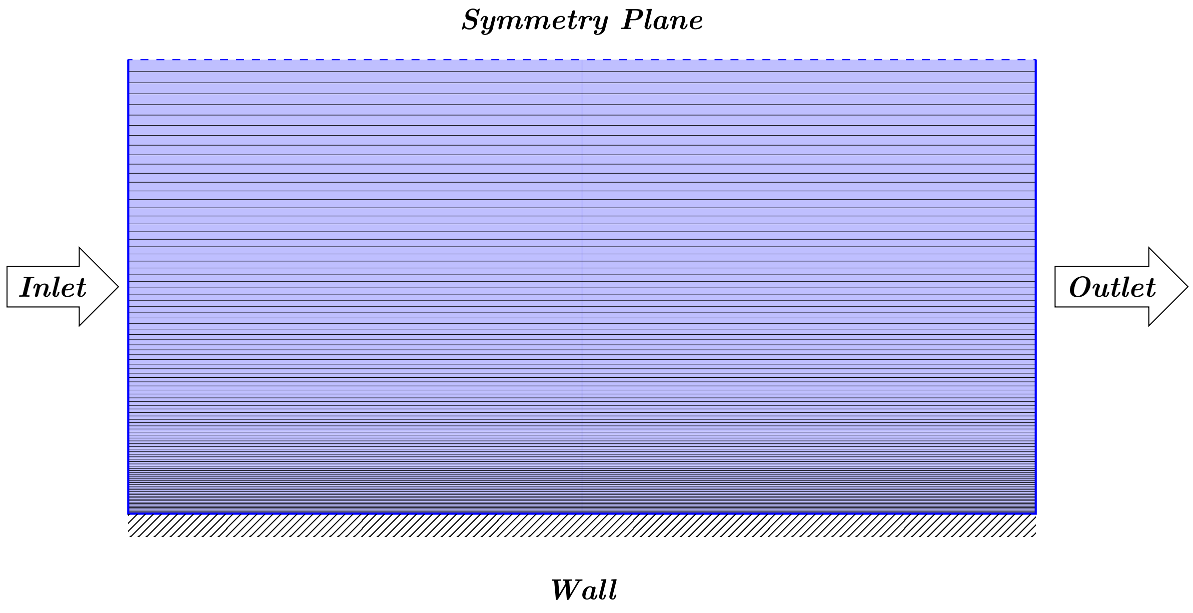


# Investigation of turbulence models for two dimensional mean flows and implementation in OpenFOAM

*Master's Thesis in Automotive Engineering*

MANAN LALIT

Department of Applied Mechanics
Chalmers University of Technology
Gothenburg, Sweden 2014

Master's Thesis 2014:78

Investigation of turbulence models for two-dimensional mean flows and implementation in OpenFOAM

Master's Thesis 2014:78

Department of Applied Mechanics
Division of Fluid Dynamics
Chalmers University of Technology
SE-41296 Göteborg
Sweden

*Tel.* +46(0)31-772 10 00

To *Begoña* and *Mike*

Investigation of turbulence models for two dimensional mean flows and implementation in OpenFOAM
Master's Thesis in the Master's programme in Automotive Engineering
MANAN LALIT
Department of Applied Mechanics
Division of Fluid Dynamics
Chalmers University of Technology

## Abstract

With an intention of discovering the best RANS model for modeling impinging flows, linear epsilon - based models('Chien's Low Reynolds' and '$\phi$-f') and non-linear($\epsilon$ and $\omega$- based) EARSM formulations were implemented on OpenFOAM-2.2.x and validated against two dimensional, developed channel flow geometries.

The project arose from the need to capture flow arising from a flame impingement on the walls of an (internal combustion) engine cylinder. The objective desired from this project is to achieve directives regarding 'optimal' turbulence models, defined as those which sufficiently reproduce the flow dynamics, and are applicable over a wide variety of meshes, while not being too costly on the computational front. A rating scheme for judging the quality of turbulence models, based on the aforementioned three characteristics, is discussed as part of the conclusions from this thesis project.

Keywords: Chien, EARSM, Wallin-Johansson, Menter, k $\omega$ BSL, Channel, OpenFOAM

## Acknowledgements

I began this thesis in September 2013 and spent a good portion of a year understanding OpenFOAM and the basics of CFD theory. During this period, I faced a lot of pitfalls and moments of self-doubt and had it not been for the support of many kind souls, I would have found it hard to bring the thesis work to where it stands now.

I am indebted to Dr. Lars Davidson, my examiner, for his insightful suggestions about work and otherwise, which often helped me land myself out of a pickle and gave new perspectives about looking at a problem. Jon-Anders Bäckar, my supervisor, was always available with time, words of counsel and links to excellent reference material. In my dealing with him, he was extremely kind and comradely. It was always a delight sharing my new findings with him, which he would listen to with a very patient ear.

I can't speak enough about the invaluable role of my colleague Ansuman. He would tranquilly listen to any doubts I would have and would give a sanguine, well-thought reply, no matter how tied up with work he was at that moment. Ansuman, and the other thesis workers, namely Sankar, Himanshu and Axel also helped me clear my head on many occasions by providing a bouncing board for my thesis-related problems or by sharing some new development from around the world. Karthik, my friend and batchmate provided many an entertaining discussion over fika or otherwise. Lastly, I am deeply grateful to my parents and sister. They have always been extremely positive about my endeavors, and that has allowed me to take my own time, constantly questioning what it is that I am doing.

To all of you, I am truly thankful!

# Contents

# 1 Introduction

The work of this thesis was inspired by the research problem of modeling heat transfer by impinging fuel jets on the inside walls of diesel engine cylinders. There was a gap in the accuracy of the results from existing RANS-based turbulence models on the one hand and the burgeoning computational and temporal cost of LES-based techniques on the other.

The following features were desired in the turbulence model:

**Fidelity** Higher degree of accuracy

**Robustness** Lack of dependence on mesh

**Cost** Lower requirement of time/computational resources

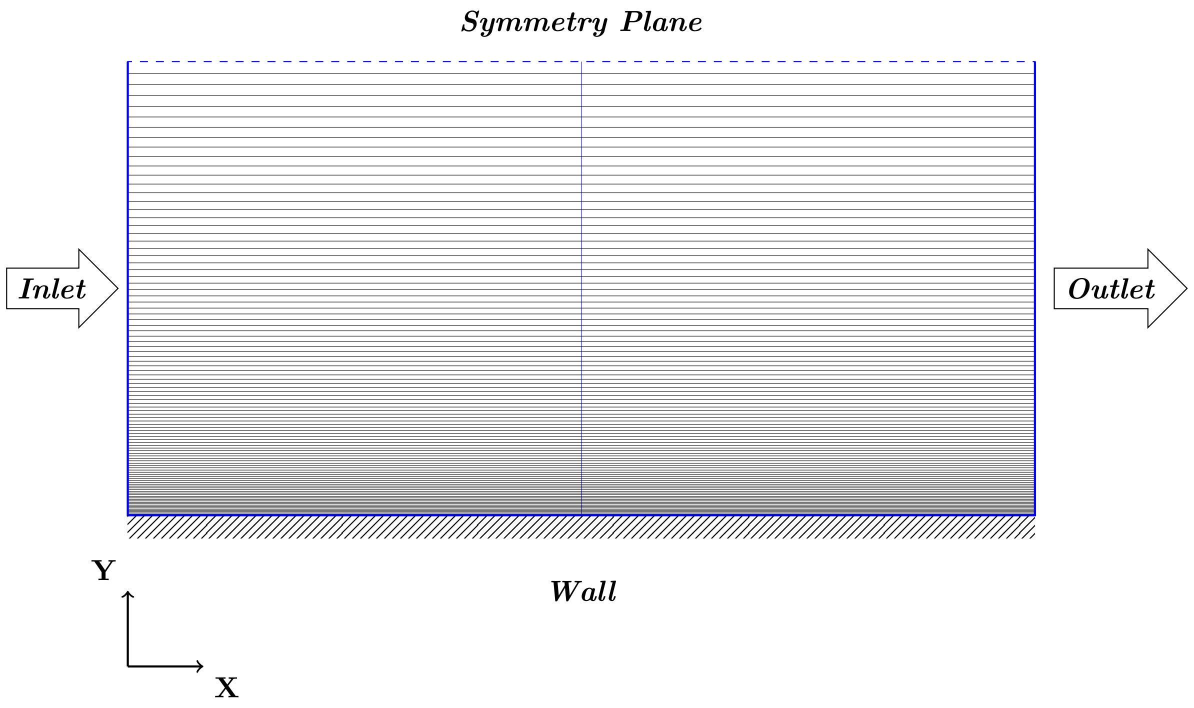
One could assume from the description of the problem statement that there would be a certain degree of streamline curvature upon impingement and flow separation caused by roughness on the walls of the engine. Hence one could safely also conclude based on existing research that linear eddy viscosity models would not suit the need.

The objective of this thesis is to propose a list of turbulence models worth investigation for modeling heat transfer by impinging jets on curved walls. The applicability of a range of models for 2-D developed channel flow in OpenFOAM-2.2-x was considered by comparing the simulation results with DNS data at a specific Reynolds Number. The models which showed interesting results are discussed in the subsequent pages.

In this process of investigation, some models were implemented for OpenFOAM-2.2-x. It is hoped that the list of suitable candidates can be further assessed in steps by looking at results with the following cases : pipe flow (instead of the current channel flow), impingement of jet flame on a flat wall, impingement of jet flame on a curved wall and finally impingement of flame on the actual inner cylinder geometry.

## 2 Constant Properties

### 2.1 Channel Geometry



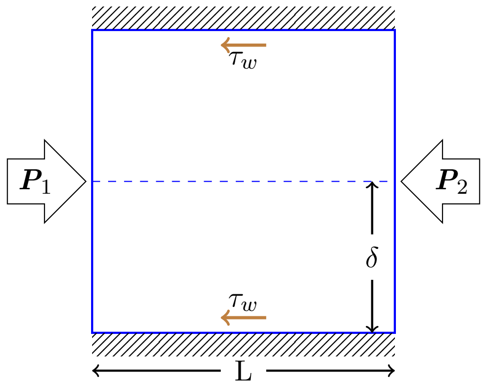
The channel mesh is generated by the ‘blockMesh’ utility. There are a total of 2 * 100 * 1 cells. Apart from the four patches in the figure above (‘inlet’, ‘outlet’, ‘wall’, ‘symmetryPlane’), the z-faces are named as ‘frontAndBack’ and are classified as ‘empty’ patches.

An expansion ratio of 10 is given in the y - direction. This means that the ratio of the thickest cell (in the mid channel region) and the thinnest cell (near the wall) is equal to 10.

$$\frac{ar^{n-1}}{a} = 10$$
$$r = 10^{\frac{1}{n-1}}$$
$$r = 1.0235$$

where a = height of the thinnest cell, r is the growth ratio and n = 100.

## 2.2 Transport Properties



By force balance on the whole channel domain,

$$(P_1 - P_2)(2\delta) = (2\tau_w) L$$
$$\frac{\Delta P}{L} = \tau_w$$

By choosing L = 1.0, $P_1$ = 1.0, $P_2$ = 0.0 and $\delta = 1.0$, the value for $\tau_w$ is obtained as 1.0 S.I. Units.

Also since

$$\tau_w = \rho u_\tau^2$$

Hence, by choosing $\rho = 1.0$, gives us $u_\tau = 1.0$.

The results have been run at $Re_\tau = 395$ to correspond with DNS data of Moser, Kim and Mansour [1]. This has been done by setting the molecular viscosity accordingly.

$$Re_\tau = \frac{\delta u_\tau}{\nu}$$
$$\nu = \frac{1}{395}$$

## 2.3 Boundary Conditions

Since fully developed flow is being modeled, the inlet and outlet properties of the velocity field have been assigned as 'zero Gradient', while the pressure field has been assigned as 'fixed Value' along the respective patches to generate a constant pressure gradient.

The 0/U file for the velocity field is specified as follows:

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

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

internalField   uniform (0 0 0);

boundaryField
{
  inlet
    {
      type            zeroGradient;
        }

  outlet
    {
      type            zeroGradient;
        }

  fixedWalls
    {
      type            fixedValue;
      value           uniform (0 0 0);

    }

  frontAndBack
    {
      type            empty;
    }

  middle
   {
    type            symmetryPlane;

  }

}
```

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

The 0/P file for the pressure field is specified as follows:

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

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

internalField   uniform 0;

boundaryField
{
  inlet
    {
      type            fixedValue;
      value           uniform 1.0;

    }

  outlet
    {
      type            fixedValue;
      value           uniform 0.0;
    }

  fixedWalls
    {
      type            zeroGradient;
    }

  frontAndBack
    {
      type            empty;
    }
```

```
 middle
   {
     type            symmetryPlane;
   }


}

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

# 3 Chien Low Reynolds kEpsilon Model

Standard kEpsilon model is known for its inability to integrate through the viscous sublayer. That motivates a need for low Reynolds kEpsilon versions which accurately account for the order of turbulent quantities near the wall.

Among the low Reynold versions, the model by Chien [2] has been chosen. Although the general treatment to account for the difference in order 'O(y)' of the modelled and the actual value of the turbulent quantities is the same as Jones and Launder (1972), the detailed proposals are quite different.

## 3.1 Theory

In general, near the wall, because of no slip conditions and continuity relations, we have the following relations for velocity $\vec{v}$ = (u,v,w) and turbulent kinetic energy k:

$$u = O(y^1) \quad v = O(y^2) \quad w = O(y^1) \quad k = O(y^2) \tag{3.1.1}$$

The standard kEpsilon formulation by Jones and Launder (1972) appears as follows. It would need to be altered to suit the Low Reynolds flow.

The Jones and Launder equations for the Standard k-Epsilon model are as follows:

$$\frac{\mathrm{d}k}{\mathrm{d}t} = \frac{\partial}{\partial y}\left(\nu + \frac{\nu_t}{\sigma_k}\right)\frac{\partial k}{\partial y} + \nu_t\left(\frac{\partial u}{\partial y}\right)^2 - \epsilon$$

$$\frac{\mathrm{d}\epsilon}{\mathrm{d}t} = \frac{\partial}{\partial y}\left(\nu + \frac{\nu_t}{\sigma_\epsilon}\right)\frac{\partial \epsilon}{\partial y} + c_1\frac{\epsilon}{k}\nu_t\left(\frac{\partial u}{\partial y}\right)^2 - c_2\frac{\epsilon^2}{k}$$

where the turbulent (eddy) viscosity is defined as:

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

and the constants are:

$$c_\mu = 0.09 \quad c_1 = 1.35 \quad c_2 = 1.8 \quad \sigma_k = 1 \quad \sigma_\epsilon = 1.3 \tag{3.1.2}$$

Chien [2] firstly considered the k equation of Jones and Launder. It is evident that near the wall, the molecular diffusion term is a constant while other terms go to 0. To balance the molecular diffusion term, he introduced $D$ such that:

$$\epsilon = \tilde{\epsilon} + D \tag{3.1.3}$$

While the order of $\epsilon$ is known to be O($y^0$), the order of $\tilde{\epsilon}$ needs to be investigated. By Taylor's expansion, k expands out as follows:

$$k = a_2 y^2 + a_3 y^3 + a_4 y^4 + ...$$

$$\frac{\partial k}{\partial y} = 2a_2 y + 3a_3 y^2 + 4a_4 y^3 + ...$$

$$\nu \frac{\partial k}{\partial y} = \nu \left[2a_2 y + 3a_3 y^2 + 4a_4 y^3 + ...\right]$$

$$\frac{\partial}{\partial y} \nu \frac{\partial k}{\partial y} = \nu \left[2a_2 + 6a_3 y^1 + 12a_4 y^2 + ...\right]$$

The term on the left hand side at the last line is the molecular diffusion near the wall. Since $a_2 \simeq \frac{k}{y^2}$, hence one concludes that the term 'D' that balances the molecular diffusion is:

$$D = 2\nu \frac{k}{y^2} \tag{3.1.4}$$

The final k equation for the Chien Model is as follows:

$$\frac{\mathrm{d}k}{\mathrm{d}t} = \frac{\partial}{\partial y}\left[\left(\nu + \frac{\nu_t}{\sigma_k}\right)\frac{\partial k}{\partial y}\right] + \nu_t \left(\frac{\partial u}{\partial y}\right)^2 - \tilde{\epsilon} - \frac{2\nu k}{y^2} \tag{3.1.5}$$

Chien [2] then re-modified the turbulent viscosity formulation to take into account the wall damping. The new equation for $\nu_t$ is:

$$\nu_t = c_\mu \frac{k^2}{\tilde{\epsilon}} \left[1 - \mathrm{e}^{-c_3 u_\tau y/\nu}\right] \tag{3.1.6}$$

The equation above gives an insight into the (near-wall) order of $\tilde{\epsilon}$. Since the order of $\nu_t$ near-wall is $O\left(y^3\right)$, as it is determined purely by the Boussinesq Assumption (Constitutive Relations between $\tau_{xy}$ which is $O\left(y^3\right)$ and $S_{12}$ which is $O\left(y^0\right)$), hence

$$O\left(y^3\right) = \frac{O\left(y^4\right)}{O\left(\tilde{\epsilon}\right)} O\left(y^1\right)$$

This implies that the near-wall order of $\tilde{\epsilon}$ is $O(y^2)$. This is so because of Taylor's expansion near the wall:

$$1 - \mathrm{e}^{-c_3 u_\tau y/\nu} = 1 - \left[1 - O\left(y^1\right) + O\left(y^2\right) - ...\right]$$

A word of caution here: Since $u_\tau$ is constant across the mesh and hence across 'y', $O\left(u_\tau\right) \simeq O\left(y^0\right)$

The final step is the conversion of Jones-Launder equation for $\epsilon$ into an equation for $\tilde{\epsilon}$. The final transport equation for $\tilde{\epsilon}$ is as follows:

$$\frac{\mathrm{d}\tilde{\epsilon}}{\mathrm{d}t} = \frac{\partial}{\partial y}\left[\left(\nu + \frac{\nu_t}{\sigma_{\tilde{\epsilon}}}\right)\frac{\partial \tilde{\epsilon}}{\partial y}\right] + c_1 \frac{\tilde{\epsilon}}{k} \nu_t \left(\frac{\partial u}{\partial y}\right)^2 - \frac{\tilde{\epsilon}}{k}\left[c_2 f_2 \tilde{\epsilon} + \frac{2\nu k \mathrm{e}^{-c_4 u_\tau y/\nu}}{y^2}\right] \tag{3.1.7}$$

where:

$$f_2 = 1 - \frac{0.4}{1.8} \mathrm{e}^{-\left(k^2/(6\nu\tilde{\epsilon})\right)^2} \quad c_3 = 0.0115 \quad c_4 = 0.5$$

## 3.2 Implementation

### 3.2.1 Wall Scaling

Current wall damping function $f_\mu$ is a function of $y^+$ as can be seen in (3.1.6). However the formulation becomes invalid in regions of separation and reattachment.

Hence, an alternate scaling $y^+$ is proposed as follows:

$$Re_y = \frac{\sqrt{k}y}{\nu} \tag{3.2.1}$$

$$y^* = 2.4\sqrt{Re_y} + 0.003Re_y^2 \tag{3.2.2}$$

Wallin and Johansson [3] have shown that $y^* \simeq y^+$ for $y^+ \leq 100$.

### 3.2.2 Boundary Conditions

The boundary conditions for turbulent kinetic energy k is as follows:

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

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

internalField   uniform 0.325;

boundaryField
{
    inlet
      {
        type            zeroGradient;
      }
    outlet
      {
        type            zeroGradient;
      }

    fixedWalls
      {
        type            fixedValue;
        value           uniform 1e-10;
      }

    frontAndBack
      {
        type            empty;
      }

    middle
   {
        type            symmetryPlane;
   }


}


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

The boundary conditions for turbulent dissipation $\epsilon$ is as follows:

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

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

internalField   uniform 0.000765;

```

```
boundaryField
{
  inlet
    {
      type            zeroGradient;
        }

  outlet
    {
      type            zeroGradient;
        }

  fixedWalls
    {
      type            fixedValue;
      value           uniform 1e-10;

    }

  frontAndBack
    {
      type            empty;
    }

  middle
   {
     type            symmetryPlane;
   }


}


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

## 3.3 Verification

Chien [2] in his 1982 publication, where he introduces the model, publishes results for $\mathrm{Re}_b = 3850$. Upon running the simulation at the same flow conditions ($\mathrm{Re}_\tau = 228$), the results match well with Chien's observations. The published data below in Figure (3.1) is taken from Figure 3 in Chien [2] 's publication and is added for the sake of verification and completeness. Results of U and $\epsilon$ were not included in the aforementioned publication and hence, performance of the current model with regards to U and $\epsilon$ has not been discussed here.

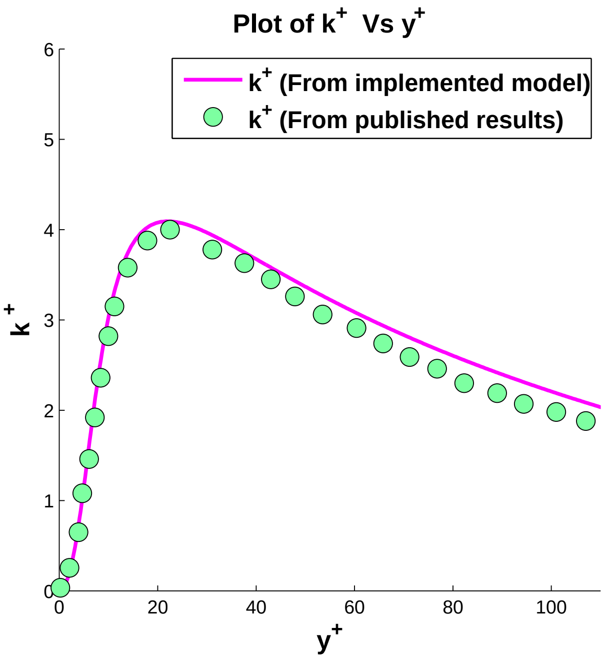


**Figure 3.1:** $k^+$ vs $y^+$ : Chien Model at $\mathrm{Re}_\tau = 228$ or $\mathrm{Re}_b = 3850$

# 4 EARSM

The linear eddy viscosity models have one major drawback: that the constitutive relations between stresses and strains is a function of an isotropic turbulent viscosity $\nu_t$.

To alter that, and to see if any improvement is visibly seen in the performance of the eddy viscosity models, the EARSM formulation is added.

Replacing the Boussinesq Assumption, EARSM seeks to redefine the constitutive relation between Reynolds stress ($\vec{\vec{\tau}}$, where $\tau_{ij} = -\rho\overline{u_i'u_j'}$) and the mean velocity strain tensor ($\vec{\vec{S}}$, where $S_{ij} = \left(\bar{U}_{i,j} + \bar{U}_{j,i}\right)$ ). $\vec{\vec{a}}$ (which is a function of higher order multiples of $\vec{\vec{S}}$ and $\vec{\vec{\Omega}}$ (where $\Omega_{ij} = \left(\bar{U}_{i,j} - \bar{U}_{j,i}\right)\big)$ ) will be determined in the equations below:

$$\overline{u_i'u_j'} = k\left(a_{ij} + \frac{2}{3}\delta_{ij}\right) \tag{4.0.1}$$

The EARSM formulation begins with the following declaration:

$$\text{RSM Equation: } C_{ij} - D_{ij} = P_{ij} + II_{ij} - \epsilon_{ij}$$

$$k \text{ Equation: } C_k - D_k = P_k - \epsilon_k$$

where C is the convection term, D is the diffusion term, P is the production term, $\epsilon$ is the dissipation term and II is the pressure strain term.

Next comes the following assumption:

$$C_{ij} - D_{ij} = \frac{\overline{u_i'u_j'}}{k}\left(C_k - D_k\right) \tag{4.0.2}$$

This implies that:

$$\boxed{P_{ij} + II_{ij} - \epsilon_{ij} = \frac{\overline{u_i'u_j'}}{k}\left(P_k - \epsilon_k\right)}$$

This assumptive tactic above seeks to find $\overline{u_i'u_j'}$ in terms of the other terms. This implies that one must have the models for $\epsilon_{ij}$ and $II_{ij}$.

Substitution of the models leads up to a simplified but still implicit equation for $\bar{\bar{a}}$.

$$\left(c_1 - 1 + \frac{P_k}{\epsilon}\right)\vec{\vec{a}} = -\frac{8}{15}\vec{\vec{S}} + \frac{4}{9}\left(\vec{\vec{a}}\vec{\vec{\Omega}} - \vec{\vec{\Omega}}\vec{\vec{a}}\right)$$

## 4.1 Wallin Johansson Formulation

### 4.1.1 Theory (for Two Dimensional Mean Flows)

Expressing $\vec{\vec{a}}$ in terms of tensorially independent groups using the Cayley Hamilton Theorem, for 2D flow, we have the following equation:

$$\vec{\vec{a}} = \beta_1 \vec{\vec{S}} + \beta_4 \left( \vec{\vec{S}}\vec{\vec{\Omega}} - \vec{\vec{\Omega}}\vec{\vec{S}} \right)$$

where,

$$\beta_1 = -\frac{6}{5} \frac{N}{N^2 - 2II_\Omega}$$

and,

$$\beta_4 = -\frac{6}{5} \frac{1}{N^2 - 2II_\Omega}$$

In the equations above 'N' is the solution to the cubic equation:

$$N^3 - c_1^{'} N^2 - \left( \frac{27}{10} II_S + 2II_\Omega \right) N + 2c_1^{'} II_\Omega = 0$$

Here, N can be solved in closed form with the solution:

$$\begin{aligned} N &= \frac{c_1'}{3} + \left(P_1 + \sqrt{P_2}\right)^{1/3} + \text{sign}\left(P_1 - \sqrt{P_2}\right) |P_1 - \sqrt{P_2}|^{1/3}, P_2 \geq 0 \\ &= \frac{c_1'}{3} + 2\left(P_1^2 - P_2\right)^{1/6} \cos\left( \frac{1}{3} \arccos\left( \frac{P_1}{\sqrt{P_1^2 - P_2}} \right) \right), P_2 < 0 \end{aligned}$$

where,

$$\begin{aligned} P_1 &= \left( \frac{1}{27} c_1'^2 + \frac{9}{20} II_s - \frac{2}{3} II_\Omega \right) c_1' \\ P_2 &= P_1^2 - \left( \frac{1}{9} c_1'^2 + \frac{9}{10} II_S + \frac{2}{3} II_\Omega \right)^3 \end{aligned}$$

where,

$$II_S = tr\left(S^2\right) \quad II_\Omega = tr\left(\Omega^2\right) \quad c_1^{'} = \frac{9}{4}\left(c_1 - 1\right) \quad c_1 = 1.8$$

For Low Reynolds flow, Wallin and Johansson [3] recommend some changes to the $\beta_1$, $\beta_2$ and $\beta_4$ coefficients. The damping factor in the equation below is of the Van Driest damping factor form:

$$f_1 = 1 - \mathrm{e}^{\left(-y^+/26\right)}$$

$$\beta_{1,low-Re} = f_1 \beta_1$$

$$\beta_{2,low-Re} = \frac{3\,(1.8) - 4}{2\sigma^2}\left(1 - f_1^2\right)$$

$$\beta_{4,low-Re} = f_1^2\beta_4 - \frac{1.8}{4\sigma^2}\left(1 - f_1^2\right)$$

The above reformulation of the $\beta$ coefficients leads to some changes in the $\bar{\bar{a}}$ tensor (or the $a_{ij}$ values)

In the tensorial format, the 2D anisotropy model would read as follows:

$$\boxed{\begin{aligned}\vec{\vec{a}} = f_1\beta_1\vec{\vec{S}} + \left(1 - f_1^2\right)\frac{3*1.8 - 4}{\max\left(II_S, 5.74\right)}\left(\vec{\vec{S}}^2 - \frac{1}{3}II_S\vec{\vec{I}}\right) \\ + \left(f_1^2\beta_4 - \left(1 - f_1^2\right)\frac{1.8}{2\max\left(II_S, 5.74\right)}\right)\left(\vec{\vec{S}}\vec{\vec{\Omega}} - \vec{\vec{\Omega}}\vec{\vec{S}}\right)\end{aligned}} \tag{4.1.1}$$

In addition, the $C_\mu$ coefficient will no longer be a constant. For two dimensional mean flows, it will be modeled as follows:

$$C_\mu^{eff} = -\frac{f_1\beta_1}{2}$$

Finally, the turbulent eddy viscosity will be calculated differently than before. It shall be:

$$\nu_t = C_\mu^{\text{eff}} k\tau$$

### 4.1.2 Theory (for Three Dimensional Mean Flows)

Expressing $\vec{\vec{a}}$ in terms of tensorially independent groups using the Cayley Hamilton Theorem, for 3D flow, we have the following equation:

$$\boxed{\begin{aligned}\vec{\vec{a}} = \beta_1\vec{\vec{S}} + \beta_2\left(\vec{\vec{S}}^2 - \frac{1}{3}II_S\vec{\vec{I}}\right) + \beta_3\left(\vec{\vec{\Omega}}^2 - \frac{1}{3}II_\Omega\vec{\vec{I}}\right) + \beta_4\left(\vec{\vec{S}}\vec{\vec{\Omega}} - \vec{\vec{\Omega}}\vec{\vec{S}}\right) \\ + \beta_6\left(\vec{\vec{S}}\vec{\vec{\Omega}}^2 + \vec{\vec{\Omega}}^2\vec{\vec{S}} - \frac{2}{3}IV\vec{\vec{I}}\right) + \beta_9\left(\vec{\vec{\Omega}}\vec{\vec{S}}\vec{\vec{\Omega}}^2 - \vec{\vec{\Omega}}^2\vec{\vec{S}}\vec{\vec{\Omega}}\right)\end{aligned}} \tag{4.1.2}$$

where,

$$\beta_1 = -\frac{N\left(2N^2 - 7II_\Omega\right)}{Q}$$

and,

$$\beta_3 = -\frac{12\left(N^{-1}IV\right)}{Q}$$

and,

$$\beta_4 = -\frac{2\left(N^2 - 2II_\Omega\right)}{Q}$$

and,

$$\beta_6 = -\frac{6N}{Q}$$

and,

$$\beta_9 = \frac{6}{Q}$$

Also,

$$Q = \frac{5}{6}\left(N^2 - 2II_\Omega\right)\left(2N^2 - II_\Omega\right) \quad IV = tr\left(\vec{\vec{S}}\vec{\vec{\Omega}}^2\right)$$

For Low Reynolds flow, Wallin and Johansson [3] recommend some changes to the $\beta_1$, $\beta_2$, $\beta_4$, $\beta_6$ and $\beta_9$ coefficients.

$$\beta_{3,low-Re} = f_1^2\beta_3$$

$$\beta_{6,low-Re} = f_1\beta_6$$

$$\beta_{9,low-Re} = f_1^2\beta_9$$

The above reformulation of the $\beta$ coefficients leads to some changes in the $\bar{\bar{a}}$ tensor (or the $a_{ij}$ values)

In the tensorial format, the 3D anisotropy model would read as follows:

$$\boxed{\begin{aligned}\vec{\vec{a}} = {} & f_1\beta_1\vec{\vec{S}} + \left(1 - f_1^2\right)\frac{3*1.8-4}{\max\left(II_S, 5.74\right)}\left(\vec{\vec{S}}^2 - \frac{1}{3}II_S\vec{\vec{I}}\right) + f_1^2\beta_3\left(\vec{\vec{\Omega}}^2 - \frac{1}{3}II_\Omega\vec{\vec{I}}\right) \\ & + \left(f_1^2\beta_4 - \left(1 - f_1^2\right)\frac{1.8}{2\max\left(II_S, 5.74\right)}\right)\left(\vec{\vec{S}}\vec{\vec{\Omega}} - \vec{\vec{\Omega}}\vec{\vec{S}}\right) + f_1\beta_6\left(\vec{\vec{S}}\vec{\vec{\Omega}}^2 + \vec{\vec{\Omega}}^2\vec{\vec{S}} - \frac{2}{3}IV\vec{\vec{I}}\right) \\ & + f_1^2\beta_9\left(\vec{\vec{\Omega}}\vec{\vec{S}}\vec{\vec{\Omega}}^2 - \vec{\vec{\Omega}}^2\vec{\vec{S}}\vec{\vec{\Omega}}\right)\end{aligned}} \tag{4.1.3}$$

In addition, the $C_\mu$ coefficient will no longer be a constant. For three dimensional mean flows, it will be modeled as follows:

$$C_\mu^{eff} = -\frac{f_1\left(\beta_1 + II_\Omega\beta_6\right)}{2}$$

### 4.1.3 Implementation (Chien Low Reynolds kEpsilon)

The Wallin Johansson EARSM implementation was tested with Chien Low Reynolds kEpsilon Model. Upon running the original two dimensional mean flow formulation, it was observed that the mid channel velocity is a little over predicted. Hence, Wallin and Johansson [3] recommend a change to $\epsilon$ in the Chien's Model as follows:

$$\epsilon = \tilde{\epsilon} + \frac{2\nu k}{y^2} \exp\left(-C_k y^+\right)$$

where $C_k$ is chosen as 0.04. This means that the original wall dissipation is multiplied by an exponential function to ensure a more 'rapid decaying'. In the implementation, this exponential function will manifest itself in both the k and the $\tilde{\epsilon}$ equations.

The $y^+$ expression in the exponential function is calculated as follows:

$$\begin{aligned} \tau_w &= -\nu \frac{\mathrm{d}U}{\mathrm{d}y} \\ u_\tau &= \sqrt{\frac{\tau_w}{\rho}} \\ y^+ &= \frac{y u_\tau}{\nu} \end{aligned}$$

### 4.1.4 Verification

The published data in Figures (4.1) and (4.2) is taken from Wallin and Johansson [3] 's publication and is added for the sake of verification. The data matches closely to the results obtained with the implemented model. Results of $\epsilon$ were not included in the aforementioned publication and hence, performance of the current model with regards to $\epsilon$ has not been discussed here.

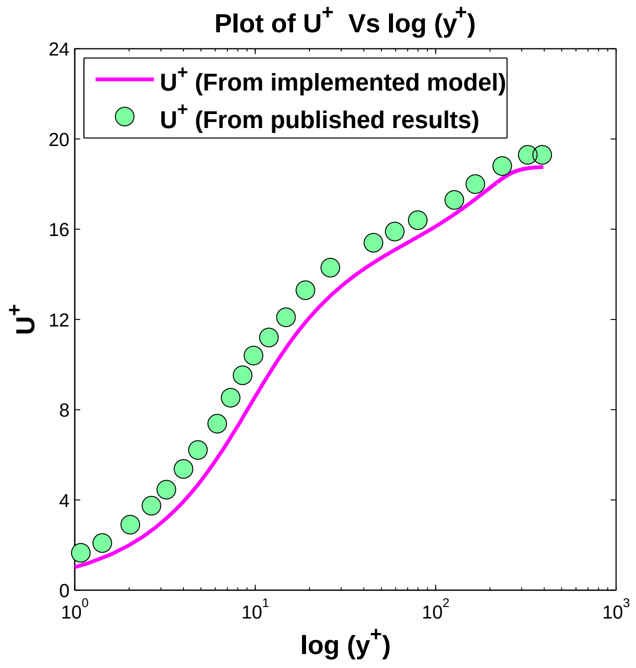


**Figure 4.1:** $U^+$ vs $y^+$ : Chien Model with EARSM at $Re_\tau = 395$

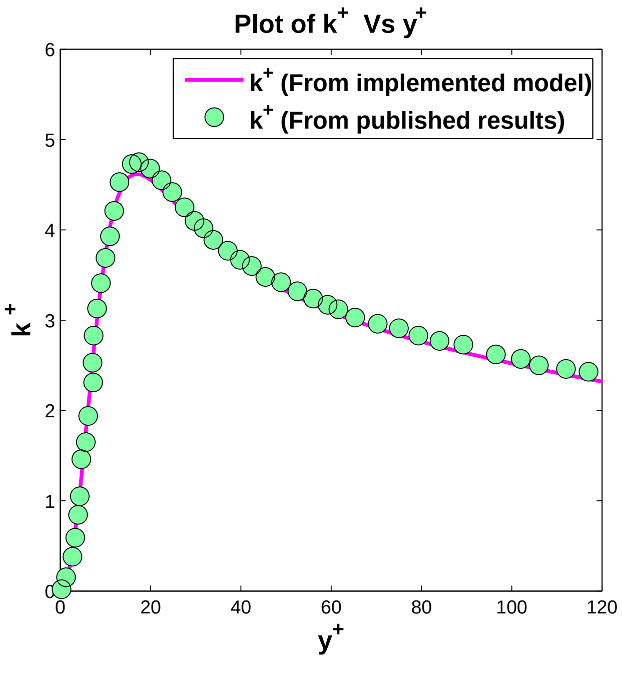


**Figure 4.2:** $k^+$ vs $y^+$ : Chien Model with EARSM at $Re_\tau = 395$

## 4.2 Menter Formulation

### 4.2.1 Theory

Menter [4] 's formulation is based on Wallin and Johansson [3]'s theory and the differences between the two are few. The ($\beta$) coefficients of the velocity strains, although fundamentally the same as in Wallin and Johansson [3]'s model have been 'expressed' differently. For the sake of completeness, all the underlying equations and constants are (re) discussed here.
Reynolds Stress Tensor is defined as:

$$\vec{\vec{\tau}} = k\left(\vec{\vec{a}} + \frac{2}{3}\vec{\vec{I}}\right)$$

where, the anisotropy tensor $\vec{\vec{a}}$ is decomposed as:

$$\vec{\vec{a}} = \beta_1\vec{\vec{T}}_1 + \beta_2\vec{\vec{T}}_2 + \beta_3\vec{\vec{T}}_3 + \beta_4\vec{\vec{T}}_4 + \beta_6\vec{\vec{T}}_6 + \beta_9\vec{\vec{T}}_9$$

Here, the velocity strains are as follows:

$$\begin{aligned}
\vec{\vec{T}}_1 &= \vec{\vec{S}} \\
\vec{\vec{T}}_2 &= \vec{\vec{S}}\vec{\vec{S}} - \frac{1}{3}II_S\vec{\vec{I}} \\
\vec{\vec{T}}_3 &= \vec{\vec{\Omega}}\vec{\vec{\Omega}} - \frac{1}{3}II_\Omega\vec{\vec{I}} \\
\vec{\vec{T}}_4 &= \vec{\vec{S}}\vec{\vec{\Omega}} - \vec{\vec{\Omega}}\vec{\vec{S}} \\
\vec{\vec{T}}_6 &= \vec{\vec{S}}\vec{\vec{\Omega}}\vec{\vec{\Omega}} + \vec{\vec{\Omega}}\vec{\vec{\Omega}}\vec{\vec{S}} - \frac{2}{3}IV\vec{\vec{I}} - II_\Omega\vec{\vec{S}} \\
\vec{\vec{T}}_9 &= \vec{\vec{\Omega}}\vec{\vec{S}}\vec{\vec{\Omega}}\vec{\vec{\Omega}} - \vec{\vec{\Omega}}\vec{\vec{\Omega}}\vec{\vec{S}}\vec{\vec{\Omega}} + \frac{1}{2}II_\Omega\left(\vec{\vec{S}}\vec{\vec{\Omega}} - \vec{\vec{\Omega}}\vec{\vec{S}}\right)
\end{aligned}$$

Here, $\vec{\vec{S}}$ and $\vec{\vec{\Omega}}$ are the non-dimensional strain and vorticity tensors and are given as follows:

$$\begin{aligned}
S_{ij} &= \frac{\tau}{2}\left(\frac{\partial U_i}{\partial x_j} + \frac{\partial U_j}{\partial x_i}\right) \\
\Omega_{ij} &= \frac{\tau}{2}\left(\frac{\partial U_i}{\partial x_j} - \frac{\partial U_j}{\partial x_i}\right)
\end{aligned}$$

Here, $\tau$ is the time scale:

$$\tau = \max\left(\frac{1}{C_\mu\omega}, 6\sqrt{\frac{\nu}{C_\mu k\omega}}\right)$$

The tensor invariants are as follows:

$$II_S = \operatorname{tr}\left(\vec{\vec{S}}\vec{\vec{S}}\right) \quad II_\Omega = \operatorname{tr}\left(\vec{\vec{\Omega}}\vec{\vec{\Omega}}\right) \quad IV = \operatorname{tr}\left(\vec{\vec{S}}\vec{\vec{\Omega}}\vec{\vec{\Omega}}\right)$$

The coefficients of the tensor basis are:

$$\beta_1 = -\frac{N}{Q}$$
$$\beta_2 = 0$$
$$\beta_3 = -\frac{2IV}{NQ_1}$$
$$\beta_4 = -\frac{N}{Q}$$
$$\beta_6 = -\frac{N}{Q_1}$$
$$\beta_9 = -\frac{1}{Q_1}$$

where,

$$Q = \frac{N^2 - 2II_\Omega}{A_1} \quad Q_1 = \frac{Q}{6}\left(2N^2 - II_\Omega\right)$$

Here, N can be solved in closed form with the solution:

$$\begin{aligned} N &= \frac{c_1'}{3} + \left(P_1 + \sqrt{P_2}\right)^{1/3} + \text{sign}\left(P_1 - \sqrt{P_2}\right)|P_1 - \sqrt{P_2}|^{1/3}, P_2 \geq 0 \\ &= \frac{c_1'}{3} + 2\left(P_1^2 - P_2\right)^{1/6} \cos\left(\frac{1}{3}\text{arccos}\left(\frac{P_1}{\sqrt{P_1^2 - P_2}}\right)\right), P_2 < 0 \end{aligned}$$

where,

$$\begin{aligned} P_1 &= \left(\frac{1}{27}c_1'^2 + \frac{9}{20}II_s - \frac{2}{3}II_\Omega\right)c_1' \\ P_2 &= P_1^2 - \left(\frac{1}{9}c_1'^2 + \frac{9}{10}II_S + \frac{2}{3}II_\Omega\right)^3 \end{aligned}$$

Here $c_1' = 1.8$. The transport equations for BSL model are mentioned as follows:

$$\frac{\mathrm{d}k}{\mathrm{d}t} = \frac{\partial}{\partial y}\left[\left(\nu + \nu_t \delta_k\right)\frac{\partial k}{\partial y}\right] + \tilde{P}_k - \beta^* k\omega \tag{4.2.1}$$

$$\frac{\mathrm{d}\omega}{\mathrm{d}t} = \frac{\partial}{\partial y}\left[\left(\nu + \nu_t \delta_\omega\right)\frac{\partial \omega}{\partial y}\right] + \frac{\gamma\omega}{k}\tilde{P}_k - \beta\omega^2 + \frac{\delta_d}{\omega}\left(\nabla k\right)\left(\nabla \omega\right) \tag{4.2.2}$$

Here, $\tilde{P}_k$ is given by:

$$\tilde{P}_k = \min\left(-\tau_{ij}\frac{\partial U_i}{\partial x_j}, 10\rho\beta^* k\omega\right) \tag{4.2.3}$$

The rest of the theory involves the fundamental concepts of the baseline model.

$$\begin{aligned}\delta_k &= F_1 0.5 + (1 - F_1)\,1.0 \\ \delta_\omega &= F_1 0.5 + (1 - F_1)\,0.856 \\ \beta &= F_1 0.075 + (1 - F_1)\,0.0828 \\ \delta_d &= 2\,(1 - F_1)\,(0.856) \\ \gamma &= \frac{\beta}{0.09} - \frac{\delta_\omega k^2}{0.3}\end{aligned}$$

where $F_1$ reads as:

$$\begin{aligned}F_1 &= \tanh\left(\arg_1^4\right) \\ \arg_1 &= \min\left[\max\left(\frac{\sqrt{k}}{0.09\omega y}, \frac{500\nu}{\omega y^2}\right), \frac{2k\omega}{y^2\,(\nabla k)\,(\nabla\omega)}\right]\end{aligned}$$

### 4.2.2 Implementation (k $\omega$ BSL)

There is one difference between Wallin and Johansson [3] 's model and Menter [4] model. A slight recalibration of the $A_1$ constant was done and it was set at 1.245 instead of 1.2, like in Wallin and Johansson [3] 's model.

The boundary condition for $\omega$ at the first cell centre next to the wall was kept as:

$$\omega_w = \frac{6\nu}{(0.075)\,y^2}$$

The complete 0/omega file is specified as follows:

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

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

internalField   uniform 400;

boundaryField
{
  inlet
    {
      type            zeroGradient;
    }

  outlet
    {
      type            zeroGradient;
    }

  fixedWalls
    {
      type            omegaWF;
      value           uniform 400;
    }

  frontAndBack
    {
      type            empty;
    }
  middle
    {
      type           symmetryPlane;
    }


}


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

# 5 $\phi$-$f$ Model

## 5.1 Theory

The v2-f model was created by Durbin [5]. It expresses turbulent viscosity as a function of wall normal stress $\left(\overline{v'v'}\right)$ and hence does away with the need of damping functions to model near wall turbulence. (Damping functions represent the kinematic blocking by walls. Instead, using wall normal stress to calculate turbulent viscosity indicates the presence of the wall, while also doing away with non linear damping functions).

The v2f model that has been implemented in OpenFOAM 2.2x is based upon Davidson [6] and Lien-Kalitzin [7]'s work.

The $\phi$ -f model, in principle, is the same as the v2f model. But casting $\phi = \frac{\overline{v'^2}}{k}$ makes it more numerically stable. There are two forms of the $\phi$ f model, one implemented by Laurence et al. [8] (University of Manchester) and the other, implemented by Hanjalic et al. [9] (Delft University of Technology). The former has been chosen for implementation as it is more similar to the original v2f than the latter's model.

The transport equations of interest are as follows:

$$\frac{\mathrm{d}k}{\mathrm{d}t} = P_k - \epsilon + \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\nu_t}{\sigma_k}\right)\frac{\partial k}{\partial x_j}\right] \tag{5.1.1}$$

$$\frac{\mathrm{d}\epsilon}{\mathrm{d}t} = \frac{C'_{\epsilon 1}P_\kappa - C_{\epsilon 2}\epsilon}{T} + \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\nu_t}{\sigma_\epsilon}\right)\frac{\partial \epsilon}{\partial x_j}\right] \tag{5.1.2}$$

where

$$P_k = 2\nu_t S^2 \quad S = \sqrt{2\left(S_{ij}S_{ij}\right)} \quad \sigma_k = 1.0 \quad \sigma_\epsilon = 1.3 \quad C_{\epsilon 1} = 1.4\left(1.0 + 0.05\sqrt{\frac{1.0}{\phi}}\right) \quad C_{\epsilon 2} = 1.85$$

The transport equations for $\phi$ and $f_0$ are as follows:

$$\frac{\mathrm{d}\phi}{\mathrm{d}t} = f - \frac{\phi}{k}P_k + \frac{\partial}{\partial x_j}\left[\left(\frac{\nu_t}{\sigma_k}\right)\frac{\partial \phi}{\partial x_j}\right] + \frac{2}{k}\frac{\nu_t}{\sigma_k}\frac{\partial \phi}{\partial x_j}\frac{\partial k}{\partial x_j} \tag{5.1.3}$$

$$L^2\frac{\partial^2 f}{\partial x_i^2} - f = \frac{1}{T}\left(C_1 - 1\right)\left[\phi - \frac{2}{3}\right] - C_2\frac{P_k}{k} - 2\frac{\nu}{k}\frac{\partial \phi}{\partial x_j}\frac{\partial k}{\partial x_j} - \nu\nabla^2\phi \tag{5.1.4}$$

where:

$$C_1 = 1.4 \quad C_2 = 0.3$$

And:

$$T = \max\left[\frac{k}{\epsilon}, 6.0\left(\frac{\nu}{\epsilon}\right)^{\frac{1}{2}}\right] \tag{5.1.5}$$

$$L = C_L \max\left[\frac{k^{3/2}}{\epsilon}, 110\left(\frac{\nu^3}{\epsilon}\right)^{\frac{1}{4}}\right] \tag{5.1.6}$$

where $C_L$= 0.25.

## 5.2 Implementation

The original implementation has been modified by adding the Davidson [6] correction factors (Modification 1). The modifications are enumerated as follows:

1. $\nu_t = \min\left[0.09k^2/\epsilon, 0.22\phi kT\right]$
2. $\phi_{source} = \min\left[f, -\frac{1}{T}\left(C_1 - 1\right)\left[\phi - \frac{2}{3}\right] + C_2\frac{P_k}{k}\right]$

The boundary condition for $\epsilon$ at the first cell centre next to the wall is calculated as:

$$\epsilon_w = \frac{2\nu k}{y^2} \tag{5.2.1}$$

The complete 0/epsilon file is specified as follows:

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

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

internalField   uniform 0.000765;

boundaryField
{
  inlet
    {
      type            zeroGradient;
    }

  outlet
    {
      type            zeroGradient;
    }

  fixedWalls
    {
      type            epsilonLRWF;

    }

  frontAndBack
    {
      type            empty;
    }

  middle
   {
     type            symmetryPlane;
   }


}


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

The boundary condition for phi and f at the wall is of type 'fixed Value' with value SMALL (1e-10). The files('0/zeta' for phi and '0/f' for f) are specified as follows:

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

dimensions      [0 0 0 0 0 0 0];

internalField   uniform 0.325;

```

```
boundaryField
{
    inlet
      {
        type            zeroGradient;
      }

    outlet
      {
        type            zeroGradient;
      }

    fixedWalls
      {
        type            fixedValue;
        value           uniform 1e-10;
      }

    frontAndBack
      {
        type            empty;
      }

 middle
   {
     type            symmetryPlane;
   }


}


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

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

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

internalField   uniform 0.000765;

boundaryField
{
  inlet
    {
      type            zeroGradient;
    }

  outlet
    {
      type            zeroGradient;
    }

  fixedWalls
    {
      type            fixedValue;
      value           uniform 1e-10;
    }

  frontAndBack
    {
      type            empty;
    }

  middle
   {
     type            symmetryPlane;
   }


}


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

## 5.3 Verification

The data below for the published results in the Figures (5.1), (5.2) and (5.3) are taken from Laurence [8] 's publication and is added for the sake of verification. The data matches closely to the results obtained with the implemented model.

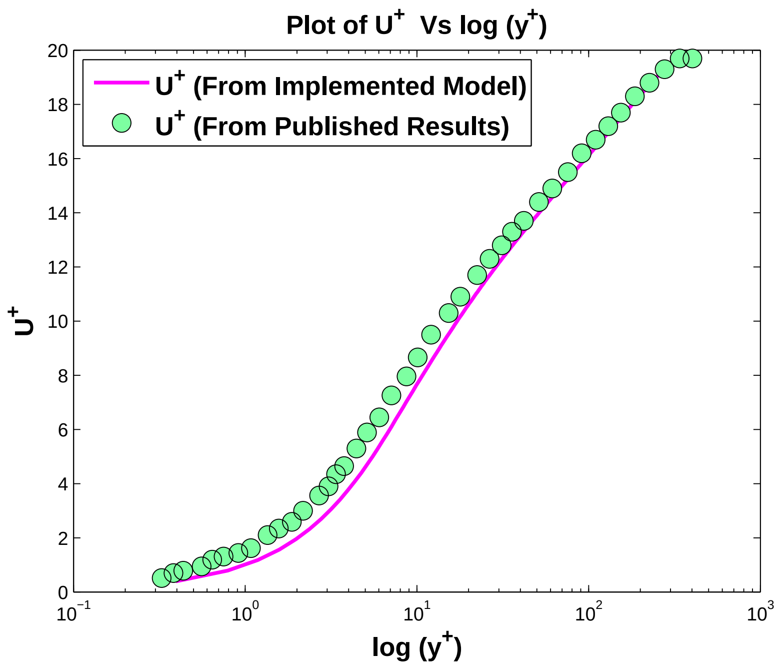


**Figure 5.1:** $U^+$ vs $y^+$ for $\phi$ -f at $Re_\tau = 395$

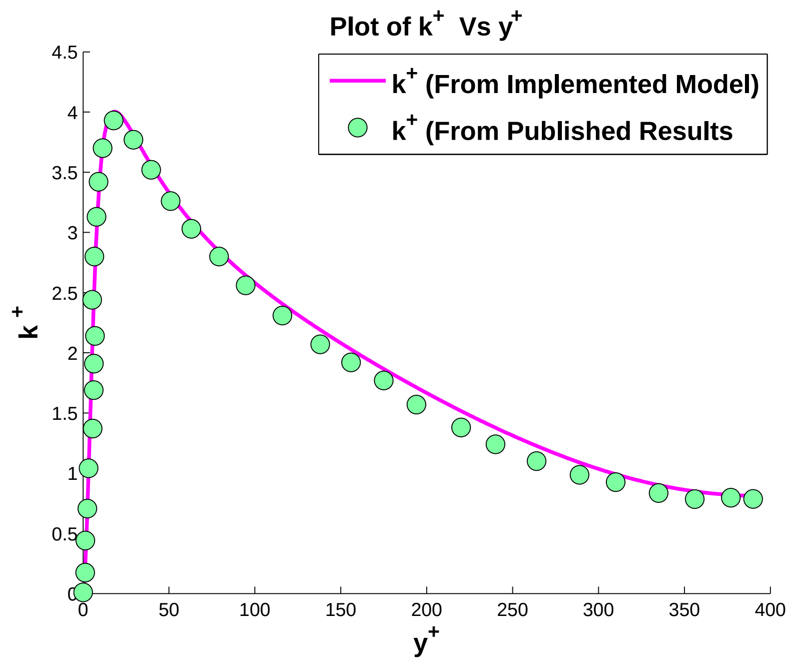


**Figure 5.2:** $\kappa^+$ vs $y^+$ for $\phi$ -f at $Re_\tau = 395$

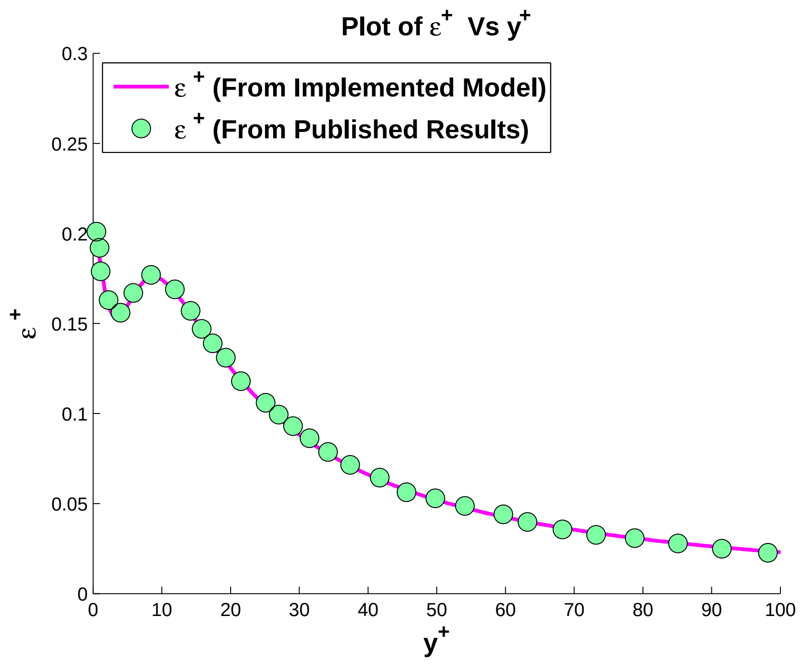


**Figure 5.3:** $\epsilon^+$ vs $y^+$ for $\phi$ -f at $Re_\tau = 395$

# 6 Validation

## 6.1 Fidelity

The figures below ((6.1), (6.2) and (6.3)) demonstrate the performance of the implemented models vis-a-vis DNS data for turbulent parameters like channel velocity ‘U’, turbulent kinetic energy ‘k’ and turbulent dissipation ‘$\epsilon$’ . The curves depicted correspond to the following cases, for channel flow at $Re_\tau = 395$:

**1** DNS data

**2** Chien Low Reynolds k $\epsilon$ Model

**3** EARSM + Chien Low Reynolds k $\epsilon$ Model (Wallin-Johansson’s Formulation)

**4** $\phi$-f Model

**5** v2-f Model

**6** k $\omega$ SST Model

**7** EARSM + k $\omega$ BSL Model (Menter’s Formulation)

For the evaluation of $\epsilon$ from $\omega$, the following formulation was used for $\omega$ based models.

$$\epsilon = C_\mu k \omega$$

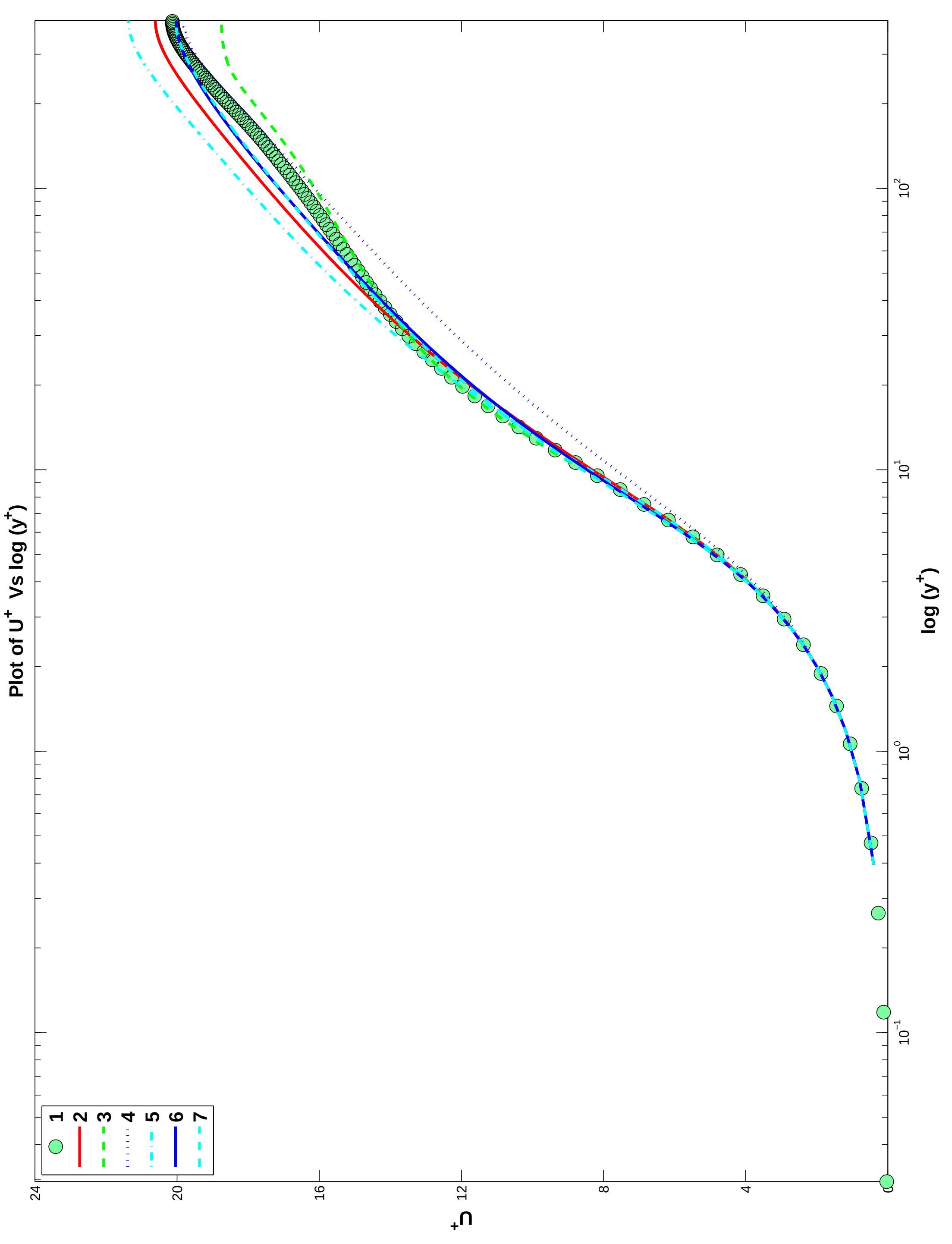


**Figure 6.1:** $U^+$ vs $y^+$ at $Re_\tau = 395$ [1 - DNS, 2 - Chien k $\epsilon$, 3- EARSM + Chien k $\epsilon$, 4- $\phi$-f, 5- v2-f, 6- k $\omega$ SST, 7-EARSM + k $\omega$ BSL]

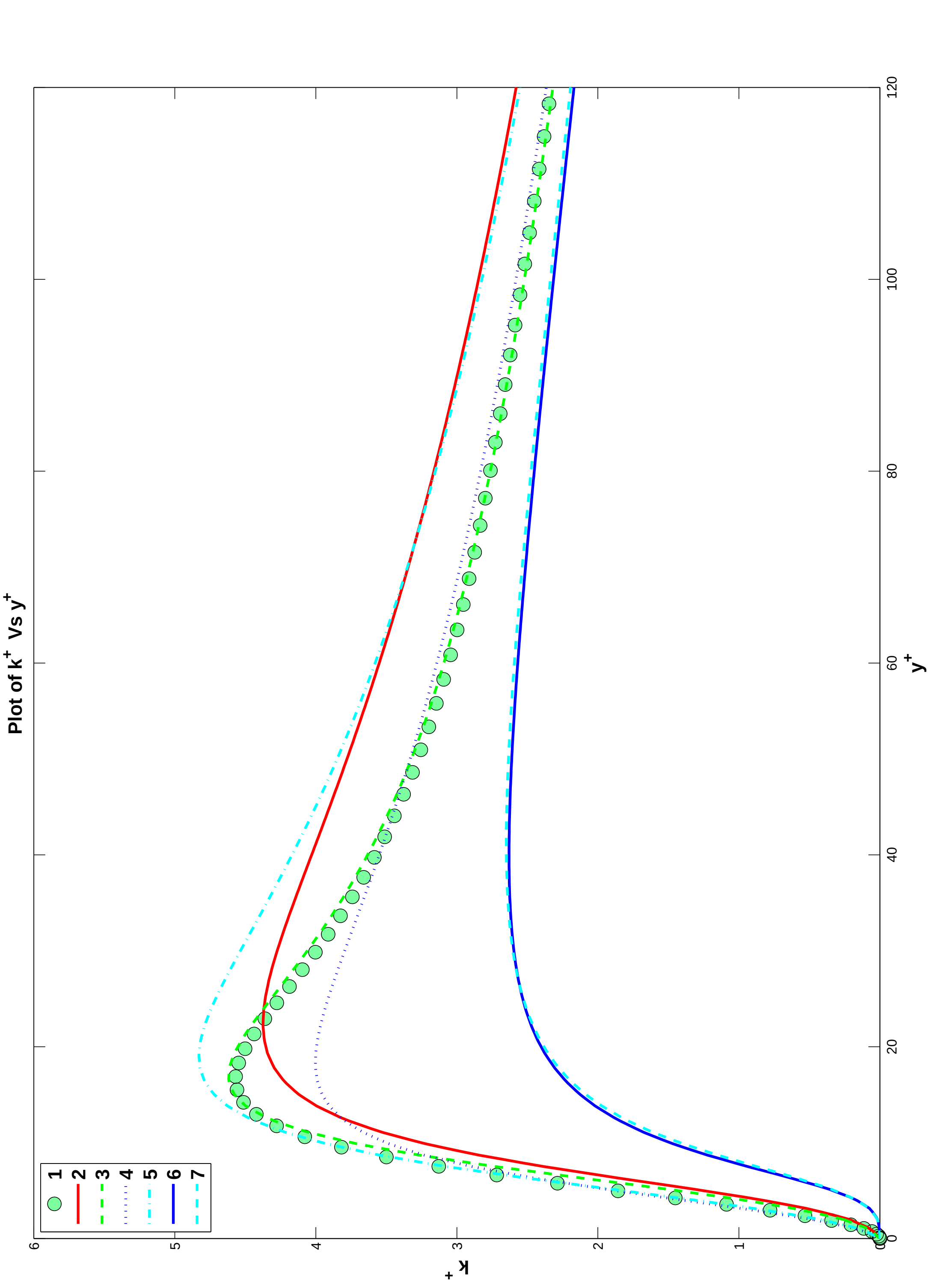


**Figure 6.2:** $\kappa^+$ vs $y^+$ at $Re_\tau = 395$ [1 - DNS, 2 - Chien k $\epsilon$, 3- EARSM + Chien k $\epsilon$, 4- $\phi$-f, 5- v2-f, 6- k $\omega$ SST, 7-EARSM + k $\omega$ BSL]

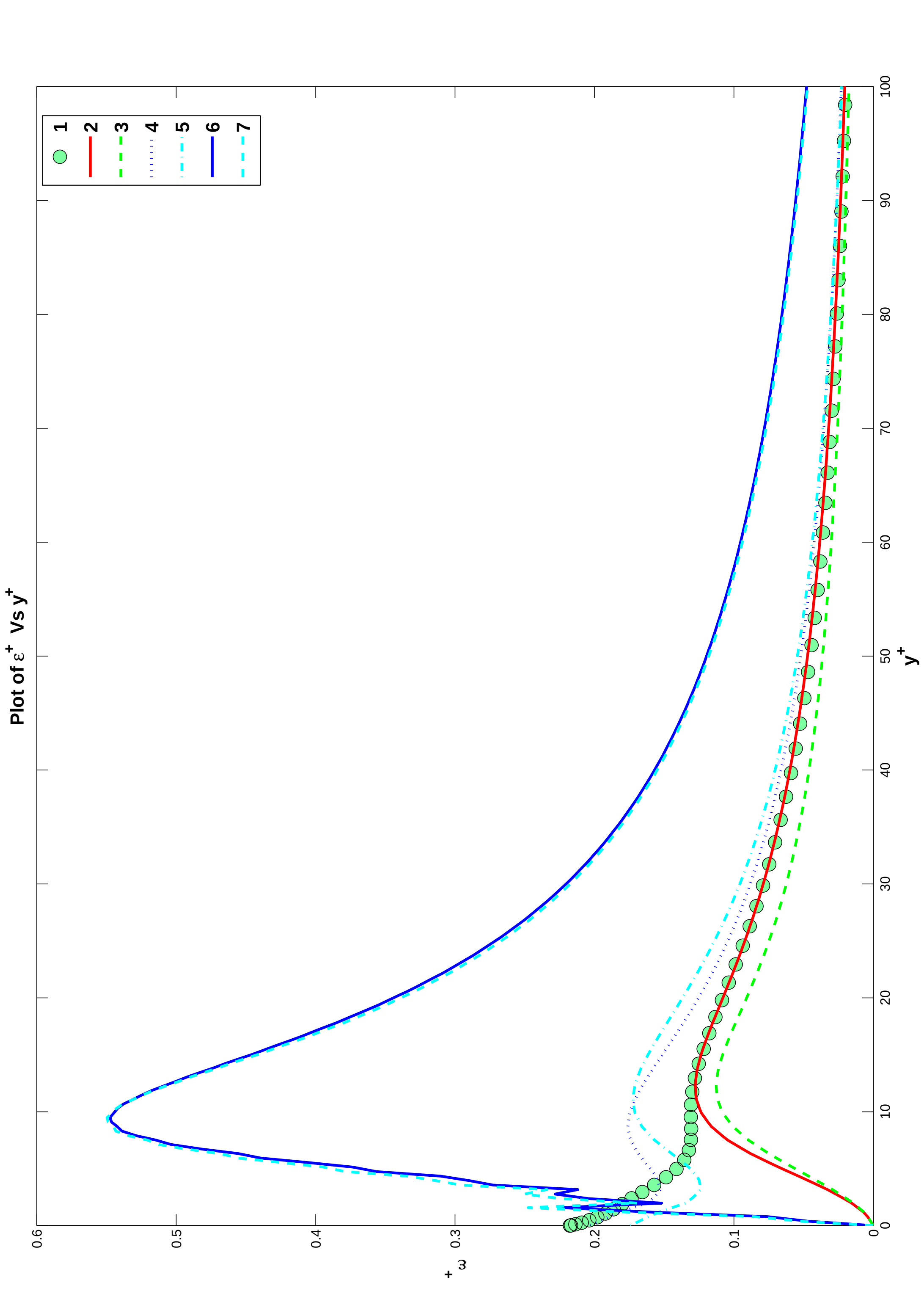


**Figure 6.3:** $\epsilon^+$ vs $y^+$ at $Re_\tau = 395$ [1 - DNS, 2 - Chien k $\epsilon$, 3- EARSM + Chien k $\epsilon$, 4- $\phi$-f, 5- v2-f, 6- k $\omega$ SST, 7-EARSM + k $\omega$ BSL]

Looking at the figures above, one can conclude the following:

| *Ranking for fidelity* | | | |
|---|---|---|---|
| | **U** | **k** | $\epsilon$ |
| 1 | k $\omega$ SST<br>EARSM + k $\omega$ BSL<br>$\phi$-f | EARSM + Chien k $\epsilon$ | Chien k $\epsilon$<br>EARSM + Chien k $\epsilon$ |
| 2 | Chien k $\epsilon$<br>EARSM + Chien k $\epsilon$ | $\phi$-f<br>Chien k $\epsilon$<br>v2-f | $\phi$-f<br>v2-f |
| 3 | v2-f | k $\omega$ SST<br>EARSM + k $\omega$ BSL | k $\omega$ SST<br>EARSM + k $\omega$ BSL |

The idea is to place the models in three broad rank fields, 1 being the most faithful and 3 being the least faithful to the DNS data. It is observed that for channel velocity, both the $\omega$ based models and $\phi$-f capture the viscous and the logarithmic flow fields quite accurately and hence are collectively assigned a rank of 1. While Chien Low Reynolds k $\epsilon$ slightly overpredicts the mid channel velocity, the EARSM attachment slightly underpredicts the same. The EARSM formulation is quite accurate until $Re_{\tau} = 170$, from where it starts to flatten out away from the DNS curve. It was felt that v2-f performs the least bit accurately compared to the other models and hence, was assigned a rank of 3 with respect to the velocity prediction.

With respect to turbulent kinetic energy, the EARSM attachment to Chien Low Reynolds k $\epsilon$ is a clear winner. The original Chien Low Reynolds k $\epsilon$ and $\phi$-f also perform satisfactorily well and are hence assigned a rank of 2. The $\omega$ based formulations perform the least satisfactorily for k, with respect to DNS data, and hence are assigned a rank 3.

Finally for turbulent dissipation, both the Low Reynolds formulations perform the best, followed by the v2-f family of models and lastly the $\omega$ family of models.

From the results' evaluation, it is a little hard to make a clear assessment of the best model. While the $\omega$ based formulations are quite accurate in predicting the channel velocity, they are not so faithful for turbulent kinetic energy and dissipation. However, in many scenarios, the channel velocity ranks as a more important parameter to predict than the other two, and hence the importance of $\omega$ based models can not be shunted away.

Also, some observations are quite evident:

- $\phi$-f model, which has been developed using v2-f as a substrate, performs better than v2-f.
- EARSM attachments to Chien Low Reynolds based model improves its performance, especially with regard to its prediction for turbulent kinetic energy. This justifies spending more computational resources for attaining higher fidelity.

## 6.2 Robustness

The robustness of the models was tested by looking at their mesh sensitivity. Two simulations with 100 ($y^+ = 0.50$) and 200 ($y^+ = 0.25$) cells in the Y (wall normal) direction respectively were evaluated. Performance with regard to the mean velocity was used as the judging criterion.

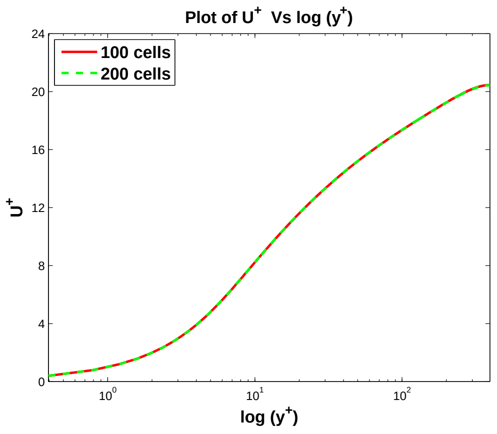


**Figure 6.4:** $U^+$ vs $y^+$ for Chien Low Reynolds k $\epsilon$

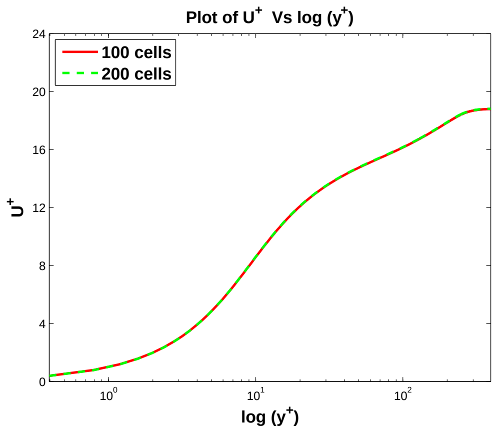


**Figure 6.5:** $U^+$ vs $y^+$ for EARSM + Chien Low Reynolds k $\epsilon$

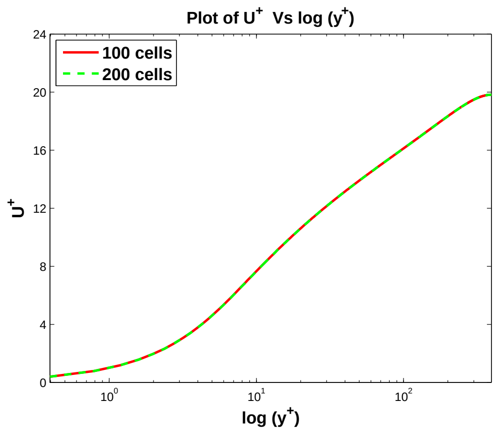


**Figure 6.6:** $U^+$ vs $y^+$ for $\phi$-f

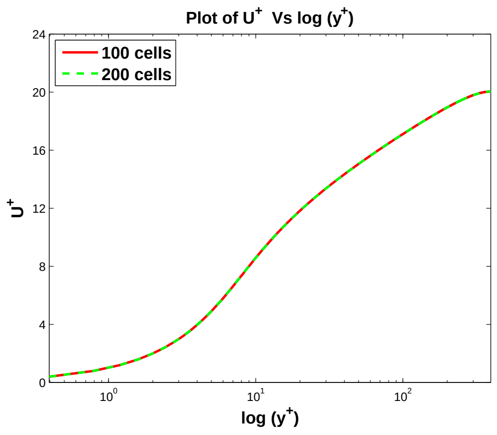


**Figure 6.7:** $U^+$ vs $y^+$ for EARSM + k $\omega$ BSL

It is concluded that the models are quite robust with respect to sensitivity to the mesh, and further mesh refinement does not lead to improvement in results.

## 6.3 Cost

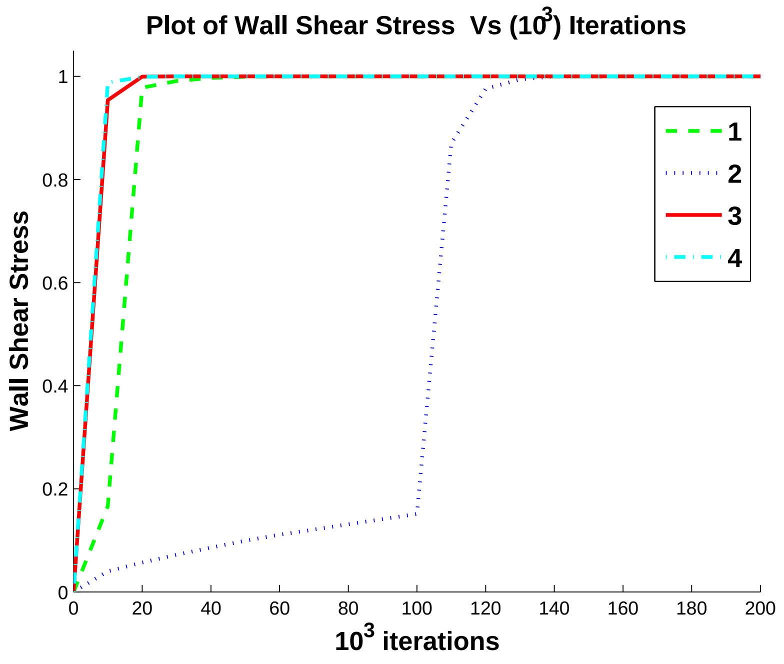


**Figure 6.8:** Wall Shear Stress vs ($10^3$) iterations at $Re_\tau = 395$ [1 -Chien k $\epsilon$, 2- EARSM + Chien k $\epsilon$, 3- $\phi$-f, 4- EARSM + k $\omega$ BSL]

The cost of a model has been judged based on how many iterations it takes to attain a wall shear stress ($\tau_w$) of 1.0 given the same mesh (100 cells in the Y direction) and the same initial conditions for the common transport quantities, as was discussed in section 3.2.

As is evident from the figure (6.8), EARSM + k $\omega$ BSL and $\phi$-f models takes the least time to reach a wall shear stress of 1.0, closely followed by the Chien k $\epsilon$ model and lastly the EARSM - Chien k $\epsilon$ model.

| | *Ranking for cost* |
|---|---|
| 1 | EARSM + k $\omega$ BSL |
| | $\phi$-f |
| 2 | Chien k $\epsilon$ |
| 3 | EARSM - Chien k $\epsilon$ |

# 7 Conclusions

In this thesis project, $\epsilon$ based models like Low Reynolds Group ('Chien', 'Launder Sharma'); $\phi$ - f and $\omega$ based models like k-$\omega$ SST, k-$\omega$ BSL were considered. In addition, the EARSM formulation was attached to some models to see if there was an improvement in results.

Although the results with standard k-$\epsilon$ and Launder Sharma k-$\epsilon$ have not been printed in this report, they were thoroughly investigated. Amongst the Low Reynolds linear eddy viscosity $\epsilon$ based models, Chien 's Low Reynolds formulation is a clear winner. All the turbulent quantities are predicted very closely to the DNS data.

The next model with several advantages is $\phi$-f. Its major advantage is that the turbulent viscosity is expressed in terms of the wall normal stress thus including the effect of the wall damping and eliminating the need of another damping function. Also this model automatically works well for Low or High Reynolds flow which gives it a powerful edge to Chien's formulation which can be used only for Low Reynolds flows. Although the number of transport equations have increased by two for this model ($\phi$ and f transport equations), the model offers performs very well across the measures of fidelity, robustness and accuracy which justifies its use.

The EARSM implementation with Chien's model achieves a close fit with turbulent quantities as well. It leads to slightly improved accuracy for channel flows, but has higher cost as well. Logic would deem that for impinging flows with streamline curvature and stagnation points, an EARSM implementation would be useful. Hence further comparison of the original Chien Model and Chien + EARSM should be done for more complex geometries.

Results with $\omega$ - based models have also been carried out. k-$\omega$ SST gave a good performance and an EARSM formulation with k-$\omega$ BSL was tested to see for any improvement in results. As such, the k-$\omega$ SST model, for its applicability to all flows and its improved accuracy over Standard Wilcox k-$\omega$, is also recommended for capturing channel flows, by the author. Not too much difference between k-$\omega$ SST and the EARSM implementation was noticed, and for that purpose a verification of the k-$\omega$ BSL + EARSM model needs to be looked into. It was not included in this report because two dimensional developed channel flow results were not included by Menter [4] in his publication.

# References

[1] R. D. Moser, J. Kim, and N. N. Mansour, “Direct numerical simulations of turbulent channel flow up to Re tau = 590,” *Physics Of Fluids*, vol. 11, pp. 943–945, 1999.

[2] K.-Y. Chien, “Predictions of Channel and Boundary-Layer Flows with a Low-Reynolds-Number Turbulence Model,” *AIAA Journal*, vol. 20, pp. 33–38, 1982.

[3] S. Wallin and A. V. Johansson, “An explicit algebraic Reynolds stress model for incompressible and compressible turbulent flows,” *J.Fluid Mech*, vol. 403, pp. 89–132, 2000.

[4] F. R. Menter, A. V. Garbaruk, and Y. Egorov, “Explicit Algebraic Reynolds Stress Models For Anisotropic Wall-Bounded Flows,” *Progress in Flight Physics*, vol. 3, pp. 89–104, 2012.

[5] P. A. Durbin, “Separated flow computations with the k-epsilon-v-squared model,” *AIAA Journal*, vol. 33, 1995.

[6] L. Davidson, P. V. Nielsen, and A. Sveningsson, “Modifications of the v2-f Model for Computing the Flow in a 3D Wall Jet,” *Turbulence, Heat and Mass Transfer*, vol. 4, pp. 577–584, 2003.

[7] F.-S. Lien and G. Kalitzin, “Computations of transonic flow with the v2-f turbulence model,” *International Journal of Heat and Fluid Flow*, vol. 22, pp. 53–61, 2001.

[8] D. R. Laurence, J. C. Uribe, and S. V. Utyuzhnikov, “A Robust Formulation of the v2-f Model,” *Flow, Turbulence and Combustion*, vol. 73, pp. 169–185, 2004.

[9] K. Hanjalic, M. Popovac, and M. Hadziabdic, “A robust near-wall elliptic-relaxation eddy-viscosity turbulence model for CFD,” *International Journal of Heat and Fluid Flow*, vol. 25, pp. 1047–1051, 2004.

# 8 Appendix

## 8.1 Chien Low Reynolds kEpsilon Model

Following are the OpenFOAM generated source files (Chien.C) for the discussed Chien model.

```
**************
***CHIEN.C****
**************

#include "Chien.H"
#include "addToRunTimeSelectionTable.H"
#include "wallDist.H"
#include "backwardsCompatibilityWallFunctions.H"

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

namespace Foam
{
namespace incompressible
{
namespace RASModels
{

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

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

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

tmp<volScalarField> Chien::fMu() const
{
   volScalarField y=wallDist(mesh_).y();
   volScalarField Rey =pow(k_,0.5)*(y/nu());
   volScalarField yStar=pow(Rey,2)*0.003 + pow(Rey, 0.5)*2.4;
   return
   scalar(1)
   - exp(-0.0115*yStar);
}
```

```
tmp<volScalarField> Chien::f2() const
{
    return
    scalar(1)
    - 0.22*exp(-sqr(sqr(k_)/(nu()*epsilonTilda_*6)));
}


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

Chien::Chien
(
    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
        )
    ),
    C1_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "C1",
            coeffDict_,
            1.35
        )
    ),
    C2_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "C2",
            coeffDict_,
            1.8
        )
    ),
    sigmaEps_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "sigmaEps",
            coeffDict_,
            1.3
        )
    ),

    k_
    (
        IOobject
        (
            "k",
            runTime_.timeName(),
            mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_
    ),

    epsilonTilda_
    (
        IOobject
        (
            "epsilon",
            runTime_.timeName(),
            mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_
    ),

    nut_
```

```
    (
        IOobject
        (
            "nut",
            runTime_.timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        autoCreateLowReNut("nut", mesh_)
    )
{
    bound(k_, kMin_);
    bound(epsilonTilda_, epsilonMin_);

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

    printCoeffs();
}


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

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


tmp<fvVectorMatrix> Chien::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 Chien::read()
{
    if (RASModel::read())
    {
        Cmu_.readIfPresent(coeffDict());
        C1_.readIfPresent(coeffDict());
        C2_.readIfPresent(coeffDict());
        sigmaEps_.readIfPresent(coeffDict());

        return true;
    }
    else
    {
        return false;
    }
}


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

    if (!turbulence_)
    {
        return;
    }

    tmp<volScalarField> S2 = 2*magSqr(symm(fvc::grad(U_)));
    volScalarField G(GName(), nut_*S2);
    volScalarField y=wallDist(mesh_).y();
    volScalarField Rey= pow(k_,0.5)*(y/nu());
    volScalarField yStar=pow(Rey,2)*0.003 + pow(Rey, 0.5)*2.4;
    const volScalarField E(-2.0*nu()*epsilonTilda_/pow(y,2)
    *exp(-0.5*yStar));
    const volScalarField D(2.0*nu()*k_/pow(y,2));


    // Dissipation rate equation
    tmp<fvScalarMatrix> epsEqn
    (
        fvm::ddt(epsilonTilda_)
      + fvm::div(phi_, epsilonTilda_)
      - fvm::laplacian(DepsilonEff(), epsilonTilda_)
     ==
        C1_*G*epsilonTilda_/k_
      - fvm::Sp(C2_*f2()*epsilonTilda_/k_, epsilonTilda_)
      + E
    );

    epsEqn().relax();
    solve(epsEqn);
    bound(epsilonTilda_, epsilonMin_);


    // Turbulent kinetic energy equation

    tmp<fvScalarMatrix> kEqn
    (
        fvm::ddt(k_)
      + fvm::div(phi_, k_)
      - fvm::laplacian(DkEff(), k_)
     ==
        G - fvm::Sp((epsilonTilda_ + D)/k_, k_)
    );

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


    // Re-calculate viscosity
    nut_ == Cmu_*fMu()*sqr(k_)/epsilonTilda_;
}


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

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

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

## 8.2 EARSM + Chien Low Reynolds kEpsilon Model

Following are the OpenFOAM generated source files (EARSM_Chien.C) for the discussed EARSM model.

```
*******************
***EARSM_CHIEN.C****
*******************


#include "EARSM_Chien.H"
#include "addToRunTimeSelectionTable.H"
#include "wallDist.H"
#include "backwardsCompatibilityWallFunctions.H"

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

namespace Foam
{
namespace incompressible
{
namespace RASModels
{

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

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

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

tmp<volScalarField> EARSM_Chien::fMu() const
{
   volScalarField y=wallDist(mesh_).y();
   volScalarField Rey = pow(k_,0.5)*(y/nu());
   volScalarField yStar=pow(Rey,2)*0.003 + pow(Rey, 0.5)*2.4;
   return
   scalar(1)
   - exp(-0.0115*yStar);
}
```

```
tmp<volScalarField> EARSM_Chien::f2() const
{
    return
        scalar(1)
      - 0.22*exp(-sqr(sqr(k_)/(nu()*epsilonTilda_*6)));
}


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

EARSM_Chien::EARSM_Chien
(
    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
        )
    ),
    C1_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "C1",
            coeffDict_,
            1.35
        )
    ),
    C2_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "C2",
            coeffDict_,
            1.8
        )
    ),
    sigmaEps_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "sigmaEps",
            coeffDict_,
            1.3
        )
    ),

    k_
    (
        IOobject
        (
            "k",
            runTime_.timeName(),
            mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_
    ),

    epsilonTilda_
    (
        IOobject
        (
            "epsilon",
            runTime_.timeName(),
            mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_
    ),
    R_
    (
        IOobject
        (
            "R",
            runTime_.timeName(),
            mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        autoCreateR("R", mesh_)
    ),
    nut_
    (
        IOobject
        (
            "nut",
            runTime_.timeName(),
            mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        autoCreateLowReNut("nut", mesh_)
    )
{
    bound(k_, kMin_);
    bound(epsilonTilda_, epsilonMin_);
    nut_ = Cmu_*fMu()*sqr(k_)/epsilonTilda_;
    nut_.correctBoundaryConditions();
    printCoeffs();
}


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

tmp<volSymmTensorField> EARSM_Chien::R() const
{
    return tmp<volSymmTensorField>
    (
        new volSymmTensorField
        (
            IOobject
            (
                "R",
                runTime_.timeName(),
                mesh_,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            R_,
            k_.boundaryField().types()
        )
    );
}


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

```
tmp<fvVectorMatrix> EARSM_Chien::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 EARSM_Chien::read()
{
    if (RASModel::read())
    {
        Cmu_.readIfPresent(coeffDict());
        C1_.readIfPresent(coeffDict());
        C2_.readIfPresent(coeffDict());
        sigmaEps_.readIfPresent(coeffDict());

        return true;
    }
    else
    {
        return false;
    }
}


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

    if (!turbulence_)
    {
        return;
    }
```

```
volScalarField tau1_ = k_/epsilonTilda_;
volScalarField tau2_ = 6.0*sqrt(nu()/epsilonTilda_);
volScalarField tau_ = max(tau1_, tau2_);

volTensorField S = 0.5*tau_*(T(fvc::grad(U_)) + fvc::grad(U_));
volTensorField W = 0.5*tau_*(T(fvc::grad(U_)) - fvc::grad(U_));


volScalarField     IIs = tr(S & S) ;
volScalarField     IIw = tr(W & W);

volScalarField P1 =
(1.8*(sqr(1.8)/27.0 + (9.0/20.0)*IIs - (2.0/3.0)*IIw));
volScalarField P2 =
sqr(P1) - pow((sqr(1.8)/9.0 +
(9.0/10.0)*IIs + (2.0/3.0)*IIw), 3);

// N is initialized first and then recalculated later.
volScalarField N_ =  1.8+9/4*pow(2*0.09*IIs,0.5);
forAll(P2, cellI)
{
if (P2[cellI] < 0.0)
{
N_[cellI] = 1.8/3.0 + 2.0*pow(pow(P1[cellI],2.0)-P2[cellI],1.0/6.0)*
cos(1.0/3.0*acos(P1[cellI]/(pow(pow(P1[cellI],2.0)-P2[cellI],0.5))));
}
else
{
N_[cellI] = 1.8/3.0 + pow(P1[cellI]+pow(P2[cellI],0.5),1.0/3.0) +
sign(P1[cellI]-pow(P2[cellI],0.5))
*pow(mag(P1[cellI]- pow(P2[cellI],0.5)), 1.0/3.0);
}
}

volScalarField beta_1=-1.2*(N_/(sqr(N_)-2*IIw));
volScalarField beta_4=-1.2*(1/(sqr(N_)-2*IIw));
volTensorField WW = W & W;
volTensorField SS = S & S;
volTensorField SW = S & W;
volTensorField WS = W & S;
volTensorField SWW = SW & W;
volTensorField WWS = WW & S;
volTensorField WSWW = (WS & W) & W;
volTensorField WWSW = (WW & S) & W;

volScalarField y=wallDist(mesh_).y();
volScalarField Re_t = pow(k_,0.5)*(y/nu());
volScalarField yStar = 2.4*pow(Re_t,0.5) + 0.003*pow(Re_t,2.0);
volScalarField f1= 1.0 - exp(-yStar/26.0);
volScalarField beta_2_lowre=(3.0*1.8-4.0)/(max(IIs,5.74))*
                            (1-pow(f1,2));
volScalarField beta_4_lowre=pow(f1,2)*beta_4 -
1.8/(2*max(IIs,5.74))*(1-pow(f1,2));

volSymmTensorField aij_= symm(f1*beta_1*S +
beta_2_lowre*(SS - (1.0/3.0)*IIs * I) +
beta_4_lowre*(SW-WS));
R_= k_*(aij_ + (2.0/3.0)*I);
volScalarField G(GName(), -R_ && T(fvc::grad(U_)));

volScalarField dUdy=mag(fvc::grad(U_));
dimensionedScalar tauw("tauw",
dimensionSet(0,0,-1,0,0,0,0), scalar(dUdy.boundaryField()[2][0]));
volScalarField yPlus=pow(tauw*(nu()),0.5)*y/nu();

const volScalarField E(-2.0*nu()*epsilonTilda_/pow(y,2)*
exp(-0.5*yStar)*exp(-0.04*yPlus));
const volScalarField D(2.0*nu()*k_/pow(y,2)*exp(-0.04*yPlus));

// Dissipation rate equation

tmp<fvScalarMatrix> epsEqn
(
     fvm::ddt(epsilonTilda_)
   + fvm::div(phi_, epsilonTilda_)
   - fvm::laplacian(DepsilonEff(), epsilonTilda_)
  ==
     C1_*G*epsilonTilda_/k_
   - fvm::Sp(C2_*f2()*epsilonTilda_/k_, epsilonTilda_)
   + E
 );

 epsEqn().relax();
 solve(epsEqn);
 bound(epsilonTilda_, epsilonMin_);
```

```
    // Turbulent kinetic energy equation

    tmp<fvScalarMatrix> kEqn
    (
        fvm::ddt(k_)
      + fvm::div(phi_, k_)
      - fvm::laplacian(DkEff(), k_)
     ==
        G - fvm::Sp((epsilonTilda_ + D)/k_, k_)
    );

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


    // Re-calculate viscosity

    nut_ == -f1*beta_1/2.0*k_*tau_;
}


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

} // End namespace RASModels_
} // End namespace incompressible
} // End namespace Foam

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

## 8.3 EARSM + k $\omega$ BSL Model

Following is the source file for the EARSM + k$\omega$ BSL model.

```
#include "EARSM_kwBSL.H"
#include "addToRunTimeSelectionTable.H"

#include "backwardsCompatibilityWallFunctions.H"
```

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

namespace Foam
{
namespace incompressible
{
namespace RASModels
{

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

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

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

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


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

    return tanh(sqr(arg2));
}


tmp<volScalarField> EARSM_kwBSL::F3() const
{
    tmp<volScalarField> arg3 = min
    (
        150*nu()/(omega_*sqr(y_)),
        scalar(10)
    );

    return 1 - tanh(pow4(arg3));
}


tmp<volScalarField> EARSM_kwBSL::F23() const
{
    tmp<volScalarField> f23(F2());

    if (F3_)
    {
        f23() *= F3();
    }

    return f23;
}


// * * * * *  * * * * * Constructors  * * * * * * * * * * * * * * //
```

```
EARSM_kwBSL::EARSM_kwBSL
(
    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_,
            0.5
        )
    ),
    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_,
            0.5532
        )
    ),
    gamma2_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "gamma2",
            coeffDict_,
            0.4403
        )
    ),
    beta1_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "beta1",
            coeffDict_,
            0.075
        )
    ),
    beta2_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "beta2",
            coeffDict_,
            0.0828
        )
    ),
    betaStar_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "betaStar",
            coeffDict_,
            0.09
        )
    ),
    a1_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "a1",
            coeffDict_,
            0.31
        )
    ),
    b1_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "b1",
            coeffDict_,
            1.0
        )
    ),
    c1_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "c1",
            coeffDict_,
            10.0
        )
    ),
    F3_
    (
        Switch::lookupOrAddToDict
        (
            "F3",
            coeffDict_,
            false
        )
    ),
    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_)
    ),

    R_
    (
        IOobject
        (
            "R",
            runTime_.timeName(),
            mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        autoCreateR("R", mesh_)
    ),

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

    nut_ =
    (
        k_/omega_

    );
    nut_.correctBoundaryConditions();

    printCoeffs();
}


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

tmp<volSymmTensorField> EARSM_kwBSL::R() const
{
    return tmp<volSymmTensorField>
    (
        new volSymmTensorField
        (
            IOobject
            (
                "R",
                runTime_.timeName(),
                mesh_,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            R_,
            k_.boundaryField().types()
        )
    );
}


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


tmp<fvVectorMatrix> EARSM_kwBSL::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 EARSM_kwBSL::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());
        F3_.readIfPresent("F3", coeffDict());

        return true;
    }
    else
    {
        return false;
    }
}


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

    if (!turbulence_)
    {
        return;
    }
```

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

 volScalarField tau1_ = 1.0/(0.09*omega_) ;
 volScalarField tau2_ = 6.0*sqrt(nu()/(0.09*k_*omega_));
 volScalarField tau_ = max(tau1_, tau2_);

 volTensorField S = 0.5*tau_*(T(fvc::grad(U_)) + fvc::grad(U_));
 volTensorField W = 0.5*tau_*(T(fvc::grad(U_)) - fvc::grad(U_));


 volScalarField     IIs = tr(S & S) ;
 volScalarField     IIw = tr(W & W);
 volScalarField     IV  = tr(S & W & W);
 volScalarField P1 =
 (1.8*(sqr(1.8)/27.0 + (9.0/20.0)*IIs - (2.0/3.0)*IIw));
 volScalarField P2 = sqr(P1) -
  pow((sqr(1.8)/9.0 + (9.0/10.0)*IIs + (2.0/3.0)*IIw), 3);

 volScalarField N_=  1.8+9/4*pow(2*0.09*IIs,0.5);


 forAll(P2, cellI)
 {
 if (P2[cellI] < 0.0)
 {
 N_[cellI]=
 1.8/3.0 +
 2.0*pow(pow(P1[cellI],2.0)-P2[cellI],1.0/6.0)*
cos(1.0/3.0*acos(P1[cellI]/(pow(pow(P1[cellI],2.0)-P2[cellI],0.5))));
 }
 else
 {
 N_[cellI]= 1.8/3.0 +
 pow(P1[cellI]+pow(P2[cellI],0.5),1.0/3.0)+
 sign(P1[cellI]-pow(P2[cellI],0.5))
 *pow(mag(P1[cellI]-pow(P2[cellI],0.5)),1.0/3.0);
 }
 }
 volScalarField Q= (1/1.245)*(pow(N_,2)-2*IIw) + 1e-10;
 volScalarField Q1= (Q/6.0)*(2.0*pow(N_,2)-IIw) + 1e-10;
 volScalarField beta_1=-N_/Q ;
 volScalarField beta_3=-2.0*IV/(N_*Q1) ;
 volScalarField beta_4= -N_/Q ;
 volScalarField beta_6= -N_/Q1 ;
 volScalarField beta_9=-1/Q1 ;

 volTensorField WW = W & W;
 volTensorField SS = S & S;
 volTensorField SW = S & W;
 volTensorField WS = W & S;
 volTensorField SWW = SW & W;
 volTensorField WWS = WW & S;
 volTensorField WSWW = (WS & W) & W;
 volTensorField WWSW = (WW & S) & W;

 volScalarField y=wallDist(mesh_).y();
 volScalarField Re_t = pow(k_,0.5)*(y/nu());
 volScalarField yStar = 2.4*pow(Re_t,0.5) + 0.003*pow(Re_t,2.0);

volSymmTensorField aij_= symm(beta_1*S +
beta_4*(SW-WS) + beta_3*(WW-(1.0/3.0)*IIw*I) +
beta_6*(SWW +WWS -2.0/3.0*IV*I-IIw*S) +
beta_9*(WSWW-WWSW+(1.0/2.0)*IIw*(SW-WS))) ;

 tmp<volScalarField> S2 = 2*magSqr(symm(fvc::grad(U_)));
 R_= k_*(aij_ + (2.0/3.0)*I);
 volScalarField G(GName(),
 min(-R_ && T(fvc::grad(U_)), c1_*betaStar_*k_*omega_));



  // 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));
```

```
volScalarField gam =beta(F1)/0.09-alphaOmega(F1)*pow(0.41,2)/0.3;
 // Turbulent frequency equation
 tmp<fvScalarMatrix> omegaEqn
 (
     fvm::ddt(omega_)
   + fvm::div(phi_, omega_)
   - fvm::laplacian(DomegaEff(F1), omega_)
  ==
     gam*omega_*G/k_
   - fvm::Sp(beta(F1)*omega_, omega_)
   - fvm::SuSp
     (
         (F1 - scalar(1))*CDkOmega/omega_,
         omega_
     )
 );

 omegaEqn().relax();

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

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

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

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


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

```
}


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

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

## 8.4 $\phi$-f Model

Following are the OpenFOAM generated source files (zf.C) for the $\phi$-f model.

```

*******************
***ZF.C****
********************


#include "zf.H"
#include "fixedValueFvPatchField.H"
#include "zeroGradientFvPatchField.H"
#include "addToRunTimeSelectionTable.H"

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

namespace Foam
{
namespace incompressible
{
namespace RASModels
{

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

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

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

wordList zf::RBoundaryTypes() const
{
    const volScalarField::GeometricBoundaryField&
    bf(k_.boundaryField());

    wordList bTypes
    (
        bf.size(),
        zeroGradientFvPatchField<symmTensor>::typeName
    );

    forAll(bf, patchI)
    {
        if (bf[patchI].fixesValue())
        {
            bTypes[patchI] = fixedValueFvPatchField<symmTensor>::
            typeName;
        }
    }

    return bTypes;
}


tmp<volScalarField> zf::davidsonCorrectNut
(
    const tmp<volScalarField>& value
) const
{
    return min(CmuKEps_*sqr(k_)/epsilon_, value);
}


tmp<volScalarField> zf::Ts() const
{
    return max(k_/epsilon_, 6.0*sqrt(nu()/epsilon_));
}


tmp<volScalarField> zf::Ls() const
{
    return CL_*max(pow(k_, 1.5)/epsilon_,
    Ceta_*pow025(pow3(nu())/epsilon_));
}
```

```
// * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //

zf::zf
(
    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_,
            0.22
        )
    ),
    CmuKEps_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "CmuKEps",
            coeffDict_,
            0.09
        )
    ),
    C1_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "C1",
            coeffDict_,
            1.4
        )
    ),
    C2_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "C2",
            coeffDict_,
            0.3
        )
    ),
    CL_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "CL",
            coeffDict_,
            0.25
        )
    ),
    Ceta_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "Ceta",
            coeffDict_,
            110.0
        )
    ),
    Ceps2_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "Ceps2",
            coeffDict_,
            1.90
        )
    ),
    sigmaK_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "sigmaK",
            coeffDict_,
            1.0
        )
    ),
    sigmaEps_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "sigmaEps",
            coeffDict_,
            1.3
        )
    ),

    k_
    (
        IOobject
        (
            "k",
            runTime_.timeName(),
            mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_
    ),
    epsilon_
    (
        IOobject
        (
            "epsilon",
            runTime_.timeName(),
            mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_
    ),
    z_
    (
        IOobject
        (
            "z",
            runTime_.timeName(),
            mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_
    ),
    f_
    (
        IOobject
        (
            "f",
            runTime_.timeName(),
            mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_
    ),
    nut_
    (
        IOobject
        (
            "nut",
            runTime_.timeName(),
            mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_
    ),
    zMin_(dimensionedScalar("zMin", z_.dimensions(), SMALL)),
    fMin_(dimensionedScalar("fMin", f_.dimensions(), 1e-20))
{
    bound(k_, kMin_);
    bound(epsilon_, epsilonMin_);
    bound(z_, zMin_);
    bound(f_, fMin_);

    nut_ = davidsonCorrectNut(Cmu_*z_*Ts()*k_);
    nut_.correctBoundaryConditions();

    printCoeffs();
}


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

```
tmp<volSymmTensorField> zf::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_)),
            RBoundaryTypes()
        )
    );
}


tmp<volSymmTensorField> zf::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> zf::divDevReff(volVectorField& U) const
```

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


tmp<fvVectorMatrix> zf::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 zf::read()
{
    if (RASModel::read())
    {
        Cmu_.readIfPresent(coeffDict());
        CmuKEps_.readIfPresent(coeffDict());
        C1_.readIfPresent(coeffDict());
        C2_.readIfPresent(coeffDict());
        CL_.readIfPresent(coeffDict());
        Ceta_.readIfPresent(coeffDict());
        Ceps2_.readIfPresent(coeffDict());
        sigmaK_.readIfPresent(coeffDict());
        sigmaEps_.readIfPresent(coeffDict());

        return true;
    }
    else
    {
        return false;
    }
}


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

    if (!turbulence_)
    {
        return;
    }

    // use N=6 so that f=0 at walls
    const dimensionedScalar N("N", dimless, 6.0);

    const volTensorField gradU(fvc::grad(U_));
    const volScalarField S2(2*magSqr(dev(symm(gradU))));

    const volScalarField G(GName(), nut_*S2);
    const volScalarField T(Ts());
    const volScalarField L2(type() + ".L2", sqr(Ls()));
    const volScalarField alpha
    (
        "zf::alpha",
        1.0/T*((C1_ - 1.0)*(z_ - 2.0/3.0))
    );


    tmp<volScalarField> Ceps1 =
        1.4*(1.0 + 0.05*min(sqrt(1.0/z_), scalar(100.0)));
    // Update epsilon (and possibly G) at the wall
    epsilon_.boundaryField().updateCoeffs();

    // Dissipation equation
    tmp<fvScalarMatrix> epsEqn
    (
        fvm::ddt(epsilon_)
      + fvm::div(phi_, epsilon_)
      - fvm::laplacian(DepsilonEff(), epsilon_)
     ==
        Ceps1*G/T
```

```
  - fvm::Sp(Ceps2_/T, epsilon_)
);

epsEqn().relax();

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

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


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

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


// Relaxation function equation
tmp<fvScalarMatrix> fEqn
(
  - fvm::laplacian(f_)
 ==

  - fvm::SuSp(
    1.0/L2
  - 2.0*nu()/f_/k_/L2*(fvc::grad(z_) & fvc::grad(k_))
  - nu()/f_/L2*(fvc::laplacian(z_))
    ,f_)
 - 1.0/L2*(alpha - C2_*G/k_)
);

fEqn().relax();
solve(fEqn);
```

```
    bound(f_, fMin_);

    // Turbulence stress normal to streamlines equation
    tmp<fvScalarMatrix> zEqn
    (
        fvm::ddt(z_)
      + fvm::div(phi_, z_)
      - fvm::laplacian(DzEff(), z_)
      ==
        min(f_, -alpha + C2_*G/k_)
      - fvm::SuSp(G/k_
      - 2.0/k_*nut_/sigmaK_*(fvc::grad(z_) & fvc::grad(k_))/z_
      , z_)
    );

    zEqn().relax();
    solve(zEqn);
    bound(z_, zMin_);

    // Re-calculate viscosity
    nut_ = davidsonCorrectNut(Cmu_*z_*k_*T);
    nut_.correctBoundaryConditions();
}


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

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

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