TABLE OF CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

Page

LIST OF SYMBOLS AND UNITS OF MEASUREMENTS

c	Speed of sound
C_p	Specific heat capacity
C_v	Volumetric heat capacity
d	Wall distance
D	Strain tensor
e	Total energy
E	Total energy per unit volume
$\mathbf{f}_v$	Body force vector
$\mathbf{F}^{adv}$	Advection flux
$\mathbf{F}^{diff}$	Diffusion flux
$\mathbf{F}^{s}$	Heat source flux
he	Characteristic length of an element
i	Internal energy
$\mathbf{I}$	Identity matrix
$\mathbf{J}$	Jacobian matrix of geometric transformation
k	Thermal conductivity
$[\mathbf{K}]$	Global stiffness matrix
$[\mathbf{M}]$	Global mass matrix
$\mathbf{n}$	Unit normal outward pointing vector
$[\mathbf{N}]$	Block diagonal matrix of shape functions

N_i	Shape function
p	Pressure
Pe	Peclet number
r	Heat source
$\mathbf{R}$	Residual vector
Re	Reynolds number
t	Time
T	Temperature
$\mathbf{V}$	Vector of conservative variables
$\{\mathbf{V}^h\}$	Vector of nodal unknowns
$\mathbf{W}$	Test function
x, y, z	Cartesian coordinates
ξ, η, ζ	Coordinates of the reference domain
Ω	Domain of integration
Γ	Boundary of the domain
σ	Stress tensor
ρ	Density
ν	Dynamic viscosity
ν_t	Turbulent viscosity
τ	Stabilization matrix
μ_c	Artificial shock capturing viscosity

ς	Sensor of pressure variation
$\gamma = \frac{C_p}{C_v}$	Adiabatic index
$\tilde{\omega}$	Module of flow vorticity
$c_{b1}, c_{b2}, \sigma, c_{w2}, c_{w3}, c_{v1}, c_{v2}, k$	Empirical constants
$f_{v1}, \tilde{f}_{v2}, \tilde{f}_{v3}, f_w, g, r$	Intermediate functions

INTRODUCTION

Generalities

Fluid dynamics is the branch of mechanics that studies the mechanics and heat transfer related to the motion of fluids, including liquids and gases. Common examples of phenomena involving fluid dynamics include airflow around an aircraft, ocean currents, engine turbines, and the circulatory system of the human body. Fluid dynamics has two subdisciplines. One is hydrodynamics, which deals with liquids in motion. The other is aerodynamics, which deals with air and gases in motion, especially flows over a plane.

The development of fluid dynamics can be traced back to Archimedes, who provided the fundamental principles of hydrostatics in his work *On Floating Bodies* (Archimedes, 287BC-212BC). He was the first person who summarized the mechanics of static fluid. Newton contributed significantly to fluid dynamics in his work *The Mathematical Principles of Natural Philosophy*, in which he discussed fluid resistance and wave motion. He established fluid dynamics as an independent branch of mechanics. The French engineer Navier (Navier, 1823) and the British mathematician Stokes (Stokes, 1845) independently proposed a theory showing how viscosity can have an effect on a fluid. This theory has led to the development of the Navier-Stokes equations.

Mathematically, it is difficult to find the exact solution for the Navier-Stokes equations. Computational fluid dynamics (CFD) methods have become powerful tools for solving such equations as a supplement to experimental analysis of complex fluid phenomena. CFD includes the finite element method (FEM), the finite volume method (FVM), the finite difference method, etc. FEM is commonly used in such industries as aerospace, mechanical manufacturing, nuclear power, and civil engineering. The first idea of FEM can be traced back to ancient times when mathematicians calculated the circular constant using polygons to approximate a circle. With the development of high-speed computation and new algorithms, FEM has gained in popularity over the years. There are currently numerous commercial FEM

software products on the market, such as Nastra (MSC Software 2015), Ansys (Ansys, 2015), and Abaqus (Dassault Systèmes, 2015).

When a fluid moves smoothly and steadily in parallel layers, the flow is said to be laminar. When a fluid moves in irregular paths, the flow is said to be turbulent. In compressible turbulent flows, the velocity, pressure, density, viscosity, and temperature fluctuate. A small Reynolds number usually results in laminar flow, and a high Reynolds number usually results in turbulent flow. The flow is said to be transitional when the Reynolds number is high but not sufficiently high to make the flow turbulent.

Turbulence will start to appear as the Reynolds number increases. Turbulence is a random phenomenon, and it is hard to predict the variations in both space and time. It is usually treated statistically. The velocity field is three-dimensional (3D) and rotational. Turbulence has a diffusive character; it increases the rate of homogenization, the transport of mass, momentum, and energy. It also has a dissipation characteristic; kinetic energy is rapidly converted into internal energy.

The earliest description of turbulence can be traced back to Leonardo da Vinci (Lumley, 1997). In 1877, Boussinesq (Boussinesq, 1877) proposed the hypothesis that turbulent stresses are linearly proportional to mean strain rates. This hypothesis greatly influenced the development of the study of turbulence. In 1883, Osborne Reynolds (Reynolds, 1883) conducted experiments to visualize the turbulence phenomenon in circular conduits. He introduced the idea of decomposing the flow variables into mean and fluctuating parts. This led to the development of the Reynolds-averaged Navier-Stokes (RANS) equations. Wilcox developed more complicated Favre-averaged Navier-Stokes equations (Wilcox, 1994). Using Reynolds averaging, we can derive a simple form of the averaged Navier-Stokes equations. The equation for conservation of mass stays the same. The equation for conservation of momentum has an additional Reynolds stress term: $-\overline{\rho u_i' \otimes u_j'}$. The equation for conservation of energy has an additional turbulence flux term.

Because many engineering problems are turbulent, turbulence modeling is crucial in CFD. The RANS model has both linear and nonlinear eddy viscosity models. The linear eddy viscosity models can be categorized based on the number of equations.

The first type of models is the zero-equation model. Zero-equation models do not introduce any new equations and simply use the existing variables. They define a relationship between the turbulent flux and the averaged value of variables. Prandtl proposed a mixing-length model. Van Driest (Van Driest, 1956) developed a viscous damping correction for the mixing-length model. The Cebeci-Smith model (Smith and Cebeci, 1967) refined the mixing-length concept. The Baldwin-Lomax model (Baldwin and Lomax, 1978) proposed a model that is suitable for high-speed flows with thin boundary layers. Another example is the Johnson-King model, which is suitable for turbulent boundary layer flows with strong adverse pressure gradients.

The second type of models is the one-equation model. One-equation models introduce one turbulent transported variable. We cite four models: Prandtl's one-equation model, the Spalart-Allmaras model (Spalart and Allmaras, 1992), the Baldwin-Barth model (Baldwin and Barth, 1990), and the Rahman-Siikonen-Agarwal Model (Rahman et al, 2011).

The third type of models is the two-equation model. Two-equation models introduce two turbulent transport equations. Two-equation models are among the most commonly used turbulence models. Most of the models introduce the turbulent kinetic energy. Here we cite the RNG $k-\varepsilon$ model (Yakhot et al, 1992), Wilcox's $k-\omega$ model (Wilcox, 1988), the SST $k-\omega$ model (Menter, 1994), and the Launder-Sharma model (Launder and Sharma, 1974).

Literature review

Many authors have proposed numerical methods to solve compressible RANS equations. Some numerical methods are FEMs. Our research is mainly based on the following works. El Kadri (El kadri, 1995) presented in his thesis presented a finite element model for two

dimensional flows. Soulaïmani and Ben Haj Ali (Soulaïmani and Ben Haj Ali, 2003) proposed a parallel-distributed computing-based approach for the solution of some multiphysics problems. They validated the method on the Agard 445.3 airfoil. Soulaïmani et al (Soulaïmani et al, 2004) proposed an efficient parallel-distributed methodology for solving multiphysics problems. They validated the results on Agard 445.6 airfoil. Ben Haj Ali and Soulaïmani (Ben Haj Ali and Soulaïmani, 2010) proposed a stabilized FEM for solving the compressible Navier-Stokes equations combined with the Spalart-Allmaras model. They validated the code on the 3D boundary layer over a flat plate and on the ONERA-M6 wing. Rebaine (Rebaine, 1997) proposed a numerical method for two-dimensional (2D) compressible laminar and turbulent flows. Rebaine and Soulaïmani (Rebaine and Soulaïmani, 2001) proposed an FEM for simulation of 2D internal compressible turbulent flows. They validated the method in 2D supersonic and thrust augmenting ejectors. Soulaïmani et al (Soulaïmani et al, 2002b) proposed a conservative finite element formulation for the coupled fluid/mesh interaction problem. Soulaïmani et al (Soulaïmani et al, 1994) proposed an FEM for simulation of 2D internal compressible turbulent flows. Soulaïmani and Fortin (Soulaïmani and Fortin, 1994) proposed a method to solve the Navier-Stokes and Euler equations in a conservative form by using the conservation variables.

Aside from FEM, some numerical methods involving FVM (Finite Volume Method) are also proposed for solving the Navier-Stokes equations. Caughey and Jameson (Caughey and Jameson, 2003) proposed an FVM for transonic flow calculation. They used this method on swept wings and wing-cylinder combinations. They showed that the FVM has the advantage of adaptability to treat a variety of complex configurations. Hafez (Hafez, 1995) proposed a cell-vertex finite volume formulation using local finite element approximations to solve inviscid and viscous compressible flow equations on unstructured grids. Feistauer et al (Feistauer et al, 1995) proposed a numerical modeling of inviscid as well as viscous gas flow. The method is based on an upwind flux vector splitting finite volume scheme on various types of unstructured grids.

Many authors have also proposed various techniques to solve the RANS equations. Pontaza and Reddy (Pontaza and Reddy, 2003) proposed a formulation of a spectral/hp algorithm to the numerical solution of the Navier-Stokes equations governing stationary incompressible and low-speed compressible flows. Rachowicz (Rachowicz, 2000) presented a technique of approximating boundary layers in viscous flow simulations with significantly stretched elements for compressible Navier-Stokes equations. Klaij et al (Klaij et al, 2006) presented a space-time discontinuous Galerkin element method for the compressible Navier-Stokes equations. They showed the space-time setting, derived the weak formulation, and discussed the choices for the numerical fluxes. Nazarov and Hoffman (Nazarov and Hoffman, 2010) presented an adaptive FEM for the compressible Euler equations. They used continuous piecewise linear approximation in space and time with componentwise weighted least-squares stabilization of convection terms and residual-based shock-capturing. Kellogg and Liu (Kellogg and Liu, 2000) developed a finite element formulation for the 2D nonlinear time-dependent compressible Navier-Stokes equations on a bounded domain. Cao (Cao, 2005) presented methods for improving the adaptive finite element simulation of compressible Navier-Stokes flow via a posteriori error estimate analysis. He used the moving space-time FEM to globally discretize the time-dependent Navier-Stokes equations on a series of adapted meshes. Banas (Banas, 2002) presented an implementation of the Newton-Krylov-Schwarz methodology for stabilized adaptive finite element approximations of compressible Navier-Stokes equations. Baruzzi et al (Baruzzi et al, 1995) presented numerical solutions for transonic inviscid and viscous laminar flows using higher-order dissipation. Martinez and Gartling (Martinez and Gartling, 2004) presented the derivation and justification for various low-speed approximations of the fully compressible Navier-Stokes equations. He and Li (He and Li, 2010) presented a fully discrete penalty FEM for the 2D time-dependent Navier-Stokes equations. Kweon (Kweon, 2000) presented a linearized steady-state compressible viscous Navier-Stokes system with an inflow boundary condition. Shan and Hou (Shan and Hou, 2009) proposed a fully discrete stabilized FEM based on two local Gauss integrations for the 2D time-dependent Navier-Stokes equations. Compared with other stabilized methods, this approach does not require specification of a stabilization parameter or calculation of higher-order derivatives. Burman (Burman, 2000) proposed

adaptive streamline diffusion FEMs with error control for compressible flow in one, two, and three dimensions. Karagiozis et al (Karagiozis et al, 2009) proposed a numerical method to solve the compressible Navier-Stokes equations around objects of arbitrary shape using Cartesian grids. This method is suitable for compressible flows without shocks. Lomtev and Karniadakis (Lomtev and Karniadakis, 1999) presented the foundations of a new discontinuous Galerkin method for simulating compressible viscous flows with shocks on standard unstructured grids. This method is based on a discontinuous Galerkin formulation for both advective and diffusive contributions. Kirk and Carey (Kirk and Carey, 2008) applied the streamline upwind/Petrov-Galerkin (SUPG) method to the unsteady compressible Navier-Stokes equations in conservation-variable form. They used a modified approach for interpolating the inviscid flux terms in the SUPG finite element formulation for the spatial discretization. They used second-order accurate time discretization. Li et al (Li et al, 1998) developed finite element-based methodology for the numerical simulation of the compressible Navier-Stokes equations on unstructured triangular meshes. They used a Galerkin finite-element discretization in space and an explicit Runge-Kutta multistage integration in time. Nigro et al (Nigro et al, 1998) presented the implementation of a local physics preconditioning mass matrix for a unified approach of 3D compressible and incompressible Navier-Stokes equations using an SUPG finite element formulation and GMRES implicit solver. Erwin et al (Erwin et al, 2013) developed a high-order flow solver for compressible flows using a stabilized finite element approach. They used streamline/upwind Petrov-Galerkin discretization for the Navier-Stokes equations, and they used a fully implicit methodology for advancing the solution at each time step.

To test our code, there are many test cases that we can use as comparisons. Liu and Li (Liu and Li, 2001) developed an unstructured algorithm for the computation of compressible RANS equations. The turbulence models they chose were the Baldwin-Lomax model and the Baldwin-Barth model. They validated the results on a flat plate, an RAE2822 airfoil, and an NACA0012 airfoil. Kersken et al (Kersken et al, 2012) proposed a computational method for solving the compressible RANS equations. They validated the method on the benchmark problem Stardard Configuration 10 and a modern ultra-high bypass ration fan. Bassi and

Rebay (Bassi and Rebay, 2014) proposed a high-order accurate discontinuous FEM for the numerical solution of the compressible Navier-Stokes equations. They showed that the method is robust in all test cases.

The Spalart-Allmaras model uses only one equation to model turbulent viscosity. The equation has one nonlinear diffusion term, one destruction term, and one production term. The Boussinesq hypothesis is employed in the model; the Reynolds stress is linearly proportional to the mean stain rates. It has advantages for applications involving wall-bounded flows and boundary layers subjected to adverse pressure gradients.

Numerous research studies have demonstrated that the Spalart-Allmaras model performs well on the external flow. Yan et al (Yan et al, 2011) validated the Spalart-Allmaras model for turbulent flow past a square cylinder. They obtained results that reasonably agree with the existing experimental results and discovered that the fluctuating pressure is more sensitive to the change in the afterbody shape. Paciorri et al (Paciorri et al, 1998) validated the Spalart-Allmaras turbulent model for hypersonic flows. They discovered that for flows involving laminar separation and turbulent reattachment, the model obtained results which agreed with the experimental data. They also demonstrated that the model correctly predicted the turbulent separation. Geng et al (Geng et al, 2011) validated four turbulence models, one of which was the Spalart-Allmaras model, on 2D supersonic expansion-compression and hypersonic flows. They obtained results that matched the experimental data and recommended compressibility for hypersonic flows at a high angle of attack. Roy and Blottner (Roy and Blottner, 2003) validated the model on hypersonic transitional flows. They presented the documentation procedure, numerical accuracy, model sensitivity, and model validation. Coratekin et al (Coratekin et al, 2004) evaluated four upwind schemes and four turbulence models, one of which was the Spalart-Allmaras model, in hypersonic flows. Their results showed that the $k-\omega$ model provided the best prediction in cases of separation. Kong et al (Kong et al, 2012) conducted simulations of crossing shock wave/turbulent boundary layer interaction using three turbulence models: Wilcox's $k-\omega$ model, the Spalart-Turbulence model, and the SST model. Their results showed that the SST model achieved

better results in the pressure and velocity vector, and all three models over-predicted the heat transfer coefficient. Nordanger et al (Nordanger et al, 2015) tested three solvers on a fixed NACA0012 airfoil at a high Reynolds number, one of which used the coupled Navier-Stokes equations with the Spalart-Allmaras turbulence model. They used the three solvers for flows over a NACA0012 airfoil at Reynolds number 3×10^6 at four different angles of attack. They noticed that beyond the angle of attack of 15°, it is difficult to predict lift and drag when the flow enters the stall regime. They also noticed that increasing the element order from 1 to 2 will give better approximation of lift, drag, and pressure coefficients for the Spalart-Allmaras model.

Several research studies have simulated the turbulent flows using averaged Navier-Stokes equations and the Spalart-Allmaras model. Soulaïmani (Soulaïmani, 2001) presented a stabilized finite element formulation for solving compressible flows. He presented three stabilization techniques: the SUPG formulation, the DG method, and the EBS method. Wervaecke et al (Wervaecke et al, 2012) presented a RANS-based Spalart-Allmaras SUPG formulation for 2D and 3D turbulent compressible flows. They tested this model on NACA0012 airfoil, RAE2822 airfoil, S809 airfoil, 3D ONERA M6 wing, and 3D turbulent flat plate.

Some authors have proposed methods to modify the Spalart-Allmaras model to achieve better performance in different cases. Liu et al (Liu et al, 2011) modified the Spalart-Allmaras model with relative helicity density to improve the predictive accuracy for corner separation flow. Yan et al (Yan et al, 2014) conducted a simulation of S825 airfoil using the Spalart-Allmaras model and compared the results to experimental data. They proposed using different values of parameter C_{b1} for the non-separating region and the separating region. Aupoix and Spalart (Aupoix and Spalart, 2003) introduced two extensions to the Spalart-Allmaras model to account for wall roughness. Developed independently by Boeing and ONERA, the two extensions assume a non-zero-eddy viscosity at the wall and change the definition of the distance d. Rung et al (Rung et al, 2003) proposed changing constant C_{b1} to a function of the strain rate for nonequilibrium flows. Deck et al (Deck et al, 2002) presented

an extension of the Spalart-Allmaras model to compressible supersonic flows. This model achieved good results for simulations on a missile. De Santis (De Santis, 2014) developed a high-order residual distribution scheme for compressible RANS equations. He changed the definition of the working variable to deactivate the production and destruction terms of the turbulence model equations when turbulent viscosity is negative. He also eliminated the diffusion contribution in the source term when the turbulent viscosity is negative. He tested the model in several 2D and 3D cases. Ishiko et al (Ishiko et al, 2014) proposed an extended nonlinear algebraic constitutive relation for the Reynolds stress tensor and modifications to improve predictions for the free jet-based Spalart-Allmaras model. Khurram et al (Khurram et al, 2012) presented a multiscale FEM with the Spalart-Allmaras turbulence model for 3D detached-eddy simulation. They decomposed the scalar field into coarse scales and fine scales. They showed that this method provided effective stabilization in turbulent computations where reaction-dominated effects strongly influence the boundary layer prediction. Lorin et al (Lorin et al, 2007) proposed a stable numerical method preserving the positivity of the turbulent viscosity in the Spalart-Allmaras model. They validated the method on the 3D boundary layer over a flat plate.

As with most numerical methods, an appropriate stabilization is important for obtaining optimal performance. Soulaïmani and Fortin (Soulaïmani and Fortin, 1994) proposed a definition of the stabilization matrix τ for several dimensions. They also proposed an artificial viscosity for shock capturing. Tezduyar and Senga (Tezduyar and Senga, 2006) proposed a definition of the SUPG stabilization matrix τ and a shock-capturing operator. Wong et al (2000) presented a stabilized finite element algorithm and proposed a definition of a stabilization matrix τ. They showed that this new matrix represents a dramatic improvement over more standard choices. Wang et al (Wang et al, 2014) presented high-order discontinuous Galerkin and SUPG methods for solutions of 3D viscous flows and 2D turbulent flows. They also proposed a definition of the matrix τ.

Objective of thesis

The objective of this thesis is to modify an existing in-house code (Ben Haj Ali and Soulaïmani, 2010) to enhance its capability to solve 3D compressible turbulent flows. The original code is limited to one type of element, and we expand its capacity to allow the use of several types of elements in a hybrid mesh.

We then must validate our code. We used our code to simulate the turbulent flows over a 3D wing model and a fuselage. The wing model we chose for the validation was extruded from NACA0012. The fuselage we chose was the DLR F11 model. Currently there are many references available for these two cases, and we compared our results with other numerical and experimental results.

Plan of thesis

Chapter 1 introduces the governing equations and the use of FEM. The Navier-Stokes equations are presented: the equations describing conservation of mass, conservation of momentum, and conservation of energy. The weak form of the Navier-Stokes equations is then developed. We show how we formulated the spatial and time discretizations, computed the elemental matrix and elemental residual for each type of element, and applied the initial and boundary conditions. This chapter also shows the stabilization and shock-capturing techniques. Finally, we use the Newton-Raphson method and the GMRES algorithm to solve the system of equations.

Chapter 2 describes the four elements used and the numerical integration. The four elements are hexahedrons, tetras, prisms, and pyramids. This chapter also shows how we obtained the shape functions and integration points for each type of element.

To facilitate the programming of the elemental matrix and residual for different types of elements, we used object-oriented programming in Fortran 2003. Chapter 3 presents the use

of the object-oriented programming method and shows how we realized this concept in our code. We then make a comparison with the non-object-oriented programming.

Chapter 4 discusses the results we obtained after running our code on several test cases. We also make a comparison with other numerical and experimental results.

CHAPTER 1

GOVERNING EQUATIONS

1.1 Introduction

Our goal is to establish a finite element model to simulate external compressible flows. The internal properties of the flow are already known: the dynamic viscosity μ, the thermal conductivity k, and the heat capacities C_p and C_v. We describe the state of the flow using the density ρ, the velocity vector **U**, the pressure p, and the temperature T. These unknown variables are then solved using the Navier-Stokes equations, which describe the conservation of mass, momentum, and energy.

The mass conservation law is expressed in the continuity equation. For any flow of mass m,

$$\frac{d}{dt}(m) = 0 \tag{1.1}$$

The momentum conservation law is expressed by Newton's second law:

$$\frac{d}{dt}(m\mathbf{u}) = \sum \mathbf{F} \tag{1.2}$$

The left side denotes the time rate of momentum per unit mass. The right side denotes the sum of applied forces, including body forces and surface forces.

The idea of energy conservation is expressed in the equation describing conservation of energy, which shows that the time rate of change for the total energy of a control volume equals the sum of the rate of work performed by the surface force, the body force, the rate of heat transfer, and the rate of heat source in the volume:

$$\frac{dE}{dt} = \mathbf{f}_s \cdot \mathbf{U} + \mathbf{f}_v \cdot \mathbf{U} - \nabla \cdot \mathbf{q} + r \tag{1.3}$$

Another necessary equation is the equation of state. For gas, the temperature, density and pressure are not independent quantities but are connected: $F(p, \rho, T) = 0$. For Mach numbers smaller than 5, we can use the perfect gas law.

To solve this system of partial differential equations, we also add the proper initial conditions and boundary conditions. This system of equations can model laminar and turbulent as well as compressible and incompressible flows. This system of equations is too complex to solve analytically and usually requires the use of numerical methods. To solve this system of equations numerically, we must choose the proper dependent variables and write the system of equations in a proper form to facilitate programming. The Navier-Stokes equations can be written in several forms. In the following sections, we will present in detail the form we used and how to implement the finite element method.

1.2 Conservative form in conservative variables

There are three general forms that we can use: the conservative form, the nonconservative form, and the conservative form with conservative variables. We write each of the Navier-Stokes equations in its conservative form with conservative variables (El Kadri, 1995).

Let ρ be the density and $\mathbf{u}$ be the vector of velocities, then we define the momentum of unit mass:

$$\mathbf{U} = \rho \mathbf{u} \tag{1.4}$$

Let e be the total energy and i be the internal energy, then we define the total energy of the unit mass:

$$E = \rho e \tag{1.5}$$

where $e = i + \frac{1}{2}|\mathbf{u}|^2$.

We can now write the system of equations in its conservative form with conservative variables.

The continuity equation reads

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\mathbf{U}) = 0 \tag{1.6}$$

The momentum equations are

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot (\mathbf{U} \otimes \frac{\mathbf{U}}{\rho}) + \nabla p - \nabla \cdot \boldsymbol{\sigma} = \rho \mathbf{f}_v \tag{1.7}$$

The energy equation is

$$\frac{\partial E}{\partial t} + \nabla \cdot \left[(E + p) \frac{\mathbf{U}}{\rho} \right] = \nabla \cdot (\boldsymbol{\sigma} \cdot \frac{\mathbf{U}}{\rho}) - \nabla \cdot \mathbf{q} + \mathbf{f}_v \cdot \mathbf{U} + r \tag{1.8}$$

The stress tensor is defined as

$$\boldsymbol{\sigma}(\mathbf{u}) = \lambda (\nabla \cdot \mathbf{u}) + 2\mu \mathbf{D} \tag{1.9}$$

where the tensor $\mathbf{D}$ has components

$$D_{ij} = D_{ij}(\mathbf{u}) = \frac{1}{2}(u_{i,j} + u_{j,i}) \tag{1.10}$$

μ and λ are Lamé constants, and they can be connected by Stokes's law:

$$2\mu + 3\lambda = 0 \tag{1.11}$$

For air, the heat transfer can be obtained by Fourier's law:

$$\mathbf{q} = -k\nabla T \tag{1.12}$$

Along with the ideal gas law, the equations can be solved.

1.3 Dimensionless form

To obtain better insight into the problem, we apply nondimensionalization to the system of equations. The geometry stays the same but is scaled. In addition, the scaled system has the same physical characteristics as the original one. In this way, we can find the solution for problems with the same boundary conditions but with different scales in the geometry.

We convert variables to their dimensionless form using scales denoted by index *r*:

$$\mathbf{u}^* = \frac{\mathbf{u}}{|\mathbf{u}_r|} \qquad \rho^* = \frac{\rho}{\rho_r} \qquad p^* = \frac{p}{\rho_r |\mathbf{u}_r|^2} \tag{1.13}$$

$$i^* = \frac{i}{|\mathbf{u}_r|^2} \qquad \mu^* = \frac{\mu}{\mu_r} \qquad t^* = \frac{t}{L_r / |\mathbf{u}_r|}$$

$$T^* = \frac{T}{|\mathbf{u}_r|^2 / C_v} \qquad x^* = \frac{x}{L_r}$$

To simplify the notation, we omit the star. The unit momentum, stress tensor, heat flux, total energy per unit mass, and pressure are defined respectively as:

$$\mathbf{U} = \rho\mathbf{u} \tag{1.14}$$

$$\boldsymbol{\sigma}(\mathbf{u}) = \frac{1}{\mathrm{Re}}\left[\nabla\mathbf{u} + (\nabla\mathbf{u})^T\right] - \frac{2}{3}\frac{1}{\mathrm{Re}}(\nabla\cdot\mathbf{u})\mathbf{I} \tag{1.15}$$

$$\mathbf{q} = -\frac{\gamma}{\mathrm{Re}\,\mathrm{Pr}}\nabla T \tag{1.16}$$

$$E = \rho e = \rho(T + \frac{|\mathbf{u}|^2}{2}) \tag{1.17}$$

$$p = (\gamma - 1)\rho T \tag{1.18}$$

When we substitute equation (1.14) into equation (1.15), we can write the stress tensor:

$$\boldsymbol{\sigma}(\mathbf{u}) = \frac{1}{\rho}\boldsymbol{\sigma}(\mathbf{u}) - \frac{1}{\rho^2\,\mathrm{Re}}\left\{\left[\mathbf{U}^T\cdot(\nabla\rho) + (\nabla\rho)^T\cdot\mathbf{U}\right] - \frac{2}{3}(\mathbf{U}\cdot(\nabla\rho)^T)\mathbf{I}\right\} \tag{1.19}$$

When we substitute equations (1.14) and (1.17) into equation (1.16), we can write the heat flux as

$$\mathbf{q} = -\frac{\gamma}{\mathrm{Re}\,\mathrm{Pr}}\nabla(\frac{E}{\rho} - \frac{|\mathbf{U}|^2}{2\rho^2}) \tag{1.20}$$

When we substitute equations (1.14) and (1.17) into equation (1.18), we can write the pressure as

$$p = (\gamma - 1)\left[E - \frac{|\mathbf{U}|^2}{2\rho}\right] \tag{1.21}$$

The dimensionless form of the set of conservation equations is

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\mathbf{U}) = 0 \tag{1.22}$$

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot (\mathbf{U} \otimes \mathbf{u}) + \nabla p - \nabla \cdot \boldsymbol{\sigma} = \mathbf{f}_v \tag{1.23}$$

$$\frac{\partial E}{\partial t} + \nabla \cdot \left[(E + p)\mathbf{u} \right] = \nabla \cdot (\boldsymbol{\sigma}.\mathbf{u}) - \nabla \cdot \mathbf{q} + \mathbf{f}_v \cdot \mathbf{U} \tag{1.24}$$

The dimensionless form is identical to the original one, but in the second system we introduce two similarity parameters: Reynolds number $\mathrm{Re} = \frac{\rho_r |\mathbf{u}_r| L_r}{\mu_r}$ and Prandtl number $\mathrm{Pr} = \frac{\mu_r Cp_r}{k_r}$.

The Reynolds number is the ratio of inertial forces to viscous forces. A small Reynolds number indicates that the flow is dominated by diffusion. Alternatively, a large Reynolds number means that the flow is dominated by convection.

The Prandtl number is the ratio of kinematic viscosity to thermal diffusivity. $Pr \ll 1$ indicates that the flow has dominantly thermal diffusivity. $Pr \gg 1$ indicates that the flow has dominantly kinematic viscosity. The variation of the Prandtl number is relatively small, and we use $Pr = 0.72$.

We can also define another similarity parameter, which is the product of *Re* and *Pr*, known as the Peclet number:

$$\mathrm{P}e = \mathrm{Re}\,\mathrm{Pr} = \frac{\rho_r \mathbf{u}_r \mathbf{L}_r}{\mu_r} \frac{\mu_r Cp_r}{k_r} = \frac{\mathbf{u}_r \boldsymbol{L}_r}{v} \tag{1.25}$$

v is the thermal diffusivity. The Peclet number is the ratio of the rate of advective transport of heat by the flow to the diffusive rate of heat. A large Pe means that the energy of the flow is dominated by the advective transport of heat.

1.4 Vectorial form

The system of equations can be written in vectors:

$$\mathbf{V}_{,t} + \mathbf{F}_{i,i}^{adv} = \mathbf{F}_{i,i}^{diff} + \mathbf{F}^{S} \tag{1.26}$$

V represents the vector of conservative variables:

$$\mathbf{V} = \begin{Bmatrix} \rho \\ \mathbf{U} \\ E \end{Bmatrix} \tag{1.27}$$

$\mathbf{F}_i^{adv}$ is the advection flux:

$$\mathbf{F}_i^{adv} = \begin{Bmatrix} \mathbf{U}_i \\ \mathbf{U}\dfrac{\mathbf{U}_i}{\rho} + p\delta_{ij} \\ (E+p)\dfrac{\mathbf{U}_i}{\rho} \end{Bmatrix} \tag{1.28}$$

$\mathbf{F}_i^{diff}$ is the diffusion flux:

$$\mathbf{F}_i^{diff} = \begin{Bmatrix} 0 \\ (\sigma_{i1}, \sigma_{i2}, \sigma_{i3})^T \\ (\sigma_{i1}, \sigma_{i2}, \sigma_{i3})^T \cdot \mathbf{u} - kT_{,i} \end{Bmatrix} \tag{1.29}$$

$\mathbf{F}^s$ is the source term:

$$\mathbf{F}^s = \begin{Bmatrix} 0 \\ \rho \mathbf{f}_v \\ \mathbf{f}_v \cdot \mathbf{U} \end{Bmatrix} \tag{1.30}$$

1.5 Weak formulation

In order to use FEM to solve this system of equations, we must write it in the weak form. We first multiply the equations with a test function **W** and integrate:

$$\int_\Omega \mathbf{W}.(\mathbf{V}_{,t} + \mathbf{F}_{i,i}^{adv} - \mathbf{F}_{i,i}^{diff} - \mathbf{F}^S) d\Omega = \mathbf{0} \tag{1.31}$$

The weighted residuals are realized by the Galerkin method. We use the Gauss theorem on the diffusion term and convert the strong form of the system of equations to its weak form:

$$\sum_e \int_{\Omega_e} \mathbf{W}(\mathbf{V}_{,t} + \mathbf{F}_{i,i}^{adv} - \mathbf{F}^S) d\Omega + \sum_e \int_{\Omega_e} \mathbf{W}_{,i} \mathbf{F}_i^{diff} d\Omega - \int_\Gamma \mathbf{W} \mathbf{F}_i^{diff} \mathbf{n}_i d\Gamma = \mathbf{0} \tag{1.32}$$

The domain Ω is divided to subdomains Ω_e, and **n** is the unit outward-pointing normal vector to Γ. The surface integration in the weak allows us to easily apply the boundary conditions.

1.6 Spatial discretization

The domain is meshed using four kinds of elements: 4-node tetras, 8-node hexahedrons, 6-node prisms, and 5-node pyramids.

The integration of any function f over the domain Ω can be written as

$$\int_{\Omega} f(x,y,z)d\Omega = \sum_{e} \int_{\Omega_e} f(x,y,z)d\Omega_e \tag{1.33}$$

The conservative variables **V** are approximated by the products of the shape functions and coefficients, which are evaluated at the nodes (Dhatt and Touzot, 1981):

$$\mathbf{V}(i) = \sum_{k=1}^{n} N_k \mathbf{V}_k(i) = \langle N_1, N_2, \ldots N_n \rangle \begin{Bmatrix} \mathbf{V}(i)_1 \\ \mathbf{V}(i)_2 \\ \cdot \\ \cdot \\ \cdot \\ \mathbf{V}(i)_n \end{Bmatrix} \tag{1.34}$$

where $\mathbf{V}(1) = \rho, \mathbf{V}(2) = \rho u, \mathbf{V}(3) = \rho v, \mathbf{V}(4) = \rho w, \mathbf{V}(5) = \rho e$. The number *n* of shape functions per element is also the number of nodes per element. By the Galerkin method, we also use the same shape functions for the test functions.

The elements forming the mesh are not identical in shape or size. It is more convenient to transform the domain Ω_e to the reference domain Ω_r (Dhatt and Touzot, 1981). The reference coordinates are denoted by $\xi_i(\xi,\eta,\zeta)$. The derivative of the function $f(x,y,z)$ with respect to ξ_i is

$$\frac{\partial f(x,y,z)}{\partial \xi_i} = \left\langle \frac{\partial N_1}{\partial \xi_i}, \frac{\partial N_2}{\partial \xi_i}, \cdots \frac{\partial N_3}{\partial \xi_i} \right\rangle \begin{Bmatrix} f_1 \\ f_2 \\ . \\ . \\ . \\ f_3 \end{Bmatrix} \tag{1.35}$$

The integration of the function $f(x,y,z)$ over the domain Ω_e can now be written as

$$\int_{\Omega_e} f(x,y,z) d\Omega_e = \int_{\Omega_r} f(\xi,\eta,\zeta) \det(\mathbf{J}) d\Omega_r \tag{1.36}$$

where $\mathbf{J}$ is the Jacobian matrix of the transformation:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta} \\ \frac{\partial x}{\partial \zeta} & \frac{\partial y}{\partial \zeta} & \frac{\partial z}{\partial \zeta} \end{bmatrix} \tag{1.37}$$

1.7 Time discretization

To discretize the time-derivative term, we use the first-order forward finite difference method:

$$\frac{\partial \mathbf{V}(t)}{\partial t} = \frac{\mathbf{V}(t+\Delta t) - \mathbf{V}(t)}{\Delta t} + \mathrm{O}(\Delta t) \tag{1.38}$$

Δt can be calculated using $\Delta t = CFL \frac{h^e}{(\|\mathbf{u}\| + c)^e}$, where h^e is the minimum length between the nodes of an element and c is the speed of sound.

1.8 SUPG stabilization

The use of the weak form by the Galerkin method will produce some numerical instability, especially if the convection flux is dominant. The stabilization method used here is SUPG. This method is popular in solving transport equations. It introduces a supplementary term to the standard Galerkin method and reinforces the stability inside the element (Soulaïmani and Fortin, 1994). This additional term adds artificial diffusion in the flow direction:

$$\sum_e \int_{\Omega^e} \mathbf{A}_i \mathbf{W}_i \boldsymbol{\tau} (\mathbf{V}_{,t} + \mathbf{A}_i \frac{\partial \mathbf{V}}{\partial x_i} - \left(\mathbf{K}_{ij} \mathbf{V}_{,j}\right)_{,i} - \mathbf{F}^s) d\Omega \tag{1.39}$$

The τ matrix has the dimension of time and depends on the element geometry. There are various definitions of this matrix. We need to choose a τ matrix without introducing excessive diffusion to the solution on the boundary layer and across the shock wave. Soulaïmani and Fortin (Soulaïmani and Fortin, 1994) proposed a τ matrix:

$$\tau = \left(\sum_i \left\| c_{ij} \mathbf{A}_j \right\| \right)^{-1} \varsigma(R_h) \tag{1.40}$$

where $c_{ij} = \frac{\partial \zeta_i}{\partial x_j}$ is the transformation matrix from the actual element to the reference element. $\varsigma(R_h)$ is defined as

$$\varsigma(R_h) = \min\left(\frac{R_h}{3}, 1 \right) \tag{1.41}$$

where $R_h = \frac{\|\mathbf{u}\| h}{2\mu}$ is the local Reynolds number, μ is the dynamic viscosity, and h is the characteristic length of the element.

This method will produce an artificial viscosity with order $\frac{\|\mathbf{u}\| h}{2}$, so there will be a large numerical diffusion, especially in the boundary layer. Ben Haj Ali and Soulaïmani (Ben Haj Ali, 2008; Ben Haj Ali and Soulaïmani 2010) proposed a new definition of the τ matrix to accommodate stretched elements. This diagonal matrix is defined as

$$\tau = \frac{h}{2\lambda}\mathbf{I} \tag{1.42}$$

where $\mathbf{I}$ is the identity matrix and λ is the largest eigenvalue of the advection matrix: $\lambda = \|\mathbf{u}\| + c$. h is a characteristic length of the element defined as $h = \min\|x_i - x_j\|$, where i and j are the nodes of the element.

1.9 Shock capturing

Along with the stabilization term, we add another term to capture the oscillations caused by the large gradients. This additional numeric dissipation across the shock waves will further add numerical stability:

$$\sum_e \int_{\Omega_e} [\mu_c] \nabla \mathbf{W} \nabla \mathbf{V} d\Omega \tag{1.43}$$

There are various definitions of the shock capturing matrix. Soulaïmani and Fortin (Soulaïmani and Fortin, 1994) proposed the following definition of the shock capturing operator:

$$\mu_c = \frac{C_k h \min\left(\|\boldsymbol{\tau}\mathbf{R}(\mathbf{V})\|, \|\mathbf{u}\|\right)}{2} \tag{1.44}$$

Ben Haj Ali and Soulaïmani (Ben Haj Ali, 2008; Ben Haj Ali and Soulaïmani, 2010) proposed the following diagonal matrix as the shock capturing operator:

$$\mu_{c,k}^e = \frac{h\lambda_k}{2} C(\varsigma^e + \varepsilon) \tag{1.45}$$

where $\varepsilon = 0.05$ and C=1.0.

$$\lambda_k = \begin{cases} \|\mathbf{u}\| + c & \text{for } k = 1 \\ 0 & \text{for } k = 2,3,4,5 \end{cases} \tag{1.46}$$

We also calculate the shock capturing viscosity:

$$\mu_c = \frac{\rho h(\|\mathbf{u}\| + c)}{2} C(\varsigma^e + \varepsilon) \tag{1.47}$$

ς^e is the sensor of pressure variation proposed in (Jameson and Mavriplis, 1986). For an element having *Ne* nodes, the sensor of shock capturing for node *i* is calculated as

$$\varsigma^i = \sum_{j=1}^{Ne} \frac{\left|p_i - p_j\right|}{\frac{p_i + p_j}{2}}, i \neq j \tag{1.48}$$

Then we can get ς^e :

$$\varsigma^e = \frac{1}{Ne} \sum_{i=1}^{Ne} \varsigma^i \tag{1.49}$$

1.10 Initial conditions and boundary conditions

Because the system of equations evolves with time, we should specify the initial conditions:

$$\rho(x,y,z,0)=\rho_0, \mathbf{U}(x,y,z,0)=\mathbf{U}_0, E(x,y,z,0)=E_0 \tag{1.50}$$

In the following parts of this section, we discuss the solid wall and the far-field boundary conditions.

On the solid wall, we impose a non-slip boundary condition for the momentum:

$$\mathbf{U}=\mathbf{0} \tag{1.51}$$

We either specify an adiabatic wall:

$$\mathbf{q}\cdot\mathbf{n}=0 \tag{1.52}$$

or we impose the energy:

$$E=\rho T_w=\rho\left(\frac{1}{\gamma(\gamma-1)M_\infty^2}\right)\left(1+\frac{(\gamma-1)}{2}M_\infty^2\right) \tag{1.53}$$

where T_w is the stagnation temperature.

For the far-field boundary, we treat the outflow and inflow differently. By the perfect gas law, $p=(\gamma-1)\rho T$. If we specify both density and energy in the whole boundary, the pressure will be fixed. Therefore, we should specify just the density or just the energy on the boundaries across which the flow enters.

Let ρ_∞, $\mathbf{U}_\infty$, and E_∞ be the values of the conservative variables in the far field. The far-field boundary is denoted by Γ_∞, which has the normal vector $\mathbf{n}$. Γ_∞ is divided into two parts: the outflow Γ_∞^+ and the inflow Γ_∞^-. For Γ_∞^+, $\mathbf{u}.\mathbf{n} > 0$. For Γ_∞^-, $\mathbf{u}.\mathbf{n} < 0$.

For the inflow Γ_∞^-, we impose the conditions

$$\begin{aligned} &\mathbf{U} = \mathbf{U}_\infty \\ &\rho = \rho_\infty = 1 \\ &E = E_\infty = \frac{1}{2} + \frac{1}{\gamma(\gamma-1)M_\infty^2} \end{aligned} \tag{1.54}$$

For boundary Γ_∞^+, we specify impose density $\rho = \rho_\infty = 1$.

Since $\mathbf{F}_i^{diff}\mathbf{n}_i = \mathbf{0}$ on boundary Γ the integration of the diffusion flux $\int_\Gamma \mathbf{W}\mathbf{F}_i^{diff}\mathbf{n}_i d\Gamma$ is not computed.

Let $\mathbf{F}_\infty^{adv}$ be the advection flux on the free boundary. By integrating by parts the advection term (Ben Haj Ali, 2008; Ben Haj Ali and Soulaïmani, 2010),

$$\int_\Omega \mathbf{W}\mathbf{F}_{i,i}^{adv} d\Omega = -\int_\Omega \mathbf{W}_i\mathbf{F}_i^{adv} d\Omega + \int_{\Gamma_s} \mathbf{W}\mathbf{F}_i^{adv}\mathbf{n}_i d\Gamma + \int_{\Gamma_\infty} \mathbf{W}\mathbf{F}_i^{adv}\mathbf{n}_i d\Gamma \tag{1.55}$$

We integrate again $\int_\Omega \mathbf{W}_i.\mathbf{F}_i^{adv} d\Omega$ by parts and replace $\mathbf{F}_i^{adv}.\mathbf{n}_i$ at the far field with $\mathbf{F}_\infty^{adv}$:

$$\int_\Omega \mathbf{W}_i\mathbf{F}_i^{adv} d\Omega = -\int_\Omega \mathbf{W}\mathbf{F}_{i,i}^{adv} d\Omega + \int_{\Gamma_s} \mathbf{W}\mathbf{F}_i^{adv}\mathbf{n}_i d\Gamma + \int_{\Gamma_\infty} \mathbf{W}\mathbf{F}_\infty^{adv} d\Gamma \tag{1.56}$$

By summing the two equations, we obtain

$$\int_{\Omega} \mathbf{W}\mathbf{F}_{i,i}^{adv} d\Omega = \int_{\Omega} \mathbf{W}\mathbf{F}_{i,i}^{adv} d\Omega + \int_{\Gamma_\infty} \mathbf{W}(\mathbf{F}_i^{adv}\mathbf{n}_i - \mathbf{F}_\infty^{adv}) d\Gamma \tag{1.57}$$

The Jacobian of the flux $\mathbf{F}_i^{adv}$ is

$$\mathbf{A}_i = \frac{\partial \mathbf{F}_i^{adv}}{\partial \mathbf{V}} \tag{1.58}$$

Let $\mathbf{A}_n$ be the product of $\mathbf{A}_i$ and $\mathbf{n}_i$:

$$\mathbf{A}_n = \sum_{i=1}^{3} \mathbf{A}_i \mathbf{n}_i \tag{1.59}$$

Let $\mathbf{S}_i$ be the i^{th} eigenvector of $\mathbf{A}_n$ and Λ_{ii} be the i^{th} eigenvalue of $\mathbf{A}_n$; because $\mathbf{\Lambda}$ is diagonal,

$$\mathbf{A}_n^{-} = \mathbf{S}\mathbf{\Lambda}^{-}\mathbf{S}^{-1} \tag{1.60}$$

We also define

$$\mathbf{A}_n^{+} = \mathbf{S}\mathbf{\Lambda}^{+}\mathbf{S}^{-1} \tag{1.61}$$

where $\Lambda_{ii}^{-} = \min(0, \lambda_i)$ is the i^{th} eigenvalue of $\mathbf{A}_n^{-}$ and $\Lambda_{ii}^{+} = \max(0, \lambda_i)$ is the i^{th} eigenvalue of $\mathbf{A}_n^{+}$.

So $\mathbf{A}_n = \mathbf{A}_n^{-} + \mathbf{A}_n^{+}$, $\mathbf{F}_\infty^{adv} = \mathbf{A}_n^{-}\mathbf{V}_\infty + \mathbf{A}_n^{+}\mathbf{V}$, $\mathbf{F}_i^{adv} n_i = \mathbf{A}_n^{-}\mathbf{V} + \mathbf{A}_n^{+}\mathbf{V}$. Here we use the Steger and Warming flux vector splitting method (Steger and Warming, 1981; Warming et al, 1975) to calculate $\mathbf{\Lambda}^{-}$, $\mathbf{S}$ and $\mathbf{S}^{-1}$.

Finally, we replace $\int_{\Gamma_\infty} \mathbf{W}(\mathbf{F}_i^{adv}\mathbf{n}_i - \mathbf{F}_\infty^{adv})d\Gamma$ with $\int_{\Gamma_\infty} \mathbf{W}\mathbf{A}_n^{-}(\mathbf{V}-\mathbf{V}_\infty)d\Gamma$.

1.11 Elemental matrices

We have showed the weak form of the Navier-Stokes equations. We now use the conservative form with conservative variables. When we add the SUPG stabilization term and the shock capturing term, the stabilized weak form of the Navier-Stokes is

$$\sum_e \int_{\Omega_e} \left[\mathbf{W}(\mathbf{V}_{,t} + \mathbf{F}_{i,i}^{adv} - \mathbf{F}^S) + \mathbf{W}_{,i}\mathbf{F}_i^{diff} \right] d\Omega \tag{1.62}$$

$$-\int_{\Gamma} \mathbf{W}\mathbf{F}_i^{diff}\mathbf{n}_i d\Gamma - \int_{\Gamma_\infty} \mathbf{W}\mathbf{A}_n^{-}(\mathbf{V}-\mathbf{V}_\infty)d\Gamma +$$

$$\sum_e \int_{\Omega_e} \nabla\mathbf{W}[\mu_c]\nabla\mathbf{V}d\Omega + \sum_e \int_{\Omega_e} \mathbf{A}_i\mathbf{W}_{,i}\boldsymbol{\tau}\mathbf{R}(\mathbf{V})d\Omega = \mathbf{0}$$

The diffusion flux $\mathbf{F}_i^{diff}$ can be written as

$$\mathbf{F}_i^{diff} = \mathbf{K}_{ij}\mathbf{V}_{,j} \tag{1.63}$$

We can obtain $\mathbf{K}_{ij}$ from (Shakib, 1989).

The advection flux can be written as

$$\mathbf{F}_{i,i}^{adv} = \frac{\partial \mathbf{F}_i}{\partial \mathbf{V}}\mathbf{V}_{,i} = \mathbf{A}_i\mathbf{V}_{,i} \tag{1.64}$$

We can obtain $\mathbf{A}_i$ from Toro (1999).

To solve equation (1.62) numerically, we write the system of equations in matrix form:

$$\mathbf{M}\left\{\dot{\mathbf{V}}^h\right\}+\mathbf{K}\left\{\mathbf{V}^h\right\}=\{\mathbf{F}\} \tag{1.65}$$

with

$$\mathbf{M}=\overset{m}{\underset{i=1}{\mathcal{A}}}\mathbf{M}^e,\ \mathbf{K}=\overset{m}{\underset{i=1}{\mathcal{A}}}\mathbf{K}^e,\ \mathbf{F}=\overset{m}{\underset{i=1}{\mathcal{A}}}\mathbf{F}^e \tag{1.66}$$

where $\left\{\mathbf{V}^h\right\}$ is the vector of all nodal unknowns, m is the number of elements, and $\overset{m}{\underset{i=1}{\mathcal{A}}}$ is the matrix assembly operator.

In this section, we demonstrate how to calculate the elemental matrix; in the next section, we demonstrate how to calculate the elemental residual.

Using the Galerkin approach, we can write $\mathbf{W}(x,y,z)$ and $\mathbf{V}(x,y,z,t)$ for each element as

$$\mathbf{W}=[\mathbf{W}][\mathbf{N}] \tag{1.67}$$

where

$$[\mathbf{W}]=\underset{matrix\ 5n\times 5n}{\begin{bmatrix}\langle w_\rho\rangle & 0 & 0 & 0 & 0\\ 0 & \langle w_{U_1}\rangle & 0 & 0 & 0\\ 0 & 0 & \langle w_{U_2}\rangle & 0 & 0\\ 0 & 0 & 0 & \langle w_{U_3}\rangle & 0\\ 0 & 0 & 0 & 0 & \langle w_E\rangle\end{bmatrix}} \tag{1.68}$$

$$[\mathbf{N}] = \underset{\text{matrix } 5n\times 5n}{\begin{bmatrix} \{N_\rho\} & 0 & 0 & 0 & 0 \\ 0 & \{N_{U_1}\} & 0 & 0 & 0 \\ 0 & 0 & \{N_{U_2}\} & 0 & 0 \\ 0 & 0 & 0 & \{N_{U_3}\} & 0 \\ 0 & 0 & 0 & 0 & \{N_E\} \end{bmatrix}} \tag{1.69}$$

where n is the number of nodes of the element, $\langle w_k \rangle (k = \rho, U_1, U_2, U_3, E)$ is a $1\times n$ row vector of scalar coefficients, and $\{N_k\}(k = \rho, U_1, U_2, U_3, E)$ is an $n\times 1$ column vector of shape functions.

The conservative variable vector $\mathbf{V}$ is approximated for each element as

$$\mathbf{V} = [\mathbf{N}]^T \underset{n\times 1 \text{ colum vector of degrees of freedom}}{\begin{Bmatrix} \rho_1 \\ \rho_n \\ U(1)_1 \\ . \\ . \\ . \\ U(3)_n \\ E_1 \\ E_n \end{Bmatrix}} \tag{1.70}$$

The elemental mass matrix is

$$\mathbf{M}^e = \frac{\alpha}{\Delta t} \int_{\Omega_e} [\mathbf{N}][\mathbf{N}]^T \, d\Omega \tag{1.71}$$

We can write the elemental stiffness matrix $\mathbf{K}^e$ as a combination of the advection matrix $\mathbf{K}^e_{adv}$, the diffusion matrix $\mathbf{K}^e_{diff}$, the SUPG stabilization matrix $\mathbf{K}^e_{SUPG}$, and the shock capturing term $\mathbf{K}^e_{sc}$:

$$\mathbf{K}^e = \mathbf{K}^e_{adv} + \mathbf{K}^e_{diff} + \mathbf{K}^e_{SUPG} + \mathbf{K}^e_{sc} \tag{1.72}$$

The advection term is

$$\mathbf{K}^e_{adv} = \int_{\Omega_e} [\mathbf{N}] \mathbf{A}_i \frac{\partial [\mathbf{N}]^T}{\partial \mathbf{x}_i} d\Omega \tag{1.73}$$

The diffusion term is

$$\mathbf{K}^e_{diff} = \int_{\Omega_e} \frac{\partial [\mathbf{N}]}{\partial \mathbf{x}_i} \mathbf{K}_{ij} \frac{\partial [\mathbf{N}]^T}{\partial \mathbf{x}_i} d\Omega \tag{1.74}$$

The shock capturing term is

$$\mathbf{K}^e_{sc} = \int_{\Omega_e} [\mu_c] \frac{\partial [\mathbf{N}]}{\partial \mathbf{x}_i} \frac{\partial [\mathbf{N}]^T}{\partial \mathbf{x}_i} d\Omega \tag{1.75}$$

The SUPG stabilization term is

$$\mathbf{K}^e_{SUPG} = \int_{\Omega_e} \frac{\partial [\mathbf{N}]}{\partial \mathbf{x}_i} \mathbf{A}^T_i \boldsymbol{\tau} \mathbf{A}_j \frac{\partial [\mathbf{N}]^T}{\partial \mathbf{x}_j} d\Omega \tag{1.76}$$

1.12 Elemental residual

The elemental residual can be calculated as

$$\mathbf{R}^e = \mathbf{M}^e \left\{ \dot{\mathbf{V}}^h \right\}^e + \mathbf{K}^e \left\{ \mathbf{V}^h \right\}^e - \left\{ \mathbf{F} \right\}^e \tag{1.77}$$

Since we already have the results of the vector $\mathbf{V}$ and its derivative $\dot{\mathbf{V}}$. The residual can be calculated when we discretize directly the weak form equation (1.61):

$$\mathbf{R}^e = \int_{\Omega_e} [\mathbf{N}] \left(\dot{\mathbf{V}} - \mathbf{F} + [\mathbf{N}] \mathbf{A}_i \frac{\partial \mathbf{V}}{\partial x_i} + \frac{\partial [\mathbf{N}]}{\partial x_i} \mathbf{A}_i^T \boldsymbol{\tau} \mathbf{A}_j \frac{\partial \mathbf{V}}{\partial x_j} + \mu_c \frac{\partial [\mathbf{N}]}{\partial x_i} \frac{\partial \mathbf{V}}{\partial x_i} \right) d\Omega \tag{1.78}$$

1.13 The standard Spalart-Allmaras turbulence model

The turbulence closure model used here is the Spalart-Allmaras model (Spalart and Allmaras, 1994). The Spalart-Allmaras model is an empirical scalar equation involving production, transport, diffusion, and destruction of turbulent viscosity. It introduces only one equation to the entire fluid domain. The dependent variable here is the turbulent viscosity ν_t, which should be always kept positive:

$$\frac{\partial \tilde{\nu}}{\partial t} + \mathbf{u}.\nabla\tilde{\nu} - \frac{1}{\sigma}\left(\nabla.\left(\left(\nu + \tilde{\nu} \right) \nabla\tilde{\nu} \right) + \mathrm{C}_{\mathrm{b2}} \left(\nabla\tilde{\nu} \right)^2 \right) - c_{b1}\tilde{\omega}\tilde{\nu} + c_{\omega 1} f_w \left(\tilde{\nu} \right) \left[\frac{\tilde{\nu}}{d} \right]^2 = 0 \tag{1.79}$$

where $\mathbf{u}$ is the velocity vector calculated from the Navier-Stokes equations and ν is the molecular viscosity.

$$\nu_t = \tilde{\nu} f_{v1}, f_{v1} = \frac{\chi^3}{\chi^3 + c_{v1}^3}, \chi = \frac{\tilde{\nu}}{\nu};$$

$$\tilde{\omega} = \omega + \frac{\tilde{\nu}}{k^2 d^2} f_{v2}, f_{v2} = 1 - \frac{\chi}{1 + \chi f_{v1}(\chi)};$$

$$f_w(g) = g\left(\frac{1 + c_{w3}^6}{g^6 + c_{w3}^6} \right)^{\frac{1}{6}}, \; g = r + c_{w2}(r^6 - r), r = \frac{\tilde{\nu}}{\omega k^2 d^2};$$

$c_{w1} = \frac{c_{b1}}{k^2} + \frac{1 + c_{b2}}{\sigma}$;

$c_{b1} = 0.1355, c_{b2} = 0.622, \sigma = \frac{2}{3}, c_{w2} = 0.3, c_{w3} = 2, c_{v1} = 7.1$.

$\tilde{\omega}$ can also be defined as:

$\tilde{\omega} = \omega \tilde{f}_{v3} + \frac{\tilde{\nu}}{k^2 d^2} \tilde{f}_{v2}$;

where $\tilde{f}_{v2} = (1 + \frac{\chi}{c_{v2}})^{-3}$, $\tilde{f}_{v3} = (1 + \chi f_{v1})(\frac{1 - \tilde{f}_{v2}}{\chi})$, $c_{v2} = 5$, $k = 0.41$, and d is the distance to the closest surface.

Here $c_{b1}\tilde{\omega}\tilde{\nu}$ is the production term and $\frac{1}{\sigma}\left(\nabla.\left(\left(\nu + \tilde{\nu}\right)\nabla\tilde{\nu}\right) + \mathrm{C_{b2}}\left(\nabla\tilde{\nu}\right)^2\right)$ is the diffusion term.

$c_{\omega 1} f_w\left(\tilde{\nu}\right)\left[\frac{\tilde{\nu}}{d}\right]^2$ is the destruction term and is negligible far from the wall.

Like the Navier-Stokes equations, we can write the equation of the turbulent viscosity in the dimensionless form using scales (indexed by *r*):

$\mathbf{u}^* = \frac{\mathbf{u}}{|\mathbf{u}_r|} \qquad \chi = \frac{\tilde{\nu}}{\nu} \qquad x^* = \frac{x}{L_r}$;

$t^* = \frac{t}{L_r / |\mathbf{u}_r|} \qquad \mathrm{Re} = \frac{|\mathbf{u}_r| L_r}{\nu}$.

For convenience, we omit the star:

$$\frac{D\chi}{Dt} - \frac{1}{Re\sigma}\left(\nabla.\left(\left(1 + \chi\right)\nabla\chi\right) + \mathrm{C_{b2}}\left(\nabla\chi\right)^2\right) - c_{b1}\tilde{\omega}\chi + \frac{c_{\omega 1}}{Re} f_w\left(\chi\right)\left[\frac{\chi}{d}\right]^2 = 0 \tag{1.80}$$

where $f_w(\chi) = g(\frac{1+c_{w3}^6}{g^6+c_{w3}^6})$, $\tilde{\omega} = \omega + \frac{1}{Re}\frac{\chi}{k^2 d^2} f_{v2}$ or $\tilde{\omega} = \omega \tilde{f}_{v3} + \frac{\chi}{Rek^2 d^2}\tilde{f}_{v2}$,

and $g(r) = r + c_{w2}\left(r^6 - r\right), r(\chi) = \frac{\chi}{Re\omega k^2 d^2}$.

For the boundary conditions, the model specifies $\chi = 0$ for the wall boundary. We specify $\chi = 1.0$ for the far field boundary.

1.14 Coupled Navier-Stokes Spalart-Allmaras model

We use the procedure proposed by Ben Haj Ali and Soulaïmani (Ben Haj Ali and Soulaïmani, 2010). At the first time step, the Navier-Stokes module sends the velocity vector $\mathbf{u}$ to the turbulence module. Given a flow field at time step *n*, and at the iteration *i*:

$$\chi_e^{n,i} = \frac{\sum_{j=1}^{nn} \chi_j^{n,i}}{n} \tag{1.81}$$

where nn is the number of nodes.

The turbulence module computes the parameters a and b back to the Navier-Stokes module:

$$\begin{cases} a = c_{b1}\tilde{\omega}^{n,i} \\ b = \frac{c_{\omega 1}}{\mathrm{Re}\, d^2} f_w^{n,i} \end{cases} \tag{1.82}$$

Then we update $\chi^{n,i+1}$ by solving

$$\frac{D\chi^{n,i+1}}{Dt} - \frac{1}{Re\sigma}\left(\nabla.\left(\left(1+\chi^{n,i+1}\right)\nabla\chi^{n,i+1}\right) + \mathrm{C}_{\mathrm{b2}}\left(\nabla\chi^{n,i+1}\right)^2\right) = a\chi^{n,i+1} - b\chi^{n,i+1^2} \tag{1.83}$$

The new turbulent viscosity $\chi^{n,i+1}$ is then sent to the Navier-Stokes module and we repeat this process until convergence.

1.15 Solution algorithms

1.15.1 Solution to the Navier-Stokes equations

We have presented how to use FEM to obtain the numerical solution. After we assemble the element matrices and element vectors, we obtain the matrix form of the system of equations:

$$\mathbf{KV} = \mathbf{F} \tag{1.84}$$

The matrix **K** is nonlinear and asymmetrical. To solve this system of equations directly would require a large amount of memory to store the matrices for large applications. It would be better if we were to linearize the system first and then use an iterative method to solve a set of linear systems.

We choose the Newton Raphson method considering the speed of convergence, although it is an expensive method. To solve the linearized system, we use the generalized minimal residual (GMRES) algorithm (Saad and Schultz, 1986). It is a stable iterative method, even for equations with matrices that are not positively definite.

The rate of convergence for all iterative methods depends on the preconditioning of the system; that is, the preconditioning of matrix **K** . We choose the ILUT preconditoner (Soulaïmani et al, 2002a) because it takes an acceptable amount of time to compute and does not consume excessive memory

1.15.2 Newton-Raphson method for the equation of turbulent viscosity

We write the turbulent viscosity as

$$[K(\chi)]\{\chi\} = \{S(\chi)\} \tag{1.85}$$

We can solve the system of equations directly, but the matrices would consume a large amount of memory to store. It would be better to linearize the system first, after which we could use iterative methods to solve the system of equations.

Our goal is to find a solution χ such that the residual $R(\chi)$ is numerically zero:

$$\{R(\chi)\} = \{S(\chi)\} - [K(\chi)]\{\chi\} \tag{1.86}$$

The Newton-Raphson method uses Taylor series around the previous iteration results. If we obtain V^{i-1} in iteration $i-1$ and $\{R(\chi^{i-1})\}$ is not numerically zero,

$$\{R(\chi^{i-1})\} = \{S(\chi^{i-1})\} - [K(\chi^{i-1})]\{\chi^{i-1}\} \neq 0 \tag{1.87}$$

At the next iteration i, we want to find χ^i such that

$$\{R(\chi^i)\} = \{R(\chi^{i-1} + \Delta\chi^i)\} \approx 0 \tag{1.88}$$

Using Taylor series around χ^{i-1},

$$\{R(\chi^{i-1} + \Delta\chi^i)\} = \{R(\chi^{i-1})\} + \left[\frac{\partial R}{\partial \chi}\right]_{\chi=\chi^{i-1}} \{\Delta\chi^i\} = 0 \tag{1.89}$$

that is,

$$[K_t(\chi^{i-1})]\{\Delta\chi^i\} = -\{R(\chi^{i-1})\} \tag{1.90}$$

where $\left[K_t(\chi^{i-1})\right]=\left[\frac{\partial R}{\partial \chi}\right]_{\chi=\chi^{i-1}}$, $\left\{\chi^i\right\}=\left\{\chi^{i-1}\right\}+\left\{\Delta\chi^i\right\}$.

$\left[K_t\right]$ is the tangent matrix, which is the derivative of the residual with respect to the turbulent viscosity vector. We then assemble the tangent matrix to calculate the residual and repeat the steps until convergence.

1.15.3 Calculation of the tangent matrix for turbulence

As shown previously, we also use the Newton-Raphson method (Ypma, 1999) to solve the equation for turbulent viscosity. We calculate the tangent matrix and the residual for each iteration. First, as with the Navier-Stokes equations, we write the equation for turbulent viscosity in its weak form:

$$\int_\Omega \delta\chi\left(\frac{D\chi}{Dt}-\frac{1}{Re\sigma}\left(\nabla\cdot\left((1+\chi)\nabla\chi\right)+\mathrm{C_{b2}}\left(\nabla\chi\right)^2\right)\right)d\Omega=\int_\Omega \delta\chi S(\chi)\,d\Omega \tag{1.91}$$

where $\delta\chi$ is the test function and $S(\chi)=c_{b1}\tilde{\omega}\chi-\frac{c_{\omega1}}{Re}f_w(\chi)\left[\frac{\chi}{d}\right]^2$.

We integrate by parts the term

$$\int_\Omega \delta\chi\nabla.\left((1+\chi)\nabla\chi\right)d\Omega=-\int_\Omega \nabla\delta\chi(1+\chi)\nabla\chi d\Omega+\int_\Gamma \delta\chi(1+\chi)\nabla\chi d\Gamma \tag{1.92}$$

The residual can be calculated as

$$R(\chi)=\int\langle\delta\chi\rangle\left\{r(\chi)\right\}d\Omega \tag{1.93}$$

where $\left\{r(\chi)\right\}=\{N\}\frac{\partial\chi}{\partial t}+\{N\}\mathbf{u}\cdot\nabla\chi+\frac{1}{\mathrm{Re}\,\sigma}(1+|\chi|)\left\{\frac{\partial N}{\partial x_i}\right\}\frac{\partial\chi}{\partial x_i}-\frac{c_{b2}}{\mathrm{Re}\,\sigma}\{N\}(\frac{\partial\chi}{\partial x_i})^2-\{N\}S(\chi)$.

The tangent matrix can be calculated as

$$\mathbf{K}_t = \frac{\partial \int r(\chi)\, d\Omega}{\partial \chi} = \mathbf{K}_t^{diff} + \mathbf{K}_t^{adv} + \mathbf{K}_t^{source} + \mathbf{K}_t^{mass} \tag{1.94}$$

The diffusion term is

$$\mathbf{K}_t^{diff} = \int_\Omega \left(\frac{1}{\mathrm{Re}\,\sigma}\left(1+|\chi|\right)\left\{\frac{\partial N}{\partial x_i}\right\}\left\langle\frac{\partial N}{\partial x_i}\right\rangle + \frac{1}{\mathrm{Re}\,\sigma}\langle N\rangle\left\{\frac{\partial N}{\partial x_i}\right\}\frac{\partial \chi}{\partial x_i} - \frac{2c_{b2}}{\mathrm{Re}\,\sigma}\{N\}\left\langle\frac{\partial N}{\partial x_i}\right\rangle\frac{\partial \chi}{\partial x_i}\right) d\Omega \tag{1.95}$$

The advection term is

$$\mathbf{K}_t^{adv} = \int_\Omega \left(\{N\} u_i \left\langle \frac{\partial N}{\partial x_i} \right\rangle \right) d\Omega \tag{1.96}$$

The implicit source term is

$$\mathbf{K}_t^{source} = \int_\Omega \left(c_{b1}\tilde{\omega} - 2\frac{C_{\omega 1}}{Re} f_w(\chi)\left[\frac{\chi}{d}\right] \right) d\Omega \tag{1.97}$$

The mass term is

$$\mathbf{K}_t^{mass} = \int_\Omega \left(\frac{1}{\Delta t}\{N\}\langle N\rangle \right) d\Omega \tag{1.98}$$

1.15.4 Preconditioning

To improve the speed of convergence, we use a preconditioner. Among several preconditioning methods, we choose incomplete LU factorization with threshold (ILUT) preconditioning (Soulaïmani et al, 2002a). This is an improvement of incomplete LU

factorizations. They proposed several ways to treat the small nonzero elements, which appear after Gaussian approximation in locations originally occupied by zero elements.

1.15.5 Additive Schwarz

To utilize parallel computing, we must first decompose the domain. In 1807, Schwarz proposed a decomposition procedure (Saad, 2003). The domain Ω is decomposed into n subdomains: $\{\Omega_1 \quad \Omega_2 \quad ... \quad \Omega_n\}$. The solution is updated at the end of the loop over all domains.

To solve the system of linear equations $\mathbf{Ax} = \mathbf{b}$, $\mathbf{A}$ is decomposed into blocks $\mathbf{A}_{ij}$, $\mathbf{x}$ is decomposed into blocks $\mathbf{x}_j$, and $\mathbf{b}$ is decomposed into blocks $\mathbf{b}_j$. $\mathbf{R}_i$ is the restriction operator for domain Ω_i, and $\mathbf{R}_i^T$ is the extension operator. The new solution $\mathbf{x}$ then can be written as

$$\mathbf{x}_{new} = \mathbf{x} + \sum_{i=1}^{n} \mathbf{R}_i^T \mathbf{A}_i^{-1} \mathbf{R}_i (\mathbf{b} - \mathbf{Ax}) \tag{1.99}$$

where $\mathbf{A}_i = \mathbf{R}_i \mathbf{A} \mathbf{R}_i^T$ is the local matrix associated with the domain Ω_i.

1.15.6 Parallel GMRES

The domain decomposition methods are simple methods for solving partial differential equations in parallel computing. We find the solution in the subdomains and then assemble the local solutions to obtain the global solution. Here we slightly modify the GMRES algorithm (Ben Haj Ali, 2002) to use parallel computing to solve the nonlinear system of equations. Our objective is to ensure the convergence of the global solution, not the local solutions.

The global residual $\mathbf{R}_{glob}$ is an assemblage of the local residuals of the n subdomains. Using the restriction operator $\mathbf{R}_i$, the global residual can be written as a combination of local residuals:

$$\mathbf{R}_{glob} = \sum_{i=1}^{n} \mathbf{R}_i^T \mathbf{R}_{loc} \tag{1.100}$$

GMRES in parallel computing also requires that we calculate the square of the global residual:

$$\left\|\mathbf{R}\right\|_{glob}^2 = \sum_{i=1}^{n} \left\|\mathbf{R}\right\|_{loc\,i}^2 \tag{1.101}$$

Similarly, the scalar product of w and v can be written as

$$\left(\mathbf{wv}\right)_{glob} = \sum_{i=1}^{n} \left(\mathbf{R}_i^T \mathbf{w}_i\right)_{loc} \left(\mathbf{R}_i^T \mathbf{v}_i\right)_{loc} \tag{1.102}$$

To ensure the continuity of the solution in the boundary between two domains, the degrees of freedom should be averaged.

CHAPTER 2

DIFFERENT ELEMENTS

2.1 Discretization

Now we choose appropriate finite elements to discretize the entire domain. Different types of elements are available, and we must weigh accuracy against computation time. More degrees of freedom and higher-order elements generally provide more accurate results but also increase the computation time. Only first-order interpolations for tetras, hexahedrons, prisms, and pyramids are used here. For fluid variables, each node of an element has five degrees of freedom: the density ρ, the unit momentum vector $\rho\mathbf{U}$, and the unit total energy ρe_t. For turbulence variables, each node of an element has one degree of freedom: turbulence viscosity.

2.2 Shape function

Because the same types of elements generally do not have identical size, it will be easier if we use a general element called the reference element. The reference element and the actual element can be transformed back and forth using geometric transformations.

Here we explain how the shape functions are derived (Dhatt and Touzot, 1981).

Let $\varphi(\xi)$ be one flow variable expressed as

$$\varphi(\xi) = \langle P(\xi) \rangle \{a\} \tag{2.1}$$

where $\langle P(\xi) \rangle$ is the polynomial basis and $\{a\}$ is the coefficient vector. We then construct the nodal matrix:

$$\left[P_n\right]=\left[P_j\left(\xi_i\right)\right];\quad i,j=1,2,...,n_d \tag{2.2}$$

where n_d is the degree of freedom for the element.

We invert the matrix $\left[P_n\right]$ and calculate the shape function:

$$\left\langle N(\xi)\right\rangle=\left\langle P(\xi)\right\rangle\left[P_n\right]^{-1} \tag{2.3}$$

Therefore, equation (2.1) becomes

$$\varphi(\xi)=\left\langle N(\xi)\right\rangle\begin{Bmatrix}\varphi_1\\ \varphi_2\\ \\ \varphi_3\end{Bmatrix} \tag{2.4}$$

2.3 Numerical integration

Integration over the actual element can be transformed to the reference element with the help of the Jacobian matrix $\mathbf{J}$:

$$\mathbf{J}=\begin{Bmatrix}\dfrac{\partial}{\partial\xi}\\ \dfrac{\partial}{\partial\eta}\\ \dfrac{\partial}{\partial\zeta}\end{Bmatrix}\begin{Bmatrix}x & y & z\end{Bmatrix} \tag{2.5}$$

The integration of the function $f(x,y,z)$ over the domain Ω_e can now be written as

$$\int_{\Omega_e} f(x,y,z)d\Omega_e = \int_{\Omega_r} f(\xi,\eta,\zeta)\det(\mathbf{J})d\Omega_r \tag{2.6}$$

Using the Gauss integration method, the integration of the function $f(\xi,\eta,\zeta)$ over the reference domain Ω_r can be written as

$$\int_{\Omega_r} f(\xi,\eta,\zeta)d\Omega_r \approx \sum_{i=1}^{r} \det(\mathbf{J})w_i f(\xi_i,\eta_i,\zeta_i) \tag{2.7}$$

where w_i is the weight function, ξ_i,η_i,ζ_i is a set of integration points, and r is the number of integration points.

2.4 The two-node line element

The two-node line element is the most basic element, and it is of class C^0. The two geometric points coincide with the interpolation points. We are interested only in the integration points and weights of this element.

We first write the polynomial as (Dhatt and Touzot, 1981)

$$f(\xi) = a_1 + a_2\xi + \cdots a_{2r}\xi^{2r-1} \tag{2.8}$$

We want the numerical integration with r points to be exact for a polynomial $f(\xi)$ with order $m \le 2r-1$:

$$\begin{aligned} &\int_{-1}^{1} f d\xi = \int_{-1}^{1} (a_1 + a_2\xi + \cdots a_{2r}\xi^{2r-1})d\xi = w_1 f(\xi_1) + w_2 f(\xi_2) + \cdots \\ &+ w_i f(\xi_i) + \cdots w_r f(\xi_r) = \sum_{i=1}^{r} w_i f(\xi_i) \end{aligned} \tag{2.9}$$

For the equation to be valid for any coefficients,

$$
\begin{aligned}
&f(\xi) = 1 \to 2 = w_1 + w_2 + \cdots + w_r \\
&f(\xi) = \xi \to 0 = w_1\xi_1 + w_2\xi_2 + \cdots + w_r\xi_r \\
&f(\xi) = \xi^2 \to \frac{2}{3} = w_1\xi_1^2 + w_2\xi_2^2 + \cdots + w_r\xi_r^2 \\
&\cdots \\
&f(\xi) = \xi^{2r-1} \to 0 = w_1\xi_1^{2r-1} + w_2\xi_2^{2r-1} + \cdots + w_r\xi_r^{2r-1}
\end{aligned}
\tag{2.10}
$$

If we choose two Gauss integration points and want the integration to be exact for the polynomial basis with a maximum order of 3, using the above equations we obtain the integration points and weights, we obtain the results in Table 2.1.

Table 2.1 Numerical Integration
(Line Reference Element)

	ξ_1	ω_1
N_1	$\frac{1}{\sqrt{3}}$	1.0
N_2	$-\frac{1}{\sqrt{3}}$	1.0

2.5 The eight-node hexahedron element

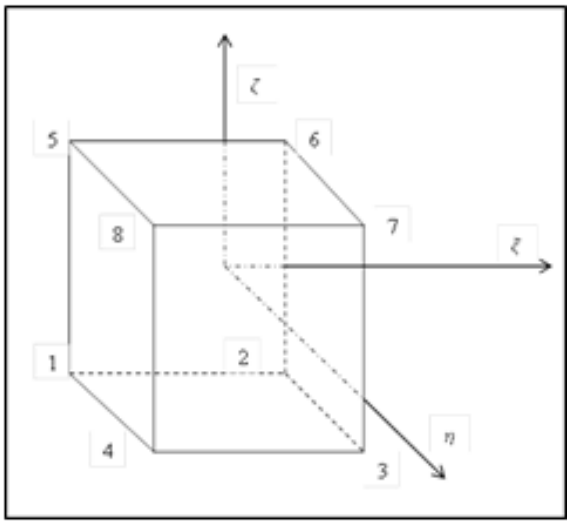


Figure 2.1 Eight-node hexahedron element

The eight-node hexahedron element is commonly used in FEM and is simple. It is of class C^0. A hexahedron element has 8 corners, 12 sides, and 6 faces. The natural coordinates for a hexahedron element are found in Table 2.2.

Table 2.2 Coordinates
(Hexahedron Reference Element)

Node	ξ	η	ζ
1	-1	-1	-1
2	1	-1	-1
3	1	1	-1
4	-1	1	-1
5	-1	-1	1

Node	ξ	η	ζ
6	1	-1	1
7	1	1	1
8	-1	1	1

In order to obtain the shape functions, we choose the polynomial basis as

$$\langle P \rangle = [1 \quad \xi \quad \eta \quad \zeta \quad \xi\eta \quad \eta\zeta \quad \zeta\xi \quad \xi\zeta] \tag{2.11}$$

The shape functions for this element are listed in Table 2.3.

Table 2.3 Shape Functions
(Hexahedron Reference Element)

	$\{N\}$
N_1	$\frac{1}{8}(1-\xi)(1-\eta)(1-\zeta)$
N_2	$\frac{1}{8}(1+\xi)(1-\eta)(1-\zeta)$
N_3	$\frac{1}{8}(1+\xi)(1+\eta)(1-\zeta)$
N_4	$\frac{1}{8}(1-\xi)(1+\eta)(1-\zeta)$
N_5	$\frac{1}{8}(1-\xi)(1-\eta)(1+\zeta)$
N_6	$\frac{1}{8}(1+\xi)(1-\eta)(1+\zeta)$
N_7	$\frac{1}{8}(1+\xi)(1+\eta)(1+\zeta)$
N_8	$\frac{1}{8}(1-\xi)(1+\eta)(1+\zeta)$

The Jacobian matrix is

$$\mathbf{J} = \begin{bmatrix} x_i \dfrac{\partial N_i}{\partial \xi} & y_i \dfrac{\partial N_i}{\partial \xi} & z_i \dfrac{\partial N_i}{\partial \xi} \\ x_i \dfrac{\partial N_i}{\partial \eta} & y_i \dfrac{\partial N_i}{\partial \eta} & z_i \dfrac{\partial N_i}{\partial \eta} \\ x_i \dfrac{\partial N_i}{\partial \zeta} & y_i \dfrac{\partial N_i}{\partial \zeta} & z_i \dfrac{\partial N_i}{\partial \zeta} \end{bmatrix} \tag{2.12}$$

where x, y, and z are replaced by $x_i N_i$, $y_i N_i$, and $z_i N_i$, respectively, and the summation convention is applied over $i = 1, 2, ...8$.

For a non-distorted cubic element, the determinant of the Jacobian matrix is constant:

$$\det(\mathbf{J}) = \frac{1}{8} V \tag{2.13}$$

where V is the volume of the actual element. For distorted hexahedrons, det(**J**) is not constant.

A two-node line element integration rule is applied in each direction. The integration can then be obtained by using products:

$$\int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} f(\xi,\eta,\zeta)\,d\xi\,d\eta\,d\zeta = \sum_{i=1}^{r_1}\sum_{j=1}^{r_2}\sum_{k=1}^{r_3} w_i w_j w_k f\left(\xi_i, \eta_j, \zeta_k\right) \tag{2.14}$$

This method using $r_1 \times r_2 \times r_3$ points calculates exactly the integration for the monomial $\xi^i \eta^j \zeta^k$ with $i \le 2r_1 - 1$, $j \le 2r_2 - 1$, $k \le 2r_3 - 1$.

We then obtain the integration points and weights for a hexahedron, which can be seen in Table 2.4.

Table 2.4 Numerical Integration
for Hexahedron Reference Element(Dhatt and Douzot, 1981)

Point	ξ	η	ζ	**Weights**
1	$-\frac{1}{\sqrt{3}}$	$-\frac{1}{\sqrt{3}}$	$-\frac{1}{\sqrt{3}}$	1
2	$-\frac{1}{\sqrt{3}}$	$-\frac{1}{\sqrt{3}}$	$\frac{1}{\sqrt{3}}$	1
3	$-\frac{1}{\sqrt{3}}$	$\frac{1}{\sqrt{3}}$	$-\frac{1}{\sqrt{3}}$	1
4	$-\frac{1}{\sqrt{3}}$	$\frac{1}{\sqrt{3}}$	$\frac{1}{\sqrt{3}}$	1
5	$\frac{1}{\sqrt{3}}$	$-\frac{1}{\sqrt{3}}$	$-\frac{1}{\sqrt{3}}$	1
6	$\frac{1}{\sqrt{3}}$	$-\frac{1}{\sqrt{3}}$	$\frac{1}{\sqrt{3}}$	1
7	$\frac{1}{\sqrt{3}}$	$\frac{1}{\sqrt{3}}$	$-\frac{1}{\sqrt{3}}$	1
8	$\frac{1}{\sqrt{3}}$	$\frac{1}{\sqrt{3}}$	$\frac{1}{\sqrt{3}}$	1

2.6 The four-node tetra element

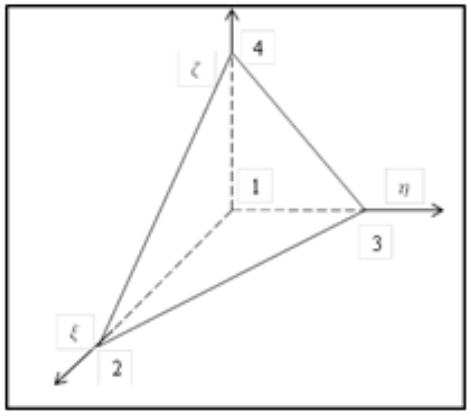


Figure 2.2 Four-node tetra element

The four-node tetra element is simple; all of its shape functions are linear polynomials. The element has 4 corners, 6 sides, and 4 faces. It is of class C^0. The sides are straight, and the faces are planar. The four nodes coincide with the interpolation points. The coordinates for a tetra reference element are listed in Table 2.5.

Table 2.5 Coordinates
(Tetra Reference Element)

Node	ξ	η	ζ
1	0	0	0
2	1	0	0
3	0	1	0
4	0	0	1

In order to obtain the shape functions, we choose the polynomial basis as

$$\langle P \rangle = \langle 1 \quad \xi \quad \eta \quad \zeta \rangle \tag{2.15}$$

The shape functions for a tetra element are in Table 2.6.

Table 2.6 Shape Functions
(Tetra Reference Element)

	$\{\boldsymbol{N}\}$
$\boldsymbol{N_1}$	$1-\xi-\eta-\zeta$
$\boldsymbol{N_2}$	ξ
$\boldsymbol{N_3}$	η
$\boldsymbol{N_4}$	ζ

The Jacobian matrix is

$$\mathbf{J}=\begin{bmatrix} x_2-x_1 & y_2-y_1 & z_2-z_1 \\ x_3-x_1 & y_3-y_1 & z_3-z_1 \\ x_4-x_1 & y_4-y_1 & z_4-z_1 \end{bmatrix} \tag{2.16}$$

The determinant is constant:

$$\det(\mathbf{J}) = 6V \tag{2.17}$$

where V is the volume of the actual element.

It should be noted that we want $\det(\mathbf{J})$ to always be positive, and the nodes should be properly numbered. We choose a face first and then number the three nodes counterclockwise looking from the other node.

To obtain the integration points and weights for a four-node tetra element, we will first use the direct method discussed in section 2.4:

$$\iiint_{\Omega^e} f(\xi,\eta,\zeta)\,d\xi d\eta = \sum_{i=1}^{r} w_i f(\xi_i,\eta_i,\zeta_i) \tag{2.18}$$

which is the exact integration for a monomial $\xi_i\eta_j\zeta_k$ with $i+j+k \le m$. For $m=3$, we obtain the integration points and weights listed in Table 2.7.

Table 2.7 Numerical Integration
(Tetra Reference Element)

Point	ξ	η	ζ	Weights
1	$\frac{(5-\sqrt{5})}{20}$	$\frac{(5-\sqrt{5})}{20}$	$\frac{(5-\sqrt{5})}{20}$	$\frac{1}{24}$
2	$\frac{(5-\sqrt{5})}{20}$	$\frac{(5-\sqrt{5})}{20}$	$\frac{(5+3\sqrt{5})}{20}$	$\frac{1}{24}$
3	$\frac{(5-\sqrt{5})}{20}$	$\frac{(5+3\sqrt{5})}{20}$	$\frac{(5-\sqrt{5})}{20}$	$\frac{1}{24}$
4	$\frac{(5+3\sqrt{5})}{20}$	$\frac{(5-\sqrt{5})}{20}$	$\frac{(5-\sqrt{5})}{20}$	$\frac{1}{24}$

2.7 The six-node prism element

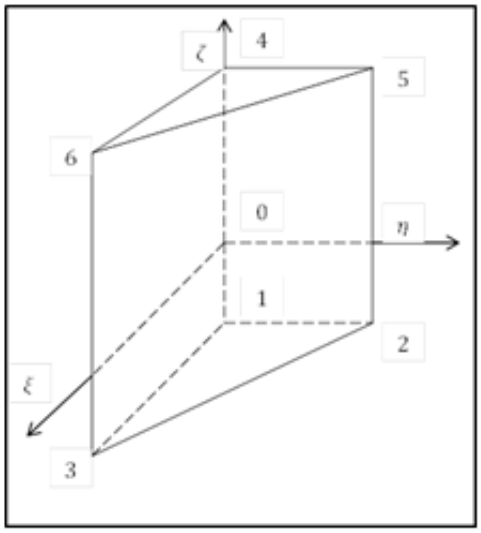


Figure 2.3 Six-node prism element

The majority of the elements used in the boundary layer are six-node prism elements, which are of class C^0. A six-node prism has 6 corners, 9 sides, and 5 faces. It is also called a wedge. The natural coordinates for a prism element are listed in Table 2.8.

Table 2.8 Coordinates
(Prism Reference Element)

Node	ξ	η	ζ
1	0	0	-1
2	1	0	-1
3	0	1	-1
4	0	0	1
5	1	0	1
6	0	1	1

The polynomial basis we choose is

$$\langle P \rangle = \langle 1 \quad \xi \quad \eta \quad \zeta \quad \xi\zeta \quad \eta\zeta \rangle \tag{2.19}$$

Then we obtain the shape functions in Table 2.9.

Table 2.9 Shape Functions
(Prism Reference Element)

	$\{\boldsymbol{N}\}$
$\boldsymbol{N}_1$	$(1-\xi-\eta)(\frac{1-\zeta}{2})$
$\boldsymbol{N}_2$	$\xi(\frac{1-\zeta}{2})$
$\boldsymbol{N}_3$	$\eta(\frac{1-\zeta}{2})$
$\boldsymbol{N}_4$	$(1-\xi-\eta)(\frac{1+\zeta}{2})$
$\boldsymbol{N}_5$	$\xi(\frac{1+\zeta}{2})$
$\boldsymbol{N}_6$	$\eta(\frac{1+\zeta}{2})$

The determinant of the Jacobian matrix is

$$\det(\mathbf{J}) = V \tag{2.20}$$

where V is the volume of the actual element.

We use a triangular reference as a base, and then we use two integration points in the ζ direction. Using the direct method discussed previously, for a monomial $\xi^i\eta^j$ with $i+j \le 3$,

the integration points and weights for the triangular reference element are listed in Table 2.10.

Table 2.10 Numerical Integration
(Triangular Reference Element)

Point	ξ	η	Weights
1	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{-27}{96}$
2	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{25}{96}$
3	$\frac{3}{5}$	$\frac{1}{5}$	$\frac{25}{96}$
4	$\frac{1}{5}$	$\frac{3}{5}$	$\frac{25}{96}$

By using a tensor product of the numerical integration rule for a triangle and a line, we obtain the integration points and weights for the prism element in Table 2.11.

Table 2.11 Numerical Integration
(Prism Reference Element)

Point	ξ	η	ζ	Weights
1	$\frac{1}{3}$	$\frac{1}{3}$	$-\frac{1}{\sqrt{3}}$	$\frac{-27}{96}$
2	0.6	0.2	$-\frac{1}{\sqrt{3}}$	$\frac{25}{96}$
3	0.2	0.6	$-\frac{1}{\sqrt{3}}$	$\frac{25}{96}$

Point	ξ	η	ζ	Weights
4	0.2	0.2	$-\frac{1}{\sqrt{3}}$	$\frac{25}{96}$
5	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{\sqrt{3}}$	$\frac{-27}{96}$
6	0.6	0.2	$\frac{1}{\sqrt{3}}$	$\frac{25}{96}$
7	0.2	0.6	$\frac{1}{\sqrt{3}}$	$\frac{25}{96}$
8	0.2	0.2	$\frac{1}{\sqrt{3}}$	$\frac{25}{96}$

2.8 The five-node pyramid element

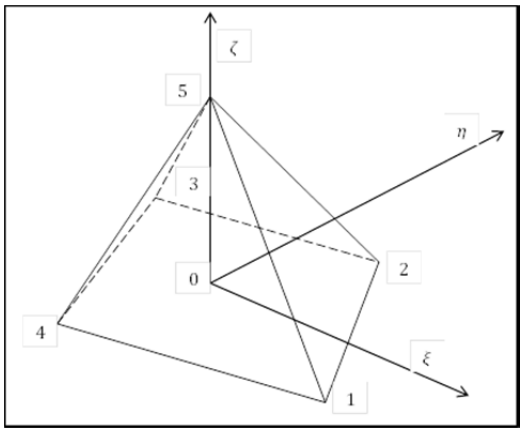


Figure 2.4 Five-node pyramid element

Here we introduce the five-node pyramid element. The pyramid element has 5 corners, 8 sides, and 5 faces. Its base is quadrilateral, and the corner opposite the base is called the apex. Pyramids can serve as a transition between tetra elements and hexahedron elements. The natural coordinates for a pyramid element are found in Table 2.12.

Table 2.12 Coordinates
(Pyramid Reference Element)

Node	ξ	η	ξ
1	1	0	0
2	0	1	0
3	-1	0	0
4	0	-1	0
5	0	0	1

The shape functions of the pyramid element can be derived from the 8-node hexahedron element. N_1, N_2, N_3, and N_4 are the same as for the hexahedron. The other four nodes of the hexahedron collapse to form the apex:

$$N_5 = N_1 + N_2 + N_3 + N_4 \tag{2.21}$$

The shape functions for this element are seen in Table 2.13.

Table 2.13 Shape Functions
(Pyramid Reference Element)

	$\{\boldsymbol{N}\}$
$\boldsymbol{N_1}$	$\frac{1}{8}(1-\xi)(1-\eta)(1+\zeta)$
$\boldsymbol{N_2}$	$\frac{1}{8}(1+\xi)(1-\eta)(1+\zeta)$
$\boldsymbol{N_3}$	$\frac{1}{8}(1+\xi)(1+\eta)(1+\zeta)$

	$\{\boldsymbol{N}\}$
$\boldsymbol{N}_4$	$\frac{1}{8}(1-\xi)(1+\eta)(1+\zeta)$
$\boldsymbol{N}_5$	$\frac{1}{2}(1+\zeta)$

The Jacobian matrix for a pyramid element is

$$\mathbf{J} = \begin{bmatrix} \frac{(1-\zeta)}{2} & 0 & 0 \\ 0 & \frac{(1-\zeta)}{2} & 0 \\ -\frac{\xi}{2} & -\frac{\eta}{2} & 1 \end{bmatrix} \tag{2.22}$$

Unlike the case with the other three elements, $\det(\mathbf{J})$ is not constant. The determinant of the Jacobian matrix is

$$\det(\mathbf{J}) = \frac{(1-\zeta)^2}{4} \tag{2.23}$$

We can notice that $\det(\mathbf{J}) = 0$ at the apex $\zeta = 1$. The inverse of the Jacobian matrix, which is used for the calculation of Cartesian derivatives of shape functions, is undefined at the apex.

The numerical integration is slightly more complex for the pyramid element. There are several definitions for numerical integration. We consider the definitions by Chen et al (2012), who proposed a second-order five-point numerical integration formula and a third-order six-point numerical integration formula.

Table 2.14 Five-Point Numerical Integration
for the Pyramid Reference Element (Chen et al, 2012)

Point	ξ	η	ζ	**Weights**
1	0	0	$\frac{70+21\sqrt{35}}{280}$	$\frac{16}{75}$
2	$\sqrt{\frac{5}{21}}$	$\sqrt{\frac{5}{21}}$	$\frac{35-2\sqrt{35}}{140}$	$\frac{7}{25}$
3	$\sqrt{\frac{5}{21}}$	$-\sqrt{\frac{5}{21}}$	$\frac{35-2\sqrt{35}}{140}$	$\frac{7}{25}$
4	$-\sqrt{\frac{5}{21}}$	$\sqrt{\frac{5}{21}}$	$\frac{35-2\sqrt{35}}{140}$	$\frac{7}{25}$
5	$-\sqrt{\frac{5}{21}}$	$-\sqrt{\frac{5}{21}}$	$\frac{35-2\sqrt{35}}{140}$	$\frac{7}{25}$

Table 2.15 Six-Point Numerical Integration
for the Pyramid Reference Element (Chen et al, 2012)

Point	ξ	η	ζ	**Weights**
1	0	0	$\frac{1}{2}$	$\frac{3}{5}$
2	$\sqrt{\frac{4}{27}}$	$\sqrt{\frac{4}{27}}$	$\frac{1}{6}$	$\frac{9}{20}$
3	$\sqrt{\frac{4}{27}}$	$-\sqrt{\frac{4}{27}}$	$\frac{1}{6}$	$\frac{9}{20}$
4	$-\sqrt{\frac{4}{27}}$	$\sqrt{\frac{4}{27}}$	$\frac{1}{6}$	$\frac{9}{20}$
5	$-\sqrt{\frac{4}{27}}$	$-\sqrt{\frac{4}{27}}$	$\frac{1}{6}$	$\frac{9}{20}$
6	0	0	$\frac{1}{4}$	$\frac{16}{15}$

CHAPTER 3

OBJECT-ORIENTED PROGRAMMING

3.1 Object-Oriented programming

Object-oriented programming (OOP) has gained in popularity during recent years (Budd, 1997). The concept of Object-oriented programming (OOP) is different from traditional flow-based programming. Commonly used object-oriented languages include C++, Java, and Python. The concept of OOP provides code reusability and more efficiency in programming. We show in this chapter how we used OOP to calculate the elemental residual and elemental matrix.

OOP uses classes, which have data, and methods, which are the subroutines and functions associated with an object. The three basic concepts of OOP use are encapsulation, polymorphism, and inheritance.

Encapsulation allows both data and methods to be defined in a single class. We can control the access of both at various levels. We can make both data and methods available only to the methods of the same class; this level of access, which is called “private,” is the strictest. We can also make data and methods available to the inheritance class; this level of access is called “protected.” The other level of access, “public,” is the least strict; methods of any other class can access its data and subroutines.

Inheritance enables the creation of a new class based on an existing class. The new class inherits and can reuse the data and methods of the old class. It can also replace the methods of the old class, which is known as overriding.

Polymorphism provides the same interface to different types. We can define the data and methods in one class and its inheritance class can interpret the data and methods in its own way. Two different inheritance classes can have different data and methods.

3.2 Object-oriented programming in calculating elemental matrix and residual

First, we define an object called **Element,** which has four attributes. The number of nodes in an element is defined by **inel**. The degree of liberty for the element is defined by **ndln**. The dimension of the problem is defined by **ndim**. The number of integration points is defined by **ipg**. There are also some allocatable variables that are used to calculate the elemental matrix and residual.

The **Element** object has five procedures. We use **GaussPoints** to obtain the integration points. We use **ShapeFunction** to obtain the shape functions for each element. We use **Initialize** to initialize the attributes. We use **Allocation** to allocate memory for the variables. We use **Finalize** to deallocate the allocated memory.

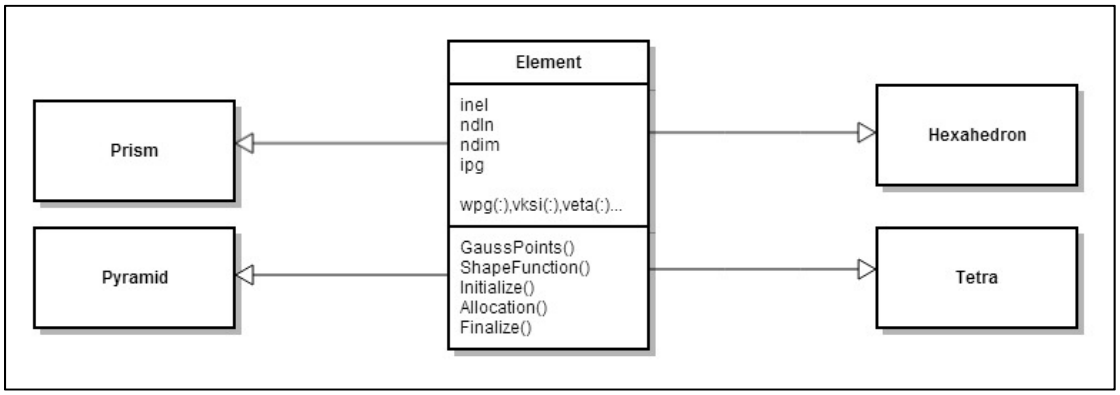


Figure 3.1 The class **Element** and its four derived types

The code for the class element module is shown below. The variable allocation and deallocation parts are not shown here.

```
module class_Element
use data_world
implicit none
public  :: Element
type, abstract    :: Element

integer :: ndim, inel,ndln,ipg

contains   ! encapsulated functionality
   procedure :: GaussPoints
   procedure :: ShapeFunction
   procedure :: Initialize
   procedure :: Allocation
   procedure :: Finalize
end type Element

contains

subroutine initialize(this) ! public constructor

class(element):: this
integer :: ndim,inel,ndln,ipg,npg
this%ndim=0
this%inel=0
this%ndln=0
this%ipg=0! four variables to define an element


endsubroutine initialize


subroutine GaussPoints(this,wpg,vksi,veta,vzeta)
class(Element),intent(in):: this
double precision :: wpg(*),vksi(*),veta(*),vzeta(*)
endsubroutine GaussPoints

subroutine ShapeFunction(this,wpg,vksi,veta,vzeta,vni)
class(Element),intent(in):: this
double precision :: vni(*),vksi(*),veta(*),vzeta(*),wpg(*)
endsubroutine ShapeFunction


end module class_Element
```

Figure 3.2 Class **Element**

In Figure 3.2 we define the class **Element**. The types **Hexahedron**, **Tetras**, **Pyramid,** and **Prism** all inherit the class **Element**. We then use the concept of polymorphism to calculate the integration points and shape functions. The subroutines to calculate integration points and shape function are first defined in the object **Element**. We then define the subroutines **Initialize**, **ShapeFunction,** and **GaussPoints** for each element.

```
module class_Tetra
use class_Element
implicit none
private
public :: Tetra

type, extends(Element) :: Tetra
contains
    procedure :: GaussPoints   => GaussTetra
    procedure :: ShapeFunction => ShapeTetra
    procedure :: Initialize    => Tetra_

end type Tetra
contains

      subroutine Tetra_(this) ! public constructor

class(Tetra):: this



this%inel=4
this%ndim=3
this%ndln=5
this%ipg=4! Four variables to define an element for fluid

     endsubroutine Tetra_


subroutine GaussTetra(this,wpg,vksi,veta,vzeta)

! The subroutine to calculate intergration points is not shown.

  endsubroutine GaussTetra

subroutine shapeTetra(this,wpg,vksi,veta,vzeta,vni)

! The subroutine to calculate shape functions is not shown.

  endsubroutine ShapeTetra

end module class_Tetra
```

Figure 3.3 Class **Tetra**

In Figure 3.3, the allocation of variables and the subroutines **ShapeFunction** and **GaussPoints** are not shown.

At the beginning, the subroutine, which calculates the element matrix, obtains the element type. We then define a pointer to the class **Element,** which can point to an object of the type **Hexahedron, Tetra, Prism,** or **Pyramid**. They all are derived types of the class **Element.**

```
...
  type(Prism),target   :: Prism1
  type(Pyramid),target :: Pyramid1
  type(Hexa),target    :: Hexa1
  type(Tetra),target   :: Tetra1
  class(element), POINTER :: e


   select case (itpe)
     case (1)
       e=> Tetra1
     case (2)
       e=> Hexa1
     case (3)
       e=> Prism1
     case (4)
       e=> Pyramid1
   endselect
...
```

Figure 3.4 Element initialization

In Figure 3.4, we define objects **Prism1**, **Pyramid1**, **Hexa1**, and **Tetra1**. They are of the type **Prism**, **Pyramid**, **Hexa**, and **Tetra**, respectively. The polymorphic pointer variable **e** can point to any object of the class **Element** or any of its derived types.

We can then simply write the following two lines to calculate shape functions and integration points for any type of element.

```
call e%GaussPoints(e%wpg,e%vksi,e%veta,e%vzeta)
call e%ShapeFunction(e%wpg,e%vksi,e%veta,e%vzeta,e%vni)
```

Figure 3.5 Shape function and integration points

In Figure 3.5, we directly access the data and modify the object. The vectors of the integration points are **vksi**, **veta,** and **vzeta**. The weight of the integration point is represented by **wpg**. The vector of shape functions is represented by **vni**. We use the concept of polymorphism here. For example, if the object is of type **Prism**, the code will use the subroutine **GaussPoints** and the subroutine **ShapeFunction** defined in the class **Prism**.

The complete code to calculate the element residual as an example of how we implemented the OOP can be found in the following code:

```
!============================================================
!      Calculation of the elemental residual

!      icode = 1 -----> interpolation function
!      icode = 2 -----> element residual : vfe
!      icode = 3 -----> physical variables
!
!      vpree(1) =  Reynolds number
!      vpree(2) =  Specific heat ratio gama  1.4
!      vpree(3) =  Prandtl number    0.72
!      vpree(4) =  Turbulent Prandtl  1.0
!      vpree(5) =  mach number
!      vpree(6) =  1.d0 if the time step is constant
!      vpree(7) =  artificial viscosity coefficient  1.0
!      vpree(8) =  1.d0
!      vpree(9) =  shock capturing coefficient
!      vpree(10)=  1.d0 stabilized formulation   1.0

!=================================================================
!
       subroutine residu_general_fluid(itpe,igre,vcore,vpree,vdle,vfe,kne)
!
!=================================================================

       USE global_data,only:probtype,VTG,VPAS,dc_coef,LapP,myid,tempicount
       USE DATA_WORLD
       USE SPALART,only:cv1
       USE class_Element
       USE class_Hexa
       USE class_Pyramid
       USE class_Prism
       Use class_Tetra
       IMPLICIT NONE
       save
     double precision:: vfe(*),vdle(*),vcore(*),vpree(*), &
     & taua,tauc ,dcs2,mpi_comm_world

     integer   kne(*), LapE(8)

     integer itpe, igre,ierr

     INTEGER :: nn1,nn2,nn3,n_vprne,n_vpree
```

```
      DOUBLE PRECISION  ::  detj, &
      &              vprne(10), &
      &              u1,u1t,u2,u2t,u3,u3t,uu, &
      &              u1x,u1y,u1z,u2x,u2y,u2z,u3x,u3y,u3z, &
      &              temp,dens,denst,enrg,enrgt,vmua,prdt, &
      &              denx,deny,denz,enrx,enry,enrz,gu, &
      &              v1,v2,v3, &
      &              v1x,v1y,v1z,v2x,v2y,v2z,v3x,v3y,v3z, &
      &              pres,prex,prey,prez,temx,temy,temz, &
      &              gradx(5),grady(5),gradz(5),divu,divv, &
      &              sig11,sig22,sig33,sig12,sig21,sig13,sig31, &
      &              sig23,sig32,heat1,heat2,heat3,s1,s2, &
      &              fadv(5),fdif1(5),fdif2(5),fdif3(5), &
      &              fstabx(5),fstaby(5),fstabz(5), &
      &              w1,w2,w3,sol1(5,5),sol2(5,5),sol3(5,5), &
      &              a1(5,5),a2(5,5),a3(5,5),gradp(5), &
      &              vk11(5,5),vk12(5,5),vk13(5,5), &
      &              vk21(5,5),vk22(5,5),vk23(5,5), &
      &              vk31(5,5),vk32(5,5),vk33(5,5), &
      &              h11(5,5),h12(5,5),h13(5,5), Lmach, &
      &              h21(5,5),h22(5,5),h23(5,5), dp,dpmax,dcs, &
      &              h31(5,5),h32(5,5),h33(5,5),dc,dc1,dcv(5), &
      &              hel,helmin,helmax,v1moy,v2moy,v3moy,pmoy,dmoy, &
      &              h1(5,5),h2(5,5),h3(5,5),g(5), &
      &              coef,gama,gama1,chi,nu_turb,nu_tield,fv1, &
      &              un,deux,trois,quatre,deuti,eps,eps1, &
      &              vmu,vlamda,vkapa,cdm1,cdm,cel,vno,PL,Lambda, &
      &              utau,zero,xsi1,xsi2,uf,xx12,xx13,xx14,yy12,yy13, &
      &              yy14,zz12,zz13,zz14,vl12,vl13,vl14,xx23,xx24,xx34,yy23, &
      &              yy24,yy34,zz23,zz24,zz34,vl23,vl24,vl34,vmu_turb, &
      &              vnorm,Pec,zeta, grad_dens,grad(5),variable(5),grad_pres, &
      &              xx11  ,vort1,vort2,vort3,w,eyplus, &
      &              Fx(5),Fy(5),Fz(5),divF(5), &
      &              volume, S,alpha
            INTEGER :: i,j,ii,jj,kk,ll,mm,is,ip,k,ielag,ig,ig1, &
      &                i1,j1,jb,is1,js1


     type(Prism),target   :: Prism1
     type(Pyramid),target :: Pyramid1
     type(Hexa),target    :: Hexa1
     type(Tetra),target   :: Tetra1
     class(element), POINTER :: e

        if (icode .eq. 1 )then
        return
        endif
        if (icode .eq. 2 )then
        return
        endif
        if (icode .eq. 4 )then
        return
        endif

!------ initialize different types of element according to itpe

      select case (itpe)
        case (1)

          e=> Tetra1
```

```
       case (2)

         e=> Hexa1


       case (3)

         e=> Prism1


       case (4)

         e=> Pyramid1


     endselect


     call e%initialize
     call e%allocation


       UN= 1.D0
       DEUX=2.D0
       trois=3.d0
       quatre= 4.d0
       zero=0.d0
       eps=1.d-16
       eps1=1.d-08

       deuti = deux/trois

       dpas=VPAS(ieeloc)

 !------ copy VDLEV into VDLEV
       idle=e%inel*e%ndln
       do k=1,idle
       e%vdlev(k)= vdlev(k)/dpas
       enddo

!----- DC jameson
         do i=1,e%inel
          LapE(i)= LapP(kne(i))
         enddo

!========== Fluid properties gama and Prandtl ================================

       gama  = vpree(2)

       gama1 = gama - un

       prdt  = vpree(3)


!    Gauss points
       call e%GaussPoints(e%wpg,e%vksi,e%veta,e%vzeta)


!    Shape functions

       call e%shapeFunction(e%wpg,e%vksi,e%veta,e%vzeta,e%vni)
```

```
!=====================================================================
!    Compute local residual
!=====================================================================

!================ Extract local vectors such as density, velocity, and their time
derivatives

      do i=1,e%inel
      e%vmut(i)=zero
!pointers
      ii = e%kro(i)
      jj = e%ku1(i)
      kk = e%ku2(i)
      ll = e%ku3(i)
      mm = e%kenr(i)


!------ density

      e%vden(i)=vdle(ii)
      e%vdent(i)=e%vdlev(ii)

!------ momentum
      e%qm1(i)  = vdle(jj)
      e%qm1t(i) = e%vdlev(jj)


      e%qm2(i)  = vdle(kk)
      e%qm2t(i) = e%vdlev(kk)


      e%qm3(i)  = vdle(ll)
      e%qm3t(i) = e%vdlev(ll)


!------ enrgy

      e%venr(i) = vdle(mm)
      e%venrt(i) = e%vdlev(mm)


!------ temperature

      gu = e%qm1(i)*e%qm1(i) + e%qm2(i)*e%qm2(i) + e%qm3(i)*e%qm3(i)
      e%vtem(i) = (e%venr(i) - gu/(2.0d0*e%vden(i)))/e%vden(i)

!------ pressure

      e%vpres(i) = gama1*e%vden(i)*e%vtem(i)
       if((e%vpres(i).le.zero).or.(e%vden(i).le.zero))then
        write(mp,*)' NODAL  DENSITY',dens
        write(mp,*)'  NODAL  PRESSURE ',pres
        call flush(mp)
      call fin_de_programme(2,'fichier :residu_general.f90')
      endif

!------ velocity

      e%veloc1(i) = e%qm1(i)/e%vden(i)
      e%veloc2(i) = e%qm2(i)/e%vden(i)
```

```
      e%veloc3(i) = e%qm3(i)/e%vden(i)
      enddo

!calculation of the maximum and the minumum distance between two nodes in the
element

      k=1
      do i=1,e%inel
      do j=1,e%inel
      e%vdis(k)=(vcore(3*i-2)-vcore(3*j-2))**2+ &
     &        (vcore(3*i-1)-vcore(3*j-1))**2+ &
     &        (vcore(3*i)-vcore(3*j))**2
      e%vdis(k)=sqrt(e%vdis(k))
      k=k+1
      enddo
      enddo

      helmax=maxval(e%vdis)

      do k=1,e%inel*e%inel
      if (abs(e%vdis(k)).eq.0.d0) then
      e%vdis(k)=helmax
      endif
      enddo
      helmin=minval(e%vdis)


!------ initialisation
      do i=1,idle
      e%vfes(i) = 0.d0
      vfe(i) = 0.d0
      enddo

!=========== tau matrix calculation with averaged values ====
        ii=0
        do i=1,5
        do j=1,5
        ii=ii+1
        tau(i,j)=vpree(10)*tauglob(ieeloc,ii)
        enddo
        enddo

!=======================================Turbulent viscosity
       vmu = 1.d0/vpree(1)


      if(probtype==5) then
      chi = 0.0d0
      dens=0.0d0
      do i=1,e%inel
       chi =  VTG(kne(i))
       dens=   e%vden(i)
      fv1=chi**3/(chi**3+cv1**3)
      nu_tield = chi*vmu
      nu_turb  = nu_tield*fv1
      e%vmut(i) = nu_turb*dens
      if(e%vmut(i).lt.zero)e%vmut(i)=0.01d0
      enddo
      endif
```

```
      is = 0

! loop over gauss points

       DO ig=1,e%ipg
!====== Jacobian matrix, its determinant and inverse

      ig1= (ig-1.d0)*e%inel*4.d0+e%inel+1.d0

      call JACOBH(e%vni(ig1),VCORE,e%ndim,e%inel,E%VJ,E%VJ1,DETJ)

      call DNIDXH(e%vni(ig1),E%VJ1,e%ndim,e%inel,E%VNIX0,E%VNIX,E%VNIY,E%VNIZ)



      if(detj.lt.eps) then
      write(mp,2040)ieeloc,itpe,detj
2040      format(/' ***ELEM ',2i8,' detj = ',e12.5)
      call fin_de_programme(2,'fichier :residu-Prism.f')
      endif

!===== Space derivatives of ============================

!----- density
      call  ScalarProduct(denx, e%vnix,e%vden,e%inel)
      call  ScalarProduct(deny, e%vniy,e%vden,e%inel)
      call  ScalarProduct(denz, e%vniz,e%vden,e%inel)

!----- Momentum
      call  ScalarProduct(u1x, e%vnix,e%qm1,e%inel)
      call  ScalarProduct(u1y, e%vniy,e%qm1,e%inel)
      call  ScalarProduct(u1z, e%vniz,e%qm1,e%inel)

      call  ScalarProduct(u2x, e%vnix,e%qm2,e%inel)
      call  ScalarProduct(u2y, e%vniy,e%qm2,e%inel)
      call  ScalarProduct(u2z, e%vniz,e%qm2,e%inel)

      call  ScalarProduct(u3x, e%vnix,e%qm3,e%inel)
      call  ScalarProduct(u3y, e%vniy,e%qm3,e%inel)
      call  ScalarProduct(u3z, e%vniz,e%qm3,e%inel)


!------ divergence of momentum
       divu = u1x + u2y + u3z

!----- Energy

      call  ScalarProduct(enrx, e%vnix,e%venr,e%inel)
      call  ScalarProduct(enry, e%vniy,e%venr,e%inel)
      call  ScalarProduct(enrz, e%vniz,e%venr,e%inel)

!----- Pressure

      call  ScalarProduct(prex, e%vnix,e%vpres,e%inel)
      call  ScalarProduct(prey, e%vniy,e%vpres,e%inel)
      call  ScalarProduct(prez, e%vniz,e%vpres,e%inel)

!------ temperature

      call  ScalarProduct(temx, e%vnix,e%vtem,e%inel)
      call  ScalarProduct(temy, e%vniy,e%vtem,e%inel)
```

```
      call  ScalarProduct(temz, e%vniz,e%vtem,e%inel)


!----- velocity



      gradx(1) = prex
      gradx(2) = u1x
      gradx(3) = u2x
      gradx(4) = u3x
      gradx(5) = enrx

      grady(1) = prey
      grady(2) = u1y
      grady(3) = u2y
      grady(4) = u3y
      grady(5) = enry

      gradz(1) = prez
      gradz(2) = u1z
      gradz(3) = u2z
      gradz(4) = u3z
      gradz(5) = enrz

      gradp(2) = prex
      gradp(3) = prey
      gradp(4) = prez
      gradp(1) = zero
      gradp(5) = zero

      ichoc=0

      is= (ig-1)*e%inel*(e%ndim+1)+1


!----- density

        call  ScalarProduct(dens, e%vni(is),e%vden,e%inel)
        call  ScalarProduct(denst, e%vni(is),e%vdent,e%inel)
        call  ScalarProduct(vmu_turb, e%vni(is),e%vmut,e%inel)

!----- momentum
        call  ScalarProduct(u1, e%vni(is),e%qm1,e%inel)
        call  ScalarProduct(u1t, e%vni(is),e%qm1t,e%inel)


        call  ScalarProduct(u2, e%vni(is),e%qm2,e%inel)
        call  ScalarProduct(u2t, e%vni(is),e%qm2t,e%inel)


        call  ScalarProduct(u3, e%vni(is),e%qm3,e%inel)
        call  ScalarProduct(u3t, e%vni(is),e%qm3t,e%inel)


!------ momentum norm
       uu = sqrt(u1*u1 + u2*u2 + u3*u3)

!------ energy

       call  ScalarProduct(enrg, e%vni(is),e%venr,e%inel)
```

```
       call  ScalarProduct(enrgt, e%vni(is),e%venrt,e%inel)


!------ pressure

       call  ScalarProduct(pres, e%vni(is),e%vpres,e%inel)

       if((pres.le.zero).or.(dens.le.zero))then
        write(mp,*)'   DENSITY',dens
        write(mp,*)'   PRESSURE ',pres
        call flush(mp)
        write(mp,2030)IEL, pres
2030      format(/' ***ELEM ',i8,' pmoyen = ',e12.5)
      call fin_de_programme(2,'fichier :residu-Prisme.f')
      endif

!------ temperature

       call  ScalarProduct(temp, e%vni(is),e%vtem,e%inel)

!--- shock capturing
       call  ScalarProduct(dcs, e%vni(is),LapE,e%inel)

!sepcial case grid veklocity is put zero

       w1=0.d0
       w2=0.d0
       w3=0.d0

!====== advection flux at gauss points==================================

         v1= u1/dens
         v2= u2/dens
         v3= u3/dens

         v1x= u1x/dens- u1*denx/(dens*dens)
         v1y= u1y/dens- u1*deny/(dens*dens)
         v1z= u1z/dens- u1*denz/(dens*dens)

         v2x= u2x/dens- u2*denx/(dens*dens)
         v2y= u2y/dens- u2*deny/(dens*dens)
         v2z= u2z/dens- u2*denz/(dens*dens)

         v3x= u3x/dens- u3*denx/(dens*dens)
         v3y= u3y/dens- u3*deny/(dens*dens)
         v3z= u3z/dens- u3*denz/(dens*dens)

         divv = v1x + v2y + v3z

         cel= sqrt(pres*gama/dens) !sound speed
         vnorm= sqrt( v1*v1+v2*v2+v3*v3)
         utau= vnorm + cel



      do i=1,e%ndln
      fadv(i) = 0.d0
      fdif1(i) = 0.d0
      fdif2(i) = 0.d0
      fdif3(i) = 0.d0
      enddo
```

```
      dc=0.d0
      dc1=0.d0
!=================================================================

        fadv(1) = divu

        fadv(2) = v1*divu + u1*v1x + u2*v1y + u3*v1z + prex

        fadv(3) = v2*divu + u1*v2x + u2*v2y + u3*v2z + prey

        fadv(4) = v3*divu + u1*v3x + u2*v3y + u3*v3z + prez

        fadv(5) = (enrg + pres)*divv  &
     &    + v1*(enrx + prex) + v2*(enry + prey) + v3*(enrz + prez)

!=================================================================
          Lambda= cel+vnorm
         Pec= (vnorm*helmin)/(2*(vmu+vmu_turb))
         zeta= min(1.0d0,Pec/3.d0)

      do i=1,e%ndln
      fstabx(i) = 0.d0
      fstaby(i) = 0.d0
      fstabz(i) = 0.d0
      enddo

!====== stabilization flux ==================================

      do i=1,e%ndln
      do j=1,e%ndln
      h1(i,j) = 0.d0
      h2(i,j) = 0.d0
      h3(i,j) = 0.d0
      enddo
      enddo



      dc = 0.d0

!compute Advection matrice: jacobian of Euler flux

      if(vpree(10).ne.zero)then
        xx11= helmin/utau
        tauc= 1.d0/((1/xx11 +12/(vpree(1)*helmin*helmin)))

!------ stabilization matrix ---------------------------------

       taua=zeta*0.5d0* (helmin *vnorm)/(cel**2+vnorm**2)
       call aimat(dens,u1,u2,u3,w1,w2,w3,enrg,gama,a1,a2,a3)

!------ stabilization matrix ---------------------------------

!compute Bj=Aj Tau

      do i=1,e%ndln
      do j=1,e%ndln
      do k=1,e%ndln
      h1(i,j) = h1(i,j) + a1(i,k)*tau(k,j)
      h2(i,j) = h2(i,j) + a2(i,k)*tau(k,j)
      h3(i,j) = h3(i,j) + a3(i,k)*tau(k,j)
```

```
      enddo
      enddo
      enddo

!compute Bj *Fadv

      do i=1,e%ndln
      do j=1,e%ndln
      fstabx(i) = fstabx(i) + h1(i,j)*fadv(j)
      fstaby(i) = fstaby(i) + h2(i,j)*fadv(j)
      fstabz(i) = fstabz(i) + h3(i,j)*fadv(j)
      enddo
      enddo

!========= compute chock capturing viscosity

        fadv(1) = denst+ fadv(1)

        fadv(2) = u1t+ fadv(2)

        fadv(3) = u2t+ fadv(3)

        fadv(4) = u3t+ fadv(4)

        fadv(5) = enrgt+fadv(5)



!dcs  averaged
           dcs2=0.d0
           do i=1,e%inel
           dcs2=(dcs2+LapP(kne(i)))
            enddo
           dcs2=dcs2/e%inel

        taua=  0.5d0*(helmin)/(cel+vnorm)

       dc1=(taua)*(vpree(7)*dcs2+0.1d0+vpree(9))
       dc=dc+(vpree(8)*dcs2+vpree(9))*dens*(vnorm*helmin/2.d0)



         dcv(1)= dc1
         dcv(2)= zero
         dcv(3)= zero
         dcv(4)= zero
         dcv(5)= zero

!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

       vmu = (1.d0/vpree(1))*( (temp)**0.76d0 )+dc

       s1 = (vmu+vmu_turb)  !molecular plus turbulent viscosity

       s2 = vmu*(gama/vpree(3))+( vmu_turb)*(gama/vpree(4))!heat diffusion

!------ viscous Stresses
        sig11 = s1*(deux*v1x - deuti*divv)
        sig22 = s1*(deux*v2y - deuti*divv)
        sig33 = s1*(deux*v3z - deuti*divv)
```

```
        sig12 = s1*(v1y + v2x)
        sig21 = sig12

        sig13 = s1*(v1z + v3x)
        sig31 = sig13

        sig23 = s1*(v2z + v3y)
        sig32 = sig23

!------ heat flux
        heat1 = - s2*temx
        heat2 = - s2*temy
        heat3 = - s2*temz

!====== diffusion flux ==================================

        fdif1(1) = 0.0d0
        fdif1(2) = sig11
        fdif1(3) = sig21
        fdif1(4) = sig31
        fdif1(5) = sig11*v1 + sig12*v2 + sig13*v3 - heat1

        fdif2(1) = 0.0d0
        fdif2(2) = sig12
        fdif2(3) = sig22
        fdif2(4) = sig32
        fdif2(5) = sig21*v1 + sig22*v2 + sig23*v3 - heat2

        fdif3(1) = 0.0d0
        fdif3(2) = sig13
        fdif3(3) = sig23
        fdif3(4) = sig33
        fdif3(5) = sig31*v1 + sig32*v2 + sig33*v3 - heat3

!=========== Residual vector ======================================

      coef = e%wpg(ig)*detj


!------ corresponding to continuity equation
       do i=1,e%inel
       is1=is-1+i
       e%vfes(i) = e%vfes(i) + coef*( &
     &   e%vni(is1)*fadv(1)         &
     &  + e%vnix(i)*fdif1(1) + e%vniy(i)*fdif2(1) + e%vniz(i)*fdif3(1) )
       enddo

!------ corresponding to x direction momentum equation
       do i=1,e%inel
       is1=is-1+i
       e%vfes(i+e%inel) = e%vfes(i+e%inel) + coef*( &
     & e%vni(is1)*fadv(2) &
     & +e%vnix(i)*fdif1(2) + e%vniy(i)*fdif2(2) + e%vniz(i)*fdif3(2))
       enddo

!------ corresponding to y direction momentum equation
       do i=1,e%inel
       is1=is-1+i
       e%vfes(i+2*e%inel) = e%vfes(i+2*e%inel) + coef*( &
     & + e%vni(is1)*fadv(3) &
     & + e%vnix(i)*fdif1(3) + e%vniy(i)*fdif2(3) + e%vniz(i)*fdif3(3) )
```

```
       enddo

!------ corresponding to z direction momentum equation
       do i=1,e%inel
       is1=is-1+i
       e%vfes(i+3*e%inel) = e%vfes(i+3*e%inel) + coef*( &
     & + e%vni(is1)*fadv(4) &
     & + e%vnix(i)*fdif1(4) + e%vniy(i)*fdif2(4) + e%vniz(i)*fdif3(4))
       enddo

!------ corresponding to energy equation
       do i=1,e%inel
       is1=is-1+i
       e%vfes(i+4*e%inel) = e%vfes(i+4*e%inel) + coef*( &
     &  + e%vni(is1)*fadv(5) &
     &  + e%vnix(i)*fdif1(5) + e%vniy(i)*fdif2(5) + e%vniz(i)*fdif3(5))
       enddo

!==== SUPG stabilization terms

      if(vpree(10).ne.zero)then
       do ib=1,e%ndln
       do i=1,e%inel
       e%vfes(i+ (ib-1)*e%inel)=e%vfes(i+ (ib-1)*e%inel)+coef*( &
     & e%vnix(i)*fstabx(ib) + e%vniy(i)*fstaby(ib) + e%vniz(i)*fstabz(ib))
       enddo
       enddo
       endif

!== chock capturing terms
       do ib=1,e%ndln
       do i=1,e%inel
       e%vfes(i+ (ib-1)*e%inel)= e%vfes(i+ (ib-1)*e%inel)+coef*dcv(ib)*( &
     & e%vnix(i)*gradp(ib) + e%vniy(i)*gradp(ib) + e%vniz(i)*gradp(ib))
       enddo
       enddo



!***** end of Gauss points loop ************************************
       enddo

         if(igre.eq.2)then
         call residu_front(vcore,vpree,vdle,vfe)
         do i=1,idle
         e%vfes(i)= e%vfes(i)+ vfe(i)
         enddo
         endif

!========== reorder  the residual vectors into vector vfe ==============

       do i=1,idle
       ii = e%kpok(i)
       vfe(i) = - e%vfes(ii)
       enddo
       tempicount =tempicount+1
!=====================================================================

     call e%Finalize

     end subroutine
```

3.3 Comparison with the flow-based programming

In flow-based programming, every subroutine has to be written individually to calculate the elemental matrix and residual for each type of element. The only difference between the subroutines for each type of element is how the shape functions and integration points are calculated. If there were a change in either subroutine, modifications would have to be made for all four elements. It would be hard to maintain consistency, and we do not need to rewrite the same code. By using OOP, thanks to polymorphism, we need only one subroutine that applies to all four types of elements. This will allow code reusability and simplify the process of making changes.

Although OOP is a superior programming method, it does not have the same level of performance as flow-based programming. Applying OOP to certain subroutines results in a increase in computation time. This may be caused by the allocation and deallocaton of the variables. We noticed that we put most of the variables in the inheritance classes of the four elements. Each time a subroutine was called to calculate an element matrix or element residual, we had to construct an object of four derived types. There was an allocation of the variables and, subsequently, a deallocation of the allocated variables. Because of the complexity of our problem, many variables required significant memory. Repeated allocation and deallocation could be the cause of the slower computation. One change we can make is to place the variables back in the subroutine and avoid the use of allocation and deallocation.

CHAPTER 4

NUMERICAL RESULTS

4.1 Introduction

In this chapter, we run our code on two models. The first example consists of a turbulent flow over the 3D Naca0012 model. We use Mach number 0.15 and Reynolds number 2.8×10^6. The flow is essentially incompressible. Three angles of attack are examined: 0°, 10° and 15°. We compare our results with those of the experiment and the results obtained by CFX. The second example consists of a turbulent flow over the DLR F11 model. We use Mach number 0.2, Reynolds number 4.3×10^6 and angle of attack 13°. We compare our results with those of other numerical and experimental results.

4.2 NACA0012

NACA 0012 airfoil has a maximum thickness of 12% at 30% chord length from the leading edge. It has no camber, so it is symmetrical. There are many experimental and numerical references for this airfoil thus it is convenient for us to validate our code.

4.2.1 Case 1 ($Re=2.88\times10^6$, M=0.15, $\alpha = 0^\circ, 10^\circ$ and 15°, hybrid mesh)

Here we test a hybrid mesh and a mesh which consists of tetras only. The hybrid mesh is composed of 2 008 211 elements in total. It includes 691 516 tetras, 1 309 891 prisms and 6 804 pyramids. It has 839 257 nodes in total. The hybrid mesh is shown in Figure 4.1.

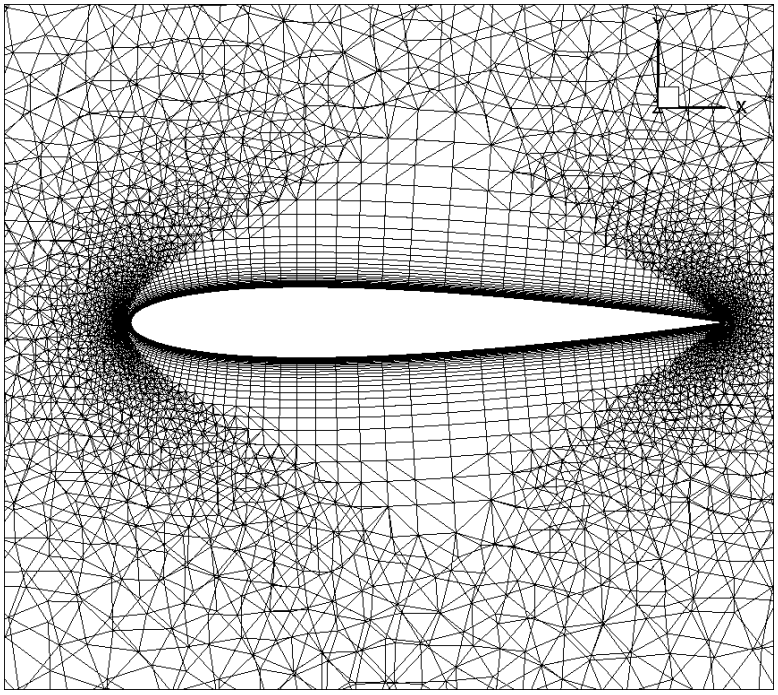


Figure 4.1 Mesh around the airfoil (hybrid mesh)

The resources used for the hybrid mesh are as follows:

Computer Platform: Guillimin compute cluster;
Number of processors: 40 (32 for fluid, 8 for turbulence);
Operating system: Unix;
Compiler: mvapich2/1.9-pgi and pgi/12.10;
Run time wall limit: 45 h;
Memory requirement: 2700 MB per processor.

Solution strategy:

To obtain a converged numerical solution and achieve the best accuracy, we must carefully set appropriate values of variable ε in the shock capturing stabilization matrix and the shock capturing viscosity. A large ε evades possible divergence of the numerical solution, but we will have a noticeable discrepancy. It is also difficult to obtain a converged solution if ε is too

small. To address this dilemma, we first set a large ε then decrease it to a smaller value. For all three angles of attack, we set $\varepsilon = 1.0$ at the first time step and decrease it to a small value after we achieve good convergence. For 0° and 10°, the change is sudden. For 15°, we gradually decrease ε . Ideally, ε should be zero, but it is much more difficult for the code to converge, especially for high angles of attack such as 10° and 15°. Currently, we decrease ε to 0.05.

To ensure that the flow stays slightly compressible, we use a low Mach number. The Reynolds number is almost 3 million, so the boundary layer is turbulent. The results are obtained after more than 3000 time steps, and the final residual produced by the parallel GMRES goes as low as 10^{-4}. In the following figures, we present the density, χ of the turbulence model, the pressure, the velocity around the trailing edge and the residual.

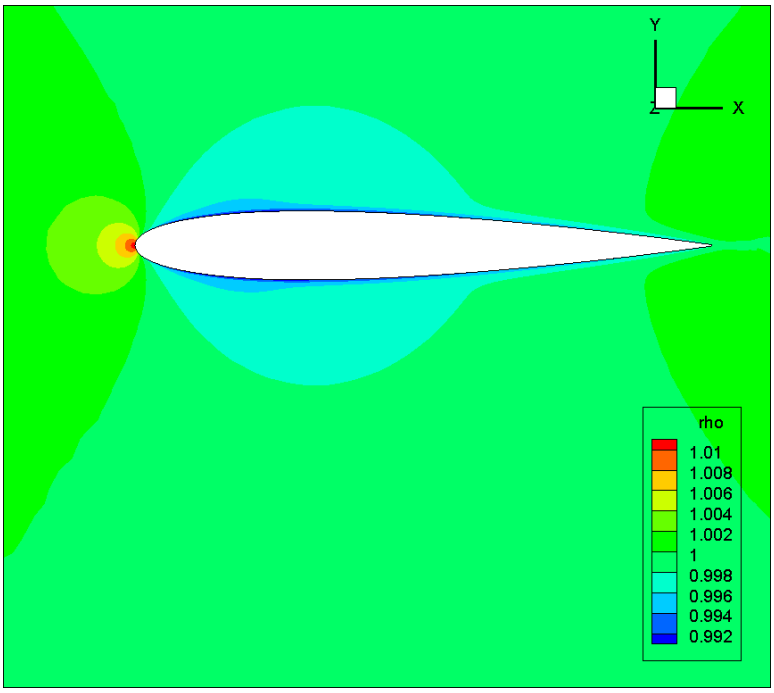


Figure 4.2 Density (M=0.15, Re=2.88×10^6, $\alpha = 0°$)

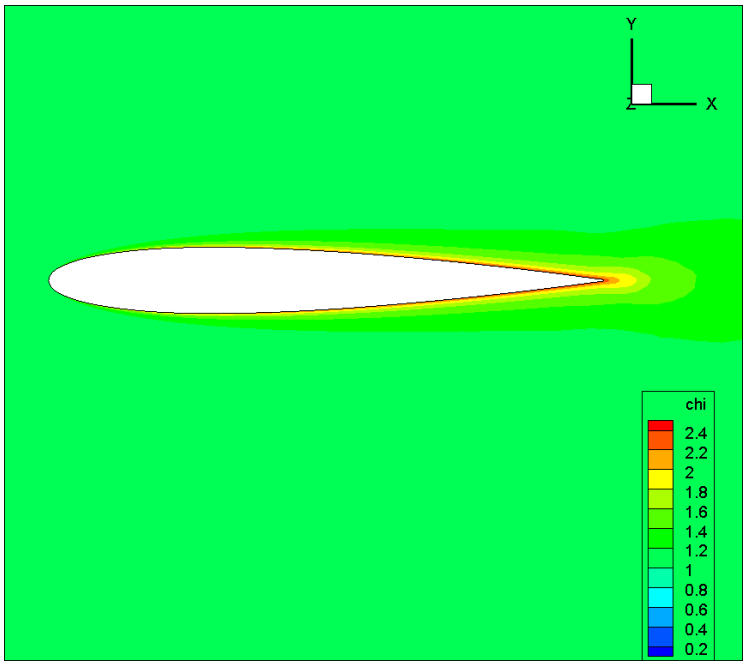


Figure 4.3 χ (M=0.15, Re=2.88×10^6, α = 0°)

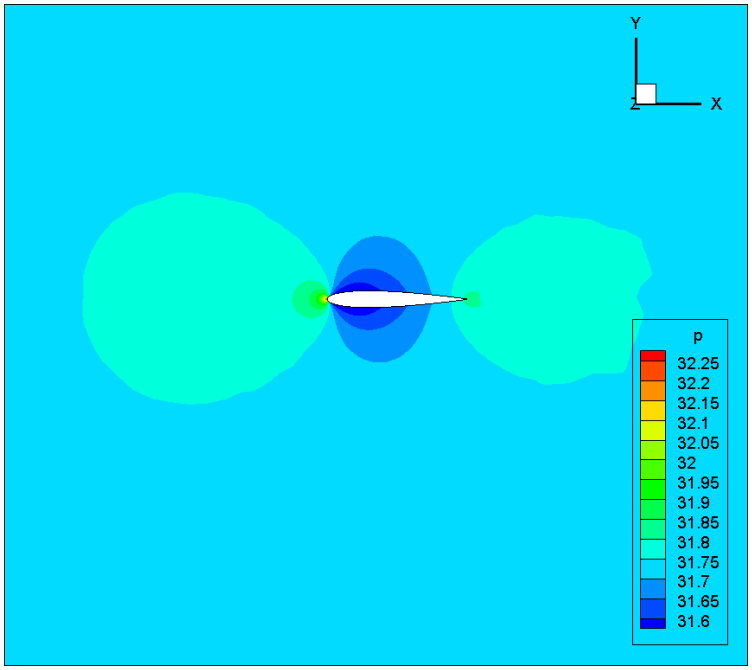


Figure 4.4 Pressure (M=0.15, Re=2.88×10^6 , α = 0°)

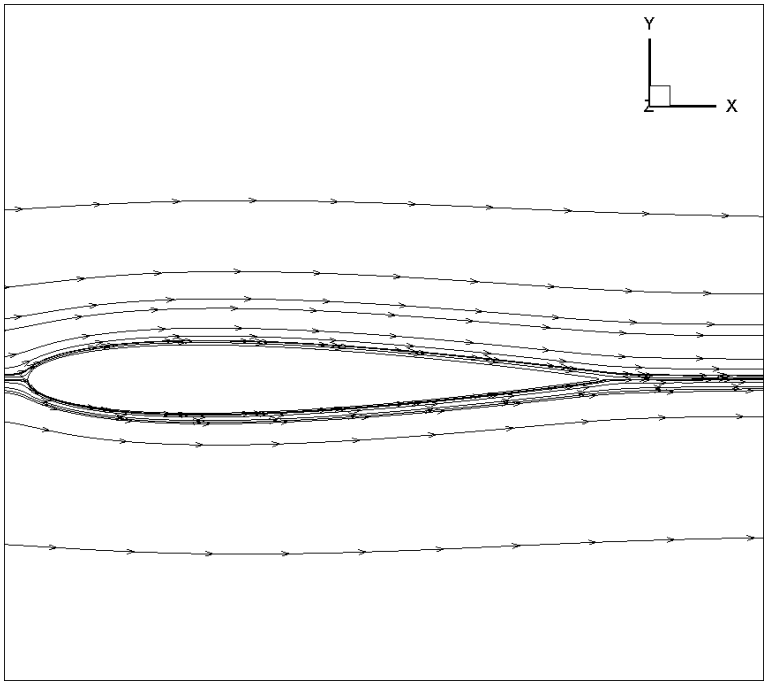


Figure 4.5 Velocity (M=0.15, Re=2.88×10^6, α = 0°)

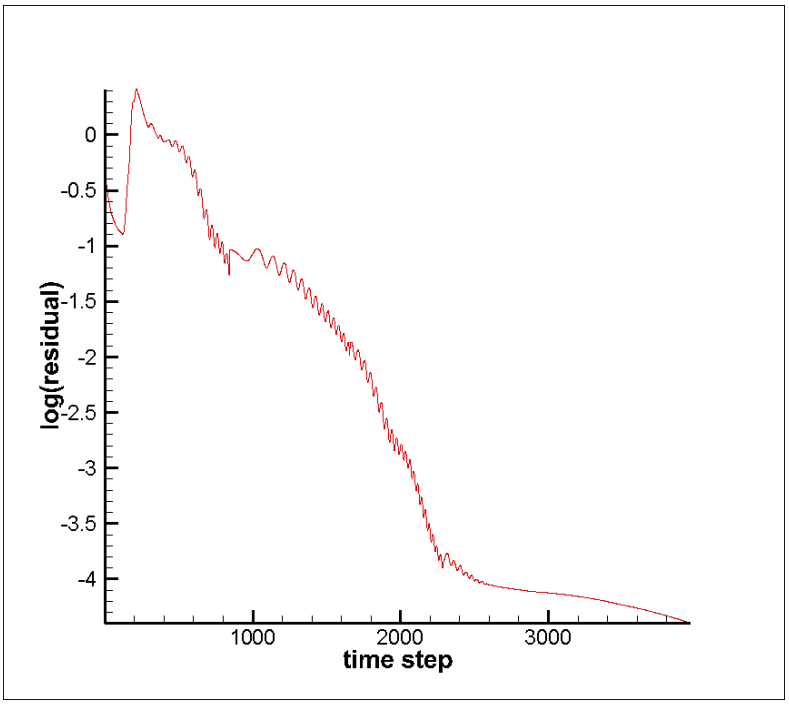


Figure 4.6 Evolution of residual with time

(M=0.15, Re=2.88×10^6, α = 0°)

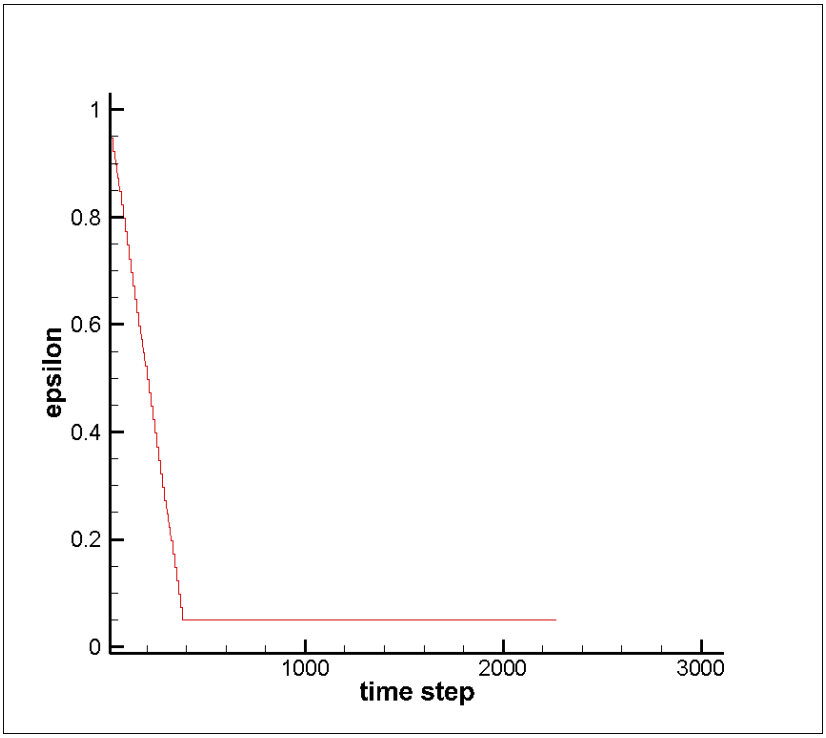


Figure 4.7 Evolution of ε with time

(M=0.15, *Re*=2.88×10^6 , α = 0°)

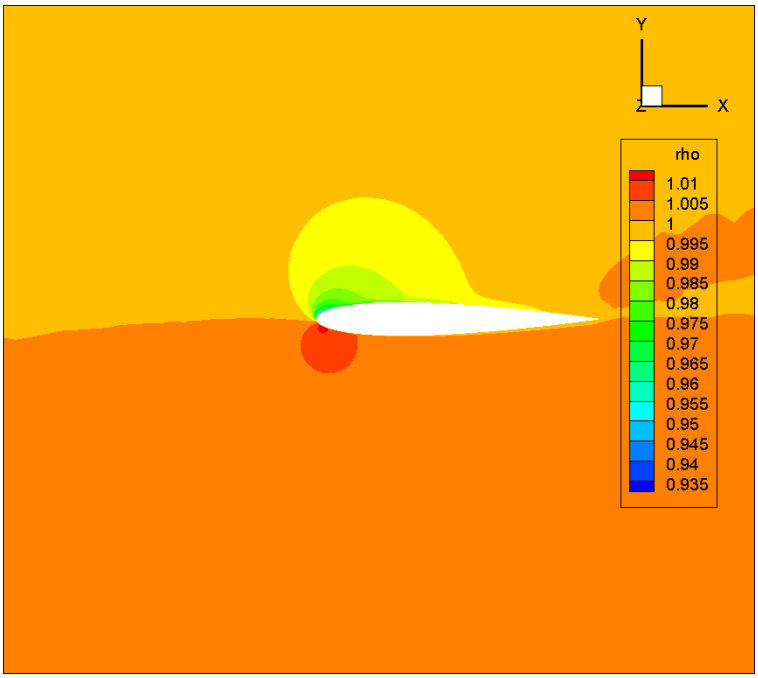


Figure 4.8 Density (M=0.15, *Re*=2.88×10^6 , α = 10°)

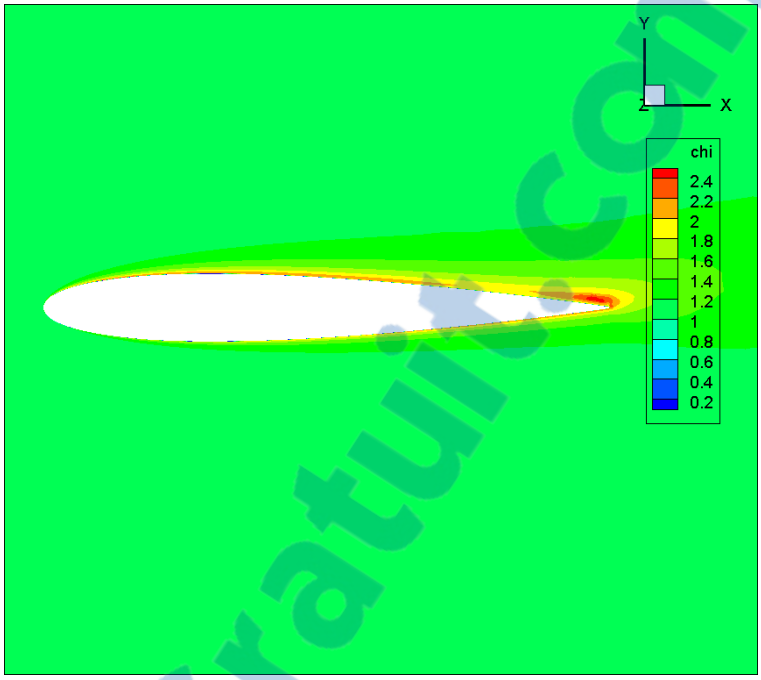


Figure 4.9 χ (M=0.15, *Re*=2.88×10^6 , $\alpha = 10°$)

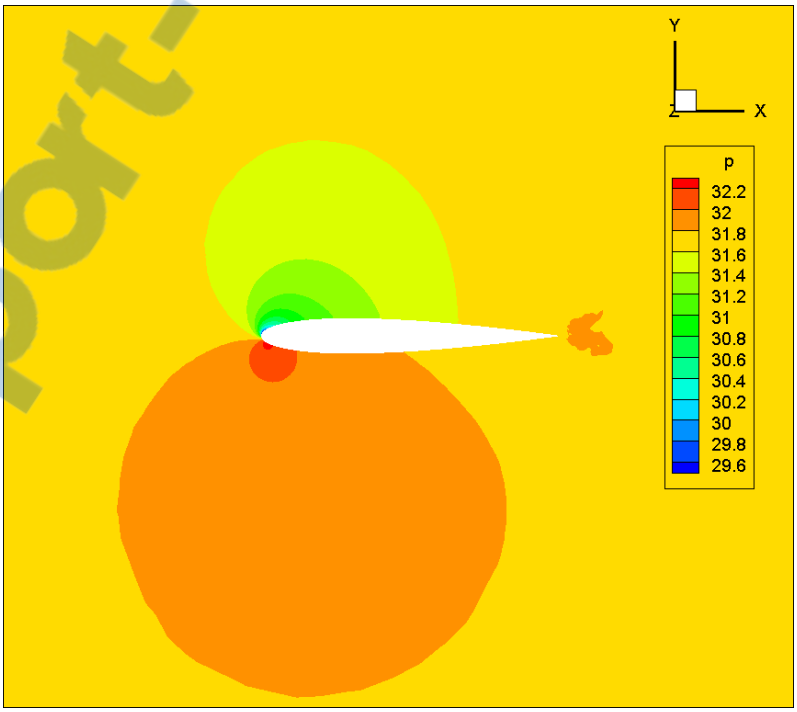


Figure 4.10 Pressure (M=0.15, *Re*=2.88×10^6, $\alpha = 10°$)

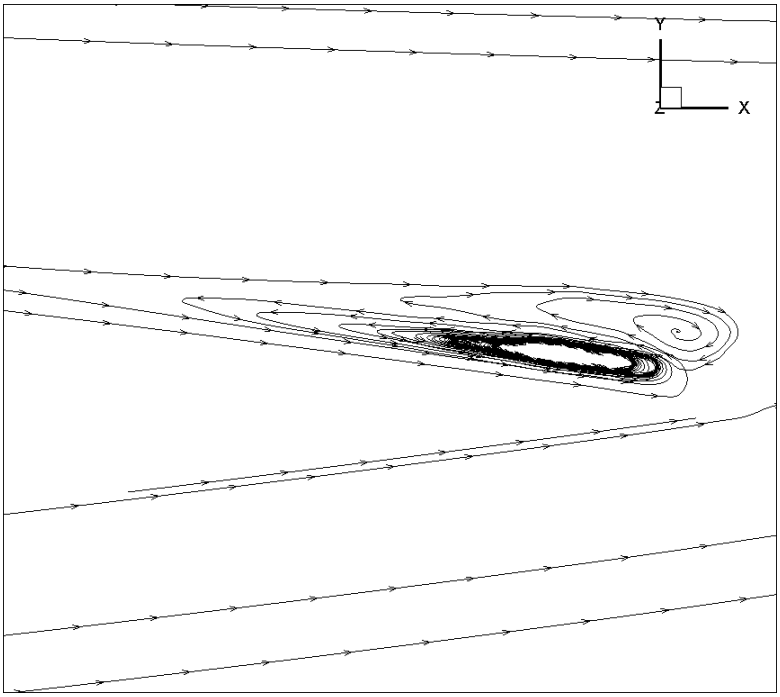


Figure 4.11 Velocity (M=0.15, Re=2.88×10^6, α = 10°)

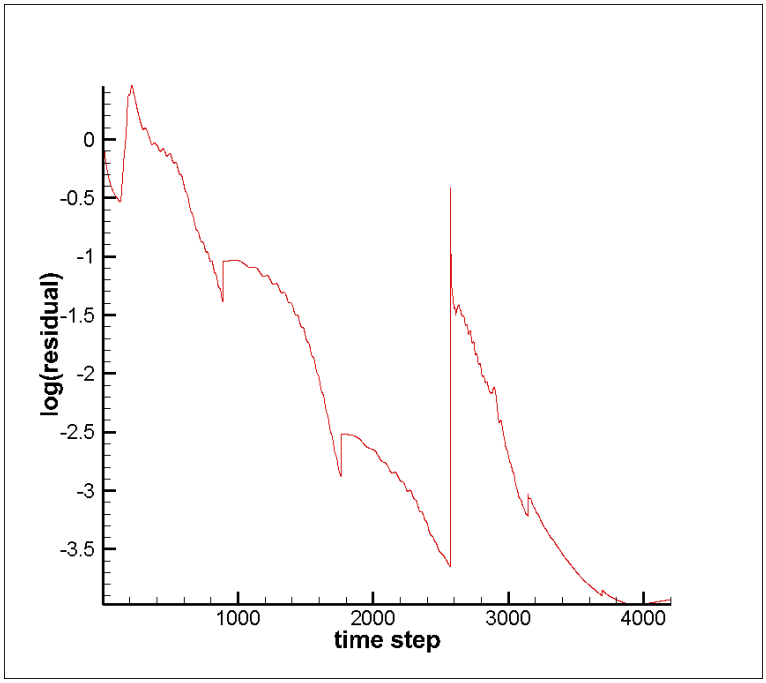


Figure 4.12 Evolution of residual with time

(M=0.15, Re=2.88×10^6, α = 10°)

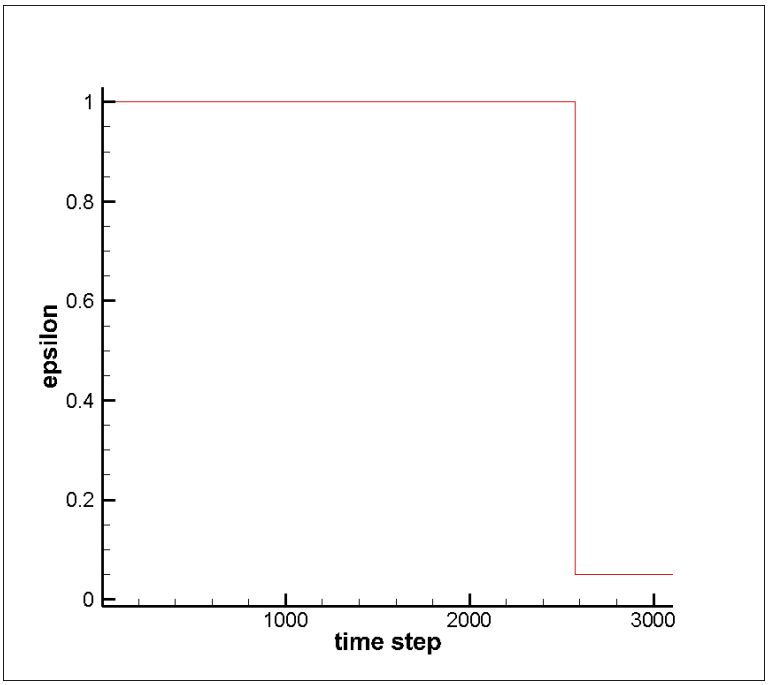


Figure 4.13 Evolution of ε with time

(M=0.15, Re=2.88×10^6 , α = 10°)

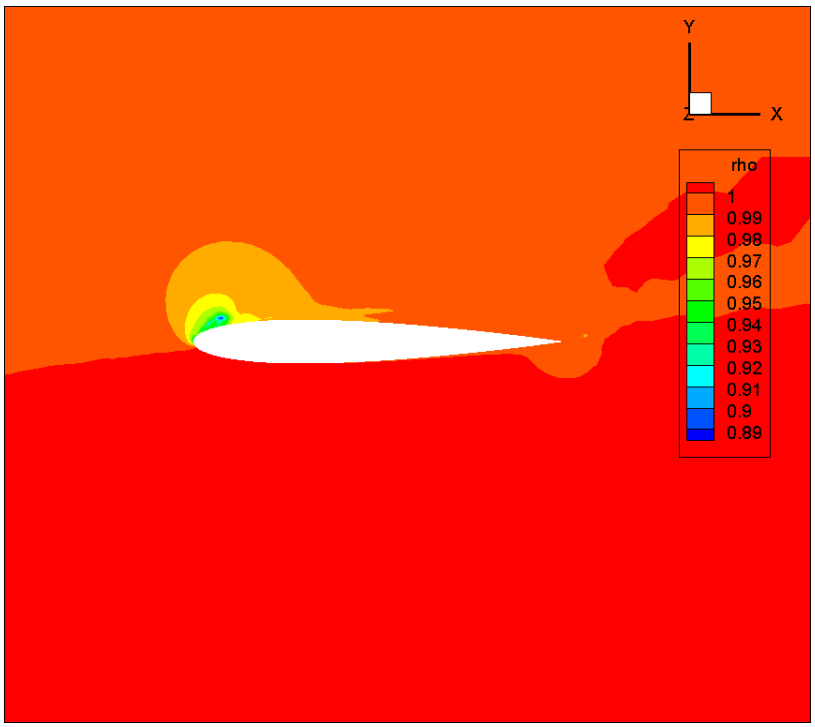


Figure 4.14 Density (M=0.15, Re=2.88×10^6, α = 15°)

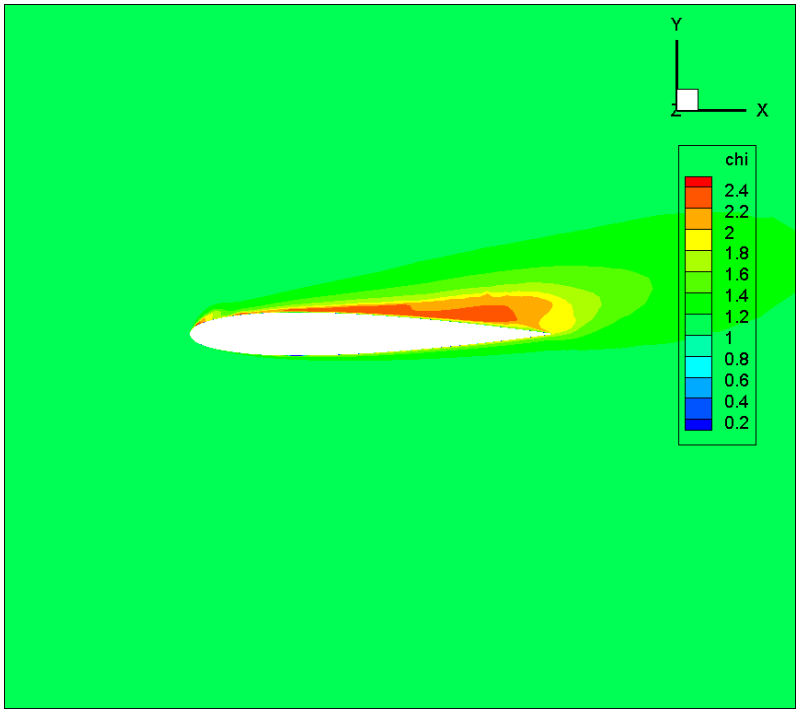


Figure 4.15 χ (M=0.15, Re=2.88×10^6, α = 15°)

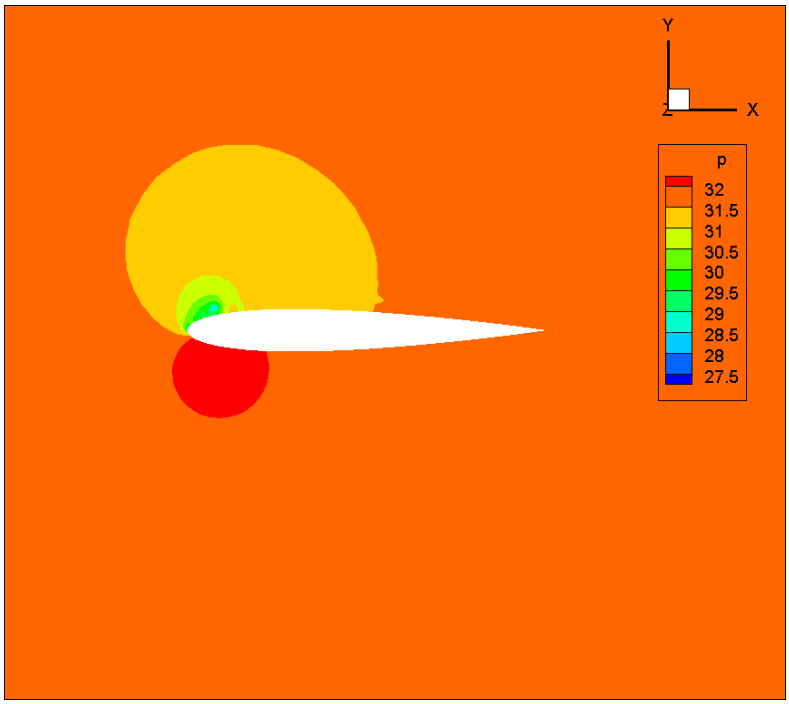


Figure 4.16 Pressure (M=0.15, Re=2.88×10^6 , α = 15°)

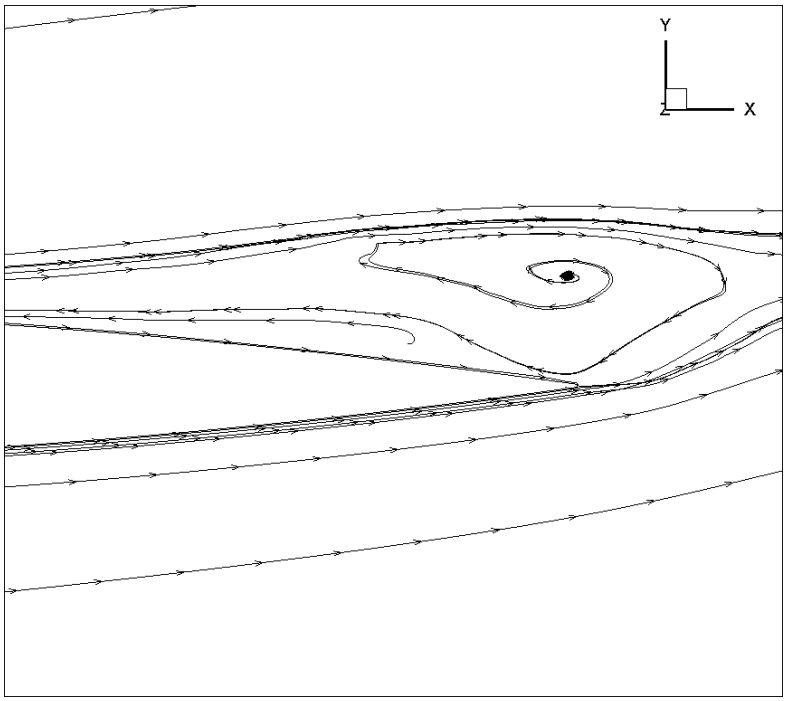


Figure 4.17 Velocity (M=0.15, Re=2.88×10^6 , α = 15°)

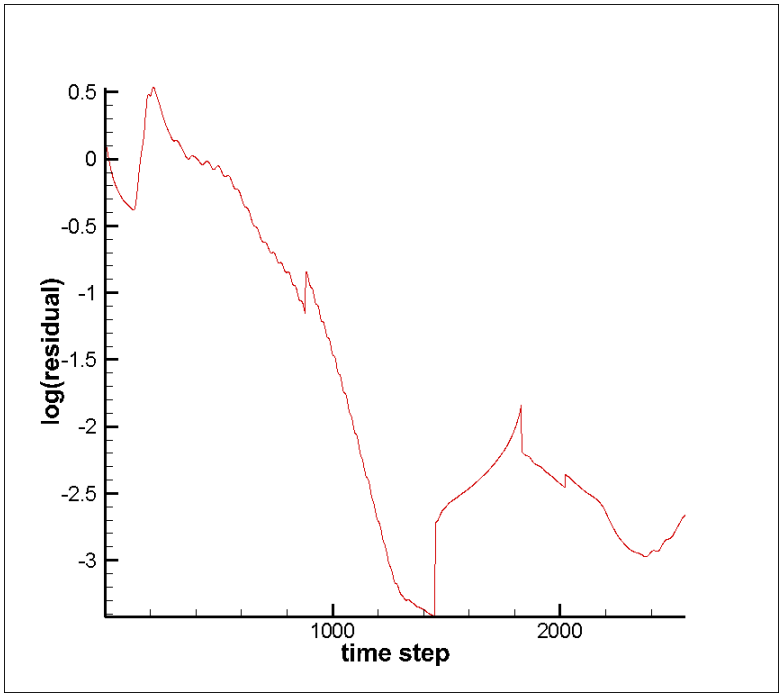


Figure 4.18 Evolution of residual with time

(M=0.15, Re=2.88×10^6, α = 15°)

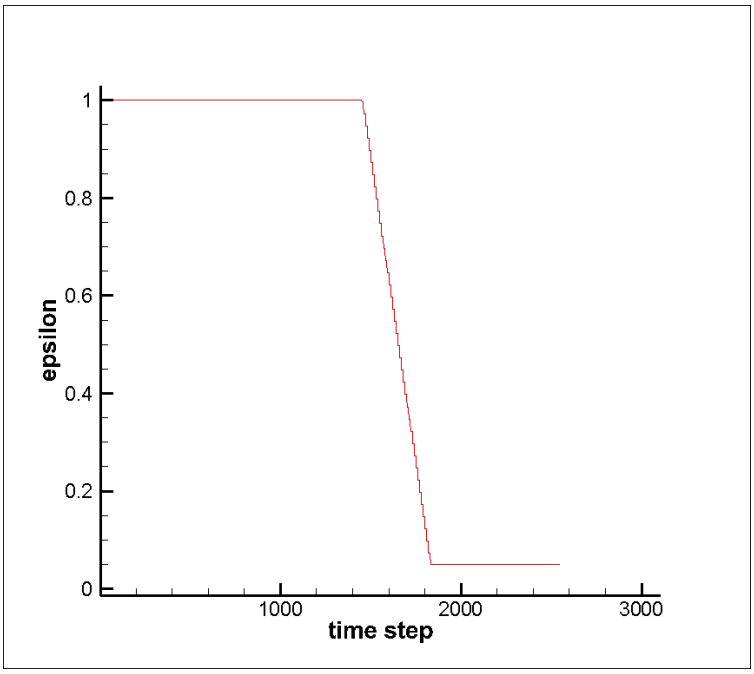


Figure 4.19 Evolution of ε with time
(M=0.15, Re=2.88×10^6, α = 15°)

For 0°, the plot is symmetric for every variable. For 10° and 15°, we can notice an increase of turbulent viscosity above the model and some vorticities around the trailing edge. Because of the solution strategy, we can notice a jump of residual when we start to decrease ε. For 0° and 10°, we can manage to achieve good convergence. For 15°, it is hard to get a steady solution after ε is decreased. ε needs to be set at a high value in order to avoid divergence. This greatly impacts the accuracy of the solution because we add too much diffusion. Here we reduce ε to 0.05 and we get this result before the solution diverges too much.

4.2.2 Case 2 (*Re*=2.88×10^6, *M*=0.15, α = 0°, 10° and 15°, tetra mesh)

The mesh which only consists of tetras is composed of 4 634 797 tetra elements. It has the same number of nodes as the hybrid mesh. We use a relatively large far-field boundary to eliminate the influence on drag and lift. The elements are denser in the boundary layer,

recirculation zones and turbulence zones. The tetra mesh is shown in Figure 4.20. We use the same solution strategy as the hybrid mesh to achieve good convergence.

The resources used for the tetra mesh are as follows:

Computer Platform: Guillimin compute cluster;
Number of processors: 40 (32 for fluid, 8 for turbulence);
Operating system: Unix;
Compiler: mvapich2/1.9-pgi and pgi/12.10;
Run time wall limit: 45 h;
Memory requirement: 2700 MB per processor.

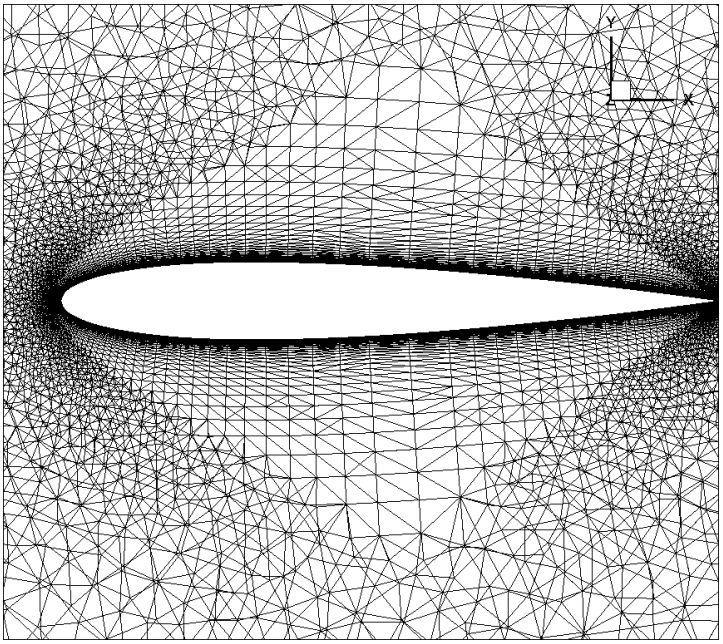


Figure 4.20 Mesh around the airfoil (tetra mesh)

In the following figures, we present the density, χ of the turbulence model, the pressure, the velocity around the trailing edge and the residual.

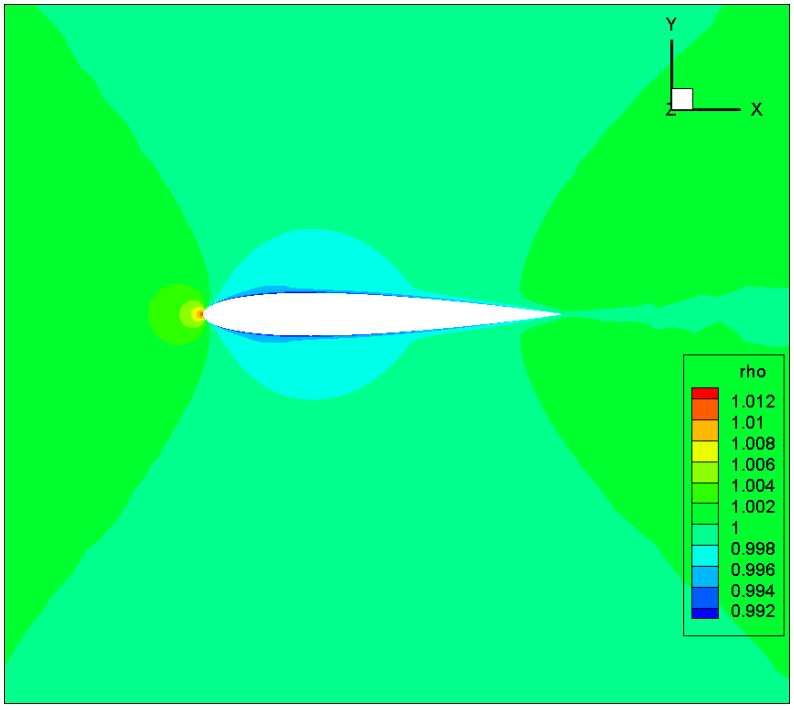


Figure 4.21 Density (M=0.15, Re=2.88×10^6 , α = 0°)

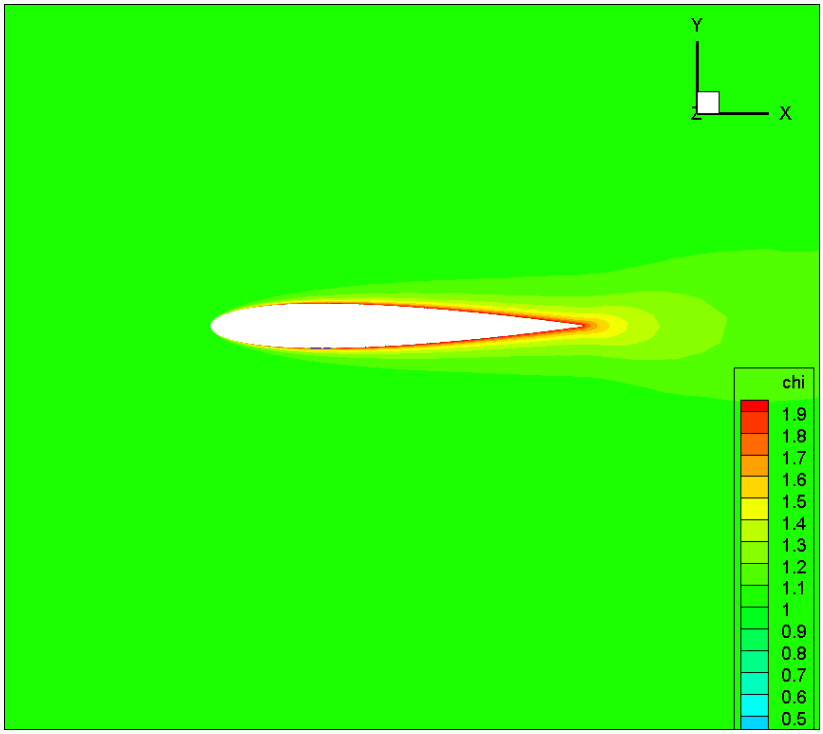


Figure 4.22 χ (M=0.15, Re=2.88×10^6, α = 0°)

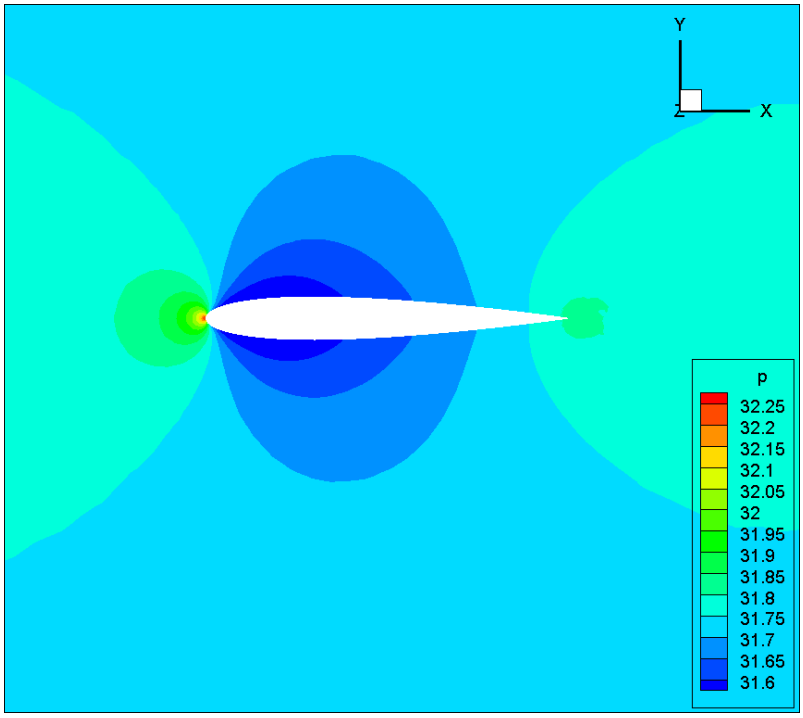


Figure 4.23 Pressure (M=0.15, Re=2.88×10^6, α = 0°)

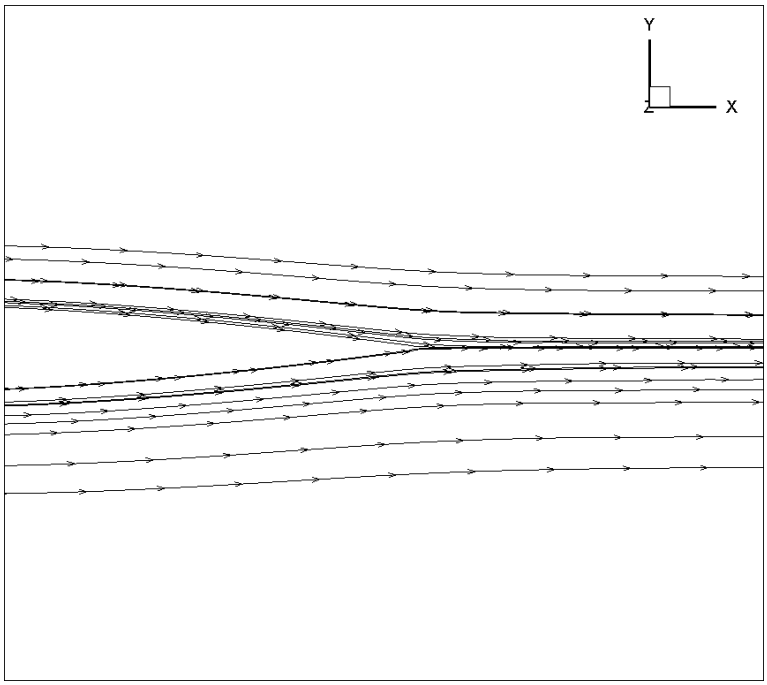


Figure 4.24 Velocity (M=0.15, Re=2.88×10^6, α = 0°)

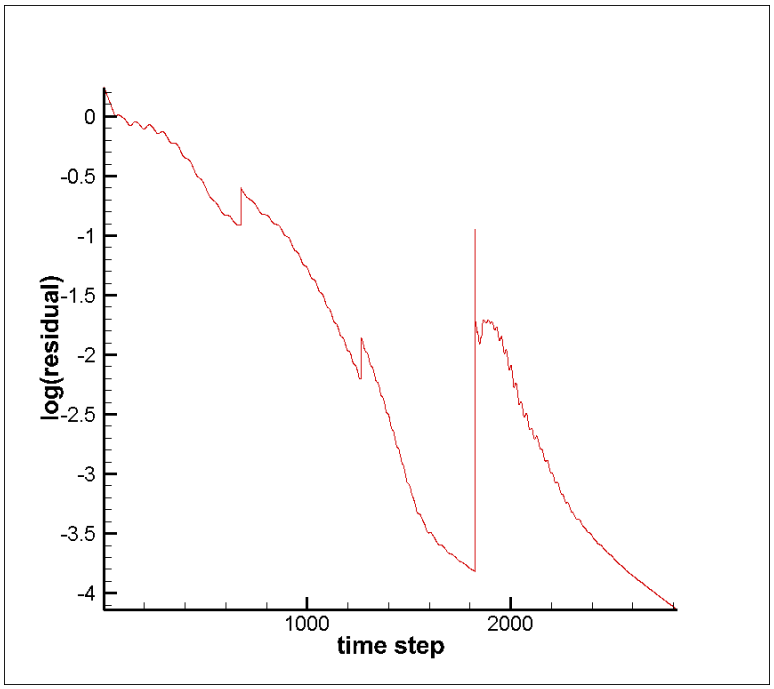


Figure 4.25 Evolution of residual with time

(M=0.15, Re=2.88×10^6, α = 0°)

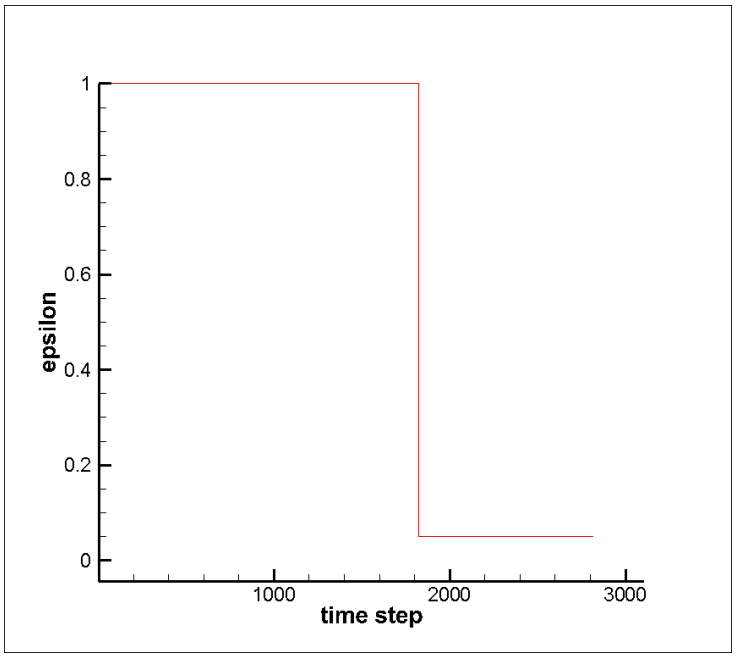


Figure 4.26 Evolution of ε with time

(M=0.15, Re=2.88×10^6 , α = 0°)

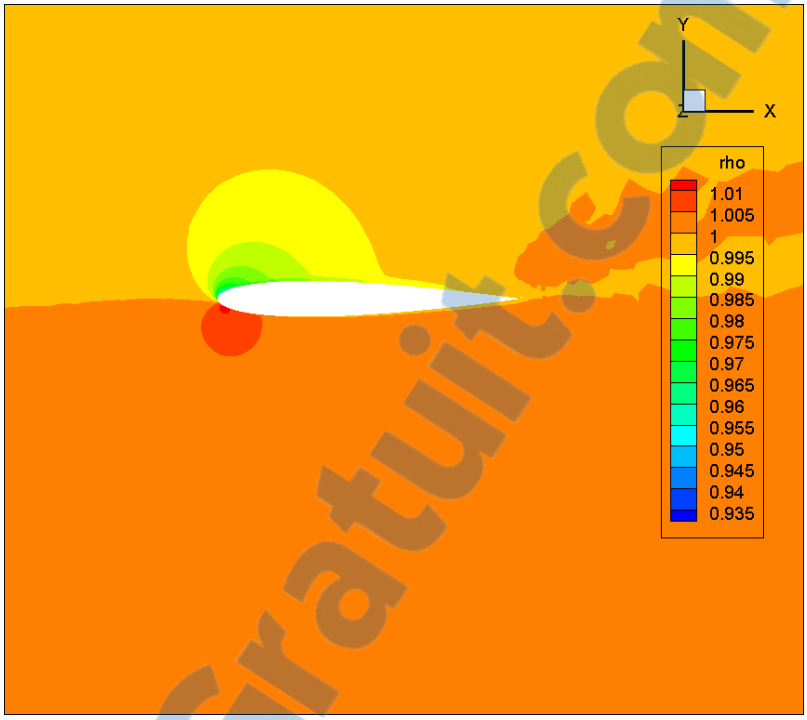


Figure 4.27 Density (M=0.15, Re=2.88×10^6, α = 10°)

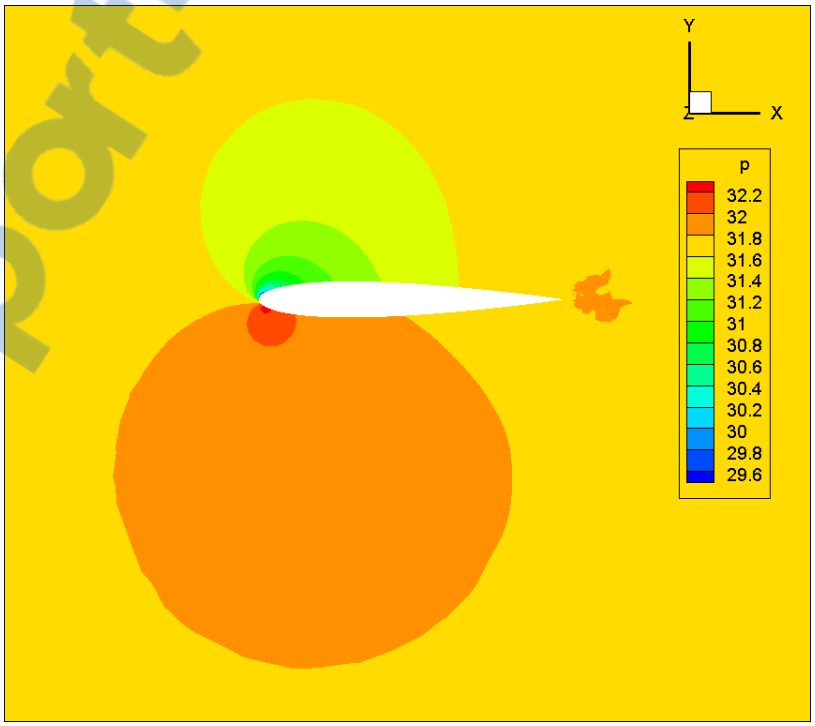


Figure 4.28 Pressure (M=0.15, Re=2.88×10^6 , α = 10°)

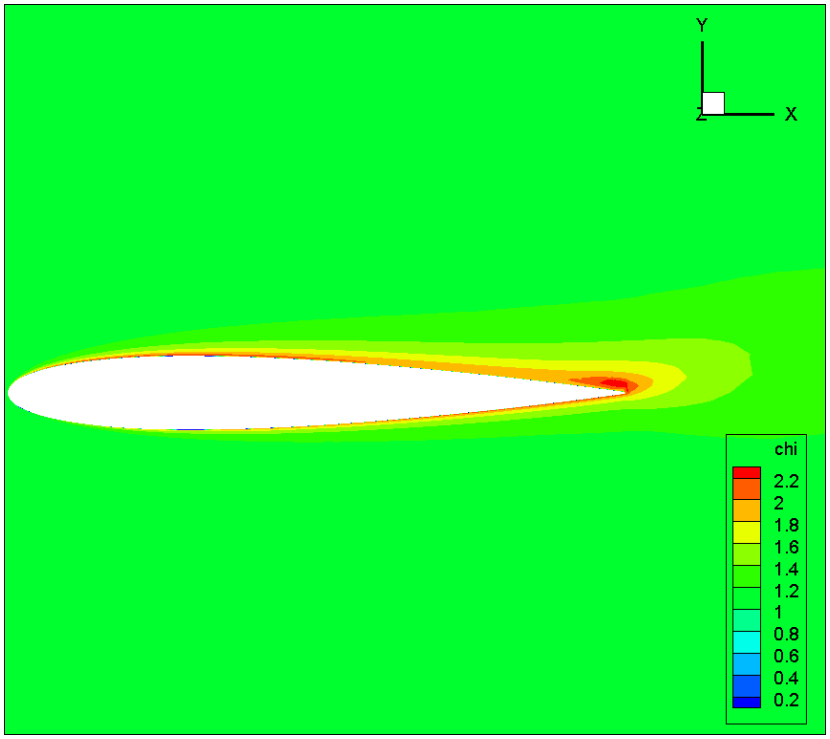


Figure 4.29 χ (M=0.15, Re=2.88×10^6, α = 10°)

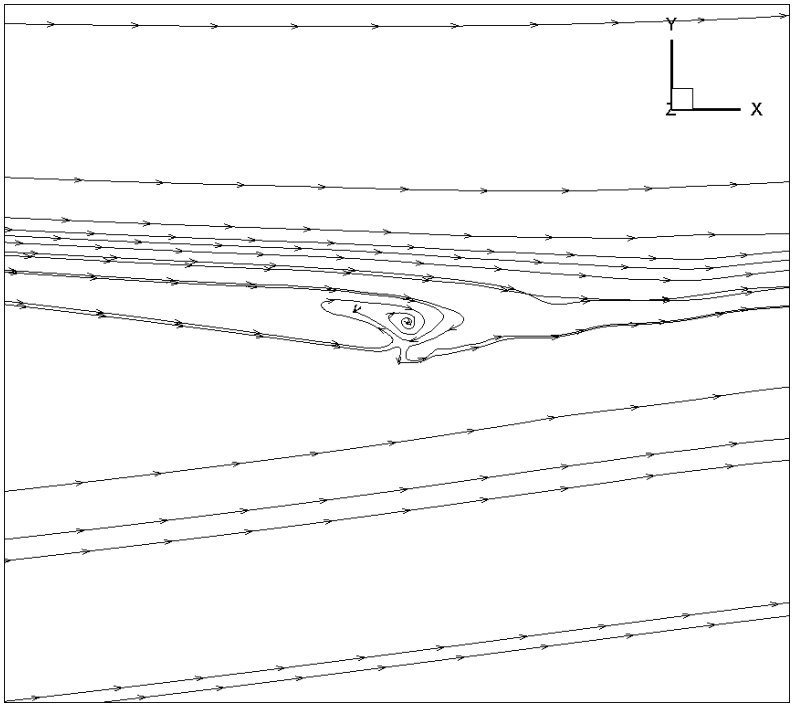


Figure 4.30 Velocity (M=0.15, Re=2.88×10^6, α = 10°)

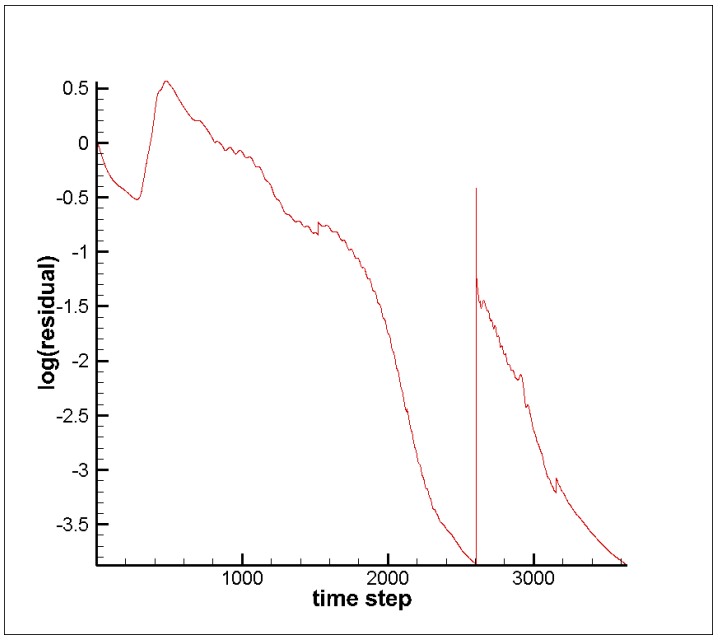


Figure 4.31 Evolution of residual with time

(M=0.15, Re=2.88×10^6, α = 10°)

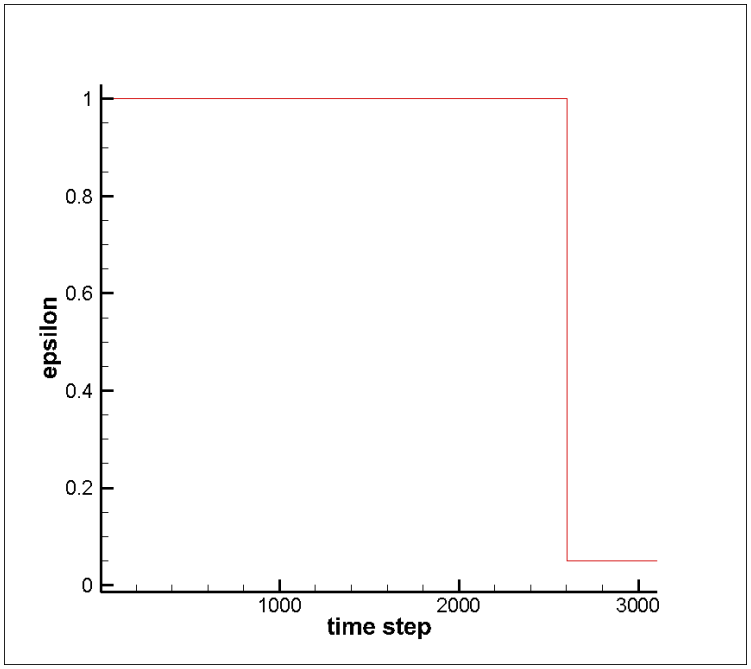


Figure 4.32 Evolution of ε with time

(M=0.15, Re=2.88×10^6 , α = 10°)

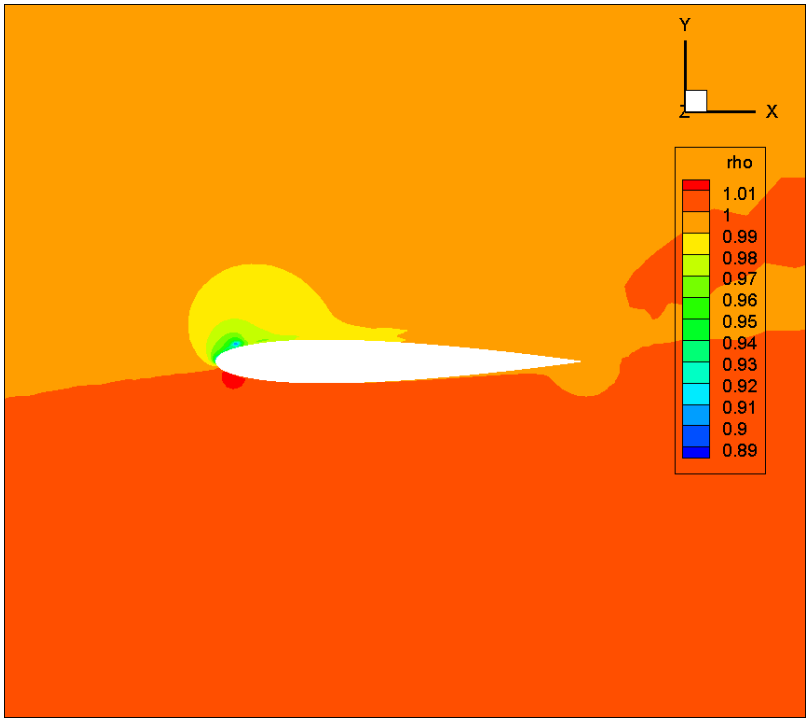


Figure 4.33 Density (M=0.15, Re=2.88×10^6, α = 15°)

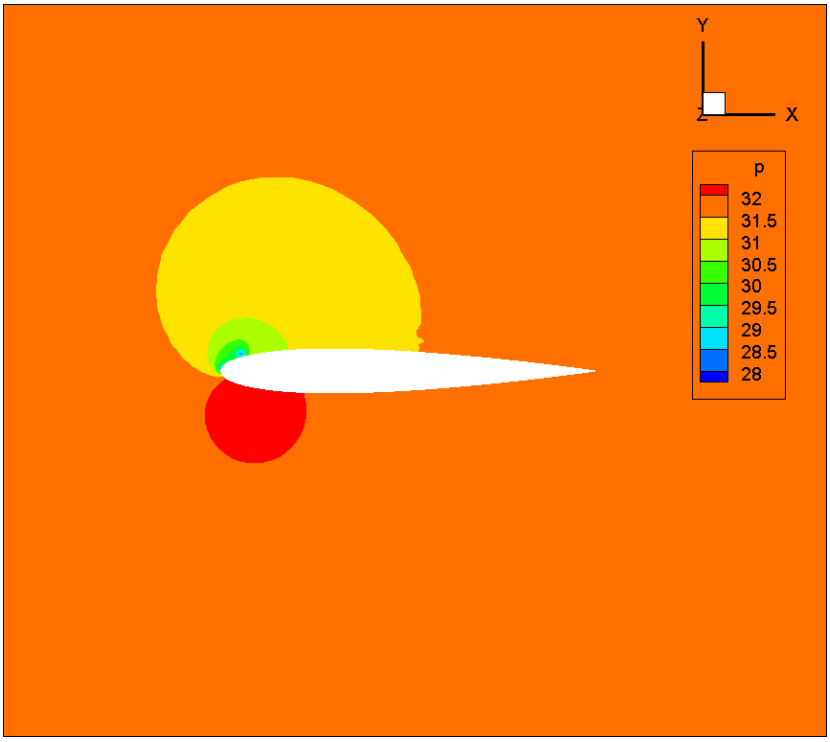


Figure 4.34 Pressure (M=0.15, Re=2.88×10^6, α = 15°)

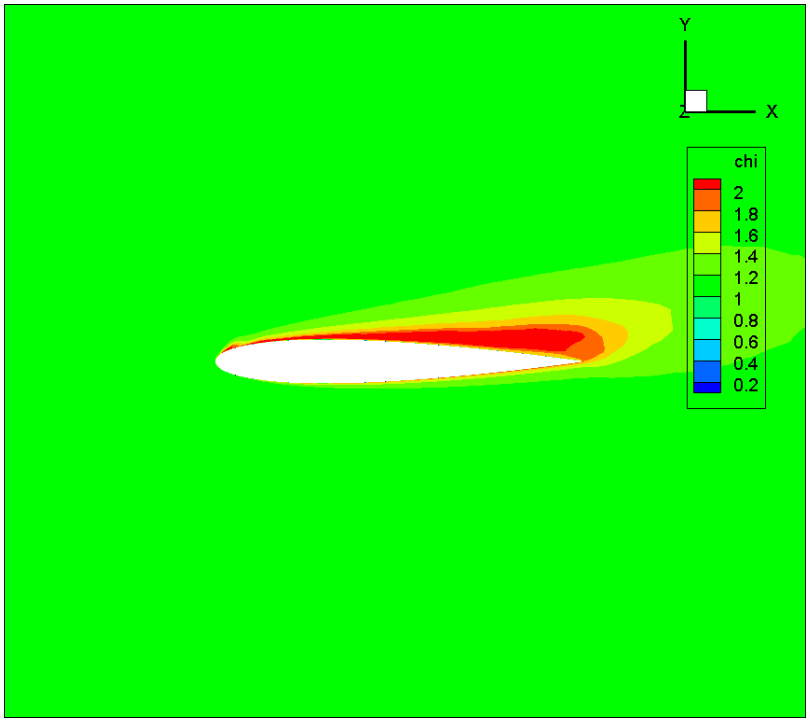


Figure 4.35 χ (M=0.15, Re=2.88×10^6, α = 15°)

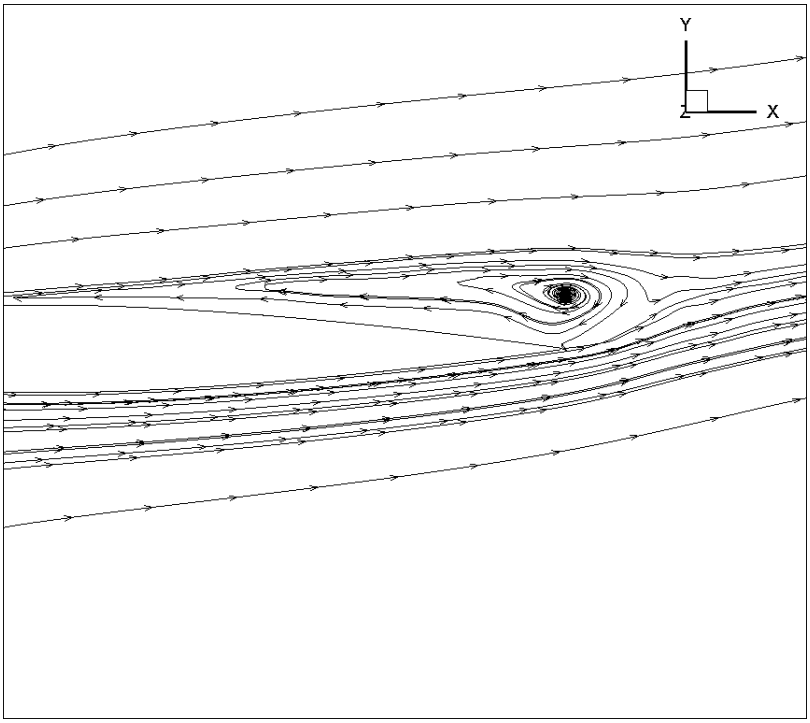


Figure 4.36 Velocity (M=0.15, Re=2.88×10^6 , α = 15°)

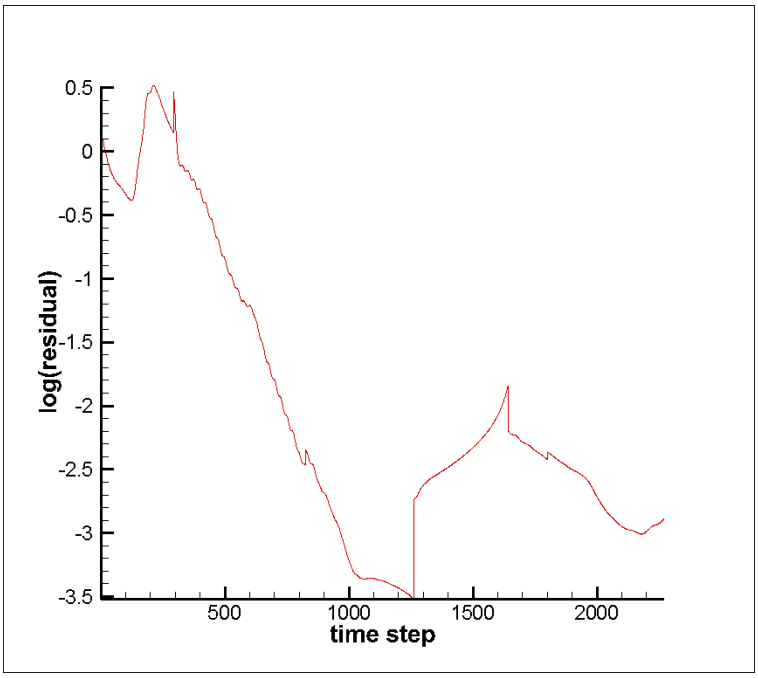


Figure 4.37 Evolution of residual with time

(M=0.15, Re=2.88×10^6 , α = 15°)

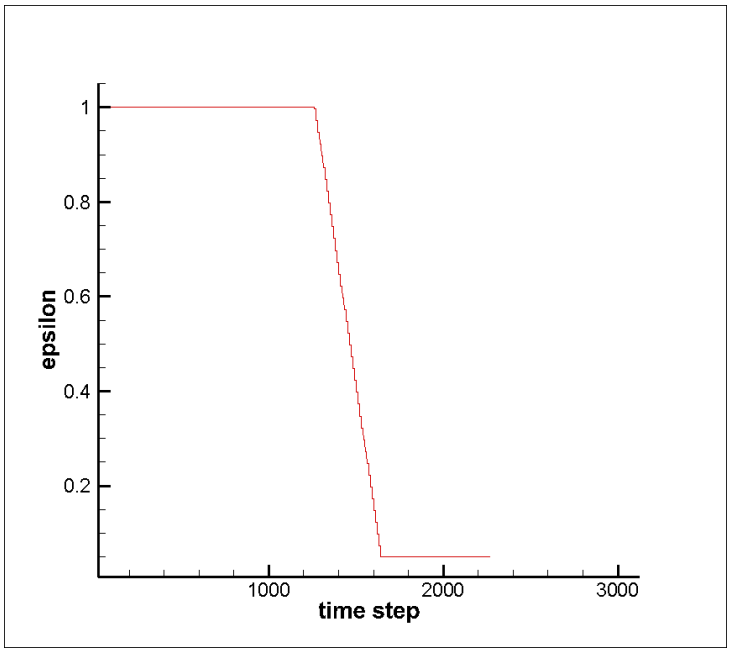


Figure 4.38 Evolution of ε with time

(M=0.15, Re=2.88×10^6 , α = 15°)

The plots of the variables of the tetra mesh are very similar to those of the hybrid mesh. It takes almost the same time for the tetra mesh to get a converged solution as for the hybrid mesh.

4.2.3 Comparison between the tetra mesh and hybrid mesh

In this section we compare the Cp contours of the hybrid mesh and the tetra mesh with the experimental results and the CFX results. The CFX results are obtained using the same tetra mesh and hybrid meshes. We also make comparisons of the lift coefficients. The Cp contours for the three angles of attack are shown as follows:

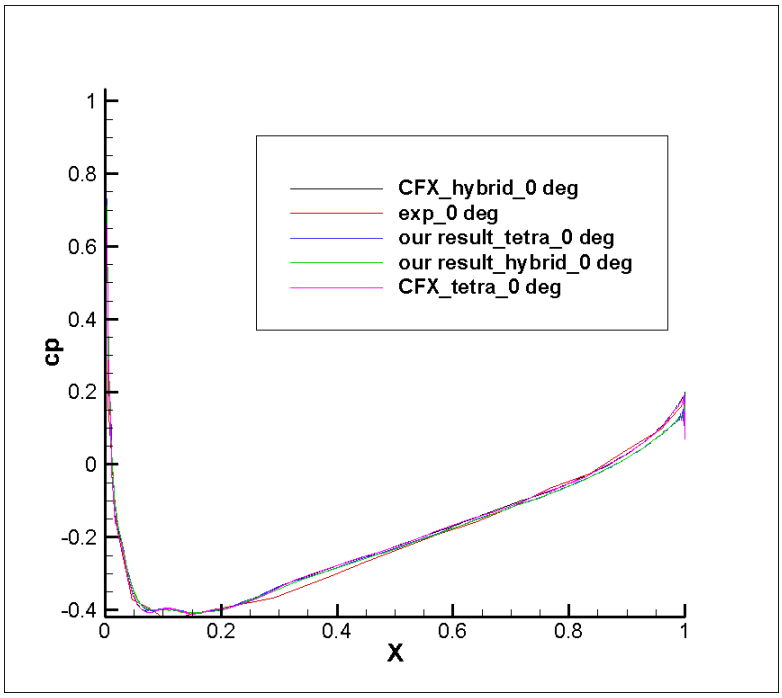


Figure 4.39 Cp (M=0.15, Re=2.88×10^6 , α = 0°)

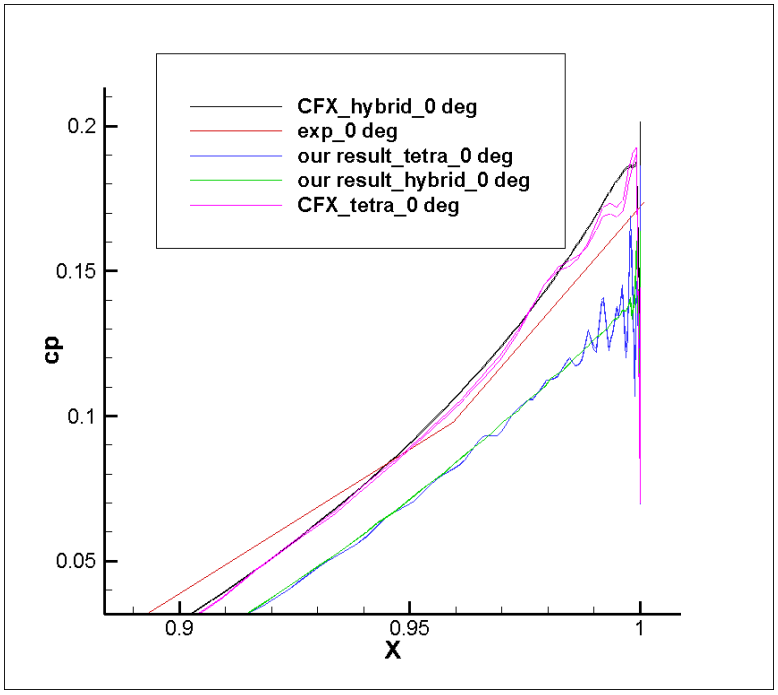


Figure 4.40 Cp around the trailing edge

(M=0.15, Re=2.88×10^6, α = 0°)

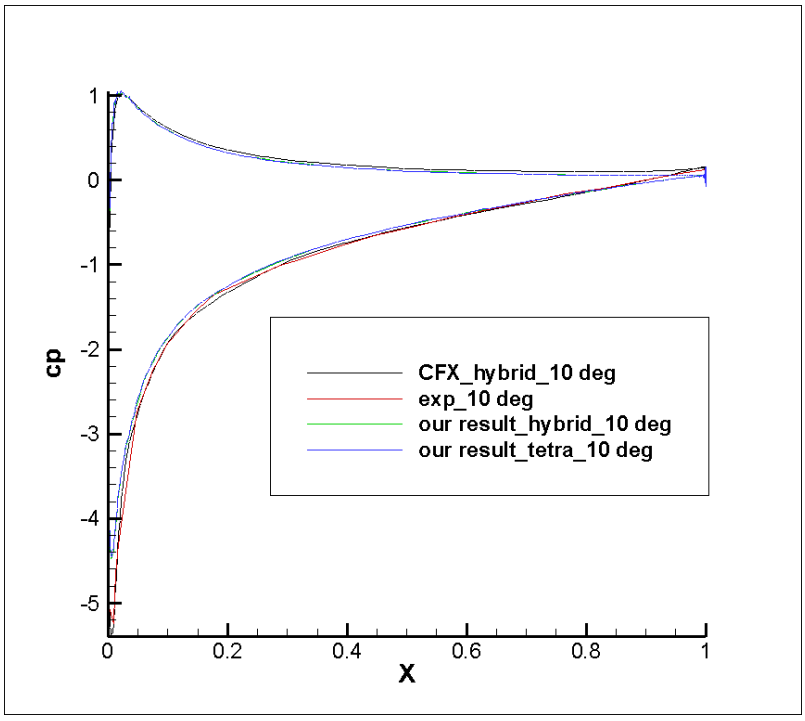


Figure 4.41 Cp (M=0.15, Re=2.88×10^6, α = 10°)

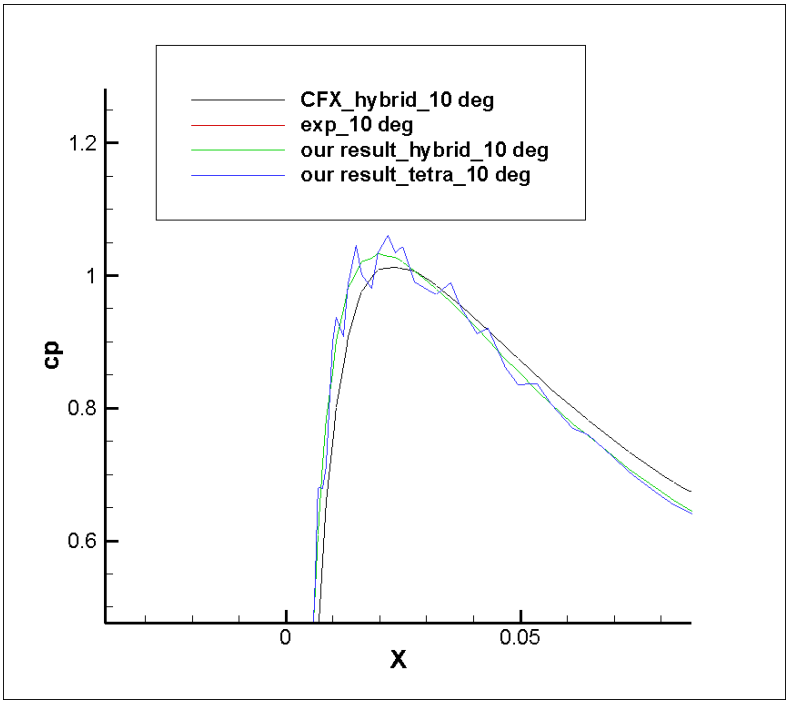


Figure 4.42 Cp around the leading edge

(M=0.15, Re=2.88×10^6, α = 10°)

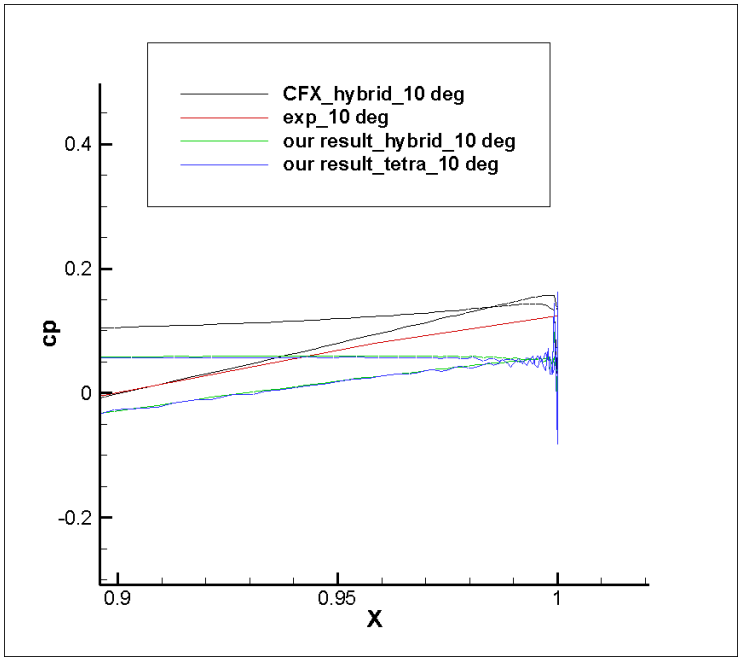


Figure 4.43 Cp around the trailing edge

(M=0.15, Re=2.88×10^6, α = 10°)

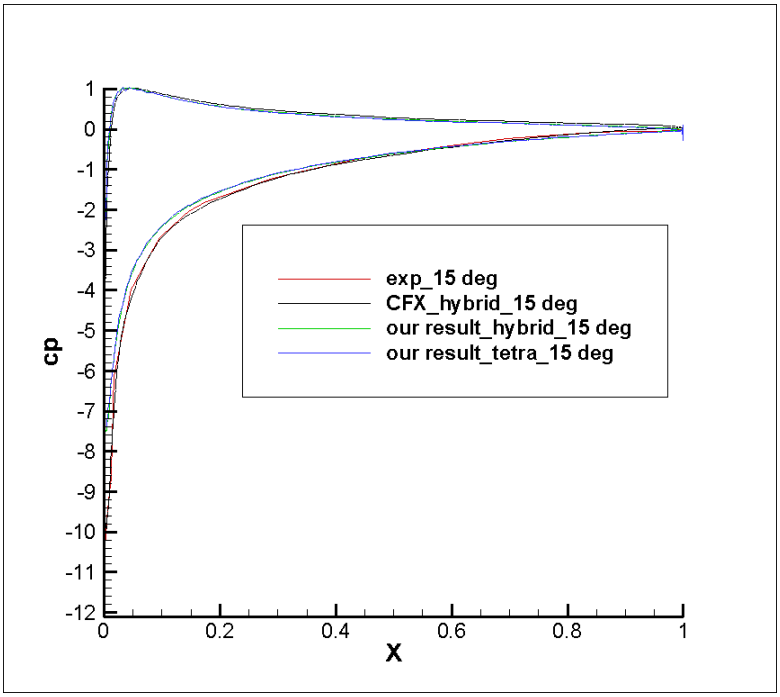


Figure 4.44 Cp (M=0.15, Re=2.88×10^6, α = 15°)

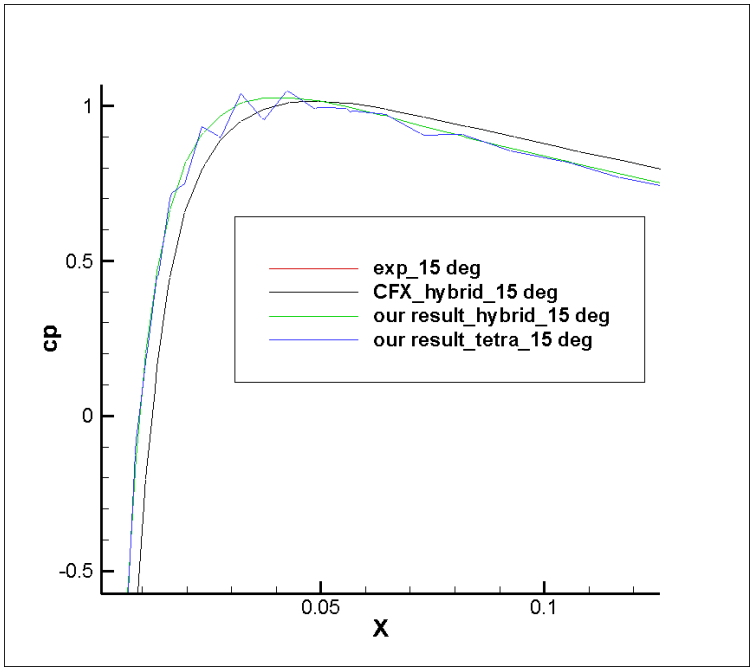


Figure 4.45 Cp around the leading edge

(M=0.15, Re=2.88×10^6, α = 15°)

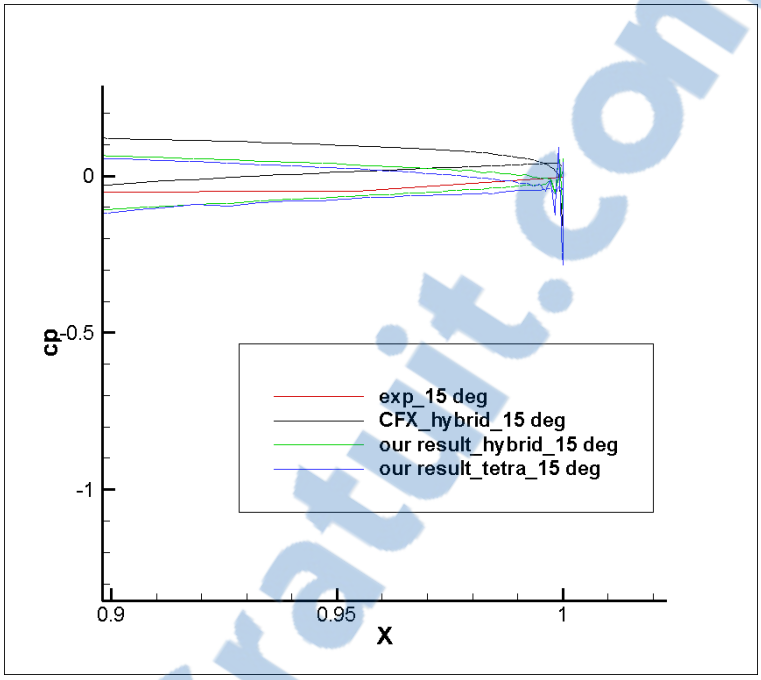


Figure 4.46 Cp around the trailing edge
(M=0.15, Re=2.88×10^6, α = 15°)

We can notice that we get good results for 0° and 10°. There are some discrepancies in our result for 15° because our code cannot get a steady solution. The hybrid mesh generally performs better than the tetra mesh. Using the tetra mesh, we can notice that there is strong oscillation near the trailing edge for 0°. CFX cannot generate converged results using the tetra mesh for 10° and 15°. Using the tetra mesh, there is slight oscillation near the leading edge in the Cp contour of 10° and 15° produced by our code.

We calculate the lift coefficients for 10° and 15°. We then compare our results with the results obtained by other numerical models (NASA, 2014b).

Table 4.1 Lift coefficient C_L for 10°

Our Result (hybrid mesh)	0.9901
Our Result (tetra mesh)	0.9963
CFX (hybrid mesh)	1.0860
SST-V by JOE	1.0805
K-e-Rt by CFD++	1.1036
Wilcox2006 by CFL3D	1.0958

Table 4.2 Lift coefficient C_L for 15°

Our Result (hybrid mesh)	1.3672
Our Result(tetra mesh)	1.3522
CFX (hybrid mesh)	1.5218
SST-V by JOE	1.5079
K-e-Rt by CFD++	1.5815
Wilcox2006 by CFL3D	1.5686

Compared with other numerical results, our code produces less lift. The difference is below 10% for 10° . The lift coefficient is too low for 15° when our code cannot generate a steady solution.

4.3 DLR F11 model

The mesh we use for the DLR F11 model is composed of 31 409 402 elements in total. It includes 7 803 763 tetras, 3 504 600 hexahedrons, 19 741 873 prisms and 359 166 pyramids. There are 14 948 380 nodes in total. The mesh of the whole domain is shown in Figure 4.47 and the mesh around the fuselage is shown in Figure 4.48.

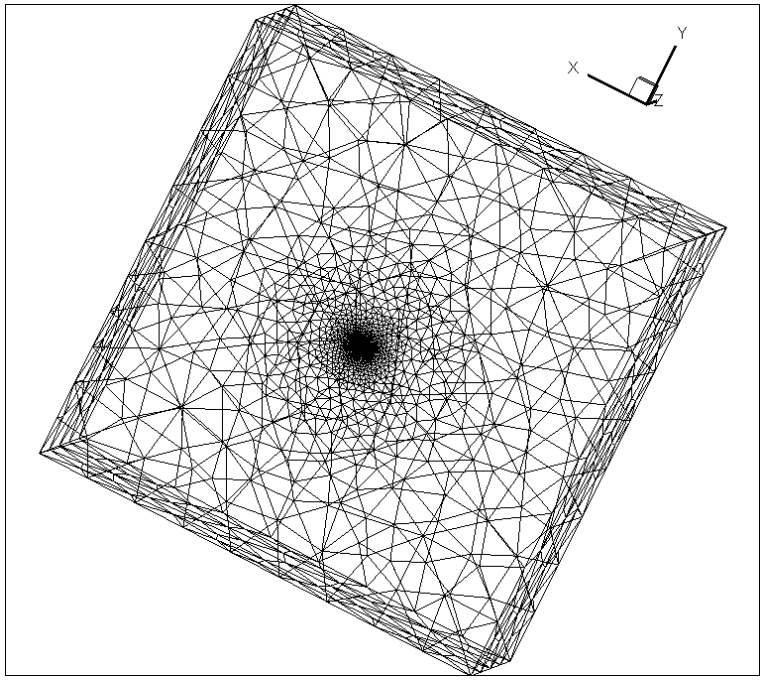


Figure 4.47 Mesh of the whole domain

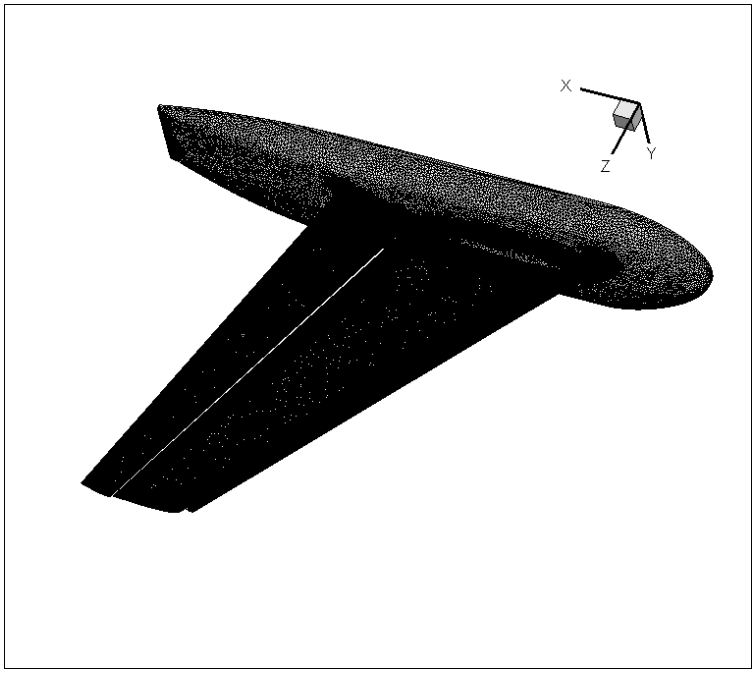


Figure 4.48 Mesh around the fuselage

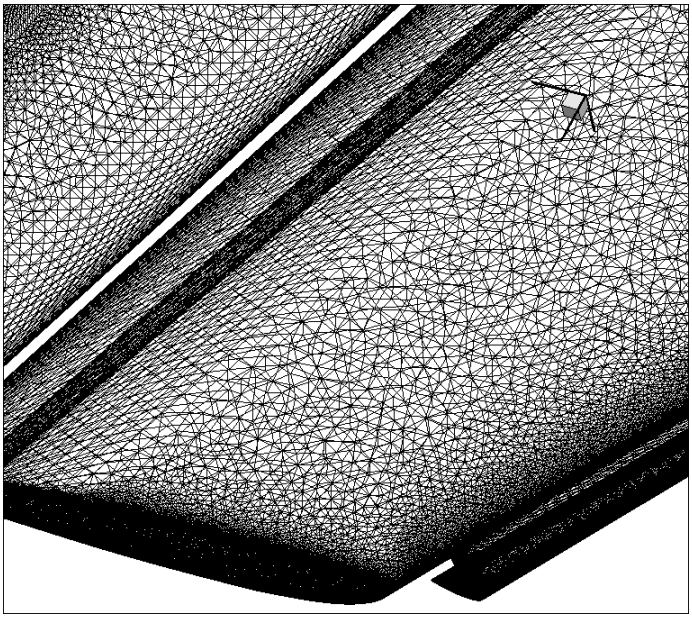


Figure 4.49 Mesh around the wing

Computer Platform: Guillimin compute cluster;
Number of processors: 288 (224 for fluid, 64 for turbulence);
Operating system: Unix;
Compiler: mvapich2/1.9-pgi and pgi/12.10;
Run time wall limit: 150 hours;
Memory requirement: 7700 MB per processor.

We validate our code under the condition Re=4.3×10^6, M=0.20, $\alpha = 13^\circ$. We use the same solution strategy that we use for the Naca0012 case. The lengths in the figures are the physical lengths.

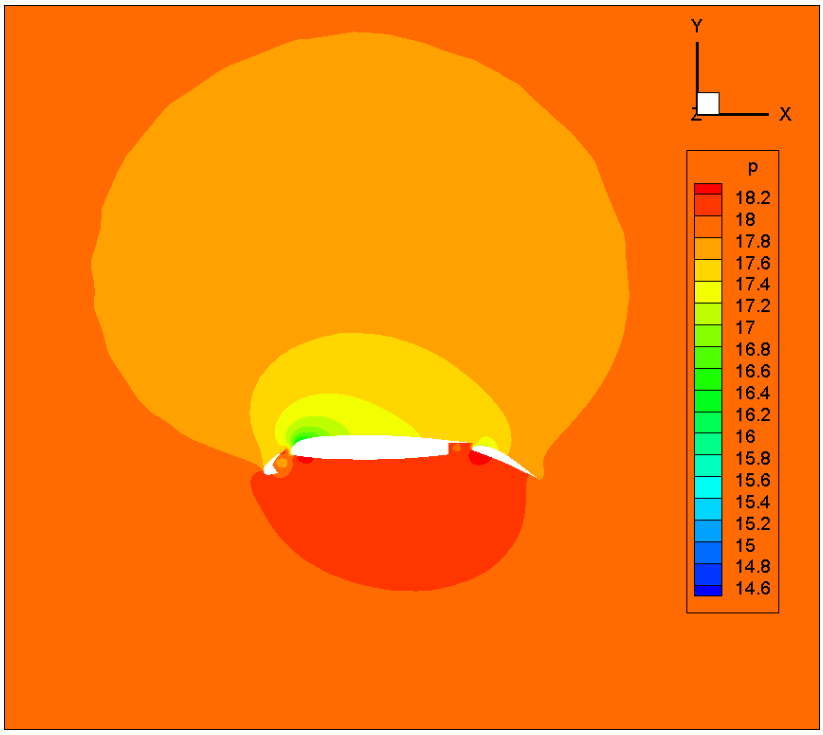


Figure 4.50 Pressure at z=30 in

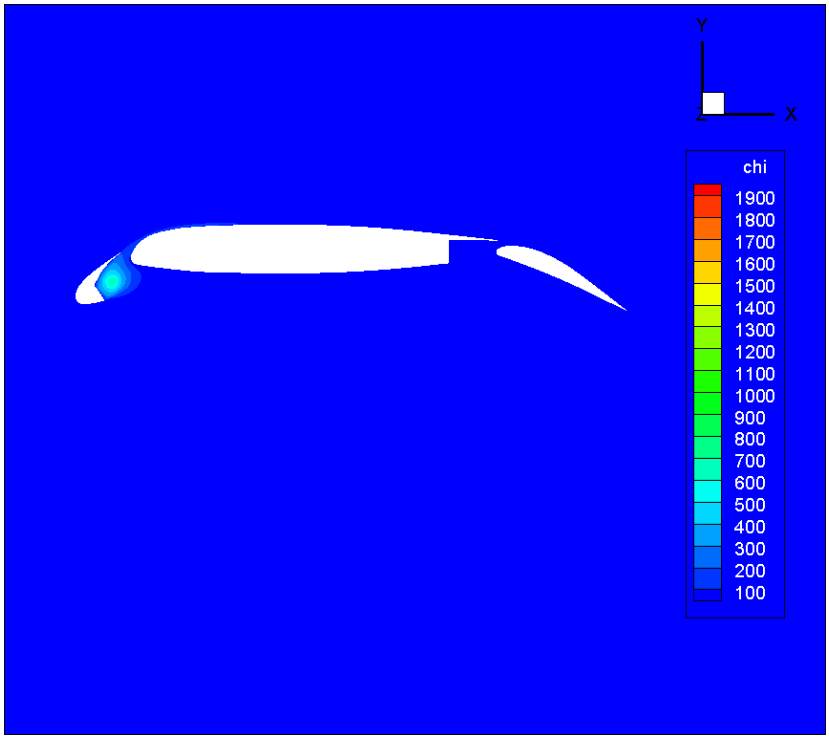


Figure 4.51 χ at z=30 in

Here we compare the Cp contour of wing with the experimental results and the numerical results obtained by CFX. We notice that we obtained good results of Cp for the upper wing. The results are relatively not satisfactory for the lower wing and areas near the wing tip.

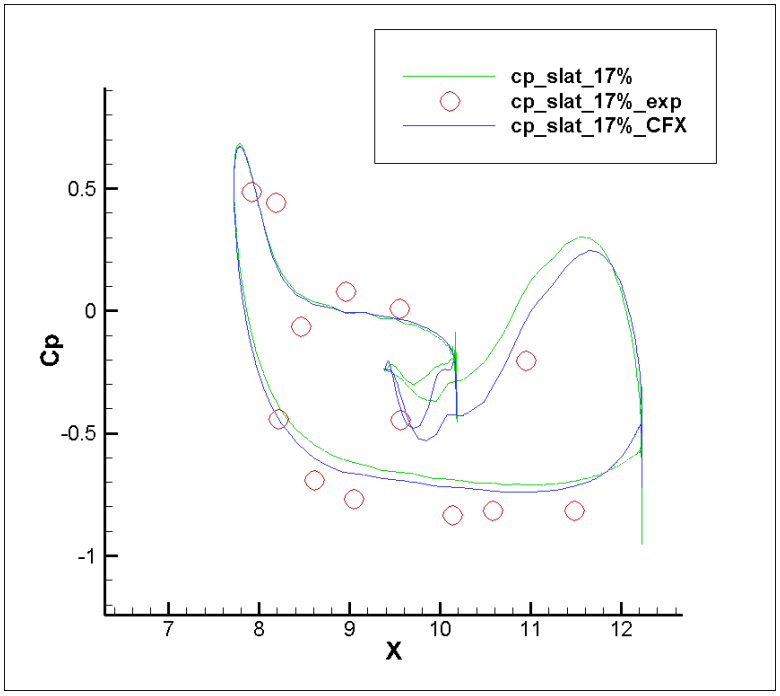


Figure 4.52 Cp contour of slat at 17% of span

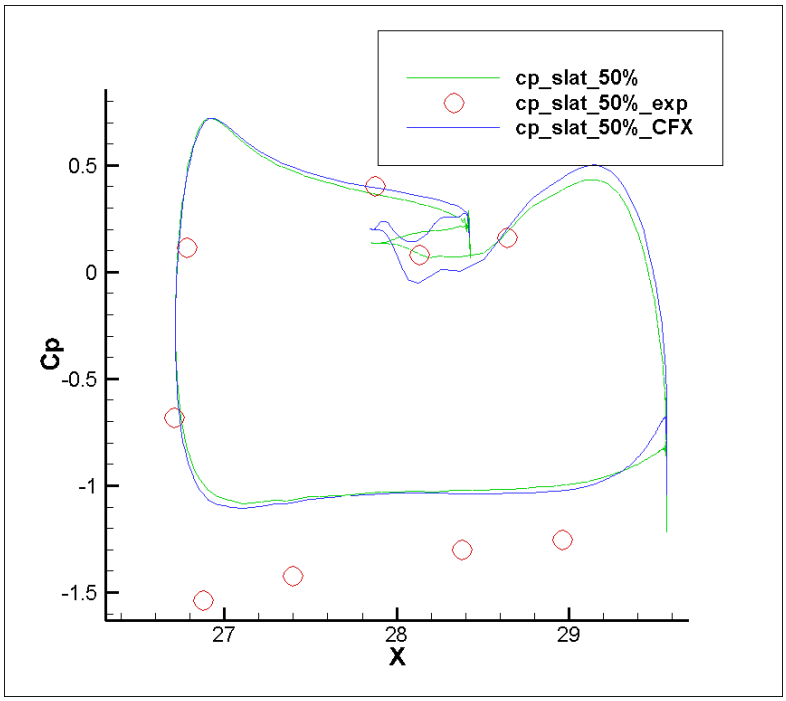


Figure 4.53 Cp contour of slat at 50% of span

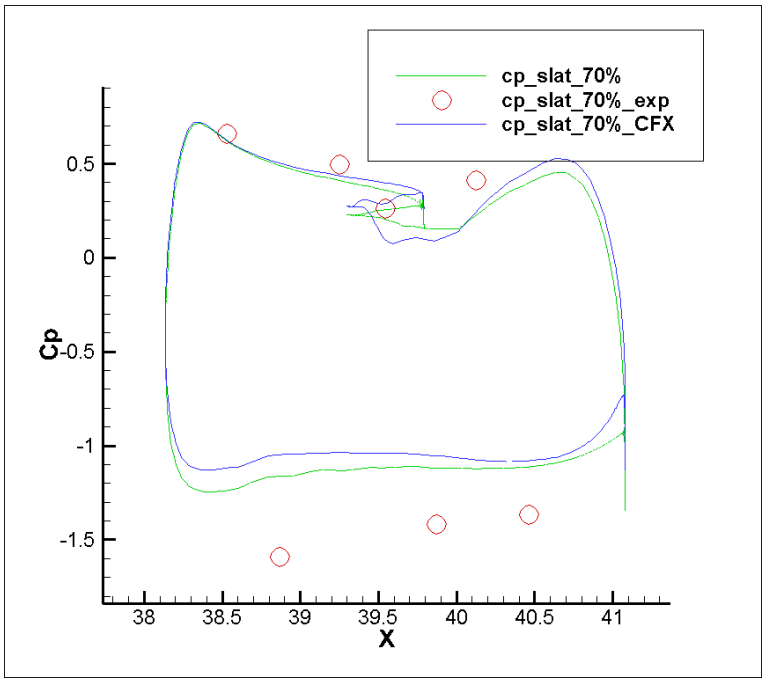


Figure 4.54 Cp contour of slat at 70% of span

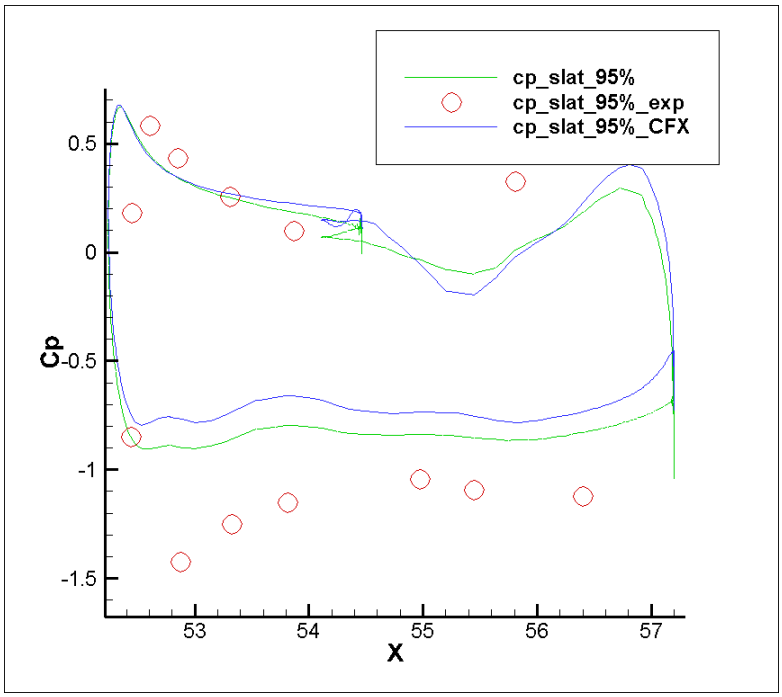


Figure 4.55 Cp contour of slat at 95% of span

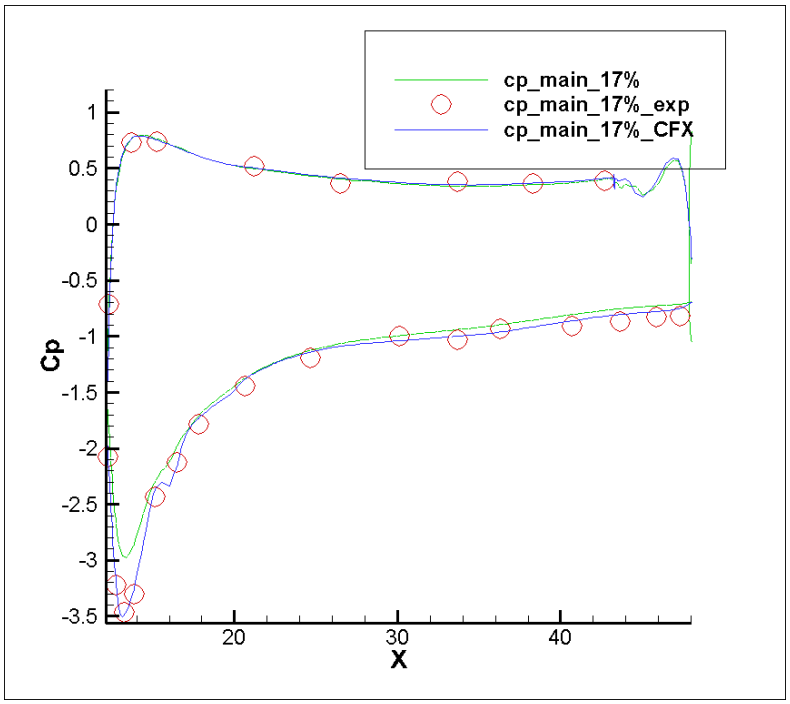


Figure 4.56 Cp contour of main-wing at 17% of span

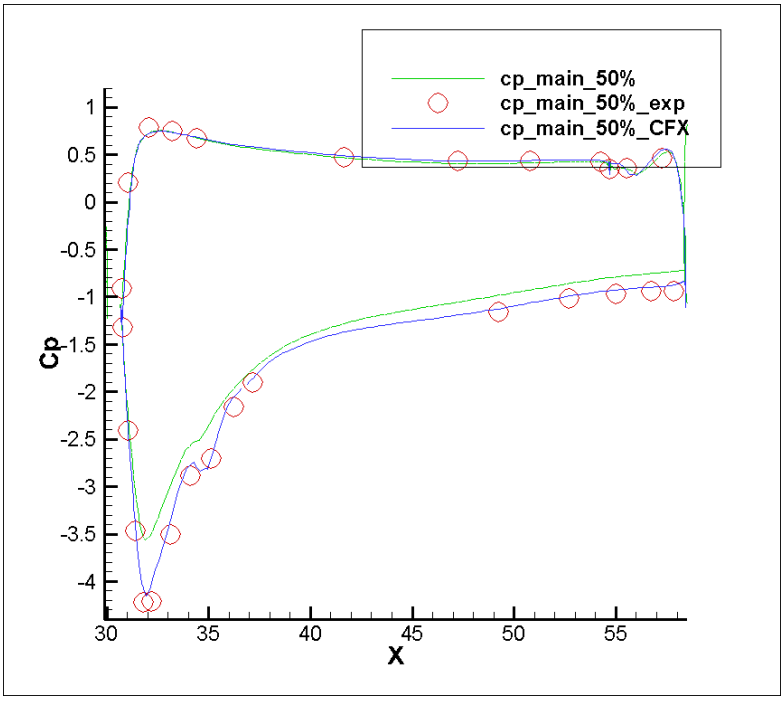


Figure 4.57 Cp contour of main-wing at 50% of span

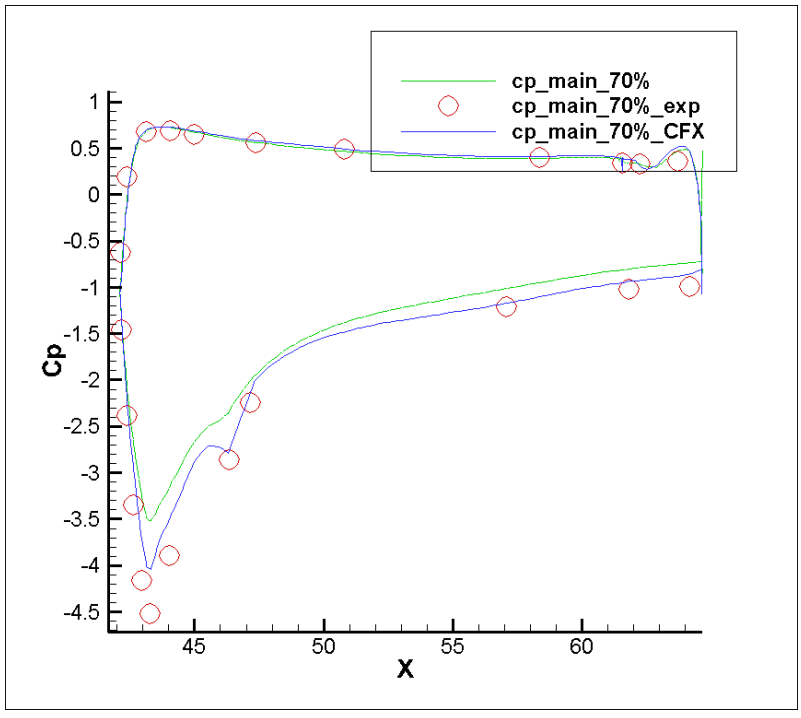


Figure 4.58 Cp contour of main-wing at 70% of span

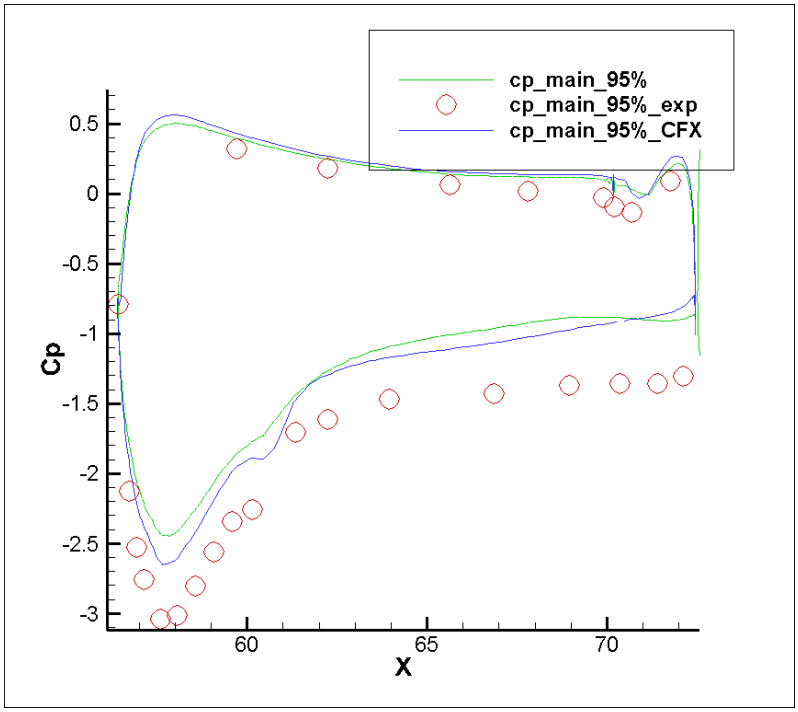


Figure 4.59 Cp contour of main-wing at 95% of span

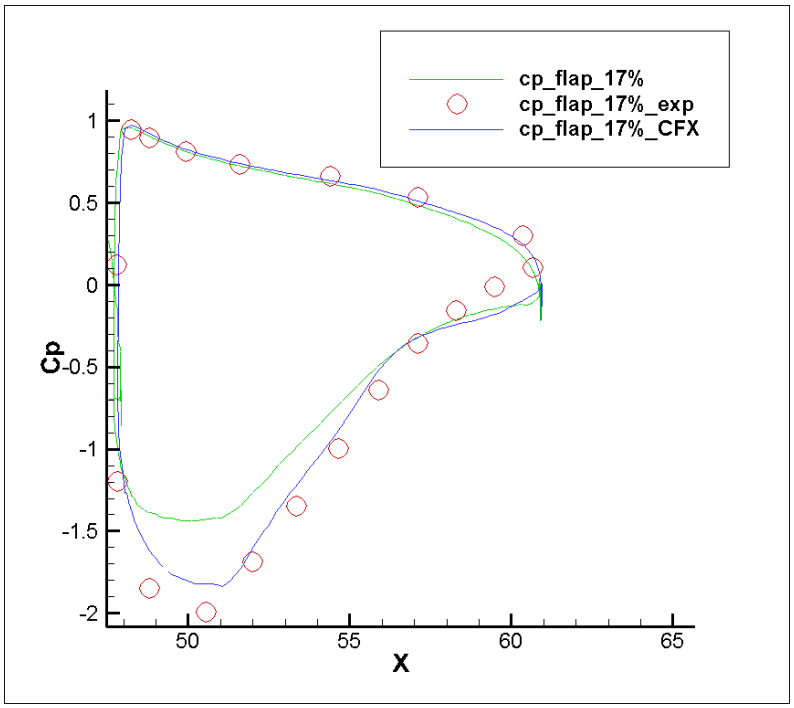


Figure 4.60 Cp contour of flap at 17% of span

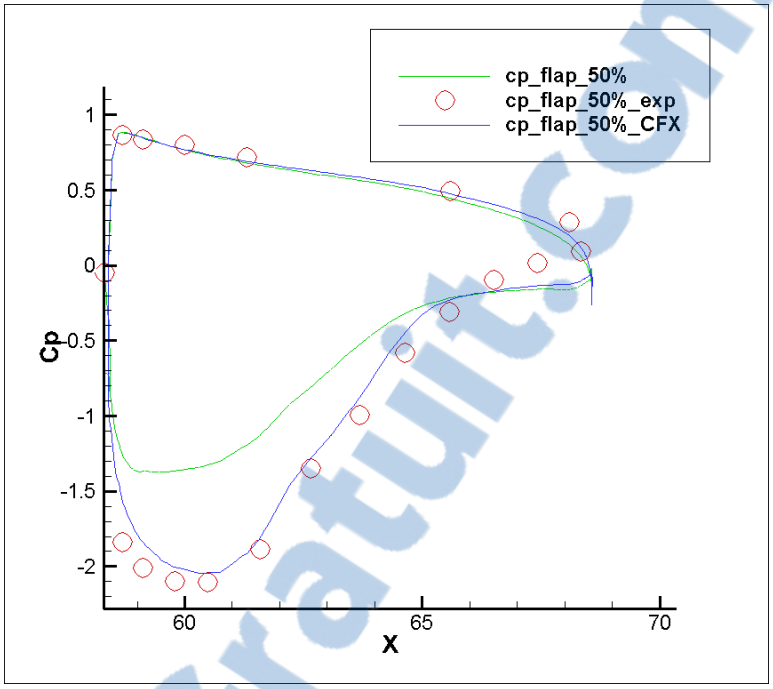


Figure 4.61 Cp contour of flap at 50% of span

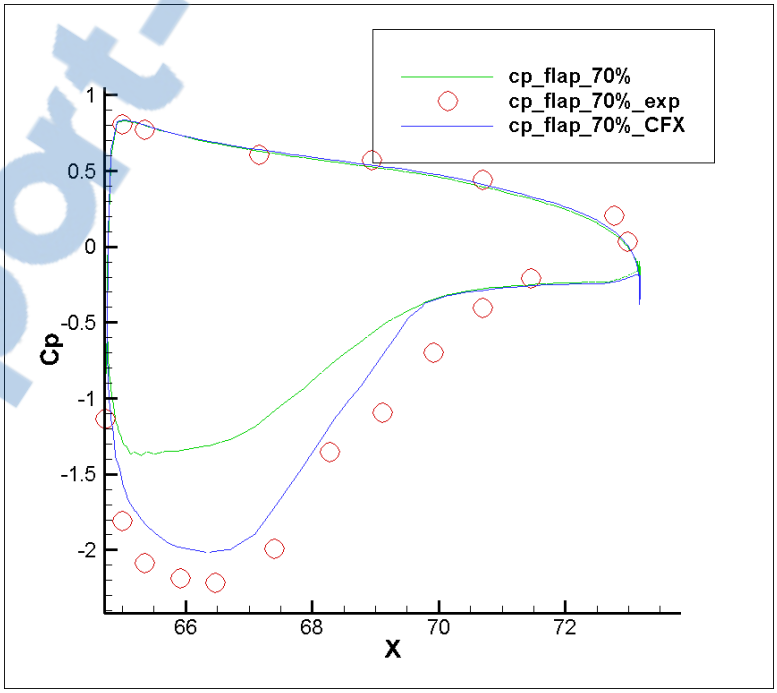


Figure 4.62 Cp contour of flap at 70% of span

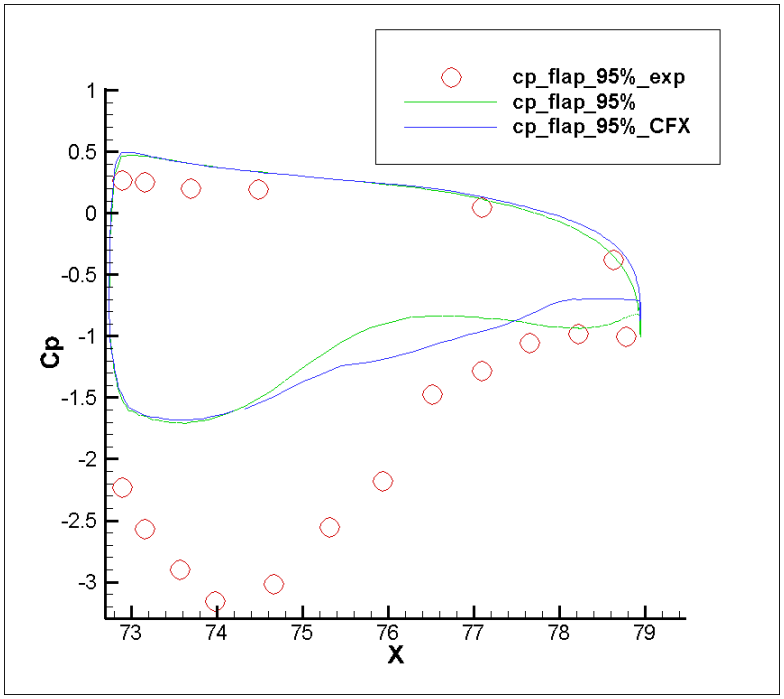


Figure 4.63 Cp contour of flap at 95% of span

The lift coefficient is compared with the experimental result and the CFX result as shown by Table 4.3.

Table 4.3 Lift coefficient C_L

Our Result ($\alpha = 13°$)	1.730	Experiment	2.047	CFX	1.882

From the lift coefficient, we notice that our code produces less lift than the experimental result. We notice that for the main wing areas, our code achieved generally satisfactory results. For the slat and flap, the results from our code and CFX are not good for the upper surface.

CONCLUSION

We presented a finite element method to simulate the coupled 3D Navier-Stokes turbulence model. There are three Navier-Stokes equations: the continuity equation, the equation of conservation of momentum and the equation of conservation of energy. The turbulence closure model we chose is the Spalart-Allmaras model, which has just a single equation. Instead of using the primitive variables, which are density, velocity and temperature, we used the conservative variables, which are density, momentum per unity mass and total energy per unit mass. We used the Galerkin approach to discretize the system of equations. We used four elements in our meshes: the eight node hexahedrons, four node tetras, six node prisms and five node pyramids. The SUPG method and a shock capturing method were employed to enforce numerical stability.

To use the finite element method, we converted the strong form of the system of equations to the weak form. We used the first-order forward finite difference method to discretize the time derivative term. We used the Newton-Raphson Method to calculate the turbulent viscosity. We used the GMRES algorithm to solve the nonlinear matrix form of the system.

To process the mesh files exported from Pointwise, we developed a preprocessing interface, which could convert the Starcd format to the format used by our code. To view the results obtained by our code, we developed a postprocessing interface, which we could use to extract surface elements and view the result by Tecplot.

The four types of elements used in our meshes were: hexahedron, tetra, prism and pyramid. We presented the shape functions and integration points for each type of element. To easily manage the four types of elements, we used the OOP method.

The two models we used to validate our code were the 3D NACA0012 model and the DLR F11 model. We compared our results to the references. We presented the pressure, density, turbulent viscosity, Cp contour and lift coefficient.

For the NACA0012 case, the Cp contours and lift coefficients obtained from our code were generally acceptable for all three angles of attack: 0°, 10° and 15°, although there was some slight discrepancy around areas of the leading edge and trailing edge. We notice that for angle of attack 15°, the solution was unsteady and it was harder for our code to converge. To curb the effect of the large diffusion, we developed a solution strategy: we gradually increased ε of the diagonal matrix of the shock capturing stabilization and the shock capturing viscosity. After we achieved good convergence with large values of the two variables, we could decrease it to a much smaller value.

For the complex case such as the DLR F11 model, many computing resources were required to run the code. Our code managed to provide generally reasonable results over an acceptable time period, although there were some noticeable discrepancies around the upper surface of the model for the Cp contours. Like the turbulent flows in the Naca0012 case, our code produced less lift than other experimental or numerical results.

From the results that we presented, it could be seen that our code was successfully extended to accept hybrid meshes. We also developed a pre-interface and a post-interface to streamline the processing phases. We noticed that with OOP, it was easier to modify the code, such as adding a new turbulence model and using high-order elements.

There are still things that we can do to improve our code. The code can be further optimized to consume fewer computing resources and less computation time. Compared to other experimental and numerical results, the accuracy of our code can still be improved. We can include the other two- and three-equation turbulence models and draw comparisons with the Spalart-Allmaras model. We can include other new stabilization methods such as the Variational Multiscale Method (VMS) in our code. Once our model is more accelerated, we can implement a DES model.

APPENDIX I

Data pre-processing interface

```
! ========================================================================
! Name        : starcd2pfes.f90
! Author      : Amine, Ben Haj Ali, modified by Wen Yang Li
! Version     : 2.0
! Copyright   : Copyright Granit
! Description : Convert STAR-CD mesh to pfes
! ========================================================================

program starcd2pfes

    implicit none
999 Format (16I10)
888 Format (I7,A10)
777 Format (10I1,3F20.16,A10)
666 Format (I15,3E16.0)
!-- Variables kind
    integer, parameter :: single = selected_real_kind( 6)     ! single precision
    integer, parameter :: double = selected_real_kind(13)     ! double precision
    integer, parameter :: quad   = selected_real_kind(30) ! quadrapule precision
    integer, parameter :: big    = selected_int_kind(12)  ! big integer to 10e12
    integer, parameter :: small  = selected_int_kind(4)  ! small integer to 10e4
    integer, parameter :: hugestring  = 512
    integer, parameter :: longstring  = 256
    integer, parameter :: midstring   =  64
    integer, parameter :: shortstring =  16


!-- Variable for upper and lower case functions
    CHARACTER( * ), PARAMETER :: LOWER_CASE = 'abcdefghijklmnopqrstuvwxyz'
    CHARACTER( * ), PARAMETER :: UPPER_CASE = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'


!-- I/O files and units

    !-- Input files
    character(hugestring)    :: starcdfile              ! STAR-CD mesh file name

    integer, parameter       :: starcd_cel   = 10   ! unit for STAR-CD cel file
    integer, parameter       :: starcd_vrt   = 11   ! unit for STAR-CD vrt file
    integer, parameter       :: starcd_bnd   = 12   ! unit for STAR-CD bnd file

    !-- Output files
    integer, parameter       :: logfile      = 20    ! unit for logging file
    integer, parameter       :: pfes_coor_f  = 21    ! unit for coordinates file
    integer, parameter       :: pfes_coor_t  = 22   !unit for  turb coordinates
    integer, parameter       :: pfes_con     = 23  ! unit for pfes connectivity
    integer, parameter       :: pfes_bc_f    = 24 ! unit for  fluide bc
    integer, parameter       :: pfes_bc_t    = 25 ! unit for pfes turb bc
    integer, parameter       :: pfes_dist    = 26 ! unit for distance to wall
    integer, parameter       :: tecplotfile  = 27 ! unit for tecplot file
    integer, parameter       :: pfes_f_ini   = 31! unit fluid initialization
    integer, parameter       :: pfes_t_ini   = 32! unit pfes turb initialization
    integer, parameter       :: pfes_metis   = 33! unit for pfes metis mesh file
```

```
!-- Data structure variables
    integer(big)              :: nnt                    ! total number of nodes
    real(double),allocatable :: nx(:)     ! x coordinate vector of all the node
    real(double),allocatable :: ny(:)     ! y coordinate vector of all the node
    real(double),allocatable :: nz(:)     ! z coordinate vector of all the node

    integer(big)              :: nelt              ! total number of elements
    integer(big),allocatable :: n1(:)        ! n1(i) is the node 1 of element i
    integer(big),allocatable :: n2(:)        ! n1(i) is the node 2 of element i
    integer(big),allocatable :: n3(:)        ! n1(i) is the node 3 of element i
    integer(big),allocatable :: n4(:)        ! n1(i) is the node 4 of element i
    integer(big),allocatable :: n5(:)        ! n1(i) is the node 5 of element i
    integer(big),allocatable :: n6(:)        ! n1(i) is the node 6 of element i
    integer(big),allocatable :: n7(:)        ! n1(i) is the node 7 of element i
    integer(big),allocatable :: n8(:)        ! n1(i) is the node 8 of element i

    integer(small),parameter :: nbctype =11  ! nuber of boundary conditions type
    character(4),parameter   :: bctype(nbctype) = ['INLE','OUTL','SYMP','WALL',&
                  'PRES','CYCL','FREE','STAG','TRAN','ATT ','NONE']
    integer(big)              :: nel2Dt      ! total nuber of 2D elements
    integer(big),allocatable :: nodebc(:)   ! BC type of the node

    real(double),allocatable :: dist(:)                  ! distcance vector
    real(double),allocatable :: distance(:)              ! distcance vector


    integer(big),allocatable :: wicount(:),nicount(:),nnwe(:),vicount(:),uicount(:)

!-- working variables
    real(double),parameter   :: eps = 1e-16
    integer(small)           :: argc,ierr
    integer(big)             :: itmp,icount,jcount
    integer(big),dimension(4):: elemnode
    character(longstring)    :: errmsg
    character(shortstring)   :: dwall,ibctype
    integer(big)             :: i1,i2,i3,i4,i5,i6,i7,i8,i9,i10
    integer(big)             :: u1,u2,u3,u4,u5,u6,u7,u8,u9,u10,u,eletype
    integer(big)             :: nbc1,nbc2,nbc3,nbc4,nbc5,nbc6,&
                                nbc7,nbc8,nbc9,nbc10 !number of nodes for each bc
    integer(big)             :: ncount1,ncount2,ncount3,ncount4,ncount5, &
    ncount6,ncount7,ncount8,ncount9,ncount10 ! indice for vectors of nodes
    integer(big),allocatable :: xbc1(:),xbc2(:),xbc3(:),xbc4(:),xbc5(:), &
                   xbc6(:),xbc7(:),xbc8(:),xbc9(:),xbc10(:)!vector of nodes
    logical                  :: bool
    CHARACTER                :: inff,inft,sp,dn
    character(4)             :: intc1,intc2,tj
    integer(big)             :: ntp1,ntp2,ntp3,ntp4 ! number of elements found
    double precision         :: refLen !reference length
    integer                  :: nwe  ! number of wall elements in an element
    integer                  :: tnwe,ii,tnwe2 ! total number of wall elements
    double precision         :: pass1,pass2,pass3


!-- the reference length used
    reflen=275.8d0
!-- Verify and parse command line arguments
    argc = command_argument_count()
    if( argc < 1 ) then
         write(*,*)"Usage : starcd2pfes starcdfile [dwall]"
         write(*,*)"starcdfile : prefix of the STAR-CD (crt, cel and bnd) files"
         write(*,*)"dwall : optional argument to calculate the distance to wall"
```

```
          write(*,*)"Error : please provide the STAR-CD mesh file name."
          stop
     endif

     dwall=""
     call getarg(1, starcdfile)
     if( argc == 2 ) then
     call getarg(2, dwall)
     endif

     if( (argc == 2) .and. (StrLowCase(trim(dwall)) /= "dwall") ) then
          write(*,*)"Usage : starcd2pfes starcdfile [dwall]"
          write(*,*)"starcdfile : prefix of the STAR-CD (crt, cel and bnd) files"
          write(*,*)"dwall : optional argument to calculate the distance to wall"
          write(*,*)"Error : please enter a valid option  ",trim(dwall)
          stop
     endif

!-- Open the logfile and check the open statement
     open(logfile,FILE=trim(starcdfile)//".log", status="UNKNOWN", &
          iostat=ierr,iomsg=errmsg)
     if ( ierr /= 0 ) then
          write(*,*) errmsg
          stop
     endif
     rewind(logfile)
     write(*,*)"Running starcd2pfes.."

!-- Write the log file header
     write(logfile,*)"============Log file of starcd2pfes  ====================="
     write(logfile,*)"============================================================="
     write(logfile,*)"! Name        : starcd2pfes.f90"
     write(logfile,*)"! Author      : Amine, Ben Haj Ali & Wen Yang Li"
     write(logfile,*)"! Version     : 2.0"
     write(logfile,*)"! Copyright   : Copyright Granit ÉTS"
     write(logfile,*)"! Description : Convert STAR-CD mesh to pfes"
     write(logfile,*)"============================================================="
     write(logfile,*)""
     call flush(logfile)

!-- Open starcd2pfes input files
     open(starcd_cel,FILE=trim(starcdfile)//".cel", status="OLD",&
          iostat=ierr,iomsg=errmsg)
     if ( ierr /= 0 ) then
          write(logfile,*) errmsg
          stop
     endif
     rewind(starcd_cel)

     open(starcd_vrt,FILE=trim(starcdfile)//".vrt", status="OLD",&
          iostat=ierr,iomsg=errmsg)
     if ( ierr /= 0 ) then
          write(logfile,*) errmsg
          stop
     endif
     rewind(starcd_vrt)

     open(starcd_bnd,FILE=trim(starcdfile)//".bnd", status="OLD",&
          iostat=ierr,iomsg=errmsg)
     if ( ierr /= 0 ) then
          write(logfile,*) errmsg
```

```
          stop
     endif
     rewind(starcd_bnd)


     write(*,*)"================================================================="

     write(*,*)"You asked to read data from ",trim(starcdfile),".* ...ok"
     if(trim(dwall)=="dwall") then
     write(*,*)"You asked to compute the wall distance  ...ok"
     else
     write(*,*)"You didn't ask to compute the wall distance  ...ok"
     endif
     write(*,*)""
     write(*,*)"let's go !"
     write(*,*)""
     write(*,*)"================================================================="

     write(*,*)"output redirected to ",trim(starcdfile)//".log"
     write(*,*)""
     write(*,*)"working.."
     write(logfile,*)"starcd2pfes reading data from ",trim(starcdfile),".*"
     if(trim(dwall)=="dwall") then
     write(logfile,*)"wall distance will be computed"
     else
     write(logfile,*)"wall distance won't be computed"
     endif
     write(logfile,*)""
     call flush(logfile)

!-- Open starcd2pfes output files
     open(pfes_coor_f,FILE=trim(starcdfile)//"_f.cor", status="UNKNOWN",&
          iostat=ierr,iomsg=errmsg)
     if ( ierr /= 0 ) then
          write(logfile,*) errmsg
          stop
     endif
     rewind(pfes_coor_f)

     open(pfes_coor_t,FILE=trim(starcdfile)//"_t.cor", status="UNKNOWN",&
          iostat=ierr,iomsg=errmsg)
     if ( ierr /= 0 ) then
          write(logfile,*) errmsg
          stop
     endif
     rewind(pfes_coor_t)

     open(pfes_con,FILE=trim(starcdfile)//".con", status="UNKNOWN",&
          iostat=ierr,iomsg=errmsg)
     if ( ierr /= 0 ) then
          write(logfile,*) errmsg
          stop
     endif
     rewind(pfes_con)

     open(pfes_bc_f,FILE=trim(starcdfile)//"_f.lim", status="UNKNOWN",&
          iostat=ierr,iomsg=errmsg)
     if ( ierr /= 0 ) then
          write(logfile,*) errmsg
          stop
     endif
```

```
    rewind(pfes_bc_f)

    open(pfes_bc_t,FILE=trim(starcdfile)//"_t.lim", status="UNKNOWN",&
         iostat=ierr,iomsg=errmsg)
    if ( ierr /= 0 ) then
         write(logfile,*) errmsg
         stop
    endif
    rewind(pfes_bc_t)

    open(pfes_dist,FILE=trim(starcdfile)//".dist", status="UNKNOWN",&
         iostat=ierr,iomsg=errmsg)
    if ( ierr /= 0 ) then
         write(logfile,*) errmsg
         stop
    endif
    rewind(pfes_dist)

    open(tecplotfile,FILE=trim(starcdfile)//".tec", status="UNKNOWN",&
         iostat=ierr,iomsg=errmsg)
    if ( ierr /= 0 ) then
         write(logfile,*) errmsg
         stop
    endif
    rewind(tecplotfile)

    open(pfes_f_ini,FILE=trim(starcdfile)//"_f.ini", status="UNKNOWN",&
         iostat=ierr,iomsg=errmsg)
    if ( ierr /= 0 ) then
         write(logfile,*) errmsg
         stop
    endif
    rewind(pfes_f_ini)


    open(pfes_t_ini,FILE=trim(starcdfile)//"_t.ini", status="UNKNOWN",&
         iostat=ierr,iomsg=errmsg)
    if ( ierr /= 0 ) then
         write(logfile,*) errmsg
         stop
    endif
    rewind(pfes_t_ini)

    open(pfes_metis,FILE=trim(starcdfile)//".metis", status="UNKNOWN",&
         iostat=ierr,iomsg=errmsg)
    if ( ierr /= 0 ) then
         write(logfile,*) errmsg
         stop
    endif
    rewind(pfes_metis)



!-- Real work starts here !
!-- read nodes coordinates
    write(logfile,*)"============================================================"
    write(logfile,*)""
    write(logfile,*)"reading coordinates from ",trim(starcdfile),".vrt ..."
    call flush(logfile)
    nnt=0
```

```
do
     read(starcd_vrt,*,iostat=ierr,iomsg=errmsg)  icount
     if (ierr > 0) then
           write(logfile,*) errmsg
           stop
     elseif (ierr < 0) then
           write(logfile,*)  'Total number of node', nnt,icount
           exit
     else
      nnt = nnt + 1
     endif
enddo

if (nnt/= icount) then
     write(logfile,*)"found a problem in ",trim(starcdfile),".vrt"
     write(logfile,*)"number of line different from the last node id",icount
     stop
endif


allocate (nx(nnt), STAT= itmp)
if (itmp /= 0) then
    write(logfile,*)"Not enough memory to allocate nx with ",nnt
    stop
endif

allocate (ny(nnt), STAT= itmp)
if (itmp /= 0) then
    write(logfile,*)"Not enough memory to allocate ny with ",nnt
    stop
endif

allocate (nz(nnt), STAT= itmp)
if (itmp /= 0) then
    write(logfile,*)"Not enough memory to allocate nz with ",nnt
    stop
endif

allocate (wicount(nnt), STAT= itmp)
if (itmp /= 0) then
    write(logfile,*)"Not enough memory to allocate wicount with ",tnwe
    stop
endif

allocate (vicount(2*nnt), STAT= itmp)
if (itmp /= 0) then
    write(logfile,*)"Not enough memory to allocate wicount with ",tnwe
    stop
endif

allocate (uicount(2*nnt), STAT= itmp)
if (itmp /= 0) then
    write(logfile,*)"Not enough memory to allocate wicount with ",tnwe
    stop
endif


do icount=1,2*nnt
    uicount(icount)=0
    vicount(icount)=0
    enddo
```

```
      rewind(starcd_vrt)
      do icount = 1, nnt
      read(starcd_vrt,666),itmp,nx(icount),ny(icount),nz(icount)
      enddo
      close(starcd_vrt)

      write(logfile,*)"reading coordinates completed"
      write(logfile,'(a5,1x,I8,3f25.16)')" Node",1,nx(1),ny(1),nz(1)
      write(logfile,'(a5,1x,I8,3f25.16)')" Node",nnt,nx(nnt),ny(nnt),nz(nnt)
      write(logfile,*)""
      call flush(logfile)
      write(logfile,*)"writing coordinates to ",trim(starcdfile)//"_*.cor"
      write(logfile,*)""


!--   read boundary conditions
      write(logfile,*)"============================================================"
      write(logfile,*)""

      write(logfile,*)"reading bc from ",trim(starcdfile),".bnd ..."
      call flush(logfile)

      nel2Dt=0
      do
           read(starcd_bnd,*,iostat=ierr,iomsg=errmsg)  icount
           if (ierr > 0) then
                 write(logfile,*) errmsg
                 stop
           elseif (ierr < 0) then
                 write(logfile,*)  'Total number of 2D element = ', nel2Dt,icount
                 exit
           else
            nel2Dt = nel2Dt + 1
           endif
      enddo

      if (nel2Dt/= icount) then
           write(logfile,*)"found a problem in ",trim(starcdfile),".bnd"
           write(logfile,*)"number of line different from last 2D elem id",icount
           stop
      endif

      allocate (nodebc(nnt), STAT= itmp)
      if (itmp /= 0) then
          write(logfile,*)"Not enough memory to allocate nodebc with ",nnt
          stop
      endif
      call flush(logfile)

      rewind(starcd_bnd)
      nodebc = 11


      nbc1 = 0
      nbc2 = 0
      nbc3 = 0
      nbc4 = 0
      nbc5 = 0
      nbc6 = 0
      nbc7 = 0
```

```
  nbc8 = 0
  nbc9 = 0
  nbc10= 0



  ncount1=0
  ncount2=0
  ncount3=0
  ncount4=0
  ncount5=0
  ncount6=0
  ncount7=0
  ncount8=0
  ncount9=0
  ncount10=0

  allocate (xbc1(nnt), STAT= itmp)
  if (itmp /= 0) then
      write(logfile,*)"Not enough memory to allocate xbc1 with ",nnt
      stop
  endif

  allocate (xbc2(nnt), STAT= itmp)
  if (itmp /= 0) then
      write(logfile,*)"Not enough memory to allocate xbc2 with ",nnt
      stop
  endif

  allocate (xbc3(nnt), STAT= itmp)
  if (itmp /= 0) then
      write(logfile,*)"Not enough memory to allocate xbc3 with ",nnt
      stop
  endif

  allocate (xbc4(nnt), STAT= itmp)
  if (itmp /= 0) then
      write(logfile,*)"Not enough memory to allocate xbc4 with ",nnt
      stop
  endif

  allocate (xbc9(nnt), STAT= itmp)
  if (itmp /= 0) then
      write(logfile,*)"Not enough memory to allocate xbc9 with ",nnt
      stop
  endif

  tnwe2=0
  do icount = 1, nel2Dt
  read(starcd_bnd,*),itmp,elemnode(1:4),itmp,itmp,ibctype
!bctype = ['INLE','OUTL','SYMP','WALL','PRES','CYCL','FREE', &
!  'STAG','TRAN','ATT ','NONE']
!bcflag = [  1,     2,     3,     4,     5,     6,     7,     8,  &
! 9,     10,    11  ]

  !find the bcflag of the actual element


       do jcount = 1, nbctype
        intc1=trim(ibctype)
        intc2=trim(bctype(jcount))
```

```
 if(intc1.eq. intc2) itmp = jcount
enddo

 if (itmp .eq. 4) then
 tnwe2=tnwe2+1

 endif


 do jcount = 1, 4

 nodebc(elemnode(jcount)) = itmp

  bool=.false.


!-- This part counts the node in each BC


                if(nodebc(elemnode(jcount)) == 3) then

                do u=1,nbc3
                if(xbc3(u)==elemnode(jcount)) then
                 goto 110
                endif
                enddo
                nbc3=nbc3+1
                ncount3=ncount3+1
                xbc3(ncount3)=elemnode(jcount)

                elseif(nodebc(elemnode(jcount)) == 4) then
                if (jcount .eq. 4) then
                   goto 110
                endif
                vicount((tnwe2-1)*3+jcount)=elemnode(jcount)
                do u=1,nbc4
                if(xbc4(u)==elemnode(jcount)) then
                   goto 110
                endif
                enddo
                nbc4=nbc4+1
                ncount4=ncount4+1
                xbc4(ncount4)=elemnode(jcount)
                if ( uicount(elemnode(jcount)) .eq. 0 ) then
                   uicount(elemnode(jcount))=nbc4
                endif


                elseif(nodebc(elemnode(jcount)) == 9) then

                do u=1,nbc9
                if(xbc9(u)==elemnode(jcount)) then
                 goto 110
                endif
                enddo
                nbc9=nbc9+1
                ncount9=ncount9+1
                xbc9(ncount9)=elemnode(jcount)
```

```
 110             endif
    enddo
!-- end of repetition check

    enddo


    close(starcd_bnd)


!-- Write pfes coord files

    rewind(pfes_coor_f)
    rewind(pfes_coor_t)


    write(pfes_coor_f,'(I10,2I4,3f8.3)')nnt,5,3,refLen,refLen,refLen
    write(pfes_coor_t,'(I10,2I4,3f8.3)')nnt,1,3,refLen,refLen,refLen

    do icount=1, nnt
       write(pfes_coor_f,*)icount,nx(icount),ny(icount),nz(icount)
       write(pfes_coor_t,*)icount,nx(icount),ny(icount),nz(icount)
    enddo

    write(pfes_coor_f,*) -1
    write(pfes_coor_t,*) -1

    close(pfes_coor_f)
    close(pfes_coor_t)


!-- Write pfes bc files

    rewind(pfes_bc_f)
    rewind(pfes_bc_t)

 !count the nodes for each bc type

    write(logfile,*)"reading boundary conditions completed"
    write(logfile,*)"Nodes/BC found :"

    write(logfile,*)nbc1,bctype(1)
    write(logfile,*)nbc2,bctype(2)
    write(logfile,*)nbc3,bctype(3)
    write(logfile,*)nbc4,bctype(4)
    write(logfile,*)nbc5,bctype(5)
    write(logfile,*)nbc6,bctype(6)
    write(logfile,*)nbc7,bctype(7)
    write(logfile,*)nbc8,bctype(8)
    write(logfile,*)nbc9,bctype(9)
    write(logfile,*)nbc10,bctype(10)
    write(logfile,*)""
    write(logfile,*)"writing boundary conditions to ",trim(starcdfile)//"_*.lim"
    write(logfile,*)""


! fill the bc tables



!--  This part only deals with limited boundary conditions
```

```
      if(nbc3 /= 0) then
      write(pfes_bc_f,888)nbc3,bctype(3)
      write(pfes_bc_f,777)0,0,0,1,0,0,0,0,0,0,0.0,0.0,0.0,"0"
      write(pfes_bc_f,999)(xbc3(icount),icount=1,nbc3)
      endif

      if(nbc4 /= 0) then

      write(pfes_bc_f,888)nbc4,bctype(4)
      write(pfes_bc_f,777)0,1,1,1,0,0,0,0,0,0,0.0,0.0,0.0,"0"
      write(pfes_bc_f,999)(xbc4(icount),icount=1,nbc4)
      write(pfes_bc_t,888)nbc4,bctype(4)
      write(pfes_bc_t,777)1,0,0,0,0,0,0,0,0,0,0.0,0.0,0.0,"0"
      write(pfes_bc_t,999)(xbc4(icount),icount=1,nbc4)
      endif

      if(nbc9 /= 0) then

      write(pfes_bc_t,888)nbc9,bctype(9)
      write(pfes_bc_t,777)2,0,0,0,0,0,0,0,0,1,0.0,0.0,0.0,"0"
      write(pfes_bc_t,999)(xbc9(icount),icount=1,nbc9)
      endif



    write(pfes_bc_f,*) -1
    write(pfes_bc_t,*) -1

    close(pfes_bc_f)
    close(pfes_bc_t)


    allocate (distance(nnt), STAT= itmp)
    if (itmp /= 0) then
      write(logfile,*)"Not enough memory to allocate distance vector with ",nnt
      stop
    endif
    distance=0.0d0

!-- if asked compute distance to wall
    if(trim(dwall)=="dwall") then
    write(logfile,*)"============================================================"
    write(logfile,*)""

    write(logfile,*)"starcd2pfes computing the distance to wall ..."
    call flush(logfile)

        rewind(pfes_dist)

        allocate (dist(nbc4), STAT= itmp)
        if (itmp /= 0) then
      write(logfile,*)"Not enough memory to allocate distance vector with ",nbc4
      stop
        endif

         distance=9999999999.0d0
         ! distance to the closer node
         do icount=1,nnt
```

```
            dist=9999999999.0d0

            do jcount=1,nbc4
               dist(jcount)=dsqrt((nx(icount)-nx(xbc4(jcount)))**2+   &
                                  (ny(icount)-ny(xbc4(jcount)))**2+   &
                                  (nz(icount)-nz(xbc4(jcount)))**2 )
            enddo !surface elements

            if(minval(dist).lt.eps)then
            distance(icount) = 0.0d0
            write(pfes_dist,*)distance(icount),icount
            else
            distance(icount) = minval(dist)
            write(pfes_dist,*)distance(icount),xbc4(minloc(dist))
            endif

         enddo
    write(logfile,*)""
    write(logfile,*)"writing distance  to ",trim(starcdfile)//"_*.dist"
    write(logfile,*)""
    deallocate(dist)
    endif

    close(pfes_dist)

!   read result file
    write(logfile,*)"============================================================"
    write(logfile,*)""
    write(logfile,*)"starcd2pfes reading results file"
    call flush(logfile)


    allocate (nicount(nnt), STAT= itmp)
    if (itmp /= 0) then
        write(logfile,*)"Not enough memory to allocate nicount with ",tnwe
        stop
    endif


!-- end of results file

!write coord,bc and distance in tecplot file
    write(logfile,*)"============================================================"
    write(logfile,*)"writing data  to  tecplot file ",trim(starcdfile)//".tec"
    write(logfile,*)""
    call flush(logfile)

    write(tecplotfile,*)'TITLE = "',trim(starcdfile),'"'
    write(tecplotfile,*)'VARIABLES = "X", "Y", "Z"'
    write(tecplotfile,*)'ZONE T="Volume", F=FEPOINT, N=',nnt,', ET=BRICK'


    call flush(logfile)

    if(nnt.ne.0) then
    do icount=1, nnt
    write(tecplotfile,*)nx(icount),ny(icount),nz(icount)
    enddo

    call flush(logfile)
    endif
```

```
    write(logfile,*)""

    if(trim(dwall)=="dwall") then
    deallocate(distance)
    endif

!-- read elements (connectivity)
    write(logfile,*)"======================================================="
    write(logfile,*)""

    write(logfile,*)"reading elements from ",trim(starcdfile),".cel ..."
    call flush(logfile)
    nelt=0

    do
         read(starcd_cel,*,iostat=ierr,iomsg=errmsg)  icount
         if (ierr > 0) then
               write(logfile,*) errmsg
               stop
         elseif (ierr < 0) then
               write(logfile,*)  'Total number of volume element= ', nelt,icount
               exit
         else
          nelt = nelt + 1
         endif
    enddo

    if (nelt /= icount) then
         write(logfile,*)"found a problem in ",trim(starcdfile),".cel"
         write(logfile,*)"the number of line in the file id different &
                          from the last element id",icount
         stop
    endif

    allocate (n1(nelt), STAT= itmp)
    if (itmp /= 0) then
        write(logfile,*)"Not enough memory to allocate n1 with ",nelt
        stop
    endif
    allocate (n2(nelt), STAT= itmp)
    if (itmp /= 0) then
        write(logfile,*)"Not enough memory to allocate n2 with ",nelt
        stop
    endif
     allocate (n3(nelt), STAT= itmp)
    if (itmp /= 0) then
        write(logfile,*)"Not enough memory to allocate n3 with ",nelt
        stop
    endif
    allocate (n4(nelt), STAT= itmp)
    if (itmp /= 0) then
        write(logfile,*)"Not enough memory to allocate n4 with ",nelt
        stop
    endif
    allocate (n5(nelt), STAT= itmp)
    if (itmp /= 0) then
        write(logfile,*)"Not enough memory to allocate n5 with ",nelt
        stop
    endif
```

```
    allocate (n6(nelt), STAT= itmp)
    if (itmp /= 0) then
        write(logfile,*)"Not enough memory to allocate n6 with ",nelt
        stop
    endif
    allocate (n7(nelt), STAT= itmp)
    if (itmp /= 0) then
        write(logfile,*)"Not enough memory to allocate n7 with ",nelt
        stop
    endif
    allocate (n8(nelt), STAT= itmp)
    if (itmp /= 0) then
        write(logfile,*)"Not enough memory to allocate n8 with ",nelt
        stop
    endif

    rewind(starcd_cel)
    do icount = 1, nelt
    read(starcd_cel,*),itmp,n1(icount),n2(icount),n3(icount),n4(icount), &
    n5(icount),n6(icount),n7(icount),n8(icount)
    enddo

    close(starcd_cel)

    write(logfile,*)"reading elements completed"
    write(logfile,'(a8,1x,5I8)')" Element",1,n1(1),n2(1),n3(1),n4(1)
    write(logfile,'(a8,1x,9I8)')" Element",nelt,n1(nelt),n2(nelt),n3(nelt),&
    n4(nelt),n5(nelt),n6(nelt),n7(nelt),n8(nelt)
    write(logfile,*)""
    call flush(logfile)


!-- Fichier de connectivit?*.con
    write(logfile,*)"starcd2pfes writing elements to ",trim(starcdfile)//".con"
    write(logfile,*)""

    rewind(pfes_con)
    rewind(pfes_metis)

    write(pfes_con,'(6I8)')nelt,8,1,2,1,1
    write(pfes_metis,'(I8)')nelt
    itmp = 2

    ntp1=0
    ntp2=0
    ntp3=0
    ntp4=0


    do icount=1,nelt

      if (n3(icount)==n4(icount)  .and. n5(icount)==n6(icount) &
        .and. n6(icount)==n7(icount).and.n7(icount)==n8(icount) )then ! tetra
      n4(icount)=n5(icount)
      n5(icount)=0
      n6(icount)=0
      n7(icount)=0
      n8(icount)=0
      eletype=1
      ntp1=ntp1+1
      elseif (n5(icount)==n6(icount) .and. n6(icount)==n7(icount) &
```

```
      .and. n7(icount)==n8(icount) )then !pyramid
      n6(icount)=0
      n7(icount)=0
      n8(icount)=0
      eletype=4
      ntp4=ntp4+1
      elseif (n3(icount)==n4(icount) .and. n7(icount)==n8(icount)) then
      n4(icount)=n5(icount)
      n5(icount)=n6(icount)
      n6(icount)=n7(icount)
      n7(icount)=0
      n8(icount)=0
      eletype=3
      ntp3=ntp3+1
      else
      eletype=2
      ntp2=ntp2+1
      endif
!************ This block check the wall boundary type and the number of nodes

!*****************************************************
      !Tetra free boundary

      if((nodebc(n1(icount)).eq. 9 ).and.(nodebc(n2(icount)).eq.9) &
      .and. (nodebc(n3(icount)).eq.9) )then
      write(pfes_con,999)icount,eletype,1,2,n1(icount),n2(icount),&
      n3(icount),n4(icount),n5(icount),n6(icount),n7(icount),n8(icount)
      write(pfes_metis,999)n1(icount),n2(icount),n3(icount),n4(icount)
      write(tecplotfile,999)n1(icount),n2(icount),n3(icount),n3(icount),&
      n4(icount),n4(icount),n4(icount),n4(icount)
      cycle
      endif

      if((nodebc(n1(icount)).eq. 9 ).and.(nodebc(n2(icount)).eq.9) &
      .and. (nodebc(n4(icount)).eq.9) )then
      write(pfes_con,999)icount,eletype,1,2,n2(icount),n1(icount),n4(icount),&
      n3(icount),n5(icount),n6(icount),n7(icount),n8(icount)
      write(pfes_metis,999)n2(icount),n1(icount),n4(icount),n3(icount)
      write(tecplotfile,999)n1(icount),n2(icount),n3(icount),n3(icount),&
      n4(icount),n4(icount),n4(icount),n4(icount)
      cycle
      endif

      if((nodebc(n2(icount)).eq. 9 ).and.(nodebc(n3(icount)).eq.9) &
      .and. (nodebc(n4(icount)).eq.9) )then
      write(pfes_con,999)icount,eletype,1,2,n3(icount),n2(icount),&
      n4(icount),n1(icount),n5(icount),n6(icount),n7(icount),n8(icount)
      write(pfes_metis,999)n3(icount),n2(icount),n4(icount),n1(icount)
      write(tecplotfile,999)n1(icount),n2(icount),n3(icount),n3(icount),&
      n4(icount),n4(icount),n4(icount),n4(icount)
      cycle
      endif

      if((nodebc(n3(icount)).eq. 9 ).and.(nodebc(n4(icount)).eq.9) &
      .and. (nodebc(n1(icount)).eq.9) )then
      write(pfes_con,999)icount,eletype,1,2,n3(icount),n4(icount),n1(icount),&
      n2(icount),n5(icount),n6(icount),n7(icount),n8(icount)
      write(pfes_metis,999)n3(icount),n4(icount),n1(icount),n2(icount)
      write(tecplotfile,999)n1(icount),n2(icount),n3(icount),n3(icount),&
      n4(icount),n4(icount),n4(icount),n4(icount)
      cycle
```

```
      endif

      write(pfes_con,999)icount,eletype,1,1,n1(icount),n2(icount),&
      n3(icount),n4(icount),n5(icount),n6(icount),n7(icount),n8(icount)


      selectcase (eletype)

      case(1)
      write(pfes_metis,999)n1(icount),n2(icount),n3(icount),n4(icount)
      write(tecplotfile,999)n1(icount),n2(icount),n3(icount),n3(icount),&
      n4(icount),n4(icount),n4(icount),n4(icount)
      case(2)

      write(pfes_metis,999)n1(icount),n2(icount),n3(icount),n4(icount),&
      n5(icount),n6(icount),n7(icount),n8(icount)
      write(tecplotfile,999)n1(icount),n2(icount),n3(icount),n4(icount),&
      n5(icount),n6(icount),n7(icount),n8(icount)

      case(3)

      write(pfes_metis,999)n1(icount),n2(icount),n3(icount),n4(icount),&
      n5(icount),n6(icount)
      write(tecplotfile,999)n1(icount),n2(icount),n3(icount),n3(icount),&
      n4(icount),n5(icount),n6(icount),n6(icount)

      case(4)

      write(pfes_metis,999)n1(icount),n2(icount),n3(icount),n4(icount),&
      n5(icount)
      write(tecplotfile,999)n1(icount),n2(icount),n3(icount),n4(icount),&
      n5(icount),n5(icount),n5(icount),n5(icount)
      endselect
    enddo

     write(tecplotfile,*)'TITLE = "',trim(starcdfile),'"'
     write(tecplotfile,*)'VARIABLES = "X", "Y", "Z"'
     write(tecplotfile,*)'ZONE T="',bctype(4),'", N=',nbc4,', E=',tnwe2,&
      ',DATAPACKING = POINT,ZONETYPE=FEQuadrilateral'



    if(nbc4.ne.0) then
    do icount=1, nbc4
    write(tecplotfile,*)nx(xbc4(icount)),ny(xbc4(icount)),nz(xbc4(icount))
    nicount(xbc4(icount))=icount
    enddo
    call flush(logfile)
    endif


! connectivity file. Far filed are made of tetras

    do icount=1, tnwe2
    write(tecplotfile,'(3I10)')uicount(vicount((icount-1)*3+1)),&
    uicount(vicount((icount-1)*3+2)),uicount(vicount((icount-1)*3+3)),&
    uicount(vicount((icount-1)*3+3))
    enddo


write(tecplotfile,*)
```

```
! end of writing conncetivity file


    write(pfes_con,'(I3)')-1


    close(pfes_con)
    close(pfes_metis)
    close(tecplotfile)


    deallocate(nicount,uicount,vicount)

    deallocate(xbc1,xbc2,xbc3,xbc4,xbc9)


    deallocate(n1,n2,n3,n4,n5,n6,n7,n8)
    deallocate(nodebc)

    write(logfile,*)ntp1,"TETRA"
    write(logfile,*)ntp2,"HEXA"
    write(logfile,*)ntp3,"PRISM"
    write(logfile,*)ntp4,"PYRAMID"

    write(logfile,*)""
    write(logfile,*)"done."

    call flush(logfile)
    close(logfile)
    write(*,*)""
    write(*,*)"done."
    write(*,*)"good bye!"

    STOP



! ======================================================================
!-- END OF PROGRAM
! ======================================================================

CONTAINS

    FUNCTION StrUpCase ( Input_String ) RESULT ( Output_String )
        ! -- Argument and result
        CHARACTER( * ), INTENT( IN )     :: Input_String
        CHARACTER( LEN( Input_String ) ) :: Output_String
        ! -- Local variables
        INTEGER :: i, n

        ! -- Copy input string
        Output_String = Input_String
        ! -- Loop over string elements
        DO i = 1, LEN( Output_String )
             ! -- Find location of letter in lower case constant string
             n = INDEX( LOWER_CASE, Output_String( i:i ) )
             ! -- If current substring is lower case , make it upper case
             IF ( n /= 0 ) Output_String( i:i ) = UPPER_CASE( n:n )
        END DO
    END FUNCTION StrUpCase
```

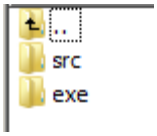
```
    FUNCTION StrLowCase ( Input_String ) RESULT ( Output_String )
         ! -- Argument and result
         CHARACTER( * ), INTENT( IN )     :: Input_String
         CHARACTER( LEN( Input_String ) ) :: Output_String
         ! -- Local variables
         INTEGER :: i, n

         ! -- Copy input string
         Output_String = Input_String
         ! -- Loop over string elements
         DO i = 1, LEN( Output_String )
              ! -- Find location of letter in upper case constant string
              n = INDEX( UPPER_CASE, Output_String( i:i ) )
              ! -- If current substring isupper case letter, make it lower case
              IF ( n /= 0 ) Output_String( i:i ) = LOWER_CASE( n:n )
         END DO

END FUNCTION StrLowCase


end program
```

Instruction on how to run the PFES code:

This instruction shows the detailed steps of how to run the code PFES on the high lift case. The above code is the preprocessing interface, and it can be slightly modified to postprocess the data.

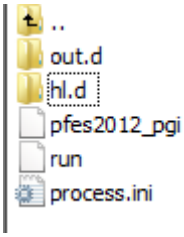
The mesh of the PointWise file is named *HiLiftPW-Fine-PFES.pw*.

The current directory is */sb/project/sks-412-aa/soulaimani/PFES2015*. We create an empty directory. In this directory, we set up two folders:

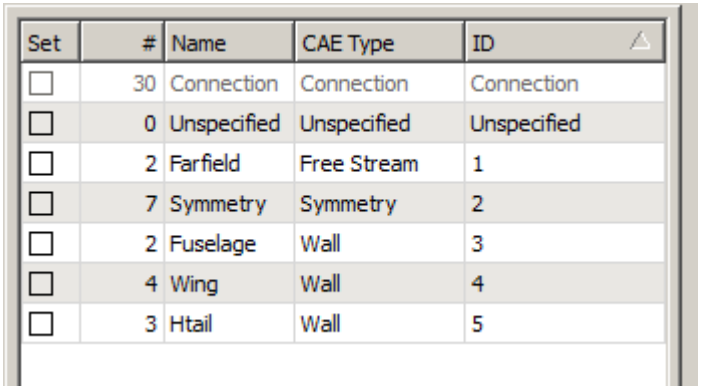


The source files are in the directory *src*.

In the directory *exe*. the structure is as follows:



1. Choose the StarCD format and set the boundary conditions. Only three boundary conditions can be set in PointWise as ‘CAE Type’ (Symmetry, Wall and Free Stream).



Set	#	Name	CAE Type	ID
☐	30	Connection	Connection	Connection
☐	0	Unspecified	Unspecified	Unspecified
☐	2	Farfield	Free Stream	1
☐	7	Symmetry	Symmetry	2
☐	2	Fuselage	Wall	3
☐	4	Wing	Wall	4
☐	3	Htail	Wall	5

2. Export the Pointwise mesh to the directory */sb/project/sks-412-aa/soulaimani/PFES2015/High_lift/hl.d*. There will be four files: *HiLiftPW-Fine-PFES.cel, HiLiftPW-Fine-PFES.vrt, HiLiftPW-Fine-PFES.bnd and HiLiftPW-Fine-PFES.inp*.
3. Add the three modules in the command editor:

   module add pgi/12.10

   module add mvapich2/1.9-pgi

   module add ParMETIS/4.0.3-mva-1.9-pgi-12.10

   Compile the source code *starcd2pfes.f* located in the directory *starcd2pfes* to the executable *starcd2pfes.o*

   mpif90 starcd2pfes.f90 –o starcd2pfes

4. Run the interface using Unix to change the StarCD format of the mesh to the format used by our code:

   mpiexec -np 1 ./starcd2pfes HiLiftPW-Fine-PFES dwall

   We will obtain eight files: *HiLiftPW-Fine-PFES_f.cor, HiLiftPW-Fine-PFES_t.cor, HiLiftPW-Fine-PFES_f.lim, HiLiftPW-Fine-PFES_t.lim, HiLiftPW-Fine-PFES.con, HiLiftPW-Fine-PFES.log, HiLiftPW-Fine-PFES.metis, and HiLiftPW-Fine-PFES.tec, HiLiftPW-Fine-PFES.dist*

   If we do not want to calculate the wall distance, the command will be:

   mpiexec -np 1 ./starcd2pfes HiLiftPW-Fine-PFES

5. We then perform the mesh partition. If we want 224 partitions for the fluid, the command will be:

   Mpmetis HiLiftPW-Fine-PFES.metis 224

   We then rename the file *HiLiftPW-Fine-PFES.metis.epart.224* to *HiLiftPW-Fine-PFES_f.met*

   Next, we perform the mesh partition for the turbulence:

   Mpmetis HiLiftPW-Fine-PFES.metis 64

We then rename the file *HiLiftPW-Fine-PFES.metis.epart.64* to *HiLiftPW-Fine-PFES_t.met*

Delete the two files *HiLiftPW-Fine-PFES.metis.npart.64*, *HiLiftPW-Fine-PFES.metis.npart.224*

6. Open the directory *High_Lift*. The source code is located in the directory *src*. The parameters and the output files are in the directory *exe*.

In the *hl.d* directory located in the folder *exe*, add the two files *HiLiftPW-Fine-PFES_f.inp* and *HiLiftPW-Fine-PFES_t.inp*. These two files set the parameters of the code.

In the *exe* directory, open the file *process.ini* and change the name of the input data files if necessary. For example, for the mesh named *HiLiftPW-Fine-PFES*:

```
Fichiers de données
COORD:
'HiLiftPW-Fine-PFES_f.cor'
'HiLiftPW-Fine-PFES_t.cor'
COND:
'HiLiftPW-Fine-PFES.con'
'HiLiftPW-Fine-PFES.con'
LIMD:
'HiLiftPW-Fine-PFES_f.lim'
'HiLiftPW-Fine-PFES_t.lim'
INPD:
'HiLiftPW-Fine-PFES_f.inp'
'HiLiftPW-Fine-PFES_t.inp'
INID:
'HiLiftPW-Fine-PFES_f.ini'
'HiLiftPW-Fine-PFES_t.ini'
DIST:
'HiLiftPW-Fine-PFES.dist'
'HiLiftPW-Fine-PFES.dist'
FMETIS:
'HiLiftPW-Fine-PFES_f.met'
'HiLiftPW-Fine-PFES_t.met'
FIN
```

Also, set the total number of processors requested. Set the number of processors requested for fluid and turbulence. For example:

```
   Nombre de processeurs
288
Proc/famille  Chemin/IN                      Chemin/OUT              nom du groupe
Type
 224    'hl.d'    'out.d'           fnaca      fluide
 64     'hl.d'    'out.d'           tnaca      turbul
```

Open the run file located in the directory *exe*. Set the number of processors requested. Set the nodes, processors per node and wall time limit.

```
#PBS -l nodes=18:ppn=16,pmem=7700mb
#PBS -l walltime=30:02:00

cd /sb/project/sks-412-aa/soulaimani/PFES2015/High Lift/exe
```

7. Compile the source code in the directory *src.* The executable file will be *pfes2012_pgi* located in the directory *exe.*
8. In Unix, set the current directory as *exe.* Submit the task using the command:

```
qsub run
```

9. The result we obtain is the file *fnaca_Tecplot.dat* located in *out.d*; we change the format of the data so that so we can obtain the results of the surfaces:

```
mpiexec -np    1   ./pfes_result HiLiftPW-Fine-PFES
```

If we also want to obtain the result of the whole mesh (volume and surfaces elements), we use the command:

```
mpiexec -np    1   ./pfes_result_wall HiLiftPW-Fine-PFES
```

We can then use Tecplot to view the result file *HiLiftPW-Fine-PFES.tec*.

BIBLIOGRAPHY

Ansys (2015), URL http://www.ansys.com/

Aupoix B. and Spalart P. R. (2003): *Extensinos of the Spalart-Allmaras turbulence model to account for wall roughness*, International Journal of Heat and Fluid Flow, Volume 24(4): 454, ISSN 0142-727X

Baldwin, B. S. and Lomax, H. (1978): *Thin Layer Approximation and Algebraic Model for Separated Turbulent Flows,* AIAA Paper 78-257

Baldwin, B. S. and Barth, T. J. (1990): *A One-Equation Turbulence Transport Model for High Reynolds Number Wall-Bounded Flows,* NASA Technical Memerandum 102847

Banas, K. (2002): *A Newton-Krylov solver with multiplicative Schwarz preconditioning for finite element compressible flow simulations,* Communications in numerical methods in engineering, Volume 18 (4): 269, ISSN 1069-8299

Baruzzi, G. S., Habashi, W. G., Guevremont, J. G., Hafez, M. M. (1995): *A second order finite element method for the solution of the transonic Euler and Navier-Stokes equations,* International Journal for numerical methods in fluids, Volume 20 (8-9): 671, ISSN 0271-2091

Bassi, F. and Rebay, S. (1997): *A high-order accurate discontinuous finite element method for the numerical solution of the compressible Navier-Stokes equations*, Journal of computational physics, Volume 131 (2): 267, ISSN 0021-9991

Ben Haj Ali, A. (2002): *Calcul distribué pour des problèmes multiphysiques,* Master thesis, École de technologie supérieure, Montréal Canada.

Ben Haj Ali, A. (2008): *Calcul de haute performance en aéroélasticité et en écoulements turbulents tridimensionnels*, Doctoral thesis, École de technologie supérieure, Montréal Canada.

Ben Haj Ali, A. and Soulaïmani, A. (2010): *An unstructured finite elements method for solving the compressible RANS equations and the Spalart-Allmaras turbulence model*, Computer Methods in Applied Mechanics and Engineering, Volume 199 (33-36) : 2261, ISSN 0045-7845

Boivin, S. (1990) : *Simulation d'écoulements compressibles à nombre de Reynolds élévé*, Doctoral thesis, Université Laval, Laval Canada

Boussinesq, J. (1877) : *Essai sur la théorie des eaux courantes,* Mémoires présentés par divers savants à l'Académie des Sciences, Volume 23 (1)

Budd, Timothy (1997) : *An introduction to object oriented programming, 2e*, Addison-Wesley

Burman, E. (2000): *Adaptive finite element methods for compressible flow*, Computer Methods in Applied Mechanics and Engineering, Volume 190 (8-10) : 1137, ISSN 0045-7845

Cambier, L. (1987): *Computation of Viscous Flows Using and Unsteady Type Method and a Zonal Grid Refinement Technique*, Office National d'Etudes et de Recherche Aérospatiales (ONERA), Chatillon France

Cao, J. (2005): *Application of a posteriori error estimation to finite element simulation of compressible Navier-Stokesflow*, Computer & fluids, Volume 34 (8) : 991, ISSN 0045-7930

Caughey, D. A., Jameson A. (2003): *Fast preconditioned multigrid solution of the Euler and Navier-Stokes equations for steady, compressible flows,* International Journal for Numerical Methods in Fluids, Volume 43 (5): 537, ISSN 0271-2091

Chen C. M., Krizek M. and Liu L. P. (2013): *Numerical integration over pyramids,* Advances in Applied Machematics and Mechanics, Volume 5 (3): 309, ISSN 0196-8858

Coratekin, T., Van Keuk, J. and Ballmann, J. (2004): *Performance of Upwind Schemes and Turbulence Models in Hypersonic Flows*, AIAA Journal, Volume 42(5): 945, ISSN 0001-1452

Dassault Systèmes (2015), URL
http://www.3ds.com/products-services/simulia/products/abaqus/latest-release/

Deck, S., Duveau, P., D'Espiney, P., and Guillen, P. (2002) : *Development and application of Spalart-Allmaras one equation turbulence model to three-dimensional supersonic complex configurations*, Aerospace Science and Technology, Volume 6(3): 171, ISSN 1270-9638

De Santis, D. (2014): *High-order linear and non-linear residual distribution schemes for turbulent compressible flows*, Computer Methods in Applied Mechanics, Volume 285: 1, ISSN 0045-7825

Dhatt, G. and Touzot, G. (1981): *Une présentation de la méthode des éléments finis*, Presses de l'Université Laval

El Kadri El Yamani, N.-E. (1995): *Une méthode d'élément finis pour la dynamique des gaz et conception orientée objet du code de calcul*, Thèse de doctorat, Université de Laval, Québec Canada.

Feistauer, M., Felcman, J. and Lukacova-Medvid'ova, M. (1995): *Combined finite element-finite volume solution of compressible flow,* Journal of computational and applied mathematics, Volume 63: 179, ISSN 0377-0427

Geng Y. F., Yan, C., Xu, J. L., Kang, H. L. (2011): *Turbulence Modeling Validation in Hypersonic Flows*, Journal of Beijing University of Aeronautics and Astronautics, Volume 37 (8): 907, ISSN 1001-5965

Hafez, M. (1995): *Finite element/finite volume solutions of full potential, Euler and Navier-Stokes equations for compressible and incompressible flows,* International journal for numerical methods in fluids, Volume 20 (8): 713, ISSN 0271-2091

He, Y.N., Li, J. (2010): *A penalty finite element method based on the Euler implicit/explicit scheme for the time-dependent Navier-Stokes equations,* Journal of computational and applied mathematics, Volume 235 (3): 708, ISSN 0377-0427

Ishiko, K., Hashimoto, A., Matsuo, Y., Yoshizawa, A., Nishiyama, Y., Mori, K., Nakamura, Y. (2014) : *One-Equation Extended Nonlinear Turbulence Modeling in Predicting Three-Dimensional Wall Jets*, Journal of Aircraft, Volume 51 (2): 584, ISSN 0021-8669

Jameson, A. and Mavriplis, D. (1986): *Finite volume solution of the two-dimensional euler equations on a regular triangular mesh*, AIAA Journal, Volume 24 (4): 611, ISSN 0001-1452

Johnson, D. A. and King, L. S. (1985): *A mathematically simple turbulence closure model for attached and separated turbulent boundary layers,* AIAA Journal Volume 23(11): 1684, ISSN 0001-1452

Karagiozis, K., Kamakoti, R., Pantano, C. (2010): *A low numerical dissipation immersed interface method for the compressible Navier-Stokes equations,* Journal of Computational Physics, Volume 229 (3): 701, ISSN 0021-9991

Kellogg, B. and Liu, B. (2000): *The analysis of a finite element method for the Navier-Stokes equations with compressibility*, Numerische Mathematik, Volume 134 (5): 153, ISSN 0029-599X

Kersken, H. P., Frey, C., Voigt, C. (2012): *Time-Linearized and time-Accurate 3D RANS methods for aeroelastic analysis in turbomachinery,* Journal of Turbomachinery, ISSN 0889-504X

Khurram, R. A., Zhang, Y., Habashi, W.G. (2012): *Multiscale finite element method applied to the Spalart–Allmaras turbulence model for 3D detached-eddy simulation,* Computer Methods in Applied Mechanics and Engineering, Volume 233: 180, ISSN 0045-7825

Kirk, B. S. and Carey, G. F. (2008): *Development and validation of a SUPG finite element scheme for the compressible Navier-Stokes equations using a modified inviscid flux discretization,* International journal for numerical methods in fluids, Volume 57 (3): 265, ISSN 0271-2091

Klaij, C. M., van der Vegt, J. J. W., van der Ven, H. (2006): *Space-time discontinuous Galerkin method for the compressible Navier-Stokes equations,* Journal of Computational Physics, Volume 217 (2): 589, ISSN 0021-9991

Kweon, J. R. (2000): *A discontinuous Galerkin method for convection-dominated compressible viscous Navier-Stokes equations with an inflow boundary condition,* SIAM journal on numerical analysis, Volume 38 (3): 699, ISSN 0036-1429

Kong, W. X., Zeng, P., Van, C. and Zhao, R. (2012): *Numerical Simulation of Crossing Shock wave/turbulent boundary Layer Interaction,* Advanced Materials Research, Volume 516-517 (2): 954, ISSN 1022-6680

Launder, B. E. and Sharma, B. I. (1974): *Application of the Energy-Dissipation Model of Turbulence to the Calculation of Flow Near a Spinning Disc,* Letters in Heat and Mass Transfer, Volume 1(2): 131, ISSN 0094-4548

Li, Y., Peiro, J., Liu, C. H. (1998): *Finite-element multigrid scheme for the Navier-Stokes solutions. II. Formulation and validation,* Numerical heat transfer. Part B, Fundamentals, Volume. 34(1): 81, ISSN 1040-7790

Liu, C. H., Li Y. (2001): *Turbulence modeling for computing viscous high-Reynolds-number flows on unstructured meshes,* Computer Methods in Applied Mechanics and Engineering, Volume 190(40-41): 5325, ISSN 0045-7825

Liu, Y. W., Lu L. L., Fang, L. and Gao, F. (2011): *Modification of Spalart–Allmaras model with consideration of turbulence energy backscatter using velocity helicity,* Physics Letters A, Volume 375 (24) : 2377, ISSN 0375-9601

Lomtev, I., Karniadakis, G. E. (1999): *A discontinuous Galerkin method for the Navier-Stokes equations,* International journal for numerical methods in fluids, Volume 29 (5) : 587, ISSN 0271-2091

Lorin, E., Ben Haj Ali, A., et Soulaïmani, A. (2007): Lomtev, I., *A positivity preserving finite element-finite volume solver for the Spalart-Allmaras turbulence model*, Computer

Methods in Applied Mechanics and Engineering, Volume 196 (17-20) : 2097, ISSN 0045-7825

Lumley, J.L.(1997): *Some comments on turbulence,* Phys. Fluids A 4 : 203

Ma, L., Lu, L. P., Fang, J. and Wang, Q. H. (2014): *A study on turbulence transportation and modification of Spalart–Allmaras model for shock-wave/turbulent boundary layer interaction flow*, Chinese Journal of Aeronautics, Volume 27 (2): 200, ISSN 1000-9361

Martinez, M. J. and Gartling, D. K. (2004): *A finite element method for low-speed compressible flows,* Computer methods in applied mechanics and engineering, Volume 193 (21-22): 1959, ISSN 0045-7825

Menter, F. R. (1994): *Two-Equation Eddy-Viscosity Turbulence Models for Engineering Applications*, AIAA Journal, Volume 32(8): 1598, ISSN 0001-1452

MSC Software (2015), URL http://www.mscsoftware.com/product/msc-nastran

NASA (2014a), URL http://commonresearchmodel.larc.nasa.gov/

NASA (2014b), URL http://turbmodels.larc.nasa.gov/naca0012_val.html

NASA (2014c), URL http://aaac.larc.nasa.gov/tsab/cfdlarc/aiaa-dpw/Workshop4/presentations/DPW4_Presentations_files/D2-5_DPW4_CAdLabIISc_upload.pdf

Nazarov, M. and Hoffman, J. (2010): *An adaptive finite element method for inviscid compressible flow*, International Journal for numerical methods in fluids, Volume 64(10-12): 1102, ISSN 0271-2091

Navier, C.L.M.H. (1823): *Mémoire sur les lois du mouvement des fluides*, Mémoire de l'Académie Royale des Sciences de l'Institut de France, Volume 6 : 389

Nigro, N., Storti, M., Idelsohn, S., Tezduyar, T. (1998): *Physics based GMRES preconditioner for compressible and incompressible Navier-Stokes equations*, Computer methods in applied mechanics and engineering, Volume. 154(3-4): 203, ISSN 0045-7825

Nordanger, K., Holdahl, R., Kvarving, A. M., Rasheed, A., Kvamsdal, T. (2015): *Implementation and Comparison of three isogeometric Navier-Stokes Solvers Applied to Simulation of Flow Past a Fixed 2D NACA0012 Airfoil at High Reynolds Number*, Computer Methods in Applied Mechanics and Engineering, Volume 284: 664, ISSN 0045-7825

Paciorri, R., Dieudonnandé, W., Degrez, G., Charbonnier, J.-M., and Deconinck, H. (1998): *Exploring the validity of the Spalart-Allmaras turbulence model for hypersonic flows*, Volume 35(2): 121, Journal of Spacecraft and Rockets, ISSN 0022-4650

Pontaza, J. P. and Reddy J. N. (2003): *Spectral/hp least-squares finite element formulation for the Navier-Stokes equations,* Journal of Computational Physics, Volume 190 (2): 523, ISSN 0021-9991

Rachowicz, W. (2000): *An h-adaptive finite element method with highly stretched elements for compressible Navier-Stokes equations,* Computer methods in applied mechanics and engineering, Volume 189 (4): 1141, ISSN 0045-7825

Rahman, M. M., Siikonen, T., and Agarwal, R. K. (2011): *Improved Low Re-Number One-Equation Turbulence Model*, Volume 49 (4)

Rebaine, A., Soulaïmani, A. (2001): *Numerical simulation of two-dimensional compressible turbulent flows in ejectors,* Transactions of the Canadian Society for Mechanical Engineering, Volume 25 (2): 227, ISSN 0315-8977

Reynolds, O. (1883): *An experimental investigation of the circumstances which determine whether the motion of water shall be direct or sinuous, and of the law of resistance in parallel channels*. Philosophical Transactions of the Royal Society, Volume 174 (0): 935

Roy, C. J. and Blottner, F. G. (2003): *Methodology for turbulence model validation: application to hypersonic flows,* Journal of Spacecraft and Rockets, Volume 40(3): 313, ISSN 0022-4650

Rung, T., Bunge, U., Schatz, M. and Thiele, F. (2003): *Restatement of the Spalart-Allmaras Eddy-Viscosity Model in Strain-Adaptive Formulation*, AIAA Journal, Volume 41 (7): 1396, ISSN 0001-1452

Saad Y. (2003), *Iterative methods for sparse linear systems*, Society for Industrial and Applied Mathematics, 2st edition, ISBN 0898715342

Saad Y. and Schultz M.H. (1986): *GMRES*: *A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM Journal on Scientific and Statistical Computing, Volume 7 (3): 856

Shakib, F. (1989): *Finite element analysis of the compressible Euler and Navier-Stokes equations*, Doctoral thesis, Stanford University, CA

Shan, L. and Hou, Y.R. (2009): *A fully discrete stabilized finite element method for the time-dependent Navier-Stokes equations,* Applied Mathematics and Computation, Volume 215 (1): 85, ISSN 0096-3003

Smith, A. M. O. and Cebeci, T. (1967): *Numerical solution of the turbulent boundary layer equations,* Douglas aircraft division report DAC 33735

Soulaïmani, A., Ben Haj Ali, A. (2003): *Calcul parallèle-distrubué pour les problèmes multiphysiques : application à l'aéroélasticité*, Revue Européenne de Mécanique Numérique European Journal of Computational Mechanics (anciennement Revue Européenne des Éléments Finis (ISSN : 1250-6559), Hermes Science publications, Volume 12(7-8) :793, ISSN 1779-7179

Soulaïmani, A., Ben Haj Ali, A., et Feng, Z. (2004): *A parallel-distributed approach for multi-physics problems with application to computational nonlinear aeroelasticity*, Canadian Aeronautics and Space Journal, Volume 50(4) :221, ISSN 0008-2821

Soulaïmani, A., Salah N. B. and Saad Y. (2002a): *Enhanced GMRES Acceleration Techniques for some CFD problems*, International Journal of Computational Fluid Dynamics, Volume 16(1):1, ISSN 1061-8562

Soulaïmani, A., Ben Haj Ali, A., et Feng, Z. (2002b): *Nonlinear computational aeroelasticity : Formulations and solution algorithms*, V RTO-MP-089, (p. 45-0145-13), NATO-AVT, Meeting Proceedings, ISBN 92-837-0027-9

Soulaïmani, A., El Yamani, N., et Deschênes, C. (1994): *Une méthode d'éléments finis pour le calcul des écoulements compressibles utilisant les variables conservatives et la méthode supg*, Revue Européennes de Eléments finis, Volume 3(2) : 211

Soulaïmani, A. and Fortin, M. (1994): *Finite element solution of compressible viscous flows using conservative variables*, Computer Methods in Applied Mechanics and Engineering, Volume 118 (3-4) :319, ISSN 0045-7825

Soulaïmani, A., Saad, Y. and Rebain, A. (2001): *An edge based stabilized finite element method for solving compressible flows: Formulation and parallel implementation*, Computer Methods in Applied Mechanics and Engineering, Volume 190 (49-50) :6735, ISSN 0045-7825

Soulaïmani, A., Saad, Y. and Rebain, A. (2000): *Parallelization of the edge based stabilized finite element method using PSPARSLIB*, Parallel computational fluid dynamics, towards teraflops, optimization and Novel Formulations

Spalart, P. R. and Allmaras, S. R. (1992): *A one-equation turbulence model for aerodynamic flows*, AIAA Paper 92-0439

Steger, L. Joseph and Warming R. F. (1981): *Flux vector splitting of the invisid gasdynamic equations with application to finite-difference methods*, Journal of computational physics, Volume 40:263

Stokes, G.G. (1845): On the theories of the internal friction of fluids in motion, and of the equilibrium and motion of elastic solids, Transaction of the Cambridge Philosophical Society, Volume 8: 287

Erwin, J. T., Anderson, W. K., Kapadia, S, Wang, L. (2013): *Three-dimensional stabilized finite elements for compressible navier-stokes*, AIAA journal, Volume. 51(6): 1404, ISSN 0001-1452

Tezduyar, T. E. and Senga, M. (2006): *Stabilization and shock-capturing parameters in SUPG formulation of compressible flows*, Computer Methods in Applied Mechanics and Engineering, Volume 195(13-16): 1621, ISSN 0045-7825

Toro, E. F. (1999): *Riemann solvers and numerical methods for fluid dynamics*, Springer, 2nd Edition

Van Driest, E. R. (1956): *On turbulent flow near a wall*, Journal of the Aeronautical Sciences, Volume 23 (11): 1007, ISSN 0001-1452

Wang, L., Anderson W. K., Erwin, J. T. and Kapadia, S. (2014): *Discontinuous Galerkin and Petrov Galerkin methods for compressible viscous flows,* Computer & Fluids, Volume 100: 13, ISSN 0045-7930

Warming, R. F. Warming, Beam Richard M. and Hyett, B. J. (1975): *Diagonalization and simultaneous symmetrization of the gas-dynamic matrices,* Mathematics of computation, Volume 29 (12): 1037

Wervaecke, C., Beaugendre, H. and Nkonga, B, (2012): *A fully coupled RANS Spalart-Allmaras SUPG formulation for turbulent compressible flows on stretched-unstructured grids*, Computer Methods in Applied Mechanics and Engineering, Volume 233-236 : 109, ISSN 0045-7825

Wilcox, D.C. (1994): *Turbulence Modeling for CFD,* DCW Industries, La Canada, CA

Wilcox, D.C. (1988): *Re-assessment of the scale-determining equation for advanced turbulence models*, AIAA Journal, Volume 26 (11): 1299, ISSN 0001-1452

Wong, J. S., Darmofal, D. L. and Peraire, J. (2001): *The solution of the compressible Euler Equations at low Mach numbers using a stabilized finite element algorithm,* Computer Methods in Applied Mechanics and Engineering, Volume 190(43-44): 5719-37, ISSN 0045-7825

Yakhot, V., Orszag, S. A., Thangam, S., Gatski, T. B. and Speziale, C.G. (1992): *Development of turbulence models for shear flows by a double expansion technique*, Physics of Fluids A, Volume 4 (7): 1510, ISSN 0899-8213

Yan, B., Zhou, D. W., Huang, C., Wu Q. and Cheng X. Q. (2011): *Numerical prediction of aerodynamic characteristics of prismatic cylinder by finite element method with Spalart-Allmaras turbulence model*, Computer & Structure, Volume 89 (3-4): 325, ISSN 0045-7949

Yan, H., Liu, Y. W., Fang, L., and Lu L. P. (2014): *Modification of Spalart-Allmaras turbulence model for predicting S825 airfoil aerodynamic performance*, Applied Mechanics and Materials, Volume 543-547: 189, ISSN 1660-9336

Ypma, T. J. (1995): *Historical development of the Newton-Raphson method*, SIAM Review, Volume 37 (4): 531