Verification of Statistical Turbulence Models in Aerodynamic Flows

Sebastian Gomez

Sebastian Gomez

Candidate

Mechanical Engineering

Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

Dr. Svetlana V. Poroseva, Chairperson

Dr. Charles R. Truman

Dr. Peter Vorobieff

VERIFICATION OF STATISTICAL TURBULENCE MODELS IN AERODYNAMIC FLOWS

by

SEBASTIAN GOMEZ

B.S., MECHANICAL ENGINEERING, UNIVERSITY OF NEW MEXICO, 2012

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science

Mechanical Engineering

The University of New Mexico
Albuquerque, New Mexico

July 2014

ACKNOWLEDGEMENTS

I would like to thank my academic advisor, Dr. Svetlana Poroseva and Elsa Maria Castillo, at the School of Engineering, for their support. The completion of my studies would not have been possible with out their help. I would also like to acknowledge my friends and family for being patient and understanding during the last couple years. Most importantly, I want to thank Sweet Baby Jesus for enabling me to complete this masterpiece.

VERIFICATION OF STATISTICAL TURBULENCE MODELS IN AERODYNAMIC FLOWS

by

Sebastian Gomez

B.S., Mechanical Engineering, University of New Mexico, 2012
M.S., Mechanical Engineering, University of New Mexico, 2014

ABSTRACT

Computational fluid dynamics (CFD) is a tool that is commonly used in industry and academia. Engineers and scientists are sometimes apprehensive about the use of CFD due to inconsistencies and/or errors in results obtained with different software packages for the same flow cases. As a result, efforts are being made to ensure that there is uniformity among results of flow simulations produced by the computer programs.

The current research makes a contribution to the verification of an open-source CFD toolbox known as OpenFOAM. In doing so, flow results for two benchmark flow cases obtained with OpenFOAM are compared with the results obtained with high-accuracy NASA CFD codes CFL3D and FUN3D. The benchmark cases are the zero pressure gradient boundary layer of flow over a flat plate and a two-dimensional bump in a channel. A number of flow profiles obtained with NASA's definitions of "standard" versions of the Spalart-Allmaras, Shear Stress Transport, and k-ω turbulence models are compared with their CFL3D and FUN3D counterparts. A grid convergence study is performed to measure the change in the results as a function of element size, specifically for the finest meshes.

The flows' mean velocity, skin friction coefficient, and turbulent variable profiles obtained with OpenFOAM are in agreement with NASA's profiles for both cases. The grid convergence studies show that the differences between OpenFOAM and NASA results are found to be of less than 5% for all variables on the finest meshes in both benchmark cases. OpenFOAM's capability to produce accurate results for the benchmark cases is confirmed.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. Introduction

The advance of computers has led to an increase in the use of computational predictions of turbulent fluid behavior in engineering. Computers are used to solve the Navier-Stokes equations, which describe the motion of a fluid. Despite of the computing improvement that has occurred over the past couple of decades, the equations describing the flow field in engineering applications cannot be solved exactly in a computer. The reason for this is that the random fluctuations associated with turbulent flows in engineering applications vary over a large range of time and space scales, which make obtaining an exact numerical solution a very computationally demanding task. A few applications of interest include turbulent flows around vehicles, inside of turbines or in manufacturing methods.

Researchers and engineers are still expected to provide estimations related to fluid flow in a timely manner with the computational resources that are currently available. A popular alternative approach to solving the exact Navier-Stokes equations is to use turbulence models, which predict the effects of turbulence by making simplifying assumptions. Turbulence models can produce reasonable solutions to flow problems but there is not a single turbulence model is capable of predicting all features for any type of flow. Specific turbulence models are often tailored for a certain type of flow (i.e.: external aerodynamics, internal, high rotation, etc.). As a result, CFD users must rely on the correct implementation of turbulence models in the computational software being used to solve a certain type of flow problem. One would hope that if the same turbulence model were used in two different computational packages, the solutions obtained would converge to the same result but that is often not the case [1][2]. To gain confidence in a turbulence model's implementation, the user may want to "verify" it. Verification consists on the use of reference solutions obtained with highly accurate numerical methods on benchmark problems. The goal of this work is to verify a number of turbulence models using the Open Field Operation and Manipulation (OpenFOAM) computational toolbox [3].

This work will provide a brief overview on Reynolds-averaged Navier-Stokes (RANS) turbulence modeling. Previous research pertinent to the topic treated in Sections II-III will be discussed after. The simulation parameters and a description of the flow geometry for each flow case will be presented in Section II. The simulation results obtained with OpenFOAM using standard versions of RANS models will be presented in Section III. The validity of the results obtained with OpenFOAM will be verified by comparing them with reference results obtained with NASA's high-order codes, CFL3D [4] and FUN3D [5], direct numerical simulation data, and experimental measurements. To finalize this document, some concluding remarks will be provided in Section IV.

a. Turbulence Modeling

***This section contains information from [6] and [7].

The equations describing fluid flow are known as the Navier-Stokes equations. They are composed of conservation of momentum and continuity equations. The incompressible version of these equations is as follows:

$$\frac{Du_i}{Dt} \equiv \frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial p}{\partial x_i} + \nu \frac{\partial^2 u_i}{\partial x_j \partial x_j} \qquad (\mathbf{1})$$

$$\frac{\partial u_i}{\partial x_i} = 0 \qquad (\mathbf{2})$$

where ν represents the kinematic viscosity, defined as $\nu = \mu/\rho$, and the subscript i represents each component of the corresponding variable. Unless specified otherwise, summation over repeated indices is implied.

In the RANS approach, the flow velocity, u_i , is decomposed into a time-averaged velocity and an instantaneous velocity fluctuation through the use of Reynolds decomposition:

$$u_i(x_i, t) = \bar{u}_i(x_i) + u_i'(x_i, t)$$

where the time-averaged or mean velocity component of a steady flow is defined as

$$\bar{u}_i(x_i) = \lim_{T\to\infty} \frac{1}{T}\int_t^{t+T} u_i(x_i, t)dt.$$

In the equation above, T is the averaging interval, which has to be large with respect to the time scale of the velocity fluctuations.

Applying Reynolds averaging to the incompressible continuity equation yields

$$\frac{\partial \bar{u}_j}{\partial x_j} = 0\,. \qquad (\mathbf{3})$$

Averaging the left hand side of Eq. 1, we get

$$\overline{\frac{Du_i}{Dt}} \equiv \frac{\partial \bar{u}_i}{\partial t} + \frac{\partial \overline{u_i u_j}}{\partial x_j} = \frac{\partial \bar{u}_i}{\partial t} + \bar{u}_i \frac{\partial \bar{u}_j}{\partial x_j} + \bar{u}_j \frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \overline{u_i' u_j'}}{\partial x_j} \,. \tag{4}$$

Employing Eq.3, Eq. 4 becomes

$$\overline{\frac{Du_i}{Dt}} = \frac{\partial \bar{u}_i}{\partial t} + \bar{u}_j \frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \overline{u_i' u_j'}}{\partial x_j} \,. \tag{5}$$

Taking Eq.5 into account and averaging each term the momentum equation yields

$$\frac{\partial \bar{u}_i}{\partial t} + \bar{u}_j \frac{\partial \bar{u}_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial \bar{p}}{\partial x_i} + \nu \frac{\partial^2 \bar{u}_i}{\partial x_j \partial x_j} - \frac{\partial \overline{u_i' u_j'}}{\partial x_j} \,. \tag{6}$$

For Newtonian fluids, the second to last term in Eq.6 is represented as a viscous stress tensor, defined as

$$t_{ij} = 2\mu S_{ij}, \tag{7}$$

where S_{ij} is the strain-rate tensor

$$S_{ij} = \frac{1}{2}\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right).$$

Substituting Eq.7 into Eq.6 and multiplying by the fluid's density, ρ, yields

$$\rho\left(\frac{\partial \bar{u}_i}{\partial t} + \bar{u}_j \frac{\partial \bar{u}_i}{\partial x_j}\right) = -\frac{\partial \bar{p}}{\partial x_i} + \frac{\partial}{\partial x_j}\left(2\mu S_{ij} - \rho \overline{u_i' u_j'}\right) . \tag{8}$$

The combination of Eq.3 and Eq.8 is known as the Reynolds-Averaged Navier-Stokes (RANS) equations.

The quantity $-\rho\overline{u_i' u_j'}$ in Eq.8 is known as the Reynolds stress tensor. The specific Reynolds stress tensor is $-\overline{u_i' u_j'}$, but it is often referred to as the Reynolds stress tensor as well. The Reynolds stress tensor is symmetric, which means that only six out of its nine components are independent. The unknown variables for a three-dimensional flow are: pressure, three velocity components and the six independent components of the Reynolds stress tensor, which makes a total of ten unknowns. The system is composed of only four equations, continuity and momentum conservation in each direction, which is six less than what is needed to close the system. The absence of the additional equations necessary to close the mathematical system is referred to as the turbulence closure problem. The closure problem is caused by the inclusion of $\overline{u_i'u_j'}$ in the equations. To compute all mean-flow properties of the turbulent flow, a prescription for computing $\overline{u_i'u_j'}$ is needed [6].

Types of Models

<u>Eddy Viscosity Models</u>

A very popular way to model the Reynolds stresses known as the Boussinesq eddy viscosity approximation was introduced by Joseph Boussinesq in 1887. Boussinesq postulated that the momentum transfer caused by turbulent eddies can be modeled with an eddy viscosity [8]. The eddy viscosity, also known as turbulent viscosity, is always positive and is computed from a mixing length that depends on the flow that is being analyzed. The use of an eddy viscosity, μ_t, assumes flow isotropy, which can sometimes lead to excessive diffusion [9]. In incompressible flows, the turbulent viscosity can be divided by the fluid's density, after which it is represented by ν_t. As will be shown in Section II, the definition of eddy viscosity varies from model to model. The Boussinesq approximation relates the Reynolds stress term found in the momentum equation to the eddy viscosity, the rate of strain, and the turbulent kinetic energy, k, in the following way:

$$-\rho\overline{u_i'u_j'} = 2\mu_t S_{ij} - \frac{2}{3}k\rho\delta_{ij}\,. \tag{9}$$

The second term in Eq.9 is present to assure that the sum of the normal stresses is equal to $2k$ [10][6], which is necessary due to the way that turbulence kinetic energy is defined:

$$k = \frac{1}{2}\rho\overline{u_i'u_i'}\,.$$

Turbulence models that employ the Boussinesq eddy viscosity approximation are often referred to as eddy viscosity models, or EVMs. Two types of EVMs are one-equation models and two-equation models. One-equation models solve an additional transport equation for a turbulent variable, usually turbulent kinetic energy, whereas two-equation models solve equations for k and a turbulence length scale, or an equivalent variable. An example of a turbulence scale of interest in two-equation models is the specific turbulence dissipation, denoted by ω. The specific turbulence dissipation represents the rate at which turbulence kinetic energy is converted into thermal internal energy per unit volume and time. Sometimes the specific turbulent dissipation is referred to as the mean frequency of the turbulence; the coining of this term is mainly based on dimensional analysis because ω has units of s^{-1} [11].

The transport equation for turbulent kinetic energy is obtained from the momentum equation by multiplying it by u_i, averaging, and performing basic mathematical manipulations. The derivation of the exact transport equation for turbulent kinetic energy is covered in most textbooks, so only the final result is shown:

$$\frac{\partial k}{\partial t} + \bar{u}_j \frac{\partial k}{\partial x_j} = \underbrace{\frac{\partial}{\partial x_j}\left(\nu \frac{\partial k}{\partial x_j} - \frac{1}{\rho}\overline{u_j' p'} - \frac{1}{2}\overline{u_i' u_i' u_j'}\right)}_{\mathcal{D}_k} \underbrace{- \overline{u_i' u_j'} \frac{\partial \bar{u}_i}{\partial x_j}}_{P_k} - \underbrace{\nu \overline{\frac{\partial u_i'}{\partial x_j}\frac{\partial u_i'}{\partial x_j}}}_{\varepsilon} . \quad \textbf{(10)}$$

The unsteady and convective terms on the left hand side of Eq.10 represent the overall change in k. On the right hand side, the first term, denoted by D_k, is known as the diffusive transport. The components of D_k represent different mechanisms for turbulence kinetic energy transport and they are known as: molecular diffusion, which represents diffusion by the fluid's natural molecular transport process, pressure diffusion, which represents diffusion via pressure–velocity fluctuations, and the triple velocity correlation, known as the turbulent transport term, which is related to transport via turbulent fluctuations. The second term, on the right hand side of Eq.10, denoted by P_k, is known as the production, and it represents the rate at which kinetic energy is transferred from the mean flow to turbulence. The last term, represented by ε, is known as dissipation and it is the rate at which kinetic energy is converted into thermal internal energy [6]. In order to close Eq.10, the Reynolds stresses, dissipation, turbulent transport, and pressure diffusion have to be specified.

The Reynolds stress tensor is modeled through the use of the Boussinesq approximation and it is defined in the following way:

$$-\overline{u_i' u_j'} \equiv \tau_{ij} = 2\nu_t S_{ij} - \frac{2}{3} k \delta_{ij} . \quad \textbf{(11)}$$

The dissipation model varies from model to model. The author in Ref.12 suggested that the dissipation be defined as

$$\varepsilon = C_D \frac{k^{3/2}}{\ell}\,, \tag{12}$$

where C_D is a closure coefficient that ranges between 0.07 and 0.09 [6] and ℓ is a turbulence length scale that depends on the type of flow that is being modeled.

Both of the diffusive terms are usually modeled as a single term, in the following way:

$$\frac{1}{\rho}\overline{u_j' p'} + \frac{1}{2}\overline{u_i' u_i' u_j'} = -\frac{\nu_t}{\sigma_k}\frac{\partial k}{\partial x_j}\,. \tag{13}$$

In the equation above, ν_t is the eddy viscosity and σ_k is a closure coefficient known as the turbulent Prandtl number, which is usually assumed to be constant and on the order of one.

The combination of Eqs.10-13 yields the modeled version of the turbulent kinetic energy equation:

$$\frac{\partial k}{\partial t} + \bar{u}_j \frac{\partial k}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\left(\nu + \frac{\nu_t}{\sigma_k}\right)\frac{\partial k}{\partial x_j}\right) + \tau_{ij}\frac{\partial \bar{u}_i}{\partial x_j} - C_D \frac{k^{3/2}}{\ell}\,. \tag{14}$$

The modeling that has been implemented to close the system leads to a significant loss of detail, but it makes the system solvable.

Reynolds Stress Transport Models

Reynolds Stress Models (RSM), or Reynolds Stress Transport (RST) Models, are a more elaborate category of turbulence models. The method of closure employed in RSM models is called a second-order closure. Second-order closure evades the use of an isotropic eddy viscosity because it calculates the Reynolds stresses from transport equations for each component. Calculating the components of the Reynolds stress tensor is beneficial because doing so accounts for directional effects of the Reynolds stress fields such as streamline curvature, sudden changes in strain rate, secondary motions, etc.[6]. Although it may seem obvious to use an RST model to simulate a given flow, an engineer must consider the expense of the increase in accuracy. Instead of only solving one or two equations, like in EVMs, transport equations must be solved for each of the six independent components of the Reynolds stress tensor and for turbulent dissipation, increasing the total number of equations to 7. A reason for choosing second-moment closures is that turbulent shear flows are not in any general sense describable by a model based on a linear eddy viscosity model [13].

The transport equation for the Reynolds stresses is defined as

$$\frac{D\overline{u_i'u_j'}}{Dt} \equiv \frac{\partial \overline{u_i'u_j'}}{\partial t} + \bar{u}_k \frac{\partial \overline{u_i'u_j'}}{\partial x_k} = \underbrace{\frac{\partial}{\partial x_k}\left[\nu \frac{\partial \overline{u_i'u_j'}}{\partial x_k} - \overline{u_i'u_j'u_k'}\right]}_{D_{ij}} \underbrace{-\left(\overline{u_i'u_k'}\frac{\partial \bar{u}_j}{\partial x_k} + \overline{u_j'u_k'}\frac{\partial \bar{u}_i}{\partial x_k}\right)}_{P_{ij}}$$

$$\underbrace{-2\nu \overline{\frac{\partial u_i'}{\partial x_k}\frac{\partial u_j'}{\partial x_k}}}_{\varepsilon_{ij}} \underbrace{-\frac{1}{\rho}\left(\overline{u_i'\frac{\partial p'}{\partial x_k}} + \overline{u_j'\frac{\partial p'}{\partial x_i}}\right)}_{\Pi_{ij}} \; . \quad \mathbf{(15)}$$

The components of Eq.15 are very similar to those found in Eq.10. Terms on the left account for unsteady and convective changes in the Reynolds stress. The right hand side terms are the diffusion, D_{ij}, production, P_{ij}, dissipation, ε_{ij}, and fluctuating pressure, Π_{ij}, related to the transport of Reynolds stresses. In order to close Eq.15, the diffusion, dissipation, and fluctuating pressure tensors have to be specified. The viscous term in the diffusion component can be obtained directly. The triple product, previously referred to as turbulent transport, is modeled through the use of the generalized gradient diffusion hypothesis (GGDH) developed by Daly and Harlow[14]. The GGDH approximation takes the following form:

$$\overline{u_i'u_j'u_k'} = -c_s \frac{k}{\varepsilon} \overline{u_k'u_l'} \frac{\partial \overline{u_i'u_j'}}{\partial x_l} \ , \qquad (\mathbf{16})$$

where c_s is a model constant and has a value of 0.2.

It is common practice to adopt an isotropic relation for ε_{ij} and to absorb any departure from isotropy in the dissipation processes into the turbulent parts of Π_{ij} [13][15]. The typical isotropic approximation of the dissipation tensor is defined as

$$\varepsilon_{ij} = \frac{2}{3} \varepsilon \delta_{ij} \ , \qquad (\mathbf{17})$$

where ε is determined from its own transport equation.

The fluctuating pressure term is usually decomposed into two parts [13]:

$$\underbrace{-\frac{1}{\rho}\left(\overline{u_i' \frac{\partial p'}{\partial x_j}} + \overline{u_j' \frac{\partial p'}{\partial x_i}}\right)}_{\Pi_{ij}} = \underbrace{-\frac{\partial}{\partial x_k}\left(\frac{1}{\rho}\, \overline{p'\left(u_i'\delta_{jk} + u_j'\delta_{ik}\right)}\right)}_{D_{ij}^{p}} + \underbrace{\frac{1}{\rho} \overline{p' \left(\frac{\partial u_i'}{\partial x_j} + \frac{\partial u_j'}{\partial x_i}\right)}}_{\phi_{ij}} \ . \qquad (\mathbf{18})$$

The first component in Eq.18 is known as the pressure diffusion and is denoted by D_{ij}^p. It accounts for the diffusion of the Reynolds stresses via pressure fluctuations and is often included in the diffusive component, D_{ij}, of Eq.15. The second component is the pressure-rate-of-strain tensor, ϕ_{ij}, which is considered to be the most challenging task in second-moment closure and is modeled differently across different RST models [13]. No explicit model for D_{ij}^p has been proposed [13].

There exist many different ways to represent the pressure-rate-of-strain tensor. In this document, Φ_{ij} will be decomposed into four components:

$$\Phi_{ij} = \Phi_{ij1} + \Phi_{ij2} + \left(\Phi_{ij}^{w1} + \Phi_{ij}^{w2}\right) f(x_n)\,.$$

The first term, Φ_{ij1}, represents the return to isotropy of non-isotropic turbulence, and is often referred to as the "slow" or Rotta term [16]. This term is traceless, which promotes return to isotropy and is defined as

$$\Phi_{ij1} = -c_1 \frac{\varepsilon}{k}\left(\overline{u_i' u_j'} - \frac{2}{3} k \delta_{ij}\right)\,. \qquad (\mathbf{19})$$

The model constant, c_1, has a value of 1.8. The second term, Φ_{ij2}, represents the isotropization of strain production and is referred to as the "rapid" term [17]. The mathematical definition of the rapid term is

$$\Phi_{ij2} = -c_2 \left(P_{ij} - \frac{1}{3} P_{kk} \delta_{ij}\right)\,, \qquad (\mathbf{20})$$

where P_{ij} is the Reynolds stress production tensor and the model constant, c_2, has a value of 0.6. The third and fourth components of Φ_{ij} are meant to account for near-wall effects. The first term, ϕ_{ij}^{w1}, was developed in [18] and the second term, ϕ_{ij}^{w2}, was developed in [19]. Their definitions are

$$\phi_{ij}^{w1} = c_1^w \frac{\varepsilon}{k}\left[\overline{u_n'^{\,2}}\, g_{nn}\delta_{ij} - \frac{3}{2}\left(\overline{u_n' u_j'} g_{in} + \overline{u_n' u_i'} g_{jn}\right)\right], \qquad (\mathbf{21})$$

and

$$\phi_{ij}^{w2} = c_2^w \frac{\varepsilon}{k}\left[\phi_{nn2}\, g_{nn}\delta_{ij} - \frac{3}{2}\left(\phi_{nj2} g_{in} + \phi_{ni2} g_{jn}\right)\right], \qquad (\mathbf{22})$$

respectively. In Eqs.21 and 22, the model coefficients, c_1^w and c_2^w, have values of 0.3, g_{ij} represents the metric tensor, and n represents the direction normal to the wall.

Summation over repeated n indices is not implied in Eqs.21 and 22. Both wall terms are multiplied by $f(x_n)$, a damping function defined as

$$f = \frac{1}{5}\frac{k^{3/2}}{\varepsilon x_n},$$

where x_n is the distance normal to the wall [20].

This concludes the general description of some of the approaches used to model the Reynolds stresses in RANS turbulence models.

b. Literature Review

The flows discussed in this document have been studied in great detail by multiple researchers. Reviewing every publication related to flow over a flat plate and a two dimensional bump in a channel would require an extensive amount of space and time, so only a small number of experimental and computational references will be mentioned.

Schwarz conducted experiments of flow over a flat plate and measured flow variables such as pressure, velocities, skin friction coefficient and Reynolds stresses to address concerns related to turbulence modeling [21]. DeGraaff and Eaton performed an experiment to verify Reynolds number scaling of a zero-pressure-gradient boundary layer over a flat plate. It was found that the log law provides a reasonably accurate universal profile for the mean velocity in the inner region of the boundary layer [22]. Experimental results for a flat plate boundary layer near a free surface in Ref.23 matched benchmark results closely. Castillo and Johansson conducted an experiment and a similarity analysis of the RANS equations on a zero pressure gradient flow over a flat plate to investigate the effect of local Reynolds number and upstream conditions on the development of the mean flow and turbulent quantities [24].

Many studies have been performed to evaluate the accuracy of different turbulence models, boundary layer structure and sensitivity to mesh size in computations. For example, researchers at NASA computed accurate numerical solutions using two-equation models for selected flows and compared them to experimental values [25]. In the study, the models' overall performance was ranked from best to worst in the following order: SST, Spalart-Allmaras, and Wilcox's 1988 version of k-ω. The authors deemed the Spalart-Allmaras model the best in terms of numerical performance, followed by the SST model and k-ω. The evaluation was based on the grid spacing required for accurate solutions and the maximum y^+ allowable at the first grid point off the wall. Simulation results obtained by Chan et al. using Wilcox's 2006 version of k-ω showed good agreement with experimental and theoretical results for flow over a 2D flat plate [26]. A comparison between the results obtained with the k-ω and SST models for flow over a flat plate showed that the SST model predicted a mean velocity profile that was very similar to that obtained with k-ω in [27]. A two-equation turbulence model developed and verified by Xu et al. showed excellent agreement with experimental values for a zero pressure gradient turbulent boundary layer on a flat plate in [28]. The computational studies mentioned so far verify the accuracy of the results by comparing them with experimental data. However, DNS results can be considered to be as good measurements obtained from experiments. As a result, they are often used to evaluate the accuracy of turbulence models. Spalart [29], Wu and Moin [30], and Sillero et al. [31] have produced some of the most widely accepted DNS results for flow over a flat plate.

A smaller amount of research has been done on flow over the 2D bump-in-channel. Computations have been performed for flow over the bump geometry and the results were used to determine the performance of RANS models with respect to large eddy simulation (LES), detached eddy simulation (DES), DNS, and experimental results. Osusky et al. used a novel solution algorithm to obtain results with RANS models that matched those obtained by NASA for flow over the bump-in-channel [32]. Furbo conducted simulations for flow over a bump using the default RANS turbulence models in OpenFOAM [7]. The results were compared to experimental and LES data and it was noticed that most of the RANS models tested didn't predict a separation zone downstream of the bump. Bensow et al. also described the difficulties of obtaining flow details using RANS instead of LES and DES for flow over the bump geometry [33]. CFD results predicted the separation location correctly, but not the reattachment location for flow over a bump in [34]. It was also found that the results obtained with the SST model showed discrepancies between detachment and reattachment locations. Disagreements between results for the pressure coefficient obtained with RANS models and experimental results were noted in [35]. However, DNS predictions were shown to have good agreement with experiments for flow over a bump in [36]. Experimental results for similar bump geometries can be found in [37], [38], [39], [40], [41], and in the European Research Community on Flow, Turbulence and Combustion Database (ERCOFTAC) [42]. An extensive list of previous research related to this specific flow geometry can also be found in the Case 3 section of [43]. The discrepancies related to RANS results that were described in literature were different from one study to the next, even when the same turbulence model was being used. As a result, the simulation results for the flow over a bump will only be compared with results obtained with NASA's high-accuracy codes and not with experiments.

The difference between results obtained with RANS and other sources has to be addressed. One way to approach the issue is to compare highly accurate numerical results to a benchmark flow problem with results obtained with lower order CFD packages. The aforementioned approach of comparing results obtained with highly accurate codes to those obtained with lower order codes is typically used to verify the numerical models or components of the lower order codes. Verification is defined as the process of determining that a model implementation accurately represents the developer's conceptual description of the model and the solution to the model [44]. Verification is important and necessary because it is used to assess the accuracy and errors in numerical modeling and solution of flow problems. Rizzi and Vos include a thorough discussion on the importance of establishing credibility in CFD simulations through verification in [45]. Roache provides a background discussion and some of the definitions and descriptions that are necessary for the verification of codes and calculations in [46]. The main conclusion to be drawn from [45] and [46] is that the verification of the components in CFD toolboxes is essential to address the types and sources of error from conducting simulations.

Two studies that emphasize the need for turbulence model verification in CFD toolboxes will be mentioned briefly. First, is a study performed by Vassberg et al. in which simulations with the "same" turbulence model implemented in different CFD packages gave varying results [47]. Wilcox performed the second study and he showed that slightly different versions of the k-ε model produce significantly different results for boundary layer flows [48]. Inconsistencies of this type can make engineers and researchers apprehensive about believing CFD results. According to Rumsey, it is often difficult to draw firm conclusions about turbulence model accuracy when performing multi-code CFD studies ostensibly using the same model because of inconsistencies in model formulation or implementation in different codes [49].

In an effort to improve consistency, verification, and validation of turbulence models within the aerospace community, NASA has established a website to provide a central location for the documentation of RANS turbulence models [50]. The website is called the Turbulence Modeling Resource (TMR) and it is a collaboration between NASA's Langley Research Center and the Turbulence Model Benchmarking Working Group (TMBWG) [51], a working group of the Fluid Dynamics Technical Committee [52] of the American Institute of Aeronautics and Astronautics (AIAA) [53]. The objective of the TMR website is to provide a resource for CFD developers to:

- Obtain accurate and up-to-date information on widely used RANS turbulence models.
- Verify that models are implemented correctly.

Correct implementation of models can be confirmed through verification cases provided on the TMR website.

II. Simulation Parameters

a. OpenFOAM

The flow simulations that will be discussed in Section III were conducted on NASA's Pleiades Supercomputer [54] using OpenFOAM. Details about Pleiades can be found in Appendix A.

OpenFOAM is a free, open source CFD software package, licensed and distributed by the OpenFOAM Foundation [55] and developed by OpenCFD Ltd[56]. OpenFOAM is used in academia and industry to solve problems ranging from complex fluid flows involving chemical reactions, turbulence, and heat transfer, to solid dynamics and electromagnetics. Almost all of the operations in OpenFOAM are capable of running in parallel, which enables users to take advantage of parallel computing. OpenFOAM is an object oriented code based on C++ and its open source nature gives users the freedom to customize and expand the existing libraries [57]. The Repository Release version of OpenFOAM (2.2.x) was used during this study [58].

An OpenFOAM simulation is defined by a group of subdirectories, each containing specific files, as shown in Fig.1. The file structure of an OpenFOAM case is composed of a system directory, where parameters associated with the solution procedure are defined, a constant directory, which contains mesh information and physical properties for the case, and the time directories, where initial/boundary conditions and results for each recorded time step are saved.

b. Turbulence Models in OpenFOAM

This subsection will describe the turbulence models that were used in this study. The transport equations for three different EVMs and an RST model implemented in OpenFOAM will be presented. The EVMs are the Spalart-Allmaras (SA) model [59], the Menter Shear Stress Transport (SST) model [60], Wilcox's 2006 version of the k-ω model [6], and the RST model is a version of the Launder-Reece-Rodi isotropization of production (LRR-IP) model [61]. The turbulence model equations for all EVMs that were originally implemented in OpenFOAM did not match the "standard" definitions found on NASA's Turbulence Modeling Resource website [51]. As a result, all model equations in OpenFOAM were modified to represent the exact definitions found on NASA's Turbulence Modeling Resource website. A brief description of the changes made to each model is included at the end of the model's subsection. The LRR turbulence model in OpenFOAM was modified to include the models that were described in Section I. Source code for all of the turbulence models can be found in Appendix B.

Spalart-Allmaras Model

A popular one-equation EVM is the Spalart-Allmaras model, which solves a transport equation for an eddy-viscosity-like variable, $\tilde{\nu}$. According to [62], the standard version of the transport equation for $\tilde{\nu}$ is

$$\frac{\partial \tilde{\nu}}{\partial t} + \bar{u}_j \frac{\partial \tilde{\nu}}{\partial x_j} = c_{b1}(1 - f_{t2})\tilde{S}\tilde{\nu} - \left[c_{w1} f_w - \frac{c_{b1}}{\kappa^2} f_{t2}\right]\left(\frac{\tilde{\nu}}{d}\right)^2 + \frac{1}{\sigma}\left[\frac{\partial}{\partial x_j}\left((\nu + \tilde{\nu})\frac{\partial \tilde{\nu}}{\partial x_j}\right) + c_{b2}\frac{\partial \tilde{\nu}}{\partial x_i}\frac{\partial \tilde{\nu}}{\partial x_i}\right].$$

The closure functions are defined as

$$f_{t2} = c_{t3} \exp(-c_{t4}\chi^2)$$

$$\chi = \frac{\tilde{\nu}}{\nu}$$

$$\tilde{S} = \Omega + \frac{\tilde{\nu}}{\kappa^2 d^2} f_{v2}$$

$$\Omega = \sqrt{2W_{ij}W_{ij}}$$

$$W_{ij} = \frac{1}{2}\left(\frac{\partial \bar{u}_i}{\partial x_j} - \frac{\partial \bar{u}_j}{\partial x_i}\right)$$

$$f_{v2} = 1 - \frac{\chi}{1 + \chi f_{v1}}$$

$$f_{v1} = \frac{\chi^3}{\chi^3 + c_{v1}^3}$$

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

$$f_w = g\left[\frac{1 + c_{w3}^6}{g^6 + c_{w3}^6}\right]^{1/6}$$

$$g = r + c_{w2}(r^6 - r)$$

$$r = \min\left[\frac{\tilde{\nu}}{\tilde{S}\kappa^2 d^2}, 10\right]$$

where d is the distance from the field point to the nearest wall.

The eddy viscosity is computed from

$$\nu_t = \tilde{\nu} f_{v1}.$$

The values for the model coefficients can be found in Table 1.

$\boldsymbol{c_{b1}}$	$\boldsymbol{c_{b2}}$	$\boldsymbol{\kappa}$	$\boldsymbol{\sigma}$	$\boldsymbol{c_{t3}}$	$\boldsymbol{c_{t4}}$	$\boldsymbol{c_{w2}}$	$\boldsymbol{c_{w3}}$
0.1355	0.622	0.41	$\frac{2}{3}$	1.2	0.5	0.3	2

Table 1: Spalart-Allmaras model coefficients

Changes to the SpalartAllmaras model in OpenFOAM were the following:

- Modified OpenFOAM's definition of the f_{v2} function to be in accordance with NASA's.
- Eliminated f_{v3} function and c_{v2} coefficient found in OpenFOAM.
- Added f_{t2}, c_{t3}, and c_{t4} as described in the equations and table above.
- Modified $\tilde{S}$ definition to take into account changes listed above.
- Modified $\tilde{\nu}$ transport equation to take into account changes listed above.

Menter Shear Stress Transport Model

The SST model is a two-equation EVM model that solves transport equations for k and ω. According to [63], the standard version of the incompressible transport equation for turbulent kinetic energy is

$$\frac{\partial k}{\partial t} + \bar{u}_j \frac{\partial k}{\partial x_j} = P^* - \beta^* \omega k + \frac{\partial}{\partial x_j}\left[(\nu + \sigma_k \nu_t)\frac{\partial k}{\partial x_j}\right].$$

Transport of the specific turbulence dissipation rate is described by

$$\frac{\partial \omega}{\partial t} + \bar{u}_j \frac{\partial \omega}{\partial x_j} = \frac{\gamma_\omega}{\nu_t} P - \beta_\omega \omega^2 + \frac{\partial}{\partial x_j}\left[(\nu + \sigma_\omega \nu_t)\frac{\partial \omega}{\partial x_j}\right] + 2(1 - F_1)\frac{\sigma_{\omega 2}}{\omega}\frac{\partial k}{\partial x_j}\frac{\partial \omega}{\partial x_j},$$

with model functions defined as

$$P^* = \min(P, 20\beta^* \omega k)$$

$$P = \tau_{ij}\frac{\partial \bar{u}_i}{\partial x_j}$$

$$\tau_{ij} = 2\nu_t S_{ij} - \frac{2}{3}k\delta_{ij}$$

$$S_{ij} = \frac{1}{2}\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right)$$

$$\nu_t = \frac{a_1 k}{\max(a_1 \omega, \Omega F_2)}$$

$$\Omega = \sqrt{2W_{ij}W_{ij}}$$

$$W_{ij} = \frac{1}{2}\left(\frac{\partial \bar{u}_i}{\partial x_j} - \frac{\partial \bar{u}_j}{\partial x_i}\right)$$

$$F_2 = \tanh(arg_2^2)$$

$$arg_2 = \max\left(2\frac{\sqrt{k}}{\beta^* \omega d}, \frac{500\nu}{d^2 \omega}\right)$$

$$F_1 = \tanh(arg_1^4)$$

$$arg_1 = \min\left[\max\left(2\frac{\sqrt{k}}{\beta^*\omega d}, \frac{500\nu}{d^2\omega}\right), \frac{4\sigma_{\omega 2}k}{CD_{k\omega}d^2}\right]$$

$$CD_{k\omega} = \max\left(2\sigma_{\omega 2}\frac{1}{\omega}\frac{\partial k}{\partial x_j}\frac{\partial \omega}{\partial x_j}, 10^{-20}\right)$$

where the variable d is the distance from the field point to the nearest wall. The constants in the transport equations that don't have a number as part of the subscript are obtained through a blending function of the following form:

$$\phi = F_1\phi_1 + (1 - F_1)\phi_2$$

where ϕ is the value of the constant without a number in the subscript and ϕ_1 and ϕ_2 represent constants 1 and 2. For example, σ_k is defined as

$$\sigma_k = F_1\sigma_{k1} + (1 - F_1)\sigma_{k2}$$

Values for the constant coefficients can be found in Table 2 and the remaining model coefficients are defined as

$$\gamma_1 = \frac{\beta_1}{\beta^*} - \frac{\sigma_{\omega 1}\kappa^2}{\sqrt{\beta^*}} \text{ and } \gamma_2 = \frac{\beta_2}{\beta^*} - \frac{\sigma_{\omega 2}\kappa^2}{\sqrt{\beta^*}} \quad .$$

$\boldsymbol{\sigma_{k1}}$	$\boldsymbol{\sigma_{k2}}$	$\boldsymbol{\sigma_{\omega 1}}$	$\boldsymbol{\sigma_{\omega 2}}$	$\boldsymbol{\beta_1}$	$\boldsymbol{\beta_2}$	$\boldsymbol{\beta^*}$	$\boldsymbol{\kappa}$	$\boldsymbol{a_1}$
0.85	1	0.5	0.856	0.075	0.0828	0.09	0.41	0.31

Table 2: Menter SST model coefficients

Changes to the kOmegaSST model in OpenFOAM were the following:

- Modified OpenFOAM's definition of the $CD_{k\omega}$, $arg1$ and $arg2$ functions to be in accordance with NASA's.
- Eliminated F_3 and F_{23}functions found in OpenFOAM.
- Modified ν_t definition to be in accordance with NASA's.

- Modified $\frac{\gamma_\omega}{\nu_t} P$ term and the sign of the last term in the ω transport equation to be in accordance with NASA.
- Substituted P in k transport equation with P^* as defined above.

Wilcox 2006 version of *k-ω*

Another turbulence model used in this study is Wilcox's 2006 version of k-ω. Similarly to SST, Wilcox's k-ω, which will be referred to as k-ω for the remainder of this document, is a two-equation model that also solves for k and ω. According to [64], the incompressible transport equations for k and ω are

$$\frac{\partial k}{\partial t} + \bar{u}_j \frac{\partial k}{\partial x_j} = P - \beta^* \omega k + \frac{\partial}{\partial x_j}\left[\left(\nu + \sigma_k \frac{k}{\omega}\right)\frac{\partial k}{\partial x_j}\right]$$

and

$$\frac{\partial \omega}{\partial t} + \bar{u}_j \frac{\partial \omega}{\partial x_j} = \frac{\gamma\omega}{k} P - \beta\omega^2 + \frac{\partial}{\partial x_j}\left[\left(\nu + \sigma_\omega \frac{k}{\omega}\right)\frac{\partial \omega}{\partial x_j}\right] + \frac{\sigma_d}{\omega}\frac{\partial k}{\partial x_j}\frac{\partial \omega}{\partial x_j},$$

where

$$P = \tau_{ij}\frac{\partial \bar{u}_i}{\partial x_j}$$

$$\tau_{ij} = 2\nu_t S_{ij} - \frac{2}{3}k\delta_{ij}$$

$$S_{ij} = \frac{1}{2}\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right)$$

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

$$\tilde{\omega} = \max\left[\omega, C_{\lim}\sqrt{\frac{2S_{ij}S_{ij}}{\beta^*}}\right]$$

The constant model coefficient values can be found in Table 3. Additional relationships are defined as

$$f_\beta = \frac{1 + 85\chi_\omega}{1 + 100\chi_\omega}, \qquad \chi_\omega = \left|\frac{\Omega_{ij}\Omega_{jk}S_{ki}}{(\beta^*\omega)^3}\right|, \qquad \Omega_{ij} = \frac{1}{2}\left(\frac{\partial \bar{u}_i}{\partial x_j} - \frac{\partial \bar{u}_j}{\partial x_i}\right).$$

$$\sigma_d = \begin{cases} 0\,, \dfrac{\partial k}{\partial x_j}\dfrac{\partial \omega}{\partial x_j} \leq 0 \\ \dfrac{1}{8}\,, \dfrac{\partial k}{\partial x_j}\dfrac{\partial \omega}{\partial x_j} > 0 \end{cases}$$

It should be noted that to model 2-D flows, Pope's correction, denoted by χ_ω, should be set equal to zero. This concludes the description of one- and two-equation models that were considered in this study.

$\boldsymbol{\sigma_k}$	$\boldsymbol{\sigma_\omega}$	$\boldsymbol{\beta^*}$	$\boldsymbol{\gamma}$	$\boldsymbol{C_{\text{lim}}}$	$\boldsymbol{\beta}$	$\boldsymbol{\beta_0}$
0.6	0.5	0.09	$\frac{13}{25}$	$\frac{7}{8}$	$\beta_0 f_\beta$	0.0708

Table 3: k-ω model coefficients

Changes to the kOmega model in OpenFOAM were the following:

- Modified β definition to make it a variable constant.
- Changed value of model coefficient σ_k from 0.5 to 0.6.
- Added f_β, χ_ω and σ_d definitions as listed above.
- Modified ν_t definition to include $\widetilde{\omega}$ as defined by NASA.
- Added $CD_{k\omega}$ as the last term in the ω transport equation.

Launder-Reece-Rodi Isotropization of Production Model

The transport equations for the LRR-IP model are composed of 7 equations: a transport equation for each of the six independent Reynolds stresses, and a transport equation for the scalar dissipation. All of the Reynolds stress equations have the same form so only a generic indexed equation will be presented. Different Reynolds stress components can be obtained by changing the value of the indices. Substituting the models discussed in Section I, the Reynolds stress transport equation implemented in OpenFOAM takes the following form:

$$\frac{\partial \overline{u_i'u_j'}}{\partial t} + \bar{u}_k \frac{\partial \overline{u_i'u_j'}}{\partial x_k} = \underbrace{\frac{\partial}{\partial x_k}\left[\nu \frac{\partial \overline{u_i'u_j'}}{\partial x_k} + c_s \frac{k}{\varepsilon}\overline{u_k'u_l'}\frac{\partial \overline{u_i'u_j'}}{\partial x_l}\right]}_{D_{ij}} \underbrace{-\left(\overline{u_i'u_k'}\frac{\partial \bar{u}_j}{\partial x_k} + \overline{u_j'u_k'}\frac{\partial \bar{u}_i}{\partial x_k}\right)}_{P_{ij}}$$

$$\underbrace{-\left(\frac{2}{3}\varepsilon\delta_{ij} + 2\nu\frac{\overline{u_i'u_j'}}{x_n^2}\right)}_{\varepsilon_{ij}} + \underbrace{\left(\Phi_{ij1} + \Phi_{ij2} + \left(\Phi_{ij}^{w1} + \Phi_{ij}^{w2}\right)f(x_n)\ \right)}_{\Pi_{ij}}$$

where the components of Π_{ij} are defined as

$$\Phi_{ij1} = -c_1 \frac{\varepsilon}{k}\left(\overline{u_i'u_j'} - \frac{2}{3}k\delta_{ij}\right)$$

$$\Phi_{ij2} = -c_2\left(P_{ij} - \frac{1}{3}P_{kk}\delta_{ij}\right)$$

$$\phi_{ij}^{w1} = c_1^w \frac{\varepsilon}{k}\left[\overline{u_n'^{\,2}}\, g_{nn}\delta_{ij} - \frac{3}{2}\left(\overline{u_n'u_j'}g_{in} + \overline{u_n'u_i'}g_{jn}\right)\right]$$

$$\phi_{ij}^{w2} = c_2^w \frac{\varepsilon}{k}\left[\phi_{nn2}\, g_{nn}\delta_{ij} - \frac{3}{2}\left(\phi_{nj2}g_{in} + \phi_{ni2}g_{jn}\right)\right]$$

$$f = \frac{1}{5}\frac{k^{3/2}}{\varepsilon x_n}\ .$$

An extra term has been added to the ε_{ij} component to account for near wall effects [20]. The term was included to eliminate the necessity of using wall functions in OpenFOAM.

The transport equation for dissipation is

$$\frac{\partial \varepsilon}{\partial t} + \bar{u}_k \frac{\partial \varepsilon}{\partial x_k} = \frac{\partial}{\partial x_k}\left[\left(\nu\delta_{jk} + c_\varepsilon \frac{k}{\varepsilon}\,\overline{u_j' u_k'}\right)\frac{\partial \varepsilon}{\partial x_j}\right] + \frac{\varepsilon}{k}\left(c_{\varepsilon 1}P - c_{\varepsilon 2}^* \varepsilon\right) - \frac{2\nu\varepsilon}{x_n^2} f_1$$

where

$$P = -\frac{1}{2}\overline{u_i' u_k'}\frac{\partial \bar{u}_i}{\partial x_k}$$

$$c_{\varepsilon 2}^* = c_{\varepsilon 2} f_2$$

$$f_2 = 1 - \frac{2}{9}\exp\left[-\left(\frac{k^2}{6\nu\varepsilon}\right)^2\right]$$

$$f_1 = \exp\left[-\frac{x_n u_\tau}{2\nu}\right]$$

$$u_\tau = \sqrt{\frac{\tau_w}{\rho}}.$$

Similarly to the Reynolds stress equation, the last term in the dissipation equation is present to account for near wall effects [20]. Model coefficients for the LRR-IP model can be found in Table 4.

$\boldsymbol{c_s}$	$\boldsymbol{c_1}$	$\boldsymbol{c_2}$	$\boldsymbol{c_1^w}$	$\boldsymbol{c_2^w}$	$\boldsymbol{c_\varepsilon}$	$\boldsymbol{c_{\varepsilon 1}}$	$\boldsymbol{c_{\varepsilon 2}}$
0.2	1.8	0.6	0.3	0.3	0.15	1.44	1.92

Table 4: LRR-IP model coefficients

Changes to the LRR model in OpenFOAM were the following:

- Incorporated function to calculate normal distance to nearest wall.
- Added wall term to dissipation tensor definition and to dissipation equation.
- Added wall reflection terms to Reynolds stress equation.
- Implemented $c_{\varepsilon 2}^*$ definition.
- Added calculation of wall proximity functions f, f_1 and f_2 as well as u_τ.

c. Numerical Methods

OpenFOAM uses the finite volume method (FVM) to obtain a numerical solution for flow problems. In FVM, the solution to the partial differential equations that describe the flow behavior is approximated by subdividing the computational domain into a finite number of control volume elements and applying conservation laws to each of them.

The process of subdividing continua into finite, or discrete, quantities is known as discretization. A general flow problem is generally composed of three types of discretization: spatial, temporal, and equation. Spatial discretization defines the solution space by specifying a set of points that bound the region in which the problem is solved. Temporal discretization is related to transient problems and it describes how the length of the time that spans the problem is divided into a finite number of smaller time steps. Equation discretization describes the way in which conservation laws are represented through a finite set of algebraic equations at specific locations defined by spatial discretization.

After a finite number of equations describing conservation laws are generated, they must be solved to find the values of the variables of interest for a given flow. Due to the nature of the partial differential equations that describe the fluid's behavior, a set of non-linear coupled equations is usually obtained. These complications make obtaining a solution to the system impossible unless iterative solution methods are employed.

Due to the immense amount of research that has been done in discretization and solution algorithms, an attempt to discuss each of them in detail in a single document is futile. Only the information pertinent to the flow cases treated in this study will be presented in the following subsections. The details of all discretization and solution methods available in OpenFOAM can be found Chapters 2 and 4 of [65], and [66], respectively.

Discretization

Spatial Discretization

The author did not perform spatial discretization in OpenFOAM. Instead, discretized representations of the flow geometries, discussed in detail in Section II, were obtained from NASA's Turbulence Modeling Resource website [50].

Temporal Discretization

The velocity of the flows treated in this study does not vary with time. Since the flows are steady, temporal discretization is not necessary. However, a pseudo-time is introduced in the OpenFOAM simulations for two reasons: i) to control the amount of iterations performed by the solver and ii) to specify the frequency of the output of the solution to the computers hard disk. OpenFOAM's controlDict dictionary, found in the case's system directory, is used to control the aforementioned i) and ii). This is achieved through the definition of values for the endTime, deltaT, writeControl, and writeInterval options in controlDict. A sample of controlDict has been included in Appendix C.

Equation Discretization

In OpenFOAM, discretization schemes used to approximate components of the conservation and turbulence model equations are specified through the fvSchemes dictionary found in the case's system directory. A brief explanation of the main keywords used in fvSchemes can be found in Table 5. A specific numerical scheme can be set as the default setting for all terms belonging to a certain category of the equations. For example, the transient term in all equations can be discretized using the Crank-Nicholson method. Additionally, specific components of each equation can be assigned a specific numerical scheme for discretization/interpolation, which gives the user full control over the computational representation of the flow equations. Using different discretization scheme settings for specific equation components can have an effect on the stability and accuracy of the solution.

The flow cases studied are steady, so the temporal derivatives in all equations are not taken into account. The second-order Gaussian integration scheme is used for every term in momentum and turbulence model equations that involves a derivative. Since OpenFOAM calculates values at each element's center, values have to be interpolated from cell to face centers. The central difference interpolation is used for all gradient terms. Upwind differencing is used for convective terms in all equations, but the scheme's order of accuracy varies between the momentum and turbulence transport equations: the second-order scheme is applied to the terms in the momentum equations and the first-order scheme is used in the turbulence transport equations. The central difference interpolation scheme is used for the diffusion coefficient in all diffusive terms, and an explicit second-order non-orthogonal correction method is employed for surface-normal gradients

Gaussian integration is the only choice of discretization for integration in OpenFOAM and it is specified as Gauss for all terms that require integration. The central difference interpolation scheme used for gradients is referred to as linear in OpenFOAM. Similarly, second- and first- order interpolation schemes used for the convective terms are referred to as linearUpwind, and upwind, respectively. The non-orthogonal correction method used in surface-normal gradients is defined as corrected. More details on the numerical schemes implemented in OpenFOAM can be found in Chapter 4 of [66]. A copy of fvSchemes has been included in Appendix C.

Solution Method

As was previously mentioned, the solution of the resulting set of discretized equations describing flow behavior requires iterative methods. A popular iterative method for solving incompressible steady-state problems in CFD is the Semi-Implicit Method for Pressure Linked Equations (SIMPLE)[67].

The iterative procedure in the SIMPLE algorithm consists of approximating the velocity field by solving momentum equations using pressure values from a previous iteration or initial conditions. The velocities that are obtained from the momentum equations do not satisfy the continuity equation unless the pressure field is corrected. The pressure field is corrected by solving a Poisson equation for pressure. Updating the pressure field causes the velocity and pressure fields to obey continuity but not momentum. Velocity values are then recalculated using the corrected pressure values to satisfy the momentum equations. The procedure described above is repeated until the velocity and pressure fields obey the continuity and momentum equations. A basic outline of the algorithm will be presented next but an in-depth discussion of the philosophy of pressure correction methods and the SIMPLE algorithm are available in [67] and in Chapter 6 of [68].

The steps in the SIMPLE algorithm can be outlined in the following way [69]:

1. Set the boundary conditions.
2. Compute the gradients of velocity and pressure.
3. Solve the discretized momentum equation to compute the intermediate velocity field.
4. Compute the uncorrected mass fluxes at cell faces.
5. Solve the pressure correction equation to produce new/corrected pressure values.
6. Update the pressure field using an under-relaxation factor.
7. Update the boundary values using the pressure corrections.
8. Correct the face mass fluxes.
9. Calculate corrected cell velocities using the pressure gradient of the pressure corrections.

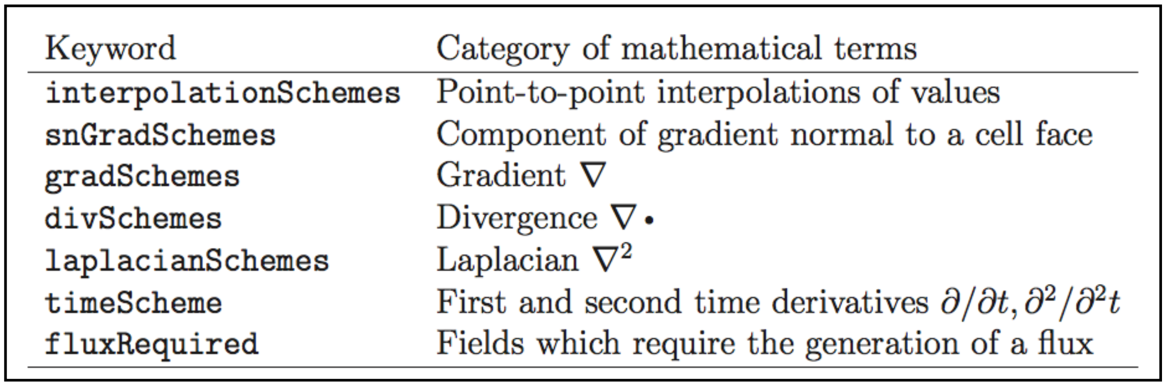
The SIMPLE algorithm was used to obtain the results discussed in Section III. OpenFOAM's incompressible version of SIMPLE is called simpleFoam. The settings for simpleFoam are specified through the fvSolution dictionary, located in the case's system directory. The first simpleFoam setting, located below the SIMPLE header in fvSolution, is defined by the keyword nNonOrthogonalCorrectors. The nNonOrthogonalCorrectors setting accounts for non-orthogonality in the mesh and its use is not necessary in this work because the meshes, described in detail in Section II, are orthogonal. Under-relaxation is used to improve the numerical stability of a computation by limiting the amount by which a variable can change from one iteration to the next. Under-relaxation factors vary from 0 to 1, with 1 corresponding to no under-relaxation. In the fvSolution dictionary, the keyword relaxationFactors is used to define under-relaxation factors for each flow variable. The relaxationFactors value for the pressure field varies from 0.2 to 0.3. For the velocity and turbulence equations, the under-relaxation values range from 0.3 to 0.7. The under-relaxation value used for the flow variables depended on simulation factors such as flow geometry, mesh quality, accuracy of numerical schemes, and turbulence model.

The type of linear solvers and solver tolerances used for each flow variable are also defined in fvSolution. The solver category specifies the type of linear-solver used to solve the set of linear equations for each discretized equation. A preconditioned bi-conjugate gradient (PBiCG) solver was used for all variables with the exception of pressure, for which a preconditioned conjugate gradient (PCG) solver was used. PBiCG is used to solve asymmetric matrices while PCG is used for symmetric matrices. The use of a preconditioned solver requires the specification of a preconditioner. The faster diagonal incomplete-Cholesky (FDIC) preconditioner was selected for all flow variables.

Due to the iterative nature of the linear solvers specified in solver, the reduction of the solution error from one iteration to the next has to be evaluated in order to establish the accuracy of the current solution. In OpenFOAM, the linear solver will stop iterating if the measure of the solution error, also known as the residual, satisfies a limit imposed by the user. OpenFOAM offers three options to stop the linear solver, all of which are defined in fvSolution. The three available options are:

1. The residual falls below the solver tolerance, defined as tolerance.
2. The ratio of current to initial residuals falls below the solver relative tolerance, defined as relTol.
3. The number of iterations exceeds a maximum number of iterations, defined as maxIter (optional).

The results presented in Section III were obtained by setting the tolerance value as 1×10^{-16} and the relTol value as 0. A copy of the fvSolution file has been included in Appendix C.



Keyword	Category of mathematical terms
`interpolationSchemes`	Point-to-point interpolations of values
`snGradSchemes`	Component of gradient normal to a cell face
`gradSchemes`	Gradient ∇
`divSchemes`	Divergence $\nabla \bullet$
`laplacianSchemes`	Laplacian ∇^2
`timeScheme`	First and second time derivatives $\partial/\partial t, \partial^2/\partial^2 t$
`fluxRequired`	Fields which require the generation of a flux

Table 5: fvSchemes keywords [66]

d. Computational Domain

The geometry for two benchmark flow cases will be discussed in this section. A description of the computational domain for the two-dimensional flow over a flat plate with zero pressure gradient will be presented first. A description of the geometry for a two-dimensional bump in a channel will follow after.

A number of structured 2-D grids obtained from NASA's Turbulence Modeling Resource website [50] were used to perform the flow simulations for both flow cases. OpenFOAM solves flow equations in all three spatial dimensions. To model the flow cases as 2-D in OpenFOAM, the two-dimensional grid must be extended one unit in the third dimension, which creates extra domain boundaries. For clarity, these extra boundaries will be referred to as front and back. OpenFOAM's empty boundary condition is assigned to front and back. As a result, the values of flow variables and their corresponding fluxes in the front-to-back direction are set equal to zero.

The meshes used in this study are nested, meaning that each coarser grid is exactly every-other-point of the finer grid [50]. The naming convention for the meshes consists of the amount of nodes in the *x*- and *y*- directions. Note that the computational domains are not defined in terms of meters, or feet, but in terms of dimensionless units. All meshes are available for download in PLOT3D format. Additional information about PLOT3D can be found at [70]. OpenFOAM's plot3dToFoam mesh conversion utility was used to import the PLOT3D mesh files into OpenFOAM.

2D Zero pressure gradient flat plate

The computational domain for the flat plate was 2.33×1 units in the x- and y-directions. The flat plate wall boundary starts at x = 0 and ends at x = 2. The plate is positioned at $y = 0$. The top boundary of the computational domain is located at y = 1. The grids used for the flat plate have vertex dimensions of 35×25, 69×49, 137×97, 273×193, and 545×385 in the x- and y-directions, respectively. An image of the 69×49 grid can be found in Figure 2. Figure 2 shows that mesh biasing is used in the wall-normal direction and near the plate leading edge. The finest grid has a minimum wall spacing of 5×10^{-7}, giving an average y^+value of about 0.07. The coarsest mesh has a minimum wall spacing of 8.32×10^{-6}, which gives an average y^+value of about 1.7. The variable y^+ is a non-dimensional wall distance used in wall-bounded flows to describe the regions of a boundary layer in a generalized manner applicable to different flows. The mathematical representation of y^+ is

$$y^+ \equiv \frac{u_\tau y}{\nu},$$

where u_τ represents the friction velocity, y represents the distance from the wall, and ν represents the kinematic viscosity of the fluid.

2D Bump-in-channel

The bump-in-channel case is similar to the flat plate case that was previously mentioned, except wall curvature is present. The curvature present in the geometry causes pressure gradients. The computational domain measures 51.5×5 units in the x- and y-directions. The wall boundary starts at location x = 0 and y = 0. The bump starts at x = 0.3, and y = 0. The top of the bump is at x = 0.75 and y = 0.05. The bump is symmetrical. The wall downstream of the bump ends at x = 1.5. The bump profile, shown in Figure 3, is defined by

$$y = \begin{cases} 0.05\left(\sin\left(\frac{\pi x}{0.9} - \frac{\pi}{3}\right)\right)^4, & if\ 0.3 \le x \le 1.2 \\ 0\,, & if\ 0 \le x < 0.3\ and\ 1.2 < x \le 1.5 \end{cases} .$$

The upstream and downstream farfields extend 25 units from the viscous wall. The upstream boundary is located at x = -25 and the downstream boundary is located at x = 26.5. The top boundary of the computational domain is located at y = 5. The grids for the bump-in-channel flow have vertex dimensions of 89×41, 177×81, 353×161, 705×321, and 1409×641 in the x- and y-directions, respectively. An image of the viscous wall section of the computational domain for the 177×81 grid can be found in Figure 4. Similarly to the flat plate case, mesh biasing is used in the wall-normal direction and near the leading and trailing edges of the wall region. The finest grid has a minimum wall spacing of 5×10^{-7}, giving an average y^+value of about 0.07. The coarsest mesh has a minimum wall spacing of 8.14×10^{-6}, which gives an average y^+value of about 0.95.

e. Boundary Conditions

The flow cases were run at a Mach number of 0.2. As is noted in [50], the Mach number of the flow is below 0.3, which allows for incompressible treatment. However, the cases' intended use is compressible code verification. According to [50], using an incompressible code may yield results that are close, but not quite the same as the grid is refined.

In an OpenFOAM simulation, the boundary and initial conditions for each flow variable are specified in the case's 0 time directory.

2D Zero pressure gradient flat plate

The flat plate case was run at a Reynolds number (based on a reference length of 1) of 5 million. Figure 5 shows the boundary conditions suggested by NASA for the flat plate flow case. A fixed velocity value of 69.3 m/s, corresponding to a Mach number of 0.2, is used as the inlet boundary condition. For the pressure at the inlet and plate boundaries, OpenFOAM's Neumann-type boundary condition, known as zeroGradient, is assigned. A no-slip boundary condition is used on the adiabatic plate surface for the velocity. The outlet is assigned the zeroGradient condition for the velocity and OpenFOAM's Dirichlet-type boundary condition, fixedValue, for pressure (1 atm). The symmetry boundary condition is applied at $x < 0$ for all variables. The zeroGradient boundary condition is assigned as to the top boundary for all variables. Temperature boundary conditions are not necessary because the simulation is run as incompressible. Boundary conditions for turbulence variables are model-specific and will be discussed in detail in Section III for each turbulence model.

2D Bump-in-channel

The bump-in-channel case was run at a Reynolds number (based on a reference length of 1) of 3 million. Figure 6 shows the boundary conditions suggested by NASA for the bump-in-channel flow case. A fixed velocity value of 69.3 m/s, corresponding to a Mach number of 0.2, is used as the inlet boundary condition. For the pressure at the inlet and wall boundaries zeroGradient is assigned. A no-slip boundary condition is used on the adiabatic wall surface for the velocity. The outlet is assigned the zeroGradient condition for the velocity and fixedValue for pressure (1 atm). The symmetry boundary condition is applied at $x < 0$ and $x > 1.5$ for all variables. The zeroGradient boundary condition is assigned as to the top boundary for all variables.

III. Results & Discussion

Results of the OpenFOAM simulations will be presented in this section. The results obtained with the incompressible EVMs will be compared with those obtained by NASA with their CFL3D and FUN3D CFD codes. The results from CFL3D and FUN3D, available on the Turbulence Modeling Resource website [50], correspond to compressible simulations. The LRR-IP model will be compared with DNS [31] and experimental [22] data. Results for the zero pressure gradient flat plate case will be presented first and the 2D bump-in-channel will follow after. Some of the plots in this section have been nondimensionalized to be in accordance with the source of the data used for comparison.

Flow variable profiles obtained with CFL3D and FUN3D are only available for the finest mesh in each flow case: 545x385 for the flat plate and 1409x641 for the 2D bump. Friction coefficient values obtained from different mesh sizes were analyzed to verify grid convergence of the solution and to determine how OpenFOAM results compared to those obtained with CFL3D and FUN3D. Grid convergence of the solution can be determined by studying the change in the value of the friction coefficient profile at a given point between meshes. The friction coefficient value was obtained near the middle of the flat plate at and at the top of the bump. The corresponding data sampling locations for the flat plate and bump are $x = 0.97$, and $x = 0.75$, respectively. The friction coefficient profile across the entire solid wall region of each flow case was also compared between meshes.

a. Zero Pressure Gradient Flat Plate

Spalart-Allmaras

In addition to the specification of the initial and boundary conditions for the velocity and pressure fields, the SA model requires a definition of the turbulent viscosity, ν_t, and the Spalart-Allmaras variable, $\tilde{\nu}$. The ν_t and $\tilde{\nu}$ values used for the initial and boundary conditions at the farfield and wall regions were calculated as suggested by NASA in [62]:

$$\tilde{\nu} = 3\nu = 4.16 \times 10^{-5}\ \mathrm{m^2/s}$$

$$\nu_t = \tilde{\nu} f_{v1} = \tilde{\nu} \frac{\chi^3}{\chi^3 + c_{v1}^3} = 2.92 \times 10^{-6}\ \mathrm{m^2/s}$$

$$\tilde{\nu}_{wall} = \nu_{t\,wall} = 0\ \mathrm{m^2/s}$$

where

$$\chi = \frac{\tilde{\nu}}{\nu}.$$

The ν_t values calculated above were used for the SST and k-ω cases.

Flow profiles for the eddy viscosity, mean velocity, and friction coefficient are shown in Figures 7a-d. Some of these plots have been nondimensionalized to be in accordance with NASA's website. Figure 7a shows the mean velocity profile nondimensionalized by the freestream velocity value as a function of distance normal to the plate. The dimensionless flow velocity, u^+, is plotted as a function of the nondimensional wall distance, y^+, in Figure 7b. The nondimensional velocity is defined as

$$u^+ \equiv \frac{u}{u_\tau},$$

where u is the mean value of the flow velocity and u_τ is the friction velocity. Figure 7c shows the eddy viscosity profile as a function of the distance normal to the wall. Figure 7d shows the skin friction coefficient profile across the entire plate. OpenFOAM results for flow over a flat plate that were obtained with the SA model are in agreement with CFL3D and FUN3D for all of the profiles.

Figure 8a shows the value of C_f obtained at $x = 0.97$ with the SA model for all of the meshes. Each marker on Figure 8a represents a mesh. On the horizontal axis of Figure 8a, the variable h represents the characteristic mesh length and N is the number of elements in the mesh. For the finest mesh, there is a 1.0% difference between the friction coefficient value calculated with OpenFOAM's incompressible solver and those calculated with CFL3D's and FUN3D's compressible solvers. However, when incompressible solver results are compared, the difference between OpenFOAM and FUN3D is only 0.16%. The friction coefficient profiles across the entire plate for each mesh, depicted in Figure 8b, show that the variation of the profile obtained with the SA model from mesh to mesh is negligible for the flat plate case.

SST

In addition to the specification of the initial and boundary conditions for the velocity and pressure fields, the SST model requires a definition of the turbulent viscosity, ν_t, turbulence kinetic energy, k, and specific dissipation rate, ω. The initial and boundary condition values for the turbulence kinetic energy and specific dissipation rate at the farfield and wall regions were calculated in accordance with [63]:

$$k_{farfield} = 9 \times 10^{-9} a^2 = 1.08 \times 10^{-3} \frac{\mathrm{m}^2}{\mathrm{s}^2} \tag{23}$$

$$k_{wall} = 0 \ \mathrm{m}^2/\mathrm{s}^2$$

$$\omega_{farfield} = 1 \times 10^{-6} \frac{\rho\, a^2}{\mu} = 8657.5 \ \mathrm{s}^{-1} \tag{24}$$

$$\omega_{wall} = 10 \frac{6\nu}{\beta_1 (\Delta d_1)^2} = 4.43 \times 10^{10} \ \mathrm{s}^{-1}. \tag{25}$$

In Eqs.23-25 a represents the local speed of sound, ρ represents the fluid's density, μ represents the fluid's dynamic viscosity, ν represents the fluid's kinematic viscosity, β_1 is a model constant with a value of 0.075, and Δd_1 represents the distance from the wall to the nearest grid point.

Flow profiles for the mean velocity, eddy viscosity, skin friction coefficient, turbulence kinetic energy, and specific dissipation rate are shown in Figures 9a-f. Figures 9a-d are arranged in the same order as they were for the SA results. Nondimensional flow profiles for the turbulent kinetic energy and specific dissipation rate are shown in Figures 9e,f. Similarly to the SA results, OpenFOAM results obtained with the SST model are in agreement with CFL3D and FUN3D for all of the profiles. There is a small discrepancy between results obtained with OpenFOAM and CFL3D/FUN3D for values of k+ and ω very close to the wall. The variable k+ represents nondimensional turbulent kinetic energy and it is defined as

$$k^+ = \frac{k}{u_\tau^2}.$$

In the specific dissipation rate profile, shown in Figure 9f, the OpenFOAM result is in better agreement with CFL3D than FUN3D.

Figure 10a shows the value of C_f obtained at $x = 0.97$ with the SST model for all of the meshes. For the finest mesh, there is a 1.3% difference between the results obtained with OpenFOAM's incompressible solver and those obtained with CFL3D's and FUN3D's compressible solvers. The difference between results obtained with OpenFOAM's and FUN3D's incompressible solvers is 0.27%. A comparison of the friction coefficient profile for each mesh can be found in Figure 10b, which shows that the variation of the profile obtained with the SST model much more pronounced than for the SA model.

k-ω

The initial and boundary values for the turbulence variables for the k-ω simulation were the same as those used in the SST case. Flow profiles for the mean velocity, eddy viscosity, skin friction coefficient, turbulence kinetic energy, and specific dissipation rate and are shown in Figure 11a-f. The flow profiles are arranged in the same order as they were in the SST results. Results obtained with the k-ω model in OpenFOAM are in agreement with CFL3D and FUN3D for all of the profiles. Similarly to the SST results, a small discrepancy is seen between OpenFOAM and CFL3D/FUN3D for values of k+ and ω very close to the wall. Contrary to the SST case, the OpenFOAM solution is in better agreement with the near wall results obtained with FUN3D for the specific dissipation rate plot shown in Figure 11f.

Figure 12a shows the value of C_f obtained at $x = 0.97$ with the k-ω model for all of the meshes. Only results calculated with CFL3D's and FUN3D's compressible solvers were available for the k-ω model. On the finest mesh, there is a 1.0% difference between OpenFOAM's incompressible and CFL3D/FUN3D compressible results. The incompressible results obtained with OpenFOAM's k-ω should presumably be within a fraction of a percent of FUN3D's incompressible solution, if provided. The reason supporting this claim is that a percentage difference of about 1.0% was seen between CFL3D/FUN3D compressible results and OpenFOAM's incompressible results for the SA and SST cases, but the difference between incompressible solvers was on the order of a fraction of a percent. A comparison of the friction coefficient profiles for all meshes is shown in Figure 12b. The variation of the profile obtained with the k-ω model from mesh to mesh is very similar to that corresponding to SST.

LRR-IP

As was previously mentioned, data for the LRR-IP model was not available on NASA's Turbulence Modeling Resource website. As a result, OpenFOAM results were compared to DNS [31] and experimental [22] data. The DNS and experiment were carried out at a momentum-thickness-based Reynolds number of 5200. Using the Reynolds number and values of measured flow variables found in [22], the flow variable values for the OpenFOAM simulation were calculated using the following relationships (See Table 6 for values):

$$\nu = \frac{\theta U_{\infty}}{Re_{\theta}}$$

$$k = \frac{3}{2}\left(U_{\infty} I\right)^{2}$$

$$\varepsilon = \frac{C_{\mu} k^{3/2}}{l}$$

$$\overline{u_i' u_i'} = \frac{2}{3} k \text{ and } \overline{u_i' u_j'} = 0$$

where *I* is the turbulence intensity and *l* is a characteristic length scale. The values of *I* and *l* depend on the wind tunnel where the experiment was conducted.

$\boldsymbol{\nu}$ (m^2/s)	$\boldsymbol{\rho}$ (kg/m^3)	$\boldsymbol{U}_{\infty}$ (m/s)	$\boldsymbol{k}$ (m^2/s^2)	$\boldsymbol{\varepsilon}$ (m^2/s^3)	$\overline{\boldsymbol{u_i' u_i'}}$ (m^2/s^2)
1.53×10^{-5}	1.19	18.95	2.16×10^{-3}	0.842	1.44×10^{-3}

Table 6: Flow variable values for LRR-IP simulation

Figures 13a-f compare results obtained with OpenFOAM against those published in [22] (shown as Exp) and [31] (shown as DNS). All profiles have been nondimensionalized to enable the inclusion of the DNS data because the DNS was performed at a mean flow velocity that was much smaller than the experiment and OpenFOAM cases (see [22] for details). There is close agreement between OpenFOAM and experimental/DNS data for the velocity profiles shown in Figure 13a. However, there is a noticeable disagreement between OpenFOAM and the experimental/DNS data in the rest of the plots. For example, Figure 13b shows that OpenFOAM results have much lower u^{+} values than the experimental/DNS data in the wake region of the boundary layer. Each of the Reynolds stress profiles in Figures 13c-f were nondimensionalized by the value of u_{τ}^{2} corresponding to each flow case. OpenFOAM doesn't seem to be able to produce the peaks seen in Figures

13c,f, which could be a cause of the results' disagreement. The $\overline{u_1'u_1'}$ and $\overline{u_3'u_3'}$ profiles obtained with OpenFOAM show close agreement with the experimental and DNS data with the exception of the absence of the large peaks. Data for $\overline{u_3'u_3'}$ was only available for the DNS. The author of this document believes that the main contributor to the discrepancy is OpenFOAM's overestimation of $\overline{u_2'u_2'}$ very close to the wall, which can be seen in Figure 13e. This error propagates to $\overline{u_1'u_2'}$, shown in Figure 13d, which is closely related to the wall shear stress used to define u_τ. The overestimation of $\overline{u_2'u_2'}$ causes the value of u_τ to be larger than it should, which leads to the aforementioned u^+ deficit. The reason for the overestimation of $\overline{u_2'u_2'}$ seems to be inherent to OpenFOAM's LRR turbulence model because similar results are obtained when the default model is used, even if wall functions are used. The cause for the difference in the results is not fully understood.

b. 2D Bump-In-Channel

Spalart-Allmaras

An approach similar to that of the flat plate was used to define initial and boundary conditions for the bump. The value of some of the flow variables is slightly different because the bump simulation was performed at a length based Reynolds number of 3 million instead of the 5 million value used for the plate. The farfield and wall values of ν_t and $\tilde{\nu}$ were calculated as suggested by NASA in [62]:

$$\tilde{\nu} = 3\nu = 6.93 \times 10^{-5}\ \mathrm{m^2/s}$$

$$\nu_t = \tilde{\nu} f_{v1} = \tilde{\nu}\frac{\chi^3}{\chi^3 + c_{v1}^3} = 2.08 \times 10^{-7}\ \mathrm{m^2/s}$$

$$\tilde{\nu}_{wall} = \nu_{t\,wall} = 0\ \mathrm{m^2/s}$$

where

$$\chi = \frac{\tilde{\nu}}{\nu}.$$

The ν_t values calculated above were used for the SST and k-ω cases.

Profiles for the mean velocity, eddy viscosity, and skin friction coefficient are shown in Figures 14a-d. Results for all profiles were obtained at $x = 0.75$. An additional velocity profile at an x-location of 1.20148 is also used to assess the accuracy of OpenFOAM's results. The velocity profiles shown in Figures 14a,b are in close agreement with CFL3D and FUN3D results. On Figure 14a, the variable y_o on the vertical axis represents the height of the bump. The results for the eddy viscosity and skin friction coefficient profiles, shown in Figures 14c,d, are slightly different at the maximum value of both profiles. OpenFOAM overestimates the eddy viscosity by 1.9%. Figure 14d shows oscillations at $x = 0.75$ and an over prediction of the friction coefficient on the downstream side of the bump. Comparing the skin friction coefficient results obtained on the 1409x641 with what was obtained on the 705x321 mesh, shown in Figure 14e, it can be concluded that the over prediction downstream of the bump appears to be inherent to the 1409x641 mesh. The oscillations seen at the top of the bump are also reduced in the profile obtained on the 705x321 mesh.

Figure 15a shows the value of C_f that was obtained with the SA model for all of the meshes. Each marker on Figure 15a represents the C_f value obtained at $x = 0.75$ for each mesh. On the finest mesh, there is a 1.4% difference between OpenFOAM and CFL3D and FUN3D compressible results. Incompressible results were not available for CFL3D and FUN3D so they could not be compared with OpenFOAM. The large percent difference between the finest mesh can be attributed to the oscillations seen at the top of the bump on Figure 14d. The difference between the C_f value obtained with OpenFOAM and CFL3D/FUN3D on the 705x321 mesh is 0.45%, which is significantly less than the difference for the 1409x641 result. The evolution of the skin friction coefficient profile as the mesh is refined is shown in Figure 15b. The SA model's results for flow over the 2D bump shows greater sensitivity of to the mesh size than it did for the flat plate.

SST

The farfield and wall turbulence kinetic energy and specific dissipation rate values were calculated in accordance with [63]:

$$k_{farfield} = 9 \times 10^{-9} a^2 = 1.08 \times 10^{-3} \frac{\mathrm{m}^2}{\mathrm{s}^2} \quad (\mathbf{26})$$

$$k_{wall} = 0\ \mathrm{m}^2/\mathrm{s}^2$$

$$\omega_{farfield} = 1 \times 10^{-6} \frac{\rho\, a^2}{\mu} = 5220.8\ \mathrm{s}^{-1} \quad (\mathbf{27})$$

$$\omega_{wall} = 10 \frac{6\nu}{\beta_1 (\Delta d_1)^2} = 7.39 \times 10^{10}\ \mathrm{s}^{-1}. \quad (\mathbf{28})$$

In Eqs.26-28 a represents the local speed of sound, ρ represents the fluid's density, μ represents the fluid's dynamic viscosity, ν represents the fluid's kinematic viscosity, β_1 is a model constant with a value of 0.075, and Δd_1 represents the distance from the wall to the nearest grid point.

Flow profiles for the mean velocity, eddy viscosity, skin friction coefficient, turbulence kinetic energy, and specific dissipation rate are shown in Figures 16 a-f. The overall trends in the OpenFOAM results obtained with the SST model on the 1409x641 mesh are in agreement with CFL3D and FUN3D for all of the profiles. However, the values in Figures 16b-d are not exactly the same as the ones obtained with NASA's software. The velocity profile downstream of the bump, shown on Figure 16b, has the same shape as the one obtained with CFL3D but it seems to be shifted to the right. The eddy viscosity shape is in agreement with CFL3D and FUN3D but OpenFOAM under predicts the values on this mesh. The difference in the eddy viscosity value between OpenFOAM and CFL3D/FUN3D on the 1409x641 mesh is 4.6%. A similar over-predictive behavior downstream of the bump observed in the SA skin friction coefficient results is also present in SST results. The oscillations seen at the top of the bump on the skin friction coefficient plot for the SA model are almost nonexistent for SST but the value is over estimated. There is a small discrepancy between OpenFOAM and CFL3D/FUN3D for values of k+ very close to the wall. In the specific dissipation profile, shown in Figure 16f, the OpenFOAM solution is in better agreement with NASA's codes near the wall for the bump than it was for the flat plate. The same profiles shown in Figure 16 are shown in Figure 17 for the 705x321 mesh. The difference in eddy viscosity value for this mesh is 1.0%, which is considerably less than the difference corresponding to the result obtained on the 1409x641 mesh. Based on the agreement between OpenFOAM and NASA's codes shown in Figure 17, it has been determined that the discrepancy between results shown in Figure 16 may be caused by solver limitations in OpenFOAM. Personal communication with the NASA employee in charge of the TMR website revealed that a similar problem has been encountered by other researchers on the 1409x641 mesh.

Figure 18a shows the value of C_f that was obtained with the SST model for all of the meshes. Each marker on Figure 18a represents the C_f value obtained at $x = 0.75$ for each mesh. On the finest mesh, the results show a 1.3% difference between OpenFOAM's incompressible solver and CFL3D's and FUN3D's compressible solvers. The large percent difference between the finest mesh can be attributed to the over prediction of the skin friction coefficient value seen at the top of the bump on Figure 16d. The difference between the C_f values obtained with OpenFOAM and CFL3D/FUN3D on the 705x321 mesh is 0.88%, which is less than the difference for the 1409x641 result. The evolution of the skin friction coefficient profile with mesh size is shown in Figure 18b. Similar sensitivity to mesh size on the friction coefficient profile was seen for flows over the flat plate and the bump when using the SST model.

k*-*ω

The initial and boundary values for the turbulence variables for the *k*-*ω* simulation were the same as those used in the SST case. The same profiles shown in the previous section are shown in Figure 19 for the 1409x641 mesh. The velocity profiles obtained with OpenFOAM using the *k*-*ω* model, shown in Figures 19a,b, are in agreement with CFL3D and FUN3D. There is an over prediction of 1.7% in the eddy viscosity profile shown in Figure 19c. The skin friction coefficient profile for the *k*-*ω* model is shown in Figure 19d. The oscillations seen at the top of the bump for the skin friction coefficient in the SA and SST results aren't present in the results obtained with *k*-*ω*. The over prediction of the skin friction coefficient downstream of the bump is reduced when using the *k*-*ω* model, but it is not eliminated completely. The *k*+ values near the wall in the profile shown in Figure 19e deviate from CFL3D and FUN3D results as they did for the SST case. The specific dissipation rate profile shown in Figure 19f matches CFL3D and FUN3D very closely. Profiles obtained on the 705x341 mesh have also been provided for comparison in Figure 20. The main difference between the results obtained on the 705x341 and the 1409x641 meshes is seen in the eddy viscosity plot on Figure 20c. The result corresponding to the 705x341 mesh is over predicted by 3.5%, which makes the error about twice as large as what was obtained on the 1409x641 mesh. Similarly to the SA and SST results, the skin friction coefficient profile calculated on the 705x341 mesh, shown in Figure 20d, does not show an over prediction downstream of the bump. Profiles for k+ and ω shown in Figures 20e,f] match CFL3D and FUN3D results.

Figure 21a shows the value of C_f that was obtained with the k-ω model for all of the meshes. Each marker on Figure 21a represents the C_f value obtained at $x = 0.75$ for each mesh. There is a 0.97% difference between results obtained with OpenFOAM’s incompressible solver and CFL3D’s and FUN3D’s compressible solvers on the 1409x641 mesh. The percent difference corresponding to the k-ω results is less those corresponding to SA and SST results. The difference between the C_f values obtained with OpenFOAM and CFL3D/FUN3D on the 705x321 mesh is 0.92%. The evolution of the skin friction coefficient profile with mesh size is shown in Figure 21b. The sensitivity to mesh coarseness is greater for the k-ω model than for SA and SST results. Comparing the change in the friction coefficient profile between the coarsest and finest mesh for each model verifies the previous claim.

IV. Conclusion

Computational fluid dynamics simulations were performed with OpenFOAM for two different benchmark flow cases developed by the TBMWG and NASA. The flow cases were a zero pressure gradient boundary layer over a flat plate and flow over a two-dimensional bump in a channel. Five nested meshes for each flow case were obtained from NASA's Turbulence Modeling Resource website. The results obtained with OpenFOAM were compared with those obtained with high-fidelity NASA codes CFL3D and FUN3D.

Flow simulations for the zero pressure gradient flat plate were run with the Spalart-Allmaras, SST, k-ω, and LRR-IP turbulence models. Only the SA, SST, and k-ω results were available for comparison on NASA's TMR website. Mean velocity, eddy viscosity, skin friction coefficient, turbulent kinetic energy, and specific dissipation profiles that were obtained with OpenFOAM for incompressible flow over the zero pressure gradient on the 545x385 mesh were in agreement with NASA's compressible results. Mesh convergence results showed that the largest difference in skin friction coefficient that was observed between OpenFOAM's incompressible results and NASA's compressible results corresponded to the SST simulation and it was 1.3%. The difference for the SA and k-ω models was of 1%. The difference in incompressible-to-incompressible results for the SA and SST models was of 0.16%, and 0.27%, respectively. Results obtained with the LRR-IP model in OpenFOAM were compared with experimental and DNS data. The velocity profile was in agreement with experimental and DNS results but discrepancies were observed in the y^+-u^+ profile and in all Reynolds stress profiles. The cause of the discrepancies is still not fully understood but it appears to be inherent to OpenFOAM's default LRR turbulence model.

Flow simulations for the 2D bump-in-channel were run with the Spalart-Allmaras, SST, and k-ω turbulence models. Only compressible results were available for this case on NASA's TMR website. The overall trends for the mean velocity, eddy viscosity, skin friction coefficient, turbulent kinetic energy, and specific dissipation profiles that were obtained with OpenFOAM were in agreement with NASA. However, an over prediction of the skin friction coefficient was seen on the top and downstream regions of the bump for all models on the 1409x641 mesh. The difference in the skin friction coefficient value for the SA, SST, and k-ω models obtained using the finest mesh was 1.4%, 1.3%, and 0.97%, respectively. On the 705x321 mesh the difference in skin friction coefficient values decreased to 0.45%, 0.88%, and 0.92% for the SA, SST, and k-ω models. It was concluded that the difference on the 1409x641 mesh was caused by OpenFOAM's solver limitations. A slight inconsistency was also observed in the eddy viscosity profile for all turbulence models.

The inconsistencies that were documented for both flow cases are small and could be attributed to slight differences in simulation parameter values (explicit values were not provided by NASA), differences in solver algorithms, and most importantly, due to the fact that incompressible results obtained with OpenFOAM are being compared to compressible results obtained with CFL3D and FUN3D. The agreement between OpenFOAM results and NASA results confirm OpenFOAM's capability to produce accurate results for benchmark flows.

Figures

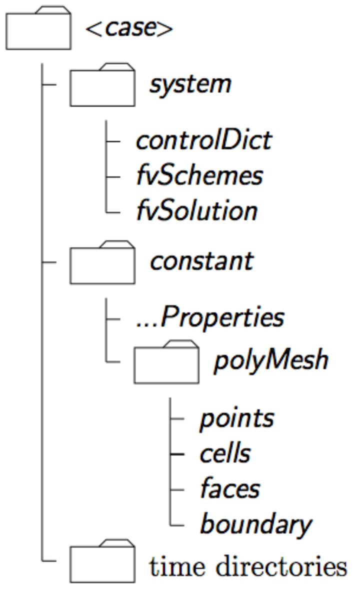


Figure 1: OpenFOAM case directory structure [66]

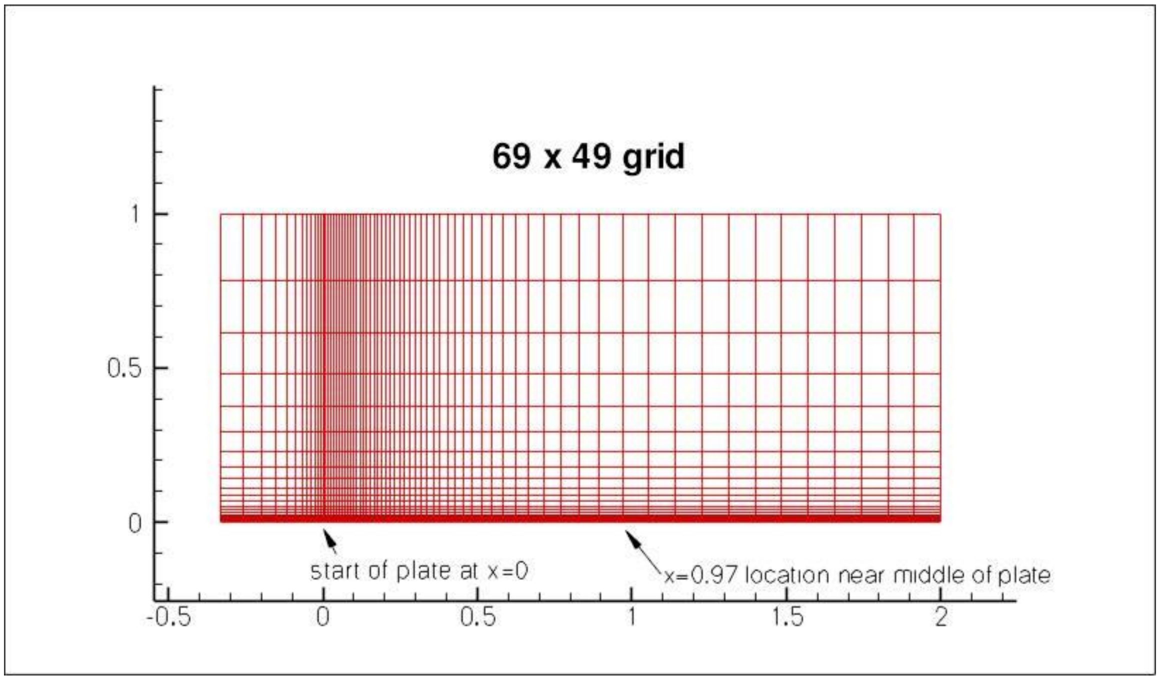


Figure 2: Zero pressure gradient flat plate mesh (69x49) [71]

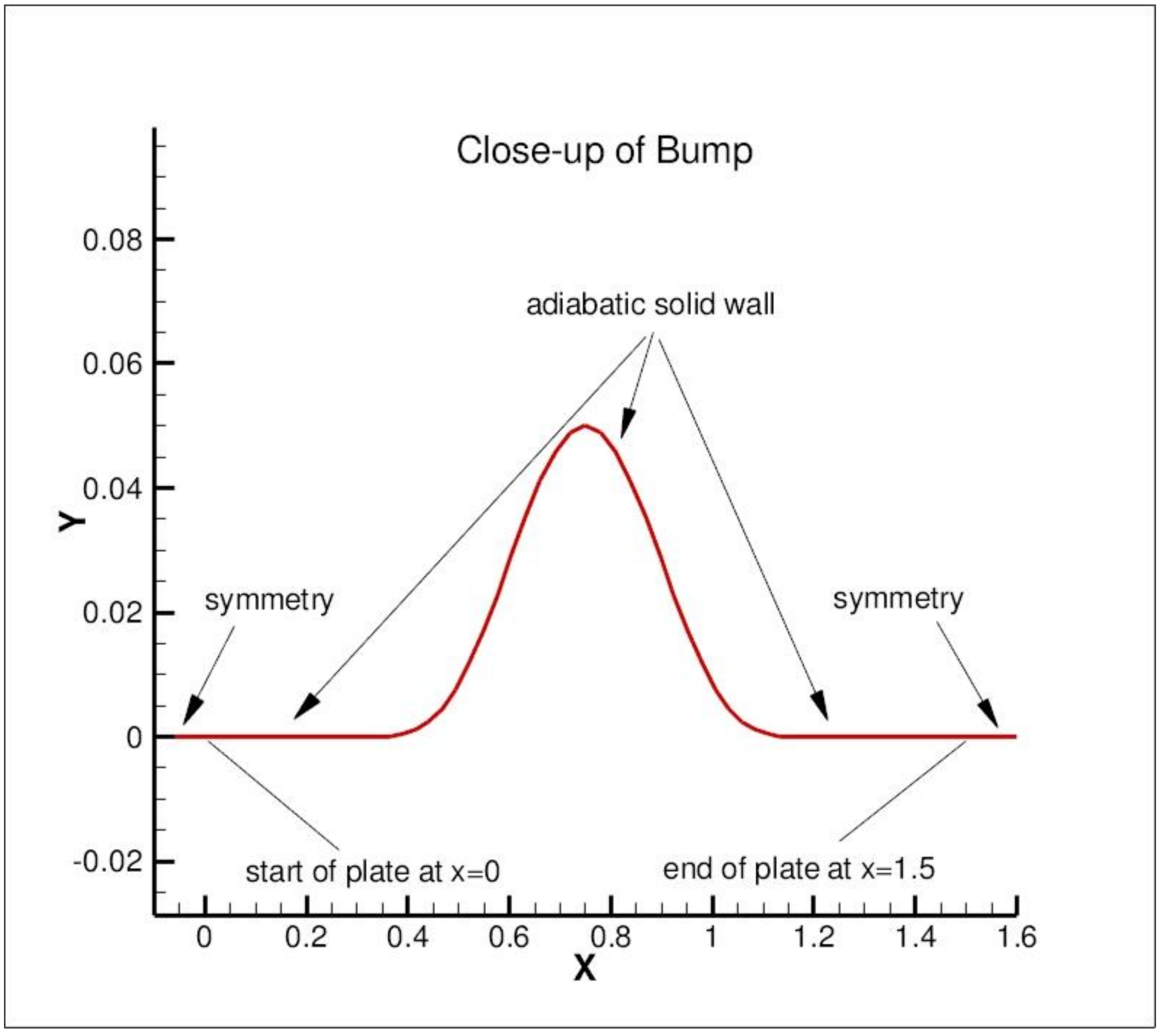


Figure 3: Enlarged view of bump profile [74]

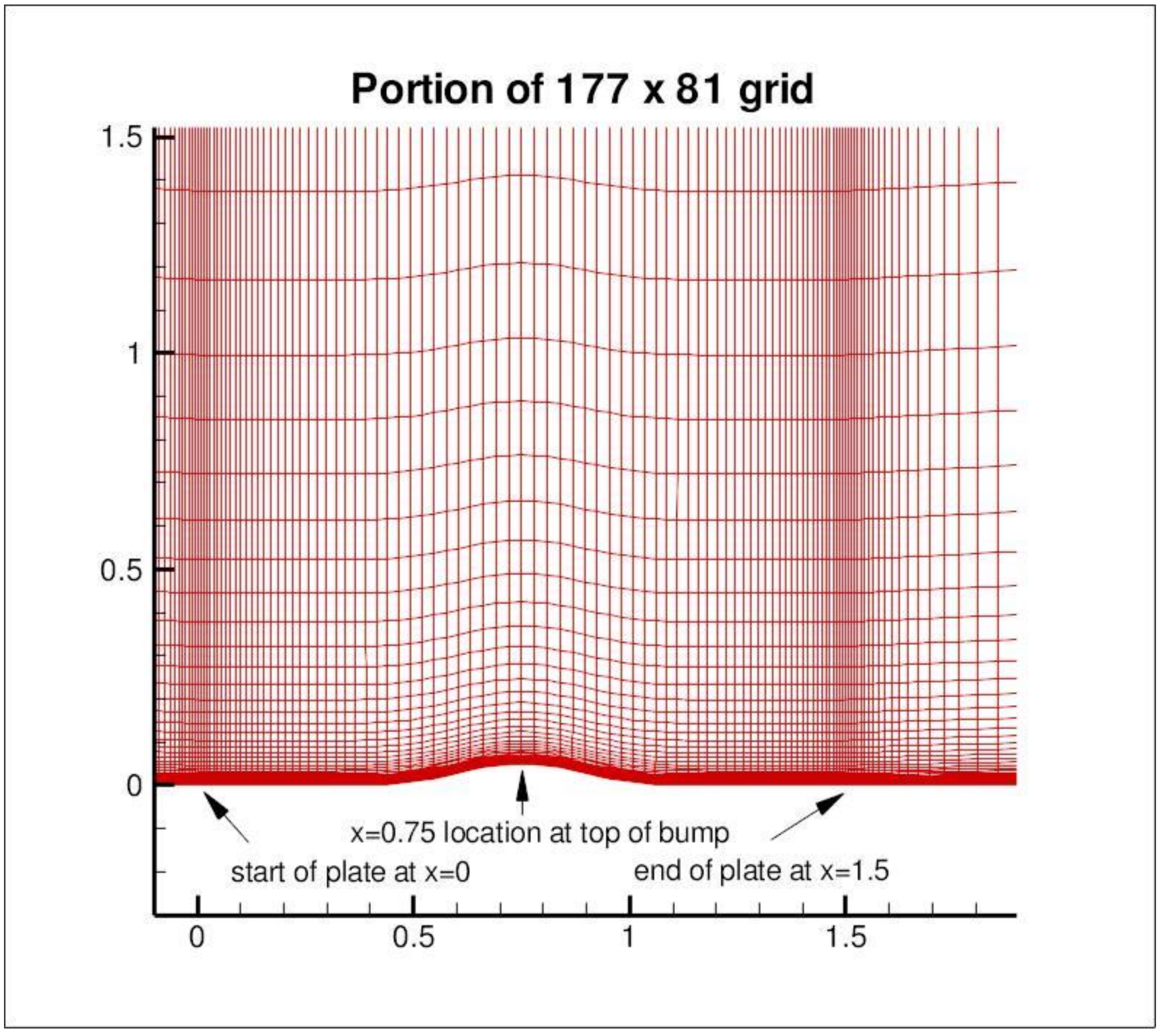


Figure 4: 2D bump-in-channel mesh (177x81) [73]

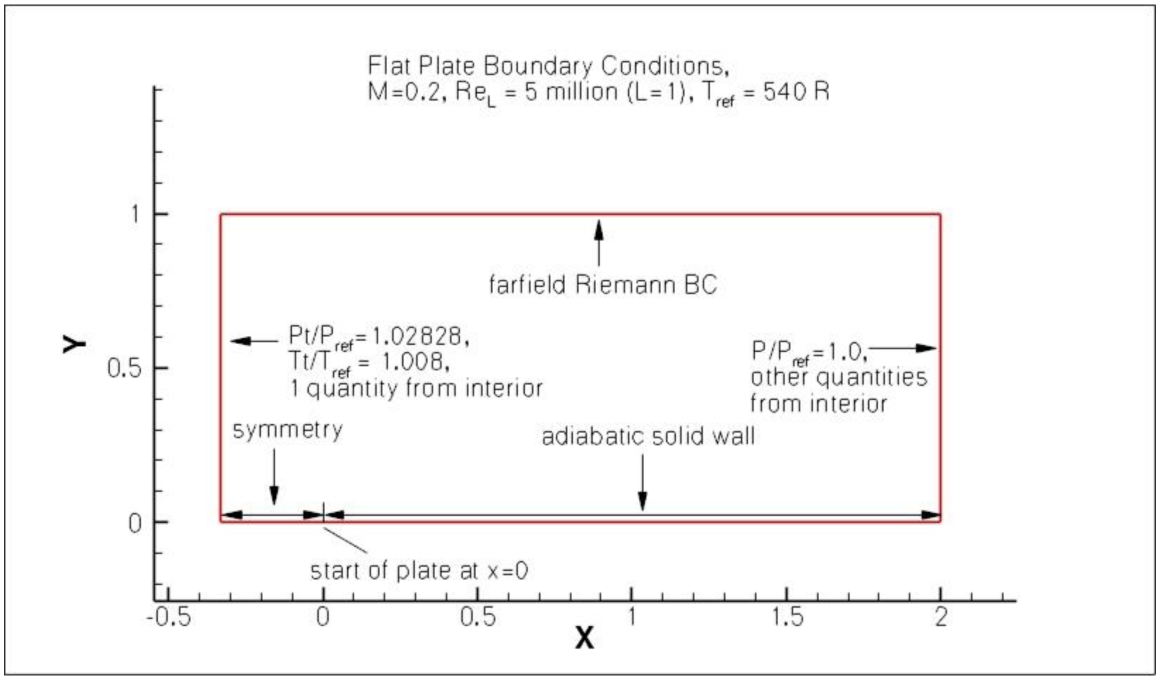


Figure 5: Boundary conditions for ZPG flat plate [72]

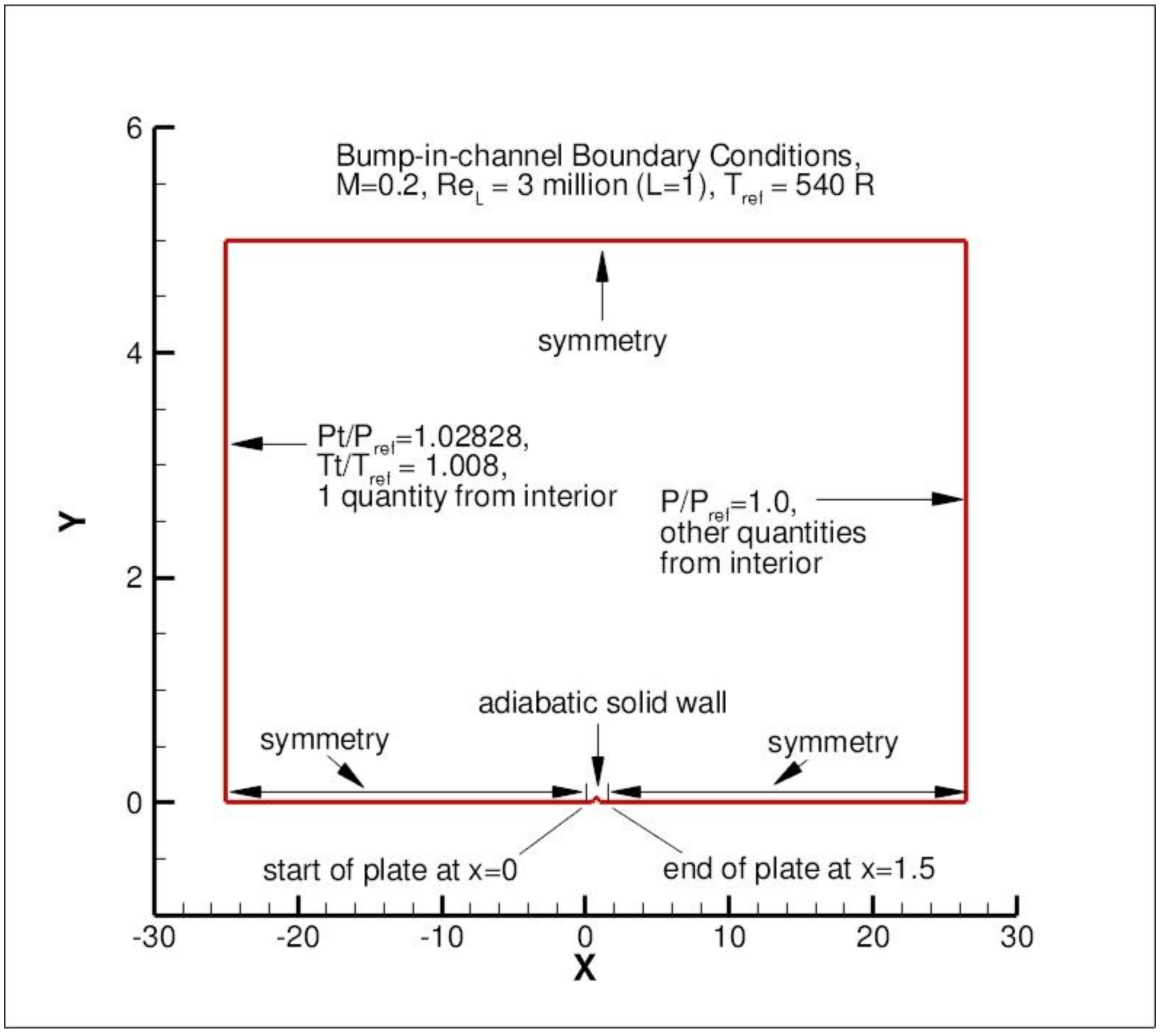
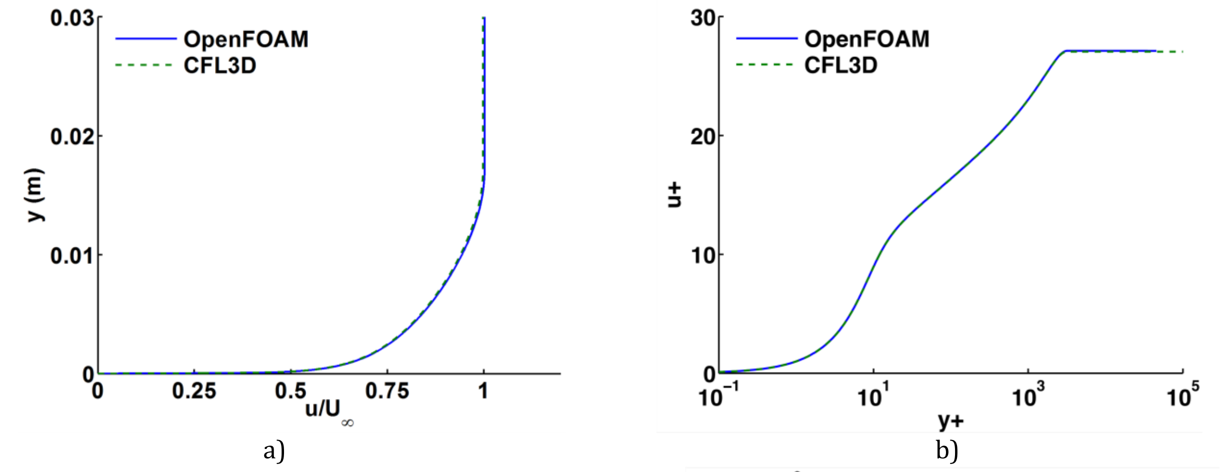


Figure 6: Boundary conditions for 2D bump-in-channel [74]



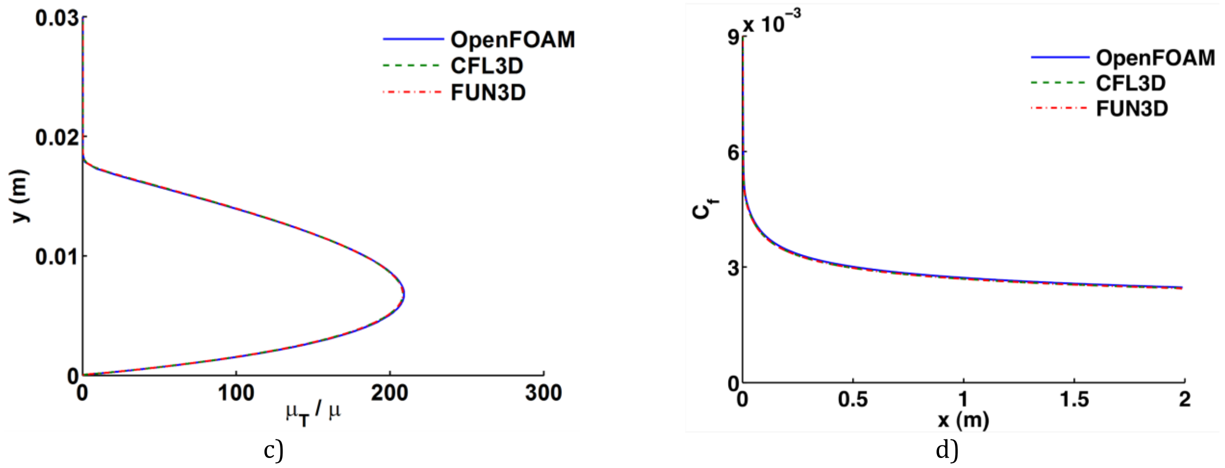


Figure 7: Results of the ZPG flat plate flow simulations with the Spalart-Allmaras turbulence model for OpenFOAM (blue), CFL3D (dashed green) and FUN3D (dashed red) at *x*=0.97:
a) mean velocity profile, b) dimensionless velocity profile, c) dimensionless eddy viscosity profile, d) skin friction coefficient profile

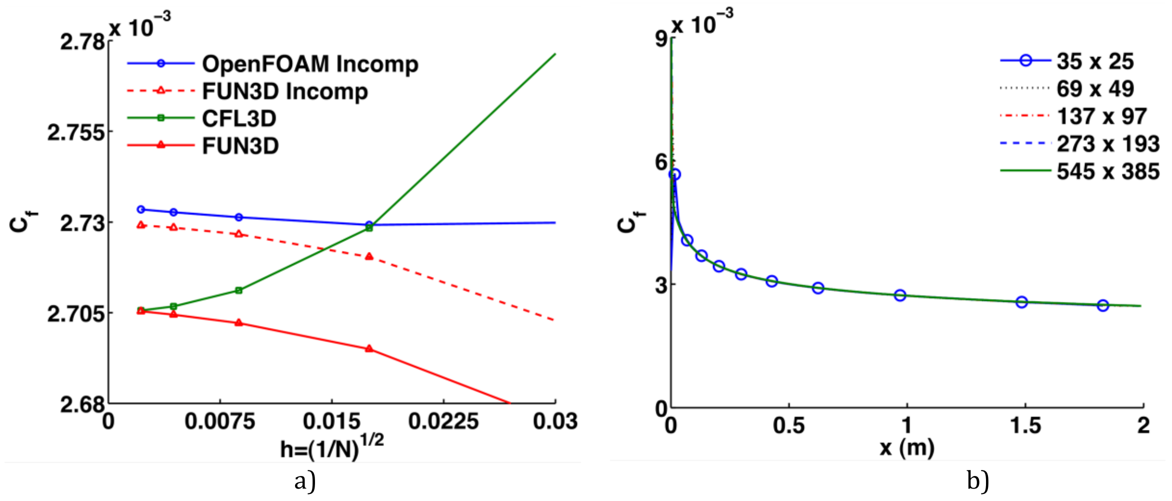
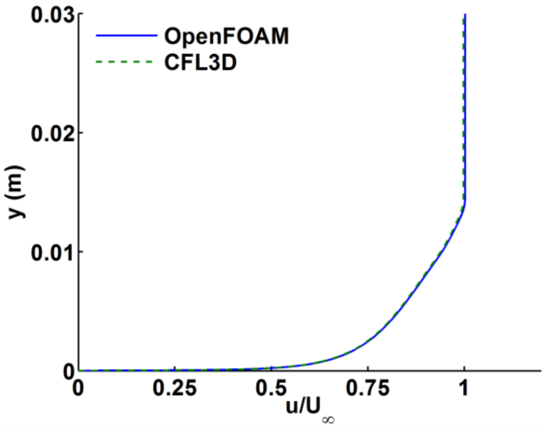
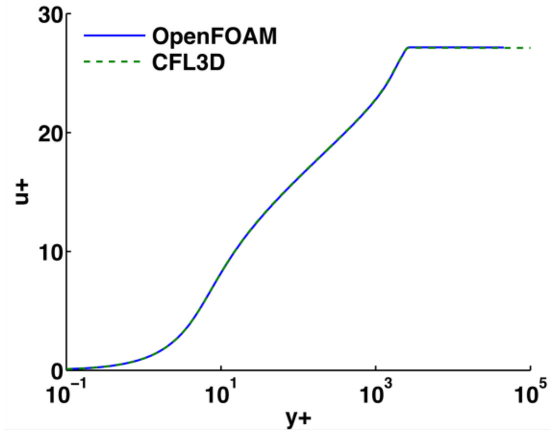


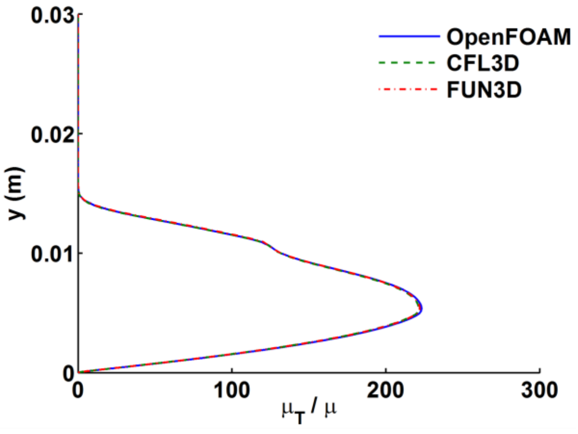
Figure 8: Grid convergence results in a ZPG flat plate flow for the SA model:
a) skin friction coefficient comparison between OpenFOAM (blue) and NASA codes (red, green) at *x*=0.97,
b) skin friction coefficient profiles obtained with OpenFOAM for all meshes



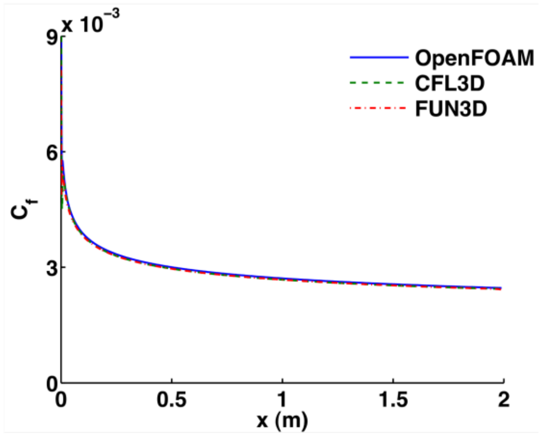
a)



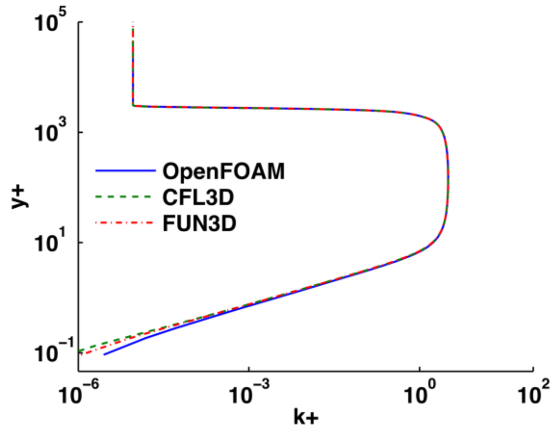
b)



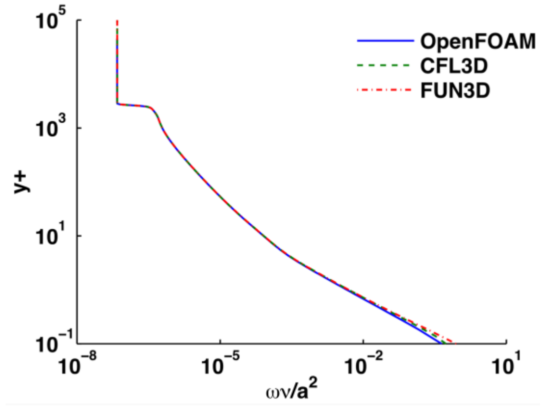
c)



d)

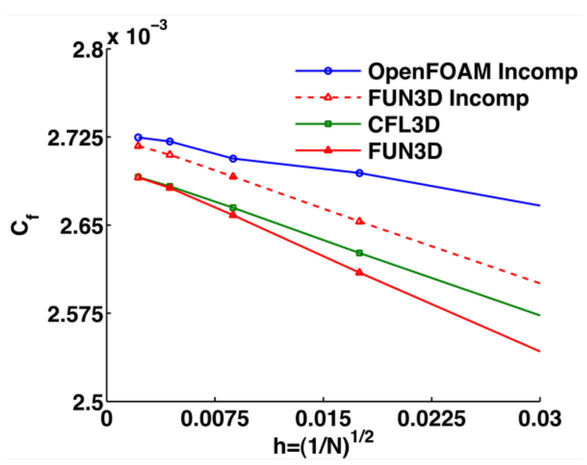


e)

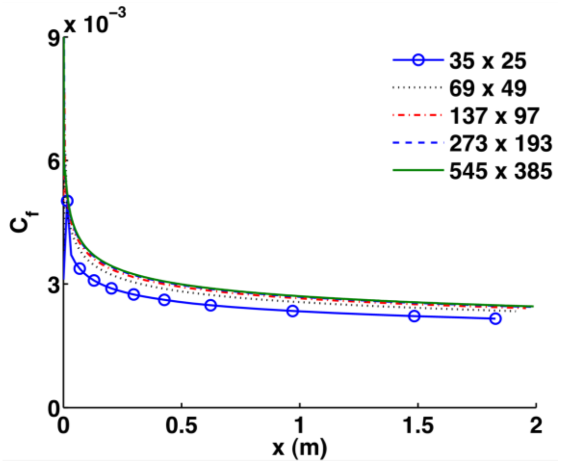


f)

Figure 9: Results of the ZPG flat plate flow simulations with the SST turbulence model for OpenFOAM (blue), CFL3D (dashed green) and FUN3D (dashed red) at *x*=0.97:
a) mean velocity profile, b) dimensionless velocity profile, c) dimensionless eddy viscosity profile, d) skin friction coefficient profile, e) dimensionless turbulent kinetic energy profile, f) specific dissipation rate profile

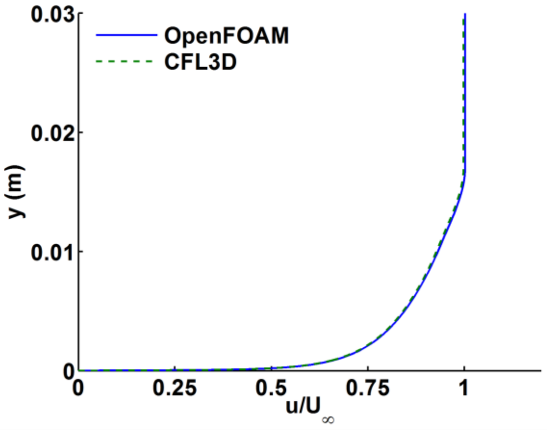


a)

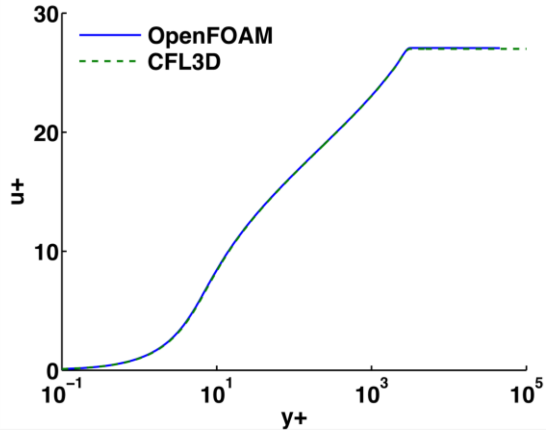


b)

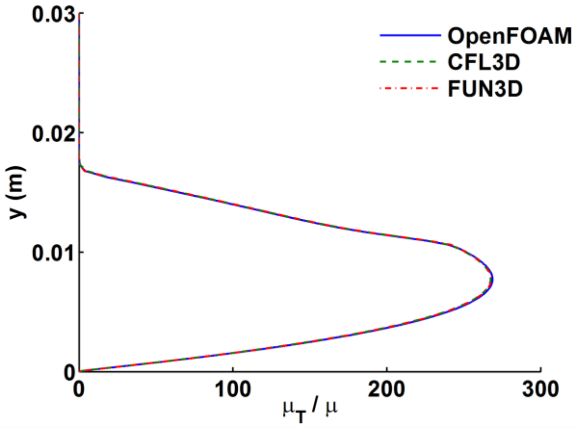
Figure 10: Grid convergence results in a ZPG flat plate flow for the SST model:
a) skin friction coefficient comparison between OpenFOAM (blue) and NASA codes (red, green) at *x*=0.97,
b) skin friction coefficient profiles obtained with OpenFOAM for all meshes



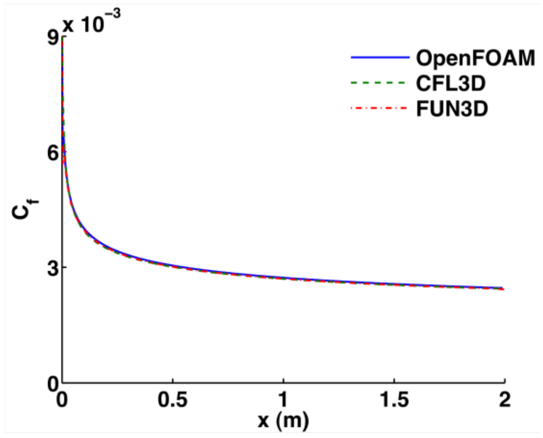
a)



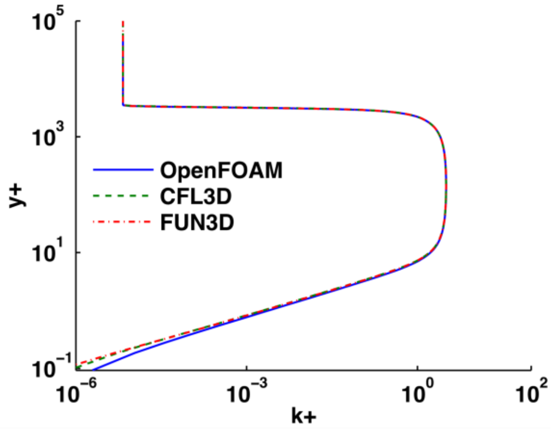
b)



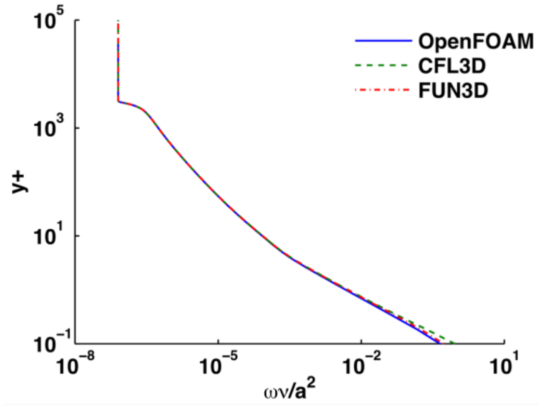
c)



d)

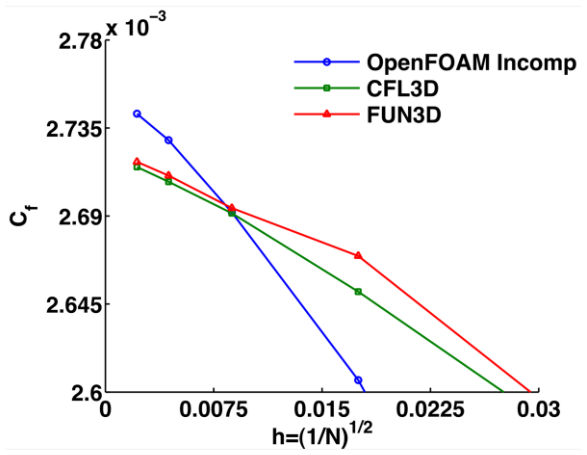


e)

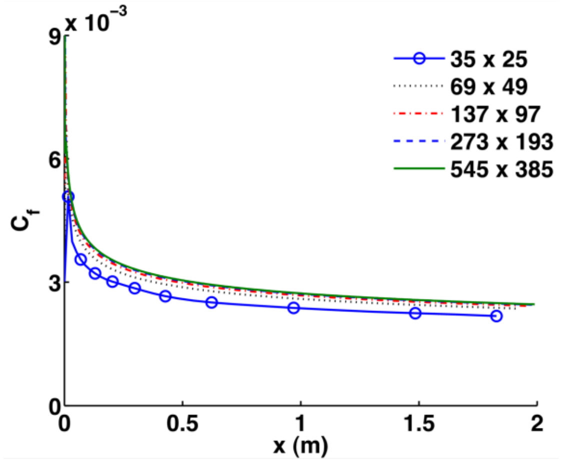


f)

Figure 11: Results of the ZPG flat plate flow simulations with the *k*-*ω* turbulence model for OpenFOAM (blue), CFL3D (dashed green) and FUN3D (dashed red) at *x*=0.97:
a) mean velocity profile, b) dimensionless velocity profile, c) dimensionless eddy viscosity profile, d) skin friction coefficient profile, e) dimensionless turbulent kinetic energy profile, f) specific dissipation rate profile

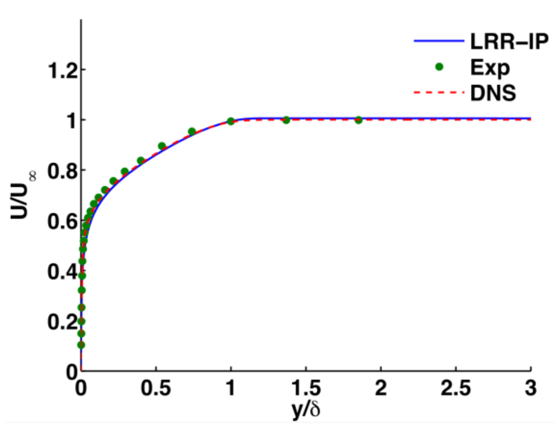


a)

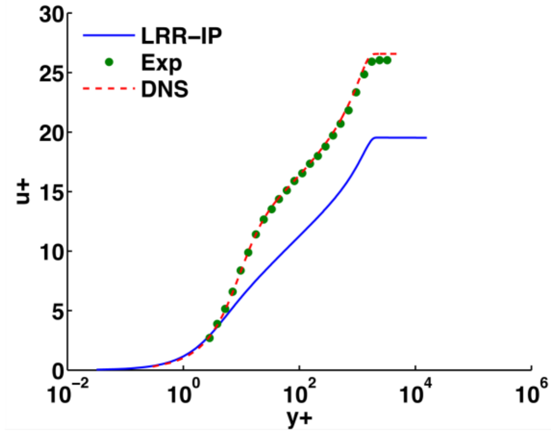


b)

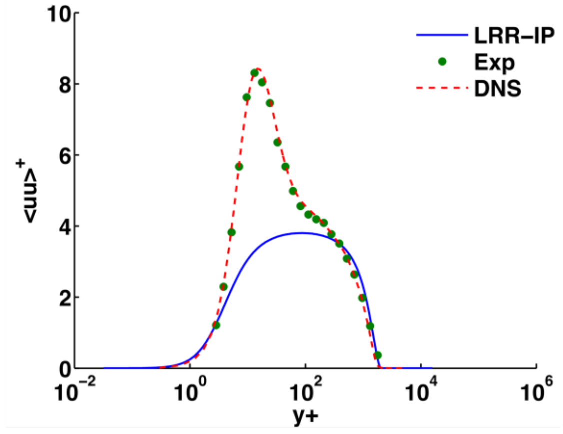
Figure 12: Grid convergence results in a ZPG flat plate flow for the *k*-*ω* model:
a) skin friction coefficient comparison between OpenFOAM (blue) and NASA codes (red, green) at *x*=0.97,
b) skin friction coefficient profiles obtained with OpenFOAM for all meshes



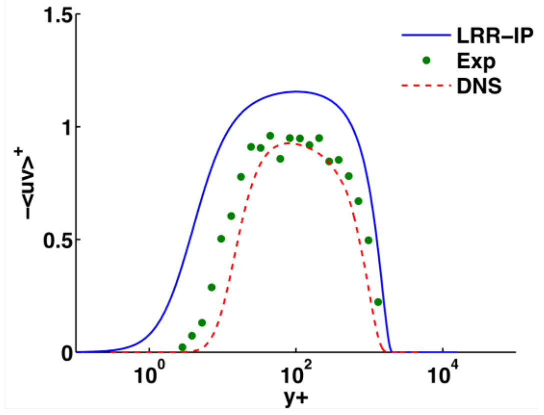
a)



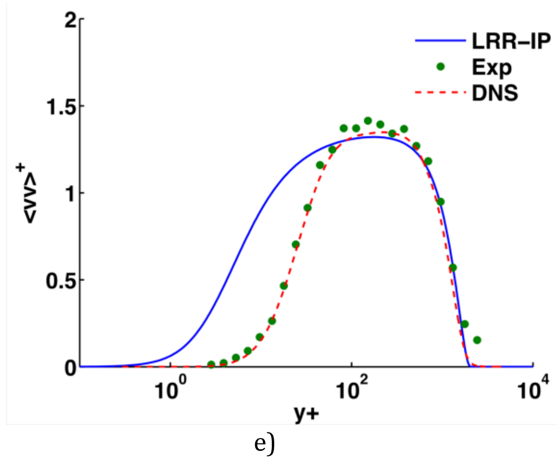
b)



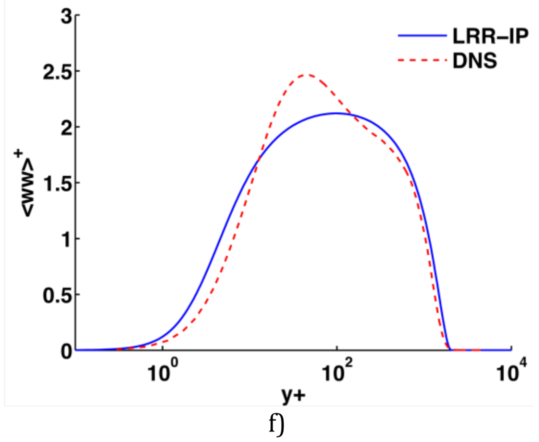
c)



d)

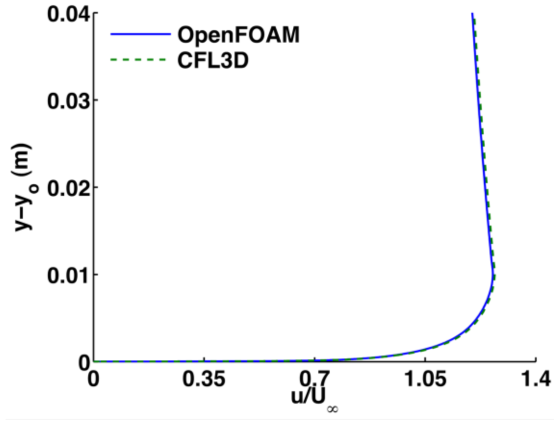


e)

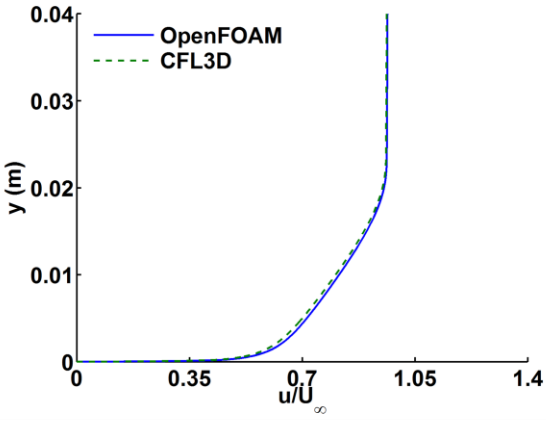


f)

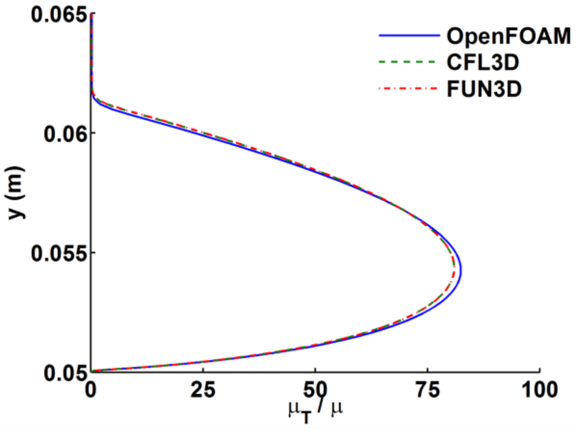
Figure 13: OpenFOAM (blue) results of the ZPG flat plate flow simulations with the LRR turbulence model compared with DNS[31] (dashed red) and experimental results[22] (green circles) at *x*=0.97: a) mean velocity profile, b) dimensionless velocity profile, c) $\overline{u_1'u_1'}^+$ profile, d) $\overline{u_1'u_2'}^+$ profile, e) $\overline{u_2'u_2'}^+$ profile, f) $\overline{u_3'u_3'}^+$ profile



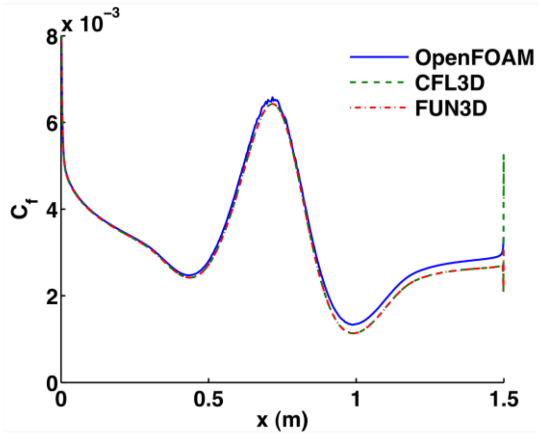
a)



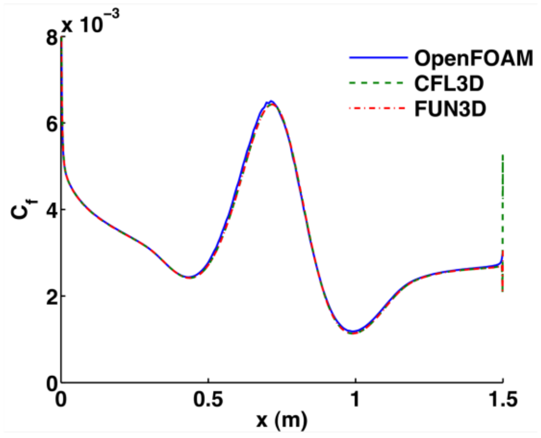
b)



c)

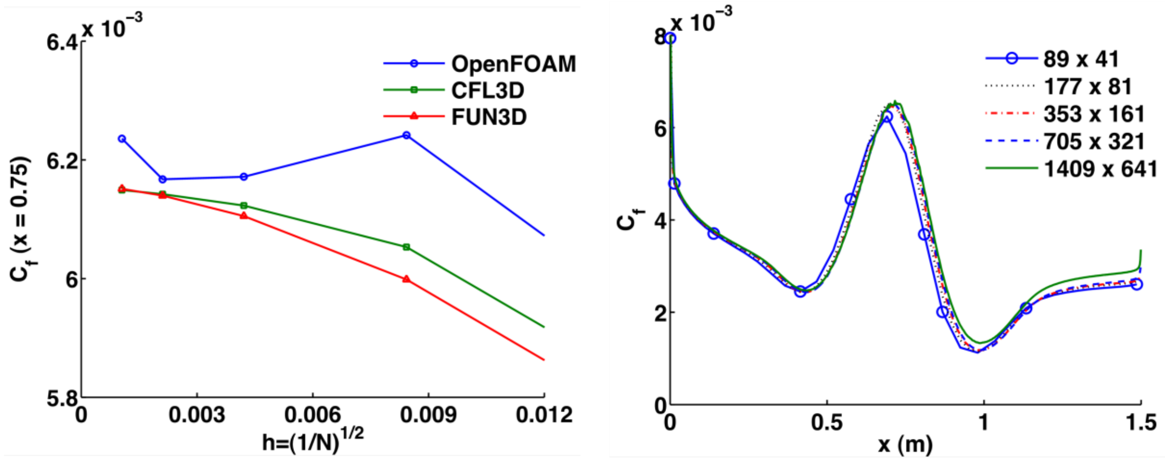


d)



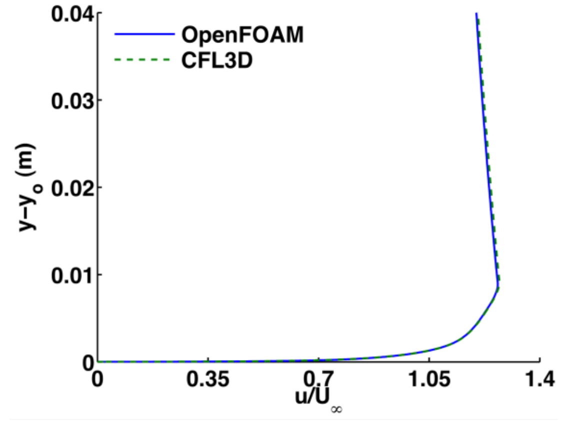
e)

Figure 14: Results of the 2D bump-in-channel simulation with the SA turbulence model for OpenFOAM (blue), CFL3D (dashed green) and FUN3D (dashed red):
a) mean velocity profile (*x*=0.75), b) mean velocity profile (*x*=1.20148), c) dimensionless eddy viscosity profile, d) skin friction coefficient profile for 1409x641 mesh, e) skin friction coefficient profile for 705x321 mesh

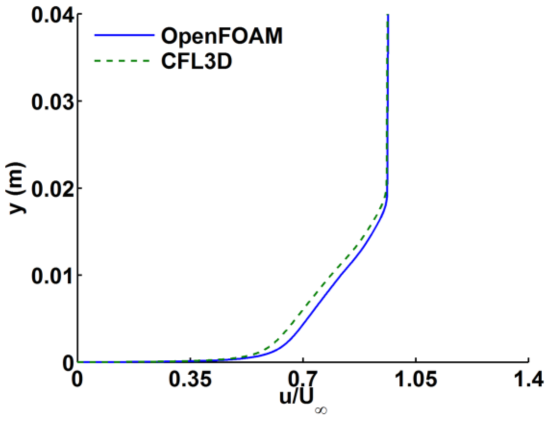


a) b)

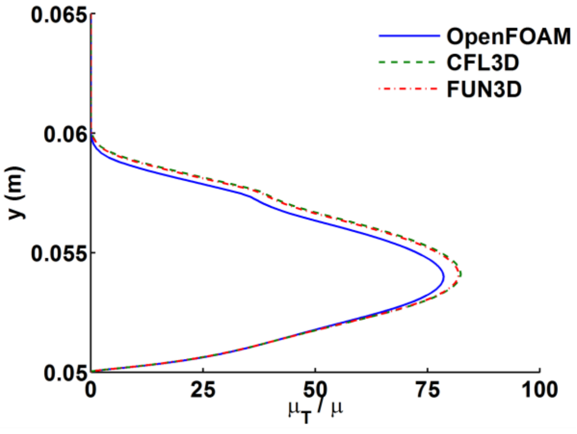
Figure 15: Grid convergence results of the 2D bump-in-channel simulation with the SA model:
a) skin friction coefficient comparison between OpenFOAM (blue) and NASA codes (red, green) at *x*=0.75,
b) skin friction coefficient profiles obtained with OpenFOAM for all meshes



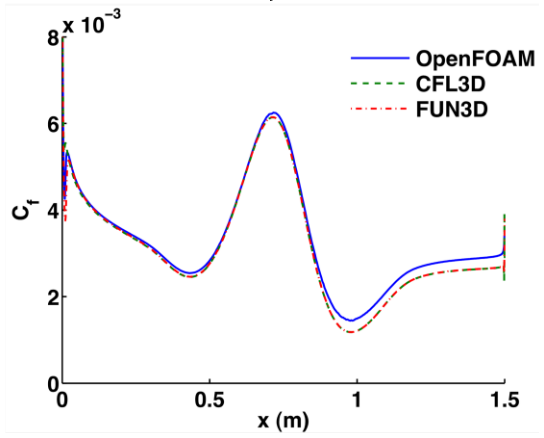
a)



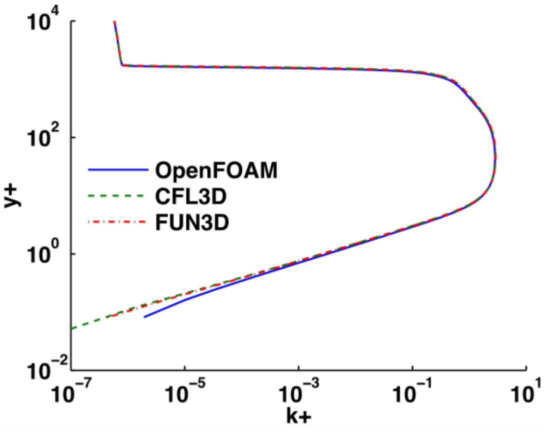
b)



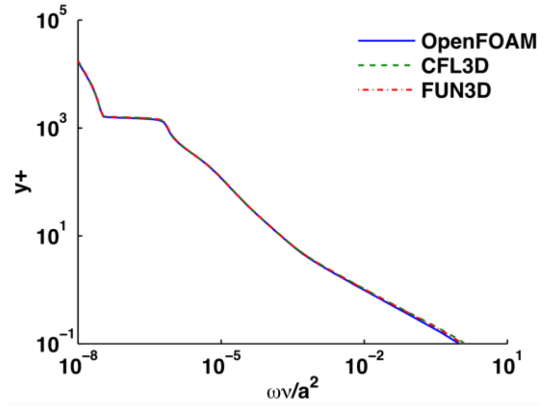
c)



d)

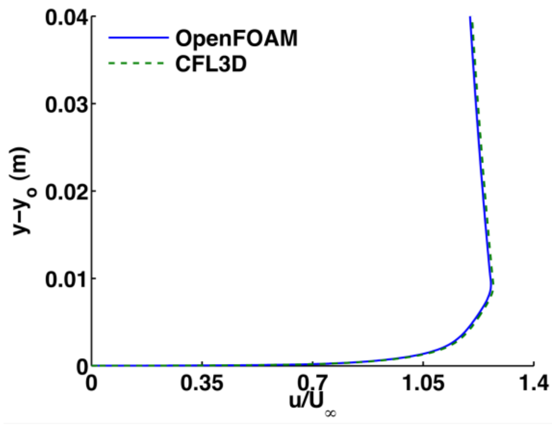


e)

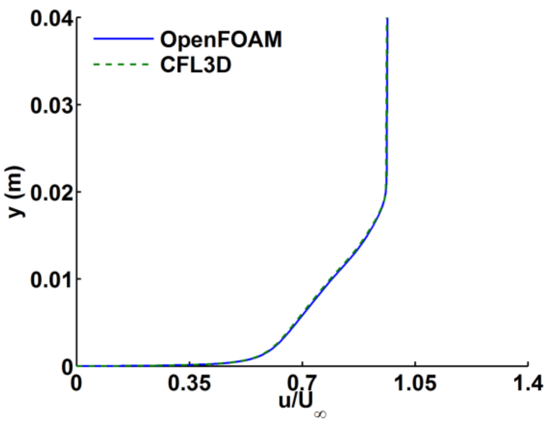


f)

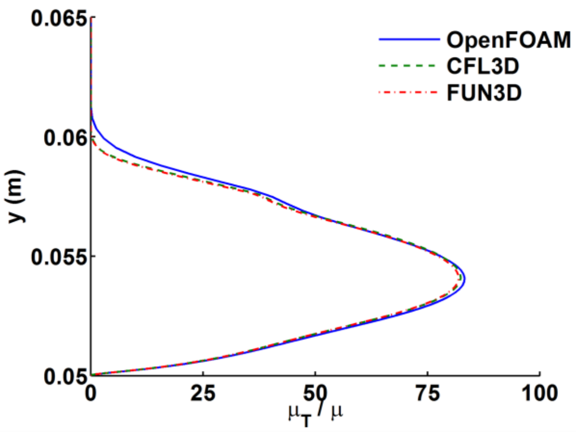
Figure 16: Results of the 2D bump-in-channel with the SST turbulence model on the 1409x641 mesh for OpenFOAM (blue), CFL3D (dashed green) and FUN3D (dashed red):
a) mean velocity profile (*x*=0.75), b) mean velocity profile (*x*=1.20148), c) dimensionless eddy viscosity profile, d) skin friction coefficient profile, e) dimensionless turbulent kinetic energy profile, f) specific dissipation profile



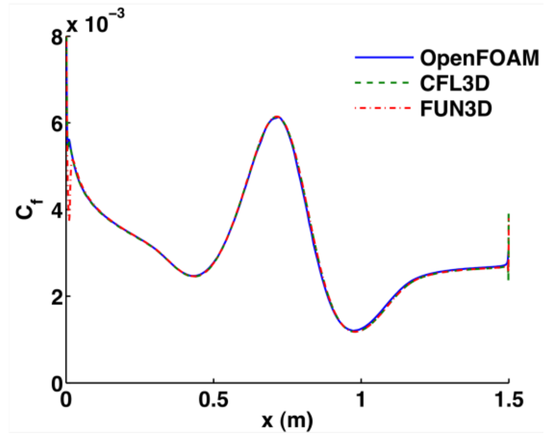
a)



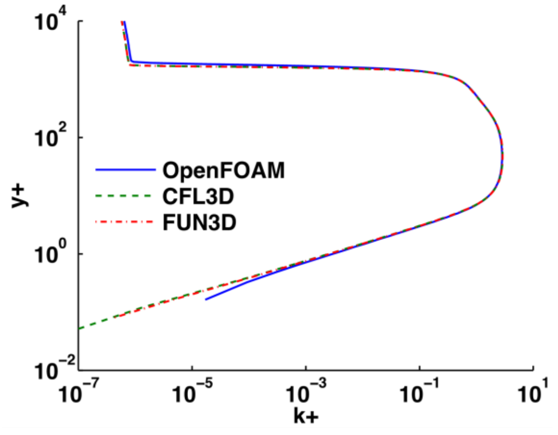
b)



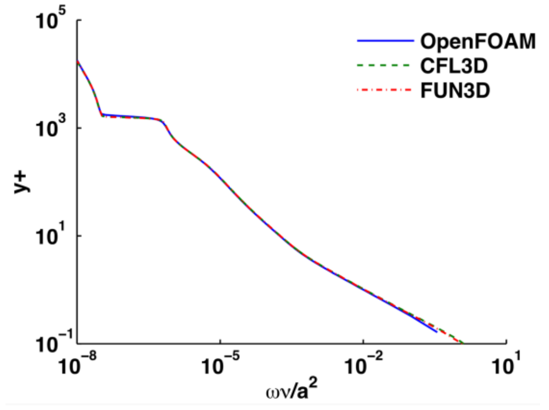
c)



d)

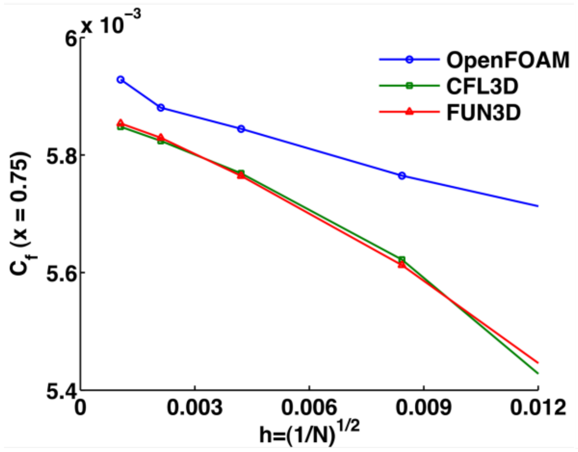


e)

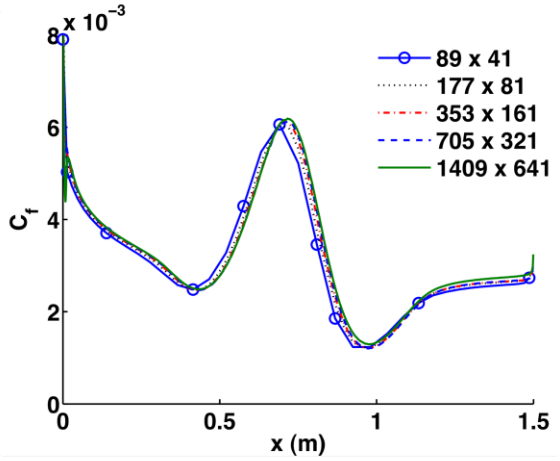


f)

Figure 17: Results of the 2D bump-in-channel with the SST turbulence model on the 705x321 mesh for OpenFOAM (blue), CFL3D (dashed green) and FUN3D (dashed red):
a) mean velocity profile (*x*=0.75), b) mean velocity profile (*x*=1.20148), c) dimensionless eddy viscosity profile, d) skin friction coefficient profile, e) dimensionless turbulent kinetic energy profile, f) specific dissipation profile

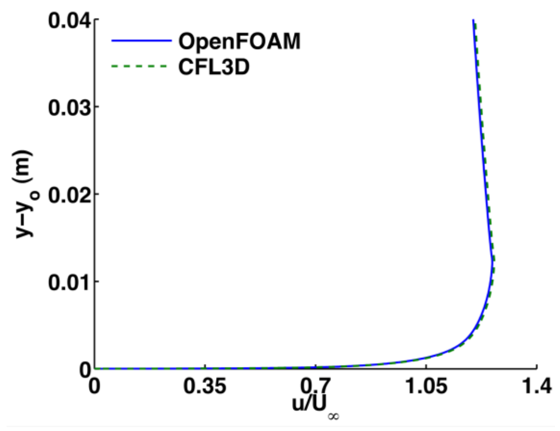


a)

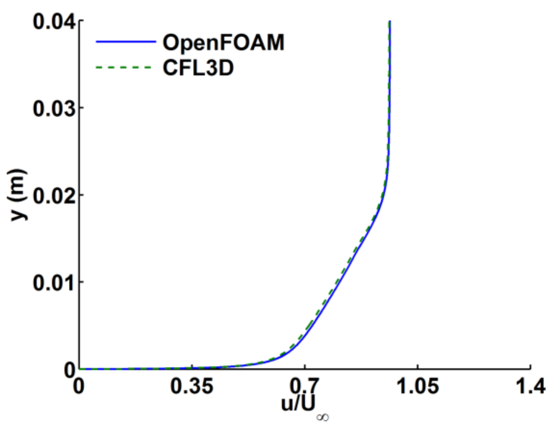


b)

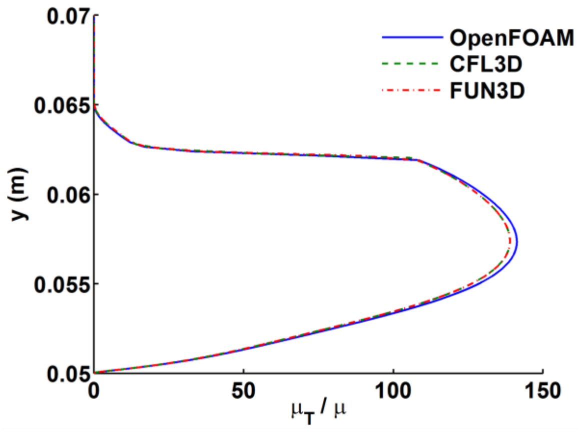
Figure 18: Grid convergence results of the 2D bump-in-channel simulation with the SST model:
a) skin friction coefficient comparison between OpenFOAM (blue) and NASA codes (red, green) at *x*=0.75,
b) skin friction coefficient profiles obtained with OpenFOAM for all meshes



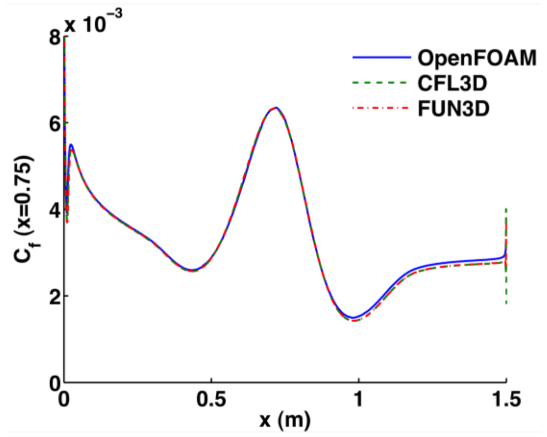
a)



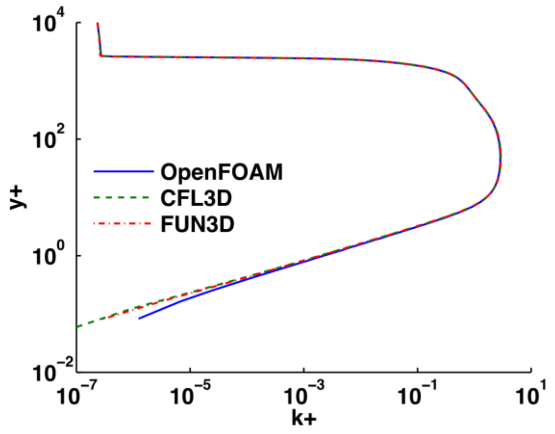
b)



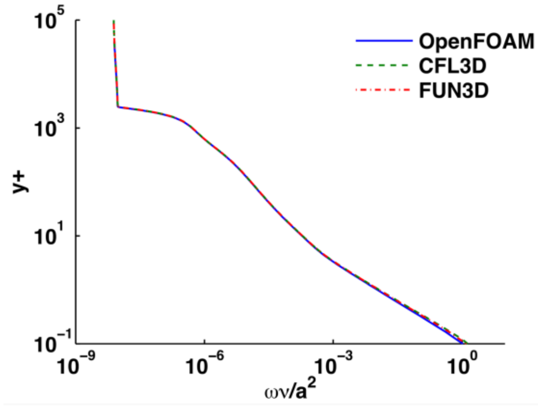
c)



d)

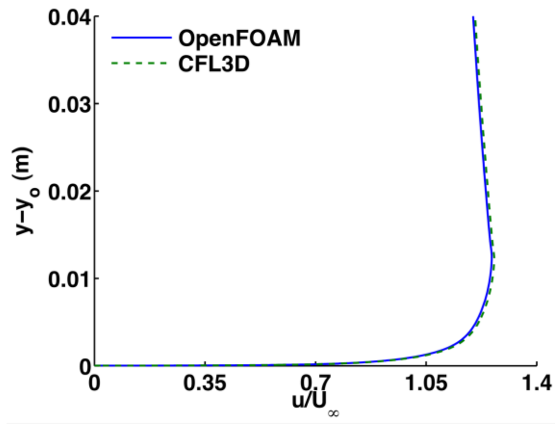


e)

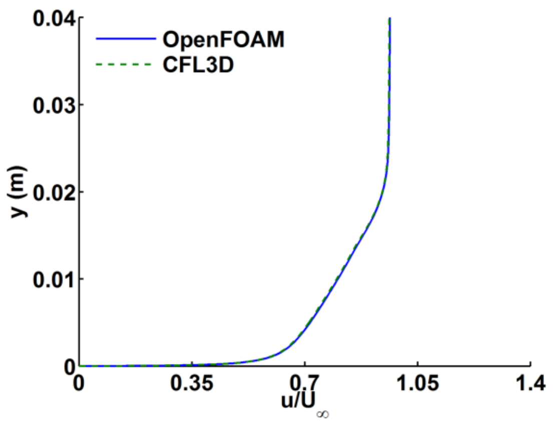


f)

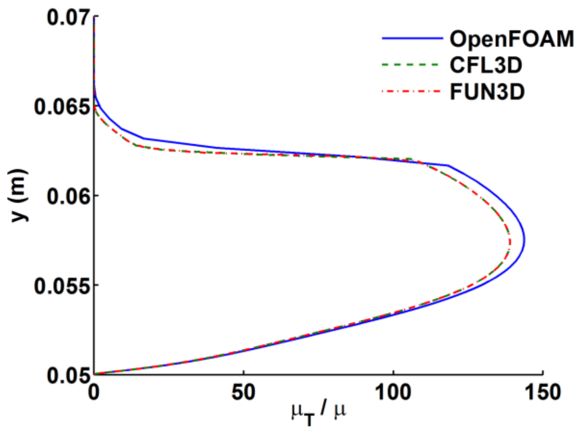
Figure 19: Results of the 2D bump-in-channel with the *k*-*ω* turbulence model on the 1409x641 mesh for OpenFOAM (blue), CFL3D (dashed green) and FUN3D (dashed red):
a) mean velocity profile (*x*=0.75), b) mean velocity profile (*x*=1.20148), c) dimensionless eddy viscosity profile, d) skin friction coefficient profile, e) dimensionless turbulent kinetic energy profile, f) specific dissipation profile



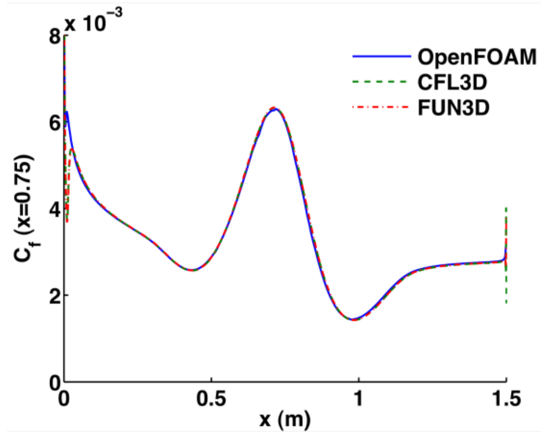
a)



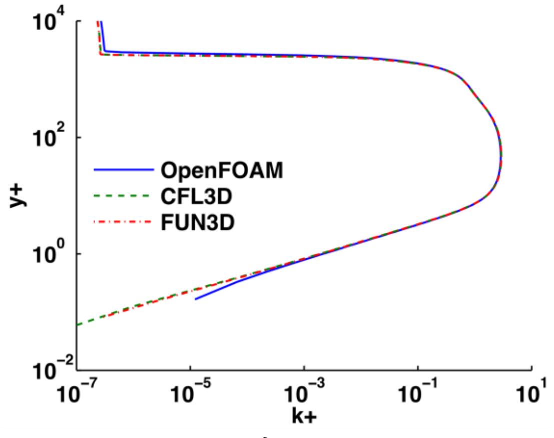
b)



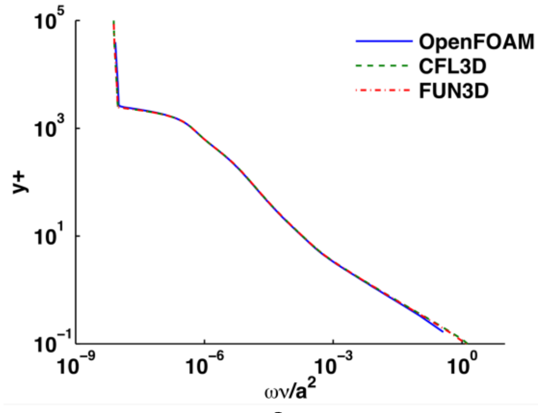
c)



d)

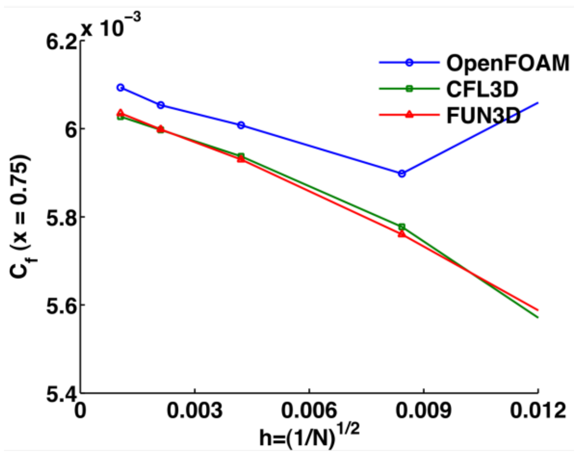


e)

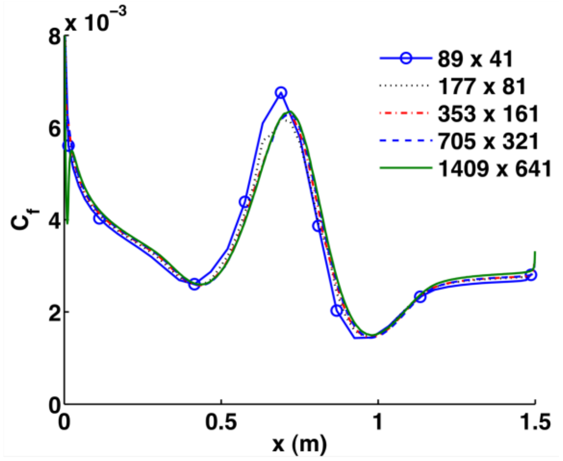


f)

Figure 20: Results of the 2D bump-in-channel with the *k*-*ω* turbulence model on the 705x341 mesh for OpenFOAM (blue), CFL3D (dashed green) and FUN3D (dashed red):
a) mean velocity profile (*x*=0.75), b) mean velocity profile (*x*=1.20148), c) dimensionless eddy viscosity profile, d) skin friction coefficient profile, e) dimensionless turbulent kinetic energy profile, f) specific dissipation profile



a)



b)

Figure 21: Grid convergence results of the 2D bump-in-channel simulation with the *k*-*ω* model:
a) skin friction coefficient comparison between OpenFOAM (blue) and NASA codes (red, green) at *x*=0.75,
b) skin friction coefficient profiles obtained with OpenFOAM for all meshes

V. References

[1] Freitas, C. J. "Perspective: selected benchmarks from commercial CFD codes." *Journal of Fluids Engineering* 117.2 (1995): 208-218.

[2] Iaccarino, G. "Predictions of a turbulent separated flow using commercial CFD codes." *Journal of Fluids Engineering* 123.4 (2001): 819-828.

[3] OpenFOAM, Open Field Operation and Manipulation, Software Package, www.openfoam.com/

[4] Rumsey, C. (2013, March 29). *CFL3D Home Page*. Retrieved from http://cfl3d.larc.nasa.gov/

[5] Carpenter, M. (2014, June 27). *FUN3D Home Page*. Retrieved from http://fun3d.larc.nasa.gov/

[6] Wilcox, D. C. (2006). *Turbulence modeling for CFD* (Vol. 3, pp. 40, 90, 109-112, 125-126, 322). La Canada, CA: DCW industries.

[7] Furbo, E. (2010). "Evaluation of RANS turbulence models for flow problems with significant impact of boundary layers". Master's thesis, Swedish Defense Research Agency, FOI. December 2010.

[8] CFD Online. (2012, January 3). *Boussinesq Eddy Viscosity Assumption*. Retrieved from http://www.cfd-online.com/Wiki/Boussinesq_eddy_viscosity_assumption

[9] Rumsey, C. L., Rubinstein, R., Salas, M. D., Thomas, J. L., "Turbulence Modeling Workshop", NASA/CR-2001-210841, ICASE Interim Report No. 37.

[10] Celik, I. B.. Introductory Turbulence Modeling. 1999 Lecture Notes, West Virginia University, Morgantown, WV.

[11] CFD Online. (2011, June 13). *Specific Turbulence Dissipation Rate*. Retrieved from http://www.cfd-online.com/Wiki/Specific_turbulence_dissipation_rate

[12] Prandtl, L, "Übe rein neues Formelsystem fur die ausgebilldete Turbulenz," Nacr. Akad. Wiss. Göttingen, Math-Phys. Kl., (1945) pp. 6-19.

[13] Hanjalić, K., & Launder, B. (2011). *Modelling turbulence in engineering and the environment: second-moment routes to closure* (Vol. 1, pp. 5, 25, 63, 75, 95, 195). Cambridge University Press. Cambridge, UK.

[14] Daly, B.J. and Harlow, F. H., "Transport equations in turbulence", *Phys. Fluids* 13, (1970) 2634-2649.

[15] Lumley, J. L., “Computational modeling of turbulent flows”, *Adv. Appl. Mech*. 18, (1978) 123-176.

[16] Rotta, J. C., “Statistische Theorie nichthomogener Turbulenz”, *Z. Phys*. 129, (1951) 547-572, 131, 51-77.

[17] Naot, D., Shavit, A. and Wolfshtein, M., “Interactions between components of the turbulent correlation tensor”, *Isr. J. Technol*. 8, (1970) 259-269.

[18] Shir, C. C., “A preliminary numerical study of atmospheric turbulent flows in the idealized turbulent boundary layer”. *J. Atmos. Sci*. 30, (1973) 1327-1339.

[19] Gibson, M. M. and Launder, B. E., “Ground effects on pressure fluctuations in the atmospheric boundary layer”, *J. Fluid Mech. 86*, (1978) 491-511.

[20] Kurbatskii, A. F., & Poroseva, S. V., “Modeling turbulent diffusion in a rotating cylindrical pipe flow”. *International journal of heat and fluid flow*, *20*(3), (1999) 341-348.

[21] Schwarz, W. R., (1992). “Experiment and Modeling of a Three-dimensional turbulent boundary layer in a 30 degree bend”*,* Dissertation, Stanford University.

[22] DeGraaff, D. B., Eaton, J. K., “Reynolds-number scaling of the flat-plate turbulent boundary layer,” *J. Fluid Mech*., Vol. 422, 2000, pp. 319-346.

[23] Longo, J., Huang, H. P., & Stern, F.,“Solid/free-surface juncture boundary layer and wake”. *Experiments in fluids*, *25*(4), (1998) 283-297.

[24] Castillo, L., & Johansson, T. G., “The effects of the upstream conditions on a low Reynolds number turbulent boundary layer with zero pressure gradient”. *Journal of Turbulence*, *3*(31), (2002) 1-19.

[25] Bardina, J. E., Huang, P. G., Coakley, T. J., “Turbulence Modeling Validation, Testing, and Development”, NASA Technical Memorandum 110446, April 1997.

[26] Chan, W. Y. K., Jacobs, P. A., & Mee, D. J., “Suitability of the k–ω turbulence model for scramjet flowfield simulations”. *International Journal for Numerical Methods in Fluids*, *70*(4), (2012) 493-514.

[27] Collie, S., Gerritsen, M., & Jackson, P. “Performance of two-equation turbulence models for flat plate flows with leading edge bubbles”. *Journal of Fluids Engineering*, *130*(2),(2008) 021201.

[28] Xu, L., Rusak, Z., & Castillo, L., “A Reduced-Order Model of the Mean Properties of a Turbulent Wall Boundary Layer at a Zero Pressure Gradient”. *Journal of Fluids Engineering*, *136*(3), (2014) 031103.

[29] Spalart, P. R., "Direct simulation of a turbulent boundary layer up to Rθ= 1410". *Journal of Fluid Mechanics*, *187*, (1988) 61-98.

[30] Wu, X., & Moin, P., "Direct numerical simulation of turbulence in a nominally zero-pressure-gradient flat-plate boundary layer". *Journal of Fluid Mechanics*, *630*, (2009) 5-41.

[31] Sillero, J. A., Jimenez, J. Moser, R. D., "One-point statistics for turbulent wall-bounded flows at Reynolds numbers of up to δ^+=2000", *Phys. Fluids*, 25, (2013) 105102.

[32] Osusky, M., Boom, P. D., & Zingg, D. W., "Results from the Fifth AIAA Drag Prediction Workshop obtained with a parallel Newton-Krylov-Schur flow solver discretized using summation-by-parts operators". 31st AIAA Applied Aerodynamics Conference, June 24-27, 2013.

[33] Persson, T., Liefvendahl, M., Bensow, R. E., & Fureby, C., "Numerical investigation of the flow over an axisymmetric hill using LES, DES, and RANS". *Journal of Turbulence*, 7, (2006).

[34] Rumsey, C. L.,"Summary of the 2004 CFD validation workshop on synthetic jets and turbulent separation control" (2004).

[35] Morgan, P. E., Rizzetta, D. P., Visbal, M. R., "High-order numerical simulation of turbulent flow over a wall-mounted hump", *AIAA journal*, 44(2), (2006) 239-251.

[36] Postl, D., & Fasel, H. F., "Direct numerical simulation of turbulent flow separation from a wall-mounted hump". *AIAA journal*, *44*(2), (2006) 263-272.

[37] Seifert, A., & Pack, L. G., "Active flow separation control on wall-mounted hump at high Reynolds numbers". *AIAA journal*, *40*(7), (2002) 1363-1372.

[38] Greenblatt, D., Paschal, K. B., Yao, C. S., Harris, J., Schaeffler, N. W., & Washburn, A. E., "A separation control CFD validation test case, Part 1: baseline and steady suction". *AIAA paper*, *2220*, (2004).

[39] Rumsey, C. (2004, March 29). *CFD Validation of Synthetic Jets and Turbulence Separation Control*. Retrieved from http://cfdval2004.larc.nasa.gov/case3.html

[40] Kim, H. G., Lee, C. M., Lim, H. C., & Kyong, N. H., "An experimental and numerical study on the flow over two-dimensional hills". *Journal of Wind Engineering and Industrial Aerodynamics*, *66*(1), (1997) 17-33.

[41] Cao, S., & Tamura, T., "Experimental study on roughness effects on turbulent boundary layer flow over a two-dimensional steep hill". *Journal of wind engineering and industrial aerodynamics*, *94*(1), (2006) 1-19.

[42] European Research Community on Flow, Turbulence and Combustion Database. (No publication date available). *ERCOFTAC Classic Database*. Retrieved from http://cfd.mace.manchester.ac.uk/ercoftac/index.html

[43] Gatski, T., Rumsey, C. (2004, March 29). *Summary of Results from the Workshop and List of Publications*. Retrieved from http://cfdval2004.larc.nasa.gov/results.html

[44] Zikanov, O. (2010). *Essential computational fluid dynamics*. John Wiley & Sons. Pg.290.

[45] Rizzi, A., & Vos, J., "Toward establishing credibility in computational fluid dynamics simulations". *AIAA journal*, *36*(5), (1998) 668-675.

[46] Roache, P. J., "Verification of codes and calculations". *AIAA journal*, *36*(5), (1998) 696-702.

[47] Vassberg, J. C., Tinoco, E. N., Mani, M., Levy, D., Zickuhr, T., Mavriplis, D. J., Murayama, M., "Comparison of NTF experimental data with CFD predictions from the third AIAA CFD drag prediction workshop". *AIAA paper*, *6918*, (2008).

[48] Wilcox, D. C., "Comparison of two-equation turbulence models for boundary layers with pressure gradient". *AIAA journal*, *31*(8), (1993) 1414-1421.

[49] Rumsey, C. L., "Consistency, Verification, and Validation of Turbulence Models for Reynolds-Averaged Navier-Stokes Applications", Paper EUCASS2009-7, 3rd European Conference for Aerospace Sciences, 2009.

[50] Rumsey, C. (2014, April 7). *Turbulence Modeling Resource: Purpose*. Retrieved from http://turbmodels.larc.nasa.gov/

[51] Smith, B., Rumsey, C., Huang, G., (2014, June 27). *Turbulence Model Benchmarking Working Group*. Retrieved from http://turbmodels.larc.nasa.gov/tmbwg.html

[52] Williams, D. (No publication date available) *AIAA Fluid Dynamics Technical Committee.* Retrieved from https://info.aiaa.org/tac/ASG/FDTC/default.aspx

[53] American Institute of Aeronautics and Astronautics. *Home Page*. Retrieved from http://www.aiaa.org/

[54] Thigpen, W. (2014, May 16). *Pleiades Supercomputer Homepage*. Retrieved from http://www.nas.nasa.gov/hecc/resources/pleiades.html

[55] OpenFOAM Foundation. (No publication date available). *The OpenFOAM Foundation*. Retrieved from http://www.openfoam.org/

[56] OpenCFD Ltd. (No publication date available). *The open source CFD toolbox.* Retrieved from http://www.openfoam.com/about/

[57] OpenFOAM Foundation. (No publication date available). *Features of OpenFOAM*. Retrieved from http://www.openfoam.org/features/

[58] GitHub. (No publication date available). *OpenFOAM-2.2.x*. Retrieved from https://github.com/OpenFOAM/OpenFOAM-2.2.x

[59] Spalart, P. R. and Allmaras, S. R., "A One-Equation Turbulence Model for Aerodynamic Flows," *Recherche Aerospatiale*, No. 1, (1994) pp. 5-21.

[60] Menter, F. R., "Two-Equation Eddy-Viscosity Turbulence Models for Engineering Applications," *AIAA Journal*, Vol. 32, No. 8, (1994) pp. 1598-1605.

[61] Launder, B. E., Reece, G. J., Rodi, W., "Progress in the development of a Reynolds Stress turbulence closure," *J. Fluid Mech*., Vol. 68, (1975) pp.537-566.

[62] Rumsey, C. (2014, June 16). *The Spalart-Allmaras Turbulence Model*. Retrieved from http://turbmodels.larc.nasa.gov/spalart.html

[63] Rumsey, C. (2013, August 29). *The Menter Shear Stress Transport Turbulence Model*. Retrieved from http://turbmodels.larc.nasa.gov/sst.html

[64] Rumsey, C. (2014, April 2). *The Wilcox k-omega Turbulence Model*. Retrieved from http://turbmodels.larc.nasa.gov/wilcox.html

[65] OpenFOAM Programmer's Guide, Chapter 2.

[66] OpenFOAM User's Guide, Figure 4.1: Case directory structure, pg.U103, Table 4.5, Chapter 4.

[67] Patankar, S. V. and Spalding, D.B.,"A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows", *Int. J. of Heat and Mass Transfer*, Volume 15, Issue 10, (1972) pp.1787-1806.

[68] Anderson, John D. *Computational fluid dynamics*. Vol. 2. New York: McGraw-Hill, 1995. Chapter 6.

[69] CFD Online. (2011, December 28). *SIMPLE Algorithm*. Retrieved from http://www.cfd-online.com/Wiki/SIMPLE_algorithm

[70] Biegel, B. (2012, September 14). *NASA Advanced Supercomputing Division*. Retrieved from https://www.nas.nasa.gov/cgi-bin/software/start

[71] Rumsey, C. (2013, March 13). *Grids 2D Zero Pressure Gradient Flate Plate Verification Case*. Retrieved from http://turbmodels.larc.nasa.gov/flatplate_grids.html

[72] Rumsey, C. (2014, April 30). *2D Zero Pressure Gradient Flate Plate Verification Case – Intro Page*. Retrieved from http://turbmodels.larc.nasa.gov/flatplate.html

[73] Rumsey, C. (2013, March 13). *Grids 2D Bump-in-channel Verification Case*. Retrieved from http://turbmodels.larc.nasa.gov/bump_grids.html

[74] Rumsey, C. (2013, March 13). *2D Bump-in-channel Verification Case – Intro Page*. Retrieved from http://turbmodels.larc.nasa.gov/bump.html

VI. Appendix

A. Pleiades Information

Pleiades is named after an astronomical open star cluster and it is one of the world's most powerful supercomputers. The system is a distributed-memory SGI ICE cluster connected with InfiniBand® in a dual-plane hypercube technology. The system contains the following types of Intel® Xeon® processors: E5-2680v2 (Ivy Bridge), E5-2670 (Sandy Bridge), and X5670 (Westmere). The cluster's information is as follows:

- System Architecture
    - Manufacturer: SGI
    - 163 racks (11,176 nodes)
    - 3.59 Pflop/s peak cluster
    - 1.54 Pflop/s LINPACK rating (November 2013)
    - 2 racks enhanced with NVIDIA graphics processing unit
    - Total cores: 184,800
    - Total memory: 502 TB
- Interconnects
    - Internode: InfiniBand®, with all nodes connected in partial hypercube topology
    - Two independent InfiniBand® fabrics
    - Infiniband® DDR, QDR and FDR
    - Gigabit Ethernet management network
- Storage
    - SGI® InfiniteStorege NEXIS 9000 home filesystem
    - 15 PB of RAID disk storage configured over several cluster-wide Listre filesystems
- Operating Environment
    - Operating system: SUSE® Linux®
    - Job scheduler: PBS®
    - Compilters: Intel and GNU C, C++ and Fortran
    - MPI SGI MPT, MVAPICH2, Intel MPI

The information presented above was obtained from NASA's Advanced Super Computer Division website [54]. More details on the specifics of each subcomponent are available at the same location.

B. Turbulence Model Source Code

Spalart-Allmaras Model

SA Source file:

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2012 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

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

#include "MySpalartAllmaras.H"
#include "addToRunTimeSelectionTable.H"

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

namespace Foam
{
namespace incompressible
{
namespace RASModels
{

// * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //
defineTypeNameAndDebug(MySpalartAllmaras, 0);
addToRunTimeSelectionTable(RASModel, MySpalartAllmaras, dictionary);

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

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

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

//OpenFOAM definition of fv2 doesn't match NASA's
/*tmp<volScalarField> MySpalartAllmaras::fv2
(
   const volScalarField& chi,
   const volScalarField& fv1
) const
{
   return 1.0/pow3(scalar(1) + chi/Cv2_);
}*/


//NASA's definition:
tmp<volScalarField> MySpalartAllmaras::fv2
(
   const volScalarField& chi,
   const volScalarField& fv1
) const
{
   return 1.0 - chi/(1.0+chi*fv1);
}

//There is no fv3 in NASA's equations (Trip term mentioned?)
/*tmp<volScalarField> MySpalartAllmaras::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> MySpalartAllmaras::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);
}

//****************************START ADDITIONS*******************************
tmp<volScalarField> MySpalartAllmaras::ft2(const volScalarField& chi) const
{
    const volScalarField chi2(pow(chi,2));
    return Ct3_*exp(-1.0*Ct4_*chi2);
}
//****************************END ADDITIONS*********************************


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

MySpalartAllmaras::MySpalartAllmaras
(
    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
        )
    ),
//No Cv2 in NASA's equations
    /*Cv2_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "Cv2",
            coeffDict_,
            5.0
        )
    ),*/
//*************************START ADDITIONS***************************************
    Ct3_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "Ct3",
            coeffDict_,
            1.2
        )
    ),
    Ct4_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "Ct4",
            coeffDict_,
            0.5
        )
    ),
//*************************END ADDITIONS****************************************

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

tmp<volScalarField> MySpalartAllmaras::k() const
{
    WarningIn("tmp<volScalarField> MySpalartAllmaras::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> MySpalartAllmaras::epsilon() const
{
    WarningIn("tmp<volScalarField> MySpalartAllmaras::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> MySpalartAllmaras::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> MySpalartAllmaras::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> MySpalartAllmaras::divDevReff(volVectorField& U) const
{
    const volScalarField nuEff_(nuEff());

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


tmp<fvVectorMatrix> MySpalartAllmaras::divDevRhoReff
(
    const volScalarField& rho,
    volVectorField& U
) const
{
    volScalarField muEff("muEff", rho*nuEff());

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


bool MySpalartAllmaras::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()); Not Used
//****************START ADDITIONS*************************************
         Ct3_.readIfPresent(coeffDict());
         Ct4_.readIfPresent(coeffDict());
//****************END ADDITIONS**************************************
```

```
        return true;
    }
    else
    {
        return false;
    }
}

void MySpalartAllmaras::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));

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

// nuTilda equation had to be modified to include ft2 terms
    tmp<fvScalarMatrix> nuTildaEqn
    (
        fvm::ddt(nuTilda_)
      + fvm::div(phi_, nuTilda_)
      - fvm::laplacian(DnuTildaEff(), nuTilda_)
      - Cb2_/sigmaNut_*magSqr(fvc::grad(nuTilda_))
     ==
        Cb1_*(1.0-ft2(chi))*Stilda*nuTilda_
      - fvm::Sp((Cw1_*fw(Stilda)*nuTilda_ - Cb1_*ft2(chi)*nuTilda_/sqr(kappa_))/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

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

SA Header file:

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2012 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Class
    Foam::incompressible::RASModels::MySpalartAllmaras

Group
    grpIcoRASTurbulence

Description
    Spalart-Allmaras 1-eqn mixing-length model for incompressible external
    flows.

    References:
    \verbatim
        "A One-Equation Turbulence Model for Aerodynamic Flows"
        P.R. Spalart,
        S.R. Allmaras,
        La Recherche Aerospatiale, No. 1, 1994, pp. 5-21.

        Extended according to:

        "An Unstructured Grid Generation and Adaptive Solution Technique
        for High Reynolds Number Compressible Flows"
        G.A. Ashford,
        Ph.D. thesis, University of Michigan, 1996.
    \endverbatim

    The default model coefficients correspond to the following:
    \verbatim
```

```
        MySpalartAllmarasCoeffs
        {
            Cb1         0.1355;
            Cb2         0.622;
            Cw2          0.3;
            Cw3          2.0;
            Cv1         7.1;
            Cv2         5.0;
            sigmaNut    0.66666;
            kappa       0.41;
        }
    \endverbatim

SourceFiles
    MySpalartAllmaras.C

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

#ifndef MySpalartAllmaras_H
#define MySpalartAllmaras_H

#include "RASModel.H"
#include "wallDist.H"

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

namespace Foam
{
namespace incompressible
{
namespace RASModels
{

/*---------------------------------------------------------------------------*\
                           Class MySpalartAllmaras Declaration
\*---------------------------------------------------------------------------*/

class MySpalartAllmaras
:
    public RASModel
{

protected:

    // Protected data

        // Model coefficients
            dimensionedScalar sigmaNut_;
            dimensionedScalar kappa_;
            dimensionedScalar Cb1_;
            dimensionedScalar Cb2_;
```

```
          dimensionedScalar Cw1_;
          dimensionedScalar Cw2_;
          dimensionedScalar Cw3_;
          dimensionedScalar Cv1_;
          //dimensionedScalar Cv2_; NOT IN NASA's model
//******************START ADDITIONS*********************************************
           dimensionedScalar Ct3_;
           dimensionedScalar Ct4_;
//******************END ADDITIONS***********************************************

        // Fields
          volScalarField nuTilda_;
          volScalarField nut_;
          wallDist d_;

    // Protected Member Functions
        tmp<volScalarField> chi() const;

        tmp<volScalarField> fv1(const volScalarField& chi) const;

        tmp<volScalarField> fv2
        (
          const volScalarField& chi,
          const volScalarField& fv1
        ) const;

        /*tmp<volScalarField> fv3
        (
          const volScalarField& chi,
          const volScalarField& fv1
        ) const;
         */
        tmp<volScalarField> fw(const volScalarField& Stilda) const;

//*****************START ADDITIONS*****************************************
tmp<volScalarField> ft2(const volScalarField& chi) const;
//*****************END ADDITIONS*******************************************

public:

    //- Runtime type information
    TypeName("MySpalartAllmaras");

    // Constructors

        //- Construct from components
        MySpalartAllmaras
        (
          const volVectorField& U,
          const surfaceScalarField& phi,
          transportModel& transport,
            const word& turbulenceModelName = turbulenceModel::typeName,
            const word& modelName = typeName
        );


    //- Destructor
    virtual ~MySpalartAllmaras()
    {}

    // Member Functions
        //- Return the turbulence viscosity
        virtual tmp<volScalarField> nut() const
        {
            return nut_;
        }

        //- Return the effective diffusivity for nuTilda
        tmp<volScalarField> DnuTildaEff() const;

        //- Return the turbulence kinetic energy
        virtual tmp<volScalarField> k() const;

        //- Return the turbulence kinetic energy dissipation rate
        virtual tmp<volScalarField> epsilon() const;

        //- Return the Reynolds stress tensor
        virtual tmp<volSymmTensorField> R() const;

        //- Return the effective stress tensor including the laminar stress
        virtual tmp<volSymmTensorField> devReff() const;

        //- Return the source term for the momentum equation
        virtual tmp<fvVectorMatrix> divDevReff(volVectorField& U) const;

        //- Return the source term for the momentum equation
        virtual tmp<fvVectorMatrix> divDevRhoReff
        (
            const volScalarField& rho,
            volVectorField& U
        ) const;

        //- Solve the turbulence equations and correct the turbulence viscosity
        virtual void correct();

        //- Read RASProperties dictionary
        virtual bool read();
};


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

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

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

#endif

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

SST Model

SST Source file:

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2012 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

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

#include "MySSTStd.H"
#include "addToRunTimeSelectionTable.H"

#include "backwardsCompatibilityWallFunctions.H"

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

namespace Foam
{
namespace incompressible
{
namespace RASModels
{

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

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

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

tmp<volScalarField> MySSTStd::F1(const volScalarField& CDkOmega) const
{
    tmp<volScalarField> CDkOmegaPlus = max //limiter, what is defined as CD_kOmega
```

```
(NASA)
    (
        CDkOmega,
        dimensionedScalar("1.0e-21", dimless/sqr(dimTime), 1.0e-21)
    );

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

    return tanh(pow4(arg1));
}


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

    );

    return tanh(sqr(arg2));
}


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

MySSTStd::MySSTStd
(
    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_,
            0.85
        )
    ),
    alphaK2_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "alphaK2",
            coeffDict_,
            1.0
        )
    ),
    alphaOmega1_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "alphaOmega1",
            coeffDict_,
            0.5
        )
    ),
    alphaOmega2_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "alphaOmega2",
            coeffDict_,
            0.856
        )
    ),
    gamma1_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "gamma1",
            coeffDict_,
            0.55316666
        )
    ),
    gamma2_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "gamma2",
            coeffDict_,
            0.44035466
        )
    ),
    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
    )
),
b1_
(
    dimensioned<scalar>::lookupOrAddToDict
    (
        "b1",
        coeffDict_,
        1.0
    )
),
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_,
            b1_*F2()*sqrt(2.0)*mag(skew(fvc::grad(U_)))
        )
    );
    nut_.correctBoundaryConditions();

    printCoeffs();
}


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

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


tmp<fvVectorMatrix> MySSTStd::divDevRhoReff
(
    const volScalarField& rho,
    volVectorField& U
) const
{
    volScalarField muEff("muEff", rho*nuEff());

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

bool MySSTStd::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());
        b1_.readIfPresent(coeffDict());
        c1_.readIfPresent(coeffDict());


        return true;
    }
    else
    {
        return false;
    }
}
```

```
void MySSTStd::correct()
{
   RASModel::correct();

   if (!turbulence_)
   {
      return;
   }

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

   const volScalarField S2(2*magSqr(symm(fvc::grad(U_))));
   volScalarField G(type() + ".G", nut_*S2);
   //volScalarField G(GName(), nut_*2*magSqr(symm(fvc::grad(U_)))); for newer OF
versions
   volScalarField G2(type() + ".G", min(G,scalar(20.0)*betaStar_*omega_*k_));
//****CHANGED

   // 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::laplacian(DomegaEff(F1), omega_)
   ==
      gamma(F1)*G/nut_  //*************DIFFERENT FROM STANDARD MODEL
    - fvm::Sp(beta(F1)*omega_, omega_)
    + fvm::Sp //changed this (see above)
      (
         (scalar(1)-F1)*CDkOmega/omega_,
         omega_
      )
   );

   omegaEqn().relax();

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

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


    // Turbulent kinetic energy equation ADDED LIMITER G2 (NASA)
    tmp<fvScalarMatrix> kEqn
    (
        fvm::ddt(k_)
      + fvm::div(phi_, k_)
      - fvm::laplacian(DkEff(F1), k_)
     ==
        G2   //************************DIFFERENT FROM STANDARD MODEL
      - fvm::Sp(betaStar_*omega_, k_)
    );

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

    // Re-calculate viscosity
    nut_ = a1_*k_/max(a1_*omega_, b1_*F2()*sqrt(2.0)*mag(skew(fvc::grad(U_))));
    nut_.correctBoundaryConditions();
}


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

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

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

SST Header file:

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2012 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Class
    Foam::incompressible::RASModels::MySSTStd

Description
    Implementation of the k-omega-SST turbulence model for incompressible
    flows.

    Turbulence model described in:
    \verbatim
        Menter, F., Esch, T.,
        "Elements of Industrial Heat Transfer Prediction",
        16th Brazilian Congress of Mechanical Engineering (COBEM),
        Nov. 2001.
    \endverbatim

    with the addition of the optional F3 term for rough walls from
    \verbatim
        Hellsten, A.
        "Some Improvements in Menter's k-omega-SST turbulence model"
        29th AIAA Fluid Dynamics Conference,
        AIAA-98-2554,
        June 1998.
    \endverbatim

    Note that this implementation is written in terms of alpha diffusion
    coefficients rather than the more traditional sigma (alpha = 1/sigma) so
    that the blending can be applied to all coefficuients in a consistent
    manner.  The paper suggests that sigma is blended but this would not be
    consistent with the blending of the k-epsilon and k-omega models.

    Also note that the error in the last term of equation (2) relating to
    sigma has been corrected.

    Wall-functions are applied in this implementation by using equations (14)
    to specify the near-wall omega as appropriate.

    The blending functions (15) and (16) are not currently used because of the
    uncertainty in their origin, range of applicability and that is y+ becomes
    sufficiently small blending u_tau in this manner clearly becomes nonsense.

    The default model coefficients correspond to the following:
    \verbatim
        MySSTStdCoeffs
        {
            alphaK1     0.85034;
            alphaK2     1.0;
            alphaOmega1 0.5;
            alphaOmega2 0.85616;
            beta1       0.075;
            beta2       0.0828;
            betaStar    0.09;
            gamma1      0.5532;
            gamma2      0.4403;
            a1          0.31;
            b1          1.0;
            c1          10.0;
            F3          no;
        }
    \endverbatim

SourceFiles
    MySSTStd.C

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

#ifndef MySSTStd_H
#define MySSTStd_H

#include "RASModel.H"
#include "wallDist.H"

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

namespace Foam
{
namespace incompressible
{
namespace RASModels
```

```
{

/*---------------------------------------------------------------------------*\
                         Class MySSTStd Declaration
\*---------------------------------------------------------------------------*/

class MySSTStd
:
    public RASModel
{

protected:

    // Protected data:
        // Model coefficients
            dimensionedScalar alphaK1_;
            dimensionedScalar alphaK2_;
            dimensionedScalar alphaOmega1_;
            dimensionedScalar alphaOmega2_;
            dimensionedScalar gamma1_;
            dimensionedScalar gamma2_;
            dimensionedScalar beta1_;
            dimensionedScalar beta2_;
            dimensionedScalar betaStar_;
            dimensionedScalar a1_;
            dimensionedScalar b1_;
            dimensionedScalar c1_;
            Switch F3_;

        //- Wall distance field
        //  Note: different to wall distance in parent RASModel
        wallDist y_;

        // Fields
            volScalarField k_;
            volScalarField omega_;
            volScalarField nut_;

    // Protected Member Functions
        tmp<volScalarField> F1(const volScalarField& CDkOmega) const;
        tmp<volScalarField> F2() const;
        tmp<volScalarField> F3() const;
        tmp<volScalarField> F23() const;

        tmp<volScalarField> blend
        (
            const volScalarField& F1,
            const dimensionedScalar& psi1,
            const dimensionedScalar& psi2
        ) const
        {
            return F1*(psi1 - psi2) + psi2;
        }

        tmp<volScalarField> alphaK(const volScalarField& F1) const
        {
            return blend(F1, alphaK1_, alphaK2_);
        }

        tmp<volScalarField> alphaOmega(const volScalarField& F1) const
        {
            return blend(F1, alphaOmega1_, alphaOmega2_);
        }

        tmp<volScalarField> beta(const volScalarField& F1) const
        {
            return blend(F1, beta1_, beta2_);
        }

        tmp<volScalarField> gamma(const volScalarField& F1) const
        {
            return blend(F1, gamma1_, gamma2_);
        }

public:
    //- Runtime type information
    TypeName("MySSTStd");

    // Constructors
        //- Construct from components
        MySSTStd
        (
            const volVectorField& U,
            const surfaceScalarField& phi,
            transportModel& transport,
            const word& turbulenceModelName = turbulenceModel::typeName,
            const word& modelName = typeName
        );

    //- Destructor
    virtual ~MySSTStd()
    {}

    // Member Functions

        //- Return the turbulence viscosity
        virtual tmp<volScalarField> nut() const
        {
            return nut_;
        }

        //- Return the effective diffusivity for k
```

```
tmp<volScalarField> DkEff(const volScalarField& F1) const
{
   return tmp<volScalarField>
   (
      new volScalarField("DkEff", alphaK(F1)*nut_ + nu())
   );
}

//- Return the effective diffusivity for omega
tmp<volScalarField> DomegaEff(const volScalarField& F1) const
{
   return tmp<volScalarField>
   (
      new volScalarField("DomegaEff", alphaOmega(F1)*nut_ + nu())
   );
}

//- Return the turbulence kinetic energy
virtual tmp<volScalarField> k() const
{
   return k_;
}

//- Return the turbulence specific dissipation rate
virtual tmp<volScalarField> omega() const
{
   return omega_;
}

//- Return the turbulence kinetic energy dissipation rate
virtual tmp<volScalarField> epsilon() const
{
   return tmp<volScalarField>
   (
      new volScalarField
      (
         IOobject
         (
            "epsilon",
            mesh_.time().timeName(),
            mesh_
         ),
         betaStar_*k_*omega_,
         omega_.boundaryField().types()
      )
   );
}

//- Return the Reynolds stress tensor
virtual tmp<volSymmTensorField> R() const;
```

```
        //- Return the effective stress tensor including the laminar stress
        virtual tmp<volSymmTensorField> devReff() const;

        //- Return the source term for the momentum equation
        virtual tmp<fvVectorMatrix> divDevReff(volVectorField& U) const;

        //- Return the source term for the momentum equation
        virtual tmp<fvVectorMatrix> divDevRhoReff
        (
            const volScalarField& rho,
            volVectorField& U
        ) const;

        //- Solve the turbulence equations and correct the turbulence viscosity
        virtual void correct();

        //- Read RASProperties dictionary
        virtual bool read();
};

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

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

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

#endif

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

k-ω Model

k-ω Source file:

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2012 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.
\*---------------------------------------------------------------------------*/

#include "kOmega20062D.H"
#include "addToRunTimeSelectionTable.H"

#include "backwardsCompatibilityWallFunctions.H"

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

namespace Foam
{
namespace incompressible
{
namespace RASModels
{

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

defineTypeNameAndDebug(kOmega20062D, 0);
addToRunTimeSelectionTable(RASModel, kOmega20062D, dictionary);

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

kOmega20062D::kOmega20062D
(
    const volVectorField& U,
    const surfaceScalarField& phi,
    transportModel& transport,
    const word& turbulenceModelName,
    const word& modelName
)
:
    RASModel(modelName, U, phi, transport, turbulenceModelName),

    Cmu_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "betaStar",
            coeffDict_,
            0.09 //Beta=9/100 in 2006
        )
    ),
    /*beta_                              ORIGINAL beta DEFINITION
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "beta",
            coeffDict_,
            0.0708 //(changed from 0.072)
        )
    ),*/
    alpha_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "alpha",
            coeffDict_,
            0.52 //alpha=13/25 in 2006
        )
    ),
    alphaK_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "alphaK",
            coeffDict_,
            0.6 //sigma*=3/5 in 2006
        )
    ),
    alphaOmega_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "alphaOmega",
            coeffDict_,
            0.5 //sigma=1/2 in 2006
        )
```

```
),
Clim_
(
    dimensioned<scalar>::lookupOrAddToDict
    (
        "Clim",
        coeffDict_,
        0.875 //Clim=7/8
    )
),
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_)
),
fBeta_
(
    IOobject
    (
        "fBeta",
            runTime_.timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        mesh_,
        dimless
    ),
    Chi_
    (
        IOobject
        (
            "Chi",
            runTime_.timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        mesh_, dimless
    ),
    absChi_
    (
        IOobject
        (
            "absChi",
            runTime_.timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        mesh_, dimless
    ),
    beta_
    (
        IOobject
        (
            "beta",
            runTime_.timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        mesh_,
        dimless
    ),
    alphad_
    (
        IOobject
        (
            "alphad",
            runTime_.timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        mesh_, dimensionedScalar("zero", dimless, 0.125)
    )
{
    bound(k_, kMin_);
    bound(omega_, omegaMin_);

   //nut_ = k_/omega_; //Standard OpenFOAM definition
    nut_ = k_/ max(omega_, Clim_*sqrt(2.0/0.09*magSqr(symm(fvc::grad(U_)))));;
    nut_.correctBoundaryConditions();


    printCoeffs();
}

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

tmp<volSymmTensorField> kOmega20062D::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> kOmega20062D::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> kOmega20062D::divDevReff(volVectorField& U) const
{
    return
    (
      - fvm::laplacian(nuEff(), U)
      - fvc::div(nuEff()*dev(T(fvc::grad(U))))
    );
}


tmp<fvVectorMatrix> kOmega20062D::divDevRhoReff
(
    const volScalarField& rho,
    volVectorField& U
) const
{
    volScalarField muEff("muEff", rho*nuEff());

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

bool kOmega20062D::read()
{
    if (RASModel::read())
    {
        Cmu_.readIfPresent(coeffDict());
        //beta_.readIfPresent(coeffDict());  Must be commented for blending function
        alphaK_.readIfPresent(coeffDict());
        alphaOmega_.readIfPresent(coeffDict());

        return true;
    }
    else
    {
        return false;
    }
}

void kOmega20062D::correct()
```

```
{
   RASModel::correct();

   if (!turbulence_)
   {
      return;
   }

   volTensorField GradU(fvc::grad(U_));
   volSymmTensorField Sij(symm(GradU));
   volTensorField Omij(-skew(GradU));
   volScalarField StressLim(Clim_*sqrt(2.0/Cmu_)*mag(Sij));
   volSymmTensorField tauij(2.0*nut_*Sij-((2.0/3.0)*I)*k_);
   volVectorField Gradk(fvc::grad(k_));
   volVectorField Gradomega(fvc::grad(omega_));
   volScalarField G(type() + ".G", tauij && GradU);
   //volScalarField G(GName(), tauij && GradU); //for newer OF versions

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

//START ADDITIONS FOR 2006 VERSION...................................

volScalarField alphadCheck_(Gradk & Gradomega);  //condition to change alphad_

forAll(alphad_,celli)
{
        if (alphadCheck_[celli] <= 0.0001)
        {
        alphad_[celli]=scalar(0);
        }else
        {
        alphad_[celli]=scalar(0.125);
        }
}

volScalarField CDkOmega(alphad_/omega_*(Gradk & Gradomega)); //last term in NASA
equations

Chi_ = (Omij & Omij) && Sij /pow((Cmu_*omega_),3);
absChi_ = mag(Chi_);
fBeta_ = 1.0; //This term should be (1.0+85.0*absChi_)/(1.0+100.0*absChi_); for 3D
beta_ = 0.0708*fBeta_;

   // Turbulence specific dissipation rate equation
   tmp<fvScalarMatrix> omegaEqn
   (
      fvm::ddt(omega_)
    + fvm::div(phi_, omega_)
    - fvm::laplacian(DomegaEff(), omega_)
    ==
        alpha_*G*omega_/k_
      - fvm::Sp(beta_*omega_, omega_)
      + CDkOmega //Crossflow diffusion term to match 2006
    );

    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::laplacian(DkEff(), k_)
     ==
        G
      - fvm::Sp(Cmu_*omega_, k_)
    );

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


    // Re-calculate viscosity
    //nut_ = k_/omega_; //Standard OpenFOAM definition
    nut_ = k_/ max(omega_, Clim_*sqrt(2.0/0.09*magSqr(symm(fvc::grad(U_)))));;
    nut_.correctBoundaryConditions();
}

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

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

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

k-ω Header file:

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2012 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Class
    Foam::incompressible::RASModels::kOmega20062DC2

Group
    grpIcoRASTurbulence

Description
    Standard high Reynolds-number k-omega turbulence model for
    incompressible flows.

    References:
        http://turbmodels.larc.nasa.gov/wilcox.html

        Turbulence Modeling for CFD (3rd Edition), David C. Wilcox, 2006

SourceFiles
    kOmega20062D.C

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

#ifndef kOmega20062D_H
#define kOmega20062D_H

#include "RASModel.H"

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

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

/*---------------------------------------------------------------------------*\
                         Class kOmega20062DC2 Declaration
\*---------------------------------------------------------------------------*/

class kOmega20062D
:
    public RASModel
{

protected:

    // Protected data

        // Model coefficients
            dimensionedScalar Cmu_;
            //dimensionedScalar beta_;    commented for blending function
            dimensionedScalar alpha_;
            dimensionedScalar alphaK_;
            dimensionedScalar alphaOmega_;
             dimensionedScalar Clim_;

        // Fields
            volScalarField k_;
            volScalarField omega_;
            volScalarField nut_;
             volScalarField fBeta_;
             volScalarField Chi_;
             volScalarField absChi_;
             volScalarField beta_;
            volScalarField alphad_;

public:

    //- Runtime type information
    TypeName("kOmega20062D");

    // Constructors
        //- Construct from components
        kOmega20062D
        (
            const volVectorField& U,
            const surfaceScalarField& phi,
            transportModel& transport,
            const word& turbulenceModelName = turbulenceModel::typeName,
            const word& modelName = typeName
        );

    //- Destructor
    virtual ~kOmega20062D()
    {}

    // Member Functions

        //- Return the turbulence viscosity
        virtual tmp<volScalarField> nut() const
        {
            return nut_;
        }

        //- Return the effective diffusivity for k
        tmp<volScalarField> DkEff() const
        {
            return tmp<volScalarField>
            (
                //new volScalarField("DkEff", alphaK_*nut_ + nu())
            new volScalarField("DkEff", alphaK_*k_/omega_ + nu())
            );
        }

        //- Return the effective diffusivity for omega
        tmp<volScalarField> DomegaEff() const
        {
            return tmp<volScalarField>
            (
                //new volScalarField("DomegaEff", alphaOmega_*nut_ + nu())
            new volScalarField("DomegaEff", alphaOmega_*k_/omega_ + nu())
            );
        }

        //- Return the turbulence kinetic energy
        virtual tmp<volScalarField> k() const
        {
            return k_;
        }

        //- Return the turbulence specific dissipation rate
        virtual tmp<volScalarField> omega() const
        {
            return omega_;
        }

        //- Return the turbulence kinetic energy dissipation rate
        virtual tmp<volScalarField> epsilon() const
        {
            return tmp<volScalarField>
            (
                new volScalarField
                (
                    IOobject
                    (
                        "epsilon",
                        mesh_.time().timeName(),
                        mesh_
                    ),
                    Cmu_*k_*omega_,
                    omega_.boundaryField().types()
                )
            );
        }

        //- Return the Reynolds stress tensor
        virtual tmp<volSymmTensorField> R() const;


        //- Return the effective stress tensor including the laminar stress
        virtual tmp<volSymmTensorField> devReff() const;

        //- Return the source term for the momentum equation
        virtual tmp<fvVectorMatrix> divDevReff(volVectorField& U) const;

        //- Return the source term for the momentum equation
        virtual tmp<fvVectorMatrix> divDevRhoReff
        (
            const volScalarField& rho,
            volVectorField& U
        ) const;

        //- Solve the turbulence equations and correct the turbulence viscosity
        virtual void correct();

        //- Read RASProperties dictionary
        virtual bool read();
};


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

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

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

#endif

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

<u>LRR-IP Model</u>

LRR-IP Source file:

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2012 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

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

#include "SPLRRIP.H"
#include "addToRunTimeSelectionTable.H"
#include "wallFvPatch.H"
#include "backwardsCompatibilityWallFunctions.H"
#include "wallDist.H"
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{
namespace incompressible
{
namespace RASModels
{

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

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

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

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

    Cmu_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "Cmu",
            coeffDict_,
            0.09
        )
    ),
    Clrr1_ //Rotta's constant
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "Clrr1",
            coeffDict_,
            1.8
        )
    ),
    Clrr2_ //Used in rapid term
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "Clrr2",
            coeffDict_,
            0.6
        )
    ),
    C1_ //First epsilon coefficient
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "C1",
            coeffDict_,
            1.35 //1.44
        )
    ),
    C2_ //Second epsilon coefficient
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "C2",
            coeffDict_,
            1.92
)
),
Cs_ //Used in Daly&Harlow GGDH correlation for u_i u_j u_k
(
dimensioned<scalar>::lookupOrAddToDict
(
    "Cs",
    coeffDict_,
    0.22 //used to be 0.25.
)
),
Ceps_ //Third epsilon coefficient
(
dimensioned<scalar>::lookupOrAddToDict
(
    "Ceps",
    coeffDict_,
    0.15
)
),
sigmaEps_ //Used in effective diffusivity of epsilon (See .H file)
(
dimensioned<scalar>::lookupOrAddToDict
(
    "sigmaEps",
    coeffDict_,
    1.3
)
),
couplingFactor_
(
dimensioned<scalar>::lookupOrAddToDict
(
    "couplingFactor",
    coeffDict_,
    0.0
)
),
R_
(
IOobject
(
    "R",
    runTime_.timeName(),
    mesh_,
    IOobject::NO_READ,
    IOobject::AUTO_WRITE
),
autoCreateR("R", mesh_)
),
k_
```

```
(
   IOobject
   (
      "k",
      runTime_.timeName(),
      mesh_,
      IOobject::NO_READ,
      IOobject::AUTO_WRITE
   ),
   autoCreateK("k", mesh_)
),
epsilon_
(
   IOobject
   (
      "epsilon",
      runTime_.timeName(),
      mesh_,
      IOobject::NO_READ,
      IOobject::AUTO_WRITE
   ),
   autoCreateEpsilon("epsilon", mesh_)
),
nut_
(
   IOobject
   (
      "nut",
      runTime_.timeName(),
      mesh_,
      IOobject::NO_READ,
      IOobject::AUTO_WRITE
   ),
   autoCreateNut("nut", mesh_)
 ),
    xn
    (
      IOobject
      (
               "xn",
               runTime_.timeName(),
               mesh_,
               IOobject::NO_READ,
               IOobject::AUTO_WRITE
      ),
      mesh_,
      dimensionedScalar("xn", dimLength, SMALL)
    ),
    utauw
    (
      IOobject
```

```
  (
          "utauw",
          runTime_.timeName(),
          mesh_,
          IOobject::NO_READ,
          IOobject::AUTO_WRITE
  ),
  mesh_,
  dimensionedScalar("utauw", U_.dimensions(), 0.0)
),
utau
(
  IOobject
  (
          "utau",
          runTime_.timeName(),
          mesh_,
          IOobject::NO_READ,
          IOobject::AUTO_WRITE
  ),
  mesh_,
  dimensionedScalar("utau", U_.dimensions(), 0.0)
),
utauFaces
(
  IOobject
  (
          "utauFaces",
          runTime_.timeName(),
          mesh_,
          IOobject::NO_READ,
          IOobject::AUTO_WRITE
  ),
  mesh_,
  dimensionedScalar("utauFaces", U_.dimensions(), 0.0)
),
f1
(
  IOobject
  (
          "f1",
          runTime_.timeName(),
          mesh_,
          IOobject::NO_READ,
          IOobject::AUTO_WRITE
  ),
  mesh_,
  dimensionedScalar("f1", dimless, 0.0)
),
argf2
(
          IOobject
          (
                   "argf2",
                   runTime_.timeName(),
                   mesh_,
                   IOobject::NO_READ,
                   IOobject::AUTO_WRITE
          ),
          mesh_,
          dimensionedScalar("argf2", dimless, 0.0)
        ),
        f2
        (
          IOobject
          (
                   "f2",
                   runTime_.timeName(),
                   mesh_,
                   IOobject::NO_READ,
                   IOobject::AUTO_WRITE
          ),
          mesh_,
          dimensionedScalar("f2", dimless, 1.0)
        ),
        yr_(mesh_),
    C1Ref_
    (
       dimensioned<scalar>::lookupOrAddToDict
       (
          "C1Ref",
          coeffDict_,
          0.3
       )
    ),
    C2Ref_
    (
       dimensioned<scalar>::lookupOrAddToDict
       (
          "C2Ref",
          coeffDict_,
          0.3
       )
     )

{
    if (couplingFactor_.value() < 0.0 || couplingFactor_.value() > 1.0)
    {
       FatalErrorIn
       (
          "MyLRRIP::MyLRRIP"
          "(const volVectorField& U, const surfaceScalarField& phi,"
            "transportModel& transport)"
        )   << "couplingFactor = " << couplingFactor_
            << " is not in range 0 - 1" << nl
            << exit(FatalError);
    }

    bound(k_, kMin_);
    bound(epsilon_, epsilonMin_);

    nut_ = Cmu_*sqr(k_)/epsilon_;
    nut_.correctBoundaryConditions();

    printCoeffs();
}


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

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


tmp<fvVectorMatrix> SPLRRIP::divDevReff(volVectorField& U) const
{
    if (couplingFactor_.value() > 0.0)
    {
        return
        (
            fvc::div(R_ + couplingFactor_*nut_*fvc::grad(U), "div(R)")
          + fvc::laplacian
            (
                 (1.0 - couplingFactor_)*nut_,
                 U,
                 "laplacian(nuEff,U)"
            )
            - fvm::laplacian(nuEff(), U)
        );
    }
    else
    {
        return
        (
            fvc::div(R_)
          + fvc::laplacian(nut_, U, "laplacian(nuEff,U)")
          - fvm::laplacian(nuEff(), U)
        );
    }
}

tmp<fvVectorMatrix> SPLRRIP::divDevRhoReff
(
    const volScalarField& rho,
    volVectorField& U
) const
{
    volScalarField muEff("muEff", rho*nuEff());

    if (couplingFactor_.value() > 0.0)
    {
        return
        (
            fvc::div
            (
                rho*R_ + couplingFactor_*(rho*nut_)*fvc::grad(U),
                "div((rho*R))"
            )
          + fvc::laplacian
            (
                (1.0 - couplingFactor_)*rho*nut_,
                U,
                "laplacian(muEff,U)"
            )
          - fvm::laplacian(muEff, U)
        );
    }
    else
    {
        return
        (
            fvc::div(rho*R_)
          + fvc::laplacian(rho*nut_, U, "laplacian(muEff,U)")
          - fvm::laplacian(muEff, U)
        );
    }
}
```

```
bool SPLRRIP::read()
{
    if (RASModel::read())
    {
        Cmu_.readIfPresent(coeffDict());
        Clrr1_.readIfPresent(coeffDict());
        Clrr2_.readIfPresent(coeffDict());
        C1_.readIfPresent(coeffDict());
        C2_.readIfPresent(coeffDict());
        Cs_.readIfPresent(coeffDict());
        Ceps_.readIfPresent(coeffDict());
        sigmaEps_.readIfPresent(coeffDict());
        C1Ref_.readIfPresent(coeffDict());
        C2Ref_.readIfPresent(coeffDict());
        couplingFactor_.readIfPresent(coeffDict());

        if (couplingFactor_.value() < 0.0 || couplingFactor_.value() > 1.0)
        {
            FatalErrorIn("SPLRRIP::read()")
                << "couplingFactor = " << couplingFactor_
                << " is not in range 0 - 1"
                << exit(FatalError);
        }

        return true;
    }
    else
    {
        return false;
    }
}

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

    if (!turbulence_)
    {
        return;
    }
    if (mesh_.changing())
    {
        yr_.correct();
    }

    volSymmTensorField P(-twoSymm(R_ & fvc::grad(U_))); //P_ij
    volScalarField G(type() + ".G", 0.5*mag(tr(P))); //P
    //volScalarField G(GName(), 0.5*mag(tr(P))); //for newer OF versions

//******************************ADDITIONS TO
LRRIP**********************************************
```

```
 xn = wallDist(mesh_).y(); //Normal distance to wall

  const fvPatchList& Boundaries = mesh_.boundary();
    forAll(Boundaries, patchi) //loops through boundaries, patchi is the index
    {
        const fvPatch& currPatch = Boundaries[patchi]; //indexed boundary definition
(current patch)
        if (isType<wallFvPatch>(currPatch))
        {
            utauw.boundaryField()[patchi] =
                        sqrt
                        (
                            nu()*mag(U_.boundaryField()[patchi].snGrad())
                        );
            forAll(currPatch, facei)
            {
                label faceCelli = currPatch.faceCells()[facei]; //indexed face in current patch
                // Assign utau[on indexed cell face] value from utauw[on boundary][at each
boundary                              face]
                utauFaces[faceCelli] = utauw.boundaryField()[patchi][facei];
            //utau[faceCelli] = utauw.boundaryField()[patchi][facei];

                 forAll(utau, celli) //assigns value of utau[at face] to utau[cells]
            {
                utau[celli] = 0.727; //value from experimental paper (should be fixed for
looping)
        }
            }
        }
    }


//Damping wall functions:
const scalarField& nuCells=nu()().internalField();
forAll(f1,celli)
{
        if (utau[celli] == 0.0)
        {
        f1[celli]= scalar(0.0);
        }else
        {f1[celli] = exp(-0.5*xn[celli]*utau[celli]/nuCells[celli]);}
}

argf2= sqr(k_)/(6.0*nu()*epsilon_);
f2 = 1-2.0/9.0*Foam::exp(-1.0*sqr(argf2));
//*******************************END ADDITIONS TO
LRRIP*******************************************

    // Update epsilon and G at the wall
    epsilon_.boundaryField().updateCoeffs();
    // Dissipation equation
    tmp<fvScalarMatrix> epsEqn
    (
        fvm::ddt(epsilon_)  //change in time
      + fvm::div(phi_, epsilon_) //convective term
      //- fvm::laplacian(DepsilonEff(), epsilon_)
      - fvm::laplacian(DissDest(), epsilon_) //NEW LINE
    //- fvm::laplacian(Ceps_*(k_/epsilon_)*R_, epsilon_) ^^DissDestruction of dissip(pg 11
of RST Doc)
      -fvm::laplacian(nu(), epsilon_) // Molecular part of DepsilonEff
      ==
        C1_*G*epsilon_/k_ //Production of dissipation
      - fvm::Sp(C2_*f2*epsilon_/k_, epsilon_) //ADDED f2 TO
LRRIP****************************
      -2.0/sqr(xn)*nu()*epsilon_*f1 // ADDITION TO
LRRIP************************************
    );

    epsEqn().relax();
    epsEqn().boundaryManipulate(epsilon_.boundaryField());

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

    // Reynolds stress equation
    const fvPatchList& patches = mesh_.boundary();

    forAll(patches, patchi)
    {
        const fvPatch& curPatch = patches[patchi];

        if (isA<wallFvPatch>(curPatch))
        {
            forAll(curPatch, facei)
            {
                label faceCelli = curPatch.faceCells()[facei];
                P[faceCelli] *= min
                (
                    G[faceCelli]/(0.5*mag(tr(P[faceCelli])) + SMALL),
                    1.0
                );
            }
        }
    }

//Reflection Equation................................
    const volSymmTensorField reflect
    (
        C1Ref_*epsilon_/k_*R_ - C2Ref_*Clrr2_*dev(P)
    );
//..................................................
```

```
    tmp<fvSymmTensorMatrix> REqn
    (
        fvm::ddt(R_)
      + fvm::div(phi_, R_)
      - fvm::laplacian(DandH(), R_) //Daly & Harlow
      //- fvm::laplacian(Cs_*(k_/epsilon_)*R_, R_) // ^^^Daly & Harlow
      - fvm::laplacian(nu(), R_) // Molecular component of DREff()
      //- fvm::laplacian(DREff(), R_)
      + fvm::Sp(Clrr1_*epsilon_/k_, R_)
      ==
        P
      +(2.0/3.0*(Clrr1_)*I)*epsilon_ //Rotta's Term  (Split OpenFOAM term into two)
      -(2.0/3.0*I)*epsilon_
      - Clrr2_*dev(P) //Second term in -IP
      -2.0/sqr(xn)*nu()*R_ // Second part of Dissipation tensor definition****

//wall reflection terms ........................................
       + symm
       (
           I*((yr_.n() & reflect) & yr_.n())
         - 1.5*(yr_.n()*(reflect & yr_.n())
         + (yr_.n() & reflect)*yr_.n())
       )*0.2*pow(k_, 1.5)/(yr_*epsilon_)
//................................................................
    );

    REqn().relax();
    solve(REqn);

    R_.max
    (
        dimensionedSymmTensor
        (
            "zero",
            R_.dimensions(),
            symmTensor
            (
                kMin_.value(), -GREAT, -GREAT,
                kMin_.value(), -GREAT,
                kMin_.value()
            )
        )
    );

    k_ = 0.5*tr(R_); //Matches
    bound(k_, kMin_);

    // Re-calculate viscosity
    nut_ = Cmu_*sqr(k_)/epsilon_;
    nut_.correctBoundaryConditions();
    // Correct wall shear stresses
    forAll(patches, patchi)
    {
        const fvPatch& curPatch = patches[patchi];

        if (isA<wallFvPatch>(curPatch))
        {
            symmTensorField& Rw = R_.boundaryField()[patchi];

            const scalarField& nutw = nut_.boundaryField()[patchi];

            const vectorField snGradU(U_.boundaryField()[patchi].snGrad());

            const vectorField& faceAreas
                = mesh_.Sf().boundaryField()[patchi];

            const scalarField& magFaceAreas
                = mesh_.magSf().boundaryField()[patchi];

            forAll(curPatch, facei)
            {
                // Calculate near-wall velocity gradient
                tensor gradUw
                    = (faceAreas[facei]/magFaceAreas[facei])*snGradU[facei];

                // Calculate near-wall shear-stress tensor
                tensor tauw = -nutw[facei]*2*symm(gradUw);

                // Reset the shear components of the stress tensor
                Rw[facei].xy() = tauw.xy();
                Rw[facei].xz() = tauw.xz();
                Rw[facei].yz() = tauw.yz();
            }
        }
    }
}


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

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

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

LRR-IP Header file:

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2012 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Class
    Foam::incompressible::RASModels::SPLRRIP

Group
    grpIcoRASTurbulence

Description
    Launder, Reece and Rodi Reynolds-stress turbulence model for
    incompressible flows.

    The default model coefficients correspond to the following:
    \verbatim
        SPLRRIPCoeffs
        {
            Cmu         0.09;
            Clrr1       1.8;
            Clrr2       0.6;
            C1          1.44;
            C2          1.92;
            Cs          0.25;
            Ceps        0.15;
            sigmaEps    1.3;
            couplingFactor  0.0;    // only for incompressible
        }
    \endverbatim

SourceFiles
```

```
    SPLRRIP.C

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

#ifndef SPLRRIP_H
#define SPLRRIP_H

#include "RASModel.H"
#include "wallDist.H" //ADDED *******************************************
#include "wallDistReflection.H"//ADDED *******************************************

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

namespace Foam
{
namespace incompressible
{
namespace RASModels
{

/*---------------------------------------------------------------*\
                Class SPLRRIP Declaration
\*---------------------------------------------------------------*/

class SPLRRIP
:
    public RASModel
{

protected:

    // Protected data

        // Model coefficients
            dimensionedScalar Cmu_;
            dimensionedScalar Clrr1_;
            dimensionedScalar Clrr2_;
            dimensionedScalar C1_;
            dimensionedScalar C2_;
            dimensionedScalar Cs_;
            dimensionedScalar Ceps_;
            dimensionedScalar sigmaEps_;
            dimensionedScalar couplingFactor_;

        // Fields
            volSymmTensorField R_;
            volScalarField k_;
            volScalarField epsilon_;
            volScalarField nut_;
//*********ADDITIONS TO SPLRRIP******************************
            volScalarField xn;
            volScalarField utauw;
            volScalarField utau;
            volScalarField utauFaces;
            volScalarField f1;
            volScalarField argf2;
            volScalarField f2;
            wallDistReflection yr_; // ADDED
                dimensionedScalar C1Ref_;// ADDED
                dimensionedScalar C2Ref_;// ADDED
//*************END ADDITIONS TO SPLRRIP*********************

public:
    //- Runtime type information
    TypeName("SPLRRIP");

    // Constructors
        //- Construct from components
        SPLRRIP
        (
            const volVectorField& U,
            const surfaceScalarField& phi,
            transportModel& transport,
            const word& turbulenceModelName = turbulenceModel::typeName,
            const word& modelName = typeName
        );

    //- Destructor
    virtual ~SPLRRIP()
    {}

    // Member Functions

        //- Return the turbulence viscosity
        virtual tmp<volScalarField> nut() const
        {
            return nut_;
        }

        //- Return the effective diffusivity for R
        tmp<volScalarField> DREff() const
        {
            return tmp<volScalarField>
            (
                new volScalarField("DREff", nut_ + nu())
            );
        }

        //- Return the effective diffusivity for epsilon
        tmp<volScalarField> DepsilonEff() const
        {
            return tmp<volScalarField>
            (
                new volScalarField("DepsilonEff", nut_/sigmaEps_ + nu())
            );
        }

        //- Return the turbulence kinetic energy
        virtual tmp<volScalarField> k() const
        {
            return k_;
        }

        //- Return the turbulence kinetic energy dissipation rate
        virtual tmp<volScalarField> epsilon() const
        {
            return epsilon_;
        }

        //- Return the Reynolds stress tensor
        virtual tmp<volSymmTensorField> R() const
        {
            return R_;
        }

        //- Return the effective stress tensor including the laminar stress
        virtual tmp<volSymmTensorField> devReff() const;

        //- Return the source term for the momentum equation
        virtual tmp<fvVectorMatrix> divDevReff(volVectorField& U) const;

        //- Return the source term for the momentum equation
        virtual tmp<fvVectorMatrix> divDevRhoReff
        (
            const volScalarField& rho,
            volVectorField& U
        ) const;

//**************START ADDITIONS***************************
        //- Return term for Dissipation equation (destruction term on line 352)
        tmp<volSymmTensorField> DissDest() const
        {
            return tmp<volSymmTensorField>
            (
                new volSymmTensorField("DissDest", Ceps_*(k_/epsilon_)*R_)
            );
        }

        //- Return term for Daly & Harlow Term in R equation (line 395)
        tmp<volSymmTensorField> DandH() const
        {
            return tmp<volSymmTensorField>
            (
                new volSymmTensorField("DandH", Cs_*(k_/epsilon_)*R_)
            );
        }
//******************END ADDITIONS**********************

        //- Solve the turbulence equations and correct the turbulence viscosity
        virtual void correct();

        //- Read RASProperties dictionary
        virtual bool read();
};

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

// * * * * * * * * * * * * * * * * * * * * * * * * * * * //
#endif
// ************************************************ //
```

C. OpenFOAM Case Files (located in system)

controlDict

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

application     simpleFoam;
startFrom       latestTime;
startTime       0;
stopAt          endTime;
endTime         1;
deltaT          .00001;
writeControl    timeStep;
writeInterval   10000;
purgeWrite      0;
writeFormat     ascii;
writePrecision  6;
writeCompression off;
timeFormat      general;
timePrecision   6;
runTimeModifiable true;
libs ("libmyIncompressibleRASModels.so");
// ************************************************************************* //
```

fvSchemes

```
/*--------------------------------*- 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)      bounded Gauss linearUpwind grad(U);
    div(phi,epsilon)  bounded Gauss upwind;
    div(phi,omega)  bounded Gauss upwind;
    div(phi,k)      bounded Gauss upwind;
    div(phi,R)      bounded Gauss upwind;
    div(R)          Gauss linear;
    div((nuEff*dev(T(grad(U))))) Gauss linear;
    div(DomegaEff,omega) bounded Gauss upwind;

}

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;
    laplacian(DomegaEff,omega) Gauss linear corrected;
    laplacian(phi,omega) Gauss linear corrected;
}

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

snGradSchemes
{
    default         corrected;
}

fluxRequired
{
    default         no;
    p               ;
}
//************************************************************ //
```

fvSolution

```
/*--------------------------------*- 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      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
solvers
{
    p
    {
        solver          PCG;
        preconditioner  FDIC;
        tolerance       1e-16;
        relTol          0;
    }
    U
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-16;
        relTol          0;
    }
    k
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-16;
        relTol          0;
    }
    epsilon
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-16;
        relTol          0;
    }
    R
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance      1e-16;
        relTol         0;
    }
    nuTilda
    {
        solver         PBiCG;
        preconditioner  DILU;
        tolerance      1e-16;
        relTol         0;
    }
    omega
    {
        solver         PBiCG;
        preconditioner  DILU;
        tolerance      1e-16;
        relTol         0;
    }
}

SIMPLE
{
    nNonOrthogonalCorrectors 0;
}
relaxationFactors
{
    fields
    {
        p             0.3;
    }
    equations
    {
        U             0.7;
        k             0.7;
        epsilon        0.7;
        R             0.7;
        nuTilda        0.7;
        omega              0.7;
    }
}
//********************************************************* //
```