Numerical implementation of turbulent solver for reacting flows within SU2 suite

POLITECNICO
MILANO 1863

Msc of Computational Science and Engineering

Advanced Programming for Scientific Computing (8 CFU)

Lorenzo Vallisa

08 08 2020

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
X_i = Mole fraction of species i
T = Temperature
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
E = Total energy per unit
H = Mixture total enthalpy
h = Mixture static enthalpy
h_i = Static enthalpy of species
C_p = Mixiture specific heat at constant pressure
P_p = Mixture specific heat at constant volume
$\hat{f}_{pi}$ = Specific heat at constant pressure of species i
μ = Mixture laminar viscosity
k = Mixture thermal conductivity

Contents

Chapter 1

Introduction

The project comprises both a challenging modelling of turbulence phenomenon happening inside a jet reactor, in which chemical reactions are happening due to the combustion process, as well as a remarkable ability to understand and implement complex C++ structures within one of the biggest and most elaborated software written for Computational Fluid Dynamics.

Under the physics point of view, the interaction between fuel, oxidizer and combustion products have been modeled using a multispecies approach: one mass conservation equation for each species, two equations for momentum (2D simulation) and one for energy conservation. Turbulent combustion has been modeled using a PaSR approach: indeed the source term of mass conservation equations accounts for the time needed by species to diffuse before reacting, species reacting in a shorter time than the one needed by turbulence to transport them will happen only on a small part of the computational cells, allowing thus the fuel not to fully react at once, but to be able to further expand in the domain. The most challenging part though was the one connected to the implementation of the code and the consequent stability workaround of the numerical scheme. After a *first phase* of understanding how the full system of drivers was interacting with *Solver* and *Numerics* classes, it was clear that a simple addition of the the turbulent closure to the already existing code was definitely not enough. The implementation of an entire new solver was compulsory if a communication between the reactive laminar solver and the turbulent one was to be brought about (in SU2, as well

as in all other CFD software, RANS and mean flow field sets of equations are solved separately and sequentially). The main goal of *phase two* was therefore to identify all the patterns involved in a SU2 solver processing and find a most efficient way to introduce a new solver without compromising the full structure of the code. Once successfully implemented a reactive-rans solver, the following obstacle to overcome was to retrieve the turbulent methods that a single fluid Navier-Stokes solver class had included thanks to the polymorphic structure, but that was unfortunately hidden by the previous laminar implementation of a multispecies reactive Navier-Stokes solver. *Third phase* was entirely devoted to numerical issues: indeed, some of the approximations introduced for building up the turbulent closure for fluxes within the viscous part, revealed themselves to be very inaccurate, causing the simulation to diverge. Moreover, during combustion, introduced PaSR constant (portion of the cell devoted to combustion) to account for turbulent combustion was assuming too low values for a combustion process to happen and be sustained. Whereas the first issue was resolved by introducing a structure in the code able to loosen the impact of the introduced approximation, the second one was curbed by imposing a lower band on the PaSR constant.

Eventually successfully results were obtained, showing, in comparison with laminar case, a much higher boundary layer and stronger diffusive properties of each species which allow the combustion to take place at lower temperature, in much more time but in a greater region of space of the engine, as expected by experimental results.

Chapter 2

Mathematical Model

2.1 Conservative Governing Equations

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

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

where

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

This set of equations is known as fluid derivation of the first three moments of the kinetic equation, within the assumption of a perturbation of the Maxwellian distribution used to obtained fluid variables from the kinetic ones. Indeed we have continuity, momentum and energy balance equations respectively, and the rest are Ns species continuity equations. The mathematical-physical model chosen to reproduce behavior of reacting species is hence

the *multispecies* one, in which each species contributes to momentum of the system mainly through pressure

$$p = \sum_{i=1}^{N_s} \rho_i R_i T = \sum_{i=1}^{N_s} \rho Y_i R_i T = \rho R T \tag{2.2}$$

where

$$Y_i = \rho_i / \rho \tag{2.3}$$

is the mass fraction of species i in the gas mixture, whereas the contribution to the energy of the system comes from internal energy

$$E = \sum_{i=1}^{N_s} Y_i e_i + \frac{\|\mathbf{u}\|}{2} \tag{2.4}$$

but most of all from the fluxes of every single species J_i, which intrinsically contributes to the heat flux according to the following

$$\mathbf{q} = -k\nabla T + \sum_{i=1}^{N_S} h_i \mathbf{J_i} \tag{2.5}$$

In cases in which the dilute approximation can not be acceptable a full multicomponent treatment is required, indeed Stefan-Maxwell equations need to be solved in order to recover the correct expression for the flux $\mathbf{J_i}$,i.e.

$$\mathbf{d_i} = \sum_{i=1}^{N_S} \frac{X_i X_j}{D_{ij}} (\mathbf{V_j} - \mathbf{V_i}) \tag{2.6}$$

where $\mathbf{V_i}$ represents species i velocity,D_{ij} is the mass diffusion coefficient for i and j species and $\mathbf{d_i}$ can be expressed as

$$d_i = \nabla X_i + (X_i - Y_i)\,\nabla \ln p - \frac{Y_i}{p}\left(\rho f_i - \rho \sum_{k=1}^{N_S} Y_k f_k\right) \tag{2.7}$$

where X_i is the molar fraction of species i, f are body forces terms and the dependence of $\mathbf{d_i}$ from $\mathbf{J_i}$ is implicit and thoroughly treated in [4].

2.2 Average Procedure

One of the approach to introduce turbulence when dealing with conservation laws is through an average procedure. More specifically *Favre-averaging* will be introduced in order to avoid explicit modeling of density fluctuation correlations, such as $\overline{\rho' u'}$, which appear if the extended Reynolds averaging is used. Every generic quantity is split into a mass-weighted mean value and a fluctuating value:

$$Q = \widetilde{Q} + Q'' \tag{2.8}$$

With the following properties:

$$\widetilde{Q} = \frac{\overline{\rho Q}}{\bar{\rho}} \tag{2.9}$$

$$\widetilde{Q''} = \frac{\overline{\rho(Q - \widetilde{Q})}}{\bar{\rho}} = 0 \tag{2.10}$$

By applying substitution (2.8) and (2.9) to the conservative variables within the model, the set of equations (2.1) will look as follows:

$$\frac{\partial \bar{\rho}}{\partial t} + \frac{\partial \bar{\rho}\tilde{u}_j}{\partial x_j} = 0 \tag{2.11}$$

$$\frac{\partial \bar{\rho}\tilde{u}_i}{\partial t} + \frac{\partial \bar{\rho}\tilde{u}_j\tilde{u}_i}{\partial x_j} = -\frac{\partial \bar{\rho}\widetilde{u_i'' u_j''}}{\partial x_j} - \frac{\partial \bar{p}}{\partial x_i} + \frac{\partial \bar{\tau}_{ij}}{\partial x_j} + \bar{f}_i \tag{2.12}$$

$$\frac{\partial \bar{\rho}\tilde{h}_t}{\partial t} + \frac{\partial \bar{\rho}\tilde{u}_j\tilde{h}_t}{\partial x_j} = -\frac{\partial \bar{\rho}\widetilde{u_j'' h_t''}}{\partial x_j} + \frac{\partial \bar{p}}{\partial t} + \frac{\partial}{\partial x_j}\left(\overline{u_i \tau_{ij}} - \bar{q}_j\right) + \overline{\dot{Q}}_{ext} \tag{2.13}$$

$$\frac{\partial \bar{\rho}\tilde{Y}_k}{\partial t} + \frac{\partial \bar{\rho}\widetilde{u_j}\tilde{Y}_k}{\partial x_j} = -\frac{\partial \bar{\rho}\widetilde{u_j'' Y_k''}}{\partial x_j} + \overline{\mathbf{J}_k} + \overline{\dot{\omega}}_k \tag{2.14}$$

Where $\bar{\tau}_{ij}$ is the laminar viscosity-dependant part of Reynolds stress tensor:

$$\bar{\tau}_{ij} = \mu\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i} - \frac{2}{3}\delta_{ij}\frac{\partial \bar{u}_k}{\partial x_k}\right) \tag{2.15}$$

and h_t is the total specific enthalpy whose contributions are respectively the static enthalpy, the sum of each species contribution to formation enthalpy and kinetic energy, i.e.

$$h_t = h_s + \sum_{i=1}^{N_S} Y_i \triangle h_0 + \frac{1}{2}u_j u_j \tag{2.16}$$

The *Favre-averaged* terms right after the equal sign are those responsible for the transport of turbulent fluctuations of the four conserved variables, indeed mass, momentum along x and y axis, and eventually energy, represented in equation 2.24.In a turbulent reacting flow, where multiple mass fractions and turbulent kinetic energy contribution are present, it is important to check all the components hidden in the turbulent energy term. The work of [3] shows in fact that the enthalpic term can be expressed as the combination of five terms:

$$\widetilde{\bar{\rho}u_i''h_t''} = \sum_k \left[\widetilde{\bar{\rho}u_i''Y_k''} \left(\widetilde{h}_{s,k} + \Delta h_{0,k} \right) \right] + \bar{\rho} \sum_k \widetilde{u_i''Y_k''h_{s,k}''} + \bar{\rho} \sum_k \widetilde{u_i''Y_k''h_{s,k}''} + \bar{\rho} \sum_i \widetilde{u}_j \widetilde{u_i''u_j''} + \bar{\rho} \sum_i u_i'' \frac{1}{2} \widetilde{u_j''u_j''} \quad (2.17)$$

the first term represents the transport of mean static enthalpy with turbulent flux of mass, the second and third term can be added to each other and represent turbulent flux of specific enthalpy, whereas the fourth term represents the work performed by the Reynolds stress tensor, as defined in next section and the last term is related to the transport of turbulence kinetic energy by velocity fluctuations. The system of equations just introduced is to be closed with models for turbulent fluxes of mass fractions, turbulent transport of energy, Reynolds stress tensor and chemical reaction source term. Turbulent transport of mass term is closed with a gradient hypothesis, defining turbulent Prandtl and Lewis [6] numbers ($\mathrm{Pr}_T \approx 0.7 - 0.9, Le_T \approx 1.0 - 1.4$), as shown here below:

$$\widetilde{\rho u_j''Y_k''} = -\frac{\mu_T}{Pr_T Le_T} \frac{\partial \tilde{Y}_k}{\partial x_j} \quad (2.18)$$

As far as the momentum part is concerned, the Reynolds stress tensor can be expressed using standard mean quantities for velocity components, under Boussinesq hypothesis, as:

$$\tau_{ij}^R = -\bar{\rho}\overline{u_i''u_j''} = \mu_T \left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i} - \frac{2}{3} \frac{\partial \bar{u}_k}{\partial x_k} \delta_{ij} \right) - \frac{2}{3} \bar{\rho} k \delta_{ij} \quad (2.19)$$

Having this in mind, the closure for the energy equation term can be actually brought about simply adding a Wilcox closure for the transport of fluctuactions connected to kinetic energy [7], and handling the terms with enthalpic variables with a simple gradient closure

$$\widetilde{\rho u_i'' h_t''} = -\left[\sum_{k=1}^{N_S} \frac{\mu_T}{\mathrm{Pr}_T\, Le_T} \frac{\partial \tilde{Y}_k}{\partial x_j}\left(\tilde{h}_s^k\right) + \sum_{k=1}^{N_S} \frac{\mu_T}{\mathrm{Pr}_T} \frac{\partial \tilde{h}_s^k}{\partial x_j} + \left(\mu + \frac{\mu_T}{\sigma_k}\right)\frac{\partial k}{\partial x_j} + \tau_{ij}^R \tilde{u}_j\right] \tag{2.20}$$

2.3 Turbulent Viscosity Model

In order to close the system for the conservation equations, it is necessary to introduce a turbulence model for the determination of the turbulent viscosity coefficient μ_T. The choice for present work is the $k-\omega$ odel, which gives better results for reacting flows in internal geometries at low Mach numbers with respect to, for instance, the $k-\epsilon$ model. Therefore turbulent viscosity is defined as:

$$\mu_T = \rho \frac{k}{\omega} \tag{2.21}$$

where k is the turbulence kinetic energy and ω is the turbulence specific dissipation rate (the characteristic frequency of turbulence). Transport equations are to be resolved for turbulence kinetic energy and specific dissipation rate:

$$\frac{\partial}{\partial t}(\bar{\rho}k) + \frac{\partial}{\partial x_j}\left(\bar{\rho}\tilde{u}_j k\right) = P - \beta^\star \bar{\rho}\omega k + \frac{\partial}{\partial x_j}\left[\left(\mu + \sigma_k \frac{\bar{\rho}k}{\omega}\right)\frac{\partial k}{\partial x_j}\right] \tag{2.22}$$

and:

$$\frac{\partial}{\partial t}(\bar{\rho}\omega) + \frac{\partial}{\partial x_j}\left(\bar{\rho}\tilde{u}_j \omega\right) = \frac{\gamma\omega}{k}P - \beta\bar{\rho}\omega^2 + \frac{\partial}{\partial x_j}\left[\left(\mu + \sigma_\omega \frac{\bar{\rho}k}{\omega}\right)\frac{\partial \omega}{\partial x_j}\right] \tag{2.23}$$

Where P is defined as:

$$P = \tau_{ij}\frac{\partial \bar{u}_i}{\partial x_j} = \left\{\mu_T\left[\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right) - \frac{2}{3}\frac{\partial \bar{u}_k}{\partial x_k}\delta_{ij}\right] - \frac{2}{3}\bar{\rho}k\delta_{ij}\right\}\frac{\partial \bar{u}_i}{\partial x_j}$$

Closure coefficients for the $k-\omega$ two-equations turbulence model from are liste lows:

$$\gamma = 13/25, \quad \beta = \beta_0 f_\beta, \quad \beta^* = 9/100, \quad \sigma = 1/2, \quad \sigma^* = 3/5, \quad \sigma_{ab} = 1/8$$

$$\beta_0 = 0.0708, \quad Pr_t = 0.9$$

$$\sigma_d = \begin{cases} 0, & \text{if } \frac{\partial k}{\partial x_j}\frac{\partial \omega}{\partial x_j} \leq 0 \\ \sigma_{d0} = 1/8, & \text{if } \frac{\partial k}{\partial x_j}\frac{\partial \omega}{\partial x_j} > 0 \end{cases}$$

$$f_\beta = \frac{1+85\chi_\omega}{1+100\chi_\omega}, \quad \chi_\omega = \left| \frac{\Omega_{ij}\Omega_{jk}\hat{S}_{ki}}{(\beta^*\omega)^3} \right|, \quad \hat{S}_{ki} = S_{ki} - \frac{1}{2}\frac{\partial \tilde{u}_m}{\partial x_m}\delta_{ki}$$

$$\Omega_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} - \frac{\partial u_j}{\partial x_i}\right), \quad S_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)$$

2.4 PaSR Algorithm for Chemistry Closure

The Partially Stirred Reactor approach was first proposed by Golovitchev [2], which accounts for reactions in a fully turbulent conditions. Indeed a simpler approach would have bben to consider a straight Perfectly Stirred Reactor (PSR) way of modeling chemical reactions, the formula for each k species' source is as follows:

$$\overline{\dot{\omega}}_k = M_k \sum_{r=1}^{NR} \left(\nu''_{k,r} - \nu'_{k,r}\right) \left\{ k_{f,r} \prod_{i=1}^{NS} \left[\frac{\bar{\rho}\tilde{Y}_i}{M_i}\right]^{\nu'_i} - k_{b,r} \prod_{i=1}^{NS} \left[\frac{\bar{\rho}\tilde{Y}_i}{M_i}\right]^{\nu''_i} \right\} \tag{2.24}$$

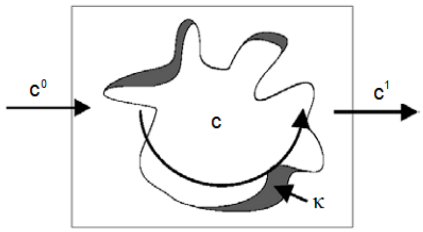

Figure 2.1: PaSR conceptual scheme

Equation 2.24 expresses the net source term for each chemical species as the sum over the NR reactions (that the considered species participate in) of the rate of production and destruction using the Law of Mass Action. This law has dependence on the stoichiometric coefficients ν (and their difference), on forward and backward reaction rates $k_{f,b}$, and on species concentration $c_k = \frac{\overline{\rho}\tilde{Y}_i}{M_i}$.PaSR model overcomes the simplified approach given by a pseudo-laminar chemistry by considering that only a partial volume of each computational cell is affected by the presence of a chemically reacting zone. In PaSR approach, combustion is considered as the combination of two different processes. The first one considers the change of concentration from c_0 (unburnt gases) to c, burnt gases. The second one considers the turbulent mixing of burnt gases with oncoming fresh reactants, therefore with the change of concentration from c to c_1. Reacting volume fraction of each computational cell is proportional to the ratio k of chemical reaction time τ_c and total time $\tau_c + \tau_{mix}$, as shown here below:

$$\overline{\dot{\omega}}_k = \sum_{r=1}^{NR} \frac{\tau_{c,r}}{\tau_{c,r} + \tau_{mix}} \overline{\dot{\omega}}_{k,r} \tag{2.25}$$

where term $\sum_{r=1}^{NR} \overline{\dot{\omega}}_{k,r}$ equals the rhs of equation 2.24 and the k term for the r reaction is $\frac{\tau_{c,r}}{\tau_{c,r}+\tau_{mix}}$. For each chemical reaction one time is chosen from the chemical system Jacobian matrix, therefore representing the sensitivity of a given reaction to variations in concentration for a given chemical species. For reactions depending on several concentrations, the one corresponding to the smallest time is chosen and calculated in dependence of the sensitivity of the r-th reaction rate to the variations of k-th chemical species partial density

(or concentration, being these two quantities strictly linked):

$$\frac{1}{\tau_{c,r}} = \max_{k \in NS} \left| \frac{\partial f_r}{\partial \rho_k} \right| \tag{2.26}$$

where

$$f_r = \left\{ k_{f,r} \prod_{i=1}^{NS} \left[\frac{\bar{\rho} \tilde{Y}_i}{M_i} \right]^{\nu_i'} - k_{b,r} \prod_{i=1}^{NS} \left[\frac{\bar{\rho} \tilde{Y}_i}{M_i} \right]^{\nu_i''} \right\} \tag{2.27}$$

This choice is due to the consideration that the influence of turbulence is greater if the chemical time is small enough. On the other hand if the chemistry time is big enough, the time ratio k could not be influenced by turbulence because τ_{mix} becomes negligible with respect to $\tau_{c,r}$ and k ratio becomes close to unity. Therefore the smallest chemical time is more influenced by turbulence while the largest chemical time is the least influenced by turbulence. For the considered reaction scheme this results in a characteristic chemical time for each one of the six reactions involved. Of course this is one of the possible approaches, which the author considers more accurate with respect to the single-time PaSR.The turbulent mixing time is chosen accordingly:

$$\tau_{mix} = \frac{1}{C_\mu \omega} \tag{2.28}$$

where C_μ is turbulence constant set to 0.09.

Chapter 3

Numerical Model

3.1 Space Integration

During the last decades, the Finite Volume method 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:

$$C_i, \quad i = 1, \ldots, N_C$$

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

$$\dot{C}_i \cap \hat{C}_j = \emptyset, \quad i, j = 1 \ldots N_C, i \neq j$$

Once a mesh has been formed, we have to create the finite volumes Ω_i on

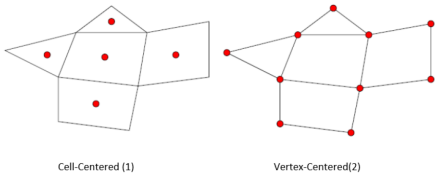

Figure 3.1: Finite Volume Strategies

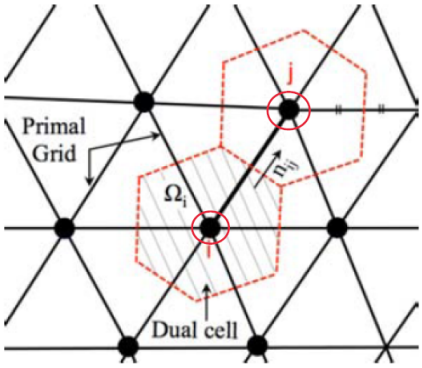

Figure 3.2: Dual Grid

which the conservation laws will be applied. This can be done in two ways depending on where the solution is stored: 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; if instead 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. 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 in 3.2:

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

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

that, through divergence theorem becomes:

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

where Σ_i is the boundary of the finite volume Ω_i and n is the outward unit normal with respect to Σ_i. At this point we rewrite the equation 3.2 introducing the residual so that

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

with

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

in which every term is discretized separately leading to an upwind treatment of convective fluxes, 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 semi-discretazion of a system of PDEs; at this point a suitable discretization in time is needed in order to obtain a numerical solution in space and in time. Through the following semplification it is possible to transform 3.3 into a system of ODEs

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

Two of the most commonly used strategies are listed here below:

$$\frac{Q_i^{n+1} - Q_i^n}{\Delta t_i^n} + R_i\left(Q^n\right) = 0 \quad \text{Explicit Euler} \tag{3.6}$$

$$\frac{Q_i^{n+1} - Q_i^n}{\Delta t_i^n} + R_i\left(Q^{n+1}\right) = 0 \quad \text{Implicit Euler} \tag{3.7}$$

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

$$\Delta t_i = N_{CFL} \min\left(\frac{|\Omega_i|}{\lambda_i^{conv}}, \frac{|\Omega_i|}{\lambda_i^{visc}}\right) \tag{3.8}$$

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

$$\lambda_i^{conv} = \sum_{f=1}^{N_f} \left(\left|u_{1/2}\right| + c_{1/2}\right) \Sigma_f \tag{3.9}$$

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

$$\lambda_i^{visc} = \sum_{f=1}^{N_f} \frac{C\mu_{1/2} + f\left(\mu_{1/2}^L\right)}{\rho_{1/2}} \Sigma_f^2 \tag{3.10}$$

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

$$f\left(\mu_{1/2}^{L}\right) = \gamma_{1/2}\frac{\mu_{1/2}^{L}}{Pr_{1/2}^{L}} \tag{3.11}$$

where $\mathrm{Pr}_{1/2}^{L}$ is the laminar Prandtl number defined as:

$$\mathrm{Pr}_{1/2}^{L} = \frac{\mu_{1/2}^{L} C_{p_{1/2}}}{\kappa_{1/2}^{L}} \tag{3.12}$$

with $C_{p_{1/2}} = \frac{C_{p_L}+C_{p_R}}{2}$ as interface specific heat at constant pressure and $\kappa_{1/2}^{L} = \frac{\kappa_L^L+\kappa_R^L}{2}$ 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 $\Delta Q_i^n = Q_i^{n+1} = Q_i^n$ is immediately found as:

$$\Delta Q_i^n = -R_i\left(Q^n\right)\Delta t_i^n \tag{3.13}$$

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

$$\boldsymbol{R}_i\left(\boldsymbol{Q}^{n+1}\right) = \boldsymbol{R}_i\left(\boldsymbol{Q}^n\right) + \frac{\partial \boldsymbol{R}_i\left(\boldsymbol{Q}^n\right)}{\partial t}\Delta t^n + \mathcal{O}\left(\Delta t^2\right) = \boldsymbol{R}_i\left(\boldsymbol{Q}^n\right) + \sum_{j=1}^{N_f}\frac{\partial \boldsymbol{R}_i\left(\boldsymbol{Q}^n\right)}{\partial \boldsymbol{Q}_j^n}\Delta \boldsymbol{Q}_j^n + \mathcal{O}\left(\Delta t^2\right) \tag{3.14}$$

Finally the following linear system should be solved to find the solution update ΔQ_i^n

$$\left(\frac{|\Omega_i|}{\Delta t_i^n}\delta_{ij} + \frac{\partial \boldsymbol{R}_i\left(\boldsymbol{Q}^n\right)}{\partial \boldsymbol{Q}_j^n}\right)\Delta \boldsymbol{Q}_j^n = -\boldsymbol{R}_i\left(\boldsymbol{Q}^n\right) \tag{3.15}$$

The term $\boldsymbol{R}_i\left(\boldsymbol{Q}^n\right)\partial \boldsymbol{Q}_j^n$ is constituted by the contribution of convective numerical flux Jacobian,viscous numerical flux Jacobian and source term Jacobian and in case the total flux F_{ij} has a stencil of points $\{i, j\}$, then contri-

butions are made to the Jacobian at four points:

$$\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{Q}} = \begin{bmatrix} \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \frac{\partial \tilde{F}_{ij}}{\partial Q_i} & \cdots & \frac{\partial \tilde{F}_{ij}}{\partial Q_j} & \cdots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \cdots & \cdots & -\frac{\partial \tilde{F}_{ij}}{\partial Q_i} & \cdots & -\frac{\partial \tilde{F}_{ij}}{\partial Q_j} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} \tag{3.16}$$

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

$$\frac{\partial Q}{\partial \tau} + R^*(Q) = 0 \tag{3.17}$$

with

$$\boldsymbol{R}^*(\boldsymbol{Q}) = \frac{3}{2\Delta t}\boldsymbol{Q} + \frac{1}{|\Omega|^{n+1}}(\boldsymbol{R}(\boldsymbol{Q})) - \frac{2}{\Delta t}\boldsymbol{Q}^n\,|\Omega^n| + \frac{1}{2\Delta t}\boldsymbol{Q}^{n-1}|\Omega|^{n-1} \tag{3.18}$$

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 $Q = Q^{n+1}$ once the steady problem is satisfied.

Chapter 4

SU2

4.1 SU2 in a nutshell: a quick guide for new solvers implementation

The main goal of this chapter is to outline the main structure behind the SU2 suite, describe how the different parts of the code communicate with each other and by doing that to give as well a quick insight on how to introduce a new solver, in order to help future developer when they first approach the code. In the following chapters the attention will be hence driven towards the description of how everything starts from a driver class and where the changings to include a new turbulent solver for reacting species have to be brought about.

4.1.1 Turbulent Reactive Solver

The very first thing to do is to make the code aware of the existence of a new solver, in practical terms when parsing a .cfg file the software will read as input the parameters that identify the kind of desired simulation, indeed this project is aimed at simulating a case with turbulent and reacting flow. The **config** class will be responsible to create the *enum* variables related to

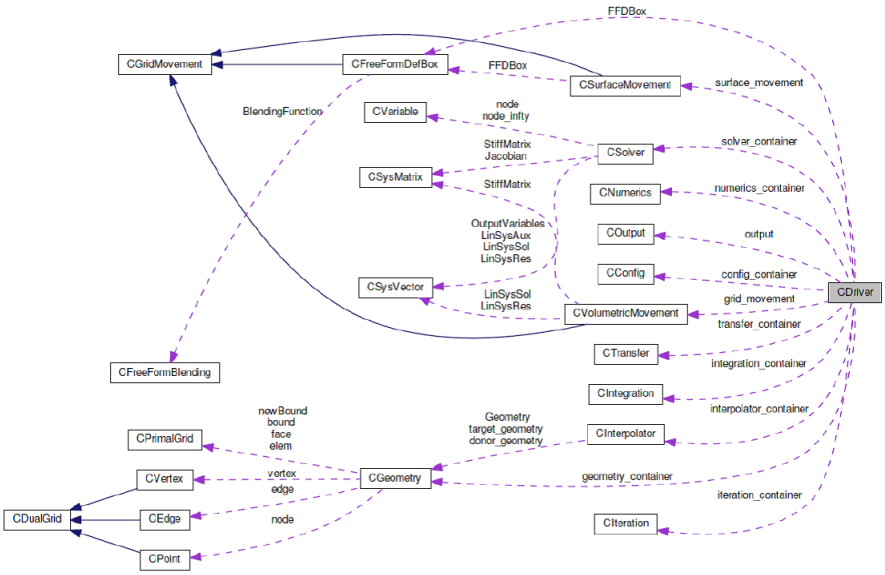

Figure 4.1: Driver structure profiling

the solver,instantiate it inside the file *option_structure.hpp* and eventually activate flags and methods related to it within the *config_structure.hpp* file.

```
  /*---Turbulent solver for multispecies simulations initialised
      ---*/
  if ((Kind_Solver == REACTIVE_NAVIER_STOKES) &&
      (Kind_Turb_Model != NONE))
    Kind_Solver = REACTIVE_RANS;

/*--- Set the solver methods ---*/
  switch (val_solver) {
    case REACTIVE_RANS:
      if (val_system == RUNTIME_REACTIVE_SYS) {
        SetKind_ConvNumScheme(Kind_ConvNumScheme_Flow,
            Kind_Centered_Flow,
                              Kind_Upwind_Flow, Kind_SlopeLimit_Flow,
                              SpatialOrder_Flow);
        SetKind_TimeIntScheme(Kind_TimeIntScheme_Flow);
      }
    case ...
```

config_structure.cpp

The need of this new solver, called REACTIVE RANS, is compulsory in order to be able to realise the communication between the solver that solves Euler or Navier-Stokes equations, that plays the role of the *mean-flow* solver, and the sequentially built *turbulent* flow, whose main solution, indeed turbulent viscosity and ω, will be used by the laminar counterpart to finalize the closures. It is therefore clear that without this bridge, the laminar solver is not able to retrieve those variables from the resolution of the turbulent model.

4.1.2 Driver Structure

The Driver class is where everything is essentially commanded down to the other classes, for this reason members of *CDriver* have to be the following:

```
/*--- Definition and of the containers for all possible zones.
    ---*/
iteration_container  = new CIteration*[nZone];
solver_container     = new CSolver***[nZone];
integration_container = new CIntegration**[nZone];
numerics_container   = new CNumerics****[nZone];
```

driver_structure.cpp

From these few lines of code the runtime polymorphic structure of SU2 is highlighted: indeed to each raw pointer is associated a precise features that comes from some input parameters the user defines within the configuration file according to the type of simulation he/she is interested in. For example the CSolver container has three raw pointers: one associated to the *iZone* of the computation if the simulation forsees a domain-splitting technique, Fig. 4.2, a second to the *iMesh* to distinguish Single-Grid or Multi-Grid, end eventually a third one for the *SolContainer_Position* to account for mean-flow and for turbulent-flow solver. In this way all the information is canalized through a cascade of raw pointers. Inside the *CDriver* class the job is essentially distributed within four main tasks/methods: *Preprocessing*,*Start_Solver*,

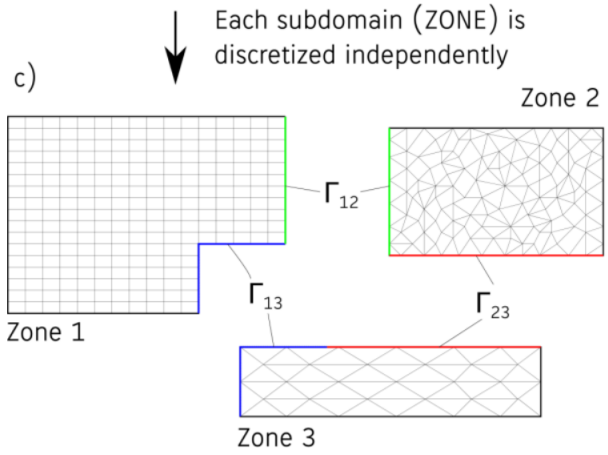

Figure 4.2: Zone splitting of the domain

Run and *Output*. The *Preprocessing* stage is of paramount importance in order to guide the runtime polymorphism through every initialization step. Within the *Solver_Preprocessing* boolean variables connected respectively to the mean reactive flow and to the turbulent one have to be activated:

```
/*--- Assign booleans ---*/
switch (config->GetKind_Solver()) {
case TEMPLATE_SOLVER: template_solver = true; break;
case EULER : euler = true; break;
case NAVIER_STOKES: ns = true; break;
case RANS : ns = true; turbulent = true; if
    (config->GetKind_Trans_Model() == LM) transition = true; break;
case POISSON_EQUATION: poisson = true; break;
case WAVE_EQUATION: wave = true; break;
case HEAT_EQUATION: heat = true; break;
case FEM_ELASTICITY: fem = true; break;
case ADJ_EULER : euler = true; adj_euler = true; break;
case ADJ_NAVIER_STOKES : ns = true; turbulent =
    (config->GetKind_Turb_Model() != NONE); adj_ns = true; break;
case ADJ_RANS : ns = true; turbulent = true; adj_ns = true;
    adj_turb = (!config->GetFrozen_Visc()); break;
case DISC_ADJ_EULER: euler = true; disc_adj = true; break;
case DISC_ADJ_NAVIER_STOKES: ns = true; disc_adj = true; break;
case DISC_ADJ_RANS: ns = true; turbulent = true; disc_adj = true;
```

```
    break;
case REACTIVE_EULER: reactive_euler = true; break;
case REACTIVE_NAVIER_STOKES: reactive_ns = true; break;
/*--- Multispecies turbulent simultions additions ---*/
case REACTIVE_RANS: reactive_ns = true; turbulent = true; break;
}

if(reactive_ns) {
solver_container[iMGlevel][FLOW_SOL] = new
    CReactiveNSSolver(geometry[iMGlevel], config, iMGlevel);
}

if (turbulent) {
  if (menter_sst) {
    solver_container[iMGlevel][TURB_SOL] = new
        CTurbSSTSolver(geometry[iMGlevel], config, iMGlevel);
    solver_container[iMGlevel][FLOW_SOL]->Preprocessing(geometry[iMGlevel],
        solver_container[iMGlevel], config, iMGlevel, NO_RK_ITER,
        RUNTIME_REACTIVE_SYS, false);
    solver_container[iMGlevel][TURB_SOL]->Postprocessing(geometry[iMGlevel],
        solver_container[iMGlevel], config, iMGlevel);
  }
```

driver_structure.cpp

The *Run* main goal is to care about the initial condition of the problem:

```
  /*--- Set initial condition in case of restart multispecies
      simulation for turbulent solver too ---*/
  if(!fsi && (config_container[ZONE_0]->GetKind_Solver() ==
      REACTIVE_EULER ||
             config_container[ZONE_0]->GetKind_Solver() ==
                 REACTIVE_NAVIER_STOKES ||
             (config_container[ZONE_0]->GetKind_Solver() ==
                 REACTIVE_RANS) ) ) {
    for(iZone = 0; iZone < nZone; ++iZone){
      solver_container[iZone][MESH_0][FLOW_SOL]->SetInitialCondition(
        geometry_container[iZone], solver_container[iZone],
        config_container[iZone], ExtIter);
  }
```

```
}
```

driver_structure.cpp

Same booleans are used for the Solver, Numerics and Integration Preprocessing stage for both the FLOW_SOL and the TURB_SOL. Within the *Run* part the methods of the $CMeanFlowIteration$ and $CMultiGridIntegration$ classes would be carried on.

```
/*--- For each zone runs one single iteration ---*/
for (iZone = 0; iZone < nZone; iZone++) {
  config_container[iZone]->SetIntIter(IntIter);

  iteration_container[iZone]->Iterate(output,
      integration_container, geometry_container,
      solver_container, numerics_container, config_container,
      surface_movement, grid_movement, FFDBox, iZone);
}
```

driver_structure.cpp

Note that in this specific work a Single-Grid iteration method were used, but modification within the code were apported for Multi Grid as well, in case any future developer would like to work on it without starting from scratch. Final *Output* part is connected to postprocessing, more into detail with the generation of either Tecplot or Paraview data to be read by these two software in order to produce a nice colorful representation of the solution.

4.1.3 Integration and Iteration Structure

Within the *Iterate* method is basically when the core of the computational structure is decided: not only in fact the global parameters specific for each type of simulation are set

```
/*--- Update global parameters ---*/
switch( config_container[val_iZone]->GetKind_Solver() ) {
```

```
  case NAVIER_STOKES: case DISC_ADJ_NAVIER_STOKES:
    config_container[val_iZone]->SetGlobalParam(NAVIER_STOKES,
        RUNTIME_FLOW_SYS, ExtIter); break;

  case RANS: case DISC_ADJ_RANS:
    config_container[val_iZone]->SetGlobalParam(RANS,
        RUNTIME_FLOW_SYS, ExtIter); break;

  /*--- Multispecies turbulent simultions additions ---*/
  case REACTIVE_RANS:
    config_container[val_iZone]->SetGlobalParam(REACTIVE_RANS,
        RUNTIME_REACTIVE_SYS, ExtIter);
    break;
   ...
}
```

iteration_structure.cpp

but also the iteration procedure is finalised, and the task is therefore transferred to a hierarchically lower class, $CMultiGridIntegration$, that is now responsible for space and time integration as well as setting the boundary conditions.

```
/*--- Reactive turbulent solver
  /*--- Solve the Euler, Navier-Stokes or Reynolds-averaged
     Navier-Stokes (RANS) equations (one iteration) ---*/
  if (config_container[val_iZone]->GetKind_Solver() ==
     REACTIVE_EULER ||
     config_container[val_iZone]->GetKind_Solver() ==
        REACTIVE_NAVIER_STOKES){
    integration_container[val_iZone][FLOW_SOL]->MultiGrid_Iteration(
    geometry_container, solver_container,
       numerics_container,config_container, RUNTIME_REACTIVE_SYS,
       IntIter, val_iZone);
  }
  else if (config_container[val_iZone]->GetKind_Solver() ==
     REACTIVE_RANS){
    integration_container[val_iZone][FLOW_SOL]->MultiGrid_Iteration(
    geometry_container, solver_container,
```

```
      numerics_container,config_container, RUNTIME_REACTIVE_SYS,
      IntIter, val_iZone);
  config_container[val_iZone]->SetGlobalParam(REACTIVE_RANS,
      RUNTIME_TURB_SYS, ExtIter);
  integration_container[val_iZone][TURB_SOL]->SingleGrid_Iteration(
  geometry_container, solver_container,
      numerics_container,config_container, RUNTIME_TURB_SYS,
      IntIter, val_iZone);
}
else
integration_container[val_iZone][FLOW_SOL]->MultiGrid_Iteration(
geometry_container, solver_container,
    numerics_container,config_container, RUNTIME_FLOW_SYS,
    IntIter, val_iZone);
```

iteration_structure.cpp

Within the *MultiGrid_Iteration* the method *MultiGrid_Cycle* is called, where first space integration is realized through imposition of boundary conditions, and lastly time integration is finalised.

4.2 Turbulent Closures

Once understood the fact that mean flow simulation and turbulent one happen sequentially in an iterative fashion and how they exactly interact at a solver level, the final step will be to present how turbulent closures are implemented within the code, putting a strong emphasis on all the numerical and coding challenges faced in order to bring this about.

4.2.1 Lost Branches

One of the main features of Polymorphism is that it allows to use methods and access members of Base classes directly through their Derived classes. Indeed this mechanism is somewhat spoiled when the information has to be retrieved

from a class not from the same branch upstream. As it is possible to observe

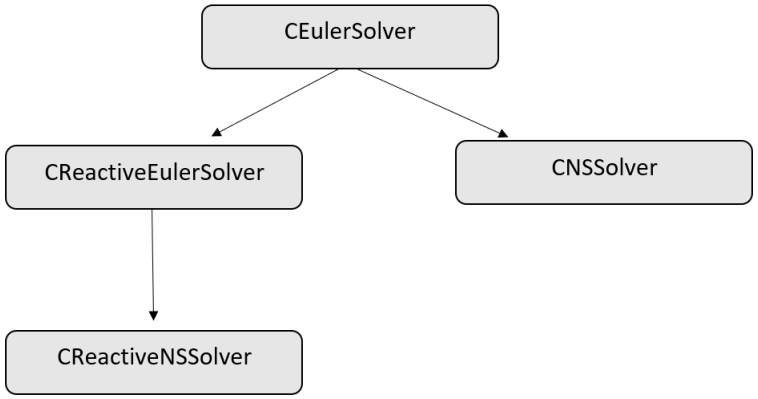

Figure 4.3: SU2 Solver Polymorphism

from the sketch proposed in Fig.4.3, in the originial structure of SU2, the class $CNSSolver$ derives directly from the $CEulerSolver$ class, and when a turbulent simulation is run, methods allowing for communication between the mean and the turbulent flows are already present within the NS Solver class, Fig.4.4. When a multi-species simulation is run instead, the mean flow Solver class derives straight from the Euler non reactive class, as shown in Fig. 4.3, missing hence all turbulent methods originally present within the NS Solver class. In this way therefore only simulation with low Reynolds number can be carried out, indeed within the boundaries of laminar regime. The main challenge of this part was therefore the one of understanding how to introduce the possibility to simulate a reactive flow in which turbulence is taken into account, without operating an invasive surgery of the original code structure: a good trade-off between efficiency and readability. A very first idea was the one of introducing a derived further class with indeed more specialised methods for the turbulence flow coupling, but also with a higher amount of already present variables that were carried out throughout the computation, burdening thus the occupied space in the heap. The final choice fell eventually on sticking to the original SU2 structure, introducing the possibility to opt for a turbulent coupling only through if clauses within the NS Solver class. As outlined in Fig.4.5, methods used in the original turbulence coupling of SU2 have to be tracked down throughout the code, properly modified in order to account for the new multi-species physics (See

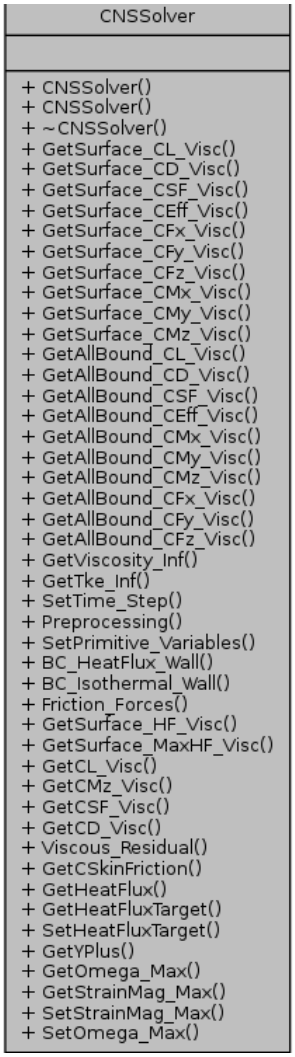

Figure 4.4: CNSSolver

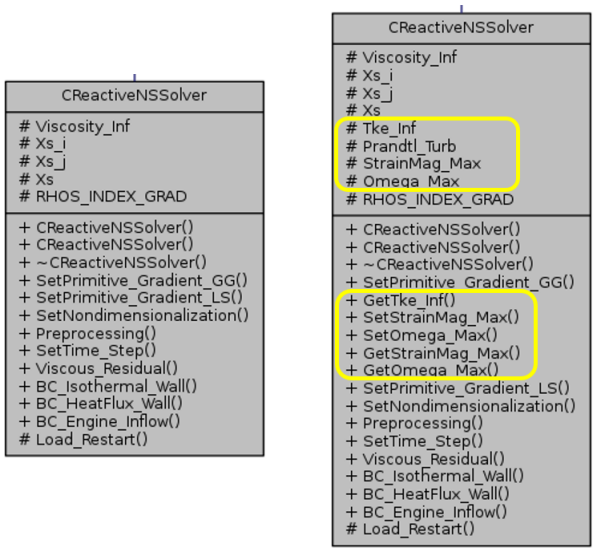

Figure 4.5: LHS: original laminar reactive Solver RHS: final turbulent reactive Solver

code examples below), and eventually added to the NS classes.

```
/*--- Set primitive and turbulent variables ---*/
//
//
bool CReactiveNSVariable::SetPrimVar(su2double eddy_visc,
    su2double turb_ke,CConfig* config) {

  /*--- Convert conserved to primitive variables using Euler
     version since primitives are the same ---*/
  bool nonPhys = CReactiveEulerVariable::SetPrimVar(config,turb_ke);

  /*--- Further turbulence coupling method ---*/
  SetEddyViscosity(eddy_visc);

  su2double dim_temp, dim_press;
  dim_temp = Primitive[T_INDEX_PRIM]*config->GetTemperature_Ref();
  dim_press =
     Primitive[P_INDEX_PRIM]*config->GetPressure_Ref()/101325.0;
  if(US_System) {
```

```
    dim_temp *= 5.0/9.0;
    dim_press *= 47.8803;
  }

  /*--- Compute transport properties --- */
  Ys = GetMassFractions();
  Laminar_Viscosity = library->ComputeEta(dim_temp,
      Ys)/config->GetViscosity_Ref();
  if(US_System)
    Laminar_Viscosity *= 0.02088553108;
  Thermal_Conductivity = library->ComputeLambda(dim_temp,
      Ys)/config->GetConductivity_Ref();
  if(US_System)
    Thermal_Conductivity *= 0.12489444444;
  /*--- Compute binary diffusion coefficents. NOTE: The empirical
      formula employed in the library should return it in cm2/s
      ---*/
  Diffusion_Coeffs = library->GetDij_SM(dim_temp,
      dim_press)/(config->GetVelocity_Ref()*config->GetLength_Ref()*1.0e4);
  if(US_System)
    Diffusion_Coeffs *= 3.28084*3.28084;
  }

  return nonPhys;
}
```

variable_direct_reactive.cpp

```
/*--- Interfacing turbulent variables with specific reactive flow
    methods (CNSReactiveVariable class) ---*/

    bool CReactiveEulerVariable::Cons2PrimVar(CConfig* config,
        su2double* U, su2double* V,su2double val_ke) {

  /*--- Including turbulent kinetic energy contribution ---*/
  rho = U[RHO_INDEX_SOL]; // Density [Kg/m3]
  rhoE = U[RHOE_INDEX_SOL] - rho*val_ke; // Density*total energy
      per unit of mass [J/m3]
```

```
...
}
```

variable_direct_reactive.cpp

It is to be highlighted that in the above mentioned analysis, only the issue with Solver classes has been mentioned, but same procedure has to be applied to the $CNSReactiveVariable$ class, the $CSourceReactive$ and the $CAvgGradReactive_Flow$ classes, respectively the Numerics derived class dealing with the source term and the viscous one.

4.2.2 Lost Variables

As mentioned in the chapter here above, one of the challenging task was to retrieve the variables necessary to complete the turbulent closure, deriving them from those already present within the Reactive class family. Within the turbulent closure proposed within this report, the variables $massfraction$ and its gradient are compulsory in order to express the viscous flux of each species and their enthalpic contribution transported around by turbulence fluctuations (see equation 2.20). Nonetheless within the laminar-reactive implementation, only $molarfraction$ variables were used as class members, and consequently only molar fraction gradients were computed. In order to retrieve $massfraction$ gradients, [5] proposed the algorithm in equation 4.1, thanks to which it is possible to obtain $massfraction$ gradients by solving a simple linear system.

$$\nabla x_i = \frac{m}{m_i} y_i \nabla\sigma - m x_i \sum_j \nabla y_j / m_j + \sigma \frac{m}{m_i} \nabla y_i = \sum_j \tilde{M}_{ij} \nabla y_j \tag{4.1}$$

with

$$\tilde{M}_{ij} = \begin{cases} \frac{m}{m_i}\left(y_i - x_i + \sigma\right) & \text{if } i = j \\ m\left(\frac{y_i}{m_i} - \frac{x_i}{m_j}\right) & \text{if } i \neq j \end{cases} \tag{4.2}$$

where m_i, x_i and y_i are respectively mass, molar fraction and mass fraction of the i-th species. As far as the code is concerned, within the $CAvgGradReactiveBoundary$ class, right before the closure implementations, the following code line were added, in which methods of the Eigen class were called to easily manage the resolution of the linear system.

```
RealMatrix M_tilde = Get_Molar2MassGrad_Operator();
Mean_Mass_Grads.resize(nSpecies,nDim);

for( iDim = 0; iDim < nDim; ++iDim)
    Mean_Mass_Grads.col(iDim) =
        M_tilde.colPivHouseholderQr().solve(Mean_GradPrimVar.col(iDim).segment(
    RHOS_INDEX_AVGGRAD,nSpecies));

    for (iSpecies = 0 ; iSpecies < nSpecies ; iSpecies++){
      for ( iDim = 0; iDim < nDim; ++iDim ){
        if((std::abs(Mean_GradPrimVar(RHOS_INDEX_AVGGRAD+iSpecies,iDim)))<1e-8)
        {
          Mean_Mass_Grads(iSpecies,iDim)=0.0;
        }
      }
    }
```

numerics_direct_reactive.cpp

where indeed the method $Get\_Molar2MassGrad\_Operator$ assembles the matrix. Not at all trivial was indeed the choice of the tolerance connected to the result of the linear system: indeed $colPivHouseholderQr$ was chosen not for its not so fast convergence properties, but for the precision level with which the final result is computed. This is of a paramount importance since even very small errors in the linear system results were giving rise to background errors like giving birth to species gradient that were not there. Having said that the matter is very delicate: on one side one has to be careful not to take too weak limit otherwise nonphysical results as well as numerical instabilities may arise, but on the other side too strong imposition may curb the precision of the result and cut correct physical information generated by the model. The tolerance chosen in this work was $1e{-}8$, but of course according to physical regimes, mesh refinement and so on results may change consis-

tently if a different tolerance is chosen. The final remark connected to this section concerns the changing needed for the solver relative to the turbulent flow: in the non-reactive SU2 structure, turbulent-flow classes communicate variables with the mean-flow classes through a pointer to primitive variables, among which indeed the turbulent viscosity and turbulent kinetic energy. Hidden in the turbulent Solver and Numerics structure there were methods that were wrongly calling those two variables from a place in which they were not anymore. Finding such small thing in a such a big code was not possible without the use of GDB debugger, that helped sort things out by smoothly going through the whole code, and through a complex structure of *DEBUG_BOOL* that turn out in the end to be very useful to double check the validity of some algorithm as well the physical correctness of some parameters.

```
/*--- DEBUG_BOOL for quick double-check of flux closure
    variables ---*/
if(config->Get_debug_visc_flow()){

  std::cout<<" --------------Turbolent add-on---------------
      "<<std::endl;

  std::cout<<" rho -----------> "<<rho<<std::endl;
  std::cout<<" TKE -----------> "<<Mean_Turbolent_KE<<std::endl;
  std::cout<<" Cp ----------->
      "<<std::accumulate(Cps.cbegin(),Cps.cend(),0.0)/nSpecies<<std::endl;
  std::cout<<" PrT -----------> "<<Prandtl_Turb<<std::endl;
  std::cout<<" mu_t ----------->
      "<<Mean_Eddy_Viscosity<<std::endl;
  std::cout<<" T_Grad ----------->
      "<<Mean_GradPrimVar(T_INDEX_AVGGRAD,0)<<" - "
           <<Mean_GradPrimVar(T_INDEX_AVGGRAD,1)<<std::endl;
  std::cout<<" Velocities ----------->
      "<<Mean_PrimVar[VX_INDEX_PRIM]<<" - "
                    <<Mean_PrimVar[VX_INDEX_PRIM +
                        1]<<std::endl;
  std::cout<<" Grad_Vel----------->
      "<<Mean_GradPrimVar(VX_INDEX_AVGGRAD,0)<<" - "
           <<Mean_GradPrimVar(VX_INDEX_AVGGRAD ,1)<<std::endl;
  std::cout<<Mean_GradPrimVar(VX_INDEX_AVGGRAD + 1,0)<<" - "
           <<Mean_GradPrimVar(VX_INDEX_AVGGRAD +
```

```
                1,1)<<std::endl;
  }
```

numerics_direct_reactive.cpp

4.2.3 Code Implementation and Numerical Issues

The idea behind turbulent closure implementations is the one of trying to save as much code repetition as possible, keeping at the same time the structure simple and close to the the original one for a clear sake of readability. Within the laminar-reactive structure fluxes were computed and projected along the normal direction of each cell side; similar thing was happening for the Jacobian second-order tensor, whose two contributions coming from the introduction of a change-of-variable operator were assembled together within the same method.

```
/*--- Set the laminar component of viscous residual tensor ---*/
SetLaminarTensorFlux(Mean_PrimVar, Mean_GradPrimVar, Normal,
  Mean_Laminar_Viscosity, Mean_Thermal_Conductivity,
  Mean_Dij,config);

if (config->GetKind_Turb_Model() == SST){

  unsigned short jDim;

  /*--- Local turbolent variables ---*/
  su2double Mean_Eddy_Viscosity, Mean_Turbolent_KE;
  Mean_Eddy_Viscosity = 2.0/(1.0/Eddy_Viscosity_i +
      1.0/Eddy_Viscosity_j);
  Mean_Turbolent_KE = 0.5*(turb_ke_i + turb_ke_j);
  Vec Mean_GradTKEVar(nDim);
  Mean_GradTKEVar.setZero();

  for(jDim = 0; jDim < nDim; ++jDim)
        Mean_GradTKEVar(jDim) = 0.5*(Grad_Tke_i[jDim] +
            Grad_Tke_j[jDim]);
```

```
  SST_Reactive_ResidualClosure(Mean_GradTKEVar,Mean_PrimVar,
  Mean_GradPrimVar,Normal,Mean_Eddy_Viscosity,Mean_Turbolent_KE,
  Mean_Laminar_Viscosity,config);

}
```

numerics_direct_reactive.cpp

As it is possible to infer from the code above, $SetLaminarTensorFlux$ include only those numerical operation connected to the build-up of laminar viscous fluxes, whereas the turbulent closure additions were included within the method $SST\_Reactive\_ResidualClosure$, presented here below.

```
/*--- Viscous flux closure connected to Momentum, Energy and
    Species equations ---*/

  for( iDim = 0; iDim < nDim; ++iDim) {

    for( jDim = 0; jDim < nDim; ++jDim) {


      Flux_Tensor[RHOVX_INDEX_SOL + jDim][iDim] +=
          tau_turb(iDim,jDim);
      Flux_Tensor[RHOE_INDEX_SOL][iDim] +=
          tau_turb(iDim,jDim)*Mean_PrimVar[VX_INDEX_PRIM + jDim];
    }

    /*--- Closure for species using molar fractions as
        approximation ---*/
    for( iSpecies = 0; iSpecies < nSpecies; ++iSpecies) {
       Proj_Flux_Tensor[RHOS_INDEX_SOL + iSpecies] +=
           Mean_Eddy_Viscosity/(Prandtl_Turb*Lewis_Turb) *
                           Mean_Mass_Grads(iSpecies,iDim)*
                               Normal[iDim];
     }

     /*--- Fick's law partial densities closure --*/
```

```
    for( iSpecies = 0; iSpecies < nSpecies; ++iSpecies) {
      Flux_Tensor[RHOE_INDEX_SOL][iDim] +=
          Mean_Eddy_Viscosity/(Prandtl_Turb*Lewis_Turb) *
                                hs[iSpecies]*Ys[iSpecies]
                                *Mean_Mass_Grads(iSpecies,iDim);
    }

   /*--- Fick's law partial sensible enthalpies closure --*/
   for( iSpecies = 0; iSpecies < nSpecies; ++iSpecies) {
   Flux_Tensor[RHOE_INDEX_SOL][iDim] +=
      Mean_Eddy_Viscosity/Prandtl_Turb*Cps[iSpecies]*Ys[iSpecies]*
                                  Mean_GradPrimVar(T_INDEX_AVGGRAD,iDim);
                                }
   /*--- Wilcox closure for turbolent ke and main stresses
      turbolent transport term --*/
   Flux_Tensor[RHOE_INDEX_SOL][iDim] += (Mean_Laminar_Viscosity
      + Mean_Eddy_Viscosity/sigma_k) * mean_tkegradvar(iDim);


  }
```

numerics_direct_reactive.cpp

Everything is therefore wrapped up, before the final projection that happens outside the closure flux methods.

```
/*--- Projected flux for momentum and second contribution for
   energy ---*/
for( iDim = 0; iDim < nDim; ++iDim) {
  for( iVar = RHOVX_INDEX_SOL; iVar < RHOVX_INDEX_SOL + nDim;
     ++iVar)
  Proj_Flux_Tensor[iVar] += Flux_Tensor[iVar][iDim]*Normal[iDim];
  Proj_Flux_Tensor[RHOE_INDEX_SOL] +=
     Flux_Tensor[RHOE_INDEX_SOL][iDim]*Normal[iDim];
}
```

numerics_direct_reactive.cpp

Same fashion as far as the Jacobian second-order tensor is concerned, where

auxiliary matrixes accounting for the splitting operator are passed by reference first to the method specialised for the laminar-reactive closure, and eventually to the one working exclusively with turbulent ones.

```
/*--- Build auxiliary matrices for jacobian components ---*/
   AuxMatrix dFdVi(nVar,RealVec(nVar));
   AuxMatrix dFdVj(nVar,RealVec(nVar));
   AuxMatrix dVdUi(nVar,RealVec(nVar));
   AuxMatrix dVdUj(nVar,RealVec(nVar));

   /*--- Compute laminar jacobian components ---*/
   SetLaminarViscousProjJacs(Mean_PrimVar, Mean_Laminar_Viscosity,
      Mean_Thermal_Conductivity, alpha, Grad_Xs_norm, Ds,
     std::sqrt(dist_ij_2), Area, UnitNormal, config, dFdVi, dFdVj,
        dVdUi, dVdUj);

     /*--- Add turbolent jacobian closure ---*/
     if (config->GetKind_Turb_Model() == SST){
       /*--- Local turbolent variables ---*/
       su2double Mean_Eddy_Viscosity, Mean_Turbolent_KE;
       Mean_Eddy_Viscosity = 2.0/(1.0/Eddy_Viscosity_i +
           1.0/Eddy_Viscosity_j);
       Mean_Turbolent_KE = 0.5*(turb_ke_i + turb_ke_j);
       SST_Reactive_JacobianClosure(UnitNormal,Mean_PrimVar,
       Mean_Turbolent_KE,Area,Mean_Eddy_Viscosity,dist_ij_2,
       dFdVi,dFdVj,Mean_Laminar_Viscosity,config);
      }

     unsigned short iDim,iVar,jVar,kVar;
     /*--- Common terms: Proj_Flux_Tensor, if turbolence is
        active, contains Reynolds stress tensor as well ---*/
     for(iDim = 0; iDim < nDim; ++iDim) {
       dFdVi[RHOE_INDEX_SOL][RHOVX_INDEX_SOL + iDim] +=
           0.5*Proj_Flux_Tensor[RHOVX_INDEX_SOL + iDim];
       dFdVj[RHOE_INDEX_SOL][RHOVX_INDEX_SOL + iDim] +=
           0.5*Proj_Flux_Tensor[RHOVX_INDEX_SOL + iDim];
     }

     for(iVar = 0; iVar < nVar; ++iVar) {
       for(jVar = 0; jVar < nVar; ++jVar) {
```

```
      val_Jacobian_i[iVar][jVar] = 0.0;
      val_Jacobian_j[iVar][jVar] = 0.0;
    }
  }

  for(iVar = 0; iVar < nVar; ++iVar) {
    for(jVar = 0; jVar < nVar; ++jVar) {
      for(kVar = 0; kVar < nVar; ++kVar) {
        val_Jacobian_i[iVar][jVar] +=
            dFdVi[iVar][kVar]*dVdUi[kVar][jVar];
        val_Jacobian_j[iVar][jVar] +=
            dFdVj[iVar][kVar]*dVdUj[kVar][jVar];
      }
    }
  }
```

numerics_direct_reactive.cpp

Analogous implementation is followed by the *CSourceReactive* class, responsible eventually for the chemical closure.

```
/*--- Initializing double tensor species-reactions through
   library method ---*/
 library -> SetSourceTerm(dim_temp, dim_rho, Ys);



 if (config->GetKind_Turb_Model()==SST){

   /*--- Initializing double tensor derivative of reaction source
      term w.r.t. species through library method ---*/
   library -> Set_DfrDrhos(dim_temp, dim_rho);
   double PaSR_lb = config-> Get_PaSR_LB();
   library->AssemblePaSRConstant(omega_turb,C_mu,PaSR_lb);

   omega.resize(Ys.size(),0.0);

   /*--- Turbolent source term for every species ---*/
   for (iSpecies = 0; iSpecies < nSpecies; iSpecies++)
```

```
      omega[iSpecies] = library ->
          GetMassProductionTerm(iSpecies);

  }
 else
 {

   omega.resize(Ys.size(),0.0);

   Eigen::VectorXd::Map(&omega[0],Ys.size())=
       library->GetMassProductionTerm();

 }
```

numerics_direct_reactive.cpp

Differently from the reactive-laminar implementation, in order to build the algorithm proposed for the PaSR (see equation 2.25) it was not just enough to build a full source term summing up each reaction contribution. A new method, within the *ReactingModelLibrary* class had to be outlined, that is *SetSourceTerm*, whose main goal is to build a second-order tensor specifying for each species (rows), the source term associated to each reaction (cols). In order to smoothly deal with matrix operations, especially those connected to summation along columns (for example in case a laminar-reactive simulation is to be chosen with original laminar closure for source term: PSR), Eigen library is used once again, and interfaced with the original structure of the reactive-laminar implementation through a *Map* feature, as it is possible to infer from the code lines above. Here below instead the use of overloading for the *GetMassProductionTerm* in order to distinguish the laminar from the turbulent algorithm to operate closures.

```
  /* ---Source term for -th species is built by a weight through
      PaSR contant method . ---*/
 double ReactingModelLibrary::GetMassProductionTerm( const
     unsigned short iSpecies){

   double omega_ith = 0.0;
```

```
  /*--- Assembling i-th species source production term ---*/
  for (unsigned short iReac = 0; iReac < nReactions ; iReac++)
  omega_ith += PaSRConstant[iReac]*omega_i_r(iSpecies,iReac);

  return omega_ith;

}

/* --- This function computes the omega term in laminar case
    (PSR). ---*/
Eigen::VectorXd ReactingModelLibrary::GetMassProductionTerm(void)
    {

  Eigen::VectorXd omega = omega_i_r.rowwise().sum();

  return omega;

}
```

reacting_model_library.cpp

The method *AssemblePaSRConstant* speaks for itself, and computes inside the method the combustion time, that is the reacting time needed by the fastest reacting species within each reaction (see equation 2.26).

```
/* Assemble the PaSR constant for the turbolence model . */
  void ReactingModelLibrary::AssemblePaSRConstant(const double
      omega_turb,const double C_mu, const double PaSR_lb){

  PaSRConstant.resize(nReactions);



  /*--- Assmebling mixing time due to turbolence ---*/
  double tau_mix = 1/(C_mu*omega_turb);

  for (unsigned short iReac = 0; iReac < nReactions ; iReac++){

    double k_th;
```

```
    /*--- Assembling slowest combustion time for r reaction---*/
    double tau_comb_r = GetTimeCombustion_r(iReac);

    if(isinf(tau_comb_r))
      k_th = 1.0;
    else if ((tau_comb_r/(tau_comb_r + tau_mix)) < PaSR_lb)
      k_th = PaSR_lb;
    else
      k_th = tau_comb_r/(tau_comb_r + tau_mix);

    /*--- Saving value for future purposes ---*/
    PaSRConstant[iReac] = k_th;
  }


}

/* Compute the smallest reaction time for each reaction among all
    species. */
double ReactingModelLibrary::GetTimeCombustion_r(const unsigned
    short iReac){

  std::vector<unsigned short> Index_list;
  std::vector<double> iReac_species;

  /*--- Searching for species taking part into iReac reaction
      ---*/
  for(unsigned short iSpecies = 0; iSpecies < nSpecies;
      iSpecies++){
    if(Stoich_Coeffs_Products(iSpecies,iReac) ||
       Stoich_Coeffs_Reactants(iSpecies,iReac))
    Index_list.push_back(iSpecies);
  }


  for(const auto it : Index_list)
    iReac_species.push_back(std::fabs(Df_rDrho_i(it,iReac)*mMasses[it]));

  double highest_derivative =
```

```
        *std::max_element(iReac_species.begin(),iReac_species.end());

    return 1/highest_derivative;

}
```

reacting_model_library.cpp

A final remark is to be brought about as far as the variable $PaSR\_lb$ is concerned. It happened in fact that during ignition procedure, that is during the very beginning of the combustion process when a sparkle is generated inside the engine in order to make reactants interact with each other, the combustion time for a certain speaker was enormously smaller than the time needed by species instead to diffuse, represented here by the tau. This was giving rise to strong numerical instabilities connected mainly to the fact that combustion was switch-off essentially as soon as it was turned-on. One of the strategy opted out within this project was to limit this nonphysical drop of the PaSR constant (indeed, as always happens with turbulent closure, algorithms presented are not analytically derived from a well-known physics, but just possible interpretation of what is physically happening), by imposing a lower bound for the k PaSR constant, in case this truned out to be too low.

Chapter 5

Results

5.1 Turbulent Flat Plate

The very first proposed test case is the flat plate: here it will not only be

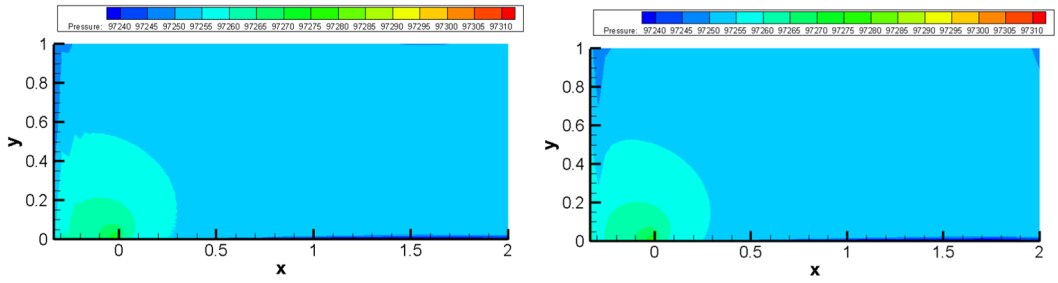

Figure 5.1: Pressure distribution: left original turbulent SU2, right the reactive counterpart

tested the capability to catch boundary layer but it is also a good validation for the reactive model to be able to reproduce a strongly analogous result of the one of the classical flat plate. The computational mesh for the plate is composed of quadrilaterals with 65 nodes in both the x and y directions (see Fig.5.2). The plate is along the lower boundary of the domain (y = 0) starting at x = 0 m and is of length 0.3048 m. As far as the physics

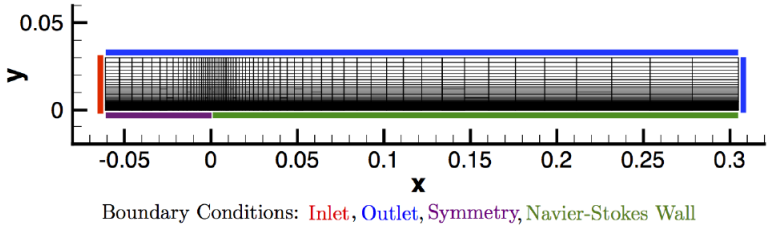

Figure 5.2: 2D flat plate mesh

is concerned, standard free-stream and boundary conditions were studied: indeed 300 K freestream temperature of air fluid travelling at 70 m/s, whereas from the inlet is coming air at 100000 Pa total pressure. Note that initial and boundary conditions for the turbulent flow are straightly derived from mean flow parameters.

```
/*--- Flow infinity initialization stuff ---*/
  su2double rhoInf, *VelInf, muLamInf, Intensity, viscRatio,
      muT_Inf;

  rhoInf    = config->GetDensity_FreeStreamND();
  VelInf    = config->GetVelocity_FreeStreamND();
  muLamInf = config->GetViscosity_FreeStreamND();
  Intensity = config->GetTurbulenceIntensity_FreeStream();
  viscRatio = config->GetTurb2LamViscRatio_FreeStream();

  su2double VelMag = 0;
  for (iDim = 0; iDim < nDim; iDim++)
  VelMag += VelInf[iDim]*VelInf[iDim];
  VelMag = sqrt(VelMag);

  kine_Inf = 3.0/2.0*(VelMag*VelMag*Intensity*Intensity);
  omega_Inf = rhoInf*kine_Inf/(muLamInf*viscRatio);

  /*--- Eddy viscosity, initialized without stress limiter at the
      infinity ---*/
  muT_Inf = rhoInf*kine_Inf/omega_Inf;
}
```

solver_direct_turbulent.cpp

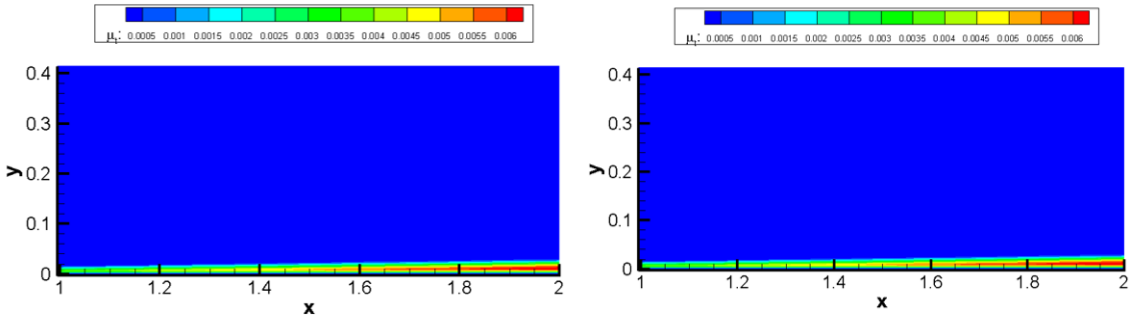

Figure 5.3: Turbulent viscosity: left original turbulent SU2, right the reactive counterpart

From Fig. 5.1 it is possible to appreciate the very good match between pressure distribution at steady state,whereas in Fig. 5.3 the resolution of turbulent viscosity inside the boundary layer.

5.2 Combustion Chamber

The second and final test case has to do with the simulation of a turbulent combustion process within a combustion chamber of turbojet engine. The chemical reactions involved are solely the ones here below:

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

The chamber is 0.125 m and 0.006 m height, whereas the computational mesh is 100 x 80 elements (see Fig. 5.3. In order to catch even better the behaviour near the walls a tapered mesh with 90 cells in y direction and 20 nodes from 0 m to 0.001 m and from 0.005 m to 0.006 m, is employed. Boundary conditions are summed up in Fig. 5.4. $TurbulenceIntensity$ and $turbulent - to - laminar$ viscosity ratio are taken respectively 0.05 and 10, according to standard value already used in analogous simulation [1]. Follow-

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 5.4: Boundary conditions for combustion chamber test-case

ing the work of [4], a first simulation in which species are simply let diffuse throughout the domain, without reacting with each other yet, is carried out, following by a second one in which a sparkle is artificially generated within the computational domain (see [1]) , in order to boost the chemical reactions finally happening within the engine liner. A peculiar feature of this

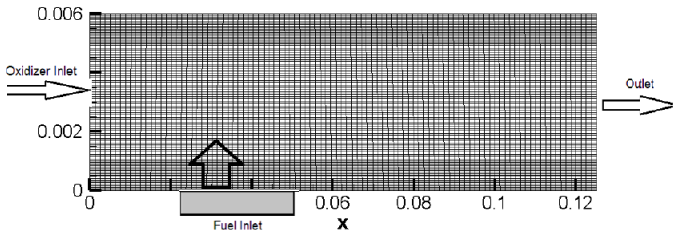

Figure 5.5: 2D combustion chamber mesh

procedure is the fact that the number of iterations chosen for the ignition step is of paramount importance in determining the convergence results of the final step, in which a steady-state solution is aimed at. Within the limits of this projects many combinations were tried: indeed being combustion process within the PaSR model more dominated by diffusion in cells where turbulence is higher, a higher number of ignition iteration needed (w.r.t. the laminar-reactive case) to be used in order to achieve an intermediate solution "burnt" enough to ensure the combustion to keep on going until the desired residual for convergence. It is to be highlighted that the numerical refinement of parameters in this part is very delicate: changing the number of iteration as well as the aforementioned lower bound on the PaSR constant gives rise

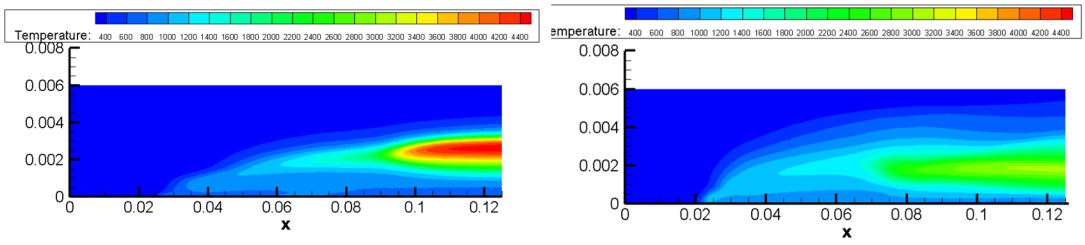

Figure 5.6: Temperature profile steady-state for (left) laminar-reactive with PSR, (right) turbulent reactive with PaSR

to consistently different results. Eventually it is to be noted the importance to further refine the steps-procedure used in [4] when turbulent combustion is to be approached: combustion propagation is strongly hindered within turbulence PaSR approach, since very fast reacting species limit combustion to certain zones of the domain in which, unavoidably, oxidizer and fuel will happen to run out quickly. From Fig.5.6 it is possible to see some of

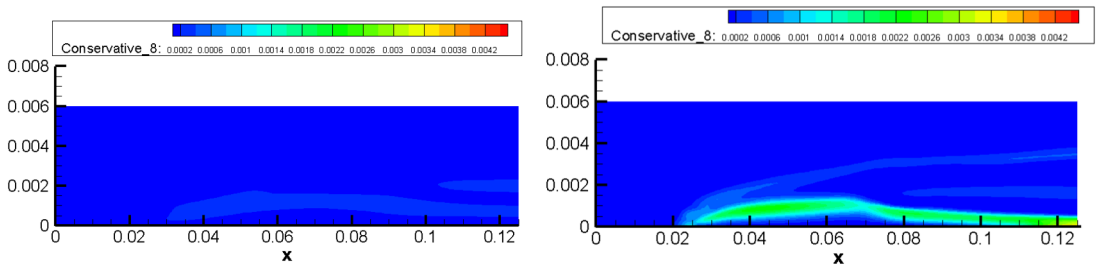

Figure 5.7: O2 Mass Fraction at steady-state for (left) laminar-reactive with PSR, (right) turbulent reactive with PaSR

the results already obtained by [3], that is a temperature profile much more spread in the PaSR case w.r.t. to the PSR one: combustion is happening in a wider region of the engine, this of course compromises with a lower highest Temperature near the liner exhaust. Figures 5.7,5.8,5.9 show in the endd another expected result, that is the behavior of combustion products. Their diffused concentration all over the engine in the PaSR case thus a strong indicator of a combustion happening in a wider region of the domain w.r.t.

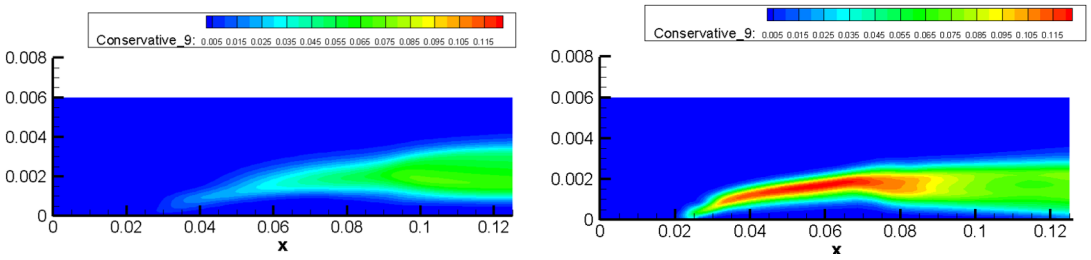

Figure 5.8: CO2 Mass Fraction at steady-state for (left) laminar-reactive with PSR, (right) turbulent reactive with PaSR

the PSR case.A final remark is to be made for the lower-bound strategy for the PaSR constant chose within the context of this work: a value of 0.2 is set in the results just shown, indeed a higher value will bring the result closer to the PSR approach, whereas lower values than 0.2 may give rise to strong instabilities, spoiling hence the desired result.

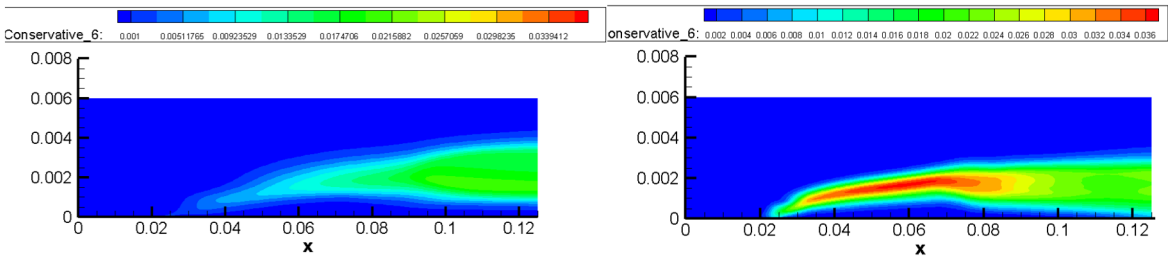

Figure 5.9: H2O Mass Fraction at steady-state for (left) laminar-reactive with PSR, (right) turbulent reactive with PaSR

Chapter 6

Conclusions

The SU2 suite is now able to perform turbulent simulations of multi-species flows in 2D and 3D. Future work will indeed regard:

- Improving the implicit formulation as far as the Jacobian is concerned: indeed within the combustion chamber test case simulation is very sensitive close to fuel inlet, and the generated turbulence creates strong instabilities that hinder the performance of the implicit scheme. For this purpose a Dual-Time-Stepping strategy may resolve this issue.
- Elaborating a stronger strategy for the choice of the PaSR constant: a lower bound was the simplest way being the time for this work very limited. A sensitivity analysis could indeed help to understand which are the stability ranges for this kind of approach and put the basis for a more elaborated strategy formulation
- Different steps procedure for generating combustion process: PaSR algorithm is indeed more physically precise than the PSR, but for this mere reason it requires a more elaborated strategy than the one proposed by [4] for the laminar combustion.

Bibliography

[1] *ANSYS: modeling of turbulent combustion*. URL: `https://www.ansys.com/blog/turbulent-combustion-flamelet-generated-manifolds`.

[2] Golovitchev. "Development of Universal Model of Turbulent Spray Combustion". In: *TFR Research Proposal* (2001).

[3] Alessandro Mazzetti. "Numerical Modeling and Simulations of Combustion Processes in Hybrid Rocket Engines," PhD thesis. Politecnico di Milano, 2014.

[4] Giuseppe Orlando. "Development of a laminar numerical solver for reacting flows based on the SU2 CFD code". MA thesis. Politecnico di Milano, 2019.

[5] K.S.C. Peerenboom. "Mass conservative finite volume discretization of the continuity equations in multi-component mixtures". In: *Journal of Computational Physics* (2011).

[6] D. Veynante. *Turbulent Combustion Modeling*. Progress in Energy and Combustion Science, 2002.

[7] D.C. Wilcox. *Turbulence Modeling for CFD*. DCW Industriese, 2006.