CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Project work:

Implementation and run-time mesh refinement for the $k - \omega$ SST DES turbulence model when applied to airfoils.

Developed for OpenFOAM-2.2.x

Author:
Daniel Lindblad

Peer reviewed by:
Adam Jareteg
Olivier Petit

January 25, 2014

Contents

Chapter 1

Introduction

This is a report for a project in the course "CFD with OpenSource software", given by Håkan Nilsson in the autumn of 2013 at Chalmers University of Technology. The work is developed for OpenFOAM 2.2.x.

In this project, the $k-\omega$ SST DES turbulence model [1] is going to be implemented and investigated when applied to airfoil simulations. In addition to this, run-time mesh refinement features in OpenFOAM 2.2.x. will be investigated and applied to the simulations using the `dynamicRefineFvMesh` class and the `pimpleDyMFoam` solver.

The $k-\omega$ SST DES turbulence model is based upon the turbulence model $k-\omega$ SST, which is an eddy viscosity model developed for aeronautic flows with pressure induced separation and adverse pressure gradient flows [1]. The model is then equipped with what is known as DES features, where DES stands for Detached Eddy Simulations. The main goal of this approach is to reduce the turbulent viscosity in regions where the mesh is fine enough to resolve large turbulent structures, and hence do LES here. Therefore the model can be viewed as a trade off between LES and URANS, with a computational cost in between the two.

In OpenFOAM, two implementations based upon the $k-\omega$ SST model are found. The first is a further developed version of the original $k-\omega$ SST model, in which additional changes have been made for various purposes. This is a RAS turbulence model, i.e. designed to be used for RANS or URANS simulations in which all of the turbulence is modeled. Next there is an implementation of the $k-\omega$ SST SAS model presented in [2], where SAS stands for Scale Adaptive Simulation. It is based upon the original $k-\omega$ SST model in which SAS features have been introduced in an extra source term in the ω equation. Since it is only different from the $k-\omega$ SST DES model with respect to one source term, see [1] and [2], it will serve as the base for the DES implementation.

Since DES models are designed to resolve turbulence where the grid is fine enough, this allows for the user to control where turbulence is to be resolved and when it is not. One way to do this is to refine the mesh run-time in desired regions until the model starts resolving turbulence in these regions. This is the second part of the project.

A test case is set to be cambered NACA four digit airfoil in 2 and 3D. The surface shape is generated according to the NACA four digit standard, and a Fortran routine found at [3] is used to then create a blockMesh dictionary. Since three dimensional simulations are very computer intensive when finer grids are employed, it is mainly the focus to run the case in 2D during development and validation.

Chapter 2

Theory

In this chapter the $k-\omega$ SST DES turbulence model which will be implemented in OpenFOAM is presented. The boundary conditions employed will also be presented as well as a short overview of how the mesh refinement will be controlled.

2.1 The $k-\omega$ SST DES turbulence model

2.1.1 Short background

The $k-\omega$ SST-DES model is a DES modification of the RANS model $k-\omega$ SST. The standard SST model is a mix between the $k-\varepsilon$ and the $k-\omega$ model, in which the former is used in the outer part of the boundary layer as well as outside of it, and the latter in the inner part of the boundary layer [1]. The transport equation for ε in the $k-\varepsilon$ model is however rewritten as an equation for ω, giving a similar equation to the original ω equation but with some additional terms and constants. This does however allow the computer to only solve for one pair of transport equations and the switching between the two formulations is done by so called blending functions. The advantage of using different models in different regions is that the two models have their individual strengths and weaknesses. The $k-\omega$ model is a low Reynolds number model, which means that it does not need extra damping functions to ensure correct behavior close to walls. On the other hand the ω equation shows great sensitivity to free stream values of ω [1]. These two features are not present in the $k-\varepsilon$ model, i.e. it is not a low Reynolds number model and neither is it sensitive to the free stream values of k or ε.

Apart from this combination of features, the $k-\omega$ SST model has been equipped with additional features over its two base models. The first feature is a shear stress limiter (reducing ν_t), which ensures that the turbulent shear stress does not become too large in adverse pressure gradient regions, typically found on the top of an airfoil. Also a production limiter is applied on the production term in the k equation in order to prevent build up of turbulence in stagnant regions [1].

The model has finally been equipped with DES features, which aims at reducing the turbulent viscosity in regions where the mesh is sufficiently fine. This means that the model effectively transfers from RANS mode to LES mode in these regions, and the turbulent viscosity should be viewed as a sub grid scale viscosity instead.

Since the introduction of the $k-\omega$ SST model it has undergone some changes in its formulation. The aim of the following section is therefore both to give an understanding of how the model works as well as to show exactly which model that will be implemented in OpenFOAM.

2.1.2 Governing transport equations

The definition of ω in terms of k and ε reads

$$\omega = \frac{\varepsilon}{\beta^* k}, \quad \beta^* = C_\mu. \tag{2.1}$$

Here C_μ is the constant present in the expression for the turbulent viscosity in the $k-\varepsilon$ model, and it is equal to 0.09. This definition is used to rewrite the transport equation for ε as a transport equation for ω, a derivation that will not be presented here.

The modeled transport equation for the turbulent kinetic energy, k, used in $k-\omega$ SST DES model reads, see [1], [2]

$$\frac{\partial k}{\partial t} + \frac{\partial(\bar{u}_i k)}{\partial x_i} = \tilde{P}_k + \frac{\partial}{\partial x_i}\left[\left(\nu + \frac{\nu_t}{\sigma_k}\right)\frac{\partial k}{\partial x_i}\right] - \beta^* k\omega F_{DES}. \tag{2.2}$$

It is the modified production $\tilde{P}_k$ as well as the term F_{DES} that differ this transport equation from those found in the base models. The bar over the velocity is indicating that the velocity field is filtered in the sense that not all turbulent fluctuations are resolved. In URANS regions, this would correspond to a time filter, and in LES regions a space filter. This is however just a formality, since no actual filtering is done in the solver. Next the transport equation for ω reads [1], [2]

$$\begin{aligned}\frac{\partial \omega}{\partial t} + \frac{\partial(\bar{u}_i \omega)}{\partial x_i} &= P_\omega + \frac{\partial}{\partial x_i}\left[\left(\nu + \frac{\nu_t}{\sigma_\omega}\right)\frac{\partial \omega}{\partial x_i}\right] - \beta\omega^2 \\ &\quad + 2(1-F_1)\sigma_{\omega 2}\frac{1}{\omega}\frac{\partial k}{\partial x_i}\frac{\partial \omega}{\partial x_i}.\end{aligned} \tag{2.3}$$

The last cross diffusion term stems from the $k-\varepsilon$ model when formulated as a $k-\omega$ model and hence should only be active away from the wall. Now, to begin with, the production term in the ω equation is evaluated as [1]

$$P_\omega = \alpha S^2, \tag{2.4}$$

where $S = \sqrt{2\bar{s}_{ij}\bar{s}_{ij}}$ is the invariant measure of the strain rate [1] and

$$\bar{s}_{ij} = \frac{1}{2}\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right), \tag{2.5}$$

is the strain rate tensor of the filtered velocity field. Next we have the first blending function, $F1$, which is evaluated as [1], [2]

$$F_1 = \tanh(\xi^4), \quad \xi = \min\left[\max\left(\frac{\sqrt{k}}{\beta^*\omega y}, \frac{500\nu}{y^2\omega}\right), \frac{4\sigma_{\omega 2}k}{CD_\omega y^2}\right]. \tag{2.6}$$

Its purpose is to switch the SST model between the $k-\omega$ and $k-\varepsilon$ formulation by changing between the value 1 close to the wall, to 0 in the outer part of the boundary layer as well as outside of it. It appears in the ω equation (2.3) in the last term, and hence can be seen to activate this term when it goes to 0, giving the $k-\varepsilon$ formulation instead. The model constants α, β, σ_k and σ_ω are in fact also blended between the two formulations to give model constants applicable for the formulation used in a certain region. This is done according to

$$\phi = F_1\phi_1 + (1-F_1)\phi_2, \tag{2.7}$$

when ϕ_i denotes either α_i or β_i, or according to

$$\frac{1}{\psi} = F_1\frac{1}{\psi_1} + (1-F_1)\frac{1}{\psi_2}, \tag{2.8}$$

when ψ_j denotes either σ_{kj} or $\sigma_{\omega j}$. The values of the constants are taken from [1]

$$\begin{aligned}\alpha_1 &= 5/9, & \alpha_2 &= 0.44,\\ \beta_1 &= 3/40, & \beta_2 &= 0.0828,\\ \frac{1}{\sigma_{k1}} &= 0.85, & \tfrac{1}{\sigma_{k2}} &= 1,\\ \frac{1}{\sigma_{\omega 1}} &= 0.5, & \tfrac{1}{\sigma_{\omega 2}} &= 0.856.\end{aligned}$$

These constants are in fact derived from the original constants of the $k-\varepsilon$ and $k-\omega$ models. Furthermore, CD_ω stems from the cross diffusion term in (2.3) and is defined by, see [1]

$$CD_\omega = \max\left(2\sigma_{\omega 2}\frac{1}{\omega}\frac{\partial k}{\partial x_i}\frac{\partial \omega}{\partial x_i}, 10^{-10}\right). \tag{2.9}$$

The SST model also features a shear stress limiter, which will switch from a eddy viscosity model to the Johnson King model in regions where the shear stress becomes too large, i.e. adverse pressure gradient flow. It is defined as [1]

$$\nu_t = \frac{a_1 k}{\max(a_1\omega, SF_2)}, \tag{2.10}$$

where $a_1 = \sqrt{\beta^*}$ and F_2 is another blending function that ensures that the Johnson King model only can be active in the boundary layer. It is defined as

$$F_2 = \tanh(\eta^2), \quad \eta = \max\left(\frac{2\sqrt{k}}{\beta^*\omega y}, \frac{500\nu}{y^2\omega}\right). \tag{2.11}$$

It should be mentioned that throughout, y denotes the distance to the closest wall. The production of turbulent kinetic energy features a limiter and is defined as [1]

$$\begin{aligned} \tilde{P}_k &= \min(P_k, 10\cdot\beta^* k\omega), \\ P_k &= \nu_t\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right)\frac{\partial \bar{u}_i}{\partial x_j}. \end{aligned} \tag{2.12}$$

It now remains to show how the DES features are incorporated. In the dissipation term of the k equation, the function F_{DES} is included. In the original $k-\omega$ SST model the dissipation is not multiplied by this term, which reads [1]

$$F_{DES} = \max\left(\frac{L_t}{C_{DES}\Delta}, 1\right) \tag{2.13}$$

Where $L_t = \sqrt{k}/(\beta^*\omega)$ is a turbulent length scale, $\Delta = \max(\Delta x_1, \Delta x_2, \Delta x_3)$ is the largest side of a cell at the present point in the grid and $C_{DES} = 0.61$. When the local grid is fine enough in all directions, compared to the turbulent length scale, the F_{DES} term grows larger than 1. This will in turn reduce k, which reduces ν_t, see (2.10), and hence allows the solution to go unsteady. Thus in regions where the mesh is fine enough to resolve turbulence, the model will reduce the amount of modeled turbulent shear stress and allow the region to be treated with LES. There is however a problem with this formulation when it comes to near wall regions. Since the grid generally is fine close to walls, it could happen that the solution is triggered to go unsteady here. This is generally a bad result since near wall turbulent structures are very small and require a very fine grid to be resolved properly with LES. These requirements might not be satisfied by the mesh and hence a poorly resolved LES simulation close to the walls might be the result. Therefore the model is also presented with DDES (Delayed Detached Eddy Simulations) features, which protects it from going unsteady near the wall. In this case the F_{DES} term is modified according to

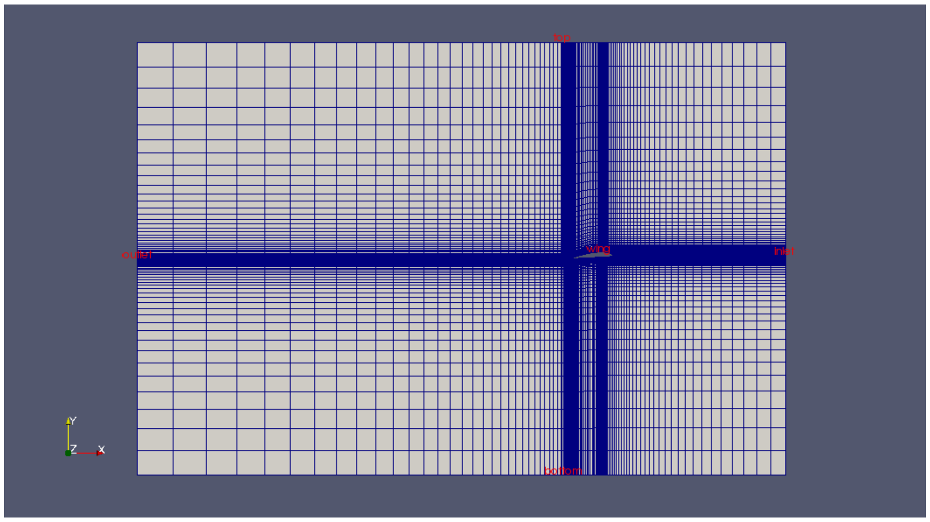
$$F_{DDES} = \max\left(\frac{L_t}{C_{DES}\Delta}(1-F_S), 1\right) \tag{2.14}$$

in which F_S is a blending function, chosen as either F_1 or F_2. As can be seen, the blending function will ensure that the term is reduced in the boundary layer, since the blending function becomes close to 1 here.

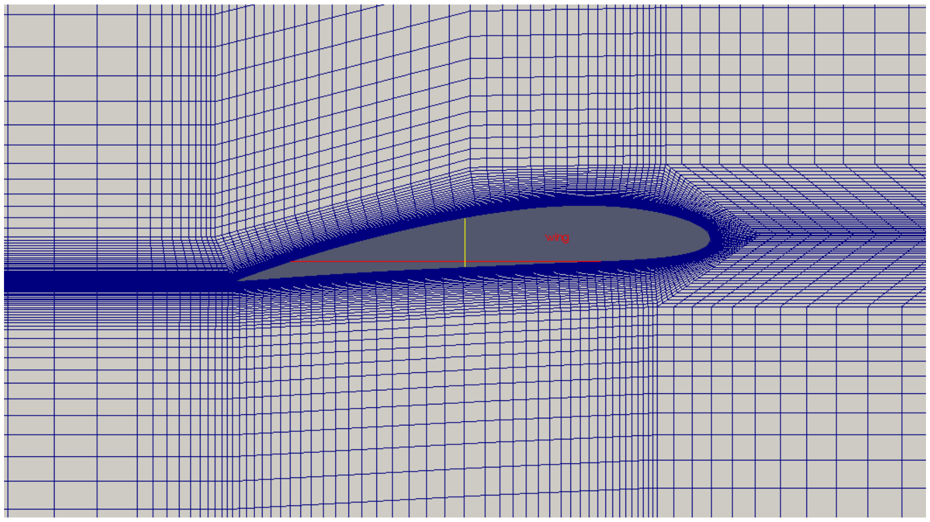
2.2 Boundary conditions

Indeed, for any CFD simulation, the boundary treatment is important in order to get accurate and reliable results. As mentioned earlier, the $k-\omega$ SST model is a blend of the $k-\varepsilon$ and the $k-\omega$ with some additional features. One reason for this is that the $k-\omega$ model shows great sensitivity to the free stream values of ω whereas it behaves much better in the near wall regions than the $k-\varepsilon$ model [1]. In addition to this, advanced wall treatment for ω has also been developed in order to make it less sensitive to near wall grid resolution, see [1] and [4].

To investigate how the new turbulence model performs, a test case was chosen to be airfoil simulations. The boundary conditions for this type of simulation can vary dependent on under which setting the airfoil is simulated. For example if it is present in a turbulent flow field or not affects the boundary conditions on k and ω and also, if large turbulent fluctuations approach the airfoil, the velocity as well. Below the boundary conditions applied in OpenFOAM are presented and discussed in connection to a typical computational domain as presented in Figure 2.1.



(a) Entire domain.



(b) Wing Patch.

Figure 2.1: Computational domain including patch names for NACA 4412 Airfoil.

The following table lists the boundary conditions applied in a two dimensional case.

Table 2.1: Boundary conditions for the $k-\omega$ SST DES model.

Field	Patch	Type	Specifications
U	Inlet	freestream	freestreamValue uniform (-1 0 0)
	Outlet	freestream	freestreamValue uniform (-1 0 0)
	Top	freestream	freestreamValue uniform (-1 0 0)
	Bottom	freestream	freestreamValue uniform (-1 0 0)
	Wing	fixedValue	uniform (0 0 0)
p	Inlet	freestreamPressure	
	Outlet	freestreamPressure	
	Top	freestreamPressure	
	Bottom	freestreamPressure	
	Wing	zeroGradient	
k	Inlet	fixedValue	uniform 1e-6
	Outlet	zeroGradient	
	Top	fixedValue	uniform 1e-6
	Bottom	fixedValue	uniform 1e-6
	Wing	fixedValue	uniform 1e-10
ω	Inlet	fixedValue	uniform 1
	Outlet	zeroGradient	
	Top	fixedValue	uniform 1
	Bottom	fixedValue	uniform 1
	Wing	omegaWallFunction	uniform 1e3
ν_t/ν_{sgs}	Inlet	calculated	uniform 0
	Outlet	calculated	uniform 0
	Top	calculated	uniform 0
	Bottom	calculated	uniform 0
	Wing	fixedValue	uniform 0

In general, the boundary conditions are selected to simulate a case where the airfoil is traveling through a completely still fluid where no turbulence is present. One specific thing that is of great importance to achieve this is to set k and ω at the boundary such that the turbulent viscosity becomes small in comparison to the kinematic one. This is achieved by noting that far away from walls, the turbulent viscosity is given by k/ω and hence the relation between the two quantities can be chosen such that ν_t/ν is small. In this case $\nu_t/\nu = 0.1$. Below follows a short description of the boundary conditions employed for the different quantities.

Velocity, U To begin with, a no-slip condition is applied to the airfoil, which is realized with the fixedValue boundary condition. At the other boundaries, the freestream boundary condition is applied. It is applicable at boundaries of type patch or wall and is a derived boundary condition of type inletOutlet. This means that for every face of the patch, the boundary condition will be assigned dependent on the local flux. If the flux goes inside the domain, it will be assigned a fixed value and if it goes out it will be assigned a zero gradient condition. This is typically what we want to achieve in this case.

Pressure, p The pressure boundary condition is of type zeroGradient on the wing. The physical motivation for this is that there is no flow through the wall and hence no pressure gradient should exist normal to the wall trying to drive the fluid through the wall. On the other boundaries, the boundary condition is of type freestreamPressure. The first thing to be noted here is that this boundary condition must be used together with the freestream boundary condition for U. It is applicable to boundaries of type patch or wall, and is a derived boundary condition of type zeroGradient. It sets what is known as a free-stream condition on the pressure. This means that it is a zero gradient condition that constrains the flux across the boundary based on the free stream velocity (U).

Turbulent kinetic energy, k The turbulent kinetic energy is prescribed by setting it to a small value on the wing. What small means is really not well defined, so therefore it is good practice to simply prescribe it arbitrarily small and check that the simulations look reasonable. In reality, $k = 0$ on the wing since no turbulent fluctuations are present at the wing. However some turbulence models include division of k in the transport equations and therefore it is better practice to choose $k \neq 0$. Note that no wall function approach is used, and hence the first internal grid point must be located at $y^+ \leq 11.63$ [5]. On the inlet, top and bottom, it has simply been prescribed to some arbitrary small number such that $\nu_t/\nu = 0.1$. A zero gradient condition is used at the outlet simply to not disturb the wake forming behind the airfoil.

Turbulent frequency, ω On the wing, a special boundary condition has been applied to ω through `omegaWallFunciton`. The reason to use special wall functions is that $\omega \to \infty$ as $y \to 0$, and hence it can not simply be specified at the wall. Instead it is set in the first internal node using a special formula. A version of this type of wall treatment is presented in [4], although it needs to be stated that this is not the exact same approach that is taken in OpenFOAM. The method applied in OpenFOAM is essentially a blend between the usual low-Re formulation and a wall function treatment dependent on if the grid is too coarse close to the wall. In short, ω is set in the first internal node as a squared average between the low-Re formulation and wall function formulation according to

$$\omega = \sqrt{\omega_{\text{Vis}}^2 + \omega_{\text{log}}^2}. \tag{2.15}$$

Here ω_{Vis} and ω_{log} are computed according to

$$\omega_{\text{Vis}} = \frac{6\nu}{\beta_1 y^2}, \tag{2.16}$$

$$\omega_{\text{log}} = \frac{\sqrt{k}}{\sqrt[4]{\beta^*}\kappa y}. \tag{2.17}$$

This allows for a smooth mixing, that automatically sets a suitable value for ω in the first node dependent on its location. At the inlet, top and bottom, the values are prescribed in order for the turbulent/sub grid scale viscosity to be small in comparison to the kinematic viscosity and hence simulate little presence of turbulence.

Turbulent viscosity/Sub grid scale viscosity, ν_t/ν_{sgs} On the wing, the turbulence goes to zero and hence any shear stresses caused by turbulence should be zero here as well. Therefore, the turbulent or sub grid scale, viscosity is set to zero at the wing. For the other boundaries, the calculated boundary condition is applied. This means that no special boundary condition is assigned to the turbulent viscosity, but it is assumed to have been assigned using other fields. This is appropriate since we want k and ω to determine ν_t/ν_{sgs}, but still some boundary condition needs to be applied since the turbulent viscosity is needed in the 0/ directory.

2.3 Applying run-time mesh refinement

In this project, mesh refinement is also going to be used. Since the mesh refinement should happen automatically at run-time, some quantity needs to exist that indicates if the mesh should be refined in a certain location. One way to do this, which is implemented in OpenFOAM, is to store a value for every cell and then refine a certain cell if the value lies within a specific interval. The limits and which type of scalar field that is used as an indicator is then up to the user to decide.

In this project, the aim is to adapt the mesh such that turbulence is resolved. This means that if turbulence is resolved, the mesh does not need to be refined. However in regions where the turbulence exists, but its length scales are smaller than the local grid size, the mesh should be refined in order to resolve turbulence here. The natural way to achieve this is to work with the F_{DES} or

F_{DDES} term, see (2.13) and (2.14). If this term is larger than 1, the DES effect has been enabled and the model should hopefully start to resolve turbulence in that region. On the other hand if it is equal to 1, the model operates in RANS mode in that region and no turbulence should be resolved. For the DES and DDES model, the ratio of the turbulent to the grid length scale is therefore defined as

$$L_{DES} = \frac{\sqrt{k}}{C_{DES}\beta^{*}\omega\Delta}, \tag{2.18}$$

$$L_{DDES} = \frac{\sqrt{k}}{C_{DES}\beta^{*}\omega\Delta}(1 - F_S). \tag{2.19}$$

Since these are the terms within the max operator in (2.13) and (2.14) respectively, it holds that the DES features are active if F_{DES} or F_{DDES} are greater than 1.

To select the regions in which mesh refinement should be applied, this term is therefore going to be used as an indicator. If it is greater than some α_u, it is sufficiently large and no mesh refinement is needed. On the other hand if it is less than some constant α_l close to 0, we have a region in which very little turbulence is present and mesh refinement in unnecessary. If it however lies in between these two limits, we have a region in which turbulence is present but the mesh is not fine enough, or just about fine enough, to resolve turbulence. Here mesh refinement is appropriate. Hence a good indicator of mesh refinement is that $\alpha_l \leq F_{DES} \leq \alpha_u$ or $\alpha_l \leq F_{DDES} \leq \alpha_u$ depending on the model used. The lower and upper limit must be tuned by repeated simulations and evaluation of results.

Chapter 3

The OpenFOAM implementation

In this chapter the implementation of a LES-class turbulence model, namely the $k - \omega$ SST SAS model, as well as the dynamic mesh features of the pimpleDyMFoam solver are described. The reason that the $k - \omega$ SST SAS model will be described is that it will later be modified into the $k - \omega$ SST DES model as mentioned in the introduction. The discussion does mainly regard high level programming and technical details on a deeper level are left out. The aim is to give a general understanding of the implementation in order to be able to modify the turbulence model as well as using the dynamic mesh features of the solver.

3.1 $k - \omega$ SST SAS model

The declaration and definition files of the $k - \omega$ SST SAS turbulence model are found at

```
$FOAM_SRC/turbulenceModels/incompressible/LES/kOmegaSSTSAS/
```

3.1.1 kOmegaSSTSAS.H

The declaration file kOmegaSSTSAS.H is presented in parts below. It begins by including some declaration files before the class declaration starts, as seen below.

```

#ifndef kOmegaSSTSAS_H
#define kOmegaSSTSAS_H

#include "LESModel.H"
#include "volFields.H"
#include "wallDist.H"

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

namespace Foam
{
namespace incompressible
{
namespace LESModels
{

/*---------------------------------------------------------------*\
                     Class kOmegaSSTSAS Declaration
\*---------------------------------------------------------------*/
```

```
75
76  class kOmegaSSTSAS
77  :
78      public LESModel
79  {
80      // Private Member Functions
81
82          //- Update sub-grid scale fields
83          void updateSubGridScaleFields(const volScalarField& D);
84
85          // Disallow default bitwise copy construct and assignment
86          kOmegaSSTSAS(const kOmegaSSTSAS&);
87          kOmegaSSTSAS& operator=(const kOmegaSSTSAS&);
88
89
90  protected:
91
92      // Protected data
93
94          // Model constants
95
96              dimensionedScalar alphaK1_;
97              dimensionedScalar alphaK2_;
98
99              dimensionedScalar alphaOmega1_;
100             dimensionedScalar alphaOmega2_;
```

Listing 3.1: file: `kOmegaSSTSAS.H`

To save some space, a set of model constant declarations have been left out. After them, the declaration of the fields the model uses and calculates follows.

```
124
125         // Fields
126
127             volScalarField k_;
128             volScalarField omega_;
129             volScalarField nuSgs_;
```

Listing 3.2: file: `kOmegaSSTSAS.H`

To begin with there is the declaration files. `LESModel.H` is included since this turbulence model inherits the `LESModel` class and hence is a sub class to this class. Furthermore, `wallDist.H` is included to enable the calculation of distance to walls, as the name indicates. It uses the utility `patchDist`, but only calculates the distance to boundaries of type wall, not a general patch. This allows the user to select which boundaries that the solver should actually interpret as physical walls and not as general boundaries such as an inlet or cyclic patch. For more details on `wallDist` and `patchDist`, please refer to the installation where they are found at

```
$FOAM_SRC/finiteVolume/fvMesh/wallDist/
```

After the declaration files, the namespace is set for this turbulence model, and as seen all declarations and so forth will be done in namespace `Foam::incompressible::LESModel`. Hence functions used in this turbulence model will be those developed for incompressible LES models.

Next, starting on line 76. the class is declared, and as seen it also inherits the attributes of the `LESModel` class and thus becomes a sub class to `LESModel`. Next follows a set of declarations, first of `updateSubgridScaleFields`, which is a function updating the turbulent/subgrid scale viscosity using call-by-reference. Since it does not say `public:` or `protected:` above we also know that this is a private member function.

After this a set of member constants and member fields are declared. They are set as protected, meaning that they are visible only within this class and classes that inherit from the class `kOmegaSSTSAS` as well as friend functions to this class. Also when run-time mesh refinement is used, a separate field depicting where mesh refinement should occur is needed as well. For this purpose a new `volScalarField` will have to be added as well. The model constants that the SAS and DES implementation share have the same values in both models. The only difference is that the models use some specific constants in their respective source terms. It should also be mentioned that the names used in this implementation differ to some extent from the names found in [2]. When describing how to do the DES implementation, the corresponding names in literature and implementation will be presented to avoid confusion.

Next the protected member functions are declared, the first one are shown below

```

    // Protected Member Functions

        tmp<volScalarField> Lvk2
        (
            const volScalarField& S2
        ) const;

        tmp<volScalarField> F1(const volScalarField& CDkOmega) const;
        tmp<volScalarField> F2() 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_);
        }
```

Listing 3.3: file: `kOmegaSSTSAS.H`

These protected functions will only be visible in the same way as for the protected member data shown above. Here, typically, different functions needed to evaluate complex terms within the transport equations as well as blending functions used to shift between the model formulations are declared. It is generally more convenient to define functions returning these terms, than to state them explicitly when setting up and discretizing the transport equations. When modifying the turbulence model, the F_{DES} or F_{DDES} should also be declared here. It is worth noting that all these functions return `volScalarFields` (related to the mesh), but they are also of class `tmp`. The wrapper class `tmp` is used for large objects (memory wise) and allows them to be returned from the function without being copied. It also allows the memory occupied by this object to be cleared as soon as it is not used anymore. In short, it is the function type of choice when calculating and returning large fields that will only be used temporarily to compute some term or quantity in the transport equations. Furthermore, the keyword `const` is predominantly used throughout member function declarations above. The second `const`, used after defining which parameters to take in, says that this function may not modify the original objects it takes in. The first ones says that the object sent in is of type constant.

The function `blend` is also defined here, not just declared. It is the implementation of (2.7) and (2.8) and is used to blend the model constants. The reason to why only one type of blend function is needed is that the values of $1/\sigma_{kj}$ and $1/\sigma_{\omega j}$ are used instead of σ_{kj} and $\sigma_{\omega j}$. An example of how `blend` is applied is seen in the next member function, where $1/\sigma_k$ is computed using the blending between the $k - \omega$ and the $k - \varepsilon$ values. As can be seen the model constants do not have the same name in the implementation as in the papers it is based upon.

Finally some public member functions are declared, the first piece of the code doing this is shown below

```

    // Member Functions

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

        //- Return omega
        virtual tmp<volScalarField> omega() const
        {
            return omega_;
        }
```

Listing 3.4: file: `kOmegaSSTSAS.H`

The public member functions are visible outside of the class, and thus are typically functions returning fields that are of interest outside the turbulence model, such as k or ω. They are also virtual, meaning that their function are determined run-time.

3.1.2 kOmegaSSTSAS.C

The main definition file is presented in parts below. First comes some include files as well as the definition of the correct namespace and and the protected member functions

```

#include "kOmegaSSTSAS.H"
#include "addToRunTimeSelectionTable.H"
#include "wallDist.H"

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

namespace Foam
{
namespace incompressible
{
namespace LESModels
{

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

```

```
defineTypeNameAndDebug(kOmegaSSTSAS, 0);
addToRunTimeSelectionTable(LESModel, kOmegaSSTSAS, dictionary);

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

void kOmegaSSTSAS::updateSubGridScaleFields(const volScalarField& S2)
{
    nuSgs_ == a1_*k_/max(a1_*omega_, F2()*sqrt(S2));
    nuSgs_.correctBoundaryConditions();
}
```

Listing 3.5: file: `kOmegaSSTSAS.C`

To begin with, the previously considered declaration files `kOmegaSSTSAS.H` and `wallDist.H` are included. In this part we will see the use of the wall distance when it comes to calculating the different terms in the transport equations.

The first protected member function is the one that computes the turbulent, or sub grid scale viscosity, called `updateSubGridScaleFields`. As can be noted, it is the exact same expression as (2.10) assuming that what is called `S2` is equal to $S^2 = 2\bar{s}_{ij}\bar{s}_{ij}$. At line 348 this field is computed as

```
348     volScalarField S2(2.0*magSqr(symm(gradU())));
```

Listing 3.6: file: `kOmegaSSTSAS.C`

The programmers guide [6], section 1.4.1 gives, since `gradU()` simply gives the gradient of the vector field, that

$$\texttt{symm(gradU())} = \frac{1}{2}\left(\nabla U + (\nabla U)^T\right).$$

Furthermore, section 1.3.6 and 1.4.1 of the programmers guide gives that the `magSqr` operation performs a double inner product to a second order tensor according to

$$\texttt{magSqr}(\mathbf{T}) = \mathbf{T} : \mathbf{T}.$$

From the programmers guide, section 1.3.2, we have that $\mathbf{T} : \mathbf{T} = T_{ij}T_{ij}$, where T_{ij} are the components of the second order tensor. Hence, the implementation computes `S2` as

$$\begin{aligned} \mathrm{S2} &= 2\left(\frac{1}{2}\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right)\frac{1}{2}\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right)\right) \\ &= 2\bar{s}_{ij}\bar{s}_{ij}. \end{aligned}$$

The second equality comes from definition (2.5). The conclusion is that the implementation uses the same formula for the turbulent/sub grid scale viscosity given in (2.10).

Next the two blending functions F_1 and F_2 are defined

```

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

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

    return tanh(pow4(arg1));
}


tmp<volScalarField> kOmegaSSTSAS::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));
}
```

Listing 3.7: file: `kOmegaSSTSAS.C`

The implementation of these two blending functions differs slightly from formulas (2.6) and (2.11). In both cases there is an extra $\min(a,b)$ operation being performed in order to evaluate ξ and η (denoted `arg1` and `arg2`) which then will go into the tanh function. For the case of F_1, the implementation instead computes the quantity $\tanh(\tilde{\xi^4})$, where $\tilde{\xi} = \min(\xi, 10)$. The reason why is simply that $\tanh(10^4)$ is so close to 1, that any larger value of ξ than 10 simply does not affect the value of tanh to anything but a very small extent. Hence the implementation avoids forcing OpenFOAM to calculate tanh of some very large value of ξ^4, hopefully giving a faster and/or more stable code. The analogous approach is used for F_2.

Keeping this small change in mind, a look at the implementation reveals that it would be the same as the formulas (2.6) and (2.11) if it holds that

$$\texttt{CDkOmega} = 2\sigma_{\omega 2}\frac{1}{\omega}\frac{\partial k}{\partial x_i}\frac{\partial \omega}{\partial x_i}.$$

At line 354 this term is calculated according to

```
354     volScalarField CDkOmega((2.0*alphaOmega2_)*(gradK & gradOmega)/omega_);
```

Listing 3.8: file: `kOmegaSSTSAS.C`

The `&` is an inner product in OpenFOAM, which for the vectors `gradK` and `gradOmega` gives the formula

$$\nabla k \cdot \nabla \omega = \frac{\partial k}{\partial x_i}\frac{\partial \omega}{\partial x_i}.$$

Hence the implementation uses the desired formula for the cross diffusion term as given in (2.9).

The rest of the implemented protected member functions are specific to the SAS model. When adding the new F_{DES} or F_{DDES} term later on, it will be convenient to include it as a function like `F1`. This will make it easy to modify and view its implementation as well as switching between the DES and DDES formulation. Later on a `volScalarField` can then be created using this function, prior to setting up and solving the discrete system of equations.

Next comes the construction of the object `kOmegaSSTSAS`

```

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

kOmegaSSTSAS::kOmegaSSTSAS
(
    const volVectorField& U,
    const surfaceScalarField& phi,
    transportModel& transport,
    const word& turbulenceModelName,
    const word& modelName
)
:
    LESModel(modelName, U, phi, transport, turbulenceModelName),

    alphaK1_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "alphaK1",
            coeffDict_,
            0.85034
        )
    ),
    alphaK2_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "alphaK2",
            coeffDict_,
            1.0
        )
    ),
```

Listing 3.9: file: `kOmegaSSTSAS.C`

To save some space, not all definitions of protected member constants have been included. They are followed by the fields used by the turbulence model as follows.

```

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

    omega_
    (
        IOobject
        (
            "omega",
            runTime_.timeName(),
            mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_
    ),

    nuSgs_
    (
        IOobject
        (
            "nuSgs",
            runTime_.timeName(),
            mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_
    )
{
    omegaMin_.readIfPresent(*this);

    bound(k_, kMin_);
    bound(omega_, omegaMin_);

    updateSubGridScaleFields(2.0*magSqr(symm(fvc::grad(U))));

    printCoeffs();
}
```

Listing 3.10: file: `kOmegaSSTSAS.C`

After the appropriate parameters have been taken in to construct the object `kOmegaSSTSAS`, the construction begins by directly calling the constructor of the `LESModel` class. After this all model constants are read from a sub dictionary in the `LESProperties` dictionary called `kOmegaSSTSASCoeffs`, or defined if not present. Also all the relevant fields, such as k and ω are read. Since the DES model includes a new model constant, it is important to include it here as well. Also, in the case of run-time mesh refinement, a new field is needed which will be computed by the turbulence model. Thus it must also be read here in order to be computed. The maybe most important thing in the body of the constructor is the call for the calculation of the turbulent viscosity through the function `updateSubGridScaleFields`.

The final part of the turbulence model includes solving for the turbulent quantities k and ω together with the calculation of ν_t/ν_{sgs}. This is done in the virtual void function `correct`, which is a public member function of the class `kOmegaSSTSAS` as can be seen in the declaration file `kOmegaSSTSAS.H`. Hence, since it is a virtual function, it will override the function `correct` in the `LESModel` class, which the `kOmegaSSTSAS` class inherited. This allows OpenFOAM to select which turbulence model that is used run-time, since the function calculating the turbulent viscosity is overridden by the chosen turbulence model. It starts by solving the equation for k

```

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

void kOmegaSSTSAS::correct(const tmp<volTensorField>& gradU)
{
    LESModel::correct(gradU);

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

    volScalarField S2(2.0*magSqr(symm(gradU())));
    gradU.clear();

    volVectorField gradK(fvc::grad(k_));
    volVectorField gradOmega(fvc::grad(omega_));
    volScalarField L(sqrt(k_)/(pow025(Cmu_)*omega_));
    volScalarField CDkOmega((2.0*alphaOmega2_)*(gradK & gradOmega)/omega_);
    volScalarField F1(this->F1(CDkOmega));
    volScalarField G(GName(), nuSgs_*S2);

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

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

    tmp<volScalarField> grad_omega_k = max
    (
        magSqr(gradOmega)/sqr(omega_),
        magSqr(gradK)/sqr(k_)
    );
```

Listing 3.11: file: `kOmegaSSTSAS.C`

What can first be noted is that support for a changing mesh is included, in which case the wall distance will be corrected by letting the function `correct()` operate on the field. After this a set of fields necessary to set up the discrete system of equations are calculated. Since the names of the different operations are very logical in OpenFOAM, it is not necessary to explain all of these fields. The fields `S2` and `CDkOmega` have already been considered, and the only one that raises some questions is the field called `G`. This is the production term P_k without the limiter applied to it, since according to the implementation we have

$$\begin{aligned}
\mathrm{G} &= \nu_t S^2 \\
&= \nu_t 2\bar{s}_{ij}\bar{s}_{ij} \\
&= 2\nu_t\bar{s}_{ij}\left(\bar{s}_{ij}+\bar{\Omega}_{ij}\right) \\
&= \nu_t\left(\frac{\partial\bar{u}_i}{\partial x_j}+\frac{\bar{u}_j}{\partial x_i}\right)\left[\frac{1}{2}\left(\frac{\partial\bar{u}_i}{\partial x_j}+\frac{\partial\bar{u}_j}{\partial x_i}\right)+\frac{1}{2}\left(\frac{\partial\bar{u}_i}{\partial x_j}-\frac{\partial\bar{u}_j}{\partial x_i}\right)\right] \\
&= \nu_t\left(\frac{\partial\bar{u}_i}{\partial x_j}+\frac{\bar{u}_j}{\partial x_i}\right)\frac{\partial\bar{u}_i}{\partial x_j} \\
&= P_k.
\end{aligned}$$

Here it was used that the product of the anti symmetric tensor $\bar{\Omega}_{ij}$ and the symmetric tensor $\bar{s}_{ij}$ is zero since $\bar{\Omega}_{ij}=-\bar{\Omega}_{ji}$.

When implementing the DES model it is the dissipation in the k equation that will be modified by multiplying it with an extra term, namely F_{DES} or F_{DDES}. This field must also be calculated in advance using the function describing it.

Turning to the creation of the discrete system of equations, the transport equation for k is set up in accordance with (2.2) apart from the fact that no DES term is included. The production term can be seen to include the limiter, as written in (2.12) since `c1_` $= 10$. Further the dissipation term comes in by using the operation `fvm::Sp(betaStar_*omega_, k_)`. Indeed the dissipation term could simply have been added by writing `betaStar_*omega_*k_`, in which case it would have been implemented explicitly. The thing is though that the entire dissipation term always is negative due to the minus sign, and the fact that k, ω and $\beta*$ are positive. To improve convergence and stability it is better to treat negative sources implicitly, which is achieved using the syntax `Sp()`. The answer to why is because when a source term is treated explicitly, it is included in the load vector $\mathbf{b}$ in the discretisized system of equations $A\mathbf{k}=\mathbf{b}$, where $\mathbf{k}$ is a vector including the nodal values of k. Hence it can prior to convergence cause the elements in $\mathbf{k}$ to go negative, which is bad when considering that $k\geq 0$ by definition. However, when it is treated implicitly, it is instead included in the diagonal of A instead. Since it was negative on the right hand side, it will instead give a positive contribution on the left hand side, making the matrix A more diagonally dominant.

After the k equation has been solved, the ω equation is set up and solved according to

```

    // Turbulent frequency equation
    {
        fvScalarMatrix omegaEqn
        (
            fvm::ddt(omega_)
          + fvm::div(phi(), omega_)
          - fvm::laplacian(DomegaEff(F1), omega_)
        ==
            gamma(F1)*S2
          - fvm::Sp(beta(F1)*omega_, omega_)
          - fvm::SuSp       // cross diffusion term
            (
                (F1 - scalar(1))*CDkOmega/omega_,
                omega_
            )
          + FSAS_
           *max
            (
                    dimensionedScalar("zero",dimensionSet(0, 0, -2, 0, 0), 0.0),
                    zetaTilda2_*kappa_*S2*sqr(L/Lvk2(S2))
                  - 2.0/alphaPhi_*k_*grad_omega_k
                )
            );

            omegaEqn.relax();
            omegaEqn.solve();
        }
        bound(omega_, omegaMin_);

        updateSubGridScaleFields(S2);
}
```

Listing 3.12: file: `kOmegaSSTSAS.C`

The implementation is the same as presented in (2.3) apart from the so called SAS term, which of course must be removed when implementing the DES model. Another interesting thing that can be noted here is the implementation of the cross diffusion term according to `fvm::SuSp(F1 - scalar(1)*CDkOmega/omega_, omega_)`. The use of `SuSp(a,b)` allows for a flexible treatment of the source term with respect to the sign of `a`. If it is negative, the source term is treated implicitly, and if it is positive it will be treated explicitly. Explicit treatment is favorable if the source term is positive and hence it will be done if possible.

Finally it can be seen that the turbulent viscosity is updated, and hence the purpose of the correct function is achieved in a sense. When a new field for mesh refinement is added, it will be calculated here as well, using the most recent values of k and ω.

The final thing that should be reviewed in order to be able to implement the DES model is the name and the values of the model constants. As it turns out, their names in the implementation do not always correspond to the literature. The following table presents the names and values used in the implementation, together with the corresponding names used in the literature.

Table 3.1: Model constant names used in implementation and corresponding names in theory.

Theory	Implementation	Value in implementation
α_1	`gamma1_`	0.5532
α_2	`gamma2_`	0.4403
β_1	`beta1_`	0.075
β_2	`beta2_`	0.0828
$1/\sigma_{k1}$	`alphaK1_`	0.85034
$1/\sigma_{k2}$	`alphaK2_`	1.0
$1/\sigma_{\omega 1}$	`alphaOmega1_`	0.5
$1/\sigma_{\omega 2}$	`alphaOmega2_`	0.85616
β^*	`betaStar_`	0.09
a_1	`a1_`	0.31
C_{DES}	`CDES_`	0.61

3.2 pimpleDyMFoam

The solver `pimpleDyMFoam` is a modification of the `pimpleFoam` solver that supports meshes of class `dynamicFvMesh`. The class `dynamicFvMesh` is a base class for meshes that can move and/or change topology. The solver `pimpleFoam` is a transient solver developed for incompressible flows, and is based on the PISO and SIMPLE algorithms. It is in addition to the `pisoFoam` solver, which is based on the PISO algorithm, developed to be able to handle larger time steps. This section will not focus on how the actual transport equations are solved using the merged PISO-SIMPLE algorithm, but on how the dynamic meshes are treated within the solver. There are a number of different sub classes to the class `dynamicFvMesh`, based upon what the mesh should be able to do. Since `pimpleDyMFoam` primarily is developed for moving meshes, special attention will also be paid towards how this solver will treat a refined mesh and if there are missing features left to be implemented for this purpose. The `pimpleDyMFoam` solver is located at

```
$FOAM_APP/applications/solvers/incompressible/pimpleFoam/\
pimpleDyMFoam/
```

Furthermore, for reference, the `pimpleFoam` solver are located at

```
$FOAM_APP/applications/solvers/incompressible/pimpleFoam/
```

3.2.1 Comparison between pimpleFoam.C and pimpleDyMFoam.C

To begin with the differences in the definition files of the original `pimpleFoam` solver and the `pimpleDyMFoam` solver will be presented. The pimple-loop, in which both solvers solve the transport equations, are essentially equal. Some differences are present to handle mesh movement in the `pimpleDyMFoam` case but otherwise they are the same. The major differences are instead present before the pimple-loop. Everything before the pimple-loop, excluding the header, is presented for the `pimpleFoam.C` file below.

```

#include "fvCFD.H"
#include "singlePhaseTransportModel.H"
#include "turbulenceModel.H"
#include "pimpleControl.H"
#include "fvIOoptionList.H"
#include "IOporosityModelList.H"
#include "IOMRFZoneList.H"

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

int main(int argc, char *argv[])
{
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "createFields.H"
    #include "createFvOptions.H"
    #include "initContinuityErrs.H"

    pimpleControl pimple(mesh);

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

    Info<< "\nStarting time loop\n" << endl;

    while (runTime.run())
    {
        #include "readTimeControls.H"
        #include "CourantNo.H"
        #include "setDeltaT.H"

        runTime++;

        Info<< "Time = " << runTime.timeName() << nl << endl;
```

Listing 3.13: file: `pimpleFoam.C`

The same content for the `pimpleDyMFoam.C` file is presented below

```

#include "fvCFD.H"
#include "singlePhaseTransportModel.H"
#include "turbulenceModel.H"
#include "dynamicFvMesh.H"
#include "pimpleControl.H"
#include "fvIOoptionList.H"

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

int main(int argc, char *argv[])
{
    #include "setRootCase.H"

    #include "createTime.H"
    #include "createDynamicFvMesh.H"
    #include "initContinuityErrs.H"
    #include "createFields.H"
    #include "createFvOptions.H"
    #include "readTimeControls.H"

    pimpleControl pimple(mesh);

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

    Info<< "\nStarting time loop\n" << endl;

    while (runTime.run())
    {
        #include "readControls.H"
        #include "CourantNo.H"

        // Make the fluxes absolute
        fvc::makeAbsolute(phi, U);

        #include "setDeltaT.H"

        runTime++;

        Info<< "Time = " << runTime.timeName() << nl << endl;

        mesh.update();

        if (mesh.changing() && correctPhi)
        {
            #include "correctPhi.H"
        }

```

```
82        // Make the fluxes relative to the mesh motion
83        fvc::makeRelative(phi, U);
84
85        if (mesh.changing() && checkMeshCourantNo)
86        {
87            #include "meshCourantNo.H"
88        }
```

Listing 3.14: file: `pimpleDyMFoam.C`

The following apparent differences can be found between the two files

1. The inclusion of the file `dynamicFvMesh.H` in the `pimpleDyMFoam` solver, which is not present in the `pimpleFoam` solver.
2. The inclusion if the file `createMesh.H` in `pimpleFoam` is changed to `createDynamicFvMesh.H` in `pimpleDyMFoam`.
3. The two declaration files `IOporosityModelList.H` and `IOMRFZoneList.H` are not present in the `pimpleDyMFoam` solver.
4. The inclusion of the file `readTimeControls.H` in `pimpleFoam` has been changed to `readControls.H` in `pimpleDyMFoam`.
5. The finite volume calculus operation `fvc::makeAbsolute(phi,U)` is added in the `pimpleDyMFoam` solver.
6. The operation `mesh.update()` in the mesh is added in the `pimpleDyMFoam` solver.
7. The inclusion of the file `correctPhi.H` is done under some conditions in the `pimpleDyMFoam` solver.
8. The finite volume calculus operation `fvc::makeRelative(phi,U)` is added in the `pimpleDyMFoam` solver.
9. The inclusion of the file `meshCourantNo.H` is done under some conditions in the `pimpleDyMFoam` solver.

3.2.2 Mesh refinement in pimpleDyMFoam

In this section, the handling of dynamic meshes in `pimpleDyMFoam` will be presented. This includes the two changed include files `dynamicFvMesh.H` and `createDynamicFvMesh.H` as well as the steps the solver performs before the pimple loop starts, i.e. line 61 - 88 in Listing 3.14.

dynamicFvMesh.H This extra file is included to define the base class of dynamic meshes, `dynamicFvMesh`. It is included from

```
$FOAM_SRC/dynamicFvMesh/lnInclude/
```

It inherits the attributes of the class `fvMesh`, which is the class for non dynamic meshes. It in addition builds upon the `polyMesh` class and adds features needed for finite volume discretization. Hence, roughly speaking, the class `dynamicFvMesh` is an extension of the `fvMesh` class with added base features for dynamic meshes.

createDynamicFvMesh.H This file is used as a substitute to the `createFvMesh.H` file. It is also included from

```
$FOAM_SRC/dynamicFvMesh/lnInclude/
```

What it does is that it uses a function in the file `dynamicFvMeshNew.C`, located in the same directory, to create a mesh of the class specified in the `dynamicMeshDict` dictionary, located in the constant directory of the case. In the case of mesh refinement, this class will be called `dynamicRefineFvMesh`, for which refinement specific functions are defined.

readControls.H This file is included instead of the file `readTimeControls.H` and is located in the same directory as the solver. It is an extension of this file and reads

```
    #include "readTimeControls.H"

    const dictionary& pimpleDict = pimple.dict();

    const bool correctPhi =
        pimpleDict.lookupOrDefault("correctPhi", false);

    const bool checkMeshCourantNo =
        pimpleDict.lookupOrDefault("checkMeshCourantNo", false);

    const bool ddtPhiCorr =
        pimpleDict.lookupOrDefault("ddtPhiCorr", true);
```

Listing 3.15: file: `readControls.H`

The file `readTimeControls.H` is located at

```
$FOAM_SRC/finiteVolume/cfdTools/general/include/
```

It is used to look up time parameters in the `controlDict` dictionary, located in the system directory of the case. The first one is the boolean variable `adjustTimeStep`, which is set to false by default. The second one is the scalar `maxCo` which is used to specify the maximum Courant number and is set to 1 by default. The last one is the scalar `maxDeltaT`, used to specify the largest allowed time step in the simulation, and it is set to a large value by default.

For the purpose of dynamic meshes, the `readControls.H` file also includes three new boolean variables, namely `correctPhi`, `checkMeshCourantNo` and `ddtPhiCorr`. These variables are set in the `fvSolution` dictionary, within the section specifying the PIMPLE controls. The specific use of these variables will be discussed when they are used later in the code.

CourantNo.H This file is located at

```
$FOAM_SRC/finiteVolume/cfdTools/incompressible/
```

It is used to calculate and print out the mean and max Courant number based on the previously used time step. In case the time step is taken as constant, i.e. `adjustTimeStep` = `false`, the Courant number calculation only serves to inform the user of which Courant number the time step and mesh gives. The Courant number is furthermore computed in OpenFOAM according to

$$\mathrm{Co} = \frac{1}{2}\frac{\sum_f |\phi_f|}{\Delta V}\Delta t.$$

Here, $\phi_f = A_f(\mathbf{n}_f \cdot \mathbf{u}_f)$, is the velocity flux normal to surface f of the mesh control volume with volume ΔV. For a hexahedral mesh with straight edges, this formula will give the following formula for the Courant number

$$\text{Co} = \left(\frac{|u_x|}{\Delta x} + \frac{|u_y|}{\Delta y} + \frac{|u_z|}{\Delta z} \right) \Delta t.$$

This is a common form to express the Courant number, which according to the Courant-Friedrichs-Lewy condition generally should be less than 1 to obtain stable solutions to time marching problems.

fvc::makeAbsolute(phi, U) This function is defined in the file `fvcMeshPhi.C` that is located at

```
$FOAM_SRC/finiteVolume/lnInclude/
```

This operation adds the flux caused by the movement of the mesh to the flux across the mesh control volume boundaries. This gives the absolute flux relative to a fixed and non moving boundary, instead of the flux relative to the movement of the mesh control volume boundaries. This operation will also only be performed in the case where the mesh is moving, and hence for mesh refinement it will not be used.

setDeltaT.H This file is also found in

```
$FOAM_SRC/finiteVolume/lnInclude/
```

This routine sets the value for Δt to be used in the next time integration. It does this to satisfy the conditions that the maximum Courant number as well as time step, if specified in the `controlDict`, are not exceeded. It does this using the Courant number calculated recently, which is based on the current velocities and previous time step. Also note that in the case of a moving mesh, the fact that the fluxes have been made absolute already does not affect this routine since the Courant number was evaluated prior to the `makeAbsolute` routine was called. Denoting the maximum Courant number and time step set in the `controlDict` by `maxCo` and `maxDeltaT`, the calculated Courant number `CoNum`, and the previous time step $\Delta\tilde{t}$, the current time step, Δt, is according to the implementation calculated as

$$\begin{aligned}
\Delta F &= \frac{\texttt{maxCo}}{\texttt{CoNum}}, \\
\Delta \tilde{F} &= \min(\min(\Delta F, 1 + 0.1\Delta F), 1.2), \\
\Delta t &= \min(\Delta \tilde{F} \Delta \tilde{t}, \texttt{maxDeltaT}).
\end{aligned}$$

Hence, the new value Δt is set based on the old value $\Delta\tilde{t}$ multiplied with a scaling factor $\Delta\tilde{F}$. The second equation serves to relax this scaling factor in order to avoid too large increases in time steps and thereby unstable solutions. If the relaxation does not become active, the algorithm calculates Δt as the largest possible time step allowed, either by the Courant number condition or the maximum time step condition. Finally this routines prints out the new time step.

mesh.update() The function `update()` operates on the mesh depending on which subclass to `dynamicFvMesh` that it belongs to. In case of mesh refinement, this class is called `dynamicRefineFvMesh`, and the corresponding implementation of `update()` is found in the file `dynamicRefineFvMesh.C` located at

```
$FOAM_SRC/dynamicFvMesh/dynamicRefineFvMesh/
```

The function `update()` itself starts at line `1073` and in Figure 3.1 a simplified block scheme of the function is presented.

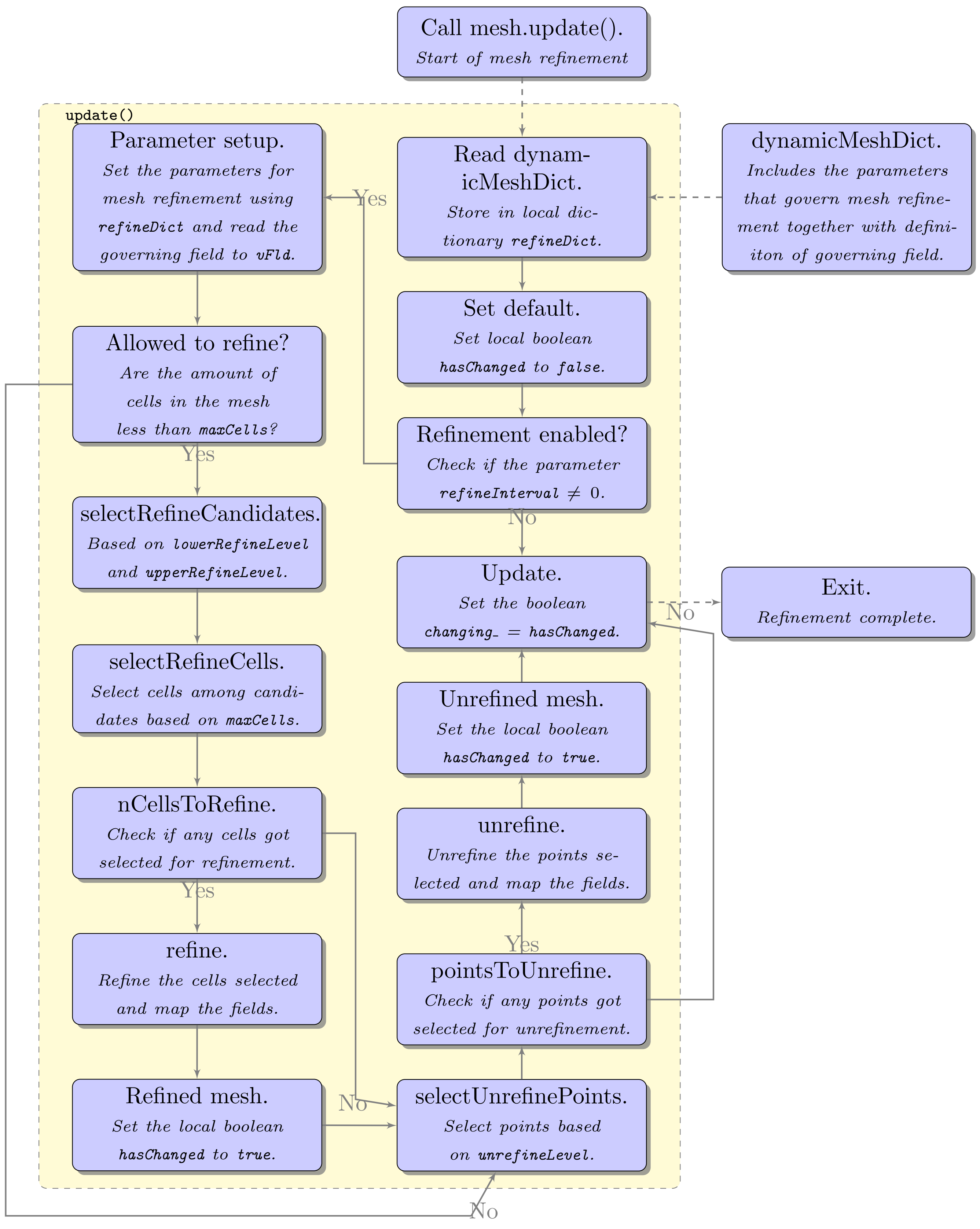

Call mesh.update().
Start of mesh refinement
update()
Parameter setup.
Set the parameters for mesh refinement using refineDict and read the governing field to vFld.
Read dynamicMeshDict.
Store in local dictionary refineDict.
dynamicMeshDict.
Includes the parameters that govern mesh refinement together with definiton of governing field.
Yes
Set default.
Set local boolean hasChanged to false.
Allowed to refine?
Are the amount of cells in the mesh less than maxCells?
Refinement enabled?
Check if the parameter refineInterval ≠ 0.
Yes
No
selectRefineCandidates.
Based on lowerRefineLevel and upperRefineLevel.
Update.
Set the boolean changing_ = hasChanged.
Exit.
Refinement complete.
No
selectRefineCells.
Select cells among candidates based on maxCells.
Unrefined mesh.
Set the local boolean hasChanged to true.
nCellsToRefine.
Check if any cells got selected for refinement.
unrefine.
Unrefine the points selected and map the fields.
Yes
Yes
refine.
Refine the cells selected and map the fields.
pointsToUnrefine.
Check if any points got selected for unrefinement.
Refined mesh.
Set the local boolean hasChanged to true.
No
selectUnrefinePoints.
Select points based on unrefineLevel.
No

Figure 3.1: Block scheme of mesh refinement in `update()`.

To begin with, it reads the dictionary `dynamicMeshDict` (`Read dynamicMeshDict`), located in the constant directory of the case. This dictionary specifies all the parameters that will govern the mesh refinement together with the name of the field that the refinement will be based on. The dictionary is stored in the local dictionary called `refineDict`.

The function uses a local boolean variable called `hasChanged`, which will be `true` if the mesh has been refined or unrefined. To begin with, no modifications to the mesh has been done and hence it is set to `false` by default (`Set default`).

After this the function will check if refinement is enabled or not (`Refinement enabled?`). This means that the parameter `refineInterval` in `refineDict` is checked. If this parameter is set to 0, the function will not enable mesh refinement, but instead go to `Update`. This enables the user to run `pimpleDyMFoam` without refinement, which can be useful if a converged solution is desired before enabling mesh refinement. If the parameter `refineInterval` is greater than 0, the function will enable mesh refinement/unrefinement with time step intervals specified by the parameter. In this case it will move on (`Parameter setup`).

The next step is simply to extract all parameters from the `refineDict` and create local variables (`Parameter setup`). In addition to this, the field that govern mesh refinement is read as well and put into the local field `vFld`. The field governs mesh refinement in the sense that it's values in every cell are used to determine if that cell should be refined/unrefined or not as we will see soon.

To avoid the mesh refinement to create too many cells, which could cause the memory or computational time to explode, there is a parameter `maxCells` specifying the maximum amount of cells the mesh may include. The function will therefore check the current amount of cells against `maxCells` before refining the mesh and creating more cells (`Allowed to refine?`). If the amount of cells are greater than `nCells`, the function will proceed to the unrefinement (`selectUnrefine`).

In case there is room for refinement, the function will proceed to select candidate cells for refinement (`selectRefineCandidates`). A cell will become a candidate if the value of `vFld` in cell i lies in between the defined limits, i.e.

$$\texttt{lowerRefineLevel} < \texttt{vFld}_i < \texttt{upperRefineLevel}.$$

This checking is done in the local function `error`, which computes the error of each cell i, defined as

$$\mathrm{err}_i = \min(\texttt{vFld}_i - \texttt{lowerRefineLevel}, \texttt{upperRefineLevel} - \texttt{vFld}_i).$$

If $\mathrm{err}_i \geq 0$ for a given cell, then that cell will be marked as a candidate for refinement. It is not hard to see that this calculation gives the criteria that the field value should lie within the specified limits. For a qualified cell, the value err_i represent the closest distance to either of the limits `lowerRefineLevel` or `upperRefineLevel`. This information is not used later on, but comments in the code suggest that it is intended to be used in later versions to do a better selection of cells to refine. For now however, a cell is either a candidate or not.

When the candidates for refinement have been selected, it is time to proceed and select the cells that will actually be refined among these candidates (`selectRefineCells`). There are two cases

Table 3.2: Parameters that can be set in the dynamicMeshDict.

Parameter	Description	Allowed values
`refineInterval`	Amount of time steps between refinement	$0 \leq$
`maxRefinement`	Maximum `cellLevel` for a cell that is refined	$0 \leq$
`maxCells`	Maximum amount of cells in mesh allowed	$0 <$
`field`	Name of field that govern refinement/unrefinement	"field name"
`lowerRefineLevel`	Lower limit of field value in a cell to allow refinement	$0 <$
`upperRefineLevel`	Upper limit of field value in a cell to allow refinement	$0 <$
`unrefineLevel`	Upper limit value of field value in a cell to allow unrefinement	≤ 0
`uBufferLayers`	Amount of buffer layers for unrefinement	< 0
`correctFluxes`	List of fluxes to be remapped on newly created faces	List of names
`dumpLevel`	Unknown boolean variable	`true/false`

that can emerge at this stage. In the first case, the number of cells after refinement of all the candidate cells will not exceed `maxCells`. In the second case, the amount of cells after refinement of all candidates cells would exceed `maxCells`, which is not allowed. To estimate the amount of cells after refinement, the function assumes that every refined cell is split into 8 new ones. This means that for every refined cell, the total amount of cells will increase by 7. Hence, the total amount of cells that can be refined, `nTotToRefine`, can be estimated as

$$\texttt{nTotToRefine} = \frac{\texttt{maxCells - nTotCells}}{7},$$

where `nTotCells` is used to denote the total amount of cells prior to refinement. Based on this estimation, the code then decides which of the two cases described above that exist. If the amount of candidates are less than `nTotToRefine`, we have the first case and vice versa if the amount of candidates are more. In the first case, the code allows, or marks, all candidates for refinement. In the second case it will simply select cells for refinement until the total amount becomes larger than `nTotToRefine`. Hence no intelligent selection is performed in this implementation, but as far as the author understands it, the foundation for better selection has been laid.

At this stage the function has a set of cells that will be refined and that satisfies all the listed criteria above. Before moving on to the refinement routine, the function will also do a trivial check to see if any cells got selected for refinement at all (`nCellsToRefine`). It could be the case that no cell was selected since the value of the field `vFld` did not match for any cell. If this is the case the function will move on to the unrefinement procedure.

In the case where cells got selected, the cells will be refined and the fields will be mapped onto the new mesh. This all happens in the routine called `refine`, which starts at line 205 in `dynamicRefineFvMesh.C`. The details of how the mapping is performed and how the mesh is split have not been investigated in any detail. The code however claims that the fields are mapped and that a new approximate flux is calculated at the newly created faces. This correction/mapping will only be done if the fluxes have been listed under `correctFluxes` in the `dynamicMeshDict`. Please note that many time integration algorithms, such as backward Euler or Crank Nicholson, use old values, not only those of the current time step, to integrate to the next time step. This means that `correctFluxes` needs to include these as well. How this is done will be shown in the tutorial later. If the fluxes are not recreated the function will still work and solver run, but the results will be redundant.

After refinement and mapping has been performed, the function will set the local boolean `hasChanged` to `true` (`Refined mesh`). It will then move on to the unrefinement.

The next thing that happens is that the function will select points to unrefine (`selectUnrefinePoints`). Of course, a cell in itself can not be unrefined, but rather a common corner of a set of cells can be removed to create a larger cell. The selection of points to remove is made among those corresponding to cells that have not been refined. In fact, the function also allows for the possibility to protect neighboring cells to cells that have been refined from being unrefined. This is possible to control through the parameter `nBufferLayers` in the `dynamicMeshDict`. This allows the user to specify how many layers of cells from those that have been refined that should be protected from unrefinement. The restoring points will now be selected for unrefinement based on the criteria that the (interpolated) value of the field `vFld` at point i satisfies

$$\texttt{pFld}_i < \texttt{unrefineLevel}.$$

The interpolation is done by taking the average value of the field `vFld` in the neighboring cells.

Before moving on to the mesh unrefinement, the function also checks if any points got selected for unrefinement (`pointsToUnrefine`).

If it turned out that some points qualified for unrefinement, the mesh will be unrefined using the routine `unrefine`. As before, the fields are also mapped and the fluxes are recreated approximately on the new faces. Also as before, the fluxes will only be recreated if they are listed under `correctFluxes` in the `dynamicMeshDict`.

Since the mesh has changed, the local boolean `hasChanged` will be set to `true` (`Unrefined mesh`). This is done since it may happen that the mesh only was unrefined.

The function concludes in the step that have been named `Update` here. In this step a function called `changing` is called with the boolean `hasChanged` as parameter. This function belongs to the class `polyMesh`, which all `dynamicFvMesh` classes are subclasses of. This function changes the boolean `changing_` in the `polyMesh` class to the value of the parameter supplied to the function. This boolean can thus be used, as will be seen later, to see if the mesh was changed this time step or not.

correctPhi.H This routine is located in the same directory as the solver, i.e. it is written specifically for this solver. As seen from Listing 3.14, it is only going to be active if the two booleans `mesh.changing()` and `correctPhi` are true. The second one we have already seen, it is set in the `fvSolution` dictionary and read by `readControls.H`. Hence, the user chooses whenever this routine will take effect or not, since the boolean `correctPhi` is set to false by default. The first boolean is returned from the member function `changing()`, whose definition can be found in the file `polyMesh.H` included from

```
$FOAM_SRC/OpenFOAM/lnInclude/
```

This file declares the class `polyMesh`, which is inherited by the class `fvMesh`, which in addition is inherited by the `dynamicFvMesh` class and its sub classes. The definition of the public member function `changing()` furthermore reads

```

        //- Is mesh changing (topology changing and/or moving)
        bool changing() const
        {
            return changing_;
        }
```

Listing 3.16: file: `polyMesh.H`

Hence, this function returns the boolean `changing_`, which was set to true if the mesh is refined, as noted previously.

In the file `correctPhi.H`, the pressure corrector equation of the PIMPLE algorithm is solved for the amount of times prescribed by the variable `nNonOrthogonalCorrectors`, set in the `fvSolution` dictionary. The part of the code that does this reads

```

        while (pimple.correctNonOrthogonal())
    {
        fvScalarMatrix pcorrEqn
        (
            fvm::laplacian(rAU, pcorr) == fvc::div(phi)
        );

        pcorrEqn.setReference(pRefCell, pRefValue);
        pcorrEqn.solve();

        if (pimple.finalNonOrthogonalIter())
        {
            phi -= pcorrEqn.flux();
        }
    }
```

Listing 3.17: file: `correctPhi.H`

The purpose of this is to obtain a so called pressure corrector that will be used to correct the fluxes over the mesh cells to obey continuity. The reason to why this is included prior to actually entering the PIMPLE loop further down the code is to compensate for the fact that the mesh may have changed. In a case where the mesh changes, the different fields need to be mapped from the old to the new mesh. This will naturally introduce some interpolation error, which for the case of face fluxes, can cause continuity to not be obeyed anymore on the new mesh. Hence, this part of the code offers the possibility to correct the fluxes to obey continuity before solving for the next time step. Speaking generally, when solving numerically for the next time step, the current time step comes in as a source term in the linear system of equations that is solved for. Therefore, since an error has been introduced in the mapping between the meshes, this error will continue to affect the solution in the next time step. Note that this argument is valid for any field that is solved for, i.e. not only the velocity field.

fvc::makeRelative(phi, U) This routine performs the opposite operation of `fvc::makeAbsolute(phi, U)`. It's defined in the file `fvcMeshPhi.C` found in

```
$FOAM_SRC/finiteVolume/lnInclude/
```

This function hence subtracts the flux of the mesh at every cell face, leaving the flux at the cell faces that is relative to the moving mesh. Since this routine only is used in cases where the mesh is moving, it is not used when mesh refinement is.

meshCourantNo.H The file `meshCourantNo.H` is included from

```
$FOAM_SRC/dynamicFvMesh/lnInclude/
```

As can be noted from Listing 3.14, this routine is used in the case where the mesh has changed (`mesh.changing()`) and if the boolean `checkMeshCourantNo` has been set to true. The second boolean we saw was set in `readControls.H`, where it was read from the `fvSolution` dictionary. The routine in `meshCourantNo.H` performs the exact same operations as the routine found in `CourantNo.H` considered previously but for the flux caused by the mesh. In other words, the flux over the cell faces caused by the mesh motion is used instead of the relative flux of the fluid over the cell faces. This means that the mesh Courant number is based on the velocity of the mesh rather than the fluid. It's use has not been further investigated, since the mesh is not moving in the refinement case and thus this routine should not be used.

3.2.3 Suggested modifications for mesh refinement

The `pimpleDyMFoam` solver is mainly developed to handle moving meshes but can also handle mesh refinement. Since it is originally written for moving meshes, there are some features that are missing and that in some way should be implemented in the future to obtain a better solver.

Smoother for turbulent quantities As noted when considering the include file `correctPhi.H`, the fluxes are corrected in order to satisfy continuity over the computational cells. This addresses the problem introduced by mapping the velocity/flux field but not the fact that turbulent quantities such as k, ε or ω have been mapped as well. As discussed before, the solution at the current time step affects the solution of the next time step and hence if the current solution have obtained a large error in the mapping procedure, this error will live on to the next time step. It is therefore suggested that a "smoothing" procedure is introduced after the velocity field has been corrected but before the PIMPLE loop start. This would constitute of a set of iterations in the solution of the turbulent equations in order to converge the solution of the turbulent quantities on the new mesh.

Adjusting Δt for the new mesh In cases where the Courant number is used to limit the value of Δt, it was noted above that this is done prior to updating the mesh. This means that the value of Δt is based on the Courant number obtained on the mesh prior to mesh refinement. If the mesh would be refined, come cells are split into smaller cells. In the current case, Δt, as well as the velocities u_x, u_y and u_z are fixed. According to the definition of the Courant number, this would mean that if for example Δx, Δy and Δz are divided by two, the Courant number for those cells would become twice as large. Hence, the way the solver operates right now will cause the actual Courant number to becomes larger in refined cells, which may cause instabilities. The solution would be to incorporate a recalculation of the Courant number after mesh refinement, as well updating Δt based on this. These routines should naturally only be active in the case of mesh refinement and be included after the velocity field/fluxes have been corrected. Note also that the update of the time through `runTime++` would have to be moved as well.

Chapter 4

Implementing $k - \omega$ SST DES

The first section of this chapter describes step by step how the turbulence model $k - \omega$ SST DES can be obtained by modifying the existing $k - \omega$ SST SAS model. The next section then describes how to further modify the turbulence model to produce a `volScalarField` that can control the mesh refinement.

4.1 $k - \omega$ SST DES

In this section a new source term in the k equation will be added, which which is a modification of the dissipation. Terms unique to the SAS model will also be taken away to avoid unnecessary term or constants being computed.

One thing that is important in order to achieve the correct turbulence model is to make sure that the different terms in the transport equations that are reused actually are implemented as described in the paper. This is due to that turbulence models often get modified over the years even though the actual name stays the same. This was done in a previous section, and this analysis will be the basis for the step by step guide below in how to implement the $k - \omega$ SST DES turbulence model.

4.1.1 Copying the kOmegaSSTSAS model

To begin with, the SAS model will be copied into the user directory. Start by opening a new terminal window and type

```
OF22x
```

to initialize the OpenFOAM 2.2.x environment. Next change directory to

```
cd $WM_PROJECT_DIR
```

Now copy the kOmegaSSTSAS turbulence model into the user directory according to

```
cp -r --parents src/turbulenceModels/incompressible/LES/\
                kOmegaSSTSAS/ $WM_PROJECT_USER_DIR
```

Finally change directory to where the turbulence model has been copied in the user directory according to

```
cd $WM_PROJECT_USER_DIR/src/turbulenceModels/incompressible/LES/
```

4.1.2 Creating the class kOmegaSSTDES

To create a new class, start by changing name of the directory according to

```
mv kOmegaSSTSAS/ kOmegaSSTDES/
```

Now continue by removing the .dep file inside the turbulence model directory and change name of the .C and .H file to an appropriate name

```
rm kOmegaSSTDES/kOmegaSSTSAS.dep
mv kOmegaSSTDES/kOmegaSSTSAS.C kOmegaSSTDES/kOmegaSSTDES.C
mv kOmegaSSTDES/kOmegaSSTSAS.H kOmegaSSTDES/kOmegaSSTDES.H
```

Now we need a Make directory where the make files will reside, it should lie in the same directory as the turbulence model directory. Create it and then go into it according to

```
mkdir Make/
cd Make/
```

Now we will create the necessary files `files` and `options`. Starting with `files`, it may be created as

```
gedit files
```

In the new empty file, we must add our new turbulence model to tell the compiler creating the dynamic turbulence model library where the new turbulence model should be included. Do this by including the lines

```
kOmegaSSTDES/kOmegaSSTDES.C

LIB = $(FOAM_USER_LIBBIN)/libmyIncompressibleLESModels
```

The content of `files` is similar to the one found in the installation. The difference is that the library should be put in `$(FOAM_USER_LIBBIN)` instead and the name should be changed from being `libIncompressibleLESModels` to something else.

Next the `options` file is created according to

```
gedit options
```

In the new file that opens, add the following lines

```
EXE_INC = \
    -I$(LIB_SRC)/turbulenceModels \
    -I$(LIB_SRC)/turbulenceModels/LES/LESdeltas/lnInclude \
    -I$(LIB_SRC)/turbulenceModels/LES/LESfilters/lnInclude \
    -I$(LIB_SRC)/transportModels \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/turbulenceModels/incompressible/LES/lnInclude

LIB_LIBS =
```

These lines are also found in the original `options` file in the installation except for the last include line. This one is needed since the turbulence model does not lie in the original directory anymore, but still needs files from there.

Now change directory to where the declaration and definition files are located

```
cd ../kOmegaSSTDES/
```

To change the name of the class, we must substitute the line ”kOmegaSSTSAS” to ”kOmegaSST-DES” in both the declaration file and the definition file. This is done according to

```
sed -i s/kOmegaSSTSAS/kOmegaSSTDES/g kOmegaSSTDES.H
sed -i s/kOmegaSSTSAS/kOmegaSSTDES/g kOmegaSSTDES.C
```

A good practice is to also look through the files and make sure that the name of the class has changed. Finish by first cleaning up the turbulence model library and then compile the turbulence model in order to see that everything works so far

```
cd ..
wclean
wmake libso
```

4.1.3 Modifying kOmegaSSTDES.H

We will start by modifying the file `kOmegaSSTDES.H`, so begin by changing directory and open it

```
cd $WM_PROJECT_USER_DIR/src/turbulenceModels/incompressible/LES/\
   kOmegaSSTDES/
gedit kOmegaSSTDES.H
```

The reader may of course choose another editor instead of gedit. We will start by adding a new header to our new file, so that it is clear which turbulence model we actually use, do this by changing the lines

```
Description
    kOmegaSSTDES LES turbulence model for incompressible flows
    based on:

    "Evaluation of the SST-SAS model: channel flow, asymmetric diffuser
    and axi-symmetric hill".
    European Conference on Computational Fluid Dynamics ECCOMAS CFD 2006.
    Lars Davidson


    The first term of the Qsas expression is corrected following:

    DESider A European Effort on Hybrid RANS-LES Modelling:
    Results of the European-Union Funded Project, 2004 - 2007
    (Notes on Numerical Fluid Mechanics and Multidisciplinary Design).
    Chapter 2, section 8 Formulation of the Scale-Adaptive Simulation (SAS)
    Model during the DESIDER Project. Published in Springer-Verlag Berlin
    Heidelberg 2009.
    F. R. Menter and Y. Egorov.
```

Listing 4.1: file: `kOmegaSSTDES.H`

to the following new description, including a small disclaimer

```
Description
    k-Omega-SST-DES LES turbulence model for incompressible flows
    based on:

    "Ten Years of Industrial Experience with the SST Turbulence Model".
    Turbulence Heat and Mass Transfer 4.
    F. R. Menter, M. Kuntz, R. Langtry.
    A note on the implementation with respect to the paper:

    - The transport equations implemented are divided by rho, as opposed to the
      presentation of them in the paper, since the flow is incompressible

    - There is a choice to use the model in DES (Detached Eddy Simulation) mode
      or DDES (Delayed Detached Eddy Simulations) mode. The latter protects the
      boundary layer from being resolved with LES and is used by default. The
      other part is commented away.

    - Model constants are not allways named the same as in the paper, see comments
      in declaration of constants below.

---------------------------------------------------------------------------
DISCLAIMER:
    This is a student project work in the course CFD with OpenSource software
    taught by Hakan Nilsson, Chalmers University of Technology, Gothenburg, Sweden.
---------------------------------------------------------------------------
```

Listing 4.2: file: `kOmegaSSTDES.H`

Now lets comment away unnecessary model constants that are unique to the SAS model. Start by commenting away the following lines

```
            dimensionedScalar Cs_;

            dimensionedScalar alphaPhi_;
            dimensionedScalar zetaTilda2_;
            dimensionedScalar FSAS_;
```

Listing 4.3: file: `kOmegaSSTDES.H`

to obtain

```
            // dimensionedScalar Cs_;

            // dimensionedScalar alphaPhi_;
            // dimensionedScalar zetaTilda2_;
            // dimensionedScalar FSAS_;
```

Listing 4.4: file: `kOmegaSSTDES.H`

After this, comment away the last two SAS model constants and also add the new constant C_{DES} as a `dimensionedScalar`. Do this by modifying the lines

```
            dimensionedScalar Cmu_;
            dimensionedScalar kappa_;
```

Listing 4.5: file: `kOmegaSSTDES.H`

to

```
            // dimensionedScalar Cmu_;
            // dimensionedScalar kappa_;

            dimensionedScalar CDES_;            // C_DES
```

Listing 4.6: file: `kOmegaSSTDES.H`

Now the new model includes the correct model constants in the class declaration. If the reader wants to, it can also add a comment after each model constant, like it has been done for CDES_, with the name of the constant in the paper. For a reference to these names please see Table 3.1.

The next thing to do is to comment away the protected member function called Lvk2. This is a function exclusive to the SAS model which computes the Von Karman length scale used to identify unsteadiness and trigger LES mode in the SAS model. After commenting, the code should read

```
        /*
        tmp<volScalarField> Lvk2
        (
            const volScalarField& S2
        ) const;
        */
```

Listing 4.7: file: kOmegaSSTDES.H

As noted earlier when studying the SAS implementation, it is convenient to use a function that creates the F_{DES} or F_{DDES} term. To give the user a choice, both terms will be added, but one of them will be commented away and hence not become active when compiling the dynamic library. As Menter implies that the DDES implementation with the boundary layer protector is the safest approach, it will be the one applied in this case [1]. Therefore add the following lines after the declaration of the blend function

```
        // Choose DES or DDES by commenting the non desired option

        // tmp<volScalarField> FDES() const;

        tmp<volScalarField> FDDES(const volScalarField& FS) const;
```

Listing 4.8: file: kOmegaSSTDES.H

If the reader would like to use DES model instead, it is merely a matter of switching the commenting or simply adding only the term preferred. As seen, the function is returning a volScalarField of type tmp, which is the preferred alternative when defining function that will be evaluated for every cell in the mesh and thus return large amount of data. As can be seen as well, the F_{DDES} term takes in the argument FS, which as stated in the theory can be chosen as either F_1 or F_2.

After this step the necessary changes to the kOmegaSSTDES.H file are done, next is the file kOmegaSSTDES.C.

4.1.4 Modifying kOmegaSSTDES.C

As seen previously, the definition file starts with defining the protected member functions. The first thing that therefore needs to be done is to remove the definition of the Von Karman length scale, Lvk2. Hence comment away its definition to obtain

```
/*
tmp<volScalarField> kOmegaSSTDES::Lvk2
(
    const volScalarField& S2
) const
{
    return max
    (
        kappa_*sqrt(S2)
```

```
            /(
                 mag(fvc::laplacian(U()))
               + dimensionedScalar
                 (
                     "ROOTVSMALL",
                     dimensionSet(0, -1 , -1, 0, 0, 0, 0),
                     ROOTVSMALL
                 )
             ),
             Cs_*delta()
        );
}
*/
```

Listing 4.9: file: `kOmegaSSTDES.C`

With this done, its time to define the F_{DES} and F_{DDES} terms. In this implementation, the DDES term is used and hence the other term should be commented away. This is achieved by adding the following lines directly after the `Lvk2` just commented away

```

// Choose DES or DDES by commenting the non desired option

// F_DES term definition
/*
tmp<volScalarField> kOmegaSSTDES::FDES() const
{
    return max
    (
        sqrt(k_)/(CDES_*betaStar_*omega_*delta()),
        scalar(1)
    );
}
*/

// F_DDES term definition, FS = F1 or F2 may be chosen as the
// boundary layer protector
tmp<volScalarField> kOmegaSSTDES::FDDES(const volScalarField& FS) const
{
    return max
    (
        sqrt(k_)/(CDES_*betaStar_*omega_*delta())*(scalar(1) - FS),
        scalar(1)
    );
}
```

Listing 4.10: file: `kOmegaSSTDES.C`

Next in the code comes the constructor of the class `kOmegaSSTDES`, and here we need to include our new model constant C_{DES} as well as remove constants no longer in use. The constants that should be removed are `Cs_`, `alphaPhi_`, `zetaTilda2_`, `FSAS_`, `Cmu_` and `kappa_` since they are all exclusive to the SAS model. Make sure to comment away them all, which for `Cs_` simply gives

```
/*
    Cs_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "Cs",
            coeffDict_,
            0.262
        )
    ),
    */
```

Listing 4.11: file: `kOmegaSSTDES.C`

After all SAS model constants have been commented away, the C_{DES} constant must be defined as well. Therefore, after the commented constant `kappa_`, add the following lines

```
    CDES_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "CDES",
            coeffDict_,
            0.61
        )
    ),
```

Listing 4.12: file: `kOmegaSSTDES.C`

With this accomplished, its time to add the DES functionality in the function `correct`, which is the one that solves the transport equations and updates the turbulent/sub grid scale viscosity. As the SAS features are also to be removed, these terms will be commented away. To begin with, the fields not used in the model should not be calculated prior to setting up and solving the discrete system of equations. Therefore comment away the calculation of the field denoted `L` according to

```
    // volScalarField L(sqrt(k_)/(pow025(Cmu_)*omega_));
```

Listing 4.13: file: `kOmegaSSTDES.C`

Next its time to calculate the field F_{DES} or F_{DDES} using the previously defined functions. This is done by adding the following lines after the production term `G` has been calculated and prior to solving for k according to

```
    // volScalarField FDES(this->FDES());
    volScalarField FDDES(this->FDDES(F1));
```

Listing 4.14: file: `kOmegaSSTDES.C`

Once again, note that it is the DDES features that are implemented here. Also note that it is crucial that the F_{DDES} is computed after the `F1` field, since it relies on it as a parameter. Next the transport equation for k is going to be modified so that the dissipation term includes the DES modification. This is done by adding the newly computed field `FDES` or `FDDES` into the first slot of the `Sp( , )` function creating the source term. The final result should look like

```
// Turbulent kinetic energy equation
    {
        fvScalarMatrix kEqn
        (
            fvm::ddt(k_)
          + fvm::div(phi(), k_)
          - fvm::laplacian(DkEff(F1), k_)
        ==
            min(G, c1_*betaStar_*k_*omega_)
          - fvm::Sp(betaStar_*FDDES*omega_, k_)          // F_DDES modification
          // - fvm::Sp(betaStar_*FDES*omega_, k_)        // F_DES modification
        );
```

Listing 4.15: file: `kOmegaSSTDES.C`

Note that it does not work to include the `FDES/FDDES` term in the second slot, i.e. multiplying it with `k_`.

After the k equation has been solved, a field unique to the SAS implementation is being calculated which is needed in a source term in the ω equation that is not used in the DES model. This field is called `grad_omega_k` and should be commented away according to

```
    /*
    tmp<volScalarField> grad_omega_k = max
    (
        magSqr(gradOmega)/sqr(omega_),
        magSqr(gradK)/sqr(k_)
    );
    */
```

Listing 4.16: file: `kOmegaSSTDES.C`

The next thing that needs to be done is to remove the source term in the ω equation that is unique to the SAS model. It is the last entity of the equation and should be commented away to get the following

```
    // Turbulent frequency equation
    {
        fvScalarMatrix omegaEqn
        (
            fvm::ddt(omega_)
          + fvm::div(phi(), omega_)
          - fvm::laplacian(DomegaEff(F1), omega_)
        ==
            gamma(F1)*S2
          - fvm::Sp(beta(F1)*omega_, omega_)
          - fvm::SuSp       // cross diffusion term
            (
                (F1 - scalar(1))*CDkOmega/omega_,
                omega_
            )
        /*
          + FSAS_
           *max
            (
                dimensionedScalar("zero",dimensionSet(0, 0, -2, 0, 0), 0.0),
                zetaTilda2_*kappa_*S2*sqr(L/Lvk2(S2))
              - 2.0/alphaPhi_*k_*grad_omega_k
            )
        */
        );
```

Listing 4.17: file: `kOmegaSSTDES.C`

Finally, the `read()` function should be modified in order to exclude SAS model constants and include the new DES model constant. The `read()` function is present at the end of the file and should be modified according to

```
bool kOmegaSSTDES::read()
{
    if (LESModel::read())
    {
        alphaK1_.readIfPresent(coeffDict());
        alphaK2_.readIfPresent(coeffDict());
        alphaOmega1_.readIfPresent(coeffDict());
        alphaOmega2_.readIfPresent(coeffDict());
        gamma1_.readIfPresent(coeffDict());
        gamma2_.readIfPresent(coeffDict());
        beta1_.readIfPresent(coeffDict());
        beta2_.readIfPresent(coeffDict());
        betaStar_.readIfPresent(coeffDict());
        a1_.readIfPresent(coeffDict());
        c1_.readIfPresent(coeffDict());
        // Cs_.readIfPresent(coeffDict());
        // alphaPhi_.readIfPresent(coeffDict());
        // zetaTilda2_.readIfPresent(coeffDict());
        // FSAS_.readIfPresent(coeffDict());
        CDES_.readIfPresent(coeffDict());

        omegaMin_.readIfPresent(*this);

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

Listing 4.18: file: `kOmegaSSTDES.C`

This concludes the modification of the turbulence model. The final thing to be done is to remove the `kOmegaSSTDES.dep` file and then recompile (`wclean` and then `wmake libso`). Therefore save and close the editor and then move into the directory where the `Make` and `kOmegaSSTDES` directories are situated. Then simply type

```
wclean
wmake libso
```

This should build a new dynamic library including the new turbulence model. It will then be possible to link to this library and run simulations using the new turbulence model.

4.2 $k - \omega$ SST DES Refine

This section shows how to modify the $k - \omega$ SST DES turbulence model implemented above to work with the dynamic mesh refinement features of OpenFOAM. As previously discussed, the mesh refinement needs a field to determine where the mesh should be refined and not. This field should be located in the time directory and hence must be computed, either by defining a function in the turbulence model or in the solver. This field will be a part of the F_{DES} term of the turbulence model and hence the creation and output of this field is what is going to be added to the turbulence model.

Before doing this, it should be stressed that the method shown here most certainly is not the only way to achieve this. The way it will be done is pretty straight forward, and follows the way the turbulent/sub grid scale viscosity is being computed and sent back. It was determined to not use the new field to further compute the F_{DES} term, even though it is a part of it. Instead this part of the F_{DES} term will be recomputed and returned, only to be used to govern mesh refinement. There are two reasons for this. The first is that the term includes the mesh size Δ, which only is defined in the node of a cell. Hence, at the boundary, there is no definition of what Δ should be and hence an appropriate boundary condition can be hard to derive. Since the term will reside in the time directory, some boundary conditions should be imposed and thus the solution would run the risk of being affected by these if this term was actually used to evaluate F_{DES}/F_{DDES}. Secondly, it was found more convenient to leave the previous implementation intact and just add new features.

4.2.1 Creating the class kOmegaSSTDESRefine

This entire guide assumes that the $k-\omega$ SST DES turbulence model already has been implemented according to above. Start by opening a new terminal window and type

```
OF22x
```

to initialize the OpenFOAM environment. Next change directory to where the user turbulence models are implemented

```
cd $WM_PROJECT_USER_DIR/src/turbulenceModels/incompressible/LES/
```

Copy the kOmegaSSTDES turbulence model directory into a new directory called kOmegaSST-DESRefine according to

```
cp -r kOmegaSSTDES/ kOmegaSSTDESRefine/
```

Change directory to the new turbulence model, remove the old `kOmegaSSTDES.dep` file and change name of the two remaining files according to

```
cd kOmegaSSTDESRefine/
rm kOmegaSSTDES.dep
mv kOmegaSSTDES.C kOmegaSSTDESRefine.C
mv kOmegaSSTDES.H kOmegaSSTDESRefine.H
```

Then substitute the line `kOmegaSSTDES` for `kOmegaSSTDESRefine` in both the files, which will change the name of the class.

```
sed -i s/kOmegaSSTDES/kOmegaSSTDESRefine/g kOmegaSSTDESRefine.H
sed -i s/kOmegaSSTDES/kOmegaSSTDESRefine/g kOmegaSSTDESRefine.C
```

Finally the model will be added to the `files` file, in order to tell the compiler that a new turbulence model should be added to the dynamic library. Open it using for example `gedit`

```
cd ..
gedit Make/files
```

and add the following line after the previous turbulence model

```
kOmegaSSTDESRefine/kOmegaSSTDESRefine.C
```

Finally recompile the dynamic library and make sure that everything works

```
wclean
wmake libso
```

4.2.2 Modifying kOmegaSSTDESRefine.H

We will start by going back into the directory where the new turbulence is situated and open it

```
cd $WM_PROJECT_USER_DIR/src/turbulenceModels/incompressible/LES/\
   kOmegaSSTDESRefine/
gedit kOmegaSSTDESRefine.H
```

As for the previous model, add a small description and disclaimer in the beginning of the file, instead of the previous one according to

```
Description
    k-Omega-SST-DES LES turbulence model for incompressible flows
    based on:

    "Ten Years of Industrial Experience with the SST Turbulence Model".
    Turbulence Heat and Mass Transfer 4.
    F. R. Menter, M. Kuntz, R. Langtry.

    A note on the implementation with respect to the paper:

    - The transport equations implemented are divided by rho, as opposed to the
      presentation of them in the paper, since the flow is incompressible

    - There is a choice to use the model in DES (Detached Eddy Simulation) mode
      or DDES (Delayed Detached Eddy Simulations) mode. The latter protects the
      boundary layer from being resolved with LES and is used by default. The
      other part is commented away.

    - Model constants are not allways named the same as in the paper, see comments
      in declaration of constants below.

    - A extra volScalarField called LDES or LDDES (dependent on which turbulence
      model that is used) is calculated. It is not influencing the solution, but
      used solely as a field to indicate where the mesh needs refinement in order
      to resolve turbulence. The same quantity is used to calculate the F_DES
      or F_DDES term in the k equation, but this is done using a different routine.
      LDES = L_t/(C_DES*delta), LDDES = L_t/(C_DES*delta)*(1 - F_S)

------------------------------------------------------------------------------
DISCLAIMER:
    This is a student project work in the course CFD with OpenSource software
    taught by Hakan Nilsson, Chalmers University of Technology, Gothenburg, Sweden.
------------------------------------------------------------------------------
```

Listing 4.19: file: `kOmegaSSTDESRefine.H`

Turning to the class declaration, a new `void` function calculating the term presented in (2.18) or (2.19) depending on turbulence model will be added. This will resemble the already existing function `updateSubGridScaleFields` which calculates the turbulent/sub grid scale viscosity. The new field will be denoted `LDES_` or `LDDES_`, where the first `L` is used to denote that it is a measure between two different length scales, the turbulent one and the mesh. Add the following lines after the declaration of the function `updateSubGridScaleFields`

```
        // Calculate the LDES/LDDES fields
        // void updateLDES();
        void updateLDDES(const volScalarField& FS);
```

Listing 4.20: file: `kOmegaSSTDESRefine.H`

As for previous implementations, the DDES formulation is implemented and the DES formulation is commented away.

Since a new field is introduced, it must also be declared. Therefore, add a declaration of the new field `LDES_` or `LDDES_` after the declaration of the `k_`, `omega_` and `nuSgs_` fields and in front of the commented declaration of `Lvk2` according to

```
        // Extra field to indicate mesh refinement

            // volScalarField LDES_;
            volScalarField LDDES_;
```

Listing 4.21: file: `kOmegaSSTDESRefine.H`

This concludes the modifications needed to the declaration file.

4.2.3 Modifying kOmegaSSTDESRefine.C

To begin with, we will add the definition of the protected member function that computes `LDES_` or `LDDES_` defined in (2.18) and (2.19) respectively. To do this, add the following lines after the definition of `updateSubGridScaleFields`

```
/*
void kOmegaSSTDESRefine::updateLDES()
{
    LDES_ == sqrt(k_)/(CDES_*betaStar_*omega_*delta());
    LDES_.correctBoundaryConditions();
}
*/

void kOmegaSSTDESRefine::updateLDDES(const volScalarField& FS)
{
    LDDES_ == sqrt(k_)/(CDES_*betaStar_*omega_*delta())*(scalar(1) - FS);
    LDDES_.correctBoundaryConditions();
}
```

Listing 4.22: file: `kOmegaSSTDESRefine.C`

Now that the function is defined, we must make sure that the new field, or object, is added in the construction of the class `kOmegaSSTDESRefine`. Therefore, add the construction of the fields `LDES_` and `LDDES_` after the construction of `nuSgs_` field. Also, make sure to add an extra comma after the construction of `nuSgs_` to obtain

```
    nuSgs_
    (
        IOobject
        (
            "nuSgs",
            runTime_.timeName(),
            mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_
    ),          // Don't forget this comma!

    /*
    LDES_
    (
        IOobject
        (
            "LDES",
            runTime_.timeName(),
            mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_
    )
    */

    LDDES_
    (
        IOobject
        (
            "LDDES",
            runTime_.timeName(),
            mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_
    )
```

Listing 4.23: file: `kOmegaSSTDESRefine.C`

As noted when `kOmegaSSTSAS.C` was considered, the turbulent/sub grid scale viscosity is computed inside the body of the constructor. This could of course be done for `LDES_` or `LDDES_` as well but there is no point in doing this. The reason is that the field is never used, only updated after k and ω have been solved for. Hence, it does not need to be calculated prior to solving for k and ω in contrast to the viscosity, which is needed for this purpose.

The last modification is therefore to make sure that the new field is calculated using the newly calculated fields k and ω. For this purpose, add the following lines at the very end of the `correct` function after both k and ω have been solved for and the turbulent/sub grid scale viscosity have been updated using the function `updateSubGridScaleFields`

```
    // Update the LDES/LDDES term
    // updateLDES();
    updateLDDES(F1);
```

Listing 4.24: file: `kOmegaSSTDESRefine.C`

This concludes the creation of the `kOmegaSSTDESRefine` model. After saving and closing, remove the `kOmegaSSTDESRefine.dep` file and then go to the directory containing the turbulence models and the Make directory and type

```
wclean
wmake libso
```

This will rebuild the dynamic library containing the user defined turbulence models to include the new turbulence model `kOmegaSSTDESRefine`. Since this model will need a new field in the 0/ directory of a case that uses it, please refer to the tutorial section for a guide on how to add this as well.

Chapter 5

Simulation with mesh refinement

This chapter describes how to set up a case to run airfoil simulations with mesh refinement. It requires that the reader has implemented the $k-\omega$ SST DES turbulence model for mesh refinement (`kOmegaSSTDESRefine`) according to the previous chapter. The case will be set up so that the solution can run in either 2 or 3D and with only the `pimpleDyMFoam` solver. For this purpose, a 2D mesh of a NACA 4412 airfoil, that is prepared for 3D simulations, have been supplied. At the end, the results of the simulation, both with respect to how the new turbulence model performs and to how the mesh refinement works, will be presented and briefly discussed.

5.1 Copying files

To begin with, a new OpenFOAM case will be created. Begin by creating a new case by copying an existing LES tutorial according to

```
run
cp -r $FOAM_TUTORIALS/incompressible/pisoFoam/les/pitzDaily .
mv pitzDaily Airfoil4412Refine
cd Airfoil4412Refine
```

Before modifying anything, we will also need a `dynamicMeshDict` for the `pimpleDyMFoam` solver. The `interDyMFoam` solver in OpenFOAM utilizes mesh refinement and hence a dictionary from one of the `interDyMFoam` tutorials will be copied into the constant directory.

```
cp $FOAM_TUTORIALS/multiphase/interDyMFoam/ras/damBreakWithObstacle/\
constant/dynamicMeshDict constant/
```

Proceed by removing the old `blockMeshDict` and the boundary file according to

```
rm constant/polyMesh/blockMeshDict
rm constant/polyMesh/boundary
```

In the supplied files to this work, a file called `blockMeshDict_4412_2D3D` has been supplied. Copy this `blockMeshDict` into the `constant/polyMesh/` directory. When this is done, rename it as well by typing

```
mv constant/polyMesh/blockMeshDict_4412_2D3D \
constant/polyMesh/blockMeshDict
```

This `blockMeshDict` will by default give a 2D mesh (one cell in third direction) when running `blockMesh`. However, the front and back patches of the mesh are not of type empty, but instead of type cyclic. Hence, cyclic boundary conditions will also be prescribed on all flow fields, but if only one cell is present in the third direction the simulation will effectively be 2D. The reason for this is to allow the user to choose either to tun the case 2D and save simulation time, or full 3D to allow for more realistic turbulence to be resolved. To obtain a 3D mesh, the 2D mesh needs to be extruded. This can be accomplished with the utility `extrudeMesh` in OpenFOAM, which needs a dictionary. This dictionary is acquired by typing

```
cp $FOAM_APP/utilities/mesh/generation/extrude/extrudeMesh/\
extrudeMeshDict system/
```

This concludes the copying of all necessary files for the case. These will now be modified in the upcoming sections.

5.2 Creating the mesh

In this section, the mesh will be created and if desired, also extruded to 3D. Start by letting `blockMesh` creating the 2D mesh by typing

```
blockMesh
```

To be able to extrude the mesh to 3D, we will have to modify the `extrudeMeshDict` according to our mesh. Open the file with a preferred editor and make the following changes.

1. Comment away `constructFrom patch` and uncomment `constructFrom mesh` to obtain

```
constructFrom mesh;
//constructFrom patch;
//constructFrom surface;
```

2. Change the source case of the mesh that will be extruded to the current case to obtain

```
sourceCase "../Airfoil4412Refine";
```

3. Tell the `extrudeMesh` utility that it is the front patch that should be extruded. This will effectively just add a set of layers in the third direction, giving a 3D mesh. It is the `sourcePatches` and `exposedPatchName` that should be changed to `front` and `back` respectively to obtain

```
sourcePatches (front);
// If construct from patch: patch to use for back (can be same as sourcePatch)
exposedPatchName back;
```

4. The mesh will be extruded linearly in the third direction, therefore uncomment the `linearNormal` option for `extrudeModel`, and comment the other options (`wedge`). The changes should look like

```
//- Linear extrusion in point-normal direction
extrudeModel        linearNormal;
```

```
//- Wedge extrusion. If nLayers is 1 assumes symmetry around plane.
//extrudeModel        wedge;
```

5. The amount of layers added on the front is specified by `nLayers`. The new mesh will therefore contain `nLayers` + 1 cells in the third direction after extrusion. Specify this parameter to obtain the desired amount of cells, the author chose 10, according to

```
nLayers             10;
```

6. Set the expansion ratio for the new layers added to 1.0 according to

   ```
   expansionRatio      1.0;
   ```

7. There are a lot of different ways `extrudeMesh` can operate on the mesh, requiring a lot of different coefficients to be specified for different cases. This extrusion only needs the coefficient specified within `linearNormalCoeffs`. Therefore, comment away all other "...`Coeffs`" sub dictionaries and make sure that `linearNormalCoeffs` is uncommented according to

   ```
   linearNormalCoeffs
   {
       thickness       0.1;
   }
   ```

   Here, the thickness of the added layer was changed to 0.1 as well. The thickness refers to the total thickness of all the new layers added, and hence the thickness of one new layer is in the case of no expansion ration `thickness/nLayers`. The original mesh have a thickness of 0.01, which will hence be obtained for the added cells here as well.

When this is done, a 3D mesh can be obtained by typing

```
extrudeMesh
```

5.3 The 0/ directory

After the mesh has been created, it's time to specify proper boundary and initial conditions for the velocity, pressure and turbulent quantities. As the mesh is created, the wing is oriented with it's leading edge in the positive x direction and it's low pressure side in the positive y direction as shown in Figure 2.1. The boundary conditions in Table 2.1 together with cyclic boundary conditions connecting the front and back patch will be implemented. Note that cyclic boundary conditions must be specified in the `blockMeshDict` as well, i.e. it must be specified that these patches are going to be connected through cyclic boundary conditions.

To begin with we will change the name of some of the patches in the existing files to fit our mesh. Therefore type the following when standing inside the case directory.

```
sed -i s/lowerWall/bottom/g 0/U
sed -i s/upperWall/top/g 0/U
sed -i s/lowerWall/bottom/g 0/p
sed -i s/upperWall/top/g 0/p
sed -i s/lowerWall/bottom/g 0/k
sed -i s/upperWall/top/g 0/k
sed -i s/lowerWall/bottom/g 0/nuSgs
sed -i s/upperWall/top/g 0/nuSgs
sed -i s/lowerWall/bottom/g 0/nuTilda
sed -i s/upperWall/top/g 0/nuTilda
```

5.3.1 Velocity

The velocity is specified in the file 0/U. The velocity will be set to 1 m/s using the freestream boundary conditions. Begin by changing the initial value of the velocity to the following

```
internalField   uniform (-1 0 0);
```

The inlet, outlet, top and bottom patch should all have the same boundary conditions, according to

```
    inlet
    {
        type            freestream;
        freestreamValue uniform (-1 0 0);
    }
```

After this, it is time to add the wing patch, at which a no slip boundary condition will be applied. Therefore, add the following lines after the last of the previous four boundary conditions.

```
    wing
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }
```

Finally, it is time to add the front and back patch, for which cyclic boundary conditions will be applied. This is done by first removing the patch called `frontAndBack` and adding the following lines

```
    front
    {
        type            cyclic;
    }

    back
    {
        type            cyclic;
    }
```

Finish by saving and closing the velocity file.

5.3.2 Pressure

The pressure is specified in the file 0/p. The pressure will be initialized to zero everywhere, and no modification is therefore needed to the initial conditions. Continue by changing the boundary conditions at the inlet, outlet, top and bottom to freestreamPressure, according to

```
    inlet
    {
        type            freestreamPressure;
    }
```

The boundary condition for the wing is set to zero gradient by adding this patch

```
    wing
    {
        type            zeroGradient;
    }
```

Finally the boundary cyclic boundary conditions needs to be added. Do this by removing the `frontAndBack` boundary condition and add the same lines as for the velocity. Finish by saving and closing the file.

5.3.3 Turbulent kinetic energy

The turbulent kinetic energy conditions are specified in 0/k. Begin by changing the initial condition to a very small number according to

```
internalField   uniform 1e-06;
```

Proceed by setting the initial condition at the inlet, top and bottom according to

```
    inlet
    {
        type            fixedValue;
        value           uniform 1e-06;
    }
```

The outlet should be of type `zeroGradient`, which means that it should be changed to

```
    outlet
    {
        type            zeroGradient;
    }
```

At the wing the turbulence is in theory zero, but for numerical stability we will avoid this and just set it to something very small. Therefore add the following lines to specify the wing

```
    wing
    {
        type            fixedValue;
        value           uniform 1e-10;  // Avoid zero
    }
```

Proceed by removing the `frontAndBack` patch boundary conditions and add the same lines as for the previous quantities to achieve cyclic boundary conditions over the front and back patch. Save and close the file.

5.3.4 Turbulent frequency

There is no file for the turbulent frequency, instead we will change the name of the file `0/nuTilda`. Begin by changing it's name

```
mv 0/nuTilda 0/omega
```

The first modification that will be done to the file is to change the name of the object to `omega`, giving the following

```
    object      omega;
```

Proceed by changing the unit of the field to $1/s$ and the initial value of the field to 1 according to

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

internalField   uniform 1;
```

After this the boundary condition at the inlet, top and bottom should be changed to look like

```
    inlet
    {
        type            fixedValue;
        value           uniform 1;
    }
```

The outlet should be `zeroGradient`. Therefore, change this boundary condition to the same as for the turbulent kinetic energy. After this, the wing will be treated with a special wall function for the turbulent frequency. To use this add the following lines for the wing patch

```
    wing
    {
        type            omegaWallFunction;
        Cmu             0.09;
        kappa           0.41;
        E               9.8;
        beta1           0.075;
        value           uniform 1000;
    }
```

Finally, remove the boundary condition for `frontAndBack` and add the cyclic boundary conditions in accordance with before. Save and close the file.

5.3.5 Turbulent/Sub grid scale viscosity

The turbulent/sub grid scale viscosity conditions is specified in the file 0/nuSgs. Begin by setting the initial value to `1e-6`, corresponding to k/ω.

```
internalField   uniform 1e-6;
```

Continue by changing the boundary conditions at the inlet, outlet, top and bottom to calculated according to

```
    inlet
    {
        type            calculated;
        value           uniform 0;
    }
```

At the wing, no turbulence is present, therefore add the following lines to prescribe the boundary condition at the wing

```
    wing
    {
        type            fixedValue;
        value           uniform 0;
    }
```

As always, finish by removing the `frontAndBack` patch boundary conditions and add those to achieve cyclic conditions. After this save and close the file.

5.3.6 LDES/LDDES

A field governing mesh refinement is also needed. It is called LDES or LDDES dependent on if the DES or DDES model is used as described earlier. In this case, it is assumed that the DDES model is used and hence the field LDDES will be added. To achieve this copy the nuSgs file, then change the name and dimension of the object according to

```
cp 0/nuSgs 0/LDDES
sed -i s/nuSgs/LDDES/g 0/LDDES
sed -i s/"0 2 -1"/"0 0 0"/g 0/LDDES
```

After opening the file change the boundary condition at the inlet, outlet, top and bottom to `zeroGradient` to achieve

```
    inlet
    {
        type            zeroGradient;
    }
```

The zeroGradient condition is used because at a patch, there is no definition of the cell size Δ. Hence, it is better to simply assume that it varies little in the normal direction to the patch instead. Also, this field is not used by the turbulence model as discussed earlier, so it's value at a boundary is not important in that sense. At the wing, we however know that this field asymptotically must go to zero, due to that the turbulent kinetic energy does. Hence, it is safe to keep the same boundary conditions as for the turbulent/sub grid scale viscosity.

5.4 The dictionaries

The next step is to set up the necessary dictionaries. The dictionaries that will modified are `controlDict`, `fvSolution`, `fvSchemes`, `LESProperties` and `dynamicMeshDict`.

5.4.1 controlDict

We start by changing the name of the solver according to

```
sed -i s/pisoFoam/pimpleDyMFoam/g system/controlDict
```

The end time must also be modified. Before enabling mesh refinement, the flow must develop properly since we are dealing with a transient simulation. The wing is 1 m long the the flow is passing it at 1 m/s, so the end time will be chosen to 3 s in order to let the flow pass the wing several times. The change is done according to

```
sed -i s/0.1/3/g system/controlDict
```

Further, change the Δt value to $1 \cdot 10^{-3}$

```
sed -i s/1e-05/1e-03/g system/controlDict
```

The write control should be set to adjustable, since the Δt value will be governed by the Courant number, hence do the following change

```
sed -i s/timeStep/adjustableRunTime/g system/controlDict
```

The time intervals for which the solver will write out the results will be set next. The reader can of course choose this as they wish, for now it will be set to 0.2 s according to

```
sed -i s/100/0.2/g system/controlDict
```

Let's also change the `purgeWrite` option to only keep the latest solutions written and overwrite older. Here, this number will be changed to 5

```
sed -i s/"purgeWrite      0"/"purgeWrite      5"/g system/controlDict
```

To save some space on the disc, especially in case of a 3D simulation in which the amount cells quickly become very large, write compression will be enabled

```
sed -i s/off/compressed/g system/controlDict
```

After this, open the file in an editor and add the following lines before the `functions`

```
adjustTimeStep  yes;

maxCo           0.4;

libs ("libmyIncompressibleLESModels.so");
```

This will enable run-time Δt adjustment based on the Courant number, and also include the $k-\omega$ SST DES turbulence model. Furthermore, since the specified functions in the `controlDict` will not be used, they can be commented away as well. Finish by saving and closing the file.

5.4.2 fvSolution

Since the copied case uses the `pisoFoam` solver and solves for other turbulent quantities this dictionary needs some changes as well. Start by changing the name of the solver

```
sed -i s/PISO/PIMPLE/g system/fvSolution
```

Also change the name of the turbulent quantity used according to

```
sed -i s/nuTilda/omega/g system/fvSolution
```

After this, open the file in an editor. To begin with, change the PIMPLE sub dictionary to include the following

```
    nOuterCorrectors 1;
    nCorrectors     2;
    nNonOrthogonalCorrectors 2;
    pRefCell        0;
    pRefValue       0;
    correctPhi      yes;
```

These numbers can not completely be justified by the author. However, some general ideas to why these settings are applied can be given. First, setting the amount of outer correctors to 1 means that the solver will effectively operate in PISO mode, since the amount of times the whole PISO loop will be done are only 1. This seems reasonable considering that the Courant number is chosen small. Most tutorials available on PISO and PIMPLE use two correctors specified on the second line, and hence it was simply kept this way. The third line is included for two reasons. First, since the mesh is very skewed, adding non orthogonal correctors will help with the ”non orthogonal” cells present. Without it, continuity will run the risk of not being properly satisfied, which of course affect the entire solution negatively. Also, it will be used to correct for continuity after mesh refinement has been used, as discussed earlier.

After the specification of the solver used for `U`, add the following

```
    UFinal
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-05;
        relTol          0;
    }
```

I.e. simply copy the specifications for the `U` solver and change the keyword to `UFinal` instead. Do the same procedure for the quantities `k`, `B` and `omega` as well.

After this, we must define the type of solver that will be used for the pressure corrector equation entered in `correctPhi.H` in cases the mesh was refined. This is due to that a separate pressure corrector equation that solves for the quantity `pcorr` is used. It may be defined by copying the definition of the solver for `p` and renaming it `pcorr` to achieve

```
    pcorr
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0.05;
    }
```

When this has been done, the type of solver used for pressure will also be used since the current solver settings will fail to converge. Therefore, change all the pressure solver settings (`p`, `pFinal` and `pcorr`) to the following settings instead

```
        solver          PCG;
        preconditioner
        {
            preconditioner      GAMG;
            tolerance           1e-5;
            relTol              0;
            smoother            DICGaussSeidel;
            nPreeSweeps         0;
            nPostSweeps         2;
            nFinestSweeps       2;
            cacheAgglomeration  false;
            nCellsInCoarsestLevel       10;
            agglomerator        faceAreaPair;
            mergeLevels         1;
        }
        tolerance       1e-05;
        relTol          0;
        maxIter         100;
```

The author however chose to keep the `relTol` to 0.05 for the solution if `p`, a choice that however can not be justified.

5.4.3 fvSchemes

Some small changes to the schemes used, as well as addition of a few more necessary definitions of schemes that are missing will be done. Begin by simply changing the name of the turbulent quantity used according to

```
sed -i s/nuTilda/omega/g system/fvSchemes
```

Furthermore, the Crank Nicholson time integration scheme will be applied, hence perform the following change as well

```
sed -i s/backward/"CrankNicolson   0.5"/g system/fvSchemes
```

Next, open the file and add the following Laplacian scheme used in the pressure corrector equation that will be entered when `nNonOrthogonalCorrectors` is different from zero

```
    laplacian(rAU,p) Gauss linear corrected;
    laplacian(rAU,pcorr) Gauss linear corrected;
```

Finally, add the following line in the sub dictionary called `fluxRequired`

```
    pcorr           ;
```

Save and close the file.

5.4.4 LESProperties

To begin with, change the name of the turbulence model applied according to

```
sed -i s/oneEqEddy/kOmegaSSTDESRefine/g constant/LESProperties
```

After this, the way the local grid size is calculated needs to be changed. From the theory, we known that $\Delta = \max\{\Delta x_1, \Delta x_2, \Delta x_3\}$, and this should be specified in `LESProperties`. After opening the file in an editor, change the type of delta to `maxDeltaxyz` to obtain

```
delta           maxDeltaxyz;
```

Also, change the name of the sub dictionary for the `cubeRootVolCoeffs` to `maxDeltaxyzCoeffs`. Finally, a sub dictionary specifying all the constants that are used in the turbulence model will be supplied. This is not necessary, the values suggested in the literature are already set by default, but it is always good to have the opportunity to easily change them. Therefore, add the following lines in the dictionary

```
kOmegaSSTDESRefineCoeffs
{
    gamma1              0.5532;
    gamma2              0.4403;
    beta1               0.075;
    beta2               0.0828;
    alphaK1             0.85034;
    alphaK2             1.0;
    alphaOmega1         0.5;
    alphaOmega2         0.85616;
    betaStar            0.09;
    a1                  0.31;
    CDES                0.61;
}
```

This concludes the modifications of `LESProperties`, save and close the file.

5.4.5 dynamicMeshDict

Finally, the `dynamicMeshDict` needs to be adapted for this case. Since the solver will be started without enabling mesh refinement, set the `refineInterval` to 0 according to

```
sed -i s/"refineInterval  1"/"refineInterval  0"/g \
constant/dynamicMeshDict
```

The name of the field that govern the refinement needs to be changed as well.

```
sed -i s/alpha1/LDDES/g constant/dynamicMeshDict
```

Note that dependent on if the DES or DDES model is used, the field changes name. Since the LDES/LDDES term must become larger than 1 in order for the DES features to kick in, it is reasonable to refine the mesh in regions where the term is close to being 1. This should effectively increase the term here due to a smaller Δ and hence allow for more turbulence to be resolved. The following changes will therefore be made to the `lowerRefineLevel` and `upperRefineLevel` respectively.

```
sed -i s/0.001/0.85/g constant/dynamicMeshDict
sed -i s/0.999/0.95/g constant/dynamicMeshDict
```

The `unrefineLevel` value is defined such that a cell will be unrefined if the field is below this value. This is typically what you want in a simulation where the mesh is used to control the errors. In such cases, when the error is sufficiently small, the mesh can be unrefined in that region to save computational power and then refined in regions where the error is larger. In the case of mesh refinement however, the situation is the opposite. In this case it would be better to be able to control how much turbulence that is resolved by setting an upper limit to how large LDES/LDDES can become. Hence, if it becomes too large, the mesh should be unrefined in order to avoid resolving too much turbulence. As it is now this is not possible unless modifications to the `dynamicRefineFvMesh` class are made. So for now, we will just suppress unrefinement by putting `unrefineLevel` to 0

```
sed -i s/10/0/g constant/dynamicMeshDict
```

The maximum amount of cells, `maxCells`, must also be specified. The current mesh, before any extrusion to 3D contains 26950 cells, so an upper limit based on this number should be chosen. Here it will be chosen to 40000 for the 2D case

```
sed -i s/200000/40000/g constant/dynamicMeshDict
```

Finally, the remapping of the velocities/fluxes after refinement needs to be specified. To do this, open the file in an editor and change the `correctFluxes` to the following

```
    (
        (phi U)
        (phi_0 U_0)
        (phi_0_0 U_0_0)
        (ddt0(phi) ddt0(U))
    );
```

The underscore followed by 0 denotes the old time step, which is used for integrating in time by some schemes. To be on the safe side, two steps back in time has been supplied, but it may happen that the solver never uses these. The last part called `ddt0()` is for the Crank Nicholson algorithm.

5.5 Running the case

To begin with, the case will be run for 3 s without mesh refinement to allow the flow to stabilize. After this, the refinement will be enabled and the simulation will be started from the 3 s solution. The case files set up according to above, as well as the solution at time 3 s has been provided with this work.

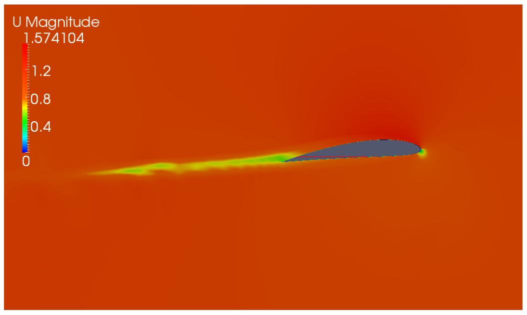
5.5.1 Without refinement

The case is now set up to run in PISO mode for 3 s without mesh refinement. To start the simulation, type

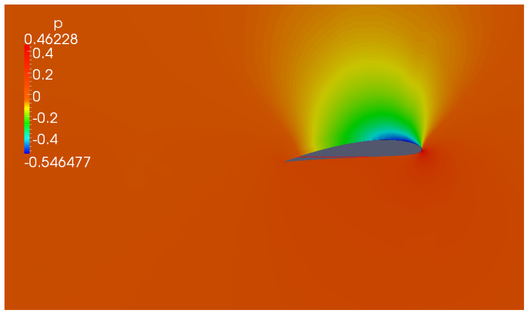
```
pimpleDyMFoam > log1 &
```

This will start the solver in the background and put the output in a log file. The simulations take time, especially in 3D. A 2D simulation will at least end up on taking a couple of hours, dependent on mesh quality, number of correctors in PIMPLE loop and mesh size.

Simulation results This work has not been concerned with evaluating the performance of the new turbulence model or the effect of mesh refinement to any larger extent. To do this, a much better mesh and deeper investigation into the numerical schemes is needed in order to get significant results to evaluate. But to give an indication of how the model performs and behaves, some simulation results are presented below.

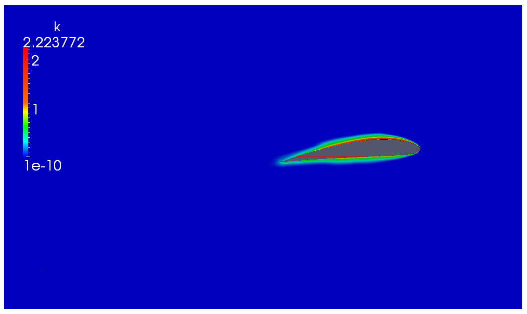


(a) Velocity.

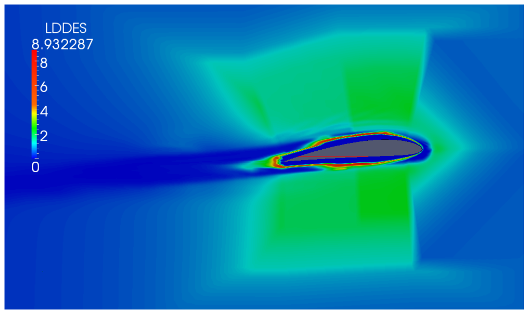


(b) Pressure.

Figure 5.1: Velocity and pressure for NACA 4412 Airfoil, $Re_L = 1 \cdot 10^5$.



(a) Turbulent kinetic energy, k.



(b) L_{DDES}.

Figure 5.2: Turbulent kinetic energy and L_{DDES} field for NACA 4412 Airfoil, $Re_L = 1 \cdot 10^5$.

5.5.2 Enabling refinement

After the flow has stabilized, its time to enable mesh refinement. Do this by changing the parameter `refineInterval` from 0 to 10 according to

```
sed -i s/"refineInterval  0"/"refineInterval  10"/g \
constant/dynamicMeshDict
```

Also change the start and end time of the simulation according to

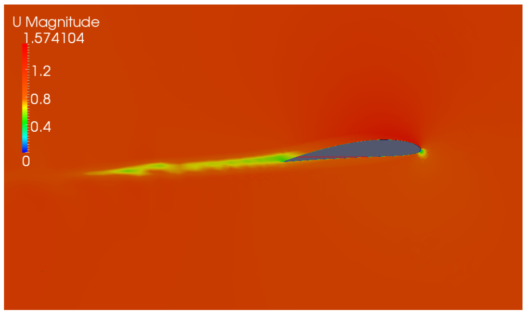
```
sed -i s/"startFrom       startTime"/"startFrom       latestTime"/g \
system/controlDict

sed -i s/"endTime         3"/"endTime         4"/g \
system/controlDict
```

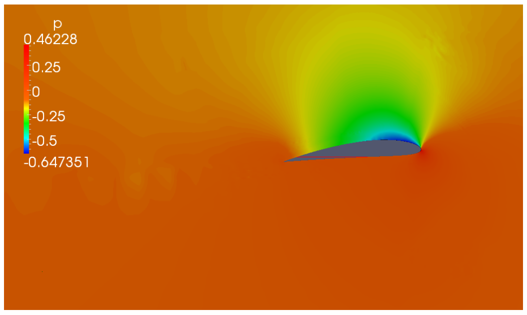
After this is done, start the solver again by typing

```
pimpleDyMFoam > log2 &
```

Simulation Results The results after mesh refinement are shown below

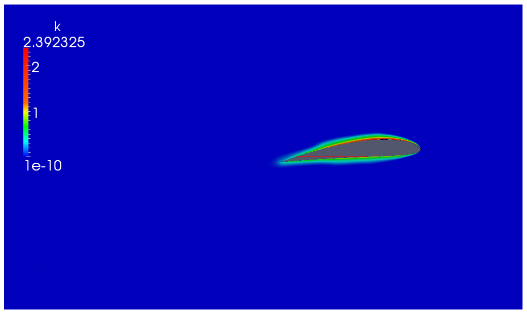


(a) Velocity.

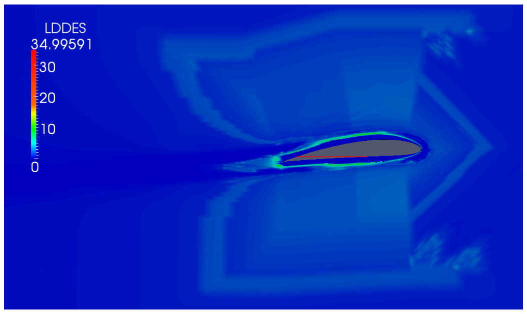


(b) Pressure.

Figure 5.3: Velocity and pressure for NACA 4412 Airfoil, $Re_L = 1 \cdot 10^5$, after refinement.

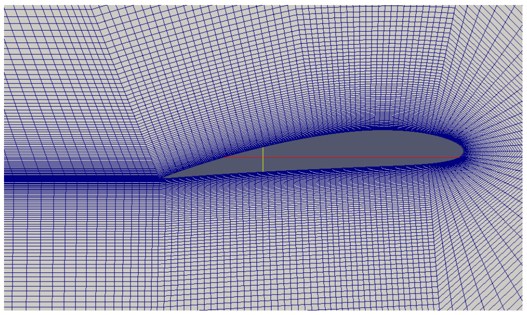


(a) Turbulent kinetic energy, k.

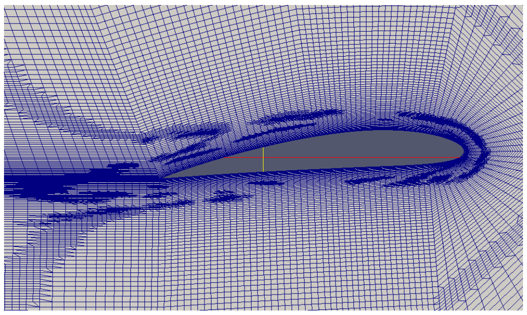


(b) L_{DDES}.

Figure 5.4: Turbulent kinetic energy and L_{DDES} field for NACA 4412 Airfoil, $Re_L = 1 \cdot 10^5$, after refinement.



(a) Turbulent kinetic energy, k.



(b) L_{DDES}.

Figure 5.5: Mesh before and after refinement.

Chapter 6

Conclusions

This work has involved a variety of topics within the scope of CFD and OpenFOAM, both with respect to theory, performing simulations and implementation of code.

To begin with the DES formulation of the $k - \omega$ SST model was presented with the aim of describing how it works. The conclusion was that the model is designed to resolve turbulence if the mesh allows in what is known as the LES mode and otherwise model all the turbulence using the original URANS model. Also from the theory it was noted that OpenFOAM includes the special wall functions used for the turbulent frequency which made it possible to easily assign proper boundary conditions to ω. The simulation was further set to be an airfoil traveling through a still fluid. This setting is not realistic in real life applications in which unsteadiness in the air is likely to be present, but allowed for a convenient and fairly easy case to set up. The aim of this project was also not to evaluate the model.

The description of the OpenFOAM implementation regarded a turbulence model, mesh refinement and a dynamic mesh solver. The turbulence model implementations in OpenFOAM are a little bit tricky to understand at first. However, after some training it is fairly easy to understand both how to modify them as well as to see what equations are actually set up and solved. This is a great advantage with OpenFOAM, since it allows the user to have full control over the turbulence modeling in contrast to closed code software. The implementation of the mesh refinement routine in OpenFOAM appears to be very well implemented and works well even though it appears to not be completely finished. The main drawback that was found was that the cells selected for mesh refinement, in cases where not all candidate cells could be refined, are not selected based on how large the governing field is in the cells. I.e., no intelligent selection appears to be performed to refine cells that are "in greater need" for refinement. As noted for the solver, it is not originally designed for refining meshes, but rather for moving meshes. It does however work for reining meshes but could use some modifications to be better suited.

The implementation of the new turbulence model proved to be convenient. Only the DDES model was enabled but the change between DES and DDES is easily made. In general, with only limited programming knowledge, it is possible to change the way a turbulence model works, which of course is a great feature.

The simulations proved that the turbulence model performs as intended, but did not try to do any validation work. It was found that the mesh quality is critical when it comes to to obtaining good results for DES simulations. A fair amount of work had to be put into obtaining meshes that produced good results. In particular, three key aspects of the mesh was found critical. First of all there is the skewness of the mesh. It was found that a very skewed mesh introduced oscillations in the flow field, which could only partly be avoided with more non-orthogonal correctors in the solver. Second there is the mesh uniformity. Since the mesh size directly control the source term in the k equation, a smoothly changing mesh size is critical. If the mesh changes a lot in size over a small distance, so does the source term, and this does not favor a smooth transition between URANS and LES mode. Last there is the resolution near walls. The DDES model includes a limiter to avoid fine near-wall mesh resolution from triggering LES here. However,this protector is not perfect and it could still happen that the model starts to resolve turbulence here if the mesh is sufficiently fine. In cases with the DES model, this effect is expected to be even more severe. Therefore, near wall mesh resolution is critical in using the DES model properly and as intended. Finally, as noted, the proper choice of solvers for different quantities, in particular pressure, was found important for fast convergence.

Appendix A

MATLAB program to generate airfoil geometry

The Fortran routine that is used to generate the `blockMeshDict` can take in an arbitrary airfoil geometry in a file called `Airfoil.data`. A MATLAB program was therefore written that generates an `Airfoil.data` file based on the NACA four digit standard for cambered airfoils. The program will generate a cross section shape of the airfoil as a set of discrete points constituting the surface. To give better resolution, the points lie closet together towards the leading edge, where the curvature generally is larger.

The program also plots the final airfoil shape together with the NACA number to allow the user to inspect the shape prior to creating the `blockMeshDict`. There is also an option to plot the airfoil shape at various angles of attack to see how it looks.

The following parameters are possible to adjust in the program

Table A.1: Parameters that can be adjusted when generating airfoil

Parameter	Description
`m`	Maximum camber
`p`	Location of maximum camber as fraction of length
`t`	Maximum thickness
`c`	Chord length (length of airfoil)

These parameters describe the shape of the airfoil. A description of how this is done is easily found on the internet and will not be presented here.

A list of the MATLAB files provided are listed in the next table

Table A.2: MATLAB files necessary to generate airfoil shape.

File	Description
`NACAxxxx.m`	Main program. Sets parameters and plot results
`y_c_calc.m`	Calculate the asymmetric camber line of the airfoil (mean line)
`y_t_calc.m`	Calculate the thickness of the airfoil
`dy_c_dx_calc.m`	Calculates the gradient, or normal, to the camber line
`profile_calc.m`	Calculates the final profile

Appendix B

Extension of `blockMeshDict` for 3D simulation

The `blockMeshDict` generated by the Fortran routine is for a 2D geometry, i.e. the front and back of the geometry is empty. The author also found that the orientation of the vertices defining the patches of the boundary was done in the counter clockwise, instead of clockwise direction. Therefore, the part of the `blockMeshDict` defining the boundary was first rewritten to allow for 2D simulations with correctly defined boundaries. It was also rewritten further to allow for 2D simulations using cyclic boundary conditions to connect the front and back of the domain. The changes to the boundary definition can be found in the files `AddToBlockMeshDictClockOrient.data` and `AddToBlockMeshDictClockOrientCyclic.data` respectively. To use these changes, first generate a `blockMeshDict` using the Fortran routine, and then substitute the part of it named `patches(...);` for the content of the file.

Bibliography

[1] Menter, F. R. Kuntz, M. Langtry, R. (2003) Ten Years of Industrial Experience with the SST Turbulence model. *Turbulence, Heat and Mass Transfer 4*, p. 624 - 632.

[2] Davidsson, L.(2006) Evaluation of the SST-SAS Model: Channel Flow, Asymmetric Diffuser and Axi-Symmetric Hill. In *European Conference on Computational Fluid Dynamics ECCOMAS CFD 2006*; September 5-8, 2006, Egmond aan Zee.

[3] Trimarchi, D. https://www.rocq.inria.fr/MACS/spip.php?rubrique69 (2013-10-05)

[4] Menter, F. R. Carregal Ferreira, J. Esch, T. Konno, B. (2003) The SST Turbulence Model with Improved Wall Treatment for Heat Transfer Predictions in Gas Turbines. In *Proceedings of the International Gas Turbine Congress 2003 Tokyo*; November 3-7, 2003, Tokyo.

[5] Versteeg, H. K. Malalasekera, W. (2007) *Computational Fluid Dynamics, The Finite Volume Method*. Second Edition. Harlow: Pearson Education Limited.

[6] Programmers Guide, Version 2.2.2.
http://foam.sourceforge.net/docs/Guides-a4/ProgrammersGuide.pdf