MSc in Mathematical Engineering

Course: Advanced Programming for scientific computing

Lecturer: Prof. Luca Formaggia
Teaching Assistant: Dr. Beatrice Crippa & Dr. Alberto Artoni

Type B project

A.Y. 2024/2025

A BGK Kinetic Model for the Study of Gas–Condensed Phase Interaction

---

Authors:

Andrea Rella

Tutors:

Prof. Paolo Francesco Barbante

Abstract

This study presents a numerical investigation of gas-condensed phase interactions, focusing on the non-linear phenomena of strong evaporation and condensation where classical hydrodynamic descriptions fail. The problem is modeled using the Bhatnagar-Gross-Krook (BGK) approximation of the Boltzmann equation to accurately capture the non-equilibrium behaviors inherent to the Knudsen layer. A robust numerical framework was developed using a Finite Volume Method (FVM) enhanced by the Quadratic Upwind Interpolation scheme to resolve the discontinuities in the velocity distribution function at the phase interface. Through a time-dependent analysis of the flow, the study characterizes the stability and structure of phase transitions. The results confirm the existence of four distinct solution regimes for strong condensation (Types I–IV), governed by the interplay between the Mach number and pressure ratio, which may lead to the formation of upstream-propagating shock waves. Furthermore, the investigation validates the theoretical constraint that steady strong evaporation is strictly limited to the subsonic regime ($M_\infty \leq 1$), with supersonic conditions resulting in unsteady expansion waves. The developed solver successfully reproduces established kinetic theory results, demonstrating its efficacy in analyzing rarefied gas dynamics

Contents

Introduction

The study of phase transitions at the interface between a vapor and its condensed phase (specifically the processes of evaporation and condensation) presents a fundamental problem in the fluid dynamics of both natural and technological processes[1]; a problem that, however, necessitates a departure from classical continuum mechanics. When a gas interacts with its condensed phase in fact, the region immediately adjacent to the interface, known as the Knudsen layer, exhibits behavior that cannot be accurately described by the Navier-Stokes equations alone. This limitation arises from the breakdown of the fundamental assumptions upon which classical hydrodynamics relies.

First, continuum mechanics assumes local thermodynamic equilibrium, implying that molecular collisions are frequent enough to maintain a Maxwellian velocity distribution. However, at the interface of a condensed phase, this assumption is invalid. The velocity distribution function of the gas molecules is inherently discontinuous: molecules leaving the surface typically follow a half-Maxwellian distribution determined by the surface temperature and saturation pressure, while molecules arriving from the bulk gas follow a distribution governed by the flow conditions at infinity.

Second, the characteristic length scale of this relaxation process—the thickness of the Knudsen layer—is comparable to the mean free path of the gas. Consequently, there is no clear separation of scales between microscopic molecular dynamics and macroscopic flow lengths in this region, rendering the continuum hypothesis inapplicable. Furthermore, the Navier–Stokes–Fourier equations rely on linear constitutive relations (Newton's law of viscosity and Fourier's law of heat conduction), which assume small deviations from local equilibrium. In the context of phase transitions, particularly within the Knudsen layer, deviations from equilibrium are significant, and higher-order moments of the velocity distribution become non-negligible.

Therefore, the flow in this region depends heavily on the microscopic velocity distribution function $f(\mathbf{x}, \boldsymbol{\xi}, t)$ rather than solely on macroscopic averages like density or pressure. To accurately model this regime, the behavior of the gas (i.e.the evolution of the distribution function in phase space) must be described by the Boltzmann equation:

$$\frac{\partial f}{\partial t} + \xi_i \frac{\partial f}{\partial x_i} + \frac{\partial (f F_i)}{\partial \xi_i} = \int (f_1' f' - f_1 f) g b \, db \, d\epsilon \, d\boldsymbol{\xi}_1$$

in particular, as it's done in a great part of the literature, due to the difficult handling of the collision term, the analysis is carried out by considering one of its kinetic models. In the present work, the Boltzmann-Krook-Welander (commonly known as the Bhatnagar-Gross-Krook (BGK) model) equation will be considered:

$$\frac{\partial f}{\partial t} + \xi_i \frac{\partial f}{\partial x_i} + \frac{\partial (f F_i)}{\partial \xi_i} = \nu (f^{(0)} - f)$$

together with other simplifying assumptions arising from the specific physics of the problem.

Furthermore, the significance of analyzing the Knudsen layer lies in its function as a boundary condition for the macroscopic regime. In classical hydrodynamics, the no-slip condition assumes that gas velocity and temperature at the surface equal those of the wall. However, in rarefied gas dynamics and phase change processes, significant jumps in temperature and velocity occur across

[1] For instance, condensation rates processes are essential in industrial power cycles, refrigeration cycles and heat exchangers. More recently, these processes have also gained relevance in microfluidics and vacuum technologies, where characteristic length scales become comparable to molecular mean free paths.

the Knudsen layer. The solution to the kinetic equations within this layer provides the necessary relations between the macroscopic variables at the surface and the uniform state at infinity (i.e. at the end of the Knudsen Layer), effectively deriving the correct boundary conditions for the Euler or Navier-Stokes equations.

While linearized kinetic theories have successfully treated weak evaporation and condensation (where deviations from equilibrium are small) [13] they are insufficient for describing "strong" processes. In cases where the flow speed is comparable to the speed of sound, or where strong condensation occurs, linearized theory fails to describe the far field and the formation of non-linear phenomena such as shock waves. Thus, a non-linear kinetic analysis is required to capture the full range of steady and unsteady flow behaviors.

In the following an introduction on the Boltzmann equation is provided in order to familiarize the reader with the equation and give her/him the necessary tools to understand the subsequent study. Then, the specific problem setting is presented and a tailored analysis is carried out; this will then lead to the numerical discretization and solve logic adopted. The key `C++` implementation details will be discussed in a dedicated chapter while the full access to code and examples can be found on the GitHub Repository and Google Drive Folder. Finally the result obtained are discussed in light of the theory.

1 A primer on the Boltzmann equation

Derived in the late 19th century, the Boltzmann equation provides the definitive link between microscopic particle dynamics and macroscopic thermodynamic phenomena. It describes the statistical behaviour of a thermodynamic system not in a state of equilibrium, effectively reconciling the reversible laws of mechanics with the irreversible nature of entropy. As such, it remains the fundamental tool for analyzing transport processes and the approach to equilibrium in dilute systems.

1.1 The Boltzmann equation

This section provides a heuristic derivation of the fundamental transport equation, followed by an analysis of the properties essential for subsequent discussions. While this approach prioritizes physical intuition, a more rigorous derivation can be formally established starting from the BBGKY hierarchy equations [3, 8].

Consider a monatomic gas comprising N molecules confined within a volume V. At any temporal instant, the state of an individual molecule is defined by its position $\mathbf{x} = (x_1, x_2, x_3)$ and velocity $\boldsymbol{\xi} = (\xi_1, \xi_2, \xi_3)$. These six variables constitute a point in the six-dimensional μ-phase space. Consequently, the ensemble of N molecules is represented by N discrete points with coordinates $(\mathbf{x}_n, \boldsymbol{\xi}_n)$ for $n = 1, 2, \ldots, N$.

The macroscopic state of the gas in μ-space is characterized by a one-particle distribution function $f(\mathbf{x}, \boldsymbol{\xi}, t)$. This function represents a density in phase space, specifically defined as the expected number of particles per unit volume of the six-dimensional space $(\mathbf{x}, \boldsymbol{\xi})$; its SI units are accordingly $\mathrm{s}^3\mathrm{m}^{-6}$. Under this framework, the quantity

$$dN = f(\mathbf{x}, \boldsymbol{\xi}, t)\, d\mathbf{x}\, d\boldsymbol{\xi} = f(\mathbf{x}, \boldsymbol{\xi}, t)\, d\mu$$

represents the number of molecules at time t contained within the infinitesimal phase-space volume element $d\mu$, corresponding to positions between $\mathbf{x}$ and $\mathbf{x} + d\mathbf{x}$ and velocities between $\boldsymbol{\xi}$ and $\boldsymbol{\xi} + d\boldsymbol{\xi}$.

By considering a time interval Δt significantly larger than the mean collision time, we may analyze the rate of change of the number of particles within an infinitesimal phase-space volume element:

$$\frac{\Delta N}{\Delta t} = \frac{N(t + \Delta t) - N(t)}{\Delta t} = \frac{f(\mathbf{x} + \Delta\mathbf{x}, \boldsymbol{\xi} + \Delta\boldsymbol{\xi}, t + \Delta t) d\mu(t + \Delta t) - f(\mathbf{x}, \boldsymbol{\xi}, t) d\mu(t)}{\Delta t}$$

where the displacements in the position and velocity coordinates during Δt are expressed as:

$$\Delta\mathbf{x} = \boldsymbol{\xi}\Delta t, \qquad \Delta\boldsymbol{\xi} = \mathbf{F}\Delta t$$

In this context, $\mathbf{F}(\mathbf{x}, \boldsymbol{\xi}, t)$ denotes the external force per unit mass acting upon the molecules. Through a first-order Taylor expansion and the application of Liouville's theorem (noting the invariance of the phase-space volume $d\mu$) [8], the following expression is obtained:

$$\frac{\Delta N}{\Delta t} = \left[\frac{\partial f}{\partial t} + \xi_i \frac{\partial f}{\partial x_i} + \frac{\partial (f F_i)}{\partial \xi_i}\right] d\mu \tag{1.1}$$

To account for intermolecular interactions, an alternative expression for $\Delta N/\Delta t$ can be derived based on the following physical assumptions:

1. Binary Collisions: For a sufficiently rarefied gas, the probability of multi-particle collisions (involving three or more molecules) is negligible compared to the frequency of binary encounters.

2. Dominance of Intermolecular Forces: The influence of external forces during the characteristic collision time τ_c is assumed to be negligible relative to the magnitude of the short-range molecular interaction forces.

3. Molecular Chaos (*Stosszahlansatz*): It is assumed that there is no correlation between the velocities of two molecules prior to their collision, and similarly for their post-collisional states.

4. Spatio-temporal Smoothness: The distribution function $f(\mathbf{x}, \boldsymbol{\xi}, t)$ is assumed to be slowly varying over scales corresponding to the range of intermolecular forces and the mean collision time, yet it remains sensitive to scales on the order of the mean free path and mean free time.

In light of this, to mathematically model the binary interactions, the collision geometry is simplified by considering the relative motion of the particles. In particular in this description (Fig. 1) one molecule is treated as a fixed scattering center at velocity $\boldsymbol{\xi}$, while the partner molecule approaches with a relative velocity $\mathbf{g} = \boldsymbol{\xi}_1 - \boldsymbol{\xi}$. The domain of interaction is visualized as a "collision cylinder" aligned with the vector $\mathbf{g}$. The trajectory of any incoming particle within this cylinder is uniquely defined by two coordinates: the impact parameter[2] (b), the radial distance from the cylinder's axis (O) before the collision occurs, and the azimuthal angle (ε) of the plane of motion. Consequently, the collision probability is determined by the volume swept out by the relative flux $g = |\boldsymbol{\xi}_1 - \boldsymbol{\xi}|$ through the differential cross-section $b\,db\,d\varepsilon$; meaning that each particle that will find itself in the volume (in the phase space $(\mathbf{x}, \boldsymbol{\xi})$) $g\Delta t\, b\, db\, d\varepsilon$ is destined to hit the particle at the center O.

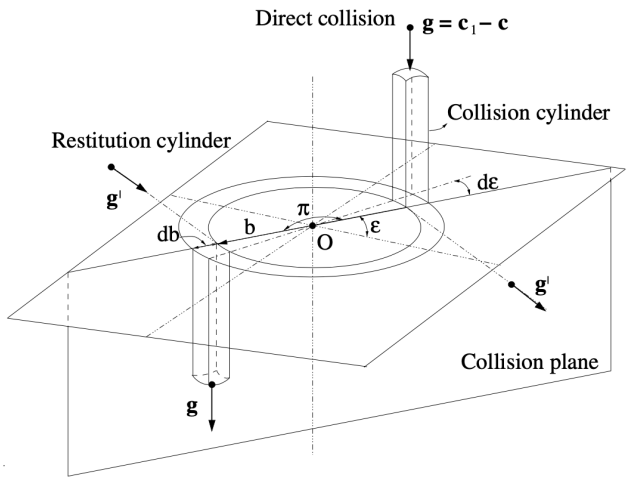


Fig. 1. Geometry of a binary collision [8]

.

With this in mind is easy to see that the number of molecules with velocities within the range $\boldsymbol{\xi}_1$ and $\boldsymbol{\xi}_1 + d\boldsymbol{\xi}_1$ inside the collision cylinder is given by $f(\mathbf{x}, \boldsymbol{\xi}_1, t) d\boldsymbol{\xi}_1\, g\Delta t\, b\, db\, d\varepsilon$. These molecules will collide with all molecules with velocities within the range $\boldsymbol{\xi}$ and $\boldsymbol{\xi} + d\boldsymbol{\xi}$ and which are in the volume element $d\mathbf{x}$ around the point O, i.e., $f(\mathbf{x}, \boldsymbol{\xi}, t)\, d\mathbf{x}\, d\boldsymbol{\xi}$. Hence, the number of collisions, during the time interval Δt, which occur in the volume element $d\mathbf{x}$, reads:

$$f(\mathbf{x}, \boldsymbol{\xi}_1, t) d\boldsymbol{\xi}_1\, g\Delta t\, b\, db\, d\varepsilon\, f(\mathbf{x}, \boldsymbol{\xi}, t)\, d\mathbf{x}\, d\boldsymbol{\xi}$$

[2] In simpler terms, it measures "how off-center" the collision is: a small b implies A nearly head-on collision, resulting in a large deflection angle while a large b results in glancing collision thus in a small deflection.

By dividing by Δt and integrating the resulting formula over all components of the velocity $\xi_i^1 \in [-\infty, \infty]$ $(i = 1, 2, 3)$, over the azimuthal angle $\varepsilon \in [0, 2\pi]$ and over all values of the impact parameter $b \in [0, \infty)$, it follows the total number of collisions per time interval Δt in the μ-phase space that annihilate points with velocity $\boldsymbol{\xi}$ in the volume element $d\mu(t)$ is:

$$\left(\frac{\Delta N}{\Delta t}\right)^{-} = d\mu(t) \int f(\mathbf{x}, \boldsymbol{\xi}_1, t) f(\mathbf{x}, \boldsymbol{\xi}, t)\, g\, b\, db\, d\varepsilon\, d\boldsymbol{\xi}_1$$

However, there exist collisions which create points with velocity $\boldsymbol{\xi}$ in the volume element $d\mu(t)$. Indeed, they result from collisions of molecules with the following characteristics: (i) asymptotic pre-collisional velocities $\boldsymbol{\xi}'$ and $\boldsymbol{\xi}_1'$ (ii) asymptotic post-collisional velocities $\boldsymbol{\xi}$ and $\boldsymbol{\xi}_1$, (iii) apsidal vector $\mathbf{k}' = -\mathbf{k}$, (iv) impact parameter $b' = b$ and (v) azimuthal angle $\varepsilon' = \pi + \varepsilon$. Such collisions are known as restitution collisions, whereas the former are called direct collisions. By analogous considerations one finds that the total number of collisions per time interval Δt, which creates points in the μ-phase space with velocity $\boldsymbol{\xi}$ in the volume element $d\mu(t)$ reads:

$$\left(\frac{\Delta N}{\Delta t}\right)^{+} = d\mu(t) \int f(\mathbf{x}, \xi_{\mathbf{1}}', t), f(\mathbf{x}, \mathbf{c}', t) g b\, db\, d\varepsilon\, d\boldsymbol{\xi}_1$$

By incorporating the assumptions previously outlined, the general balance equation (1.1) may be reformulated into its most recognized kinetic form:

$$\frac{\partial f}{\partial t} + \xi_i \frac{\partial f}{\partial x_i} + \frac{\partial (f F_i)}{\partial \xi_i} = \int (f_1' f' - f_1 f) g b\, db\, d\epsilon\, d\boldsymbol{\xi}_1 \tag{1.2}$$

This expression is the *Boltzmann equation*, a non-linear integro-differential equation that serves as the foundation for describing the evolution of the distribution function within the μ-phase space. Physically, the equation suggests that the total rate of change of f results from two competing effects: a *streaming term* associated with the continuous motion of molecules and the influence of external fields, and a *collision term* representing the discontinuous jumps in velocity space caused by molecular encounters. To simplify the notation in the integral above, we've defined the shorthand for pre- and post-collisional states as:

$$f' \equiv f(\mathbf{x}, \boldsymbol{\xi}', t), \quad f_1' \equiv f(\mathbf{x}, \boldsymbol{\xi}_1', t), \quad f \equiv f(\mathbf{x}, \boldsymbol{\xi}, t), \quad f_1 \equiv f(\mathbf{x}, \boldsymbol{\xi}_1, t)$$

In many practical scenarios, the external force field $\mathbf{F}$ possesses properties that allow for further simplification. If the force is independent of velocity—as with gravity—or if it depends on velocity only through a cross product—as seen with the Coriolis force or the Lorentz force in plasmas—the term $\partial F_i / \partial \xi_i$ vanishes. In such cases, the Boltzmann equation (1.2) takes the simpler form:

$$\frac{\partial f}{\partial t} + \xi_i \frac{\partial f}{\partial x_i} + F_i \frac{\partial f}{\partial \xi_i} = \int (f_1' f' - f_1 f) g b\, db\, d\epsilon\, d\boldsymbol{\xi}_1 \tag{1.3}$$

Throughout the remainder of this analysis, we shall focus on systems where this condition holds. Finally, it is standard practice to denote the integral on the right-hand side as the *collision operator*, defined as:

$$\mathcal{Q}(f, f) = \int (f_1' f' - f_1 f) g b\, db\, d\epsilon\, d\boldsymbol{\xi}_1 \tag{1.4}$$

This operator is central to the theory, as it captures the statistical nature of particle interactions and the relaxation of the gas toward equilibrium.

1.2 Transfer equation - collision invariants

By multiplying the reduced Boltzmann equation (1.3) by an arbitrary weight function $\psi = \psi(\mathbf{x}, \boldsymbol{\xi}, t)$ and integrating over the entire velocity domain, we obtain the general moment equation:

$$\int \psi \frac{\partial f}{\partial t}\, d\boldsymbol{\xi} + \int \psi \xi_i \frac{\partial f}{\partial x_i}\, d\boldsymbol{\xi} + \int \psi F_i \frac{\partial f}{\partial \xi_i}\, d\boldsymbol{\xi} = \int \psi \mathcal{Q}(f, f)\, d\boldsymbol{\xi}$$

To simplify the left-hand side, we consider the behavior of the distribution function at the boundaries of the velocity space. Given that f decreases rapidly as the velocity magnitude increases, the integral of the divergence term can be evaluated via the divergence theorem in velocity space:

$$\int \frac{\partial(\psi F_i f)}{\partial \xi_i}\, d\boldsymbol{\xi} = \oint_{S_\infty} \psi F_i f n_i\, dS = 0$$

By applying the product rule and utilizing the vanishing boundary integral, the general transport of the property ψ is expressed as:

$$\frac{\partial}{\partial t}\int \psi f\, d\boldsymbol{\xi} + \frac{\partial}{\partial x_i}\int \psi \xi_i f\, d\boldsymbol{\xi} - \int f\left[\frac{\partial \psi}{\partial t} + \xi_i \frac{\partial \psi}{\partial x_i} + F_i \frac{\partial \psi}{\partial \xi_i}\right] d\boldsymbol{\xi} = \int \psi \mathcal{Q}(f, f)\, d\boldsymbol{\xi}$$

Through the exploitation of the symmetries inherent in the collision integral—specifically the exchange of indices between colliding particles and the reversibility of the collision process—the right-hand side can be manipulated into a highly symmetric form:

$$\int \psi \mathcal{Q}(f, f)\, d\boldsymbol{\xi} = \frac{1}{4}\int (\psi + \psi_1 - \psi' - \psi_1')(f_1' f' - f_1 f) g b\, db\, d\epsilon\, d\boldsymbol{\xi}_1\, d\boldsymbol{\xi} \tag{1.5}$$

This identity is known as the *Transfer Equation*. A significant result arising from this formulation is that the collision production term vanishes for any distribution function f whenever the function ψ satisfies the condition:

$$\psi + \psi_1 = \psi' + \psi_1' \tag{1.6}$$

A function ψ that fulfills (1.6) is defined as a *summational invariant*. The existence and form of these invariants are governed by the following theorem [8]:

Theorem 1.2.1. *A continuous function $\psi = \psi(\boldsymbol{\xi})$ is a summational invariant if and only if it takes the form:*

$$\psi(\boldsymbol{\xi}) = a + \mathbf{b} \cdot \boldsymbol{\xi} + d\xi^2 \tag{1.7}$$

where a and d are scalar functions and $\mathbf{b}$ is a vectorial function, all of which are independent of the velocity $\boldsymbol{\xi}$.

Consequently, any such $\psi(\boldsymbol{\xi})$ can be constructed as a linear combination of five *elementary collision invariants*:

$$\psi_0 = 1, \quad (\psi_1, \psi_2, \psi_3) = \boldsymbol{\xi}, \quad \psi_4 = \xi^2$$

As will be discussed shortly, these five quantities correspond to the physical conservation of mass, momentum, and energy during molecular interactions.

1.3 Connection between macroscopic and microscopic

In the framework of kinetic theory, the macroscopic state of a gas is fundamentally defined by moments of the distribution function $f(\mathbf{x}, \boldsymbol{\xi}, t)$. By considering the microscopic properties of individual molecules—specifically their mass m, linear momentum $m\xi_i$, and kinetic energy

$m\xi^2/2$—one can derive the local mass density ρ, momentum density ρv_i, and total energy density ρu through the following integral relations:

$$\rho(\mathbf{x},t) = \int m f(\mathbf{x},\boldsymbol{\xi},t)\, d\boldsymbol{\xi} \qquad \text{(Mass density)} \tag{1.8}$$

$$\rho v_i(\mathbf{x},t) = \int m\xi_i f(\mathbf{x},\boldsymbol{\xi},t)\, d\boldsymbol{\xi} \qquad \text{(Momentum density)} \tag{1.9}$$

$$\rho u(\mathbf{x},t) = \frac{1}{2}\int m\xi^2 f(\mathbf{x},\boldsymbol{\xi},t)\, d\boldsymbol{\xi} \qquad \text{(Energy density)} \tag{1.10}$$

The quantity $\mathbf{v} = \frac{1}{\rho}\int m\boldsymbol{\xi} f\, d\boldsymbol{\xi}$ represents the *macroscopic flow velocity*, which corresponds to the observable bulk motion of the fluid. On a microscopic scale, however, individual molecules exhibit stochastic deviations from this mean motion. This fluctuation is captured by the *peculiar velocity* $\boldsymbol{\xi}$, defined such that:

$$\boldsymbol{\xi} = \mathbf{v} + \mathbf{C}$$

By substituting this decomposition into the expression for the total energy density (1.10), the energy may be partitioned as follows:

$$\rho u = \frac{1}{2}\rho v^2 + \rho\varepsilon, \qquad \text{where} \qquad \rho\varepsilon(\mathbf{x},t) = \frac{1}{2}\int mC^2 f(\mathbf{x},\boldsymbol{\xi},t)\, d\boldsymbol{\xi}$$

This decomposition demonstrates that the total energy density of the gas is the sum of its macroscopic *kinetic energy density*, $\rho v^2/2$, and its *internal energy density*, $\rho\varepsilon$, the latter of which accounts for the thermal motion relative to the mean flow.

Furthermore, the transport of physical properties is characterized by the flux of specific quantities across a surface. If G denotes the density of a property in μ-phase space, the associated flux through a surface element dS—representing the amount of that property crossing the surface per unit area and unit time—is given by:

$$\Phi_G = \frac{\int G\xi_n\, dt\, dS\, d\boldsymbol{\xi}}{dt\, dS} = \int G\xi_n\, d\boldsymbol{\xi}$$

where ξ_n is the projection of the molecular velocity $\boldsymbol{\xi}$ onto the unit normal of the surface element dS. This definition allows for the systematic derivation of the conservation laws by considering the flux of the elementary collision invariants.

An analysis of particular interest involves the momentum flux, specifically the transport of the j-th component of momentum in the i-th direction. This flux is expressed as:

$$\int \xi_i(m\xi_j f)\, d\boldsymbol{\xi} = m\int \xi_i\xi_j f\, d\boldsymbol{\xi} = \rho v_i v_j + m\int C_i C_j f\, d\boldsymbol{\xi}$$

The first term on the right-hand side represents the macroscopic convective momentum flow, while the second term accounts for the "hidden" momentum transport resulting from the stochastic thermal motion of the particles. This latter contribution defines the *pressure tensor*, P_{ij}. Its negative, $T_{ij} = -P_{ij}$, is identified as the *stress tensor*[3].

The *hydrostatic pressure* p is defined as the mean of the normal stresses, corresponding to the trace of the pressure tensor:

$$p(\mathbf{x},t) = \frac{1}{3}P_{ii} = \frac{1}{3}\int mC^2 f(\mathbf{x},\boldsymbol{\xi},t)\, d\boldsymbol{\xi}$$

[3]This identification is justified by the fact that $-P_{ij}$ plays the same functional role in the macroscopic transport equations derived from the Boltzmann equation as the stress tensor does in classical continuum mechanics [3].

Given that $P_{ii} = 2\rho\varepsilon$, we recover the caloric *equation of state* for a monatomic gas:

$$p = \frac{2}{3}\rho\varepsilon$$

By comparing this result with the ideal gas law, $p = \rho RT = nkT$, we deduce the relationship $\varepsilon = \frac{3}{2}RT$. This allow us to provide a kinetic definition for the absolute temperature T in terms of the distribution function:

$$T(\mathbf{x}, t) = \frac{p}{nk} = \frac{2}{3}\frac{m}{k}\varepsilon = \frac{m}{3nk}\int C^2 f(\mathbf{x}, \boldsymbol{\xi}, t)\, d\boldsymbol{\xi} \tag{1.11}$$

In a similar manner, we examine the total energy flux in the i-th direction:

$$\frac{1}{2}\int m\xi_i\xi^2 f\, d\boldsymbol{\xi} = v_i\left(\frac{1}{2}\rho v^2 + \rho\varepsilon\right) + v_j P_{ij} + \frac{1}{2}\int mC_i C^2 f\, d\boldsymbol{\xi}$$

This flux is composed of three distinct physical mechanisms: the convective transport of total energy, the rate of macroscopic work performed by the internal stresses, and a residual energy flow termed the *heat flux vector* (or *non-convective energy flux*):

$$q_i = \frac{1}{2}\int mC_i C^2 f\, d\boldsymbol{\xi}$$

Consistent with our treatment of the stress tensor, q_i is identified as the heat flux because it assumes the identical role of the thermal conduction vector within the macroscopic conservation equations.

To complete the bridge between the microscopic and macroscopic descriptions, we now derive the five differential equations satisfied by the aforementioned macroscopic variables as a direct mathematical consequence of the Boltzmann equation. These relations are formally referred to as the *Conservation Equations*, as they provide the physical representation of the conservation of mass, momentum, and energy.

The derivation is performed by multiplying the Boltzmann equation (1.3) by the mass-weighted elementary collision invariants $m\psi_\alpha$ and integrating over the entire velocity space[4]. Recalling that the integral of the collision operator vanishes for these invariants, and utilizing the macroscopic definitions established previously, one obtains the following transport equations [8]:

- For $\alpha = 0$:

$$\frac{\partial\rho}{\partial t} + \frac{\partial}{\partial x_i}(\rho v_i) = 0 \qquad \text{(Continuity equation)} \tag{1.12}$$

- For $\alpha = 1, 2, 3$:

$$\frac{\partial}{\partial t}(\rho v_j) + \frac{\partial}{\partial x_i}(\rho v_i v_j + P_{ij}) = \rho F_j \qquad \text{(Momentum conservation)} \tag{1.13}$$

- For $\alpha = 4$:

$$\frac{\partial}{\partial t}\left[\rho\left(\frac{1}{2}v^2 + \varepsilon\right)\right] + \frac{\partial}{\partial x_i}\left[\rho v_i\left(\frac{1}{2}v^2 + \varepsilon\right) + P_{ij}v_j + q_i\right] = \rho F_i v_i \qquad \text{(Energy conservation)} \tag{1.14}$$

[4]It is assumed that the distribution function decays sufficiently fast such that $\lim_{\xi^2\to\infty}\psi_\alpha f = 0$, ensuring the convergence of the integrals.

By means of the kinetic approach, we have recovered the fundamental equations of continuum mechanics. However, in their current form, these five equations involve thirteen unknown macroscopic quantities (ρ, v_i, P_{ij}, and q_i), thus constituting an open system. To achieve a solution, the system must be closed through constitutive models that relate the higher-order moments (stress and heat flux) to the lower-order variables. In fluid dynamics, two primary closure models are widely recognized:

- **Euler (Ideal) Fluid:** Assuming the gas remains in local equilibrium, the effects of viscosity and heat conduction are neglected:

$$P_{ij} = p\delta_{ij}, \qquad q_i = 0 \tag{1.15}$$

- **Navier-Stokes-Fourier Fluid:** This model accounts for dissipation in viscous and thermally conducting fluids:

$$\begin{cases} P_{ij} = p\delta_{ij} - \mu\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right) - \lambda\frac{\partial v_k}{\partial x_k}\delta_{ij} \\ q_i = -\kappa\frac{\partial T}{\partial x_i} \end{cases} \tag{1.16}$$

1.4 Equilibrium states

The right-hand side of the Boltzmann equation, the collision operator $\mathcal{Q}(f, f)$, accounts for the net rate of change in the number of molecules within a μ-phase space volume element due to molecular encounters. In a state of thermodynamic equilibrium, this net exchange must necessarily vanish:

$$\mathcal{Q}(f, f)\big|_E = 0$$

where the subscript E denotes the evaluation of the distribution function at equilibrium. To characterize these states, we rely on a fundamental result in kinetic theory [3]:

Theorem 1.4.1 (**Boltzmann Inequality**). *The collision operator satisfies the following integral inequality:*

$$\int \mathcal{Q}(f, f)\log(f)\, d\boldsymbol{\xi} \leq 0 \tag{1.17}$$

Furthermore, the equality holds if and only if $\log(f)$ *is a summational invariant, which implies that the distribution function must take the exponential form* $f = \exp(a + \mathbf{b}\cdot\boldsymbol{\xi} + d\xi^2)$.

The connection between the vanishing of the collision operator and the functional form of f is thus established: $\mathcal{Q}(f, f)\big|_E = 0$ implies that $f\big|_E$ must be a summational invariant. For the distribution to be physically integrable over the velocity domain, the coefficient d must be negative; we therefore define $d = -\beta$ with $\beta > 0$. By further reparameterizing the vector $\mathbf{b}$ as $2\beta\mathbf{w}$ and performing algebraic manipulations, the equilibrium distribution is expressed as:

$$f\big|_E = Ae^{-\beta|\boldsymbol{\xi}-\mathbf{w}|^2} \qquad \text{where} \quad A = e^{a+\beta|\mathbf{w}|^2}$$

This functional form is recognized as the *Maxwellian distribution*. The parameters A, β, and $\mathbf{w}$ are uniquely determined by requiring the moments of this distribution to match the macroscopic mass density, momentum, and temperature as defined in (1.8), (1.9), and (1.11). This leads to the following properties [3]:

Proposition 1.4.2. *A gas characterized by a Maxwellian distribution satisfies the following relations:*

1. $\mathbf{w} = \mathbf{v}$, *where* $\mathbf{v}$ *is the macroscopic flow velocity;*

2. $\beta = \frac{1}{2RT} = \frac{m}{2kT}$;

3. $A = \rho(2\pi RT)^{-\frac{3}{2}} = n\left(\frac{m}{2\pi kT}\right)^{\frac{3}{2}}$.

Under these conditions, the pressure tensor reduces to $P_{ij} = p\delta_{ij}$ and the heat flux vector vanishes, $q_i = 0$. Thus, a gas in Maxwellian equilibrium strictly adheres to the constitutive relations of an Euler (ideal) fluid.

The normalized equilibrium distribution, often denoted as $f^{(0)}$, is therefore given by:

$$f^{(0)} = n\left(\frac{m}{2\pi kT}\right)^{\frac{3}{2}} e^{-\frac{m}{2kT}|\boldsymbol{\xi}-\mathbf{v}|^2} \tag{1.18}$$

Physically, $f^{(0)}$ represents a state of detailed balance, where the rate of particles being scattered into a specific velocity class is exactly compensated by the rate of particles being scattered out of it.

1.5 $\mathcal{H}$-theorem

In this section, we examine a fundamental result concerning the irreversible nature of gaseous processes as described by the kinetic theory. We provide here a heuristic derivation of the $\mathcal{H}$-theorem; for a more rigorous and generalized treatment, the reader is referred to [8].

Consider the Boltzmann equation in its standard form:

$$\frac{\partial f}{\partial t} + \xi_i \frac{\partial f}{\partial x_i} + F_i \frac{\partial f}{\partial \xi_i} = \mathcal{Q}(f, f)$$

We define the $\mathcal{H}$-functional, a quantity representative of the statistical state of the gas, as:

$$\mathcal{H}(t) = \iint f \log(f)\, d\boldsymbol{\xi}\, d\mathbf{x} \tag{1.19}$$

The temporal evolution of this functional is obtained by differentiating under the integral sign:

$$\frac{d\mathcal{H}}{dt} = \iint (1 + \log(f)) \frac{\partial f}{\partial t}\, d\boldsymbol{\xi}\, d\mathbf{x} = \iint \log(f) \frac{\partial f}{\partial t}\, d\boldsymbol{\xi}\, d\mathbf{x}$$

The simplification in the final step arises from the principle of mass conservation: since the total number of molecules $N = \iint f\, d\boldsymbol{\xi}\, d\mathbf{x}$ is constant in time, the integral of the partial derivative $\frac{\partial f}{\partial t}$ over the entire phase space vanishes. By substituting the expression for $\frac{\partial f}{\partial t}$ from the Boltzmann equation, we obtain:

$$\frac{d\mathcal{H}}{dt} = \iint \log(f) \left[-\xi_i \frac{\partial f}{\partial x_i} - F_i \frac{\partial f}{\partial \xi_i} + \mathcal{Q}(f, f) \right] d\boldsymbol{\xi}\, d\mathbf{x}$$

Upon applying the divergence theorem and assuming that the distribution function and its fluxes vanish at the boundaries of the spatial and velocity domains, the transport terms (convective and forced) contribute nothing to the total integral. Consequently, the rate of change of $\mathcal{H}$ is governed solely by the collision operator:

$$\frac{d\mathcal{H}}{dt} = \iint \log(f) \mathcal{Q}(f, f)\, d\boldsymbol{\xi}\, d\mathbf{x}$$

Invoking the Boltzmann Inequality (Theorem 1.4.1), we arrive at the following theorem:

Theorem 1.5.1 ($\mathcal{H}$**-Theorem**). *For a gas evolving according to the Boltzmann equation, the functional $\mathcal{H}$ satisfies:*

$$\frac{d\mathcal{H}}{dt} \leq 0$$

where the equality $\frac{d\mathcal{H}}{dt} = 0$ holds if and only if the distribution function f is a Maxwellian.

This result provides a microscopic basis for physical irreversibility[5]. Furthermore, it can be formally demonstrated [3] that the H-functional is related to the thermodynamic entropy $\mathcal{S}$ via the relation:

$$\mathcal{H} = -\frac{\mathcal{S}}{k}$$

where k is the Boltzmann constant. In this sense, the $\mathcal{H}$-theorem constitutes a statistical mechanical derivation of the Second Law of Thermodynamics, suggesting that the increase in entropy is a direct consequence of molecular collisions driving the system toward a Maxwellian equilibrium.

1.6 The BGK Model

A primary challenge in the practical application of the Boltzmann equation stems from the complexity of the collision operator, $\mathcal{Q}(f, f)$. Its non-linear structure, resulting from the product of distribution functions within the fivefold integral, renders analytical and numerical solutions exceedingly difficult:

$$\mathcal{Q}(f, f) = \int (f_1' f' - f_1 f) g b \, db \, d\epsilon \, d\boldsymbol{\xi}_1$$

To circumvent this difficulty while preserving the essential physics of particle interactions, various simplified expressions for the collision term—collectively known as *kinetic models*—have been proposed. One of the most foundational models is the BGK model, developed independently by Bhatnagar, Gross, and Krook, as well as by Welander [2].

Any simplified collision model $\mathcal{J}(f)$ must adhere to the fundamental physical constraints satisfied by the original operator $\mathcal{Q}(f, f)$. These include:

1. **Conservation Laws:** For all summational invariants $\psi \in \{m, m\xi_i, m\xi^2/2\}$, the model must satisfy the vanishing integral condition:

$$\int \psi \mathcal{Q}(f, f) \, d\boldsymbol{\xi} = 0 \quad \longrightarrow \quad \int \psi \mathcal{J}(f) \, d\boldsymbol{\xi} = 0$$

   ensuring the conservation of mass, momentum, and energy.

2. **Entropy Production:** The model must satisfy the requirement of irreversibility and the tendency toward equilibrium, consistent with the $\mathcal{H}$-theorem:

$$\int \log(f) \mathcal{Q}(f, f) \, d\boldsymbol{\xi} \leq 0 \quad \longrightarrow \quad \int \log(f) \mathcal{J}(f) \, d\boldsymbol{\xi} \leq 0$$

In the BGK approximation, the collision operator is replaced by a linear relaxation term:

$$\mathcal{J}(f) = \nu(f^{(0)} - f) \tag{1.20}$$

where $f^{(0)}$ is the local Maxwellian distribution function defined in (1.18), and ν denotes an effective collision frequency. Under this approximation, the Boltzmann equation (1.3) simplifies to:

$$\frac{\partial f}{\partial t} + \xi_i \frac{\partial f}{\partial x_i} + F_i \frac{\partial f}{\partial \xi_i} = \nu(f^{(0)} - f)$$

[5]This conclusion was historically the subject of intense debate, most notably through the criticisms of Loschmidt and Zermelo, who pointed out the apparent paradox between the irreversibility of the Boltzmann equation and the time-reversible nature of the underlying Newtonian mechanics.

It can be formally demonstrated that this kinetic model satisfies the aforementioned physical properties, providing a robust yet manageable framework for the study of non-equilibrium gas dynamics [8].

2 Modelisation of the problem

We consider a semi-infinite domain filled with a rarefied gas, bounded by its planar condensed phase maintained at a uniform and constant surface temperature T_w [1, 10, 11]. The gas occupies the region $X_1 > 0$, where X_i denotes the Cartesian coordinate system. At time $t = 0$, a uniform equilibrium flow with pressure p_∞, temperature T_∞, and velocity $(v_\infty, 0, 0)$ with $v_\infty \in \mathbb{R}$ (the sign will depend on the problem considered), oriented perpendicularly toward the condensed phase, impinges on the boundary (Fig. 2). This uniform flow is continuously supplied from infinity. This study addresses both the unsteady condensation and evaporation occurring on the condensed phase and the temporal evolution of the disturbance generated by the interaction between the incident flow and the condensed boundary. Through a time-dependent analysis, we further elucidate the characteristics of steady gas flows condensing on a planar condensed surface, focusing on the transition region between the condensed phase and the far-field equilibrium state, as well as on the range of far-field and boundary variables that ensure the existence of a steady solution.

The analysis is conducted under the following assumptions: (i) the dynamics of the gas are governed by the Boltzmann–Krook–Welander equation [2, 8], and (ii) the molecules leaving the condensed phase follow the stationary Maxwellian distribution associated with the saturated gas at the surface temperature of the condensed phase.

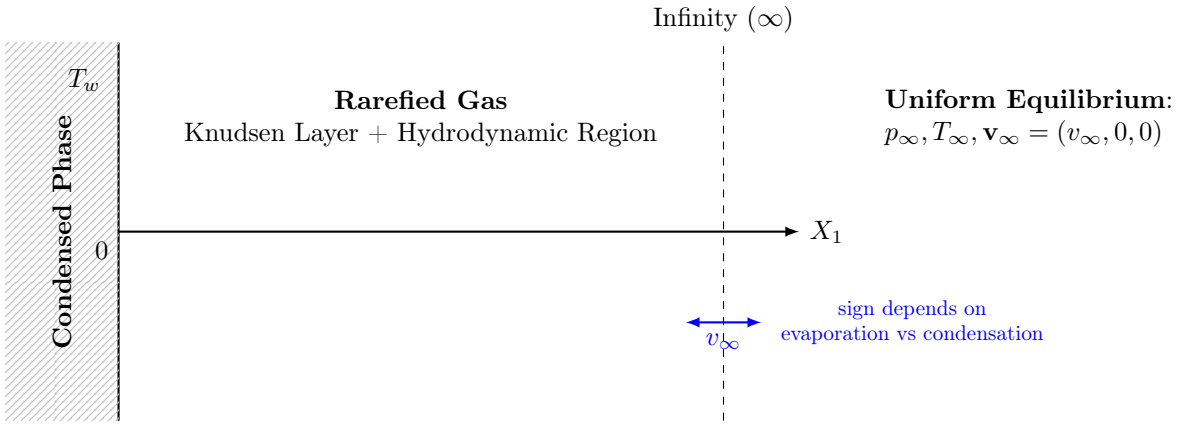


Fig. 2. Schematic representation of the problem configuration. A semi-infinite domain filled with a rarefied gas ($X_1 > 0$) is bounded by a planar condensed phase at $X_1 = 0$ with temperature T_w. The gas at infinity is in a uniform equilibrium state characterized by p_∞, T_∞, and velocity v_∞.

In this "one-dimensional" spatial configuration, the Bhatnagar-Gross-Krook model equation is reduced (due to symmetric arguments) to:

$$\frac{\partial f}{\partial t} + \xi_1 \frac{\partial f}{\partial X_1} = A_c \rho (f^{(0)} - f), \tag{2.1}$$

$$f^{(0)} = \frac{\rho}{(2\pi RT)^{3/2}} \exp\left(-\frac{(\xi_1 - v_1)^2 + \xi_2^2 + \xi_3^2}{2RT} \right), \tag{2.2}$$

$$\rho = \int f \, d\xi_1 \, d\xi_2 \, d\xi_3, \tag{2.3}$$

$$v_1 = \frac{1}{\rho} \int \xi_1 f \, d\xi_1 \, d\xi_2 \, d\xi_3, \tag{2.4}$$

$$T = \frac{1}{3R\rho} \int \left[(\xi_1 - v_1)^2 + \xi_2^2 + \xi_3^2\right] f \, d\xi_1 \, d\xi_2 \, d\xi_3, \tag{2.5}$$

Where $f(t, X_1, \xi_1)$ is the velocity distribution function[6] of the gas molecules, ξ_i is the molecular velocity, $\rho(t, X_1)$ is the gas density, $v_i(t, X_1) = [v_1(t, X_1), 0, 0]$ is the gas flow velocity, $T(t, X_1)$ is the gas temperature, R is the specific gas constant, $A_c\rho$ is the collision frequency of the gas molecules with A_c a constant, and the range of integration of Eqs. (2.3) – (2.5) is the whole space $\boldsymbol{\xi}$ i.e. $\mathbb{R}^3$. The initial condition for Eq (2.1) is

$$f = \frac{\rho_\infty}{(2\pi R T_\infty)^{3/2}} \exp\left(-\frac{(\xi_1 - v_\infty)^2 + \xi_2^2 + \xi_3^2}{2RT_\infty}\right), \quad \text{at } t = 0, \tag{2.6}$$

where $\rho_\infty = (RT_\infty)^{-1} p_\infty$. The boundary condition is given by

$$f = \left[\frac{\rho_w}{(2\pi R T_w)^{3/2}}\right] \exp\left(-\frac{\xi_1^2 + \xi_2^2 + \xi_3^2}{2RT_w}\right), \quad \text{for } \xi_1 > 0, \quad \text{at } X_1 = 0, \tag{2.7}$$

$$f = \frac{\rho_\infty}{(2\pi R T_\infty)^{3/2}} \exp\left(-\frac{(\xi_1 - v_\infty)^2 + \xi_2^2 + \xi_3^2}{2RT_\infty}\right), \quad \text{for } \xi_1 < 0, \quad \text{at } X_1 = \infty. \tag{2.8}$$

where ρ_w is the saturation gas density at temperature T_w.

A desired thing would be now to reduce the dimensionality keeping only the characteristic direction of the problem; to do we proceed by eliminating the molecular velocity components ξ_2 and ξ_3 along the condensed phase from the initial-boundary-value problem —Eqs. (2.1), (2.6), (2.7), and (2.8))— by multiplying them by 1 or $\xi_2^2 + \xi_3^2$ and integrating over the whole space of ξ_2 and ξ_3. Both these integrations are needed to retrieve at the end the macroscopic quantities of interest: $\overline{\rho}$, $\overline{v}$ and $\overline{T}$.

The process of multiplying Eq. (2.1) by 1 and integrating over ξ_2 and ξ_3 gives:

$$\frac{\partial u}{\partial t} + \xi_1 \frac{\partial u}{\partial X_1} = A_c \rho \left(\frac{\rho}{\sqrt{2\pi RT}} e^{-\frac{(\xi_1 - v_1)^2}{2RT}} - u\right)$$

with

$$u = \int f \, d\xi_2 \, d\xi_3$$

Another natural thing to do is to adimensionalize the equation. In such way the characteristic constants of the problem are combined into a smaller set of dimensionless groups and the problem is effectively rendered scale invariant. The new set of variables is:

$$\begin{aligned} &\overline{\rho} = \rho_w^{-1}\rho, \qquad \overline{v} = (2RT_w)^{-1/2} v_1, \qquad \overline{T} = T_w^{-1} T \\ &\overline{t} = (2RT_w)^{1/2} l_w^{-1} t, \qquad x = l_w^{-1} X_1, \qquad \zeta = (2RT_w)^{-1/2} \xi_1 \end{aligned} \tag{2.9}$$

where $l_w = (8RT_w/\pi)^{1/2}(A_c\rho_w)^{-1}$ is the mean free path of the uniform saturated equilibrium state at rest at temperature T_w. We observe that in such way the operators can be equivalently identified with:

$$\frac{d}{dt} \longleftrightarrow \frac{\sqrt{2RT_w}}{l_w} \frac{d}{d\overline{t}} \qquad\qquad \frac{d}{dX_1} \longleftrightarrow \frac{1}{l_w} \frac{d}{dx}$$

and so, after some simple simplifications, the equation becomes:

$$\frac{\partial g}{\partial \overline{t}} + \zeta \frac{\partial g}{\partial x} = \frac{2}{\sqrt{\pi}} \overline{\rho}(G - g)$$

[6] With respect to Sec. 1 here $f(t, X_1, \xi_1)$ represents the expected *mass* of particles per unit volume $dX_1 \, d\xi_1$ i.e., if we call $\tilde{f}$ the distribution function as introduced in Sec 1 we have $f = m\tilde{f}$

with

$$g = \frac{\sqrt{2RT_w}}{\rho_w} u \qquad G = \frac{1}{\sqrt{\pi}} \bar{\rho} \overline{T}^{-1/2} \exp\left(-\frac{(\zeta - \bar{v})^2}{\overline{T}} \right)$$

What was just shown can be repeated analogously for the process of multiplying by $\xi_2^2 + \xi_3^2$ and integrating over (ξ_2, ξ_3) thus obtaining from Eqs. (2.1) – (2.5) the following system of equations

$$\frac{\partial g}{\partial \bar{t}} + \zeta \frac{\partial g}{\partial x} = \frac{2}{\sqrt{\pi}} \overline{\rho} (G - g) \tag{2.10a}$$

$$\frac{\partial h}{\partial \bar{t}} + \zeta \frac{\partial h}{\partial x} = \frac{2}{\sqrt{\pi}} \overline{\rho} (H - h) \tag{2.10b}$$

$$G = \frac{1}{\sqrt{\pi}} \bar{\rho} \overline{T}^{-1/2} \exp\left(-\frac{(\zeta - \bar{v})^2}{\overline{T}} \right) \tag{2.11a}$$

$$H = \frac{1}{\sqrt{\pi}} \bar{\rho} \overline{T}^{1/2} \exp\left(-\frac{(\zeta - \bar{v})^2}{\overline{T}} \right) \tag{2.11b}$$

$$\bar{\rho} = \int_{-\infty}^{\infty} g \, d\zeta \tag{2.12}$$

$$\bar{v} = \bar{\rho}^{-1} \int_{-\infty}^{\infty} \zeta g \, d\zeta \tag{2.13}$$

$$\bar{T} = \frac{2}{3} \bar{\rho}^{-1} \left(\int_{-\infty}^{\infty} (\zeta - \bar{v})^2 g \, d\zeta + \int_{-\infty}^{\infty} h \, d\zeta \right) \tag{2.14}$$

where we define

$$g = (2RT_w)^{1/2} \rho_w^{-1} \iint_{-\infty}^{\infty} f \, d\xi_2 \, d\xi_3 \tag{2.15}$$

$$h = (2RT_w)^{-1/2} \rho_w^{-1} \iint_{-\infty}^{\infty} (\xi_2^2 + \xi_3^2) f \, d\xi_2 \, d\xi_3 \tag{2.16}$$

The initial and boundary conditions follow a analogous fate and so, from Eqs. (2.6) – (2.8), we obtain the subsequent conditions for Eqs. (2.10a) and (2.10b):

$$g = (\sqrt{\pi})^{-1} \exp(-\zeta^2) \tag{2.17a}$$

$$h = (\sqrt{\pi})^{-1} \exp(-\zeta^2) \tag{2.17b}$$

for $\zeta > 0$, at $x = 0$

$$g = \frac{1}{\sqrt{\pi}} \frac{p_\infty}{p_w} \left(\frac{T_\infty}{T_w} \right)^{-3/2} \exp\left\{ -\left[\zeta - \left(\frac{5T_\infty}{6T_w} \right)^{1/2} M_\infty \right]^2 \left(\frac{T_\infty}{T_w} \right)^{-1} \right\} \tag{2.18a}$$

$$h = \frac{1}{\sqrt{\pi}} \frac{p_\infty}{p_w} \left(\frac{T_\infty}{T_w} \right)^{-5/2} \exp\left\{ -\left[\zeta - \left(\frac{5T_\infty}{6T_w} \right)^{1/2} M_\infty \right]^2 \left(\frac{T_\infty}{T_w} \right)^{-1} \right\} \tag{2.18b}$$

for $\zeta < 0$, at $x = \infty$ (boundary condition) or $\forall \zeta$ at $\bar{t} = 0$ (initial condition). $M_\infty = v_\infty [(5/3) R T_\infty]^{-1/2}$ is the Mach number at the initial state (or at infinity) and $p_w = R \rho_w T_w$ is the saturation gas pressure at temperature T_w.

The problem is characterized by three dimensionless parameters: M_∞, p_∞ / p_w and T_∞ / T_w. To achieve our purpose, we solve the system [Eqs. (2.10a) – (2.14) and (2.17a) – (2.18b)] numerically for a large number of sets of M_∞, p_∞ / p_w and T_∞ / T_w.

The saturation gas pressure p_w is a function of T_w given by the Clausius-Clapeyron relation [9]. In the following analysis, however, the relation between p_w and T_w is never used, and thus p_w and T_w are chosen freely in the results. The ρ_w is determined by the two as $\rho_w = p_w / (RT_w)$.

2.1 Numerical Analysis

We analyze the initial-boundary-value problem [Eqs. (2.10a) – (2.14) and (2.17a) – (2.18b)] through a finite volume method which is described in this section.

In the actual computation we consider the problem in a finite region $0 \leq x \leq D$ and impose the condition at infinity, Eqs. (2.18a) – (2.18b), at $x = D$ i.e.:

$$g = \frac{1}{\sqrt{\pi}} \frac{p_\infty}{p_w} \left(\frac{T_\infty}{T_w}\right)^{-3/2} \exp\left\{ - \left[\zeta - \left(\frac{5T_\infty}{6T_w}\right)^{1/2} M_\infty \right]^2 \left(\frac{T_\infty}{T_w}\right)^{-1} \right\} \tag{2.19a}$$

$$h = \frac{1}{\sqrt{\pi}} \frac{p_\infty}{p_w} \left(\frac{T_\infty}{T_w}\right)^{-5/2} \exp\left\{ - \left[\zeta - \left(\frac{5T_\infty}{6T_w}\right)^{1/2} M_\infty \right]^2 \left(\frac{T_\infty}{T_w}\right)^{-1} \right\} \tag{2.19b}$$

for $\zeta < 0$, at $x = D$. Here D is chosen in such a way that the disturbance does not reach the edge $x = D$ within the time interval of our interest. Furthermore, we limit the molecular velocity space to a finite range $-Z \leq \zeta \leq Z$. The quantities $|g|$ and $|h|$ must be negligibly small for $\zeta > |Z|$. This is confirmed from the result of computation.

We adopt a finite volume scheme with cell centered computational points and Quadratic Upwind Interpolation [5] (QUICK) (cf. Appendix A.1). It's worth noting that the following choice of discretization of the domain in the $\mu-$space is given by the fact that Eqs. (2.10a) – (2.10b), after a discretization of the ζ space, are all independent with respect to $\zeta^{(j)}$. We then define:

$$\begin{aligned} &x^{(i)} \quad i = 0, 1, \ldots, N \quad x^{(0)} = 0, \quad x^{(N)} = D \\ &x^{(i+1/2)} \quad i = 0, 1, \ldots, N-1 \quad x^{(i+1/2)} = (x^{(i+1)} + x^{(i)})/2 \\ &x^{(-1/2)} \equiv x^{(0)} \qquad x^{(N+1/2)} \equiv x^{(N)} \end{aligned} \tag{2.20}$$

the computational points in the x space and the volume boundary respectively. Moreover let

$$\zeta^{(j)} \quad j = -\overline{N}, \ldots, \overline{N} \quad \zeta^{(0)} = 0, \quad \zeta^{(\pm\overline{N})} = \pm Z, \quad \zeta^{(-j)} = -\zeta^{(j)} \tag{2.21}$$

be the lattice points in the ζ space. Specifically we adopt the following lattice systems:

$$\begin{aligned} x^{(i)} &= d_1 i + \tfrac{1}{4}(d_2 - d_1)\left(i^4/N_0^3\right) \quad (i = 0, 1, \ldots, N_0) \\ x^{(i)} &= x^{(N_0)} + d_2(i - N_0) \quad (i = N_0 + 1, \ldots, N) \end{aligned}$$

for x space (If $N \leq N_0$, we use only the first equation), and

$$\zeta^{(j)} = a_1 j + a_2 j^3 \quad (j = -\overline{N}, -\overline{N} + 1, \ldots, \overline{N})$$

for ζ space. Integers N, N_0, and $\overline{N}$ and constants d_1, d_2, a_1, and a_2 are chosen properly depending on the solution regime.

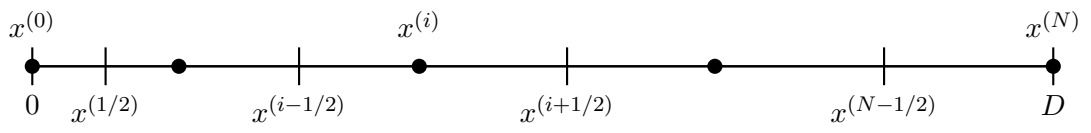


Fig. 3. Finite Volume space grid with refined spacing. Dark circles indicate to the computational points while vertical lines the volume boundaries.

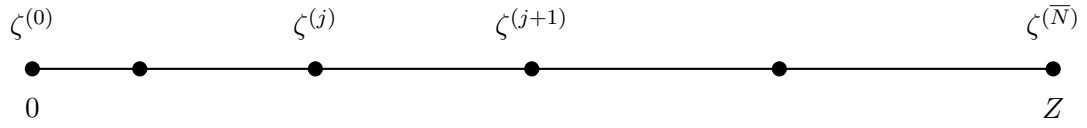


Fig. 4. Positive-Velocity space discretization. The computational points are highlighted by the dark circles.

Finally we consider a time discretization grid as the following:

$$t^k = t_0 + k\Delta t \quad k = 0, \dots, \tilde{N}; \qquad t_0 = 0$$

with $\tilde{N} = \min\{\text{max iterations}, k : err < \text{tol}\}$.

We denote the values at the points $(x^{(i)}, \zeta^{(j)})$ and time t^k of the reduced functions g, h, the macroscopic variables $\overline{\rho}, \overline{v}, \overline{T}$ and the reduced local Maxwellians G, H (which are also functions of $\bar{t}$, x and ζ), with a superscript and subscripts, i.e.,

$$\begin{aligned} \phi_{ij}^k &= \phi(k\Delta t, x^{(i)}, \zeta^{(j)}) \qquad (\phi = g \text{ or } h) \\ \Phi_{ij}^k &= \Phi(k\Delta t, x^{(i)}, \zeta^{(j)}) \qquad (\Phi = G \text{ or } H) \\ S_{ij}^k &= S(k\Delta t, x^{(i)}) \qquad (S = \overline{\rho}, \overline{v} \text{ or } \overline{T}) \end{aligned} \tag{2.22}$$

We start with the μ-space discretization and consider the LHS of (2.10a):

$$\begin{aligned} &= \int_{x^{(i-1/2)}}^{x^{(i+1/2)}} \left(\frac{\partial g_j}{\partial t} + \zeta^{(j)} \frac{\partial g_j}{\partial x} \right) dx \\ &= \frac{d}{dt} \int_{x^{(i-1/2)}}^{x^{(i+1/2)}} g_j \, dx + \zeta^{(j)} \int_{x^{(i-1/2)}}^{x^{(i+1/2)}} \frac{\partial g_j}{\partial x} \, dx \\ &= \frac{d}{dt} \int_{x^{(i-1/2)}}^{x^{(i+1/2)}} g_j \, dx + \zeta^{(j)} \left[g_{(i+1/2)j} - g_{(i-1/2)j} \right] \\ &= \Delta x^{(i)} \frac{dg_{ij}}{dt} + \zeta^{(j)} \left[g_{(i+1/2)j} - g_{(i-1/2)j} \right] \end{aligned}$$

similarly, the discretization of the RHS of the same equation gives:

$$\begin{aligned} &= \int_{x^{(i-1/2)}}^{x^{(i+1/2)}} \left(\frac{2}{\sqrt{\pi}} \overline{\rho} (G_j - g_j) \right) dx \\ &= \frac{2}{\sqrt{\pi}} \overline{\rho}_i (G_{ij} - g_{ij}) \Delta x^{(i)} \end{aligned}$$

Given that the case $\zeta^{(j)} > 0$, $\zeta^{(j)} < 0$ and $\zeta^{(j)} = 0$ differ slightly we separate the three treatments.

2.1.1 Case $\zeta^{(j)} > 0$

In the case $\zeta^{(j)} > 0$ the QUICK interpolation scheme defines:

$$\begin{aligned} g_{(i+1/2)j} &= g_{ij} + \alpha_1^{(i+1)} (g_{(i+1)j} - g_{ij}) + \alpha_2^{(i+1)} (g_{ij} - g_{(i-1)j}) \\ g_{(i-1/2)j} &= g_{(i-1)j} + \alpha_1^{(i)} (g_{ij} - g_{(i-1)j}) + \alpha_2^{(i)} (g_{(i-1)j} - g_{(i-2)j}) \end{aligned}$$

from which we derive the following expression for the difference $g_{(i+1/2)j} - g_{(i-1/2)j}$:

$$g_{(i+1/2)j} - g_{(i-1/2)j} = \eta^{(i)} g_{ij} + \alpha_1^{(i+1)} g_{(i+1)j} + \omega^{(i)} g_{(i-1)j} + \alpha_2^{(i)} g_{(i-2)j}$$

with

$$\eta^{(i)} = 1 - \alpha_1^{(i+1)} + \alpha_2^{(i+1)} - \alpha_1^{(i)}$$
$$\omega^{(i)} = -\alpha_2^{(i+1)} - 1 + \alpha_1^{(i)} - \alpha_2^{(i)}$$

The generic equation for g_{ij} $i = 1, \ldots, N-1$ then becomes:

$$\frac{dg_{ij}}{dt} + \frac{\zeta^{(j)}}{\Delta x^{(i)}}\left[\eta^{(i)} g_{ij} + \alpha_1^{(i+1)} g_{(i+1)j} + \omega^{(i)} g_{(i-1)j} + \alpha_2^{(i)} g_{(i-2)j}\right] + \frac{2}{\sqrt{\pi}}\overline{\rho}_i g_{ij} = \frac{2}{\sqrt{\pi}}\overline{\rho}_i G_{ij} \quad (2.23)$$

with g_{0j} given by the boundary condition (2.17a). As for the g_{Nj}–equation we observe that, as shown in Figure 3, the computational point g_{Nj} and the volume boundary $g_{(N+1/2)j}$ coincide. We can then simplify Eq. (2.26) by imposing

$$\begin{aligned} g_{(N+1/2)j} - g_{(N-1/2)j} &= g_{Nj} - g_{(N-1/2)j} \\ &= (1 - \alpha_1^{(N)}) g_{Nj} + (\alpha_1^{(N)} - 1 - \alpha_2^{(N)}) g_{(N-1)j} + \alpha_2^{(N)} g_{(N-2)j} \end{aligned}$$

so that, for g_{Nj}, it holds

$$\frac{dg_{Nj}}{dt} + \frac{\zeta^{(j)}}{\Delta x^{(N)}}\left[(1 - \alpha_1^{(N)}) g_{Nj} + (\alpha_1^{(N)} - 1 - \alpha_2^{(N)}) g_{(N-1)j} + \alpha_2^{(N)} g_{(N-2)j}\right] + \frac{2}{\sqrt{\pi}}\overline{\rho}_N g_{Nj} = \frac{2}{\sqrt{\pi}}\overline{\rho}_N G_{Nj} \quad (2.24)$$

It is also important to note that the equation for g_{1j} involves the term $g_{(-1)j}$. To address this, a computational ghost node at $x^{(-1)} = -x^{(1)}$ is introduced, with its value prescribed by the boundary condition g_{0j}.

All things considered, the equations for g_{ij} can be summarized as:

- $i = 3, \ldots, N-1$: Eq. (2.26)
- $i = N$: Eq. (2.24)
- $i = 1$:

$$\frac{dg_{1j}}{dt} + \frac{\zeta^{(j)}}{\Delta x^{(1)}}\left[\eta^{(1)} g_{1j} + \alpha_1^{(2)} g_{2j}\right] + \frac{2}{\sqrt{\pi}}\overline{\rho}_1 g_{1j} = \frac{2}{\sqrt{\pi}}\overline{\rho}_1 G_{1j} - \frac{\zeta^{(j)}}{\Delta x^{(1)}}\left(\alpha_2^{(1)} + \omega^{(1)}\right) g_{0j}$$

- $i = 2$:

$$\frac{dg_{2j}}{dt} + \frac{\zeta^{(j)}}{\Delta x^{(2)}}\left[\eta^{(2)} g_{2j} + \alpha_1^{(3)} g_{3j} + \omega^{(2)} g_{1j}\right] + \frac{2}{\sqrt{\pi}}\overline{\rho}_2 g_{2j} = \frac{2}{\sqrt{\pi}}\overline{\rho}_2 G_{2j} - \frac{\zeta^{(j)}}{\Delta x^{(2)}}\alpha_2^{(2)} g_{0j}$$

This results in an algebraic formulation of the following form:

$$\frac{d\mathbf{g}_j}{dt} + (\zeta^{(j)} A + R)\mathbf{g}_j = \mathbf{U}_j$$

with $A \in \mathbb{R}^{N \times N}$ being the advection quadri-diagonal matrix:

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 & \cdots & 0 \\ a_{21} & a_{22} & a_{23} & 0 & \cdots & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & \cdots & 0 \\ 0 & a_{42} & a_{43} & a_{44} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & a_{N-1,N} \\ 0 & \cdots & 0 & a_{N,N-2} & a_{N,N-1} & a_{NN} \end{bmatrix}$$

computed as:

$$\Delta x^{(i)} a_{ik} = \begin{cases} \eta^{(i)} & \text{if } k = i \\ \alpha_1^{(i+1)} & \text{if } k = i+1 \\ \omega^{(i)} & \text{if } k = i-1 \text{ and } k > 0 \\ \alpha_2^{(i)} & \text{if } k = i-2 \text{ and } k > 0 \\ 0 & \text{otherwise} \end{cases} \qquad i = 1, \dots, N-1$$

$$\Delta x^{(N)} a_{Nk} = \begin{cases} 1 - \alpha_1^{(N)} & \text{if } k = N \\ \alpha_1^{(N)} - 1 - \alpha_2^{(N)} & \text{if } k = N-1 \\ \alpha_2^{(N)} & \text{if } k = N-2 \end{cases}$$

$R \in \mathbb{R}^{N \times N}$ is instead a diagonal matrix representing part of the reaction:

$$[R]_{ik} = r_{ik} = \frac{2}{\sqrt{\pi}} \overline{\rho}_i \delta_{ik} \quad i = 1, \dots, N$$

Finally $\mathbf{U}_j \in \mathbb{R}^N$ computed as:

$$\begin{aligned} (\mathbf{U}_j)_i &= \frac{2}{\sqrt{\pi}} \overline{\rho}_i G_{ij} \qquad i = 3, \dots, N \\ (\mathbf{U}_j)_1 &= \frac{2}{\sqrt{\pi}} \overline{\rho}_1 G_{1j} - \frac{\zeta^{(j)}}{\Delta x^{(1)}} \left(\alpha_2^{(1)} + \omega^{(1)} \right) g_{0j} \\ (\mathbf{U}_j)_2 &= \frac{2}{\sqrt{\pi}} \overline{\rho}_2 G_{2j} - \frac{\zeta^{(j)}}{\Delta x^{(2)}} \alpha_2^{(2)} g_{0j} \end{aligned} \tag{2.25}$$

Following an analogous line of reasoning we discretize Eq. (2.10b) thus constructing the algebraic system:

$$\frac{d\mathbf{h}_j}{dt} + (\zeta^{(j)} A + R)\mathbf{h}_j = \mathbf{W}_j$$

with:

$$\begin{aligned} (\mathbf{W}_j)_i &= \frac{2}{\sqrt{\pi}} \overline{\rho}_i H_{ij} \qquad i = 3, \dots, N \\ (\mathbf{W}_j)_1 &= \frac{2}{\sqrt{\pi}} \overline{\rho}_1 H_{1j} - \frac{\zeta^{(j)}}{\Delta x^{(1)}} \left(\alpha_2^{(1)} + \omega^{(1)} \right) h_{0j} \\ (\mathbf{W}_j)_2 &= \frac{2}{\sqrt{\pi}} \overline{\rho}_2 H_{2j} - \frac{\zeta^{(j)}}{\Delta x^{(2)}} \alpha_2^{(2)} h_{0j} \end{aligned}$$

2.1.2 Case $\zeta^{(j)} < 0$

If $\zeta^{(j)} < 0$ the QUICK scheme computes:

$$\begin{aligned} g_{(i+1/2)j} &= g_{(i+1)j} + \beta_1^{(i+1)}(g_{ij} - g_{(i+1)j}) + \beta_2^{(i+1)}(g_{(i+1)j} - g_{(i+2)j}) \\ g_{(i-1/2)j} &= g_{ij} + \beta_1^{(i)}(g_{(i-1)j} - g_{ij}) + \beta_2^{(i)}(g_{ij} - g_{(i+1)j}) \end{aligned}$$

so that

$$g_{(i+1/2)j} - g_{(i-1/2)j} = \theta^{(i)} g_{ij} + \sigma^{(i)} g_{(i+1)j} - \beta_2^{(i+1)} g_{(i+2)j} - \beta_1^{(i)} g_{(i-1)j}$$

with

$$\begin{aligned} \theta^{(i)} &= \beta_1^{(i+1)} - 1 + \beta_1^{(i)} - \beta_2^{(i)} \\ \sigma^{(i)} &= 1 - \beta_1^{(i+1)} + \beta_2^{(i+1)} + \beta_2^{(i)} \end{aligned}$$

We have then that for the generic g_{ij} it holds that

$$\frac{dg_{ij}}{dt} + \frac{\zeta^{(j)}}{\Delta x^{(i)}}\left[\theta^{(i)} g_{ij} + \sigma^{(i)} g_{(i+1)j} - \beta_2^{(i+1)} g_{(i+2)j} - \beta_1^{(i)} g_{(i-1)j}\right] + \frac{2}{\sqrt{\pi}}\overline{\rho}_i g_{ij} = \frac{2}{\sqrt{\pi}}\overline{\rho}_i G_{ij} \quad (2.26)$$

Again we account for the presence of the term $g_{(N+1)j}$ in the g_{N-1} equation with a ghost node $x^{(N+1)} = 2x^{(N)} - x^{(N-1)}$ set to the boundary condition g_{Nj}. Moreover, as before, we take into account that the computational point and the boundary of the volume in $x^{(0)}$ coincide to set $g_{(0+1/2)j} - g_{(0-1/2)j} = g_{(1/2)j} - g_{0j}$.

As before this gives rise to an algebraic formulation of the form:

$$\frac{d\mathbf{g}_j}{dt} + (\zeta^{(j)} B + R)\mathbf{g}_j = \mathbf{U}_j$$

with $B \in \mathbb{R}^{N\times N}$ being a quadri-diagonal matrix

$$B = \begin{bmatrix} b_{00} & b_{01} & b_{02} & 0 & \cdots & 0 \\ b_{10} & b_{11} & b_{12} & b_{13} & \cdots & 0 \\ 0 & b_{21} & b_{22} & b_{23} & \ddots & \vdots \\ 0 & 0 & b_{32} & b_{33} & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & b_{N-2,N-1} \\ 0 & 0 & \cdots & 0 & b_{N-1,N-2} & b_{N-1,N-1} \end{bmatrix}$$

computed as

$$\Delta x^{(i)} b_{ik} = \begin{cases} \theta^{(i)} & \text{if } k = i \\ \sigma^{(i)} & \text{if } k = i+1 \text{ and } k < N-1 \\ -\beta_2^{(i+1)} & \text{if } k = i+2 \text{ and } k < N-1 \\ -\beta_1^{(i)} & \text{if } k = i-1 \\ 0 & \text{otherwise} \end{cases} \qquad i = 1, \ldots, N-1$$

$$\Delta x^{(0)} b_{0k} = \begin{cases} \beta_1^{(1)} - 1 & \text{if } k = 0 \\ 1 - \beta_1^{(1)} + \beta_2^{(1)} & \text{if } k = 1 \\ -\beta_2^{(1)} & \text{if } k = 2 \end{cases}$$

$R \in \mathbb{R}^{N\times N}$, $r_{ik} = \frac{2}{\sqrt{\pi}}\overline{\rho}_i \delta_{ik}$ $i = 0, \ldots, N-1$ and $\mathbf{U}_j \in \mathbb{R}^N$ constructed as

$$\begin{aligned} (\mathbf{U}_j)_i &= \frac{2}{\sqrt{\pi}}\overline{\rho}_i G_{ij} \qquad i = 0, \ldots, N-3 \\ (\mathbf{U}_j)_{N-1} &= \frac{2}{\sqrt{\pi}}\overline{\rho}_{N-1} G_{(N-1)j} + \frac{\zeta^{(j)}}{\Delta x^{(N-1)}}\left(\beta_2^{(N)} - \sigma^{(N-1)}\right) g_{Nj} \\ (\mathbf{U}_j)_{N-2} &= \frac{2}{\sqrt{\pi}}\overline{\rho}_{N-2} G_{(N-2)j} + \frac{\zeta^{(j)}}{\Delta x^{(N-2)}}\beta_2^{(N-1)} g_{Nj} \end{aligned}$$

Again, Eq. (2.10b) can be discretized using an analogous treatment thus obtaining

$$\frac{d\mathbf{h}_j}{dt} + (\zeta^{(j)} B + R)\mathbf{h}_j = \mathbf{W}_j$$

with

$$(\mathbf{W}_j)_i = \frac{2}{\sqrt{\pi}}\overline{\rho}_i H_{ij} \qquad i = 0, \ldots, N-3$$
$$(\mathbf{W}_j)_{N-1} = \frac{2}{\sqrt{\pi}}\overline{\rho}_{N-1} H_{(N-1)j} + \frac{\zeta^{(j)}}{\Delta x^{(N-1)}}\left(\beta_2^{(N)} - \sigma^{(N-1)}\right) h_{Nj}$$
$$(\mathbf{W}_j)_{N-2} = \frac{2}{\sqrt{\pi}}\overline{\rho}_{N-2} H_{(N-2)j} + \frac{\zeta^{(j)}}{\Delta x^{(N-2)}}\beta_2^{(N-1)} h_{Nj}$$

2.1.3 Case $\zeta^{(j)} = 0$

If $\zeta^{(j)} = 0$ then the problem at hand is an initial value one:

$$\frac{dg_{ij}}{dt} + \frac{2}{\sqrt{\pi}}\overline{\rho}_i g_{ij} = \frac{2}{\sqrt{\pi}}\overline{\rho}_i G_{ij} \quad i = 0, \ldots, N-1$$

which is equivalent the the algebraic system of ODEs:

$$\frac{d\mathbf{g}_j}{dt} + R\mathbf{g}_j = \mathbf{U}_j$$

with $R \in \mathbb{R}^{N\times N}$ and $U \in \mathbb{R}^N$ defined as follows

$$r_{ik} = \frac{2}{\sqrt{\pi}}\overline{\rho}_i \delta_{ik} \qquad (\mathbf{U}_j)_i = \frac{2}{\sqrt{\pi}}\overline{\rho}_i G_{ij}$$

Analogously, for $\mathbf{h}_0$ we have

$$\frac{d\mathbf{h}_j}{dt} + R\mathbf{h}_j = \mathbf{W}_j$$

with

$$(\mathbf{W}_j)_i = \frac{2}{\sqrt{\pi}}\overline{\rho}_i H_{ij}$$

2.1.4 Temporal Discretization

Temporal discretization is performed through a semi-implicit scheme of the form:

$$\frac{\mathbf{u}^{k+1} - \mathbf{u}^k}{\Delta t} + (M + R^k)\mathbf{u}^{k+1} = \mathbf{F}^k$$

where by R^k we mean the R matrix computed with $\overline{\rho}_i^k$. This gives us 6 systems of equations:

$$\begin{aligned}\mathbf{g}_j^{k+1} &= \left[I + \Delta t\left(\zeta^{(j)}A + R^k\right)\right]^{-1}\left(\mathbf{g}_j^k + \Delta t\mathbf{U}_j^k\right)\\ \mathbf{h}_j^{k+1} &= \left[I + \Delta t\left(\zeta^{(j)}A + R^k\right)\right]^{-1}\left(\mathbf{h}_j^k + \Delta t\mathbf{W}_j^k\right)\end{aligned} \qquad j : \zeta^{(j)} > 0 \tag{2.27}$$

$$\begin{aligned}\mathbf{g}_j^{k+1} &= \left[I + \Delta t\left(\zeta^{(j)}B + R^k\right)\right]^{-1}\left(\mathbf{g}_j^k + \Delta t\mathbf{U}_j^k\right)\\ \mathbf{h}_j^{k+1} &= \left[I + \Delta t\left(\zeta^{(j)}B + R^k\right)\right]^{-1}\left(\mathbf{h}_j^k + \Delta t\mathbf{W}_j^k\right)\end{aligned} \qquad j : \zeta^{(j)} < 0 \tag{2.28}$$

$$\begin{aligned}\mathbf{g}_j^{k+1} &= \left[I + \Delta t R^k\right]^{-1}\left(\mathbf{g}_j^k + \Delta t\mathbf{U}_j^k\right)\\ \mathbf{h}_j^{k+1} &= \left[I + \Delta t R^k\right]^{-1}\left(\mathbf{h}_j^k + \Delta t\mathbf{W}_j^k\right)\end{aligned} \qquad j : \zeta^{(j)} = 0 \tag{2.29}$$

2.2 Solution Iteration

In this section an outline of the process of the computation of the quantities $\mathbf{g}_j^k$ and $\mathbf{h}_j^k$ $\forall j$ is given. We can devide the core procedure into 4 steps:

(i). Using the initial data $\mathbf{g}_j^0$ and $\mathbf{h}_j^0$ compute $\overline{\rho}_i^0$, $\overline{v}_i^0$ and $\overline{T}_i^0$ using Eqs. (2.12) – (2.14) and integrating according to Simpson's rule. Using the same initial data compute also $\mathbf{G}_j^0$ and $\mathbf{H}_j^0$ using Eqs (2.11a) and (2.11b).

(ii). Compute $\mathbf{g}_j^1$ and $\mathbf{h}_j^1$ ($j > 0$) solving (2.27) using the boundary data g_{0j}^1 and h_{0j}^1 given by Eqs (2.17a) and (2.17b) and the initial data at the previous stage $\mathbf{g}_j^0$, $\mathbf{h}_j^0$, $\mathbf{G}_j^0$, $\mathbf{H}_j^0$ and $\overline{\rho}_i^0$.

(iii). Compute $\mathbf{g}_j^1$ and $\mathbf{h}_j^1$ ($j \leq 0$) solving respectively (2.28) and (2.29) using the boundary data g_{Nj}^1 and h_{Nj}^1 given by Eqs (2.19a) and (2.19b) and the initial data at the previous stage $\mathbf{g}_j^0$, $\mathbf{h}_j^0$, $\mathbf{G}_j^0$, $\mathbf{H}_j^0$ and $\overline{\rho}_i^0$.

(iv). Repeat process (i)-(iii) with the shift of superscripts (0 to k and 1 to $k+1$) to obtain $\mathbf{g}_j^{k+1}$ and $\mathbf{h}_j^{k+1}$ until satisfactory convergence is reached or the max iterations limit is met.

As for the stopping criterion a few things have to be accounted for. First of all the variables at play are two (g and h) therefore, at a fixed iteration k and at a fixed velocity j there are two errors to be accounted for. Moreover, considering only one of the two, we have $2\bar{N}+1$ independent equations (one for each velocity point) therefore, the errors from this different but linked in time equations need to be combined in some way too. Finally, we observe that the quantity at play are in general very small, therefore, the order of magnitude of the tolerance has to be chosen wisely. Taking all of this into account, the following stopping criterion is proposed.

Let $err_{u,k}$ be the error for the quantity $u \in \{g, h\}$ at timestep k; its computation is performed as follows:

$$err_{u,k} = \varphi\left(\left\{\frac{\|\mathbf{u}_j^k - \mathbf{u}_j^{k-1}\|}{\|\mathbf{u}_j^{k-1}\|}\right\}_{j=-\overline{N}}^{\overline{N}}\right) \tag{2.30}$$

where $\varphi(\cdot) : \mathbb{R}^n \to \mathbb{R}$ is an aggregating function (either "max(·)" or "avg(·)") that takes as input the sequence of the relative errors between two subsequent timesteps of all the $2\bar{N}+1$ velocity equations and $\|\cdot\|$ is a norm on $\mathbb{R}^{N+1}$ (e.g. L^1, L^2 and L^∞). The final error for the timestep—i.e. err_k—is then computed as:

$$err_k = \sqrt{err_{g,k}^2 + err_{h,k}^2}$$

Algorithm 1 Iterative computation of $\mathbf{g}_j$ and $\mathbf{h}_j$

Require: Initial data $\mathbf{g}_j^0, \mathbf{h}_j^0$, tolerance ϵ, max iterations K

$k \leftarrow 0$
converged $\leftarrow$ FALSE
while $k < K$ **and not** *converged* **do**
  {Step (i): Macro-quantities and source terms}
  Compute $\overline{\rho}_i^k, \overline{v}_i^k, \overline{T}_i^k$ using Simpson's rule and Eqs. (2.12)–(2.14)
  Compute $\mathbf{G}_j^k, \mathbf{H}_j^k$ using Eqs. (2.11a)–(2.11b)
  {Step (ii) & (iii): Solve kinetic equations for all j}
  for $j = -\overline{N}$ **to** $\overline{N}$ **do**
    if $j > 0$ **then**
      Solve (2.27) for $\mathbf{g}_j^{k+1}, \mathbf{h}_j^{k+1}$ using boundary data $g_{0j}^{k+1}, h_{0j}^{k+1}$
    else if $j < 0$ **then**
      Solve (2.28) for $\mathbf{g}_j^{k+1}, \mathbf{h}_j^{k+1}$ using boundary data $g_{Nj}^{k+1}, h_{Nj}^{k+1}$
    else
      Solve (2.29) for $\mathbf{g}_j^{k+1}, \mathbf{h}_j^{k+1}$
    end if
  end for
  {Step (iv): Convergence check}
  $err_{g,k+1} \leftarrow \varphi\left(\left\{\frac{\|\mathbf{g}_j^{k+1} - \mathbf{g}_j^k\|}{\|\mathbf{g}_j^k\|}\right\}\right)$
  $err_{h,k+1} \leftarrow \varphi\left(\left\{\frac{\|\mathbf{h}_j^{k+1} - \mathbf{h}_j^k\|}{\|\mathbf{h}_j^k\|}\right\}\right)$
  $err_{k+1} \leftarrow \sqrt{err_{g,k+1}^2 + err_{h,k+1}^2}$
  if $err_{k+1} < \epsilon$ **then**
    converged $\leftarrow$ TRUE
  else
    $k \leftarrow k + 1$
  end if
end while
return $\mathbf{g}_j^k, \mathbf{h}_j^k$

3 Implementation

The present section is dedicated to discussing the main implementation choices made during the drafting of the code together with presenting the main structural nodes that constitute it. For the complete code and doxygen documentation please refer to the GitHub Repository; meanwhile, in the appendix, the signatures of the main classes are provided.

3.1 `utilities` File

The `utilities.hpp` file provides a collection of foundational tools, concepts, and error-handling mechanisms that ensure consistency across the different modules of the library.

To ensure the flexibility of the solver while maintaining strict performance requirements, the code utilizes `C++20` Concepts. These formalize the requirements for various template parameters used throughout the meshes and the solver:

1. `MeshContainer1D`: This concept defines the structural requirements for any container representing a mesh. It ensures that the container is a standard range, holds elements of the correct floating-point precision, and provides $O(1)$ random access through contiguous memory, which is critical for the efficiency of the numerical schemes.

```cpp
template <typename Container, typename T>
concept MeshContainer1D =
    std::ranges::range<Container> &&
    std::same_as<std::ranges::range_value_t<Container>, T> &&
    requires(const Container &c) {
        { c.size() } -> std::integral;
    } &&
    requires(const Container &c, std::size_t n) {
        { c[n] } -> std::convertible_to<const T &>;
    } &&
    std::random_access_iterator<std::ranges::iterator_t<Container>>;
```

2. `SpacingFunction` and `SingleArgFunction`: These concepts define the interface for callable objects used within the library. For instance, `SpacingFunction` ensures that a provided lambda or function object can correctly map indices to spatial coordinates, which is essential for defining non-uniform mesh distributions

```cpp
template <typename SpacingFunc, typename T>
concept SpacingFunction = requires(SpacingFunc func, size_t index) {
    { func(index) } -> std::convertible_to<T>;
};
```

```cpp
template <typename Func, typename T>
concept SingleArgFunction = requires(Func func, T arg) {
    { func(arg) } -> std::convertible_to<T>;
};
```

Moreover, the `error_message()` function is provided, it is a diagnostic tool that leverages `std::source_location` to generate highly detailed logs. Unlike standard exception messages, this utility automatically captures the file name, line number, and function name where the error occurred, significantly reducing the time required for debugging simulation failures.

```cpp
std::string error_message(const std::string &message, const std::source_location &loc)
{
     std::string error_msg;
     error_msg += "Error at " + std::string(loc.file_name()) + ":" + std::to_string(loc.line()) +
```

```
                    " in " + loc.function_name() + ": " + message + "\n";

    return error_msg;
};
```

3.2 ConfigData Object

The configuration and management of simulation parameters is addressed through the `ConfigData` class, which serves as a centralized repository for all physical, geometric, and computational parameters required to define the kinetic problem. The class manages four primary type of data:

1. Mesh Parameters: This section defines the discretization of the problem, including the total number of spatial points (N), the subset of polynomially spaced points (N_0), and the resolution of the velocity mesh ($\overline{N}$). It also includes the parameters which govern the non-uniform distribution of nodes in the computational domain: d_1, d_2, a_1, a_2.

2. Physical Parameters: These members represent the fundamental physics of the plasma model, such as the ratio of infinity to wall temperature (T_∞/T_w), the pressure ratio (p_∞/p_w), and the Mach number at infinity (M_∞).

3. Simulation Controls: The temporal evolution and numerical convergence are managed by variables such as the time-stepping size (dt), the relative tolerance for convergence (tol), and the maximum number of iterations allowed (`max_iter`). Additionally, it monitors the visualization frequency via the `save_every_k_steps` parameter.

4. General Settings: This includes metadata such as the folder name used for saving results, ensuring that simulation outputs are organized systematically.

Access to these parameters is strictly controlled through a comprehensive set of constant getter methods, which preserve the encapsulation of the data while allowing other components of the system to query the simulation state without the risk of unintended modification.

The class also provides a specialized constructor that accepts a filename string, allowing the system to load parameters directly from a `.json` configuration file which will be the subject of Sec. 5.

3.3 BaseMesh1D, SpaceMeshFV and VelocityMesh Objects

The core of the discretization layer is the `BaseMesh1D` class, a pure virtual template that serves as the primary abstraction for all one-dimensional grids within the simulation. It is designed to be agnostic of the specific coordinate system, utilizing a `MeshNature` enumeration to differentiate[7] between Space and Velocity domains at compile time.

This base class manages the fundamental lifecycle of a mesh, storing the total number of computational points N and the primary container for the coordinates, `x_comp`. To ensure memory efficiency and flexibility, the container type is templated, allowing the use of various STL-compatible sequences.

```
template <FloatingPoint T,
          MeshContainer1D<T> Container = std::vector<T>,
          MeshNature Nature = MeshNature::SPACE>
class BaseMesh1D
{...}
```

[7] This differentiation is needed mainly for the output functions

The class provides a standardized interface for:

- State Management: Tracking the distinct initialization phase via the `is_initialized` flag to prevent operations on unallocated or uncomputed data.
- Data Access: Overloading the `[]` operator and providing a bounds-checked `at()` method to allow for intuitive, array-like interaction with the mesh points.
- Iteration: Exposing standard `begin()` and `end()` iterators, which enables the mesh to be used seamlessly with `C++20` ranges and standard algorithms.
- Writing: Outputting a `.txt` file with the indices and values of the computational points for further postprocessing.

By defining a virtual destructor and a pure virtual `initialize_mesh()` method, `BaseMesh1D` enforces a contract that requires derived classes to provide specific logic for point distribution, whether uniform, polynomial, or custom-defined.

The specialization of the pure virtual class concerns the two type of mesh needed in our problem: a finite volume mesh for the space dimension and classical discretization for the velocity space (Fig. 5). Both of them will be described in the following.

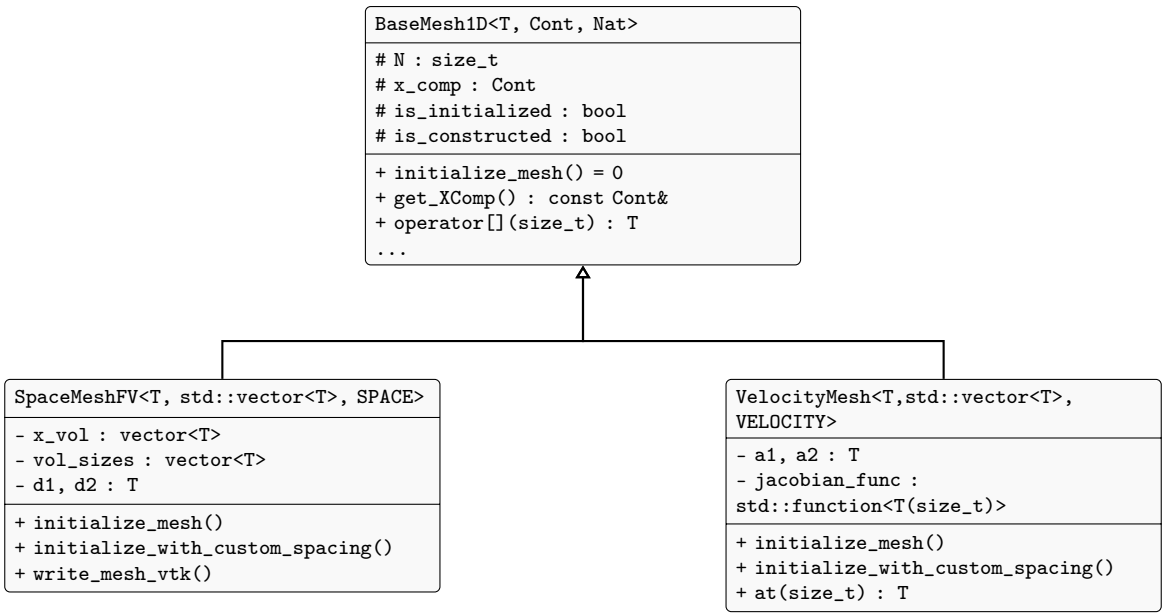


Fig. 5. Class hierarchy for the 1D mesh discretization system.

The `SpaceMeshFV` class specializes the base mesh for Finite Volume (FV) applications. Unlike a simple point-wise discretization, a finite volume approach requires the definition of cell boundaries and the calculation of the volumes (or lengths in 1D) associated with each computational node. The implementation of `SpaceMeshFV` extends the base functionality by introducing specific members for volume management: `x_vol` stores the boundaries of each cell, while `vol_sizes` captures the measure of each volume element. The class defines the parameters d_1, d_2 and N_0 which are used in the default `initialize_mesh()` method to fill the computational points, volume boundaries and sizes containers according to the spacings of Eq. (2.20). Furthermore, the class provides support for custom spacing functions through `initialize_with_custom_spacing()`; this allows the user to inject arbitrary mathematical distributions $x(\cdot) : i \in \mathbb{N} \to x^{(i)} \in \mathbb{R}^+$ into the mesh generation process, provided they satisfy the required functional concepts.

```cpp
template <typename T>
template <SpacingFunction<T> Spacing>
void SpaceMeshFV<T>::initialize_with_custom_spacing(Spacing &&spacing_func)
{
        if (this->is_initialized)
        {
                std::cerr << "SpaceMesh already initialized. Skipping initialization." << std::endl;
                return;
        }

        // Computational points x^{(i)}
        this->x_comp.resize(this->N + 1);

        for (size_t i = 0; i <= this->N; ++i)
           this->x_comp[i] = spacing_func(i);

        // volume boundaries
        x_vol.resize(this->N + 2);
        vol_sizes.resize(this->N + 1);

        x_vol[0] = this->x_comp[0];
        for (size_t i = 0; i < this->N; ++i)
                x_vol[i + 1] = T{0.5} * (this->x_comp[i] + this->x_comp[i + 1]);
        x_vol[this->N + 1] = this->x_comp[this->N];

        for (size_t i = 0; i < this->N + 1; ++i)
                vol_sizes[i] = x_vol[i + 1] - x_vol[i];

        this->is_initialized = true;
}
```

As a final additional feature, the class handles data persistence by offering methods to export the mesh configuration to both raw text formats and VTK files, the latter enabling high-fidelity visualization of the spatial grid in tools such as ParaView (Fig. 6).

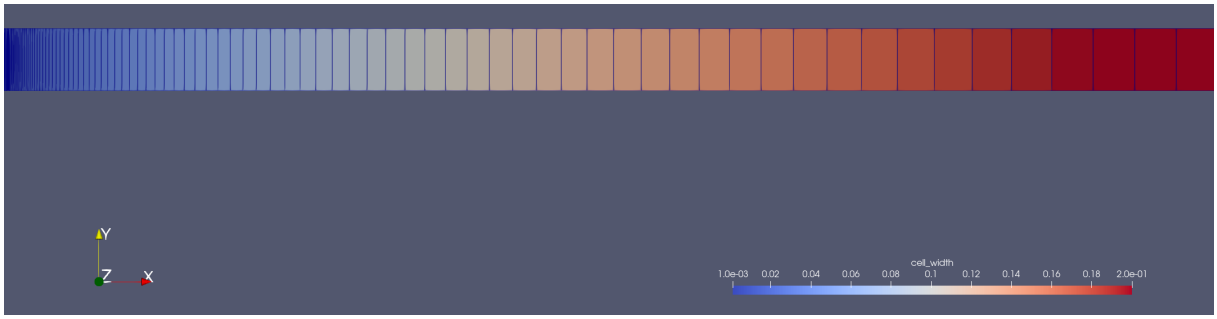


Fig. 6. Visualization of a `space_mesh.vtk` file in ParaView. Each cell is characterized by both a cell ID and a cell width.

The `VelocityMesh` is more in line with `BaseMesh1D` by needing only `x_comp`. It specializes itself by defining the parameters a_1 and a_2 needed for filling the container according to the rule described in Eq. (2.21); a process that takes place through the default `initialize_mesh()` method. As in `SpaceMeshFV`, a `initialize_with_custom_spacing()` is provided which takes $\zeta(\cdot) : j \in \mathbb{N} \to \zeta^{(j)} \in \mathbb{R}^+$ and fills the mesh in a *symmetric* way:

```cpp
template <typename T>
template <SpacingFunction<T> Spacing>
void VelocityMesh<T>::initialize_with_custom_spacing(Spacing &&spacing_func)
{
    if (this->is_initialized)
    {
        std::cerr << "Velocity Mesh already initialized. Skipping initialization." << std::endl;
        return;
    }
```

```
10
11      // Computational points x^{(i)}
12
13      this->x_comp.resize(2 * this->N + 1);
14      this->x_comp[this->N] = T{0}; // Center point at zero
15
16      T value;
17      // initialize positive side and mirror to negative side
18      for (size_t i = this->N + 1; i <= 2 * this->N; ++i)
19      {
20          value = spacing_func(i - this->N);
21          this->x_comp[i] = value;
22          this->x_comp[2 * this->N - i] = -value; // Symmetric point
23      }
24      this->is_initialized = true;
25  }
```

Tt's important to mention that the symmetric nature is *not* reflected in the access operator `[]` and neither in`at()`. This means that an usage like `velocity_mesh[-1]` will throw an error given that the methods expect still `size_t` variables.

Finally, for reasons that we will shortly see, a `jacobian_function` functional is needed to store $\partial\zeta^{(j)}/\partial j$ (as `std::function<T(size_t)>`). In particular `jacobian_function` : $k \in \{0, \dots, 2\bar{N}+1\} \to \mathbb{R}$ and should deal internally with the negative index convention.

Given their slight different logic, in the following a short summary of the main methods for `VelocityMesh` (with examples) is made taking as reference. Eq.(2.21):

- `initialize_with_custom_spacing()`: requires as `SpacingFunction<T>` (`spacing_funct`) that takes $j \geq 0$ and construct the velocity mesh $\zeta^{(j)}$ ($2\bar{N}+1$ points) dealing with the symmetric nature internally e.g.:

```
this->x_comp[this->N] = T{0}; // Center point at zero

T value;

// initialize positive side and mirror to negative side
for (size_t i = this->N + 1; i <= 2 * this->N; ++i)
{
    value = spacing_func(i - this->N);
    this->x_comp[i] = value;
    this->x_comp[2 * this->N - i] = -value; // Symmetric point
}
```

```
auto default_spacing = [this](size_t i) -> T
{
    return a1 * i + a2 * std::pow<T>(i, 3);
};
```

- `[]` & `.at()`: Access operators, they treat the mesh as a generic `C++` container and therefore accept $k \in \{0, \dots, 2\bar{N}+1\}$ e.g.:

```
assert(index <= 2 * this->N && "Index out of range in VelocityMesh::operator[]");
return this->x_comp[index];
```

  It's the user responsibility to be familiar with the correspondence: $j = 0 \iff k = \overline{N}$, $j = -1 \iff k = \overline{N} - 1$, $j = +1 \iff k = \overline{N} + 1$

- `jacobian_function`: Requires a requires as `SpacingFunction<T>` (`jacobian_func`) that represents $\partial\zeta^{(j)}/\partial j$. Given that it will be used for integral computation its nature is in

line with both `.at()` in the sense that should accept values $k \in \{0, \dots, 2\bar{N}+1\}$ and `initialize_with_custom_spacing()` in the sense that should internally deal with the "negative index convention", i.e.:

```
auto get_jacobian = [&](size_t k) -> T
{
    // Cast to signed integer to handle negative relative indices correctly
    long long j_signed = static_cast<long long>(k) - static_cast<long long>(this->N);
    return config.get_a1() + 3.0 * config.get_a2() * (static_cast<T>(j_signed) \
           * static_cast<T>(j_signed));
};
jacobian_func = get_jacobian;
```

3.4 numeric_utils File

The implementation of numerical utilities for the project is primarily encapsulated within the `Bgk::numerics` namespace, providing a modular and template-based framework for calculating the geometric weights required by the QUICK scheme (cf. Appendix A.1). The architectural focus of these functions is to translate the theoretical three-point upstream-biased stencil into a robust computational procedure that interacts directly with the `SpaceMeshFV` class.

The calculation of coefficients for positive velocity fields is handled by the `QUICKcoefficients_p()` function, which accepts a reference to the spatial mesh and returns a vector of coordinate pairs representing the $\{\alpha_1^{(i)}, \alpha_2^{(i)}\}$ weights. Before proceeding with the computation, the function performs a critical validation check to ensure the mesh has been properly initialized, throwing an invalid argument exception if the geometric data is unavailable. To optimize memory management during the simulation's setup phase, the implementation utilizes the `reserve` method to allocate the necessary memory for the entire vector of coefficients upfront, thereby avoiding the overhead of multiple reallocations.

The treatment of boundaries within this function is particularly important for the stability of the finite volume formulation. The coefficients for the first and last boundaries are explicitly set to zero as, in our formulation, they are governed by external boundary conditions or terminal cell logic rather than internal interpolation. A specialized logic is applied at the second volume boundary ($i = 1$) to maintain the quadratic accuracy of the scheme; following what said in Sec. 2.1.1 the implementation incorporates a symmetric ghost node $x^{(-1)}$ through a reflective geometric calculation. For all subsequent internal boundaries, the function enters a general loop that calculates the interpolation weights based on the relative distances between the current boundary and the surrounding computational nodes $x^{(i)}$, $x^{(i-1)}$ and $x^{(i-2)}$.

```
template <typename T>
std::vector<std::pair<T, T>> QUICKcoefficients_p(const SpaceMeshFV<T> &mesh)
{
    if (!mesh.validate_mesh())
        throw std::invalid_argument(error_message(
            "Invalid mesh: trying to compute QUICK coefficients for an uninitialized mesh."));

    std::vector<std::pair<T, T>> coefficients;
    const std::vector<T> &vol_boundaries = mesh.get_volume_boundaries();
    const std::vector<T> &x_comp = mesh.get_XComp();
    size_t V = vol_boundaries.size();
    T a1, a2;
    coefficients.reserve(V);

    // Case i = 0
    coefficients.emplace_back(T{0}, T{0});

```

```
18      // Case i = 1: Account for the symmetric ghost node x^{(-1)} (x_comp[-1])
19      a1 = ((vol_boundaries[1] - x_comp[0]) * (vol_boundaries[1] + x_comp[1])) /
20           ((x_comp[1] - x_comp[0]) * (x_comp[1] + x_comp[1]));
21      a2 = ((vol_boundaries[1] - x_comp[0]) * (x_comp[1] - vol_boundaries[1])) /
22           ((x_comp[0] + x_comp[1]) * (x_comp[1] + x_comp[1]));
23
24      coefficients.emplace_back(a1, a2);
25
26      // General case:
27      for (size_t i = 2; i < V - 1; ++i)
28      {
29          a1 = ((vol_boundaries[i] - x_comp[i - 1]) * (vol_boundaries[i] - x_comp[i - 2])) /
30               ((x_comp[i] - x_comp[i - 1]) * (x_comp[i] - x_comp[i - 2]));
31          a2 = ((vol_boundaries[i] - x_comp[i - 1]) * (x_comp[i] - vol_boundaries[i])) /
32               ((x_comp[i - 1] - x_comp[i - 2]) * (x_comp[i] - x_comp[i - 2]));
33          coefficients.emplace_back(a1, a2);
34      }
35
36      // Case i = V - 1
37      coefficients.emplace_back(T{0}, T{0});
38
39      return coefficients;
40  }
```

Conversely, the `QUICKcoefficients_n()` function provides the necessary coefficients for regimes where the velocity is negative. While the general structure of this function mirrors that of its positive counterpart, the implementation must account for the reversal of the upstream direction. The general loop calculates the $\{\beta_1^{(i)}, \beta_2^{(i)}\}$ coefficients by evaluating the distances relative to the downstream nodes $x^{(i)}$ and $x^{(i+1)}$.

To support the three-point stencil at the end of the domain, the code implements an extrapolated ghost node $x^{(N+1)}$ for the boundary at $i = V-2$ ($V = N+2$ total number of boundaries). This ghost node is calculated as $2 \cdot x^{(V-2)} - x^{(V-3)}$, effectively mirroring the mesh spacing to allow for a consistent quadratic reconstruction even when the physical nodes are exhausted. Just as in the positive case, the terminal boundaries are nullified to ensure that the numerical flux remains consistent with the global boundary conditions of the BGK model.

In addition to the bulk coefficient generators, the implementation provides granular access through the `QUICKcoefficients_p_at()` and `QUICKcoefficients_n_at()` functions, which allow for the calculation of weights at a specific mesh index. These functions are particularly useful for localized flux updates or debugging. They incorporate rigorous error handling, including checks to ensure that the requested index i does not exceed the physical boundaries of the mesh. If an out-of-bounds index is targeted, the system throws a descriptive `std::out_of_range` error, reinforcing the defensive programming style adopted throughout the implementation to ensure numerical reliability.

3.5 `phys_utils` File

The `phys_utils` file contains the computational logic for the physical quantities characterizing the gas evolution, as defined in Eqs. (2.12)–(2.14). The numerical evaluation of these integrals is governed by two primary methodological choices: the selection of an appropriate integration quadrature for non-uniform meshes and the treatment of the kinetic discontinuity at the velocity origin.

First, a modified Simpson's rule is employed to account for the non-uniformity of the velocity mesh. To map the non-uniform coordinates x_j (where $j \in \{-\overline{N}, \dots, \overline{N}\}$) to a uniform computational space j, a change of variables is performed. For a generic integrand $y(\zeta)$, the

transformation is expressed as:

$$\int y(\zeta)\, d\zeta = \int y(\zeta(j)) \cdot \frac{d\zeta}{dj}\, dj.$$

Following this transformation, the classical Simpson's 1/3 rule is applied over the uniform index space $\{-\overline{N}, -\overline{N}+1, -\overline{N}+2, \ldots, \overline{N}\}$ ($\overline{N}$ must then be even). We remember that for a general domain discretized into n intervals of constant width h, the integral is approximated as:

$$\int_a^b y(\zeta)\, d\zeta = \frac{h}{3}\left[y(\zeta_0) + y(\zeta_n) + 4 \sum_{i=1,\ i=\text{odd}}^{n-1} y(\zeta_i) + 2 \sum_{i=2,\ i=\text{even}}^{n-2} y(\zeta_i) \right]$$

The second critical design choice concerns the physical discontinuity at the interface ($x = 0$) between incoming and outgoing molecules. At the condensed phase boundary, the molecular distribution is bifurcated based on the sign of the molecular velocity ζ:

1. Incoming Molecules ($\zeta < 0$): Particles arriving at the wall from the bulk gas, whose distribution is determined by the macroscopic evolution of the flow.

2. Outgoing Molecules ($\zeta > 0$): Particles evaporated or reflected from the condensed phase, modeled by a stationary Maxwellian distribution at the saturation pressure p_w and wall temperature T_w.

Due to these distinct boundary conditions, the distribution functions g and h exhibit a sharp jump discontinuity at $\zeta = 0$. Although intermolecular collisions gradually smooth this jump as the flow propagates, a residual "kink" or slope discontinuity persists at $\zeta = 0$ throughout the flow field. Applying a standard numerical integration across the entire domain would lead to significant numerical errors, particularly within the Knudsen layer, as the quadrature would attempt to fit a continuous parabola across the discontinuity at $\{\zeta_{-1}, \zeta_0, \zeta_1\}$. To mitigate this, the integrals are evaluated separately for the semi-infinite intervals $\zeta < 0$ and $\zeta > 0$.

By splitting the integration at the origin, the scheme independently incorporates the limiting values $g(0^-)$ and $g(0^+)$. At the wall, $g(0^-)$ is obtained from the gas-phase computation, while $g(0^+)$ is prescribed by the boundary condition. From a numerical perspective, a single-loop Simpson's rule effectively assigns a weight of 2 to the central node (corresponding to index $\overline{N}$):

$$I_{\text{central}} \propto 2 \cdot g(\overline{N}), \tag{3.1}$$

which necessitates an arbitrary choice between the incoming or outgoing value. In contrast, the split-loop approach yields:

$$I_{\text{central}} \propto (1 \cdot g_{\text{incoming}}) + (1 \cdot g_{\text{outgoing}}). \tag{3.2}$$

This formulation correctly accounts for the contribution of both populations at the interface. As noted in [1] and discussed in Sec. 4, this specialized treatment is strictly necessary only in the vicinity of the wall, as the discontinuity dissipates rapidly for $x_i > 0$.

As far as implementation goes, each physical quantities has a `compute_quantity()` functions that returns the the vector containing the value of such quantity at each of the space computational points, and a `compute_quantity_at(i)` that return the quantity at a specific space index. Morover, for the $\overline{v}$ and $\overline{T}$ family of functions, two versions for each of the two are implemented: one that requires a precomputed $\overline{\rho}$ and or $\overline{v}$ and one that does not. All these functions are contained in the `Bgk::phys` namespace.

In the following, the `compute_density()` function is provided together with the signature of the `compute_temperature()` functions:

```cpp
template <typename T, typename JacobianFunc>
Eigen::Vector<T, Eigen::Dynamic> compute_density(
                                const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor> &g,
                                const VelocityMesh<T> &velocity_mesh,
                                T wall_g_val,
                                JacobianFunc get_jacobian)
{
    Eigen::Vector<T, Eigen::Dynamic> densities(g.cols());
    if (g.cols() == 0)
    {
        return densities;
    }
    const size_t zero_idx = velocity_mesh.get_N();
    const size_t max_idx = velocity_mesh.size() - 1;

    // --- Case i = 0 (Wall) ---
    {
        constexpr Eigen::Index i = 0;
        T integral_neg = T{0};
        T integral_pos = T{0};
        // 1. Negative Velocity Integral (Incoming)
        for (size_t k = 0; k <= zero_idx; ++k)
        {
            T weight = (k == 0  k == zero_idx) ? 1.0 : ((zero_idx - k) % 2 != 0 ? 4.0 : 2.0);
            integral_neg += weight * g(k, i) * get_jacobian(k);
        }
        integral_neg *= (1.0 / 3.0);
        // 2. Positive Velocity Integral (Outgoing) with discontinuity fix
        for (size_t k = zero_idx; k <= max_idx; ++k)
        {
            T weight = (k == zero_idx  k == max_idx) ? 1.0 : ((k - zero_idx) % 2 != 0 ? 4.0 : 2.0);
            T f_val = (k == zero_idx) ? wall_g_val : g(k, i);
            integral_pos += weight * f_val * get_jacobian(k);
        }
        integral_pos *= (1.0 / 3.0);
        densities.coeffRef(i) = integral_neg + integral_pos;
    }

    // --- Cases i > 0 (Away from Wall) ---
    for (Eigen::Index i = 1; i < g.cols(); ++i)
    {
        T integral_neg = T{0};
        T integral_pos = T{0};
        // 1. Negative Velocity Integral (Incoming)
        for (size_t k = 0; k <= zero_idx; ++k)
        {
            T weight = (k == 0  k == zero_idx) ? 1.0 : ((zero_idx - k) % 2 != 0 ? 4.0 : 2.0);
            integral_neg += weight * g(k, i) * get_jacobian(k);
        }
        integral_neg *= (1.0 / 3.0);
        // 2. Positive Velocity Integral (Outgoing)
        for (size_t k = zero_idx; k <= max_idx; ++k)
        {
            T weight = (k == zero_idx  k == max_idx) ? 1.0 : ((k - zero_idx) % 2 != 0 ? 4.0 : 2.0);
            integral_pos += weight * g(k, i) * get_jacobian(k);
        }
        integral_pos *= (1.0 / 3.0);
        densities.coeffRef(i) = integral_neg + integral_pos;
    }

    return densities;
}
```

```cpp
namespace Bgk{

    namespace phys{

```

```
        template <typename T, typename JacobianFunc>
        Eigen::Vector<T, Eigen::Dynamic> compute_temperature(
                                    const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor> &g,
                                    const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor> &h,
                                    const VelocityMesh<T> &velocity_mesh,
                                    const Eigen::Vector<T, Eigen::Dynamic> &densities,
                                    const Eigen::Vector<T, Eigen::Dynamic> &mean_velocities,
                                    T wall_g_val, // Value of g at wall for zeta=0+
                                    T wall_h_val, // Value of h at wall for zeta=0+
                                    JacobianFunc get_jacobian);

        template <typename T, typename JacobianFunc>
        Eigen::Vector<T, Eigen::Dynamic> compute_temperature(
                                    const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor> &g,
                                    const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor> &h,
                                    const VelocityMesh<T> &velocity_mesh,
                                    T wall_g_val, // For velocity & temp
                                    T wall_h_val, // For temp
                                    JacobianFunc get_jacobian);

        template <typename T, typename JacobianFunc>
        T compute_temperature_at(
                                const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor> &g,
                                const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor> &h,
                                const VelocityMesh<T> &velocity_mesh,
                                const Eigen::Index i,
                                const T density_i,       // Pre-computed
                                const T mean_velocity_i, // Pre-computed
                                T wall_g_val,            // For temp
                                T wall_h_val,            // For temp
                                JacobianFunc get_jacobian);

        template <typename T, typename JacobianFunc>
        T compute_temperature_at(
                                const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor> &g,
                                const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor> &h,
                                const VelocityMesh<T> &velocity_mesh,
                                const Eigen::Index i,
                                T wall_g_val, // For internal velocity & temp
                                T wall_h_val, // For temp
                                JacobianFunc get_jacobian);
    }
}
```

3.6 `metric_utils` File

Situated within the `Bgk::metrics` namespace, the implementation utilizes a combination of the Strategy and Factory design patterns to provide a flexible system for computing vector and matrix norms. This structure plays a role in determining when the numerical scheme has reached a steady state or converged to a required tolerance by implementing the error computation described in Eq. (2.30).

At the foundation of this system is the `VectorNorm` abstract base class, which establishes a common interface for various mathematical norms acting on Eigen vectors. The design supports both standard norm calculations and weighted variants, the latter being particularly vital for simulations on non-uniform meshes where individual cell volumes or lengths must weight the error contribution of each node.

```
template <typename T>
class VectorNorm
{
public:
    virtual ~VectorNorm() = default;
    virtual T compute(const Eigen::Vector<T, Eigen::Dynamic> &v) const = 0;
    virtual T compute(const Eigen::Vector<T, Eigen::Dynamic> &v,
```

```
                        const std::vector<T> &) const { return compute(v); }
};
```

The framework (which can be easily extended) provides three concrete implementations of this interface:

- `L2VectorNorm`: implements the Euclidean norm, calculated as the square root of the sum of squares. For weighted scenarios, it computes $\sqrt{\sum v_i^2 w_i}$, allowing for a geometrically informed error metric.
- `L1VectorNorm`: Computes the Manhattan norm, or the sum of absolute values. For weighted scenarios, it computes $\sum |v_i| w_i$.
- `LinfVectorNorm`: Represents the maximum (Infinity) norm, identifying the largest absolute error across the entire vector. For weighted scenarios, it computes $\max\{|v_i| w_i\}$.

These objects are instantiated via the `VectorNormFactory`, which encapsulates the creation logic and allows the rest of the system to remain decoupled from the specific norm implementation being used.

```
template <typename T>
class VectorNormFactory
{
public:

    static std::unique_ptr<VectorNorm<T>> create(VectorNormType type)
    {
        switch (type)
        {
        case VectorNormType::L1:
            return std::make_unique<L1VectorNorm<T>>();
        case VectorNormType::L2:
            return std::make_unique<L2VectorNorm<T>>();
        case VectorNormType::Linf:
            return std::make_unique<LinfVectorNorm<T>>();
        default:
            throw std::invalid_argument("Unknown vector norm type");
        }
    }
};
```

To handle multi-dimensional data, such as the distribution functions encountered in BGK models, the library introduces the concept of row-wise matrix error quantification. This is achieved through a two-stage process involving `RowAggregator` and `MatrixErrorNorm` classes.

The `RowAggregator` interface defines how individual error values collected from different rows are combined into a single scalar metric, i.e. the function $\varphi(\cdot)$ of Eq. (2.30).

```
template <typename T>
class RowAggregator
{
public:
    virtual ~RowAggregator() = default;
    virtual T aggregate(const std::vector<T> &values) const = 0;
};
```

The `AverageRowAggregator` computes the arithmetic mean to provide a global sense of error, while the `MaxRowAggregator` focuses on the worst-case error across all rows. Similar to the vector norms, these are managed by a `RowAggregatorFactory` to maintain architectural consistency

```cpp
template <typename T>
class RowAggregatorFactory
{
public:
    static std::unique_ptr<RowAggregator<T>> create(RowAggregateType type)
    {
        switch (type)
        {
        case RowAggregateType::Average:
            return std::make_unique<AverageRowAggregator<T>>();
        case RowAggregateType::Max:
            return std::make_unique<MaxRowAggregator<T>>();
        default:
            throw std::invalid_argument("Unknown row aggregate type");
        }
    }
};
```

The `RowWiseMatrixErrorNorm` class, a specialization of the more general `MatrixErrorNorm`, serves as the primary engine for assessing the difference between successive simulation steps. It accepts two matrices—typically representing the current and previous states—and calculates a normalized error for each row. Specifically, it computes the ratio of the norm of the difference to the norm of the current row, ensuring that the error is evaluated relative to the magnitude of the solution. In our specific formulation, it computes the sequence:

$$\left\{ \frac{\|\mathbf{u}_j^k - \mathbf{u}_j^{k-1}\|}{\|\mathbf{u}_j^{k-1}\|} \right\}_{j=-\overline{N}}^{\overline{N}}$$

The final result is produced by passing these individual row metrics to a `RowAggregator`.

```cpp
template <typename T>
class RowWiseMatrixErrorNorm : public MatrixErrorNorm<T>
{
public:
    RowWiseMatrixErrorNorm(std::unique_ptr<VectorNorm<T>> vec_norm,
                           std::unique_ptr<RowAggregator<T>> aggregator)
        : vec_norm_(std::move(vec_norm)), aggregator_(std::move(aggregator)) {}

    T compute(const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic> &current,
              const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic> &prev) const override
    {
        const auto rows = static_cast<int>(current.rows());

        if (rows == 0)
            throw std::invalid_argument("RowWiseMatrixNorm: matrix has zero rows");
        std::vector<T> row_norms;

        row_norms.reserve(static_cast<size_t>(rows));
        Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic> diff = current - prev;

        for (int i = 0; i < rows; ++i)
        {
            // Copy row into a column vector for the VectorNorm interface
            Eigen::Vector<T, Eigen::Dynamic> diff_v = diff.row(i).transpose();
            Eigen::Vector<T, Eigen::Dynamic> current_v = current.row(i).transpose();
            row_norms.push_back(vec_norm_->compute(diff_v) / vec_norm_->compute(current_v));
        }
        return aggregator_->aggregate(row_norms);
    }

    T compute(const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic> &current,
              const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic> &prev,
              const std::vector<T> &weights) const override
    {...}

private:
```

```
37      std::unique_ptr<VectorNorm<T>> vec_norm_;
38      std::unique_ptr<RowAggregator<T>> aggregator_;
39  };
```

The culmination of this architectural design is the `MatrixNormFactory`. This factory serves as a high-level "object factory" that simplifies the creation of complex error-tracking objects. By specifying a `VectorNormType` and a `RowAggregateType` through the apposite enumerations, the user can instantly generate a fully configured `MatrixErrorNorm` instance. This approach ensures that the numerical scheme's convergence criteria can be swapped or modified with minimal changes to the core simulation logic, providing a robust and extensible toolset for scientific error analysis.

```
1   template <typename T>
2   class MatrixNormFactory
3   {
4   public:
5       static std::unique_ptr<MatrixErrorNorm<T>> create(VectorNormType vec_norm_type,
6                                                         RowAggregateType agg_type)
7       {
8           auto vec_norm = VectorNormFactory<T>::create(vec_norm_type);
9           auto aggregator = RowAggregatorFactory<T>::create(agg_type);
10          return std::make_unique<RowWiseMatrixErrorNorm<T>>(std::move(vec_norm), std::move(aggregator));
11      }
12  };
```

3.7 SolverFV Object

The `SolverFV` class represents the numerical core of the code by blending physical parameters, mesh geometry, and numerical utilities into an unified interface; this class is in fact responsible for advancing in time the discrete kinetic distribution functions defined on a spatial mesh by means of a finite volume formulation. The solver encapsulates the entire workflow associated with a finite volume scheme, including flux computation, time integration, collision handling, boundary condition enforcement, and post-processing output; in the following the main implementation ideas and strategies will be discussed.

At its foundation, the class maintains references to three critical components: a `ConfigData` object, which encapsulates the simulation parameters, and two specialized mesh objects, `SpaceMeshFV` and `VelocityMesh`. These meshes define the discrete domain over which the kinetic equations are solved. To represent the state of the system, the solver utilizes several primary member attributes:

- Distribution Functions (g, h): These are implemented as dynamic Row-wise stored Eigen matrices (`Eigen::Matrix`) where the rows represent velocity space and the columns represent physical space. A Row-wise storage is enforced since it's perfectly in line (thus more efficient) for the operations carried out during the solve procedure i.e. extracting and modifying rows of the g and h matrices.
- Macroscopic Quantities: Vectors (`Eigen::Vector`) for $\overline{\rho}$, $\overline{v}$ and $\overline{T}$ are maintained to store the macroscopic properties of the plasma, which are updated at each iteration through numerical integration of the distribution functions.
- Initial and Boundary conditions: The boundary and initial conditions (Eqs. (2.17a)–(2.17b) and (2.19a)– (2.19b)) are implemented as lambda expressions together with the Collision terms G and H (Eqs. (2.11a) and (2.11b)).

- Numerical Operators: The solver pre-assembles quadri-diagonal matrices A and B, stored in `Eigen::SparseMatrix`, to represent the advection operators for positive and negative velocities, respectively. A diagonal reaction matrix, R, is also stored in a `Eigen::Vector` form to enhance computational performance during the semi-implicit time integration.

While the class constructors handle the basic allocation of memory and the assignment of configuration parameters, the formal setup of the numerical problem is deferred to the `initialize()` method to ensure that all dependencies—spatial grids, velocity spaces, and operator matrices—are synchronized before the simulation begins. This method purpose is to transition the solver from a "dormant", configured state to a fully operational computational environment and it does so following a rigorous procedural sequence:

1. Mesh Construction: The method first triggers the initialization of the spatial and velocity meshes, which computes the geometric properties and cell volumes necessary for finite volume integration.

2. Initial State Definition: Once the meshes are validated, the solver populates the distribution functions g and h across the phase space according to the prescribed initial conditions and boundary profiles.

3. Macroscopic Seeding: Immediately following the distribution of particles, the solver calculates the starting physical quantities—density, mean velocity, and temperature—to ensure the collision operator has valid inputs for the first iteration.

4. Numerical Operator Assembly: The method concludes with the assembly of diagonal reaction matrix (R) and the sparse advection matrices (A and B) by retrieving the QUICK coefficients from numerical utility functions. Specific handling is provided for boundary adjacent cells (the first and last rows of the matrices), where the scheme must revert to or incorporate boundary condition data to close the system. This is further refined in the "solve helper" functions (e.g., `assemble_U_pos` for Eq. 2.25), which calculate the source and boundary correction vectors required for each velocity index in accordance to what described in Sec. 2.

```
template <typename T>
void SolverFV<T>::initialize()
{
    std::cout << "Initializing SolverFV..." << std::endl;
    initializeMeshes();
    std::cout << "Meshes initialized." << std::endl;
    setInitialState();
    std::cout << "Initial state set." << std::endl;
    assemble_A();
    assemble_B();
    assemble_R();
    std::cout << "Numerical matrices assembled." << std::endl;
    std::cout << "Assembly complete." << std::endl;
    is_initialized = true;
}
```

This design pattern allows for a "lazy initialization" approach, where the user can modify configuration parameters at runtime, only committing to the full memory and computational setup when the global `initialize()` is explicitly or implicitly called. This ensures that the numerical matrices are perfectly tailored to the final state of the meshes and physical parameters. If a custom space or velocity mesh is required the following two methods are provided:

```
template <SpacingFunction<T> custom_spacing>
void initialize_custom_spacemesh(custom_spacing &&custom_space_spacing);

template <SpacingFunction<T> custom_spacing, SpacingFunction<T> Jacobian>
void initialize_custom_velocitymesh(custom_spacing &&custom_velocity_spacing,
                                    Jacobian &&jacobian);
```

an example usage is:

```
auto custom_spacing = [Data](size_t i) -> double
{
    return i * Data.get_d2();
};
solver.initialize_custom_spacemesh(custom_spacing);
solver.initialize();
```

the usage of `initialize()` is safe because the initialization methods for the meshes have a sanity check on the `is_initialized` status.

The core solution logic is implemented in the `solve()` method, which follows a semi-implicit temporal scheme. The process is divided into three distinct phases within each time step:

1. Positive and Negative Velocity Sweeps: The method `solve_timestep_pos()` and iterate through the (positive) velocity mesh. For each velocity, the linear system represented by Eqs. (2.27) is solved. In particular:

```
template <typename T>
void SolverFV<T>::solve_timestep_pos()
{
    const size_t Velocity_N = Velocity_mesh.get_N();
    const size_t Space_N = Space_mesh.get_N();
    const T dt = Data.get_dt();

    // Positive velocities indices
    const size_t j_begin = Velocity_N + 1;
    const size_t j_end = 2 * Velocity_N;

    // Pre-calculate coefficients once
    const std::pair<T, T> a1 = numerics::QUICKcoefficients_p_at<T>(Space_mesh, 1);
    const std::pair<T, T> a2 = numerics::QUICKcoefficients_p_at<T>(Space_mesh, 2);
    const T omega1 = -a2.second - T{1} + a1.first - a1.second;

    // --- Pre-compute diagonal offsets ---
    // This maps where the diagonal elements live in the raw value array.
    // Complexity: One-time O(NNZ).
    std::vector<ptrdiff_t> diag_offsets(Space_N);

    // Ensure A is compressed to guarantee valuePtr safety
    // (Assuming A is already built; if not, call A.makeCompressed())
    if (!A.isCompressed())
        A.makeCompressed();

    for (int k = 0; k < A.outerSize(); ++k)
    {
        for (typename Eigen::SparseMatrix<T>::InnerIterator it(A, k); it; ++it)
        {
            if (it.row() == it.col())
            {
                // Store the offset from the beginning of the value array
                diag_offsets[it.row()] = &it.value() - A.valuePtr();
            }
        }
    }

    // --- Structure Reuse ---
    // Initialize M with A's pattern ONCE. We never change the pattern, only values.
    Eigen::SparseMatrix<T> M = A;
```

```
42
43      // Analyze pattern once (Symbolic Factorization)
44      Eigen::SparseLU<Eigen::SparseMatrix<T>, Eigen::NaturalOrdering<int>> solver;
45      solver.analyzePattern(A);
46
47      // Buffers
48      Eigen::Vector<T, Eigen::Dynamic> U(Space_N);
49      Eigen::Vector<T, Eigen::Dynamic> W(Space_N);
50      Eigen::Vector<T, Eigen::Dynamic> rhs(Space_N); // Reused for both g and h
51
52      // Pre-fetch constant vector parts
53      const Eigen::Vector<T, Eigen::Dynamic> R_loc = R.tail(Space_N);
54      const T *A_vals = A.valuePtr(); // Read-only source
55      T *M_vals = M.valuePtr();       // Write target
56      const Eigen::Index nnz = A.nonZeros();
57
58      for (size_t j = j_begin; j <= j_end; ++j)
59      {
60          // 1. Matrix Assembly: M = (v_j * dt) * A + (I + dt * R)
61          const T v_dt = Velocity_mesh[j] * dt;
62
63          // Step A: Reset M values by scaling A (Vectorized copy)
64          // This is much faster than M = A (which copies indices too) followed by M *= scalar
65          for (Eigen::Index k = 0; k < nnz; ++k)
66          {
67              M_vals[k] = A_vals[k] * v_dt;
68          }
69
70          // Step B: Update diagonal (Direct access via offsets)
71          for (size_t i = 0; i < Space_N; ++i)
72          {
73              M_vals[diag_offsets[i]] += (dt * R_loc[i] + T{1});
74          }
75
76          // 2. Numerical Factorization
77          solver.factorize(M);
78          if (solver.info() != Eigen::Success)
79              throw std::runtime_error("LU factorization failed in solve_timestep_pos");
80
81          // 3. Assemble RHS vectors
82          U = assemble_U_pos(j, a1.second, a2.second, omega1);
83          W = assemble_W_pos(j, a1.second, a2.second, omega1);
84
85          // 4. Solve for g
86          rhs = g.row(j).tail(Space_N).transpose();
87          rhs += dt * U;
88          auto x = solver.solve(rhs);
89          if (solver.info() != Eigen::Success)
90              throw std::runtime_error("Solve (g) failed");
91          g.row(j).tail(Space_N) = x.transpose();
92
93          // 5. Solve for h
94          rhs = h.row(j).tail(Space_N).transpose();
95          rhs += dt * W;
96          auto y = solver.solve(rhs);
97          if (solver.info() != Eigen::Success)
98              throw std::runtime_error("Solve (h) failed");
99          h.row(j).tail(Space_N) = y.transpose();
100     }
101 }
```

The efficiency of this functions relies heavily on the fact that the sparse matrices A and $M = I + \Delta t R + \Delta t \zeta^{(j)} A$ have the same non-zero pattern (quadri-diagonal) and this pattern doesn't change during the loop; this simple but crucial fact lets us employ several shortcuts. Firstly, the code is able to call `analyzePattern` once outside the loop; inside the loop it only calls `factorize` effectively skipping the expensive (and redundant in our case) structural analysis for every velocity step.

Furthermore, the sparse matrix operation `M = A * dt * v[j]`, which would generally im-

ply a check on M reserved space, a copy of both the innerIndices and outerPointers and then the multiplication and copy of the actual values, can be avoided. Instead, we iterate directly over the contiguous value array via `valuePtr()` performing a single fused multiply: `M_vals[k] = A_vals[k] * v_dt`. This loop is also a sequential read-write over a flat memory region, which is both cache friendly and amenable to compiler auto-vectorization (SIMD). The result is a $O(\texttt{nnz})$ pass with minimal memory traffic and no reduntant structural copies.

Finally, performing `M += I + dt * R` (an operation that acts on the diagonal entries) would require the machine to search through each row's non-zero entries to find the one whose column index matches its row index ($O(\log k)$ with k being the number of non zero elements in the column). To eliminate this, the offset of every diagonal entry within the value array is computed once, before the velocity loop begins, and stored in a precomputed offset vector. During assembly, diagonal updates reduce to direct indexed writes `M_vals[diag_offsets[i]] += (...)` each of which is $O(1)$.

Together, the two techniques reduce each matrix assembly to two lightweight linear passes: one over all nonzero entries for the scaling step, and one over the diagonal entries for the update step. No memory is allocated, no sparsity structure is copied or searched, and both loops are amenable to hardware-level optimisation.

The `solve_timestep_neg()` for the negative velocity system represented by Eqs. (2.28) is analogously implemented.

2. Zero-Velocity Treatment: A specialized `solve_timestep_zero()` method handles the case where $\zeta^{(j)} = 0$, reducing the partial differential equation (Eq. (2.29)) to a purely algebraic update that can be solved with a simple element-wise division.

```cpp
template <typename T>
void SolverFV<T>::solve_timestep_zero()
{
    const size_t Velocity_N = Velocity_mesh.get_N();
    const size_t Space_N = Space_mesh.get_N();
    const T dt = Data.get_dt();

    // 1. Assemble sources
    Eigen::Vector<T, Eigen::Dynamic> U = assemble_U_zero();
    Eigen::Vector<T, Eigen::Dynamic> W = assemble_W_zero();

    // 2. Pre-calculate Inverse Denominator
    Eigen::Vector<T, Eigen::Dynamic> inv_denom =
        (Eigen::Vector<T, Eigen::Dynamic>::Ones(Space_N) + dt * R.head(Space_N)).cwiseInverse();

    // 3. Update 'g' in-place (No intermediate allocations)
    g.row(Velocity_N).head(Space_N) = (g.row(Velocity_N).head(Space_N).transpose() + dt * U)
                                        .cwiseProduct(inv_denom)
                                        .transpose();

    // 4. Update 'h' in-place
    h.row(Velocity_N).head(Space_N) = (h.row(Velocity_N).head(Space_N).transpose() + dt * W)
                                        .cwiseProduct(inv_denom)
                                        .transpose();
}
```

3. Macroscopic Convergence: After distribution functions are updated, macroscopic quantities are recalculated, and the reaction terms are re-assembled. Convergence is monitored by evaluating the relative error between iterations using a configurable matrix norm (Sec. 3.6), continuing until the system reaches a steady state or the maximum iteration limit is met.

```cpp
template <typename T>
template <PlotStrategy Strategy>
void SolverFV<T>::solve(const metrics::VectorNormType vec_norm_type,
                        const metrics::RowAggregateType agg_type)
{
    if (!is_initialized)
        initialize();

    size_t max_iter = Data.get_max_iter();
    size_t k = 0;
    T tol = Data.get_tol();
    T rel_err = std::numeric_limits<T>::max();

    size_t plot_every_k_steps = 0;
    if constexpr (Strategy == PlotStrategy::EACHSTEP)
    {
        plot_every_k_steps = Data.get_plot_every_k_steps();
        std::cout << "Plotting every " << plot_every_k_steps << " steps." << std::endl;
        write_phys_instant(Data.get_saving_folder_name(), 0);
    }

    auto mat_norm = metrics::MatrixNormFactory<T>::create(vec_norm_type, agg_type);

    Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor> g_old, h_old;

    while (k < max_iter && rel_err > tol)
    {
        g_old = g;
        h_old = h;

        solve_timestep_neg();
        solve_timestep_zero();
        solve_timestep_pos();

        set_physical_quantities();
        assemble_R();

        // Error calculation logic
        rel_err = std::sqrt(std::pow(mat_norm->compute(g, g_old, Space_mesh.get_volume_sizes()), T{2}) +
                            std::pow(mat_norm->compute(h, h_old, Space_mesh.get_volume_sizes()), T{2}));

        ++k;

        // Compile-time branching
        if constexpr (Strategy == PlotStrategy::EACHSTEP)
        {
            if (k % plot_every_k_steps == 0  k == 1 /* ... other conditions ... */)
            {
                write_phys_instant(Data.get_saving_folder_name(), k);
            }
        }
    }

    // Always plot at the end regardless of strategy
    write_phys_txt(Data.get_saving_folder_name());

    std::cout << "Solver finished after " << k << " iterations with relative error " << rel_err << std::endl;
    return;
}
}
```

Finally, to facilitate the analysis of the simulation results, the class provides a suite of output methods. `write_sol_txt()` and `write_phys_txt()` export the raw distribution and macroscopic data into structured text files. The inclusion of `write_phys_instant()` allows for the recording of "snapshots" during the simulation, which is essential for diagnosing the convergence behavior and transient phases of the BGK model.

3.7.1 Parallelization

In Sec. 2 a great stress has been put on the fact that the equations of systems (2.27) and (2.28) are independent in the velocity space $\zeta^{(j)}$, that's because this characteristic makes them prone to parallelization. In the following the parallel version of `solve_timestep_pos()` (and it's negative counterpart) implemented using `OpenMP` will be discussed.

```
    template <typename T>
    void SolverFV<T>::solve_timestep_pos_parallel()
    {
        const size_t Velocity_N = Velocity_mesh.get_N();
        const size_t Space_N = Space_mesh.get_N();
        const T dt = Data.get_dt();

        // Positive velocities indices
        const size_t j_begin = Velocity_N + 1;
        const size_t j_end = 2 * Velocity_N;

        // Pre-calculate coefficients once (Shared)
        const std::pair<T, T> a1 = numerics::QUICKcoefficients_p_at<T>(Space_mesh, 1);
        const std::pair<T, T> a2 = numerics::QUICKcoefficients_p_at<T>(Space_mesh, 2);
        const T omega1 = -a2.second - T{1} + a1.first - a1.second;

        // Ensure A is compressed for raw pointer access
        if (!A.isCompressed())
            A.makeCompressed();

        // --- Pre-compute diagonal offsets (Shared) ---
        // This map is valid for any matrix with A's pattern (including thread-local M)
        std::vector<ptrdiff_t> diag_offsets(Space_N);
        const T *A_global_ptr = A.valuePtr(); // Pointer to shared A values

        for (int k = 0; k < A.outerSize(); ++k)
        {
            for (typename Eigen::SparseMatrix<T>::InnerIterator it(A, k); it; ++it)
            {
                if (it.row() == it.col())
                {
                    diag_offsets[it.row()] = &it.value() - A_global_ptr;
                }
            }
        }

        // Shared pointers/refs for loop access
        const Eigen::Vector<T, Eigen::Dynamic> &R_ref = R;
        const Eigen::Index nnz = A.nonZeros();

        // Exception capture for OpenMP
        std::atomic<bool> error_occurred{false};

// --- Parallel Region ---
// Note: We create thread-local resources HERE, outside the loop
#pragma omp parallel
        {
            // 1. Thread-Local Allocations
            Eigen::SparseMatrix<T> M_loc = A;

            // Solver instance for this thread
            Eigen::SparseLU<Eigen::SparseMatrix<T>, Eigen::NaturalOrdering<int>> solver_loc;
            solver_loc.analyzePattern(A); // Analyze pattern ONCE per thread

            // Thread-local Buffers
            Eigen::Vector<T, Eigen::Dynamic> U_loc(Space_N);
            Eigen::Vector<T, Eigen::Dynamic> W_loc(Space_N);
            Eigen::Vector<T, Eigen::Dynamic> rhs_loc(Space_N);

            // Pointers for fast access
            T *M_vals = M_loc.valuePtr();
            const Eigen::Vector<T, Eigen::Dynamic> R_loc = R_ref.tail(Space_N);

```

```
64  // 2. Parallel Loop
65  // schedule(dynamic) is usually better if solve times vary,
66  // otherwise schedule(static) has lower overhead.
67  #pragma omp for schedule(static)
68              for (size_t j = j_begin; j <= j_end; ++j)
69              {
70                  if (error_occurred)
71                      continue; // Skip work if another thread failed
72
73                  try
74                  {
75                      // --- Matrix Assembly ---
76                      const T v_dt = Velocity_mesh[j] * dt;
77
78                      // Step A: Vectorized reset using raw pointers
79                      // We use A_global_ptr (Shared read-only) to reset M_vals (Thread-local)
80                      for (Eigen::Index k = 0; k < nnz; ++k)
81                      {
82                          M_vals[k] = A_global_ptr[k] * v_dt;
83                      }
84
85                      // Step B: Update diagonal using pre-calc offsets
86                      for (size_t i = 0; i < Space_N; ++i)
87                      {
88                          M_vals[diag_offsets[i]] += (dt * R_loc[i] + T{1});
89                      }
90
91                      // --- Factorization ---
92                      solver_loc.factorize(M_loc);
93                      if (solver_loc.info() != Eigen::Success)
94                          throw std::runtime_error("LU factorization failed");
95
96                      // --- RHS Assembly ---
97                      U_loc = assemble_U_pos(j, a1.second, a2.second, omega1);
98                      W_loc = assemble_W_pos(j, a1.second, a2.second, omega1);
99
100                     // --- Solve for g ---
101                     // @note: .row(j) write is thread-safe as j is unique per thread
102                     rhs_loc = g.row(j).tail(Space_N).transpose();
103                     rhs_loc += dt * U_loc;
104
105                     auto x = solver_loc.solve(rhs_loc);
106                     if (solver_loc.info() != Eigen::Success)
107                         throw std::runtime_error("Solve g failed");
108                     g.row(j).tail(Space_N) = x.transpose();
109
110                     // --- Solve for h ---
111                     rhs_loc = h.row(j).tail(Space_N).transpose();
112                     rhs_loc += dt * W_loc;
113
114                     auto y = solver_loc.solve(rhs_loc);
115                     if (solver_loc.info() != Eigen::Success)
116                         throw std::runtime_error("Solve h failed");
117                     h.row(j).tail(Space_N) = y.transpose();
118                 }
119                 catch (...)
120                 {
121                     error_occurred = true;
122                 }
123             }
124         } // End of parallel region
125
126         if (error_occurred)
127             throw std::runtime_error(error_message("An error occurred during parallel \
128                                                    execution of solve_timestep_pos."));
129
130         return;
131     }
```

To ensure thread safety and minimize overhead, a thread-privatization strategy was adopted: the `#pragma omp parallel` is opened before the loop; this allows us to allocate "heavy" objects

(the M matrix copy, the SparseLU solver, and vectors U, W and rhs) once per thread, rather than once per iteration while isolating memory management and preventing race conditions. Instead A, diag_offsets, R and Data remain shared given that their role is "read only".

As for the schedule schedule(static) was chosen since the sparsity pattern is constant across $\zeta^{(j)}$. Therefore no j^{th} solve should take more than the others or, at least, not in a way that would cause remarkable waiting to the remaining threads thus justifying a dynamic schedule.

The parallel solver is then called with:

```
template <typename T>
template <PlotStrategy Strategy>
void SolverFV<T>::solve_parallel(const metrics::VectorNormType vec_norm_type,
                                 const metrics::RowAggregateType agg_type)
{...}
```

4 Time Development of the solution

Depending on the gas and condensed phase conditions, either condensation or evaporation takes place, inducing a disturbance that propagates into the gas. In particular, the condensed phase emits molecules into the gas at a rate determined by the surface temperature (T_w) and the saturation pressure (p_w) while the gas "sends" molecules toward the surface at a rate that depends on the state of the gas at infinity ($p_\infty, T_\infty, M_\infty$). Whether the net process is condensation or evaporation depends on which of these two fluxes is stronger.

For condensation to occur, there must be a net flux of mass from the gas into the condensed phase which implies that the macroscopic velocity of the gas is directed toward the surface ($v < 0$). However, as we will shortly see, simply blowing the gas toward the surface is not sufficient to guarantee condensation: the pressure ratio (p_∞/p_w) is critical; if the pressure is too low evaporation can still happen. Similarly, in steady evaporation the net mass flux is outward so the gas velocity is directed away from the condensed phase ($v > 0$) however, if the pressure ration (p_∞/p_w) is sufficiently high.

After a certain period of time, a steady condensation or evaporation flow will be established with precise parameters at infinity [1, 10, 13].

4.1 Condensation

The key to understanding the time development of the solution lies in the interaction between the uniform equilibrium flow, characterized by the set ($p_\infty, T_\infty, M_\infty$), and the plane condensed phase ($x_1 = 0$) where molecules leaving the surface are governed by the stationary Maxwellian distribution corresponding to the saturated gas at the surface temperature T_w and saturation pressure p_w.

The immediate encounter between the uniform incoming flow and the outgoing flux creates an initial discontinuity in the velocity distribution function. This discontinuity generates a disturbance—a region of highly non-equilibrium-flow—near the wall in the Knudsen layer. The nature of this initial disturbance is critical and is classified based on whether the local conditions near the wall show:

- Compression: The flow interaction causes the gas density and pressure near the condensed phase to increase
- Rarefaction: The flow interaction causes the gas density and pressure near the condensed phase to decrease

Once generated, the disturbance begins its time development, propagating or diffusing upstream into the semi-infinite expanse of gas. The manner in which this disturbance evolves determines the final steady state of the system and falls into one of the four solution types [11]:

I. A compression region forms, but its speed of propagation slows down and finally vanishes. The disturbance is confined to the neighborhood of the condensed phase, meaning the compression wave is "pushed back to the condensed phase and merged with the Knudsen layer". A steady state compatible with the data at infinity and characterized by a supersonic speed at infinity ($M_\infty > 1$) is established

II. The compression disturbance is strong enough to develop into a compression wave (shock wave) that propagates up to upstream infinity. Because this shock wave alters the upstream region, the flow behind the wave approaches a steady state with a new subsonic state at infinity. In this regime, the initial prescribed data ($M_\infty,\ p_\infty/p_w,\ T_\infty/T_w$) cannot support a steady solution

III. A rarefaction region forms, which diffuses but does not reach upstream infinity. A steady state is finally established with the prescribed $(M_\infty,\ p_\infty/p_w,\ T_\infty/T_w)$ at infinity, also characterized by a supersonic speed at infinity.

IV. The rarefaction disturbance cannot be confined and it develops into an expansion wave that propagates up to upstream infinity. The region behind this wave approaches a steady state with a new subsonic or sonic state at infinity.

Thus in the Type I and III regimes the upstream flow conditions $(M_\infty,\ p_\infty/p_w,\ T_\infty/T_w)$ are compatible with the boundary condition p_w, T_w such that the flow can settle into a steady state without altering the conditions at infinity. For the Type II and IV the initial upstream conditions are not compatible with the flow physics required for a steady state, leading to a permanent adjustment of the upstream conditions.

The boundary between these regimes is crucial because it marks the precise set of upstream parameters where a steady solution matching $(M_\infty,\ p_\infty/p_w,\ T_\infty/T_w)$ is possible. The schematic view of the cross section at $T_\infty/T_w = \text{const}$ for the three-dimensional space of prescribed data is shown in Fig. 7

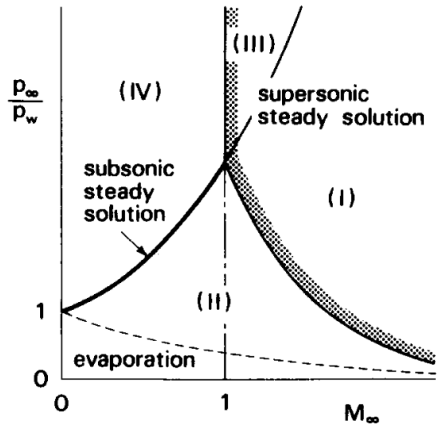


Fig. 7. Schematic view of the map of the solution type on the cross section $T_\infty/T_w = \text{const}$ in the three-dimensional space $(M_\infty,\ p_w/p_\infty,\ T_w/T_\infty)$ [1]

We can observe that the boundary between the regions I and II and that between regions II and IV intersect on $M_\infty = 1$. For p_∞/p_w larger than the intersection value region III solutions appear between regions I and IV.

As mentioned before the steady, solution with prescribed data at infinity exists in regions I and III (supersonic steady solution) moreover, it also exists on the boundary between region II and IV. That's because as the point in region II (IV) approaches the boundary between regions II and IV, the amplitude of the shock (expansion) wave decreases and vanishes on the boundary. Therefore no finite disturbance propagates up to upstream infinity, and the steady state with the prescribed data at infinity (subsonic steady solution) is formed[8].

Finally, region II is subdivided into two regions. For p_∞/p_w smaller than some constant that depends on M_∞ the gas evaporates from the condensed phase, although the gas flow is blowing toward the condensed phase from infinity (cf. Fig. 7). The evaporation problem is studied in [11, 12]. The solution approaches a steady solution of evaporation with subsonic or sonic speed

[8]Therefore, for a given T_w (and p_w), a subsonic steady solution (that preserves the data at infinity) is determined by specifying two parameters at infinity (e.g., T_∞, p_∞), whereas a supersonic steady solution is determined by the three parameters there (T_∞, p_∞, v_∞).

at infinity. In any case, a compression wave propagates up to upstream infinity, and no steady solution with the prescribed data $(M_\infty,\ p_w/p_\infty,\ T_w/T_\infty)$ exists in region II.

The map of the type of the time development of solution in the cross section $T_\infty/T_w = 1$ of the three dimensional space $(M_\infty,\ p_w/p_\infty,\ T_w/T_\infty)$ is shown in Fig 8 where the symbols ∘, •, △, ▲ stand for I-, II-, III-, IV-type solutions respectively [Photoshoppare sta immagine con i punti].

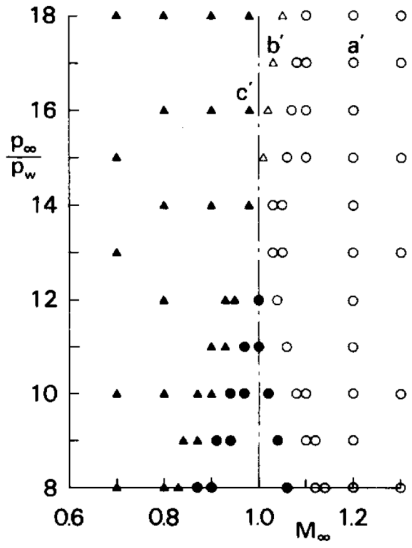


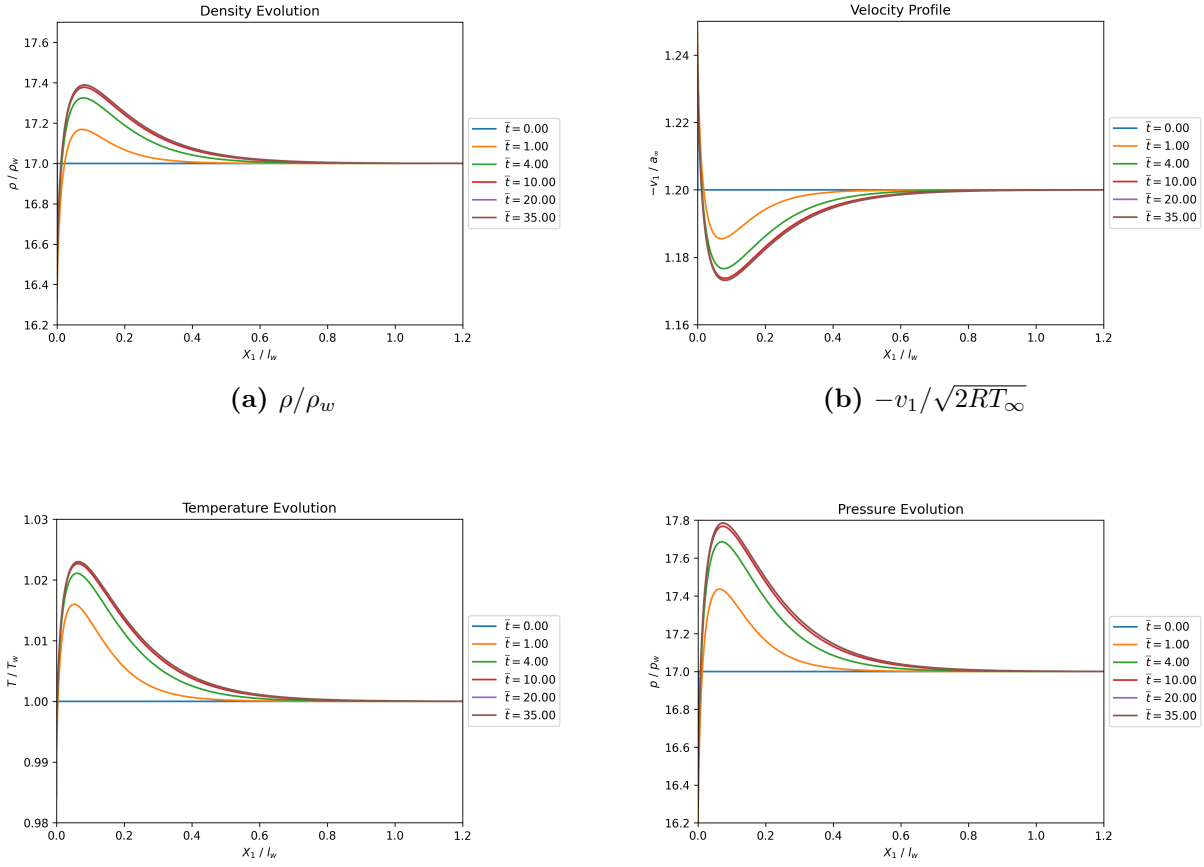
Fig. 8. Type of solution on the cross section $T_w/T_\infty = 1$ of the $(M_\infty,\ p_w/p_\infty,\ T_w/T_\infty)$ space. Here ∘, •, △, ▲ indicate the I-, II-, III-, and IV-type solutions, respectively [1].

The profiles corresponding to points a', b', c' and d' in Fig. 8 are shown respectively in Fig. 9, 10, 11 and 12.

To gain more insight from the plot we remember that in a gas flow, the total energy is distributed among several contributions. Primarily the thermodynamic energy comprising internal energy and pressure–volume work, here represented by T/T_w and p/p_w, and the macroscopic kinetic energy associated with the bulk velocity v_1 of the flow. Energy conservation governs the exchange between these forms.

For Type I solutions (Fig. 9), as the gas molecules interact with the boundary or with an increasingly dense region of gas near it, the bulk flow is progressively decelerated. The resulting reduction in macroscopic kinetic energy is converted into random molecular motion, which manifests macroscopically as an increase in temperature. At the same time, molecular compression leads to higher density and pressure. A similar mechanism is observed for Type II solutions (Fig. 10), where a shock wave propagates upstream: downstream of the shock, the gas is characterized by reduced velocity and elevated temperature and density.

In contrast, Type III and IV solutions correspond to rarefaction processes, in which the gas undergoes expansion (Fig. 11 and 12). Such behavior typically arises when the gas is drawn toward the condensed phase more rapidly than it is replenished, or when it expands into a region of lower pressure. During rarefaction, the molecules become more widely spaced, leading to a drop in density and pressure; this is also accompanied by a reduction in random thermal motion, resulting in a decrease in temperature. Thus, thermodynamic energy stored in the form of pressure and internal energy is converted into macroscopic kinetic energy, causing the gas to accelerate.



(a) ρ/ρ_w **(b)** $-v_1/\sqrt{2RT_\infty}$

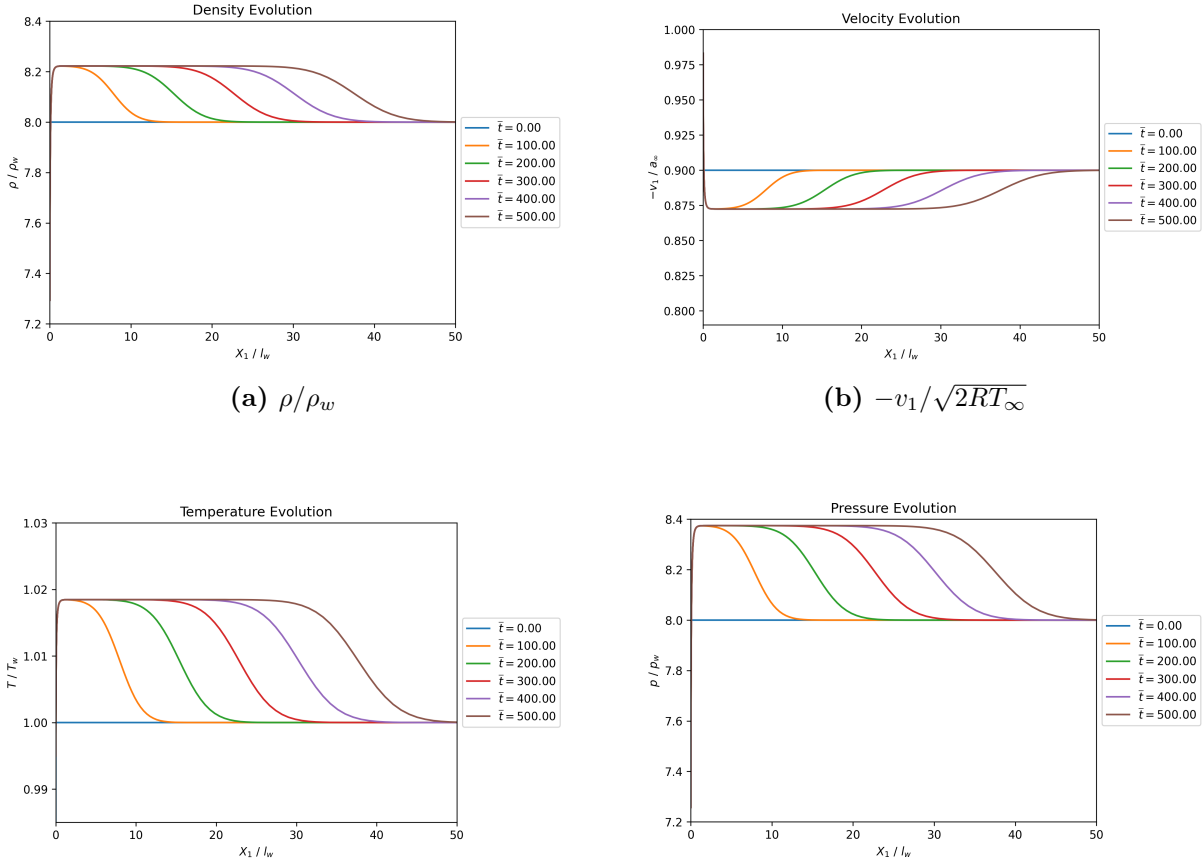
(c) T/T_w **(d)** p/p_w

Fig. 9. Time development of the solution corresponding to a' in Fig. 8: $M_\infty = -1.2$, $p_\infty/p_w = 17$ and $T_\infty/T_w = 1$. Data on the lattice system: $\Delta t = 5 \times 10^{-2}$; $d_1 = 5 \times 10^{-4}$; $d_2 = 0.1$; $N + 2 = N_0 = 399$; $a_1 = 1 \times 10^{-2}$; $a_2 = 2.5 \times 10^{-5}$; $\overline{N} = 60$

Proceeding with our analysis, from the previous section we've understood that a subsonic steady solution is possible only if we are in two specific scenarios:

- If the initial set of parameters $(M_\infty,\ p_\infty/p_w,\ T_\infty/T_w)$ lies on the surface dividing the II and IV region. In such case a steady state with the prescribed data at infinity is formed.
- If the initial set of parameters $(M_\infty,\ p_\infty/p_w,\ T_\infty/T_w)$ lies fully in the II or IV region. In such case a steady state with a new subsonic (or sonic in the limiting case) state is formed at infinity.

The interesting fact suggested by the numerical results of the previous chapter and already put forward in [11, 13] is that in the second case the new values at infinity land exactly on the "subsonic solution surface". What could be concluded then is that for the steady state a subsonic gas flow condensing onto a surface, you cannot arbitrarily choose the conditions at infinity, the three parameters $(M_\infty, p_\infty, T_\infty)$ must satisfy a specific relationship $p_\infty/p_w = F_s(M_\infty, T_\infty/T_w)$. A numerical derivation of this surface is proposed in [1] while a theoretical derivation based on the Moments Method [8] is presented in [4, 13].



(a) ρ/ρ_w **(b)** $-v_1/\sqrt{2RT_\infty}$

(c) T/T_w **(d)** p/p_w

Fig. 10. Time development of the solution corresponding to b' in Fig. 8: $M_\infty = -0.9$, $p_\infty/p_w = 8$ and $T_\infty/T_w = 1$. Data on the lattice system: $\Delta t = 5 \times 10^{-2}$; $d_1 = 5 \times 10^{-3}$; $d_2 = 1.0$; $N = 220$; $N_0 = 150$; $a_1 = 2 \times 10^{-2}$; $a_2 = 5 \times 10^{-5}$; $\overline{N} = 60$

4.2 Evaporation

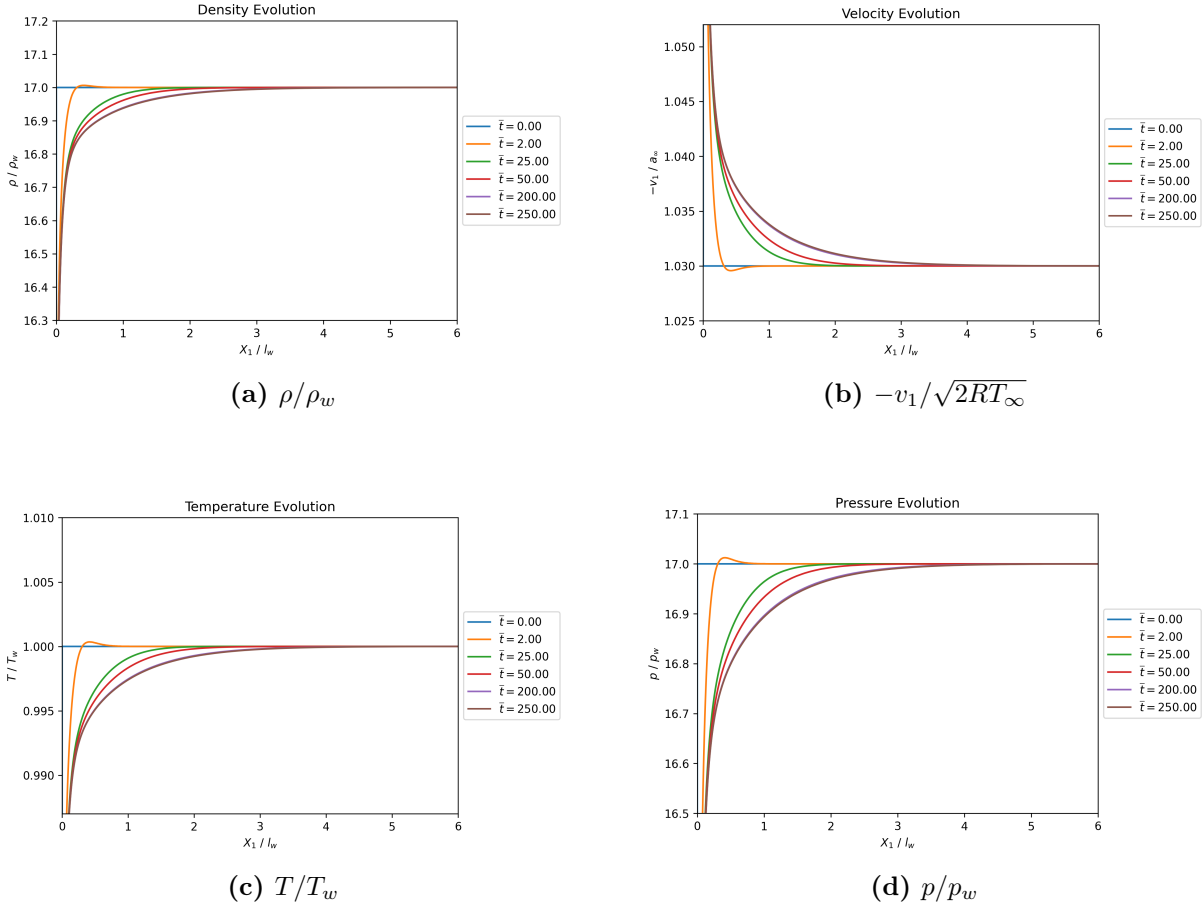
Also in the case of evaporation, upon the initiation of the process, a discontinuity at the interface resolves into a disturbance that propagates into the semi-infinite gas region, typically separating into distinct hydrodynamic features: a shock wave or expansion wave, a contact layer, and a Knudsen layer adjacent to the condensed phase [10].

In [13] Ytrehus presents a closed-form steady solution for the problem of strong evaporation from a planar surface. This solution is derived using a four-moment method (an extension of the Hertz-Knudsen and Schrage approaches) [8, 13] and provides explicit formulas for the coupling between the interphase surface conditions and the external (downstream) vapor flow and for the molecular distribution function $f(X_1, \boldsymbol{\xi})$.

The author models the vapor Knudsen layer as a transition between the molecules emitted from the surface and those in the downstream equilibrium state. The analytical formula (tri-modal ansatz) for the distribution function at any distance X_1 from the surface is:

$$f(X_1, \boldsymbol{\xi}) = a_e^+(X_1) f_e^+(\boldsymbol{\xi}) + a_\infty^+(X_1) f_\infty^+(\boldsymbol{\xi}) + a_\infty^-(X_1) f_\infty^-(\boldsymbol{\xi}) \tag{4.1}$$

where $f_e^+(\boldsymbol{\xi})$ is the half range Maxwellian distribution of molecules emitted from the surface at the condensed phase temperature T_w (for $\xi_1 > 0$) while $f_\infty^+(\boldsymbol{\xi})$ and $f_\infty^-(\boldsymbol{\xi})$ are the half-range components ($\xi_1 > 0$ and $\xi_1 < 0$, respectively) of the downstream external Maxwellian f_∞. The



(a) ρ/ρ_w **(b)** $-v_1/\sqrt{2RT_\infty}$

(c) T/T_w **(d)** p/p_w

Fig. 11. Time development of the solution corresponding to c' in Fig. 8: $M_\infty = -1.03$, $p_\infty/p_w = 17$ and $T_\infty/T_w = 1$. Data on the lattice system: $\Delta t = 5 \times 10^{-2}$; $d_1 = 5 \times 10^{-4}$; $d_2 = 0.25$; $N + 2 = N_0 = 399$; $a_1 = 1 \times 10^{-2}$; $a_2 = 9 \times 10^{-5}$; $\overline{N} = 36$

$a_e^+(X_1)$, $a_\infty^+(X_1)$ and $a_\infty^-(X_1)$ functions describe how the different modes evolve through the Knudsen layer.

By integrating the general conservation equations (Eq. (1.13)–(1.14)) over the space X_1, inserting Eq. (4.1) and then performing the half range integrations in velocity space, the solution identifies the external vapor state as being controlled by a single parameter, the local mach number at infinity $S_\infty = v_\infty/\sqrt{2RT_\infty}$ $(= \sqrt{5/6}\, M_\infty)$. The primary closed form results for the external state (T_∞, p_∞) relative to the surface state (T_w, p_w) are given as follows:

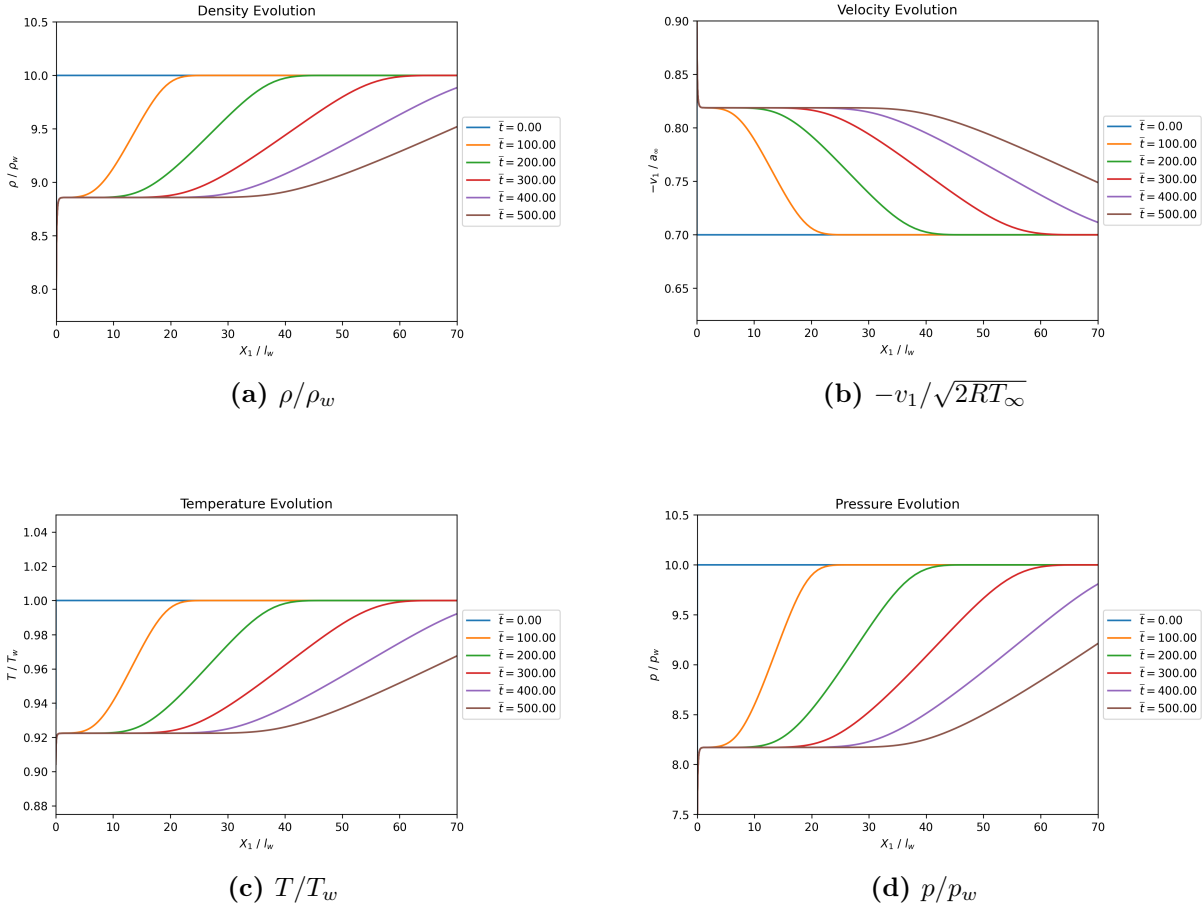
$$\sqrt{\frac{T_\infty}{T_w}} = -\frac{\sqrt{\pi}}{8} S_\infty + \sqrt{1 + \frac{\pi}{64} S_\infty^2} \tag{4.2}$$

$$\frac{p_w}{p_\infty} = \frac{2e^{-S_\infty^2}}{F^-(S_\infty) + G^-(S_\infty)\sqrt{\frac{T_\infty}{T_w}}} \tag{4.3}$$

$$\frac{\rho_\infty}{\rho_w} = \frac{p_\infty}{p_w} \cdot \frac{T_\infty}{T_w}$$

Some computed values can be found in Table 1. As for the amplitudes, they are given as:

1. $a_e^+(X_1) = \dfrac{a_\infty^-(X_1) - 1}{\beta - 1}$



(a) ρ/ρ_w **(b)** $-v_1/\sqrt{2RT_\infty}$

(c) T/T_w **(d)** p/p_w

Fig. 12. Time development of the solution corresponding to d' in Fig. 8: $M_\infty = -0.7$, $p_\infty/p_w = 10$ and $T_\infty/T_w = 1$. Data on the lattice system: $\Delta t = 5 \times 10^{-2}$; $d_1 = 5 \times 10^{-3}$; $d_2 = 1.0$; $N = 240$; $N_0 = 150$; $a_1 = 2 \times 10^{-2}$; $a_2 = 6 \times 10^{-5}$; $\overline{N} = 60$

2. $a_\infty^+(X_1) = \dfrac{\beta - a_\infty^-(X_1)}{\beta - 1}$

3. $a_\infty^-(X_1)$ is determined as the solution to the following a non conserved moment equation:

$$\frac{da_\infty^-(X_1)}{dX_1} = -\frac{P}{l_w}(a_\infty^-(X_1) - 1)(a_\infty^-(X_1) - r)$$

The case $M_\infty \leq 1$ results in:

$$\frac{a_\infty^-(X_1) - 1}{\beta - 1} = \left(\frac{a_\infty^-(X_1) - r}{\beta - r}\right) \exp\left(-P(1-r)\frac{X_1}{l_w}\right)$$

which gives us:

$$a_\infty^-(X_1) = \frac{(\beta - r) - r(\beta - 1)E(X_1)}{(\beta - r) - (\beta - 1)E(X_1)} \tag{4.4}$$

with $E(X_1) = \exp\left(-P(1-r)\frac{X_1}{l_w}\right)$. In the sonic case ($M_\infty = 1, r = 1$) instead, the dependence becomes a slower algebraic relaxation:

$$a_\infty^-(X_1) = 1 + \frac{\beta - 1}{1 + P(\beta - 1)\frac{X_1}{l_w}}$$

this implies that for sonic evaporation, the Knudsen layer is significantly thicker, requiring a distance of roughly $20\, l_w$ for relaxation to be completed.

S_∞	p_w/p_∞	ρ_∞/ρ_w	T_∞/T_L
0.0	1.000	1.000	1.000
0.1	1.231	0.849	0.957
0.2	1.500	0.728	0.915
0.3	1.812	0.630	0.876
0.4	2.170	0.550	0.838
0.5	2.577	0.484	0.802
0.6	3.037	0.429	0.767
0.7	3.553	0.383	0.734
0.8	4.127	0.345	0.703
0.9	4.764	0.312	0.673
0.907	4.813	0.310	0.671

Table 1. Flow properties as a function of S_∞.

The parameters at play are defined in the following way: β (Backscatter Factor) is the value of $a_\infty^-(0)$, representing the amplitude of the molecules moving toward the surface at the interface. It varies from $\beta \approx 1$ at the equilibrium to $\beta \approx 6.13$ at the sonic limit and it's defined as:

$$\beta = \frac{2(2S_\infty^2+1)\sqrt{T_\infty/T_w} - 2\sqrt{\pi}S_\infty}{F^-(S_\infty) - \sqrt{T_\infty/T_w}G^-(S_\infty)}$$

where $F^-(S_\infty)$ and $G^-(S_\infty)$ are two auxiliary function defined, using the notation $\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$, as:

$$F^-(S_\infty) = e^{-S_\infty^2} - \sqrt{\pi}S_\infty \mathrm{erfc}(S_\infty)$$
$$G^-(S_\infty) = (2S_\infty^2+1)\,\mathrm{erfc}(S_\infty) - \frac{2}{\sqrt{\pi}}S_\infty e^{-S_\infty^2}$$

The Collision Parameter r is the second root of the quadratic moment equation. It represents the "lower limit" toward which the amplitude function would move if the downstream state were not fixed at unity. It is defined as:

$$r = 1 - \frac{2}{\phi_1} + \frac{4S_\infty^2}{\phi_2}$$

and it ranges from 0.617 at the equilibrium $S_\infty = 0$ to exactly 1.0 at the sonic limit. The parameter P is a positive coefficient that determines the overall rate of spatial decay (the "slope" of the Knudsen layer profile). It is defined based on the gas-dynamic ratios as:

$$P = \frac{\pi}{12}\frac{\rho_\infty}{\rho_w}(\beta-1)^2\frac{\phi_1\phi_2}{(p_w/p_\infty)(1-T_\infty/T_w)}$$

Both are defined by means of the two intermediate functions:

$$\phi_1 = \frac{(\rho_w/\rho_\infty - 2 + \beta\,\mathrm{erfc}(S_\infty))}{\beta-1}$$
$$\phi_2 = \frac{(p_w/p_\infty - 2 + \beta\,\mathrm{erfc}(S_\infty))}{\beta-1}$$

Looking now at the amplitude coefficients we see that at the interface ($X_1 = 0$) $a_e^+(0) = 1$, this ensures that for $\xi_1 > 0$ the vapor perfectly mathes the wall's maxwellian f_e^+; $a_\infty^+(0) = 0$ which indicates that the downstream state has not yet contributed to the molecules moving away from the wall. Finally, as mentioned above, $a_\infty^-(0) = \beta$, this means that in the Ytreus model the

unknown distribution of molecules returning to the wall ($\xi_1 > 0$ at $X_1 = 0$), which normally would be approximated by the numerical scheme, is approximated as the dowstream distribution f_∞^- scaled by the factor β.

At the far field ($X_1 \to \infty$) instead, we must have that $a_e^+ \to 0$, $a_\infty^+ \to 1$ and $a_\infty^- \to 1$ to be coherent with the fact that the effect of the wall-emitted molecules disappears as they collide and equilibrate while the total distribution f becomes the uniform maxwellian f_∞. Since $\beta > 1$ expect in the case of net mass transfer (for which $S_\infty = 0$ and $\beta = 1$) this can happen only if $r \leq 1$; hence we have the condition:

$$\frac{2}{\phi_1} - \frac{4S_\infty^2}{\phi_2} \geq 0$$

This equality is satisfied for $M_\infty \in [0, 0.994]$ ($S_\infty \in [0, 0.907]$) [13] thus suggesting that the critical upper limit for the existence of Knudsen Layer solutions is $M_\infty = 1$.

Summarizing, a fundamental constraint revealed through kinetic theory analysis is that the resulting steady evaporation flow is strictly limited to the subsonic regime ($M_\infty \leq 1$) [7, 10]. Unlike the condensation problem, which admits supersonic steady solutions, no steady solution exists for supersonic evaporation; if a supersonic region forms during the transient phase, it propagates away from the condensed phase and vanishes at infinity, asymptotically leaving behind a subsonic steady flow. Consequently, the thermodynamic parameters in the steady state are not independent; the pressure ratio (p_∞/p_w) and temperature ratio (T_∞/T_w) at infinity are uniquely determined by the Mach number (M_∞), reducing the degrees of freedom of the system to a single parameter [7, 10]. As the evaporation speed (M_∞) increases from 0 to 1 the pressure ratio decreases significantly together with T_∞/T_w decreases.

Furthermore, the spatial structure of the Knudsen layer is highly sensitive to the flow velocity, expanding significantly in thickness as the evaporation rate approaches the sonic limit ($M_\infty \to 1$).

Still, while the final steady evaporation flow is strictly subsonic and determined by a single parameter (usually the Mach number), the asymptotic behavior—how the gas behaves far from the wall as it settles into a steady state—is classified into four distinct patterns (plus a fifth case which turns into condensation). In fact, if the gas starts at a condition that does not exactly match the specific steady-state curve described above, the system generates disturbances (waves) to adjust. Based on numerical analysis, these behaviors are classified into four cases (excluding the condensation case) each defined by the sequence of hydrodynamic features that develop moving away from the condensed phase [10]:

1. Knudsen layer - Contact layer - Shock wave (R): The flow field is characterized by a kinetic transition at the condensed phase, a contact discontinuity where density varies at constant pressure, and a leading shock wave that creates an abrupt jump in gas properties (increase in pressure and density) as it propagates away from the condensed phase relative to the gas motion.

2. Knudsen layer - Contact layer - Expansion wave (R): A steady Knudsen layer and contact layer is established, but the transition to the initial state is managed by a right-moving expansion wave (R) that allows pressure and density to decay smoothly toward infinity rather than jumping across a shock.

3. Knudsen layer - Expansion wave (L) - Contact layer - Shock wave (R): This regime features an internal expansion wave (L) propagating toward the condensed phase relative to the gas motion with a sonic front, positioned between the Knudsen layer and a contact layer, while the entire system is led by an outward-moving shock wave.

4. Knudsen layer - Expansion wave (L) - Contact layer - Expansion wave (R): The flow develops into a double-expansion structure containing an internal left-moving expansion wave (L) with a sonic front and an external right-moving expansion wave (R), separated by a contact layer.

The key distinction is given by the pressure ration p_∞/p_w and the velocity at which the gas moves away from the condensed phase (M_∞). A shock wave occurs when the pressure of the initial gas (p_∞) is lower than the pressure of the gas being produced by the evaporation process at the surface (p_w). Physically, the evaporating gas acts like a piston, pushing into the slower/thinner initial gas and compressing it. This compression creates an abrupt physical jump, where the pressure and density spike instantly to "make room" for the incoming flow.

An expansion wave occurs when the initial gas state is already at a very low pressure or moving at a very high velocity away from the phase. In this case, instead of the evaporating gas pushing into the existing gas, the existing gas is effectively "pulling away" faster than the evaporation can fill the space. This creates a rarefaction zone where the pressure and density decay smoothly as the gas expands to fill the volume, rather than jumping across a sharp front.

If one or both the conditions are more extreme (Type 3 and 4) the flow cannot bridge the gap between the wall and the outer gas directly. Physically, it must develop an Expansion Wave (L) propagating toward the wall, which creates a distinct “hump” or additional step in the velocity and pressure profiles between the Knudsen layer and the contact layer. This extra wave acts as a sonic “buffer” at its front to satisfy high-speed boundary requirements.
The map of the different behaviors in the plane defined by $T/T_w = 1$ can be seen in Fig 13

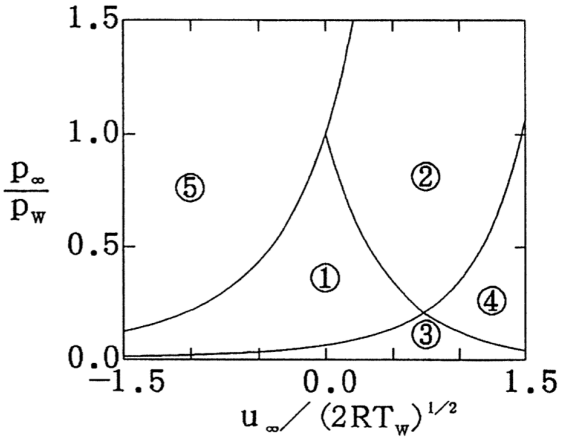
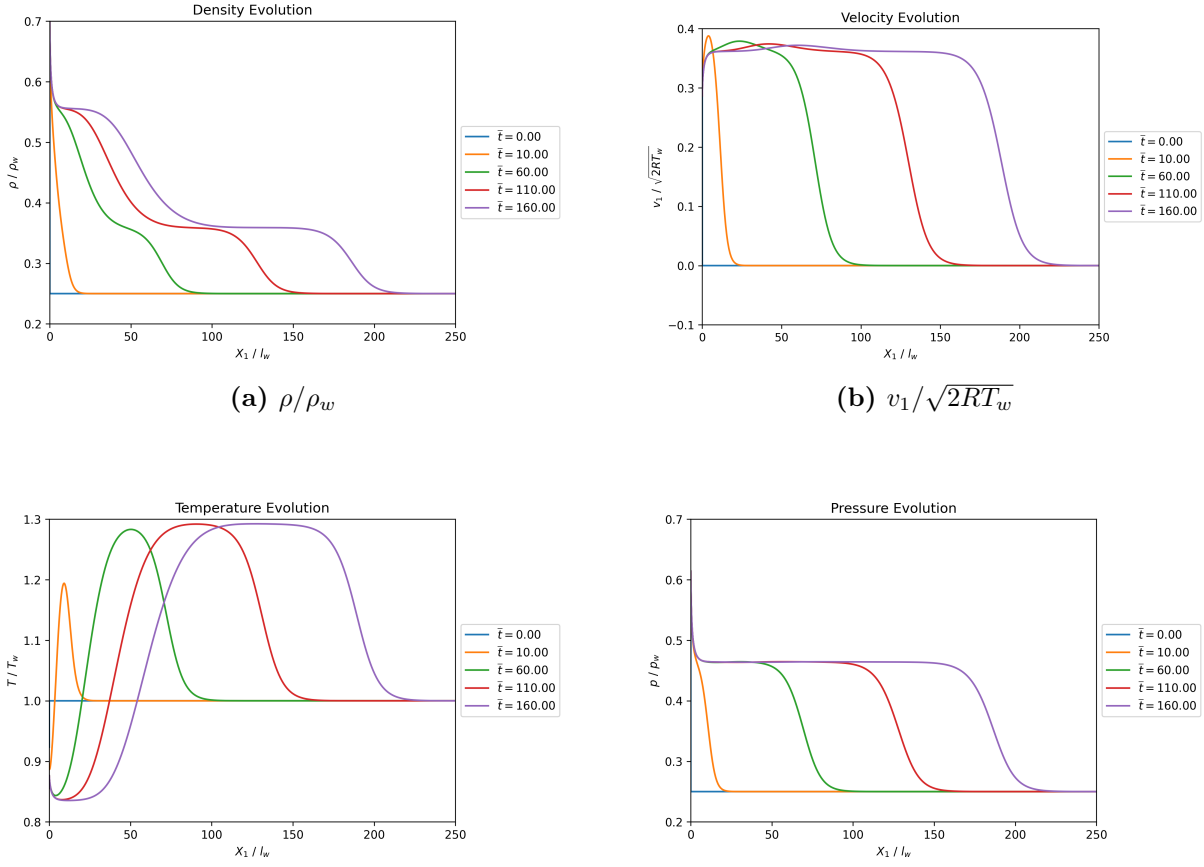


Fig. 13. Classification of the asymptotic behavior for $T/T_w = 1$ with (5) being the zone where condensation happens [10].

In all these cases, the "Knudsen layer" eventually becomes the steady evaporation region discussed above. The waves (shock or expansion) travel away to infinity, leaving behind the steady solution near the wall. If a supersonic region exists initially, it acts like an expansion wave that moves away from the condensed phase. It eventually disappears from the flow field at infinity, leaving behind a subsonic steady flow.

Figures 14, 15, 16, 17 represent respectively Type 1, 2, 3 and 4 and the behavior just discussed can be observed.

To conclude we plot the g and h quantities over the velocity space ζ at different points $x^{(i)}$. What can be seen from Fig. 18 is that, apart from the condensed phase ($x^{(0)} = 0$) the discontinuities



(a) ρ/ρ_w **(b)** $v_1/\sqrt{2RT_w}$

(c) T/T_w **(d)** p/p_w

Fig. 14. Time development of the solution corresponding to Type 1 in Fig. 13: $M_\infty = 0$, $p_\infty/p_w = 0.25$ and $T_\infty/T_w = 1$. Data on the lattice system: $\Delta t = 5 \times 10^{-2}$; $d_1 = 5 \times 10^{-3}$; $d_2 = 1$; $N = 330$; $N_0 = 100$; $a_1 = 5 \times 10^{-2}$; $a_2 = 9 \times 10^{-5}$; $\overline{N} = 48$

in the distribution functions quickly decay thus supporting the choices made for the integrals computations in Sec. 3.5.

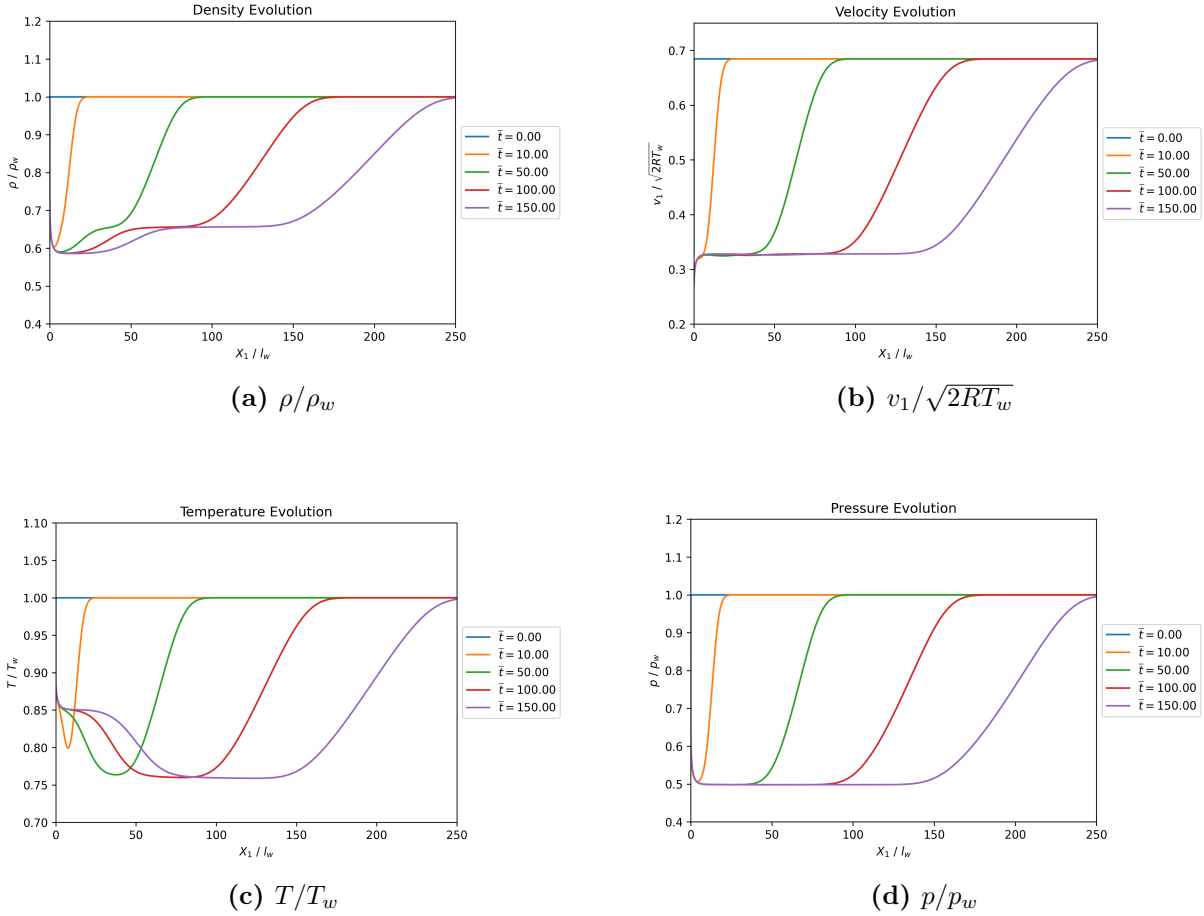
4.3 Scalability Results

To test the parallelization efficiency the following configuration was used: $M_\infty = -0.9$, $p_\infty/p_w = 8.0$ and $T_\infty/T_w = 1$. Data on the lattice system: $\Delta t = 5{\times}10^{-2}$; $d_1 = 5{\times}10^{-4}$; $d_2 = 0.1$; $N = 397$; $N_0 = 399$; $a_1 = 5 \times 10^{-2}$; $a_2 = 1 \times 10^{-6}$; $\overline{N} = 2^p$ with $p \in \{4, 5, 6, 7\}$. The max number of iterations was fixed to 2000 and the parameters defined above are such that it is always reached. All the material regarding the scalability test can be found in the project's Google Drive Folder.

The results of the scalability test are shown in Table. 2.

The performance data obtained from the OpenMP scalability tests reveals a nuanced landscape of parallel efficiency, characterized by tangible but not optimal performance gains. In terms of strong scalability, the implementation successfully reduces wall-clock time for the largest problem size ($\overline{N} = 128$), improving from 59.04 seconds on a single processor to 35.72 seconds on four processors. This represents a significant 40% reduction in solution latency, demonstrating that the solver effectively exploits parallel resources in the low-processor regime.

However, the scaling behavior is sub-linear and eventually exhibits diminishing returns. While the move from one to four processors yields a speedup of approximately 1.65× for the largest



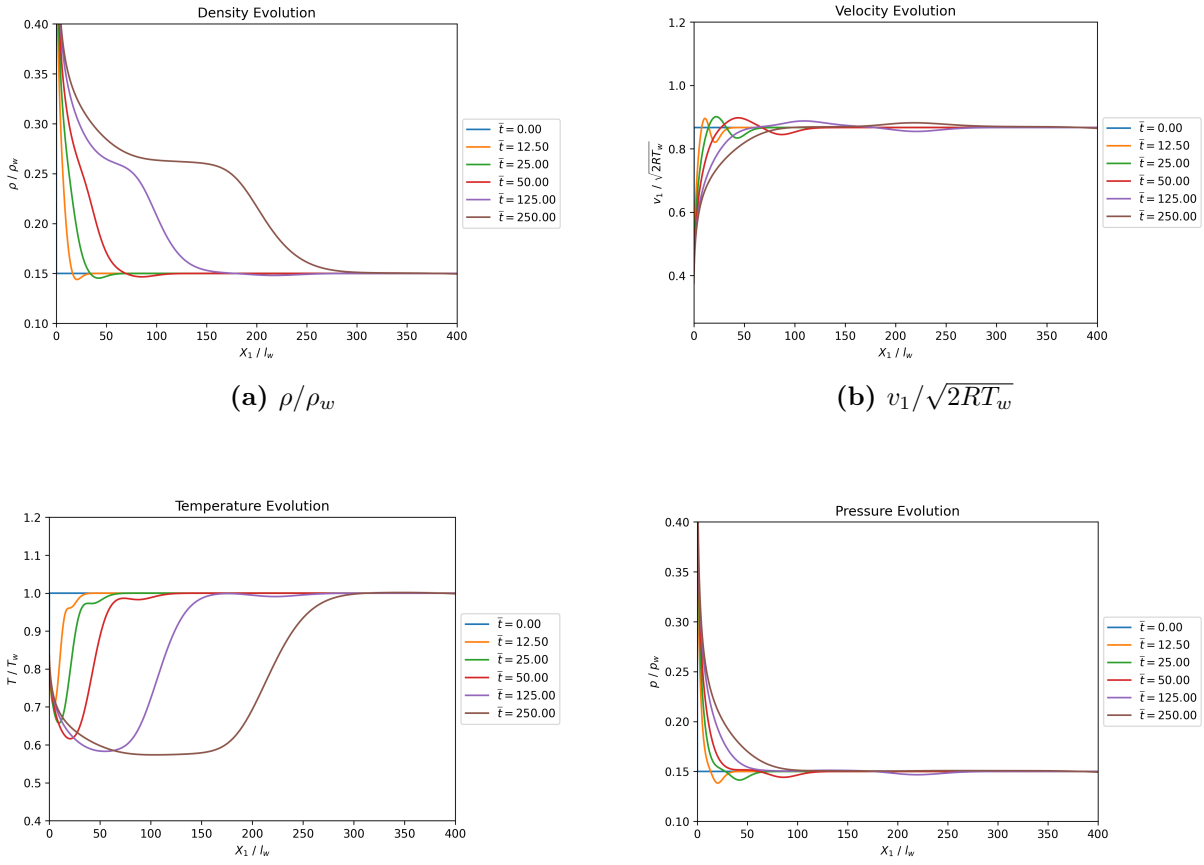
(a) ρ/ρ_w **(b)** $v_1/\sqrt{2RT_w}$

(c) T/T_w **(d)** p/p_w

Fig. 15. Time development of the solution corresponding to Type 2 in Fig. 13: $M_\infty = 0.75$, $p_\infty/p_w = 1$ and $T_\infty/T_w = 1$. Data on the lattice system: $\Delta t = 5 \times 10^{-2}$; $d_1 = 5 \times 10^{-3}$; $d_2 = 1$; $N = 330$; $N_0 = 100$; $a_1 = 5 \times 10^{-2}$; $a_2 = 9 \times 10^{-5}$; $\overline{N} = 48$

grid, scalability limits are reached beyond this point; at eight processors, execution time regresses slightly to 36.82 seconds. This behavior seems to identify four processors as the sweet spot for resource allocation, maximizing throughput while minimizing parallel overhead. Regarding weak scalability, while the execution time does rise when scaling from $\bar{N} = 16$ on one processor (6.55s) to $\bar{N} = 32$ on two processors (9.21s), the system successfully mitigates the computational load. By utilizing additional cores, the solver avoids the proportional doubling of runtime seen in serial execution (13.41s), thereby allowing for larger problem dimensions to be solved with a more manageable increase in latency.

Ultimately, while synchronization overheads dominate at higher concurrencies, the OpenMP implementation delivers valuable acceleration for computationally intensive grids ($\overline{N} \geq 64$) when constrained to an optimal processor count.



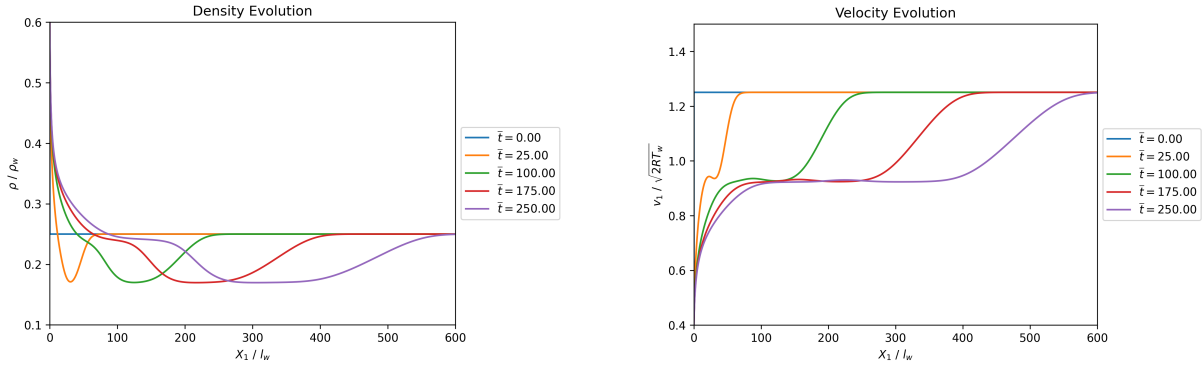
(a) ρ/ρ_w **(b)** $v_1/\sqrt{2RT_w}$

(c) T/T_w **(d)** p/p_w

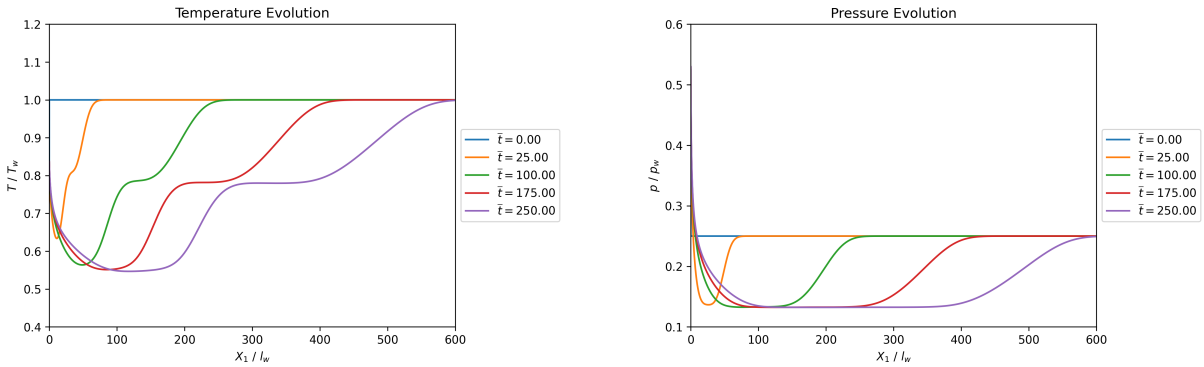
Fig. 16. Time development of the solution corresponding to Type 3 in Fig. 13: $M_\infty = 0.95$, $p_\infty/p_w = 0.15$ and $T_\infty/T_w = 1$. Data on the lattice system: $\Delta t = 5 \times 10^{-2}$; $d_1 = 1 \times 10^{-2}$; $d_2 = 3$; $N = 640$; $N_0 = 200$; $a_1 = 2 \times 10^{-2}$; $a_2 = 6 \times 10^{-5}$; $\overline{N} = 52$

	Number of Processors			
Grid Size ($\overline{N}$)	**1**	**2**	**4**	**8**
16	6.547	4.687	4.579	6.718
32	13.410	9.209	8.415	10.971
64	27.473	20.199	17.941	18.275
128	59.042	44.464	35.718	36.822

Table 2. Scalability Test Results: Time Elapsed (seconds)

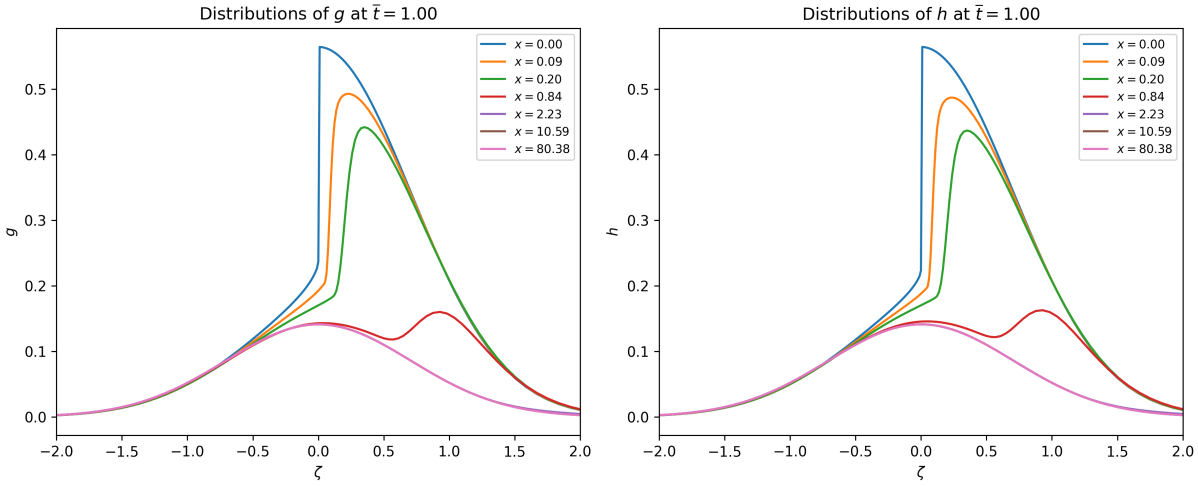


(a) ρ/ρ_w **(b)** $v_1/\sqrt{2RT_w}$

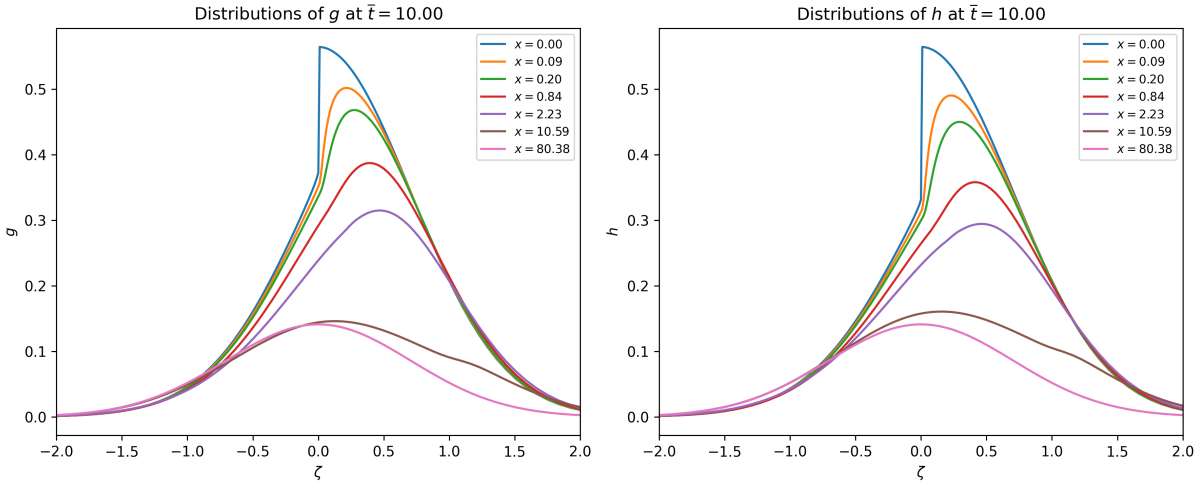


(c) T/T_w **(d)** p/p_w

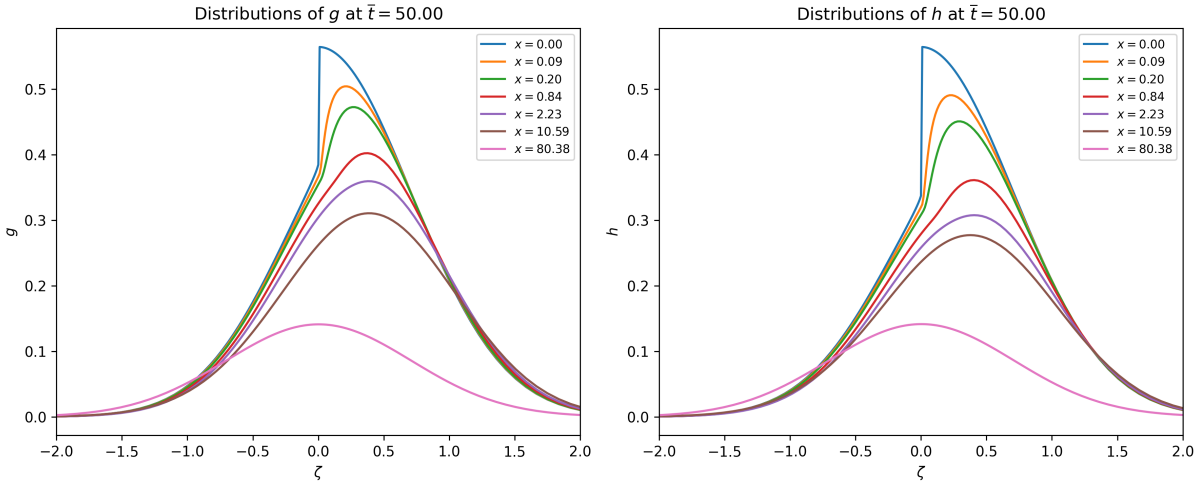
Fig. 17. Time development of the solution corresponding to Type 4 in Fig. 13: $M_\infty = 1.37$, $p_\infty/p_w = 0.25$ and $T_\infty/T_w = 1$. Data on the lattice system: $\Delta t = 5 \times 10^{-2}$; $d_1 = 1 \times 10^{-2}$; $d_2 = 3$; $N = 640$; $N_0 = 200$; $a_1 = 2 \times 10^{-2}$; $a_2 = 6 \times 10^{-5}$; $\overline{N} = 52$



(a) $\bar{t} = 1$



(b) $\bar{t} = 10$



(c) $\bar{t} = 50$

Fig. 18. Time development of g and h: $M_\infty = 0$, $p_\infty/p_w = 0.25$ and $T_\infty/T_w = 1$. Data on the lattice system: $\Delta t = 5 \times 10^{-2}$; $d_1 = 5 \times 10^{-3}$; $d_2 = 1$; $N = 330$; $N_0 = 100$; $a_1 = 1 \times 10^{-2}$; $a_2 = 1 \times 10^{-5}$; $\overline{N} = 84$

5 How to Run the Solver

Given what discussed in Sec.3, the `Plasma_BGK` solver as a standalone library appears as a command-line tool that leverages JSON configuration files to define the physical and numerical parameters of the simulation. Even though it can be easily integrated with bigger projects (not necessarily in a command line fashion) in this section we shall detail the basic workflow from compilation to post-processing.

The code can be downloaded by cloning the GitHub Repository with:

```
git clone git@github.com:andrea-rella/Plasma_BGK.git
cd Plasma_BGK
```

Before execution, ensure that the system satisfies the following requirements:

- `C++20` Compiler: Required for modern features.
- `Eigen3`: Utilized for high-performance linear algebra operations.
- `nlohmann/json`: Used for parsing simulation parameters.
- `OpenMP`: (Optional) Used for parallelizing the solver execution.

For build management the project uses a `Makefile` .

```
# Compiler and flags
CXX      = g++-15

# Flags
OMPFLAGS = -fopenmp # Needed only if you intend to use the parallel solver
CXXSTD   = -std=c++20
WARNINGS = -Wall -Wextra
OPTFLAGS = -O2 -g -march=native
DEFINES = -DNDEBUG

# Includes
INCLUDES = -Iinclude \
           -I/usr/local/Cellar/nlohmann-json/3.12.0/include \
           -I/usr/local/Cellar/eigen/3.4.0_1/include/eigen3

CXXFLAGS = $(OMPFLAGS) $(CXXSTD) $(WARNINGS) $(OPTFLAGS)

CPPFLAGS = $(INCLUDES) $(DEFINES)


# Targets
EXEC = main.exe
SRC  = main.cpp
OBJ  = $(SRC:.cpp=.o)

# Default target
all: $(EXEC)

# Compile source files into object files
%.o: %.cpp
        $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $< -o $@

# Link object files into the final executable
$(EXEC): $(OBJ)
        $(CXX) $(CXXFLAGS) $(LDFLAGS) $^ $(LIBS) -o $@

# Run the program
run: $(EXEC)
        ./$(EXEC)

# Clean up build artifacts
```

```
clean:
	$(RM) *.o

distclean: clean
	$(RM) *~ $(EXEC)
```

After having ensured that the generic include paths in the `Makefile` match your system's library locations and that the desired compiler flags are set, to compile the executable navigate to the root directory and run:

```
make
```

To remove object files or perform a full reset of the build environment, use `make clean` or `make distclean` respectively.

The solver behavior is governed by a `.json` file of the following style:

```
{
  "mesh": {
    "space_points": 397,
    "N0": 399,
    "velocity_points": 60,
    "d1": 5e-4,
    "d2": 0.1,
    "a1": 1e-2,
    "a2": 2.5e-5
  },

  "physical": {
    "T_infty_w": 1.0,
    "p_infty_w": 17.0,
    "M_infty": -1.2
  },

  "simulation": {
    "time_step": 0.05,
    "tolerance": 1e-7,
    "max_iterations": 1500,
    "save_every_k_steps": 50
  },

  "general": {
    "saving_folder_name": "./output/Cond/type1_data"
  }
}
```

The structure of the file is organised into four main blocks:

1. `mesh`: Defines the number of spatial points (N), the number of non-uniformly spaced spatial points (N_0), non-uniform spacing parameters (d_1, d_2), number of velocity points ($\overline{N}$) (which has to be *even* due to simpson integration rule constraints) and velocity grid resolution (a_1, a_2) (c.f. Eqs. (2.20) and (2.21)).

2. `physical`: Sets the boundary conditions and flow regime, including the temperature ratio (T_∞/T_w), pressure ratio (p_∞/p_w), and the Mach number (M_∞).

3. `simulation`: Controls numerical convergence, including the time step (Δt), relative tolerance (e.g., 10^{-7}), and maximum iterations.

4. `general`: Specifies the output directory for the resulting data.

Once compiled, supposing a standard `main.cpp` like:

```cpp
#include <iostream>
#include "ConfigData.hpp"
#include "SolverFV.hpp"

int main(int argc, char *argv[])
{
    std::string configPath = ""; // default path

    if (argc > 1)
        configPath += std::string(argv[1]);
    else
        throw std::runtime_error("No config file specified. Please provide a config file \
                                  name as a command line argument.");

    Bgk::ConfigData<double> Data(configPath);
    Bgk::SolverFV<double> solver(Data);

    std::cout << "Solver initialized with config: " << configPath << "\n" << std::endl;

    solver.initialize();

    solver.solve<Bgk::PlotStrategy::ONLYEND>(Bgk::metrics::VectorNormType::L2,
                                             Bgk::metrics::RowAggregateType::Max);

    solver.write_all(Data.get_saving_folder_name());

    return 0;

}
```

the solver is invoked by passing the path of the configuration file as a command-line argument:

```
./main.exe path/to/config.json
```

The solver initializes the `SpaceMeshFV` and `VelocityMesh` classes, executes the BGK kinetic iteration, and writes the macroscopic profiles to the specified output folder upon completion.

To visualize the results (e.g., density profiles, temperature, or velocity), use the provided Python scripts in the `postprocessing/` directory. The virtual environment `Bgk-venv` is recommended to handle dependencies like `numpy` and `matplotlib` and can be found in the dedicated Google Drive Folder together with all the configuration files used in the present report. An example usage is the following:

```python
import postprocessing.BGK_plot as pl
import postprocessing.BGK_read as rd
import json
from math import sqrt
import numpy as np

with open("data/Cond/type1.json", "r", encoding="utf-8") as f:
    problem_data = json.load(f)

folder = problem_data["general"]["saving_folder_name"] + "/"

timesteps = [0, 20, 80, 200, 400,700]
dt = problem_data["simulation"]["time_step"]

# Type 1
xlim = (0, 1.2)
temp_y_lim = (0.98, 1.03)
velocity_y_lim = (1.16, 1.25)
pressure_y_lim = (16.2, 17.8)
density_y_lim = (16.2, 17.7)

x = rd.read_mesh(folder + "space_mesh.txt")

pl.draw_temperature_evolution(
```

```
    x,
    timesteps,
    dt,
    folder,
    xlims=xlim,
    ylims=temp_y_lim,
    save_path=folder + "temperature_evolution.png"
    )

pl.draw_velocity_evolution(
    x,
    timesteps,
    problem_data["physical"]["T_infty_w"],
    dt,
    folder,
    xlims=xlim,
    ylims=velocity_y_lim,
    save_path=folder + "velocity_evolution.png"
    )

pl.draw_pressure_evolution(
    x,
    timesteps,
    dt,
    folder,
    xlims=xlim,
    ylims=pressure_y_lim,
    save_path=folder + "pressure_evolution.png"
    )

pl.draw_density_evolution(
    x,
    timesteps,
    dt,
    folder,
    xlims=xlim,
    ylims=density_y_lim,
    save_path=folder + "density_evolution.png"
    )
```

```
source path/to/Bgk-venv/bin/activate
python -m postprocessing.my_plot
```

Conclusions

The present study has addressed the kinetic modeling of gas-condensed phase interactions, specifically focusing on the non-linear phenomena of strong evaporation and condensation. By adopting the Bhatnagar-Gross-Krook (BGK) model of the Boltzmann equation, the code successfully captured the non-equilibrium behaviors inherent to the Knudsen layer phenomena that classical hydrodynamic descriptions fail to resolve.

The core of this work consisted of the development and implementation of a rigorous numerical framework, employing a Finite Volume Method (FVM) enhanced by the Quadratic Upwind Interpolation (QUICK) scheme. The solver, capable of handling the discontinuities in the velocity distribution function at the phase interface, was validated by comparing the time-dependent evolution of the flow against the established literature on rarefied gas dynamics. In particular, the following behaviors were observed:

- Strong Condensation: The numerical results successfully reproduced the four distinct solution regimes (Types I–IV) governed by the interplay between the Mach number (M_∞) and the pressure ratio (p_∞/p_w). The simulation confirmed that while steady supersonic condensation is physically permissible (Type I and III), specific configurations of upstream parameters trigger the formation of shock waves that propagate upstream (Type II), permanently altering the far-field conditions.
- Strong Evaporation: In strict agreement with the theoretical constraints identified by Ytrehus, our solver corroborated that steady evaporation flows are limited to the subsonic regime ($M_\infty \leq 1$). Attempts to impose supersonic evaporation conditions resulted in unsteady solutions characterized by the detachment of expansion waves or shock waves from the boundary. The code successfully resolved the four transient patterns (Knudsen layer interacting with expansion waves, contact layers, and shock waves) necessitated by the pressure mismatch between the surface and the bulk gas.

Although the code can live as a standalone project, taking it as a starting points many improvements and extensions can be made from both a pure coding prospective and a broadly physical one. In the following some ideas are proposed that may spark the interest of future contributors:

1. Sub-Iterations: Looking at the systems of equations defined by Eqs. (**??**)–(2.29) is clear theat the weak link of the chain is the temporal discretization given by the semi-implicit scheme:
$$\frac{\mathbf{u}^{k+1}-\mathbf{u}^k}{\Delta t} + (M + R^k)\mathbf{u}^{k+1} = \mathbf{F}^k$$
The fully implicit scheme would be:
$$\frac{\mathbf{u}^{k+1}-\mathbf{u}^k}{\Delta t} + (M + R(\mathbf{u}^{k+1}))\mathbf{u}^{k+1} - \mathbf{F}(\mathbf{u}^{k+1})$$
but $\mathbf{F}(\cdot)$ and $R(\cdot)$ are nonlinear. The idea is then to introduce sub-iterations to approximate that nonlinear solve inside each time step; to achieve this we define the non linear residual:
$$\mathbf{G}(\mathbf{u}) = \frac{\mathbf{u}-\mathbf{u}^k}{\Delta t} + (M + R(\mathbf{u}))\mathbf{u} - \mathbf{F}(\mathbf{u})$$
with the intent of finding $\mathbf{u}^{k+1} : \mathbf{G}(\mathbf{u}^{k+1}) = 0$. If there was a way (and in the present context there is) to re-frame $\mathbf{G}(\mathbf{u}^{k+1}) = 0$ as $\mathbf{u}^{k+1} = \mathbf{V}(\mathbf{u}^{k+1})$ then (ideally[9]):
$$\mathbf{u}^{k+1} = \lim_{m\to\infty} \mathbf{u}^{k+1,m} \qquad \text{with} \quad \mathbf{u}^{k+1,m+1} = \mathbf{V}(\mathbf{u}^{k+1,m})$$

[9] You have to prove that $\mathbf{V}$ is a contraction

As a consequence, instead of advancing from k to $k+1$ in one shot, we introduce an inner iteration index m and proceed with the following logic:

(a) Initialize: $\mathbf{u}^{k+1,0} = \mathbf{u}^k$

(b) Iterate for $m = 0, 1, 2, \ldots$: solve

$$\frac{\mathbf{u}^{k+1,m+1} - \mathbf{u}^k}{\Delta t} + (M + R(\mathbf{u}^{k+1,m}))\mathbf{u}^{k+1,m+1} = \mathbf{F}(\mathbf{u}^{k+1,m})$$

(c) Optional Relaxation: Set $\mathbf{u}^{k+1,m+1} \leftarrow \omega\mathbf{u}^{k+1,m+1} + (1-\omega)\mathbf{u}^{k+1,m}$ with $\omega \in [0, 1]$

(d) Convergence test: stop when $\|\mathbf{u}^{k+1,m+1} - \mathbf{u}^{k+1,m}\| < \varepsilon$ then set $\mathbf{u}^{k+1} = \mathbf{u}^{k+1,m+1}$

(e) Repeat setting $k \leftarrow k+1$

this procedure, called Picard (fixed-point) sub-iterations, is the most common and easier to implement improvement. If before we were treating R^k and $\mathbf{F}^k$ as constant constant source terms for the duration of the timestep (i.e. implementing a first order approximation in time), now we acknowledge that they depend on $\mathbf{u}$ and we search, by means of fixed-point iterations, the state $\mathbf{u}^{k+1}$ that is self-consistent with its own operators.

Therefore, two loops are in place: an outer loop represented by the time stepping (k) and an inner one represented by the sub iterations to compute $\mathbf{u}^{k+1}$ (m).

In general if R and $\mathbf{F}$ are highly non-linear, a simple fixed-point iteration might converge slowly. In those cases, moving toward a Newton-Raphson approach (where you use the Jacobian) would be more robust, though computationally more expensive per iteration. The general idea is to define again the nonlinear residual $\mathbf{G}(\mathbf{u})$ with the purpose of finding $\mathbf{G}(\mathbf{u}^{k+1}) = 0$ through fixed point iterations. In particular, supposing to have a guess $\mathbf{u}^{k+1,m}$, the Newton approach frames the iterations as looking for a correction $\delta\mathbf{u}^m$ so that:

$$\mathbf{u}^{k+1,m+1} = \mathbf{u}^{k+1,m} + \delta\mathbf{u}^m$$

the next step is approximate the residual with a taylor expansion:

$$\mathbf{G}(\mathbf{u}^{k+1,m+1}) = \mathbf{G}(\mathbf{u}^{k+1,m} + \delta\mathbf{u}^m) = \mathbf{G}(\mathbf{u}^{k+1,m}) + J(\mathbf{u}^{k+1,m})\delta\mathbf{u}^m \qquad \text{with} \quad J(\mathbf{u}) = \frac{\partial\mathbf{G}}{\partial\mathbf{u}}$$

then impose $\mathbf{G}(\mathbf{u}^{k+1,m+1}) = 0$ which gives the linear system:

$$J(\mathbf{u}^{k+1,m})\delta\mathbf{u}^m = -\mathbf{G}(\mathbf{u}^{k+1,m})$$

you then update the solution as $\mathbf{u}^{k+1,m+1} = \mathbf{u}^{k+1,m} + \delta\mathbf{u}^m$ until $\|\delta\mathbf{u}^m\| < \varepsilon$. Finally you accept $\mathbf{u}^{k+1} = \mathbf{u}^{k+1,m+1}$ and you go on with the outer timestepping loop.

2. Plasma sheet: While the Knudsen layer is typically associated with neutral vapor-liquid interfaces, its underlying physics extends to the behavior of ionized gases in electric propulsion systems, such as ionic thrusters. In these systems, the "interface" exists between the bulk plasma and the acceleration grids or the thruster walls.

   More precisely, in ion thrusters (such as Hall effect thrusters), the propellant (typically Xenon) is ionized and accelerated. Near the walls and the acceleration grids, the gas density is so low that the continuum assumption fails. Here, when a high-energy ion hits the solid wall it grabs an electron from the surface, becomes a neutral atom, and effectively "condense" (neutralize) from the plasma phase; at the same time the electrons follow the same fate but given that they are much lighter and move faster the result is that the wall becomes negatively charged while the region just in front of it is positively charged. An

electric field is thus established that pulls the ion forward and the electrons back; eventually, the system reaches a "steady state" where the electric field is just strong enough that only the very fastest electrons can make it to the wall, and the ions are accelerated just enough to match the electron flow. A plasma sheet (Debye sheet) is created.

The configuration is described by a system of equations where each species has its own evolution:

$$\begin{cases} \dfrac{\partial f_{(i)}}{\partial t} + \xi_j \dfrac{\partial f_{(i)}}{\partial x_j} + \dfrac{eE_j}{m_{(i)}} \dfrac{\partial f_{(i)}}{\partial \xi_j} = \nu_{(i)}(f_{(i)}^{(0)} - f_{(i)}) \\ \dfrac{\partial f_{(e)}}{\partial t} + \xi_j \dfrac{\partial f_{(e)}}{\partial x_j} - \dfrac{eE_j}{m_{(e)}} \dfrac{\partial f_{(e)}}{\partial \xi_j} = \nu_{(e)}(f_{(e)}^{(0)} - f_{(e)}) \\ E_j = -\dfrac{\partial \phi}{\partial x_j} \\ \Delta\phi = -\dfrac{e}{\epsilon_0}\left(\displaystyle\int (f_{(i)} - f_{(e)})\, d\boldsymbol{\xi}\right) \end{cases}$$

where $f_{(i)}(\mathbf{x}, \boldsymbol{\xi}, t)$ and $f_{(e)}(\mathbf{x}, \boldsymbol{\xi}, t)$ are the distributions functions in the phase space of respectively ions and electrons and the Xenon ions are supposed to be singly ionized (Xe^+).

Even though this is a very simplified formalization it can be already seen that the difficulty of this problem is much grater than what treated in the present study. Apart from the additional Poisson equation for the electric field the greatest obstacle is the non-separability of the equations in the discretized velocity space $\zeta^{(j)}$.

3. More realistic wall boundary conditions: As of now we problem was addressed under the conventional boundary condition on the condensed phase where the velocity distribution function of the molecules leaving the condensed phase is independent of the velocity distribution of the molecules incident on the condensed phase and its shape is the half of a stationary Maxwellian (Eq. (2.7)). In reality however, some molecules hitting the surface might simply be diffusely reflected (bounce off) rather than being absorbed and replaced by new ones. Therefore, the leaving gas should more accurately be treated as a mixture: a fraction $\alpha_c \in (0, 1]$ (Condensation Factor) is newly evaporated gas, and the remaining fraction $1 - \alpha_c$ is gas that has been "diffusely reflected" [10].

   In practice, the velocity distribution function f for molecules leaving the wall ($X_1 = 0$ for $\xi_1 > 0$) is still a Maxwellian, but the effective pressure ($\hat{p}_w$) changes to account for the incoming flux:

$$f = \left[\frac{\hat{p}_w}{RT_w(2\pi RT_w)^{1/2}}\right] \exp\left(-\frac{\xi_1^2 + \xi_2^2 + \xi_3^2}{2RT_w}\right)$$

   where the modified pressure $\hat{p}_w$ is defined as:

$$\hat{p}_w = \alpha_c p_w - (1 - \alpha_c)(2\pi RT_w)^{1/2} \int_{\xi_1 < 0} \xi_1 f(0, \boldsymbol{\xi}, t)\, d\boldsymbol{\xi}$$

   Note that because $\hat{p}_w$ depends on the integral of f for $\xi_1 < 0$, the molecules leaving the wall are now mathematically coupled to the molecules arriving at the wall.

4. Polyatomic Gas: In the present treatment only monoatomic gasses where considered however, the same scenario can be analyzed in the presence of polyatomic gas [6].

   Polyatomic molecules are modeled with j internal degrees of freedom (e.g., $j = 2$ for linear rotators like N_2, $j = 3$ for non-linear rotators). This changes the specific heat ratio (γ):

$$\gamma = \frac{5 + j}{3 + j}$$

Moreover, for polyatomic gases, the distribution function is extended to include rotational energy (ϵ) alongside molecular velocity ($\boldsymbol{\xi}$). If we take for example the downstream equilibrium state, we see that it's described by a Maxwellian that incorporates these internal degrees:

$$f_\infty(\boldsymbol{\xi}, \epsilon) = \frac{\rho_\infty}{(2\pi R T_\infty)^{3/2}} \exp\left(-\frac{(\boldsymbol{\xi} - \mathbf{v}_\infty)^2}{2RT_\infty}\right) \frac{\epsilon^{j/2-1}}{\Gamma(j/2)(kT_\infty)^{j/2}} \exp\left(-\frac{\epsilon}{kT_\infty}\right)$$

Mathematically, the gas no longer has a single temperature. It is divided into translational and rotational components, which only equalize at equilibrium:

- Translational Temperature (T_t): Derived from the kinetic energy of center-of-mass motion.
$$T_t = \frac{1}{3RN} \int (\boldsymbol{\xi} - \mathbf{v})^2 f \, d\boldsymbol{\xi} \, d\epsilon$$
- Rotational Temperature (T_r): Derived from the average internal energy.
$$T_r = \frac{2}{kjN} \int \epsilon f \, d\boldsymbol{\xi} \, d\epsilon$$
- Overall Temperature (T): The weighted average of the two.
$$T = \frac{3T_t + jT_r}{3 + j}$$

5. Functionality extension: The solve is prone and will benefit from functionality extensions, among some examples are: better ParaView support, iteration checkpoints system, object factory for integral method choice, better user interface, better error checking and exception handling, Docker support, MPI parallelization etc.

References

[1] K. Aoki, Y. Sone, and T. Yamada. Numerical analysis of gas flows condensing on its plane condensed phase on the basis of kinetic theory. *Physics of Fluids A: Fluid Dynamics*, 2(10): 1867–1878, 1990.

[2] P. L. Bhatnagar, E. P. Gross, and M. Krook. A model for collision processes in gases. i. small amplitude processes in charged and neutral one-component systems. *Physical review*, 94(3):511, 1954.

[3] C. Cercignani. *The Boltzmann Equation and Its Applications*. Springer New York, New York, NY, 1988. ISBN 978-1-4612-1039-9. doi: 10.1007/978-1-4612-1039-9_2. URL `https://doi.org/10.1007/978-1-4612-1039-9_2`.

[4] C. Cercignani. *Rarefied gas dynamics: from basic concepts to actual calculations*, volume 21. Cambridge university press, 2000.

[5] J. H. Ferziger, M. Perić, and R. L. Street. *Computational methods for fluid dynamics*. springer, 2019.

[6] A. Frezzotti. A numerical investigation of the steady evaporation of a polyatomic gas. *European Journal of Mechanics-B/Fluids*, 26(1):93–104, 2007.

[7] A. Frezzotti. Boundary conditions at the vapor-liquid interface. *Physics of fluids*, 23(3), 2011.

[8] G. M. Kremer. *An introduction to the Boltzmann equation and transport processes in gases*. Springer Science & Business Media, 2010.

[9] F. Reif. *Fundamentals of statistical and thermal physics*. Waveland Press, 2009.

[10] Y. Sone and H. Sugimoto. Strong evaporation from a plane condensed phase. *Shinku*, 31 (5):420–423, 1988.

[11] Y. Sone, K. Aoki, and I. Yamashita. A study of unsteady strong condensation on a plane condensed phase with special interest in formation of steady profile. In *15th International Symposium on Rarefied Gas Dynamics*, volume 2, pages 323–333, 1986.

[12] H. Sugimoto and Y. Sone. Numerical analysis of steady flows of a gas evaporating from its cylindrical condensed phase on the basis of kinetic theory. *Physics of Fluids A: Fluid Dynamics*, 4(2):419–440, 1992.

[13] T. Ytrehus. Molecular-flow effects in evaporation and condensation at interfaces. *Multiphase Science and Technology*, 9(3), 1997.

A Appendix

A.1 Quadratic Upwind Interpolation (QUICK)

In a Finite Volume setting the approximations to the integrals require the values of variables at locations other than computational nodes (CV centers) thus they have to be expressed in terms of the nodal values by interpolation. Numerous possibilities are available for calculating a certain quantity ϕ and its gradients at a cell face, in this work we shall adopt the Quadratic Upwind Interpolation (QUICK) [5].

Given a 1D finite volume grid as the following –the reasoning can be easily extended to 2D and 3D–:

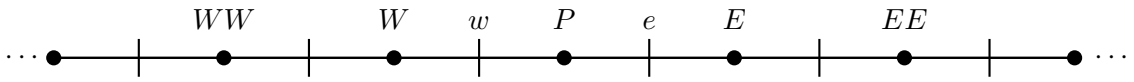


Fig. 19. Uniformly spaced mesh finite volume mesh. Dark circles indicate to the computational points while vertical lines the volume boundaries

the QUICK interpolation scheme interpolates the variable profile between P and E by a parabola rather than a straight line. To construct a parabola we need to use data at one more point in addition to P and E; in accord with the nature of convection, the third point is taken on the upstream side so that we have the following approximation formula:

$$\phi_e = \phi_U + \alpha_1(\phi_D - \phi_U) + \alpha_2(\phi_U - \phi_{UU})$$

where

$$\alpha_1 = \frac{(x_e - x_U)(x_e - x_{UU})}{(x_D - x_U)(x_D - x_{UU})} \qquad \alpha_2 = \frac{(x_e - x_U)(x_D - x_e)}{(x_U - x_{UU})(x_D - x_{UU})}$$

with where D, U, and UU denoting the downstream, the first upstream, and the second upstream node, respectively (E, P, and W or P, E, and EE, depending on the flow direction).

Furthermore we observe that each pair $\{\alpha_1, \alpha_2\}$ is unique to each volume boundary; therefore, if we consider the finite volume mesh adopted in this work (Figure 3) with $N+1$ computational points and $N+2$ volume boundaries we'll have a vector of pairs $\boldsymbol{\alpha} \in (\mathbb{R} \times \mathbb{R})^{N+2}$, $(\boldsymbol{\alpha})_i = \{\alpha_1^{(i)}, \alpha_2^{(i)}\}$ that fully characterizes the coefficients necessary for the finite volume approximation at hand.

B Class signatures

B.1 ConfigData

```cpp
template <typename T>
class ConfigData
{
private:
// ------MESH PARAMETERS ------
size_t N;
size_t N0;
size_t barN;
T d1;
T d2;
T a1;
T a2;

// ------PHYSICAL PARAMETERS ------
T T_infty_w;
T p_infty_w;
T M_infty;

// ------SIMULATION PARAMETERS ------
T dt;
T tol;
size_t max_iter;
size_t save_every_k_steps;

// ------SAVING PARAMETERS ------
std::string saving_folder_name;

public:
// ------CONSTRUCTORS AND DESTRUCTORS ------
ConfigData() = default;
ConfigData(std::string filename);
~ConfigData() = default;

// ------GETTERS ------
size_t get_N() const;
size_t get_N0() const;
size_t get_BarN() const;
T get_d1() const;
T get_d2() const;
T get_a1() const;
T get_a2() const;
T get_T_infty_w() const;
T get_p_infty_w() const;
T get_M_infty() const;
T get_dt() const;
T get_tol() const;
size_t get_max_iter() const;
size_t get_save_every_k_steps() const;
std::string get_saving_folder_name() const;
};
```

B.2 Mesh Classes

```cpp
template <typename T,
          MeshContainer1D<T> Container = std::vector<T>,
          MeshNature Nature = MeshNature::SPACE>
class BaseMesh1D
{
protected:
    size_t N;
    Container x_comp;
    bool is_initialized = false;

```

```
public:
    // ------ CONSTRUCTORS AND DESTRUCTORS ------
    BaseMesh1D() = default;
    BaseMesh1D(ConfigData<T> config) : is_initialized(false);
    virtual ~BaseMesh1D() = default;

    // ------ INITIALIZATION ------
    virtual void initialize_mesh() = 0;
    virtual bool validate_mesh() const;
    virtual void reset_mesh();
    bool isInitialized() const { return is_initialized; }

    // ------ GETTERS FOR MESH PARAMETERS ------
    size_t get_N() const;

    // ------ SETTERS FOR MESH PARAMETERS ------
    void set_N(size_t n);

    // ------ GETTERS FOR MESH COMPONENTS ------
    const Container &get_XComp();
    auto begin() const;
    auto end() const;
    auto size() const;

    // ------ OPERATORS ------
    virtual T &operator[](size_t index);
    virtual T operator[](size_t index) const;
    virtual T &at(size_t index);
    virtual T at(size_t index) const;

    // ------ OUTPUT MESH ------
    virtual void write_mesh_txt(const std::string &folder_path) const;
};
```

```
template <typename T>
class SpaceMeshFV : public BaseMesh1D<T, std::vector<T>, MeshNature::SPACE>
{
private:
    size_t N0;
    T d1;
    T d2;

    std::vector<T> x_vol;
    std::vector<T> vol_sizes;
public:
    // ------ CONSTRUCTORS AND DESTRUCTORS ------
    SpaceMeshFV() = default;
    SpaceMeshFV(const ConfigData<T> &);
    ~SpaceMeshFV() = default;

    // ------ MESH INITIALIZATION METHODS ------
    template <SpacingFunction<T> Spacing>
    void initialize_with_custom_spacing(Spacing &&spacing_func);
    void initialize_mesh() override;

    // ------ GETTERS ------
    const std::vector<T> &get_volume_boundaries() const { return x_vol; }
    const std::vector<T> &get_volume_sizes() const { return vol_sizes; }

    // ------ OUTPUT METHODS ------
    void write_mesh_txt(const std::string &folder_path) const override;
    void write_mesh_vtk(const std::string &folder_path) const;
};
```

```
template <typename T>
class VelocityMesh : public BaseMesh1D<T, std::vector<T>, MeshNature::VELOCITY>
{
private:
    T a1;
```

```
    T a2;

    std::function<T(size_t)> jacobian_func;

public:
    // ------CONSTRUCTORS AND DESTRUCTORS ------
    VelocityMesh() = default;
    VelocityMesh(const ConfigData<T> &config);
    ~VelocityMesh() = default;

    // ------SETTERS ------
    template <SpacingFunction<T> Jacobian>
    void set_jacobian_function(Jacobian &&jacobian);

    // ------GETTERS ------
    std::function<T(size_t)> get_jacobian_function() const { return jacobian_func; }

    // ------OPERATORS ------
    T operator[](size_t index) const override;
    T &operator[](size_t index) override;
    T at(size_t index) const override;
    T &at(size_t index) override;

    // ------INITIALIZATION ------
    template <SpacingFunction<T> Spacing>
    void initialize_with_custom_spacing(Spacing &&spacing_func);
    void initialize_mesh() override;
};
```

B.3 Solver

```
template <typename T>
class SolverFV
{
private:

    bool is_initialized = false;
    ConfigData<T> Data;
    SpaceMeshFV<T> Space_mesh;
    VelocityMesh<T> Velocity_mesh;

    // ------PHYSICAL QUANTITIES ------
    Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor> g;
    Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor> h;
    Eigen::Vector<T, Eigen::Dynamic> density;
    Eigen::Vector<T, Eigen::Dynamic> mean_velocity;
    Eigen::Vector<T, Eigen::Dynamic> temperature;

    // ------NUMERICAL MATRICES ------
    Eigen::SparseMatrix<T> A;
    Eigen::SparseMatrix<T> B;
    Eigen::Vector<T, Eigen::Dynamic> R;

    // ------BOUNDARY AND INITIAL CONDITIONS ------
    std::function<T(T z)> g0, g_infty;
    std::function<T(T z)> h0, h_infty;
    std::function<T(T z)> g_init, h_init;

    // ------COLLISION TERMS ------
    std::function<T(T z, T rho, T v, T Temp)> G, H;

    // ------SOLVE HELPERS ------
    Eigen::Vector<T, Eigen::Dynamic> assemble_U_pos(size_t j, T a1_2, T a2_2, T omega1) const;
    Eigen::Vector<T, Eigen::Dynamic> assemble_W_pos(size_t j, T a1_2, T a2_2, T omega1) const;
    Eigen::Vector<T, Eigen::Dynamic> assemble_U_zero() const;
    Eigen::Vector<T, Eigen::Dynamic> assemble_W_zero() const;
    Eigen::Vector<T, Eigen::Dynamic> assemble_U_neg(size_t j, T bN1_2, T bN2_2, T sigmaN1) const;
    Eigen::Vector<T, Eigen::Dynamic> assemble_W_neg(size_t j, T bN1_2, T bN2_2, T sigmaN1) const;
```

```
38
39      // ------SOLVE ------
40      void solve_timestep_pos();
41      void solve_timestep_neg();
42      void solve_timestep_zero();
43      void solve_timestep_pos_parallel();
44      void solve_timestep_neg_parallel();
45
46  public:
47      // ------CONSTRUCTORS AND DESTRUCTORS ------
48      SolverFV() = default;
49      SolverFV(const ConfigData<T> &InputData);
50      SolverFV(const std::string &config_file_path);
51      ~SolverFV() = default;
52
53      // ------INITIALIZE METHODS ------
54      void initializeMeshes();
55      template <SpacingFunction<T> custom_spacing>
56      void initialize_custom_spacemesh(custom_spacing &&custom_space_spacing);
57      template <SpacingFunction<T> custom_spacing, SpacingFunction<T> Jacobian>
58      void initialize_custom_velocitymesh(custom_spacing &&custom_velocity_spacing, Jacobian &&jacobian);
59      void set_physical_quantities();
60      void setInitialState();
61
62      // ------BUILD NUMERICAL MATRICES / SETUP ------
63      void assemble_A();
64      void assemble_B();
65      void assemble_R();
66
67      // ------GLOBAL INITIALIZE AND RESET ------
68      void initialize();
69      void reset();
70
71      // ------SOLVE ------
72      template <PlotStrategy Strategy>
73      void solve(const metrics::VectorNormType vec_norm_type, const metrics::RowAggregateType agg_type);
74      template <PlotStrategy Strategy>
75      void solve_parallel(const metrics::VectorNormType vec_norm_type, const metrics::RowAggregateType agg_type);
76
77      // ------OUTPUT ------
78      void write_sol_txt(const std::string &folder_path) const;
79      void write_sol_instant_txt(const std::string &folder_path, size_t iter) const;
80      void write_phys_txt(const std::string &folder_path) const;
81      void write_meshes_txt(const std::string &folder_path) const;
82      void write_space_mesh_vtk(const std::string &folder_path) const;
83      void write_initial_state_txt(const std::string &folder_path) const;
84      void write_phys_instant(const std::string &folder_path, size_t iter) const;
85      void write_all(const std::string &folder_path) const;
86  };
```