Cite as: Liu, S.N.: Implementation of a Complete Wall Function for the Standard $k-\epsilon$ Turbulence Model in OpenFOAM 4.0. In Proceedings of CFD with OpenSource Software, 2016,
Edited by Nilsson. H., `http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2016`

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Implementation of a Complete Wall Function for the Standard $k-\epsilon$ Turbulence Model in OpenFOAM 4.0

Developed for OpenFOAM-4.0

Author:
Shengnan Liu
University of Stavanger
shengnan.liu@uis.no

Peer reviewed by:
Mohammad Arabnejad
Håkan Nilsson

Disclaimer: This is a student project work, done as part of a course where OpenFOAM and some other OpenSource software are introduced to the students. Any reader should be aware that it might not be free of errors. Still, it might be useful for someone who would like learn some details similar to the ones presented in the report and in the accompanying files. The material has gone through a review process. The role of the reviewer is to go through the tutorial and make sure that it works, that it is possible to follow, and to some extent correct the writing. The reviewer has no responsibility for the contents.

February 16, 2017

Contents

1 Introduction **3**

2 Near-wall Physics **4**
2.1 Overview on $k-\epsilon$ model 4
2.2 Wall Functions 4

3 Wall Functions Implementation for Standard $k-\epsilon$ Turbulence Model in OpenFOAM 4.0 **7**
3.1 $k-\epsilon$ turbulence model code in OpenFOAM 4.0 7
3.2 Summary of available wall functions of $k-\epsilon$ turbulence model in OpenFOAM 4.0 . 10
3.2.1 k wall functions in OpenFOAM 4.0 10
3.2.2 ϵ wall functions in OpenFOAM 4.0 13
3.2.3 ν_T wall functions in OpenFOAM 4.0 20

4 New Wall Function Implementation for Standard $k-\epsilon$ Turbulence Model in OpenFOAM 4.0 **22**
4.1 Implementation of new wall function in OpenFOAM 4.0 22
4.2 Modifications to existing wall functions 23
4.2.1 Modification to kOngWallFunction 24
4.2.2 Modification to epsilonOngWallFunction 29
4.2.3 Modification to nutOngWallFunction 30
4.2.4 Compile Ong wall functions in OpenFOAM 4.0 31

5 Test Cases **33**
5.1 Test case 1 33
5.1.1 Case set up 33
5.1.2 Post-processig in paraFoam 34
5.2 Test Case 2 35
5.2.1 Case set up 35
5.2.2 Results 36

Learning outcomes

The reader will learn:

- how to build a new wall function.
- the theory of wall function.
- how to add new member functions.
- how to realize Newton interation.
- how to create nonuniform inlet input profile.
- how to test new wall function and post-processing.

Chapter 1

Introduction

Most turbulent flows are bounded by one or more solid surfaces, such as channel flow, pipe flow and flow around offshore foundations or ships. In turbulent flow, the presence of a wall causes a number of different effects, some of which are shown as follows:

- Low Reynolds number - the turbulence Reynolds number $Rel = k^2/(\epsilon v)$ decreases as the wall is approached. Here k is the turbulent kinetic energy, ϵ is the turbulence dissipation rate and v is the kinematic velocity.
- High shear rate - the highest mean shear rate $\partial < U > /\partial y$ ($< U >$ is the mean shear velocity, y is the distance in normal direction) occurs at the wall. The velocity changes from the no-slip condition at the wall to its stream value.
- Wall blocking - through the pressure field, the impermeability condition the v=0 (at y=0) affects the flow up to an integral scale from the wall.

The present work is based on the $k - \epsilon$ turbulent model. The form of the basic $k - \epsilon$ and shear stress models have not changed since 1970s (pioneered by Jones and Launder, 1972; Launder and Sharma, 1974). However, researchers still have different ideas about the near wall treatment until present (Kalitzin et al., 2005; Ong et al., 2009; Parente et al., 2011 and Balogh et al., 2012). In the late 1980s (pioneered by Rogallo, 1981), detailed direct numerical simulation (DNS) data for viscous near-wall region started to be available, which the current wall function mainly based on. If DNS is used to simulate the near-wall region, very fine mesh close to the wall is required to resolve the flow field for the direct integration, especially at high Reynolds number flow condition. The smallest turbulence scales decrease with the increase of Reynolds numbers, moreover the boundary layer will be thin and there will be high mean velocity gradient on the wall, so a large number of grids are needed to capture near-wall pressure and velocity gradients. The idea of the $wallfunction$ approach is to use appropriate wall boundary conditions at some distance away from the wall, so that fine grids are not required to resolve the near-wall flow condition. In this way, it will reduce the computational cost significantly.

Chapter 2

Near-wall Physics

2.1 Overview on $k-\epsilon$ model

The $k-\epsilon$ turbulence model belongs to the two-equation models, in which model transport equations are solved for two turbulence quantities (Launder and Spalding, 1972; Rodi, 1993), see equations 2.1 and 2.2.

$$\frac{\partial k}{\partial t} + u_j \frac{\partial k}{\partial x_j} = \frac{\partial}{\partial x_j}(\frac{\nu_T}{\sigma_k}\frac{\partial k}{\partial x_j}) + \nu_T(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i})\frac{\partial u_i}{\partial x_j} - \epsilon \tag{2.1}$$

$$\frac{\partial \epsilon}{\partial t} + u_j \frac{\partial \epsilon}{\partial x_j} = \frac{\partial}{\partial x_j}(\frac{\nu_T}{\sigma_\epsilon}\frac{\partial \epsilon}{\partial x_j}) + C_1\frac{\epsilon}{k}\nu_T(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i})\frac{\partial u_i}{\partial x_j} - C_2\frac{\epsilon^2}{k} \tag{2.2}$$

The main components of $k-\epsilon$ model are concluded as follow.

- The first is the transport equation for turbulent kinetic energy k, which determines the energy in the turbulence.
- The second is the transport equation for turbulent dissipation ϵ, which determines the rate of dissipation of the turbulent kinetic energy.
- The turbulent viscosity is specified as $\nu_T = C_\mu k^2/\epsilon$ and $C_1 = 1.44$, $C_2 = 1.92$, $C_\mu = 0.09$, $\sigma_k = 1.0$, $\sigma_\epsilon = 1.3$.

2.2 Wall Functions

Typical boundary layer flow over a flat plate includes three regions: linear viscous sub-layer, buffer layer and log-law layer (also called inertial sub-layer) (Tennekes and Lumley, 1972). Viscous effect dominates in the viscous sub-layer. For the log-law layer, the viscous effect is small, and it is dominated by turbulence. For the buffer layer, both viscous and turbulent effects are important. figure 2.1 shows the relation between y^+ and u^+, where $y^+ = (u^* y)/\nu$, y is the normal distance from the wall. $u^* = (\tau_w/\rho)^{1/2}$ is the friction velocity, where τ_w is the wall shear stress and ρ is the density of water, and ν is the kinematic viscosity of the fluid, $u^+ = u/u^*$, u is the tangential velocity of the fluid.

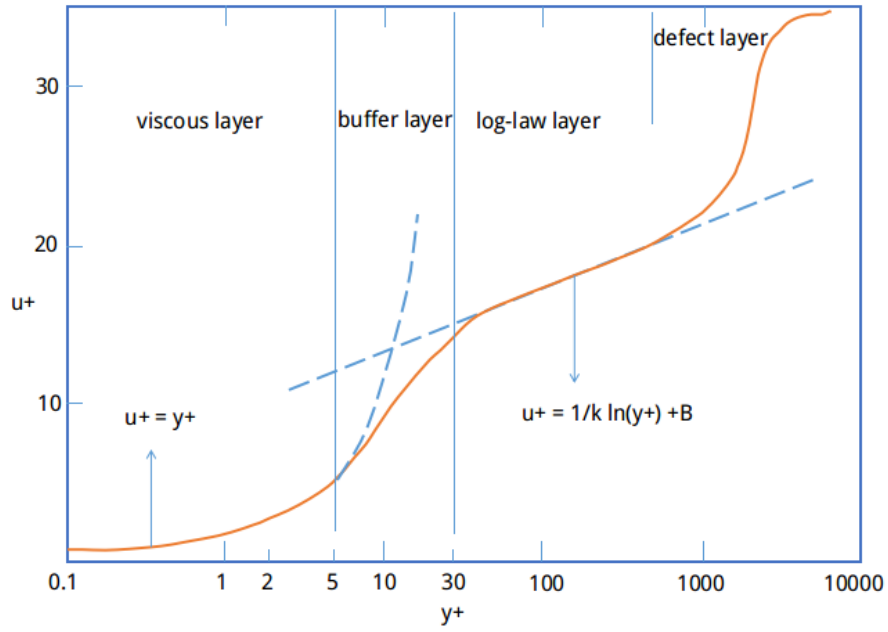

Figure 2.1: Wall function of different layers.

- Viscous sub-layer

  The fluid very close to the wall is dominated by viscous shear in absence of the turbulent shear stress effects for $y^+ < 5$. It can be assumed that the shear stress is almost equal to the wall shear stress τ_ω throughout the viscous layer.

$$\tau_\omega = \mu \frac{\partial u}{\partial y} \tag{2.3}$$

  The expression shows the velocity gradient on the wall, where u is the tangential velocity. In this region $u^+ = y^+$, but the standard specification $\nu_T = C_\mu k^2/\epsilon$ yields too large for turbulent viscosity in the near wall region. Jones and Launder (1972) include various damping functions to allow the model to be used within the viscous near wall region, $\nu_T = f_\mu C_\mu k^2/\epsilon$. Rodi and Mansour (1993) suggested one relation according to the DNS data: $f_\mu = 1 - exp(-0.0002y^+ - 0.00065y^{+2})$, which is used in the present study. Applying boundary conditions and manipulations, we obtain the following equation set to be used in the viscous near wall region:

$$\begin{cases} y^+ = u^* y/\nu \\ u^+ = u/u^* \\ u^+ = y^+ \\ k = u^{*2}/\sqrt{C_\mu} \\ \epsilon = C_\mu^{3/4} k^{3/2}/\kappa y \\ \nu_T = f_\mu C_\mu k^2/\epsilon \end{cases} \tag{2.4}$$

- Buffer layer (mixed layer)

  Outside the viscous sub-layer ($5 < y^+ < 30$) buffer layer region exists. The largest variation occurs from either law occurring approximately where the two equations intercept, at

$y^+ = 11$. That is, before 11 wall units the linear approximation is more accurate and after 11 wall units the logarithmic approximation should be used. Considering both the linear and logarithmic approximation by a weighted average (linear interpolation), u^+ can be obtained as follows (Ong et al., 2009).

$$u^+ = \frac{1}{(\frac{\kappa\omega}{ln(Ey^+)}) + (\frac{1-\omega}{y^+})} \tag{2.5}$$

where the weighting factor $\omega = (y^+ - 5)/25$, $E = 9.8$, von Karman constant $\kappa = 0.41$. Similarly, we can obtain the following equation set to be used in the near wall region:

$$\begin{cases} y^+ = u^* y/\nu \\ u^+ = u/u^* \\ u^+ = \frac{1}{(\frac{\kappa\omega}{ln(Ey^+)}) + (\frac{1-\omega}{y^+})} \\ k = u^{*2}/\sqrt{C_\mu} \\ \epsilon = C_\mu^{3/4} k^{3/2}/\kappa y \\ \nu_T = f_\mu C_\mu k^2/\epsilon \end{cases} \tag{2.6}$$

- Log-law layer

  At some distance from the wall and outside the buffer layer ($30 < y^+ < 100$) a region exists where turbulent effects are important. Within this inner region the shear stress is assumed to be constant and equal to wall shear stress and varying gradually with distance from the wall. The relationship between y^+ and u^+ in the log-law region is given as:

$$u^+ = \frac{1}{\kappa} ln(Ey^+) \tag{2.7}$$

  where E=9.8. As the relationship between y^+ and u^+ is logarithmic, the above expression is known as log-law and the layer where y^+ takes the values between 30 and 100 is known as log-law layer. We can obtain the following equation set to be used in the near wall region:

$$\begin{cases} y^+ = u^* y/\nu \\ u^+ = u/u^* \\ u^+ = \frac{1}{\kappa} ln(Ey^+) \\ k = u^{*2}/\sqrt{C_\mu} \\ \epsilon = C_\mu^{3/4} k^{3/2}/\kappa y \\ \nu_T = f_\mu C_\mu k^2/\epsilon \end{cases} \tag{2.8}$$

- Defect layer

  In an defect layer (overlap region, $y^+ >= 100$) with approximately constant shear stress and far enough from the wall for (direct) viscous effects to be negligible.

Due to different wall functions in different regions, the height of the first layer must be accurately calculated, so that the first node results will be obtained from the right functions. However, in OpenFOAM, the wall function for $k - \epsilon$ model are not defined strictly according to the method stated above, which are only available for one region (log-law region) or at most two regions (viscous region and log-law region), as the green dash line in Figure 2.1. The objective of this project is to modify the wall functions for the $k - \epsilon$ model in OpenFOAM in order to cover all the regions in boundary layer .

Chapter 3

Wall Functions Implementation for Standard $k-\epsilon$ Turbulence Model in OpenFOAM 4.0

3.1 $k-\epsilon$ turbulence model code in OpenFOAM 4.0

The code of $k-\epsilon$ turbulence model can be found in $http://cpp.openfoam.org/v4/a10852\_source.html$ or in folder \$FOAM_SRC/ TurbulenceModels/ turbulenceModels/ RAS/ kEpsilon/ kEpsilon.C can be checked by command:OF4x && vi \$FOAM_SRC/TurbulenceModels/turbulenceM odels/RAS/kEpsilon/kEpsilon.C. $kEpsilon < BasicTurbulenceModel >$ calls three functions, i.e. $GeometricField$, $eddyViscosity$ and $dimensioned$. The collabration diagram for $kEpsilon < BasicTurbulenceModel >$ is shown as figure 3.1 for settings.

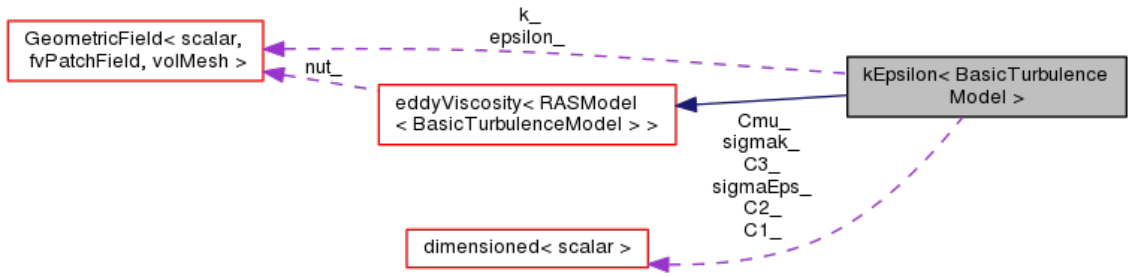

Figure 3.1: Collabration diagram for kEpsilon

The main code of $kEpsilon < BasicTurbulenceModel >$ is shown as follows.

```
39 template<class BasicTurbulenceModel>
40 void kEpsilon<BasicTurbulenceModel>::correctNut()
41 {
42  this->nut_ = Cmu_*sqr(k_)/epsilon_;
43  this->nut_.correctBoundaryConditions();
44  fv::options::New(this->mesh_).correct(this->nut_);
45
46  BasicTurbulenceModel::correctNut();
47 }
48
49
```

```
50 template<class BasicTurbulenceModel>
51 tmp<fvScalarMatrix> kEpsilon<BasicTurbulenceModel>::kSource() const
52 {
53  return tmp<fvScalarMatrix>
54  (
55  new fvScalarMatrix
56  (
57  k_,
58  dimVolume*this->rho_.dimensions()*k_.dimensions()
59  /dimTime
60  )
61  );
62 }
63
64
65 template<class BasicTurbulenceModel>
66 tmp<fvScalarMatrix> kEpsilon<BasicTurbulenceModel>
::epsilonSource() const
67 {
68  return tmp<fvScalarMatrix>
69  (
70  new fvScalarMatrix
71  (
72  epsilon_,
73  dimVolume*this->rho_.dimensions()*epsilon_.dimensions()
74  /dimTime
75  )
76  );
77 }
```

Firstly, *kSource*() is defined to obtain the value of k, and *epsilonSource*() is defined to obtain the value of *epsilon*, *correctNut*() is used to correct ν_T in the whole field. Then the function *correct*() is defined which is also the main function of *kEpsilon.C* (shown below). The main calculation process stated in the code will be explained.

```
void kEpsilon<BasicTurbulenceModel>::correct()
222 {
223  if (!this->turbulence_)
224  {
225  return;
226  }
227
228  // Local references
229  const alphaField& alpha = this->alpha_;
230  const rhoField& rho = this->rho_;
231  const surfaceScalarField& alphaRhoPhi = this->alphaRhoPhi_;
232  const volVectorField& U = this->U_;
233  volScalarField& nut = this->nut_;
234  fv::options& fvOptions(fv::options::New(this->mesh_));
235
236  eddyViscosity<RASModel<BasicTurbulenceModel>>::correct();
237
238  volScalarField::Internal divU
239  (
```

```
240   fvc::div(fvc::absolute(this->phi(), U))().v()
241   );
242
243   tmp<volTensorField> tgradU = fvc::grad(U);
244   volScalarField::Internal G
245   (
246   this->GName(),
247   nut.v()*(dev(twoSymm(tgradU().v())) && tgradU().v())
248   );
249   tgradU.clear();
250
251   // Update epsilon and G at the wall
252   epsilon_.boundaryFieldRef().updateCoeffs();
253
254   // Dissipation equation
255   tmp<fvScalarMatrix> epsEqn
256   (
257   fvm::ddt(alpha, rho, epsilon_)
258   + fvm::div(alphaRhoPhi, epsilon_)
259   - fvm::laplacian(alpha*rho*DepsilonEff(), epsilon_)
260   ==
261   C1_*alpha()*rho()*G*epsilon_()/k_()
262   - fvm::SuSp(((2.0/3.0)*C1_ + C3_)*alpha()*rho()*divU, epsilon_)
263   - fvm::Sp(C2_*alpha()*rho()*epsilon_()/k_(), epsilon_)
264   + epsilonSource()
265   + fvOptions(alpha, rho, epsilon_)
266   );
267
268   epsEqn.ref().relax();
269   fvOptions.constrain(epsEqn.ref());
270   epsEqn.ref().boundaryManipulate(epsilon_.boundaryFieldRef());
271   solve(epsEqn);
272   fvOptions.correct(epsilon_);
273   bound(epsilon_, this->epsilonMin_);
274
275   // Turbulent kinetic energy equation
276   tmp<fvScalarMatrix> kEqn
277   (
278   fvm::ddt(alpha, rho, k_)
279   + fvm::div(alphaRhoPhi, k_)
280   - fvm::laplacian(alpha*rho*DkEff(), k_)
281   ==
282   alpha()*rho()*G
283   - fvm::SuSp((2.0/3.0)*alpha()*rho()*divU, k_)
284   - fvm::Sp(alpha()*rho()*epsilon_()/k_(), k_)
285   + kSource()
286   + fvOptions(alpha, rho, k_)
287   );
288
289   kEqn.ref().relax();
290   fvOptions.constrain(kEqn.ref());
291   solve(kEqn);
292   fvOptions.correct(k_);
293   bound(k_, this->kMin_);
```

```
294
295   correctNut();
296 }
```

From the code above, the turbulence calculation can be concluded as following:

- Calculate turbulent kinetic energy production term G and correct the value of G at first layer mesh close to the wall by '$epsilon\_.boundaryFieldRef().updateCoeffs()$'. The correction on ϵ and G is achieved by updateCoeffs() function of ϵ.
- After updating G, the ϵ equation is built by this new G. Then the ϵ equation is revised by '$epsEqn.ref().boundaryManipulate(epsilon\_.boundaryFieldRef())$;'
- Solve ϵ equation and obtain the updated ϵ field.
- Solve k equation using the new ϵ, and k field including the k on the wall is renewed.
- Calculate ν_T, and update the ν_T at wall by correctNut();

3.2 Summary of available wall functions of $k-\epsilon$ turbulence model in OpenFOAM 4.0

The key parameters of $k-\epsilon$ wall function include k, ϵ, ν_T. The available wall functions of k, ϵ, ν_T of OpenFOAM 4.0 are concluded here. In addition the wall functions used for further modification are explained in detail.

3.2.1 k wall functions in OpenFOAM 4.0

In OpenFOAM 4.0, there are two available wall functions for k, i.e., kqRWallFunction and kLowReWallFunction. Normally, kqRWallFunction is used for high Reynolds numbers and kLowReWallFunction can be used for both low Reynolds numbers and high Reynolds numbers. See Table 3.1.

Type name	kqRWallFunction	kLowReWallFunction
Available scope (first layer cell position)	Log-law region	Viscous and log-law region
Class	Foam::kqRWallFunctionFvPatchField	Foam::kLowReWallFunctionFvPatchField
Inherit from	Foam::zeroGradientFvPatchField	Foam::fixedValueFvPatchField
Other references	zeroGradient	fixedValue

Table 3.1: Available k wall functions in OpenFoam 4.0

kqRWallFunction is a simple wrapper around the zero-gradient condition. It provides a suitable condition for turbulence k, q and R fields for the case of high Reynolds number flow. kLowReWallFunction provides a turbulence kinetic energy wall function condition for both low- and high-Reynolds number turbulent flow cases. The model operates in two modes, based on the computed laminar-to-turbulent switch-over y^+ value derived from kappa and E.
k wall function is modified based on kLowReWallFunction, therefore, kLowReWallFunctionFvPatchScalarField.C is explained here.

First, the function *yPlusLam* is used to calculate the switching point of y^+, and the return value will be stored at yPlusLam_.

```
scalar kLowReWallFunctionFvPatchScalarField::yPlusLam
57 (
58  const scalar kappa,
59  const scalar E
60 )
61 {
62  scalar ypl = 11.0;
63
64  for (int i=0; i<10; i++)
65  {
66  ypl = log(max(E*ypl, 1))/kappa;
67  }
68
69  return ypl;
70 }
71
```

According to the code above, y^+ at switching point is calculated by 10-step iteration. The iteration formula used here is $y^+ = log(max(E*y^+,1))/\kappa$; because in viscous sublayer $u^+ = y^+$, and $u^+ = 1/kln(Ey^+)$ in log-law layer. Through the iteration, the switching point of y^+ between viscous sublayer and log-law layer can be obtained and will also be used for the mode change for k, ϵ and nut. The initial value of y^+ is set to 11, because theoretically the switch point will be around 11. After the iteration, the value is around 11.53. And this value will be stored in yPlusLam_.

The value of k is set by the following function '$updateCoeffs()$'.

```
void kLowReWallFunctionFvPatchScalarField::updateCoeffs()
165 {
166     if (updated())
167     {
168         return;
169     }
170
171     const label patchi = patch().index();
172
173     const turbulenceModel& turbModel
        = db().lookupObject<turbulenceModel>
174     (
175         IOobject::groupName
176         (
177             turbulenceModel::propertiesName,
178             internalField().group()
179         )
180     );
181     const scalarField& y = turbModel.y()[patchi];
182
183     const tmp<volScalarField> tk = turbModel.k();
184     const volScalarField& k = tk();
185
```

```
186     const tmp<scalarField> tnuw = turbModel.nu(patchi);
187     const scalarField& nuw = tnuw();
188
189     const scalar Cmu25 = pow025(Cmu_);
190
191     scalarField& kw = *this;
192
193     // Set k wall values
194     forAll(kw, facei)
195     {
196         label faceCelli = patch().faceCells()[facei];
197
198         scalar uTau = Cmu25*sqrt(k[faceCelli]);
199
200         scalar yPlus = uTau*y[facei]/nuw[facei];
201
202         if (yPlus > yPlusLam_)
203         {
204             scalar Ck = -0.416;
205             scalar Bk = 8.366;
206             kw[facei] = Ck/kappa_*log(yPlus) + Bk;
207         }
208         else
209         {
210             scalar C = 11.0;
211             scalar Cf = (1.0/sqr(yPlus + C)
                        + 2.0*yPlus/pow3(C) - 1.0/sqr(C));
212             kw[facei] = 2400.0/sqr(Ceps2_)*Cf;
213         }
214
215         kw[facei] *= sqr(uTau);
216     }
217
218     // Limit kw to avoid failure of the turbulence
        model due to division by kw
219     kw = max(kw, SMALL);
220
221     fixedValueFvPatchField<scalar>::updateCoeffs();
222
223     // TODO: perform averaging for cells sharing
        more than one boundary face
224 }
```

This function shows how the k value at the current wall calculated. This refers to Kalitzin et al. (2005) and is designed for v^2-f model. At first friction velocity u^* is calculated by k_c (subscript c means the value of the cell close to the wall) and y^+ is calculated based on this u^* with the following expression.

$$\begin{cases} u^* = C_\mu^{1/4}\sqrt{k_c} \\ y^+ = u^* y/\nu_w \end{cases} \tag{3.1}$$

Then k_w (value at the wall) is calculated through:

$$k_w = \begin{cases} (C_k/\kappa ln(y^+) + B_k) * \sqrt{C_\mu} * k_c, \; y^+ > yPlusLam \\ 2400 * C_f/C_{eps2}^2 * \sqrt{C_\mu} * k_c, \; y^+ < yPlusLam \end{cases} \tag{3.2}$$

where $C_f = (1.0/(y^+ + C)^2 + 2.0 * y^+/C^3 - 1.0/C^2)$; $C = 11.0$; C_{μ}=0.09, $\kappa_- = 0.41$, $E_- = 9.8$, $C_{eps2} = 1.9$, $C_k = -0.416$; $B_k = 8.366$.

3.2.2 ϵ wall functions in OpenFOAM 4.0

Correspondingly, there are two available wall functions for ϵ in OpenFOAM 4.0, i.e., epsilonWallFunction and epsilonLowReWallFunction. Normally, epsilonWallFunction is used for high Reynolds numbers and epsilonLowReWallFunction can be used for both low Reynolds numbers and high Reynolds numbers (see Table 3.2). epsilonWallFunction provides a turbulence dissipation wall function condition for high Reynolds number turbulent flow cases. This wall function calculates ϵ and G (produnction term), and inserts near wall epsilon values directly into the epsilon equation to act as a constraint. epsilonLowReWallFunction can be used for both low- and high-Reynolds number turbulent flow cases. The model operates in two modes, based on the computed laminar-to-turbulent switch-over y+ value derived from kappa and E, which is the same with the calculation in kLowReWallFunction. epsilonWallFunctionFvPatchScalarField.C is explained here.

Type name	epsilonWallFunction	epsilonLowReWallFunction
Available scope (first layer cell position)	Log-law region	Viscous and log-law region
Class	Foam::epsilonWallFunctionFvPatchField	Foam::epsilonLowReWallFunctionFvPatchField
Formula	$\epsilon_c = \frac{1}{N}\sum_{f=i}^{N}(\frac{C_\mu^{3/4}k_c^{3/2}}{\kappa y_i})$	$\epsilon_c = \frac{1}{N}\sum_{f=i}^{N}(\frac{C_\mu^{3/4}k_c^{3/2}}{\kappa y_i})$ $\epsilon_c = \frac{1}{N}\sum_{f=i}^{N}(\frac{2k^{c\nu}}{y_i^2})$
Inherit from	Foam::fixedInternalValueFvPatchField	Foam::epsilonWallFunctionFvPatchScalarField

Table 3.2: Available k wall functions in OpenFoam 4.0

The main external function of $epsilonWallFunctionFvPatchScalarField.C$ is $updateWeighted-Coeffs()$, first it checks whether the epsilon value at the wall is updated, if not, it calls the $setMaster()$ function to set the master patch, then update the values of epsilon on master patches (wall patches). The code of $updateWeightedCoeffs()$ is shown below.

```
void Foam::epsilonWallFunctionFvPatchScalarField::updateWeightedCoeffs
  449 (
  450     const scalarField& weights
  451 )
  452 {
  453     if (updated())
  454     {
  455         return;
  456     }
  457
  458     const turbulenceModel& turbModel
          = db().lookupObject<turbulenceModel>
  459     (
```

```
460             IOobject::groupName
461             (
462                 turbulenceModel::propertiesName,
463                 internalField().group()
464             )
465         );
466
467         setMaster();
468
469         if (patch().index() == master_)
470         {
471             createAveragingWeights();
472             calculateTurbulenceFields(turbModel, G(true)
                , epsilon(true));
473         }
474
475         const scalarField& G0 = this->G();
476         const scalarField& epsilon0 = this->epsilon();
477
478         typedef DimensionedField<scalar, volMesh> FieldType;
479
480         FieldType& G =
481             const_cast<FieldType&>
482             (
483                 db().lookupObject<FieldType>(turbModel.GName())
484             );
485
486         FieldType& epsilon = const_cast<FieldType&>(internalField());
487
488         scalarField& epsilon = *this;
489
490         // only set the values if the weights are > tolerance
491         forAll(weights, facei)
492         {
493             scalar w = weights[facei];
494
495             if (w > tolerance_)
496             {
497                 label celli = patch().faceCells()[facei];
498
499                 G[celli] = (1.0 - w)*G[celli] + w*G0[celli];
500                 epsilon[celli] = (1.0 - w)*epsilon[celli]
                    + w*epsilon0[celli];
501                 epsilonf[facei] = epsilon[celli];
502             }
503         }
504
505         fvPatchField<scalar>::updateCoeffs();
506     }
```

The code of function *setMaster*() is shown as follows.

```
    void Foam::epsilonWallFunctionFvPatchScalarField::setMaster()
66  {
```

```
67      if (master_ != -1)
68      {
69          return;
70      }
71
72      const volScalarField& epsilon =
73          static_cast<const volScalarField&>(this->internalField());
74
75      const volScalarField::Boundary& bf = epsilon.boundaryField();
76
77      label master = -1;
78      forAll(bf, patchi)
79      {
80          if (isA<epsilonWallFunctionFvPatchScalarField>(bf[patchi]))
81          {
82              epsilonWallFunctionFvPatchScalarField& epf
                = epsilonPatch(patchi);
83
84              if (master == -1)
85              {
86                  master = patchi;
87              }
88
89              epf.master() = master;
90          }
91      }
92  }
```

First, this '*setMaster*' function judges that if the *master_* value of the current member is not equal to -1. If it is true, then the return action will be executed. Otherwise it will obtain the epsilon boundary and store the value at bf. Then for all the bf, if the boundary type is 'epsilonWallFunctionFvPatchScalarField', then it will do another judgement, i.e. whether the temporary variable 'master' is equal to 1, if it is true, pass the value of 'patchi' to 'master', then pass the value of temporary variable 'master' to the corresponding epf.master(). Overall, if there are several boundaries use the 'epsilonWallFunctionFvPatchScalarField' boundary type, the the boundary with smallest index will be set as master, the non-master boundary can obtain information from master.

According to the $updateWeightedCoeffs()$ code, after setMaster(), check whether the patch type is master type '$epsilonWallFunctionFvPatch$'. If it is ture, then '$createAveragingWeights()$' and '$calculateTurbulenceFields()$' will be executed. These two functions are explained as follows.

```
  void
  Foam::epsilonWallFunctionFvPatchScalarField::createAveragingWeights()
  96 {
  97      const volScalarField& epsilon =
  98          static_cast<const volScalarField&>(this->internalField());
  99
 100      const volScalarField::Boundary& bf = epsilon.boundaryField();
 101
 102      const fvMesh& mesh = epsilon.mesh();
 103
 104      if (initialised_ && !mesh.changing())
 105      {
 106          return;
```

```
107     }
108
109     volScalarField weights
110     (
111         IOobject
112         (
113             "weights",
114             mesh.time().timeName(),
115             mesh,
116             IOobject::NO_READ,
117             IOobject::NO_WRITE,
118             false // do not register
119         ),
120         mesh,
121         dimensionedScalar("zero", dimless, 0.0)
122     );
123
124     DynamicList<label> epsilonPatches(bf.size());
125     forAll(bf, patchi)
126     {
127         if (isA<epsilonWallFunctionFvPatchScalarField>(bf[patchi]))
128         {
129             epsilonPatches.append(patchi);
130
131             const labelUList& faceCells
                = bf[patchi].patch().faceCells();
132             forAll(faceCells, i)
133             {
134                 weights[faceCells[i]]++;
135             }
136         }
137     }
138
139     cornerWeights_.setSize(bf.size());
140     forAll(epsilonPatches, i)
141     {
142         label patchi = epsilonPatches[i];
143         const fvPatchScalarField& wf
            = weights.boundaryField()[patchi];
144         cornerWeights_[patchi] = 1.0/wf.patchInternalField();
145     }
146
147     G_.setSize(internalField().size(), 0.0);
148     epsilon_.setSize(internalField().size(), 0.0);
149
150     initialised_ = true;
151 }
```

'*createAveragingWeights*()' is used to set the weight of every patch cell. The weight will be used in the calculation of G and epsilon later. Inside this function, '*DynamicList*' means checking all the boundary fields. If its type is '*epsilonWallFunctionFvPatchScalarField*', it will then be put in DynamicList. '*weight*' is the number of wall boundary faces in one cell, it is used to weight how many boundary faces with '*epsilonWallFunctionFvPatchScalarField*' type the celli

use. '*cornerWeights*' is used to save the inversed value of '*weight*', and weight of every boundary faces equal to the weight of the cell which the faces belong to. Then line 147 and line 148 set initial value of G and epsilon to 0. Actually, G_ and epsilon_ save the value of the whole internal field instead of the boundary cell value. It means the data member G_ and epsilon_ of master patch contain the information of both master patches and non-master patches. So the value of non-master patches can also be obtained from G_ and epsilon_ according to the corresponding cell id.

Subsequently, *calculateTurbulenceFields* function is called, the details are explained as follows.

```
void Foam::
epsilonWallFunctionFvPatchScalarField::calculateTurbulenceFields
170 (
171     const turbulenceModel& turbulence,
172     scalarField& G0,
173     scalarField& epsilon0
174 )
175 {
176     // accumulate all of the G and epsilon contributions
177     forAll(cornerWeights_, patchi)
178     {
179         if (!cornerWeights_[patchi].empty())
180         {
181             epsilonWallFunctionFvPatchScalarField& epf
              = epsilonPatch(patchi);
182
183             const List<scalar>& w = cornerWeights_[patchi];
184
185     epf.calculate(turbulence, w, epf.patch(), G0, epsilon0);
186         }
187     }
188
189     // apply zero-gradient condition for epsilon
190     forAll(cornerWeights_, patchi)
191     {
192         if (!cornerWeights_[patchi].empty())
193         {
194             epsilonWallFunctionFvPatchScalarField& epf
              = epsilonPatch(patchi);
195
196         epf == scalarField(epsilon0, epf.patch().faceCells());
197         }
198     }
199 }
```

In function '*calculateTurbulenceFields*', first 'if !cornerWeights_[patchI].empty()' is used to judge the wall type. If it is '*epsilonWallFunctionFvPatchScalarField*', then the calculate function is called to update the G0 and epsilon value. Then zero-gradient condition is applied for epsilon. The first layer mesh value is assigned to the wall patch value.

The statements '$const scalarField\&G0 = this->G();$' and '$const scalarField\&epsilon0 = this->epsilon();$' return the value of member function G and epsilon back to variable G0 and epsilon0. The definition of member function G and epsilon are shown as follows.

```
Foam::scalarField&
```

```
Foam::epsilonWallFunctionFvPatchScalarField::G(bool init)
365 {
366     if (patch().index() == master_)
367     {
368         if (init)
369         {
370             G_ = 0.0;
371         }
372
373         return G_;
374     }
375
376     return epsilonPatch(master_).G();
377 }
378
379
380 Foam::scalarField&
    Foam::epsilonWallFunctionFvPatchScalarField::epsilon
381 (
382     bool init
383 )
384 {
385     if (patch().index() == master_)
386     {
387         if (init)
388         {
389             epsilon_ = 0.0;
390         }
391
392         return epsilon_;
393     }
394
395     return epsilonPatch(master_).epsilon(init);
396 }
```

The two functions G and *epsilon* return the values of G_ and epsilon_, for the 'epsilonWallFunctionFvPatchScalarField' type wall boundary patches. Then the returned values of non-master patches come from the data member of master_.

As a summary, the patch cell with smaller id number will be set to 'master' when using this wall function. When the 'master' type is called, the parameters will be calculated, and 'non-master' members can obtain required information from 'master'.

The *calculate* function is the function to calculate the value of epsilon and G, it is shown as follows.

```
void Foam::epsilonWallFunctionFvPatchScalarField::calculate
203 (
204     const turbulenceModel& turbulence,
205     const List<scalar>& cornerWeights,
206     const fvPatch& patch,
207     scalarField& G,
208     scalarField& epsilon
209 )
```

```
210 {
211     const label patchi = patch.index();
212
213     const scalarField& y = turbulence.y()[patchi];
214
215     const scalar Cmu25 = pow025(Cmu_);
216     const scalar Cmu75 = pow(Cmu_, 0.75);
217
218     const tmp<volScalarField> tk = turbulence.k();
219     const volScalarField& k = tk();
220
221     const tmp<scalarField> tnuw = turbulence.nu(patchi);
222     const scalarField& nuw = tnuw();
223
224     const tmp<scalarField> tnutw = turbulence.nut(patchi);
225     const scalarField& nutw = tnutw();
226
227     const fvPatchVectorField& Uw =
        turbulence.U().boundaryField()[patchi];
228
229     const scalarField magGradUw(mag(Uw.snGrad()));
230
231     // Set epsilon and G
232     forAll(nutw, facei)
233     {
234         label celli = patch.faceCells()[facei];
235
236         scalar w = cornerWeights[facei];
237
238         epsilon[celli] +=
           w*Cmu75*pow(k[celli], 1.5)/(kappa_*y[facei]);
239
240         G[celli] +=
241             w
242            *(nutw[facei] + nuw[facei])
243            *magGradUw[facei]
244            *Cmu25*sqrt(k[celli])
245            /(kappa_*y[facei]);
246     }
247 }
```

The calculate function calculates the value of G and epsilon using the following expression:

$$\begin{aligned} \epsilon_c &= \frac{1}{N}\sum_{f=i}^{N}\left(\frac{C_\mu^{3/4}k_c^{3/2}}{\kappa y_i}\right) \\ G_c &= \frac{1}{N}\sum_{f=i}^{N}\left(\frac{(\nu+\nu_T)*|\frac{U_i-U_c}{d}|C_\mu^{1/4}k_c^{1/2}}{\kappa y_i}\right) \end{aligned} \tag{3.3}$$

Subscript c means the value of the cell close to the wall; i represents the index of boundary cell; y is the normal distance from the cell center to the wall patch. '*ManipulateMatrix*' function (line 525-575) renews the parameters of each patch cell into the matrix.

3.2.3 ν_T wall functions in OpenFOAM 4.0

There are many types of ν_T wall functions in OpenFOAM 4.0 which are built based on one virtual base class '*nutWallFunction*'. The ν_T wall functions calculat the turbulence viscosity on the wall by using virtual function '*calcNut*', and return the ν_T value to the boundary through function '*updateCoeffs*'. Different ν_T wall functions are summerized in Table 3.3.

Type name	nutkWallFunction	nutkRoughWallFunction
Available scope (first layer cell position)	Log-law region	Log-law region
Class	Foam::nutkWallFunction-FvPatchScalarField	Foam::nutkRoughWallFunction-FvPatchScalarField
Calculate from	k	k
Inherit from	Foam::nutWallFunction-FvPatchScalarField	-
Type name	nutUWallFunction	nutURoughWallFunction
Available scope (first layer cell position)	Log-law region	Log-law region
Class	Foam::nutUWallFunction-FvPatchScalarField	Foam::nutURoughWallFunction-FvPatchScalarField
Calculate from	U	U
Inherit from	Foam::nutWallFunction-FvPatchScalarField	Foam::nutWallFunction-FvPatchScalarField
Type name	nutLowReWallFunction	nutkAtmRoughWallFunction
Available scope (first layer cell position)	Log-law and viscous region	Log-law region
Class	Foam::nutLowReWallFunction-FvPatchScalarField	Foam::nutkAtmRoughWallFunction-FvPatchScalarField
Calculate from	k	k
Inherit from	Foam::nutWallFunction-FvPatchScalarField	Foam::nutWallFunction-FvPatchScalarField
Type name	nutUSpaldingWallFunction	nutUTabulatedWallFunction
Available scope (first layer cell position)	All regions	All regions
Class	Foam::nutUSpaldingWall-FunctionFvPatchScalarField	Foam::nutUTabulatedWall-FunctionFvPatchScalarField
Calculate from	U	U
Inherit from	Foam::nutWallFunction-FvPatchScalarField	Foam::nutWallFunction-FvPatchScalarField

Table 3.3: Available ν_T wall functions in OpenFoam 4.0

These ν_T wall functions can be divided into two categories, i.e., (i) calculated from U (ii) calculated from k, this can be easily identified by checking the type name. nutkWallFunction provides a turbulent kinematic viscosity condition based on turbulence kinetic energy. nutUWallFunction provides turbulent kinematic viscosity condition based on U. In addition, nutkRoughWallFunction and nutkRoughWallFunction manipulate the E parameter to account for the effects of roughness. nutLowReWallFunction provides a turbulent kinematic viscosity condition for low Reynolds number models, it sets ν_T to zero, and provides an access function to calculate y^+. nutUSpaldingWallFunction is used for rough walls to give a continuous ν_T profile to the wall based on one fitting formula of y^+ and u^+ proposed by Spalding (1961). '*nutUTabulatedWallFunction*' needs one user-defined table of U+ as a function of near-wall Reynolds number. The table should be located in the $FOAM_CASE/constant directory.nutkAtmRoughWallFunction provides ν_T for atmospheric veloc-

ity profiles. The '*atmBoundaryLayerInletVelocity*' boundary condition is needed here.

In the present study, a new ν_T wall function is built based on '*nutkWallFunction*' which is explained as follows. The main function is '*calcNut*()', and the value of ν_T will be returned to '*updateCoeffs*()'.

```
tmp<scalarField> nutkWallFunctionFvPatchScalarField::calcNut() const
{
    const label patchi = patch().index();

    const turbulenceModel& turbModel =
    db().lookupObject<turbulenceModel>
    (
        IOobject::groupName
        (
            turbulenceModel::propertiesName,
            internalField().group()
        )
    );

    const scalarField& y = turbModel.y()[patchi];
    const tmp<volScalarField> tk = turbModel.k();
    const volScalarField& k = tk();
    const tmp<scalarField> tnuw = turbModel.nu(patchi);
    const scalarField& nuw = tnuw();

    const scalar Cmu25 = pow025(Cmu_);

    tmp<scalarField> tnutw(new scalarField(patch().size(), 0.0));
    scalarField& nutw = tnutw.ref();

    forAll(nutw, facei)
    {
        label faceCelli = patch().faceCells()[facei];

        scalar yPlus = Cmu25*y[facei]
       *sqrt(k[faceCelli])/nuw[facei];

        if (yPlus > yPlusLam_)
        {
            nutw[facei] =
           nuw[facei]*(yPlus*kappa_/log(E_*yPlus) - 1.0);
        }
    }

    return tnutw;
}
```

This achieves the standard wall function, ν_T is set to zero when $yPlus < yPlusLam\_$, and $\nu_T = \nu * (\frac{\kappa y^+}{ln(Ey^+)} - 1)$ when $yPlus > yPlusLam\_$.

Chapter 4

New Wall Function Implementation for Standard $k-\epsilon$ Turbulence Model in OpenFOAM 4.0

4.1 Implementation of new wall function in OpenFOAM 4.0

As mentioned in Chapter 2, the new wall function includes three section functions which cover all the wall regions. The name of the new wall function is given as kOngWallFunction, epsilonOngWallFunction and nutOngWallFunction (based on Ong et al., 2009). According to equation 2.4, 2.6 and 2.8, the new expression of k, ϵ and ν_T can be obtained. The methodology of the wall function implementation is first calculating turbulence kinetic energy k based on velocity; then ϵ and ν_T are calculated based on k.

First, k is calculated by $k = u^{*2}/\sqrt{C_\mu}$. C_μ is constant. Therefore, the key is how to obtain u^*. Solving the equation set 2.4, 2.6 and 2.8, equation of u^* can be obtained in different regions, see equation 4.1.

$$\begin{cases} u^* = \dfrac{u\nu}{y} & (y^+ < 5) \\ \dfrac{\kappa(yu^*/\nu - 5)yu^*/\nu - (30 - yu^*/\nu)ln(Eyu^*/\nu)}{25yu^*/\nu ln(Eyu^*/\nu)} - \dfrac{u^*}{u} = 0 & (5 <= y^+ <= 30) \\ \dfrac{Eyu^*}{\nu} - exp(\dfrac{\kappa u}{u^*}) = 0 & (y^+ > 30) \end{cases} \tag{4.1}$$

Equation 4.1 shows the u^* expression when y^+ is in different regions. For the case of $y^+ < 5$, the expression of u^* is simple and can be directly used for the calculation of k. However, for the cases of $5 <= y^+ <= 30$ and $y^+ > 30$, u^* can not be solved directly. The approach here is using Newton iteration method to solve the equation of u^*.The equations of u^* are defined as follows:

$$\begin{cases} f1(u^*) = \dfrac{\kappa(yu^*/\nu - 5)yu^*/\nu - (30 - yu^*/\nu)ln(Eyu^*/\nu)}{25yu^*/\nu ln(Eyu^*/\nu)} - \dfrac{u^*}{u} & (5 <= y^+ <= 30) \\ f2(u^*) = \dfrac{Eyu^*}{\nu} - exp(\dfrac{\kappa u}{u^*}) & (y^+ > 30) \end{cases} \tag{4.2}$$

Then the equation set to be solved is:

$$\begin{cases} f1(u^*) = 0 & (5 <= y^+ <= 30) \\ f2(u^*) = 0 & (y^+ > 30) \end{cases} \tag{4.3}$$

According to Newton iteration method, the iteration relation of $f1(u^*)$ and $f2(u^*)$ are:

$$\begin{cases} u^*_{n+1} = u^*_n - \dfrac{f1(u^*_n)}{f1'(u^*_n)} & (5 <= y^+ <= 30) \\ u^*_{n+1} = u^*_n - \dfrac{f2(u^*_n)}{f2'(u^*_n)} & (y^+ > 30) \end{cases} \tag{4.4}$$

where $f1'(u^*_n)$ and $f2'(u^*_n)$ is the derivative with respect to u^*.

$$\begin{cases} f1'(u^*_n) = -\dfrac{1}{u} + \dfrac{(2\kappa y u^*_n/\nu - 5\kappa + ln(Eyu^*_n/\nu))y}{\nu} - \dfrac{(30 - yu^*_n/\nu)}{25(yu^*_n/\nu)u^*_n ln(Eyu^*_n/\nu)} \\ \qquad\qquad \dfrac{(\kappa(5 - yu^*_n/\nu)yu^*_n/\nu + (30 - yu^*_n/\nu)ln(Eyu^*_n/\nu))(y + ln(Eyu^*_n/\nu)y)}{25\nu(yu^*_n/\nu ln(Eyu^*_n/\nu))^2} & (5 <= y^+ <= 30) \\ f2'(u^*_n) = \dfrac{Ey}{\nu} + \dfrac{\kappa u}{u^{*2}_n} exp(\dfrac{\kappa u}{u^*_n}) & (y^+ > 30) \end{cases} \tag{4.5}$$

where u is the velocity, y is the height of the first layer cell, u^* is the only unknown variable. Replace $f1(u^*)$ and $f2(u^*)$ with the expression 4.2, after a certain number of iteration steps, the value of u^* can be obtained. Subsequently k can be obtained by $k = u^{*2}/\sqrt{C_\mu}$. ϵ and ν_T are calculated based on equation (2.4). This section gives a basic idea of implementation. Next section will introduce how to modify the code in OpenFOAM 4.0.

4.2 Modifications to existing wall functions

The Ong wall functions are modified based on kLowReWallFunction, epsilonWallFunction and nutkWallFunction. First, copy these three files to the corresponding folder usig the following commands.

Change work directory to wall function directory.

```
OF4X
cd $FOAM_SRC/TurbulenceModels/turbulenceModels/derivedFvPatchFields/wallFunctions
```

Copy the required wall function files in the same folder.

```
cp -r kqRWallFunctions/kLowReWallFunction kqRWallFunctions/kOngWallFunction
cp -r epsilonWallFunctions/epsilonWallFunction epsilonWallFunctions/\
epsilonOngWallFunction
cp -r nutWallFunctions/nutkWallFunction nutWallFunctions/nutOngWallFunction
```

4.2.1 Modification to kOngWallFunction

Change the .H and .C files' names to the new wall function names.

```
cd kqRWallFunctions/kOngWallFunction
mv kLowReWallFunctionFvPatchScalarField.C kOngWallFunctionFvPatchScalarField.C
mv kLowReWallFunctionFvPatchScalarField.H kOngWallFunctionFvPatchScalarField.H
```

Change all the key words from *kLowReWallFunction* to *kOngWallFunction*

```
sed -i s/kLowReWallFunction/kOngWallFunction/g kOngWallFunctionFvPatchScalarField.C
sed -i s/kLowReWallFunction/kOngWallFunction/g kOngWallFunctionFvPatchScalarField.H
```

'*yPlusLam*' function is not required in the kOngWallFunction, the declaration and definition of '*yPlusLam*' function are deleted in case of any conflict. Then modify the original '*updateCoeffs*()' into:

```
void kMukWallFunctionFvPatchScalarField::updateCoeffs()
{
    if (updated())
    {
        return;
    }

    const label patchi = patch().index();

    const turbulenceModel& turbModel = db().lookupObject<turbulenceModel>
    (
        IOobject::groupName
        (
            turbulenceModel::propertiesName,
            internalField().group()
        )
    );
    const scalarField& y = turbModel.y()[patchi];


    const tmp<volScalarField> tk = turbModel.k();
    const volScalarField& k = tk();

    const tmp<scalarField> tnuw = turbModel.nu(patchi);
    const scalarField& nuw = tnuw();

    const scalar Cmu25 = pow025(Cmu_);

    const fvPatchVectorField& Uw = turbModel.U().boundaryField()[patchi];
    const scalarField magUp(mag(Uw.patchInternalField() - Uw));
    const scalarField magGradU(mag(Uw.snGrad()));
    scalarField& kw = *this;
```

```
    // Set k wall values
    forAll(kw, facei)
    {
       label faceCelli = patch().faceCells()[facei];

        scalar uTau = Cmu25*sqrt(k[faceCelli]);

        scalar yPlus = uTau*y[facei]/nuw[facei];
      tmp<scalarField> tuTau = calcUTau(magGradU);
    scalarField& uts = tuTau.ref();

     tmp<scalarField> tuTau2 = calcUTau2(magGradU);
    scalarField& uts2 = tuTau2.ref();

         if (yPlus <= 5)
        {

            kw[facei] = magUp[facei]*nuw[facei]/y[facei];
        }
        else if (yPlus > 5 && yPlus < 30)
      {

       kw[facei] = uts2[facei]*uts2[facei];

        }
        else
        {

       kw[facei] = uts[facei]*uts[facei];

        }

        kw[facei] /= sqrt(Cmu_);
    }

    kw = max(kw, SMALL);

  fixedValueFvPatchField<scalar>::updateCoeffs();

}
```

This is the main function calculating k according to equation (2.4), (2.6) and (2.8). Two new member functions are used here to calculate u^* in different regions. Member function *calcUTau* is defined to calculate the u^* in Log-law region. Then the value of u^* will be returned back to the *else* statement in $updateCoeffs()$:

```
        else
         {

       kw[facei] = uts[facei]*uts[facei];

        }
```

The other member function '*calcUTau*2' is defined to calculate the u^* in buffer layer region and the value of u^* will be returned back to the '*else if*' statement in '*updateCoeffs*()':

```
    else if (yPlus > 5 && yPlus < 30)
     {

     kw[facei] = uts2[facei]*uts2[facei];

      }
```

The definition of *calcUTau* is shown as follows:

```
tmp<scalarField> kMukWallFunctionFvPatchScalarField::calcUTau
(
    const scalarField& magGradU
) const
{
    const label patchi = patch().index();

    const turbulenceModel& turbModel = db().lookupObject<turbulenceModel>
    (
        IOobject::groupName
        (
            turbulenceModel::propertiesName,
            internalField().group()
        )
    );
    const scalarField& y = turbModel.y()[patchi];

    const fvPatchVectorField& Uw = turbModel.U().boundaryField()[patchi];
    const scalarField magUp(mag(Uw.patchInternalField() - Uw));

    const tmp<scalarField> tnuw = turbModel.nu(patchi);
    const scalarField& nuw = tnuw();

    const scalarField& nutw = *this;

    tmp<scalarField> tuTau(new scalarField(patch().size(), 0.0));
    scalarField& uTau = tuTau.ref();

    forAll(uTau, facei)
    {
        scalar ut = sqrt((nutw[facei] + nuw[facei])*magGradU[facei]);

        if (ut > ROOTVSMALL)
        {
            int iter = 0;
            scalar err = GREAT;

            do
            {
                scalar kUu = max(kappa_*magUp[facei]/ut, 13.86);
                scalar fkUu = exp(kUu);
```

```
                scalar f =
                    - ut*y[facei]/nuw[facei]+ 1/E_*fkUu;

                scalar df =
                    y[facei]/nuw[facei] + 1/E_*kUu*fkUu/ut;

                scalar uTauNew = ut + f/df;
                err = mag((ut - uTauNew)/ut);
                ut = uTauNew;

            } while (ut > ROOTVSMALL && err > 0.01 && ++iter < 10);

            uTau[facei] = max(0.0, ut);
        }
    }

    return tuTau;
}
```

First u^* is assigned with a initial value. ROOTVSMALL and GREAT are simply constant defined in

`src/OpenFOAM/primitives/Scalar/scalar/scalar.H`
`src/OpenFOAM/primitives/Scalar/floatScalar/floatScalar.H`

The value of ROOTVSMALL is 1.0e-18, GREAT is 1.0e+6. u^+ is defined as kUu here. This function is used when y^+ is larger the 30(corresponding $u^+ > 13.86$), therefore, the value of u^+ keeps larger than 13.86. The relative error is defined as $err = \frac{|u_n^* - u_{n+1}^*|}{u_n^*}$. Then do at least 10 steps interation of $f2(u^*)$(refer to equation(4.4) which is introduced in Section 4.1) until the relative error is less than 0.01. Then return back the value of u^*.
The definition of '$calcUTau2$' is shown as follows:

```
tmp<scalarField> kMukWallFunctionFvPatchScalarField::calcUTau2
(
    const scalarField& magGradU
) const
{
    const label patchi = patch().index();

    const turbulenceModel& turbModel = db().lookupObject<turbulenceModel>
    (
        IOobject::groupName
        (
            turbulenceModel::propertiesName,
            internalField().group()
        )
    );
    const scalarField& y = turbModel.y()[patchi];

    const fvPatchVectorField& Uw = turbModel.U().boundaryField()[patchi];
```

```
    const scalarField magUp(mag(Uw.patchInternalField() - Uw));

    const tmp<scalarField> tnuw = turbModel.nu(patchi);
    const scalarField& nuw = tnuw();

    const scalarField& nutw = *this;

    tmp<scalarField> tuTau2(new scalarField(patch().size(), 0.0));
    scalarField& uTau2 = tuTau2.ref();

    forAll(uTau2, facei)
    {
        scalar ut2 = sqrt((nutw[facei] + nuw[facei])*magGradU[facei]);

        if (ut2 > ROOTVSMALL)
        {
            int iter = 0;
            scalar err = GREAT;

            do
            {

            scalar lg = log(E_*y[facei]*ut2/nuw[facei]);
            scalar yp = min(y[facei]*ut2/nuw[facei], 30);
                   yp = max(5,yp);

                scalar f =
                    -ut2/magUp[facei]+
                    (kappa_*(yp-5)*yp-(30-yp)*lg)/(25*yp*lg);

                scalar df1=
                   ((-2*kappa_*yp+5*kappa_-lg)*y[facei]/nuw[facei]
                   +(30-yp)/ut2)/(25*yp*lg);
               scalar df2=
                   -(-kappa_*(yp-5)*yp+(30-yp)*lg)*(yp/ut2+lg*y[facei]
                   /nuw[facei]) /(25*sqr(yp*lg));
                scalar df =
                  1/magUp[facei]+df1+df2;

                scalar uTauNew = ut2 + f/df;
                err = mag((ut2 - uTauNew)/ut2);
                ut2 = uTauNew;

            } while (ut2 > ROOTVSMALL && err > 0.01 && ++iter < 10);

            uTau2[facei] = max(0.0, ut2);
        }
    }

    return tuTau2;
}
```

The structure of '*calcUTau*2' function is similar with '*calcUTau*', the difference is the iteration part

which uses the Newton iteration of $f1(u^*)$ (refer to equation(4.4)).

The declaration of these functions must be added into kOngWallFunctionFvPatchScalarField.H:

```
virtual tmp<scalarField> calcUTau(const scalarField& magGradU) const;
virtual tmp<scalarField> calcUTau2(const scalarField& magGradU) const;
```

The modification to kOngWallFunction has been completed so far.

4.2.2 Modification to epsilonOngWallFunction

The ϵ expression in epsilonOngWallFunction is the same with which in epsilonWallFunction:

```
    // Set epsilon and G
    forAll(nutw, facei)
    {
        label celli = patch.faceCells()[facei];

        scalar w = cornerWeights[facei];

        epsilon[celli] += w*Cmu75*pow(k[celli], 1.5)/(kappa_*y[facei]);

        G[celli] +=
            w
           *(nutw[facei] + nuw[facei])
           *magGradUw[facei]
           *Cmu25*sqrt(k[celli])
           /(kappa_*y[facei]);
    }
```

Therefore, make the epsilonOngWallFunction a simple wrapper of epsilonWallFunction. The commands used here are:

```
cd epsilonWallFunctions/epsilonOngWallFunction

mv epsilonWallFunctionFvPatchScalarField.C\
epsilonOngWallFunctionFvPatchScalarField.C
mv epsilonWallFunctionFvPatchScalarField.H\
epsilonOngWallFunctionFvPatchScalarField.H
```

Change the file name first, then rename all the class name in the .C and .H file.

```
sed -i s/epsilonWallFunction/epsilonOngWallFunction/g\
epsilonOngWallFunctionFvPatchScalarField.C
sed -i s/epsilonWallFunction/epsilonOngWallFunction/g\
epsilonOngWallFunctionFvPatchScalarField.H
```

4.2.3 Modification to nutOngWallFunction

Similarly, the files and classes should be renamed first.

```
cd nutWallFunctions/nutOngWallFunction

mv nutkWallFunctionFvPatchScalarField.C nutOngWallFunctionFvPatchScalarField.C
mv nutkWallFunctionFvPatchScalarField.H nutOngWallFunctionFvPatchScalarField.H

sed -i s/nutkWallFunction/kOngWallFunction/g nutOngWallFunctionFvPatchScalarField.C
sed -i s/nutkWallFunction/kOngWallFunction/g nutOngWallFunctionFvPatchScalarField.H
```

Then, the main function '*calcNut*()' should be renewed to the following one.

```
tmp<scalarField> nutMukWallFunctionFvPatchScalarField::calcNut() const
{
    const label patchi = patch().index();

    const turbulenceModel& turbModel = db().lookupObject<turbulenceModel>
    (
        IOobject::groupName
        (
            turbulenceModel::propertiesName,
            internalField().group()
        )
    );

    const scalarField& y = turbModel.y()[patchi];
    const tmp<volScalarField> tk = turbModel.k();
    const volScalarField& k = tk();
    const tmp<scalarField> tnuw = turbModel.nu(patchi);
    const scalarField& nuw = tnuw();

    const scalar Cmu25 = pow025(Cmu_);

    tmp<scalarField> tnutw(new scalarField(patch().size(), 0.0));
    scalarField& nutw = tnutw.ref();

    forAll(nutw, facei)
    {
        label faceCelli = patch().faceCells()[facei];

        scalar yPlus = Cmu25*y[facei]*sqrt(k[faceCelli])/nuw[facei];

        nutw[facei] = (1-exp(-0.0002*yPlus-0.00065*sqr(yPlus)))
        *Cmu25*y[facei]*sqrt(k[faceCelli])*kappa_;
    }

    return tnutw;
}
```

Here ν_T is calculated by equation (2.4).

4.2.4 Compile Ong wall functions in OpenFOAM 4.0

First change the working directory to turbulenceModels/Make:

```
cd $FOAM_SRC/TurbulenceModels/turbulenceModels/Make
```

Open the 'files' file, add the following statement inside under the 'wallFunctions = derivedFvPatch-Fields/wallFunctions' accordingly.

```
$(nutWallFunctions)/nutOngWallFunction/nutOngWallFunctionFvPatchScalarField.C
$(epsilonWallFunctions)/epsilonOngWallFunction\
/epsilonOngWallFunctionFvPatchScalarField.C
$(kqRWallFunctions)/kOngWallFunction/kOngWallFunctionFvPatchScalarField.C
```

Then change the last line 'LIB = $(FOAM_LIBBIN)/libturbulenceModels' to 'LIB = $(FOAM_ USER _LIBBIN)/ libturbulenceModels' Touch the change of wall functions:

```
cd $FOAM_SRC/TurbulenceModels
wclean
touch turbulenceModels/derivedFvPatchFields/wallFunctions\
/epsilonWallFunctions/epsilonOngWallFunction\
/epsilonOngWallFunctionFvPatchScalarField.C
touch turbulenceModels/derivedFvPatchFields/wallFunctions\
/epsilonWallFunctions/epsilonOngWallFunction\
/epsilonOngWallFunctionFvPatchScalarField.H
touch turbulenceModels/derivedFvPatchFields/wallFunctions\
/kqRWallFunctions/kOngWallFunction\
/kOngWallFunctionFvPatchScalarField.C
touch turbulenceModels/derivedFvPatchFields/wallFunctions\
/kqRWallFunctions/kOngWallFunction\
/kOngWallFunctionFvPatchScalarField.H
touch turbulenceModels/derivedFvPatchFields/wallFunctions\
/nutWallFunctions/nutOngWallFunction\
/nutOngWallFunctionFvPatchScalarField.C
touch turbulenceModels/derivedFvPatchFields/wallFunctions\
/nutWallFunctions/nutOngWallFunction\
/nutOngWallFunctionFvPatchScalarField.H
```

Compile the turbulence model by using the following commands.

```
cd $FOAM_SRC/TurbulenceModels
wmake libso turbulenceModels/derivedFvPatchFields/wallFunctions\
/nutWallFunctions/nutOngWallFunction
wmake libso turbulenceModels/derivedFvPatchFields/wallFunctions\
/kqRWallFunctions/kOngWallFunction
wmake libso turbulenceModels/derivedFvPatchFields/wallFunctions\
```

```
/epsilonWallFunctions/epsilonOngWallFunction
./Allwmake
```

After compiling successfully, restart the terminal window and prepare for the case Test.

Chapter 5

Test Cases

A verification study will be performed to ensure that the new wall function (Ong wall function) is implemented correctly. Two test cases are introduced here, i.e. (i) Case 1: The uniform velocity flow past a long flat plate; (ii) Case 2: A fully developed boundary layer flow past a short flat plate. A summary of test cases are shown in Table 5.1.

Case	Test case 1	Test case 2
Fisrt layer cell position	buffer layer	buffer layer
First layer cell height	0.0005	0.0005
Mesh quantity	244600	143700

Table 5.1: Summary of test cases

5.1 Test case 1

5.1.1 Case set up

A uniform velocity profile is introduced in the inlet. The main improvement of Ong wall function is including the buffer layer. Therefore, in these two test cases, the first layer cell is set in buffer layer ($5 < y^+ < 30$). The set up of Case 1 is shown as Figure 5.1. The outlet velocity profile are used to check whether the newly-implemented wall function can produce physically-sound velocity profiles.

The calculation process of inlet velocity can be concluded as: (1) According to the experience, assume that at the position $y = 0.22$ it is the switching point to fully developed flow; (2) According the wall function equation (2.8), the value of u^* can be calculated; (3) Combine the equation (2.4) (2.6)and (2.8) with the known u^*, the velocity profile can be obtained. The key step during this process is selecting the position of first point, we must make sure that this point is above the real boundary layer switching point, thus, it will not cause that the boundary layer region is forced to be fully developed flow.

The computational domain of a simple 2-D flow along a horizontal plate is shown below.

where H is the height of the 2D domain. The inlet velocity is uniform value, the bottom is set as

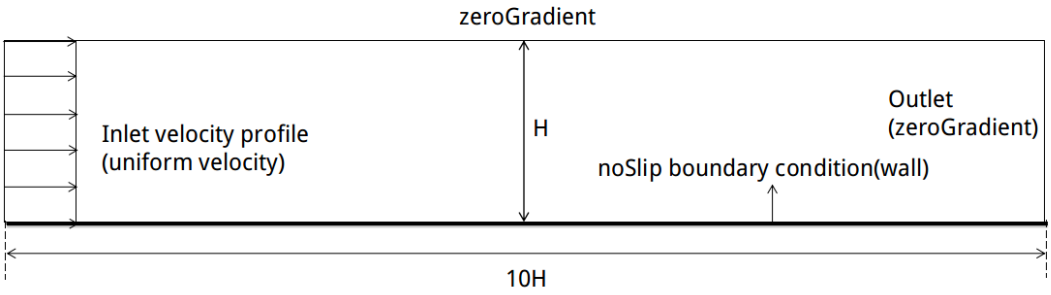

Figure 5.1: Test case 1 set up

noSlip (wall), top and outlet is set as zeroGradient (patch). The summary of the test cases is shown as Table 5.1. At inlet boundary, the following expression are used for k, ϵ and ν_T (Ong et al., 2009).

$$\begin{aligned} k &= \frac{3}{2}(I_u U_\infty)^2 \\ \epsilon &= \frac{C_\mu k^(3/2)}{0.1L} \\ \nu_T &= C_\mu \frac{k^2}{\epsilon} \end{aligned} \tag{5.1}$$

For the 0 folder, the setting for U, p, k, ϵ and ν_T are required. On the wall, the k, ϵ and ν_T are set to the wall function name we defined (kOngWallFunction, epsilonOngWallFunction, nutOngWallFunction), ν_T are set to '*calculated*' on inlet patches. Detailed setting in 0, cnstant and system folder can be found in Appendix C.

After finishing the setting, run *simpleFoam* in the test case folder.

5.1.2 Post-processig in paraFoam

When the simulation is done, type *paraFoam* in terminal. Click 'Apply', then select U in the upper toolbar. Click 'slice' then input the coordinate (9, 0.5, 0.5) and apply.
Then click '$splithorizontal' -> ' spreadsheetview$', the data can be seen as Figure 5.3. Then save the data to csv format file by '$File -> savedata'$. In this way, after plate flow is stable, outlet velocity profile can be compared with the theoretical value to check whether this new wall function is applicable.

According to the stable velocity profile and the equations of Ong wall function, the theoretical velocity profile can be obtained. Different wall function will cause a big difference in the real simulation especially for the cases that the separating point varies with the flow.

The results comparison of present simulated velocity profile and the velocity profile calculated based on Ong et al., (2009) is shown as Figure 5.4.

It can be seen that he results of simulated results and the calculated results do not agree so good. The reason is that the plate flow is still not fully developed, which can be seen by comparing the velocity profiles between $x = 8H$ and $x = 10H$. It means the flow around the horizontal plate needs longer computation domain and longer time to achieve the fully developed boundary layer. In addition, the meshes must be fine enough to capture the velocity profile. Therefore, Case 2 is suggested for further verification study.

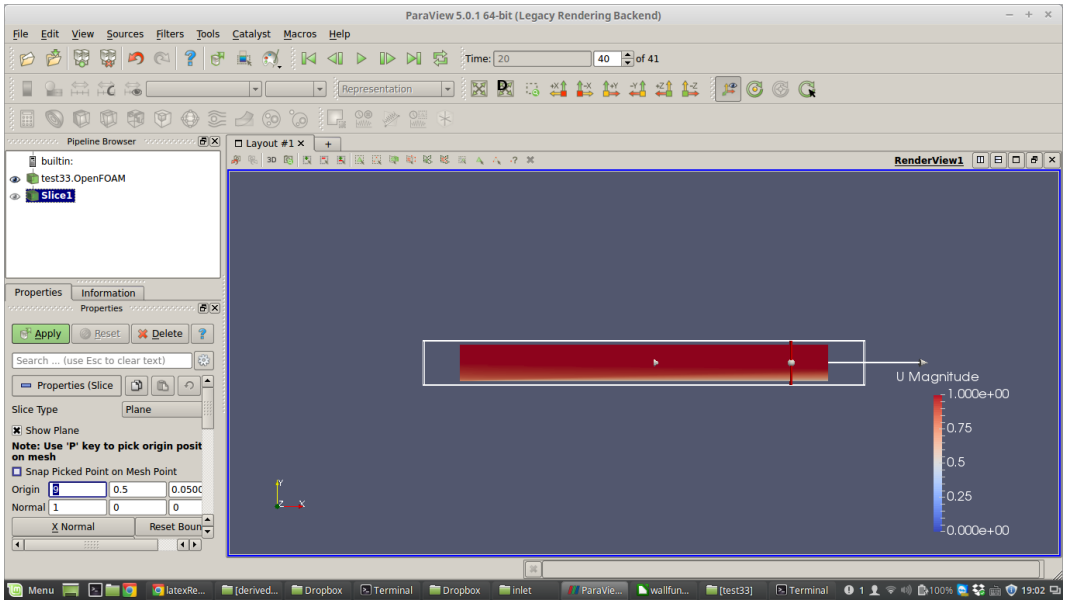

Figure 5.2: Slice in paraview a

5.2 Test Case 2

5.2.1 Case set up

Test Case 1 requires lone simulation time to achieve the converged results, Test Case 2 is less time-consuming and also a good way to verify the newly implemented wall function. The basic idea is to calculate a fully developed boundary layer flow according to the equations of Ong wall function. Set this boundary layer flow on the inlet pathes, and compare the velocity profile of inlet and outlet. If these two velocity profiles fit well, it means the new wall function can maintain the velocity and produce the correct velocity profile.

The case set up is shown in Figure 5.5. The inlet velocity in case 2 is a boundary layer flow. A nonuniform setting for U, k, ϵ and ν_T is required in 0' folder. In addition, boundaryData folder is required in 'constant' folder, which contains all the nonuniform inlet information. This process can also be done by swak4Foam, but in the present study, the data points are used to set up the boundary condition at the inlet. The inlet velocity profile calculated from the Ong wall function is shown in Figure 5.6. The detailed setting can be found in Attachment C.

The inlet boundary conditions for k, ϵ and ν_T are based on Ong et al., (2010) :

$$\begin{aligned} k &= max\{C_\mu^(- 1/2)(1 - y/y^f)^2 u^{*2}, 0.0001U_\infty^2\} \\ \epsilon &= \frac{C_\mu (3/4) k y^{3/2}}{l} \\ \nu_T &= C_\mu \frac{k^2}{\epsilon} \end{aligned} \tag{5.2}$$

where U_∞ is the velocity of infinity. y^f is the height of first layer cell. The expression of l is:

$$l = min\{ky(1 + 3.5y/y^f)^{-1}, C_\mu y\} \tag{5.3}$$

Run *simpleFoam* in the main folder.

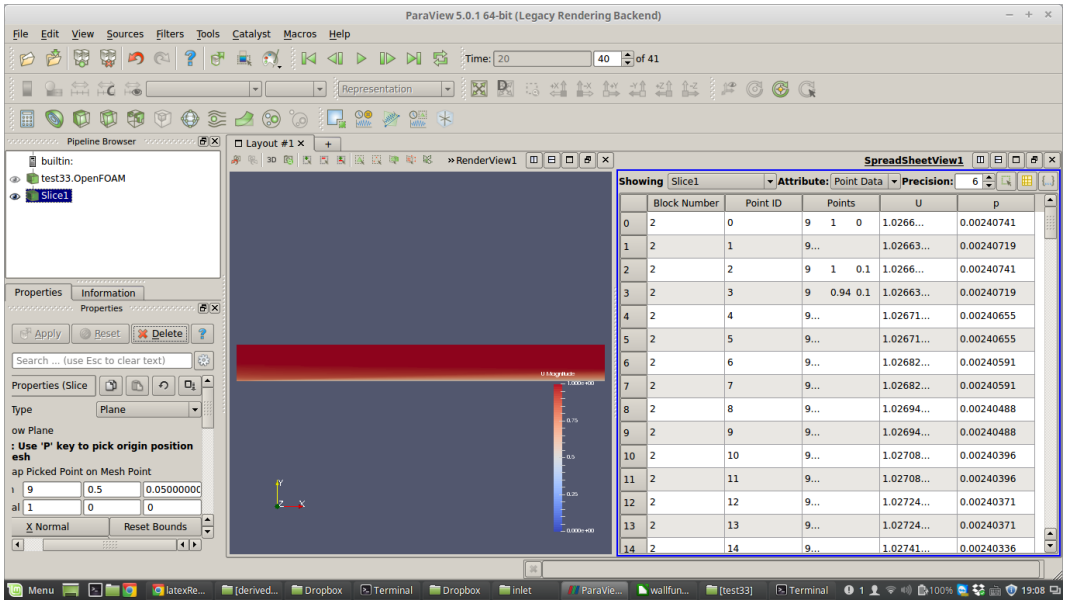

Figure 5.3: Slice in paraview b

5.2.2 Results

The post-processing is similar with those shown section 5.1.2. The comparison of inlet and outlet velocity profile is shown in Figure 5.7.

The result shows that the Ong wall functions keep the fully-developed velocity profile stable. It also means that the new wall function is implemented successfully.

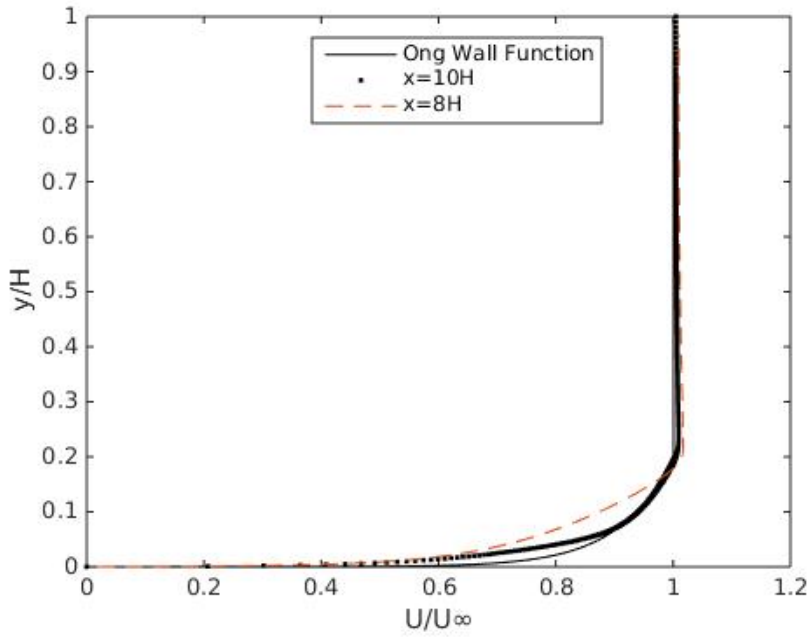

Figure 5.4: Inlet velocity profile

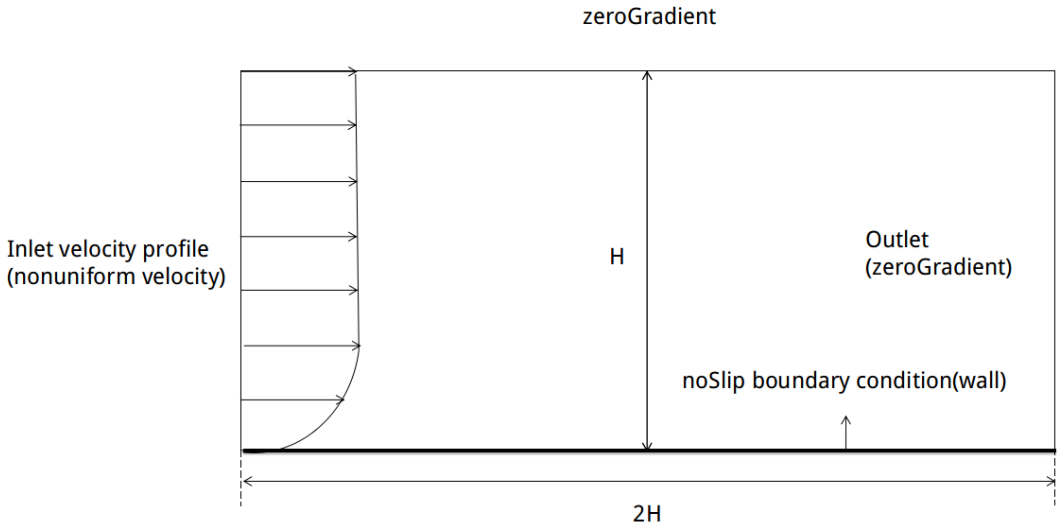

Figure 5.5: Test case 2 set up

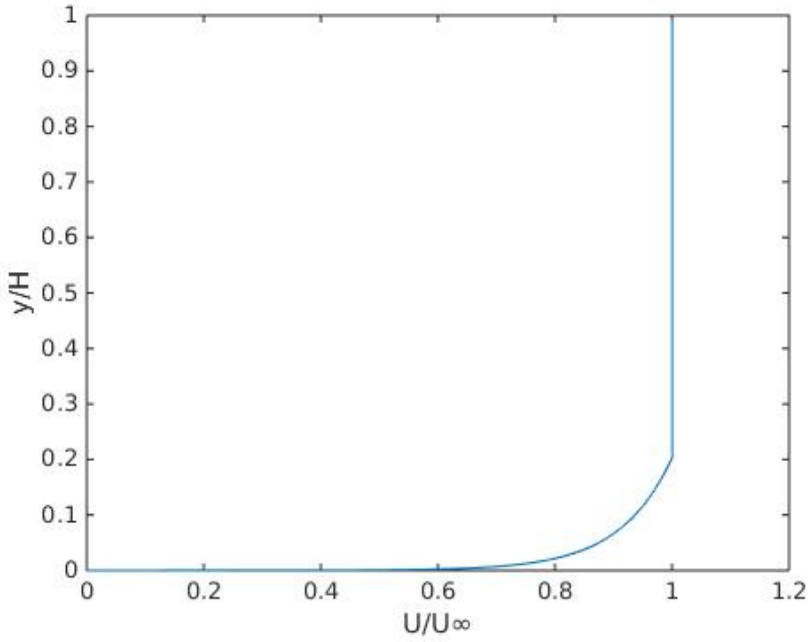

Figure 5.6: Inlet velocity profile

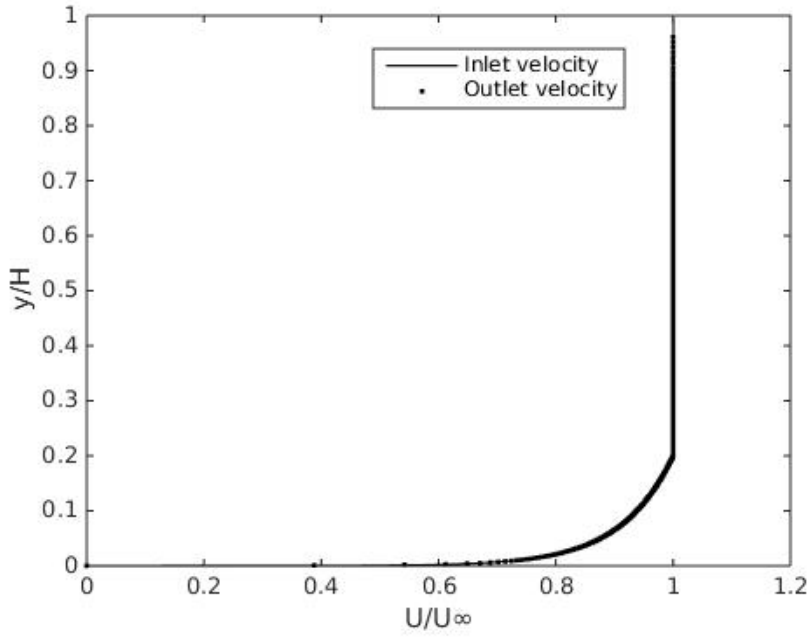

Figure 5.7: Inlet and outlet velocity profile

Study questions

1. How many near-wall regions are usually used around the wall? What are them? What does the division depends on?
2. Which folder are the wall functions situated?
3. What commands should be used to copy existing wall functions to an aimed folder? Please give an example.
4. How can you add non-uniform inlet velocity? Please give more available ways.
5. How to verify a newly complemented wall function? Please describe what kind of case should be used and what results are expected?

Reference

Balogh, M., Parente, A. and Carlo B. "RANS simulation of ABL flow over complex terrains applying an enhanced $k-\epsilon$ model and wall function formulation: Implementation and Comparison for Fluent and OpenFOAM." Journal of Wind Engineering and Industrial Aerodynamics 104 (2012): 360-368.

Jones, W. P., and Launder B.E. "The prediction of laminarization with a two-equation model of turbulence." International journal of heat and mass transfer 15.2 (1972): 301-314.

Kalitzin, G., Medic, G., Iaccarino, G. and Durbin, P. "Near-wall behavior of RANS turbulence models and implications for wall functions." Journal of Computational Physics 204.1 (2005): 265-291.

Launder, B. E., and Sharma, B. I. "Application of the energy-dissipation model of turbulence to the calculation of flow near a spinning disc." Letters in heat and mass transfer 1.2 (1974): 131-137.

Liestyarini, U. C. CFD Analysis of Internal Pipe Flows, Master Thesis, University of Stavanger, Norway (2016).

Ong, M.C., Trygsland, E. and Myrhaug, D. "Numerical Study of Seabed Boundary Layer Flow around Monopile and Gravity-Based Wind Turbine Foundation", Proceedings of 35th International Conference on Ocean, Offshore and Arctic Engineering, ASME, OMAE2016-54643, Busan, Korea (2016).

Ong, M.C., Utnes, T., Holmedal, L.E., Myrhaug, D. and Pettersen, B. "Numerical simulation of flow around a smooth circular cylinder at very high Reynolds numbers." Marine Structures 22.2 (2009): 142-153.

Ong, M.C., Utnes, T., Holmedal, L.E., Myrhaug, D. and Pettersen, B. "Numerical simulation of flow around a circular cylinder close to a flat seabed at high Reynolds numbers using a $k-\epsilon$ model." Coastal Engineering 57.10 (2010): 931-947.

Parente, A., Gorle, C., van Beeck, J. and Benocci, C. "Improved $k-\epsilon$ model and wall function formulation for the RANS simulation of ABL flows." Journal of wind engineering and industrial aerodynamics 99.4 (2011): 267-278.

Rodi, W., and Mansour, N. N. "Low Reynolds number $k-\epsilon$ modelling with the aid of direct simulation data." Journal of Fluid Mechanics 250 (1993): 509-529.

Spalding, D. B. "A single formula for the "law of the wall". "Journal of Applied Mechanics 28.3 (1961): 455-458.

Tennekes, H., and Lumley, J.L. A first course in turbulence. MIT press, 1972.