are overkill.

If you have never coded with C++ before, I'd recommend Derek Banas' YouTube video *C++ Tutorial: C++ Full Course*. In one video, Derek goes through all the essential parts of C++, and it will give you an excellent overview. To get started, you only need the first 40 minutes, though. You can watch the rest once you feel settled with C++.

Resource 1.4 C++ Tutorial: C++ Full Course

Once you feel more comfortable with C++ (or perhaps you already know the basics), I would recommend you to have a look at my series on *What every CFD developer needs to know about C++*. It covers all the features of C++ that make our life easier when writing CFD solvers.

Typical textbook discussions of C++ features use very simplistic code examples, and

it can be challenging to see how these translate to CFD solvers. This series is bridging this gap by showing how we can use these advanced features of C++ in the context of a CFD problem and why they are so important.

Resource 1.5 What every CFD developer needs to know about C++

Now that we have a coding environment, let us look at the basic structure any CFD solver will follow.

2

The structure of a CFD solver

In this section, we will look at what is required to write a CFD solver in the first place. We start by reviewing what I call the *generalised CFD solver framework*. This is a pattern I have observed that any CFD solver will follow.

We will then look into the equations we will solve with our CFD solver and finally review the numerical schemes we will implement. Implementing different schemes and observing their behaviour is the simplest yet most effective thing you can do to develop an intuition for Monte CarloCFD (and, by extension, expertise). Let's jump straight in:

2.1. The generalised CFD solver framework

I have worked on countless CFD solvers myself, many of which I authored from start to end as the sole developer. In the process, I realised that writing a CFD solver always follows a predictable pattern.

I have worked on incompressible and compressible Navier-Stokes solvers, the lattice Boltzmann method, molecular dynamics, smooth particle hydrodynamics, and the direct simulation Monte Carlo method. I mixed laminar and turbulent flows, used the finite difference, finite volume, and finite element methods, and worked on small and large-scale codes (as in millions of lines of code). In each case, the same general CFD solver framework revealed itself.

Thus, in this section, I will show you this general CFD solver framework, which we will use to write our own first CFD solver.

The framework is shown in Figure 2.1 which can be separated into 3 main categories:

- **Pre-processing**: In this step, we prepare the simulation. All operations performed in this step are done once.
- **Solving**: This is where we obtain the solution. This requires an iterative solution procedure, and hence, this step is repeated until we reach a converged solution.
- **Post-processing**: After we have a solution, we process it. This step is also only ever performed once.

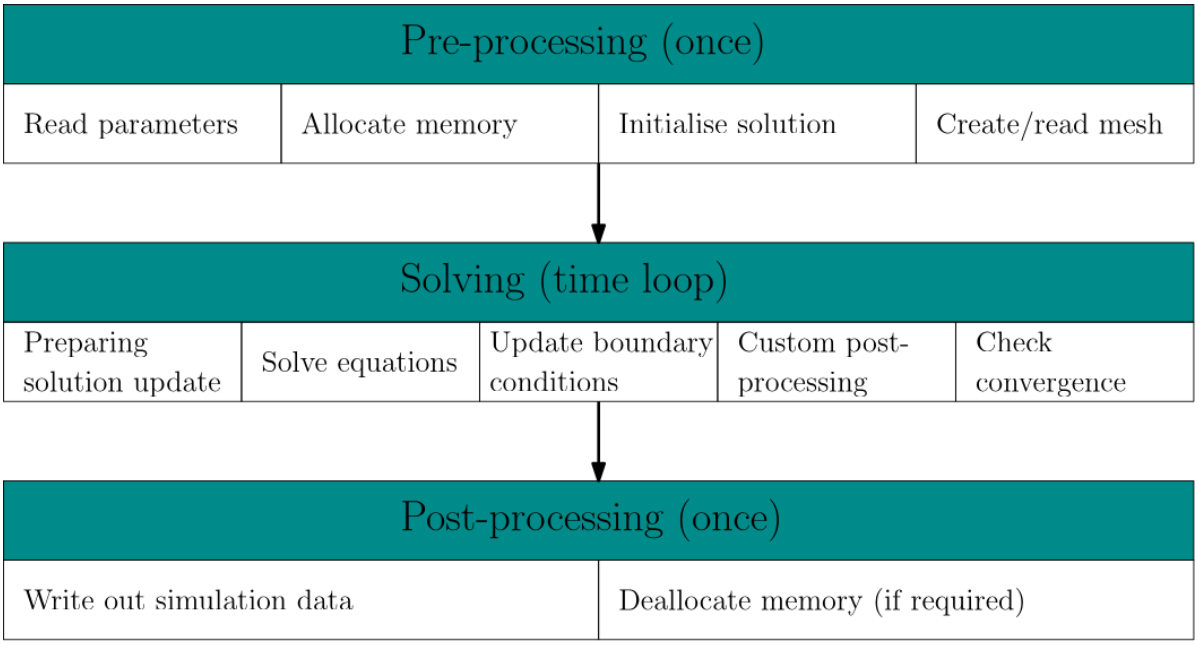


Figure 2.1: The generalised CFD solver framework. It is split into three sections, each with its subsections. A general CFD solver will go through all of these steps.

We go through each category one after the other (and in the order shown in Figure 2.1). Within each category, we have several steps that we have to perform.

The pre- and post-processing steps are done only once, while the solving step is repeated in a time loop (or, if it is a steady-state solution, an interaction loop). This means most of the computational cost is associated with the solving step.

Going through each category, these are the steps we have to perform:

Pre-processing (once)

1. **Read parameters**: Each simulation requires some solver settings. This is the first step in setting up a CFD solver. These can either come from a text input file or by setting the parameters in a graphical user interface.
2. **Allocate memory**: Next, we allocate the required memory for all arrays. This would typically include the solution arrays (pressure, velocity, temperature, etc.), but it can also include memory allocation for the mesh.
3. **Initialise solution**: With memory allocated, we initialise the solution for our solution vectors.
4. **Create/read mesh**: Finally, once the simulation is set up and memory has been allocated, we proceed with either creating a mesh or reading one. This is the last step in the pre-processing stage.

Solving (time loop)

1. **Preparing solution update**: Before we can proceed with the current timestep, we need to calculate a few things before we can continue. We may want to store

the previous solution in a separate array to calculate the residuals later, or we may want to calculate a stable timestep if we have a time-dependent flow. All of these required calculations come at the beginning of the timestep.

2. **Solve equations**: This is where we solve the governing equations, along with any additional equations (e.g. turbulence). We also compute the residual as part of the solution procedure, which we can use to judge convergence.
3. **Update boundary conditions**: Once the simulation has been advanced in time or iteration, we need to update the boundary conditions to reflect changes to the solutions.
4. **Custom post-processing**: We may want to perform custom post-processing during each timestep/iteration, like calculating the lift and drag coefficient for each timestep to obtain a time history. These calculations are performed at the end of each timestep/iteration.
5. **Check convergence**: We want to check if the simulation has ended at the end of each time step (or iteration). Typically, this means checking for convergence in residuals or integral quantities. If the simulation is unsteady, we may run the simulation until we have reached the targeted simulation time.

Post-processing (once)

1. **Write out simulation data**: Once the simulation has completed, we typically want to write out the solution. This can be as simple as a **.csv* file to process in Excel, Matlab, Python, etc., or more sophisticated by writing out the solution in a format that a post-processing software like ParaView or Tecplot can read.
2. **Deallocate memory**: Finally, in languages where we have to allocate and deallocate memory ourselves, we should clean up ourselves and free memory again.

This is it. It doesn't get more complicated than that. There may be cases, however, where we have to augment the solver framework. For example, when we are dealing with parallel computation, we also need to account for processor synchronisation, initialisation, and additional clean-up at the end. However, the framework shown in Figure 2.1 will still be present.

Now that we understand how to write our solver, it is time to review the governing equations of the CFD solver we will implement.

2.2. The governing equations

In this section, we will introduce the governing equation of our CFD solver. We will use the Euler equation to write our first CFD solver, which represents an inviscid approximation of the Navier-Stokes equations.

The Euler and Navier-Stokes equations share many similarities; mainly, they both retain the non-linear (convective) term, which is responsible for creating shock waves and turbulence.

If we understand how to write a CFD solver for the Euler equation, we'll see that going to Navier-Stokes isn't much more complicated. Furthermore, high-speed flows are

mostly inviscid anyway, and only small regions are dominated by viscous forces.

Only if we want to study these small regions do we need to consider viscous forces, and most of the CFD work in the early 1980s and onwards was performed using the Euler equation (especially in the aerospace industry).

2.2.1. From Navier-Stokes to the Euler equation

We will start with the compressible form of the momentum equation of the Navier-Stokes equations. This is given as:

$$\frac{\partial \rho \mathbf{u}}{\partial t} + \nabla \cdot (\rho \mathbf{u} \otimes \mathbf{u}) = -\nabla p + \nabla \cdot \left(\mu \left[\nabla \mathbf{u} + (\mathbf{u})^T - \frac{2}{3} (\nabla \cdot \mathbf{u}) \mathbf{I} \right] \right) + \nabla \left[\xi (\nabla \cdot \mathbf{u}) \right] \quad (2.1)$$

This equation has several terms, where each term is responsible for different physical phenomena:

- The term $\partial \rho \mathbf{u} / \partial t$ is responsible for the time-dependence of the Navier-Stokes equations. It allows for physical phenomena to develop in time, i.e. it is a dynamic equation.
- The term $\nabla \cdot (\rho \mathbf{u} \otimes \mathbf{u})$ is the convective, or non-linear term, and is responsible for, let's say, all the excitement in the Navier-Stokes equations. It produces shock waves, generates and sustains turbulence, and keeps people like me in business.
- The term ∇p is the pressure gradient, and it is responsible for inducing flow where there is a pressure difference between different regions. Think of an open valve on a bicycle tyre; if the valve is opened, a flow will be induced from the high-pressure (bicycle tyre) region to the low-pressure (atmospheric) region.
- The term $\nabla \cdot \left(\mu \left[\nabla \mathbf{u} + (\mathbf{u})^T - (2/3)(\nabla \cdot \mathbf{u}) \mathbf{I} \right] \right)$ is responsible for diffusive and mixing processes. For example, if you place a tea bag in hot but non-moving water, you will see how diffusion will mix the tea in your tea leaves into the water.
- The term $\nabla \left[\xi (\nabla \cdot \mathbf{u}) \right]$ contains the second-viscosity parameter ξ and thus enhances the mixing process. For monoatomic gases, we have $\xi = 0$. For non-monoatomic gases or liquids, we have $\xi \neq 0$.

Thus, if we want to create an inviscid approximation to the above equation, we need to remove all terms containing μ and ξ. This simplifies our equation considerably, and we arrive at the following equation, which we call the Euler equation:

$$\frac{\partial \rho \mathbf{u}}{\partial t} + \nabla \cdot (\rho \mathbf{u} \otimes \mathbf{u}) = -\nabla p \quad (2.2)$$

We can also bring the pressure onto the left-hand side of the equation, which provides us with the following form:

$$\frac{\partial \rho \mathbf{u}}{\partial t} + \nabla \cdot (\rho \mathbf{u} \otimes \mathbf{u} + p \mathbf{I}) = 0 \quad (2.3)$$

The non-linear term containing the so-called dyad product ($\otimes$) can be written as follows, assuming that the velocity vector is given as $\mathbf{u} = [u, v, w]^T$:

$$\rho\mathbf{u} \otimes \mathbf{u} = \rho \begin{pmatrix} u \\ v \\ w \end{pmatrix} \begin{pmatrix} u & v & w \end{pmatrix} = \begin{bmatrix} \rho u^2 & \rho uv & \rho uw \\ \rho uv & \rho v^2 & \rho vw \\ \rho uw & \rho vw & \rho w^2 \end{bmatrix} \tag{2.4}$$

Since Eq.(2.4) is a 3x3 tensor, we need to multiply the pressure in Eq.(2.3) by the identity matrix ($\mathbf{I}$), as we can't add a tensor and a scalar.

Eq.(2.3) is a vector equation for x, y and z. Thus, we have a total of 3 equations but 5 unknowns (ρ, p, u, v, w). Thus, we need to construct an additional 2 equations so that we can solve for all unknown quantities.

The first equation we consider is the conservation of mass. This can be written as:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho\mathbf{u}) = 0 \tag{2.5}$$

We obtain the density ρ from Eq.(2.5), and thus only need one additional equation for the pressure. However, there is no conservation law that we can use for the pressure. Thus, we need to use some form of thermodynamic relation to obtain the pressure.

We start by solving the conservation law for the energy. Different versions are available, depending on which type of energy we are solving (e.g. total, internal, potential. kinetic, etc.). We will use the total energy here, which can be decomposed as:

$$E = \text{internal energy} + \text{kinetic energy} = \rho e + \frac{1}{2}\rho\mathbf{u}^2 \tag{2.6}$$

The conservation law for the total energy is defined as:

$$\frac{\partial E}{\partial t} + \nabla \cdot \mathbf{u}(E + p) = 0 \tag{2.7}$$

This equation is valid for an inviscid flow only, which is in line with the inviscid approximation we made in the momentum equation, i.e. Eq.(2.3) where we have dropped any viscous contributions.

While we now have a total of 5 equations, we are still not able to close the system for all 6 unknowns. To do that, we need to borrow some thermodynamic relations, which we will review in the next section.

2.2.2. Thermodynamic relations for the Euler equation

As alluded to above, to solve Eq.(2.3), Eq.(2.5), and Eq.(2.7), we need some thermodynamic relation. Since we obtain u, v, and w from Eq.(2.3), ρ from Eq.(2.5), and E from Eq.(2.7), we need to find a relation for the pressure p.

This can be done using an equation of state, where the pressure is related to other known thermodynamic quantities. For example, the equation of state for an ideal gas (ideal gas law) relates the pressure p to the density ρ, temperature T, and specific gas constant R as:

$$p = \rho R T \tag{2.8}$$

Eq.(2.8) is derived based on the assumption that intermolecular forces can be neglected. This works as long as the pressure is low. At higher pressure, a fluid compresses, and molecules are placed closer and closer together. Once they are a few atomistic diameters apart, intermolecular forces dominate, making the ideal gas law inaccurate.

Similarly, at low temperatures, fluid particles slow down, and a gas may condense into a liquid, changing the state of the fluid. At this point, the ideal gas law becomes invalid. Related to that are high-speed flows, i.e. those where the Mach number is greater than 5. Here, chemical reactions become important, making the ideal gas law again inaccurate.

Despite these limitations, the ideal gas law applies to a wide range of applications and is still widely used these days, despite more sophisticated equations of states being available, such as the Van der Waals, Redlich-Kwong, or Peng-Robinson equation of state. We will use the ideal gas law to relate the pressure p to the total energy E in our CFD solver.

We need a few additional thermodynamic relations before we can relate p to E. The first is for internal energy, which can be calculated as:

$$e = c_v T \tag{2.9}$$

Here, c_v is the specific heat capacity at constant volume and T the temperature. We can also define the specific heat capacity at constant pressure, which is called c_p. It is related to c_v for an ideal gas by:

$$c_p - c_v = R \tag{2.10}$$

Here, R again, is the specific gas constant. For an ideal gas, we can define the ratio of specific heats as:

$$\gamma = \frac{c_p}{c_v} \tag{2.11}$$

Dividing Eq.(2.10) by c_v, we obtain:

$$\gamma - 1 = \frac{R}{c_v} \tag{2.12}$$

Or, multiplying Eq.(2.12) by c_v, we obtain:

$$c_v(\gamma - 1) = R \tag{2.13}$$

We now have all the ingredients to relate the pressure p to the total energy E. First, we solve Eq.(2.8) for T and put it into Eq.(2.9). This results in:

$$e = c_v T = c_v \frac{p}{\rho R} \tag{2.14}$$

Next, we insert Eq.(2.13) into Eq.(2.14). This results in:

$$e = c_v \frac{p}{\rho c_v (\gamma - 1)} \tag{2.15}$$

Both values for c_v cancel out in Eq.(2.15), so we can rewrite this equation as:

$$\rho e = \frac{p}{\gamma - 1} \tag{2.16}$$

We can insert Eq.(2.16) into Eq.(2.6) and obtain:

$$E = \rho e + \frac{1}{2}\rho \mathbf{u}^2 = \frac{p}{(\gamma - 1)} + \frac{1}{2}\rho \mathbf{u}^2 \tag{2.17}$$

This equation can now be solved for the pressure, which results in:

$$p = (\gamma - 1)\left[E - \frac{1}{2}\rho \mathbf{u}^2\right] \tag{2.18}$$

2.2.3. Scalar form of the Euler equation

Thus far, we have looked at the conservation of momentum (Eq.(2.3)), conservation of mass (Eq.(2.5)), and conservation of total energy (Eq.(2.7)) in a compact divergence form. However, we can't implement these equations.

To do that, we always need to derive the scalar form first, which shows us how to discretise our equations. This is what we will look at in this section.

Let's start with Eq.(2.3), i.e. the conservation of momentum. We already reviewed how to write out the dyad product in Eq.(2.4), which resulted in a tensor. Furthermore, we need to know how to write the nabla operator. This is defined differently for different coordinate systems, but typically, we use Cartesian coordinates for which we have:

$$\nabla = \begin{pmatrix} \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \end{pmatrix}^T \tag{2.19}$$

We can now use Eq.(2.4) and Eq.(2.19) to rewrite Eq.(2.3) as:

$$\frac{\partial}{\partial t}\rho\begin{pmatrix}u\\v\\w\end{pmatrix}+\begin{pmatrix}\frac{\partial}{\partial x}\\\frac{\partial}{\partial y}\\\frac{\partial}{\partial z}\end{pmatrix}\cdot\left(\rho\begin{pmatrix}u\\v\\w\end{pmatrix}\begin{pmatrix}u & v & w\end{pmatrix}+p\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}\right)=0 \tag{2.20}$$

Carrying out the multiplication and addition for the last term in brackets results in:

$$\frac{\partial}{\partial t}\rho\begin{pmatrix}u\\v\\w\end{pmatrix}+\begin{pmatrix}\frac{\partial}{\partial x}\\\frac{\partial}{\partial y}\\\frac{\partial}{\partial z}\end{pmatrix}\cdot\left(\begin{bmatrix}\rho u^2+p & \rho uv & \rho uw\\ \rho uv & \rho v^2+p & \rho vw\\ \rho uw & \rho vw & \rho w^2+p\end{bmatrix}\right)=0 \tag{2.21}$$

From this, we can now derive the scalar form of the Euler equation in three-dimensional space as:

$$\frac{\partial \rho u}{\partial t}+\frac{\partial \rho u^2+p}{\partial x}+\frac{\partial \rho uv}{\partial y}+\frac{\partial \rho uw}{\partial z}=0 \tag{2.22}$$

$$\frac{\partial \rho v}{\partial t}+\frac{\partial \rho uv}{\partial x}+\frac{\partial \rho v^2+p}{\partial y}+\frac{\partial \rho vw}{\partial z}=0 \tag{2.23}$$

$$\frac{\partial \rho w}{\partial t}+\frac{\partial \rho uw}{\partial x}+\frac{\partial \rho vw}{\partial y}+\frac{\partial \rho w^2+p}{\partial z}=0 \tag{2.24}$$

For the conservation of mass, i.e. Eq.(2.5), we get:

$$\frac{\partial \rho}{\partial t}+\begin{pmatrix}\frac{\partial}{\partial x}\\\frac{\partial}{\partial y}\\\frac{\partial}{\partial z}\end{pmatrix}\cdot\begin{pmatrix}\rho u\\\rho v\\\rho w\end{pmatrix}=0 \tag{2.25}$$

Carrying out the scalar (dot) product, we obtain the final form of the conservation of mass equation as:

$$\frac{\partial \rho}{\partial t}+\frac{\partial \rho u}{\partial x}+\frac{\partial \rho v}{\partial y}+\frac{\partial \rho w}{\partial z}=0 \tag{2.26}$$

We proceed similarly for the conservation of total energy, i.e. Eq.(2.7). This can be written as:

$$\frac{\partial E}{\partial t}+\begin{pmatrix}\frac{\partial}{\partial x}\\\frac{\partial}{\partial y}\\\frac{\partial}{\partial z}\end{pmatrix}\cdot\begin{pmatrix}u\\v\\w\end{pmatrix}(E+p)=0 \tag{2.27}$$

Carrying out the scalar (dot) product, just as we did before, we obtain the final scalar form of the conservation of total energy equation as:

$$\frac{\partial E}{\partial t}+\frac{\partial u(E+p)}{\partial x}+\frac{\partial v(E+p)}{\partial y}+\frac{\partial w(E+p)}{\partial z}=0 \tag{2.28}$$

At this point, we can make one simplification. Since we are only interested in a one-dimensional flow, we can ignore any terms that contain a derivative in the y and z directions. Equally, any term containing the v and w velocity component can be ignored. Thus, we end up with the following final system of equations to solve the inviscid Navier-Stokes (Euler) equations:

- Conservation of mass:

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u}{\partial x} = 0 \tag{2.29}$$

- Conservation of momentum:

$$\frac{\partial \rho u}{\partial t} + \frac{\partial \rho u^2 + p}{\partial x} = 0 \tag{2.30}$$

- Conservation of total energy:

$$\frac{\partial E}{\partial t} + \frac{\partial u(E+p)}{\partial x} = 0 \tag{2.31}$$

- Equation of state for an ideal gas:

$$p = (\gamma - 1)\left[E - \frac{1}{2}\rho \mathbf{u}^2\right] \tag{2.32}$$

This system is typically written in a vector form, which lends itself to easy implementation into code. It is given as:

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}(\mathbf{U})}{\partial x} = 0 \tag{2.33}$$

Here, we have defined the vector of conserved quantities $\mathbf{U}$ as:

$$\mathbf{U} = \begin{pmatrix} \rho \\ \rho u \\ E \end{pmatrix} \tag{2.34}$$

The fluxes $\mathbf{F}(\mathbf{U})$, on the other hand, are written as:

$$\mathbf{F}(\mathbf{U}) = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ u(E+p) \end{pmatrix} \tag{2.35}$$

2.2.4. Discretising the Euler equation using the finite volume method

Now that we have found a scalar form of the system of equations we want to solve, i.e. Eqs.(2.29)–(2.31), we have to decide how we want to transform the system into algebraic equations. This is done by the discretisation process.

There are three main discretisation approaches that can be used in CFD. These are:

- The finite difference method (FDM)
- The finite volume method (FVM)
- The finite element method (FEM)

The finite difference method (FDM) was historically the first method we used for CFD applications. It takes the derivatives in our partial differential equations and replaces these with approximations of the derivatives. We use Taylor-Series expansions here. It is the most straightforward method to implement, but it is limited to structured grids, and it does not cope well with shock waves (discontinuities), for which derivatives are infinite!

The finite volume method (FVM) reformulates the differential equations into an integral form. This avoids the issues of not being able to resolve discontinuities faithfully. Furthermore, we can also use unstructured grids. Therefore, the finite volume method is the de facto standard discretisation method used in CFD. Most, if not all, CFD solvers you will ever come across (commercial and open-source) are based on this method.

Finally, we have the finite element method (FEM). It is mainly used in structural mechanics, but some CFD solvers are based on it as well. It uses test functions within a cell to reconstruct the solution locally. This allows us to reconstruct the solution with an arbitrary accuracy (it is a local user-defined property), which makes it very lucrative. It has gained popularity in CFD through the discontinuous Galerkin method (DG), but it remains primarily a research tool. I can only think of two commercial CFD solvers that use either FEM or DG as a discretisation procedure.

We will use the finite volume method here, as this is in line with most of the literature that you will come across. It strikes a nice balance between mathematical complexity (not being too complex like FEM or DG) and computational flexibility (not being too restrictive in terms of accuracy and the grids we can use, as in FDM).

The finite volume method first of all starts by dividing a computational domain into so-called finite volumes. Finite volumes are nothing else than the computational cells a mesh generator would provide us. We assume that each variable we are solving for (e.g. velocity, pressure, temperature, density, etc.) stays constant within each finite volume (cell).

We then go about calculating the fluxes that go through the faces. For example, for the continuity equation (conservation of mass, i.e. Eq.(2.29)), we can see that the flux is defined as ρu (see flux definition in Eq.(2.35)). Similarly, we can define fluxes in the momentum and energy equation as $\rho u^2 + p$ and $u(E + p)$, respectively.

We compute the flux that goes through faces between finite volumes (cells). However, the variables are typically stored at the centre (centroid) of the finite volume (cell).

Thus, we need to interpolate the values from the cells' centroids to the cells' faces. This process requires a numerical scheme, and we will review common options in section 2.3, which are then implemented in section 3.3.2.

So let's do that with Eq.(2.33). First, we need to integrate the terms over a finite volume V. This results in:

$$\int_V \frac{\partial \mathbf{U}}{\partial t} \mathrm{d}V + \int_V \frac{\partial \mathbf{F}(\mathbf{U})}{\partial x} \mathrm{d}V = 0 \tag{2.36}$$

We then apply the Gauss (or divergence) theorem to the second term in Eq.(2.36). This transforms the volume integral into a surface integral, removing the derivative. It may not seem clear why we do that, but suffice it to say at this point that this will conserve mass, momentum, and energy, while a volume integral won't.

The resulting equation becomes:

$$\int_V \frac{\partial \mathbf{U}}{\partial t} \mathrm{d}V + \int_A \vec{n} \cdot \mathbf{F}(\mathbf{U}) \mathrm{d}A = 0 \tag{2.37}$$

We can now carry out the integration. While the quantities defined in $\mathbf{U}$ most certainly change in space, we make one simplification here. We say that within each cell, all quantities are constant. And, if they are constant, we can take it out of the volume integration in Eq.(2.37). This results in:

$$\frac{\partial \mathbf{U}}{\partial t} \int_V \mathrm{d}V + \int_A \vec{n} \cdot \mathbf{F}(\mathbf{U}) \mathrm{d}A = 0 \tag{2.38}$$

The first term simplifies, but the second term remains the same, as the fluxes at the faces will be different. This is something we will see shortly. This simplification is one assumption we make in the finite volume method. If the values are constant, we only need a single integration point per cell.

If we wanted to allow more than one integration point, and thus a solution which changes within a cell, then we need to use the finite element method, which allows for an arbitrary number of points per cell and face.

Returning back to Eq.(2.38), this equation can now be integrated into the following form:

$$\frac{\partial \mathbf{U}}{\partial t} V + \sum_{face=0}^{nFaces} \vec{n} \cdot \mathbf{F}(\mathbf{U})_{face} A_{face} = 0 \tag{2.39}$$

The first term is simply the volume integral, which produces the volume of the cell. The second term, however, was a surface integral. Thus, we need to integrate over each face of the cell separately.

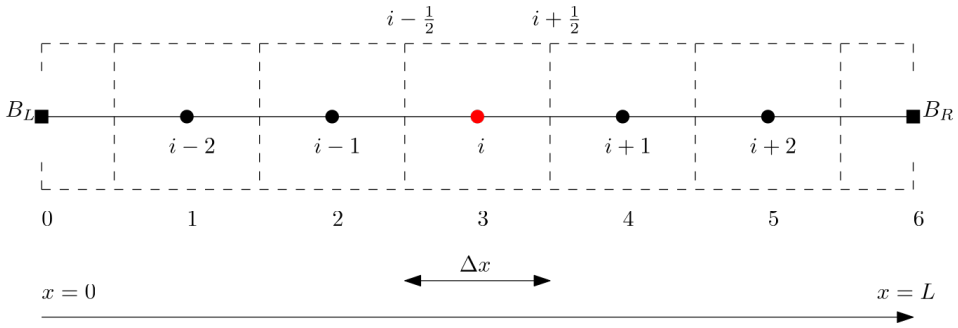


Figure 2.2: 1D finite volume discretisation for a domain spanning from $x = 0$ to $x = L$ using 6 grid points, two of which are located on the left (B_L) and right (B_R) boundaries.

To see how this term comes about, consider Figure 2.2. Here, we are looking at a simple 1D mesh with 6 points. The left and right points are the left and right boundaries, denoted by B_L and B_R, respectively. The domain starts at $x = 0$ and goes to $x = L$.

We can pick any of the points we want, for example, point 3 (shown in red), and find its neighbouring points using a local index i. For example, its neighbour to the left can be found with the index $i - 1$ (point 2), and its neighbour to the right can be found with $i + 1$ (point 4).

Even though this is a 1D domain, the finite volume method assumes, as the name suggests, a 3D domain. However, we can always approximate a 3D domain with a 1D mesh by assuming that we have some finite extent in the top and bottom, as well as front and back direction (in other words, we assume the flow only changes in one direction, so we don't consider the other two directions).

Shown in Figure 2.2 , the finite volumes (cells) are in dashed lines. Granted, these are only 2D cells, but imagine these cells stretch to the front and back, then we would have a 3D mesh.

OK, so let's return to Eq.(2.39). We said that we need to integrate over all surfaces of the cell (finite volume). If we pick cell 3 in Figure 2.2, we can see its surfaces located at $i + 1/2$ and $i - 1/2$, i.e. at the dashed lines where two cells meet.

In Eq.(2.39), we have $nFaces = 2$, i.e. at $i+1/2$ and $i-1/2$. We also have the normal vector $\vec{n}$, which always points out of the cell. Thus, if x is going from the left to the right, we have $\vec{n} = 1$ at $i + 1/2$ (pointing along the x direction) and $\vec{n} = -1$ at $i - 1/2$ (pointing against the x direction).

Furthermore, the terms $\mathbf{F}(\mathbf{U})_{face}$ and A_{face} will be evaluated at $i+1/2$ and $i-1/2$. With this information at hand, we can carry out the summation and arrive at the following form:

$$\frac{\partial \mathbf{U}}{\partial t} V + \left(\mathbf{F}(\mathbf{U})_{i+\frac{1}{2}} A_{i+\frac{1}{2}} - \mathbf{F}(\mathbf{U})_{i-\frac{1}{2}} A_{i-\frac{1}{2}} \right) = 0 \tag{2.40}$$

Since we are considering a 1D domain, we can see from Figure 2.2 that the surface area in the x-direction does not change from cell to cell. We can, therefore, write that

$A_{i+\frac{1}{2}} = A_{i-\frac{1}{2}} = A$, i.e. the surface area is constant. We can take the constant out of the integration and arrive at:

$$\frac{\partial \mathbf{U}}{\partial t} V + \left(\mathbf{F}(\mathbf{U})_{i+\frac{1}{2}} - \mathbf{F}(\mathbf{U})_{i-\frac{1}{2}} \right) A = 0 \tag{2.41}$$

Now, let's consider the volume and area for a moment. We can see from Figure 2.2 that we have a spacing in the x-direction labelled as Δx. We also said that we have a constant but arbitrary value in the top and bottom direction (e.g. y-direction), and in the front and back direction (e.g. z-direction).

Thus, if we say that the spacing in the y-direction is set arbitrarily at $\Delta y = 1$, and similar in the z-direction to $\Delta z = 1$, then we can say that the surface area is $A = \Delta y \Delta z$ and the volume is $V = \Delta x \Delta y \Delta z$. Of course, if we had a 3D solver, we could calculate the real values for Δy and Δz from the grid (we only set it here because of the 1D nature of the solver).

Inserting the definitions for A and V into Eq.2.41 results in:

$$\frac{\partial \mathbf{U}}{\partial t} \Delta x \Delta y \Delta z + \left(\mathbf{F}(\mathbf{U})_{i+\frac{1}{2}} - \mathbf{F}(\mathbf{U})_{i-\frac{1}{2}} \right) \Delta y \Delta z = 0 \tag{2.42}$$

Now, we divide by $\Delta x \Delta y \Delta z$, i.e. the volume, to arrive at:

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{1}{\Delta x} \left(\mathbf{F}(\mathbf{U})_{i+\frac{1}{2}} - \mathbf{F}(\mathbf{U})_{i-\frac{1}{2}} \right) = 0 \tag{2.43}$$

Eq.(2.43) represents the final, finite volume discretised form of our system of equations shown in Eq.(2.33). This is the one we implement into our CFD solver. Let's see what this equation will look like.

First, we substitute the vector quantities $\mathbf{U}$ and $\mathbf{F}(\mathbf{U})$ into Eq.(2.43) to arrive at:

$$\frac{\partial}{\partial t} \begin{pmatrix} \rho \\ \rho u \\ E \end{pmatrix} + \frac{1}{\Delta x} \left(\begin{pmatrix} \rho u \\ \rho u^2 + p \\ u(E+p) \end{pmatrix}_{i+\frac{1}{2}} - \begin{pmatrix} \rho u \\ \rho u^2 + p \\ u(E+p) \end{pmatrix}_{i-\frac{1}{2}} \right) = 0 \tag{2.44}$$

From Eq.(2.44), we can then re-write our system of equations as:

- Conservation of mass:

$$\frac{\partial \rho}{\partial t} + \frac{1}{\Delta x} \left[(\rho u)_{i+\frac{1}{2}} - (\rho u)_{i-\frac{1}{2}} \right] = 0 \tag{2.45}$$

- Conservation of momentum:

$$\frac{\partial \rho u}{\partial t} + \frac{1}{\Delta x} \left[(\rho u^2 + p)_{i+\frac{1}{2}} - (\rho u^2 + p)_{i-\frac{1}{2}} \right] = 0 \tag{2.46}$$

- Conservation of total energy:

$$\frac{\partial E}{\partial t} + \frac{1}{\Delta x}\left[(u(E+p))_{i+\frac{1}{2}} - (u(E+p))_{i-\frac{1}{2}}\right] = 0 \tag{2.47}$$

We need to find a suitable approximation for our time derivative $\partial/\partial t$, as well as a way to obtain fluxes at the faces, i.e. at $i+1/2$ and $i-1/2$. We will find approximations for them in section 2.3.

2.2.5. Obtaining primitive variables

Let's review the vector of conserved variables $\mathbf{U}$ and fluxes $\mathbf{F}(\mathbf{U})$. We defined these in Eq.(2.34) and Eq.(2.35), respectively. These are given below again for convenience:

$$\mathbf{U} = \begin{pmatrix} \rho \\ \rho u \\ E \end{pmatrix} \qquad \mathbf{F}(\mathbf{U}) = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ u(E+p) \end{pmatrix}$$

We have the values of $\mathbf{U}$ available at each cell i, and we have to use this vector to compute the fluxes at faces $i+1/2$ and $i-1/2$. If we inspect the flux vector above, we note that we need the velocity u, the density ρ, the pressure p, and the total energy E.

We need to establish a mechanism to obtain the primitive variables (which are u, ρ, p, and E) from the conserved variables (which are ρ, ρu, and E).

Obtaining the density is trivial, as this is the first component of the conserved quantity vector $\mathbf{U}$. We can write this explicitly as:

$$\rho = \mathbf{U}(1) = \rho \tag{2.48}$$

Obtaining the velocity u is also rather straightforward by dividing the second component with the first. I.e. we have:

$$u = \frac{\mathbf{U}(2)}{\mathbf{U}(1)} = \frac{\rho u}{\rho} = u \tag{2.49}$$

The total energy can also be obtained directly from the conserved quantities. This is done using the following relationship:

$$E = \mathbf{U}(3) = E \tag{2.50}$$

The pressure p is more complicated, but we already reviewed how to get this from thermodynamic relations in Section 2.2.2, specifically, Eq.(2.18). For convenience, the equation is repeated below again:

$$p = (\gamma - 1)\left[\mathbf{U}(3) - \frac{1}{2}\rho\mathbf{u}^2\right] = (\gamma - 1)\left[E - \frac{1}{2}\rho\mathbf{u}^2\right] \tag{2.51}$$

With these values obtained, we can now compute the fluxes at $i + 1/2$ and $i - 1/2$ defined by Eq.(2.35) and use them in Eq.(2.44).

2.3. Numerical schemes

Much of the simulation will be spent on approximating quantities to solve our governing equation. For compressible flows in particular, numerical schemes play a crucial role and can make the difference between convergence and divergence of the simulation.

In this section, we will first review how we can approximate solutions at the faces of our cells (this is either called the interpolation or reconstruction step). We use these reconstructed variables to compute fluxes across the faces, i.e. the fluxes at $i + 1/2$ and $i - 1/2$ as seen in Eq.(2.44).

We will see that we end up with two separate approximations for the fluxes on our faces, and we use an approximate Riemann solver to consolidate these two fluxes into a single value.

The fluxes obtained from the Riemann problem are then used in our governing equation to update the solution in time. At the end of this section, we will review a common technique to integrate our simulation in time.

2.3.1. Numerical approximation in space

In this section, we look at spatial numerical schemes that we can use to find approximations to $\mathbf{F}(\mathbf{U})_{i\pm\frac{1}{2}}$ in Eq.(2.44). We will look at two common choices and see how they compare against one another later when we run the code.

2.3.1.1. Piecewise constant reconstruction (1st-order)

The piecewise constant reconstruction is the simplest numerical scheme we can use. It is dead simple to implement and usually a good idea to ensure things are working as expected. Once we have the first working version of our solver available, we may want to implement a more accurate scheme as for example shown in the next section.

The piecewise constant scheme is simply copying values that are stored at the cells' centroid to the faces' centroid. In other words, we assume that the quantities within a cell do not change. This may seem like a reasonable approximation at first, especially if our cells are made smaller and smaller. However, in the presence of strong non-linear effects, such as shock waves, this leads to an overly diffusive approximation and excessive numerical dissipation.

We won't go into much detail about numerical dissipation here, but if you want to learn more about it, I have written about it previously, and you may want to check out the article linked below:

Resource 2.1 What is numerical dissipation in CFD and why do we need it?

We can formally define the piecewise constant scheme as:

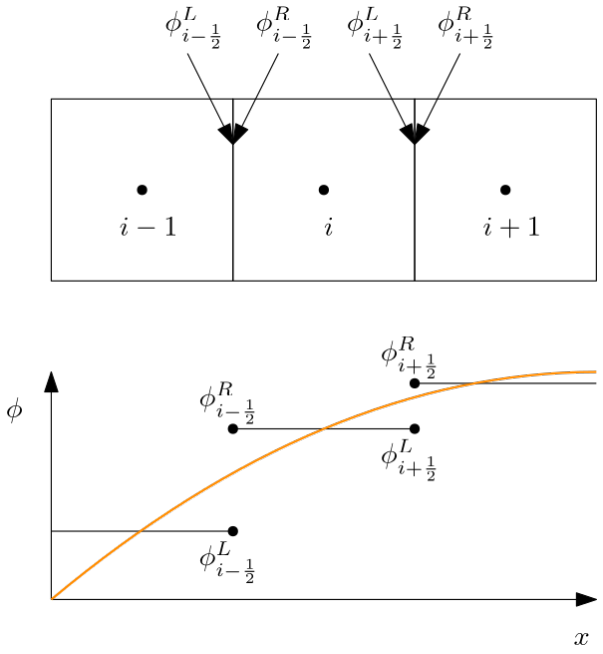


Figure 2.3: Piecewise constant reconstruction for the function shown in orange.

$$\begin{aligned} \phi^L_{i+\frac{1}{2}} &= \phi_i \\ \phi^R_{i-\frac{1}{2}} &= \phi_i \end{aligned} \tag{2.52}$$

Let's see how this scheme works graphically. For this, let us look at Figure 2.3, where we have three cells on which we want to approximate the values of ϕ, which are given by the orange line.

The finite volume method assumes that we average the function values within each cell and only store this average at the cell centroid. This is indicated by the black solid lines.

The piecewise constant reconstruction now says that we take this constant value within each cell and copy that to the left and right face.

Let's look at the face at $i + 1/2$ closely. If we approximate $\phi_{i+1/2}$ from cell i, then we get some approximation that we denote by $\phi^L_{i+1/2}$, i.e. we say it is a left-sided reconstruction (left as in the left-hand side of the face). We can also obtain a right-sided reconstruction by starting in cell $i + 1$. Now, we obtain $\phi^R_{i+1/2}$, and we can see from the figure that $\phi^L_{i+1/2} \neq \phi^R_{i+1/2}$!

Is this a problem for us? Not quite. Godunov realised that this poses essentially a Riemann problem, and he postulated that we should use a Riemann solver to consolidate both the left-sided and right-sided reconstruction into a single value $\tilde{\phi}_{i+1/2}$.

This is a very influential concept in the field of CFD and if we employ a Riemann solver in our solution, then we are using the so-called Godunov method. Because of

its importance, We will review the Riemann problem in Section 2.3.3 in more detail and look at one such Riemann solver in Section 2.3.3.1.

2.3.1.2. MUSCL scheme (2nd-order)

The piecewise constant approximation is great to get started, but we will also see later that it is really diffusive and thus not very accurate for capturing strong non-linear phenomena such as shock waves. As a result, researchers developed numerical schemes that were able to capture these non-linear behaviour with greater accuracy.

In the context of compressible flows with strong shock waves, two main schemes are used nowadays.

- The Monotonic Upstream-centered Scheme for Conservation Laws (MUSCL) by Van Leer
- The Weighted Essentially Non-oscillatory (WENO) scheme by Liu, Osher, and Chan.

Both of these schemes offer the reconstruction of high-resolution schemes, that is, they provide a higher-order accuracy while ensuring that strong non-linear effects such as shock waves do not introduce unwanted side effects (oscillations in the solution).

WENO schemes achieve this by constructing different approximations for $\phi^{L,R}_{i+1/2}$, which are then combined using different weights. The weights are calculated based on the smoothness of the solution, i.e. if strong shock waves are present, then the weights will be adjusted so that any non-physical effects are suppressed.

MUSCL schemes, on the other hand, reconstruct values for $\phi^{L,R}_{i+1/2}$ using only a single polynomial. Instead of weighing different polynomials, van Leer introduced a limiter to avoid non-physical oscillations. We will review the role of the limiter in Section 2.3.2 in more detail.

Both schemes can be formulated for higher orders of numerical accuracy and achieve similar results. The MUSCL scheme, though, is computationally easier to implement and understand and thus used in this section to give you a gentle introduction to high-resolution schemes.

However, if you are interested in implementing the WENO scheme as well, I provide some guidelines on how to achieve that in Section 5.3. A great review of WENO schemes can be found in the article linked below:

Resource 2.2 Scholarpedia - WENO methods

In the second-order version of the MUSCL scheme, the values for $\phi^{L,R}_{i+1/2}$ are reconstructed using the following equations:

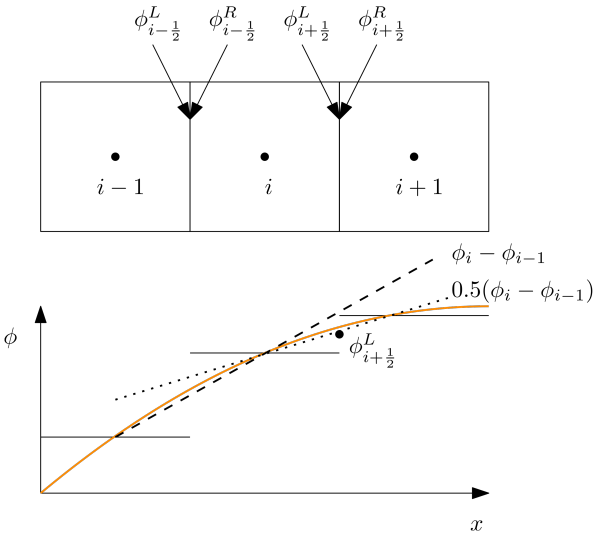


Figure 2.4: MUSCL reconstruction for the function shown in orange.

$$\phi^L_{i+1/2} = \phi_i + \frac{\Psi(r_{i+\frac{1}{2}})}{2}(\phi_i - \phi_{i-1})$$
$$\phi^R_{i-1/2} = \phi_i - \frac{\Psi(r_{i-\frac{1}{2}})}{2}(\phi_{i+1} - \phi_i) \tag{2.53}$$

Here, $\Psi(r_{i\pm\frac{1}{2}})$ is the TVD limiter discussed in the next section and $r_{i\pm\frac{1}{2}}$ is the smoothness indicator of ϕ. For the moment, we will assume that the value of $\Psi(r_{i\pm\frac{1}{2}})$ is going to be 1. In the next section, we will look at cases when that may not be true anymore.

Similar to the piecewise constant reconstruction, let us review how the MUSCL scheme works graphically. For this, have a look at Figure 2.4.

We are approximating here the same function shown in orange as we saw previously for the piecewise constant reconstruction; see Figure 2.3. As the MUSCL scheme is a bit more involved, we only look at how $\phi^L_{i+1/2}$ is calculated, to keep the figure clean.

First, we must calculate the slope between ϕ_i and ϕ_{i-1} according to Eq.(2.53). This is shown by the dashed black line in Figure 2.4. Then, we multiply this slope by 0.5, which is shown by the dotted black line.

Then, starting at ϕ_i (first term in the approximation of $\phi^L_{i+1/2}$ in Eq.(2.53)) and moving along the slope to the face on the right, we obtain the left-sided interpolation of the face, i.e. $\phi^L_{i+1/2}$.

We can see that this is closer to the orange function than the piecewise constant reconstruction. If we used a higher-order polynomial in the MUSCL scheme, we would likely get even closer to the orange function.

2.3.2. Flux limiters

In Eq.(2.53), we saw the flux limiter $\Psi(r \pm 1/2)$ and said that this is responsible for providing a smooth approximation for our face reconstructed values. In a mathematical

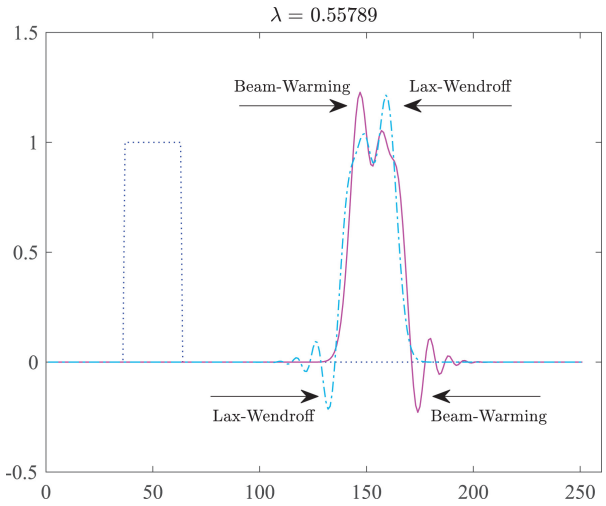


Figure 2.5: Example of how a square profile on the left is approximate by the Lax-Wendroff and Beam-Warming method shown on the right. Reproduced from Winnicki *et al.* (2019).

sense, flux limiters ensure that our approximation is TVD (total variation diminishing).

A numerical scheme can either be TVD or non-TVD. Though, in CFD, we prefer to use TVD schemes as these suppress non-physical oscillations. Take the simple solution of a square profile in Figure 2.5.

We see the initial profile on the left of the figure. If we apply the Lax-Wendroff or Beam-Warming method (two classical numerical schemes used before high-resolution schemes, i.e. MUSCL and WENO) to approximate its evolution over time, we obtain the solution shown on the right. We see some non-physical oscillations near the discontinuities.

Imagine the profile we see is that of the density. Then, we see that the oscillations near the discontinuities can go below zero. A negative density violates the laws of physics, and thus, this would introduce non-physical results.

The role of a TVD limiter, thus, is to identify these non-physical oscillations, mainly due to strong shock waves (e.g. discontinuities) and then remove them from the simulation.

To do that, the TVD limiter requires the smoothness indicator as an input. It is calculated as:

$$
\begin{aligned}
r_{i+\frac{1}{2}} &= \frac{\phi_{i+1} - \phi_i}{\phi_i - \phi_{i-1}} \\
r_{i-\frac{1}{2}} &= \frac{\phi_i - \phi_{i-1}}{\phi_{i+1} - \phi_i}
\end{aligned}
\tag{2.54}
$$

The smoothness indicator provides us with a numerical evaluation of the smoothness of the solution. Consider the three scenarios presented in Figure 2.6.

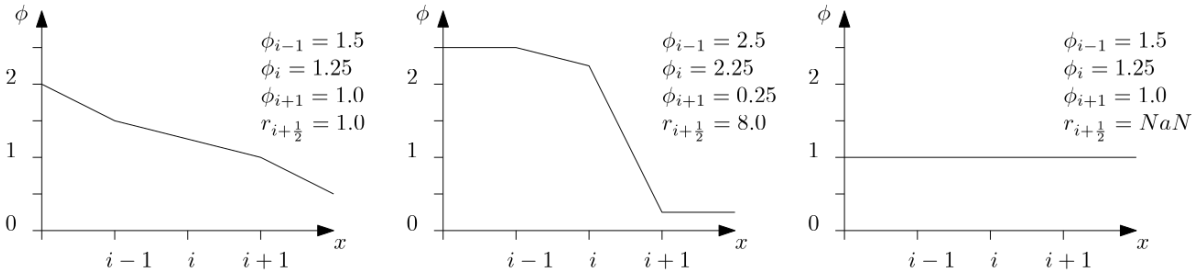


Figure 2.6: Three possible scenarios for smooth and non-smooth solutions.

For the first case (left side of the figure), we have a smooth solution in space, and if we plug in values to compute $r_{i+1/2}$, we see that we obtain a value of $r_{i+1/2} = 1.0$.

The second case (middle of the figure) shows a developing shock wave. In this case, if we plug in numbers, we get a smoothness indicator $r_{i+1/2} = 8.0$. If the shock wave (discontinuity) grows stronger (larger differences between the top and bottom limit of ϕ, i.e. $0.25 \geq \phi \geq 2.5$ in this case), then we get a larger value for the smoothness indicator.

The third case (right side of the figure) shows an extreme case of no change in the solution. You and me may say that this solution is smooth, but if we ask the smoothness indicator, it will give us a result of $r_{i+1/2} = NaN$ (not a number). This is because we have a division by zero in the denominator of Eq.(2.54), as both ϕ_i and ϕ_{i-1} have the same numerical value. To avoid this, we typically modify the smoothness indicator in the following way:

$$
\begin{aligned}
r_{i+\frac{1}{2}} &= \frac{\phi_{i+1} - \phi_i}{\phi_i - \phi_{i-1} + \epsilon} \\
r_{i-\frac{1}{2}} &= \frac{\phi_i - \phi_{i-1}}{\phi_{i+1} - \phi_i + \epsilon}
\end{aligned}
\tag{2.55}
$$

Here, ϵ is a small number close to zero that we are free to choose. A typical value for ϵ is 10^{-8}. Returning back to Figure 2.6, Eq.(2.55) would now provide us with a value of $r_{i+1/2} = 10^8$.

It is a rather large value, but it is no longer a NaN value, and it will be passed through the limiter function$\Psi(r)$, which will provide an upper bound.

So, let's look at what bounds the limiter will impose. Sweby showed that there is a general region for $\Psi(r)$ in which it is said to be TVD based on its smoothness indicator r. This is shown in Figure 2.7 on the left-hand side.

To make the limiter itself 2nd-order accurate, it is further restricted to the region shown on the right of Figure 2.7. So the job of the limiter $\Psi(r)$ is to return a value that is within the grey-shaded region in Figure 2.7, regardless of the value of r.

This is not a particularly difficult problem, and a myriad of functions can be thought of (and have! Just have a look at the list on Wikipedia, which isn't even complete and shows just the most popular ones).

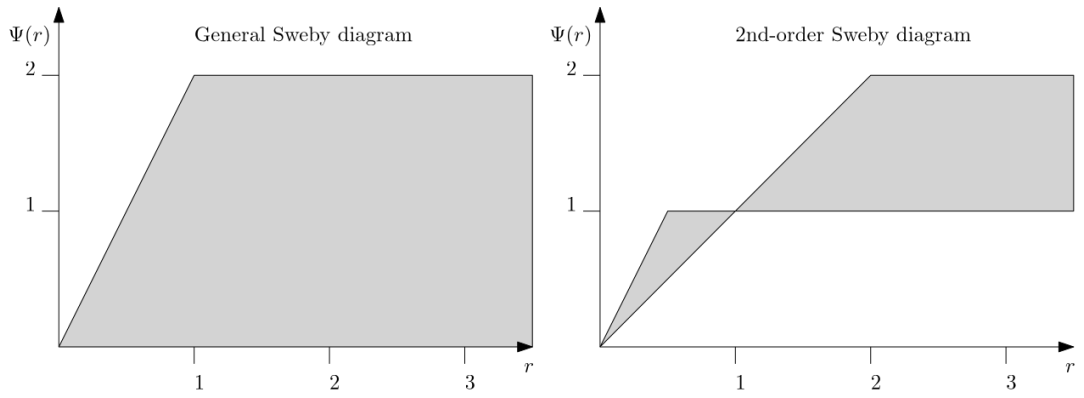


Figure 2.7: General Sweby diagram (left) and 2nd-order Sweby diagram (right).

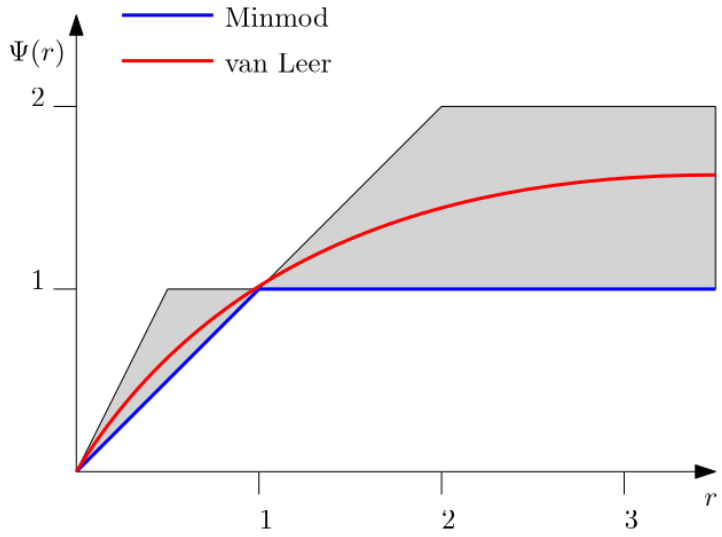


Figure 2.8: Examples of the minmod and van Leer limiter, which both pass through the 2nd-order shaded region.

I'm sure you could come up with your own limiter function. Go ahead and try it if you want (you can later extend the solver we will develop in Chapter 3 and check if your limiter does indeed work).

In the next two sections, we will look at two common limiters you may find in the wild and see how they are implemented, which are the minmod limiter (Section 2.3.2.1) and the van Leer limiter (Section 2.3.2.2). Both of these are shown in Figure 2.8.

At this point, I should point out that there is no consensus on which limiter is the best or worst. As is so often the case in CFD (e.g. selection of numerical schemes, turbulence model, boundary conditions, etc.), we have a wealth of them available, and some of them work better than others for certain situations. It is up to us to perform tests and determine which is best suited for the problem we are trying to solve.

2.3.2.1. Minmod limiter

As we can see from Figure 2.8, the minmod limiter traces the lower part of the 2nd-order TVD region. It is a common choice that works rather well, but it also has a discontinuity at $r = 1$. Quite a few limiters have a similar discontinuity, which can

sometimes affect convergence.

The minmod limiter is defined as:

$$\Psi(r)_{\text{minmod}} = \max[0, \min(1, r)] \tag{2.56}$$

2.3.2.2. Van Leer limiter

To avoid the discontinuity in the function definition, we may want to use a smooth profile instead. One such limiter is due to van Leer, which we can see in Figure 2.8. This function is defined as:

$$\Psi(r)_{\text{van Leer}} = \frac{r + |r|}{1 + |r|} \tag{2.57}$$

We can see that as r tends to infinity, Eq.(2.57) approaches a value of 2, thus staying within the 2nd-order TVD region.

2.3.3. The Riemann problem and approximate Riemann solvers

The Riemann problem is a very important concept in CFD. Compressible flows with strong discontinuities (shock waves) are virtually impossible to resolve well without solving the Riemann problem. OpenFOAM, for example, is famous for disregarding the Riemann problem altogether (in most solvers), and it is known for producing poor results for compressible flows (which even the OpenFOAM developers have admitted).

The Riemann problem, in its most basic form, is rather straightforward. We require a hyperbolic conservation law (i.e. the Euler equations in our case) with a discontinuous initial solution. Let's unpack this statement a bit.

We saw the conservation of mass (Eq.(2.5)), momentum (Eq.(2.3)), and total energy (Eq.(2.7)). As long as the system of equations has real and distinct eigenvalues, it is said to be hyperbolic.

In a more physical sense, real and distinct eigenvalues mean that we have different wave speeds within our system. In the case of the Euler equations, we obtain three distinct eigenvalues corresponding to the characteristic wave speed of a shock wave, a contact discontinuity, and rarefaction waves.

When we later run our code, we can see these different wave speeds by animating through our solution. Each wave will expand at its own speed, which corresponds to one of the eigenvalues of the system.

The discontinuous initial data can be written mathematically as:

$$\mathbf{U}(x, t = 0) = \begin{cases} \mathbf{U}_L & x < x_0 \\ \mathbf{U}_R & x \geq x_0 \end{cases} \tag{2.58}$$

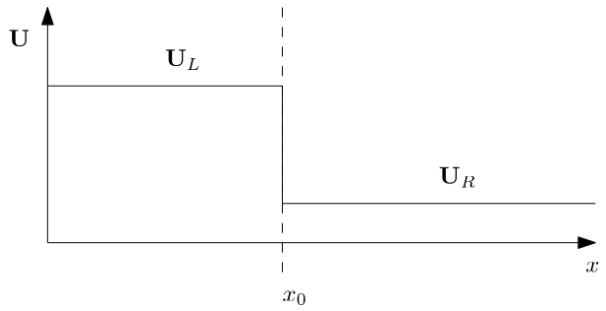


Figure 2.9: Discontinuous initial data according to the Riemann problem.

This can also be shown visually, as shown in Figure 2.9. Thus, as long as there is a discontinuity somewhere in the initial data, and we have a hyperbolic conservation law (or system of conservation laws, i.e. conservation of mass, momentum, and total energy), we have a Riemann problem.

Each problem requires a solution, and this is where Riemann solvers come in. A Riemann solver simply looks at the left and right states (i.e. $\mathbf{U}_L$ and $\mathbf{U}_R$ in Eq.(2.58) and in Figure 2.9), and aims to consolidate these into a single state $\tilde{\mathbf{U}}$.

More specifically, in the case of conservation laws, we first approximate the left-sided and right-sided states of $\mathbf{U}$ using an appropriate reconstruction scheme (e.g. the piecewise constant scheme (Section 2.3.1.1) or the MUSCL scheme (Section 2.3.1.2)), and then compute the fluxes based on these reconstructed states. The Riemann solver will then consolidate these fluxes into a single flux.

In a mathematical sense, we can write the solution of a Riemann solver as:

$$\tilde{\mathbf{F}}(\mathbf{U})_{i+\frac{1}{2}} = f[\mathbf{F}(\mathbf{U}^L_{i+\frac{1}{2}}), \mathbf{F}(\mathbf{U}^R_{i+\frac{1}{2}})] \tag{2.59}$$

If we return to Figure 2.3 for a second, where we looked at the piecewise constant reconstructed states, we saw that the left-sided and right-sided approximations ϕ were rather far apart. The Riemann solver now tries to correct these predictions so as to bring the solution closer to the actual solution (here, shown by the orange curve).

There are several different Riemann solvers out there with varying degrees of complexity. In my opinion, the Rusanov Riemann solver (also sometimes referred to as the local Lax-Friedrichs flux) is a fairly robust and useful Riemann solver that is easy to implement and understand. We will review this in the next section.

2.3.3.1. The Rusanov Riemann solver

As alluded to above, the Rusanov (local Lax-Friedrichs) Riemann solver is straightforward and easy to implement. It contains only two steps:

1. Obtain a suitable, characteristic wave speed.
2. Compute the flux based on the left-sided and right-sided approximations of $\mathbf{U}$.

There is no shortage of wave speed estimations, all more or less following a similar approach. This typically involves some form of the eigenvalues of the system. A

common wave speed estimation is the one provided below:

$$S = \max(|u_L| + a_L, |u_R| + a_R) \tag{2.60}$$

Here, $u_{L,R}$ are the left-sided and right-sided reconstructed x velocity component and $a_{L,R}$ is the speed of sound and calculated as:

$$a_{L,R} = \sqrt{\frac{\gamma p_{L,R}}{\rho_{L,R}}} \tag{2.61}$$

The Rusanov Riemann solver then computes the flux as:

$$\tilde{\mathbf{F}}(\mathbf{U})_{i+\frac{1}{2}} = \frac{1}{2}\left[\mathbf{F}(\mathbf{U}^L_{i+\frac{1}{2}}) + \mathbf{F}(\mathbf{U}^R_{i+\frac{1}{2}})\right] - S\left(\mathbf{U}^R_{i+\frac{1}{2}} - \mathbf{U}^L_{i+\frac{1}{2}}\right) \tag{2.62}$$

There are two observations we can make at this point. For cases where the wave speed estimation reaches zero, the second term in Eq.(2.62) becomes negligible. In this case, only the first term contributes to the flux calculation, in which case we simply take an average of the left-sided and right-sided fluxes.

Secondly, if we set $S = 0.5\Delta x/\Delta t$, then Eq.(2.62) becomes the Lax-Friedrichs scheme. Thus, since we compute the wave speed S for each cell in the Rusanov Riemann solver, it is also sometimes called the local Lax-Friedrichs scheme.

Similar to Eq.(2.62), we can also compute the fluxes for the face at $i-1/2$, i.e. $\tilde{\mathbf{F}}(\mathbf{U})_{i-\frac{1}{2}}$. Once we have obtained these fluxes, we can substitute the fluxes in Eq.(2.43) with those obtained by Eq.(2.62). This results in:

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{1}{\Delta x}\left(\tilde{\mathbf{F}}(\mathbf{U})_{i+\frac{1}{2}} - \tilde{\mathbf{F}}(\mathbf{U})_{i-\frac{1}{2}}\right) = 0 \tag{2.63}$$

A solution to Eq.(2.63) provides a high-resolution approximation to the Euler equation. The ingredients we have just looked at are essentially the same for any compressible flow solver (e.g. higher-order reconstructions, flux limiters, and the Riemann problem).

While we now have a good idea of how to approximate the fluxes in Eq.(2.63), we have not discussed how to approximate the time derivative. We will look at this in the next section.

2.3.4. Numerical approximation in time

In this section, we will look at a common scheme to advance the solution in time. This will provide us with an approximation for the time derivative, i.e. $\partial \mathbf{U}/\partial t$ in Eq.(2.63).

2.3.4.1. Euler scheme (1st-order)

The 1st-order Euler scheme in time is the simplest form of time integration. It assumes we can replace the time derivative with a first-order approximation based on the Taylor-Series. Developing a Taylor-series in time around $\mathbf{U}(t + \Delta t)$ results in:

$$\mathbf{U}(t+\Delta t) = \mathbf{U}(t) + \frac{\partial \mathbf{U}}{\partial t}\Delta t + \mathcal{O}(\Delta t^2) \tag{2.64}$$

Here, we are only concerned with the first two terms in the Taylor-series expansion. We will see that this is responsible for the first-order accuracy in a second. We can solve Eq.(2.64) for the time derivative $\partial \mathbf{U}/\partial t$ to arrive at:

$$\frac{\partial \mathbf{U}}{\partial t}\Delta t = \mathbf{U}(t+\Delta t) - \mathbf{U}(t) + \mathcal{O}(\Delta t^2) \tag{2.65}$$

Dividing both sides by Δt results in:

$$\frac{\partial \mathbf{U}}{\partial t} = \frac{\mathbf{U}(t+\Delta t) - \mathbf{U}(t)}{\Delta t} + \mathcal{O}(\Delta t) \tag{2.66}$$

The truncation error $\mathcal{O}(\Delta t)$ was also divided by Δt, and as a result, the truncation error was reduced from $\mathcal{O}(\Delta t^2)$ to $\mathcal{O}(\Delta t)$. Since the exponent determines the order of the approximation, we can see that our approximation is now first-order accurate (i.e. the exponent in $\mathcal{O}(\Delta t) = \mathcal{O}(\Delta t^1)$ is 1).

In CFD, we typically introduce subscripts for dealing with time, i.e. Eq.(2.66) can be rewritten as

$$\frac{\partial \mathbf{U}}{\partial t} = \frac{\mathbf{U}^{n+1} - \mathbf{U}^{n}}{\Delta t} + \mathcal{O}(\Delta t) \tag{2.67}$$

Values at t (or time level n) are always known, either from the previous time step or the initial conditions (if we are just starting with the simulation). Thus, if we insert the approximation of Eq.(2.67) into Eq.(2.63), we obtain:

$$\frac{\mathbf{U}_i^{n+1} - \mathbf{U}_i^{n}}{\Delta t} + \frac{1}{\Delta x}\left(\tilde{\mathbf{F}}(\mathbf{U})_{i+\frac{1}{2}} - \tilde{\mathbf{F}}(\mathbf{U})_{i-\frac{1}{2}}\right) = 0 \tag{2.68}$$

We need to make an assumption now. At what time level do we want to approximate the fluxes, i.e. $\tilde{\mathbf{F}}(\mathbf{U})_{i\pm 1/2}$? We have two choices, either at the time level n or at the time level $n+1$. The former will lead to an explicit discretisation in time, while the latter results in an implicit discretisation.

Explicit time integrations are easy to implement, and for the sake of simplicity, this is what we will do here. We will see why in a second. So let's introduce this time level into Eq.(2.68), which results in:

$$\frac{\mathbf{U}_i^{n+1} - \mathbf{U}_i^{n}}{\Delta t} + \frac{1}{\Delta x}\left(\tilde{\mathbf{F}}(\mathbf{U})^{n}_{i+\frac{1}{2}} - \tilde{\mathbf{F}}(\mathbf{U})^{n}_{i-\frac{1}{2}}\right) = 0 \tag{2.69}$$

We can now put all the values of n on the right-hand side of the equation and collect all terms with $n+1$ on the left-hand side of the equation. This provides us with the following equation:

$$\mathbf{U}_i^{n+1} = \mathbf{U}_i^n - \frac{\Delta t}{\Delta x}\left(\tilde{\mathbf{F}}(\mathbf{U})_{i+\frac{1}{2}}^n - \tilde{\mathbf{F}}(\mathbf{U})_{i-\frac{1}{2}}^n\right) \tag{2.70}$$

From Eq.(2.70), we can see that there is only a single unknown quantity, i.e. $\mathbf{U}_i^{n+1}$. Thus, we can directly (explicitly) calculate the solution $\mathbf{U}$ at the next step (time level $n+1$). In contrast, had we assumed the fluxes would be evaluated at time level $n+1$ in Eq.(2.68), we would get:

$$\frac{\mathbf{U}_i^{n+1} - \mathbf{U}_i^n}{\Delta t} + \frac{1}{\Delta x}\left(\tilde{\mathbf{F}}(\mathbf{U})_{i+\frac{1}{2}}^{n+1} - \tilde{\mathbf{F}}(\mathbf{U})_{i-\frac{1}{2}}^{n+1}\right) = 0 \tag{2.71}$$

Putting again all unknowns (at time level $n+1$) on the left-hand side of the equation and all terms with n on the right-hand side of the equation, we obtain the following implicit time discretisation:

$$\frac{1}{\Delta t}\mathbf{U}_i^{n+1} + \frac{1}{\Delta x}\tilde{\mathbf{F}}(\mathbf{U})_{i+\frac{1}{2}}^{n+1} - \frac{1}{\Delta x}\tilde{\mathbf{F}}(\mathbf{U})_{i-\frac{1}{2}}^{n+1} = \frac{1}{\Delta t}\mathbf{U}_i^n \tag{2.72}$$

We now have several unknowns on the left-hand side of the equation. To solve this, we would have to insert the reconstruction scheme for each flux, until we have the fluxes expressed in terms of $\mathbf{U}_{i\pm1}$. We will end up with a linear system of equations that we need to solve.

This is great for stability, but if you are just starting out writing your first solver, it is a bad idea to jump straight to implicit methods. Furthermore, for certain applications, implicit methods are not the right choice anyway. Scale-resolved turbulence modelling is mainly done with explicit schemes as they are faster than implicit schemes, and we do not need the added stability here (as physical constraints limit us to $CFL < 1$).

I won't go into further details here about implicit schemes, but I have written about them at length on cfd.university. If you want to see how to discretise an equation using an implicit scheme, and how to implement the resulting linear system of equations, then feel free to check out the article linked below:

Resource 2.3 How to write a CFD library: Discretising the model equation

2.4. Boundary conditions

In this section, I want to give you the essentials to master boundary conditions. There is a lot more to them than what I can cover here, and I could probably write an entire guide just about boundary conditions! Honestly, they are rather tricky to get right.

There are only 3 basic types of boundary conditions. These are periodic, Dirichlet, and Neumann boundary conditions (although you could argue that periodic boundary conditions are a special case of Dirichlet boundary conditions). If you master these three types, you can handle any flows.

We will review these three basic types of boundary conditions first and then discuss how we can combine different variations of them for our conserved variables so that

we can impose boundary conditions such as solid walls, symmetry faces, inflows, and outflows. We won't need all of them for our solver, but I have included them here for completeness.

2.4.1. Periodic boundary conditions

Periodic boundaries are the simplest type of boundary conditions we can think of. They state that what leaves the domain on one side must enter the domain on the other side. Have a look at Figure 2.2 again. Here, we can see a domain that has two boundaries. B_L is the boundary on the left and B_R on the right.

If we need information to the left of B_L, then we simply look to the left of B_R. In Figure 2.2, this means that the next centroid to the left of B_L is the cell with ID 5. Similarly, if we require information to the right of B_R, then we simply look to the right of B_L. The next point past B_R is the cell with ID 1 in Figure 2.2. We can put this in equation form as:

$$\begin{aligned} \phi_{B_L-1} &= \phi_{B_R-1} \\ \phi_{B_R+1} &= \phi_{B_L+1} \end{aligned} \tag{2.73}$$

Periodic boundary conditions are particularly useful when we have a domain with repeated features, and we want to focus on just one element that is repeated through the domain (e.g. a single rotor element on a wind turbine).

2.4.2. Dirichlet boundary conditions

Dirichlet boundary conditions require us to impose a value at the boundary. For example, at an inlet/inflow boundary, we typically prescribe the velocity vector. Giving explicit values for the velocity (or, indeed, any other quantity) is a form of a Dirichlet boundary condition.

Mathematically speaking, and referring back to Figure 2.2 again, we can write Dirichlet boundary conditions for either end of the domain as:

$$\begin{aligned} \phi_{B_L} &= \phi_A \\ \phi_{B_R} &= \phi_B \end{aligned} \tag{2.74}$$

Here, ϕ_A and ϕ_B are some values we want to impose at the boundary for ϕ. Thus, Dirichlet boundary conditions always require us to know the value on the boundary, and typically, we have a uniform distribution across the boundary as a result, e.g. a uniform inflow velocity profile.

2.4.3. Neumann boundary conditions

Finally, we have Neumann-type boundary conditions. These are formulated as a gradient of a quantity. Let's look at the definition first and then develop an intuition for it (if

you are seeing it for the first time, it likely won't make sense). The Neumann boundary condition for the domain in Figure 2.2, is written for either side as:

$$\begin{aligned} \left.\frac{\partial \phi}{\partial n}\right|_{B_L} &= \phi_A \\ \left.\frac{\partial \phi}{\partial n}\right|_{B_R} &= \phi_B \end{aligned} \tag{2.75}$$

Here, ϕ_A and ϕ_B are, again, some values that we can impose. A fairly common choice is $\phi_A = \phi_B = 0$, i.e. there is no gradient of ϕ across the boundary. The denominator ∂n indicates that this gradient is to be evaluated along the normal direction to the boundary.

Let's use temperature as an example. If we have a Neumann-type boundary condition for the temperature, and we write it as:

$$\left.\frac{\partial T}{\partial n}\right|_{B_L} = 0 \tag{2.76}$$

In this case, we say that there is no temperature gradient across the boundary. We use Eq.(2.76) for adiabatic solid walls, for example. Of course, if we wanted to have a heat flux across the boundary (i.e. we wanted to impose heating on our system), then we would have:

$$\left.\frac{\partial T}{\partial n}\right|_{B_L} > 0 \tag{2.77}$$

Here, we allow for some temperature gradient at the boundary, i.e. we allow for heat to flow over the boundary into our domain.

Ok, so let's see how we would implement Eq.(2.75). First, we need to discretise the gradient. For the boundary on the left, i.e. B_L, we use a right-sided, first-order approximation of the gradient and use Figure 2.2 again as a reference. Then, we obtain the following approximation:

$$\left.\frac{\partial \phi}{\partial n}\right|_{B_L} \approx \frac{\phi_1 - \phi_{B_L}}{\Delta x} = \phi_A \tag{2.78}$$

Here, we use a Taylor series again to obtain an approximation for the gradient, as we did in Section 2.3.4.1. The only difference is that our Taylor series is now in space (i.e. in terms of x rather than in time t).

We can see that Eq.(2.78) contains the value of ϕ_{B_L}. Thus, we can simply solve for it as follows:

$$\phi_{B_L} = \phi_1 - \phi_A \Delta x \tag{2.79}$$

In case $\phi_A = 0$, Eq.(2.79) further simplifies to:

$$\phi_{B_L} = \phi_1 \tag{2.80}$$

We can also repeat the same steps for the Boundary on the right, i.e. B_R. In this case, we have:

$$\left.\frac{\partial \phi}{\partial n}\right|_{B_R} \approx \frac{\phi_{B_R} - \phi_5}{\Delta x} = \phi_B \tag{2.81}$$

This can be solved for ϕ_{B_R} as:

$$\phi_{B_R} = \phi_5 + \phi_B \Delta x \tag{2.82}$$

If we assume a zero-gradient condition again as in Eq.(2.80), then we can simplify Eq.(2.82) again to:

$$\phi_{B_R} = \phi_5 \tag{2.83}$$

In other words, we look at the values of ϕ at the interior points next to the boundary and copy them into the boundary points for ϕ. In a sense, a Neumann boundary condition is a Dirichlet boundary condition, but we don't know the value we are imposing (rather, it is calculated from the interior domain).

Another way to look at the Neumann boundary condition is this: If we don't know what values to prescribe at a boundary, then we probably want a Neumann-type boundary condition. Or, if we are interested in quantities on the boundary, then it is also likely a Neumann boundary condition.

For example, think of a simple simulation of the flow around an airfoil. Typically, we are interested in the lift and drag of the arifoil. The lift is mainly calculated from the pressure, and so, since we are interested in the pressure (i.e. it is a solution of our simulation), it must be a Neumann-type boundary condition. Indeed, the pressure is usually given as a Neumann-type boundary condition at solid walls.

Another example is the outflow velocity profile within a channel flow. We don't know the shape of the velocity profile, and thus, we need to impose the velocity as a Neumann-type boundary condition. Since we copy the velocity profile from the interior to the boundary, we essentially look at the computed velocity profile at the points next to the boundary, which was computed on the interior.

2.4.4. Common boundary conditions

Hopefully, the discussion in the previous sections clearly demonstrated how to use periodic, Dirichlet, and Neumann-type boundary conditions. In this section, I want to build upon that knowledge and show which type we need to use for which variables to achieve a very specific type of boundary condition.

In the solver we develop in Chapter 3, we'll impose some form of boundary conditions, but they are inconsequential for the development of the solution (as we never reach the boundary). So, you won't need any of the following discussion on boundary conditions, but I thought of including them so that you know how you could extend the solver should you wish to.

2.4.4.1. Solid wall boundary conditions

A solid wall is likely something you will encounter in most applications. Any type of solid surface will be represented as a solid wall. The only question we need to ask ourselves is whether it is a viscous or inviscid simulation.

In our case, we deal with inviscid flows, and this means that the flow should simply flow over a surface but not slow down. Only viscous forces will impose a boundary layer and thus slow down the flow near a solid surface.

Thus, we can write the boundary conditions for solid walls as follows:

- Inviscid wall:
  - Velocity (wall-normal direction): Dirichlet (zero velocity)
  - Velocity (wall parallel direction): Neumann (zero gradient)
  - Pressure: Neumann (zero gradient)
  - Density: Neumann (zero gradient)
  - Temperature: Neumann (zero gradient or specified heat flux)
  - Energy: Neumann (zero gradient or specified energy flux)
- Viscous wall:
  - Velocity (wall-normal direction): Dirichlet (zero velocity)
  - Velocity (wall parallel direction): Dirichlet (zero velocity)
  - Pressure: Neumann (zero gradient)
  - Density: Neumann (zero gradient)
  - Temperature: Neumann (zero gradient or specified heat flux)
  - Energy: Neumann (zero gradient or specified energy flux)

Thus, the only difference between an inviscid and a viscous wall is how we specify the velocity in the wall's parallel direction.

2.4.4.2. Symmetry boundary conditions

A symmetry boundary condition is one where we assume that we have flow on the other side of the boundary, which is symmetrical to the flow we are solving for. We

often employ symmetry boundary conditions to reduce the domain size and thus speed up the simulation.

For example, if we were to simulate the flow through a pipe or channel, we could just model half of the pipe or channel and impose a symmetry boundary condition at the centerline of the pipe or channel. This reduces the number of cells in our mesh by a factor of 2 without losing any accuracy.

A symmetry and inviscid wall are virtually the same, with the exception that all quantities are given as a Neumann boundary condition (apart from the velocity in the wall-normal direction, which remains a Dirichlet-type boundary condition). In other words, a symmetry boundary condition is imposed using the following:

- Velocity (wall-normal direction): Dirichlet (zero velocity)
- Velocity (wall parallel direction): Neumann (zero gradient)
- Pressure: Neumann (zero gradient)
- Density: Neumann (zero gradient)
- Temperature: Neumann (zero gradient)
- Energy: Neumann (zero gradient)

2.4.4.3. Inlet boundary conditions

Inlet (as well as outlet boundary conditions, discussed in the next section) are where things get interesting. You'd think these are simple to impose, but in reality, there are some nuances which can make life rather difficult.

Luckily for us, these nuances only really start to appear once we deal with turbulence and scale-resolved turbulence modelling in particular. If we are dealing with inviscid flows, life is easy again.

For inlet boundary conditions, we first have to determine the inflow Mach number, as this will determine how many Dirichlet and Neumann boundary conditions we have. For completeness, the Mach number is calculated as $Ma = u/a$, where u is the local velocity and a the local speed of sound. If the Mach number is below 1, then the flow is said to be subsonic. If the Mach number is above 1, then the flow is said to be supersonic.

With this definition in mind, we can specify the boundary conditions for an inlet as follows:

- Supersonic inlet:
  - All variables: Dirichlet
- Subsonic inlet:
  - All variables apart from one: Dirichlet
  - One variable: Neumann

For supersonic inlets, we impose values for velocity, density, pressure, temperature, and so on at the inlet directly (Dirichlet). However, if the Mach number is below 1 at

the inlet, then we have to specify Dirichlet boundary conditions for all but one variable. One of the variable has to be a Neumann-type boundary condition.

Which variable is a Neumann-type boundary condition is up to us to decide. For example, if we wanted to specify a certain velocity at the inlet, we would impose the velocity as a Dirichlet boundary condition and then use a Neumann-type for the pressure or density. On the other hand, we may want to specify a pressure drop between the inlet and outlet, in which case we would specify the pressure as Dirichlet boundary conditions and then use a Neumann type for the velocity.

The choice is ours, and depending on the choice, we end up with either a velocity inlet or pressure/density inlet boundary condition.

2.4.4.4. Outlet boundary conditions

Outlet boundary conditions are very similar to inlet boundary conditions in that we first need to establish whether the flow is supersonic or subsonic at the outlet. Based on the type of flow, we can then impose the following conditions:

- Supersonic outlet:
  - All variables: Neuamnn
- Subsonic outlet:
  - All variables apart from one: Neumann
  - One variable: Dirichlet

If the flow is supersonic, the case is easy. We simply apply Neumann-type boundary conditions everywhere. If we reach subsonic flows locally, we have to specify one Dirichlet boundary condition for a variable of our choosing. Typically, we fix the outlet pressure and set it to an ambient pressure.

However, we could also let the pressure develop and require a specific, fixed value for the velocity at the outlet. This may make sense if we want to target a specific mass flow rate (for which we can compute the required velocity at the outlet) from which we can then determine the pressure drop.

Depending on the type of problem we want to solve, we have to choose between a Dirichlet or a Neumann-type boundary condition accordingly. There is no right or wrong answer, and with a bit of practice, you'll develop an intuition for which type to use.

But if in doubt, remember what we discussed above. If you are interested in a specific quantity at the boundary (e.g. the pressure for the calculation of the pressure drop), you'll likely need to impose it as a Neumann-type boundary condition.

And with that, you should now have a good basic understanding of how to deal with boundaries. Remember that we can always bring back any type of boundary condition to a Dirichlet or a Neumann-type boundary condition (and, in some cases, to a periodic type boundary condition). Let us now continue our discussion by looking at initial conditions.

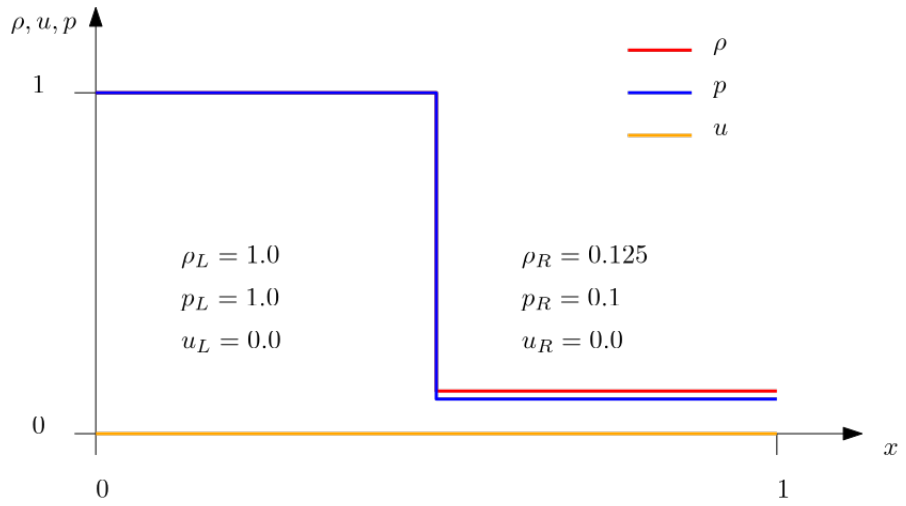


Figure 2.10: The Sod shock tube problem.

2.5. Initial conditions

Initial conditions are those conditions we set at the beginning of the simulation. Our simulation will advance the solution in time based on these initial conditions.

Typically, the initial conditions we set are not important, as our simulation will always converge to our boundary conditions.

However, the case we will look at depends only on the initial conditions since we stop our simulation early, before any disturbance from the initial profile has reached the boundary. However, if we run our simulation for long enough, it should always reach the same solution, regardless of the initial conditions.

The initial condition may also influence the convergence speed, i.e. if we are seeking a steady-state solution, then an initial condition close to the final solution may provide faster convergence than any arbitrary initial condition (like setting inflow values for all interior points).

2.5.1. The Sod shock tube problem

If you are writing your first (compressible) CFD solver, the Sod shock tube problem is likely the first problem you encounter. It is one of the simplest and most well-known and well-studied examples out there. It is insensitive to the boundary conditions, and with relatively little effort (i.e. a 1D inviscid solver is sufficient), we can study complex non-linear phenomena of the Euler equations.

The Sod shock tube problem is essentially a Riemann problem. Return to Figure 2.9. Here, we saw that we have two states, i.e. the left and right state. The Sod shock tube problem is using this as the basis and defines the initial conditions with a discontinuity for all primitive variables.

Specifically, on a domain of Length $L = 1$, where we start at $x = 0$ and go to $x = L$, we have the following initial values:

$$\begin{pmatrix} \rho_L \\ p_L \\ u_L \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \tag{2.84}$$

$$\begin{pmatrix} \rho_R \\ p_R \\ u_R \end{pmatrix} = \begin{pmatrix} 0.125 \\ 0.1 \\ 0 \end{pmatrix} \tag{2.85}$$

This setup is also shown in Figure 2.10, where we see the initial data for the density ρ, the pressure p, and the velocity u for the left and right states. We can see that the discontinuity appears at $x = 0.5$.

With these initial conditions specified, we now have everything we need to start writing our solver. Before we do, though, let's have a quick review of what we just went through, as it was quite a lot. The next section provides you with a quick summary of the steps and equations required to solve the 1D Euler equations.

2.6. Step-by-step summary

In this section, I want to condense all of the information we just went through and put that into a short summary. We review all the steps we need to take and see all of the key equations again. The steps we need to take are:

1. Define the governing equations. We have settled for the 1D Euler equations. These were given in discretised form by Eq.(2.44) as:

$$\frac{\partial}{\partial t}\begin{pmatrix} \rho \\ \rho u \\ E \end{pmatrix} + \frac{1}{\Delta x}\left(\begin{pmatrix} \rho u \\ \rho u^2 + p \\ u(E+p) \end{pmatrix}_{i+\frac{1}{2}} - \begin{pmatrix} \rho u \\ \rho u^2 + p \\ u(E+p) \end{pmatrix}_{i-\frac{1}{2}}\right) = 0$$

2. We see that we need to approximate the values for the flux vector $\mathbf{F}(\mathbf{U})$ at the faces $i + 1/2$ and $i - 1/2$. We either use a piecewise constant reconstruction (Eq.(2.52)) or the MUSCL scheme (Eq.(2.53)). These were given as:

   - Piecewise constant reconstruction (Eq.(2.52)):

$$\phi^L_{i+\frac{1}{2}} = \phi_i$$
$$\phi^R_{i-\frac{1}{2}} = \phi_i$$

   - MUSCL scheme (Eq.(2.53)):

$$\phi^L_{i+1/2} = \phi_i + \frac{\Psi(r_{i+\frac{1}{2}})}{2}(\phi_i - \phi_{i-1})$$
$$\phi^R_{i-1/2} = \phi_i - \frac{\Psi(r_{i-\frac{1}{2}})}{2}(\phi_{i+1} - \phi_i)$$

3. If we are using the MUSCL scheme, we also need to evaluate the flux limiter $\Psi(r)$. There are countless flux limiters available, and we have reviewed the minmod (Eq.(2.56)) and the van Leer (Eq.(2.57)) flux limiters. These are given as:

   - Minmod limiter (Eq.(2.56)):

   $$\Psi(r)_{\text{minmod}} = \max[0, \min(1, r)]$$

   - Van Leer limiter (Eq.(2.57)):

   $$\Psi(r)_{\text{van Leer}} = \frac{r + |r|}{1 + |r|}$$

   - Here, r is the smoothness indicator defined by Eq.(2.55) and given as:

   $$r_{i+\frac{1}{2}} = \frac{\phi_{i+1} - \phi_i}{\phi_i - \phi_{i-1} + \epsilon}$$
   $$r_{i-\frac{1}{2}} = \frac{\phi_i - \phi_{i-1}}{\phi_{i+1} - \phi_i + \epsilon}$$

   ϵ is a small number to avoid divisions by zero and set here to $\epsilon = 10^{-8}$.

4. Using either the piecewise constant reconstruction or the MUSCL scheme, we end up with left-sided and right-sided interpolated values for our variables, i.e. we obtain $\phi^{L,R}_{i\pm1/2}$. These values are typically not the same, and thus, we need to consolidate these fluxes into a single flux. This is the job of the Riemann solver, and we are using the Rusanov (or local Lax-Friedrichs) Riemann solver here, as given by Eq.(2.62). This is given as:

   $$\tilde{\mathbf{F}}(\mathbf{U})_{i+\frac{1}{2}} = \frac{1}{2}\left[\mathbf{F}(\mathbf{U}^L_{i+\frac{1}{2}}) + \mathbf{F}(\mathbf{U}^R_{i+\frac{1}{2}})\right] - S\left(\mathbf{U}^L_{i+\frac{1}{2}} - \mathbf{U}^R_{i+\frac{1}{2}}\right)$$

5. With a single flux for each face in the mesh, we can now update the solution in time using a first-order Euler time integration scheme. This is given by Eq.(2.70) as:

   $$\mathbf{U}^{n+1}_i = \mathbf{U}^n_i - \frac{\Delta t}{\Delta x}\left(\tilde{\mathbf{F}}(\mathbf{U})^n_{i+\frac{1}{2}} - \tilde{\mathbf{F}}(\mathbf{U})^n_{i-\frac{1}{2}}\right)$$

Returning back to the generalised CFD solver framework in Figure 2.1, the step-by-step summary outlined below is what we do in the *Solving (time loop)* block, specifically within the *Solve equations* sub-block.

3

Writing a CFD solver

In the previous chapter, we developed all of the theoretical knowledge we needed to write our Euler equation-based CFD solver. While the focused discussion of the previous chapter can be found in many CFD textbooks, what follows typically is omitted.

We will go through each line of code, discuss what it is doing, and link it back to equations from the previous chapter where applicable. I really hope this section will provide clarity on how to write a CFD solver, as going from theory to working code is not always straightforward.

Before we look at the code, I need to address the (conscious) choice I made about the code structure. The way I have written the code is not how you would actually write a production-ready CFD solver.

However, I have always found that writing optimised and efficient code is completely the opposite of what you need when you want to show how something is implemented, i.e. writing code for education purposes.

Too often have I looked at code that was written to demonstrate a particular algorithm, only to find that it was full of unnecessary optimisations that made it more difficult to understand the code.

Thus, I have decided to write the code in the clearest form possible; this includes putting the entire code into a single `main()` function and not structuring the code into smaller functions, potentially split into several header and source files, or using any advanced C++ features.

You should be able to open the code and read through it like you would read a book. This isn't how you would write a C++ application, but this is how you *should* write a C++ application for education purposes. I hope this will help translate the theory into practice.

3.1. The main structure of the code

Before we jump into the implementation of the equations, I want to first introduce the main structure of the code and get some definitions out of the way. We won't look at

the entire code all at once but rather at chunks of code, one at a time.

Thus, it helps to have an understanding of the structure of the code first so that we know where each of the code snippets will go.

3.1.1. Folder structure

Within the `code/` directory, you will find three files:

- `euler.cpp`: This is the main c++ source file. It contains the entire code of the CFD solver that implements the Euler equations for a 1D, inviscid compressible flow.
- `CMakeLists.txt`: This file is used by CMake to build the executable in a platform-independent way. You will need to have CMake installed to make use of it (which you should if you followed the resources listed in Chapter 1)
- `QuickStartGuide.pdf`: This file shows you how to compile the code using CMake and then how to execute it.

I have debated for a while whether I should include CMake here or not. In the end, I have decided that if you know how to install a C++ compiler, installing CMake shouldn't be that much more complicated (and I have provided a dedicated guide on how to install everything for Windows, Linux, and macOS in Chapter 1).

In the end, knowing how to use a build system like CMake is just as important as knowing how to write code itself, especially once your project becomes more complex and consists of many more files.

I have an entire series on how to use CMake for CFD applications, and if this is the first time you have heard about CMake, I would recommend having a brief look through my introductory article on CMake linked below:

Resource 3.1 Introduction to CMake for CFD practitioners

We won't use any advanced features here, and our CMake file is so trivial that it seems almost pointless to use it in the first place. I have included it, however, because it gives you the best chance of quickly compiling your solver regardless of your operating system.

If you already feel comfortable with a C++ compiler, you can safely ignore CMake and compile the code directly through the terminal. I won't stop you!

3.1.2. The main() function

In Listing 1, we can see the `main()` function of our CFD solver. As discussed at the beginning of this chapter, we are dealing here only with a single `main()` function that contains all code.

I have removed all of the implementations and have only retained the structure of the code. Where I have removed the code, I have left a comment indicating what is being implemented here, along with a section reference. In the next sections, we will look

```
// header include files (Section 3.1.3)

// enum and struct definitions (Section 3.1.4)

int main() {

  // pre-processing (Section 3.2)

  while (parameters.time < parameters.endTime) {
    // solving (section 3.3)
  }

  // post-processing (Section 3.4)

  return 0;
}
```

Listing 1: Structure of the main() function.

at the code implementation in detail for each comment.

In case you ever get lost at any point, you can check the entire code in Appendix A where the entire content of the `euler.cpp` file can be found.

3.1.3. Header include files

The first part of any C++ program is the header include section, and this code is no different. Listing 2 shows the required header files for the solver. As you can see, we only use C++ standard header files and no third-party libraries. Thus, as long as you have a C++ compiler (with the C++ 2017 standard at a minimum) available, the code provided will compile.

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <array>
#include <cmath>
#include <fstream>
#include <sstream>
#include <string>
```

Listing 2: Required header files for the solver

We include `iostream` and `iomanip` to output results to files and change the formatting of the output.

The `vector` and `array` headers are used later for defining the data structures in our code, e.g. the conserved variable vector $\mathbf{U}$ and the fluxes $\mathbf{F}(\mathbf{U})$.

The `cmath` header includes basic math operations such as the square root, raising a number to a specific power, and taking the absolute value of a variable.

The `fstream` header allows us to write files to disk, and it handles things such as file creation and overwriting if required.

The string stream (`sstream`) and `string` headers allow us to work with strings instead of just the primitive characters that are a basic C type. This makes our life easier when we dynamically create the file name for our output files based on the current timestep.

3.1.4. Enum and struct definitions

Along with the header files we need, we also define a few enums and one struct (custom data type) to help us organise our code a bit more and make the code, in general, more readable. The definitions are provided in Listing 3.

```
1  // global enums for easy variable access
2  enum SCHEME { CONSTANT = 0, MUSCL};
3  enum LIMITER { NONE = 0, MINMOD, VANLEER};
4  enum FACE {WEST = 0, EAST};
5
6  // definition for case parameter structure to hold case-specific settings
7  struct caseParameters {
8    int numberOfPoints;
9    double gamma;
10   double domainLength;
11   double endTime;
12   double CFL;
13   double dx;
14   double time;
15   int timeStep;
16 };
```

Listing 3: Enum and struct definitions for easier variable access

Let's look at the enums on lines 2–4 first. An enum allows us to assign a string to a specific numeric value. For example, the enum `FACE` on line 4 assigns the value of 0 to the `WEST` face (i.e. $i - 1/2$) and 1 to the `EAST` face (i.e. $i + 1/2$).

What do we gain? Well, if we want to check which face is currently being processed, we can write the following:

```
1  auto currentFace = /* either 1 for i+1/2 (east) or 0 for i-1/2 (west) */;
2
3  // without enum
4  if (currentFace == 0) {
5    // perform processing on west face
6  } else if (currentFace == 1) {
7    // perform processing on east face
8  }
9
10 // with enum
11 if (currentFace == FACE::WEST) {
12   // perform processing on west face
13 } else if (currentFace == FACE::EAST) {
14   // perform processing on east face
15 }
```

We can probably all agree that the code on lines 11–15 is more readable than the code on lines 4–8. Not only that, but it is also easy to debug and less error-prone.

An even more evil version of the above code would be to use strings:

```
std::string currentFace = "west"; /* or "west" */

if (currentFace == "west") {
  // perform processing on west face
} else if (currentFace == "east") {
  // perform processing on east face
}
```

This seems reasonable, but if you have a typo in your string, then the if/else statement will not match strings and will not work properly, but no error will be detected by either the compiler or the runtime environment.

If you have a typo in an enum, the compiler will warn you, and you won't even get the code compiled. Thus, enums help us to protect ourselves from bugs before they have a chance to materialise while providing an easy-to-read string that we can use for comparisons.

There are also enum classes, which are even safer than plain enums (like we use on lines 2–4 in Listing 3). However, for the simple (short) code that we are about to write, it does not really make a difference.

Moving on, on lines 7–16, we define a struct called `caseParameters`, which holds the case-specific settings. This will help us organise all case-specific settings into a single variable.

A struct is a special type of a C++ class where the access specifier is by default `public`. This means that any types declared within a struct can be accessed from anywhere. In contrast, the default behaviour of classes is such that all access is denied (and we have to provide functions to get and set values typically).

3.2. Pre-processing

OK, in the previous section, we have only really looked at the scaffold of our code, but we have not really touched upon anything CFD-related. This will change in this section.

Referring back to the general CFD framework in Figure 2.1, this section will deal with everything that is contained within the *Pre-processing* block. We perform all these tasks once before we start the time loop, where we solve the governing equations (in this case, the Euler equations).

3.2.1. Read parameters

Listing 4 shows the code responsible for setting up the case parameters. In a real CFD solver, you would read all of these settings from a file, but this is one of the points where I decided to simply hard-code them into the C++ file so we avoid having to deal with file reading here as well. The downside is that we must compile the solver each time we change some settings.

Lines 1–2 deal with the numerical scheme (as discussed in Section 2.3.1) and the flux limiter (discussed in Section 2.3.2) we want to use. We simply set one of the enum

```
1  auto numericalScheme = SCHEME::MUSCL;
2  auto limiter = LIMITER::VANLEER;
3  caseParameters parameters;
4
5  parameters.numberOfPoints = 101;
6  parameters.gamma = 1.4;
7  parameters.domainLength = 1.0;
8  parameters.endTime = 0.2;
9  parameters.CFL = 0.1;
10
11 parameters.dx = parameters.domainLength / (parameters.numberOfPoints - 1);
12 parameters.time = 0.0;
13 parameters.timeStep = 0;
```

Listing 4: Setting of the case parameters.

values discussed in Section 3.1.4 here.

Lines 5–9 set some case-specific parameters. On line 5, we specify the number of points in our domain. More points mean more accuracy and longer compute times (though they are still reasonably short for this simple solver).

Line 6 sets γ to 1.4 (the default value for air, which is used in this case), and line 7 specifies the length of the domain.

We state on line 8 that we want the simulation to stop at 0.2 seconds and that we want to compute our timestep using a CFL number of 0.1 on line 9.

Line 11 computes the size of our finite volumes (cells), i.e. Δx as seen in Figure 2.2, and we initialise the start time (`time`) to 0.0 as well as the time step (`timeStep`) to 0. Both of these values will be incremented within the time loop.

3.2.2. Allocate memory

Once all parameters are available, especially the size of the domain (i.e. the number of points specified on line 5 in Listing 4), we can allocate the memory we need. This is shown in Listing 5.

```
1 std::vector<double> x(parameters.numberOfPoints);
2 std::vector<std::array<double, 3>> U(parameters.numberOfPoints);
3 std::vector<std::array<std::array<double, 3>, 2>> Ufaces(parameters.
     numberOfPoints);
4 std::vector<std::array<std::array<double, 3>, 2>> Ffaces(parameters.
     numberOfPoints);
```

Listing 5: Memory allocation.

Line 1 allocates memory for the x vector. This will contain the centroids of our finite volumes (cells), i.e. the black dots (and squares, i.e. the boundary centroids, too) as seen in Figure 2.2.

We use a `std::vector` data type here, which allows us to allocate memory for as many elements as we have.

On line 2, we allocate memory for the conserved variable vector $\mathbf{U}$, i.e. see Eq.(2.34). We use a `std::vector` to have an entry available for each cell in the domain. Then, we want to store, for each cell, an `std::array` containing 3 elements. These elements are the ones given in Eq.(2.34).

Both `std::vector` and `std::array` are very similar, with the only difference in how we can allocate memory for them.

For a `std::vector`, we don't have to specify how large (or small) our vector is, and we can, at runtime, specify the size. This means we can read the number of elements, for example, from a file and then have the vector sized based on this input.

A `std::array`, on the other, requires us to specify the size at compile time. This means that we have to hardcode the size in the code. For the conserved variable vector, we don't know how many cells we have, so we use a `std::vector` here. However, we know we always have 3 entries for each cell (e.g. ρ, ρu, and E), so we can use a `std::array` here.

Why don't we always use a `std::vector` if it is more flexible, you ask? Because of performance. A `std::array` is allocated on the stack while the `std::vector` goes onto the heap. The stack is faster than the heap. So, if we have a choice, we prefer a `std::array` over a `std::vector`.

If you have never heard about the stack and heap and why one is more performant than the other, I have a dedicated article for you that explains this in detail:

Resource 3.2 The complete guide to memory management in C++ for CFD

Finally, the variables `Ufaces` and `Ffaces` in Listing 5 on lines 3–4 are the conserved variable and flux vectors at the faces of the cells (i.e. at $i \pm 1/2$). They are similarly defined as the `U` variable on line 2, with an additional `std::array` to allow us to store their values for each face (i.e. the east ($i + 1/2$) and west ($i - 1/2$) faces).

Since we have three data types here (i.e. one `std::vector` and two `std::arrays`), we have a three-dimensional data structure and both `UFaces` and `Ffaces` are accessed with three indices, e.g. `Ufaces[cell][face][variable]`.

If you find Listing 5 confusing, you are not alone. C++ is a type-safe programming language, meaning that we have to always specify the types used. The syntax isn't great and leads easily to confusion (especially when you are dealing with multi-dimensional data structures like we are).

If you want to take some time to review data structures in C++ (such as the `std::vector` and `std::array` data structure) and how they really are an amazing baked-in feature of C++, you can read my article on the standard template library in C++, of which data structures are an integral part:

Resource 3.3 The power of the standard template library (STL) in C++

3.2.3. Create/read mesh

Once we have allocated memory for the grid, we can go ahead and fill our `x` vector we saw in Listing 5 on line 1. This is done in Listing 6.

```
1 for (int i = 0; i < parameters.numberOfPoints; i++) {
2   x[i] = i * parameters.dx;
3 }
```

Listing 6: Create the 1D mesh.

We loop over all points in our domain and then simply multiply the loop index `i` by the size of the finite volumes (cells), which we have labelled Δx in Figure 2.2 and `parameters.dx` in Listing 6.

3.2.4. Initialise solution

After we have populated the mesh, it is time to initialise our solution. This is the last step in the pre-processing stage and is shown in Listing 7.

```
1  double rho = 0.0;
2  double u = 0.0;
3  double p = 0.0;
4
5  for (int i = 0; i < parameters.numberOfPoints; i++) {
6    if (x[i] <= 0.5) {
7      rho = 1.0;
8      u = 0.0;
9      p = 1.0;
10   } else {
11     rho = 0.125;
12     u = 0.0;
13     p = 0.1;
14   }
15
16   U[i][0] = rho;
17   U[i][1] = rho * u;
18   U[i][2] = p / (parameters.gamma - 1.0) + 0.5 * rho * std::pow(u, 2);
19 }
```

Listing 7: Initialising the conserved variable vector $\mathbf{U}$ according to Sod's shock tube problem.

We first define our primitive variables ρ, u and p on lines 1–3 and then loop over all cells in our 1D mesh. If the x-coordinate of the current cell is less than or equal to 0.5, we are on the left side of the discontinuous profile as seen in Figure 2.10, and assign values according to lines 7–9. Otherwise, we use values for the right side, as seen on lines 11–13.

Once we have defined the values of the primitive variables, we have to compute the conserved variables, i.e. see Eq.(2.34). We do that on lines 16–18, where we write the conserved variables into our conserved variable vector `U`.

3.3. Solving

Now that we have specified all case-relevant parameters, allocated memory, assigned initial values and created our mesh, it is time to advance the initial solution in time according to our governing equations, i.e. Eq.(2.44).

The steps required to solve this equation are outlined in the next sections. All subsections within this section are executed within the `while()` loop as seen in Listing 1 on line 10.

3.3.1. Preparing solution update

In this section, we set up the solution update, i.e. we set or update variables required for the solution update. This typically involved calculating a stable time step storing the solution from the previous iteration, which is what is shown in Listing 8.

```
1  // Preparing solution update (store old solution and calculate stable
       timestep)
2  auto UOld = U;
3
4  // calculate stable time step
5  double speedMax = 0.0;
6  for (int i = 0; i < parameters.numberOfPoints; i++) {
7    // we need the primitive variables first to compute the wave speed (
       based on speed of sound and local velocity)
8    auto rho = U[i][0];
9    auto u = U[i][1] / rho;
10   auto p = (parameters.gamma - 1.0) * (U[i][2] - 0.5 * rho * std::pow(u,
       2));
11
12   // calculate wave speed for each cell
13   double speedOfSound = std::sqrt(parameters.gamma * p / rho);
14   if (speedOfSound + std::fabs(u) > speedMax)
15     speedMax = speedOfSound + std::fabs(u);
16 }
17 double dt = (parameters.CFL * parameters.dx) / speedMax;
```

Listing 8: Storing the solution from the previous iteration and calculating a stable time step.

First, we store the conserved variable vector `U` into `UOld`. This is a common naming convention, e.g. OpenFOAM uses the same terminology.

Within the loop starting on line 6, we calculate the fastest speed waves in our solution. To do that, we first need to determine what the fastest wave speeds are. The wave speeds are equivalent to the eigenvalues of the flux Jacobian. Great, but what is the flux Jacobian, you ask?

Well, without going into too much detail here (because it is not necessary to understand this step to write your own solver), we start with our conserved variable vector $\mathbf{U}$, i.e. Eq.(2.34) and the flux vector $\mathbf{F}(\mathbf{U})$, i.e. Eq.(2.35). Then, we compute the Jacobian matrix as:

$$\mathbf{A}(\mathbf{U}) = \frac{\partial \mathbf{F}(\mathbf{U})}{\mathbf{U}} = \begin{bmatrix} \frac{\partial F_1}{\partial U_1} & \frac{\partial F_1}{\partial U_2} & \frac{\partial F_1}{\partial U_3} \\ \frac{\partial F_2}{\partial U_1} & \frac{\partial F_2}{\partial U_2} & \frac{\partial F_2}{\partial U_3} \\ \frac{\partial F_3}{\partial U_1} & \frac{\partial F_3}{\partial U_2} & \frac{\partial F_3}{\partial U_3} \end{bmatrix} \tag{3.1}$$

We have to transform all fluxes $\mathbf{F}(\mathbf{U})$ into variables given by $\mathbf{U}$. For example, $F_1 = \rho u$, and $U_2 = \rho u$ (see Eq.(2.34) and Eq.(2.35)). Thus, in this simple case we have $F_1 = U_2$ and the first row of the Jacobian matrix in Eq.(3.1) is computed as:

$$\frac{\partial F_1}{\partial U_1} = 0, \quad \frac{\partial F_1}{\partial U_2} = 1, \quad \frac{\partial F_1}{\partial U_3} = 0 \tag{3.2}$$

The book *Riemann Solvers and Numerical Methods for Fluid Dynamics* by Toro, Chapter 3, is a great reference if you want to read more about the eigenstructure of the Euler equations.

Once we have determined the whole Jacobian matrix, we can compute its eigenvalues. These are given as:

$$\begin{aligned} \lambda_1 &= u - a \\ \lambda_2 &= u \\ \lambda_3 &= u + a \end{aligned} \tag{3.3}$$

Here, u is the local velocity in the x-direction, while a is the local speed of sound. Since the speed of sound a is, by definition, always positive, we could guess that the fastest wave speed must be λ_3. However, if we have a left-travelling wave, the fastest velocity is negative, and thus λ_1 is greatest.

There are many different ways how to compute the fastest wave speeds, but the simplest one is the one we are using in Listing 8, i.e. we are looking at a modified definition of λ_3 as $S_{max} = |u| + a$. This ensures the local velocity is always positive, and this should give us the fastest wave speed.

OK, so let's return to Listing 8. Lines 6–16 loop through all cells. First, we must find the local primitive variables ρ, u and p. This is done on lines 8–10, and we are implementing the procedure discussed in Section 2.2.5.

With these available, we can compute the local speed of sound from ρ and p (line 13) and then check if $|u| + a$ is greater than the currently largest wave speed stored in the variable `speedMax` (line 14). If that is the case, we overwrite it (line 15).

Finally, we can compute a stable time step based on the CFL condition. We haven't really touched upon the CFL number in this document, but if this is the first time you hear it, think of it as a non-dimensional time step.

Imagine we are releasing a particle in the flow, and we want to measure the distance it has travelled after one time step. Let's assume that the cell in which we release the particle has a size (length) of Δx.

Then, if the particle is moving exactly one cell distance, i.e. Δx, then we have a CFL number of $CFL = 1$. Equally, if it has only travelled half a cell's distance, we have a CFL number of $CFL = 0.5$.

Thus, the CFL number takes the cell's size and local flow velocity within that cell into account to determine how far *disturbances* propagate within one time step. I have used an analogy of a particle moving with the flow, but a disturbance is a more general way of looking at this.

Typically, a disturbance is some form of wave. If we know the fastest wave speed (i.e. from the eigenvalues), then we can determine the cell with the fastest wave speed and use that to determine our CFL condition as:

$$CFL = \frac{\Delta t}{\Delta x} S_{max} \tag{3.4}$$

The CFL number is a crucial concept in CFD and is typically used for stability considerations. Typically, explicit time integration schemes, such as the first-order Euler scheme discussed in Section 2.3.4.1, are stable if the CFL number is less than or equal to 1.

We saw, however, that the CFL number is set to 0.1 in Listing 4. Why is that? Well, the flow is highly non-linear, and we are making some approximations along our discretisation (for example, the piecewise constant reconstruction in Section 2.3.1.1 is a pretty severe simplification).

To give us some additional room for stability, we use a low CFL number of 0.1, but you can try to increase it and still get usable results. However, you may also start to notice some numerical instabilities. We will investigate this further in Section 4.3.3.

Solving Eq.(3.4) for the time step Δt gives us:

$$\Delta t = CFL \frac{\Delta x}{S_{max}} \tag{3.5}$$

Eq.(3.5) is implemented on line 17 in Listing 8 and provides us with a stable time step to advance our solution time, i.e. we need Δt in the discretised form of the Euler equations as seen in Eq.(2.70).

We need to compute a stable time for each iteration, as the local velocities and local speed of sound change constantly. Thus, this is the first computational we perform at the beginning of each time step.

3.3.2. Solve equations

We have a stable time step, so now it is time to advance our solution in time. This is done in the current section. Most of the computational cost will be spent on solving

the equations, and, indeed, most of the code is within this section.

This includes first reconstructing (interpolating) variables to the cell faces, then computing local fluxes, and then solving the Riemann problem at each face. We use the Riemann solver consolidated fluxes to update Eq.(2.70) in time. Let us review each step in detail below.

3.3.2.1. Reconstruct states at faces $i \pm 1/2$

We saw from Eq.(2.70) that variables are required at $i \pm 1/2$. This means we must reconstruct (interpolate) variables to the cell faces. In Section 2.3.1, we looked at the piecewise constant reconstruction (Section 2.3.1.1) and the MUSCL scheme (Section 2.3.1.2).

In this section, we will look at how to implement both of them. The first thing we need to check is which numerical scheme to use. This is done in Listing 9. Here, the variable `numericalScheme` was set on line 1 in Listing 4. We are now using our enum `SCHEME` (see Listing 3) to check which scheme to use.

```
1 if (numericalScheme == SCHEME::CONSTANT) {
2   // piecewise constant reconstruction
3 } else if (numericalScheme == SCHEME::MUSCL) {
4   // MUSCL scheme
5 }
```

Listing 9: Choosing the reconstruction method

If we are using the piecewise constant reconstruction (line 2 in Listing 9), then we execute the code provided in Listing 10. If we have set up the simulation to use the MUSCL scheme instead, we execute the code provided in Listing 11 located on line 4 in Listing 9.

```
1 for (int i = 0; i < parameters.numberOfPoints; i++) {
2   for (int variable = 0; variable < 3; ++variable) {
3     Ufaces[i][FACE::WEST][variable] = U[i][variable];
4     Ufaces[i][FACE::EAST][variable] = U[i][variable];
5   }
6 }
```

Listing 10: Piecewise constant reconstruction

The piecewise constant reconstruction is given by Eq.(2.52) in Section 2.3.1.1. We said that this simply copies the values from the cell centres to the cell faces, and we can see that this is done here.

We can also see that we have two loops, one over all cells (the first loop) and one over all variables (the second loop). We saw from the conserved variable vector `U`, i.e. Eq.(2.34) that we have 3 entries in total for a 1D case, thus our second loop is going from 0 to 2 (see line 2 in Listing 10).

I should point out that `FACE::EAST` corresponds to the face located at $i + 1/2$ and `FACE::WEST` corresponds to the face at $i - 1/2$. This just reads better than, say, `ip12`

and `im12` for $i + 1/2$ and $i - 1/2$, respectively. Thus, whenever you see `FACE::EAST` and `FACE::WEST`, you will need to replace that with $i + 1/2$ and $i - 1/2$, respectively.

```
1  // use lower-order scheme near boundaries
2  for (int variable = 0; variable < 3; ++variable) {
3    Ufaces[0][FACE::WEST][variable] = U[0][variable];
4    Ufaces[0][FACE::EAST][variable] = U[0][variable];
5
6    Ufaces[parameters.numberOfPoints - 1][FACE::WEST][variable] = U[
       parameters.numberOfPoints - 1][variable];
7    Ufaces[parameters.numberOfPoints - 1][FACE::EAST][variable] = U[
       parameters.numberOfPoints - 1][variable];
8  }
9
10 // use high-resolution MUSCL scheme on interior nodes / cells
11 for (int i = 1; i < parameters.numberOfPoints - 1; i++) {
12   for (int variable = 0; variable < 3; ++variable) {
13     auto du_i_plus_half = U[i + 1][variable] - U[i][variable];
14     auto du_i_minus_half = U[i][variable] - U[i - 1][variable];
15
16     double rL = du_i_minus_half / (du_i_plus_half + 1e-8);
17     double rR = du_i_plus_half / (du_i_minus_half + 1e-8);
18
19     double psiL = 1.0;
20     double psiR = 1.0;
21
22     // apply limiter to make scheme TVD (total variation diminishing)
23     if (limiter == LIMITER::MINMOD) {
24       psiL = std::max(0.0, std::min(1.0, rL));
25       psiR = std::max(0.0, std::min(1.0, rR));
26     } else if (limiter == LIMITER::VANLEER) {
27       psiL = (rL + std::fabs(rL)) / (1.0 + std::fabs(rL));
28       psiR = (rR + std::fabs(rR)) / (1.0 + std::fabs(rR));
29     }
30
31     Ufaces[i][FACE::WEST][variable] = U[i][variable]
32       - 0.5 * psiL * du_i_plus_half;
33     Ufaces[i][FACE::EAST][variable] = U[i][variable]
34       + 0.5 * psiR * du_i_minus_half;
35   }
36 }
```

Listing 11: MUSCL scheme implementation, including flux limiters

Listing 11 shows the implementation of the MUSCL scheme. Lines 2–8 look a lot like the piecewise constant reconstruction, and indeed, this is exactly what we are doing here. Why? Well, look back at Eq.(2.53), here, we nee variables at $i+1$ and $i-1$. If we were to loop over all cells, the cells near the boundaries would not have a neighbour cell, and thus, we could not access $i + 1$ and $i - 1$.

Thus, near the boundaries, we typically resort to a lower-order scheme if we have to. Another approach is to use so-called ghost cells, adding extra cells beyond the boundaries to reconstruct the variables near the boundary with the same scheme (e.g. the MUSCL scheme).

Ghost cells are not that easy to generalise for unstructured grids (though it can be done), but using a lower-order scheme near the boundary is, so that is why we are using it here.

Once we have obtained values for the cells near boundaries (e.g. lines 2–8 in Listing 11), we can reconstruct the variables on the internal domain using the MUSCL scheme.

Lines 13 and 14 help us write the code a bit more compactly. On line 15, we compute the modified smoothness indicator as given by Eq.(2.55), where $\epsilon = 10^{-8}$.

We set a default value of one for the flux limiter on lines 19–20, discussed in Section 2.3.2. If we are not using any flux limiter, then this value will remain, and we will not influence the MUSCL scheme reconstruction later.

However, if we do want to use a limiter (and we really do!), then we compute the flux limiter for the left (west) and right (east) faces. This is done on lines 23–29, where we implement both the minmod (Section 2.3.2.1) and the van Leer limiter (Section 2.3.2.2).

With values for the flux limiters obtained, we can compute the face reconstructed variables on lines 31–34 in Listing 11, which simply implement Eq.(2.53). Similar to the piecewise constant reconstruction, we are also looping over all variables (i.e. 3 in this 1D case) to repeat this reconstruction step for all variables listed in Eq.(2.34).

3.3.2.2. Compute fluxes at faces $i \pm 1/2$ and solve the Riemann problem

Once we have the left-sided and right-sided variables reconstructed at each face, i.e. $\phi^L_{i+1/2}$ and $\phi^R_{i+1/2}$, we can compute the fluxes at each side of a face. We will use these fluxes then in our Riemann solver, i.e. Eq.(2.62), to get a single flux at each face. This is shown in Listing 12.

On line 2, we define a temporary array for the left-sided and right-sided fluxes that we compute for each face. Then, we loop over each cell on line 3, and on line 4, we loop over each face. In this 1D case, we only have faces to the east (right) and west (left).

Remember that `FACE::WEST` and `FACE::EAST` are just integer values, where we have `FACE::WEST = 0` and `FACE::EAST = 1`. Thus, we could have also written line 4 as:

```
1 for (int face = 0; face <= 1; ++face)
```

Both would have worked the same, but line 4 in Listing 12 is more readable.

Lines 5–7 define the variable `indexOffset`. This is used to distinguish between the east and west faces. For the west (left) face, we have `indexOffset = 0` and for the east (right) face, we have `indexOffset = 1`.

Let's see why we have to do this. On lines 9–13, we compute the left-sided primitive variables for each face. For example, we get the density on line 9. We see that we access `Ufaces` here, which holds all reconstructed variables at the faces $i \pm 1/2$ (i.e. the east and west faces).

The following mapping can be used to relate our equations to the code provided in Listing 12:

```
// compute fluxes at faces
std::array<double, 3> fluxL, fluxR;
for (int i=1; i < parameters.numberOfPoints - 1; i++) {
  for (int face = FACE::WEST; face <= FACE::EAST; ++face) {
    int indexOffset = 0;
    if (face == FACE::WEST) indexOffset = 0;
    else if (face == FACE::EAST) indexOffset = 1;

    auto rhoL = Ufaces[i - 1 + indexOffset][FACE::EAST][0];
    auto uL = Ufaces[i - 1 + indexOffset][FACE::EAST][1] / rhoL;
    auto EL = Ufaces[i - 1 + indexOffset][FACE::EAST][2];
    auto pL = (parameters.gamma - 1.0) * (EL - 0.5 * rhoL * std::pow(uL,
   2));
    auto aL = std::sqrt(parameters.gamma * pL / rhoL);

    auto rhoR = Ufaces[i + indexOffset][FACE::WEST][0];
    auto uR = Ufaces[i + indexOffset][FACE::WEST][1] / rhoR;
    auto ER = Ufaces[i + indexOffset][FACE::WEST][2];
    auto pR = (parameters.gamma - 1.0) * (ER - 0.5 * rhoR * std::pow(uR,
   2));
    auto aR = std::sqrt(parameters.gamma * pR / rhoR);

    fluxL[0] = rhoL * uL;
    fluxL[1] = pL + rhoL * std::pow(uL, 2);
    fluxL[2] = uL * (EL + pL);

    fluxR[0] = rhoR * uR;
    fluxR[1] = pR + rhoR * std::pow(uR, 2);
    fluxR[2] = uR * (ER + pR);

    // Rusanov Riemann solver
    auto speedMax = std::max(std::fabs(uL) + aL, std::fabs(uR) + aR);
    for (int variable = 0; variable < 3; ++variable) {
      const auto &qL = Ufaces[i - 1 + indexOffset][FACE::EAST][variable];
      const auto &qR = Ufaces[i + indexOffset][FACE::WEST][variable];
      const auto &fL = fluxL[variable];
      const auto &fR = fluxR[variable];
      Ffaces[i][face][variable] = 0.5 * (fL + fR) - speedMax * (qR - qL);
    }
  }
}
```

Listing 12: Compute the fluxes at the faces and solve the Riemann problem using the Rusanov Riemann solver (or local Lax-Friedrichs scheme)

$$\begin{aligned}
\phi^L_{i+1/2} &= \phi[i][\text{FACE :: EAST}] \\
\phi^R_{i+1/2} &= \phi[i+1][\text{FACE :: WEST}] \\
\phi^L_{i-1/2} &= \phi[i-1][\text{FACE :: EAST}] \\
\phi^R_{i-1/2} &= \phi[i][\text{FACE :: WEST}]
\end{aligned} \tag{3.6}$$

Here, the first index of `Ufaces` is used to loop over all cells. The index we use for that is `i - 1 + indexOffset`. If `indexOffset = 0` (i.e. we are currently working on the west face), then we take values at `i - 1`.

Since line 9 in Listing 12 looks at the left-sided reconstructed density at the west face (`indexOffset = 0`), we can see from Eq.(3.6) that line 9 is for $\rho^L_{i-1/2}$. The corresponding code is `Ufaces[i - 1][FACE::EAST][0]`.

We can confirm that this is correct by looking at Figure 2.3, where $\phi^L_{i-1/2}$ would be stored at the cell with index `i - 1`, specifically, at its east face.

Once we have obtained the primitive variables at both the east and west faces for each cell, we can compute the flux vector as given by Eq.(2.35). This is done on lines 21–23 and 25–27 in Listing 12 for the left and right side of the face, respectively.

Now that we have both the conserved variables and the fluxes available at the face, both with their left and right state, we can go ahead and solve the Riemann problem for it. This is what is done on lines 30–37.

First, we have to compute a suitable wave speed. We are using the Rusanov Riemann solver (or local Lax-Friedrichs scheme), where we have established the simple yet effective wave speed estimate in Eq.(2.60). This equation is implemented on line 30 in Listing 12.

Lines 31–37 loop over all three entries in our conserved variable vector $\mathbf{U}$ and the flux vector $\mathbf{F}(\mathbf{U})$. To make the equation shorter to write, we introduce the temporary on lines 32–35. This isn't required but just makes line 36 more compact to write.

Line 36 is the Rusanov Riemann solver, corresponding to Eq.(2.62). Thus, we obtain a single flux at the face from the left and right-sided states and fluxes. We store this flux in the new flux vector `Ffaces`, which will be used in the next part when integrating the Euler equation in time.

3.3.2.3. Integrate solution in time

Ok, at this point, we have everything we need. Let's review Eq.(2.70), which is reproduced below for convenience again:

$$\mathbf{U}^{n+1}_i = \mathbf{U}^n_i - \frac{\Delta t}{\Delta x}\left(\tilde{\mathbf{F}}(\mathbf{U})^n_{i+\frac{1}{2}} - \tilde{\mathbf{F}}(\mathbf{U})^n_{i-\frac{1}{2}}\right)$$

We know $\mathbf{U}^n_i$, this is just `UOld[i]` as seen in Section 3.3.1. Within the same section, we also computed a stable time step denoted by Δt above. When we set up the case

parameter in Section 3.2.1, we defined Δx, and in the previous section, we computed the fluxes at the faces, i.e. $\tilde{\mathbf{F}}(\mathbf{U})^n_{i\pm1/2}$ through the Rusanov Riemann solver.

All that is left to do is to implement Eq.(2.70) into code, which is done in Listing 13. We loop over all points (except the boundary points, these will be discussed in the next section), and for each cell with index i, we loop over all 3 quantities in our conserved variable vector $\mathbf{U}$, using index j.

```
1 for (int i=1; i < parameters.numberOfPoints - 1; i++)
2   for (int j=0; j<3; j++) {
3     const auto &dF = Ffaces[i][FACE::EAST][j] - Ffaces[i][FACE::WEST][j];
4     U[i][j] = UOld[i][j] - (dt / parameters.dx) * dF;
5   }
```

Listing 13: Integrate solution in time using a first-order Euler time integration method.

3.3.3. Update boundary conditions

In the previous section, I mentioned that we are not integrating the solution at the first and last points of the domain. Returning to Figure 2.2, we introduce these points as boundary points, and thus, we have to apply our boundary conditions here.

In this simple case, the flow is driven by the initial discontinuity at the centre of the domain, and we stop the simulation after 0.2 seconds. The generated waves do not have sufficient time to reach the boundaries within that time, so the boundary conditions we are imposing do not matter (in fact, we could ignore this section).

However, generally speaking, boundary conditions are absolutely crucial, and our simulation will always converge towards the boundary conditions we set. If our boundary conditions are wrong or inaccurate, our simulation will be, too. Thus, let's be good citizens and provide our solver with some boundary conditions.

In Section 2.4, we looked at different types of boundary conditions we can impose. In this particular case, we want to assume an inviscid wall on either side so that the generated waves can reflect at boundaries. Go ahead and change the simulation end time in Listing 4, say, to 0.5 seconds. You will see waves reflecting off from the right boundary and new structures forming.

The boundary conditions for an inviscid wall, as discussed in Section 2.4.4.1, are implemented in Listing 14. We use a Dirichlet-type boundary condition for the velocity (it has to be 0 so that no flow is going through the wall) and a Neumann-type boundary condition for the pressure, density, and energy.

Even though we set the density for the left and right boundary on lines 1 and 4, we only do so for completeness, as we have to impose ρu in the conserved variable vector, i.e. see Eq.(2.34). The product of ρu will be zero, and thus, the second entry in the vector will be zero.

We see the imposition of boundary conditions on lines 7–9 for the left boundary and lines 11–13 for the right boundary. We already established that we use Dirichlet boundary condition for the second entry in $\mathbf{U}$. However, the first and third entries use a Neumann condition.

```
auto rhoL = 1.0;
auto uL = 0.0;

auto rhoR = 0.125;
auto uR = 0.0;

U[0][0] = U[1][0];
U[0][1] = rhoL * uL;
U[0][2] = U[1][2];

U[parameters.numberOfPoints - 1][0] = U[parameters.numberOfPoints - 2][0];
U[parameters.numberOfPoints - 1][1] = rhoR * uR;
U[parameters.numberOfPoints - 1][2] = U[parameters.numberOfPoints - 2][2];
```

Listing 14: Update boundary conditions

We can see that using a Neumann condition with a zero-gradient, e.g. $\partial\phi/\partial n = 0$, results in copying whatever value is in the cell next to the boundary into the boundary cell. This is in agreement with Eq.(2.80) and Eq.(2.83).

3.3.4. Custom post-processing

Now that we have an updated solution for the current time step, we want to output it to look at the animated solution. There are many different ways to produce a solution file. However, we opt here for a simple `*.csv` file that we write out at each time step.

A `*.csv` file simply stores a list of *comma separated values* (hence the name csv). In our case, we want to write out ρ, u and p at each point in the domain. A simple `*.csv` file with 11 grid points containing the initial solution may look like this:

```
x,rho,u,p
0.0, 1.0, 0.0, 1.0
0.1, 1.0, 0.0, 1.0
0.2, 1.0, 0.0, 1.0
0.3, 1.0, 0.0, 1.0
0.4, 1.0, 0.0, 1.0
0.5, 1.0, 0.0, 1.0
0.6, 0.125, 0.0, 0.1
0.7, 0.125, 0.0, 0.1
0.8, 0.125, 0.0, 0.1
0.9, 0.125, 0.0, 0.1
1.0, 0.125, 0.0, 0.1
```

That's all. The advantage of a `*.csv` file is that we can open it with pretty much any processing tool we want. For example, we can easily read it into Excel and plot the columns, read it into Python using the pandas package, or even read it with the post-processor Paraview and animate the solution in time.

Whatever route we choose, for 1D data, a `*.csv` file is likely the easiest solution to implement. The algorithm to do that is shown in Listing 15.

We are using the `std::ofstream` class here, which allows us to write out data to a file. Since we want to write out the files for each time step, we need to make the current time step part of the filename.

```
1  // Output solution to csv file for plotting
2  std::ofstream outputFile;
3
4  // convert time step and points into 6 digits string with leading zeros
5  std::ostringstream timeStepTemp, pointsTemp;
6
7  timeStepTemp << std::setfill('0') << std::setw(6);
8  timeStepTemp << parameters.timeStep;
9  auto timeStep = timeStepTemp.str();
10
11 pointsTemp << std::setfill('0') << std::setw(6);
12 pointsTemp << parameters.numberOfPoints;
13 auto points = pointsTemp.str();
14
15 outputFile.open("solution_" + points + "_" + timeStep + ".csv");
16 outputFile << "x,rho,u,p" << std::endl;
17 for (int i = 0; i < parameters.numberOfPoints; i++) {
18   auto rho = U[i][0];
19   auto u = U[i][1] / rho;
20   auto p = (parameters.gamma - 1.0) * (U[i][2] - 0.5 * rho * std::pow(u,
       2));
21   outputFile << x[i] << "," << rho << "," << u << "," << p << std::endl;
22 }
23 outputFile.close();
```

Listing 15: Output primitiv variables ρ, u and p to a CSV file for each timestep.

Thus, we first convert the current time step into a string using the string stream class on lines 7–9. The string stream class allows us to perform some useful modifications on our strings, for example, fill a string with leading zeros, which is what we want here. i.e. lines 7–9 take the current timestep and convert it into a string, making sure that the string is always 6 characters long (by using leading zeros if necessary).

The `std::setfill` function declares the leading characters to use when filling the string (here, `0`), while the `std::setw(6)` states that the width of the string should be 6 characters.

Lines 11–13 repeat this step for the number of points, and this allows us also to encode this information in the filename so that we can have different solutions for different grid sizes stored on our disk for comparisons.

Line 15 creates a new file, where the file name is dynamically generated using the number of points and time step as variables. We then write the required header on line 16 and then loop over all cells on lines 17–22.

Within the loop, we first need to calculate the primitive variables ρ, u and p from the conserved variable vector $\mathbf{U}$ (lines 18–20), and then we proceed on line 21 to write out these variables to the file. Once the loop is completed, we close the file on line 23.

3.3.5. Check if the simulation has ended

Finally, we need to check if the simulation has finished. In our case, we specify that after a certain amount of time, we want to stop the simulation. And, we saw in Listing 1 that within the `while()` loop on lines 9–11, we check if we have reached the end time of the simulation.

So, in a sense, the check for the end of the simulation is really done here. However, we also need to have a mechanism to increase the time and time step variable. This is done in Listing 16.

```
1 parameters.time += dt;
2 parameters.timeStep++;
```

Listing 16: Check if the solution has ended by incrementing the time and time step.

In this case, we simply increase the time step and add the current Δt value to the total time (line 1). For transient simulations, this is a common stopping criterion.

However, steady-state codes, or even unsteady codes, which march a solution towards a steady-state-like solution, we may also want to implement some form of residual checking in this part. For our solver, though, we are happy with just checking for the end time.

If you want to read up more on the difficulties of checking convergence and how to do it properly, along with a framework to ensure you have optimal stopping criteria available, you might want to check my article on checking convergence in the article linked below:

Resource 3.4 How to determine the best stopping criterion for CFD simulations

Furthermore, we should output some information to the console about the progress of the simulation. We are probably interested in how much (simulation) time has elapsed and how many time steps we have used thus far. This information is provided by Listing 17.

```
1 // output current time step information to screen
2 std::cout << "Current time: " << std::scientific << std::setw(10) << std::
     setprecision(3) << parameters.time;
3 std::cout << ", End time: " << std::scientific << std::setw(10) << std::
     setprecision(3) << parameters.endTime;
4 std::cout << ", Current time step: " << std::fixed << std::setw(7) <<
     parameters.timeStep;
5 std::cout << "\r";
```

Listing 17: Output the current simulation time and time step to the console.

We are using here some formatting features of C++, which are frustratingly complicated, especially if you are used to things such as *f-strings* in Python. Anyhow, that's the price we have to pay to use C++.

We are printing three quantities to screen. First, we print the current (simulated) time, then the time at which we want to stop the simulation, followed by the current timestep.

In C++, we can represent numbers as either floating point notation, e.g. 0.00123, or exponential notation, e.g. $1.23e-3$. The `std::fixed` keyword tells C++ to use floating point notation, while `std::scientific` uses exponential notation.

The current (simulated) time and timestep will be printed using exponential notation. The time step, which is just an integer value, does not need exponential notation, i.e. it would be strange to write time step (iteration) 123 as $1.23e2$.

Then, we can influence how many digits we want to use after the comma. This is what the command `std::setprecision(3)` is doing. Here, 3 means we want to use 3 digits after the comma. The command `std::setw(10)` tells C++ that we want to use a total of 10 characters to print a given number. If we do not need 10 characters to print the entire number, it will be padded with leading spaces.

Finally, we use a little trick on line 5. Here, the command `std::cout << "\r"` instructs C++ to avoid a line break and, instead, go back to the beginning of the current line. The next time we print to the console, we will simply overwrite the previous line.

Since we have specified the width of each number using the `std::setw()` command, we can ensure that each line that is being printed will have the same length. The only thing that is changing is the numerical values themselves. Thus, we only see a single line that is constantly being updated, which will keep our terminal clean.

3.4. Post-processing

Finally, we have the post-processing section. Here, we typically write out the solution to file once the simulation has finished and de-allocate any memory we need to take care of. This is discussed in the following sections.

3.4.1. Write out solution

We could implement some additional solution output writing, for example, in a native Paraview format. However, since the `*.csv` output files provide us with all the information we need, there is no need to complicate things further here and write out additional files. Thus, there is nothing to be done in this section for our solver.

3.4.2. Deallocate memory

Memory management is crucial in C++ and can make or break your code. C++ is a low-level programming language, meaning we are responsible for all memory allocations and de-allocations.

However, if we use C++ as the designers have intended us to do, then we can actually offload quite a lot of memory management.

Whenever we use a built-in data type of C++, i.e. one from the standard template library (STL) such as a `std::vector`, then all memory management is done for us.

We only ever need to worry about memory when we are dealing with pointers; how-

ever, that has become an issue of the past since C++ introduced smart pointers. I have, again, written a dedicated article on smart pointers and why they are truly amazing, and you can find the article linked below:

Resource 3.5 Reduce memory bugs with smart pointers in C++

Thus, since we are good programmers and have exclusively used STL data types (so-called containers), we don't have to worry about memory de-allocation, and we don't have to do anything in this section either.

3.5. Summary

This brings us to the end of the code implementation. Hopefully, after going through the theory in Chapter 2, this Chapter did make sense, and you were able to follow the discussion.

Remember that you can always look at the full source code in Appendix A or by consulting the source code that you should have received with this document.

In the next section, we will look at how to compile the code, how to run it, and then how to process and inspect the results.

4

Running simulations and visualising results

Now that we have an idea of the theory as discussed in Chapter 2, as well as how to put that theory into code as discussed in Chapter 3, it is time to compile and run our solver. In this chapter, we will look at how to do just that.

First, we will look through several ways to compile the code. Since we are using C++, which is a compiled language, we need to translate the source code into machine-readable code (typically called an object file). This is what the compiler will do for us.

Then, the linker will take all object files (in our case, just one, as we only have a single source file) and link them into one executable file. On Windows, that would be a file ending in `*.exe`, while on Linux or macOS, it is customary to use either no extension or `*.out`.

Even though we only need to compile a single source file, this can sometimes be a challenge already, depending on the operating system you are using and the development set-up you have (or a lack thereof). Thus, I present a few solutions with the hope that at least one of them will work for you.

If you have not already done so, I suggest you have at least a glimpse at the recommended way to set up your system so that you can compile C++ code. The link was provided in Chapter 1, which is repeated below for convenience:

Resource 4.1 Setting up a programming environment to develop CFD codes

4.1. Compiling and running our CFD solver

Right, let us jump straight into the process and compile our solver. I will show you two ways of doing it. First, we will look at the manual way, where we must do all the compilation ourselves. Afterwards, I'll show you a more streamlined approach which you can use if you have CMake installed, which I would generally recommend for reasons discussed later.

4.1.1. Compiling the code manually

Compiling the code ourselves requires a bit of typing, and what exactly we have to type into our console depends on the operating system. In the following, we will look at instructions for Windows and Linux / macOS.

4.1.1.1. Windows

While there are a lot of different C++ compilers available that you can use for Windows, I am only interested in compiling code natively on Windows. That means using Microsoft's MSVC compiler, which will be installed when we download Visual Studio and install the required C++ components.

There are botched ways of bringing back a UNIX (Linux or macOS-like) development environment on Windows (e.g. MinGW or WSL), but these, to me, qualify as a UNIX solution and are not helpful if you are trying to understand how to compile code natively on Windows.

Thus, if you have set up your system correctly, you should be able to open a PowerShell (either the *Developer PowerShell for VS 20XX* or a customised one as discussed in the above-linked article on setting up a programming environment), and then type `cl` into it and get an output similar to the one below:

```
Microsoft (R) C/C++ Optimizing Compiler Version 19.42.34435 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

usage: cl [ option... ] filename... [ /link linkoption... ]
```

This means we have Microsoft's MSVC compiler installed, and it is ready to use. Of course, we could just throw the compiler at our source file and let it create an executable, but that would be the same as going to a fancy restaurant and eating your steak using your hands. Let's be civilised and use some common conventions.

First, we want to keep our source code separate from anything the compiler generates, i.e. any object or executable files. We also want to store any output our solver generates in a separate folder. It is common to create a `build/` folder containing all the generated files.

Using the PowerShell, the following commands can be used for this:

```
Remove-Item .\build -Force -Recurse
New-Item -Name "build" -ItemType "directory"
```

First, we remove the `build/` folder, if already available, and then re-created it. This will eliminate any previously generated files, and we will ensure that we always get a clean build.

Next, we compile the code into an object file. We use the `cl` compile here, using the following command:

```
cl.exe /nologo /EHsc /std:c++17 /I. /c /O2 .\euler.cpp /Fo".\build\euler.
    obj"
```

Whenever we use the `cl` compiler, it reminds us that Microsoft developed it. To suppress that behaviour, we can use the /logo flag. The `/EHsc` flag will turn on proper exception handling. Without it, you will get a lot of warnings printed to the console,

even if your code is compiling just fine. This is really annoying when tracking down bugs and error messages being hidden in a lot of warnings.

We use the C++ 2017 standard, which is necessary for certain string manipulation features. We specify this with the /std:c++17 flag. Additionally, we instruct the compiler to include the current directory using the /I. flag, to compile the file with the /c flag, and to optimise the code with the /O2 flag, which can significantly enhance the performance of our code.

We tell the compiler that we want to compile the file called `euler.cpp` and output the generated object file into the `build/` folder using the `/Fo` flag. The generated object file will be called `euler.obj`.

The next step involves calling the linker, which is done with the following command:

```
cl.exe /nologo .\build\euler.obj /Fe".\build\euler.exe"
```

This command is pretty similar to the previous one, where the input now changes to `.\build\euler.obj`, i.e. we provide the linker with the object file, and then we tell it to generate an executable within the `build/` folder using the `Fe` flag. We are calling the executable `euler.exe`.

I should say that we call the `cl` compiler in the command above, though it will pass all the input to the actual linker for us so that we don't have to deal with the linker ourselves. This is for a good reason, as the linker can require some additional flags that the compiler can deduce.

With the above commands executed, we should have the `euler.exe` file within our `build/` folder.

4.1.1.2. Linux and macOS

Linux and macOS behave pretty much the same, and thus, we can discuss both of them in the same section. We will follow the same steps as outlined above in the Windows section; that is, we will first create a `build/` folder to house all of the compiler output and then proceed to generate the solver.

First, we remove any `build/` folder that may exist. Afterwards, we recreate it to ensure no previous compiler-generated files are present. This can be achieved with the following two commands:

```
rm -rf build
mkdir build
```

Then, we go ahead and compile our code. In this case, I will assume you are using the GNU compiler suite, i.e. `g++`, to compile C++ code. If you are on macOS (or on Linux and just a bit eccentric), then you might have the LLVM compilers installed, i.e. `clang++` for compiling C++ code. If that is the case, just replace `g++` with `clang++`.

To compile the code, turn on the C++ 2017 standard, including the current directory and optimising the code, we can use the following command:

```
g++ -std=c++17 -I. -c -O2 euler.cpp -o build/euler.o
```

This will generate an object file within the `build/` directory called `euler.o`. The linker can now be called on the object file to generate the executable using the following syntax:

```
g++ build/euler.o -o build/euler
```

We generate the executable `euler` within the `build/` folder. Note that we do not give it any extension as on Windows. This is customary on UNIX platforms and the convention we will adopt here. And, again, if you are using LLVM, replace `g++` with `clang++`.

4.1.2. Compiling the code using CMake

Finally, let's have a look at compiling the code using CMake. This assumes that you have CMake installed on your PC. CMake is a build system that compiles the code for you by selecting the appropriate compiler for your operating system.

Thus, it does not matter if you are on Windows, Linux, or macOS. CMake will always generate a correct build for your platform, and thus, it is the recommended way of compiling code.

CMake is a beast. It not only helps you compile your code (and link it against dependencies), but it offers so much more. So much so that I have a full series just on how to use CMake and all of its basic and advanced features, as well as how you would use it for developing an actual CFD code. This can be found in the series linked below:

Resource 4.2 Automating CFD solver and library compilation using CMake

The idea of CMake is simple. First, generate a project file that contains all build instructions. From it and some potentially user-specified inputs, build instructions for your operating system and compiler will be generated. CMake then executes these build instructions to create the executable.

The project file that contains all build instructions is named `CMakeLists.txt`, and CMake will always look for this file when it is executed. It is always in the root folder of your project, and in our case, it has the following content:

```
1 cmake_minimum_required(VERSION 3.10)
2 project(euler)
3
4 # ensure that the C++ 2017 standard is used
5 set(CMAKE_CXX_STANDARD 17)
6 set(CMAKE_CXX_STANDARD_REQUIRED ON)
7
8 add_executable(euler euler.cpp)
```

First, we have to define a minimum version of CMake to use. If we try to run it with a lower version of CMake, it will stop. Version 3.10 is pretty old, and your version of CMake should be much newer than this.

Then, we give the project a unique identifier, here, `euler`. Lines 5–6 ensure that the C++ 2017 standard is enabled, and then, on line 8, we define the executable called `euler`, which requires `euler.cpp` to be generated.

This is all. From this file, we can create the executable for Windows, Linux, and macOS. Let's do that.

First, we generate a `build/` folder again. This is essential and must be done with CMake, but you can give the folder any name you want. Then, we have to go inside this newly created `build/` folder and invoke CMake once using the following command:

```
cmake .. -DCMAKE_BUILD_TYPE=Release
```

This will run CMake and look in the parent folder (`..`) for a file called `CMakeLists.txt` (which we have just looked at above). CMake will then check what your operating system is, which compiler should be used, and so on. All of that information gets stored in a file called `CMakeCache.txt`.

The additional flag `-DCMAKE_BUILD_TYPE=Release` informs CMake that we want to turn on compiler optimisations. This is somewhat equivalent to using the `/O2` on Windows or the `-O2` flag on Linux / macOS, though CMake will set a few more compiler flags.

We can now go ahead and build our project using the following command:

```
cmake --build . --config Release
```

This will generate the executable either within the current `build/` folder or within a sub-folder called `Release/`. This will depend on your platform, and thus, you will need to check, though when you run the above command, it will tell you where the executable was placed in the console (that path will be printed).

If you want to clean up the current folder from all of the generated `*.csv` files, you can achieve that using the `rm *.csv` command. This command works on Windows, Linux, and macOS.

4.1.3. Running the code

Now that we have a compiled code available, let's run it to make sure it is working. Depending on which route you have taken, your executable will be located in one of two locations. It is going to be in either of the two:

- `build/euler.exe` (or `build/euler` on UNIX)
- `build/Release/euler.exe` (or `build/Release/euler` on UNIX)

Change into the `build/` directory and run the executable. This can be achieved using one of the following commands:

- `./euler.exe` (or `./euler` on UNIX)
- `./Release/euler.exe` (or `./Release/euler` on UNIX)

We want to stay in the `build/` folder so that all `*.csv` files will be generated in this folder. This will help us with the post-processing, which we will look at in the next section, where one of the scripts assumes that all solution files are stored in the `build/` folder.

If you managed to get the code compiled and executed, you should see the following message after running the code:

```
Current time:  2.000e-01, End time:  2.000e-01, Current time step:     385
Simulation finished
```

If this is the case, the solver executes correctly, and you are ready to look at the results. To make your life easier, I have included two shell scripts that will compile and run the code for you using the steps outlined above. These are called `compileAndRun.ps1` for Windows and `compileAndRun.sh` for Linux and macOS. You can run them inside your terminal using the following commands:

On Windows: `.\compileAndRun.ps1`

On Linux and macOS: `./compileAndRun.sh`

4.2. Post-processing results with Python, Jupyter notebook, and plotly

In this section, we will look at how we can post-process the results using Python. If you are already familiar with Python, this is probably the simplest way to look at the results. If not, you can open the `*.csv` files with any other program you like (e.g. Excel, Matlab), and then process the results. If you do, simply take the last created `*.csv` file and use that to post-process your results.

Remember that the structure of the `*.csv` file is such that all values are comma-separated, and the first line represents the header (column names), followed by the columns with the values for each column.

If you want to follow along in this Section, then you will need Python installed on your system, along with its package manager pip. If you have that, the first thing you will need to do is install all required dependencies using pip. This can be achieved with the following command:

```
pip install -r requirements.txt
```

This will install all dependencies listed in the `requirements.txt` file. This step will install all required packages system-wide, which isn't the best choice, and sometimes, you aren't even allowed to do so. If that is the case, you want to create a virtual environment first, which allows you to install all required packages locally. You can do that using the following command:

```
python -m venv euler
```

Then, depending on your operating system, you can activate the virtual environment using one of the following commands. On Windows, you have to type:

```
.\euler\Scripts\Activate.ps1
```

And, on UNIX (Linux or macOS):

```
source euler/bin/activate
```

This will load the virtual environment, and any packages you download and install will be local to this environment. Once the environment has been activated, you can install all required packages using the same command as above, i.e.:

`pip install -r requirements.txt`.

Once all packages have been installed (either system-wide or inside your virtual environment), we will be ready to post-process the results. There is just one assumption here: The Python code expects all `*.csv` files to be located within the `build/` directory. If not, then the code won't find the solution files generated by the CFD solver.

The post-processing code is written as a Jupyter Notebook. If you have never used that before, then think of it as a dynamic code execution framework. You can run either the entire code or just parts of it and then inspect the results as you step through each block of code.

To launch the Jupyter notebook, run the following command:

```
jupyter notebook eulerPlot.ipynb
```

This will launch the code within your default browser. Here, you can go to the `Run` menu and select `Run all Cells`. This will go through all the code and execute them. This will generate two plots.

The first plot is shown in the middle of the notebook and shows the solution at the end of the simulation. The second plot, shown at the bottom of the notebook, is an animated version of the entire solution. It has a play and a pause button, allowing you to animate the solution for each time step.

If you can see the results, then you are ready to move on to the next section and start investigating the results.

4.3. Results and preliminary analysis

In this section, I want to highlight some of the characteristics of the shock tube problem and, by extension, the behaviour of the Euler equations. We will see that capturing shock waves is rather difficult, and even a simple inviscid 1D solver already poses great challenges for our numerical schemes to overcome.

But before we jump in, let us have a look at the solution that we would expect. This is shown for the density, velocity, and pressure profile in Figure 4.1.

Think of the shock tube problem as a long cylinder filled with air, where we keep two separate chambers at different pressures and densities. This is achieved using a thin sheet of metal in the middle of the cylinder, which separates these states.

Then, at $t = 0s$, we remove the thin sheet of metal (or pierce it) so that both regions are brought into contact with one another, where we have discontinuities between the two separate regions.

As a result, we will form 3 different types of waves which are:

- A rarefaction wave travelling from the initial discontinuity to the left

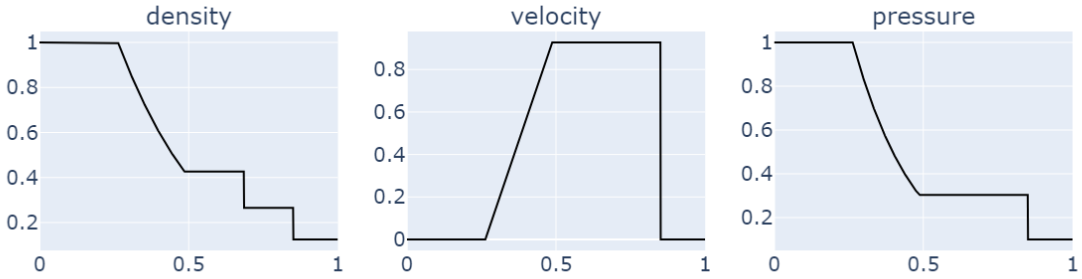


Figure 4.1: Expected solution for the density, velocity, and pressure profile for the shock tube problem at $t = 0.2s$.

- A shock wave travelling from the initial discontinuity to the right
- A contact discontinuity which is located between the rarefaction and shock wave

We can see that at a time of $t = 0.2s$, i.e. the solution shown in Figure 4.1, we have zero velocities at the boundaries (the rarefaction and shock waves have still not reached the boundaries). We see that the contact discontinuity is present in the density profile but absent in the pressure profile (which otherwise looks fairly similar).

Thus, we study the expansion of these three separate and characteristic waves in the shock tube problem, and numerical results are presented in the subsequent sections.

4.3.1. Influence of the numerical schemes

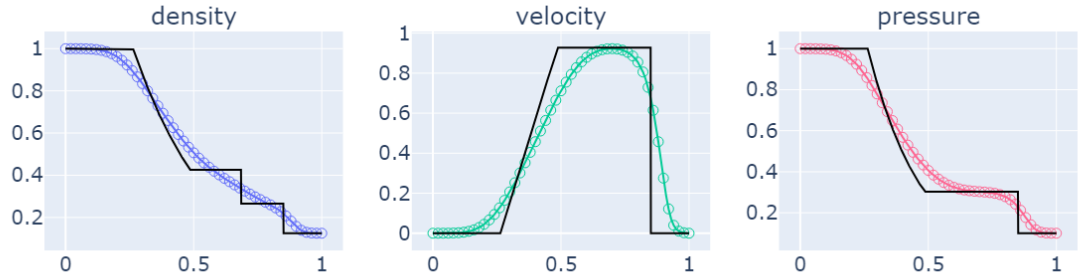
Let's start with the numerical schemes first. We can choose between the piecewise constant reconstruction and the MUSCL scheme. Using the MUSCL scheme, we also have to pick a limiter. We can either pick the minmod, or van Leer limiter, or, switch it off if we wish.

The results for both the piecewise constant reconstruction as well as the MUSCL reconstruction using both the minmod and van Leer flux limiter are shown in Figure 4.2.

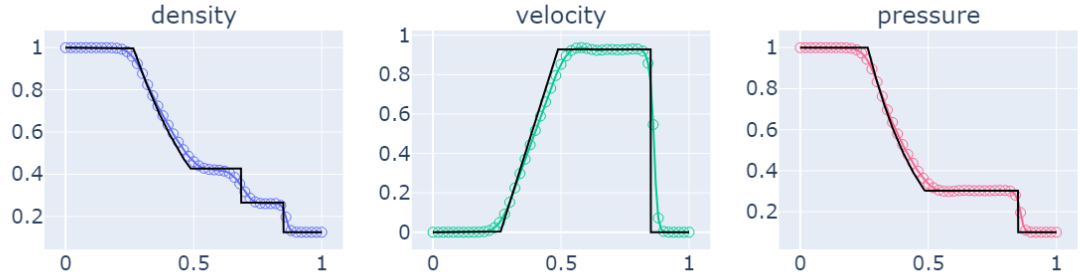
The piecewise constant reconstruction struggles to capture discontinuities, which becomes especially noticeable when comparing it against the MUSCL scheme using either of the two flux limiters. However, the resolution of the discontinuities is also not that great with the MUSCL scheme, and this is particularly exposed by the contact discontinuity seen in the density profile.

This is not entirely the fault of the MUSCL scheme alone. The Riemann solver we use here can also positively (or negatively) affect our solution. We use the simple Rusanov Riemann solver here, which isn't great at capturing these discontinuities with high accuracy.

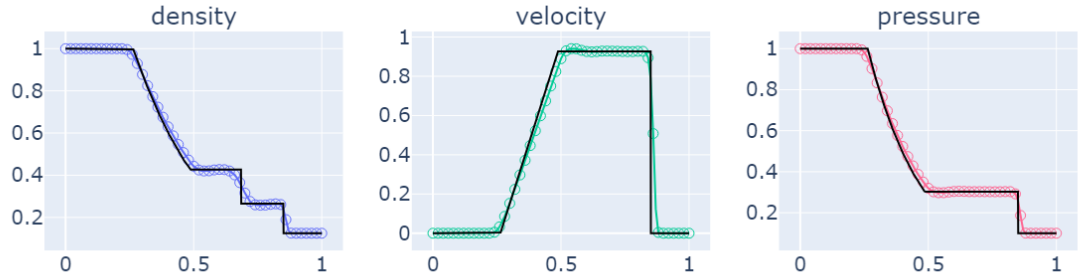
There is another Riemann solver proposed by Harten, Lax, and van Leer (aptly named the HLL Riemann solver, based on their names), which is a bit more sophisticated compared to the Rusanov Riemann solver, although it also struggles to capture this discontinuity. An extension proposed by Toro and co-workers recovers this contact



(a) Piecewise constant reconstruction



(b) MUSCL scheme using the minmod flux limiter



(c) MUSCL scheme using the van Leer flux limiter

Figure 4.2: Solution for the shock tube problem using the piecewise constant reconstruction and the MUSCL scheme with 101 grid points at $t = 0.2s$.

discontinuity, and their Riemann solver is known as the HLLC Riemann solver.

There is another point worth noting, which isn't clear from Figure 4.2. We are only looking at results that converged and not the ones that diverged. If you try to run the MUSCL scheme without a limiter (which we can set up in our solver), you won't get results (in fact, the simulation will diverge).

This is because the MUSCL scheme is second order, which is a problem. Godunov (do you remember him from Section 2.3.1.1?) worked on this problem quite a bit and

postulated the following, which is nowadays known as Godunov's theorem:

Linear numerical schemes for solving partial differential equations (PDE's), having the property of not generating new extrema (monotone scheme), can be at most first-order accurate.

To translate that into plain English, it means that if you have any numerical scheme where the order is greater than one, we will introduce oscillations (new extrema) into our solutions. These oscillations will become stronger in magnitude until they diverge the simulation, and these oscillations are entirely driven by discontinuities in our solution.

Thus, van Leer introduced the flux limiters into the MUSCL scheme, which made them stable, even at higher orders, which gave us high-resolution numerical schemes. Therefore, only results for the minmod and van Leer flux limiter are provided, as only these converged.

From Figure 4.2, we can also see that 101 points for the domain are likely not sufficient to resolve the flow accurately, at least not with the current numerical schemes and Riemann solver setup. Thus, let us explore what happens when we use more points for the solution.

4.3.2. Influence of the grid size

If we want to enhance the accuracy of our solution, one option is to increase the number of cells in our domain. As we add more and more cells, the cell sizes decrease, and thus, we have more cells near discontinuities to resolve them better.

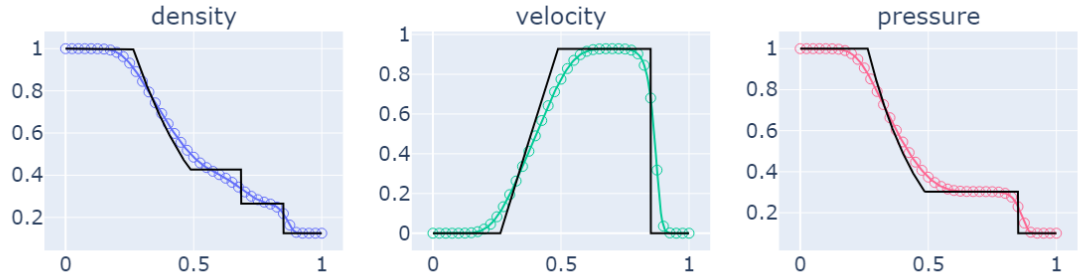
In Figure 4.3, we can see results for the piecewise constant reconstruction using 201, 401, 801, and 1601 grid points. We see that despite increasing the number of cells, the accuracy is only modestly improving.

While we do have an overall good resolution at 1601 cells, the contact discontinuity is still not resolved well. We could, of course, continue to increase the number of cells, but this would just unnecessarily increase the computational cost.

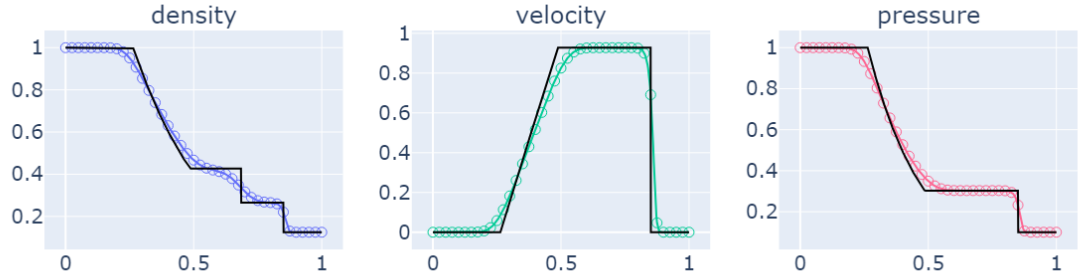
Instead, let's look at the results for the MUSCL scheme using the van Leer flux limiter. These are shown in Figure 4.4. Comparing these results with those obtained with the piecewise constant reconstruction, we see a drastic improvement in resolution.

At 201 grid points using the MUSCL scheme, the results look qualitatively as good as the piecewise constant reconstruction at 1601 grid points. However, the contact discontinuity has still not been resolved well enough. But, as we increase the number of cells, the contact discontinuity will eventually be resolved much better. Looking at the 801 and 1601 grid points, we see that the contact discontinuity is finally resolved well with a sharp gradient.

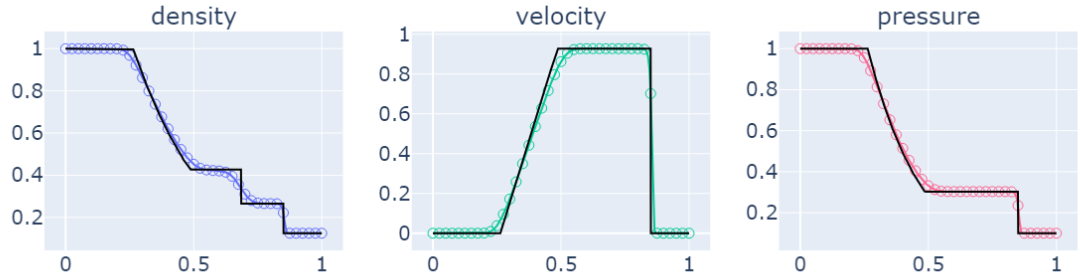
Thus, we can conclude the following: The piecewise constant reconstruction is a first-order method, while we are using a second-order MUSCL scheme here. The higher the order, the better the resolution on the same grid. Or, we can use a coarser grid using a higher-order scheme compared to a lower-order scheme for the same level of accuracy and thus save computational costs.



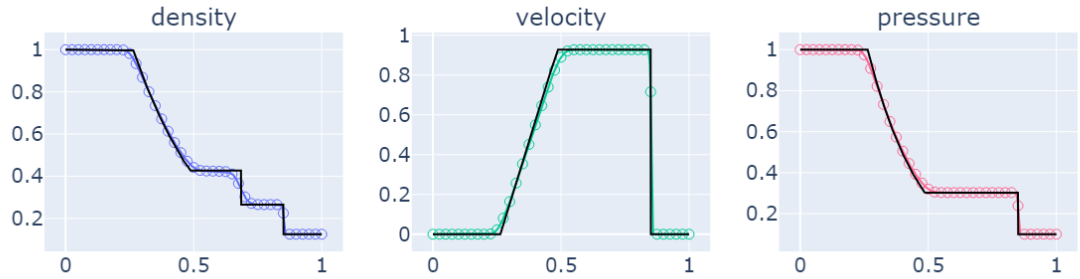
(a) Piecewise constant reconstruction using 201 grid points



(b) Piecewise constant reconstruction using 401 grid points

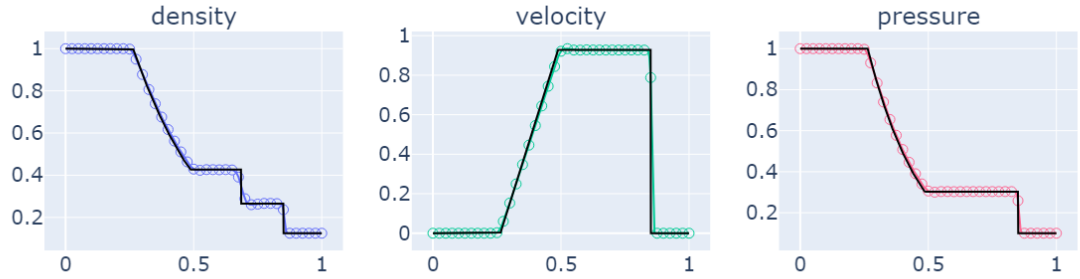


(c) Piecewise constant reconstruction using 801 grid points

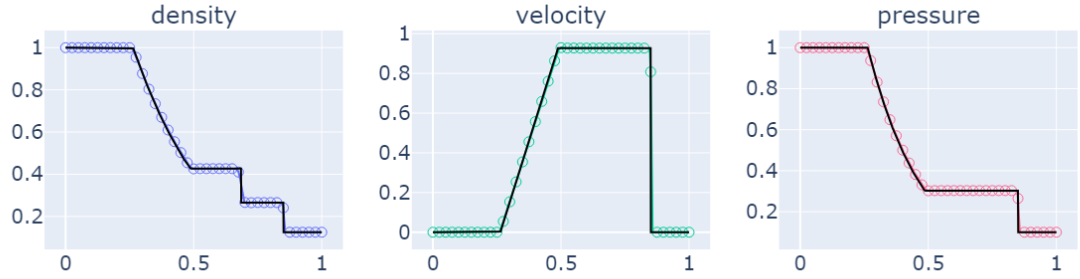


(d) Piecewise constant reconstruction using 1601 grid points

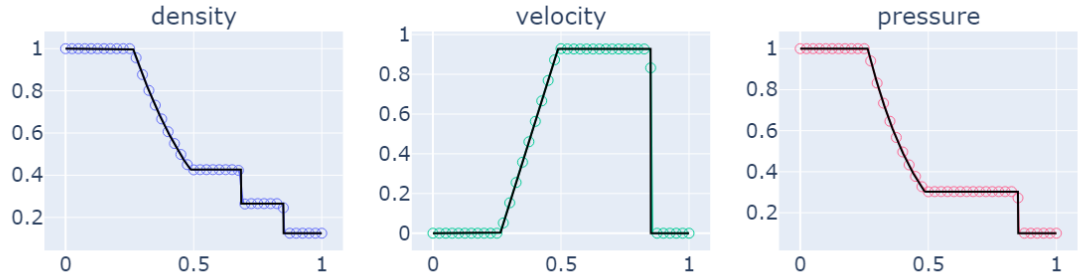
Figure 4.3: Solution for the shock tube problem using the piecewise constant reconstruction with 201, 401, 801, and 1601 grid points at $t = 0.2s$.



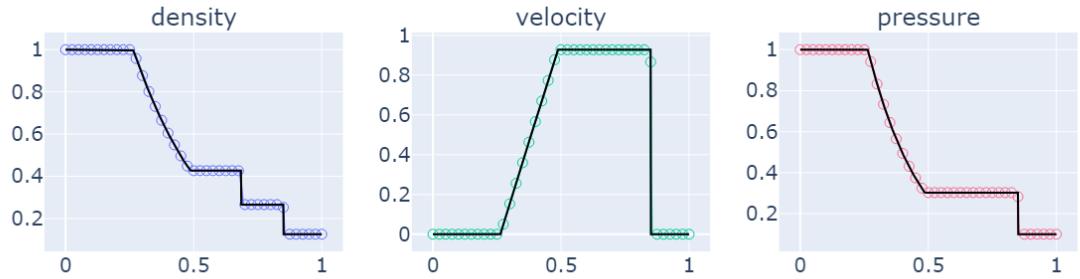
(a) MUSCL scheme with the van Leer flux limiter using 201 grid points



(b) MUSCL scheme with the van Leer flux limiter using 401 grid points



(c) MUSCL scheme with the van Leer flux limiter using 801 grid points



(d) MUSCL scheme with the van Leer flux limiter using 1601 grid points

Figure 4.4: Solution for the shock tube problem using the MUSCL scheme with the van Leer flux limiter with 201, 401, 801, and 1601 grid points at $t = 0.2s$.

To achieve higher-order accuracy, however, a flux limiter is mandatory to stabilise the numerical approximation in space. Using a flux limiter transforms a higher-order method (second-order and above) into a high-resolution method. Charles Hirsch writes in his book ”Numerical Computations of Internal & External Flows”:

It has to be said that the development of high-resolution schemes is one of the most remarkable achievements of the history of CFD.

Indeed, CFD without high-resolution schemes is unthinkable, like pizza without cheese. We really are lucky that so many before us have devoted so much time to developing schemes that can be used to predict strong non-linear phenomena, such as shock waves so that we can use them to predict the behaviour of real-world flows.

The grid resolution is not the only thing we can play with. Another important factor is the time step. This is investigated in the next section.

4.3.3. Influence of the time step

Table 4.1: Influence of the CFL number (time step) on the convergence of the MUSCL scheme with van Leer limiter on a grid with 101 points.

CFL	Iterations	Converged?	Oscillation?
0.10	430	Yes	No
0.25	172	Yes	No
0.50	83	Yes	No
0.57	91	Yes	Yes
0.58	47	No	Yes

As alluded to in the previous section, the time step plays a crucial role in our simulation as well. If we set it too low, we will wait for quite a long time to get results. If we set it too high, then our simulation may diverge.

We discussed in Section 3.3.1 that to have a stable time step, we first need to express it in some form of non-dimensional unit, which we do using the CFL number. Then, depending on our numerical scheme in time (e.g. here, the first-order Euler scheme), we can derive the max CFL condition.

While we could go through the stability analysis, this would be a bit over the top here and, for pretty much all convection-dominated problems (i.e. the Euler or Navier-Stokes equations) using a Reynolds number larger than one and an explicit time integration scheme (which we do), the upper limit is typically at $CFL = 1$. If diffusion dominates, this typically reduces to $CFL = 0.5$. Thus, if in doubt, set $CFL_{max} = 0.5$, and you should always be able to compute a stable time step from Eq.(3.4).

This is the theory, but in reality, we are making so many additional approximations, simplifications, and modifications that we rarely achieve divergence at $CFL = 1$. Sometimes, it is slightly above that limit (typically low Reynolds number, laminar, incompressible flows) or below for flows involving strong non-linear behaviour (turbulence or shock formation).

Table 4.1 shows the influence of the CFL number on the convergence of the MUSCL scheme with the van Leer limiter on a grid with 101 points. We start with $CFL = 0.1$ and go all the way up to $CFL = 0.58$, at which point divergence is detected.

As we increase the CFL number, we see that the number of iterations required to march the solution until the required end time of $t = 0.2s$ decreases. This makes sense. From Eq.(3.5), we see that the time step is proportional to the CFL number, and if the CFL number increases, then so does the time step.

A higher value of Δt leads to fewer time steps required to reach the simulation end time; however, at $CFL = 0.5$, we see this trend reversing. If we increase the CFL number further beyond this point, we see that we require more time steps than before. This is because we are now introducing some non-physical oscillations which destabilise the simulation, leading to smaller time steps and thus more time steps in total required to reach the simulation end time.

Let's examine the solution and see how the CFL number influences the results. This is shown in Figure 4.5. Look at the density profile and see how the result changes from a CFL number of $CFL = 0.1$, i.e. Figure 4.5a to a CFL number of $CFL = 0.5$, i.e. Figure 4.5c.

We can see that the solution is smeared, and the discontinuities are less well resolved. A larger CFL number brings about numerical diffusion in time, as it damps out any sharp features such as discontinuities. Thus, a lower CFL number may result in better accuracy at the cost of longer simulation times.

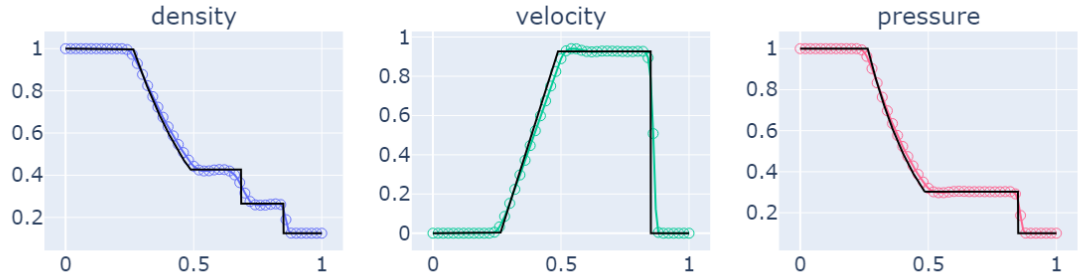
But now observe Figure 4.5d, where we have used $CFL = 0.57$. We saw before that, in this case, the number of steps required to reach the simulation end time is higher than in the $CFL = 0.5$ case. We can also see the oscillations (non-physical results) that appear in the simulations.

We have a few options to get rid of these oscillations. The simplest would be to implement further flux limiters and see if they can handle the oscillations better. Otherwise, a better Riemann solver may be another option. We could also switch from MUSCL to WENO schemes, another family of schemes commonly used to treat discontinuities (discussed further in Section 5.3). Finally, the time integration could be made more robust by using a higher-order numerical scheme in time, potentially with some TVD properties (e.g. a third-order Runge-Kutta explicit time stepping with TVD properties, which is commonly used for compressible flows).

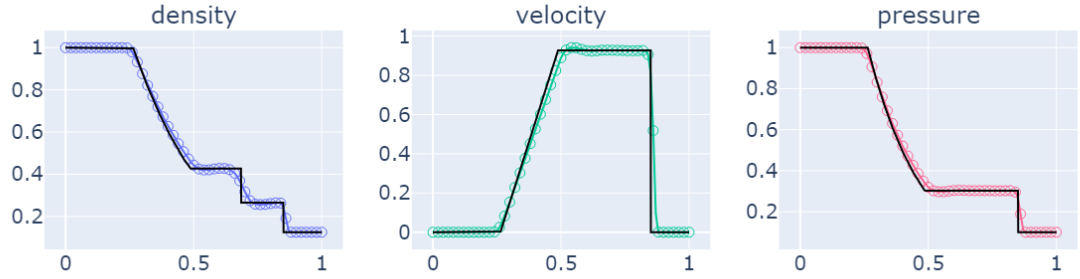
There are a few options, but for our current discussion, it is sufficient to realise that we cannot arbitrarily increase the time step as we want. If we need larger CFL numbers, we need to switch to an implicit time integration, which is unconditionally stable and thus allows for arbitrarily large CFL numbers.

4.4. Summary

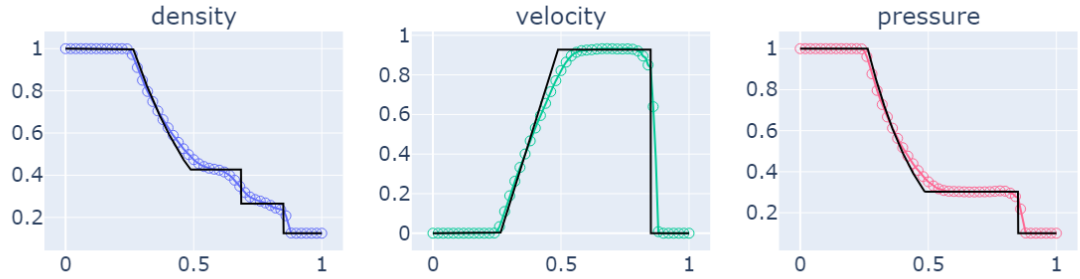
We have now examined the influence of the numerical schemes, the grid size, and the time step. We saw that they all influence the simulation, and certain combinations can lead to divergence.



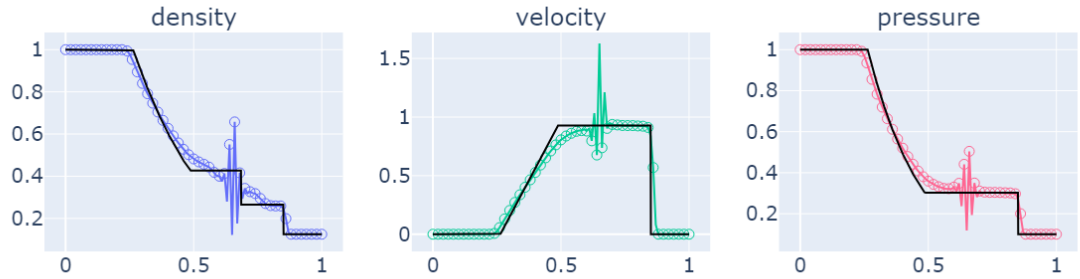
(a) MUSCL scheme with the van Leer flux limiter at $CFL = 0.1$



(b) MUSCL scheme with the van Leer flux limiter at $CFL = 0.25$



(c) MUSCL scheme with the van Leer flux limiter at $CFL = 0.5$



(d) MUSCL scheme with the van Leer flux limiter at $CFL = 0.57$

Figure 4.5: Solution for the shock tube problem using the MUSCL scheme with the van Leer flux limiter for different CFL values

I would encourage you to play around with the settings and observe how the results change. This is the best way to get an intuition for how our numerical schemes and settings influence results.

You might consider implementing additional numerical schemes or extending the solver to 2D or 3D. Use this solver as a playground to experiment and observe how different concepts are implemented. This experience will make it much easier for you to write your own solver in the future.

The following section will offer ideas for using the solver, allowing you to experiment further and test additional concepts.

But this is it. We have now seen how to take the theory discussed in Chapter 2 and implement that into code in Chapter 3. In the current chapter, we have discussed results that we have obtained with our CFD solver to gain some intuition for what the settings and schemes do to our results.

5

Where to go from here

You have made it to the end. If you followed all the steps and could understand and implement the code (or, instead, follow the implementation and understand it), then you have come a long way.

Sure, there will be concepts that will still feel a bit strange, and you may not have grasped everything in the first read. This is normal. CFD isn't an easy field to get into, as there are so many separate aspects we need to master.

However, I hope that you got the gist of how to write a simple 1D, inviscid CFD solver. While it is a simplified solver, do not mistake that for a simple solver. The steps we had to go through to get to a solution were anything but simple.

Every commercial and open-source CFD solver that wants to solve the compressible form of the Navier-Stokes equation has to implement high-resolution schemes (e.g. MUSCL scheme), flux limiters, and a Riemann solver.

The additional terms in the Navier-Stokes equations just make our discretised equation longer, and the addition of 3D instead of 1D also just makes the programming effort more significant, but at its core, you will find the same topics discussed in this guide in every other more mature CFD solver.

Thus, feel free to consult some of the references sprinkled in within this guide or perform your own research and read up on concepts you feel require some more reading to understand fully.

In the meantime, I want to give you some ideas you could try to do next. The following sections propose some extensions you could do that vary in the degree of complexity. You may not be comfortable with all of them, but there should be something here for everyone.

5.1. Implementing additional flux limiters

This is probably the most straightforward extension for you to work on. Initially, I had quite a few more flux limiters implemented, but I decided to go for two basic options so that you have some room to explore additional ones.

As a reminder, you can find a good overview of common flux limiters on Wikipedia. Pick one and then implement it within the MUSCL scheme as discussed in Section 2.3.1.2.

You will then also need to extend the enum on flux limiters as discussed in Section 3.1.4. This will allow you to pick your newly implemented flux limiter.

Try the Superbee, Sweby, and perhaps the van Albada flux limiter. They are straightforward to implement and should be a good place to start. Note that some flux limiters are not TVD or 2nd order. This isn't necessarily a problem. In fact, implement one of those as well and see how they stack up against other flux limiters that are TVD and second-order.

5.2. Implementing additional Riemann solvers

We have looked at the simplest of all Riemann solvers: the Rusanov Riemann solver. This is a great way to get started, but there are much better versions available.

Whenever I write a CFD solver and need a decent selection of Riemann solvers, I start with the Rusanov Riemann solver and then implement the HLL Riemann solver next. Once that is working, extending it to the HLLC Riemann solver isn't that much of a problem.

The HLL and HLLC Riemann solvers are great starting points for capturing more of the flow with greater accuracy. We spent quite a bit of time in Chapter 4 to look at the contact discontinuity, and the HLLC Riemann solver was specifically designed to capture this with greater accuracy (the C in HLLC stands for contact, i.e. capturing the contact discontinuity).

The best place to learn about both of them is the book by Toro, who also introduced the HLLC Riemann solver. If you have access to Springer's ebook collection (e.g. through your university), then you can access the chapter on the HLL and HLLC Riemann solver here.

If you don't have access, then you can find an overview of the HLL Riemann solver here, and for the HLLC Riemann solver here. However, the above-linked book is really the place you want to start to get some more understanding, including a sensible summary that walks you through step by step on how to implement both Riemann solvers.

5.3. Implementing WENO schemes

When we were looking at our numerical schemes to approximate the values at the faces, i.e. at $i \pm 1/2$, then we talked about the piecewise constant reconstruction (Section 2.3.1.1) and the MUSCL scheme (Section 2.3.1.2).

However, we saw that the piecewise constant reconstruction really needs a lot of points to capture discontinuities well when compared to the MUSCL scheme (i.e. see Figure 4.3 and Figure 4.4).

I briefly mentioned that there is a second type of scheme available, which is called the WENO (Weighted Essentially Non-Oscillatory) scheme. These involve a bit more

work to implement, but they are really powerful in capturing discontinuities, even at lower grid resolutions.

Probably the best description at a very accessible level is provided over at Scholarpedia, and you could go through this section and try to implement the scheme.

You'll need to implement Eq.(1–3) first in the article above. Then, you have to compute the smoothness indicator β_j as given in Eq.(8), which allows you to compute the non-linear weights as given by Eq.(9). With that, you are in a position to obtain the face reconstructed values at $i \pm 1/2$ as shown in Eq.(6).

The article is nicely written, but given that the equations are not presented in the order in which they are implemented, I thought of including it here in case you want to try it.

Additionally, you will need to use a lower-order approximation (e.g. piecewise constant) near the boundary faces, similar to what we have done in the MUSCL scheme (e.g. see lines 2–8 in Listing 11).

In the case of the MUSCL scheme, we only needed to do that for the cells directly attached to the boundary. However, in the case of the WENO scheme discussed above, we need the first and last two cells to use a lower-order approximation.

```
1  // use lower-order scheme near boundaries
2  for (int variable = 0; variable < 3; ++variable) {
3      Ufaces[0][FACE::WEST][variable] = U[0][variable];
4      Ufaces[0][FACE::EAST][variable] = U[0][variable];
5      Ufaces[1][FACE::WEST][variable] = U[1][variable];
6      Ufaces[1][FACE::EAST][variable] = U[1][variable];
7
8      Ufaces[parameters.numberOfPoints - 2][FACE::WEST][variable] = U[
       parameters.numberOfPoints - 2][variable];
9      Ufaces[parameters.numberOfPoints - 2][FACE::EAST][variable] = U[
       parameters.numberOfPoints - 2][variable];
10     Ufaces[parameters.numberOfPoints - 1][FACE::WEST][variable] = U[
       parameters.numberOfPoints - 1][variable];
11     Ufaces[parameters.numberOfPoints - 1][FACE::EAST][variable] = U[
       parameters.numberOfPoints - 1][variable];
12 }
13
14 // use high-resolution WENO scheme on interior nodes / cells
15 for (int i = 2; i < parameters.numberOfPoints - 2; i++) {
16   // Implement Eq.(1-3) (polynomial reconstruction at i+(1/2))
17   // Implement Eq.(8)   (smoothness indicator)
18   // Implement Eq.(9)   (non-linear weights)
19   // Implement Eq.(6)   (face reconstruction values at i+(1/2))
20 }
```

Listing 18: Boilerplate code to start the WENO scheme implementation.

To make things easier, I have provided you with some boilerplate code which you could use to get started. This is provided in Listing 18.

Look at lines 3–11. Here, we use the piecewise constant reconstruction for the first and last two cells to obtain the face values at $i \pm 1/2$. Then, on line 15, we loop over all

cells starting with index `i = 2` and going to `i < parameters.numberOfPoints - 2`.

As a reminder, when we implemented the MUSCL scheme, we were looping from `i = 1` to `i < parameters.numberOfPoints - 1`.

5.4. Extending the solver to 2D

Finally, to make things more interesting, we could also extend our solver from 1D to 2D (or even 3D). It isn't much more complicated than the 1D case, and, in fact, we only need to add one more dimension to our conserved variable vector $\mathbf{U}$ and the flux vector $\mathbf{F}(mathbfU)$.

We also need to add an additional loop over the y direction, i.e. whenever we loop over the internal cells in the x direction, we also need to loop over the internal cells in the y direction. For example, when we have:

```
for (int variable = 0; variable < 3; ++variable) {
  for (int i = 0; i < parameters.numberOfPoints ; i++) {
    // do something, e.g. reconstruct values at i+1/2
    Ufaces[i][FACE::WEST][variable] = U[i][variable];
  }
}
```

We need to replace that with a 2D version of the loop, i.e.:

```
for (int variable = 0; variable < 3; ++variable) {
  for (int i = 0; i < parameters.numberOfPoints; i++) {
    for (int j = 0; j < parameters.numberOfPoints; j++) {
    // do something, e.g. reconstruct values at i+1/2
    Ufaces[i][j][FACE::WEST][variable] = U[i][j][variable];
    }
  }
}
```

Notice how I have also added the additional index `j` to access information in the conserved variable vector `U`.

Now, you just have to go through the code and make extensions wherever you need. The governing equations, by the way, can be derived in a similar way as we did in Section 2.2.3, where we obtained the 1D system from the general 3D system (i.e. we take the 3D system and remove any derivative that contains either the z direction or the w velocity component).

Granted, making our solver 2D (or 3D) is probably the most involved, as we need to make changes at many different points in the code, but it is also not that difficult as long as we think which parts need to be modified and which can remain the same.

Once we have figured that out, it is usually a case of allocating memory for an additional index and looping over an additional loop in the y direction.

If this feels to big of a project, then that's fine. However, once you get 2D or even 3D work, you can investigate some really interesting problems, such as the double Mach reflection problem.

A

Full source code for the Euler CFD solver

This section contains the entire content of the `euler.cpp` file for quick reference. All code sections are discussed in detail in Chapter 3.

```cpp
#include <iostream>
#include <iomanip>
#include <vector>
#include <array>
#include <cmath>
#include <fstream>
#include <sstream>
#include <string>

// global enums for easy variable access
enum SCHEME { CONSTANT = 0, MUSCL};
enum LIMITER { NONE = 0, MINMOD, VANLEER};
enum FACE {WEST = 0, EAST};

// definition for case parameter structure to hold case-specific settings
struct caseParameters {
  int numberOfPoints;
  double gamma;
  double domainLength;
  double endTime;
  double CFL;
  double dx;
  double time;
  int timeStep;
};

int main() {

  /* ---------- Pre-processing ---------- */

    // Read parameters
    auto numericalScheme = SCHEME::MUSCL;
    auto limiter = LIMITER::VANLEER;
```

```
34      caseParameters parameters;
35
36      parameters.numberOfPoints = 101;
37      parameters.gamma = 1.4;
38      parameters.domainLength = 1.0;
39      parameters.endTime = 0.2;
40      parameters.CFL = 0.58;
41
42      parameters.dx = parameters.domainLength / (parameters.numberOfPoints -
        1);
43      parameters.time = 0.0;
44      parameters.timeStep = 0;
45
46      // Allocate memory
47      std::vector<double> x(parameters.numberOfPoints);
48      std::vector<std::array<double, 3>> U(parameters.numberOfPoints);
49      std::vector<std::array<std::array<double, 3>, 2>> Ufaces(parameters.
       numberOfPoints);
50      std::vector<std::array<std::array<double, 3>, 2>> Ffaces(parameters.
       numberOfPoints);
51
52      // Create/read mesh
53      for (int i = 0; i < parameters.numberOfPoints; i++) {
54        x[i] = i * parameters.dx;
55      }
56
57      // Initialise solution
58      double rho = 0.0;
59      double u = 0.0;
60      double p = 0.0;
61
62      for (int i = 0; i < parameters.numberOfPoints; i++) {
63        if (x[i] <= 0.5) {
64          rho = 1.0;
65          u = 0.0;
66          p = 1.0;
67        } else {
68          rho = 0.125;
69          u = 0.0;
70          p = 0.1;
71        }
72
73        U[i][0] = rho;
74        U[i][1] = rho * u;
75        U[i][2] = p / (parameters.gamma - 1.0) + 0.5 * rho * std::pow(u, 2);
76      }
77
78    /* ---------- Solving ---------- */
79      while (parameters.time < parameters.endTime) {
80
81        // Preparing solution update (store old solution and calculate
         stable timestep)
82        auto UOld = U;
83
84        // calculate stable time step
85        double speedMax = 0.0;
```

```
86      for (int i = 0; i < parameters.numberOfPoints; i++) {
87        // we need the primitive variables first to compute the wave speed
     (based on speed of sound and local velocity)
88        auto rho = U[i][0];
89        auto u = U[i][1] / rho;
90        auto p = (parameters.gamma - 1.0) * (U[i][2] - 0.5 * rho * std::
    pow(u, 2));
91
92        // calculate wave speed for each cell
93        double speedOfSound = std::sqrt(parameters.gamma * p / rho);
94        if (speedOfSound + std::fabs(u) > speedMax)
95          speedMax = speedOfSound + std::fabs(u);
96      }
97      double dt = (parameters.CFL * parameters.dx) / speedMax;
98
99      // Solve equations
100
101     // compute/interpolate conserved variables at faces
102     if (numericalScheme == SCHEME::CONSTANT) {
103       for (int i = 0; i < parameters.numberOfPoints; i++) {
104         for (int variable = 0; variable < 3; ++variable) {
105           Ufaces[i][FACE::WEST][variable] = U[i][variable];
106           Ufaces[i][FACE::EAST][variable] = U[i][variable];
107         }
108       }
109     } else if (numericalScheme == SCHEME::MUSCL) {
110       // use lower-order scheme near boundaries
111       for (int variable = 0; variable < 3; ++variable) {
112         Ufaces[0][FACE::WEST][variable] = U[0][variable];
113         Ufaces[0][FACE::EAST][variable] = U[0][variable];
114
115         Ufaces[parameters.numberOfPoints - 1][FACE::WEST][variable] = U[
    parameters.numberOfPoints - 1][variable];
116         Ufaces[parameters.numberOfPoints - 1][FACE::EAST][variable] = U[
    parameters.numberOfPoints - 1][variable];
117       }
118
119       // use high-resolution MUSCL scheme on interior nodes / cells
120       for (int i = 1; i < parameters.numberOfPoints - 1; i++) {
121         for (int variable = 0; variable < 3; ++variable) {
122           auto du_i_plus_half = U[i + 1][variable] - U[i][variable];
123           auto du_i_minus_half = U[i][variable] - U[i - 1][variable];
124
125           double rL = du_i_minus_half / (du_i_plus_half + 1e-8);
126           double rR = du_i_plus_half / (du_i_minus_half + 1e-8);
127
128           double psiL = 1.0;
129           double psiR = 1.0;
130
131           // apply limiter to make scheme TVD (total variation
    diminishing)
132           if (limiter == LIMITER::MINMOD) {
133             psiL = std::max(0.0, std::min(1.0, rL));
134             psiR = std::max(0.0, std::min(1.0, rR));
135           } else if (limiter == LIMITER::VANLEER) {
136             psiL = (rL + std::fabs(rL)) / (1.0 + std::fabs(rL));
```

```
137             psiR = (rR + std::fabs(rR)) / (1.0 + std::fabs(rR));
138           }
139
140           Ufaces[i][FACE::WEST][variable] = U[i][variable]
141             - 0.5 * psiL * du_i_plus_half;
142           Ufaces[i][FACE::EAST][variable] = U[i][variable]
143             + 0.5 * psiR * du_i_minus_half;
144         }
145       }
146     }
147
148     // compute fluxes at faces
149     std::array<double, 3> fluxL, fluxR;
150     for (int i=1; i < parameters.numberOfPoints - 1; i++) {
151       for (int face = FACE::WEST; face <= FACE::EAST; ++face) {
152         int indexOffset = 0;
153         if (face == FACE::WEST) indexOffset = 0;
154         else if (face == FACE::EAST) indexOffset = 1;
155
156         auto rhoL = Ufaces[i - 1 + indexOffset][FACE::EAST][0];
157         auto uL = Ufaces[i - 1 + indexOffset][FACE::EAST][1] / rhoL;
158         auto EL = Ufaces[i - 1 + indexOffset][FACE::EAST][2];
159         auto pL = (parameters.gamma - 1.0) * (EL - 0.5 * rhoL * std::pow
    (uL, 2));
160         auto aL = std::sqrt(parameters.gamma * pL / rhoL);
161
162         auto rhoR = Ufaces[i + indexOffset][FACE::WEST][0];
163         auto uR = Ufaces[i + indexOffset][FACE::WEST][1] / rhoR;
164         auto ER = Ufaces[i + indexOffset][FACE::WEST][2];
165         auto pR = (parameters.gamma - 1.0) * (ER - 0.5 * rhoR * std::pow
    (uR, 2));
166         auto aR = std::sqrt(parameters.gamma * pR / rhoR);
167
168         fluxL[0] = rhoL * uL;
169         fluxL[1] = pL + rhoL * std::pow(uL, 2);
170         fluxL[2] = uL * (EL + pL);
171
172         fluxR[0] = rhoR * uR;
173         fluxR[1] = pR + rhoR * std::pow(uR, 2);
174         fluxR[2] = uR * (ER + pR);
175
176         // Rusanov Riemann solver
177         auto speedMax = std::max(std::fabs(uL) + aL, std::fabs(uR) + aR)
    ;
178         for (int variable = 0; variable < 3; ++variable) {
179           const auto &qL = Ufaces[i - 1 + indexOffset][FACE::EAST][
    variable];
180           const auto &qR = Ufaces[i + indexOffset][FACE::WEST][variable
    ];
181           const auto &fL = fluxL[variable];
182           const auto &fR = fluxR[variable];
183           Ffaces[i][face][variable] = 0.5 * (fL + fR) - speedMax * (qR -
     qL);
184         }
185       }
186     }
```

```
187
188       // calculate updated solution
189       for (int i=1; i < parameters.numberOfPoints - 1; i++)
190         for (int j=0; j<3; j++) {
191           const auto &dF = Ffaces[i][FACE::EAST][j] - Ffaces[i][FACE::WEST
    ][j];
192           U[i][j] = UOld[i][j] - (dt / parameters.dx) * dF;
193         }
194
195       // Update boundary conditions
196       auto rhoL = 1.0;
197       auto uL = 0.0;
198
199       auto rhoR = 0.125;
200       auto uR = 0.0;
201
202       U[0][0] = U[1][0];
203       U[0][1] = rhoL * uL;
204       U[0][2] = U[1][2];
205
206       U[parameters.numberOfPoints - 1][0] = U[parameters.numberOfPoints -
    2][0];
207       U[parameters.numberOfPoints - 1][1] = rhoR * uR;
208       U[parameters.numberOfPoints - 1][2] = U[parameters.numberOfPoints -
    2][2];
209
210       // Output solution to csv file for plotting
211       std::ofstream outputFile;
212
213       // convert time step and points into 6 digits string with leading
    zeros
214       std::ostringstream timeStepTemp, pointsTemp;
215
216       timeStepTemp << std::setfill('0') << std::setw(6);
217       timeStepTemp << parameters.timeStep;
218       auto timeStep = timeStepTemp.str();
219
220       pointsTemp << std::setfill('0') << std::setw(6);
221       pointsTemp << parameters.numberOfPoints;
222       auto points = pointsTemp.str();
223
224       outputFile.open("solution_" + points + "_" + timeStep + ".csv");
225       outputFile << "x,rho,u,p" << std::endl;
226       for (int i = 0; i < parameters.numberOfPoints; i++) {
227         auto rho = U[i][0];
228         auto u = U[i][1] / rho;
229         auto p = (parameters.gamma - 1.0) * (U[i][2] - 0.5 * rho * std::
    pow(u, 2));
230         outputFile << x[i] << "," << rho << "," << u << "," << p << std::
    endl;
231       }
232       outputFile.close();
233
234       // output current time step information to screen
235       std::cout << "Current time: " << std::scientific << std::setw(10) <<
     std::setprecision(3) << parameters.time;
```

```
236       std::cout << ", End time: " << std::scientific << std::setw(10) <<
      std::setprecision(3) << parameters.endTime;
237       std::cout << ", Current time step: " << std::fixed << std::setw(7)
      << parameters.timeStep;
238       std::cout << "\r";
239
240       // Increment solution time and time step
241       parameters.time += dt;
242       parameters.timeStep++;
243     }
244
245   std::cout << "\nSimulation finished" << std::endl;
246
247   return 0;
248 }
```