POLITECNICO DI MILANO

SCHOOL OF INDUSTRIAL AND INFORMATION ENGINEERING

MSc of Mathematical Engineering

Project for the course *Advanced programming for scientific computing (8 CFU)*

Development of a laminar numerical solver for reacting flows based on the SU2 CFD code

Giuseppe ORLANDO

Matr. 878776

Academic Year 2018-2019

List Of Symbols

δ_{ij} = Kronecker's delta
p = Mixture pressure
ρ = Mixture density
ρ_i = Density of species i
Y_i = Mass fraction of species i
X_i = Mole fraction of species i
T = Temperature $\boldsymbol{u}$ = Velocity
u_x = Component of the velocity along x direction
u_y = Component of the velocity along y direction
u_z = Component of the velocity along z direction
$\frac{\partial \cdot}{\partial(\rho \boldsymbol{u})} = \begin{pmatrix} \frac{\partial \cdot}{\partial(\rho u_x)} \\ \frac{\partial \cdot}{\partial(\rho u_y)} \\ \frac{\partial \cdot}{\partial(\rho u_z)} \end{pmatrix}$
E = Total energy per unit of mass
H = Mixture total enthalpy
h = Mixture static enthalpy
h_i = Static enthalpy of species i
C_p = Mixture specific heat at constant pressure
C_p = Mixture specific heat at constant volume
C_{p_i} = Specific heat at constant pressure of species i μ = Mixture laminar viscosity k = Mixture thermal conductivity

Contents

Chapter 1

Introduction

This project explores the formulation and implementation of the equations that model chemically reacting flows.
Specific attention has been paid to the modelling of molecular diffusion fluxes, which are the one that "feed the chemistry" using rigorous results of the kinetic theory of gases. For this purpose the well-known complete Stefan-Maxwell equations have been used in order to compute the diffusion fluxes. The problem with this kind of model is that singularities arise in the equation due to the conservation property of fluxes, especially in case of zero or vanishing mass fractions. In order to overcome this issue ad hoc modifications of Stefan-Maxwell equations are presented.

The implementation has been performed on the SU2 suite: it is a general purpose Computational Fluid Dynamics code provided by the Stanford University. In this work its capabilities have been extended in order to study multispecies reacting flows with the development of a library to compute physical and chemical properties.
Therefore in the first part we briefly describe the mathematical model that governs reacting flows and then we present a description of the coding part with the new classes and routines added to SU2 and the implementation of the aforementioned library.

Eventually some test cases are performed in order to validate the code: first we present an inviscid case, then we show the diffusion of two species inside a channel in order to verify the correct behaviour of this phenomenon and finally we simulate the processes of combustion inside aerospace engines.

1.1 Additions

In this section we briefly introduce some novelties applied to the code. First of all we needed all the numerical methods to compute the fluxes with distinction between convective, diffusive and source terms: therefore we built a class for each contribution all derived by a base class **CNumerics** as we will see later on.
Secondly we need to store opportunely the physical state of a multispecies simulation and we created two classes (**CReactiveEulerVariable** and **CReactiveNSVariable**) to store all the desired quantities; unlike the original implementation we employed some suitable **unsigned** type variables (T_INDEX_PRIM, RHO_INDEX_SOL,...) in order to rely on the position of a certain variable inside the conserved or primitive array: in this way another developer can simply modify these values if a different order is needed.
Eventually we adapted the main routines of the **CSolver** class in order to adequately call the functions implemented in the other classes for computing residuals and imposing boundary conditions.

Moreover we added all the extra information needed by the configuration file such as free-stream mass fractions or mass fractions at inlet and we chose to rely on an external library in order to compute physical-chemical properties.
Eventually we added some exceptions to indicate specifically that a particular function has not been implemented or that the library has not been correctly set up and some useful functions to compute the spline polynomials for thermodynamic and transport properties and functions to read the chemical reactions from a text file.

In the following sections as summary we will focus on some relevant aspects of the classes related to **CSolver**, **CNumerics** and **Cvaribale** and on the library.

Chapter 2

Mathematical Model

2.1 Conservative Governing Equations

We consider the unsteady, viscous chemically reacting flow equations in three spatial dimensions:

$$\frac{\partial \mathbf{Q}}{\partial t} + \nabla \cdot \mathbf{F} - \nabla \cdot \mathbf{G} = \mathbf{S} \tag{2.1}$$

where

$$\boldsymbol{Q} = \begin{pmatrix} \rho \\ \rho \boldsymbol{u} \\ \rho E \\ \rho_1 \\ \rho_2 \\ \dots \\ \rho_{N_s} \end{pmatrix}, \boldsymbol{F} = \begin{pmatrix} \rho \boldsymbol{u} \\ \rho \boldsymbol{u} \otimes \boldsymbol{u} + p\boldsymbol{I} \\ (\rho E + p)\boldsymbol{u} \\ \rho_1 \boldsymbol{u} \\ \rho_2 \boldsymbol{u} \\ \dots \\ \rho_{N_S} \boldsymbol{u} \end{pmatrix}, \boldsymbol{G} = \begin{pmatrix} 0 \\ \boldsymbol{\tau} \\ \boldsymbol{\tau u} - \boldsymbol{q} \\ -\boldsymbol{J}_1 \\ -\boldsymbol{J}_2 \\ \dots \\ -\boldsymbol{J}_{N_S} \end{pmatrix}, \boldsymbol{S} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \dot{\omega}_1 \\ \dot{\omega}_2 \\ \dots \\ \dot{\omega}_{N_s} \end{pmatrix}$$

Here $\otimes$ denotes the tensor product and $\nabla\cdot$ denotes the divergence operator.

This set of equations is known as **Navier-Stokes** equations: the first three are the continuity, momentum and energy balance equations respectively, and the rest are N_s species continuity equations, describing the mass conservation of each gas species.
In (2.1), ρ is the density of the gas mixture, while ρ_{i} is the density for species $i, \boldsymbol{u} = (u_x, u_y, u_z)$ is the velocity of the gas mixture and E is the total energy of the gas mixture per unit mass defined as

$$E = e + \frac{\|\mathbf{u}\|^2}{2} \tag{2.2}$$

where $\|\cdot\|$ denotes the Euclidean norm and e is the internal energy of the gas mixture per unit mass, and it is calculated based on a mass-weighted average of the internal energy per unit mass of each species e_i, i.e.,

$$e = \sum_{i=1}^{N_s} Y_i e_i. \tag{2.3}$$

Note that

$$Y_i = \rho_i / \rho$$

is the mass fraction of species i in the gas mixture. As will be shown later, the definitions of internal energy e and total energy E include the heat of formation of chemical species. Therefore, no source term exists in the energy equation. Moreover, mass is conserved through chemical reactions and the summation of all source terms is zero, i.e.

$$\sum_{i=1}^{N_s} \dot{\omega}_i = 0$$

Eventually we assume that individual species behave as thermally perfect gases, i.e. they satisfy the relation:

$$p_i = \rho_i R_i T \tag{2.4}$$

where p_i is the partial pressure, R_i is the gas constant of species i and T is the temperature. The gas constant R_i is computed based on

$$R_i = \frac{R_u}{M_i}$$

where $R_u = 8.31\mathrm{J\,mol^{-1}\,K^{-1}}$ is the universal gas constant and M_i is the molar mass of species i.
As Dalton's law prescribes that the pressure p of a mixture is equal to the sum of partial pressure, we find:

$$p = \sum_{i=1}^{N_s} \rho_i R_i T = \sum_{i=1}^{N_s} \rho Y_i R_i T = \rho R T \tag{2.5}$$

where $R = \sum_{i=1}^{N_S} Y_i R_i$ is the gas constant of the gas mixture.
Eventually $\boldsymbol{\tau}$ is the stress tensor whose definition is

$$\boldsymbol{\tau} = \mu \left(\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T - \frac{2}{3}(\nabla \cdot \boldsymbol{u})\boldsymbol{I} \right)$$

and $\boldsymbol{q}$ is the heat flux due to conduction and diffusion whose expression will be specified later on.

In equation (2.1) the term $\boldsymbol{J}_i$ represents the diffusion flux of species i whose contribution must be taken into account in the expression of $\boldsymbol{q}$:

$$\boldsymbol{q} = -k\nabla T + \sum_{i=1}^{N_S} h_i \boldsymbol{J}_i$$

where k represents the thermal conductivity and h_i is the static enthalpy of species i defined as:

$$h_i = e_i + \frac{p_i}{\rho_i} \tag{2.6}$$

The static enthalpy can be related to the specific heat at constant pressure C_{p_i}thanks to first law of thermodynamics; indeed if we define

$$C_{p_i} = \left.\frac{\partial q_i}{\partial T}\right|_{p_i}$$

we find that

$$dq_i = T ds_i = de_i + p_i dv_i = dh_i - v_i dp_i$$

Hence it follows immediately that

$$C_{p_i} = \left.\frac{\partial h_i}{\partial T}\right|_{p_i} \tag{2.7}$$

Herein q_i is the reversible heat transfer per unit of mass, s_i is the entropy of species i and v_i is the volume occupied by species i.
We immediately note that h_i is a function of T only and can be proper expressed as integration of (2.7):

$$h_i = \int_{T_{ref}}^{T} C_{p_i} dT + h_{f_i} \tag{2.8}$$

where T_{ref} is a reference temperature and h_{f_i} is the assigned value of enthalpy at $T = T_{ref}$.

Moreover the following conservation constraint must hold

$$\sum_{i=1}^{N_S} \boldsymbol{J}_i = 0$$

in order to ensure the total mass conservation as expressed by (2.1).

In case the dilute approximation (also known as Fick's law) holds we can say through Ramshaw self-consistent modification of fluxes that

$$\boldsymbol{J}_i = -\rho D_{i,m} \nabla Y_i + Y_i \sum_{i=1}^{N_S} \rho D_{i,m} \nabla Y_i \tag{2.9}$$

where $D_{i,m}$ is the mass diffusion coefficient for species i in the mixture.
In other cases the dilute approximation may not be acceptable and full multicomponent diffusion is required; in such cases, Stefan-Maxwell equations must be solved:

$$\boldsymbol{d}_i = \sum_{j=1}^{N_S} \frac{X_i X_j}{D_{ij}} \left(\boldsymbol{V_j} - \boldsymbol{V_i} \right) \tag{2.10}$$

where $\boldsymbol{V_i}$ and $\boldsymbol{V_j}$ represent the diffusion velocity of species i and j respectively, while $\boldsymbol{d}_i$ can be expressed as

$$\boldsymbol{d}_i = \nabla X_i + (X_i - Y_i) \nabla \ln p - \frac{Y_i}{p} \left(\rho \boldsymbol{f_i} - \rho \sum_{k=1}^{N_S} Y_k \boldsymbol{f_k} \right) \tag{2.11}$$

where X_i is the molar fraction of species i and $\boldsymbol{f}$ are body forces terms.
In this work no body forces are considered and we neglect the contribution due to the pressure so that:

$$\boldsymbol{d}_i = \nabla X_i \tag{2.12}$$

2.2 Analysis of Stefan-Maxwell Equations

Commonly the treatment of Stefan-Maxwell equations exploits the conservation of molecular fluxes and therefore plans the solution of a linear system with $N_S - 1$ unknowns. This approach has a significant drawback: you have to choose which of the $N_S - 1$ species you solve and so there is the presence of an asymmetry since one species is treated differently from the other and moreover some problems can arise from a numerical point of view if the selected species is not present in excess.

Another approach has been proposed by Giovangigli in [17]: the analysis carried out takes into account some criticalities of the Stefan-Maxwell equations once they are considered as a set of N_S independent relations.
Let us recall the Stefan-Maxwell equations as reported in (2.10):

$$\boldsymbol{d}_i = \sum_{j=1}^{N_S} \frac{X_i X_j}{D_{ij}} \left(\boldsymbol{V_j} - \boldsymbol{V_i} \right) = X_i \sum_{\substack{j=1 \\ j \neq i}}^{N_S} \frac{X_j \boldsymbol{V_j}}{D_{ij}} - X_i \boldsymbol{V_i} \sum_{\substack{j=1 \\ j \neq i}}^{N_S} \frac{X_j}{D_{ij}}$$

which can be expressed as a linear system of the form

$$\boldsymbol{F V} = -\boldsymbol{d}_i \tag{2.13}$$

where

$$F_{ii} = X_i \sum_{\substack{j=1 \\ j \neq i}}^{N_S} \frac{X_j}{D_{ij}} \qquad i = 1 ... N_S$$

$$F_{ij} = -\frac{X_i X_j}{D_{ij}} \qquad i, j = 1 ... N_S, \quad i \neq j$$

It is immediate to verify that $\sum_{j=1}^{N_S} F_{ij} = 0$ and therefore the matrix $\boldsymbol{F}$ is singular and hence not invertible.

The idea is to modify the Stefan-Maxwell equations in such a way that the constraint $\sum_{i=1}^{N_S} Y_i = 1$ is not imposed *a priori* but will be satisfied in the whole domain under the imposition of suitable boundary and initial conditions.
First of all since the continuity equations are solved for the mass fractions of the species, the driving forces of the Stefan-Maxwell equations must be expressed in terms of mass fractions as well. The relation between the mole and mass fractions is given by:

$$X_i = \frac{M}{M_i} Y_i$$

so that

$$\nabla X_i = \frac{M}{M_i} \nabla Y_i + \frac{Y_i}{M_i} \nabla M$$

Since

$$\frac{1}{M} = \sum_{j=1}^{N_S} \frac{Y_j}{M_j}$$

then

$$\nabla M = -M^2 \sum_{j=1}^{N_S} \frac{\nabla Y_j}{M_j}$$

and so

$$\nabla X_i = \frac{M}{M_i} \nabla Y_i - M^2 \frac{Y_i}{M_i} \sum_{j=1}^{N_S} \frac{\nabla Y_j}{M_j} = \frac{M}{M_i} \nabla Y_i - M X_i \sum_{j=1}^{N_S} \frac{\nabla Y_j}{M_j} = \sum_{j=1}^{N_S} M_{ij} \nabla Y_j$$

where

$$M_{ii} = \frac{M}{M_i} (1 - X_i) \qquad i = 1...N_S$$
$$M_{ij} = -\frac{M X_i}{M_j} \qquad i, j = 1...N_S, \quad i \neq j$$

Note that $\sum_{i=1}^{N_S} M_{ij} = 0$ and so also the matrix $\boldsymbol{M} = M_{ij}$ is singular; in order to eliminate the singularity of the aforementioned matrix we adopt the following relation between mass and mole fractions as proposed by [17]:

$$X_i = \sigma \frac{M}{M_i} Y_i$$

where $\sigma = \sum_{i=1}^{N_S} Y_i$.
Note that $\sum_{i=1}^{N_S} X_i = \sum_{i=1}^{N_S} Y_i$ and the gradient is now given by:

$$\nabla X_i = \frac{\sigma M}{M_i} \nabla Y_i - M X_i \sum_{j=1}^{N_S} \frac{\nabla Y_j}{M_j} + \frac{M Y_i}{M_i} \nabla \sigma = \widetilde{M_{ij}} \nabla Y_i$$

where

$$\widetilde{M_{ii}} = \frac{M}{M_i} (Y_i - X_i + \sigma) \qquad i = 1...N_S$$
$$\widetilde{M_{ij}} = M \left(\frac{Y_i}{M_i} - \frac{X_i}{M_j} \right) \qquad i, j = 1...N_S, \quad i \neq j$$

Note that $\sum_{i=1}^{N_S} \widetilde{M_{ij}} = 1$: the matrix $\widetilde{\boldsymbol{M}} = \widetilde{M_{ij}}$ is not singular and $\sum_{i=1}^{N_S} \nabla X_i = \sum_{i=1}^{N_S} \nabla Y_i$.

As shown by [17] singularities will appear for flux boundary conditions and homogeneous Neumann conditions as well. The singular behaviour is due to the lack of a diffusion term for the "*species*" $\sum_{i=1}^{N_S} Y_i$: adding an artificial diffusion would suppress the singularity and this is achieved by adding $\alpha Y_i (\alpha > 0)$ times the mass flux constraint to each Stefan-Maxwell equation that becomes:

$$\sum_{j=1}^{N_S} F_{ij} \boldsymbol{V_J} + \alpha Y_i \sum_{j=1}^{N_S} Y_j \boldsymbol{V_J} = -\nabla X_i \tag{2.14}$$

The modified Stefan-Maxwell equations can be expressed as:

$$\widetilde{\boldsymbol{F}} \boldsymbol{V} = -\boldsymbol{d}_i$$

where $\widetilde{\boldsymbol{F}} = \boldsymbol{F} + \alpha \boldsymbol{Y} \otimes \boldsymbol{Y}$.
As shown by Giovangigli the matrix $\widetilde{\boldsymbol{F}}$ is symmetric and positive definite and therefore it is invertible. If we introduce the matrix $\boldsymbol{G} = \widetilde{\boldsymbol{F}}^{-1}$ we find

$$\boldsymbol{V} = -\boldsymbol{G} \boldsymbol{d_i}$$

Switching from velocities to mass fluxes $\boldsymbol{J} = J_i$ since $J_i = \rho Y_i V_i$ we find

$$\boldsymbol{J} = -\boldsymbol{H} \boldsymbol{d_i}$$

where $\boldsymbol{H} = \boldsymbol{R}\boldsymbol{G}$ where $\boldsymbol{R} = \mathrm{diag}(\rho Y_1, ...\rho Y_{N_S})$.
For positive mass fractions the matrices $\boldsymbol{R}$ and $\boldsymbol{G}$ are non-singular and in this case $\boldsymbol{H}$ is not singular as well; however difficulties arise in the case of zero or vanishing mass fractions: in this situation the matrices $\boldsymbol{R}$ and $\widetilde{\boldsymbol{F}}$ are singular or ill-conditioned: indeed since $\widetilde{F_{ij}} = F_{ij} + \alpha Y_i Y_j$ we find that

$$\sum_{j=1}^{N_S} \widetilde{F_{ij}} = \sum_{j=1}^{N_S} F_{ij} + \sum_{j=1}^{N_S} \alpha Y_i Y_j = \alpha Y_i \sigma$$

which is equal or tends to 0 in presence of at least one zero or vanishing mass fraction.

Therefore we need a rescaled version of Stefan-Maxwell equations in terms of diffusion fluxes. We are led to introduce the matrix $\boldsymbol{\Gamma}$ such that:

$$\Gamma_{ii} = \sigma \frac{M}{\rho M_i} \sum_{\substack{j=1 \\ j \neq i}}^{N_S} \frac{X_j}{D_{ij}}, \qquad i = 1...N_S \tag{2.15}$$

$$\Gamma_{ij} = -\sigma \frac{M}{\rho M_j} \frac{X_i}{D_{ij}} \qquad i, j = 1...N_S, \quad i \neq j \tag{2.16}$$

so that:

$$\boldsymbol{\Gamma} \boldsymbol{J} = -\boldsymbol{d}_i$$

Since $\sum_{i=1}^{N_S} \Gamma_{ij} = 0$, the matrix $\boldsymbol{\Gamma}$ is singular and we have to modify it according to [17] as:

$$\widetilde{\boldsymbol{\Gamma}} = \boldsymbol{\Gamma} + \frac{\alpha}{\rho} \boldsymbol{Y} \otimes \boldsymbol{U}$$

where $\boldsymbol{U} = (1, ..., 1)^T$.

Since $\widetilde{\Gamma_{ij}} = \Gamma_{ij} + \frac{\alpha}{\rho} Y_i$ then $\sum_{i=1}^{N_S} \widetilde{\Gamma_{ij}} = \frac{\alpha}{\rho}\sigma$, hence we are able to determine the diffusive fluxes even in the presence of zero or vanishing mass fractions solving the linear system

$$\widetilde{\boldsymbol{\Gamma}} \boldsymbol{J} = -\boldsymbol{d}_i \tag{2.17}$$

In his work, Giovangigli developed also an iterative method to solve the aforementioned system; it is based on the following consideration: if we define $\boldsymbol{S} = \boldsymbol{I} - \boldsymbol{L}^{-1}\boldsymbol{\Gamma}$, with $\boldsymbol{L}$ to be determined later on, then $\boldsymbol{S}\boldsymbol{J} - \boldsymbol{L}^{-1}\boldsymbol{d}_i = \boldsymbol{J} + \boldsymbol{L}^{-1}\boldsymbol{d}_i - \boldsymbol{L}^{-1}\boldsymbol{d}_i = \boldsymbol{J}$.
Let us introduce the following vector space $\boldsymbol{U}^{\perp} = \left\{ \boldsymbol{x} \in \mathbb{R}^{N_S} : \boldsymbol{U} \cdot \boldsymbol{x} = 0 \right\}$ where

$$\boldsymbol{U} \cdot \boldsymbol{x} = \sum_{i=1}^{N_S} x_i.$$

There is only one solution $\boldsymbol{J}$ such that, given $\boldsymbol{d}_i \in \boldsymbol{U}^{\perp}$, it satisfies the conservation constraint; in particular the following theorem holds:

Theorem 1 *Let $\boldsymbol{\Gamma}$ be as in (2.15),(2.16), let $Y_i \geq 0, i = 1...N_S$ such that $\exists i \; Y_i \neq 0$ and let $\boldsymbol{L} = diag(L_1, ...L_{N_S})$ such that*

$$L_i = \frac{\Gamma_{ii}}{1 - \frac{Y_i}{\sigma}}$$

so that $L_i > \Gamma_{ii}$ if $Y_i > 0$ and $L_i \geq \Gamma_{ii}$ if $Y_i = 0$. Denote by $\boldsymbol{Q} = \boldsymbol{I} - \frac{\boldsymbol{Y} \otimes \boldsymbol{U}}{\sigma}$ and $\boldsymbol{S} = \boldsymbol{I} - \boldsymbol{L}^{-1}\boldsymbol{\Gamma}$ where $\boldsymbol{I}$ is the identity matrix; let $\boldsymbol{x_0} \in \mathbb{R}^{N_S}, \boldsymbol{y_0} = \boldsymbol{Q}\boldsymbol{x_0}$ and define

$$\boldsymbol{y}^{N+1} = \boldsymbol{Q}\left(\boldsymbol{S}\boldsymbol{y}^N - \boldsymbol{L}^{-1}\boldsymbol{d}_i\right).$$

Then

$$\boldsymbol{J} = \lim_{N \to \infty} \boldsymbol{y}^N$$

The drawback of this method is that the matrix $\boldsymbol{Q}$ projects each vector that belongs to $\mathbb{R}^{N_S}$ to $\boldsymbol{U}^{\perp}$; indeed we get:

$$[\boldsymbol{Q}\boldsymbol{x}]_i = x_i - \sum_{j=1}^{N_S} \frac{(\boldsymbol{Y} \otimes \boldsymbol{U})_{ij}}{\sigma} x_j = x_i - \sum_{j=1}^{N_S} \frac{Y_i}{\sigma} x_j = x_i - \frac{Y_i}{\sigma} \sum_{j=1}^{N_S} x_j$$

and therefore:

$$\begin{aligned} \sum_{i=1}^{N_S} [\boldsymbol{Q}\boldsymbol{x}]_i &= \sum_{i=1}^{N_S} \left(x_i - \frac{Y_i}{\sigma} \sum_{j=1}^{N_S} x_j \right) = \sum_{i=1}^{N_S} x_i - \sum_{i=1}^{N_S} \left(\frac{Y_i}{\sigma} \sum_{j=1}^{N_S} x_j \right) \\ &= \sum_{i=1}^{N_S} x_i - \sum_{j=1}^{N_S} x_j \sum_{i=1}^{N_S} \frac{Y_i}{\sigma} = \sum_{i=1}^{N_S} x_i - \sum_{j=1}^{N_S} x_j = 0 \end{aligned}$$

Hence this iterative method is not able to compute the correct diffusive fluxes in case $\boldsymbol{d}_i \notin \boldsymbol{U}^{\perp}$.

For this reason we choose to solve the linear system (2.17) with a "standard" iterative scheme, in particular the biconjugate gradient stabilized (**BiCGSTAB**) method (see Appendix A.2) because it does not need any particular requirement and this matches our case since the matrix $\widetilde{\boldsymbol{\Gamma}}$ is neither symmetric in general.

Eventually an appropriate choice for the free parameter α needed by the definition of $\widetilde{\boldsymbol{\Gamma}}$ is given by $\alpha = 1/\max_{i,j=1,...N_S} D_{ij}$ because it guarantees the same order of magnitude

for the elements of $\widetilde{\mathbf{\Gamma}}$.

At this point we need to focus on the application of the mass flux constraint: the original Stefan-Maxwell equations are subjected to $\sum_{i=1}^{N_S} \boldsymbol{J}_i = 0$, but the use of artificial diffusivity changes this constraint; indeed summing up the relations (2.17) we find:

$$\alpha\sigma \sum_{i=1}^{N_S} \boldsymbol{J}_i = -\nabla\sigma\rho \tag{2.18}$$

and therefore we modify the continuity equation to include this theoretical diffusion effect:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}) - \nabla \cdot \left(\frac{\rho}{\sigma\alpha} \nabla\sigma \right) = 0 \tag{2.19}$$

Therefore the matrix associated to viscous fluxes in (2.1) is modified as follows:

$$\boldsymbol{G} = \begin{pmatrix} \frac{\rho}{\sigma\alpha}\nabla\sigma \\ \boldsymbol{\tau} \\ \boldsymbol{\tau}\boldsymbol{u} - \boldsymbol{q} \\ -\boldsymbol{J}_1 \\ -\boldsymbol{J}_2 \\ \dots \\ -\boldsymbol{J}_{N_S} \end{pmatrix}$$

Let us notice that in case $\sigma = 1$ the constraint (2.18) reduces to $\sum_{i=1}^{N_S} \boldsymbol{J}_i = 0$ recovering the "*standard*" continuity equation.

Chapter 3

Numerical Methods

The system of equations in (2.1) must be discretized both in space and in time in order to provide a numerical solution.

3.1 Space Integration

3.1.1 Vertex-Centered Finite Volume

During the last decades, the Finite Volume method [9, 10] has become one of the most employed technique for simulating a wide variety of flows governed by hyperbolic equations.
The basic idea of this method is to subdivide the computational domain Ω into a disjoint set of finite cells or *volumes* and to apply inside each cell the conservation laws.
Let us introduce some useful notation: the division of Ω into N_C elements gives raise to the computational **mesh** or **grid** and the cell C_i is composed by a set of **vertices** V so that:

$$C_i, \qquad i = 1, ..., N_C$$

$$\Omega = \bigcup_{i=1}^{N_C} C_i$$

$$\mathring{C}_i \cap \mathring{C}_j = \emptyset, \qquad i, j = 1...N_C, i \neq j$$

Once a mesh has been formed, we have to create the finite volumes Ω_i on which the conservation laws will be applied. This can be done in two ways depending on where the solution is stored:

(1) If the solution is stored at the center of each C_i, then C_i itself is the finite volume, namely $\Omega_i = C_i$: this is the so called **cell-centered** finite volume method.

(2) If the solution is stored at the vertices of the mesh, then the finite volume Ω_i must be constructed around each vertex and this gives raise to the **vertex-centered** finite volume method.

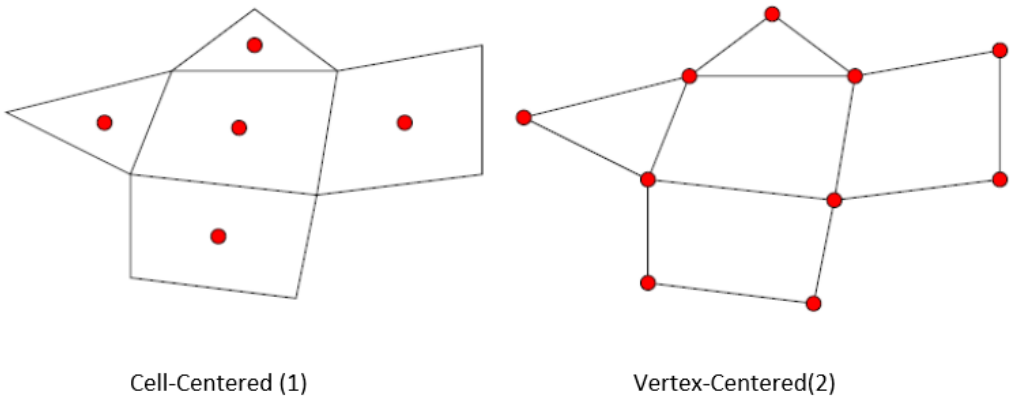

Figure 3.1: Finite volumes strategies

In either case we get a collection of finite volumes such that

$$\Omega = \bigcup_{i=1}^{N_C} \Omega_i$$

$$\mathring{\Omega}_i \cap \mathring{\Omega}_j = \emptyset, \qquad i, j = 1...N_C, i \neq j$$

The points where the solution is stored are called **nodes**.

The software that we will exploit adopts this second strategy and the finite volumes are formed by the centroids, face, and edge-midpoints of all cells sharing a particular node: their union is known as **dual grid** as shown is Figure (3.2).

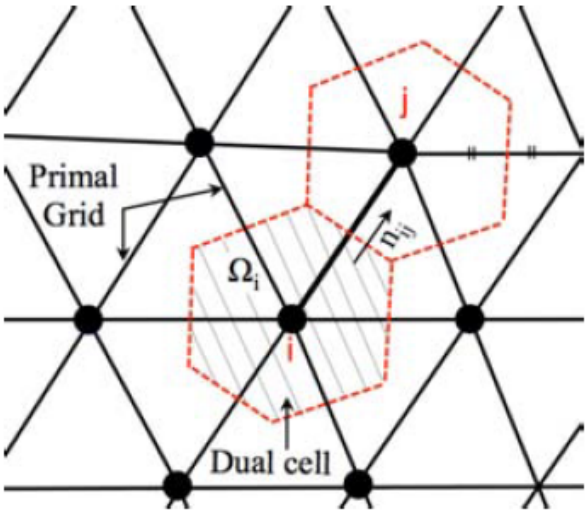

Figure 3.2: Schematic of the mesh and the control volume on a dual mesh.

The algorithm discretizes the system of PDE's in (2.1) written in an integral form:

$$\int_{\Omega_i} \frac{\partial \boldsymbol{Q}}{\partial t} d\Omega + \int_{\Omega_i} \nabla \cdot \boldsymbol{F} d\Omega - \int_{\Omega_i} \nabla \cdot \boldsymbol{G} d\Omega - \int_{\Omega_i} \boldsymbol{S} d\Omega = 0 \tag{3.1}$$

Now we exploit the divergence theorem to find:

$$\int_{\Omega_i} \frac{\partial \boldsymbol{Q}}{\partial t} d\Omega + \int_{\Sigma_i} \boldsymbol{F}\boldsymbol{n} d\Sigma - \int_{\Sigma_i} \boldsymbol{G}\boldsymbol{n} d\Sigma - \int_{\Omega_i} \boldsymbol{S} d\Omega = 0 \tag{3.2}$$

where Σ_i is the boundary of the finite volume Ω_i and $\boldsymbol{n}$ is the outward unit normal with respect to Σ_i. At this point we rewrite the equation (3.2) introducing the so called *residual* (or *steady residual*) so that

$$\int_{\Omega_i} \frac{\partial \boldsymbol{Q}}{\partial t} d\Omega + \boldsymbol{R_i}(\boldsymbol{Q}) = 0 \tag{3.3}$$

with

$$\boldsymbol{R_i}(\boldsymbol{Q}) = \int_{\Sigma_i} (\boldsymbol{F} - \boldsymbol{G})\, \boldsymbol{n} d\Sigma - \int_{\Omega_i} \boldsymbol{S} d\Omega \tag{3.4}$$

Every term in (3.4) is discretized separately leading to an upwind treatment of convective fluxes (see Appendix A.1), a central discretization of diffusive fluxes and a vertex-centered treatment of the source term in combination with an explicit(forward Euler, Runge Kutta) or implicit scheme for the time stepping. Moreover second order accuracy in space can be obtained evaluating the fluxes with a linear polynomial reconstruction, while high order in time can be achieved by selected a time integration scheme like $n-th$ order Runge-Kutta method.

3.2 Time Integration

The system of equations in (3.3) is an example of semidiscretazion of a system of PDEs; at this point we need a suitable discretization in time of (3.3) to obtain a numerical solution in space and in time. First of all let us reduce ourselves to a system of ODEs with the following semplification

$$\int_{\Omega_i} \frac{\partial \boldsymbol{Q}}{\partial t} d\Omega + \boldsymbol{R}_i(\boldsymbol{Q}) \approx \frac{d\boldsymbol{Q}_i}{dt} |\Omega_i| + \boldsymbol{R}_i(\boldsymbol{Q}) \tag{3.5}$$

where $|\Omega_i|$ is the volume of cell i.
Now we have several possibilities to discretize in time, for instance:

$$\frac{\boldsymbol{Q}_i^{n+1} - \boldsymbol{Q}_i^n}{\Delta t_i^n} + \boldsymbol{R}_i(\boldsymbol{Q}^n) = 0 \qquad \text{Explicit Euler (EE)} \tag{3.6}$$

$$\frac{\boldsymbol{Q}_i^{n+1} - \boldsymbol{Q}_i^n}{\Delta t_i^n} + \boldsymbol{R}_i\left(\boldsymbol{Q}^{n+1}\right) = 0 \qquad \text{Implicit Euler (IE)} \tag{3.7}$$

where the superscripts n and $n+1$ denote that the numerical solutions are evaluated at step n and $n+1$ respectively.
Δt_i^n is the time step for the cell i at time n: indeed **local-time stepping** strategies are applied so that each volume can advance at a different time step according to the local values of the variables of the problem. The following definition applies:

$$\Delta t_i = N_{CFL} \min\left(\frac{|\Omega_i|}{\lambda_i^{conv}}, \frac{|\Omega_i|}{\lambda_i^{visc}}\right)$$

where N_{CFL} is the Courant-Friedrichs-Lewy (CFL) number and λ_i^{conv} is the integrated convective spectral radius computed as

$$\lambda_i^{conv} = \sum_{f=1}^{N_f} \left(\left|u_{1/2}\right| + c_{1/2}\right) \Sigma_f$$

where $\left|u_{1/2}\right|$ is the absolute value of the interface velocity computed as $\left|\frac{\boldsymbol{u}_L + \boldsymbol{u}_R}{2} \cdot \boldsymbol{n}_f\right|$ and $c_{1/2}$ is the interface sound speed. On the other hand the viscous spectral radius λ_i^{visc} is computed as:

$$\lambda_i^{visc} = \sum_{f=1}^{N_f} \frac{C\mu_{1/2} + f(\mu_{1/2}^L)}{\rho_{1/2}} \Sigma_f^2$$

Here $\rho_{1/2}$ is the interface density already defined as the arithmetic mean of left and right state densities, C is a constant, $\mu_{1/2}$ is the interface viscosity defined as the sum of the interface laminar viscosity $\mu_{1/2}^L = \frac{\mu_L^L + \mu_R^L}{2}$ and f is a suitable function defined as

$$f(\mu_{1/2}^L) = \gamma_{1/2} \frac{\mu_{1/2}^L}{Pr_{1/2}^L} \tag{3.8}$$

where $Pr^L_{1/2}$ is the laminar Prandtl number defined as:

$$Pr^L_{1/2} = \frac{\mu^L_{1/2} C_{p_{1/2}}}{\kappa^L_{1/2}}$$

with $C_{p_{1/2}} = \frac{C_{p_L} + C_{p_R}}{2}$ as interface specific heat at constant pressure and $\kappa^L_{1/2} = \frac{\kappa^L_L + \kappa^L_R}{2}$ as laminar and turbulent interface thermal conductivity.
Finally $\gamma_{1/2} = \frac{C_{p_{1/2}}}{C_{v_{1/2}}}$ where $C_{v_{1/2}} = \frac{C_{v_L} + C_{v_R}}{2}$ is the interface specific heat at constant volume.

In case of Explicit Euler scheme the solution update $\Delta \boldsymbol{Q}^n_i = \boldsymbol{Q}^{n+1}_i = \boldsymbol{Q}^n_i$ is immediately found as:

$$\Delta \boldsymbol{Q}^n_i = -\boldsymbol{R}_i(\boldsymbol{Q}^n)\Delta t^n_i \tag{3.9}$$

while in case of Implicit Euler scheme the residuals at time $n+1$ are unknown and therefore a linearization about t^n is needed:

$$\boldsymbol{R}_i(\boldsymbol{Q}^{n+1}) = \boldsymbol{R}_i(\boldsymbol{Q}^n) + \frac{\partial \boldsymbol{R}_i(\boldsymbol{Q}^n)}{\partial t}\Delta t^n + \mathcal{O}(\Delta t^2) = \boldsymbol{R}_i(\boldsymbol{Q}^n) + \sum_{j=1}^{N_f} \frac{\partial \boldsymbol{R}_i(\boldsymbol{Q}^n)}{\partial \boldsymbol{Q}^n_j}\Delta \boldsymbol{Q}^n_j + \mathcal{O}(\Delta t^2)$$

Finally the following linear system should be solved to find the solution update $\Delta \boldsymbol{Q}^n_i$:

$$\left(\frac{|\Omega_i|}{\Delta t^n_i}\delta_{ij} + \frac{\partial \boldsymbol{R}_i(\boldsymbol{Q}^n)}{\partial \boldsymbol{Q}^n_j} \right) \Delta \boldsymbol{Q}^n_j = -\boldsymbol{R}_i(\boldsymbol{Q}^n)$$

The term $\frac{\partial \boldsymbol{R}_i(\boldsymbol{Q}^n)}{\partial \boldsymbol{Q}^n_j}$ is constituted by the contribution of convective numerical flux Jacobian, viscous numerical flux Jacobian (see Appendix A.4) and source term Jacobian and in case the total flux $\tilde{F}_{ij}$ has a stencil of points $\{i, j\}$, then contributions are made to the Jacobian at four points:

$$\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{Q}} = \begin{bmatrix} \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \frac{\partial \tilde{F}_{ij}}{\partial \boldsymbol{Q}_i} & \cdots & \frac{\partial \tilde{F}_{ij}}{\partial \boldsymbol{Q}_j} & \cdots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \cdots & \cdots & -\frac{\partial \tilde{F}_{ij}}{\partial \boldsymbol{Q}_i} & \cdots & -\frac{\partial \tilde{F}_{ij}}{\partial \boldsymbol{Q}_j} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Finally we set $\boldsymbol{Q}^{n+1} = \boldsymbol{Q}^n + \Delta \boldsymbol{Q}^n$.

The software allows also the use of a **dual time-stepping** strategy in order to achieve high-order accuracy in time. The main idea of this method is to transform an unsteady problem into a steady one at each physical time step; therefore the implementation of the dual-time stepping approach solves the following problem:

$$\frac{\partial \boldsymbol{Q}}{\partial \tau} + \boldsymbol{R}^* (\boldsymbol{Q}) = 0$$

with

$$\boldsymbol{R}^* (\boldsymbol{Q}) = \frac{3}{2\Delta t}\boldsymbol{Q} + \frac{1}{|\Omega|^{n+1}} (\boldsymbol{R}(\boldsymbol{Q})) - \frac{2}{\Delta t}\boldsymbol{Q}^n |\Omega^n| + \frac{1}{2\Delta t}\boldsymbol{Q}^{n-1} |\Omega|^{n-1}$$

where Δt is the physical time step and τ is a fictitious time step used for the convergence of the steady problem: therefore we set $\boldsymbol{Q} = \boldsymbol{Q}^{n+1}$ once the steady problem is satisfied.

3.3 Boundary Conditions

A set of boundary conditions is necessary to solve the governing equations (2.1) at each time step. For node-centric finite volume solvers the solution is stored directly on the computational boundary: this allows boundary conditions to be enforced either weakly or strongly. In weak enforcement, the governing equations are written on the boundary and a flux is computed such that when the solution achieves convergence, the condition is satisfied, while in strong enforcement Dirichlet conditions are set for one or more scalar variables at boundary and any contribution to the solution residual or Jacobian from flux calculations is eliminated to preserve the specified boundary condition. Herein we report some considerations about inlet and outlet boundary conditions.

3.3.1 Subsonic Inlet

More attention is needed for what concerns the inlet boundary conditions; two cases must be distinguished: supersonic and subsonic inlet.
In both cases the Riemann invariants have to be taken into account [18]; in case of supersonic inlet there are no outgoing characteristics and therefore all variables can be directly imposed at inlet, while in case of subsonic inlet there is one outgoing characteristic. Usually it is expressed as

$$V_n + 2\frac{c}{\gamma - 1} = const$$

but unluckily this relation holds only considering the isentropic relations for ideal gas, i.e. $\gamma = const$, as specified in [21]

$$\frac{p}{\rho^\gamma} = const$$

In our case, since we are considering thermally perfect gases and γ is not constant, we can not deal a priori to this relation. One possible novel approch can be starting from differential relations and then when no exact integral approximate it through suitable quadrature formulas.
In this analysis we will focus on the so called **total inlet boundary conditions** which impose the velocity, the **total temperature** and the **total pressure**, i.e. the temperature and the pressure when the fluid is at rest. Let us express the conditions of isentropicity and adiabaticity between the "total" state and the boundary state and the Riemann invariant in differential form:

$$\begin{cases} C_v \dfrac{dT}{T} - R\dfrac{d\rho}{\rho} = 0 & (3.10) \\ dh + V dV = 0 & (3.11) \\ dV_n + \dfrac{c}{\rho} d\rho = 0 & (3.12) \end{cases}$$

where the subscripts b and i denote the boundary and inner state respectively. If we develop and we start integrating we get:

$$\begin{cases} C_p dT - R_b dT - R_b \frac{T}{\rho} d\rho = 0 \\ \int_{H_{tot}}^{h_b} dh + \int_0^{V_b} dV = 0 \\ \int_{V_{n,i}}^{V_{n,b}} dV_n + \int_{\rho_i}^{\rho_b} \frac{c}{\rho} d\rho = 0 \end{cases}$$

where H_{tot} is the total enthalpy computed at the imposed total temperature T_{tot}. Therefore we find:

$$\begin{cases} \int_{T_{tot}}^{T_b} C_p dT - \int_{T_{tot}}^{T_b} R_b dT - \int_{T_{tot}}^{T_b} R_b \frac{T(\rho)}{\rho} d\rho = 0 \\ h_b + \frac{V_b^2}{2} = H_{tot} \\ V_{n,b} + \int_{\rho_i}^{\rho_b} \frac{c}{\rho} d\rho = V_{n,i} \end{cases}$$

where ρ_{tot} is computed using the gas equation with T_{tot} and the total pressure p_{tot}. Applying the trapezoidal rule we find:

$$\begin{cases} h_b - H_{tot} - R_b(T_b - T_{tot}) - R_b\left(\dfrac{\rho_b - \rho_{tot}}{2}\right)\left[\dfrac{T_b}{\rho_b} + \dfrac{T_{tot}}{\rho_{tot}}\right] = 0 & (3.13) \\ h_b + \dfrac{V_b^2}{2} = H_{tot} & (3.14) \\ V_b\alpha + \left(\dfrac{\rho_b - \rho_i}{2}\right)\left[\dfrac{c_b}{\rho_b} + \dfrac{c_i}{\rho_i}\right] = V_{n,i} & (3.15) \end{cases}$$

where *alpha* is the inner product between the inlet velocity direction and the outward unit normal.

Let us derive V_b from (3.15):

$$\begin{aligned} V_b &= \frac{1}{\alpha}\left[V_{n,i} - \left(\frac{\rho_b - \rho_i}{2}\right)\left(\frac{c_b}{\rho_b} + \frac{c_i}{\rho_i}\right)\right] \\ &= \frac{1}{\alpha}\left[V_{n,i} + \frac{1}{2}\left(\rho_i\frac{c_b}{\rho_b} + c_i - \rho_b\frac{c_i}{\rho_i} - c_b\right)\right] = V_b(\rho_b(T_b)) \end{aligned} \tag{3.16}$$

Now we need to express $\rho_b = \rho_b(T_b)$ in order then to substitute into (3.14) and apply an iterative scheme like the secant method (see Appendix A.3) to solve the implicitly defined relation; from (3.13) we find:

$$\begin{aligned} & h_b - H_{tot} - R_b(T_b - T_{tot}) - \frac{R_b}{2}\left(T_b + \rho_b\frac{T_{tot}}{\rho_{tot}} - \rho_{tot}\frac{T_b}{\rho_b} - T_{tot}\right) = 0 \implies \\ & h_b - H_{tot} - \frac{3}{2}R_b\left(T_b - T_{tot}\right) - \frac{R_b}{2}\rho_b\frac{T_{tot}}{\rho_{tot}} + \frac{R_b}{2}\rho_{tot}\frac{T_b}{\rho_b} = 0 \implies \\ & -\frac{R_b}{2}\frac{T_{tot}}{\rho_{tot}}\rho_b^2 + \left[n_b - H_{tot} - \frac{3}{2}R_b(T_b - T_{tot})\right]\rho_b + \frac{R_b}{2}\rho_{tot}T_{tot} = 0 \implies \\ & \frac{R_b}{2}\frac{T_{tot}}{\rho_{tot}}\rho_b^2 - \left[n_b - H_{tot} - \frac{3}{2}R_b(T_b - T_{tot})\right]\rho_b - \frac{R_b}{2}\rho_{tot}T_{tot} = 0 \end{aligned} \tag{3.17}$$

The relation (3.17) is a second degree equation whose only physical solution is

$$\rho_b = \frac{\left[h_b - H_{tot} - \frac{3}{2}R_b(T_b - T_{tot})\right] + \sqrt{\left[h_b - H_{tot} - \frac{3}{2}R_b(T_b - T_{tot})\right]^2 + R_b^2 T_{tot}T_b}}{\frac{R_b T_{tot}}{\rho_{tot}}} = \rho_b(T_b) \tag{3.18}$$

Eventually if we substitute (3.18) into (3.16) and (3.16) into (3.14) we get:

$$h_b(T_b) + \frac{V_b(\rho_b(T_b))^2}{2} = H_{tot} \tag{3.19}$$

It is worth to notice that the choice of trapezoidal rule has been due to the fact that we need to derive explicitly ρ_b as a function of T_b but more accurate quadrature rules can be applied.

3.3.2 Subsonic outlet

An analogous discussion holds also for the outlet boundary condition: in case of supersonic outlet there are no incoming characteristics and therefore all the variables can be extrapolated from the interior state, while in case of subsonic outlet there is one incoming characteristics and therefore one variable must be specified and used to update the value of all the other variables.
The typical choice, applied also in this work, is to impose the static back pressure p_b at outlet boundary and then to rely on isentropicity between the outer and the inner state in order to update the boundary variables.

As stated in [21], the isentropicity condition can be also expressed in differential form as:

$$C_p \frac{dT}{T} - R\frac{dp}{p} = 0 \tag{3.20}$$

which can be integrated:

$$\int_{T_i}^{T_b} \frac{C_p}{T} dT - R \ln\left(\frac{p_b}{p_i}\right) = 0 \tag{3.21}$$

where R is constant because we keep on the boundary the same mass fractions of the inner node.
Now we need to approximate the integral that appears in (3.21) using a quadrature formula; herein we choose to employ the so-called Cavalieri-Simpson rule which represents a good compromise between accuracy and computational cost. Therefore we get:

$$\int_{T_i}^{T_b} \frac{C_p}{T} dT = \frac{T_b - T_i}{6}\left[\frac{C_p(T_i)}{T_i} + 4\frac{C_p(\frac{T_i+T_b}{2})}{\frac{T_i+T_b}{2}} + \frac{C_p(T_b)}{T_b}\right] = f(T_b) \tag{3.22}$$

from which we find the following implicit equation:

$$F(T_b) = f(T_b) - R\ln\left(\frac{p_b}{p_i}\right) = 0 \tag{3.23}$$

which can be solved iteratively through the secant method.
Up to now we did not mention anything about the information from the Riemann invariant: now it's time to exploit the incoming characteristic in order to find the boundary velocity; indeed the aforementioned Riemann invariant can be stated in the form:

$$V_{b,n} - \int_{\rho_i}^{\rho_b} \frac{c}{\rho} d\rho = V_{i,n} \tag{3.24}$$

The integral in (3.24) can be approximated through quadrature rules: herein we choose a three-point Gaussian quadrature formula; since we can map the interval between ρ_i and ρ_b (determined through (2.5)) exploiting the isentropicity, this scheme represents a good compromise between accuracy and computational cost.
More in detail we get:

$$\int_{\rho_i}^{\rho_b} \frac{c(T)}{\rho} d\rho = \int_{\rho_i}^{\rho_b} f(\rho) d\rho \approx \frac{\rho_b - \rho_i}{2} \sum_j w_j f\left(\frac{\rho_b - \rho_i}{2}\rho_j + \frac{\rho_b + \rho_i}{2}\right) = \frac{\rho_b - \rho_i}{2} \sum_j w_j f(\tilde{\rho}_j) \tag{3.25}$$

with

$$w_j = \begin{bmatrix} \frac{5}{9} & \frac{8}{9} & \frac{5}{9} \end{bmatrix} \qquad \rho_j = \begin{bmatrix} -\frac{\sqrt{3}}{5} & 0 & \frac{\sqrt{3}}{5} \end{bmatrix}$$

The intermediate temperature $\tilde{T}_j$ at corresponding density $\tilde{\rho}_j$ to compute the speed of sound is determined through the isentropicity, as stated before:

$$\int_{\tilde{T}_{j-1}}^{\tilde{T}_j} \frac{C_p}{T} dT - R \int_{\tilde{T}_{j-1}}^{\tilde{T}_j} \frac{dT}{T} - R \int_{\tilde{\rho}_{j-1}}^{\tilde{\rho}_j} \frac{d\rho}{\rho}$$

Anyway both this approach for subsonic outlet and the one previously described for subsonic inlet do not guarantee a significant improvement in the accuracy of boundary condition especially if compared with the computational cost:

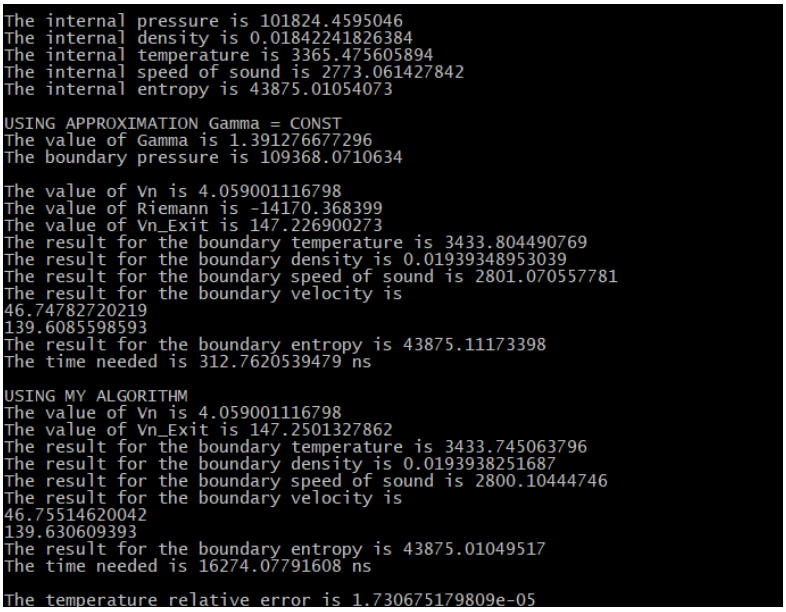

Figure 3.3: Comparison between two approaches for subsonic boundary conditions

Therefore we choose to pick $\gamma = const$ approximation taking the internal value for the subsonic outlet and the harmonic average between the internal value and the total one for the subsonic inlet.
Eventually it is worth to notice that since we use Burcat polynomials to compute the enthalpy as we will see later on, we still have to rely on secant method to determine the boundary temperature at inlet through (3.14) because temperature is defined by enthalpy only implicitly.

Chapter 4

SU2 Code

The system of equations previously introduced is implemented in the **SU2** suite, an open-source collection of C++ based software tools for performing Partial Differential Equations(PDE) analysis and solving PDE-constrained optimization problems.
For a more detailed analysis, please refer to [2]

4.1 General Features

At the highest level, SU2 has a driver class, **CDriver**, that controls the solution of a multiphysics simulation. The CDriver class is responsible for instantiating all of the geometry, physics packages, and numerical methods needed to solve a particular problem as we can see in (4.1). In particular the constructor of this class calls all the routine

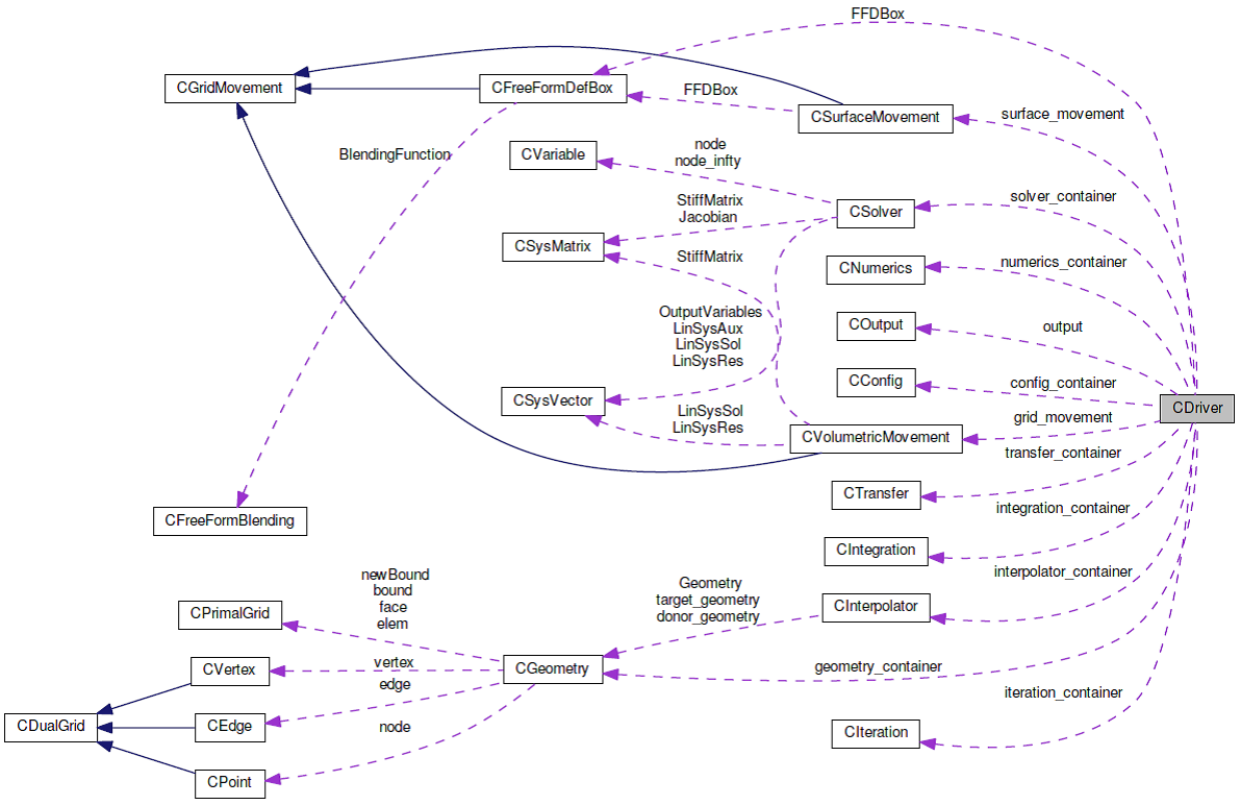

Figure 4.1: General structure of SU2

involved to the preprocessing of the simulation that ensure the physical reliability of data.
For our purposes, since we are interested in the implementation of a new kind of problem,

we have to focus on the classes which represent the interface for physical problems which are **CVariable**, **CSolver** and **CNumerics**.

4.1.1 CSolver

In this class the solution procedure is defined and each child represent a solver for a particular set of governing equations: ours will be solved through class **CReactiveEulerSolver** or **CReactiveNSSolver** depending on whether Euler or Navier-Stokes equations have to be investigated.

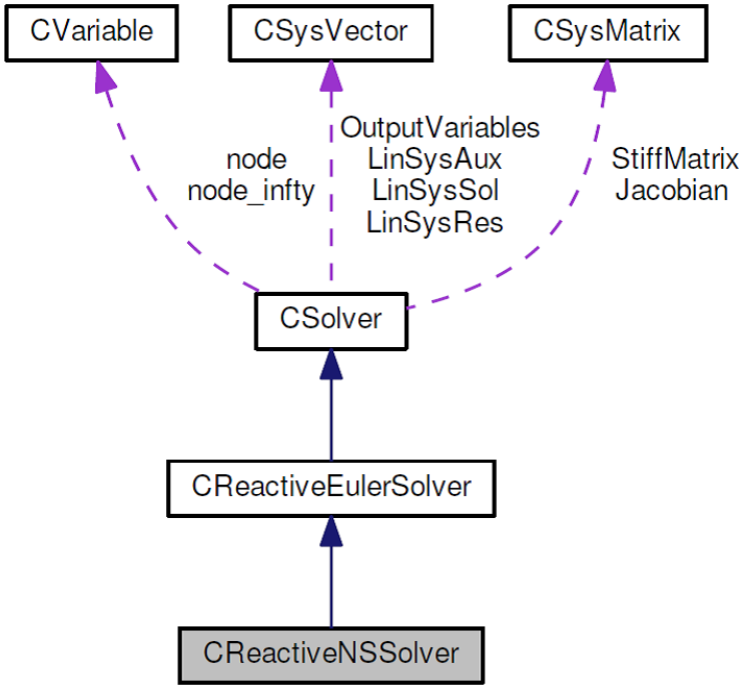

Figure 4.2: Diagram for solver class

These solver classes contain functions for computing each spatial term of the PDE: we find loops over the mesh edges to compute convective and viscous fluxes and loops over the mesh nodes to compute source terms as well as routines for imposing suitable boundary conditions.
Let us report here some parts of the constructor for the class **CReactiveEulerSolver**

```
/*--- Read freestream mass fractions. ---*/
MassFrac_Inf = RealVec(config->GetMassFrac_FreeStream(),
                config->GetMassFrac_FreeStream() + nSpecies);

/*--- Check right order of species in configuration file ----*/
int rank = MASTER_NODE;
#ifdef HAVE_MPI
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
#endif
if(iMesh == MESH_0 && rank == MASTER_NODE)
  Check_FreeStream_Species_Order(config);

/*--- Perform the non-dimensionalization for the flow equations using the
    specified reference values. ---*/
US_System = (config->GetSystemMeasurements() == US);
SetNondimensionalization(geometry, config, iMesh);
...
Density_Inf     = config->GetDensity_FreeStreamND();
Pressure_Inf    = config->GetPressure_FreeStreamND();
```

```
Temperature_Inf = config->GetTemperature_FreeStreamND();

Velocity_Inf    = RealVec(config->GetVelocity_FreeStreamND(),
    config->GetVelocity_FreeStreamND() + nDim);

/*--- Initialize the far-field state ----*/
node_infty = new CReactiveEulerVariable(Pressure_Inf, MassFrac_Inf,
    Velocity_Inf, Temperature_Inf, nDim, nVar, nSpecies,
nPrimVar, nPrimVarGrad, nPrimVarLim, library, config);

/*--- Allocate a CVariable array for each node of the mesh ---*/
node = new CVariable*[nPoint];

if(!restart || iMesh != MESH_0) {
  /*--- Initialize the solution to the far-field state everywhere. ---*/
  for(iPoint = 0; iPoint < nPoint; ++iPoint)
    node[iPoint] = new CReactiveEulerVariable(Pressure_Inf, MassFrac_Inf,
        Velocity_Inf, Temperature_Inf, nDim, nVar, nSpecies, nPrimVar,
        nPrimVarGrad, nPrimVarLim, library, config);
}
else
  Load_Restart(geometry, config);

/*--- Use a function to check that the initial solution is physical ---*/
Check_FreeStream_Solution(config);
```

As it can be easily noticed there is an auxiliary function called **Load_Restart** that allows the software to start from a previously stored solution. Its main core is the following:

```
/*--- The first line is the header ---*/
std::getline(restart_file, text_line);
unsigned long index;
su2double dull_val;
for(iPoint_Global = 0; iPoint_Global < geometry->GetGlobal_nPointDomain();
    ++iPoint_Global) {
  std::getline(restart_file, text_line);
  std::istringstream point_line(text_line);
  if(iPoint_Global >= geometry->GetGlobal_nPointDomain()) {
    sbuf_NotMatching = 1;
    break;
  }

  /*--- Retrieve local index. If this node from the restart file lives
  on the current processor, we will load and instantiate the vars. ---*/
  MI = Global2Local.find(iPoint_Global);
  if(MI != Global2Local.end()) {
    iPoint_Local = Global2Local[iPoint_Global];
    if(nDim == 2)
     point_line >> index >> dull_val >> dull_val >> Solution[RHO_INDEX_SOL]
         >> Solution[RHOVX_INDEX_SOL] >>
     Solution[RHOVX_INDEX_SOL + 1] >> Solution[RHOE_INDEX_SOL];
    else if(nDim == 3)
      point_line >> index >> dull_val >> dull_val >> dull_val >>
          Solution[RHO_INDEX_SOL] >> Solution[RHOVX_INDEX_SOL] >>
      Solution[RHOVX_INDEX_SOL + 1] >> Solution[RHOVX_INDEX_SOL + 2] >>
          Solution[RHOE_INDEX_SOL];
      for(unsigned short iSpecies = 0; iSpecies < nSpecies; ++iSpecies)
        point_line >> Solution[RHOS_INDEX_SOL + iSpecies];
```

```
    node[iPoint_Local] = new CReactiveEulerVariable(Solution, nDim, nVar,
        nSpecies, nPrimVar, nPrimVarGrad, nPrimVarLim, library, config);
    iPoint_Global_Local++;
  }
}
```

Moreover we report briefly the secant method to find temperature in case of subsonic inlet which has been discussed in the previous section

```
/*--- Auxiliary function to impose adiabaticity ---*/
auto f = std::function<su2double(su2double)>([&](su2double T){
  su2double hb = library->ComputeEnthalpy(T, Ys);
  su2double cb = std::sqrt(Gamma*Rgas*T);
  su2double Vb = (Riemann - 2.0*cb/Gamma_Minus_One)/alpha;
  return hb + 0.5*Vb*Vb;
});

/*--- Set parameters for secant method to find temperature ---*/
bool NRconvg, Bconvg;
su2double NRtol = 1.0e-9; // Tolerance for the Secant method
su2double Btol = 1.0e-6; // Tolerance for the Bisection method
unsigned short maxNIter = 15;     // Maximum Secant method iterations
unsigned short maxBIter = 100;     // Maximum Bisection method iterations
unsigned short iIter;

su2double Told = Ttot + 1.0;
su2double Tcurr = Ttot;
su2double Tnew;
NRconvg = false;

/*--- Execute a secant root-finding method to find the inlet temperature
    (TRAPEZOIDAL) ---*/
for(iIter = 0; iIter < maxNIter; ++iIter) {
  su2double tmp = f(Tcurr);
  su2double F = tmp - Tot_Enthalpy;
  su2double dF = tmp - f(Told);
  Tnew = Tcurr - F*(Tcurr - Told)/dF;

  /*--- Check for convergence ---*/
  if(std::abs(Tnew - Tcurr) < NRtol) {
   NRconvg = true;
   break;
  }
  else {
   Told  = Tcurr;
   Tcurr = Tnew;
  }
}

if(NRconvg)
  V_inlet[T_INDEX_PRIM] = Tcurr;
else {
  /*--- Execute the bisection root-finding method ---*/
  Bconvg = false;
  su2double Ta = 300.0/config->GetTemperature_Ref();
  su2double Tb = Ttot;
  for(iIter = 0; iIter < maxBIter; ++iIter) {
   Tcurr = (Ta + Tb)/2.0;
   su2double F = f(Tcurr) - Tot_Enthalpy;
```

```
   if(std::abs(F) < Btol) {
     V_inlet[T_INDEX_PRIM] = Tcurr;
     Bconvg = true;
     break;
   }
   else {
    if(F > 0.0)
      Ta = Tcurr;
    else
      Tb = Tcurr;
   }
 }

 /*--- If absolutely no convergence, then something is going really wrong
     ---*/
 if(!Bconvg)
   throw std::runtime_error("Convergence not achieved for bisection method
       in inlet boundary condition");
}
```

Eventually we report a typical loop used in order to call the functions that evaluate the flux and that are implemented in the **CNumerics** class as explained later on.

```
/*--- Loop over all the edges ---*/
for(iEdge = 0; iEdge < geometry->GetnEdge(); ++iEdge) {
  /*--- Points in edge and normal vectors ---*/
  iPoint = geometry->edge[iEdge]->GetNode(0);
  jPoint = geometry->edge[iEdge]->GetNode(1);
  numerics->SetNormal(geometry->edge[iEdge]->GetNormal());

  /*--- Get primitive variables ---*/
  auto V_i = node[iPoint]->GetPrimitive();
  auto V_j = node[jPoint]->GetPrimitive();
  ...
  /*--- Set primitive variables without reconstruction ---*/
  numerics->SetPrimitive(V_i, V_j);
  if(implicit)
    numerics->SetSecondary(node[iPoint]->GetdPdU(),node[jPoint]->GetdPdU());

  /*--- Compute the residual ---*/
  numerics->ComputeResidual(Res_Conv, Jacobian_i, Jacobian_j, config);

  /*--- Check for NaNs before applying the residual to the linear system
      ---*/
  bool err = !std::none_of(Res_Conv, Res_Conv + nVar,
      [](su2double elem){return std::isnan(elem);});
  if(implicit) {
   if(!err) {
    for(iVar = 0; iVar < nVar; ++iVar) {
      err = !std::none_of(Jacobian_i[iVar], Jacobian_i[iVar] + nVar,
          [](su2double elem){return std::isnan(elem);});
      err = err || !std::none_of(Jacobian_j[iVar], Jacobian_j[iVar] + nVar,
          [](su2double elem){return std::isnan(elem);});
      if(err)
        break;
     }
    }
  }
```

```
  /*--- Update residual value ---*/
  if(!err) {
    LinSysRes.AddBlock(iPoint, Res_Conv);
    LinSysRes.SubtractBlock(jPoint, Res_Conv);

    /*--- Set implicit Jacobians ---*/
    if(implicit) {
      Jacobian.AddBlock(iPoint, iPoint, Jacobian_i);
      Jacobian.AddBlock(iPoint, jPoint, Jacobian_j);
      Jacobian.SubtractBlock(jPoint, iPoint, Jacobian_i);
      Jacobian.SubtractBlock(jPoint, jPoint, Jacobian_j);
    }
  }
  else
    throw std::runtime_error("NaN found in the upwind residual");
} /*--- End loop over edges ---*/
```

4.1.2 CVariable

As we can see in (4.2) the solver classes refer to the CVariable class for storing unknowns and other variables pertinent to the PDE at each mesh node and for our simulations we created, accordingly to the specific problem, a suitable child class called **CReactiveEulerVariable** or **CReactiveNSVariable**

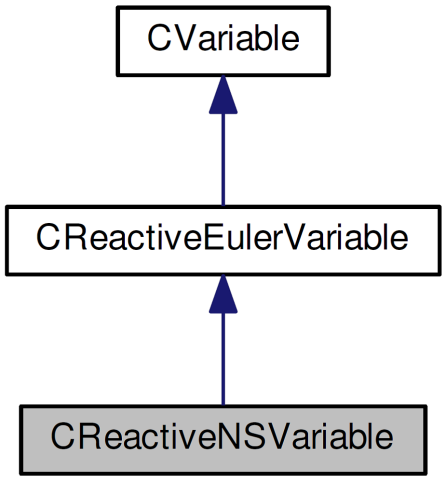

Figure 4.3: Diagram for variable class

An important consideration when we deal with equations or system of equations like (2.1) is the non-dimensionalization in order to avoid undesirable scaling effects due to units. The particular scheme chosen for SU2 can be found in Table (4.1) and (4.2)

Variables	Value	SI
Length	l_{ref}(input)	m
Pressure	p_{ref}(input)	$\mathrm{kg\,m^{-1}\,s^{-2}}$
Density	ρ_{ref}(input)	$\mathrm{kg\,m^{-3}}$
Temperature	T_{ref}(input)	K

Table 4.1: Independent variables for which reference values can be arbitrarily chosen

Variables	Value	SI
Velocity	$u_{ref} = \sqrt{p_{ref}/\rho_{ref}}$	$\mathrm{m\,s^{-1}}$
Time	$t_{ref} = l_{ref}u_{ref}$	s
Dynamic Viscosity	$\mu_{ref} = \rho_{ref}u_{ref}l_{ref}$(input)	$\mathrm{kg\,m^{-1}\,s^{-1}}$
Specific energy	$e_{ref} = u_{ref}^2$	$\mathrm{m^2\,s^{-2}}$
Specific enthalpy	$h_{ref} = e_{ref}$	$\mathrm{m^2\,s^{-2}}$
Gas Constant	$R_{ref} = e_{ref}/T_{ref}^2$	$\mathrm{m^2\,s^{-2}\,K^{-1}}$
Heat capacity (constant pressure)	$c_{p_{ref}} = R_{ref}$	$\mathrm{m^2\,s^{-2}\,K^{-1}}$
Heat capacity (constant volume)	$c_{v_{ref}} = R_{ref}$	$\mathrm{m^2\,s^{-2}\,K^{-1}}$
Turbulent kinetic energy	$k_{ref} = u_{ref}^2$	$\mathrm{m^2\,s^{-2}}$
Turbulent specific dissipation	$\omega_{ref} = u_{ref}/l_{ref}$	$\mathrm{s^{-1}}$

Table 4.2: Independent variables for which reference values can be arbitrarily chosen

Unlike the original implementation we allow the user to choose the reference length (which was prior fixed to 1.0 m) in order to allow more flexibility and to take into account the mesh dimension which will be hopefully related to the characteristic length of the problem.

```
//Length_Ref = 1.0; //<---- NOTE: this should be given an option or set as
    a const
/*!\brief REF_LENGTH\n DESCRIPTION: Reference length for
    adimensionalitazion (1.0 m by default) \ingroup Config*/
addDoubleOption("REF_LENGTH", Length_Ref, 1.0);
```

The most important function is the one that allows at the beginning of each iteration to pass from conserved to primitive variables before computing fluxes: it is called **Cons2PrimVar** and its interesting part is the computation of temperature from total energy

```
/*--- Translational-Rotational Temperature ---*/
const su2double Rgas =
    library->ComputeRgas(Ys)/config->GetGas_Constant_Ref();
const su2double C1 = (-rhoE + 0.5*rho*sqvel)/(rho*Rgas);
const su2double C2 = 1.0/Rgas;

/*--- Pick initial state and start algorithm ---*/
T = V[T_INDEX_PRIM];
Told = T + 1.0;
for(iIter = 0; iIter < maxNIter; ++iIter) {
  /*--- Execute a secant root-finding method to find T ---*/
  su2double dim_temp, dim_temp_old;
  dim_temp = T*config->GetTemperature_Ref();
  dim_temp_old = Told*config->GetTemperature_Ref();
  if(US_System) {
   dim_temp *= 5.0/9.0;
   dim_temp_old *= 5.0/9.0;
  }
  hs_old = library->ComputeEnthalpy(dim_temp_old,
      Ys)/config->GetEnergy_Ref();
  hs = library->ComputeEnthalpy(dim_temp, Ys)/config->GetEnergy_Ref();
  if(US_System) {
   hs_old *= 3.28084*3.28084;
   hs *= 3.28084*3.28084;
  }
```

```
  f = T - C1 - C2*hs;
  df = T - Told + C2*(hs_old-hs);
  Tnew = T - f*(T-Told)/df;

  /*--- Check for convergence ---*/
  if(std::abs(Tnew - T) < NRtol) {
   NRconvg = true;
   break;
  }
  else {
   Told = T;
   T = Tnew;
  }
}

/*--- If the secant method has converged, assign the value of T.
     Otherwise execute a bisection root-finding method ---*/
if(NRconvg)
  V[T_INDEX_PRIM] = T;
else {
  Bconvg = false;
  su2double Ta = Tmin;
  su2double Tb = Tmax;
  for(iIter = 0; iIter < maxBIter; ++iIter) {
   T = (Ta + Tb)/2.0;
   su2double dim_temp = T*config->GetTemperature_Ref();;
   if(US_System)
    dim_temp *= 5.0/9.0;
    hs = library->ComputeEnthalpy(dim_temp, Ys)/config->GetEnergy_Ref();
    if(US_System)
     hs *= 3.28084*3.28084;
     f = T - C1 - C2*hs;

     if(std::abs(f) < Btol) {
       V[T_INDEX_PRIM] = T;
       Bconvg = true;
       break;
     }
     else {
      if(f > 0)
        Ta = T;
      else
       Tb = T;
    }
  }

  /*--- If absolutely no convergence, then something is going really wrong
      ---*/
  if(!Bconvg)
    throw std::runtime_error("Convergence not achieved for bisection
        method");
}
```

4.1.3 CNumerics

This class discretizes each system of governing equations using the numerical schemes specified in the input file. There are several child that provide discretization techniques

for convective fluxes,viscous fluxes and source terms. During a single iteration methods in the CNumerics classes would compute the flux contributions and Jacobians (in case of implicit computations) at each node (using the variables stored in the CVariable class). These flux and Jacobian values are transferred back to the CSolver class which calls routines within CSysMatrix in order to solve the resulting linear system of equations for the solution update.

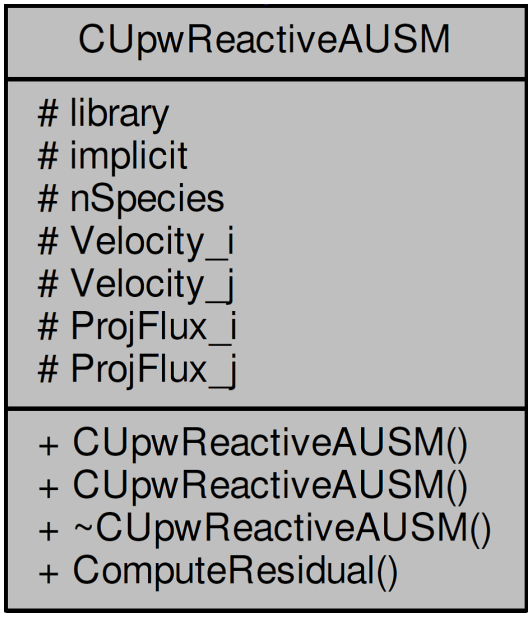

Figure 4.4: UML diagram for the class CReactiveUpwAUSM

Herein we reported the UML diagram with some members of the new class **CUpwReactiveAUSM** used for the convective fluxes: the function that effectively computes the flux and eventually the Jacobians is called **ComputeResidual**.

It is worth to notice that the SU2 Code widely employes **run-time polymorphism**: therefore in order to implement any kind of new physical model it is necessary to derive from the base class **CNumerics** and to implement accordingly the **virtual** function **ComputeResidual**. For our purpose three more classes were needed: one for viscous fluxes at boundary nodes (**CAvgGradReactive_Boundary**), one for viscous fluxes at internal nodes(**CAvgGradReactive_Flow**) and one for source fluxes (**CSourceReactive**).
The reason why we need two classes for viscous fluxes is that for the internal nodes we apply an averaged corrected gradient formula which can not be applied on the boundary (see [2]): for this reason we choose to take **CAvgGradReactive_Boundary** as **parent** class and **CAvgGradReactive_Flow** as **child** as it can be noticed by the diagram in Figure 4.5.

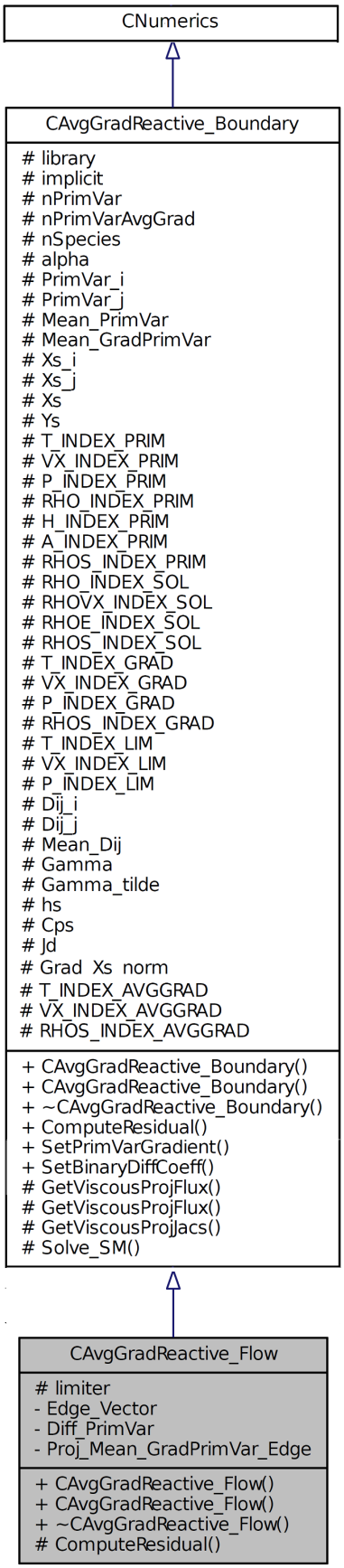

Figure 4.5: Diagram for the numerics classes

Finally we underline one important aspect from a computational point of view: for

viscous fluxes we need the values of diffusion fluxes from Stefan-Maxwell equations (2.10) introduced in previous chapters: the simplest way to solve the linear system using BiCGStab is to rely on the external library **EIGEN** [22] and this procedure is encapsulated in the routine **Solve_SM**

```
/*--- Solution of Stefan-Maxwell equation using artificial diffusion
    modified matrix ---*/
void CAvgGradReactive_Boundary::Solve_SM(const su2double val_density, const
    su2double val_alpha, const RealMatrix& val_Dij, const RealVec& val_xs,
    const Vec& val_grad_xs, const RealVec& val_ys) {

  const su2double toll = 1e-11;

  /*--- Rename for convenience ---*/
  su2double rho = val_density;
  su2double alpha = val_alpha;

  /*--- Compute original matrix of Stefan-Maxwell equations ---*/
  Gamma = library->GetGamma(rho, val_xs, val_ys, val_Dij);

  /*--- Add artificial diffusion part ---*/
  for(unsigned short iSpecies = 0; iSpecies < nSpecies; ++iSpecies)
    for(unsigned short jSpecies = 0; jSpecies < nSpecies; ++jSpecies)
      Gamma_tilde(iSpecies,jSpecies) = Gamma(iSpecies,jSpecies) +
          alpha*val_ys[iSpecies];

  Eigen::BiCGSTAB<RealMatrix> bicg(Gamma_tilde);
  bicg.setTolerance(toll);
  Jd = bicg.solve(-val_grad_xs);
}
```

4.2 Reacting Model Library

This section describes the structure of the library implemented for computing the physical and chemical properties in the mixture.
In order to allow each user to use its own version of the library and to handle the creation of polymorphic object we use a simple version of the factory design pattern.

```
/*!
* \class Factory
* \brief Class for loading libraries at run-time.
* \author G. Orlando
*/
template<class Base>
class Factory: public Common::NotCopyable<Factory<Base>> {
public:

/*
* \brief Constructor of this simple factory
* \param[in] lib_name - Name of the desired library
* \param[in] config_name - Name of the file to read in order to configure
    the library
* \param[in] lib_path - Path where the library is present
*/
Factory(const std::string& lib_name, const std::string& config_name, const
    std::string& lib_path);

/*
* \brief Factory destrcutor
*/
~Factory() {}

/*
* \brief Get the library
*/
std::shared_ptr<Base> GetLibraryPtr(void) const {
  return my_library;
}

private:
  std::shared_ptr<Base> my_library; /*!< \brief Pointer to Base in order to
      access concrete version. */

}; /*--- End of class Factory ---*/
```

Class **Factory** is templatized with the polymorphic *Base* type as a parameter and serves as a registry point for provider of different types if needed.

In our case, within the namespace **Framework**, the *Base* type is represented by the class **PhysicalChemicalLibrary**, while **ReactingModelLibrary** is the concrete implementation of the functions virtually declared in the *Base*.

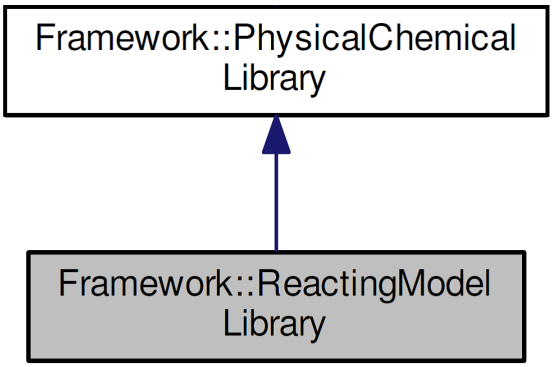

Figure 4.6: Hierarchy for reacting model library

In the solver classes we add as a member a static shared pointer to the *Base* class so that we have one version of the library and all functions and classes can rely on that if necessary.

```
class CReactiveEulerVariable: public CVariable {
public:
  typedef std::vector<su2double> RealVec;
  typedef su2double** SU2Matrix;
  typedef std::shared_ptr<Framework::PhysicalChemicalLibrary> LibraryPtr;
  ....
}
/*! \class CReactiveEulerSolver
*  \brief Main class for defining a solver for chemically reacting inviscid
    flows.
*  \author G. Orlando.
*/
class CReactiveEulerSolver: public CSolver {
public:
  using RealVec = CReactiveEulerVariable::RealVec;
  using RealMatrix = CReactiveNSVariable::RealMatrix;
  using LibraryPtr = CReactiveEulerVariable::LibraryPtr;

protected:
  static LibraryPtr library; /*!< \brief Smart pointer to the library that
      computes physical-chemical properties. */
  ....
}
```

To the library is also delegated the reading of data mixture,of chemistry and thermodynamical and transport properties data through a suitable function called **Setup**

```
/*--- Setup library ---*/
void ReactingModelLibrary::Setup(void) {
  if(!Lib_Setup) {
   Le = 1.0;
   /*--- If nobody has configured the library path, we try to do it here
       with a default value ---*/
   if(Lib_Path == "") {
     std::cout<<"Library path set to default"<<std::endl;
     auto base_dir = std::experimental::filesystem::current_path().string();
     Lib_Path = base_dir;
   }

   std::vector<std::string> list_file;
   std::ifstream config_file(Config_File);
   if(config_file.is_open()) {
     while(config_file.good() && !config_file.eof()) {
       std::string curr_line;
       std::getline(config_file,curr_line);
       if(!curr_line.empty() && !std::ispunct(curr_line.at(0)))
         list_file.push_back(curr_line);
     }
   }
   else {
      std::cerr<<"Unable to open the specified file with all the file names
          for setting library."<<std::endl;
      std::exit(1);
   }

   /*--- Read mixture file: it needs to be the first to check exactness of
       chemical reactions and properties ---*/
   std::string file_mix = list_file.at(0);
   ReadDataMixture(file_mix);
   std::cout<<"Mixture Data read"<<std::endl;

  /*--- Check we have the right number of files ---*/
  using size_type = std::vector<std::string>::size_type;
  size_type max_n_file = 2*nSpecies + 2;
  size_type n_file = list_file.size();
  SU2_Assert((n_file == max_n_file) || (n_file == max_n_file - 1), "The
      number of files present in the configuration file is wrong");

  /*--- Set the specific gas constants ---*/
  SetRiGas();

  /*--- Read chemistry file (if present) ---*/
  int buffer_chemistry = 1;
```

```
  nReactions = 0;
  if(n_file == max_n_file) {
    /*--- We assume that chemistry file is the second in the list if its
        present ---*/
    std::string file_chem = list_file[1];
    ReadDataChem(file_chem);
    std::cout<<"Chemical Reactions read"<<std::endl;
    buffer_chemistry = 0;
  }

  /*--- We assume that data correspond to the species declared at the
      beginning of the file
        and a transport file is followed by a thermodynamic file
        (I can't check the content so it seems reasonable) ---*/
   std::string file_transp, file_thermo;
   for(unsigned short iSpecies = 0; iSpecies < nSpecies; ++iSpecies) {
     file_transp = list_file[iSpecies*2 + 2 - buffer_chemistry];
     ReadDataTransp(file_transp);
     file_thermo = list_file[iSpecies*2 + 3 - buffer_chemistry];
     ReadDataThermo(file_thermo);
   }
   Lib_Setup = true;
   std::cout<<"Library set."<<std::endl;
   std::cout<<std::endl;
}
else
   throw Common::NotSetup("Trying to setup again without calling unsetup
       first.");
}
```

where **Common::NotSetup** is a simple **exception** to underline that the library has not been correctly instantiated.

We report here also the function **ReadDataChem** employed to read all data related to chemical reactions

```
/*--- Reading data about chemistry. ---*/
void ReactingModelLibrary::ReadDataChem(const std::string& f_name) {
  /*--- Local variables ---*/
  std::string line;
  unsigned n_line = 0;
  unsigned n_reac_read = 0;

  std::ifstream chemfile(Lib_Path + "/" + f_name);
  if(chemfile.is_open()) {
    /*--- Clear for safety ---*/
    Stoich_Coeffs_Reactants.resize(0,0);
```

```
Stoich_Coeffs_Reactants.resize(0,0);
Stoich_Coeffs_Products_Exp.resize(0,0);
Stoich_Coeffs_Reactants_Exp.resize(0,0);
Reversible_Reactions.clear();
As.clear();
Betas.clear();
Temps_Activation.clear();
while(chemfile.good() && !chemfile.eof()) {
  std::getline(chemfile,line);

  /*--- Check if we encounter the termination character ---*/
  if(line == "STOP")
    break;
  /*--- We avoid clearly reading empty lines and comments in the file
      ---*/
  if(!line.empty() && !std::ispunct(line.at(0))) {
    if(n_line == 0) {
      std::istringstream curr_line(line);
      curr_line>>nReactions;
      SU2_Assert(!curr_line.fail(), "You have to specify the number of
          reactions before proceding");

      /*--- Resize and reserve space for vectors ---*/
      Stoich_Coeffs_Reactants.resize(nSpecies,nReactions);
      Stoich_Coeffs_Reactants.setZero();
      Stoich_Coeffs_Products.resize(nSpecies,nReactions);
      Stoich_Coeffs_Products.setZero();
      Stoich_Coeffs_Reactants_Exp.resize(nReactions,nSpecies);
      Stoich_Coeffs_Reactants_Exp.setZero();
      Stoich_Coeffs_Products_Exp.resize(nReactions,nSpecies);
      Stoich_Coeffs_Products_Exp.setZero();

      Forward_Rates.resize(nReactions);
      Backward_Rates.resize(nReactions);
      Kc.resize(nReactions);
      Kc_Derivatives.resize(nReactions);

      Reversible_Reactions.reserve(nReactions);
      As.reserve(nReactions);
      Betas.reserve(nReactions);
      Temps_Activation.reserve(nReactions);

      n_line++;
    }
    else if(n_line == 1) {
      std::istringstream curr_line(line);
      std::string kind_units;
      curr_line>>kind_units;
      if(kind_units == "CGS")
```

```
        CGS_Units = true;
      else if(kind_units == "SI")
        CGS_Units = false;
      else
        throw std::out_of_range("Unknown option for the type of units
            measure");

      n_line++;
    }
    else {
      bool is_rev;
      if(n_line 'mod' 2 == 0 && n_line < nReactions + 3) {
        is_rev = (line.find('<') != std::string::npos);
        Reversible_Reactions.push_back(is_rev);
        n_reac_read++;
        ReadReactSpecies(line,is_rev,n_reac_read);
      }
      else if(n_line 'mod' 2 == 1 && n_line < nReactions + 4) {
        ReadChemCoefs(line);
      }
      else {
        ReadExtraData_Rates(line);
        ReadExtraData_ForwardExponent(line);
        ReadExtraData_BackwardExponent(line);
      }

      n_line++;
    }
  }
}
SU2_Assert(n_reac_read == nReactions, "The number of reactions detected
    doesn't match nReactions");
chemfile.close();
unsigned short iReac, iSpecies;

/*--- Try automatic computations of exponents of products in case
    backward data were not already available ---*/
for(iReac = 0; iReac < nReactions; ++iReac) {
  if(Reversible_Reactions[iReac] &&
      Available_Backward_Rate.count(iReac) == 0) {
    for(iSpecies = 0; iSpecies < nSpecies; ++iSpecies)
      Stoich_Coeffs_Products_Exp(iReac,iSpecies)=
          Stoich_Coeffs_Reactants_Exp(iReac,iSpecies) +
      Stoich_Coeffs_Products(iSpecies,iReac) -
          Stoich_Coeffs_Reactants(iSpecies,iReac);
  }
}

/*--- Update to SI units the Arrhenius constants if needed ---*/
```

```
    if(CGS_Units) {
      for(iReac = 0; iReac < nReactions; ++ iReac) {
        double sum_forward_exp =
            Stoich_Coeffs_Reactants_Exp.row(iReac).sum();
        As[iReac] *= std::pow(10.0, 6.0*(1.0 - sum_forward_exp));
        if(Available_Backward_Rate.count(iReac) == 1) {
          double sum_backward_exp =
              Stoich_Coeffs_Products_Exp.row(iReac).sum();
          As_back[iReac] *= std::pow(10.0, 6.0*(1.0 - sum_backward_exp));
        }
      }
    }

    /*--- Save species with negative exponents for backward rates---*/
    Species_Negative_Exponent.resize(nReactions);
    for(iReac = 0; iReac < nReactions; ++iReac) {
      for(iSpecies = 0; iSpecies < nSpecies; ++iSpecies) {
        if(Stoich_Coeffs_Products_Exp(iReac,iSpecies) < 0.0)
          Species_Negative_Exponent[iReac].push_back(iSpecies);
      }
    }
  }
  else {
    std::cerr<<"Unable to open the chemical file: "<<f_name<<std::endl;
    std::exit(1);
  }
}
```

As it can be easily noticed we allow the user to express the Arrhenius constants both in **CGS** units (mol/cm^3) and **SI** units (mol/m^3) and the same holds also for the activation temperature where we can express the activation energy in /mol or directly the temperature in K. Moreover we save the species with negative exponents in order to avoid the rates going to infinity in case of zero or vanishing mass fractions.

Chapter 5

Numerical results

Single species entropy s_i, static enthalpy h_i and specific heat at constant pressure $C_{p,i}$ are taken from NASA polynomials as given by [4], while regarding transport properties (molecular viscosity and thermal conductivity) of the single species, the interpolations from [3] is used.
In this work we do not apply directly the aforementioned polynomials to compute transport and thermodynamic properties, but we employ spline interpolation techniques in order to obtain flexibility in the use of several user-defined models and to gather the computational way to proceed; this is performed in the following manner: in a preprocessing stage data tables for thermodynamic/transport properties are generated as text files using literature polynomial data with user-defined temperature interval ΔT and then the spline interpolation coefficients are generated during the setup phase.
Eventually in order to determine the interval which a desired temperature belongs to, the integer search algorithm has been employed: in this way the extrema of interval can be found through a simple division with a noticeable speed-up in the algorithm.

```
//
/*--- Sets the second derivative coefficients for natural spline ---*/
void SetSpline(const RealVec& x, const RealVec& y, const double yp1,
               const double ypn, RealVec& y2) {
  auto n = x.size();
  SU2_Assert(n > 1,"You have only one datum: cannot generate spline");
  SU2_Assert(y.size() == n,"The dimension of vectors x and y from data are not
      the same");
  SU2_Assert(std::is_sorted(x.cbegin(),x.cend()),"The vector of x is not
      ordered");
  auto it = std::adjacent_find(x.begin(), x.end());
  SU2_Assert(it == x.end(),"Some values that have been tabulated are not
      unique");

  bool is_equispaced = true;
  const auto step = x[1] - x[0];
  for(std::size_t i=2; i < n-1 && is_equispaced; ++i) {
```

```
    const auto curr_step = x[i] - x[i-1];
    if(curr_step != step)
      is_equispaced = false;
  }
  SU2_Assert(is_equispaced,"This version of spline works only with equispaced
      data since it uses integer division");

  unsigned long i, k;
  double p, qn, sig, un;
  RealVec u(n);
  y2.clear();
  y2.reserve(n);

  if(yp1 > 0.99e30) // The lower boundary condition is set either to be
    y2.push_back(0.0); //"natural"
  else {                 // or else to have a specified first derivative.
    y2.push_back(-0.5);
    u[0]=(3.0/(x[1]-x[0]))*((y[1]-y[0])/(x[1]-x[0])-yp1);
  }

  for(i=2; i<n; ++i) { // This is the decomposition loop of the tridiagonal
      algortihm
   sig = (x[i-1]-x[i-2])/(x[i]-x[i-2]);  // y2 and u are used for temporary
   p = sig*y2[i-2] + 2.0;           // storage of the decomposed
   y2.push_back((sig-1.0)/p);       // factors.
   u[i-1] = (y[i]-y[i-1])/(x[i]-x[i-1]) - (y[i-1]-y[i-2])/(x[i-1]-x[i-2]);
   u[i-1] = (6.0*u[i-1]/(x[i]-x[i-2])-sig*u[i-2])/p;
  }

  if(ypn > 0.99e30)      // The upper boundary condition is set either to be
    qn=un=0.0;         // "natural"
  else {            // or else to have a specified first derivative.
   qn=0.5;
   un=(3.0/(x[n-1]-x[n-2]))*(ypn-(y[n-1]-y[n-2])/(x[n-1]-x[n-2]));
  }
  y2[n-1]=(un-qn*u[n-2])/(qn*y2[n-2]+1.0);
  for (k=n-1; k>0; --k) // This is the backsubstitution loop of the tridiagonal
    y2[k-1]=y2[k-1]*y2[k]+u[k-1];  // algorithm.
}

//
/*--- Gets the value from natural spline ---*/
double GetSpline(const RealVec& x, const RealVec& y, const RealVec& y2,
                 const double value) {
  if(value < x[0] || value > x.back())
    throw std::out_of_range("The required temperature (" +
        std::to_string(value) + " K) is out of data range");

  unsigned long klo;
```

```
  double h, b, a, result;
  h = x[1] - x[0];
  klo = (value - x[0])/h + 1;
  a = (x[klo] - value)/h;
  b = (value - x[klo-1])/h;
  // Cubic spline polynomial is now evaluated.
  result = a*y[klo-1] + b*y[klo] + ((a*a*a - a)*y2[klo-1] +
                            (b*b*b - b)*y2[klo])*(h*h)/6.0;

  return result;
}
```

5.1 Inviscid Bump

The first case to test the correct behaviour of the code is represented by the inviscid bump in a channel: this example uses a 2D geometry that features a circular bump along the lower wall. This kind of flow is one of the benchmark problems for subsonic inlet and outlet boundary conditions in SU2 and therefore we can employ the already tested mesh. In detail the channel is of length 3 m with a height of 1 m and a circular bump centered along the lower wall with height 0.1 m, while the mesh is composed of quadrilaterals with 256 nodes along the length of the channel and 128 nodes along the height.

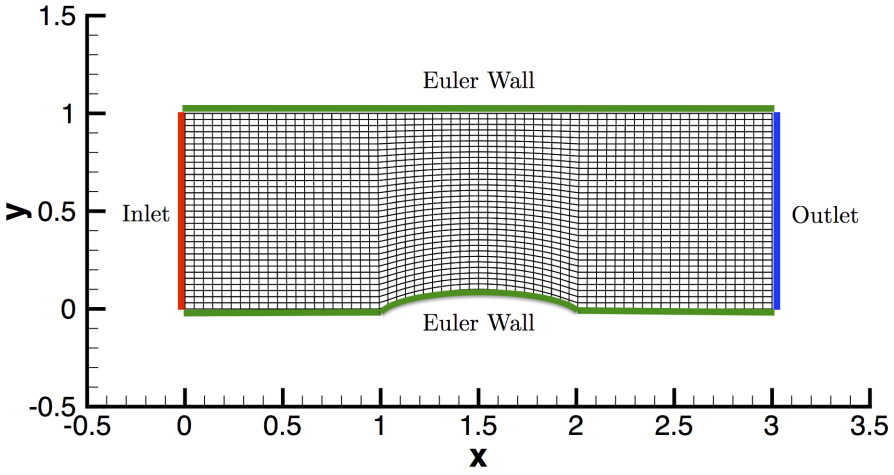

Figure 5.1: Computational mesh with highlighted boundary conditions

At inlet boundary **total boundary conditions** are imposed with a total temperature of 288.6 K and a total pressure of 102010.0 Pa, while the static pressure imposed

at outlet is equal to 101325.0 Pa.
Eventually the species chosen for the simulation is O_2. Let us compare the results at steady state with a second order scheme:

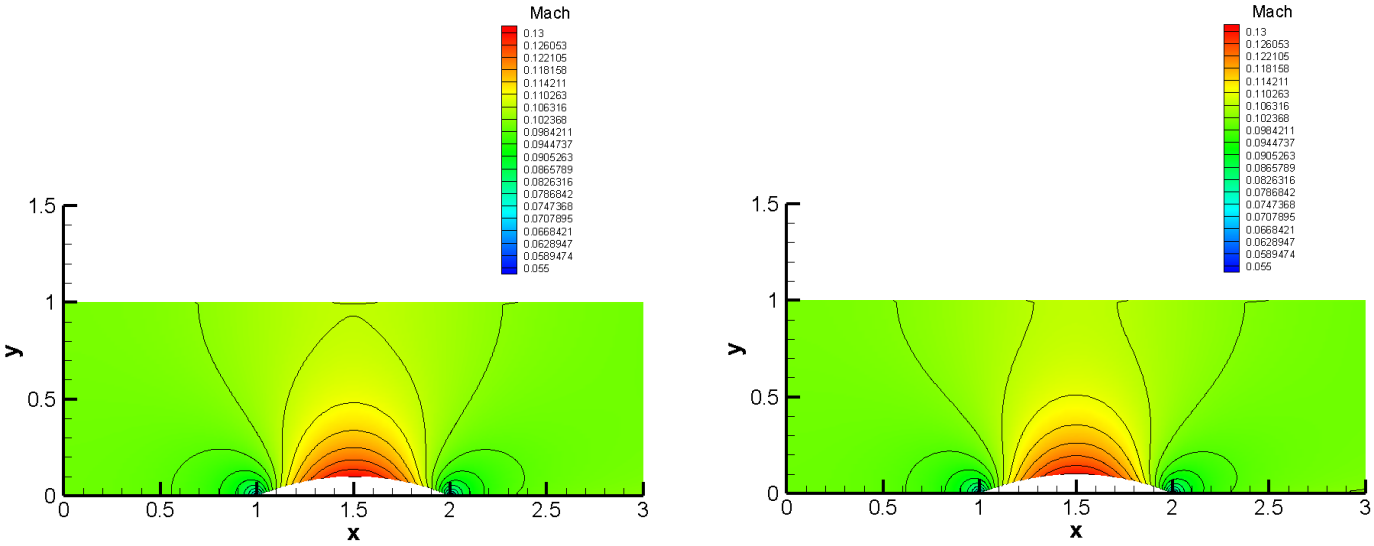

Figure 5.2: Comparison of Mach number contours between my version (left) and the original one (right)

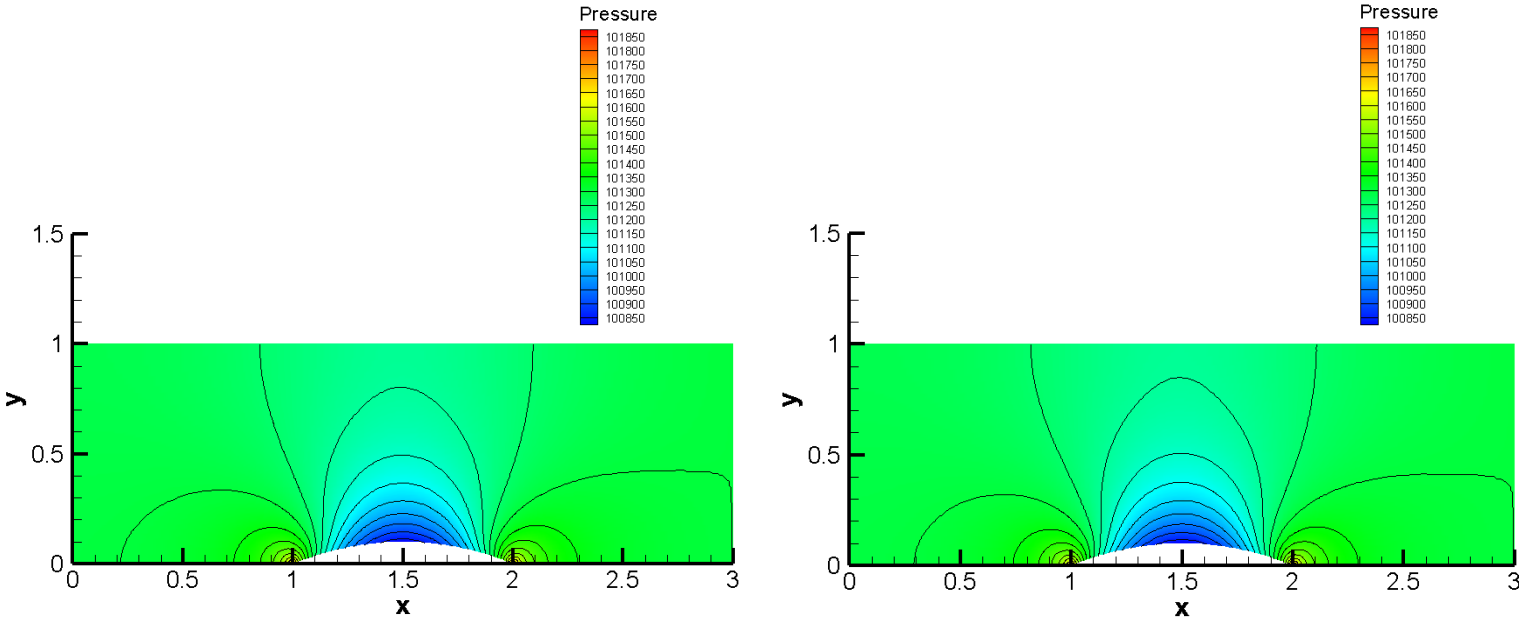

Figure 5.3: Comparison of pressure contours between my version (left) and the original one (right)

The results obtained with my version are in good agreement with respect to the original one and the expected symmetry due to the geometry is well caught. Finally another simulation involving three species (78% N_2, 20% O_2 and 2% CO_2) has been performed in order to verify the behaviour in case of multispecies flows.

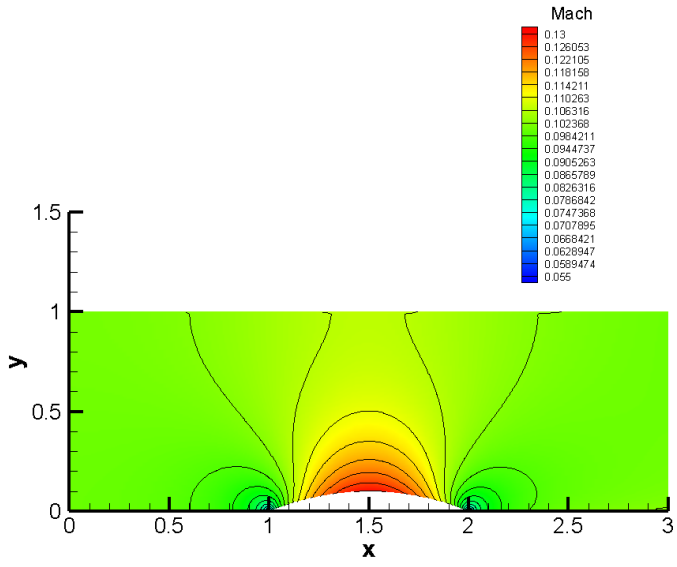

Figure 5.4: Mach number contour for multispecies simulation

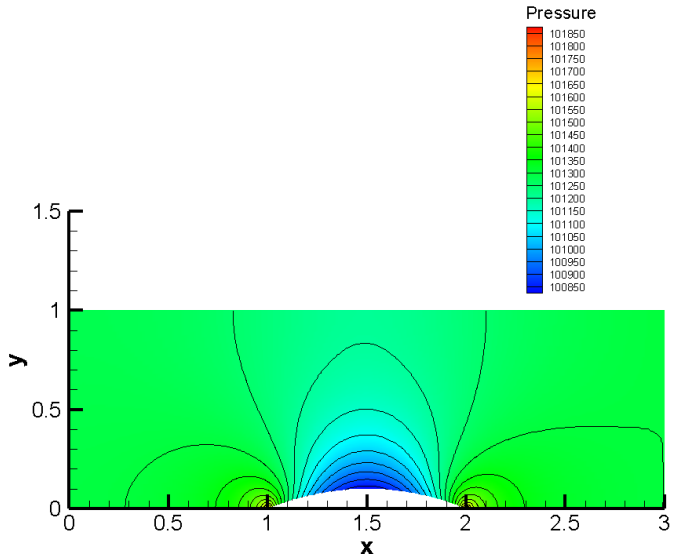

Figure 5.5: Pressure contour for multispecies simulation

Also in this case the simulation shows the expected symmetric behaviour with values very similar to the previous simulations since the gas constant values are not so different.

5.2 Diffusion in a channel

The second test case is the diffusion of two species inside a channel in order to verify if the modifications of Stefan-Maxwell equations previously described work properly.
The channel is long 1 m and height 0.06 m and the mesh consists of 120 x 40 nodes. As initial condition the channel is full of CO and in order to avoid pressure gradient at inlet the species denoted as CO_2 has the same mass fraction of O_2.
Eventually at inlet on the left we inject O_2 in the upper part and CO_2 in the lower part with a total temperature of 500K and a total pressure of 102000Pa.
These are the results:

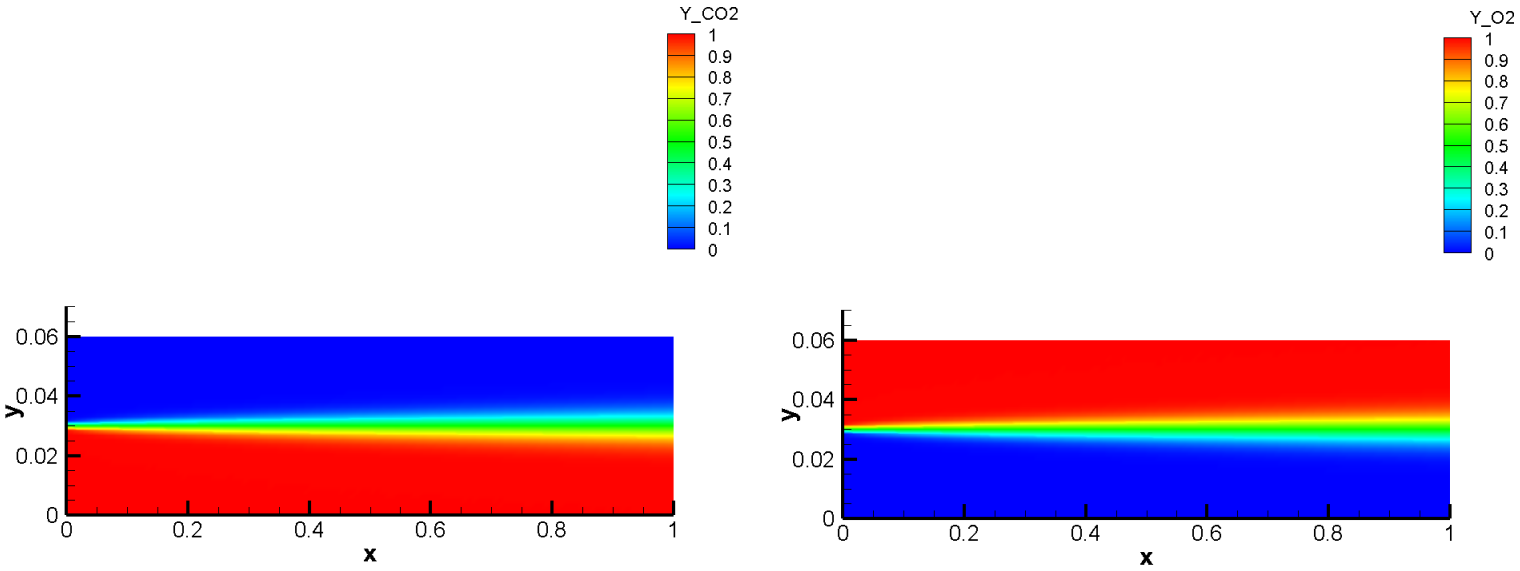

Figure 5.6: Steady state distribution of CO_2 mass fraction (left) and O_2 mass fraction (right)

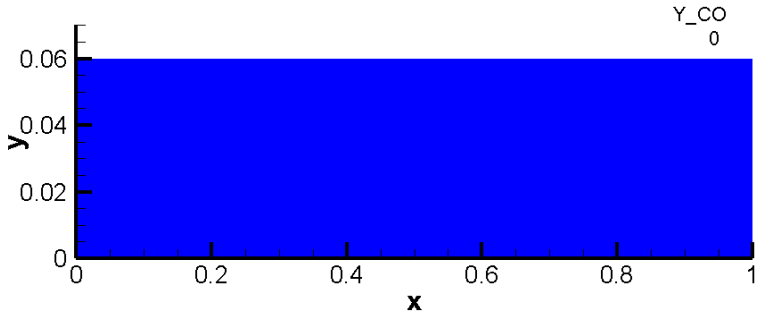

Figure 5.7: Steady state distribution of CO mass fraction

As it can be seen the diffusion phenomenon is well caught at interface and there are no residuals of CO as expected.

5.3 Laminar Flat Plate

Another interesting test case regards the capability to catch boundary layer; therefore a comparison analogous to the one previously described for the bump is performed.
The computational mesh for the flat plate is composed of quadrilaterals with 65 nodes in both the x and y directions. The flat plate is along the lower boundary of the domain (y = 0) starting at x = 0 m and is of length 0.3048 m.

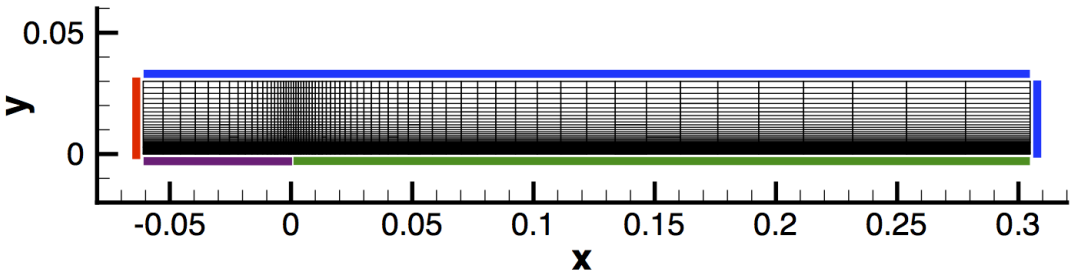

Figure 5.8: Computational mesh with highlighted boundary conditions

At inlet boundary total are imposed with a total temperature of 300.0 K and a total pressure of 100000.0 Pa, while the static pressure imposed at outlet is equal to 97250.0 Pa. Also in this case we choose O_2 and we compare the results between the original code and my version.

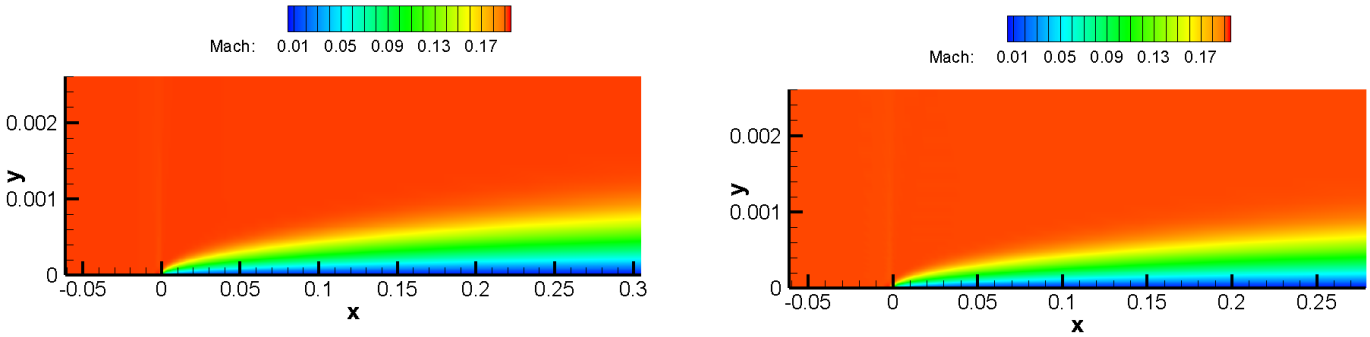

Figure 5.9: Comparison of Mach number contours between my version (left) and the original one (right)

Also in this case the two simulations match very well. Eventually we perform a simulation with three species taking the same composition described for the bump

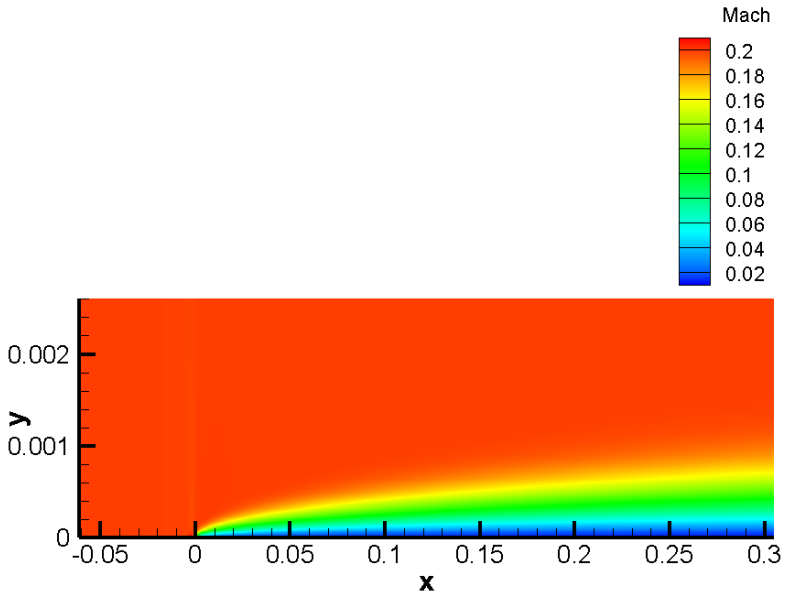

Figure 5.10: Mach number contour for multispecies simulation

The development of the boundary layer is present even in this case and this shows the capability of the code to treat multispecies flows.

5.4 Combustion

Eventually we perform a simulation of combustion processes inside aerospace engines. We chose a simple chemical model:

$$\begin{aligned} C_4H_6 + 3.5O_2 &\longrightarrow 4CO + 3H_2O \\ CO + 0.5O_2 &\longleftrightarrow CO_2 \end{aligned} \tag{5.1}$$

The chamber is 0.125 m and 0.006 m height. For what concerns the computational mesh we rely on [1] and therefore we know that a mesh with 100 x 80 elements is sufficient to reach convergence. Anyway in order to catch even better the behaviour near the walls we employ a tapered mesh with 90 cells in y direction and more specifically 20 nodes from 0 m to 0.001 m and from 0.005 m to 0.006 m
Since the chemistry is always stiff to treat, we first run a simulation without chemistry in order to reach a configuration more adapt to turn on the process. This simulation is run using a first order scheme up to convergence below 10^{-6} for density residual; as initial condition the chamber is full of O_2 at $T = 300$K with a speed of 6m/s along x direction.

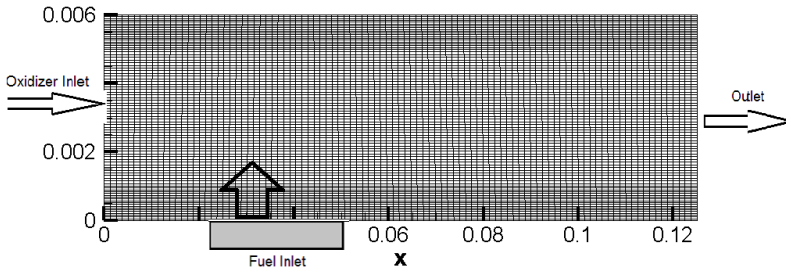

Figure 5.11: Computational mesh for chamber

Let us summarize now the employed boundary conditions: we underline that the walls are treated as isothermal no-slip walls.

Table 5.1: Boundary conditions

Boundary	**u [m/s]**	**v [m/s]**	**T [K]**	**p [Pa]**
Oxidizer Inlet	20	0	300	-
Fuel Inlet	0	0.86	800	-
Upper Wall	0	0	300	-
Lower Wall - Pre-Inlet	0	0	300	-
Lower Wall - Post-Inlet	0	0	600	-
Outlet	-	-	-	101325

Figure 5.12: Summary of boundary conditions

We show now the temperature contour of the simulation without chemistry.

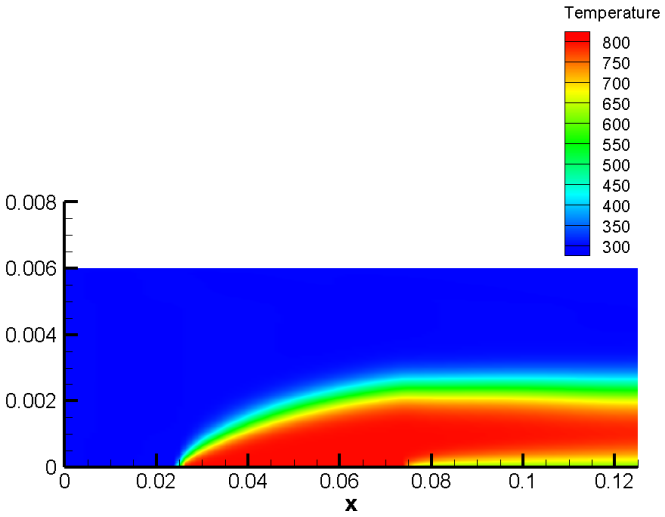

Figure 5.13

As we can see the convective and diffusive part worked very well and this is a good starting point to turn on the chemistry. As reported in [6] the forward and backward rate constants of this combustion model are computed as follows:

$$k_{f,1}(T) = 8.80 \cdot 10^{11} \exp(-30000/(R_{gas}T))$$
$$k_{f,2}(T) = 1.0 \cdot 10^{14.6} \exp(-40000/(R_{gas}T))$$
$$k_{b,2}(T) = 5.0 \cdot 10^{8} \exp(-40000/(R_{gas}T))$$

where R_{gas} is the universal gas constant in cal/mol. Moreover the rate exponents for the reactions are 1.0 for both C_4H_6 and O_2 in the first reaction, 1.0 for CO and 0.5 for O_2 in the forward contribution of second reaction and 1.0 for CO_2 in the backward contribution.

Another physical issue must be overtaken: if we introduce fuel to an oxidant, spontaneous ignition does not occur unless the temperature of the mixture exceeds the activation energy threshold required to maintain combustion: therefore we have to supply an ignition source to initiate combustion. This ignition source may be a heated surface or inlet mass flow that heats the gas mixture above the required ignition temperature. Often, however, it is the equivalent of a spark: a region at high temperature for a sufficient number of iterations where fuel/oxidizer is present enough that causes combustion to proceed: it is worth to notice that for this kind of simulations the CFL must be kept quite low in order to reach convergence.

```
if(ignition) {
  if(config->GetExtIter() < config->GetIgnitionIter()) {
    if(node[iPoint]->GetMassFraction(FUEL_INDEX) > 0.4 &&
        node[iPoint]->GetMassFraction(OXD_INDEX) > 0.2 &&
        node[iPoint]->GetTemperature() < config->GetIgnitionTemperature())
        bool tmp =
            node[iPoint]->SetTemperature(config->GetIgnitionTemperature());
  }
}
```

This piece of code is inside the function **SetPrimitiveVariables** of the class **CReactiveEulerSolver**.

In our case we set CFL to 0.2, the ignition temperature to 1700 K for 700 iterations and we increased also the Arrhenius parameter of first reaction to $2,90 \cdot 10^{13}$ in order to accelerate the kinetic chemistry. Then the simulation has been restarted using the right values up to residual norm below 10^{-4} and these are the results:

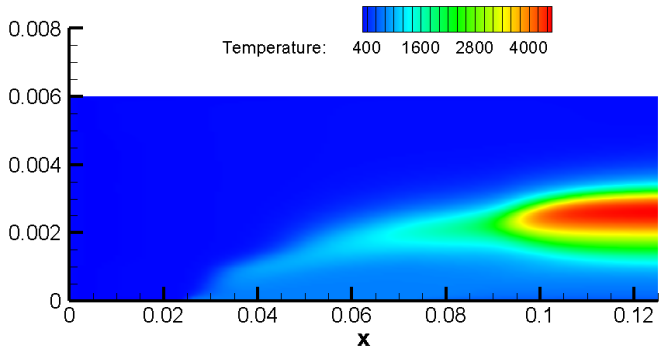

Figure 5.14: Temperature contour at steady state for combustion chamber

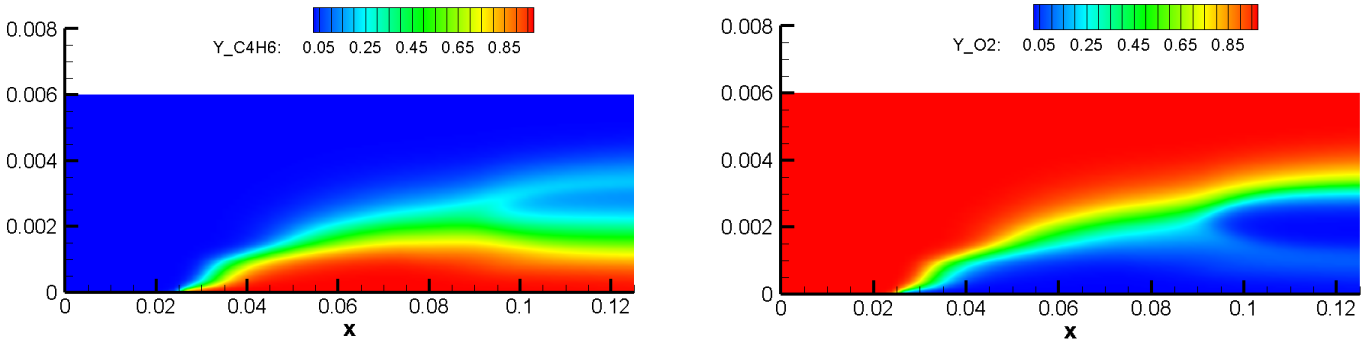

Figure 5.15: C4H6 (left) and O2 (right) contours at steady state

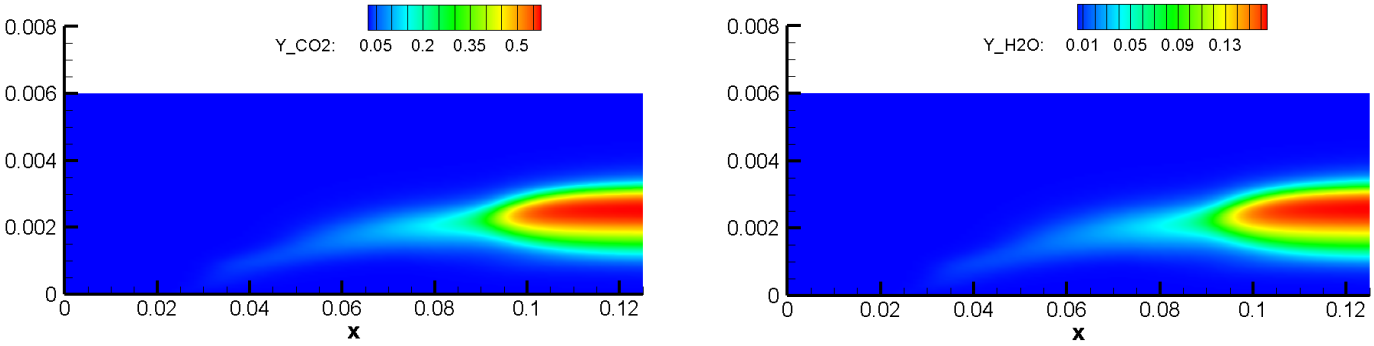

Figure 5.16: CO2 (left) and H2O (right) contours at steady state

Reactant distribution is coherent with inlet positioning: fuel (C4H6) remain in lower region and then it is progressively burnt, while oxidizer (O2) oxidizer remains in the top

region and reacts in the flame region.
Products distribution is coherent with peak temperature region, were chemical reactions are more intense.
Finally the peak temperature is higher than expected but this is due to the simplicity of the model as explained by [1].

Chapter 6

Conclusions

In previous sections we described the development of a numerical solver able to treat multispecies flows with and without chemistry; for future purposes it is worth to notice that the code is able to support 3D simulations.

Future developments and extensions are possible, specially in three directions:

- The employment of collision integrals to compute binary diffusion coefficients instead of semi-empirical formulas as in this work
- The possible implementation of the contribution to Jacobian due to diffusion fluxes with a different procedure either numerical or with another analytical approximation.
- The capability to treat turbulent flows: for this purpose it is necessary to pay particular attention to the modelling of energy terms (see [1]) and diffusion fluxes which is not standard.

Appendix A

A.1 AUSM Family

The discretization of the convective term leads to:

$$\int_{\Sigma_i} \boldsymbol{F}\boldsymbol{n} d\Sigma = \sum_{f=1}^{N_f} \boldsymbol{F_f} \Sigma_f \tag{A.1}$$

where $\boldsymbol{F_f} = \widetilde{\boldsymbol{F}}_{\boldsymbol{f}} \boldsymbol{n_f}$ is the numerical convective flux projected onto the normal $\boldsymbol{n_f}$ to the interface f with area Σ_f and N_f is the number of neighbour nodes of node i.

The numerical flux $\widetilde{\boldsymbol{F}}$ depends on the state vectors corresponding to the left and right neighboring points of the considered interface, namely:

$$\widetilde{\boldsymbol{F}} = \widetilde{\boldsymbol{F}}\left(\boldsymbol{Q}_i, \boldsymbol{Q}_j\right)$$

as reported in Figure (3.2).
The choice for multi-species reacting flows is a scheme of Advection Upstream Splitting Method (AUSM) family which is described in the next subsection.

AUSM Family

The schemes proposed by Liou and Steffen in [12–14] combine the efficiency of flux vector splitting methods with the accuracy and robustness of Godunov methods. The main idea is to split the projected numerical flux $\boldsymbol{F_f}$ into a convective term $\boldsymbol{F}^{(c)}$ and a pressure term $\boldsymbol{F}^{(p)}$ so that at a continuum level we find:

$$\boldsymbol{F_f} = \boldsymbol{F}^{(c)} + \boldsymbol{F}^{(p)} = \dot{m}\boldsymbol{\Psi} + \boldsymbol{F}^{(p)}$$

Herein $\boldsymbol{\Psi}$ is a vector quantity that represents the problem variables; in our context the mass flux $\dot{m}$, the vector $\boldsymbol{\Psi}$ and the pressure flux $\boldsymbol{F}^{(p)}$ can be therefore expressed as:

$$\dot{m} = \rho \boldsymbol{u} \cdot \boldsymbol{n}, \boldsymbol{\Psi} = \begin{pmatrix} 1 \\ \boldsymbol{u} \\ H \\ Y_i \end{pmatrix}, \boldsymbol{F}^{(p)} = \begin{pmatrix} 0 \\ p\boldsymbol{n} \\ 0 \\ 0 \end{pmatrix} \tag{A.2}$$

As we can see in (A.2) the convective term is constituted by a common scalar mass flux that takes into account the flow direction(the upwinding nature of AUSM) and problem variables which are convected by this mass flux, while the pressure term takes into account the contribution due only to pressure.

The discretization of the component of the numerical flux normal to a given interface depends on the left and right state vectors $\boldsymbol{Q}_L$ and $\boldsymbol{Q}_R$ and it can be defined as:

$$\boldsymbol{F}_{1/2}\left(\boldsymbol{Q}_L, \boldsymbol{Q}_R, \boldsymbol{n}\right) = \dot{m}_{1/2}\boldsymbol{\Psi}_{L/R} + \boldsymbol{p}_{1/2}$$

where $\boldsymbol{\Psi}_{L/R}$ will be determined in a simple upwind fashion,

$$\boldsymbol{\Psi}_{L/R} = \begin{cases} \boldsymbol{\Psi}_L, & \text{if } \dot{m}_{1/2} > 0 \\ \boldsymbol{\Psi}_R, & \text{otherwise} \end{cases}$$

Different choices for $\dot{m}_{1/2} = \dot{m}_{1/2}\left(\boldsymbol{Q}_L, \boldsymbol{Q}_R, \boldsymbol{n}\right)$ and $\boldsymbol{p}_{1/2} = \boldsymbol{p}_{1/2}\left(\boldsymbol{Q}_L, \boldsymbol{Q}_R, \boldsymbol{n}\right)$ determine different schemes: in this work the **AUSM$^+$-up** is chosen. [14]

A.2 BiCGSTAB Algorithm

The biconjugate gradient stabilized gradient method, abbreviated as **BiCGSTAB**, is an iterative scheme for the numerical solution of linear system $\boldsymbol{Ax} = \boldsymbol{b}$ with $\boldsymbol{A}$ non-symmetric developed by H. A. van Der Vorst; it is a variant of the biconjugate gradient method (**BiCG**, see [20]) and therefore it belongs to the Krylov subspace methods.

The idea is to define residual vectors of the form :

$$\tilde{\boldsymbol{r}_i} = \Psi_i(\boldsymbol{A})\Phi_i(\boldsymbol{A})\boldsymbol{r}_0$$

instead of the definition:

$$\boldsymbol{r}_i = \Phi_i(\boldsymbol{A})\boldsymbol{r}_0$$

provided by standard BiCG with the hope that $\Psi_i(\boldsymbol{A})$ will enable faster and smoother convergence. Specifically $\Psi_i(\boldsymbol{A})$ is defined by simply recurrence:

$$\Psi_i(\boldsymbol{A}) = \prod_{j=1}^{i} (\boldsymbol{I} - \omega_j \boldsymbol{A})$$

with ω_j scalars to be determined.
Let us recall the recursive relations that characterize the BiCG algorithm:

$$\begin{aligned}\Phi_i(\boldsymbol{A}) &= \Phi_{i-1}(\boldsymbol{A}) - \alpha_i \boldsymbol{A}\Pi_{i-1}(\boldsymbol{A}) \\ \Pi_i(\boldsymbol{A}) &= \Phi_i(\boldsymbol{A}) + \beta_{i+1}\Pi_{i-1}(\boldsymbol{A})\end{aligned}$$

with α_i and β_i suitable coefficients we immediately obtain:

$$\Psi_i(\boldsymbol{A})\Phi_i(\boldsymbol{A})\boldsymbol{r}_0 = (\boldsymbol{I} - \omega_i \boldsymbol{A}) \left(\Psi_{i-1}(\boldsymbol{A})\Phi_{i-1}(\boldsymbol{A})\boldsymbol{r}_0 - \alpha_i \boldsymbol{A}\Psi_{i-1}(\boldsymbol{A})\Pi_{i-1}(\boldsymbol{A})\boldsymbol{r}_0\right)$$

which entails for a recursive relation for the term $\Psi_i(\boldsymbol{A})\Pi_i(\boldsymbol{A})\boldsymbol{r}_0$; this can be also derived from BiCG:

$$\Psi_i(\boldsymbol{A})\Pi_i(\boldsymbol{A})\boldsymbol{r}_0 = \Psi_i(\boldsymbol{A})\Psi_i(\boldsymbol{A})\boldsymbol{r}_0 + \beta_{i+1} \left(\boldsymbol{I} - \omega_i \boldsymbol{A}\right) \Psi_{i-1}(\boldsymbol{A})\Phi_{i-1}(\boldsymbol{A})\boldsymbol{r}_0$$

Moreover we define:

$$\tilde{\boldsymbol{p}_{i+1}} = \Psi_i(\boldsymbol{A})\Pi_i(\boldsymbol{A})\boldsymbol{r}_0$$

so that in vector form we get:

$$\begin{aligned}\tilde{\boldsymbol{p}_i} &= \tilde{\boldsymbol{r}_{i-1}} + \beta_i \left(\boldsymbol{I} - \omega_{i-1}\boldsymbol{A}\right) \tilde{\boldsymbol{p}_{i-1}} \\ \tilde{\boldsymbol{r}_i} &= \left(\boldsymbol{I} - \omega_i \boldsymbol{A}\right) \left(\tilde{\boldsymbol{r}_{i-1}} - \alpha_i \boldsymbol{A}\tilde{\boldsymbol{p}}\right)\end{aligned}$$

In order to derive a recursive relation for $\boldsymbol{x}_i$ we define:

$$\boldsymbol{s}_i = \tilde{\boldsymbol{r}_{i-1}} - \boldsymbol{A}\tilde{\boldsymbol{p}_i}$$

so that the recurrence relation for $\tilde{\boldsymbol{r}}_i$ becomes:

$$\tilde{\boldsymbol{r}}_i = \tilde{\boldsymbol{r}_{i-1}} - \alpha_i \boldsymbol{A}\tilde{\boldsymbol{p}}_i - \omega_i \boldsymbol{A}\boldsymbol{s}_i$$

and therefore since $\boldsymbol{r}_i = \boldsymbol{A}\boldsymbol{x}_i - \boldsymbol{b}$ we get:

$$\boldsymbol{x}_i = \boldsymbol{x}_{i-1} + \alpha_i \tilde{\boldsymbol{p}}_i + \omega_i \boldsymbol{s}_i$$

Now we need to determine the constants α_i and β_i and choose a suitable ω_i; in standard BiCG we define:

$$\begin{aligned}
\boldsymbol{r}_i &= \boldsymbol{A}\boldsymbol{x}_i - \boldsymbol{b} = \Phi_i(\boldsymbol{A})\boldsymbol{r}_0 \\
\hat{\boldsymbol{r}}_i &= \boldsymbol{A}^T\boldsymbol{x}_i - \boldsymbol{b} = \Phi_i(\boldsymbol{A}^T)\boldsymbol{r}_0 \\
\gamma_i &= (\hat{\boldsymbol{r}}_{i-1}, \boldsymbol{r}_{i-1}) = \left(\Phi_{i-1}(\boldsymbol{A}^T)\hat{\boldsymbol{r}}_0, \Phi_{i-1}(\boldsymbol{A})\boldsymbol{r}_0\right)
\end{aligned}$$

where $(\cdot,\cdot)$ denotes the inner product and from these we impose:

$$\begin{aligned}
\alpha_i &= \frac{\gamma_i}{(\hat{\boldsymbol{p}}_i, \boldsymbol{A}\boldsymbol{p}_i)} \\
\beta_i &= \frac{\gamma_i}{\gamma_{i-1}}
\end{aligned}$$

with

$$\begin{aligned}
\boldsymbol{p}_i &= \boldsymbol{r}_{i-1} + \beta_i \boldsymbol{p}_{i-1} \\
\hat{\boldsymbol{p}}_i &= \hat{\boldsymbol{r}}_{i-1} + \beta_i \hat{\boldsymbol{p}}_{i-1}
\end{aligned}$$

Anyway, since BiCGSTAB does not explicitly keep track of $\hat{\boldsymbol{r}}_i$ or $\boldsymbol{r}_i$, we cannot directly use the previous formula for γ_i. However it can be related to the scalar

$$\tilde{\gamma}_i = \left(\Psi_{i-1}(\boldsymbol{A}^T)\hat{\boldsymbol{r}}_0, \Phi_{i-1}(\boldsymbol{A})\boldsymbol{r}_0\right) = (\hat{\boldsymbol{r}}_0, \Psi_{i-1}(\boldsymbol{A})\Phi_{i-1}(\boldsymbol{A})\boldsymbol{r}_0) = (\hat{\boldsymbol{r}}_0, \boldsymbol{r}_{i-1})$$

Due to biorthogonality $\boldsymbol{r}_{i-1} = \Phi_{i-1}(\boldsymbol{A})\boldsymbol{r}_0$ is orthogonal to $U_{i-2}(\boldsymbol{A}^T)\hat{\boldsymbol{r}}_0$, where $U_{i-2}(\boldsymbol{A}^T)$ is any polynomial of degree $i-2$ ni $\boldsymbol{A}^T$. Hence only the highest order term of $\Phi_{i-1}(\boldsymbol{A}^T)$ and $\Psi_{i-1}(\boldsymbol{A}^T)$ are relevant in the scalar products $\left(\Phi_{i-1}(\boldsymbol{A}^T)\hat{\boldsymbol{r}}_0, \Phi_{i-1}(\boldsymbol{A})\boldsymbol{r}_0\right)$ and $\left(\Psi_{i-1}(\boldsymbol{A}^T)\hat{\boldsymbol{r}}_0, \Phi_{i-1}(\boldsymbol{A})\boldsymbol{r}_0\right)$. The leading coefficients of $\Phi_{i-1}(\boldsymbol{A}^T)$ and $\Psi_{i-1}(\boldsymbol{A}^T)$ are $(-1)^{i-1}\alpha_1\alpha_2...\alpha_{i-1}$ and $(-1)^{i-1}\omega_1\omega_2...\omega_{i-1}$ respectively. It follows that:

$$\gamma_i = \frac{\alpha_1\alpha_2...\alpha_{i-1}}{\omega_1\omega_2...\omega_{i-1}}\tilde{\gamma}_i$$

and

$$\beta_i = \frac{\gamma_i}{\gamma_{i-1}} = \frac{\tilde{\gamma}_i}{\tilde{\gamma}_{i-1}}\frac{\alpha_{i-1}}{\omega_{i-1}}$$

A similar approach can be used to determine α_i; since

$$\alpha_i = \frac{\gamma_i}{(\hat{\boldsymbol{p}}_i, \boldsymbol{A}\boldsymbol{p}_i)} = \frac{\Phi_{i-1}(\boldsymbol{A}^T)\hat{\boldsymbol{r}}_0, \Phi_{i-1}(\boldsymbol{A})\boldsymbol{r}_0}{\Pi_{i-1}(\boldsymbol{A}^T)\hat{\boldsymbol{r}}_0, \boldsymbol{A}\Pi_{i-1}(\boldsymbol{A})\boldsymbol{r}_0}$$

only the highest order terms of $\Phi_{i-1}(\boldsymbol{A}^T)$ and $\Pi_{i-1}(\boldsymbol{A}^T)$ matter in inner products thanks to biorthogonality and biconjugacy and it happens that $\Phi_{i-1}(\boldsymbol{A}^T)$ and $\Psi_{i-1}(\boldsymbol{A}^T)$ have the same leading coefficient. Thus, replacing simultaneously with $\Psi_{i-1}(\boldsymbol{A}^T)$, we are led to:

$$\alpha_i = \frac{\Psi_{i-1}(\boldsymbol{A}^T)\hat{\boldsymbol{r}}_0, \Phi_{i-1}(\boldsymbol{A})\boldsymbol{r}_0}{\Psi_{i-1}(\boldsymbol{A}^T)\hat{\boldsymbol{r}}_0, \boldsymbol{A}\Pi_{i-1}(\boldsymbol{A})\boldsymbol{r}_0} = \frac{\tilde{\gamma}_i}{\hat{\boldsymbol{r}}_0, \boldsymbol{A}\Psi_{i-1}(\boldsymbol{A})\Pi_{i-1}(\boldsymbol{A})\boldsymbol{r}_0} = \frac{\tilde{\gamma}_i}{(\hat{\boldsymbol{r}}_0, \boldsymbol{A}\tilde{\boldsymbol{p}}_i)}$$

Finally, BiCGSTAB selects ω_i to minimize $\tilde{\boldsymbol{r}}_i = (\boldsymbol{I} - \omega_i \boldsymbol{A})\, \boldsymbol{s}_i$ in 2-norm as function of ω_i; this happens when:

$$((\boldsymbol{I} - \omega_i \boldsymbol{A})\, \boldsymbol{s}_i, \boldsymbol{A}\boldsymbol{s}_i) = 0$$

giving the value:

$$\omega_i = \frac{(\boldsymbol{A}\boldsymbol{s}_i, \boldsymbol{s}_i)}{(\boldsymbol{A}\boldsymbol{s}_i, \boldsymbol{A}\boldsymbol{s}_i)}$$

A.3 Secant Method

The secant method is a root-finding algorithm that uses a succession of roots of secant lines to better approximate a root of a function $f(x)$.
Starting with initial values x_0 and x_1, the recurrence relation is defined as:

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}, \quad n = 1, 2, ...$$

If the function f is twice continuously differentiable, the root in question is simple and the initial values x_0 and x_1 are sufficiently close to the root, the iterates x_n converge to a root of f with order of convergence $\phi = \frac{1+\sqrt{5}}{2}$.

It is also a **quasi-Newton** method because it can be derived from the Newton's formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad n = 1, 2....$$

by replacing $f'(x_n)$ with a finite-difference approximation:

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

If we compare Newton's method with the secant method, we see that Newton's method converges faster (order 2 against $\phi \approx 1.6$), but Newton's method requires the evaluation of both f and its derivative while the secant method requires only the evaluation of f.

A.4 Numerical Viscous Jacobians

For implicit time-stepping a Jacobian of viscous fluxes is required. Analytically, the viscous Jacobian depends on second order derivative information within the flow domain, which is difficult and expensive to acquire in general as it requires the use of non-local, "neighbour of neighbour" nodes. This can be avoided by applying the **Thin Shear Layer** (**TSL**) approximation [18], where dual-grid face tangential gradient information are neglected. Under TSL approximation, gradients of a generic scalar quantity ξ are the finite difference gradients between nodes i and j, projected into the cartesian directions,

$$\frac{\partial \xi}{\partial \cdot} = \frac{\xi_j - \xi_i}{d_{ij}} n_\cdot, \qquad \cdot = x, y, z$$

where d_{ij} is the distance between node i and node j.
Under the TSL approximation, only local quantities on opposing sides of the dual-grid interface are required, which makes the approximation well-suited to edge-based, unstructured solvers. Moreover in SU2 an other approximation is introduced for what concerns the Jacobian contribution due to momentum and energy balance equations: the value of the gradient at node j is computed using the mean value at interface and then its opposite is taken for node i, i.e:

$$\frac{\partial \xi}{\partial \cdot}|_j = -\frac{\partial \xi}{\partial \cdot}|_i = \frac{\bar{\xi}}{d_{ij}} n_\cdot, \qquad \cdot = x, y, z$$

where $\bar{\xi}$ is the mean value of a generic quantity ξ at face interface. Let us analyse first the contribution of the following part of the projected viscous flux:

$$\hat{\boldsymbol{G}_{ij}} = \begin{pmatrix} .. \\ \boldsymbol{\tau n} \\ (\boldsymbol{\tau u} - \boldsymbol{q}) \cdot \boldsymbol{n} \\ ... \end{pmatrix}, \tag{A.3}$$

If the transport coefficients are held fixed under differentiation and (A.3) is differentiated with respect to the primitive variables

$$\boldsymbol{P} = \begin{pmatrix} \boldsymbol{u} \\ T \end{pmatrix}$$

the TSL Jacobian takes a simpler form,

$$\frac{\partial \hat{\boldsymbol{G}_{i,j}}}{\partial \boldsymbol{P}_{ij}} = \frac{\partial \hat{\boldsymbol{G}_{i,j}}^{(1)}}{\partial \boldsymbol{P}_{ij}} + \frac{\partial \hat{\boldsymbol{G}_{i,j}}^{(2)}}{\partial \boldsymbol{P}_{ij}}$$

where

$$\frac{\partial \hat{\boldsymbol{G}_{ij}}^{(1)}}{\partial \boldsymbol{P}_j} = \begin{bmatrix} \mu_{ij}\theta_x/d_{ij} & \mu_{ij}\eta_z/d_{ij} & \mu_{ij}\eta_y/d_{ij} & 0 \\ \mu_{ij}\eta_z/d_{ij} & \mu_{ij}\theta_y/d_{ij} & \mu_{ij}\eta_z/d_{ij} & 0 \\ \mu_{ij}\eta_y/d_{ij} & \mu_{ij}\eta_x/d_{ij} & \mu_{ij}\theta_z/d_{ij} & 0 \\ \mu_{ij}\pi_x/d_{ij} & \mu_{ij}\pi_y/d_{ij} & \mu_{ij}\pi_z/d_{ij} & \kappa_{ij} * \theta/d_{ij} \end{bmatrix} = -\frac{\partial \hat{\boldsymbol{G}_{ij}}^{(1)}}{\partial \boldsymbol{P}_i}$$

Herein $\mu_i j$ and κ_{ij} are the laminar viscosity and the thermal conductivity at the interface respectively, while the other quantities are defined as follows:

$$\begin{aligned}
\theta &= n_x^2 + n_y^2 + n_z^2 \\
\theta_x &= \theta + \frac{n_x^2}{3} \\
\theta_y &= \theta + \frac{n_y^2}{3} \\
\theta_z &= \theta + \frac{n_z^2}{3} \\
\eta_x &= \frac{n_y n_z}{3} \\
\eta_y &= \frac{n_z n_z}{3} \\
\eta_z &= \frac{n_x n_y}{3} \\
\pi_x &= \bar{u_x}\theta_x + \bar{u_y}\eta_z + \bar{u_z}\eta_y \\
\pi_y &= \bar{u_x}\eta_z + \bar{u_y}\theta_y + \bar{u_z}\eta_x \\
\pi_z &= \bar{u_x}\eta_y + \bar{u_y}\eta_x + \bar{u_z}\theta_z
\end{aligned}$$

where $\bar{u_\cdot}$, with $\cdot = x, y, z$ is the value of velocity at face interface.

Eventually

$$\frac{\partial \hat{\boldsymbol{G}_{\boldsymbol{ij}}}^{(2)}}{\partial \boldsymbol{P}_j} = \frac{\partial \hat{\boldsymbol{G}_{\boldsymbol{ij}}}^{(2)}}{\partial \boldsymbol{P}_i} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0.5(\boldsymbol{\tau}_{ij} n)_x & 0.5(\boldsymbol{\tau}_{ij} n)_y & 0.5(\boldsymbol{\tau}_{ij} n)_z & \sum_{k=1}^{N_S} 0.5 \boldsymbol{J}_{k_{ij}} \cdot \boldsymbol{n} C_{p_k}(\bar{T}) \end{bmatrix}$$

where $\bar{T}$ is the interface temperature.
The whole computation is very long to report; anyway in order to show the general procedure let us show the computation of $\frac{\partial (\boldsymbol{\tau n})_x}{\partial u_x}|_j$; first of all we recall that

$$(\boldsymbol{\tau n})_x = \mu_{ij} \left(2 \frac{\partial \bar{u_x}}{\partial x} - \frac{2}{3} \nabla \cdot \bar{\boldsymbol{u}} \right) n_x + \mu_{ij} \left(\frac{\partial \bar{u_x}}{\partial y} + \frac{\partial \bar{u_y}}{\partial x} \right) n_y$$

Therefore

$$\frac{\partial (\boldsymbol{\tau n})_x}{\partial u_x} | j = \mu_{ij} \left(2 \frac{\partial}{\partial u_x} \left(\frac{\partial \bar{u_x}}{\partial x} \right) - \frac{2}{3} \frac{\partial}{\partial u_x} \left(\nabla \cdot \bar{\boldsymbol{u}} \right) \right) n_x + \mu_{ij} \left(\frac{\partial}{\partial u_x} \left(\frac{\partial \bar{u_x}}{\partial y} \right) + \frac{\partial}{\partial u_x} \left(\frac{\partial \bar{u_y}}{\partial x} \right) \right) n_y$$

Changing the derivatives we get:

$$\frac{\partial (\boldsymbol{\tau n})_x}{\partial u_x} | j = \mu_{ij} \left(2 n_x - \frac{2}{3} n_x \right) \frac{n_x}{d_{ij}} + \mu_{ij} \frac{n_y^2}{d_{ij}} = \mu_{ij} \frac{\left(\frac{4}{3} n_x^2 - n_y^2 \right)}{d_i j} = \mu_{ij} \frac{\theta_x}{d_{ij}}$$

Since we need the derivatives with respect to the conserved variables, the following transformation matrix is also applied:

$$\frac{\partial \boldsymbol{P}}{\partial \boldsymbol{Q}} = \begin{bmatrix} -\frac{u_x}{\rho} & \frac{1}{u_x} & 0 & 0 & 0 & 0 & 0 & .... & 0 \\ -\frac{u_y}{\rho} & 0 & \frac{1}{u_y} & 0 & 0 & 0 & 0 & .... & 0 \\ -\frac{u_z}{\rho} & 0 & 0 & \frac{1}{u_z} & 0 & 0 & 0 & .... & 0 \\ \frac{\partial T}{\partial \rho} & \frac{\partial T}{\partial \rho u_x} & \frac{\partial T}{\partial \rho u_y} & \frac{\partial T}{\partial \rho u_x} & \frac{\partial T}{\partial \rho E} & \frac{\partial T}{\partial \rho_1} & \frac{\partial T}{\partial \rho_2} & .... & \frac{\partial T}{\partial \rho_{N_S}} \end{bmatrix}$$

More attention is needed instead for the contribution due to species diffusion fluxes; indeed since the fluxes $\boldsymbol{J}_i, i = 1...N_S$ are defined implicitly through (**??**), it is not feasible to derive an analytic expression even with the use of TSL. Another approach could be the computation through a numerical Jacobian but even this case is too expensive from a computational point of view because it requires to solve one more set of Stefan-Maxwell equations for each interface and, above all, it necessitates the computation of gradient of molar fractions after perturbation which involves the costly algorithms of Green-Gauss or least squares.
Therefore the alternative exploited in order to compute an approximate Jacobian is to apply the TSL to Ramshaw self-consistent relations (**??**) keeping fixed the mean effective diffusion coefficients and the molar mass computed with the value of molar fractions at interface.
After some attempts, it has been noted that only the contribution due to single species guarantees the convergence of implicit Euler method; therefore let us report first the derivative of a generic molar fraction X_i with respect to a partial density ρ_k

$$\begin{aligned} \frac{\partial X_i}{\partial \rho_k} =& \frac{\partial \left(\sigma \frac{M}{M_i} Y_i \right)}{\partial \rho_k} = \frac{M}{M_i} \left(\frac{\partial \sigma}{\partial \rho_k} Y_i + \frac{\partial Y_i}{\partial \rho_k} \sigma \right) = \frac{M}{M_i} \left(\frac{Y_i}{\rho} + \delta_{ik} \frac{\sigma}{\rho} \right) \\ &= \frac{M}{M_i} \left(\frac{Y_i}{\rho} \frac{\sigma}{\sigma} + \delta_{ik} \frac{\sigma}{\rho} \right) = \frac{X_i}{\sigma \rho} + \frac{M}{M_i} \frac{\delta_{ik}}{\rho} \end{aligned}$$

If we denote by R and L the right and left state respectively we get:

$$\frac{\partial \boldsymbol{J}_i \cdot \boldsymbol{n}}{\partial \rho_k}|_R = -\bar{\rho} \frac{M_i}{\bar{M}} \frac{\bar{D_{i,m}}}{d_{ij}} \left(\frac{X_{i,R}}{\sigma_R \rho_R} + \frac{M_R}{M_i} \frac{\sigma_R}{\rho_R} \delta_{ik} \right) + \bar{Y_i} \frac{\bar{\rho}}{\bar{M} d_{ij}} \sum_{j=1}^{N_S} \left(M_j \bar{D_{j,m}} \frac{X_{j,R}}{\sigma_R \rho_R} \right) + \bar{Y_i} \frac{\bar{\rho}}{\bar{M} d_{ij}} \frac{M_R \sigma_R}{\rho_R}$$

$$\frac{\partial \boldsymbol{J}_i \cdot \boldsymbol{n}}{\partial \rho_k}|_L = \bar{\rho} \frac{M_i}{\bar{M}} \frac{\bar{D_{i,m}}}{d_{ij}} \left(\frac{X_{i,L}}{\sigma_L \rho_L} + \frac{M_L}{M_i} \frac{\sigma_L}{\rho_L} \delta_{ik} \right) - \bar{Y_i} \frac{\bar{\rho}}{\bar{M} d_{ij}} \sum_{j=1}^{N_S} \left(M_j \bar{D_{j,m}} \frac{X_{j,L}}{\sigma_L \rho_L} \right) - \bar{Y_i} \frac{\bar{\rho}}{\bar{M} d_{ij}} \frac{M_L \sigma_L}{\rho_L}$$

where the symbol $-$ denotes a value taken at interface between state L and R.

It is worth to notice that for the diffusion fluxes we do not use the classical SU2 approximation of taking the mean value of interface for right state and then take the opposite for the left one, but we distinguish between the value at node L and the value at node R.

Bibliography

[1] Numerical Modeling and Simulations of Combustion Processes in Hybrid Rocket Engines, Mazzetti, Alessandro

[2] SU2 Detailed documentation: `https://su2code.github.io/docs/home/`

[3] R. A. Svehla: Transport Coefficients for the NASA Lewis Chemical Equilibrium Program, NASA Technical Memorandum 4647, NASA-TM-4647, National Aeronautics and Space Administration Lewis Research Center, Cleveland, Ohio; USA; 1995; available online at `https://www.grc.nasa.gov/www/CEAWeb/TM-4647.pdf`

[4] Burcat, A., Ruscic, B., Third Millennium Ideal Gas and Condensed Phase Thermochemical Database for Combustion with Updates from Active Thermochemical Tables, Argonne Report ANL 05/20 and Technion Aerospace Report TAE N. 960, September 2005.

[5] Multi-Component Diffusion with Application To Computational Aerothermodynamics, Sutton, Kenneth and Gnoffo, Peter, 02/1998

[6] Venkateswaran, S., Merkle, C.L., Size Scale-Up in Hybrid Rocket Motors, AIAA Aerospace Sciences Meeting and Exhibit, 34th, Reno, NV; USA; 15-18 January, 1996, AIAA Paper 1996-0647.

[7] Kuo, K.K., Principles of Combustion, 2nd Ed., Chapter 3. John Wiley & Sons, New York, NY; 2005.

[8] Fuller, E.N., Schettler, P.D., Giddings, J.C., A new method for prediction of binary gas-phase diffusion coefficients, Industrial and Engineering Chemistry, 58(5):19-27, 1966.

[9] Quarteroni, A., Numerical Models for Differential Problems,2014

[10] Praaven, C., Finite volume method on unstructured grid, 2013, `http://math.tifrbng.res.in/~praveen/notes/acfd2013/fvm.pdf`

[11] Veynante, D., Poinsot, T., Theoretical and Numerical Combustion, R.T. Edwards, P.O. Box 27388, Philadelphia, PA, 19118; USA; 2001.

[12] Liou, M.-S. and Steffen, C., A New Flux Splitting Scheme, Journal of Computational Physics, 107: 23-39, 1993.

[13] Liou, M.-S., A Sequel to AUSM: AUSM+, Journal of Computational Physics, 129: 364-382, 1996.

[14] Liou, M.-S., A Sequel to AUSM, Part II: AUSM+-up, Journal of Computational Physics, 214: 137- 170, 2006.

[15] Wilke, C.R., A Viscosity Equation for Gas Mixtures, The Journal of Chemical Physics, 18(4): 517 - 519, 1950.

[16] Mason, E.A., Saxena, S.C., Approximate Formula for the Thermal Conductivity of Gas Mixtures, The Physics of Fluids, 1(5): 361 - 369, 1958.

[17] Giovangigli, V., Convergent iterative methods for multicomponent diffusion, IMPACT Comput. Sci. Eng., 1991, Vol. 3, 244-276

[18] Blazek, J., Computational Fluid Dynamics: Principles and Applications, Elsevier Science, 2001

[19] Wilcox, D.C., Turbulence Modeling for CFD, 3rd Ed. Chapter 5. DCW Industries, 5354 Palm Drive, La Canada, California, 91011; USA; 2006.

[20] Saad, Y., Iterative methods for sparse linear systems, Second Edition, Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 2003

[21] Ames Research Staff, Equations, tables and charts for compressible flows, Ames Aeronautical Laboratory, Moffett Field, California, Report 1135, 1953

[22] EIGEN Documentation: `http://eigen.tuxfamily.org/dox/`