Team Assignment
Internal flow in a channel with a thin symmetric bump (2D Euler equations)

Alice de Carolis
12402529
e12402529@student.tuwien.ac.at

Joshua Kofler
61806971
joshua.kofler@tuwien.ac.at

February 16, 2025

Contents

1 Introduction

The aim of this project is to develop a numerical solver for the two-dimensional Euler equations using a finite volume method to model inviscid, compressible flow through a channel with a symmetric bump. Spatial discretization is performed on a structured, quasi-uniform grid, employing a central scheme for the fluxes, with artificial dissipation incorporated to ensure numerical stability and suppress spurious oscillations. Temporal discretization will be performed using a 4-stage low-storage Runge-Kutta method to advance the solution toward a steady-state.

The computational domain extends from $x = -L$ to $x = 2L$ with a height of $1L$. It features a bump of height $y_0(x) = \varepsilon\, x\left(1 - \frac{x}{L}\right)$ located between $x = 0$ and $x = L$, where $\varepsilon = 0.08$. The boundary condition at the inlet prescribes axial flow ($v = 0$) and specifies the stagnation pressure and stagnation temperature. At the outlet, the static pressure is set to the atmospheric pressure.

The source code used in this work is available at [DOI/link] for reproducibility and further analysis.

1.1 Objective

The primary objective of this project is to obtain a steady-state solution that satisfies the governing equations and the prescribed boundary conditions. Key quantities of interest, such as the pressure coefficient C_p along the bottom wall and the local Mach number across the computational domain, will be investigated. Additionally, the compressibility effects will be explored by varying the Mach number, allowing for an analysis of how these changes influence the flow dynamics.

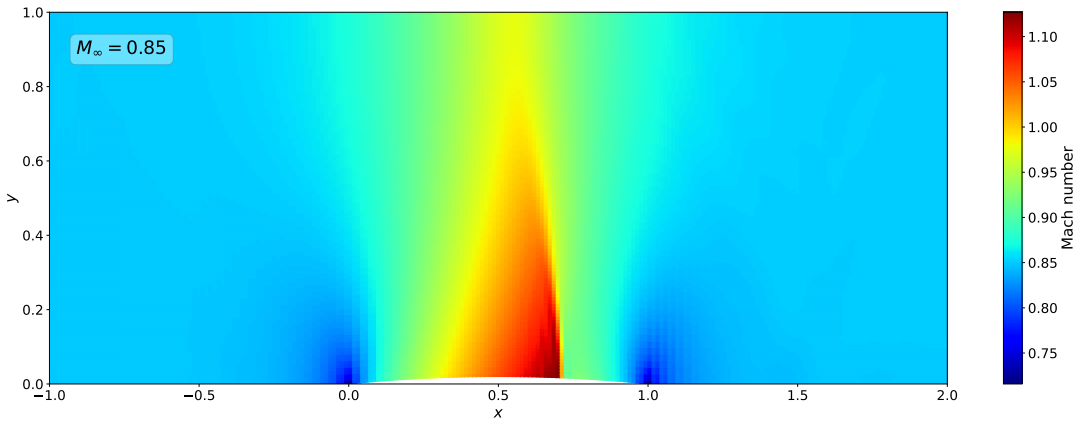


Figure 1: Contours of local Mach number as computed on the (225×113) mesh

2 Overview

At high Reynolds numbers, except within viscous regions that develop near solid surfaces, viscosity and heat conduction can be disregarded, giving rise to an inviscid model referred to as the **Euler equations**. The Euler equations describe the most general flow configuration for a non-viscous, non-heat-conducting fluid. Mathematically, they are represented as a system of coupled first order partial differential equations hyperbolic in time and space.

Below is the simplified, two-dimensional form of the Euler equations. The problem involves three independent variables: time t and the spatial coordinates x and y. In addition, there are four dependent variables: pressure p, density ρ, and the two components of the velocity vector $\vec{u} = (u, v)$. The velocity component u corresponds to the x-direction, while v corresponds to the y-direction. Each of these dependent variables is a function of both x and y, as well as time.

The governing equations are as follows:

$$\frac{\partial \rho}{\partial t} + \frac{\partial (\rho u)}{\partial x} + \frac{\partial (\rho v)}{\partial y} = 0 \tag{2.1}$$

$$\frac{\partial (\rho u)}{\partial t} + \frac{\partial (\rho u^2)}{\partial x} + \frac{\partial (\rho uv)}{\partial y} = -\frac{\partial p}{\partial x} \tag{2.2}$$

$$\frac{\partial (\rho v)}{\partial t} + \frac{\partial (\rho uv)}{\partial x} + \frac{\partial (\rho v^2)}{\partial y} = -\frac{\partial p}{\partial y} \tag{2.3}$$

$$\frac{\partial (\rho E)}{\partial t} + \frac{\partial (\rho uH)}{\partial x} + \frac{\partial (\rho vH)}{\partial y} = 0 \tag{2.4}$$

This set of equations can be rewritten in conservative form as:

$$\frac{\partial U}{\partial t} + \nabla \cdot \vec{F} = Q, \quad \text{with } \vec{F} = \begin{pmatrix} f \\ g \end{pmatrix} \tag{2.5}$$

Here, U denotes the vector of conserved variables, $\vec{F}$ represents the flux vector, and Q is the source term. In this case, Q equals zero, since no body forces or external energy sources are considered. In two dimensions, the vectors are given by:

$$U = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho E \end{pmatrix}, \quad f = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uH \end{pmatrix}, \quad g = \begin{pmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ \rho vH \end{pmatrix} \tag{2.6}$$

To fully define the problem, the equation of state and the corresponding thermodynamic relations must be incorporated. In this work, these relations are derived under the assumption of a **perfect gas** with **constant specific heats**.

2.1 Finite Volume Method for Euler Equations

The **finite volume method** (FVM) is employed to solve the governing equations, which are first formulated in their integral form:

$$\frac{\partial}{\partial t}\int_{\Omega} U \, d\Omega + \oint_{S} \vec{F} \cdot d\vec{S} = 0 \tag{2.7}$$

The integral formulation is discretized directly in the physical space. Thus, Equation 2.7 is reformulated by substituting the volume integrals with the cell-averaged values and the surface integral with a sum over all the bounding faces of the considered volume Ω_{ij}.

$$\frac{d}{dt}\left[\overline{U}_{ij}\Omega_{ij}\right] = -\sum_{faces} \vec{F}^{*} \cdot \Delta\vec{S} \equiv -R_{ij} \tag{2.8}$$

Here, the right-hand side defines the **residual** R_{ij} as the net flux balance across all faces of cell (i, j). Within the FVM framework, the solution $\overline{U}_{ij}$ represents the cell-averaged conservative variable U. During post-processing, these values are assigned to the cell centers, introducing in general a second-order discretization error. Note that the numerical flux $\vec{F}^{*}$ represents the discretized physical flux.

2.2 Space discretization

Spatial discretization is carried out on a **structured** grid, more specifically, an H-grid, employing a **cell-centered** approach. The cell faces are assumed to be straight lines, as typical for second-order approximations. Cell vertices are denoted as A, B, C, and D, while its faces are labeled by the cardinal directions: south, east, north, and west.

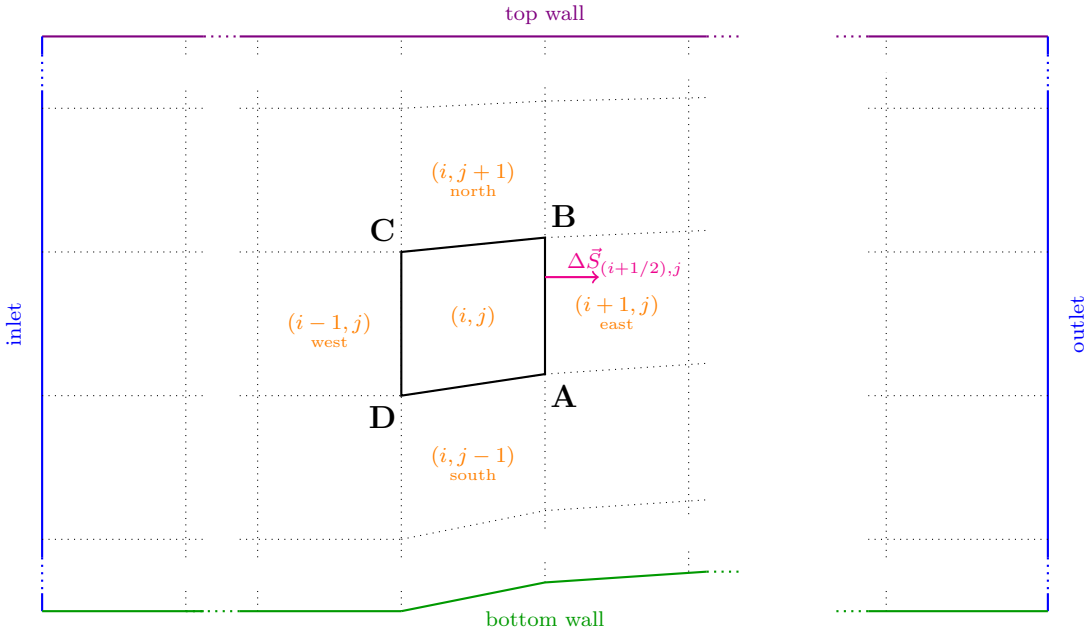


Figure 2: Structured mesh used for the channel flow

To compute the flux across a cell face, it is necessary to determine the corresponding surface vector $\Delta\vec{S}$. This is achieved by first calculating the face length and defining the unit normal vector. The surface vector is then obtained as the product of the unit normal vector and the face length. This procedure is applied to each face; for instance, the surface vector for the east face is given by:

$$\Delta\vec{S}_{AB} = \vec{n}_{AB} \cdot \Delta S_{AB} = \begin{pmatrix} y_B - y_A \\ -(x_B - x_A) \end{pmatrix}. \tag{2.9}$$

In addition to the surface vectors, the cell areas must also be computed. for a general quadrilateral $ABCD$, the area Ω_{ij} is determined using the vector cross product of its diagonals, given by:

$$\Omega_{ij} = \frac{1}{2}(\vec{x}_{AC} \times \vec{x}_{BD}) \tag{2.10}$$

2.2.1 Implementation

The implementation can be found in the `mesh.py` module, where the surface vector is computed in the `_calculate_ndS()` function. As an example, the calculation for the east face surface vector is shown below. In the code, the `ndS` array represents the surface vector for each face, with the first two indices corresponding to the cell, and the third index indicating the face. The values 0, 1, 2, and 3 represent the south, east, north, and west face, respectively. The final index denotes the x or y component.

src/mesh.py: Code for calculating the surface vector for the east face

```
291         # Face 1 (East): Between points A and B
292         gv.ndS[cell_x_index, cell_y_index, 1, :] = np.array([
293             (y_point_b - y_point_a),  # x-component
294             -(x_point_b - x_point_a)  # y-component
295         ])
```

The cell area is calculated within the function `_calculate_cell_area()`. In both cases, the coordinates of the cell vertices are obtained using the `get_vertex_coordinates()` function.

```
238             # Calculate the area using the vector cross product
239             gv.cell_area[cell_x_index, cell_y_index] = 0.5 * ((x_point_c - x_point_a) * (y_point_d - y_point_b)
240                                                               - (x_point_d - x_point_b) * (y_point_c - y_point_a))
```

2.3 Evaluation of fluxes through cell faces

As noted in Equation 2.8, the evaluation of flux components along cell faces depends on the selected numerical scheme, as well as on the location of the flow variables relative to the mesh. A detailed discussion of various approaches can be found in [1].

In this work, a central discretization is applied to the fluxes along the faces, supplemented by an **artificial dissipation term** to enhance numerical stability. For central schemes and cell-centered finite volume methods, the most straightforward approach is to compute the flux at the face as the arithmetic mean. As shown for the east face below, the arithmetic mean and, consequently, the flux at the face, $\vec{F}_{(i+1/2),j}$, is defined as:

$$\vec{F}_{(i+1/2),j} = \frac{1}{2}(\vec{F}_{i,j} + \vec{F}_{(i+1),j}) \tag{2.11}$$

where the flux for a given cell, $\vec{F}_{i,j}$, is defined as:

$$\vec{F}_{i,j} = \begin{pmatrix} f_{i,j} \\ g_{i,j} \end{pmatrix} \quad \text{where} \quad f_{i,j} = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uH \end{pmatrix}_{i,j}, \quad g_{i,j} = \begin{pmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ \rho vH \end{pmatrix}_{i,j} \tag{2.12}$$

2.3.1 Implementation

The calculation is implemented in the `update_flux` function within `calculate_flux.py`. Initially, the components f and g in the cell center are updated using the previously updated cell quantities.

src/calculate_flux.py: Code for calculating the components of the flux vector

```
9    # Update the flux vector components (f) in the x-direction
10   gv.f[:, :, 0] = gv.rho[:, :] * gv.u[:, :, 0]
11   gv.f[:, :, 1] = gv.rho[:, :] * np.power(gv.u[:, :, 0], 2) + gv.p[:, :]
12   gv.f[:, :, 2] = gv.rho[:, :] * gv.u[:, :, 0] * gv.u[:, :, 1]
13   gv.f[:, :, 3] = gv.rho[:, :] * gv.u[:, :, 0] * gv.H[:, :]
14
15   # Update the flux vector components (g) in the y-direction
16   gv.g[:, :, 0] = gv.rho[:, :] * gv.u[:, :, 1]
17   gv.g[:, :, 1] = gv.rho[:, :] * gv.u[:, :, 0] * gv.u[:, :, 1]
18   gv.g[:, :, 2] = gv.rho[:, :] * np.power(gv.u[:, :, 1], 2) + gv.p[:, :]
19   gv.g[:, :, 3] = gv.rho[:, :] * gv.u[:, :, 1] * gv.H[:, :]
```

Using the flux vectors in the cell center, the flux vectors at the cell faces can be calculated. Consider first the flux vector at the east face. As illustrated in Figure 2, the y component of the normal vector is zero for both the east and west faces. Consequently, the y component of the flux vector, g, does not contribute and can be disregarded. The flux vector at the west face is then obtained as the negative of the east face flux vector from the neighboring cell.

src/calculate_flux.py: Code for calculating the fluxes of the east and west face

```
36   # East flux (index 1)
37   gv.F[:-1, :, 1, :] = (0.5 * (gv.f[:-1, :, :] + gv.f[1:, :, :]) * gv.ndS[:-1, :, 1, 0][..., np.newaxis])
```

```
45   # West flux (index 3)
46   gv.F[1:, :, 3, :] = - gv.F[:-1, :, 1, :]
```

Similarly, the flux vectors at the north and south faces can be computed. Unlike the east and west faces, the normal vectors in these directions can have arbitrary orientations. As a result, both the x- and y-components of the flux vector must be considered.

src/calculate_flux.py: Code for calculating the fluxes of the east and west face

```
59    # North flux (index 2)
60    gv.F[:, :-1, 2, :] = (0.5 * (gv.f[:, :-1, :] + gv.f[:, 1:, :]) * gv.ndS[:, :-1, 2, 0][..., np.newaxis]
61                          + 0.5 * (gv.g[:, :-1, :] + gv.g[:, 1:, :]) * gv.ndS[:, :-1, 2, 1][..., np.newaxis])
68    # South flux (index 0)
69    gv.F[:, 1:, 0, :] = - gv.F[:, :-1, 2, :]
```

Flux vectors at boundary cells require special treatment, which is discussed in detail in Section 2.6.

2.4 Artificial dissipation

Due to the possibility of the development of discontinuities, the numerical scheme will develop oscillations. Therefore, additional diffusion needs to be added near the discontinuity. The aim is to add a dissipative contribution to the scheme, that is an even order derivative, with a truncation error lower than the truncation error of the selected discretization. A popular scheme was designed by Jameson-Schmidt-Turkel (JST). Find more details in [2].

As previously mentioned, the numerical flux $\vec{F}^*$ is given by the sum of the flux at the face and an artificial dissipation term, as illustrated for the east and west face below:

$$(\vec{F}^* \cdot \Delta\vec{S})_{(i\pm1/2),j} = \vec{F}_{(i\pm1/2),j} \cdot \Delta\vec{S} \mp D_{(i\pm1/2),j} \tag{2.13}$$

The JST blends a high and low diffusion term, preserving second-order accuracy far from discontinuities, while maintaining first-order accuracy near them. In **subsonic** flow, adding a third-order derivative to the flux is typically sufficient. However, in **supersonic** flows with shocks, it is recommended to use this blend of first- and third-order derivatives for the artificial dissipation. Accordingly, for the east face, it is given by:

$$\begin{aligned} D_{(i+1/2),j} = {} & \eta_{(i+1/2),j} \left(U_{(i+1),j} - U_{i,j}\right) \\ & - \gamma_{(i+1/2),j} \left(U_{(i+2),j} - 3U_{(i+1),j} + 3U_{i,j} - U_{(i-1),j}\right) \end{aligned} \tag{2.14}$$

The coefficients η and γ at the cell faces are defined as follows:

$$\eta_{(i+1/2),j} = \frac{1}{2}\kappa^{(2)} \left[|\vec{u} \cdot \Delta\vec{S}| + c|\Delta\vec{S}|\right]_{(i+1/2),j} \cdot \nu_{(i+1/2),j} \tag{2.15}$$

$$\gamma_{(i+1/2),j} = \max\left(0,\ \frac{1}{2}\kappa^{(4)} \left[|\vec{u} \cdot \Delta\vec{S}| + c|\Delta\vec{S}|\right]_{(i+1/2),j} - \eta_{(i+1/2),j}\right) \tag{2.16}$$

where $\kappa^{(2)} = 1$ and $\kappa^{(4)} = 1/256$.

To maintain accuracy, the first artificial dissipation term must be deactivated far from discontinuities, while the second term remains essential to ensure consistent convergence to the steady state. To achieve this, a pressure-based sensor ν is introduced:

$$\nu_{ij} = \left| \frac{p_{(i+1),j} - 2p_{i,j} + p_{(i-1),j}}{p_{(i+1),j} + 2p_{i,j} + p_{(i-1),j}} \right| \tag{2.17}$$

The sensor value at the cell face is determined by the maximum pressure variation in the surrounding stencil:

$$\nu_{(i+1/2),j} = \max\left(\nu_{(i-1),j},\ \nu_{i,j},\ \nu_{(i+1),j},\ \nu_{(i+2),j}\right) \tag{2.18}$$

2.4.1 Implementation

In the module `calculate_artificial_dissipation.py` , the calculation of artifical dissipation is implemented, structured into two primary sections. The first section defines the **subsonic** artifical dissipation, while the second addresses the **supersonic** artificial dissipation. The function names are identical, with a `_subsonic` / `_supersonic` appendix. Due to the inherent complexity, the detailed implementation is not presented directly; instead, references to the supersonic implementation are provided to direct the user to the relevant sections of the code.

The pressure sensor is computed by the function `_calculate_pressure_sensor()`. It is important to note that, at the domain boundaries, the stencil used for the calculation of the face maximum values is reduced from four points to three points.
The required artificial dissipation coefficients, η and γ, are determined within the function `_calculate_coefficient_supersonic()`. Initially, the function performs the pressure sensor calculation, which is then used to compute the coefficients, accordingly. It is worth noting that the coefficients are calculated only for the east and north directions, with the coefficients for the west and south directions being identical, respectively.
The function `update_artificial_dissipation_supersonic()` computes the actual artificial dissipation for each cell face based on the vector of conserved variables. For interior cells, the artificial dissipation is calculated using Equation 2.14. However, for boundary cells, a simplified first-order approximation is applied due to the limited number of neighboring points available.

2.5 Time integration

In this work, the time integration is performed using a low-storage, fourth-order Runge-Kutta (RK4) method, expressed mathematically as:

$$Y_1 = U_{ij}^{(n)} - \frac{\Delta t}{\Omega_{ij}} \alpha_1 \, R_{ij}^{(n)} \tag{2.19}$$

$$Y_2 = U_{ij}^{(n)} - \frac{\Delta t}{\Omega_{ij}} \alpha_2 \, R_{ij}(Y_1) \tag{2.20}$$

$$Y_3 = U_{ij}^{(n)} - \frac{\Delta t}{\Omega_{ij}} \alpha_3 \, R_{ij}(Y_2) \tag{2.21}$$

$$U_{ij}^{(n+1)} = U_{ij}^{(n)} - \frac{\Delta t}{\Omega_{ij}} \, R_{ij}(Y_3) \tag{2.22}$$

with the following coefficients:

$$\alpha_1 = \frac{1}{4}, \quad \alpha_2 = \frac{1}{3}, \quad \alpha_3 = \frac{1}{2} \tag{2.23}$$

The time step Δt is determined based on the stability criterion, which ensures that the physical domain of dependence is entirely contained within the numerical domain of dependence. Thus, for the RK4 method, the CFL number must be less than 2.8. In the present work, the CFL is set to 2. The time step is computed using the following formula:

$$\Delta t = \frac{\mathrm{CFL}}{\frac{u_{\max}}{\Delta x} + \frac{v_{\max}}{\Delta y}} \tag{2.24}$$

where the maximum velocity components are given by:

$$\vec{u}_{\max} = \max\left(|\vec{u} + c|, \, |\vec{u} - c|\right) \tag{2.25}$$

2.5.1 Implementation

The Runge-Kutta method is implemented in the `RK.py` module. During each sub-step, the time step, cell properties, and residuals are updated to ensure the accurate evolution of the solution. After completing all stages of the Runge-Kutta method, the conserved variables vector $\overline{U}_{ij}$ is updated with the final computed values.

src/RK.py: Runge-Kutta method

```
47    # Perform the four-step Runge-Kutta time integration
48    for step in range(4):
49        # Calculate the timestep based on the maximum velocity in the domain
50        _calculate_timestep()
51
52        # Vectorized computation of the intermediate solution Y using the RK method
53        Y[:, :, :] = (gv.state_vector[:, :, :]
54                      - (gv.dt / gv.cell_area[:, :, np.newaxis] * RK_ALPHA[step] * gv.R[:, :, :]))
```

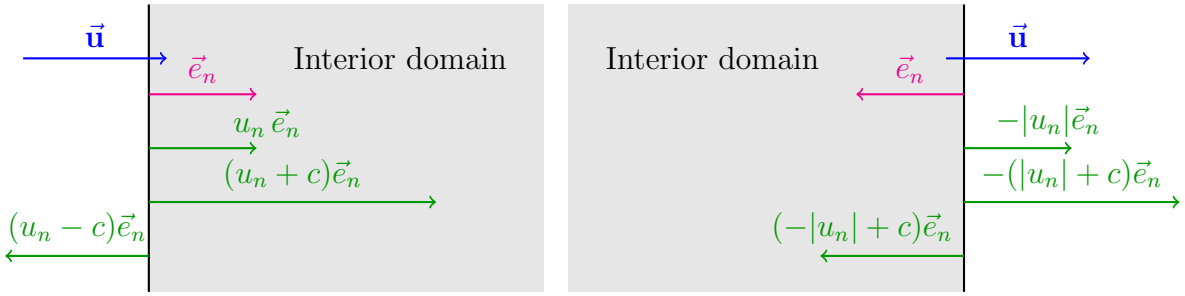
```
        # Update the physical properties of each cell based on the new intermediate state vector Y
        cell.update_cell_properties(Y)

        # Recalculate residuals based on the updated intermediate conserved variables (Y)
        cR.update_residual(Y)

    # After completing all RK steps, overwrite the state vector with the final computed values
    gv.state_vector[:, :, :] = Y[:, :, :]
```

2.6 Boundary conditions for the Euler equations

The boundaries represent a most critical component of any CFD code and must be consistent with both the physical and numerical properties of the problem addressed. A concise overview is presented in [1].



(a) inlet (b) outlet

Figure 3: Characteristic propagation properties at an boundary with subsonic conditions

2.6.1 Inlet

Referring to Figure 3a, for an inlet boundary with subsonic flow in the direction normal to the inlet surface, three eigenvalues are positive, while one eigenvalue, $\lambda_4 = u_n - c$, is negative. Therefore, three quantities will have to be fixed by the physical flow conditions at the inlet of the flow domain, while the remaining one will be determined by the interior conditions, through a numerical boundary condition.

A common approach, often applied to internal flows, is to specify the **stagnation pressure** and **temperature**, along with the **inlet flow angle**. To begin, the inlet flow angle can be fixed by setting the tangential velocity v_{in} to zero:

$$v_{\text{in}} = 0 \tag{2.26}$$

Using the Riemann invariants, a system of equations can be derived to determine the streamwise velocity u_{in} and the speed of sound c_{in} at the inlet.

$$u_{\text{in}} + \frac{2}{\gamma - 1} c_{\text{in}} = u_\infty + \frac{2}{\gamma - 1} c_\infty \tag{2.27}$$

$$u_{\text{in}} - \frac{2}{\gamma - 1} c_{\text{in}} = u_1 - \frac{2}{\gamma - 1} c_1 \tag{2.28}$$

Next, the inlet temperature T_{in} is calculated using the Newton-Laplace equation.

$$T_{\text{in}} = \frac{c_{\text{in}}^2}{\gamma R} \tag{2.29}$$

The inlet pressure p_{in} and inlet density ρ_{in} are then computed using the isentropic relations and the equation of state for an ideal gas:

$$\frac{p_{\text{in}}}{\rho_{\text{in}}^{\gamma}} = \frac{p_{\infty}}{\rho_{\infty}^{\gamma}}, \qquad p_{\text{in}} = \rho_{\text{in}} R T_{\text{in}} \tag{2.30}$$

Finally, the total inlet enthalpy, H_{in}, is the sum of static enthalpy and the kinetic energy.

$$H_{\text{in}} = c_p T_{\text{in}} + \frac{1}{2}\left(u_{\text{in}}^2 + v_{\text{in}}^2\right) \tag{2.31}$$

The computed quantities can then be applied to determine the flux vector at the western boundary of the inlet cell.

2.6.2 Outlet

At an outlet boundary with subsonic normal velocity, as illustrated in Figure 3b, three eigenvalues are negative because the normal vector is defined as pointing inward toward the flow domain. Thus, three numerical boundary conditions must be specified, while the fourth propagates information from the boundary into the flow domain and corresponds to a physical boundary condition.
The most suitable physical boundary condition for internal flows, in line with most experimental setups, is to set the downstream static pressure, assumed to be equal to atmospheric pressure:

$$p_{\text{out}} = p_a \tag{2.32}$$

Using the isentropic relation, the outlet density, ρ_{out}, is calculated. The outlet temperature, T_{out}, is derived from the equation of state, and the speed of sound, c_{out}, follows from the Newton-Laplace equation.
Once the speed of sound at the oulet ist known, the Riemann invariant R^+ is used to calculate the streamwise velocity u_{out}:

$$u_{\text{out}} = u_{N_x} + \frac{2}{\gamma - 1}\left(c_{N_x} - c_{\text{out}}\right). \tag{2.33}$$

The tangential velocity component at the outlet, v_{out}, is given directly by:

$$v_{\text{out}} = v_{N_x}. \tag{2.34}$$

At last, the total outlet enthalpy, H_{out}, is calculated. As with the inlet, these quantities are then used to determine the flux vector at the eastern boundary of the outlet cell.

2.6.3 Top and bottom wall

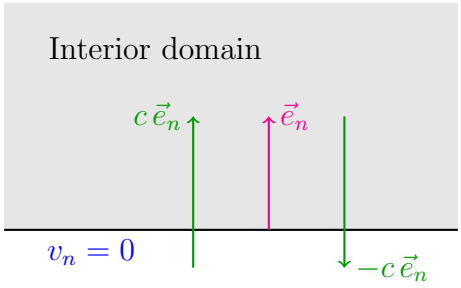


Figure 4: Characteristic propagation properties at an solid boundary

At a solid wall boundary, the normal velocity is zero, since no mass or other convective fluxes can penetrate the solid body. Hence only one eigenvalue is positive and only one physical condition can be imposed, namely $v_n = 0$. Substituting this condition into the flux expression yields:

$$\vec{F} \cdot \Delta\vec{S} = f\,\Delta S_x + g\,\Delta S_y = \begin{pmatrix} 0 \\ p\,\Delta S_x \\ p\,\Delta S_y \\ 0 \end{pmatrix} \tag{2.35}$$

In this work, zero-order extrapolation is used to determine the extrapolated value. This method directly projects the interior value onto the adjacent surface, assuming that it is piecewise constant within the cell, so that the boundary value equals the cell's mean value.

$$p_{\text{top}} = p_{N_y}, \qquad p_{\text{bot}} = p_0 \tag{2.36}$$

Note that the top boundary represents a special case, as the normal vector points directly in the vertical direction, so that ΔS_x equals zero.

2.6.4 Implementation

The stagnation properties, which remain constant throughout the simulation, are computed in the `calculate_inlet_properties()` function within `cell.py`. This function is called right at the start of the simulation.
On the other hand, the flux calculations at the boundaries, as derived in Section 2.6, are implemented in the `_calculate_boundary_flux()` function of the `calculate_flux.py` module. This function is invoked after each iteration step, including every Runge-Kutta sub-step, to update the boundary fluxes in response to changes in the interior cell values.

3 Results

As outlined in the introduction, the primary objective of this project was to obtain a steady-state solution, which was successfully achieved. In addition, a more in-depth analysis of the obtained solution was conducted. The results are presented below.

3.1 Study on Compressibility Effects

In a first analysis, the effects of compressibility are examined by studying the local Mach number distribution, as shown in Figure 5. The figure highlights the impact of varying incident Mach numbers on the flow field, particularly the formation of a supersonic region as the Mach number increases. The presented results are derived from numerical simulations conducted on a 65×33 computational mesh, considering incident Mach numbers in the range of $M = 0.2$ to $M = 0.85$.

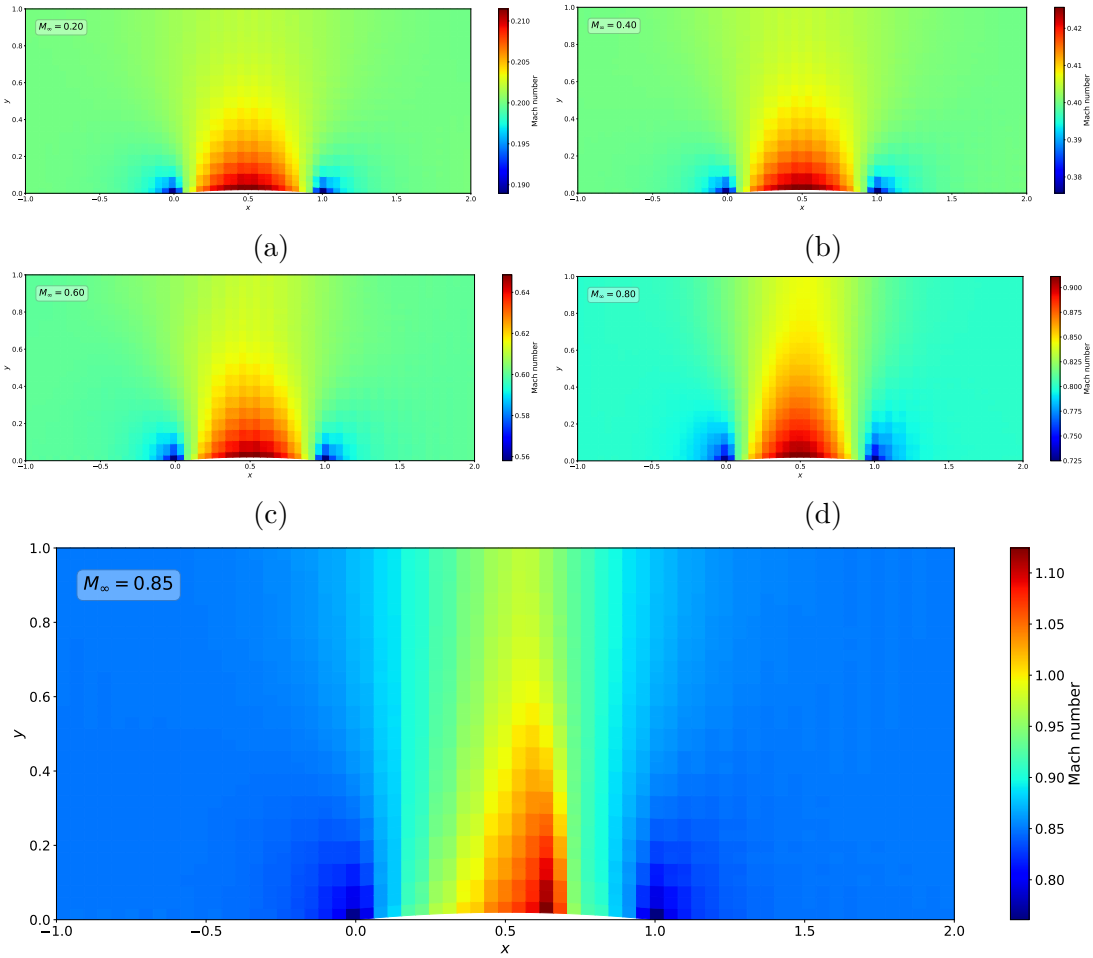


Figure 5: Distribution of local Mach number as computed on a (65×33) mesh, for different values of the incident Mach number, from 0.2 to 0.85. Observe the supersonic region appearing at M=0.85.

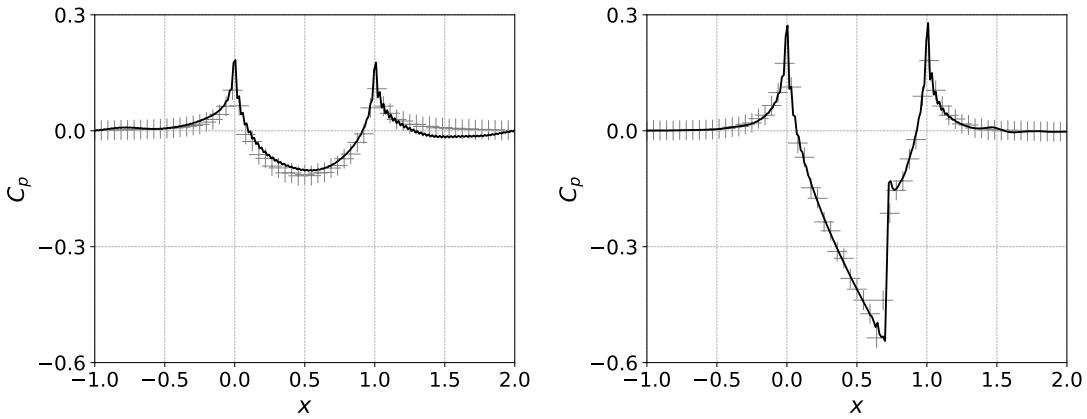
At lower Mach numbers (Figures 5a–5d), the flow remains subsonic. The solution shows only a slight dependence to variations in the Mach number. The velocity distribution transistions gradually as the Mach number increases, with peak velocities occuring near the center of the domain, at the maximum height of the bump.
As the Mach number increases, the compressibility effects become increasingly significant. At an incident Mach number of just above $M = 0.8$, the flow reaches the sonic speed ($M = 1$) on the bump surface, leading to the formation of a supersonic region that is subsequently terminated by a shock. This is evident in both Figure 5e and Figure 1, the latter presented in the introduction, which depicts the same flow under identical conditions but with a refined mesh of resolution 255×113.

3.2 Pressure coefficient at the bottom wall

A second, detailed examination of the solution is performed by analyzing the pressure coefficient along the bottom wall. The pressure coefficient, C_p, is a dimensionless parameter defined as:

$$C_p = \frac{p - p_0}{\frac{1}{2}\rho_0 u_0^2}, \tag{3.1}$$

where the subscript 0 refers to the mean value at the inlet, corresponding to the first column of cells.



(a) $M = 0.2$ (b) $M = 0.85$

Figure 6: Axial distribution of pressure coefficient as computed using the (65×33) mesh (plus signs) and compared to the (225×113) mesh (continuous line)

Figure 6 illustrates the distribution of the pressure coefficient along the bottom wall for two different Mach numbers and two mesh resolutions: a coarse grid (65×33) and a fine grid (225×113). In Figure 6a, the pressure coefficient distribution for subsonic flow conditions is presented. The results exhibit a smooth, well-resolved solution. Conversely, Figure 6b displays the solution for a supersonic flow condition ($M = 0.85$), where a dis-

tinct shock wave is evident. The rapid pressure variation across the shock is reflected by a steep gradient in the pressure coefficient, aligning well with the findings from the first analysis. A comparison between the results obtained from the two mesh sizes indicates that the grid resolution has minimal influence on the overall pressure coefficient distribution. However, due to computational constraints, the solution with the fine mesh did not converge to the desired accuracy, with high residuals persisting and spurious reflections continuing to appear downstream of the bump.

4 Discussion

The results indicate that the implementation of the underlying code is largely correct; however, further investigation is required to fully validate this assessment. Besides, there is still potential for optimization in terms of memory usage and computational efficiency. For example, the storage of face fluxes could be improved to **eliminate redundancies**, and the computation of the time step could be restructured to utilize a **local time-stepping** approach. Since the analysis focuses solely on the steady-state solution, each cell could progress independently at its maximum permissible time step, significantly accelerating convergence. Additionally, the implementation of ghost cells at the boundaries could be explored to minimize spurious reflections.

The results validate the ability of the employed model to accurately capture the effects of compressibility. At subsonic Mach numbers ($M = 0.2$ to $M = 0.8$), the flow remains smooth with gradual velocity transitions. At $M = 0.85$, a supersonic region emerges, terminated by a shock, characterized by steep gradients in velocity and pressure, consistent with theoretical predictions. The agreement between the coarse and fine mesh solutions underscores the robustness and numerical stability of the method.

The pressure coefficient analysis reinforces these findings. For $M = 0.2$, C_p exhibits a smooth distribution, while at $M = 0.85$, a sharp gradient reflects the shock. The comparison of mesh resolutions shows that the pressure coefficient distribution is largely unaffected by numerical dissipation.

Looking ahead, an important next step would be to perform a grid refinement study to further validate the accuracy and convergence of the implemented code. Moreover, a valuable extension would be to analyze the entropy distribution, which offers insights into regions influenced by numerical dissipation. As emphasized by [1], it is common practice to first run an Euler simulation of the same test case on the same grid before conducting a full Navier-Stokes simulation. This initial Euler simulation helps identify regions where numerical dissipation might significantly affect the solution by tracking the growth of entropy. Such a process is crucial for ensuring the reliability of the Navier-Stokes results, as it allows for better understanding and mitigation of dissipation effects.

References

[1] Charles Hirsch. *Numerical Computation of Internal and External Flows: The Fundamentals of Computational Fluid Dynamics*. Accessed February 2025. Elsevier, 2007. ISBN: 9780080550022.

[2] Antony Jameson. *Computational Aerodynamics*. Accessed February 2025. Cambridge University Press, 2022. ISBN: 978-1108943345. DOI: 10.1017/9781108943345.