Politecnico di Milano

FACOLTÀ DI INGEGNERIA INDUSTRIALE
DIPARTIMENTO DI INGEGNERIA AEROSPAZIALE
Corso di Laurea in Ingegneria Aeronautica

PERFORMANCE OF REYNOLDS AVERAGED NAVIER-STOKES MODELS IN PREDICTING SEPARATED FLOWS:

STUDY OF THE HUMP FLOW MODEL PROBLEM

Master's Thesis in Aeronautical Engineering

Supervisor:
Prof. Maurizio Quadrio

Advisor:
Nagi N. Mansour

Student:
DANIELE CAPPELLI
751032

Academic Year 20011-2012

Once you have tasted flight, you will
forever walk the earth with your eyes
turned skyward, for there you have been,
and there you will always long to return.

Leonardo da Vinci

Acknowledgments

Special thanks go to my parents for their love, understanding and financial support during my graduate studies.
I would like to thank my advisor Nagi N. Mansour who gave me the possibility to work for NASA during the realization of this work, in a stimulating, innovative environment.
I would also like to thank my supervisor at Politecnico di Milano, prof. Maurizio Quadrio, and my supervisors at the Ecole National Supérieure de l'Aéronautique et de l'Espace, prof. Allan Bonnet and prof. Emmanuel Bénard for the support they provided.
Last but not least, I would like to thank all the friends, who accompanied me in this wonderful experience of the double degree, in Italy, in France and in the USA, and all the friends who are eagerly waiting for my return home.
Thank you all.

Abstract

The separation problem affects most aerodynamic flows, but the accurate prediction of these flows is still a challenging problem for CFD.
This thesis will study the behavior of Reynolds Averaged Navier-Stokes (RANS) models in predicting the flow over a wall-mounted hump.
The goal of the work is to evaluate strengths and weaknesses of the most used and popular RANS models (*Spalart-Allmaras*, *k-epsilon*, *k-omega*, *k-omega-SST*) by using the open source software *OpenFOAM*.
Particular attention will be given to the size of the recirculation bubble, to the position of the reattachment point, and to the trend of velocity profiles downstream of the hump.
The hump modeled in this work is the same used in Rumsey's workshops in 2004 [1] and 2008 [2] and in the experiments conducted by Greenblatt et al. [3]; but here, since only the baseline case is treated, the slot for the flow control is not included.
Both 2D and 3D simulations are run in order to reproduce the effects of the blockage due to the end-side plates and particular attention is given to all parameters which can affect the accuracy of results: mesh resolution, boundary conditions, location of the inlet and development of the boundary layer upstream and downstream of the hump.
All models predict the separation point, the pressure distribution and the velocity profiles upstream and downstream of the hump in a proper way, but k-ϵ is the only model able to catch the exact location of the reattachment point.
The boundary condition used for the upper wall, (*symmetry plane* or *viscous wall*), does not change substantially the physics of the problem, even if the *viscous wall* condition can help in adjusting and improving the pressure coefficient distribution over the hump for the k-ω-SST model.
This thesis also shows that the development of the boundary layer upstream of the hump, which depends on the position of the numerical inlet, strongly affects the velocity profiles downstream of it.
Mesh resolution is another important aspect taken into account: in order to compute pressure and skin friction at the wall properly, the mesh has to be very fine in the y-direction, close to the hump.

Sommario

Il problema della separazione riguarda la maggior parte dei flussi aerodinamici, ma l'accurata predizione di tali flussi resta una grande sfida in ambito di calcolo numerico (CFD).
Questa tesi tratta il comportamento dei modelli RANS (Reynolds Averaged Navier Stokes models) nella predizione del flusso attorno da una parete con gobba.
L'obiettivo di questo lavoro è di valutare i punti di forza e di debolezza dei modelli RANS più usati e diffusi (*Spalart-Allmaras*, *k-epsilon*, *k-omega*, *k-omega-SST*), utilizzando il software open-source *OpenFOAM.*
Particolare attenzione sarà dedicata alle dimensioni della bolla di ricircolo, alla posizione del punto di riattacco e all'andamento dei profili di velocità a valle della gobba.
La gobba modellata in questo lavoro è la stessa usata da Rumsey nei suoi workshops del 2004 [1] e del 2008 [2] e negli esperimenti condotti da Greenblatt et al. [3]; ma questa tesi tratta il solo caso base, mentre il problema del controllo attivo e passivo non è affrontato.
Verranno eseguite simulazioni 2D e 3D, al fine di riprodurre gli effetti del bloccaggio delle pareti di estremità e verrà dedicata grande attenzione a tutti i parametri che possono influenzare l'accuratezza dei risultati: raffinatezza della mesh, condizioni al contorno, posizione della sezione di ingresso e sviluppo dello strato limite a monte e a valle della gobba.
Tutti i modelli predicono il punto di separazione, la distribuzione di pressione e i profili di velocità a monte e a valle della gobba in modo accurato, ma soltanto k-ϵ riesce a riprodurre l'esatta posizione del punto di riattacco.
La condizione al contorno usata per la parete superiore (*piano di simmetria* o parete viscosa) non cambia sensibilmente la fisica del problema, anche se la condizione di *parete viscosa* aiuta a migliorare la distribuzione del coefficiente di pressione attorno al modello per il modello k -ω-SST.
La tesi mostra che lo sviluppo dello strato limite a monte dell' "hump", che a sua volta dipende dalla posizione dell' inlet numerico, ha forte influenza sui profili di velocità a valle.
La risoluzione della mesh è un altro aspetto importante da tenere in conto: al fine di calcolare la pressione e il coefficiente d'attrito accuratamente, la griglia deve essere molto raffinata in direzione y, in prossimità della gobba.

Contents

List of Figures

List of Tables

Nomenclature

a_{ij}	Anisotropic stresses
c	Chord of the hump
$\bar{c}$	Speed of sound
C_1	Coefficient of the production term of ϵ for the $k-\epsilon$ model
C_2	Coefficient of the dissipation term of ϵ for the $k-\epsilon$ model
C_f	Skin friction coefficient
C_μ	Coefficient of eddy viscosity for the $k-\epsilon$ model
D_k	Viscous diffusion of turbulent kinetic energy
D_ϵ	Viscous diffusion of dissipation
k	Turbulent kinetic energy
Kn	Knudsen number
M	Mach number
P	Production
Re_c	Reynolds number based on the chord of the hump
S	Outward-pointing face area vector
S_{ij}	Rate-of-strain tensor
t	Time
T	Time interval
T_k	Transport of turbulent kinetic energy
T_ϵ	Transport of dissipation
$\mathbf{x}$	Space vector
u	Scalar component of velocity
$\mathbf{u}$	Velocity vector
u_τ	Frcition velocity $\sqrt{\tau/\rho}$
U_∞	Asymptotic velocity
α	Coefficient of the production term of ω for the $k-\omega$ model
α_ω	Coefficient of the viscous term of the ω equation
β	Coefficient of the dissipation term of ω in the $k-\omega$ model
ϵ	Rate of dissipation of turbulent kinetic energy
Γ	Dissipation, equation for ϵ
λ	Molecular free path
ω	Specific dissipation of turbulent kinetic energy
Π_k	Pressure diffusion, equation for turbulent kinetic energy
Π_ϵ	Pressure transport, equation for dissipation
σ_ϵ	Coefficient of the viscous term in the ϵ equation
ρ	Density
ν	Kinematic viscosity
ν_t	Turbulent kinematic viscosity

ν_{eff}	Effective viscosity ($\nu + \nu_t$)
τ	Time scale
Subscript	
i	Index for the scalar component of a vector
j	Index for the scalar component of a vector
N	Center of the neighboring cells
P	Center of the cell
f	Face
Apex	
$-$	Time average
$'$	Time fluctuation

Chapter 1

Introduction

This thesis is inspired by Rumsey's workshops of 2004 [1] and 2008 [2] and its goal is to update the state of the art for what concerning the "hump-flow" model problem and RANS models in predicting separated flows.
Even though, since the time of the workshops, many research centers have been working on this problem, many aspects, concerning the characteristics of this flow and the behavior of CFD methods are not totally clear, so that further investigations are required.
This work compares the performances of the most popular *RANS* models and discusses the effects that procedural variables have on the accuracy of results.
A sketch of the geometry used in experiments is shown in figure 1.1.
The model was constructed from aluminum over a splitter-plate and the assembly was mounted between two end-plates with aluminum frames and glass interiors[3].
Numerical simulations of the hump model were object of study during the CFD Validation Workshop, held in Williamsburg, Virginia, in 2004. The goal of this workshop, fully described in Rumsey's paper[1], was to bring together an international group of computational fluid dynamics practitioners to assess the current capabilities of different classes of different flow solution methodologies to predict flow fields induced by synthetic jets and separation control geometries.
For the base-line (no flow control), most CFD results missed the pressure levels over the hump between $0.2 < x/c < 0.6$ and also predicted higher results in the separated region upstream of $x/c = 1$.
This trend was in part due to the blockage of the side plates used in the experiments that caused a decrease of pressure, when compared to 2D simulations.
The most significant results concerned with the separated flow downstream of the model: the separation location was predicted reasonably well by most of the CFD methods, while the reattachment location was predicted significantly downstream of the experimental location $x/c = 1.1$.
According to the author, a possible reason for the reattachment being predicted too late was that most of the methods predicted the magnitude of the turbulent shear stress to be too small in the separated region.

In 2008 a new survey was made by Rumsey[2]. According to Rumsey, no major progress was made since the time of the first workshop in terms of RANS/URANS. Results seemed to be still very consistent in under predicting the eddy viscosity in the separated region and over-predicting the size of the bubble.
He et Al.[9] obtained reasonable good results by using the commercial software *Fluent* with second order upwind and SIMPLE algorithm of pressure-velocity coupling. In this case, k-ϵ agreed well with the experimental reattachment location but still underpriced the magnitude of the

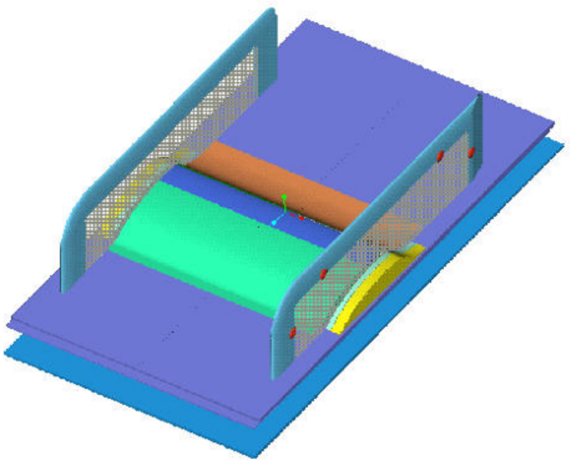


Figure 1.1: Isometric view of the hump model[3].

turbulent shear-stress.
These results brought into the question as why the same software yielded different results for Brettini and Cravero[11].
The answer may be in the fact that this flow is strongly sensitive to the resolution of the mesh, the position of the inlet and to the particular boundary conditions employed.

Our work will focus on these aspects and will put in evidence how these settings affect the solution.
After a brief introduction to the SIMPLE algorithm implemented in *OpenFOAM*, we will show our results for the baseline[1] by using 4 different turbulent models (*Spalart-Allmaras*, *k-epsilon*, *k-omega*, *k-omega-SST*) and by analyzing weaknesses and strengthens of each of them.
The effect that the position of the inlet, the boundary condition on the upper wall and the resolution of the mesh have on the hump-flow will be discussed, as well.
The work is structured as follows:

- **Chapter 2** will provide a theoretical background about the *RANS* equations, Boussinesq's hypothesis and *RANS* models (the most popular 1 equation and 2 equation models) used to model the eddy viscosity. It will be useful to understand the limits of the theory and the range of applicability of *RANS* models.

- **Chapter 3** will show the state of the art for the "hump-flow" model problem. The most

[1]The baseline will not include the plenum, since the problem of the flow-control is not dealt with.

important problematics encountered in the 2004 and in the 2008 workshops are shown, with particularly attention to RANS equations, which represent the less accurate tool to compute separated flows.

- **Chapter 4** will describe the basics of the *SIMPLE* algorithm implemented in *OpenFOAM* and how a case is generated. The code will be translated into math and the final equations for momentum and turbulent models will be compared to the theoretical ones.
- **Chapter 5** will describe the hardware and the software used to run the simulations. After a brief introduction to the super-computer facility at *NASA Ames*, the procedure to run and to post-process the simulations will be shown.
- **Chapter 6** contains most of the significant results concerning the simulations run by using the most popular *RANS* models: $Spalart - Allmaras$, $k - \epsilon$, $k - \omega$ and $k - \omega - SST$. Numerical results will be compared to experimental data in terms of pressure coefficient, skin friction coefficient, separation point, reattachment point, size of the bubble, velocity profile upstream and downstream of the hump and turbulent shear stress.
- **Chapter 7** contains the conclusion and the future developments required to improve the reliability of *RANS* models.
- In the **Appendix** some of the source files required to generate a case are attached. Here, more specific information about how to generate the mesh, how to set initial and boundary conditions, how to specify the numerical schemes and the properties of the fluid can be found.

Chapter 2

Turbulence Models

2.1 RANS Equations

Navier Stokes equations, for an incompressible flow can be written as follows:

$$\begin{cases} \nabla \cdot (\mathbf{u}) = 0 \\ \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\frac{1}{\rho}\nabla p + \nu \Delta \mathbf{u} \end{cases}$$

Remark that:

- Mass conservation is expressed by an equation which is not a rate equation;
- Pressure does not satisfy a state equation: it only acts as a Lagrange multiplier to force the mass conservation incompressibility constraint;
- Energy equation decouples, so that momentum equation is enough to determine the unknowns of the problem.

This equation includes and models all the physical phenomena concerning incompressible flows. A direct numerical simulation implementing this system of equations would be able to provide the exact[1] solution for the problem, after having fixed the appropriate boundary conditions and initial conditions.
Unfortunately, even considering the increasing evolution of modern computer devices, the computational power required to solve a direct numerical simulation is still too big, for a problem of practical interest. This is why it is required to model some of the terms of the momentum equations which would otherwise require too much computational effort to be calculated.
To highlight these terms, velocity can be decomposed in the following way:

$$u'(x,t) = u(x,t) - \bar{u}(x) \tag{2.1}$$

This approach, derived by Reynolds for the first time in 1894, is also known as *Reynolds decomposition*, where $\overline{u}$ is the average velocity and u' its fluctuation.

The average velocity is defined as:

$$\bar{u}(x) = \langle u(x,t) \rangle = \lim_{T \to \infty} \frac{1}{T} \int_0^T U(x,t) \tag{2.2}$$

[1]When we run numerical simulations, speaking of "exact" results is improper, since our solutions are always affected by numerical errors. We suppose that this error is small enough to be neglected.

and it is independent of time, while $u'(x,t)$ is the fluctuating part, and depends on time.
It follows from the continuity equation that both $\bar{u}(x)$ and $u'(x,t)$ are solenoidal: the average velocity obeys the following equation

$$\nabla\bar{\mathbf{u}} = 0 \tag{2.3}$$

and by subtraction:

$$\nabla\mathbf{u}' = 0 \tag{2.4}$$

For what concerning the momentum equation, taking the mean is less simple, because of the non linear convective term:

$$(\mathbf{u}\cdot\nabla)\mathbf{u} = ((\bar{\mathbf{u}}+\mathbf{u}')\cdot\nabla)(\bar{\mathbf{u}}+\mathbf{u}') \tag{2.5}$$

The means of this equation leads to:

$$\overline{(\mathbf{u}\cdot\nabla)\mathbf{u}} = \overline{\bar{\mathbf{u}}\nabla\bar{\mathbf{u}}} + \overline{\mathbf{u}'\nabla\bar{\mathbf{u}}} + \overline{\mathbf{u}'\nabla\mathbf{u}'} + \overline{\bar{\mathbf{u}}\nabla\mathbf{u}'} \tag{2.6}$$

It can be demonstrated that the term $\overline{\bar{\mathbf{u}}\nabla\mathbf{u}'}$ is zero [12]:

$$\overline{\bar{\mathbf{u}}\nabla\mathbf{u}'} = \lim_{T\to\infty}\int_0^T \bar{\mathbf{u}}\nabla\mathbf{u}'dt = \bar{\mathbf{u}}\left[\lim_{T\to061\infty}\int_0^T \nabla\mathbf{u}'dt\right] = \bar{\mathbf{u}}\nabla\left[\lim_{T\to\infty}\int_0^T \mathbf{u}'dt\right] = \bar{\mathbf{u}}\nabla\bar{\mathbf{u}'} = 0 \tag{2.7}$$

While,

$$\overline{\bar{\mathbf{u}}\nabla\bar{\mathbf{u}}} = \overline{\nabla\bar{\mathbf{u}}^2} = \nabla\bar{\mathbf{u}}^2 \tag{2.8}$$

$$\overline{\mathbf{u}'\nabla\mathbf{u}'} = \overline{\nabla\mathbf{u}'^2} = \nabla\overline{\mathbf{u}'^2} \tag{2.9}$$

The final form of momentum equation is, for the steady case,

$$\nabla\cdot(\bar{\mathbf{u}}\bar{\mathbf{u}}) + \nabla\cdot(\overline{\mathbf{u}'\mathbf{u}'}) = -\frac{1}{\rho}\nabla\bar{p} + \nu\nabla^2\bar{\mathbf{u}} \tag{2.10}$$

also known as Reynolds Averaged Navier-Stokes equations or, more simply, *RANS* equations. RANS equations do not differ from the Navier-Stokes equations, except for the highlighted term which represents the covariance of velocities, also known as *Reynolds stresses*[2]. Like $p(x,t)$, $\bar{p}(\mathbf{x})$ satisfies a Poisson equation. This may be obtained by taking the divergence of the Reynolds equations [3].
The four equations written above (the continuity equation - or, alternatively, Poisson's equation - and the 3 scalar equations for momentum) are not enough anymore to solve the problem, since these equations contain more than four unknowns, because of the appearance of the Reynolds stresses: this issue leads to the problem of *closure*. In order to close the system, Reynolds stresses have to be written as function of the other unknowns. In order to close the equations, turbulence models are used.

2.2 Reynolds stresses

By bringing the Reynolds stresses to the right-hand side of the RANS equations, equation 2.10 can be rewritten as follows:

$$\nabla\cdot(\bar{\mathbf{u}}\bar{\mathbf{u}}) = -\frac{1}{\rho}\nabla\bar{p} + \nu\nabla^2\bar{\mathbf{u}} - \nabla\cdot(\overline{\mathbf{u}'\mathbf{u}'}) \tag{2.11}$$

[2]The reason for which they are called "stresses" will be explained in the next section

[3]We obtain: $-\frac{1}{\rho}\nabla^2\bar{p} = \langle\frac{\partial u_i}{\partial x_j}\frac{\partial u_j}{\partial x_i}\rangle = \frac{\partial\overline{u_i}}{\partial x_j}\frac{\partial\overline{u_j}}{\partial x_i} + \frac{\partial^2\overline{u_i'u_j'}}{\partial x_i\partial x_j}$

By multiplying by density and by using Einstein notation, the following relation is derived:

$$\rho \bar{u}_i \frac{\partial \bar{u}_j}{\partial x_i} = \frac{\partial}{\partial x_i}\left[\mu\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right) - \bar{p}\delta_{ij} - \rho\overline{u'_i u'_j}\right] \tag{2.12}$$

The right-hand side terms represent now the sum of three stresses[13]: the viscous stress, the isotropic stress on the mean pressure field and the apparent stress arising from the fluctuating velocity field, $-\rho\overline{u'_i u'_j}$. The viscous stress ultimately stems from momentum transfer at molecular level; the Reynolds stress stems from momentum transfer by the fluctuating velocity field. This explains why it is common to refer to this term as a stress.
It is important to remark that, while viscosity is an intrinsic propriety of the fluid, Reynolds stresses are an exclusive propriety of the motion of the fluid: the mathematical analogy between these two terms is not justified from a physical point view.
The most important proprieties of Reynolds stresses are now listed:

- The Reynolds stresses are identified by a second order tensor which is obviously symmetric: $\overline{u'_i u'_j} = \overline{u'_j u'_i}$;
- The diagonal components are *normal stresses*, while the off-diagonal components are *shear stresses*;
- The turbulent kinetic energy is defined as half of the trace of the Reynolds stress tensor: $k = \frac{1}{2}\overline{u'_i u'_i}$;
- The distinction between normal and shear stresses depends on the choice of the coordinate system; an intrinsic distinction can be made between *isotropic* and *anisotropic* stresses: if $\frac{2}{3}k\delta_{ij}$ is the isotropic stress, the anisotropic part can be defined as:

$$a_{ij} = \overline{u'_i u'_j} - \frac{2}{3}k\delta_{ij} \tag{2.13}$$

  The importance of this part is that it is effective in transporting momentum.

2.3 Appraisal of Boussinesq's Hypothesis

In order to close the system, Reynolds stresses must be expressed as a function of the other dependent variables, otherwise the system would not admit any solution (we would have more unknowns than equations).
By making the hypothesis that Reynolds stresses are aligned to the mean rate-of-strain tensor, a positive scalar quantity, called *eddy viscosity*, $\nu_t(\mathbf{x})$, can be defined, so that the anisotropic term can be modeled as follows:

$$a_{ij} = -(\rho\overline{u'_i u'_j} - \frac{2}{3}\rho k\delta_{ij}) = -\rho\nu_t(\mathbf{x})\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right) \tag{2.14}$$

By making the *intrinsic* hypothesis that a_{ij} only depends upon mean velocity gradients, and the specific *hypothesis* that:

$$a_{ij} = -2\nu_t(\mathbf{x})\overline{S_{ij}} \tag{2.15}$$

the final form for a_{ij} is obtained. This result is also known as *Boussinesq's hypothesis*. $\overline{S_{ij}}$ is the characteristic mean strain rate, defined as $S_{ij} = \frac{1}{2}(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i})$. An effective viscosity, $\nu_e = \nu + \nu_t$ can be now defined, so that 2.10 can be written as follows:

$$\nabla \cdot (\bar{\mathbf{u}}\bar{\mathbf{u}}) = -\nabla\bar{p} + \nabla \cdot (\nu_e(\mathbf{x})\nabla\bar{\mathbf{u}}) \tag{2.16}$$

The problem is now closed (ν_t needs to be modeled since it is the only unknown of the problem) but, before going on, it is interesting to evaluate strengths and weaknesses of this approach. The advantages of this method are its simplicity, coming from the direct analogy between viscous stresses and Reynolds stresses, and its being the point of departure for the development of models[4] able to provide good results in several cases and applications.
However, there are some weaknesses that need to be discussed.

- The relation 2.15 is *inconsistent*: if we look at it, it is evident that in turbulence shear flows, for instance, where $\overline{Sii}$ is zero, a_{ii} is zero as well, even if normal Reynolds stresses are not;
- a_{ij} is *not invariant*;
- a_{ij} is not aligned with $\bar{S}_{ij}$.

The similitude between viscosity and eddy viscosity presents another important aspect that has to be considered: viscosity is a fluid property which represents its molecular behavior[5], while eddy viscosity is a property of motion. In order to calculate the *molecular* timescale and the *turbulent* timescale, the following relationships are applied:

$$\tau_m = \frac{\lambda}{\bar{c}} \tag{2.17}$$

where λ is the molecular free path, and $\bar{c}$ is the speed of sound; while turbulent timescale, is:

$$\tau_t = \frac{k}{\epsilon} \tag{2.18}$$

where k is the production of turbulent kinetic energy, while ϵ is its dissipation[6]. By comparing the molecular and the turbulent timescale to that of the shear[7], the following relationship is derived :

$$\frac{\tau_m}{\tau_s} \sim \frac{\lambda}{\bar{c}} S = \frac{\lambda}{L} \frac{U}{\bar{c}} = KnM << 1 \tag{2.19}$$

while, for turbulent timescale:

$$\frac{\tau_t}{\tau_S} \sim \frac{Sk}{\epsilon} = O(1) \tag{2.20}$$

This means that, while molecular timescale is independent from shear timescale, turbulent timescale is not. Ultimately, turbulence is not independent from the change of the mean velocity gradients.
The fact that turbulence has memory is evident in Uberoi (1956) and Tucker(1970) experiments.

[4]Models which will be described further, such as *k-epsilon*, *k-omega*, *Spalart-Allmaras...*

[5]Viscosity is the statistic representation of molecular momentum exchange in y-direction [14]

[6]These two terms will be fully desribed and calculated in the next sections

[7]Shear is defined as $S_{ij} = \frac{1}{2}(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i})$, that is, for a simple shear flow, $S = \frac{\partial \bar{u}_1}{\partial x_2}$

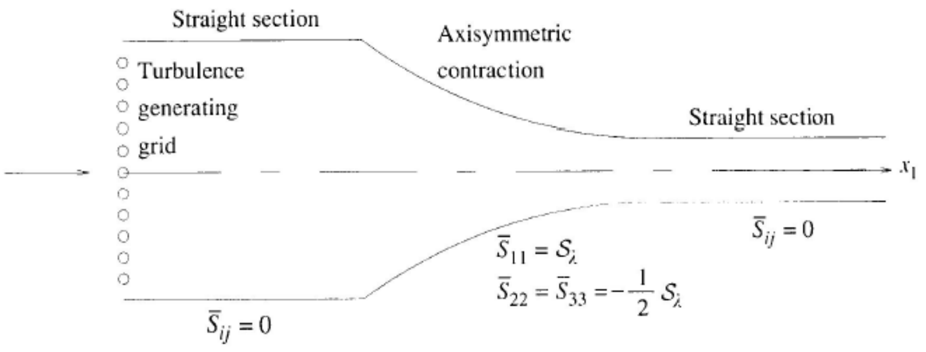

Straight section
Axisymmetric contraction
Turbulence generating grid
Straight section
x_1
$\bar{S}_{ij} = 0$
$\bar{S}_{11} = \mathcal{S}_\lambda$
$\bar{S}_{22} = \bar{S}_{33} = -\frac{1}{2}\mathcal{S}_\lambda$
$\bar{S}_{ij} = 0$

Figure 2.1: The apparatus used by Uberoi and Tucker in their experiments.

Figure 2.1 is a sketch of the wind tunnel used by Uberoi and Tucker to study the effects on turbulence of an axisymmetric contraction. By theory, Reynolds stress anisotropies should be zero on the straight sections, since there is no mean straining in there.
Unfortunately, because of the considerations about turbulent timescale, experimental data clearly shows that anistotropy is different from zero after the contraction. (fig 2.2).

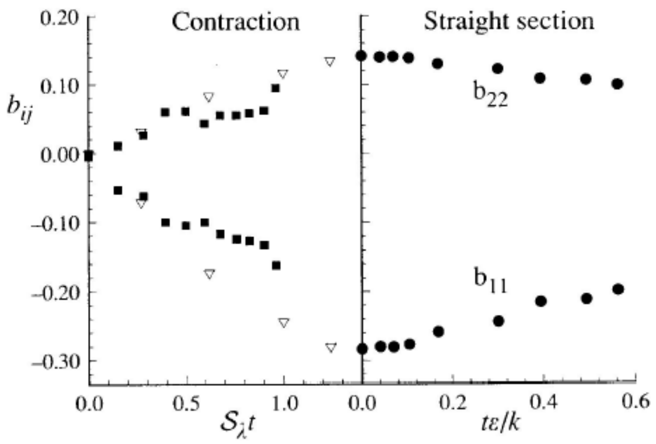


Figure 2.2: Reynolds-stress anisotropies during and after axisymmetric contraction: experimental data of Tucker(1970) and DNS data of Lee and Reynolds(1985)[4]

This leads to the conclusion that eddy viscosity is not fully adequate for:

- Flows with abrupt change of shear rate;
- Flows over curved surfaces;
- Flows in ducts with secondary motions and/or separations;

- Rotating or stratified flows;
- Three-dimensional flows;
- Many others...

2.4 One equation models

The first model implementing only one equation was the model for k. This model only implemented a PDE transport equation for turbulent kinetic energy, as follows:

$$\bar{u}_i \frac{\partial k}{\partial x_i} = \frac{\partial}{\partial x_i}\left[\left(\nu + \frac{\nu_T}{\sigma_k}\right)\frac{\partial k}{\partial x_i}\right] - \overline{u_i' u_j'}\frac{\partial \bar{u}_i}{\partial x_j} - C_d \frac{k^{3/2}}{l_m} \tag{2.21}$$

where the black terms are the exact ones, while the red terms are the ones which are modeled. The first term of this equation represents the mean convenction, the second term is the redistribution term, the last two are, respectively, production and dissipation.
After solving the equation for k, it is possible to calculate the turbulent kinematic viscosity, defined as:

$$\nu_t = c\sqrt{k} l_m$$

c and C_d are empirical coefficient, σ_K is the turbulent Prandtl number for k and it is generally chosen to be 1, while l_m represents the mixing length and it is unknown.
This is why this model is incomplete and makes l_m depend on the particular flow geometry.
The first complete one-equation model has been the *Spalart-Allmaras* model which represents the lowest level at which a model can be complete.
It consists in resolving a transport equation for ν_t composed by 3 empirical functions and 8 empirical coefficients.
The model is designed for aerodynamic flows, such as transonic flow over airfoils, including boundary-layer separation.
The equation for ν_t can be schematised as follows:

$$\frac{\bar{D}\nu_t}{\bar{D}t} = \nabla \cdot \left(\frac{\nu_t}{\sigma_\nu}\nabla\nu_t\right) + S_\nu \tag{2.22}$$

where the source term S_ν depends on the laminar and turbulent viscosities, ν and ν_t; the mean vorticity (or rate of rotation) Ω; the turbulent viscosity gradient and the distance to the nearest wall.
The details of the model are quite complicated: the reader is referenced to the original paper or to the *OpenFOAM* source code in Annex, for the complete equation.

2.5 Two equation models

This section illustrates the two equation models used to solve the closure problem. These models are so called because two turbulence quantities are modeled, so that they provide "more accurate" solutions, whereas calculation time increases.
By definition, two equation models include two extra transport equations to represent the turbulent properties of the flow. This allows to account for historical effects like convection and diffusion of turbulent energy.
Most often one of the two transported variables is the *turbulent kinetic energy*, k, while the second

transported variable depends on the type of two-equation model. The first variable determines the energy in the turbulence, while the second determines the scale of the turbulence (length-scale or time scale). Common choices are the *turbulent dissipation*, ϵ, or the *specific dissipation*, ω.
Models like the k-ϵ model and the k-ω model have become industry standard models and are commonly used for many types of engineering problems.

2.5.1 k-ϵ Model

If the turbulent-viscosity hypothesis is accepted, all that remains is to determine an appropriate specification of the turbulent viscosity $\nu_t(\mathbf{x},t)$. This can be written as the product of a velocity $u^*(\mathbf{x},t)$ and a length $l^*(\mathbf{x},t)$:

$$\nu_t = u^* l^* \tag{2.23}$$

The k-ϵ modelers approximate the eddy viscosity ν_t as follows[15]:

$$\nu_t = C_\mu \left(\frac{k^2}{\epsilon} \right) \tag{2.24}$$

The reason is that k and ϵ can be combined into a lengthscale ($L = k^{(3/2)}/\epsilon$) and a timescale ($\tau = k/\epsilon$), as a consequence from 2.23 it is possible to derive 2.24.
The value of the constant is chosen to be $C_\mu = 0.09$. In simple turbulent shear flow, in fact, the $k-\epsilon$ model yields:

$$\frac{\overline{u'v'}}{k} = \left(C_\mu \frac{P}{\epsilon} \right)^{\frac{1}{2}} \tag{2.25}$$

Empirical observation leads to $\frac{\overline{u'v'}}{k} = 0.3$ in regions where production is equal to dissipation, so that $C_\mu = 0.09 = (0.3)^2$

Equation for k

For an incompressible flow, the exact equation governing the transport of k is

$$\frac{\partial k}{\partial t} + \overline{u_i} \frac{\partial k}{\partial x_i} = P_k + T_k + \Pi_k + D_k - \epsilon \tag{2.26}$$

Where the different terms on the right-hand side are given as rate of [5]:

- Production:
$$P_k = -\overline{u_i' u_j'} \frac{\partial \overline{u_i}}{\partial x_j} \tag{2.27}$$

- Turbulence transport:
$$T_k = -\frac{1}{2} \frac{\partial}{\partial x_i} (\overline{u_i' u_j' u_j'}) \tag{2.28}$$

- Pressure diffusion:
$$\Pi_k = -\frac{1}{\rho} \frac{\partial}{\partial x_i} (\overline{u_i' p'}) \tag{2.29}$$

- Viscous diffusion:
$$D_k = -\frac{\partial}{\partial x_i} 2\nu \overline{u_j' \frac{\partial u_i'}{x_j}} \tag{2.30}$$

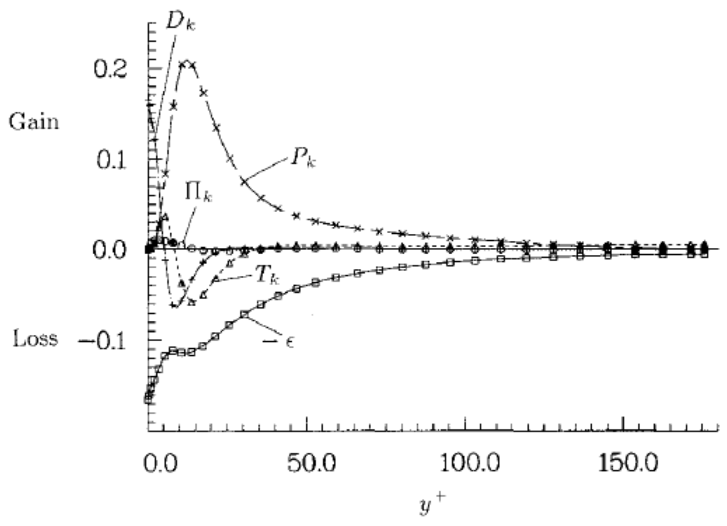


Figure 2.3: Terms in the budget of the turbulence kinetic energy in wall coordinate [5].

- Dissipation

$$\epsilon = -2\nu \overline{\frac{\partial u_i'}{\partial x_j}\frac{\partial u_i'}{\partial x_j}} \tag{2.31}$$

Figure 2.3 shows the various terms computed from the channel data of KMM as a function of variable y^+. The salient feature of this plot is that, away from the wall, the production rate is almost balanced by the dissipation rate. Close to the wall the production rate and dissipation are still the dominant terms, but the turbulent transport rate and viscous diffusion rate are no longer negligible.
The turbulent diffusion rate has a positive peak at $y^+ = 6$ and a negative peak $y^+ = 15$. At the wall, the left-hand side of equation 2.26 vanishes and the diffusion rate exactly balances the dissipation rate. After having analyzed the exact equation for k, the goal is to model it.
The viscous diffusion term needs not be modeled, but the pressure-diffusion term and turbulent-transport term are usually added and modeled as one term:

$$T_k + \Pi_k = \frac{\partial}{\partial x_i}\left(\nu_T \frac{\partial k}{\partial x_i}\right) \tag{2.32}$$

Figure 2.4 shows the distribution of the turbulence transport term compared to the eddy-viscosity model using $\nu_t = \frac{-\overline{u_1'u_2'}}{\frac{\partial U}{\partial y}}$, from the data. In the vicinity of the wall the model has a different slope than the data would indicate [5].

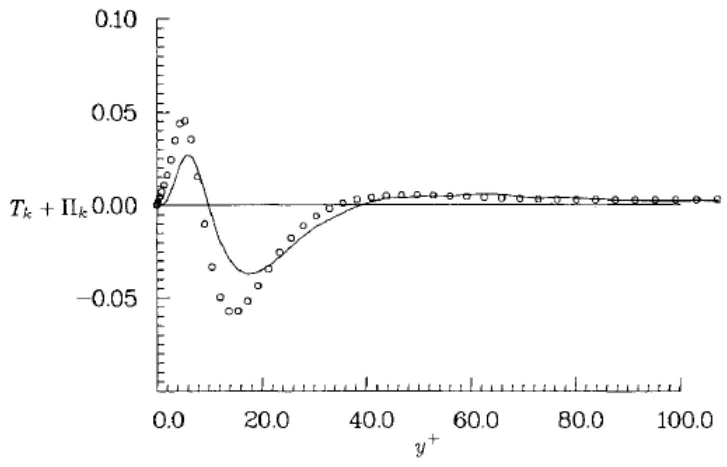


Figure 2.4: Pictures taken from [5]. Transport terms $T_k + \Pi_k$ across the channel: circles = term computed by [5], from the channel data [6], line = model.

The production term is modeled by substituting the model for $-\overline{u_i'u_j'}$ in the production rate expression,

$$P_k = -\overline{u_i'u_j'}\frac{\partial \overline{u_i}}{\partial x_j} = 2\nu_t\overline{S_{ij}S_{ij}} \tag{2.33}$$

In this case, the data match exactly. The final form of the modelled equation for k is:

$$\frac{\partial k}{\partial t} + \frac{\partial}{\partial x_i}(k\bar{u}_i) = \frac{\partial}{\partial x_i}\left(\frac{\nu_t}{\sigma_k}\frac{\partial k}{\partial x_i}\right) + P_k - \epsilon \tag{2.34}$$

Where σ_k is the "turbulent Prandtl number" for kinetic energy and it is generally taken to be $\sigma_k = 1$.

Equation for ϵ

The exact equation describing the evolution of ϵ is:

$$\frac{\partial \epsilon}{\partial t} + \frac{\partial}{\partial x_i}(\epsilon\bar{u}_i) = P_\epsilon^1 + P_\epsilon^2 + P_\epsilon^3 + P_\epsilon^4 + T_\epsilon + \Pi_\epsilon + D_\epsilon - \Gamma \tag{2.35}$$

The different terms on the right-hand can be treated as rate of:

- Mixed production:

$$P_\epsilon^1 = -2\nu\left(\overline{\frac{\partial u_i'}{\partial x_k}\frac{\partial u_j'}{\partial x_k}}\right)\frac{\partial \bar{u}_i}{\partial x_j} \tag{2.36}$$

- Production by mean velocity gradient:

$$P_\epsilon^1 = -2\nu\overline{\frac{\partial u_k'}{\partial x_i}\frac{\partial u_k'}{\partial x_j}}\frac{\partial \bar{u}_i}{\partial x_j} \tag{2.37}$$

- Gradient production:

$$P_\epsilon^3 = -2\nu \overline{u_k' \frac{\partial u_k'}{\partial x_j}} \frac{\partial^2 \bar{u}_i}{\partial x_k \partial x_j} \tag{2.38}$$

- Turbulent production:

$$P_\epsilon^4 = -2\nu \overline{\frac{\partial u_i'}{\partial x_k} \frac{\partial u_i'}{\partial x_m} \frac{\partial u_k'}{\partial x_m}} \tag{2.39}$$

- Turbulent transport:

$$T_\epsilon = -\frac{\partial}{\partial x_j}\left(\nu \overline{u_j' \frac{\partial u_i'}{\partial x_m} \frac{\partial u_i'}{\partial x_m}}\right) \tag{2.40}$$

- Pressure transport:

$$\Pi_\epsilon = -\frac{\partial}{\partial x_j}\left(2\frac{\nu}{\rho} \overline{\frac{\partial p'}{\partial x_m} \frac{\partial u_j'}{\partial x_m}}\right) \tag{2.41}$$

- Viscous diffusion:

$$D_\epsilon = \frac{\partial}{\partial x_j}\left(\nu \frac{\partial \epsilon}{\partial x_j}\right) \tag{2.42}$$

- Dissipation:

$$\Gamma = -\frac{\partial}{\partial x_j}\left(\nu \overline{u_j' \frac{\partial u_i'}{\partial x_m} \frac{\partial u_i'}{\partial x_m}}\right) \tag{2.43}$$

The various terms in the balance equation for ϵ are shown in Fig. 2.5. These results indicate that P_ϵ^4 and Γ are the largest terms in the core region of the channel. Near the wall, these terms are still significant but are not larger than the other terms. Close to the wall ($y^+ < 8$), the production rate P_ϵ^1 becomes of the same order as P_ϵ^4. In the range $6 < y^+ < 15$, the production rate by mean velocity gradient P_ϵ^2 is of the same order as P_ϵ^4.

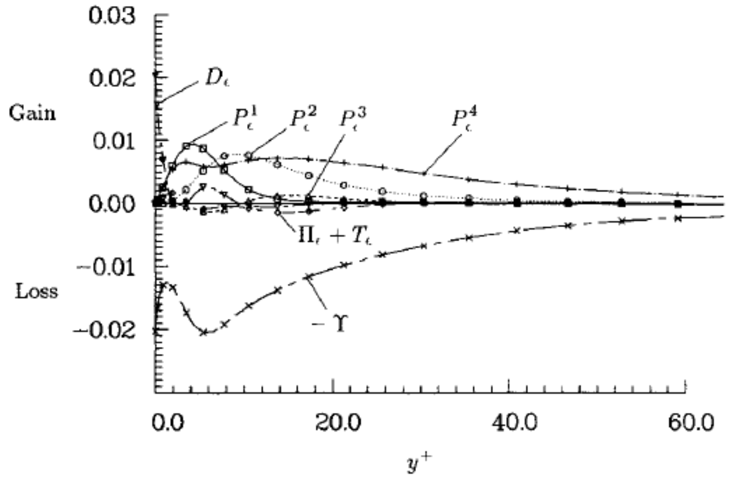


Figure 2.5: Terms in the budget of the dissipation rate of the turbulence kinetic energy ϵ in wall coordinates [5].

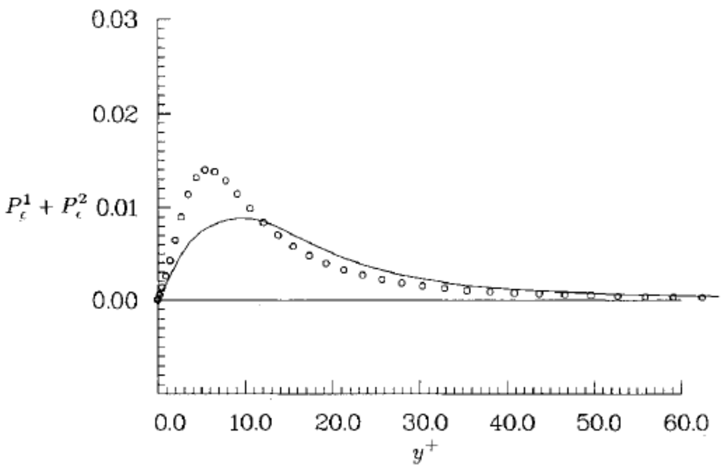


Figure 2.6: Production of the dissipation rate of turbulent kinetic energy: Circlets = terms computed from the channel data[6], line = model [5].

The equation for ϵ is closed by modeling the terms of the equation 2.35. To represent the first two production terms, an expression for them in terms of k, ϵ and other mean quantities is required. These two terms have the same trace and they are related for homogeneous flows through the vorticity fluctuation. By assuming that Rotta's approximation is valid for both terms, the following relationship is obtained:

$$-2\nu\left(\overline{\frac{\partial u_i'}{\partial x_k}\frac{\partial u_j'}{\partial x_k}}\right)\frac{\partial \bar{u}_i}{\partial x_j} - 2\nu\overline{\frac{\partial u_k'}{\partial x_i}\frac{\partial u_k'}{\partial x_j}}\frac{\partial \bar{u}_i}{\partial x_j} = -C_1\frac{-\overline{u_i'u_j'}}{k}\epsilon \tag{2.44}$$

and by substituting the Boussinesq approximation for the Reynolds stresses, we obtain:

$$P_\epsilon^1 + P_\epsilon^2 = C_1\frac{\epsilon}{k}\nu_T\bar{s_{ij}}\bar{s_{ij}} \tag{2.45}$$

Fig. 2.6 shows the model compared to the exact expression, using the constant recommended by CH, $C_1 = 1.35$. The model yields a lower peak than the data would indicate. The term P_ϵ^3 is negligible compared to the other terms in the channel flow, so no explicit expression is used to model it. The turbulent production term P_ϵ^4 is expected to be non negligible even in isotropic flows; an appropriate model for this model will be a function of k and ϵ. Dimensional analysis yields $P_\epsilon^4 \sim \epsilon^2/k$. The same arguments are used in modeling dissipation Γ, so that the model for the combined terms is given as:

$$P_\epsilon^4 - \Gamma = -C_2\frac{\epsilon^2}{k} \tag{2.46}$$

Figure 2.7 shows the comparison of this model ($C_2 = 1.8$) with the data: the model adequately compare with the data in the $y^+ > 11$ range but underpredicts the data close to the wall. The remaining terms in the balance equation of ϵ are transport rate terms, which are grouped together and modeled using an eddy-viscosity-diffusion model:

$$T_\epsilon + \Pi_\epsilon = \frac{\partial}{\partial x_j}\left(\frac{\nu_T}{\sigma}\frac{\partial \epsilon}{\partial j}\right) \tag{2.47}$$

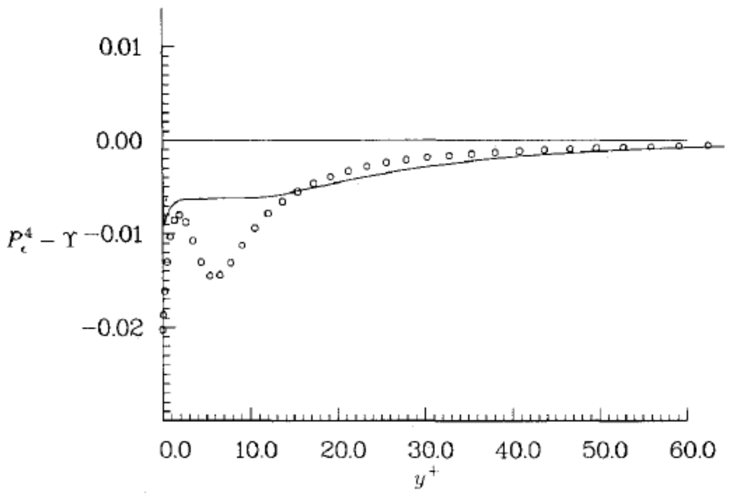


Figure 2.7: Net dissipation rate of ϵ: Circlets = terms computed from the channel data[6], line = model [5].

Where $\sigma = 1.3$(J&L, CH). Figure 2.8 compares the model with the terms representing the left-hand side of Eq.2.47. As in the case for the k-equation, the comparison is not good in the vicinity of the wall; in fact, it can be shown from Taylor series expansion that this model does not have the proper asymptotic behavior as $y^+ \to 0$. The final model for ϵ is:

$$\frac{\partial \epsilon}{\partial t} + \frac{\partial}{\partial x_i}(\epsilon \bar{u}_i) = \frac{\partial}{\partial x_i}\left[\left(\frac{\nu_t}{\sigma_\epsilon}\right)\frac{\partial \epsilon}{\partial x_i}\right] + C_{\epsilon_1}\frac{P\epsilon}{k} - C_{\epsilon_2}\frac{\epsilon^2}{k} \tag{2.48}$$

Where the value of constants is chosen by comparing the results of the model to experimental data, DNS simulations and theoretical results for simple cases. The standard $k - \epsilon$ model implemented in most of CFD applications, at present days, uses the following values for the constants: $C_{\epsilon_1 = 1.44}$, $C_{\epsilon_2 = 1.92}$,$C_{\sigma_\epsilon = 1.3}$.

2.5.2 k-ω Model

In the k-ω model, the first transported variable is turbulent kinetic energy, k, while the second transported variable is the specific dissipation ω. The first to use a two equation model based on the dissipation per unit turbulence kinetic energy ω was Kolmogorov in 1942. The model was improved by Saffman in 1970 and then by Wilcox in 1974.
The differences between $k - \epsilon$ and $k - \omega$ will be discussed without going into the theoretical details of $k - \omega$ which are not far from those derived form $k - \epsilon$ (The reader is referenced to [16]).
The equation for k is the same as described in section 2.5.1, while the equation for ω becomes:

$$\frac{\partial \omega}{\partial t} + \frac{\partial}{\partial x_i}(\omega \bar{u}_i) = \frac{\partial}{\partial x_i}\left[\left(\frac{\nu_t}{\sigma_\omega}\right)\frac{\partial \omega}{\partial x_i}\right] + C_{\omega_1}\frac{P\omega}{k} - C_{\omega_2}\omega^2 \tag{2.49}$$

The question is: how does the k-ω model based on this equation differ from the k-ϵ model?
One way to answer this question is to derive the ω equation implied by the k-ϵ. Taking $\sigma_k =$

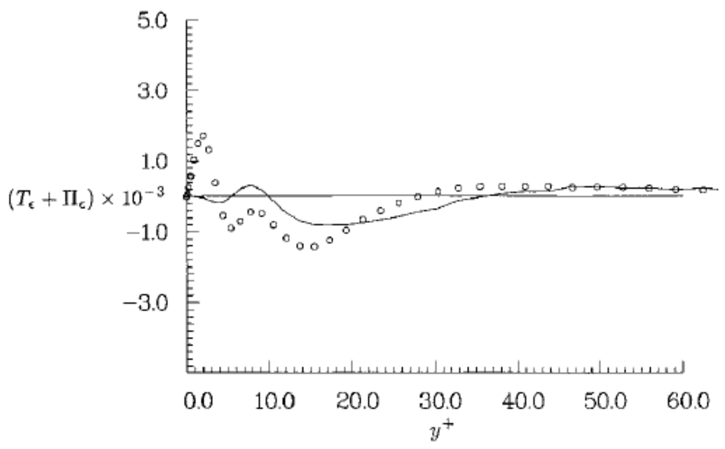


Figure 2.8: Turbulent transport rate of ϵ: Circlets = terms computed from the channel data[6], line = model [5].

$\sigma_\epsilon = \sigma_\omega$ for simplicity, the result is

$$\frac{\partial \omega}{\partial t} + \frac{\partial}{\partial x_i}(\omega \bar{u}_i) = \frac{\partial}{\partial x_i}\left[\left(\nu + \frac{\nu_t}{\sigma_\omega}\right)\frac{\partial \omega}{\partial x_i}\right] + (C_{\epsilon_1} - 1)\frac{P\omega}{k} - (C_{\epsilon_2} - 1)\omega^2 + \frac{2\nu_T}{\sigma_{\omega k}}\frac{\partial \omega}{\partial x_i}\frac{\partial k}{\partial x_i} \tag{2.50}$$

Evidently, for homogeneous turbulence, the choices $C_{\omega_1} = C_{\epsilon_1} - 1$ and $C_{\omega_2} = C_{\epsilon_2} - 1$ make the models identical. However, for inhomogeneous flows, the k-ϵ, written as k-ω model, contains an additional term, that is the final term of equation 2.50. An improved version of the k-ω model, able to model this term properly, is the k-ω-SST model.

2.6 Wall functions

Near wall effects require additions or modifications to the basic turbulent models, because of the physics problem which requires to solve steep profiles of velocity and ϵ / ω: the idea of the "wall function" approach is to apply boundary conditions some distance away from the wall, so that turbulent equations are not solved close to the wall.
By combining the equation for k, ϵ and ω, we the following values of wall functions are obtained [17]:

$$k = \frac{u_\tau^2}{\sqrt{C_\mu}}$$

$$\omega = \frac{k^{1/2}}{C_\mu^{1/4} ky}$$

$$\epsilon = C_\mu^{3/4}\frac{k^{3/2}}{ky}$$

where the location at which these functions are calculated is $y = y^+ \cong 50$

2.7 Strengths and weaknesses of turbulence models

The considerations made in this sections are taken from [18] [19] and refer to the range of applicability of the turbulence models described in the previous sections.

- *Spalart-Allmaras* is the simplest model among the ones used in this work, since it is a 1-equation model. For this reason, this model is expected to work worst than the others for complicated flows: this turbulence model is intended only for aerodynamic applications and it has clear limitations as a general model: it is incapable, for example, of accounting for the decay of ν_T in isotropic turbulence.
  Anyway this model is able to predict skin friction for attached boundary layers that is as close to measurement as algebraic models. The model's predictions are far superior to those of algebraic models for separated flows and the differential equation presents no serious numerical difficulties.

- The k-ϵ model is the widely used two-equation model and it can be applied to any turbulent flow. It performs reasonably well for two-dimensional thin shear-flows in which the streamline curvature and the pressure gradient are small.
  For boundary layers with strong pressure gradients and for flows with adverse pressure gradients it performs poorly; it is also inaccurate for separated flows.

- The k-ω model performs satisfactorily for boundary layers with strong pressure gradients and its performance is superior for many flows. However, like all of the *RANS* models based on the Boussinesq-hypothesis, it may not perform properly for 3D flows.

Chapter 3

Flow Over a Hump Wall: Overview

This section summarizes the state of the art for what concerns the study of the model problem studied in this work: that is the flow over a wall-mounted hump.
The understanding and prediction of separated flows have posed a significant challenge for many decades, so that many rstudies have been involved in this experimental setup, which has proven to be a useful tool to test the reliability of numerical simulations and turbulence models for separated flows.
A collection of the most important results is reported concerning both numerical and experimental works.
The experimental data are also available on the *NASA Lanley Research Center* website [20].
Experiments and simulations were conducted on both the baseline and the controlled cases (steady suction and zero-flux oscillatory blowing were used to control the flow);the attention is focused on the baseline case, which is the object of study of this work.
For further information about the “hump flow”, it is recommended to read the papers and the bibliographic sources mentioned in this section.

3.1 Test case

Before beginning the computational study, low speed separation over the wall-mounted hump was studied experimentally by the Langley group[3] in order to generate a data sets for the workshop aimed at validating CFD turbulence models.
The test case involved a wall-mounted hump model, represented in figures 3.1, 3.2.

The model was constructed from aluminum over a splitter-plate and it was mounted between two endplates with aluminum frames and glass interiors. The experiments were performed in the NASA Langley 20”x 28” shear flow tunnel. The flow was nominally two-dimensional with side-wall effects expected near the endplates. The characteristic reference “chord” length of the model was defined as the length of the hump on the wall i.e. c=420mm and its maximum thickness was $h = 53.7mm$.
The boundary layer was tripped at the splitter-plate leading edge, resulting in a fully developed turbulent boundary layer at 2.14 chord lengths upstream of the model leading-edge[1].

[1]For further details, it is recommended to consult the paper [3]

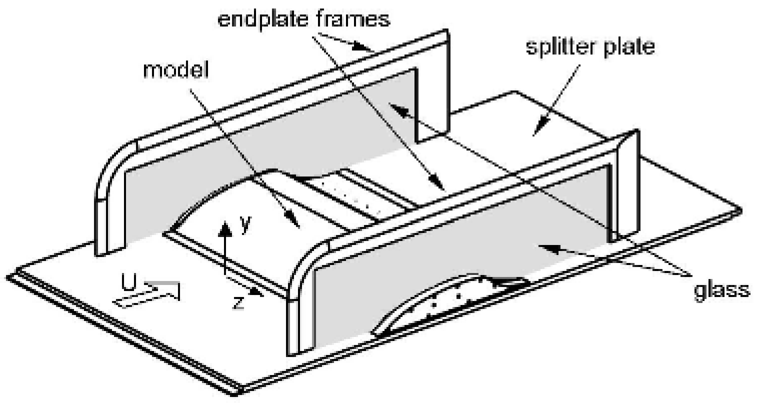


Figure 3.1: Isometric view showing the model mounted on the splitter plate with end plates in place [3].

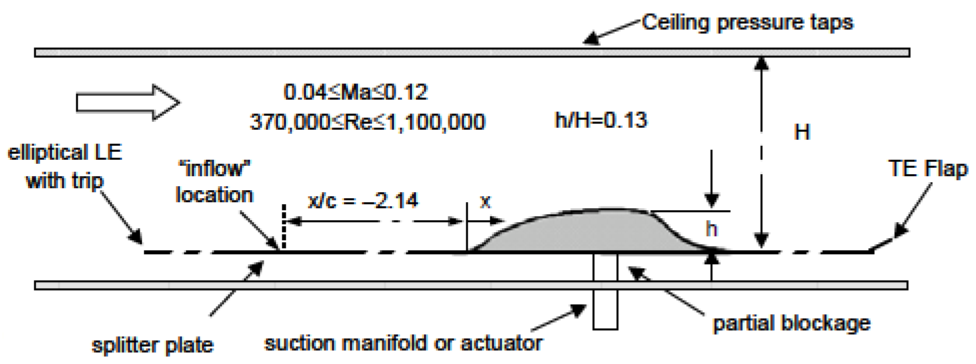


Figure 3.2: 2-D Sketch of the experimental setup [3].

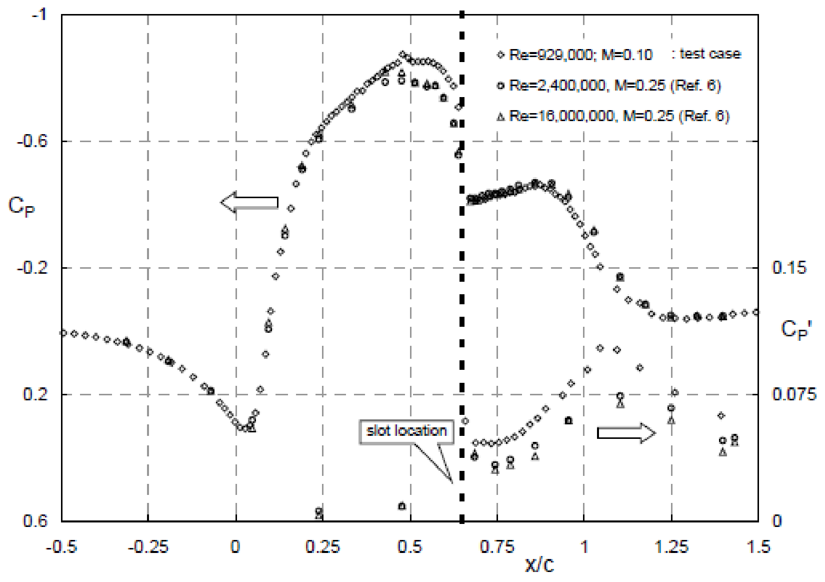


Figure 3.3: A comparison of the test case surface pressure with that at high Reynolds number [3]

For the uncontrolled case[2] a test case was chosen at $Re = 929000$ and $M = 0.1$, even if this does not play an essential role, since the model appears to be virtual insensitive to both *Reynolds number* and inflow conditions, as we can see from the pressure coefficient distribution for different *Reynolds* numbers, Fig. 3.3.
The results of pressure measurements and of 2-D and 3-D PIV flow field measurements are shown in figure 3.5.

Figures 3.3 and 3.4 show that the flow, approaching the model leading-edge decelerates but does not separate. Immediately downstream of the leading edge, the boundary layer is subjected to a strong favorable pressure gradient: low C_f in this region, followed by large C_f change between x/c=0.07 and 0.11 may indicate relaminarization close to the leading edge, followed immediately downstream by re-transition. At $x/c \approx 0.6$ in the region of strong convex curvature the pressure increases abruptly and separation occurs. The flow remains separated over the relatively short concave ramp in the aft part of the body and reattaches downstream of the trailing edge, at $x/c \approx 1.1$.
Downstream of reattachment, the boundary layer recovers under a near zero pressure gradient[3].

3.2 Computational Fluid Dynamics Validation Workshop

Numerical simulations of the hump model were object of study during the CFD Validation Workshop, held in Williamsburg, Virginia in March 2004.
The goal of this workshop, fully describe in [1] was to bring together an international group of computational fluid dynamics practitioners to assess the current capabilities of different classes

[2]The uncontrolled case is the only one we are studying in this work.

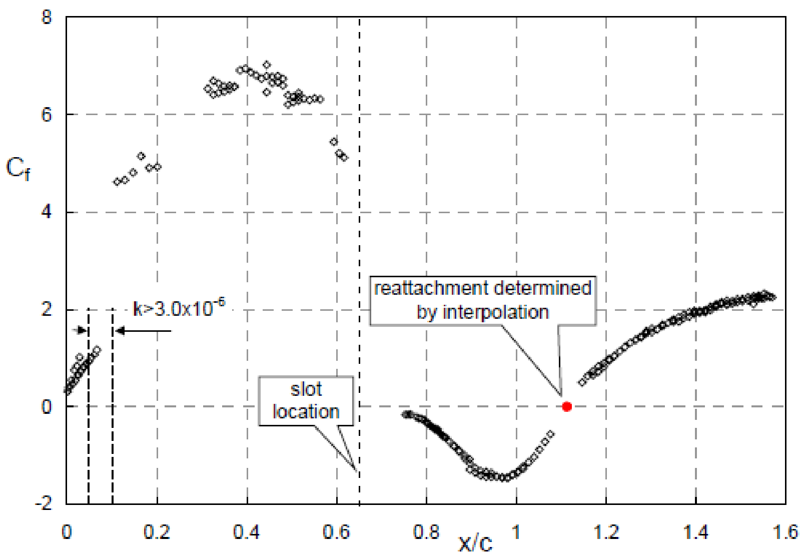


Figure 3.4: Wall shear stress coefficient data over the model. [3]

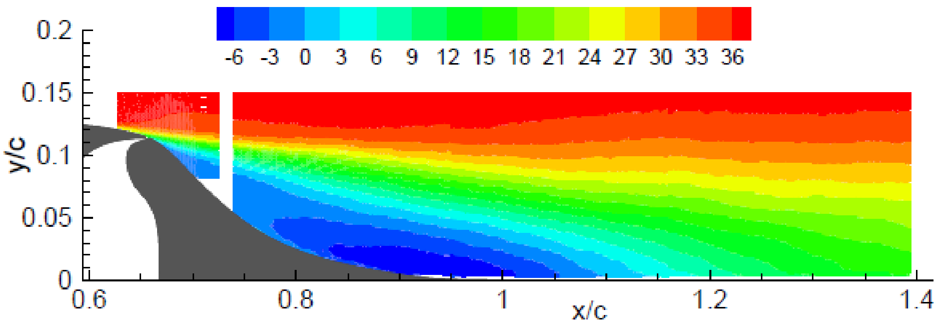


Figure 3.5: The superposition of four blocks of 2-D PIV U-component data, from upstream of separation to downstream of the reattachement region for the baseline case (in m/s). [3]

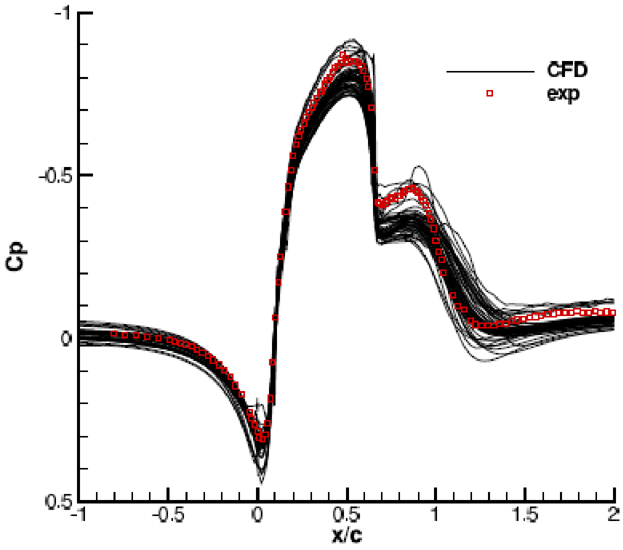


Figure 3.6: Simulated pressure coefficients compared to experimental data. [1]

of different flow solution methodologies to predict flow fields induced by synthetic jets and separation control geometries.
There were 75 attendees at this workshop and 7 countries were represented, including the United States, France, Italy, Germany, Japan, United Kingdom and Switzerland.
To encourage broad participation and to determine the general state-of-the-art, the decision was made *not* to dictate particular boundary conditions, grids or methods of solution.
Specifications concerning the geometry and the inlet conditions were given, in order to respect the fluid dynamics similitude. Like in the experimental study described in the previous chapter, a turbulent flow at $M = 0.1$ passed over a hump of chord 420mm.
The most important results, for the test case, are shown in figures 3.6, 3.7, 3.8

As a whole, most CFD results missed the pressure levels over the hump between $0.2 < x/c < 0.6$ and also predicted higher pressures than experimental results in the separated region upstream of $x/c = 1$. An explanation of this trend could be in the fact that blockage effects caused by the side plates in the experiment caused a decrease of pressure, if compared to 2D simulations.
It can also be observed that the separation location was predicted reasonably well by most of the CFD methods, and the reattachment location was predicted significantly downstream of the experimental location of $x/c = 1.11$.
The only exception was the RANS simulation run by *CIRA & CTR*[3], which predicted separation later and reattachment earlier than other results with SST model, and the simulation also run by *CIRA & CTR*, which predicted reattachment further downstream than other results using a k-ϵ model.
A possible reason for reattachment being predicted too late is that most of the current models

[3]Authors: Marongiu, Iaccarino, Catalano, Amato

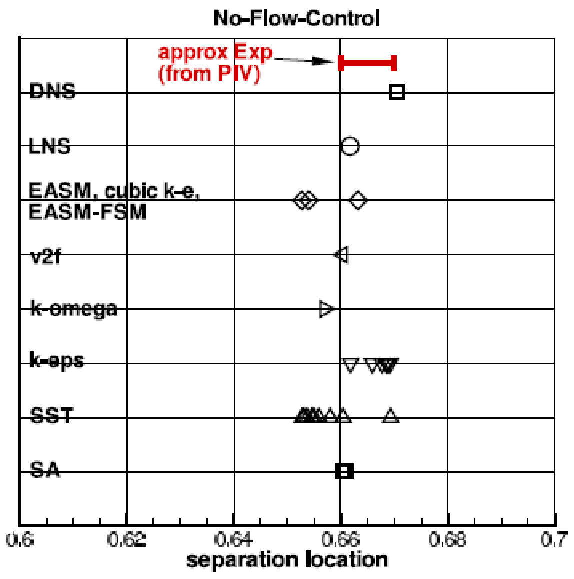


Figure 3.7: Simulated separation location compared to experimental data. [1]

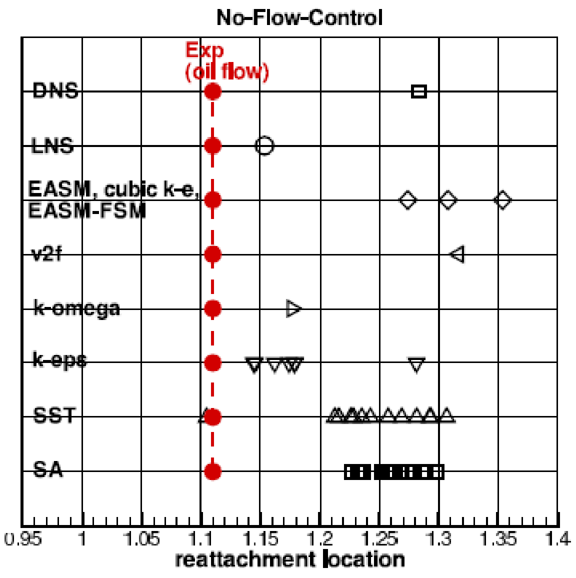


Figure 3.8: Simulated reattachment location compared to experimental data [1]

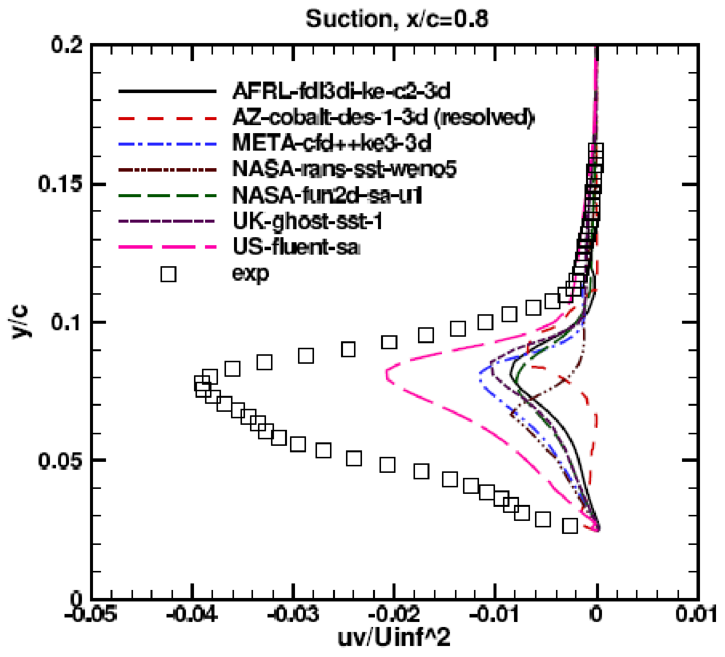


Figure 3.9: Turbulent shear stresses inside separation bubble for the controlled case, steady suction. ($x/c = 0.8$) [1]

and methods under-predict the magnitude of the turbulent shear stress in the separated region. Fig. 3.9 shows turbulent shear stress for the suction condition[4] at the location $x/c = 0.8$. It can be remaked that the magnitude of the turbulent stress was seriously underpredicted. Rumsey concluded his paper by saying that after the workshop it was discovered that the side plates used in the tunnel caused blockage that, if not modeled, resulted in relatively minor (but noticeable) overprediciton of the pressures over most of the hump. Flow structures near the back of the plates could constrict the flow even further, so that, in spite of this flow appearing to be relatively simple, computing the wall pressures accurately can require full 3-D modeling or else some sort of blockage corrections. By the way CFD results were deficient in another important regard: they consistently predicted the reattachment location to be significantly further downstream than the location documented in the experiment. This same behavior occurred regardless of turbulence model or method: even a DNS computation predicted a relatively long separation bubble.

3.3 k-ω Model

In 2005, Balakumar computed turbulent separated flow over a two-dimensional hump, by implementing a higher order method for the RANS equations with $k - \omega - SST$ turbulence model [7]. In order to have a fully developed turbulent flow upstream of the hump, Balakumar extended the computational domain from $x/c = -10.0$ to 4.0 in the streamwise direction (Fig. 3.10). The length of the splitter plate ($-6 < x/c < 0$) was selected to match the measured velocity profiles at $x/c = -2.1$. The free-stream Mach number was 0.1 and the chord Reynolds number was 936000

[4]This is not strictly our case (our case is the not controlled one), but the physical phenomena are the same.

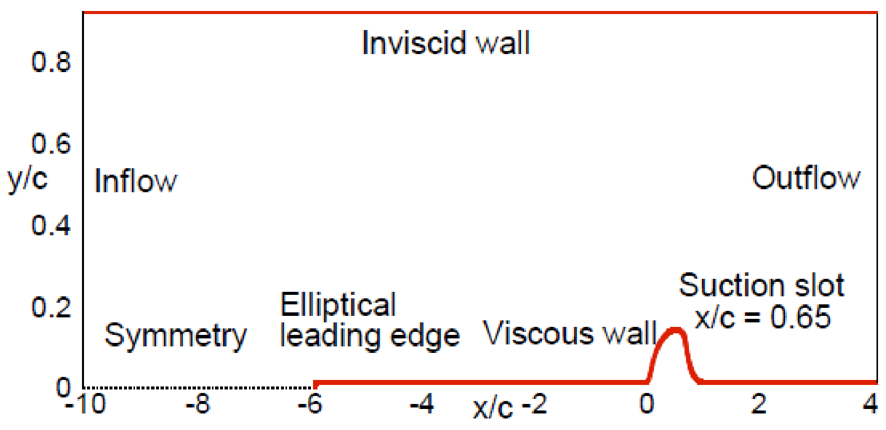


Figure 3.10: Schematic diagram of Balakumar's hump model [7]

(very similar to the experimental setup described in section 3.1 . To keep as close as possible to the experimental data the effects of the grid upstream of the splitter plate are reproduced, as well (Fig. 3.11).
The governing equations, the flow equations and the turbulent equations, are solved using the 5th order accurate weighted essentially non-oscillatory (WENO) scheme for space discretization and using explicit third order total-variation-diminishing (TVD)[5]. Figure 3.12 shows the computed and the measured C_p distribution for the baseline and the oscillatory cases. Qualitatively, the same phenomena described in section 3.2 were observed: in the experiment, the pressure decreased immediately downstream of the slot, then increased up to the reattachment point; in the computations, it remained flat near the slot and then increased. The computed C_p is about 10% smaller than the experimental value because of the blockage of the end plates and the computed separation region is longer than the experimental one.
The steady suction control reduces the length of the bubble, which, in the computed case, is still over-predicted if compared to the experimental data.

3.4 Large Eddy Simulation

Since none of the RANS techniques gave satisfactory prediction of the important flow features, in 2005 Large Eddy Simulation was employed to solve the problem of separation control over the wall-mounted hump [8].
In their study, the authors employed a dynamic subgrid-scale model[6] and non-dissipative numerics to predict the turbulent flow separation and its control by synthetic jets in the same hump-model configuration as described in the previous works.
LES produced superior results compared to those obtained from RANS simulations for both the controlled and uncontrolled cases.
Figure 3.15 shows comparison of pressure coefficients for different numerical methods. It can be remarked that LES results are the closest ones to experimental data. In what concerning the

[5]The WENO and TCD methods and the formulas are explained in Ref.13-14 of the paper [7]. In this paper all information about inlet and boundary conditions are listed, as well.
[6]Dynamic subgrid-scale model for LES simulations were introduced by Germano for the first time, in 1991.

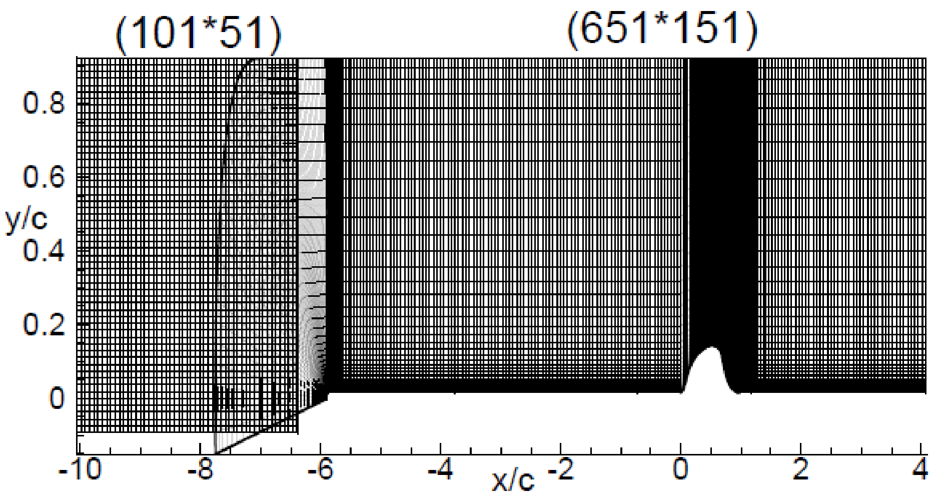


Figure 3.11: Sketch of the mesh used by Balakumar in his simulations. [7]

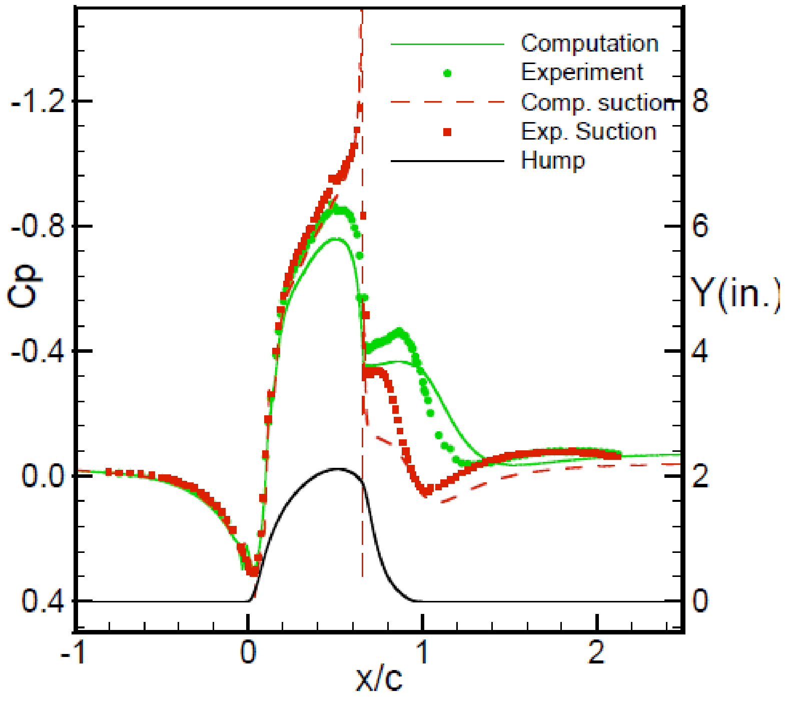


Figure 3.12: C_p distribution for the baseline and the oscillatory control cases[7].

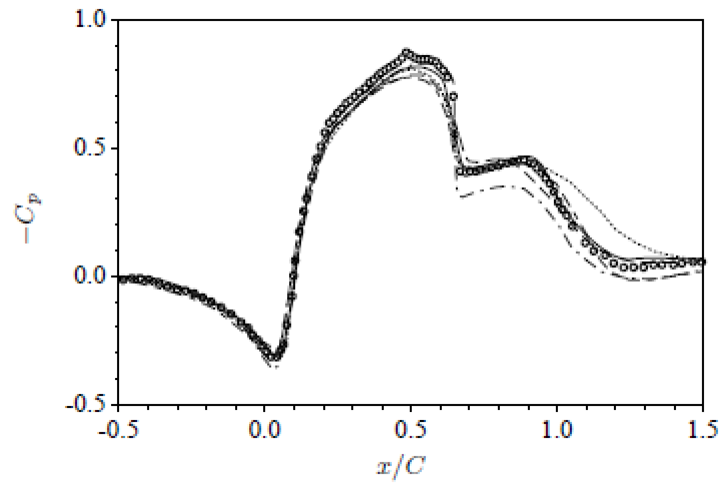


Figure 3.13: C_p distribution for the baseline.[8]. Continuous line: LES, Circlets: experimental data, dashed line: RANS, dot line: URANS

prediction of separation and reattachment of the bubble, figure 3.14 shows the trend of Cf, for the controlled and uncontrolled cases, compared to the experimental results. For the baseline, the LES curve perfectly matches the experimental data downstream of the hump, so that the position and the length of the bubble are exactly calculated.
LES also shows favorable agreement with experimental data except for the front convex region of the hump, even if, in this case, LES over-predicts the skin friction coefficient.
The Reynolds stress profiles from numerical simulations and experiments are compared in Fig. 3.16.
Once again, it can be observed that LES is consistently better in predicting the Reynolds shear stress. The goal of this overview was just to expose the state-of-the-art for what concerning the hump flow model problem. Even if this chapter deals with Large Eddy Simulation, from now on, this work will only focus on RANS equations, since for these equations, the state-of-the-art is still incomplete and they require more efforts in order to be modeled properly and to be consistent with experiments.

3.5 2008 Workshop

In 2008, Rumsey published another paper to update the state-of-the-art on the hump-flow, from 2004 to 2008 [2].
This is the most recent survey about the model problem of the flow over a hump.
Rumsey's conclusions about *RANS* equations are:
"No major progress has been made since the time of the first worksop in terms of RANS-URANS. Results have for the most part been very consistent in terms of predicting too little eddy viscosity in the separated region and too long a bubble. It was noted that models can sometimes predict a particular feature like reattachment location correctly for the wrong reasons. For example, because k-ϵ turbulence models tend to predict smooth body separation caused by adverse pressure-gradient too late, they also tend to predict earlier reattachment. This reattachment location may appear

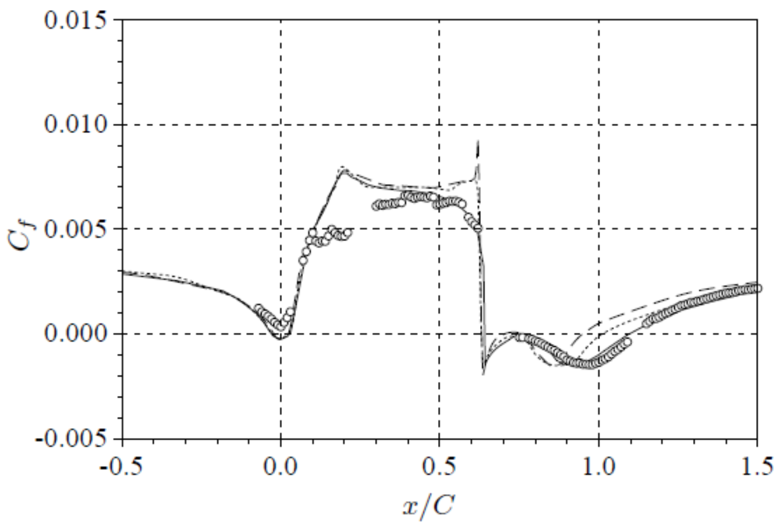


Figure 3.14: C_f distribution for the baseline.[8]. Continuous: baseline, Circlets: experimental data, dashed line: steady suction, dot line: oscillatory jet.

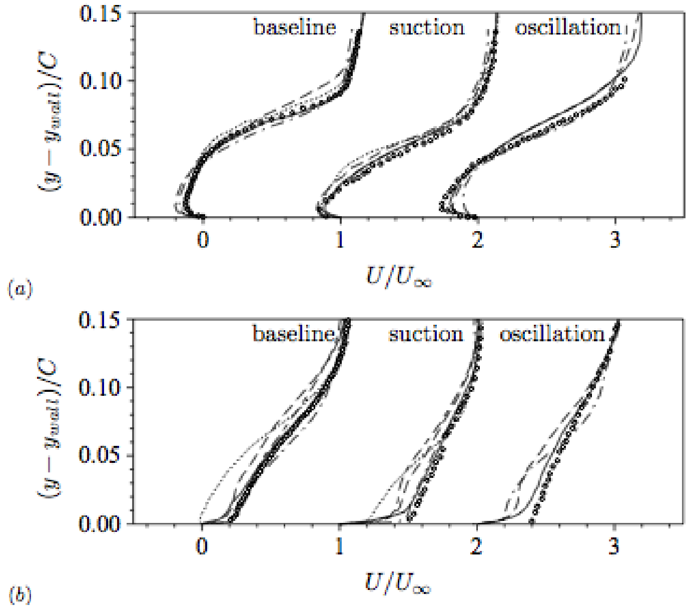


Figure 3.15: Velocity profile at different locations[8]. Continuous: baseline, Circlets: experimental data, dashed line: steady suction, dot line: oscillatory jet.

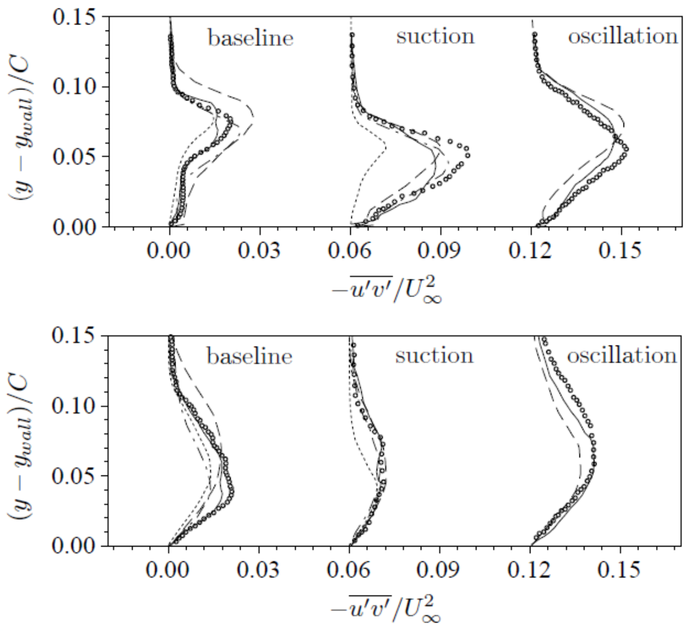


Figure 3.16: Reynolds shear stress profile.[8]. Continuous line: present LES, Circlets: experimental data, dashed line: ILES, dot line: RANS.

to agree better with experiment in the hump case, for example, but it is not due better modeling of the turbulent mixing in the separated region. " *He et al.* [9] obtained very reasonable results for the baseline, by using a second order upwind and a SIMPLE algorithm. The software used to run their simulations was the commercial one, *Fluent.*
Their results, however, were questioned by the fact that Bettini and Cravero [11] obtained different results, by using the same commercial software: the effects that the refinement of the mesh, the position of the *Inlet* have on the solution may have been neglected: this work will focus on these important aspects.
Figures 3.17, 3.18, 3.19 shows the behavior of pressure coefficients, velocity profiles and skin-friction in the simulations run by *He et al.*

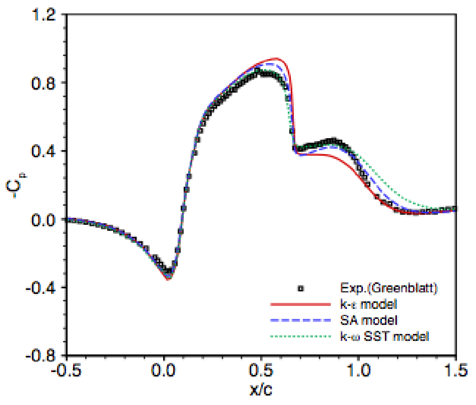


Figure 3.17: *He et Al.* Pressure coefficient for the baseline [9].

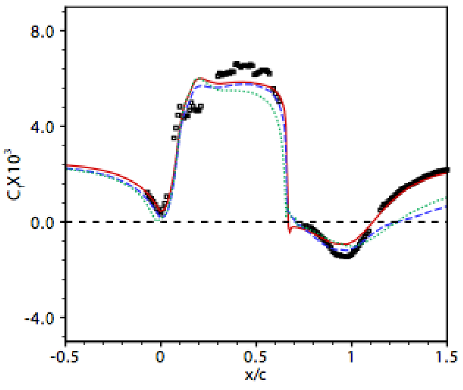


Figure 3.18: *He et Al.* Skin friction for the baseline [9].

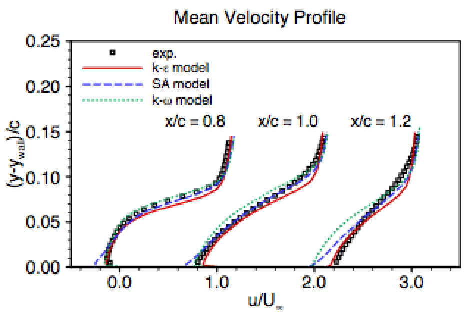


Figure 3.19: *He et Al.* Velocity profile for the baseline [9].

Chapter 4

OpenFOAM

4.1 An Introduction to *OpenFOAM*

OpenFOAM, **Open Source Field Operation and Manipulation** is a free, open source CFD software package written in C++ and produced by *OpenCFD Ltd.* The code is released as free and open source software under the GNU General Public License and it is maintained by the *OpenFOAM Foundation*, which is sponsored by *Silicon Graphics International.*
By being open, OpenFOAM offers users complete freedom to customize and extend its existing functionality. At present *OpenFOAM* includes over 80 solver applications that simulate specific problems in engineering mechanics and over 170 utility applications that perform pre- and post-process tasks (meshing, data visualization ...).
The original development of *OpenFOAM* started in 1980 at *Imperial College*, London to develop a more powerful and flexible general simulation platform than *FORTRAN*. This led to the choice of *C++* as programming language, due to its highest modularity and object oriented features. *OpenFOAM* was one of the first major scientific packages in *C++* and it has also been the first major general-purpose CFD package to use polyhedral cells.
One distinguishing feature of *OpenFOAM* is its syntax for tensors operations and partial differential equations that closely resembles the equations being solved. For example, the equation

$$\frac{\partial \rho \mathbf{U}}{\partial t} + \nabla \cdot (\phi \mathbf{U}) = -\nabla p + \nabla \cdot \mu \nabla \mathbf{U} \tag{4.1}$$

is represented by the code:

Example of equation in OpenFOAM

```
solve
 (
    fvm::ddt(rho,U)
 +  fvm::div(phi,U)
    ==
 -  fvc::grad(p)
 +  fvm::laplacian(mu,U)
 );
```

OpenFOAM solver capabilities include:

- Incompressible Flows;

- Multiphase Flows;
- Combustion;
- Conjugate heat transfer;
- Particle methods (DEM, DSMC, MD);
- Other ...

While its libraries include:

- Turbulence Models;
- Transport/rheology models;
- Lagrangian particle tracking;
- Reacting kinetic / chemistry.

The solver which will be used in this work is that for steady, incompressible flows: **simpleFOAM**, by implementing the turbulence models available for RANS equations, in particular: $SpalartAllmaras$, $k-\epsilon$, $k-\omega$, $k-\omega-SST$.

4.2 OpenFOAM structure

The *OpenFOAM* code is structured as follows[1]:

- applications: this folder contains the source files of all executables:
  - solvers;
  - utilities;
  - bin;
  - test;
- bin: basic executable scripts;
- doc: pdf and Doxygen documentation;
- lib: compiled libraries;
- src: source library files;
- test: library test source files;
- tutorials: tutorial cases;
- wmake: compiler settings.

[1]Type "*foam*" and then "*ls*" on your terminal window".

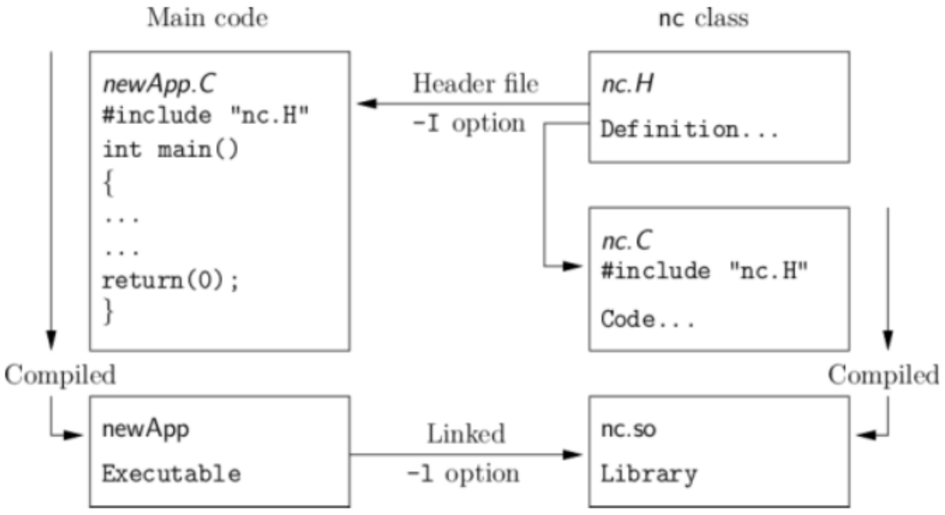


Figure 4.1: Header files, source files, compilation and linking. [10]

The code for the incompressible flows solver, **simpleFOAM**, is in the **solver** folder, while the turbulence models employed to solve RANS equations are in the **turbulenceModels** folder, which is inside **src**.
Programming languages that are object-oriented, such as *C++*, provide the mechanism - *classes* - to declare types and associated operations that are part of the verbal and mathematical languages used in science and engineering. The velocity U, for example, would be an instance, that is an *object*, of the *vectorField* class; hence the term object-oriented.
New classes can be derived or inherit properties from other classes; for instance the *vectorField* can be derived from a *vector* class and a *Field* class.
To understand the compilation process in *OpenFOAM* we should explain certain aspects of C++ and its file structure, shown schematically in figure 4.1.

A class is defined through a set of instructions such as object construction, data storage and class member functions. The file containing the class definition takes a .C extension, e.g. a class *nc* would be written *nc.C*. This file can be compiled indipendently of other code into a binary executable library file known as a shared object library with the *.so* file extension, i.e. *nc.so*. When compiling a piece of code, say *newApp.C* that uses the *nc* class, *nc.C* need not be recompiled, rather *newApp.C* calls *nc.so* at runtime. This is known as *dynamic linking*.
As a means of checking errors, the piece of code being compiled must know that the classes it uses and the operations they perform actually exist. Therefore each class requires a class *declaration*, contained in a header file with a *.H* extension, e.g. *nc.H*, that includes the names of the class and its functions. This file is included at the beginning of any piece of code using the class, including the class declaration code itself.

4.2.1 Compiling with *wmake*

OpenFOAM applications are organized using a standard convention that the source code of each application is placed in a directory whose name is that of the application. The top level source file takes the application name with the *.C* extension. For example, the source code for an application called *newApp* would reside in a directory *newApp* and the top level file would be *newApp.C*, as shown in figure 4.2.

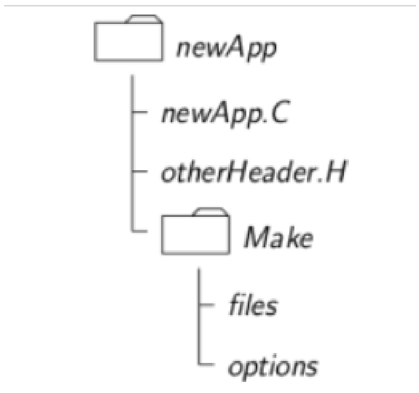


Figure 4.2: Directory structure for an application. [10]

When the user compiles, by using the *wmake* command [2] the compiler searches for the included header files. The full directory paths of the header files is located in the *Make/options* file by using the following syntax:

```
EXE_INC = \
-I<directoryPath1> \
-I<directoryPath2> \
...                \
-I<directoryPathN> \
```

4.3 The *simpleFoam* application: Algorithm

SimpleFOAM is a steady-state solver for incompressible, turbulent flow. We recall that the Navier-Stokes equations for a single-phase flow with a constant density and viscosity are the following:

$$\nabla \cdot \mathbf{u} = 0 \tag{4.2}$$

$$\nabla \cdot (\mathbf{uu}) - \nabla \cdot (\nu \nabla \mathbf{u}) = -\frac{1}{\rho}\nabla p \tag{4.3}$$

The solution of these equations is not straightforward because of the non-linear term $\nabla \cdot (\mathbf{uu})$ and because an explicit equation for the pressure is not available. The approach used in *OpenFOAM* is to derive an equation for the pressure by taking the divergence of the momentum equation and substituting it in the continuity equation. Temporal discretization is performed using some implicit temporal scheme, such as:[21]

$$\int_t^{t+\Delta t} f(t, \mathbf{U}(\mathbf{x}, t))dt = (1 - C)\Delta t f(t, \mathbf{U}(\mathbf{x}, t^0)) + C\Delta t f(t, \mathbf{U}(\mathbf{x}, t^n)) \tag{4.4}$$

[2] *OpenFOAM* is supplied with the *wmake* compilation script that is based on *make* but is considerably more versatile and easier to use...

where different values of C can recover temporal schemes defined in Juretic's thesis[21]. When the momentum equation is approximated by using the equation 4.4, the following relationship is derived[21, 22]:

$$a_p \mathbf{u}_p = \mathbf{H(u)} - \nabla p \tag{4.5}$$

where the subscript P refers to the center of cell P. Velocity can be rewritten as follows:

$$\mathbf{u}_p = \frac{\mathbf{H(u)}}{a_P} - \frac{\nabla p}{a_P} \tag{4.6}$$

The term $\mathbf{H(u)}$ includes all terms apart from the pressure gradient at the new time step and the diagonal term $a_P^n \mathbf{U}_P^n$, where the subscript n represents the new time level. The continuity equation is discretized as:

$$\nabla \cdot \mathbf{u} = \sum_f \mathbf{S} \mathbf{u}_f = 0 \tag{4.7}$$

where $\mathbf{S}$ is outward-pointing face area vector and $\mathbf{u}_f$ is the velocity on the face, which is obtained by interpolating the semi-discretized form of the momentum equation (4.6) as follows:

$$\mathbf{u}_f = \left(\frac{\mathbf{H(u)}}{a_P}\right)_f - \left(\frac{\nabla p}{a_P}\right)_f \tag{4.8}$$

By substituting this equation into the discretized continuity equation above, the pressure equation is derived:

$$\nabla \cdot \left(\frac{1}{a_P} \nabla p\right) = \nabla \cdot \left(\frac{\mathbf{H(u)}}{a_P}\right) \tag{4.9}$$

The mass flux through a cell face can be obtained by using equation 4.8 as follows:

$$F = \mathbf{S} \cdot \left[\left(\frac{\mathbf{H(u)}}{a_p}\right)_f - \left(\frac{1}{a_P}\right)(\nabla p)_f\right] \tag{4.10}$$

4.4 The *simpleFoam* application: Implementation

The SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) allows to couple the Navier-Stokes equations with an iterateve procedure, which can be summed up as follows:

1. Set the boundary conditions;
2. Solve the discretized momentum equation to compute the intermediate velocity field;
3. Compute the mass fluxes at the cells faces;
4. Solve the pressure equation and apply under-relaxation;
5. Correct the mass fluxes at the cell faces;
6. Correct the velocities on the basis of the new pressure field;
7. Update the boundary conditions;
8. Repeat till convergence.

The complete implementation of the algorithm can be seen in the source code of the *simpleFoam* solver provided with **OpenFOAM** (file *simpleFoam.C*)[3]. It is based on the following steps:

[3]Directory:*openfoam211/applications/solvers/incompressible/simpleFoam*

- **Writing an equation for velocity**, (file *UEqn.H*), where a RANS equation is defined, including all the terms of (ref rans eq), except pressure.The operators **turbulence** and **sources** are defined in the file *createFields.H* and represent the non-linear terms. Reynolds stresses are modeled using the eddy viscosity model, $\overline{u'v'} = -2\nu_t S_{ij}$. Once the equation is relaxed, the pressure term is added to the equation (last line of the following script):

The UEqn.H file in the *simpleFoam* directory

```
tmp<fvVectorMatrix> UEqn
(
    fvm::div(phi, U)
  + turbulence->divDevReff(U)
  ==
    sources(U)
);

UEqn().relax();

sources.constrain(UEqn());

solve(UEqn() == -fvc::grad(p));
```

The first term in the implementation of the momentum equation (*div(phi,U)*) is translated into:

$$\frac{\partial(\bar{u}_i\bar{u}_j)}{\partial x_j} = \bar{u}_j\frac{\partial \bar{u}_i}{\partial x_j} + \bar{u}_i\frac{\partial \bar{u}_j}{\partial x_j} = \bar{u}_j\frac{\partial \bar{u}_i}{\partial x_j} \tag{4.11}$$

where the last step is given by incompressibility of the mean flow fields[4].

The function *divdevReff* in the *kEpsilon.C* file

```
tmp<fvVectorMatrix> kEpsilon::divDevReff(volVectorField& U) const
{
    return
    (
      - fvm::laplacian(nuEff(), U)
      - fvc::div(nuEff()*dev(T(fvc::grad(U))))
    );
}
```

The term *divDevReff(U)* is defined in the same file where turbulence models for *RANS* equations are implemented and it has two components:

$$-\frac{\partial}{\partial x_j}\left(\nu_{eff}\frac{\partial \bar{u}_i}{\partial x_j}\right) \tag{4.12}$$

and

$$-\frac{\partial}{\partial x_j}\left[\nu_{eff}\left(\frac{\partial \bar{u}_i}{\partial x_j} - \frac{1}{3}\frac{\partial \bar{u}_k}{\partial x_k}\delta_{ij}\right)\right] \tag{4.13}$$

[4]According to continuity equation $\frac{\partial \bar{u}_j}{\partial x_j} = 0$

which, thanks to continuity equation, becomes:

$$-\frac{\partial}{\partial x_j}\nu_{eff}\left(\frac{\partial \bar{u}_j}{\partial x_i}\right) \tag{4.14}$$

so that the total equation for *divDevReff* is:

$$-\frac{\partial}{\partial x_j}\left[\nu_{eff}\left(\frac{\partial \bar{u}_i}{\partial x_j}+\frac{\partial \bar{u}_j}{\partial x_i}\right)\right] \tag{4.15}$$

The last part of the implementation is the pressure gradient. The final momentum equation is:

$$\bar{u}_j\frac{\partial \bar{u}_i}{x_j}-\frac{\partial}{\partial x_j}\left[\nu_{eff}\left(\frac{\partial \bar{u}_i}{\partial x_j}+\frac{\partial \bar{u}_j}{\partial x_i}\right)\right]=-\frac{\partial p}{\partial x_i} \tag{4.16}$$

which is the same equation as 2.16. The connection to previous time step is done through the command $UEqn().relax()$.

- **Prediction of velocity**; in this part of code the coefficients of the matrix A and H are extracted, basing on the equation for velocity defined above, and a value of velocity defined as $\mathbf{u} = \frac{H}{a_P}$ is predicted. The flux $\sum_f \mathbf{S}(\frac{H}{a_P})_f$ is calculated after interpolation through the surfaces of the mesh (file *pEqn.H*).

Prediction of velocity, *simpleFoam.C* file

```
p.boundaryField().updateCoeffs();

    volScalarField rAU(1.0/UEqn().A());
    U = rAU*UEqn().H();
    UEqn.clear();

    phi = fvc::interpolate(U, "interpolate(HbyA)") & mesh.Sf();
    adjustPhi(phi, U, p);
```

- **Non-othogonal pressure corrector loop**; pressure is calculated in according with the relationship 4.9, and then it is relaxed:

Corrector loop for pressure, *simpleFoam.C* file

```
    // Non-orthogonal pressure corrector loop
    while (simple.correctNonOrthogonal())
    {
        fvScalarMatrix pEqn
        (
            fvm::laplacian(rAU, p) == fvc::div(phi)
        );

        pEqn.setReference(pRefCell, pRefValue);

        pEqn.solve();

        if (simple.finalNonOrthogonalIter())
        {
            phi -= pEqn.flux();
        }
    }

    #include "continuityErrs.H"

    // Explicitly relax pressure for momentum corrector
    p.relax();
```

- **Momentum corrector**, where the corrected value of velocity is calculated by using the relationship 4.6

Correction of velocity, *simpleFoam.C* file

```
// Momentum corrector
    U -= rAU*fvc::grad(p);
    U.correctBoundaryConditions();
    sources.correct(U);
```

- **Resolution of equations for turbulence**: In the *simpleFoam.C* file, the command *turbulence* $-$ $>$ *correct* is used to inherit the value of turbulent viscosity from the classes in which the equations for turbulence are implemented (*k*-ϵ, *k*- ω, *Spalart Allmaras* ...)

```
turbulence->correct();
runTime.write();
```

The equations for k, ϵ and ω are shown as thy appear in the *kEpsilon.C* and *kOmega.C* files[5]. The equations for turbulent kinetic energy and dissipation energy are are implemented as follows:

Turbulent kinetic energy equation, *kEpsilon.C* file

```
    // Turbulent kinetic energy equation
    tmp<fvScalarMatrix> kEqn
    (
        fvm::ddt(k_)
      + fvm::div(phi_, k_)
      - fvm::Sp(fvc::div(phi_), k_)
      - fvm::laplacian(DkEff(), k_)
     ==
        G
      - fvm::Sp(epsilon_/k_, k_)
    );

    kEqn().relax();
    solve(kEqn);
    bound(k_, kMin_);
```

[5]The whole code is available in open foam, at the directory: *openfoam211/src/turbulenceModels/incompressible/RAS*

The first term in the implementation of the k-equation is the time derivative for k: this term will not be treated, since the case studied in this work is steady.
The second term is the divergence of the velocity times k:

$$\frac{\partial(u_j k)}{x_j} = k\frac{\partial \bar{u}_j}{\partial x_j} + \bar{u}_j\frac{\partial k}{\partial x_j} \tag{4.17}$$

while the third term is the source term and it is written as $-k\frac{\partial \bar{u}_j}{\partial x_j}$. The last term on the left hand side is written $-\frac{\partial}{\partial x_j}(\nu_{eff}\frac{\partial k}{\partial x_j})$. The G term is the production term and the OpenFOAM implementation is quite difficult to understand [23].

Definition of G, file *kEpsilon.C* and of *DkEff*, file *kEpsilon.H*

```
...
volScalarField G("RASModel::G", nut_*2*magSqr(symm(fvc::grad(U_))));
...
//- Return the effective diffusivity for k
        tmp<volScalarField> DkEff() const
        {
            return tmp<volScalarField>
            (
                new volScalarField("DkEff", nut_ + nu())
            );
        }
.....
```

The symmetric part of the gradient of u_i is:

$$\frac{1}{2}\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right) \tag{4.18}$$

and the translation of the *OpenFOAM* code of G is therefore

$$\begin{aligned} G &= 2\nu_T\left[\frac{1}{2}\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right)\right]^2 \\ &= \frac{\nu_T}{2}\left(\frac{\partial \bar{u}_i}{\partial x_j}\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\frac{\partial \bar{u}_j}{\partial x_i} + 2\frac{\partial \bar{u}_i}{\partial x_j}\frac{\partial \bar{u}_j}{\partial x_i}\right) \\ &= \nu_T\frac{\partial \bar{u}_i}{\partial x_j}\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right) \end{aligned} \tag{4.19}$$

The last term of k equation represents the dissipation, ϵ, so that, by putting together all the terms of the equation, the following relation is derived:

$$\frac{\partial k}{\partial t} + k\frac{\partial \bar{u}_j}{\partial x_j} + \bar{u}_j\frac{\partial k}{\partial x_j} - k\frac{\partial \bar{u}_i}{\partial x_j} - \frac{\partial}{\partial x_j}\left(\nu_{eff}\frac{\partial k}{\partial x_j}\right) = \nu_T\frac{\partial \bar{u}_i}{\partial x_j}\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right) - \epsilon \tag{4.20}$$

which simplifies to the final expression:

$$\frac{\partial k}{\partial t} + \bar{u}_j\frac{\partial k}{\partial x_j} - \frac{\partial}{\partial x_j}\left[\left(\nu_{eff}\right)\frac{\partial k}{\partial x_j}\right] = \nu_T\frac{\partial \bar{u}_i}{\partial x_j}\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right) - \epsilon \tag{4.21}$$

By comparing this equation to 2.34, it is observed that the k-equation implemented in *OpenFOAM* corresponds to the standard one, except for the terms $\nu_{eff} = \nu + \nu_t$, and σ_k which is

missing in *OpenFOAM*. However, by assuming that $\sigma_k = 1$, the only difference between the two models is that, in this case, viscosity is added to ν_t.
This difference does not affect the results sensitively, how it will be discussed further in this section. Moreover, most of two equation models, at present days, use the same formulation as this one implemented by *OpenFOAM*.

Dissipation equation, *kEpsilon.C* file

```
// Dissipation equation
   tmp<fvScalarMatrix> epsEqn
   (
        fvm::ddt(epsilon_)
      + fvm::div(phi_, epsilon_)
      - fvm::Sp(fvc::div(phi_), epsilon_)
      - fvm::laplacian(DepsilonEff(), epsilon_)
     ==
        C1_*G*epsilon_/k_
      - fvm::Sp(C2_*epsilon_/k_, epsilon_)
   );
   epsEqn().relax();
   epsEqn().boundaryManipulate(epsilon_.boundaryField());

   solve(epsEqn);
   bound(epsilon_, epsilonMin_);
```

The equation for ϵ is very similar to the equation for k, with the only difference that the term *DepsilonEff* is now defined as $\frac{\nu_T}{\sigma_\epsilon} + \nu$. The production term G is defined in the same way as in the k-equation, so that the translation of the *OpenFOAM* code into math is, in this case,

$$\frac{\partial \epsilon}{\partial t} + \bar{u}_j \frac{\partial \epsilon}{\partial x_j} - \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\nu_T}{\sigma \epsilon}\right)\frac{\partial \epsilon}{\partial x_j}\right] = C_1 \frac{\epsilon}{k} \nu_T \frac{\partial \bar{u}_i}{\partial x_j}\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right) - C_2 \frac{\epsilon^2}{k} \tag{4.22}$$

Once again, by comparing this equation to 2.48, we can remark that the only difference is that, in this equation, the viscosity is added to $\frac{\nu_T}{\sigma_\epsilon}$. In order to show the effect that this term has on the final solution, in figure 4.3, two different solutions are shown, in term of velocity profiles downstream of the hump geometry. The source code of *OpenFOAM* has been modified, in order to estimate the effect of the "added " viscosity on the equations. The two different solutions "with and without added viscosity" match very well.
The last step is to define the turbulent kinematic viscosity: for the k-ϵ model, this parameter is defined as:

$$\nu_t = C_\mu \frac{k^2}{\epsilon}$$

where C_μ is a constant of the problem, while k and ϵ have been calculated above.

Definition of the turbulent kinematic viscosity, file *kEpsilon.C*

```
   // Re-calculate viscosity
   nut_ = Cmu_*sqr(k_)/epsilon_;
   nut_.correctBoundaryConditions();
```

While, for what concerning k-ω, the system is defined as follows:

Equation for ω, file *kOmega.C*

```
// Turbulence specific dissipation rate equation
    tmp<fvScalarMatrix> omegaEqn
    (
        fvm::ddt(omega_)
      + fvm::div(phi_, omega_)
      - fvm::Sp(fvc::div(phi_), omega_)
      - fvm::laplacian(DomegaEff(), omega_)
     ==
        alpha_*G*omega_/k_
      - fvm::Sp(beta_*omega_, omega_)
    );

    omegaEqn().relax();

    omegaEqn().boundaryManipulate(omega_.boundaryField());

    solve(omegaEqn);
    bound(omega_, omegaMin_);
```

Equation for k and definition of ν_T, file *kOmega.C*

```
    // Turbulent kinetic energy equation
    tmp<fvScalarMatrix> kEqn
    (
        fvm::ddt(k_)
      + fvm::div(phi_, k_)
      - fvm::Sp(fvc::div(phi_), k_)
      - fvm::laplacian(DkEff(), k_)
     ==
        G
      - fvm::Sp(Cmu_*omega_, k_)
    );

    kEqn().relax();
    solve(kEqn);
    bound(k_, kMin_);


    // Re-calculate viscosity
    nut_ = k_/omega_;
    nut_.correctBoundaryConditions();
```

In the k-ω equation, viscosity is defined as

$$\nu_t = \frac{k}{\omega}$$

σ_ϵ	1.3
$C1$	1.44
$C2$	1.92
C_μ	0.09

Table 4.1: Coefficients for the k-ϵ model.

α_ω	0.5
α	0.52
β	0.072
C_μ	0.09

Table 4.2: Coefficients for the k-ω model.

The equation for k is the same as the one used in the k-ϵ model, while the equation for ω, translated to mathematics is:

$$\frac{\partial \omega}{\partial t} + \bar{u}_j \frac{\partial \omega}{\partial x_j} - \frac{\partial}{\partial x_j}\left[\left(\nu + \alpha_\omega \nu_T\right)\frac{\partial \omega}{\partial x_j}\right] = \alpha \frac{\omega}{k}\nu_T \frac{\partial \bar{u}_i}{\partial x_j}\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right) - \beta\omega^2 \tag{4.23}$$

the value of coefficients used in *OpenFOAM* for the k-ϵ and the k-ω methods are summarized in tables 4.1 and 4.2.

In Annex, the source codes for the Spalart-Allmaras model and the k-ω-SST model are reported.

4.5 *OpenFOAM* cases

The basic directory structure for an *OpenFOAM* case, that contains the minimum set of files required to run an application, is shown in figure 4.4. It contains:

- A *constant* directory that contains a full description of the case mesh in a subdirectory *polyMesh* and files specifying physical properties for the application concerned;
- A *system* directory for setting parameters associated with the solution procedure itself. It contains at least the following three files: *controlDict* where run control parameters are set including start and end time, time step and parameters for data output; *fvSchemes* where discretization schemes used in the solution may be selected at run-time; and, *fvSolution* where the equation solvers, tolerances and other algorithm controls are set for the run;
- The *time* directories containing individual files of data for particular fields. These data include: initial values and boundary conditions that the user must specify to define the problem and results, written to file by *OpenFOAM.* The name of each time directory is based on the simulated time at which the data is written; since we start our simulations at time 0, the initial conditions are stored in a directory named *0.*

In the next chapter, the commands used to run a case are shown, while the files used in the 0, *system* and *constant* directories are reported in Annex.

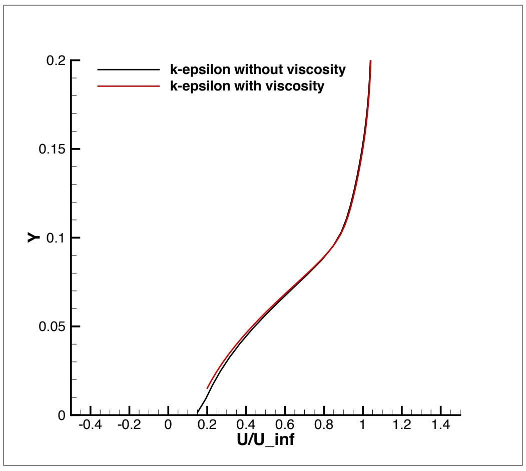


Figure 4.3: Velocity profile downstream of the hump, $x/c = 1.2$. $k - \epsilon$ methods with and without added viscosity.

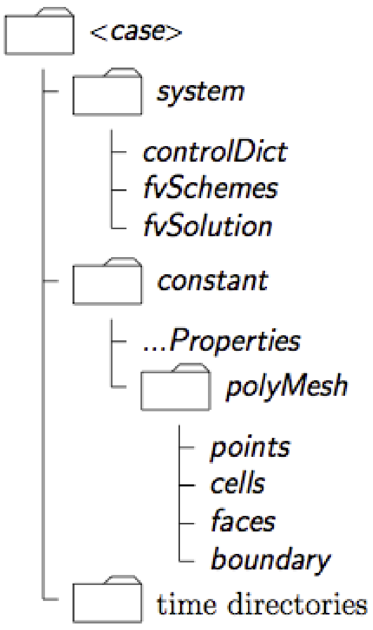


Figure 4.4: Case directory structure. [10]

Chapter 5

Equipment: Hardware and Software

In the previous chapter, the structure and the algorithms of the open source software *OpenFOAM* have been treated. This section explains how simulations are run and how solutions have been post-processed and analyzed after having listed all the instruments and the softwares used for that purpose.
After a brief introduction to Pleiades, the *NASA* super computer, a description of all the most important commands to run simulations and to post-process data will follow.

5.1 Pleiades

Pleiades, which ranks 11th on the TOP500 list of the world's most powerful supercomputers, represents NASA's state-of-the-art technology for meeting the agency's supercomputing requirements, enabling NASA scientists and engineers to conduct modeling and simulation for NASA missions. This distributed-memory SGI ICE cluster is connected with *InfiniBand* in a dual-plane hypercube technology.

The system contains the following types of **Intel-Xeon** processors: E5-2670 (Sandy Bridge), X5670 (Westmere), X5570 (Nehalem), and E5472 (Harpertown). Pleiades is named after the astronomical open star cluster of the same name. [24]

5.1.1 History

Built in 2008, the supercomputer debuted as the third most powerful supercomputer in the world at 487 teraflops. It originally contained 100 SGI Altix ICE 8200EX racks with 12,800 Intel Xeon quad-core E5472 Harpertown processors connected with more than 20 miles of InfiniBand double data rate (DDR) cabling.
With the addition of ten more racks of quad-core X5570 Nehalem processors in 2009, Pleiades ranked sixth on the November 2009 TOP500 with 14,080 processors running at 544 teraflops.
In January 2010, the scientists and engineers at NAS successfully completed a "live integration" of another ICE 8200 rack by connecting the new rack's InfiniBand dual port fabric via 44 fibre cables while the supercomputer was still running a full workload, saving 2 million hours in productivity that would previously have been lost.

Another expansion in 2010 added 32 new SGI Altix ICE 8400 racks with Intel Xeon six-core X5670 Westmere processors, bringing up to 18,432 processors (81,920 cores in 144 racks) at a theoretical peak of 973 teraflops and a LINPACK rating of 773 teraflops. NASA also put an emphasis on keeping Pleiades energy efficient, increasing the power efficiency with each expansion so that in 2010 it was three times more power-efficient than the original 2008 components, which were the most power-efficient at the time. The integration of the six-core Westmere nodes also required new quad data rate (QDR) and hybrid DDR/QDR InfiniBand cabling, making the world's largest InfiniBand interconnect network with more than 45 miles of cable.
After another 14 ICE 8400 racks containing Westmere processors were added in 2011, Pleiades ranked seventh on the TOP500 list in June of that year at a LINPACK rating of 1.09 petaflops, or 1.09 quadrillion floating point operations per second.
InfiniBand DDR and QDR fiber cables are used to connect the all of nodes to each other, as well as to the mass storage systems at NAS and the hyperwall visualization system, creating a network of made up of more than 65 miles of InfiniBand fabric, the largest of its kind in the world. Pleiades is built in a partial 11-D hypercube technology, where each node has eleven connections to eleven other nodes, with some making up to twelve connections to form a 12-D hypercube.
In 2012, NASA and partners SGI and Intel began working on the integration of 24 new Altix ICE X racks with Intel Xeon eight-core E5-2760 Sandy Bridge processors to replace 27 of the original Alitx 8200 racks containing quad-core Harpertown processors. With a total of 126,720 processor cores and over 233 terabytes of RAM across 182 racks, the expansion increased Pleiades' available computing capacity 40 percent. Each new Sandy Bridge node has four networking links using fourteen data rate (FDR) InfiniBand cable for a total transfer bandwidth of 56 gigabits (about 7 gigabytes) per second.

5.1.2 System architecture

The operating system of **Pleiades** is **SUSE - Linux**; the job scheduler is **PBS** and the compilers are: **Intel**, **GNU**, **C**, **C++**, **Fortran**. The system architecture and the **Pleiades** Node details can be resumed as follows:

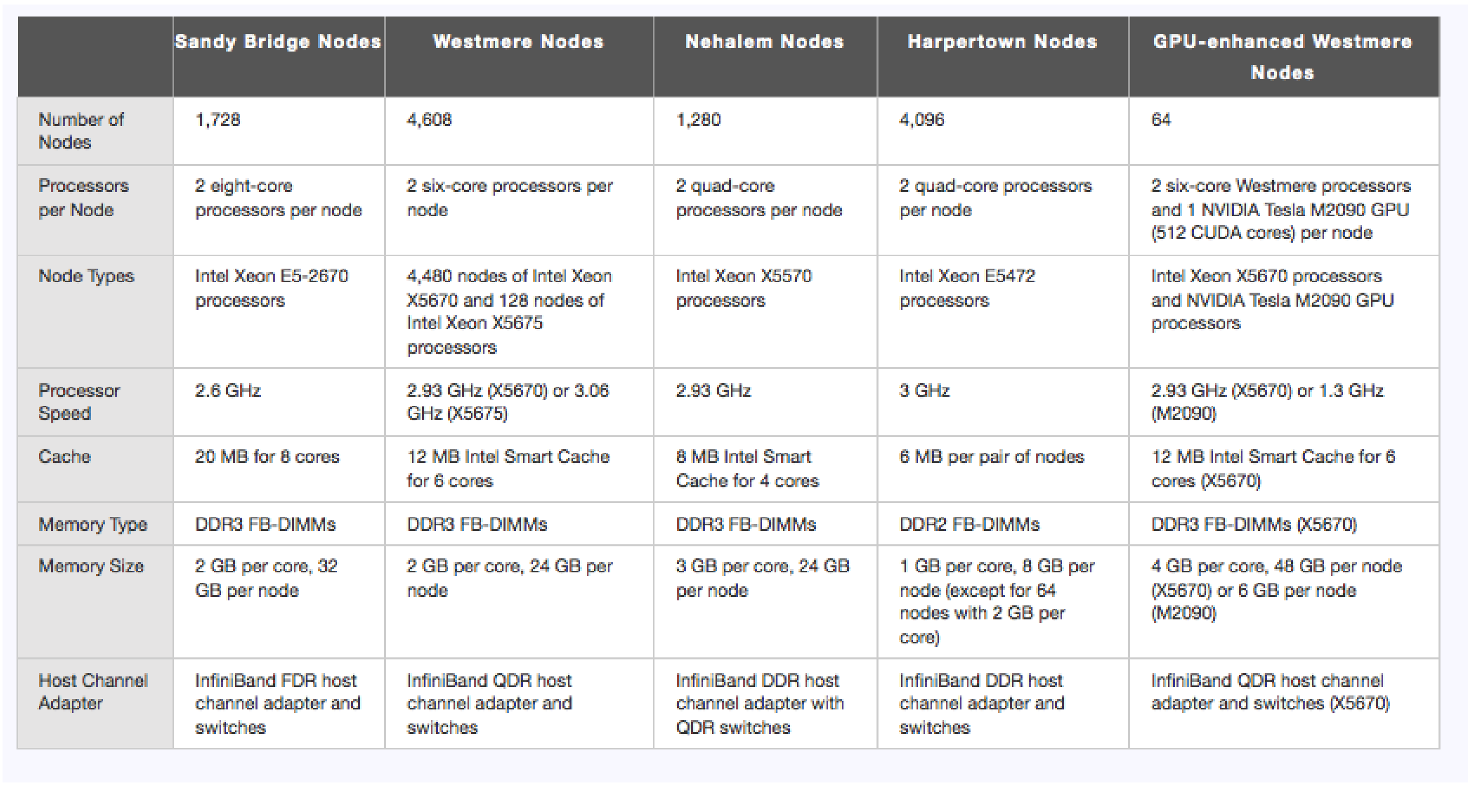
- Manufacturer: **SGI**;
- 182 racks;
- 1.75 Pflop/s peak cluster;
- 2 racks (64 nodes total) enhanced with **NVIDIA** graphics processing unit (GPU): 43 teraflops total;
- Total processors: 23.552;
- Total cores: 126.720 (32.768 additional GPU cores);
- Total memory: 233 TB

In the fig 5.2 the details of **Pleiades** nodes are illustrated:

5.1.3 Role at NASA

Pleiades is part of NASA's High-End Computing Capability (HECC) Project and represents NASA's state-of-the-art technology for meeting the agency's supercomputing requirements, enabling NASA scientists and engineers to conduct high-fidelity modeling and simulation for NASA

Figure 5.1: A picture of Pleiades in the super-computer department. NASA Ames Research Center. Moffett Field, CA.



	Sandy Bridge Nodes	Westmere Nodes	Nehalem Nodes	Harpertown Nodes	GPU-enhanced Westmere Nodes
Number of Nodes	1,728	4,608	1,280	4,096	64
Processors per Node	2 eight-core processors per node	2 six-core processors per node	2 quad-core processors per node	2 quad-core processors per node	2 six-core Westmere processors and 1 NVIDIA Tesla M2090 GPU (512 CUDA cores) per node
Node Types	Intel Xeon E5-2670 processors	4,480 nodes of Intel Xeon X5670 and 128 nodes of Intel Xeon X5675 processors	Intel Xeon X5570 processors	Intel Xeon E5472 processors	Intel Xeon X5670 processors and NVIDIA Tesla M2090 GPU processors
Processor Speed	2.6 GHz	2.93 GHz (X5670) or 3.06 GHz (X5675)	2.93 GHz	3 GHz	2.93 GHz (X5670) or 1.3 GHz (M2090)
Cache	20 MB for 8 cores	12 MB Intel Smart Cache for 6 cores	8 MB Intel Smart Cache for 4 cores	6 MB per pair of nodes	12 MB Intel Smart Cache for 6 cores (X5670)
Memory Type	DDR3 FB-DIMMs	DDR3 FB-DIMMs	DDR3 FB-DIMMs	DDR2 FB-DIMMs	DDR3 FB-DIMMs (X5670)
Memory Size	2 GB per core, 32 GB per node	2 GB per core, 24 GB per node	3 GB per core, 24 GB per node	1 GB per core, 8 GB per node (except for 64 nodes with 2 GB per core)	4 GB per core, 48 GB per node (X5670) or 6 GB per node (M2090)
Host Channel Adapter	InfiniBand FDR host channel adapter and switches	InfiniBand QDR host channel adapter and switches	InfiniBand DDR host channel adapter with QDR switches	InfiniBand DDR host channel adapter and switches	InfiniBand QDR host channel adapter and switches (X5670)

Figure 5.2: Pleiades Node Detail

missions in Earth studies, space science, aeronautics research, as well as human and robotic space exploration.
Some of the scientific and engineering projects run on Pleiades include:

- The Kepler Mission, a space observatory launched in March 2009 to locate Earth-like planets, monitors a section of space containing more than 200,000 stars and takes high-resolution images every 30 minutes. After the operations center gathers this data, it is pipelined to Pleiades in order to calculate the size, orbit, and location of the planets surrounding these stars. As of February 2012, the Kepler mission has discovered 1,235 planets, 5 of which are approximately Earth-sized and orbit within the "habitable zone" where water can exist in all three forms (solid, liquid, gas).

- Research and development of next generation space launch vehicles is done on Pleiades using cutting-edge analysis tools and computational fluid dynamics (CFD) modeling and simulation in order to create more efficient and affordable space launch system and vehicle designs. Research has also been done on reducing noise created by the landing gear of aircraft using CDF code application to detect where the sources of noise are within the structures.

- Astrophysics research into the formation of galaxies is run on Pleiades to create simulations of how our own Milky Way Galaxy was formed and what forces might have caused it to form in its signature disk-shape. Pleiades has also been the supercomputing resource for dark matter research and simulation, helping to discover gravitationally bound "clumps" of dark matter within galaxies in one of the largest simulations ever done, in terms of particle numbers.

- Visualization of the Earth's ocean currents using a NASA-built data synthesis model for the Estimating the Circulation and Climate of the Ocean (ECCO) Project between MIT and the NASA Jet Propulsion Laboratory in Pasadena, California. According to NASA, the "ECCO model-data syntheses are being used to quantify the ocean's role in the global carbon cycle, to understand the recent evolution of the polar oceans, to monitor time-evolving heat, water, and chemical exchanges within and between different components of the Earth system, and for many other science applications."

5.2 Running in parallel

Our simulations were run on 500 processors in parallel for the 3D cases and 100 processors for the 2D cases.
In order to have access to at least 100 processors in **Pleiades**, the following command must be used:

```
qsub -I -lselect=10:ncpus=12:mpiprocs=12:model=wes -lwalltime=5:00:00
```

Where the first digit refers to the number of nodes the user requires while the second and the third digit refer to the number of processors available on each node, for a total of 120 processors. In this example *Westmere* nodes (see fig 5.2) are going to be used for 5 hours.
The instruction changes, if the user wants to use the nodes of another sub-system, in according with its characteristic. If the user wants to use *Sandy Bridge*, the command becomes:

```
qsub -I -lselect=7:ncpus=16:mpiprocs=16:model=san -lwalltime=5:00:00
```

The number of processors available on each node has now changed, so that the number of nodes to require to get at least 100 processors is now 7, instead of 10.
Once the user has access to the processors he required, he can enter into the directory of the case he wants to simulate and type:

```
decomposePar
```

In this example, the mesh is decomposed in 100 parts. *OpenFOAM* reads the file *decomposeParDict* (see the chapter 8.4 in Annex, to understand how this file is written), and it sends each portion of the mesh to a specific processor.
At this point it is enough to type:

```
mpirun -np 100 simpleFoam -parallel
```

and the simulation is launched. Once the simulation is over, the mesh has to be reconstructed, by using the command:

```
reconstructPar
```

At this point the solution at different iteration steps is available.
Since the waiting time to have access to the processors may be very long, especially at the increasing of the *wall time*, the user may prefer to run simulations in the background.
An import ant advantage of running in the background is that several simulations can be run by *Pleiades* at the same time.
The procedure to run in the background is very similar to that illustrated above. The only difference is that the user has to create a text file in which he writes the commands that the supercomputer has to execute. The directory of the case the user wants to simulate must be specified.
Suppose the user generates a file called:

```
 launch_simulation.txt
```

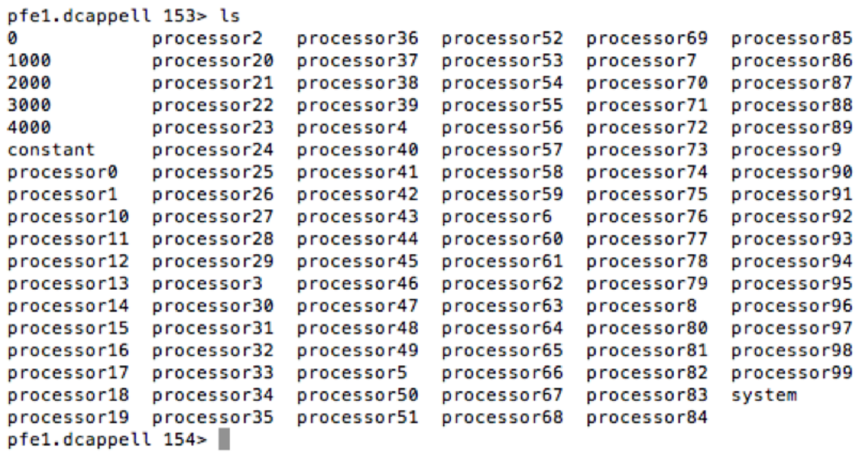
In this file he will write all the commands described above, by specifying the *directory* where the case is. An example could be:

```
decomposePar -case /nobackupp2/dcappell/Hump_model_SpalartAllmaras

mpirun -np 100 simpleFoam -case /nobackupp2/dcappell/Hump_model_SpalartAllmaras -parallel

reconstructPar -case /nobackupp2/dcappell/Hump_model_SpalartAllmaras
```

In order to let *Pleiades* read the instructions contained in the file and to access the processors that are needed, the following instruction has to be launched:

```
 qsub -l select=7:ncpus=16:mpiprocs=16:model=san -lwalltime=10:00:00 launch_simulation
```

The command is very similar to that used to run the simulations in the foreground, except that here the name of the file created above must be specified.
After the simulation is run, the folder where the case is run will contain all the directories shown in picture 5.3:
de



```
pfe1.dcappell 153> ls
0            processor2   processor36  processor52  processor69  processor85
1000         processor20  processor37  processor53  processor7   processor86
2000         processor21  processor38  processor54  processor70  processor87
3000         processor22  processor39  processor55  processor71  processor88
4000         processor23  processor4   processor56  processor72  processor89
constant     processor24  processor40  processor57  processor73  processor9
processor0   processor25  processor41  processor58  processor74  processor90
processor1   processor26  processor42  processor59  processor75  processor91
processor10  processor27  processor43  processor6   processor76  processor92
processor11  processor28  processor44  processor60  processor77  processor93
processor12  processor29  processor45  processor61  processor78  processor94
processor13  processor3   processor46  processor62  processor79  processor95
processor14  processor30  processor47  processor63  processor8   processor96
processor15  processor31  processor48  processor64  processor80  processor97
processor16  processor32  processor49  processor65  processor81  processor98
processor17  processor33  processor5   processor66  processor82  processor99
processor18  processor34  processor50  processor67  processor83  system
processor19  processor35  processor51  processor68  processor84
pfe1.dcappell 154> 
```

Figure 5.3: Screenshot of the window terminal, showing all the directories of our case after the execution "*recontructPar*"

5.3 Post-processing

The default visualization application integrated in *OpenFOAM* is the open-source software *Paraview.*
In order to convert the results in the *Paraview* format, the following command is used:

```
foamToVTK
```

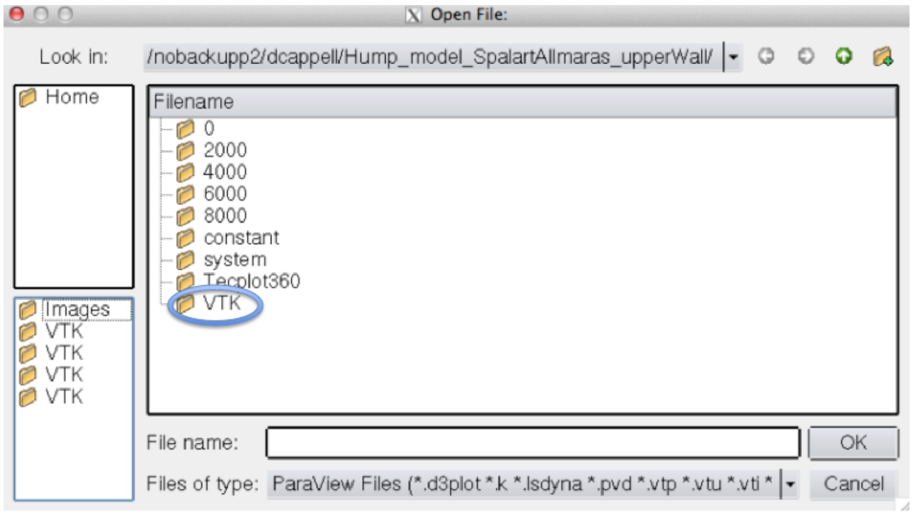
Once opened *Paraview*, the user goes into the directory **VTK**, automatically generated after the execution of the previous instruction (figure 5.4) and he selects the file with the *VTK* extension. After clicking on *Apply*, in the *Properties* window, the case will be uploaded. At this point, the user has to move into the *Display* window to select the field that he wants to visualize (pressure, velocity, viscosity...) as shown in figure 5.5

OpenFOAM also disposes of different filters, to extract a plane (*Slice*), to plot the parameters along a line -*plot over a line*- (this tool is very useful if we want to plot the velocity profile in y-direction, for example) to seed a vector field with points and then traces those seed points through the vector field, and so on...
The limit of *Paraview* is that it is not able to plot the parameters over a curved line. This means that it does not give the possibility to plot the coefficient pressure over the wall of the hump.
For this reason, in this work, a more complete visualization software was used: *TecPlot.*
Results are post-processed in *TecPlot* through the command:

```
foamToTecplot360
```

After having launched the instruction above, a folder called *Tecplot360* will be generated. In this folder there are all files containing solutions over the time and a file containing information about the mesh[1].

[1]The case studied in this work is steady: time is only a counter of the iterations required before getting numerical convergence.

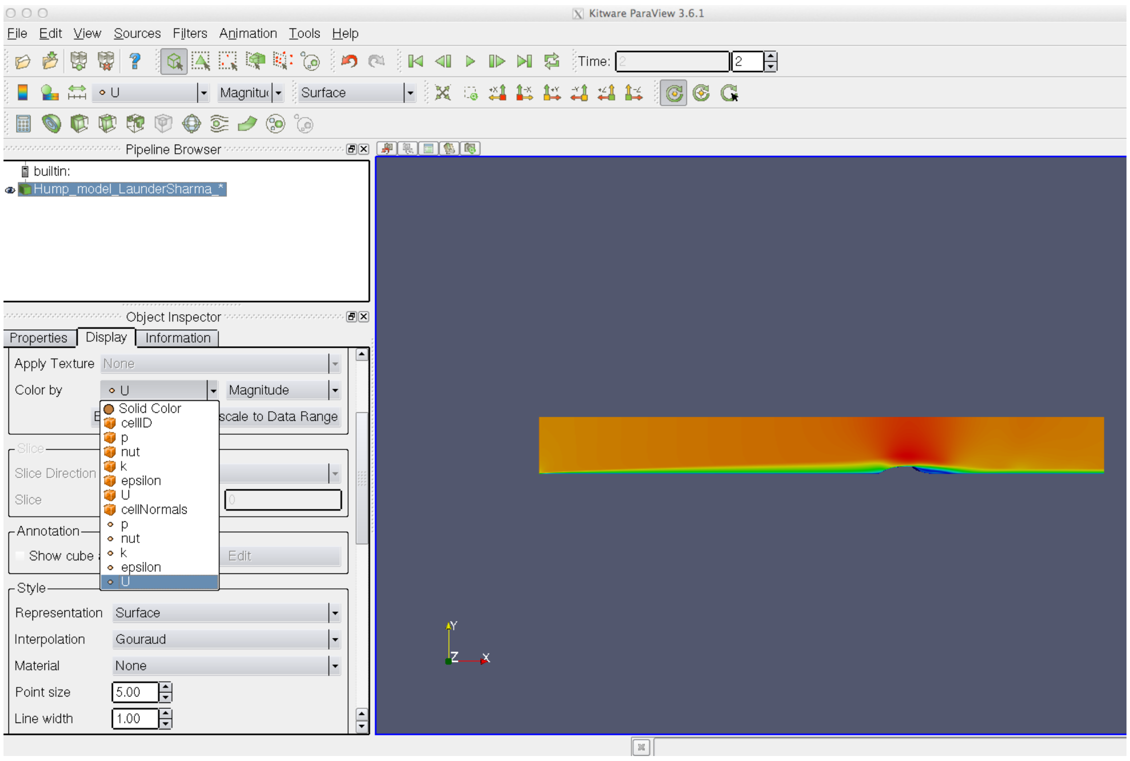


Look in: /nobackupp2/dcappell/Hump_model_SpalartAllmaras_upperWall/

File name:

Files of type: ParaView Files (*.d3plot *.k *.lsdyna *.pvd *.vtp *.vtu *.vti *

Figure 5.4: Opening a VTK file in Paraview



Time: 2

Apply Texture: None

Color by: U, Magnitude

Rescale to Data Range

Slice Direction

Slice: 0

Representation: Surface

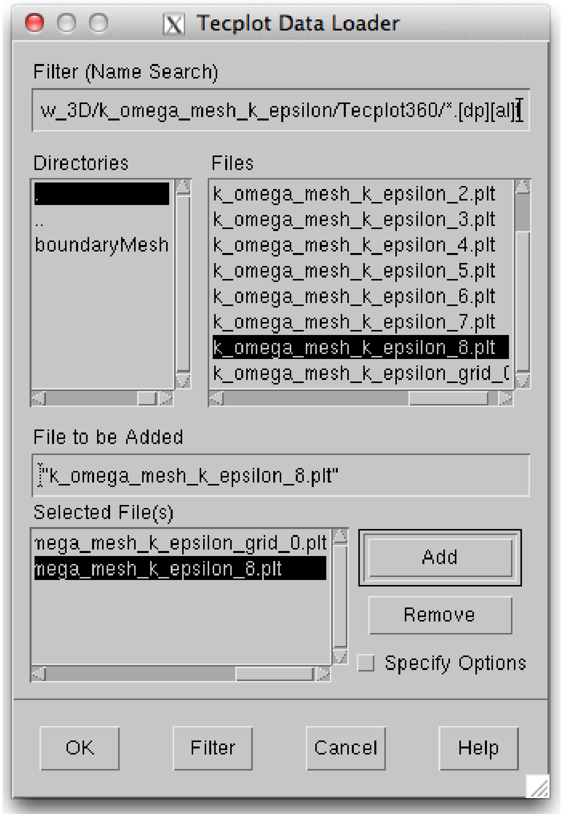
Interpolation: Gouraud

Material: None

Point size: 5.00

Line width: 1.00

Figure 5.5: Selection of the field to visualize



Tecplot Data Loader

Filter (Name Search): w_3D/k_omega_mesh_k_epsilon/Tecplot360/*.[dp][al]

Directories:
```
.
..
boundaryMesh
```

Files:
```
k_omega_mesh_k_epsilon_2.plt
k_omega_mesh_k_epsilon_3.plt
k_omega_mesh_k_epsilon_4.plt
k_omega_mesh_k_epsilon_5.plt
k_omega_mesh_k_epsilon_6.plt
k_omega_mesh_k_epsilon_7.plt
k_omega_mesh_k_epsilon_8.plt
k_omega_mesh_k_epsilon_grid_(
```

File to be Added: "k_omega_mesh_k_epsilon_8.plt"

Selected File(s):
```
nega_mesh_k_epsilon_grid_0.plt
nega_mesh_k_epsilon_8.plt
```

Figure 5.6: Uploading a case in Tecplot.

We have to select the file containing the mesh, and at least one of the files containing the solution over the time (the last one has to be selected to visualize the solution at the last time-step). In order to do that, the user opens the *File* window and he selects the *TecPlot Data Loader* command. He goes through the *Tecplot* path of his case, he selects the option *multiple files* and uploads the files containing the mesh and the solution, as shown in picture 5.6.
At this point, At this point, the case is uploaded and the boundaries of the geometry are visible. If the user wants to investigate the behavior of the solution at the center plane, he has to extract a slice of plane by clicking on *Data* - *Extract* - *Slice From Plane*, as shown in figure 5.7 and select the Z coordinate in which the plane is located.
After having selected the *2D Cartesian* view (top-left of the screen, as highlighted in figure 5.8) a contour of pressure or velocity[2] is shown. If the user wants to extract pressure over the wall of the geometry, he has to select *Extract* - *FE-Boundary* from the *Data* menu, as shown in figure 5.8. If we are interested in extracting the velocity profile, we can select *Extract* - *Points from Polyline* and then draw a vertical line over the X location where profile wants to be extracted.
Once he has extracted all the fields he needs, in order to have the 2-D plot of the fields, he selects the *XY Line* view, and sets the variables he wants to plot in the *Mapping Style* window (figure 5.9). For further information about *TecPlot*, the reader is referenced to the official guide[25].

[2]It depends on which parameter is set by default. To change it, click on the three-point button at the right of *Contour* button

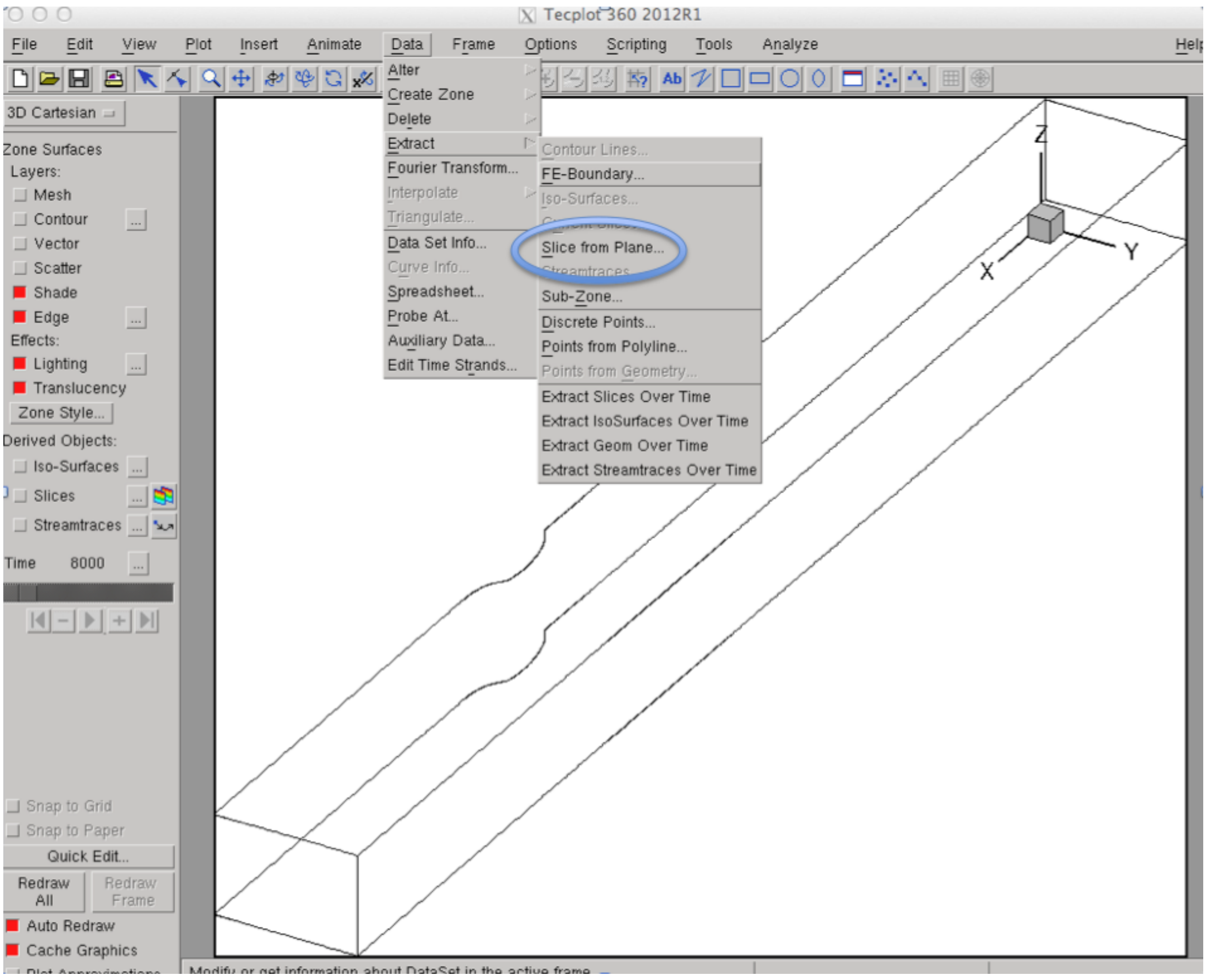


Figure 5.7: Extraction of a plane in Tecplot.

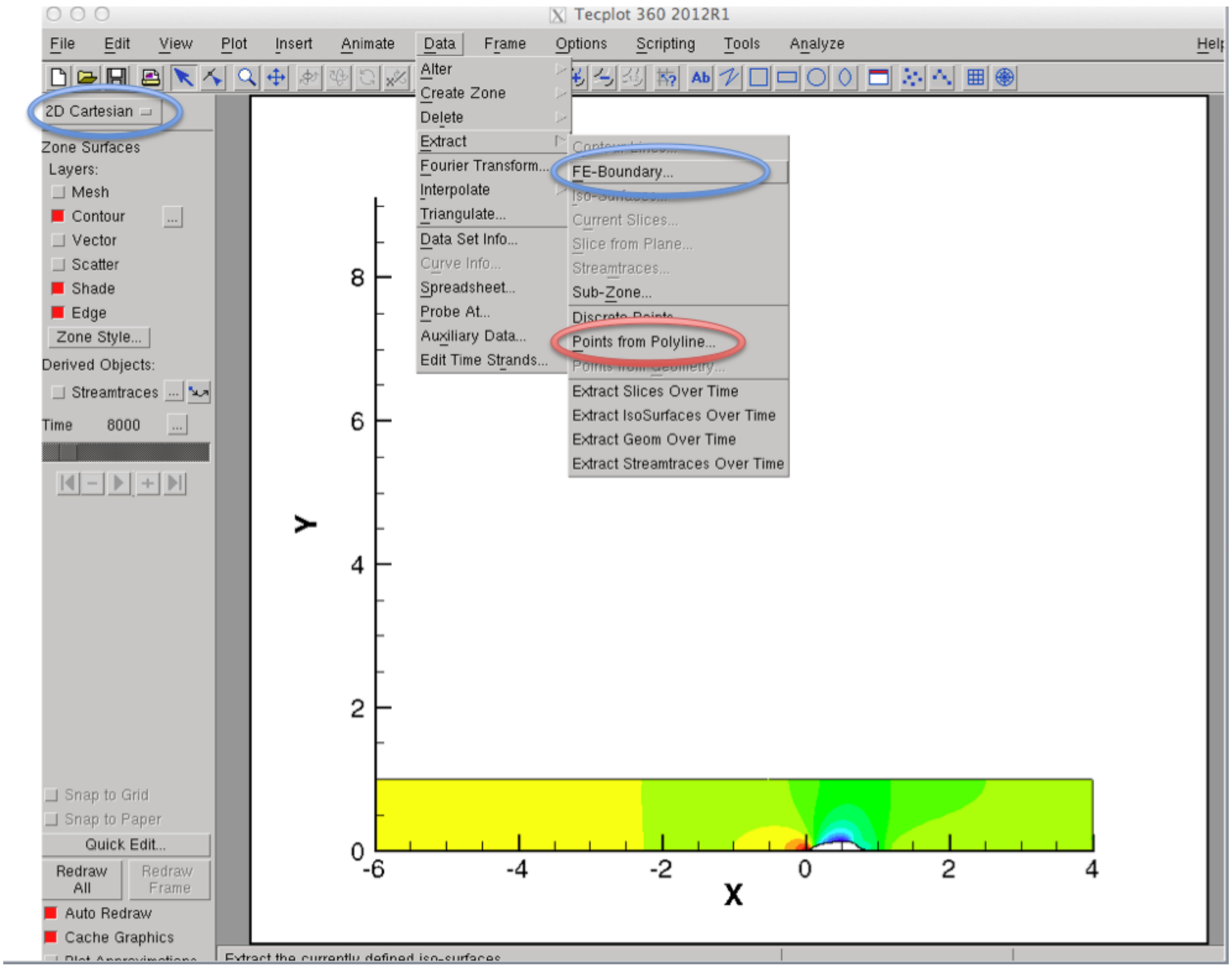
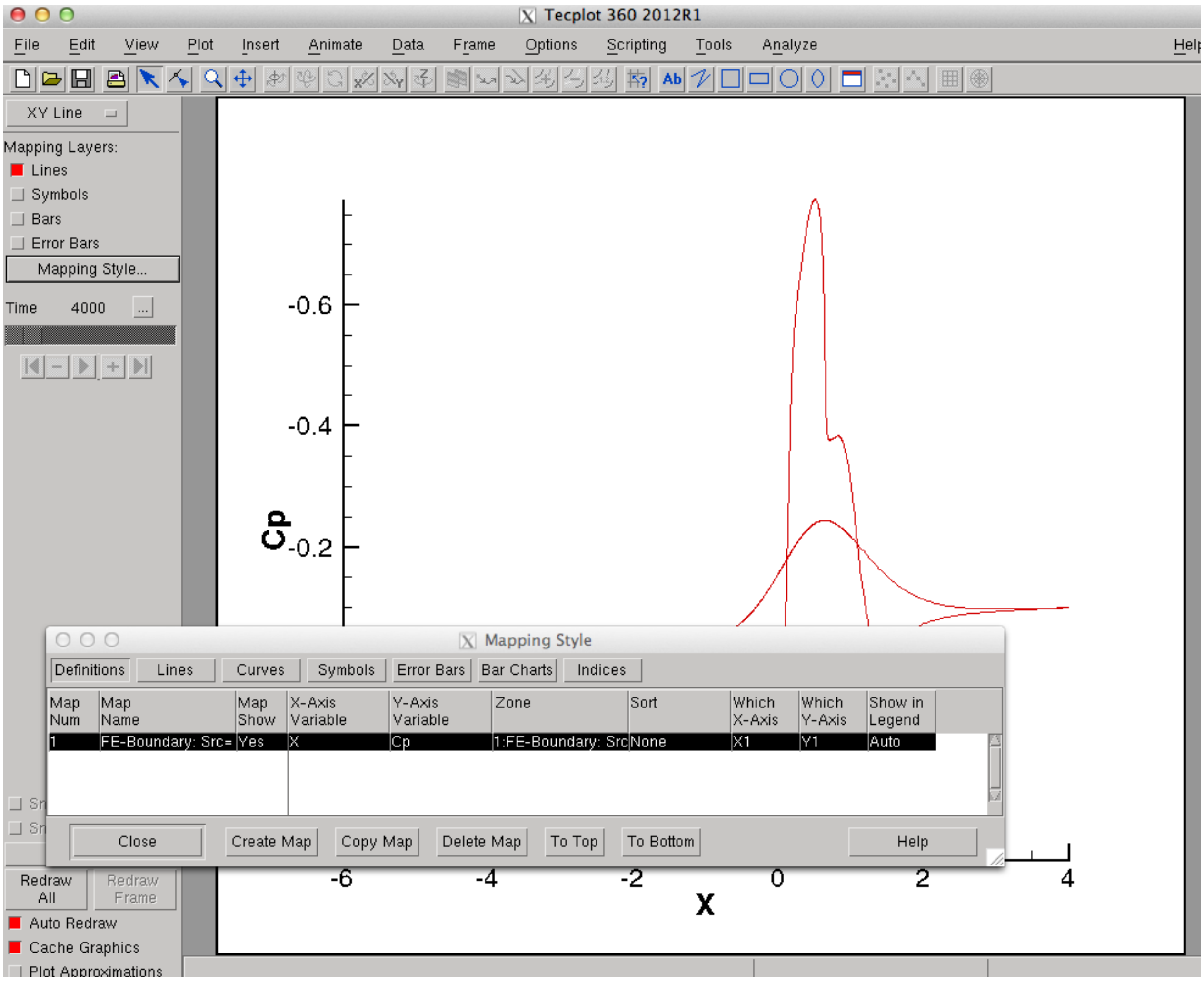


Figure 5.8: Extraction of the boundary and of Points from Polyline .



Map Num	Map Name	Map Show	X-Axis Variable	Y-Axis Variable	Zone	Sort	Which X-Axis	Which Y-Axis	Show in Legend
1	FE-Boundary: Src=	Yes	X	Cp	1:FE-Boundary: Src	None	X1	Y1	Auto

Figure 5.9: Display of the extracted variables on a 2D graph.

Chapter 6

Hump Flow: Results

This section shows the most significant results obtained by running the *hump flow* model in *OpenFOAM*.
In order to compare the numerical results to the experimental ones, we respected the fluid-dynamic similitude was respected, by fixing *Reynolds* number and *Mach* number like in the experiments.

6.1 Fluid dynamic similitude

In order to operate in subsonic, velocity is chosen to be 34 m/s, so that $M = 0.1$
The chord of the model used in this work goes from 0 to 1m (in order to easily compare this case with the dimensionless results provided by the other works: x/c, y/c, U/U_∞). This means that in order to use the same *Reynolds* number[1] as the experimental one (the Reynolds number for the baseline case is $\sim$ 930000, see [3],[1], [20]) the value of the kinematic viscosity in the *transportProperties* file[2] has to be changed as follows:[3]

$$\nu = \frac{U_\infty c}{Re_c} = \frac{34 \cdot 1}{930000} = 3.66\ 10^{-5} \frac{m^2}{s} \tag{6.1}$$

In the following table, the basic properties of the "Hump flow" are summarized.

[1]We refer to the Reynolds number based on the chord of our model: Re_c

[2]All the parameters defined in this file are shown in Appendix.

[3]This is the advantage of CFD: we can easily change the proprieties of the gas , without employing difficult devices consisting in changing the pressure and the temperature of the wind tunnel.

c	1 m
U_∞	34 m/s
ν	3.66 10^{-5} m^2/s
M	0.1
Re_c	930 000

Table 6.1: Properties of our "hump" flow

	nCells	y points	grading	nIT	nProc	Time
Spal-Allmar	336 000	300	500	30.000	100	2h30
k-ϵ	224 000	200	100	8000	100	2h
k-ω	336 000	500	300	30.000	100	2h30
k-ω-SST	336 0002	500	300	30.000	100	2h30

Table 6.2: Size of mesh and computational time for the 2D case

6.2 Mesh

For the simulations run in this work, three different meshes have been used: one with the inlet set at -6 x/c upstream of the leading edge of the hump which wants to reproduce the experiments as well as possible, the second is with inlet located at -1 x/c upstream of the hump which is interesting to estimate the influence of the boundary layer on the flow: the goal is to compare the effects of the development of the boundary layer along the lower wall of the mesh [4]; the third includes the end-side plates for the study of the 3-D flow.
The condition on the upper wall has been changed in order to study the theoretical case (no wall perturbing the flow) and the experimental one: in this case, the effects of the upper wall of the wind tunnel on the flow have been taken into account.
For the inlet located at -6 x/c, the length of the splitter plate has been chosen in order to match the measured velocity profiles at $x/c = -2.1$ (see figure 6.2), [7]. Both 2D and 3D simulations have been run in order to evaluate the effects of the end side plates on the flow. Since the mesh generated by the *blockMesh* utility is 3D by default, an *empty* condition has to be used for the side walls, in order to switch from a 3D to a 2D case.
The mesh has been built by using the *OpenFOAM* utility *blockMesh*[5], and it is built in order to be finer in proximity of the lower wall and of the hump[6]. In figure 6.1 a simplified sketch of the mesh used to run our simulations is shown. The script of the mesh (*blockMeshDict* file) is available in the appendix.
In table 6.2 the characteristics of the mesh used in the 2-D case and the computational time to get to convergence are summarized.
The grading command defines the factor scale of the cells in y direction. A value of 100 means that the cells near the lower wall are 100 times smaller than the cell close to the upper wall.

6.3 Boundary Conditions

Although one of the most important characteristic of the "hump" flow is its insensitiveness to the inlet conditions, it is important to set coherent values at the inlet in order not to have numerical problems in the solution convergence.
An important parameter to define is the turbulent intensity I, defined as $I = \frac{\sqrt{u^2}}{U_\infty}$, where u is the value of velocity fluctuation in the x-direction.
The physics of the problem can be simplified, by supposing isotropic turbulence, so that the following relationship for the turbulent kinetic energetic can be written:

$$k = \frac{3}{2}(IU_\infty)^2 \tag{6.2}$$

[4]The lower wall of the mesh reproduces the splitter plate used in the experiments
[5]By default, OpenFOAM handles both structured and unstructured meshes as unstructured
[6]The points are scaled by using the option *simpleGrading*

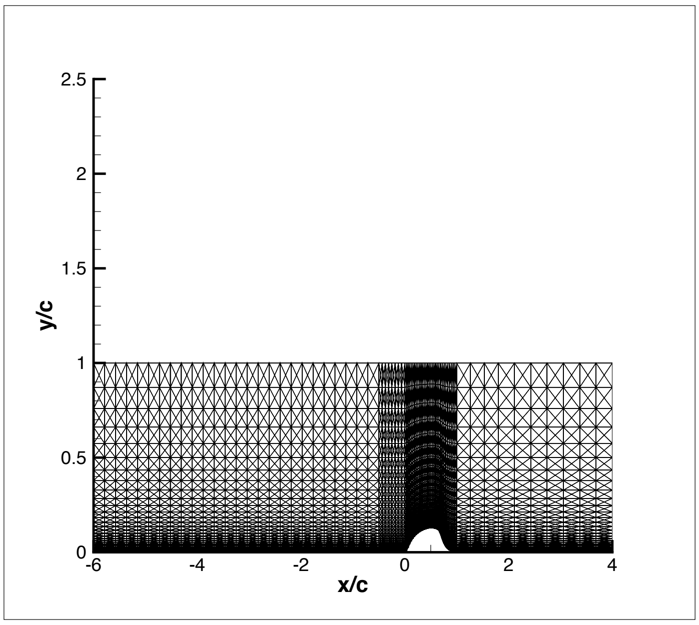


Figure 6.1: Sketch of our mesh. 1/10 lines plotted

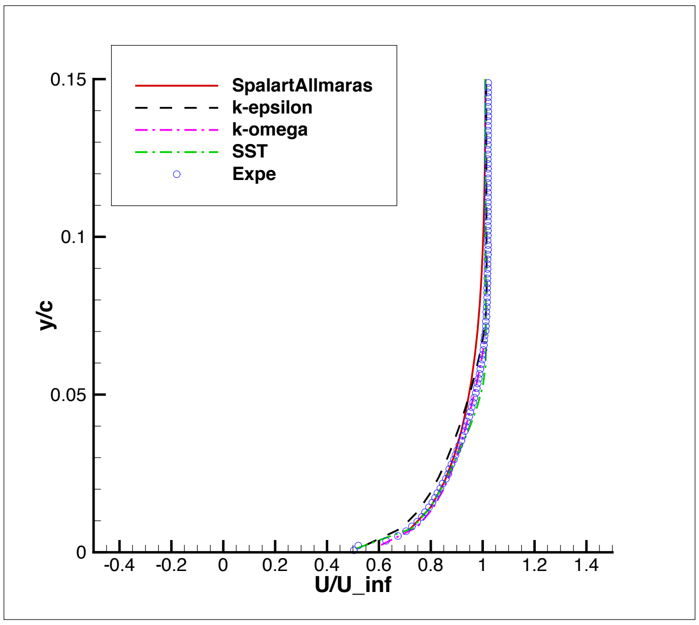


Figure 6.2: Velocity profiles at the inlet, $x/c = -2.14$.

	inlet	**lower wall**	**upper wall**	**outlet**
U	34 m/s	0 m/s	symmetry plane	zero gradient
p	zero gradient	zero gradient	symmetry plane	$p = p_\infty$
ν_t	3.66e-4 m^2/s	wall function	symmetry plane	zero gradient
k	0.39 m^2/s^2	0 m^2/s^2	symmetry plane	zero gradient
ϵ	37.4 m^2/s^3	wall function	symmetry plane	zero gradient
ω	500 1/s	wall function	symmetry plan	zero gradient

Table 6.3: Boundary conditions for the 2D case. Upper Wall considered as a symmetry plane.

Turbulent viscosity, ν_t is the unknown of the problem and it will be calculated after the solution of turbulence equations (k-ϵ, k-ω, *Spalart Allmaras...*). To a first approximation, the following relationship can be used[26]:

$$\frac{\nu_t}{\nu} \simeq 10 \text{ to } 100 \tag{6.3}$$

By considering the relationship between k, ϵ, ω and ν_t, ϵ and ω can be estimated, once k and ν_t are known.
By supposing a value of 1.5% for I, we have $k = \frac{3}{2}(IU_\infty)^2 = 0.39$.
Note that ν is fixed, in order to respect the Reynolds number, and its value is $3.66 \cdot 10^{-5}$; it means that the turbulent kinematic viscosity will be in the range $\nu_t \simeq 3.66 \cdot 10^{-4}$ to $3.66 \cdot 10^{-3}$.
By using the relationships between k, ϵ, ω and ν_t (chapter 2), the following relationships for ϵ and ω are obtained:

$$\epsilon = C_\mu \frac{k}{\nu_t} \simeq 4 \text{ to } 40 \tag{6.4}$$

$$\omega \simeq \frac{k}{\nu_t} \simeq 100 \text{ to } 1000 \tag{6.5}$$

Velocity is forced by Mach number to be 34 m/s at the inlet, while for pressure, the *zero gradient* condition has been chosen.
At the lower wall, viscosity imposes the values for the velocity and the turbulent kinetic energy to be 0, while, since pressure does not change in the y-direction through the boundary layer (hypothesis of thin boundary layers) the *zero gradient* condition for pressure has been used. For what concerning ϵ and ω, they are calculated through the use of wall functions (*epsilonWallFunction* and *omegaWallFunction*[7]).
For the 2D case, two different conditions have been used for the upper wall: in one case, the upper wall has been supposed far enough not to be perturbed by the hump - *symmetric plane* condition - while in the other case, a viscous wall condition has been used, in order to evaluate the effects of the wind tunnel wall on the flow.
At the outlet, the *zero gradient* condition is used for all the variables, except from pressure which is forced to be equal to the asymptotic pressure. Since in the simpleFoam solver $p_{simpleFoam} = \frac{\Delta p}{\rho}$, p is zero at the outlet.
The boundary values are summarized in table 6.3 and in table 6.4.

[7]Boundary conditions are specified in the 0 directory of our case. We report them in Appendix

	inlet	**lower wall**	**upper wall**	**outlet**
U	34 m/s	0 m/s	0 m/s	zero gradient
p	zero gradient	zero gradient	zero gradient	$p = p_\infty$
ν_t	3.66e-4 m^2/s	wall function	wall function	zero gradient
k	0.39 m^2/s^2	0 m^2/s^2	0 m^2/s^2	zero gradient
ϵ	37.4 m^2/s^3	wall function	wall function	zero gradient
ω	500 1/s	wall function	wall function	zero gradient

Table 6.4: Boundary conditions for the 2D case. Effects of the upper wall on the flow.

6.4 Pressure

In this section the results of pressure for the 2D case are shown. In figures 6.3, 6.4, 6.5, 6.6, the trend of pressure for the different turbulence methods is shown over the entire domain, while in figures 6.7, 6.8, 6.9, pressure coefficients are shown in order to compare numerical results to the experimental data.
These simulations are run by using both 1 equation and 2 equation turbulence models, more specifically: *Spalart-Allmaras*[8], k-ϵ, k-ω, k-ω-SST[9]. Pressure shows how the flow is accelerated up to around the mid-chord of the hump, where a peak magnitude C_p is observed. A sudden drop of pressure afterwards leads to the separation at around x/C=0.65, which corresponds to the location of the cavity slot for the controlled case[10].
Two experimental curves were measured: one is measured in the presence of the side plates, the other is measured without side plates. This work will refer to both. In order to differentiate them, the label *Exp* and *Exp-no-plates* will be used in the legend of graphs. The effect of the blockage provoked by the walls significantly affects experimental measurements, how remarked by Rumsey in his workshop (section 3.1, figure 3.6).
The boundary layer on the side plates makes the flow accelerate more, so that pressure decrease over the hump is higher.
In figure 6.7, numerical results[11] are compared to experiments: all the methods are very close to the experimental results without plates, except $k - \omega - SST$ which over-predicts pressure.
In figure 6.8, simulations are run by taking into account the effects of the upper wall and they are compared to the experiments: in this case a slightly improvement in pressure prediction can be observed, especially for what concerns $k - \omega - SST$ which matches the experimental results very well. Downstream of the separation point, pressure is over predicted by *Spalart Allmaras* and $k - \epsilon$, while $k - \omega$ and SST provides accurate results.
When the inlet is located at -1 x/c, higher values of depressions are obtained over the hump: all the methods get closer to the experimental data with plates; some numerical methods (like $k - \omega - SST$) under predict pressure.
This means that the boundary layer development upstream of the hump - along the splitter plate (or the lower wall of the mesh) - significantly affects the flow over the hump.
The comparison of velocity profiles downstream of the hump clarifies this behavior as shown in section 6.5.

[8]1 equation model
[9]2 equation models
[10]In this study the controlled case will not be studied
[11]In this plot, the upper wall is treated as a symmetric plane

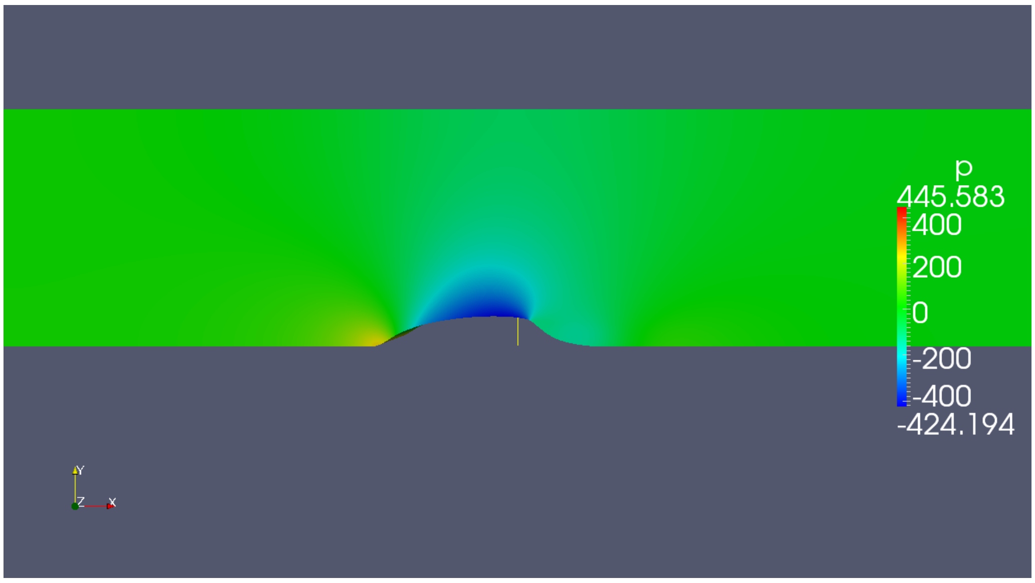


Figure 6.3: $\Delta p/\rho$ distribution over the plane. 2D case. Inlet = -6 x/c. Spalart-Allmaras model

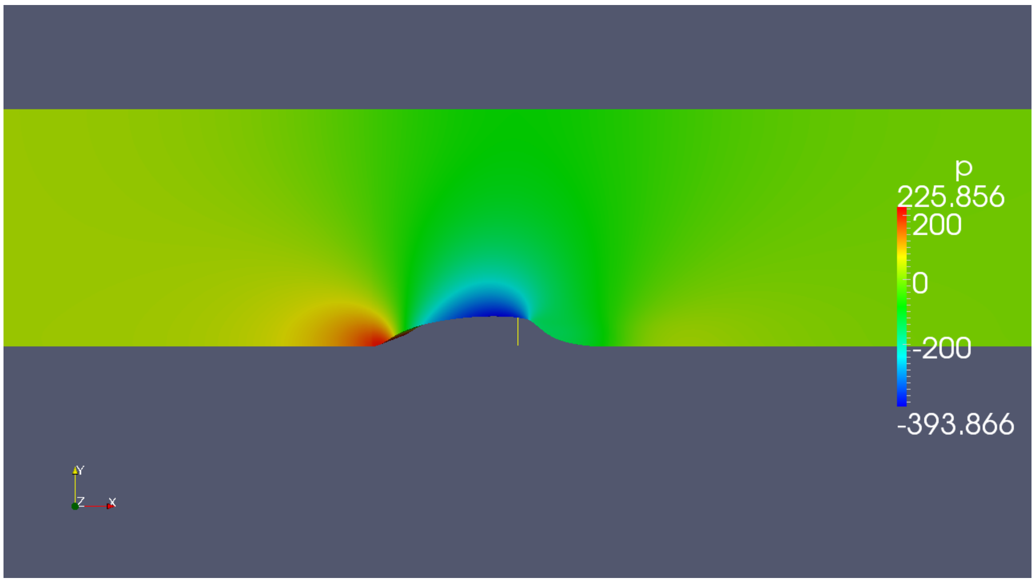


Figure 6.4: $\Delta p/\rho$ distribution over the plane. 2D case. Inlet =-6 x/c. k-ϵ model

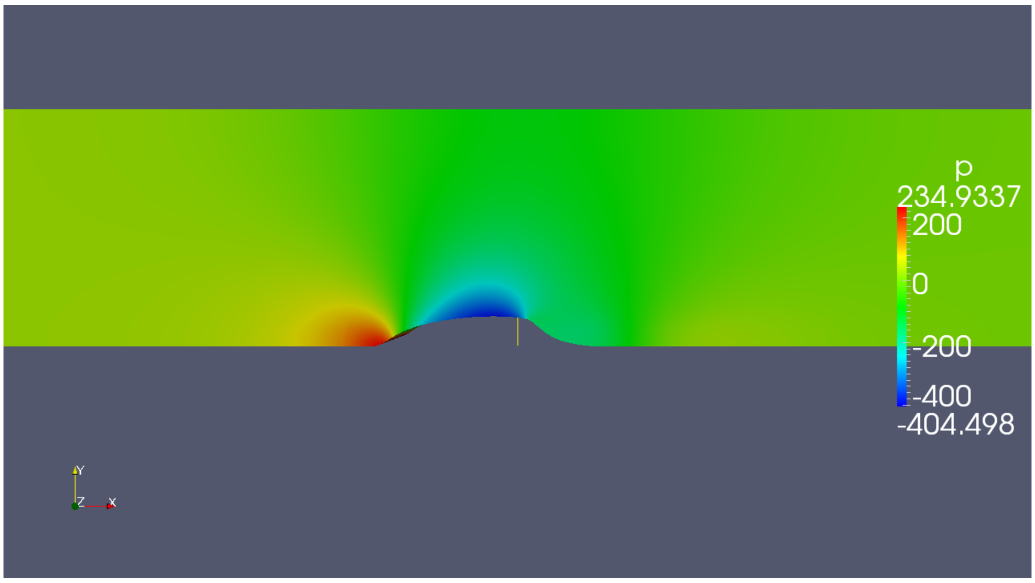


Figure 6.5: $\Delta p/\rho$ distribution over the plane. 2D case. Inlet = -6 x/c. k-ω model

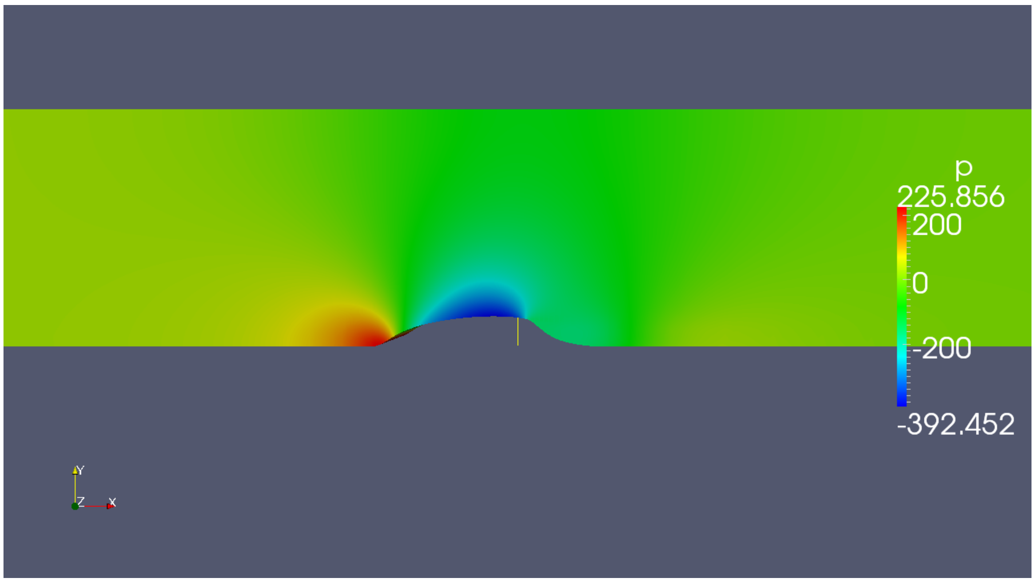


Figure 6.6: $\Delta p/\rho$ distribution over the plane. 2D case. Inlet = -6 x/c. k-ω-SST model

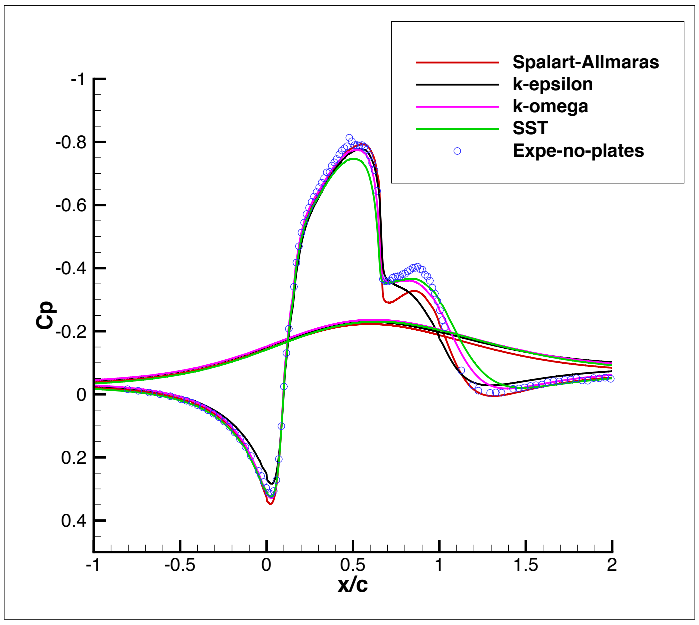


Figure 6.7: Comparison between numerical and experimental pressure coefficients. Theoretical case. Inlet = -6 x/c.

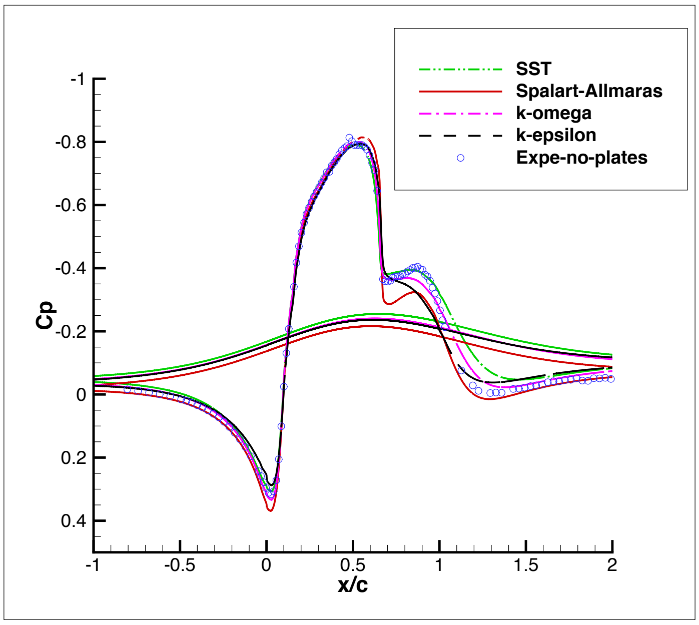


Figure 6.8: Comparison between numerical and experimental pressure coefficients, by taking into account the effects of the upper wall. 2D case. Inlet = -6 x/c.

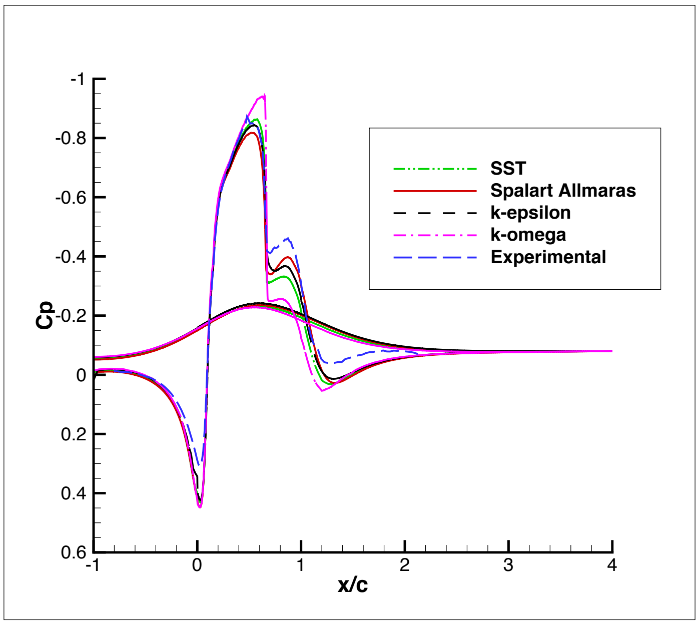


Figure 6.9: Comparison between numerical and experimental pressure coefficients. 2D case.Inlet = -1 x/c.

6.5 Velocity

In figures 6.10, 6.11, 6.12, 6.13 velocity contours are shown for all turbulence models: the cases are 2D and they do not take into account the presence of the upper wall. In figures 6.14, 6.15, 6.16, 6.17, velocity profiles at $x/c = 0.8$, $x/c = 1.0$, $x/c = 1.1$, $x/c = 1.2$ are compared with the experimental results (inlet located at -6 x/c).
Grid origin is at the beginning of the hump, so that 1.0 x/c is the last point of the hump.
In the contour plots, the recirculation bubble (low speed flow marked by blue color) is visible for all of the turbulence models employed. The trend of velocity is the same for all models: the flow accelerates where pressure drops and separates soon after. The size of the bubble may change for different models; this aspect will be discussed in the following section.The most significant results are obtained by comparing the velocity profiles downstream of the hump. Velocity profiles are extracted in the bubble ($x/c = 0.8$ and $x/c = 1.0$), at the experimental reattachment point ($x/c = 1.1$), and outside the bubble ($x/c = 1.2$). The numerical profiles match the experimental results very well at all locations except $x/c = 1.2$ which is a critical location.
As it will be shown in the next section, most turbulence models predict the reattachment point to be very close to $x/c = 1.2$, while the experience suggests $x/c = 1.1$. *Spalart-Allmaras*, $k - \omega$, and $k - \omega - SST$ fail to reproduce the velocity profile at $x/c = 1.2$ close to the wall, while $k - \epsilon$ provides good results probably because it is the most accurate one in predicting the position of the reattachment point [12]. In figure 6.18 the effects of the upper wall on velocity are shown. We can see the boundary layer developing all along the upper wall. This is responsible for the decrease of pressure shown in figure 6.8.
The presence of the upper wall does not affect the bubble and the velocity profiles downstream of

[12]These aspects will be discussed deeper in the next section.

the hump. Figure 6.19 shows the velocity profiles at the critical location $x/c = 1.2$. Their trend is very similar to that obtained by using a symmetry plane as boundary condition for the upper wall (figure 6.17).
To have an idea of the accuracy of these solutions, the velocity profiles collected by Rumsey ($x/c = 1.2$) are shown in figure 6.20. In this case, two methods predict the experimental results with a good accuracy:*AZ-cobalt-des-1-3d* and *META-cfd++lns-3D*. These are two hybrid LES/RANS methods. One of them implements the *DES* method (Detached Eddy Simulation), the other one implements the *LNS* method (Limited Numerical Scales). All the other curves, obtained by using RANS models, are quite far from the experimental results: an evidence is the fact that, in proximity of the wall, velocities are negative, while the experimental curve predicts positive values. This is a consequence of the fact that the RANS simulations used in this workshop over-predicted the length of the bubble, so that the location x/c=1.2 was still inside the recirculation zone.
The good behavior of the k-ϵ method used in our simulations is a consequence of predicting the size of the bubble properly.

6.5.1 Inlet located at $x/c = -1$

In the previous section the influence of the position of the inlet section was mentioned. In this section, the effects that the location of the inlet has on the velocity profiles are discussed.
So far, the inlet has been located -6 x/c upstream of the hump. Figures 6.21, 6.22, 6.23 show the behavior of velocity profiles for the inlet located -1 x/c upstream of the hump.
In this case the boundary layer over the lower wall is thinner than that calculated for the inlet located -6 x/c upstream of the hump: when the flow separates, the wake remains thinner, so that by moving from the lower wall to the upper wall, it can be observed that the velocity profile reaches the external flow velocity faster than experimental results.
This consideration is very important, because it shows that, even if the "hump flow" is not sensitive to different inlet conditions (Reynolds number, Mach number and turbulent parameters do not affect sensitively the solution[13]), the position of the inlet plays an important role in the accuracy of solution.

6.6 Recirculation zone

The size of the bubble is the most important parameter to evaluate how accurate our simulations are. In order to estimate the length of the bubble, both the separation point and the reattachment point have to be calculated.
This is possible by looking at the curve of the skin friction coefficient. When the curve cuts the zero axis the term du/dy is zero as well. It means that the flow separates or reattaches.
In order to get the skin friction coefficient trend, the derivative of velocity along y has to be computed.
This field is not one of the standard outputs in the software, so we need to modify the source code of OpenFOAM by defining the new parameters.
It is now shown how the new parameters have been defined and implemented in order to compute the skin friction and the turbulent shear stress.
We remind the analytical definitions of C_f and $\overline{u'v'}$:

$$C_f = \frac{\tau}{\frac{1}{2}\rho U_\infty^2} = \frac{\mu\frac{\partial \bar{U}}{\partial y}}{\frac{1}{2}\rho U_\infty^2} \tag{6.6}$$

[13]This is shown by Greenblatt in [27]

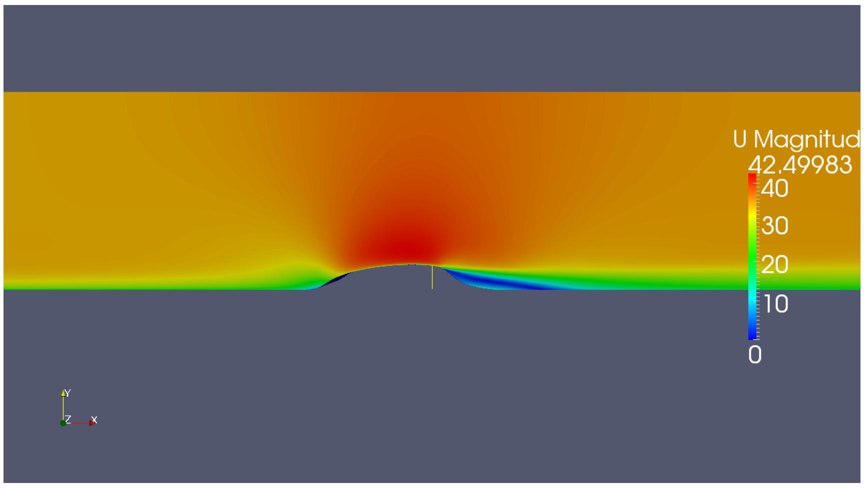


Figure 6.10: Zoom of velocity field in $x-y$ plane. 2D case. Inlet = -6 x/c. Spalart-Allmaras model.

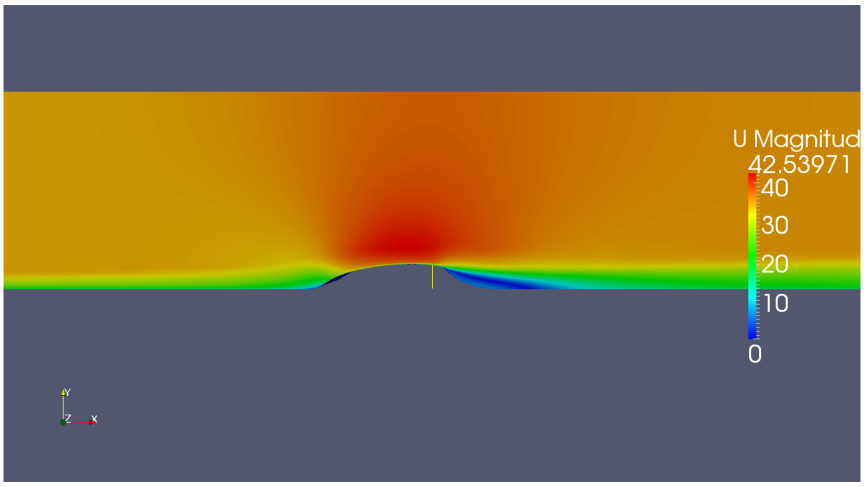


Figure 6.11: Zoom of velocity field in $x-y$ plane. 2D case. Inlet = -6 x/c. k-ϵmodel.

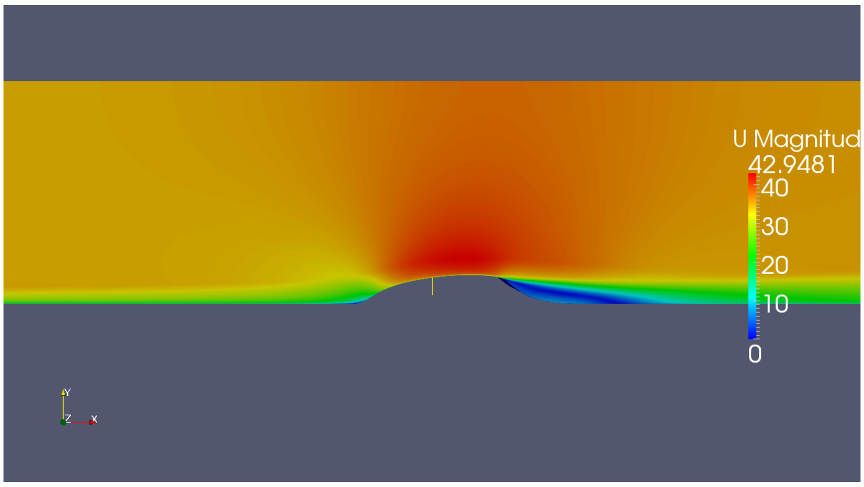


Figure 6.12: Zoom of velocity field in $x - y$ plane. 2D case. Inlet = -6 x/c. k-ω model.

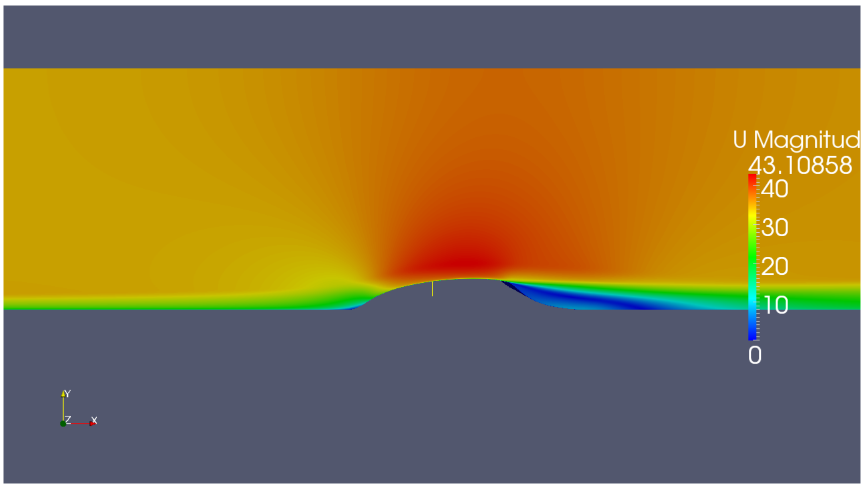


Figure 6.13: Zoom of velocity field in $x - y$ plane. 2D case. Inlet = -6 x/c. k-ω-SST model.

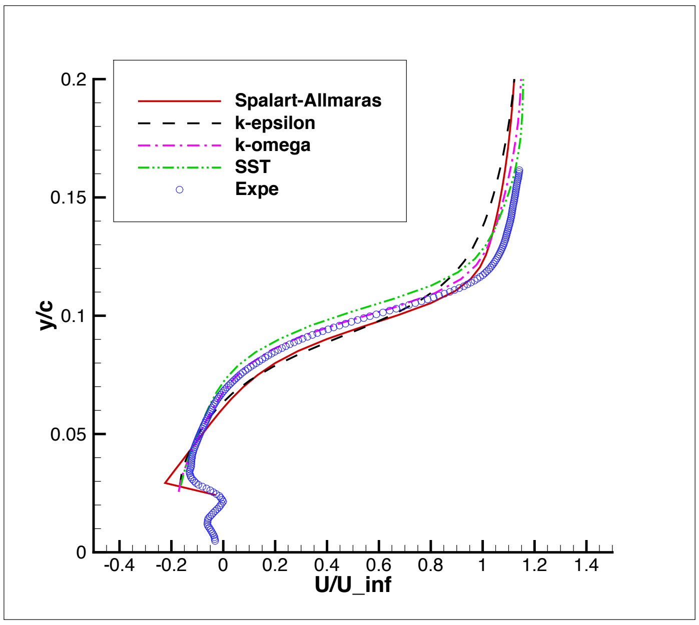


Figure 6.14: Simulated velocity profiles compared to experimental data. x/c=0.8. Inlet located at -6 x/c.

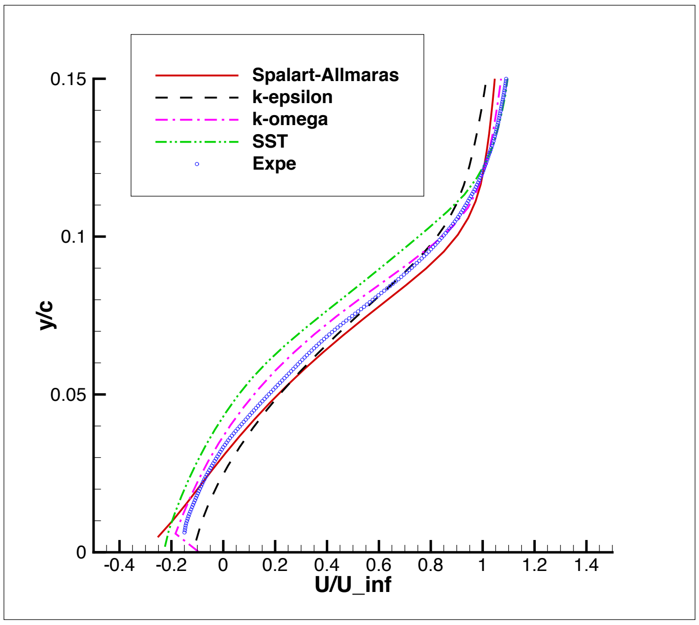


Figure 6.15: Simulated velocity profiles compared to experimental data.x/c=1.0. Inlet located at -6 x/c.

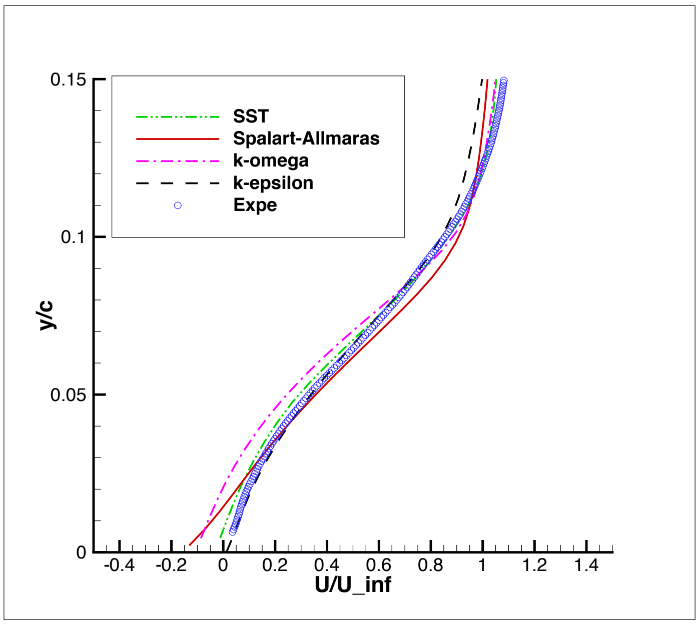


Figure 6.16: Simulated velocity profiles compared to experimental data.x/c=1.1. Inlet located at -6 x/c.

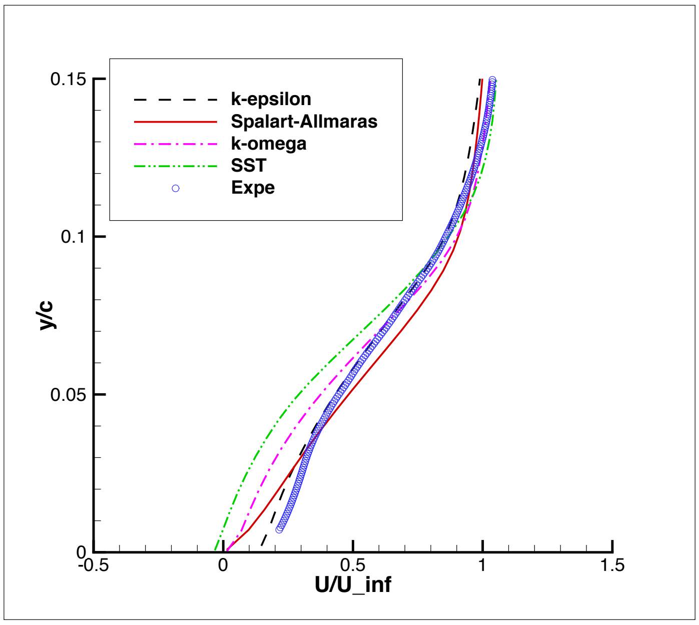


Figure 6.17: Simulated velocity profiles compared to experimental data.x/c=1.2. Inlet located at -6 x/c.

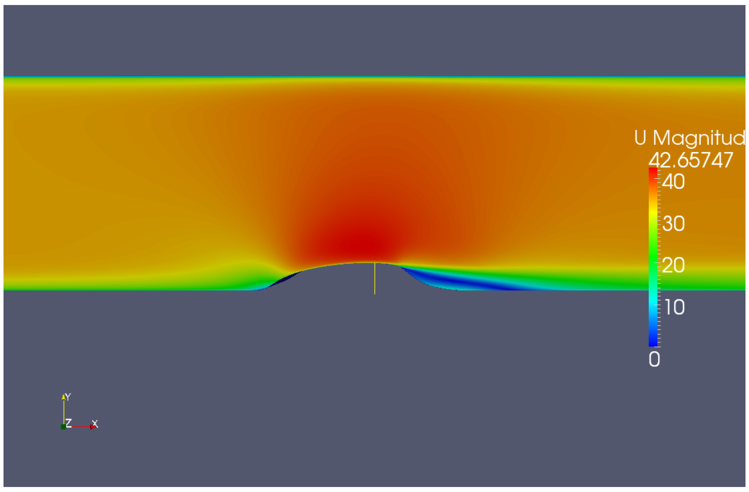


Figure 6.18: Velocity field in the x-y plane, by taking into account the effects of the upper wall. Inlet located at -6x/c.

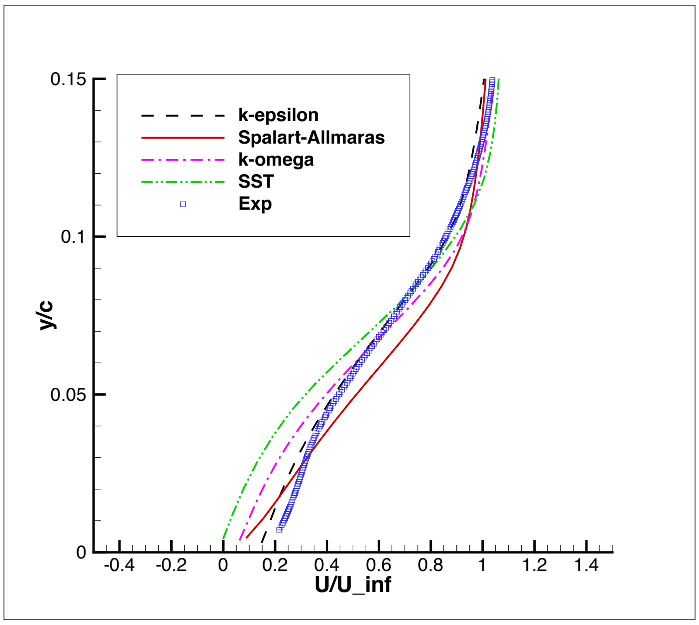


Figure 6.19: Simulated velocity profiles compared to experimental data, in the presence of the upper wall. x/c=1.2. Inlet located at -6 x/c.

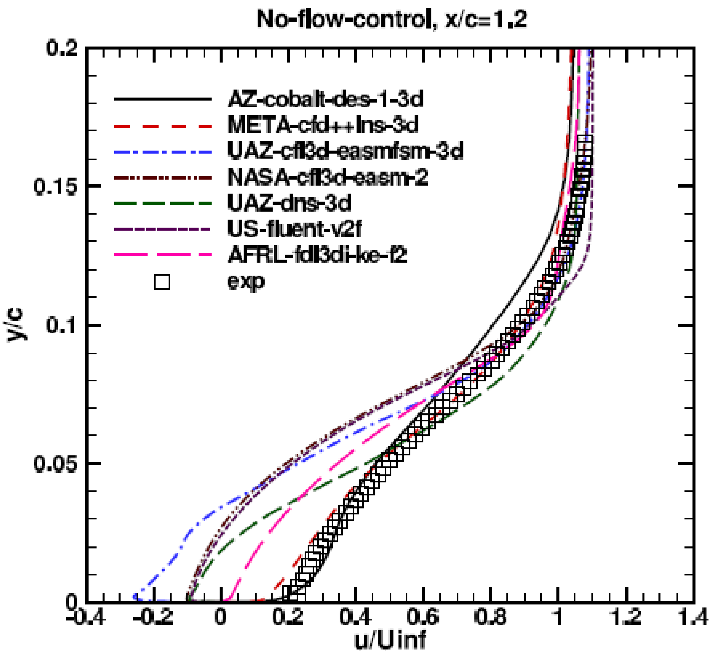


Figure 6.20: Velocity profiles collected by Rumsey. $x/c = 1.2$ [1]

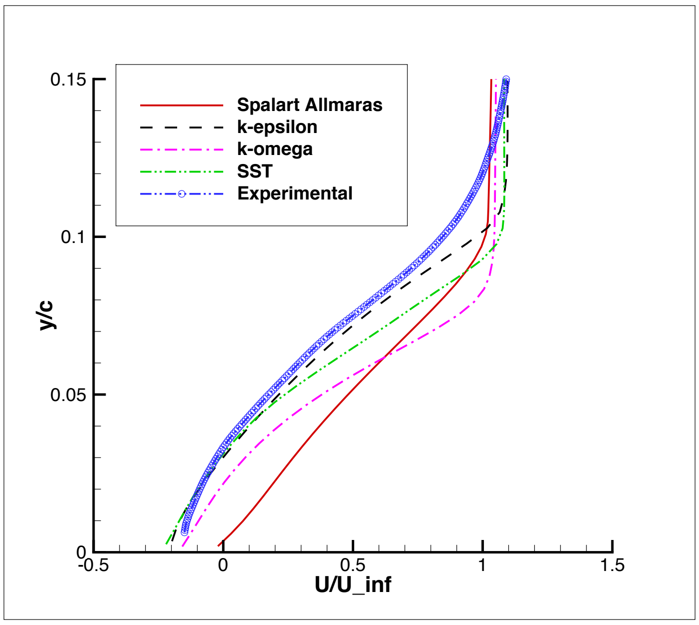


Figure 6.21: Simulated velocity profiles compared to experimental data. x/c=1.0. Inlet located at -1 x/c.

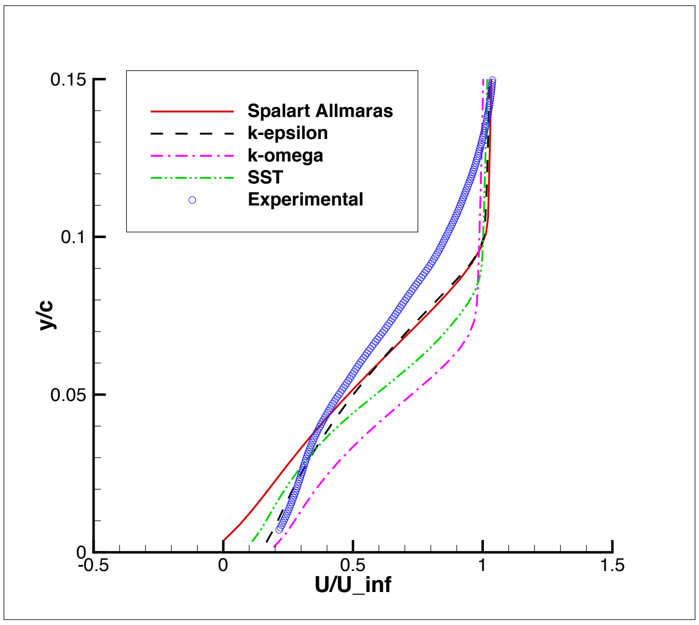


Figure 6.22: Simulated velocity profiles compared to experimental data. x/c=1.2. Inlet located at -1 x/c.

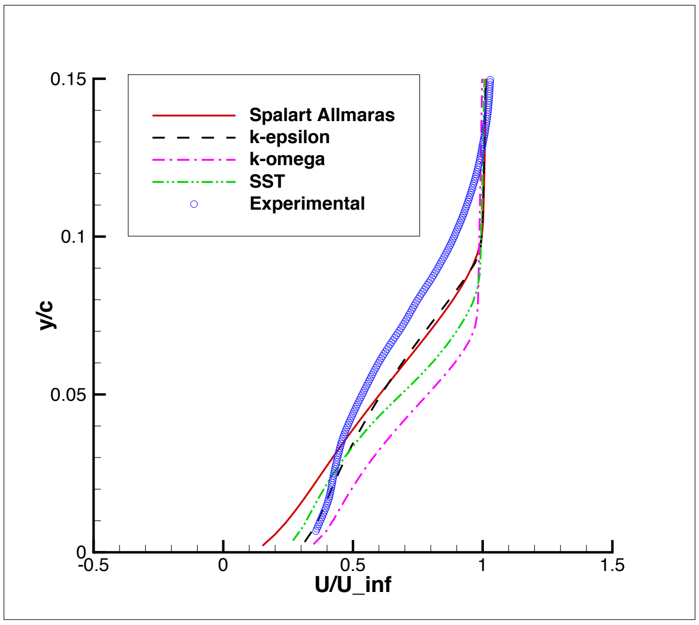


Figure 6.23: Simulated velocity profiles compared to experimental data. x/c=1.3. Inlet located at -1 x/c.

While the turbulent shear stress is given by:

$$\overline{u'v'} = -2\nu_t\overline{S_{ij}} = -\nu_t\left(\frac{\partial\bar{u}}{\partial y} + \frac{\partial\bar{v}}{\partial x}\right) \tag{6.7}$$

Where S_{ij} is the rait-of-strain tensor already defined in the chapter 2.
The following lines represent the modifications to the source code, to obtain these new fields:

Definition of new parameters in the file *createFields.H*

```
  ...

  Info<< "Reading field vectorX\n" << endl;
  vector  vectorX(1, 0, 0); //vector definition

 Info<< "Reading field vectorY\n" << endl;
 vector vectorY(0, 1, 0);

   Info<< "Reading field U_x\n" << endl;
   volScalarField U_x  // Assuming that U_x is a scalar: 'volScalarField'
   (
        IOobject
        (
            "U_x",
            runTime.timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        U.component(vector::X)
            );

   Info<< "Reading field U_y\n" << endl;
   volScalarField U_y
   (
        IOobject
        (
            "U_y",
            runTime.timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        U.component(vector::Y)
   );

Info<< "Reading field mu\n" << endl;
    volScalarField mu
    (
        IOobject
        (
            "mu",
            runTime.timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh
    );


Info<< "Reading field tau\n" << endl;
    volScalarField tau
    (
        IOobject
        (
            "tau",
           runTime.timeName(),
            mesh,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),

       // tau=mu*fvc::grad(U_x)&vectorY
       mesh,
       dimensionedScalar("zero", dimensionSet(1,-1,-2,0,0),0.0)

    );

Info<< "Reading field dU_dx\n" << endl;
    volScalarField dU_dx
    (
        IOobject
        (
            "dU_dx",
           runTime.timeName(),
            mesh,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),

       // tau=mu*fvc::grad(U_x)&vectorY
       mesh,
       dimensionedScalar("zero", dimensionSet(0,0,-1,0,0),0.0)

    );
Info<< "Reading field dU_dy\n" << endl;
    volScalarField dU_dy
    (
        IOobject
        (
            "dU_dy",
           runTime.timeName(),
            mesh,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),

      // tau=mu*fvc::grad(U_x)&vectorY
      mesh,
      dimensionedScalar("zero", dimensionSet(0,0,-1,0,0),0.0)

   );

);
```

The new parameters defined in the *createFields.H* file were then declared and computed in the file *UEqn.H.*

Implementation of τ and $\overline{u'v'}$ in the file *UEqn.H*

```
....
    U_x=U.component(vector::X);
    U_y=U.component(vector::Y);
    dU_dx=fvc::grad(U_y)&vectorX;
    dU_dy=fvc::grad(U_x)&vectorY;
    tau=mu*fvc::grad(U_x)&vectorY;
  uv=-nut*fvc::grad(U_x)&vectorY-nut*fvc::grad(U_y)&vectorX;
 .....
```

Figure 6.24 shows the skin friction coefficient over the hump for the turbulence models used. The upper wall does not affect the value of the skin friction at lower wall, so that these curves are valid for both cases - viscous wall and symmetry plane - with the inlet located at -6 x/c.
By identifying the points where the C_f curves cut the axis of zero, it is possible to locate the separation point and the reattachment point.
As it can seen in table 6.5 and in figure 6.24 the separation point is accurately predicted by all turbulence models, while the only method able to catch the reattachment point is $k-\epsilon$. This explains why the velocity profiles downstream of the reattachment are more accurate for $k-\epsilon$ rather than they are for the other methods.
Figures 6.25, 6.26, 6.27, 6.28 show the size of the bubble computed with the different *RANS* methods. The smallest bubble is predicted by $k-\epsilon$, the biggest by $k-\omega-SST$ as summarized in table 6.5.

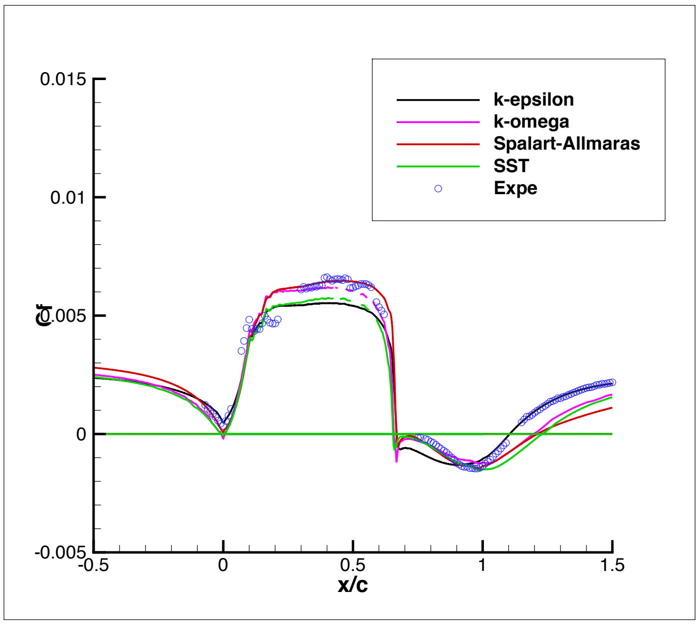


Figure 6.24: Skin friction coefficient. Inlet located at $x/c = -6$

	Separation point [x/c]	**Reattachment point [x/c]**
Spalart Allmaras	0.667	1.198
k-ϵ	0.669	1.104
k-ω	0.656	1.196
k-ω-SST	0.656	1.229
Experiments	0.65-0.67	1.11

Table 6.5: Location of the the separation point and of the reattachment point for the different RANS models.

6.7 Turbulent Shear Stress

As reported in Rumsey's survey, one of the parameters which can mostly affect the prediction of the size of the bubble is the turbulent shear stress. As long as this parameter is not predicted properly, the turbulence in the bubble can be compromised and so its length and size.
Figure 3.9 showed that CFD tends to under predict the turbulent shear stress: the consequence is that the mixing of velocity in the bubble decreases too, by provoking a longer bubble as result.
In figure 6.29, the turbulent shear stress computed by using the algorithm shown above are compared with the experiments. It can be remarked that in our case the turbulent shear stress is under-predicted, too.
This is an intrinsic limit of *RANS* equations which are not modeled to compute separated boundary layers properly.

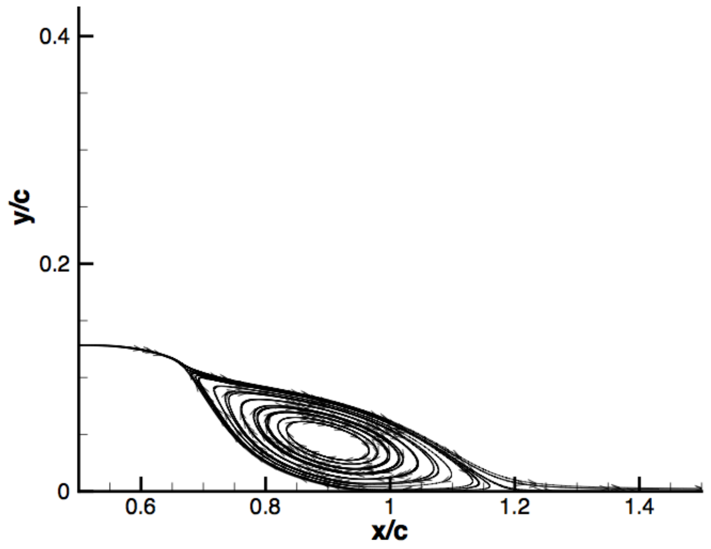


Figure 6.25: Visualization of the bubble in *TecPlot* for the *Spalart-Allmaras* model

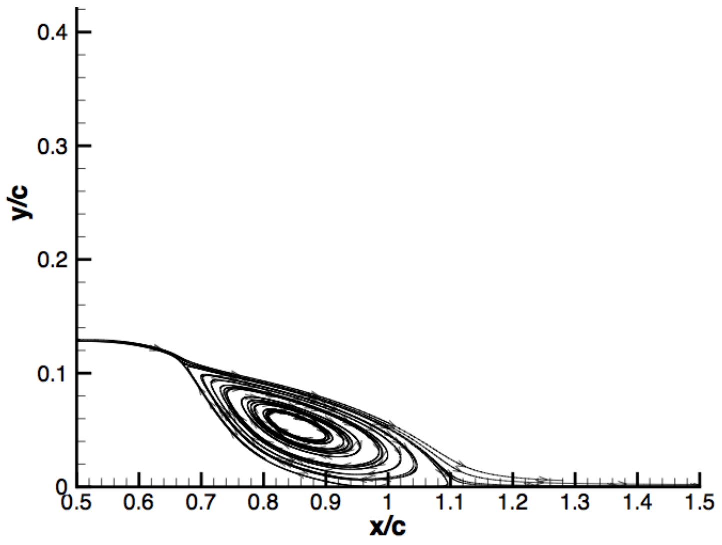


Figure 6.26: Visualization of the bubble in *TecPlot* for the $k - \epsilon$ model

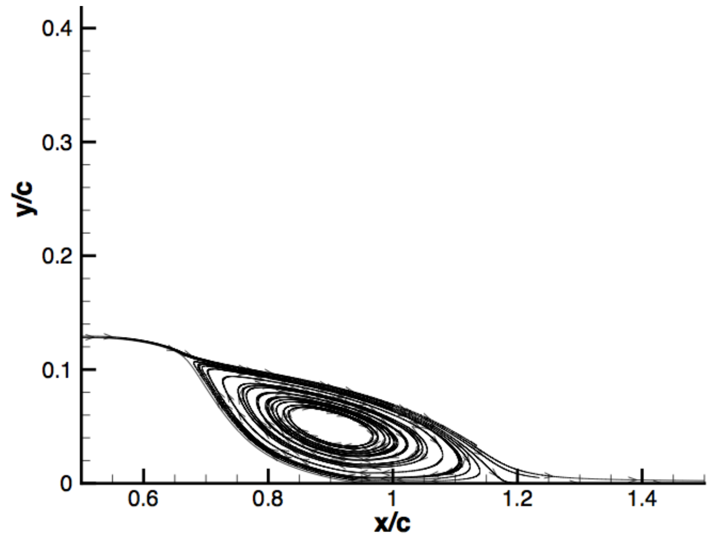


Figure 6.27: Visualization of the bubble in *TecPlot* for the $k-\omega$ model

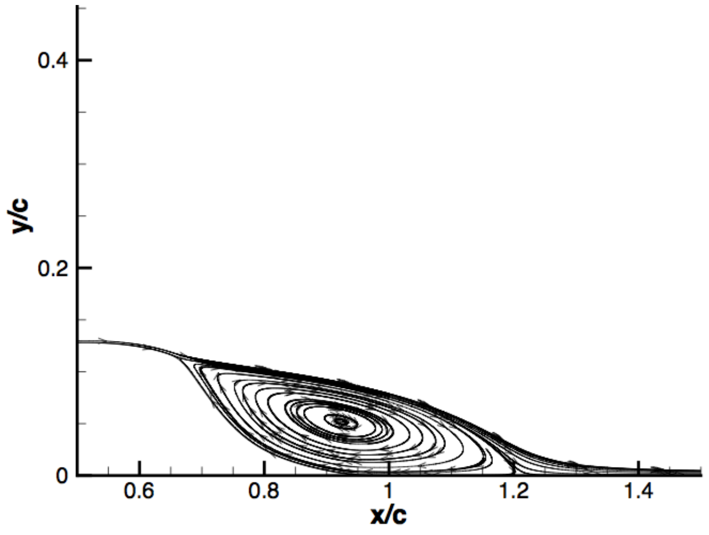


Figure 6.28: Visualization of the bubble in *TecPlot* for the *SST* model

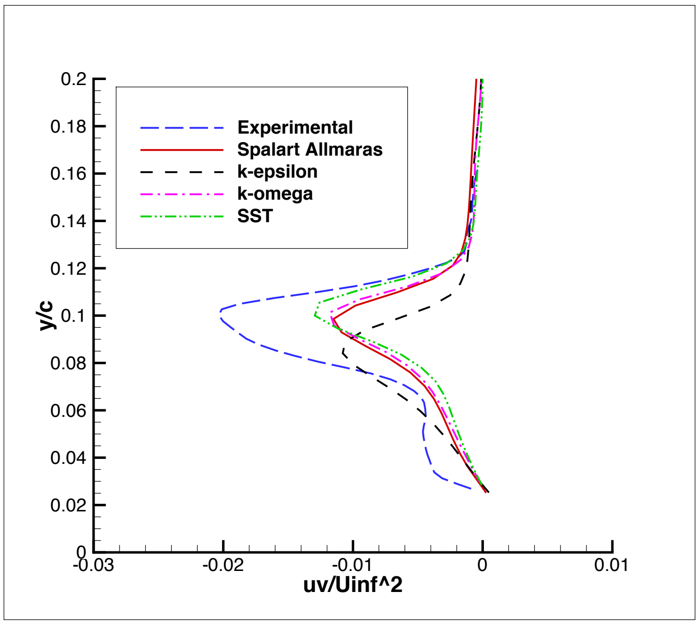


Figure 6.29: Turbulent shear stress at x/c=0.8.

6.8 Effects of the resolution of the Mesh

In section 2.6 the effects of the wall functions have been treated. In order not to compute the velocity profile too close to the wall, where the gradient is very high and the profile too sharp, the boundary conditions are applied somewhere away from the wall.
Unfortunately, in order to calculate the behavior of some parameters, like the skin friction coefficient, which depends on the velocity profile at the wall, the boundary conditions have to be forced as close as possible to the wall.
In order to do that, a very fine mesh close to the wall, in the y direction is required.
Figure 6.30 shows the skin friction coefficient obtained with a mesh of 100 points and scaling of 300 in y direction[14] We can remark how far the computed results are from the experiments.
This problem can affect not only the skin friction but all the others parameters calculated at the wall, like, for example, the pressure coefficient. Figure 6.31 shows the pressure coefficient distribution for a non-scaled mesh (in y direction, 400 equidistant cells have been used). We can remark that both the methods employed ($k-\omega$ and $k-\omega-SST$) fail in predicting pressure. Results are much less accurate than those shown for a mesh scaled at the lower wall (figure 6.7 and 6.8) .

6.9 3D Model

In this section the effect of the end-side plates are taken into account. One of the causes of pressure being over predicted by CFD in Rumsey's first workshop in 2004[1] was the effect of blockage due to the end-side plates used in the experiments.
A set of experimental measurements of pressure was done in the absence of the plates, showing

[14]The number of points and the scale factors chosen for the different cases are listed in table 6.2

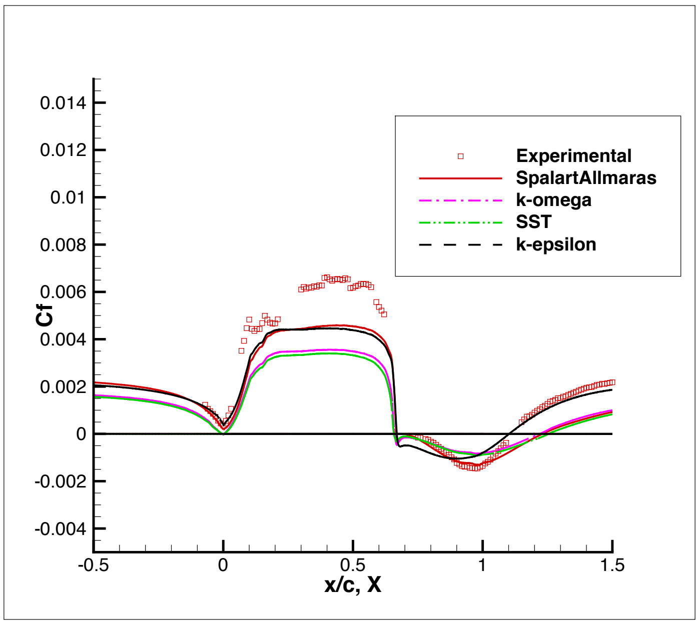


Figure 6.30: Skin friction coefficient. Coarse mesh.

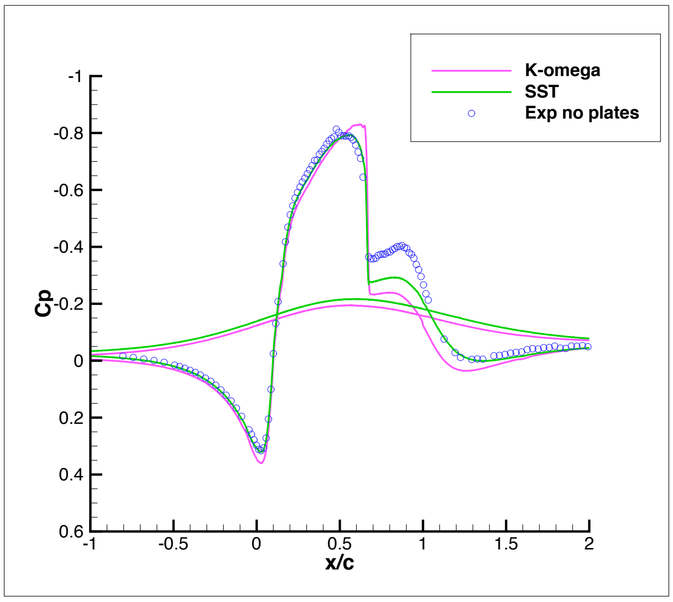


Figure 6.31: Example of pressure coefficient distribution for a coarse mesh.

a significant variation of results.
In order to reproduce the effects of the side-plates, a viscous-wall condition on the lateral faces of the mesh has been used.
The blue zone in figure 6.32 shows the region where velocity has zero value. The goal is to reproduce the 3D model shown in figure 3.1, whereas figure 6.33 gives an idea of the grid used to discretize the mesh.
Figure 6.34 shows the computed pressure coefficients compared to experimental data.
Both experimental measurements of Cp, with and without plates, are shown here.
The effect of blockage does not influence numerical solutions as it does in experiments.
A possible reason could be that, in this work, the end-side plates are modeled as flat plates, so that, for high Reynolds number they do not affect the solution because of the modest thickness of the boundary layer.
In reality other components, like the aluminum frame shown in picture 3.1, could have a major impact on the solution. Another possible reason could be the intrinsic limit of RANS equations in predicting three-dimensional flows.
Figure 6.35 shows the evolution of pressure coefficients for the 2D cases run (the upper wall is modeled as a symmetry plane and as a viscous wall) and for the 3D case, including the end plates.
The effects of the plates are important downstream of the separation point, while they do not affect the solution too much, in the front part of the hump.
This is consistent with the fact that the effects of the boundary layer along the side-plates are more important far downstream of the leading edge of the plate.
The presence of the plates does not affect the solution at the center line, as remarked by Rumsey in his workshop.
At the centerline, the position of the the reattachment point does not change significantly if compared to D results, as shown in figure 6.36. Moreover, velocity profiles have the same trend of the 2D simulations as shown in figure 6.37.
Table 6.6 summarizes the number of processors used and the time and number of iterations required to get the convergence.

	n cells	n Faces	n iter	n procs	time
k-ϵ	11.325.000	34.178.750	18.000	500	20 h

Table 6.6: Number of iterations and time of convergence for the $k - \epsilon$ model. 3D case.

6.10 Launder-Sharma

The hump geometry used in this workshop can be considered as the extrados of a wing airfoil. In this term, the choice of the Reynolds number based on the chord of the hump gives the possibility to compare this flow to that over the airfoil of an aircraft.
Reynolds number is high enough so that *low-Reynolds-number* methods can not be used.
As a proof of that, figure 6.38 shows the coefficient pressure computed by using a low-Reynolds RANS model: *Launder Sharma*. Results seem to be far away from experimental data.

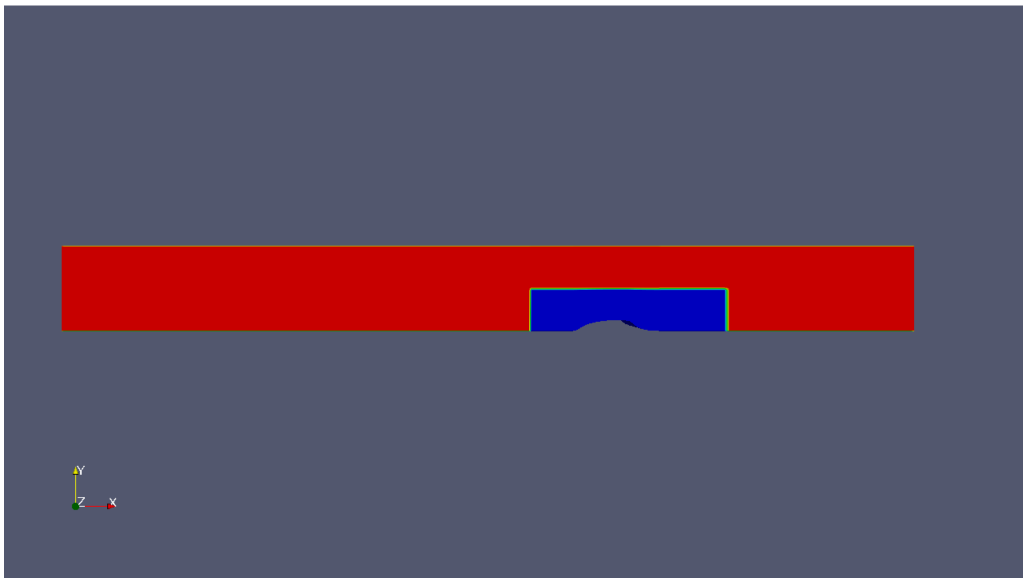


Figure 6.32: Sketch of the mesh used for 3D simulations.

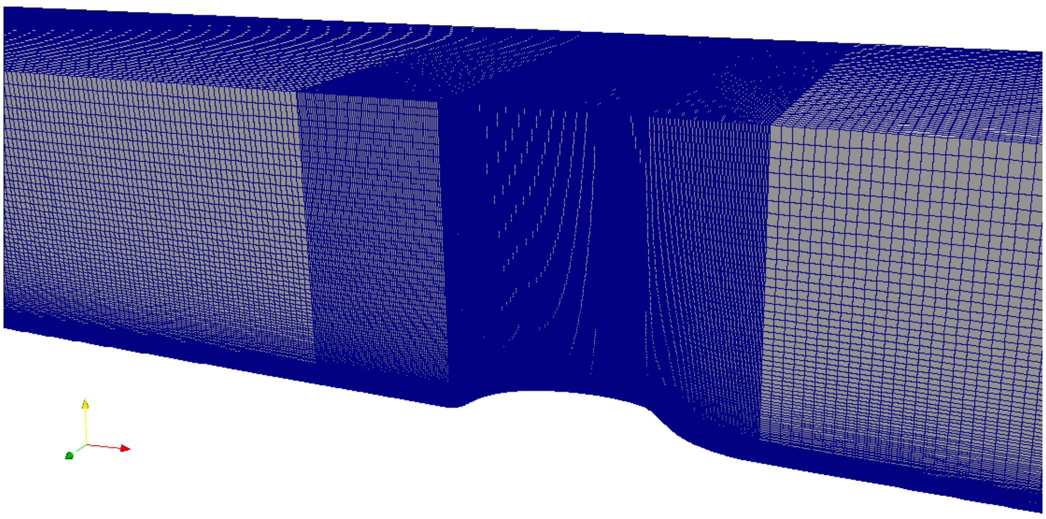


Figure 6.33: Sketch of the grid used for 3D simulations.

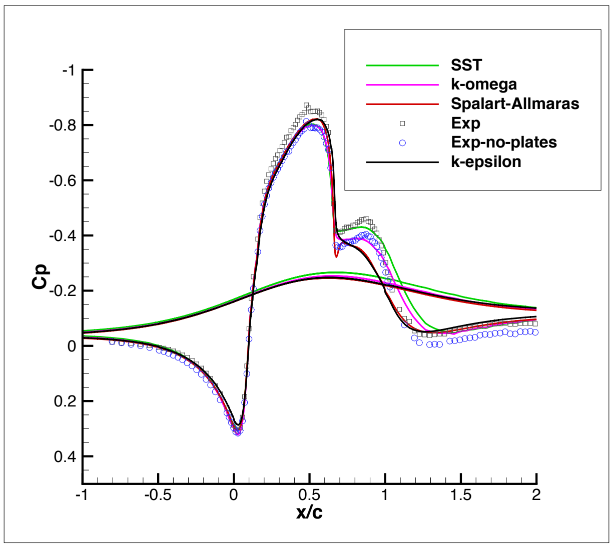


Figure 6.34: Comparison between numerical and experimental pressure coefficients at the centerline. 3D case. Inlet located at -6 x/c.

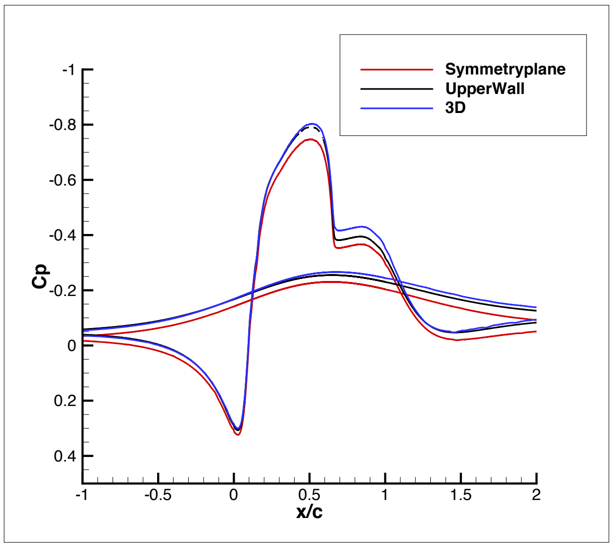


Figure 6.35: Pressure coefficient computed by using the $k - \omega - SST$ method with different boundary conditions: 2D model with upper wall modeled like a symmetry plane, 3D case with upper wall modeled like a viscous wall, 3D case with upper wall and end side plates.

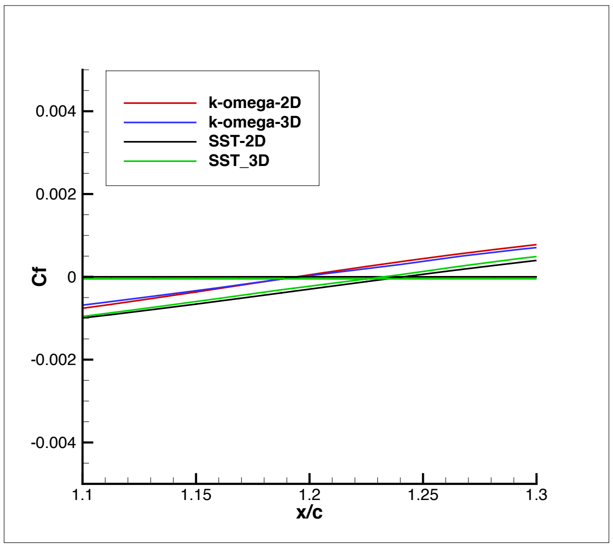


Figure 6.36: Localization of the reattachment point ($Cf = 0$) for 2D and 3D cases, for the methods $k - \omega$ and SST.

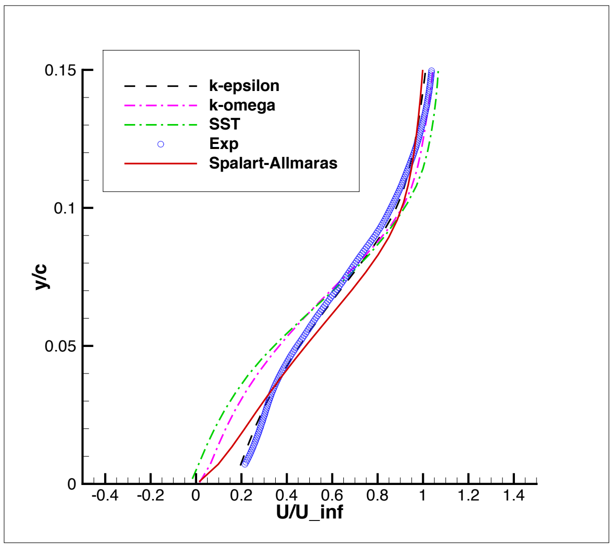


Figure 6.37: Simulated velocity profiles compared to experimental data for the 3D case. x/c=1.2. Inlet located at -6 x/c.

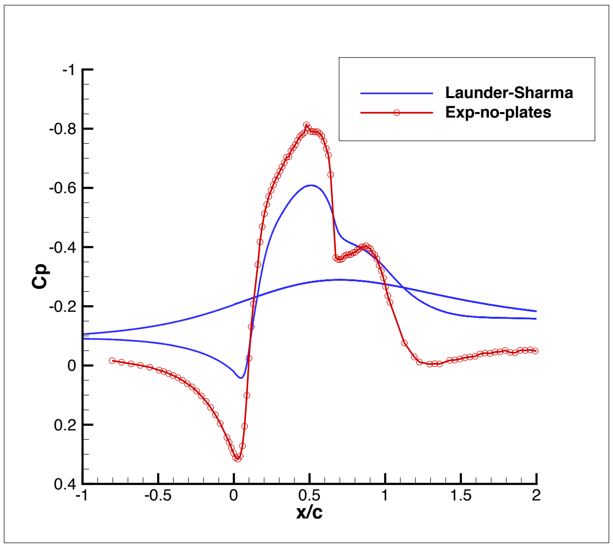


Figure 6.38: Pressure coefficient computed with the Launder-Sharma model.

6.11 Modifications to the k-ϵ model

Since the k-ϵ is the method performing better, some modifications were tried, in order to increase the turbulent shear stress predicted by this method.
For the k-ϵ model, the term $\overline{u'v'}$ is proportional to the constant C_μ, which is chosen to be 0.09 in the standard implementation.
The equation of the turbulent shear stress for the k-ϵ model is recalled:

$$\overline{u'v'} = -2\nu_t S_{ij} = -2C_\mu \frac{k^2}{\epsilon} S_{ij} \tag{6.8}$$

Figure 6.39 shows the trend of the turbulent shear stress when C_μ is increased from the standard value, 0.09, to 0.18. The magnitude of $\overline{u'v'}$ increases, but the physics of the problem is seriously compromised. The length of the separation bubble decreases until it disappears: turbulent viscosity increases, but not at the same location of the experimental data. Turbulent viscosity increases close to the wall and enhances the mixing. This makes the bubble disappear.
Table 6.7 summarizes the location of the reattachment point for the same values of C_μ used in figure 6.39.

C_μ	**0.09**	**0.10**	**0.18**
Reattachment point	1.10	1.08	–

Table 6.7: Location of the reattachment point for different values of C_μ. k-ϵ model.

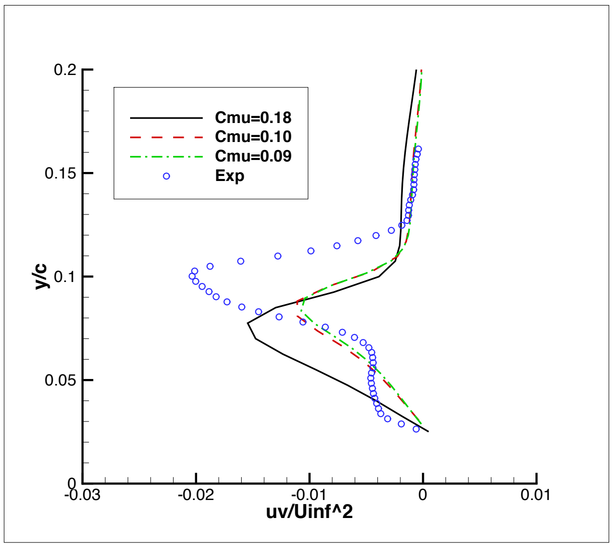


Figure 6.39: Turbulent shear stress for different values of C_μ. k-ϵ model. $x/c = 0.8$

Chapter 7

Conclusion

The goal of this work is to demonstrate that this problem is very sensitive to the procedural variables used to implement the mesh and the boundary conditions and that a complete 3D study can give results that differ from the 2D case because of the blockage introduced by the end-side plates.
The reason for which the k-ϵ model gives very accurate results in terms of pressure coefficients, velocity profiles and position of separation and reattachment points may be in the fact that the turbulent shear stress predicted by this model is quite close to the experimental data in the region close to the wall ($y/c < 0.08$). An evidence of this fact is that, if the magnitude of the turbulent shear stress is increased more, the length of the bubble decreases until it disappears.
In what concerning the other models, the magnitude of the turbulent shear stress is seriously under-predicted in all the domain, so that the mixing in the bubble is not intense enough to determine the exact position of the reattachment point: the length of the bubble is over predicted, as shown by Rumsey in the workshops of 2004 and 2008 .
Even if the hump-flow is considered insensitive to inlet conditions in terms of Reynolds number and Mach number, the development of the boundary layer upstream of the hump strongly affects the velocity profiles downstream. This is evidenced by the fact that by moving the location of the *inlet*, the velocity field computed downstream of the model changes in a substantial way.
Since the Reynolds number influences the development of the boundary layer over the splitter plate, a big variation of the Reynolds number during the experiments could have led to the same consequences, too.
The reason for which particular attention has to be paid to the mesh resolution, is that velocity profile is very sharp at the wall, because of the turbulent nature of the boundary layer. In order to catch the derivatives of velocity properly, the mesh has to be very fine, close to the lower wall. In this work, the mesh is scaled so that the points in proximity of the hump are 300 or 500 times smaller than the free stream points.
A mesh which does not satisfy these requirements could not be able to provide accurate values of pressure, skin friction and other parameters at the wall .
Our results are consistent with those obtained by He et al[9], who used the commercial software *Fluent*, but they could be different from other results, even if computed by using the same software, because of the great sensitiveness of the solution to all of the variables discussed above.

7.0.1 Subsequent work

A complete direct numerical simulation (DNS) is suggested to fully understand the statistics and the physics of the flow over a hump. This work would better help adjust the RANS models, which would provide more accurate results.
This study should also focus on the turbulent shear stress: this quantity may play an important role in mixing and recovery, so a complete understanding of this problem could definitively improve the accuracy of solutions.
As well, the behavior of *OpenFOAM* for other separated flows could be tested: RANS simulations on a 3D hill would be interesting to better understand the performance of the software in different geometries.
Last but not least, other softwares, different from *OpenFOAM*, should be used to run simulations on the same case, with the same settings: this would allow to better evaluate the performance of *OpenFOAM*.

Chapter 8

Annex

8.1 Initial Conditions

The initial and the boundary conditions are in the directory 0. The files shown are those needed to generate the initial conditions for the "Hump" case, when the k-ϵ model is used. They refer to: velocity, pressure, turbulent kinetic energy k, dissipation of turbulent kinetic energy ϵ, turbulent viscosity, ν_t.
These conditions are specified on the boundary of our geometry which we remember to be composed by: a lower wall (our hump), an upper wall (the upper wall of the wind tunnel), an inlet and an outlet.
Initial and boundary conditions for velocity:

```
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (34 0 0);

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform (34 0 0);
    }

    outlet
    {
        type            zeroGradient;
    }

    upperWall
    {
        type            symmetryPlane;
    }

    lowerWall
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }

    frontAndBack
    {
        type            empty;
    }
}

// ************************************************************************* //
```

Initial/boundary conditions for pressure:

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            zeroGradient;
    }

    outlet
    {
        type            fixedValue;
        value           uniform 0;
    }
    upperWall
    {
        type            symmetryPlane;
    }

    lowerWall
    {
        type            zeroGradient;
    }

    frontAndBack
    {
        type            empty;
    }
}

// ************************************************************************* //
```

Initial/boundary conditions for k

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    location    "0";
    object      k;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 0.39;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform 0.39;
    }
    outlet
    {
        type            zeroGradient;
    }
    upperWall
    {
        type            symmetryPlane;
    }
    lowerWall
    {
        type            kqRWallFunction;
        value           uniform 0;
    }
    frontAndBack
    {
        type            empty;
    }
}


// ************************************************************************
```

Initial/boundary conditions for ϵ

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    location    "0";
    object      epsilon;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 2 -3 0 0 0 0];

internalField   uniform 37.4;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform 37.4;
    }
    outlet
    {
        type            zeroGradient;
    }
    upperWall
    {
        type            symmetryPlane;
    }
    lowerWall
    {
        type            epsilonWallFunction;

    }
    frontAndBack
    {
        type            empty;
    }
}


// ************************************************************************* //
```

8.2 Mesh

The description of the mesh is in the *blockMeshDict* file, whose directory is *constant-¿polyMesh*. Since our geometry is curved, we use the *spline* command to force the line connecting two distinct points of our mesh to interpolate all the others. The number of points used for the interpolation is higher, here we only report some of them for reasons of space:

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

convertToMeters 1;
//vertices of our geometry
vertices
(
    (0.000000000e+00     0.000000000e+00 0)           //0         coordinates of the hump
    (2.504898603e-01     1.004583950e-01 0)            //1
    (5.003331531e-01     1.279137016e-01 0) //2
    (6.541569678e-01     1.150131168e-01 0) //3
    (1.000001861e+00     0.000000000e+00 0) //4
    (0.000000000e+00     0.000000000e+00 1) //5
    (2.504898603e-01     1.004583950e-01 1) //6
    (5.003331531e-01     1.279137016e-01 1) //7
    (6.541569678e-01     1.150131168e-01 1) //8
    (1.000001861e+00     0.000000000e+00 1) //9
    (0.000000000e+00     1 0) //10
    (2.504898603e-01     1 0) //11
    (5.003331531e-01     1 0) //12
    (6.541569678e-01     1 0) //13
    (1.000001861e+00     1 0) //14
    (0.000000000e+00     1 1) //15
    (2.504898603e-01     1 1) //16
    (5.003331531e-01     1 1) //17
    (6.541569678e-01     1 1) //18
    (1.000001861e+00     1 1) //19
    (-1 0 0) //20         coordinates of the inlet
    (-1 0 1) //21
    (-1 1 1) //22
    (-1 1 0) //23
    (4 0 0) //24         coordinates of the outlet
    (4 0 1) //25
    (4 1 1)       //26
    (4 1 0) //27
    );

blocks
 //Definition of the blocks and of the number of cells in (x, z, y) direction.
 //One cell in z direction: 2D case
// The grading option allows to specify a variable the degree of finishing of the mesh
 // along one direction
(
    hex (5 6 1 0 15 16 11 10) (125 1 400) simpleGrading (1 1 50)
    hex (6 7 2 1 16 17 12 11) (125 1 400) simpleGrading (1 1 50)
    hex (7 8 3 2 17 18 13 12) (75 1 400) simpleGrading (1 1 50)
    hex (8 9 4 3 18 19 14 13) (175 1 400) simpleGrading (1 1 50)
    hex (9 25 24 4 19 26 27 14) (150 1 400) simpleGrading (1 1 50)
    hex (21 5 0 20 22 15 10 23) (50 1 400) simpleGrading (1 1 50)

);

edges
(
    // The spline allows to connect two point with a curve line,
    // interpolating all the points specified
  spline 0 1 (
    (2.642555636e-04     2.118387155e-05    0)
    (1.196187107e-03     1.013596154e-04    0)
    (2.391164127e-03     2.169787857e-04    0)
    (3.337323718e-02     8.488049619e-03
    (7.975447508e-02     3.539180777e-02   0)
    (8.094403000e-02     3.615196887e-02   0)
    (1.011604723e-01     4.846462459e-02   0)
     (1.692736726e-01     7.810842660e-02  0)
     (2.487983224e-01     1.000949998e-01  0)
            )

   spline 1 2 (
     (2.521801294e-01     1.008182144e-01 0)
     (2.995579493e-01     1.097353663e-01 0)
     (3.926100562e-01     1.219958860e-01 0)
     (4.946946172e-01     1.278379919e-01 0)
     (4.992053078e-01     1.279006964e-01 0)
              )

   spline 2 3 (
       (5.014609990e-01     1.279255533e-01 0)
       (5.195059519e-01     1.279621465e-01 0)
       (5.364229439e-01     1.277221132e-01 0)
       (5.612348549e-01     1.268510549e-01 0)
       (6.007160256e-01     1.239144952e-01 0)
       (6.526311811e-01     1.154531877e-01 0)
       (6.537463389e-01     1.151343734e-01 0)
              )

  spline 3 4 (
       (6.584814913e-01     1.135819998e-01   0)
       (6.670954723e-01     1.096670242e-01 0)
       (6.781899528e-01     1.022275024e-01 0)
       (7.005702715e-01     8.357909926e-02  0)
       (8.505962294e-01     1.225083792e-02  0)
       (9.363485982e-01     1.280259906e-03  0)
       (9.997102679e-01     1.369738752e-07  0)
             )

spline 5 6 (
       (2.642555636e-04     2.118387155e-05    1)
       (4.763002238e-02     1.553935551e-02   1)
       (1.035564236e-01     4.980249228e-02   1)
       (1.046353622e-01     5.038915247e-02   1)
       (2.403368275e-01     9.822725788e-02  1)
       (2.487983224e-01     1.000949998e-01  1)
             )
spline 6 7 (
      (2.521801294e-01     1.008182144e-01 1)
      (3.570851113e-01     1.180858127e-01 1)
      (4.834101633e-01     1.276072870e-01 1)
      (4.992053078e-01     1.279006964e-01 1)
              )
 spline 7 8 (

          (5.477001312e-01     1.274103783e-01 1)
          (5.713927322e-01     1.263019395e-01 1)
          (5.725205781e-01     1.262329486e-01 1)
          (6.503894467e-01     1.160492197e-01 1)
          (6.537463389e-01     1.151343734e-01 1)
              )
spline 8 9 (
         (6.584814913e-01     1.135819998e-01   1)
         (6.694054910e-01     1.083136210e-01 1)
         (6.804952425e-01     1.004385556e-01 1)
         (8.155152479e-01     2.002839700e-02  1)
         (9.077660516e-01     3.870971317e-03  1)
         (9.987028423e-01     2.678174891e-06  1)
         (9.997102679e-01     1.369738752e-07  1)
```

```
             )

);

boundary
   //Definition of the nature of each path (type wall, type inlet, type outlet...)
(
    inlet
    {
        type patch;
        faces
        (
            (20 21 22 23)
        );
    }


    outlet
    {
        type patch;
        faces
        (
            (24 25 26 27)
        );
    }



    lowerWall
    {
        type wall;
        faces
        (
            (5 6 1 0)
            (6 7 2 1)
            (7 8 3 2)
            (8 9 4 3)
            (9 25 24 4)
            (21 5 0 20)

        );
    }


   upperWall
    {
      type patch;
      faces
       (
        (22 15 10 23)
        (15 16 11 10)
        (16 17 12 11)
        (17 18 13 12)
        (18 19 14 13)
        (19 26 27 14)
       );
     }



   frontAndBack
    {
        type empty;
        faces
        (
            (5 6 16 15)
            (6 7 17 16)
            (7 8 18 17)
            (8 9 19 18)
            (9 25 26 19)
            (0 1 11 10)
            (1 2 12 11)
            (2 3 13 12)
            (3 4 14 13)
            (4 24 27 14)
            (21 5 15 22)
            (20 0 10 23)



        );
    }
);

mergePatchPairs
(
);

// ************************************************************************* //
```

8.3 *constant* Directory

In the *constant* directory we also specify the model of turbulence that we want to use (*RASProperties* file)

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      RASProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

RASModel        kEpsilon;

turbulence      on;

printCoeffs     on;
```

and the properties of our flow (we set the appropriate value of viscosity, in order to fix the Reynolds number).

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

transportModel  Newtonian;

nu              nu [ 0 2 -1 0 0 0 0 ] 3.66e-05;

CrossPowerLawCoeffs
{
    nu0             nu0 [ 0 2 -1 0 0 0 0 ] 1e-06;
    nuInf           nuInf [ 0 2 -1 0 0 0 0 ] 1e-06;
    m               m [ 0 0 1 0 0 0 0 ] 1;
    n               n [ 0 0 0 0 0 0 0 ] 1;
}

BirdCarreauCoeffs
{
    nu0             nu0 [ 0 2 -1 0 0 0 0 ] 1e-06;
    nuInf           nuInf [ 0 2 -1 0 0 0 0 ] 1e-06;
    k                  k [ 0 0 1 0 0 0 0 ] 0;
    n                  n [ 0 0 0 0 0 0 0 ] 1;
}
```

8.4 *system* Directory

In the *system* directory, we set the parameters to control our solution, we specify the operators to use to solve the equations and to interpolate the data, and the tolerances on the residual that we want to be respected. This is *controlDict* file:

```
 FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

application     simpleFoam;

startFrom       latestTime;

startTime       0;

stopAt          endTime;

endTime         1000;

deltaT          1;

writeControl    timeStep;

writeInterval   300;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable true;

functions
{
    streamLines
    {
        type            streamLine;

        // Where to load it from (if not already in solver)
        functionObjectLibs ("libfieldFunctionObjects.so");

        // Output every
        outputControl   outputTime;
        // outputInterval 10;

        setFormat       vtk; //gnuplot; //xmgr; //raw; //jplot;

        // Velocity field to use for tracking.
        UName U;

        // Tracked forwards (+U) or backwards (-U)
        trackForward    true;

        // Names of fields to sample. Should contain above velocity field!
        fields (p k U);

        // Steps particles can travel before being removed
        lifeTime        10000;

        // Number of steps per cell (estimate). Set to 1 to disable subcycling.
        nSubCycle 5;

        // Cloud name to use
        cloudName       particleTracks;

        // Seeding method. See the sampleSets in sampleDict.
        seedSampleSet   uniform;  //cloud;//triSurfaceMeshPointSet;

        uniformCoeffs
        {
            type        uniform;
            axis        x;  //distance;

            start       (-0.0205 0.001  0.00001);
            end         (-0.0205 0.0251 0.00001);
            nPoints     10;
        }
    }
}

// ************************************************************************* //
```

This is the *fvSolution* file:

```
 FoamFile
```

```
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{
    p
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0.01;
    }

    U
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-05;
        relTol          0.1;
    }

    k
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-05;
        relTol          0.1;
    }

    epsilon
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-05;
        relTol          0.1;
    }

    R
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-05;
        relTol          0.1;
    }

    nuTilda
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-05;
        relTol          0.1;
    }
}

SIMPLE
{
    nNonOrthogonalCorrectors 0;

    residualControl
    {
        p               1e-2;
        U               1e-3;
        "(k|epsilon|omega)" 1e-3;
    }
}

relaxationFactors
{
    fields
    {
        p               0.3;
    }
    equations
    {
        U               0.7;
        k               0.7;
        epsilon         0.7;
        R               0.7;
        nuTilda         0.7;
    }
}


// ************************************************************************* //
```

The third file contained in the *system* folder is the *fvSchemes* file. Here the numerical methods for each term of the equations are specified as follows:

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.1.1                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

ddtSchemes
{
    default         steadyState;
}

gradSchemes
{
    default         Gauss linear;
    grad(p)         Gauss linear;
    grad(U)         Gauss linear;
}

divSchemes
{
    default         none;
    div(phi,U)      Gauss upwind;
    div(phi,k)      Gauss upwind;
    div(phi,epsilon) Gauss upwind;
    div(phi,R)      Gauss upwind;
    div(R)          Gauss linear;
    div(phi,nuTilda) Gauss upwind;
    div((nuEff*dev(T(grad(U))))) Gauss linear;
}

laplacianSchemes
{
    default         none;
    laplacian(nuEff,U) Gauss linear corrected;
    laplacian((1|A(U)),p) Gauss linear corrected;
    laplacian(DkEff,k) Gauss linear corrected;
    laplacian(DepsilonEff,epsilon) Gauss linear corrected;
    laplacian(DREff,R) Gauss linear corrected;
    laplacian(DnuTildaEff,nuTilda) Gauss linear corrected;
}

interpolationSchemes
```

```
{
    default         linear;
    interpolate(U)  linear;
}

snGradSchemes
{
    default         corrected;
}

fluxRequired
{
    default         no;
    p               ;
}


// ************************************************************************* //
```

If the user wants to run parallel calculation, the number of parts in which the mesh has to be decomposed must be specified. There are three different methods to decompose the mesh:

- *simple*: it decomposes the domain in equal parts according in the directions specified in the corresponding subdictionary simpleCoeffs. For example, do decompose a case in 100 parts in the x direction, n has to be set to (100 1 1). To split the domain into 100 parts, 50 in the x direction and 2 in the y direction, n has to be set to (50 2 1);
- *hierarchical*: it's the same than *simple*, but the user can specify the order according to which the decomposition is done;
- *manual*: it lets the user directly specify the allocation of each cell to a particular processor.

delta is the cell skew factor. It's default value is 0.001 For our simulations, the *simple* method is used.

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      decomposeParDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

numberOfSubdomains 100;
```

```
method       simple;

simpleCoeffs
{
    n               ( 100 1 1 );
    delta           0.001;
}

hierarchicalCoeffs
{
    n               ( 50 2 1 );
    delta           0.001;
    order           xyz;
}

manualCoeffs
{
    dataFile        "";
}

distributed     no;

roots           ( );
```

8.5 Spalart-Allmaras equation

The k-ϵ and k-ω equation are fully treated in the chapter 3. Here the file *SpalartAllmaras.C* is reported for further information about the equation implemented in *OpenFOAM* for this turbulent model.

```
\*---------------------------------------------------------------------------*/

#include "SpalartAllmaras.H"
#include "addToRunTimeSelectionTable.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{
namespace incompressible
{
namespace RASModels
{

// * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //

defineTypeNameAndDebug(SpalartAllmaras, 0);
addToRunTimeSelectionTable(RASModel, SpalartAllmaras, dictionary);
```

```
// * * * * * * * * * * * * Private Member Functions  * * * * * * * * * * * * //

tmp<volScalarField> SpalartAllmaras::chi() const
{
    return nuTilda_/nu();
}


tmp<volScalarField> SpalartAllmaras::fv1(const volScalarField& chi) const
{
    const volScalarField chi3(pow3(chi));
    return chi3/(chi3 + pow3(Cv1_));
}


tmp<volScalarField> SpalartAllmaras::fv2
(
    const volScalarField& chi,
    const volScalarField& fv1
) const
{
    return 1.0/pow3(scalar(1) + chi/Cv2_);
}


tmp<volScalarField> SpalartAllmaras::fv3
(
    const volScalarField& chi,
    const volScalarField& fv1
) const
{
    const volScalarField chiByCv2((1/Cv2_)*chi);

    return
        (scalar(1) + chi*fv1)
       *(1/Cv2_)
       *(3*(scalar(1) + chiByCv2) + sqr(chiByCv2))
       /pow3(scalar(1) + chiByCv2);
}


tmp<volScalarField> SpalartAllmaras::fw(const volScalarField& Stilda) const
{
    volScalarField r
    (
        min
        (
            nuTilda_
           /(
               max
               (
                   Stilda,
                   dimensionedScalar("SMALL", Stilda.dimensions(), SMALL)
               )
              *sqr(kappa_*d_)
            ),
            scalar(10.0)
        )
    );
    r.boundaryField() == 0.0;

    const volScalarField g(r + Cw2_*(pow6(r) - r));

    return g*pow((1.0 + pow6(Cw3_))/(pow6(g) + pow6(Cw3_)), 1.0/6.0);
}


// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //

SpalartAllmaras::SpalartAllmaras
(
    const volVectorField& U,
    const surfaceScalarField& phi,
    transportModel& transport,
    const word& turbulenceModelName,
    const word& modelName
)
:
    RASModel(modelName, U, phi, transport, turbulenceModelName),

    sigmaNut_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "sigmaNut",
            coeffDict_,
            0.66666
        )
    ),
    kappa_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "kappa",
            coeffDict_,
            0.41
        )
    ),
    Cb1_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "Cb1",
            coeffDict_,
            0.1355
        )
    ),
    Cb2_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "Cb2",
            coeffDict_,
            0.622
        )
    ),
    Cw1_(Cb1_/sqr(kappa_) + (1.0 + Cb2_)/sigmaNut_),
    Cw2_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "Cw2",
            coeffDict_,
            0.3
        )
    ),
    Cw3_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "Cw3",
            coeffDict_,
            2.0
        )
    ),
    Cv1_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "Cv1",
            coeffDict_,
            7.1
        )
    ),
    Cv2_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "Cv2",
            coeffDict_,
            5.0
        )
    ),

    nuTilda_
    (
        IOobject
        (
            "nuTilda",
            runTime_.timeName(),
            mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_
    ),

    nut_
    (
        IOobject
        (
            "nut",
            runTime_.timeName(),
            mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_
    ),

    d_(mesh_)
{
    printCoeffs();
}


// * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * //

tmp<volScalarField> SpalartAllmaras::DnuTildaEff() const
{
    return tmp<volScalarField>
    (
        new volScalarField("DnuTildaEff", (nuTilda_ + nu())/sigmaNut_)
    );
}
```

```
tmp<volScalarField> SpalartAllmaras::k() const
{
    WarningIn("tmp<volScalarField> SpalartAllmaras::k() const")
        << "Turbulence kinetic energy not defined for Spalart-Allmaras model. "
        << "Returning zero field" << endl;

    return tmp<volScalarField>
    (
        new volScalarField
        (
            IOobject
            (
                "k",
                runTime_.timeName(),
                mesh_
            ),
            mesh_,
            dimensionedScalar("0", dimensionSet(0, 2, -2, 0, 0), 0)
        )
    );
}


tmp<volScalarField> SpalartAllmaras::epsilon() const
{
    WarningIn("tmp<volScalarField> SpalartAllmaras::epsilon() const")
        << "Turbulence kinetic energy dissipation rate not defined for "
        << "Spalart-Allmaras model. Returning zero field"
        << endl;

    return tmp<volScalarField>
    (
        new volScalarField
        (
            IOobject
            (
                "epsilon",
                runTime_.timeName(),
                mesh_
            ),
            mesh_,
            dimensionedScalar("0", dimensionSet(0, 2, -3, 0, 0), 0)
        )
    );
}
```

```
tmp<volSymmTensorField> SpalartAllmaras::R() const
{
    return tmp<volSymmTensorField>
    (
        new volSymmTensorField
        (
            IOobject
            (
                "R",
                runTime_.timeName(),
                mesh_,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            ((2.0/3.0)*I)*k() - nut()*twoSymm(fvc::grad(U_))
        )
    );
}


tmp<volSymmTensorField> SpalartAllmaras::devReff() const
{
    return tmp<volSymmTensorField>
    (
        new volSymmTensorField
        (
            IOobject
            (
                "devRhoReff",
                runTime_.timeName(),
                mesh_,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
           -nuEff()*dev(twoSymm(fvc::grad(U_)))
        )
    );
}


tmp<fvVectorMatrix> SpalartAllmaras::divDevReff(volVectorField& U) const
{
    const volScalarField nuEff_(nuEff());

    return
    (
      - fvm::laplacian(nuEff_, U)
      - fvc::div(nuEff_*dev(T(fvc::grad(U))))
    );
```

```
}


bool SpalartAllmaras::read()
{
    if (RASModel::read())
    {
        sigmaNut_.readIfPresent(coeffDict());
        kappa_.readIfPresent(coeffDict());

        Cb1_.readIfPresent(coeffDict());
        Cb2_.readIfPresent(coeffDict());
        Cw1_ = Cb1_/sqr(kappa_) + (1.0 + Cb2_)/sigmaNut_;
        Cw2_.readIfPresent(coeffDict());
        Cw3_.readIfPresent(coeffDict());
        Cv1_.readIfPresent(coeffDict());
        Cv2_.readIfPresent(coeffDict());

        return true;
    }
    else
    {
        return false;
    }
}


void SpalartAllmaras::correct()
{
    RASModel::correct();

    if (!turbulence_)
    {
        // Re-calculate viscosity
        nut_ = nuTilda_*fv1(this->chi());
        nut_.correctBoundaryConditions();

        return;
    }

    if (mesh_.changing())
    {
        d_.correct();
    }

    const volScalarField chi(this->chi());
    const volScalarField fv1(this->fv1(chi));

    const volScalarField Stilda
    (
        fv3(chi, fv1)*::sqrt(2.0)*mag(skew(fvc::grad(U_)))
      + fv2(chi, fv1)*nuTilda_/sqr(kappa_*d_)
    );

    tmp<fvScalarMatrix> nuTildaEqn
    (
        fvm::ddt(nuTilda_)
      + fvm::div(phi_, nuTilda_)
      - fvm::Sp(fvc::div(phi_), nuTilda_)
      - fvm::laplacian(DnuTildaEff(), nuTilda_)
      - Cb2_/sigmaNut_*magSqr(fvc::grad(nuTilda_))
     ==
        Cb1_*Stilda*nuTilda_
      - fvm::Sp(Cw1_*fw(Stilda)*nuTilda_/sqr(d_), nuTilda_)
    );

    nuTildaEqn().relax();
    solve(nuTildaEqn);
    bound(nuTilda_, dimensionedScalar("0", nuTilda_.dimensions(), 0.0));
    nuTilda_.correctBoundaryConditions();

    // Re-calculate viscosity
    nut_.internalField() = fv1*nuTilda_.internalField();
    nut_.correctBoundaryConditions();
}


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

} // End namespace RASModels
} // End namespace incompressible
} // End namespace Foam

// ************************************************************************* //
```

8.6 k-ω-SST equation

In this section we report the file *kOmegaSST.C* containing the source code of the SST turbulent model.

```
\*---------------------------------------------------------------------------*/

#include "kOmegaSST.H"
#include "addToRunTimeSelectionTable.H"

#include "backwardsCompatibilityWallFunctions.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

```
namespace Foam
{
namespace incompressible
{
namespace RASModels
{

// * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //

defineTypeNameAndDebug(kOmegaSST, 0);
addToRunTimeSelectionTable(RASModel, kOmegaSST, dictionary);

// * * * * * * * * * * * * Private Member Functions  * * * * * * * * * * * * //

tmp<volScalarField> kOmegaSST::F1(const volScalarField& CDkOmega) const
{
    tmp<volScalarField> CDkOmegaPlus = max
    (
        CDkOmega,
        dimensionedScalar("1.0e-10", dimless/sqr(dimTime), 1.0e-10)
    );

    tmp<volScalarField> arg1 = min
    (
        min
        (
            max
            (
                (scalar(1)/betaStar_)*sqrt(k_)/(omega_*y_),
                scalar(500)*nu()/(sqr(y_)*omega_)
            ),
            (4*alphaOmega2_)*k_/(CDkOmegaPlus*sqr(y_))
        ),
        scalar(10)
    );

    return tanh(pow4(arg1));
}

tmp<volScalarField> kOmegaSST::F2() const
{
    tmp<volScalarField> arg2 = min
    (
        max
        (
            (scalar(2)/betaStar_)*sqrt(k_)/(omega_*y_),
            scalar(500)*nu()/(sqr(y_)*omega_)
        ),
        scalar(100)
    );

    return tanh(sqr(arg2));
}


// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * //

kOmegaSST::kOmegaSST
(
    const volVectorField& U,
    const surfaceScalarField& phi,
    transportModel& transport,
    const word& turbulenceModelName,
    const word& modelName
)
:
    RASModel(modelName, U, phi, transport, turbulenceModelName),

    alphaK1_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "alphaK1",
            coeffDict_,
            0.85034
        )
    ),
    alphaK2_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "alphaK2",
            coeffDict_,
            1.0
        )
    ),
    alphaOmega1_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "alphaOmega1",
            coeffDict_,
            0.5
        )
    ),
    alphaOmega2_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "alphaOmega2",
            coeffDict_,
            0.85616
        )
    ),
    gamma1_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "gamma1",
            coeffDict_,
            0.5532
        )
    ),
    gamma2_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "gamma2",
            coeffDict_,
            0.4403
        )
    ),
    beta1_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "beta1",
            coeffDict_,
            0.075
        )
    ),
    beta2_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "beta2",
            coeffDict_,
            0.0828
        )
    ),
    betaStar_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "betaStar",
            coeffDict_,
            0.09
        )
    ),
    a1_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "a1",
            coeffDict_,
            0.31
        )
    ),
    c1_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "c1",
            coeffDict_,
            10.0
        )
    ),

    y_(mesh_),

    k_
    (
        IOobject
        (
            "k",
            runTime_.timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        autoCreateK("k", mesh_)
    ),
    omega_
    (
        IOobject
        (
            "omega",
            runTime_.timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        autoCreateOmega("omega", mesh_)
    ),
    nut_
    (
        IOobject
        (
            "nut",
            runTime_.timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        autoCreateNut("nut", mesh_)
    )
{
    bound(k_, kMin_);
    bound(omega_, omegaMin_);

    nut_ =
    (
        a1_*k_
      / max
        (
            a1_*omega_,
            F2()*sqrt(2.0)*mag(symm(fvc::grad(U_)))
        )
    );
    nut_.correctBoundaryConditions();

    printCoeffs();
}


// * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * //

tmp<volSymmTensorField> kOmegaSST::R() const
{
    return tmp<volSymmTensorField>
    (
        new volSymmTensorField
        (
            IOobject
            (
                "R",
                runTime_.timeName(),
                mesh_,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            ((2.0/3.0)*I)*k_ - nut_*twoSymm(fvc::grad(U_)),
            k_.boundaryField().types()
        )
    );
}


tmp<volSymmTensorField> kOmegaSST::devReff() const
{
    return tmp<volSymmTensorField>
    (
        new volSymmTensorField
        (
            IOobject
            (
                "devRhoReff",
                runTime_.timeName(),
                mesh_,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
           -nuEff()*dev(twoSymm(fvc::grad(U_)))
        )
    );
}


tmp<fvVectorMatrix> kOmegaSST::divDevReff(volVectorField& U) const
{
    return
    (
      - fvm::laplacian(nuEff(), U)
      - fvc::div(nuEff()*dev(T(fvc::grad(U))))
    );
}


bool kOmegaSST::read()
{
    if (RASModel::read())
    {
        alphaK1_.readIfPresent(coeffDict());
        alphaK2_.readIfPresent(coeffDict());
        alphaOmega1_.readIfPresent(coeffDict());
        alphaOmega2_.readIfPresent(coeffDict());
        gamma1_.readIfPresent(coeffDict());
        gamma2_.readIfPresent(coeffDict());
        beta1_.readIfPresent(coeffDict());
        beta2_.readIfPresent(coeffDict());
        betaStar_.readIfPresent(coeffDict());
        a1_.readIfPresent(coeffDict());
        c1_.readIfPresent(coeffDict());
        return true;
    }
    else
    {
        return false;
    }
}


void kOmegaSST::correct()
{
    RASModel::correct();

    if (!turbulence_)
    {
        return;
    }

    if (mesh_.changing())
    {
        y_.correct();
    }

    const volScalarField S2(2*magSqr(symm(fvc::grad(U_))));
    volScalarField G("RASModel::G", nut_*S2);

    // Update omega and G at the wall
    omega_.boundaryField().updateCoeffs();

    const volScalarField CDkOmega
    (
        (2*alphaOmega2_)*(fvc::grad(k_) & fvc::grad(omega_))/omega_
    );

    const volScalarField F1(this->F1(CDkOmega));

    // Turbulent frequency equation
    tmp<fvScalarMatrix> omegaEqn
    (
        fvm::ddt(omega_)
      + fvm::div(phi_, omega_)
      - fvm::Sp(fvc::div(phi_), omega_)
      - fvm::laplacian(DomegaEff(F1), omega_)
     ==
        gamma(F1)*S2
      - fvm::Sp(beta(F1)*omega_, omega_)
      - fvm::SuSp
        (
            (F1 - scalar(1))*CDkOmega/omega_,
            omega_
        )
    );

    omegaEqn().relax();

    omegaEqn().boundaryManipulate(omega_.boundaryField());

    solve(omegaEqn);
    bound(omega_, omegaMin_);

    // Turbulent kinetic energy equation
    tmp<fvScalarMatrix> kEqn
    (
        fvm::ddt(k_)
      + fvm::div(phi_, k_)
      - fvm::Sp(fvc::div(phi_), k_)
      - fvm::laplacian(DkEff(F1), k_)
     ==
        min(G, c1_*betaStar_*k_*omega_)
      - fvm::Sp(betaStar_*omega_, k_)
    );

    kEqn().relax();
    solve(kEqn);
    bound(k_, kMin_);


    // Re-calculate viscosity
    nut_ = a1_*k_/max(a1_*omega_, F2()*sqrt(S2));
    nut_.correctBoundaryConditions();
}


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

} // End namespace RASModels
} // End namespace incompressible
} // End namespace Foam

// ************************************************************************* //
```

Bibliography

[1] C.L. Rumsey, T.B. Gatski, W.L. Sellers, V.N. Vatsa, and S.A. Viken. "summary of the 2004 cfd validation workshop on synthetic jets and turbulent separation control". *Nasa Langley Research Center, AIAA 2005-1270*, 2005.

[2] Christopher L. Rumsey. "successes and challenges for flow control simulations". *Nasa Langley Research Center - AIAA 2008-4311*, 2008.

[3] David Greenblatt, Keith B. Paschal, Chung-Sheng Yao, Jerome Harris, Norman Schaeffler, and Anthony E. Washburn. "a separation control cfd validation test case. part 1". *AIAA 2004-2220*, 2004.

[4] Stephen B. Pope. *Turbulent Flows*, chapter 10, pages 358–351. Cambridge University Press, 2000.

[5] N.N. Mansour, J. Kim, and P. Moin. "near-wall k-ϵ turbulence modeling". *AIAA Journal, Vol.27, No.8*, 1989.

[6] R. Moser, J. Kim, and P. Moin. "turbulent statistics in fully developed channel flow at low reynolds number". *Journal of Fluid Mechanics. Vol 177*, 1987.

[7] P.Balakumar. "computations of flow over a hump model using higher order method with turbulence modeling". *Nasa Langley Research Center, AIAA 2004-2220*, 2004.

[8] D. You, M. Wang, and P.Moin. "large eddy simulation of flow over a wall-mounted hump with separation control". *Center for Turbulence Research*, 2005.

[9] Chuan He and Thomas C. Corke. "numerical and experimental analysis of plasma flow control over a hump model". *University of Notre Drame, IN*, 2007.

[10] OpenFOAM foundation. "www.openfoam.org/doc". *User Guide*, 2012.

[11] C. Bettini and C Cravero. "computational analysis of flow separation control for the flow over a wall-mountedhump using a synthetic jet". *AIAA Paper 2007-0516*, 2007.

[12] Maurizio Quadrio. *Slides of the course "Turbulence"*. Politecnico di Milano, 2009.

[13] Stephen B. Pope. *Turbulent Flows*, chapter 4. Cambridge University Press, 2000.

[14] Arturo Baron. *Notes of the course "Fluid Dynamics"*. http:
www.aero.polimi.it, 2009.

[15] V.C. Patel, W. Rodi, and G. Schuerer. "turbulence models for near-wall and low reynolds number". *AIAA Journal, Vol.33*, 1985.

[16] David C. Wilcox. *Turbulence Modeling for CFD.* DCW Industries, third edition, 2010.

[17] David C. Wilcox. *Turbulence Modeling for CFD*, chapter 4, pages 180–182. DCW Industries, third edition, 2010.

[18] David C. Wilcox. *Turbulence Modeling for CFD*, chapter 4.11, pages 227–229. DCW Industries, third edition, 2010.

[19] Stephen B. Pope. *Turbulent Flows*, chapter 11, pages 460–462. Cambridge University Press, 2000.

[20] http://cfdval2004.larc.nasa.gov/. "cfd validation of synthetic jets and turbulent separation control". *NASA Langley Research Center.*

[21] Franjo Juretic. "it error analysis in finite volume cfd". 2004.

[22] Jasak H. "error analysis and estimation for the finite volume method with applications to fluid flows". *PhD Thesis, Imperial College*, 1996.

[23] Eric Furbo. "evaluation of rans turbulence models for flow problems with significant impact of boundary layers". *M. Sc. THesis, Uppsala Universitet*, 2010.

[24] High-End Computing Capability. "http://www.nas.nasa.gov/hecc/resources/pleiades.html". *NASA Ames Research Center*, 2012.

[25] Inc. Tecplot. "getting started manual". *tecplot360 user Guide*, 2012.

[26] Platteeuw P.D.A. "application of the probabilistic collocation method to uncertainty in turbulence models". *M. Sc. THesis*, 2008.

[27] David Greenblatt, Keith B.Paschal, Chung-Sheng Yo, Jerome Harris, Norman W. Schaeffler, and Anthony E. Washburn. "a separation control cfd validetion test case". *AIAA 2004-2220*, 2004.