Cite as: Fangqing Liu.: A Thorough Description Of How Wall Functions Are Implemented In OpenFOAM.
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

Project work:

A Thorough Description Of How Wall Functions Are Implemented In OpenFOAM

Developed for OpenFOAM-4.0.x

Author:
Fangqing Liu
Chalmers university of techology
fangqing@student.chalmers.se

Peer reviewed by:
Håkan Nilsson
Mohsen Irannezhad

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.

January 22, 2017

Learning outcomes

This tutorial will mainly teach four points: How to use it, The theory of it, How it is implemented, and How to modify it. Therefore the list of learning outcomes is organized with those headers.

The reader will learn:

How to use it:

- How to set wall functions as boundary conditions.

The theory of it:

- The basic principle and derivation of wall functions.

How it is implemented:

- The difference of wall functions between theroy and codes in OpenFOAM.
- How a solver set boundary condition using wall function.

How to modify it:

- The first step of how to implement a new wall function.

Contents

Chapter 1

General theoretical description about wall functions

1.1 Introduction

In the present report simple principle of wall functions will be illustrated and the details of the wall functions code in OpenFOAM are explained. The way how wall functions are used in solver will be illustrated, together with the first step of implementing a new wall function.

1.2 Why using wall functions approach

Nowadays turbulence models are heavily used when doing CFD simulations. However, some mature turbulence models such as k-ε are only valid in the area of turbulence fully developed, and don't perform well in the area close to the wall. In order to deal with the near wall region, two ways are usually proposed. One way is to integrate the turbulence to the wall. Turbulence models are modified to enable the viscosity-affected region to be resolved with all the mesh down to the wall, including the viscous sublayer. When using modified low Reynolds turbulence model to solve the near-wall region the first cell center must be placed in the viscous sublayer (preferably y^+=1). This approach leads to requirement of abundant mesh number, which means a substantial computational resource is needed. Another way is to use so-called wall functions, which can model the near wall region. Wall functions are empirical equations used to satisfy the physics of the flow in the near wall region and the first cell center needs to be placed in the log-law region to ensure the accuracy of the result. Wall functions are used to bridge the inner region between the wall and the turbulence fully developed region, in order to provide near-wall boundary conditions for the momentum and turbulence transport equations, rather than to specify those conditions at the wall itself[1].
It can be noted that when using wall functions approach there is no need to resolve the boundary layer, which yields to a significant reduction of the mesh size and the computational domain. Despite wall functions are based on the empirical relations that are only valid under some conditions, the result is relatively accurate if used correctly ([2]).
The wall function approach was first proposed by Launder and Spalding in 1972 ([3]) and the requirement of the earliest wall functions is the first cell center at the wall to be located in the logarithmic layer. If the cell center lies in the viscous sublayer, the results from wall functions approach are very inaccurate. However, that is a very strict constraint which will often be violated by the grid. Thus some noble wall functions are given to reduce this restrain.
In OpenFOAM the existing wall functions have been modified to ensure that they can provide the accurate result so wherever the position of the first cell center. The theory of wall functions used in OpenFOAM is based on the paper proposed by Georgi Kalitzin ([4]).

1.3 What are wall functions

The wall boundary layer, together with some important parameters which are important to define and derive wall functions are illustrated. Then the process of wall functions derivation are explained. It should be noted that the main purpose of introducing the derivation of wall functions is to compare them with that in OpenFOAM to check how wall functions are implemented.

1.3.1 The near wall region

To know in which condition wall functions approach can be used correctly, the near-wall region should be introduced. Some parameters which be used to introduce the near wall region ([1] [2]).

$$y^+ = \frac{y \times u_\tau}{\nu} \tag{1.1}$$

$$u_\tau = \sqrt{\frac{\tau_w}{\rho}} \tag{1.2}$$

where u_τ is the friction velocity, which can be used to define dimensionless velocity later. τ_w is the wall shear stress, y is the distance to the wall and y^+ is a non-dimensional. The viscosity and density of fluid which are used in equation (1.1) and (1.2) are represented by greek letter nu and rho differently. The value of y^+ of the first cell is essential since it judge where the first cell located. The dimensionless velocity is given by

$$u^+ = \frac{u}{u_\tau} \tag{1.3}$$

Then it comes to the near-wall boundary layer.

The near-wall region can be divided into 3 parties: the viscous sub-layer, the buffer layer and the logarithmic region. In figure 1.1 these sublayers, together with the relationship between y^+ and u^+ is demonstrated[1].

1. The viscous sublayer ($y^+ < 5$)
   In the viscous layer the fluid is dominated by the viscous affect so it can be assumed that the shear stress of fluid is equal to the wall shear stress τ_w. In viscous layer viscous stress decide the flow and the velocity profile is liner, given by

   $$u^+ = y^+ \tag{1.4}$$

   In figure 1.1, the blue line in the viscous sublayer represents the liner relation.

2. The logarithmic area ($30 < y^+ < 200$).
   In the logarithmic layer turbulence stress dominate the flow and velocity profile varies very slowly with a logarithmic function along the distance y, given by

   $$u^+ = \frac{1}{\kappa} \times In(Ey^+) \tag{1.5}$$

   where k is the von karmon constant equal to 0.41 and E is equal to 9.8 for the smooth walls. In figure 1.1, the red line in the logarithmic region represents the logarithmic relation.

3. The Buffer layer
   Viscous and turbulent stresses are of similar magnitude and since it is complex velocity profile is not well defined and the original wall functions avoid the first cell center located in this region. However, the improved wall functions allow the first cell center is located in buffer layer. In OpenFOAM buffer layer is divided 2 parties. One is using the liner relation as that in viscous sublayer, the other is to use logarithmic function as that in logarithmic region.

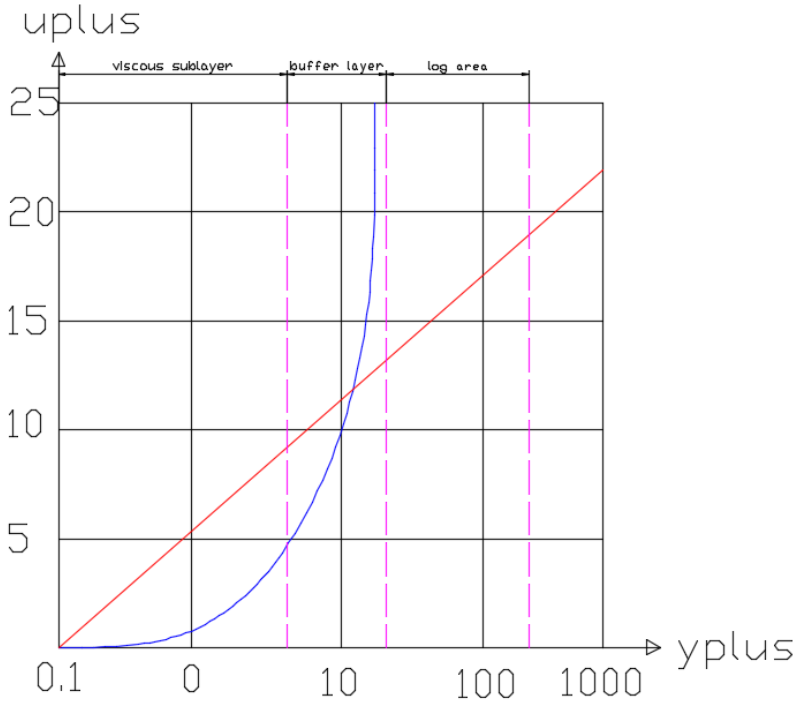

Figure 1.1: Relation between uplus and yplus in three parties of inner region

1.3.2 Derivations of wall functions

The process of deriving wall functions, including ε, f, k, ν, $\overline{v^2}$ and ω are here explained. Velocity and turbulence parameters are given in non-dimensional form , given by

$$U^+ = \frac{U}{u_\tau};\ k^+ = \frac{k}{{u_\tau}^2};\ \varepsilon^+ = \frac{\varepsilon\nu}{{u_\tau}^4};\ \overline{v^2}^+ = \frac{\overline{v^2}}{{u_\tau}^2};\ f^+ = \frac{f\nu}{{u_\tau}^2};\ \omega^+ = \frac{\nu\omega}{{u_\tau}^2} \tag{1.6}$$

Two turbulence models are discussed. The first one is $v^2 - f$ turbulence model which needs wall functions k, ε, v^2 and f wall functions. The second one is $k - \omega$ which needs wall functions (the greek letter omega)ω.

The $v^2 - f$ turbulence model

In the viscous sub-layer, the $v^2 - f$ turbulence model equations can be simplified as ([4])

$$-\varepsilon^+ + \frac{d^2k^+}{(dy^+)^2} = 0 \tag{1.7}$$

$$-\frac{C_{\varepsilon 2}}{6}(\varepsilon^+)^{1.5} + \frac{d^2\varepsilon^+}{(dy^+)^2} = 0 \tag{1.8}$$

$$-N\frac{\varepsilon^+(\overline{v^2})^+}{k^+} + k^+f^+ + \frac{d^2(\overline{v^2})^+}{(dy^+)^2} = 0 \tag{1.9}$$

$$\frac{2}{3}(C_{f1} - 1) - (N - C_{f1})\frac{(\overline{v^2})^+}{k^+} - \frac{6}{(\varepsilon^+)^{0.5}}f^+ + \frac{C_\eta^2 C_L^2 6}{\varepsilon^+}\frac{d^2f^+}{(dy^+)^2} = 0 \tag{1.10}$$

Based on the above equations ε^+ is given by

$$\varepsilon^+ = \frac{14400}{C_{\varepsilon 2}^2}\frac{1}{(y^+ + C)^4} \tag{1.11}$$

With the boundary condition $k^+(0)$ is equal to zero and the gradient $dk^+/dy^+(0)$ is zero, we can use result of ε^+ to do the integration analytically in order to get k^+.
So the k^+ is given by

$$k^+ = \frac{2400}{C_{\varepsilon 2}^2} \times [\frac{1}{(y^+ + C)^4} + \frac{2y^+}{C^3} - \frac{1}{C^2}] \tag{1.12}$$

A local quadratic behavior is imposed on k^+ by assuming $k^+ = C_k\ (y^+)^2$ and put this in the turbulence equations we get a unique condition for ε^+, as

$$\varepsilon^+ = 2\frac{k^+}{(y^+)^2} \tag{1.13}$$

The boundary condition f^+ is derived by assuming that $\overline{(v^2)^+}$ behaves locally as $\overline{(v^2)}^+ = C_{V^2}((y^+)^4)$ and put it in equations of the $v^2 - f$ model and then getting the f^+ [5].

$$f^+ = \frac{-4(6-N)\overline{(v^2)}^+}{\varepsilon^+(y^+)^4} \tag{1.14}$$

In the logarithmic layer, it can be assumed that the diffusion term is small, yielding

$$\kappa y^+ \frac{dk^+}{dy^+} = C_k \tag{1.15}$$

Integrating equation (1.15) k^+ is given by

$$k^+ = \frac{C_k}{\kappa} log(y^+) + B_k \tag{1.16}$$

$\overline{v^2}^+$ is derived in the same way as

$$\overline{v^2}^+ = \frac{C_{V^2}}{\kappa} log(y^+) + B_{V^2} \tag{1.17}$$

The ε^+ is given by

$$\varepsilon^+ = \frac{1}{\kappa y^+} \tag{1.18}$$

f^+ is given by

$$f^+ = N\frac{\overline{v^2}^+}{(k^+)^2}\varepsilon^+ \tag{1.19}$$

In these derived equations some constants used are listed in table 1.1. These values are the same as that in OpenFOAM.

Table 1.1: The constants values

$C_k = -0.416$	$B_k = 8.366$	$C_{V^2} = 0.193$	$B_{V^2} = -0.940$

The k-ω turbulence model

Similar to the $v^2 - f$ turbulence model, $k - \omega$ model equations are simplified to get the behaviour of ω in viscous sub-layer and logarithmic region.
In the viscous layer the standard equations of $k - \omega$ can be simplified as

$$C_\mu \omega^+ k^+ + \frac{d^2 k^+}{(dy^+)^2} = 0 \tag{1.20}$$

$$-\beta_1(\omega^+)^2 + \frac{d^2\omega^+}{(dy^+)^2} = 0 \tag{1.21}$$

According to the above two equations the singular solution of ω^+ is given by

$$\omega^+ = \frac{6}{\beta_1(y^+)^2} \tag{1.22}$$

In the logarithmic layer, ω^+ is given by

$$\omega^+ = \frac{1}{\kappa\sqrt{C_\mu}y^+} \tag{1.23}$$

In the intermediate region ω^+ is usually obtained according a combination of the viscous sub-layer and the log-law layer value, which is given by

$$\omega^+ = \sqrt{[\omega^+_{vis}]^2 + [\omega^+_{log}]^2} \tag{1.24}$$

Chapter 2

Wall functions in OpenFOAM

In this chapter the details of wall functions is explained. Firstly, there are some knowledge background needed to be introduced.

Wall functions are boundary conditions and in OpenFOAM all of them inherit from the abstract class `FvPatchField`. They provide Dirichlet and Neumann boundary condition, which inherit from FixedValue and zeroGradient differently. However, in order to understand the exact definitions of boundary conditions, it is efficient to check the function `updateCoeffs()` and `evaluate()` in specific wall functions classes since in OpenFOAM a lot of boundary conditions are defined in one of them.

Another essential characteristics of wall functions is ypluslam. As mentioned before in OpenFOAM buffer layer is divided into viscous sublayer and logarithmic region in order to provide a effective definition of boundary condition when the first cell center is located in the buffer layer. Ypluslam can be understood as the intersection between the viscous and logarithmic layer. In some wall functions which can provide two calculation modes, yplusLam is usually calculated at the beginning of definition class.

Last but not least, the calculation process (or calculation position) should be demonstrated. In OpenFOAM, based on the calculation process wall functions can be divided into 2 types. One is calculating the value on the face of the first cell, such as kwallfunctions etc. Another type is calculating the value of center of the cell adjacent to the wall, such as epsilon and omega wall functions. The reason we need to distinguish calculation process is that there are some differences in the final result, which is explained latter.

The reader should keep the above three important points about wall functions in mind, the following description will be made based on them.

Wall functions directory in OpenFOAM is located in the path

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

In `wallFunctions` directory there are six types of wall function sub-directories

- epsilonWallFunctions
- kqRWallFunctions
- omegaWallFunctions
- fWallFunctions
- nutWallFunctions

- v2WallFunctions

So in this chapter, each section will introduce one type of wall functions. For some types of wall functions which provide more than 1 kind, there is description in a sub-section under relative section. At the end there is a section where an general conclusion, such as characteristics of each wall function, is made.

2.1 KqRWallFunctions

In OpenFOAM there are two kinds of k wall functions and in `kqRWallFunctions` directory there are two sub-directories.

- kqRWallFunction
- kLowReWallFunction

Each one will be illustrated.

2.1.1 kqRWallFunction

kqRWallFunction inherits from the `zeroGradientFvPatchField`, which means it provides Neumann boundary (the only Neumann boundary of wall functions). As said at the beginning in this chapter, to know the exact meaning, checking the function `evaluate()`

```
120 template<class Type>
121 void Foam::kqRWallFunctionFvPatchField<Type>::evaluate
122 (
123     const Pstream::commsTypes commsType
124 )
125 {
126     zeroGradientFvPatchField<Type>::evaluate(commsType);
127 }
```

It provides a pure zero-gradient boundary condition.

2.1.2 kLowReWallFunction

kLowReWallFunction provides a turbulence kinetic energy boundary condition for low- and high-Reynolds number turbulent flow cases. And it can supply boundary condition if the first cell center is located in buffer layer, so ypluslam is calculated at the beginning.

```
34 Foam::scalar Foam::epsilonLowReWallFunctionFvPatchScalarField::yPlusLam
35 (
36     const scalar kappa,
37     const scalar E
38 )
39 {
40     scalar ypl = 11.0;
41
42     for (int i=0; i<10; i++)
43     {
44         ypl = log(max(E*ypl, 1))/kappa;
45     }
46
47     return ypl;
48 }
```

The parameter yPlusLam is calculated through ten iteration steps and the result is approximately 11.5. kappa is von karman constant with value of 0.41, E is the roughness coefficient with value of 9. In most wall functions which can operate two modes, the ypluslam is calculated at the beginning using the same member function as above. The equation is given by

$$ypl = \frac{log(max(E \times ypl^{+}, 1))}{\kappa} \tag{2.1}$$

Then checking function to know exact definition. `updateCoeffs()`.

```
164 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
```

There is a if statement in the beginning. If `update()` then return without doing anything. To know the reason we should check the definition of `update()`.
Declaration class `kLowReWallFunctionFvPatchScalarField.H` inherits from the class `fixedValueFvPatchField.H`. But the function update() cannot be found in this function. So checking higher level class. `fixedValueFvPatchField.H` inherits from `fvPatchField.H`. In `fvPatchField.H` function `update()` is found and the code is given by

```
//- Return true if the boundary condition has already been updated
bool updated() const
{
       return updated_;
}
```

Checking the updateCoeffs()

```
void Foam::fvPatchField<Type>::updateCoeffs()
{
    updated_ = true;
}
```

Based on the definitions of functions of `update()` and `updateCoeffs()` in the class `fvPatchField.H`, we can understand the purpose of if statement in the beginning of `updateCoeffs()` in kLowReWallFunction. If the update() has been used, if it has been updated return the `updated_`. In the `updateCoeffs()` if statement condition is satisfied so `updateCoeffs()` do nothing. The above steps are done in `updateCoeffs()` function of most wall function definition classes in order to judge if the boundary condition has been updated.

Let us get back to the updateCoeffs() functions in kLowReWallFunction. Firstly boundary is divided into different patches, and label every patch with patchi. Then some parameters are defined. Line 181 defined y. y() is the member function of `turbulenceModel.H`. To check y(), k() and nu() go to `turbulenceModel.H` to check

```
157   //- Return the near wall distances
158       const nearWallDist& y() const
159       {
160           return y_;
161       }
```

The distance of the first cell center to the wall y is defined here. For k() and nu() the definition processes are the same as y().

Going back to `updateCoeffs()` (in page 10), `nuw` at line 187 means the friction velocity and `cmu25` is a scalar which uses constant `Cmu_` to calculate. From line 193 the calculation based on the two modes starts.

First going all faces of one patch. The important parameter yplus(y^+) is calculated through uTau(u_τ). uTau is friction velocity, which is calculated through the equation given by

$$u_\tau = C_\mu^{1/4}\sqrt{k} \tag{2.2}$$

Where k(`k(faceCelli)`) is the value of kinematic energy of the cell center adjacent to the wall.

After getting y^+ value, it comes to the calculation of k^+.
When $y^+ >$ yPlusLam

$$k^+ = \frac{C_k}{\kappa} \times log(y^+) + B_k \tag{2.3}$$

The constant value C_k and B_k are the same as the theory described in chapter one.
When $y^+ <$ yPlusLam

$$k^+ = \frac{2400}{C_{eps2}^2} \times C_f. \tag{2.4}$$

where C_f is calculated through the equation

$$C_f = [\frac{1}{(y^+ + C)^2} + \frac{2y^+}{C^3} - \frac{1}{C^2}] \tag{2.5}$$

The final step is to change the non-dimensional k^+ into k value on the wall(face of cell).

$$k = k^+ \times u_\tau^2 \tag{2.6}$$

Then doing the above steps for all patches.

2.2 V2WallFunction

V2Wallfunction provide the stress normal to streamlines boundary condition. The code structure is similar to kLowReWallFunction. So just checking the calculation part. v^2 is calculated in the non-dimensional form.

```
197     forAll(V^2, facei)
198     {
205         if (yPlus > yPlusLam_)
206         {
207             scalar CV^2 = 0.193;
208             scalar BV^2 = -0.94;
209             V^2[facei] = CV^2/kappa_*log(yPlus) + BV^2;
210         }
211         else
212         {
213             scalar CV^2 = 0.193;
214             V^2[facei] = CV^2*pow4(yPlus);
215         }
216
217         V^2[facei] *= sqr(uTau);
218     }
```

$(v^2)^+$ is calculated depending on the position of y^+.

$$(v^2)^+ = [\frac{C_{V^2}}{\kappa}\ln(y^+) + B_{V^2}] \tag{2.7}$$

When $y^+ <$ yPlusLam

$$(v^2)^+ = C_{V^2}(y^+)^2 \tag{2.8}$$

Here the constant value of C_{V^2} and B_{V^2} have the same value as theory description before. Finally the value of (v^2) on the wall is obtained, as

$$(v^2) = (v^2)^+ \times u_\tau^2 \tag{2.9}$$

The equations in OpenFOAM is the same as that in the theroy description.

2.3 fWallFunctions

fWllfunctions provides the damping function both for low and high Reynolds numbers. The code structure is similar as v^2 and provides two modes.

When $y^+ >$ yPlusLam

$$f = \frac{Nv^2\varepsilon}{k^2u_\tau^2} \tag{2.10}$$

When $y^+ <$ yPlusLam

$$f = 0 \tag{2.11}$$

Where N is constant with value of 6.
However formula is totally different from theory, which means when using f wall function as boundary condition, checking your case requirement in advance.

2.4 epsilonWallFunctions

Unlike k, v^2 or f wall function, in the epsilon wall function the value of the cell center, rather than the value on the cell face, is calculated.

In `epsilonWallFunctions` directory there are two sub-directories:

- `epsilonWallFunction`
- `epsilonLowReWallFunction`

Before explaining the code the parameter 'weights' should be introduced first. Weights means that how many faces of one cell use this boundary condition. If three faces of one cell use the epsilonWallFunctionFvPatchScalarField boundary condition, the weights of this cell is three. The purpose of calculating the weights is obtaining the value of the cell center. In figure 2.1 the weight is 3.

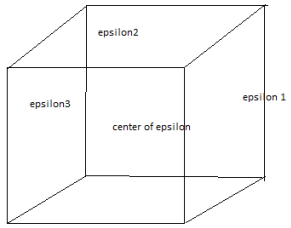

Figure 2.1: The calculation of cell center of epsilon

In figure 2.1 the epsilon value of each face is calculated and then summing them up through equation (2.12) to get the cell center's value.

$$\varepsilon_{center} = \frac{1}{3}\varepsilon_1 + \frac{1}{3}\varepsilon_2 + \frac{1}{3}\varepsilon_3 \tag{2.12}$$

2.4.1 epsilonWallFunction

As before, `updateCoeffs()` is mainly focused. Three functions should be analysed Since they have big influence on the calculation process.

1. setMaster()

   `setMaster()` is used to set the master patch, the master patch which be set here is used in the following calculation steps. In this function `master_` is the ID of the master patch and it is the member data of patch.

2. createAveragingWeights()

   `createAveragingWeights()` is used to create the averaging corner weights when doing the calculations. Corner weight is parameter to be used in the calculation, which is illustrated later. In the function `cornerWeights_` is used to list the averaging corner weights.

3. calculateTurbulenceFields()

   The main process of calculation is done in this function and it call member function **calculate** to do the exact calculation.

Initially the code of `updateCoeffs()` is show as

```
void Foam::epsilonWallFunctionFvPatchScalarField::updateCoeffs()
400 {
401     if (updated())
402     {
403         return;
404     }
405
406     const turbulenceModel& turbModel = db().lookupObject<turbulenceModel>
407     (
408         IOobject::groupName
409         (
410             turbulenceModel::propertiesName,

411             internalField().group()
412         )
413     );
414
415     setMaster();
416
417     if (patch().index() == master_)
418     {
419         createAveragingWeights();
420         calculateTurbulenceFields(turbModel, G(true), epsilon(true));
421     }
422
423     const scalarField& G0 = this->G();
424     const scalarField& epsilon0 = this->epsilon();
425
```

```
426     typedef DimensionedField<scalar, volMesh> FieldType;
427
428     FieldType& G =
429         const_cast<FieldType&>
430 (
431             db().lookupObject<FieldType>(turbModel.GName())
432         );
433
434     FieldType& epsilon = const_cast<FieldType&>(internalField());
435
436     forAll(*this, facei)
437     {
438         label celli = patch().faceCells()[facei];
439
440         G[celli] = G0[celli];
441         epsilon[celli] = epsilon0[celli];
442     }
443
444     fvPatchField<scalar>::updateCoeffs();
445 }
```

The function `setMaster()` is called in line 415. `createAveragingWeights()` is called in line 419 and `calculateTurbulenceFields(turbModel, G(true), epsilon(true))` is called in line 420. In the beginning `if (update())` is used again to judge whether this boundary condition has been updated. Then it comes to the `setMaster()`. First start going through `setMaster()`.

setMaster()

```
65 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 }
```

As said before, the purpose of this function is to set the master patch. `master_`is the ID of master patch. The initial value of `master_` is -1.

Line 67-71 is judging whether this master patch has been settled.

Then in the line 75 variable bf has gotten the boundary field of the epsilon.

In the line 77 a temporary variable master has been set to -1.

From the line 77 to 89. For all the patches and bf. If the patch of boundary using epsilonWallFunctionFvPatchScalarField, then judging the master is whether equal to -1.
when master $= -1$, the value of patchI has been given to master, and then give the master value to the member data `master_` of this patchI.(since the function `master()` return `master_` to the patch) Thus the match which has gotten `master_` value is master patch. After setting master patch the relevant calculation is done around master patch.

An simple example can be given to illustrat. There are two patches: the first patch with its patch index patchI=0 and second patch with patch index patchI=1. Firstly when it comes to the patch with patchI=0, its member data `master_` is of course -1 (haven't been handled) so `setMaster()` will handle it. Because the patchI of first patch is 0 so when the function has been run over to the end the member data `master_` of the first patch is zero. When the second patch with its patch index equal to 1 the `setMaster()` also will handle it. However when going to line 84 `master` is now 0 rather than 1. (it has been changed to 0 when handling the first patch) so the second patch won't give its patch index patchI=1 to its member data `master` since the condition of if statement cannot be satisfied. Its member data `master` will be given value of 0 directly. Thus the patch index patchI and member data `master` of the first patch are both equal to 0 but second patch has its patchI=1 and its member data `master_`=0. The purpose is the function can using the judge sentence to pick the master patch (can be understood as the patchI and `master_` of master patch are equal) to do calculation.

After understanding how `setMaster()` set the master patch going back to `updateCoeffs()` to line 417. There is an if statement. It is for picking master patch since only master patch has the same `patchI`(patch index) and `master_`, in the big parentheses two functions describe the calculation for the master patch which will discussed later.
In the `updateCoeffs()`line 423 and 424 member functions `G()` and `epsilon()` give the value to the data G0 and epsilon0. These two functions are very similar so only opening one to check what value has been return in these functions.

epsilon()

```
364 Foam::scalarField& Foam::epsilonWallFunctionFvPatchScalarField::epsilon(bool init)
365 {
366     if (patch().index() == master_)
367     {
368         if (init)
369         {
370             epsilon_ = 0.0;
371         }
372
373         return epsilon_;
374     }
375
```

```
376     return epsilonPatch(master_).epsilon();
377 }
```

In the line 366 of function `epsilon()` the same if statement appeared in function `updateCoeffs()` is used and it is clear now that it picks the master patch.

In the line 368 if statement is used, since the `init` value is false, so `epsilon()` will return `epsilon_`(local copy of turbulence epsilon field) directly. In the line 366 if statement is not satisfied (not master patch) the function `epsilon()` will return the `epsilon` of master patch. (since the function epsilonPatch(patchi) return access to the epsilon patch and the pathI of master patch is `master_`).

An simple conclusion can be made: an under study boundary has a lot of patches and the patch with the smallest patch index patchI will be set into master patch , the relevant calculation is done together with the master patch. These non-master patches just read the results (turbulence field `G_`and turbulence epsilon field `epsilon_`) from master patch since the master patch getting the whole internal values so there is no need for other patches to do the same calculation again.

In the end of the `upfateCoeffs()` the main process is that give the value of temporary parameter G0 and epsilon0 to the cells adjacent to the wall.

After understanding how the function `updateCoeffs()` works the important part of calculation should be analysed. As said before the function `createAveragingWeights()` and `calculateTurbulenceFields` are mainly related to calculation. First checking the `createAveragingWeights()`.

createAveragingWeights()

```
 95 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 }
```

From the line 109 to 122 the volScalarField `weights` has been initialed as an IOobject and set the initial value of 0.

In the line 124 a dynamicList `epsilonPatches` has been defined and line 125 to 129 illustrates if the patch (with index of patchI) of boundary use the epsilonWallFunctionFvPatchScalarField condition then add this patch in the list of epsilonPatches which will used in line 140 later.

In the line 134 parameter weights is calculated.

From line 139 an important parameter cornerWeight is obtained. For all the patches which use epsilonWallFunctionFvPatchScalarField boundary (in line 140), the cornerWeight (line 143 and 144) of one face is the reciprocal of weights of the cell which this face belongs to. In the previous example cornerweight is 1/3 (the cornerWeight of each face is equal).

In the line 147 and 148 the member data `G_` has been initialized to zero (`interfield()` means the value is inside the cells).

Then checking the function `calculateTurbulenceFields()`.

calculateTurbulenceFields()

```
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 line 177, statement `!cornerWeights_[patchi].empty()` is used to judge whether the patch uses epsilon boundary condition. In line 185 member function `calculate` is called to do the calculation. In line 189 using the zero-gradient condition and put the value of the cell center to the boundary.

The function `calculate` does the exact calculation. So opening it and checking the equations of boundary condition.

calculate(turbulence, w, epf.patch(), G0, epsilon0)

```
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 }
```

It can be found that the cell center value is the sum of value of each face times the cornerweight.

$$\varepsilon = \frac{1}{W}\sum_{f=i}^{W}\left(\frac{c_\mu^{3/4}k^{3/2}}{\kappa y_i}\right) \tag{2.13}$$

It calculates the sum of the value on each face based on the weights.
Where W is the weight of the cell, k is the value of the cell center and y is the distance from the center of cell to the wall.

2.4.2 epsilonLowReWallFunction

The `epsilonLowReWallFunctionFvPatchScalarField` inherits from `epsilonWallFunctionFvPatchScalarField` So process of this wall function is similar as before, except that this wall function provides two calculation models based on the position of y^+, Only part of calculation need to paid attention (checking `calculate` function).

```
 86     // Set epsilon and G
 87     forAll(nutw, facei)
 88     {
 95         if (yPlus > yPlusLam_)
 96         {
 97             epsilon0[celli] += w*Cmu75*pow(k[celli], 1.5)/(kappa_*y[facei]);
 98
 99             G0[celli] +=
100                 w
101                *(nutw[facei] + nuw[facei])
102                *magGradUw[facei]
103                *Cmu25*sqrt(k[celli])
104                /(kappa_*y[facei]);
105         }
106         else
107         {
108             epsilon0[celli] += w*2.0*k[celli]*nuw[facei]/sqr(y[facei]);
109             G0[celli] += G[celli];
110         }
111     }
112 }
```

When $y^+ >$ yPlusLam the equation is same as in the epsilonWallFunction.
When $y^+ <$ yPlusLam

$$\varepsilon = \frac{1}{W}\sum_{f=i}^{W}(\frac{2\cdot k\nu_i}{y_i^2}) \tag{2.14}$$

In the theory non-dimensional epsilon value in the viscous sublayer is given by

$$\varepsilon^+ = 2\frac{k^+}{(y^+)^2} \tag{2.15}$$

This function provide boundary condition for low Reynolds $k-\varepsilon$ and v^2-f turbulence model.

2.5 OmegaWallFunctions

Omega wall functions provide the constraint on turbulence specific dissipation.

As the theory mentioned before, the omega wall function provides the combination of viscous and log equation. In the OpenFOAM omegaWallFunction is a special wall function which can switch between viscous and logarithmic region according to the position of y^+. In the intersection of the viscous sublayer and log-law region value is calculated through blending the viscous and log-law sublayer value.

Value of the cells center is calculated (same as epsilon), which considers two or more than two faces using omegaWallFunctionFvPatchScalarField boundary.

Function `updateCoeffs()` is very similar to the that of epsilonWallFunction, thus only focusing on the function `calculate`.

```
234
235     // Set omega and G
236     forAll(nutw, facei)
237     {
238         label celli = patch.faceCells()[facei];
239
240         scalar w = cornerWeights[facei];
241
242         scalar omegaVis = 6.0*nuw[facei]/(beta1_*sqr(y[facei]));
243
244         scalar omegaLog = sqrt(k[celli])/(Cmu25*kappa_*y[facei]);
245
246         omega[celli] += w*sqrt(sqr(omegaVis) + sqr(omegaLog));
247
248         G[celli] +=
249             w
250            *(nutw[facei] + nuw[facei])
251            *magGradUw[facei]
252            *Cmu25*sqrt(k[celli])
253            /(kappa_*y[facei]);
254     }
255 }
```

Only focusing the main calculation part of function. From the equation it can be seen that the omegaVis and omegaLog is calculated on the face of the cell, Then blending them together and calculating the value of cell center.
In the viscous layer, the equation is given by

$$\omega_{Vis} = \frac{6.0\nu}{\beta_1 y^2} \tag{2.16}$$

Where β_1 is constant with value of 0.075
In the log-law layer the equation is given by

$$\omega_{Log} = \frac{k^{1/2}}{C_\mu^{1/4}\kappa y} \tag{2.17}$$

Where k is the value of cell. Finally getting the combination of equation (2.16) and equation (2.17)

$$\omega = \sqrt{\omega_{Vis}^2 + \omega_{Log}^2} \tag{2.18}$$

So when using the turbulence model based on the omega (such as $k-\omega$), this wall function can be used.

2.6 nutWallFunctions

nutWallFunctions is simple but important which provides eight kinds of turbulence viscosity of the wall boundary condition.

Firstly what is the turbulence viscosity? The shear stress can be represented as ([1])

$$\tau_w = \nu \cdot \frac{\partial U}{\partial n} \mid_w \tag{2.19}$$

In order to solve the equation (2.19), there are two ways. The first way is doing the DNS simulations to get the curve of between velocity profile and y, then getting the derivative to get the correct gradient. The second way is using the equation (2.20) to calculate.

$$\tau_w = \nu \frac{U_c - U_w}{y} \tag{2.20}$$

Where U_c is the velocity of first cell center and U_w is velocity of the flow on the wall.

But in fact since the velocity gradient near the wall is very large so two equations are not equal. But we can make an effective viscosity to make them equal. The second way is much better since we don't need to modify the momentum equations but only setting the new viscosity to correct the shear stress on the wall.
In logarithmic area kinematic energy k is represented as ([1])

$$k^+ = \frac{1}{\sqrt{C_\mu}} = k/u_\tau^2 \tag{2.21}$$

Friction velocity can be obtained as

$$u_\tau = C_\mu^{1/4} k^{1/2} \tag{2.22}$$

And the shear stress on the wall can be also represented as

$$\tau_w = \rho u_\tau^2 \tag{2.23}$$

Friction velocity is equal to u divided by the non-dimensional u^+. So the equation (2.23) can be represented as

$$\tau_w = \rho u_\tau \cdot \frac{U}{U^+} = \frac{\rho u_\tau (U_c - U_w)}{\frac{1}{\kappa}\ln(Ey^+)} \tag{2.24}$$

U is the velocity of the first cell center parallel to the wall.

Compared with the equation (2.44), a new effective viscosity can be redefined

$$\nu_{new} = \frac{u_\tau y}{\frac{1}{\kappa}\ln(Ey^+)} = \frac{y^+ \nu}{\frac{1}{\kappa}\ln(Ey^+)} = \nu + \nu_t \tag{2.25}$$

So the turbulence viscosity can be obtained as

$$\nu_t = \nu \cdot \left(\frac{\kappa y^+}{\ln(Ey^+)} - 1 \right) \tag{2.26}$$

This is a simple example of getting turbulence viscosity. y^+ can be obtained through different ways. Turbulence viscosity can be used to correct the viscosity to get the proper wall shear stress when using turbulence model to do the CFD simulations.

In the nutwallFunctions directory, there are a lot of types of turbulence viscosity wall functions. But not each of them provides one specific type of turbulence viscosity boundary condition. The `nutWallFunction` is the abstract class which don't provide a specific turbulence viscosity. Figure 2.2 can illustrate the inheriting relationship between different classes.

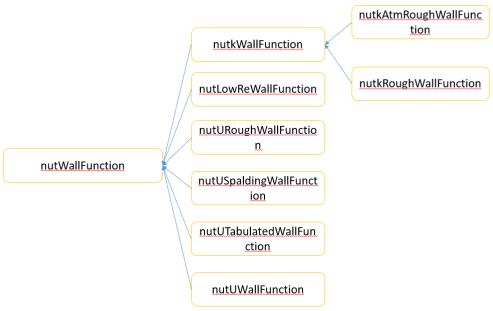

Figure 2.2: Inherit Relationship among turbulence viscosity wall functions

2.6.1 nutWallFunction

The class `nutWallFunction` is an abstract class in the nutWallFunctions directory and it inherits from the fixedValueFvPatchScalarField so it provides Dirichlet boundary condition.

Checking `updateCoeffs()`

```
169 void nutWallFunctionFvPatchScalarField::updateCoeffs()
170 {
171     if (updated())
172     {
173         return;
174     }
175
176     operator==(calcNut());
177
178     fixedValueFvPatchScalarField::updateCoeffs();
179 }
```

An operator is defined and equal to the function `calcNut()` which means the calculation will be done in `calcNut()`, then turbulence viscosity value is aligned to each patch. Function `calcNut()` is defined as a virtual function which has been initialized to zero and for different nutwallfunction, thus this function will be redefined. In order to understand what kind of turbulence viscosity that each nutwallfunction supply it is better to check the function `calcNut()` in each class.

What should also been mentioned that the value of yPlusLam is calculated in the nutWallFunction. In other nut wall functions if some of them provide two models for viscous and logarithm layer they can use the value directly.

2.6.2 nutLowReWallFunction

Checking the function `calcNut()`

```
tmp<scalarField> nutLowReWallFunctionFvPatchScalarField::calcNut() const
{
    return tmp<scalarField>(new scalarField(patch().size(), 0.0));
}
```

It is obvious that nutLowReWallFunction set the turbulence viscosity into zero.

2.6.3 nutkWallFunction

The wall function `nutkWallFunction` provides a turbulence viscosity condition based on the turbulence kinetic energy. First checking the function `CalcNut()`

```
40 tmp<scalarField> nutkWallFunctionFvPatchScalarField::calcNut() const
41 {
42     const label patchi = patch().index();
43
44     const turbulenceModel& turbModel = db().lookupObject<turbulenceModel>
45     (
46         IOobject::groupName
47         (
48             turbulenceModel::propertiesName,
49             internalField().group()
50         )
51     );
52
53     const scalarField& y = turbModel.y()[patchi];
54     const tmp<volScalarField> tk = turbModel.k();
55     const volScalarField& k = tk();
56     const tmp<scalarField> tnuw = turbModel.nu(patchi);
57     const scalarField& nuw = tnuw();
58
59     const scalar Cmu25 = pow025(Cmu_);
60
61     tmp<scalarField> tnutw(new scalarField(patch().size(), 0.0));
62     scalarField& nutw = tnutw.ref();
63
64     forAll(nutw, facei)
65     {
66         label faceCelli = patch().faceCells()[facei];
67
68         scalar yPlus = Cmu25*y[facei]*sqrt(k[faceCelli])/nuw[facei];
69
70         if (yPlus > yPlusLam_)
71         {
72             nutw[facei] = nuw[facei]*(yPlus*kappa_/log(E_*yPlus) - 1.0);
73         }
74     }
75
76     return tnutw;
77 }
```

From line 56 to 57 the viscosity(`nuw`) of each patch is obtained.

In line 61 the turbulence viscosity (`tnutw`) is initialized with the value of 0. In the calculation `nutw` is the turbulence viscosity. This function supply two modes
When $y^+ >$ yPlusLam

$$\nu_t = \nu \cdot \left(\frac{\kappa y^+}{\ln(Ey^+)} - 1 \right) \tag{2.27}$$

When $y^+ <$ yPlusLam

$$\nu_t = 0 \tag{2.28}$$

It is same as it in theory and in the theory part u^+ is obtained from logarithmic region so when using this boundary condition it is better to put the first cell center in the logarithmic area.

The member function `yPlus()` is defined here but not use. However for other functions it can be used to calculate yplus. For example there is a class called `yPlusRAS()` uses this function to calculate the yplus for all wall patches when using RAS turbulence models.
`yPlusRAS()` is located in the FOAM-extend-4.0

```
$HOME/foam/foam-extend-4.0/applications/utilities/postProcessing/wall/yPlusRAS+
```

Opening `yPlusRAS.C`

```
45 #include "nutWallFunction/nutWallFunctionFvPatchScalarField.H"
```

In the beginning the nutWallFunctionFvPatchScalarField.H has been included.
In the line 63 there is typedef command for using wallFunctionPatchField to represent nutWallFunctionFvPatchScalarField.

```
63     typedef incompressible::RASModels::nutWallFunctionFvPatchScalarField
64     wallFunctionPatchField;
```

In the calculation process of yPlus, `nutPW` means nutWallFunctionFvScalarField. In the line 169 function `yPlus()` which is defined in the nutwallfunction has been called to calculated the yPlus at each patch.

```
bool foundNutPatch = false;
 79     forAll(nutPatches, patchi)
 80     {
 81         if (isA<wallFunctionPatchField>(nutPatches[patchi]))
 82         {
 83             foundNutPatch = true;
 84
 85             const wallFunctionPatchField& nutPw =
 86                 dynamic_cast<const wallFunctionPatchField&>
 87                     (nutPatches[patchi]);
 88
 89             yPlus.boundaryField()[patchi] = nutPw.yPlus();
 90             const scalarField& Yp = yPlus.boundaryField()[patchi];
```

Yp is the final result.

2.6.4 nutUWallFunction

The structure of this class is similar to nutkWallFunction. So just checking the calculation part.

```
 62     forAll(yPlus, facei)
 63     {
 64         if (yPlus[facei] > yPlusLam_)
 65         {
 66             nutw[facei] =
 67                 nuw[facei]*(yPlus[facei]*kappa_/log(E_*yPlus[facei]) - 1.0);
 68         }
 69     }
 70
 71     return tnutw;
```

The calculation equation is same as that in nutkWallFunction. So when using nutUWallFunction as boundary condition trying to put the first cell center in the logarithmic area.

Here function `CalcYPlus` should be paid attention. The principle can be summarized as.

In logarithmic area the velocity profile and distance is given by

$$U^+ = \frac{U_c}{u_\tau} = \frac{1}{\kappa}\ln(Ey^+) \tag{2.29}$$

Doing some modifications we can get

$$\frac{U_c}{yu_\tau/\nu}\cdot(y/\nu) = \frac{U_c}{y^+}\cdot(y/\nu) = \frac{1}{\kappa}\ln(Ey^+) \tag{2.30}$$

Then we can get

$$y^+\ln(Ey^+) - \frac{\kappa y U_p}{\nu} = 0 \tag{2.31}$$

We solve it using Newton-Raphson as ([1])

$$y^+_{n+1} = y^+_n - \frac{f(y^+)}{f'(y+)} = y^+_n - \frac{y^+_n\ln(Ey^+_n) - \frac{\kappa y U_c}{\nu}}{1+\ln(Ey^+_n)} = \frac{y^+_n + \frac{\kappa y U_c}{\nu}}{1+\ln(Ey^+_n)} \tag{2.32}$$

This is an iteration process. It can be seen that the calculation part in the code is the same as that in theroy. The code is given by

```
 97     forAll(yPlus, facei)
 98     {
 99         scalar kappaRe = kappa_*magUp[facei]*y[facei]/nuw[facei];
100
101         scalar yp = yPlusLam_;
102         scalar ryPlusLam = 1.0/yp;
103
104         int iter = 0;
105         scalar yPlusLast = 0.0;
106
107         do
108         {
109             yPlusLast = yp;
110             yp = (kappaRe + yp)/(1.0 + log(E_*yp));
111
112         } while (mag(ryPlusLam*(yp - yPlusLast)) > 0.01 && ++iter < 10 );
113
114         yPlus[facei] = max(0.0, yp);
115     }
116
117     return tyPlus;
118 }
```

In the line 99 it calculate the second item of equation (2.31). And from the 107 to 114 Newton-Raphson iteration is done. The equation (2.32) is implemented in the code from line 110.

2.6.5 nutUSpaldingWallFunction

This wall function is based on the special relationship between y^+ and u^+.The curve can fit the curve of $u^+ = y^+$ in the viscous layer and $u^+ = Ey^+/\kappa$ in the log area.

$$y^+ = u^+ + \frac{1}{E}\left[e^{\kappa u^+} - 1 - \kappa u^+ - \frac{1}{2}(\kappa u^+)^2 - \frac{1}{6}(\kappa u^+)^3\right] \tag{2.33}$$

```
42
43     const turbulenceModel& turbModel = db().lookupObject<turbulenceModel>
44     (
45         IOobject::groupName
46         (
47             turbulenceModel::propertiesName,
48             internalField().group()
49         )
50     );
51     const fvPatchVectorField& Uw = turbModel.U().boundaryField()[patchi];
52     const scalarField magGradU(mag(Uw.snGrad()));
53     const tmp<scalarField> tnuw = turbModel.nu(patchi);
54     const scalarField& nuw = tnuw();
55
56     return max
57     (
58         scalar(0),
59         sqr(calcUTau(magGradU))/(magGradU + ROOTVSMALL) - nuw
60     );
61 }
```

Focusing from line 56 to 61. It supplies the maximum value of 0 or equation. The function `mag(Uw.snGrad()` means the gradient of the value. (here it means gradient of velocity).

$$\nu_t = \frac{(u_\tau)^2}{\frac{\partial U}{\partial n}} - \nu \tag{2.34}$$

In this class the yplus is calculated based on the function `calcUTau()` which is also done through Newton-Raphton method. Since this wall function can operate in two modes so if the position of first cell center can not be guarantee in viscous or logarithmic area, just choosing this wall function.

2.7 Conclusion

Some types of wall functions have been described thoroughly and characteristics and purpose of wall functions are made in table 2.1.

Table 2.1: The wall function

kqRWallFunction kLowReWallFunction	For high Re, zero gradient condition for high and low Re, the condition based on the position of y^+
epsilonWallFunction epsilonLowReWallFunction	High Re for High and Low Re, the condition based on the position of y^+
V2WallFunction	For High and Low Re, The condition based on the position of y^+
fWallFunction	For High and Low Re, The condition based on the position of y^+
omegaWallFunction	For High and Low Re, The condition based the blending of viscous and log layer
nutWallFunction nutLowReWallFunction nutUWallFunction nutkWallFunction utUSpaldingWallFunction	abstract class, not provide specific nut for low Re, the condition is zero condition based on the velocity condition based on the kinematic for the whole area

Chapter 3

Structure of wall function and implementation

3.1 Structure of wall functions

In this section how wall functions are used in a solver is illustrated. Based on OpenFOAM lectures we use simpleFoam to explain.

In the simpleFoam.C it can be found that.

```
turbulence->correct();
```

We realize that the object named turbulence is a pointer, since the functions of the object are called using `->`. The operation of those member functions depend on which turbulence model we are using, which is specified when we run the case. So the object name called turbulence can call any member function in the turbulence model we use ([6]).

So it means it will call the function **correct()** in the relatively turbulence model ,since here we use KEpsilon turbulence model. So we check the correct() in KEpsilon.C

```
251 // Update epsilon and G at the wall
252 epsilon_.boundaryFieldRef().updateCoeffs();
```

Here `epsilon_` is an object of class, and it use its member function, Checking the function boundaryFieldRef(). It is defined in the function GeometricField.C, The location is

```
$WM_PROJECT_DIR/src/OpenFOAM/fields/GeometricFields/GeometricField/GeometricField.C
```

In the line 739 the definition is found.

```
 736 template<class Type, template<class> class PatchField, class GeoMesh>
 737 typename
 738 Foam::GeometricField<Type, PatchField, GeoMesh>::Boundary&
 739 Foam::GeometricField<Type, PatchField, GeoMesh>::boundaryFieldRef()
 740 {
 741     this->setUpToDate();
 742     storeOldTimes();
 743     return boundaryField_;
 744 }
```

It is used to return the reference to the internal field and `boundaryField_` is boundary type field containing boundary field values. It return the `boundaryField_` so it called the boundary field of

epsilon and updateCoeffs() is called. updateCoeffs() is the function which is defined in the epsilon wall function.

In the kEpsilon.C there is another important function `epsEqn.ref().boundaryManipulate(epsilon_.boundaryFieldRef` This function is used to find the adjacent cells to the boundaries whose conditions are "epsilonWall-Function" and eliminate the corresponding equations in these cells from the matrix.

3.2 Implementing a new wall function

when building a new turbulence model such as ζ-f turbulence model, a new kind of wall function ζ should be redefined. In this section an example of how to implement new wall function is showed. Here no modifications of existing wall functions are going to presented.
First initializing the environment and copying the whole TurbulenceModels under the src directory to user directory

```
OF4x
cd $WM_PROJECT_USER_DIR/src
cp -r --parents  $WM_PROJECT_DIR/src/TurbulenceModel .
cd TurbulenceModels/turbulenceModels/derivedFvPatchFields/wallFunctions
```

As said above, the list of wall function can be found here.
Then go back to the directory where Make directory appear:

```
cd ../..
vi Make/files
```

And change to

```
LIB = $(FOAM_USER_LIBBIN)/libturbulenceModels
```

Compiling with wmake.

The next step is copy from an existing wall function and name zetaWallFunctions

```
cd derivedFvPatchFields/wallFunctions/
cp -r fWallFunctions zetaWallFunctions
```

Since etaWallFunctions contain fWallFunction so re-name it. And then rename the defination class and declaration class. And replace all the fWallFunctionFvPatchScalarField to zetaWallFunction-FvPatchScalarField.

```
mv zetaWallFunctions/fWallFunction zetaWallFunctions/zetaWallFunction
cd zetaWallFunctions/zetaWallFunction
mv fWallFunctionFvPatchScalarField.C  zetaWallFunctionFvPatchScalarField.C
mv fWallFunctionFvPatchScalarField.H  zetaWallFunctionFvPatchScalarField.H
sed -i s/fWallFunctionFvPatchScalarField/zetaWallFunctionFvPatchScalarField/g
zetaWallFunctionFvPatchScalarField.*
sed -i s/fWallFunction/zetaWallFunction/g
zetaWallFunctionFvPatchScalarField.*
```

The come back and open the Make files

```
/* Wall function BCs */
wallFunctions = derivedFvPatchFields/wallFunctions
```

```
nutWallFunctions = $(wallFunctions)/nutWallFunctions
$(nutWallFunctions)/nutWallFunction/nutWallFunctionFvPatchScalarField.C
$(nutWallFunctions)/nutkWallFunction/nutkWallFunctionFvPatchScalarField.C
$(nutWallFunctions)/nutkRoughWallFunction/nutkRoughWallFunctionFvPatchScalarField.C
$(nutWallFunctions)/nutkAtmRoughWallFunction/nutkAtmRoughWallFunctionFvPatchScalarField.C
```

So just add the following line in here

```
zetaWallFunctions = $(wallFunctions)/zetaWallFunctions
$(zetaWallFunctions)/zetaWallFunction/zetaWallFunctionFvPatchScalarField.C
```

Compiling it

```
wmake libso
```

As said before, wall function is one kind of boundary condition.

To valid whether it can be chosen, we can write a wrong name in the boundaryField file and seeing how it happens. So just copying a pitzDaily of simpleFoam tutorial and then entering 0 direcory. Finding a lot of turbulence properties, just opening a random file such as epsilon.

Changing from the epsilonWallFunction to a random name and running the simulation.

The system reports that an Unknown patchField type, together with a list of patchField you can use. zetaWallFunction is one of them, which indicates zetaWallFunction can be used to set the pitch Field. If you have your own case and idea how to make a new wall function, you can follow the above steps.

Chapter 4

Future work

In the present report only the description of existing wall functions in OpenFOAM is made and the first step of implementing a new wall function is showed. The next step is introducing a new wall function.

In FOAM-extent there are a lot of different kinds of wall functions provided. Some of them are similar to the wall functions which has been described in chapter 2, while others are not.

The description of f wall function in the OpenFOAM is different from that in the theory part. The reason could be that the equations have been modified to make it more stable when using it. Or maybe it use another theory when making codes. So it could be better to figure out it before using f wall functions.

The y^+ calculation in some nutWallFunction is very complex so it is better to know how are they be defined.

Chapter 5

Study question

1. Q: Why do we want to use wall function.

2. Q: In terms of calculation process, how many kinds of wall funcion exist in OpenFOAM.

3. Q: In epsilonWallFunction, which function should be modified if you want to introduce a new wall function.

4. Q :Which function provide definition of turbulence viscosity in nutWallFunctions.

5. Q : If you want to use yplus function, which class should be used.

Bibliography

[1] H K Versteeg and W Malalasekera. *An Introduction to Computational Fluid Dynamics*. Bell Bain, 2003.

[2] Lars Davidson. *An Introduction to Turbulence Models*. Chalmers university of technology, 2003.

[3] D. B. SPALDING. A single formula for the law of the wall. *Applied Mechanics*, 28(3):455, 1961.

[4] Georgi Kalitzin ; Gorazd Medic; Gianluca Iaccarino; Paul Durbin. Near-wall behavior of rans turbulence models and implications for wall functions. *Computational physics*, 204(205):265–291, 2004.

[5] B.A. Pettersson Reif P.A. Durbin. *Statistical Theory and Modeling for Turbulent Flow*. Wiley, 2011.

[6] Håkan Nilsson. Implement turbulence model, https://pingpong.chalmers.se/courseid/7056/content.do?id=3256524.