Chapter 18
Boundary Conditions in OpenFOAM® and uFVM

Abstract This chapter reviews the implementation of boundary conditions in OpenFOAM® and uFVM. Details on the data structure needed for their implementation are presented along with information on how to add new boundary conditions. The procedure is illustrated through the implementation of the no-slip wall boundary condition. This is shown to differ from the Dirichlet type currently implemented in OpenFOAM®.

18.1 Boundary Conditions in OpenFOAM®

Each boundary condition has a physical meaning described mathematically via an equation, which in the context of a numerical method has to be translated into an algebraic relation. An *inlet* boundary condition for instance, describes a known flow behavior where velocity and pressure satisfy specified physical conditions expressed using proper mathematical equations. These include a Dirichlet and a Neumann condition, which should be defined in order to connect the mathematical model with the boundary conditions of the problem. The implementation of these conditions will affect the mathematical operator or term to which they apply (i.e., divergence, laplacian, gradient, etc.).

In OpenFOAM® [1] almost all definitions of boundary conditions are stored in the following directory (Listing 18.1):

```
src/finiteVolume/fields/fvPatchFields
```

Listing 18.1 OpenFOAM® directory where boundary condition definitions are stored

with the main implemented *types* of boundary conditions stored in the sub-directories listed in Listing 18.2.

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*,
Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_18

```
basic
constraint
derived
fvPatchField
```

Listing 18.2 Sub-directories where the main types of boundary conditions are implemented

A brief description of these sub-directories is given in what follows.

The fvPatchField directory contains the general class definition of a boundary condition, which represents the base class. This class defines the main functions and data structures that will be used and will be inherited by the genuine classes.

The *basic* directory contains the basic mathematically defined boundary conditions. These are the Dirichlet type (*fixedValue*), the Neuman type (*zeroGradient* and *fixedGradient*), and the Robin type (*mixed*) boundary conditions. One additional entry included in *basic* is the *coupled* boundary condition that implements a patch to patch type condition, i.e., coupling two boundary patches together (coupled boundaries).

The *constraint* directory contains geometric type boundary conditions that derive from the *coupled* boundary class. An example is the periodicity boundary condition depicted schematically in Fig. 18.1. In this case each cell is related to the cell of the corresponding patch allowing the boundary cells to be treated as internal ones.

Finally the *derived* directory includes all boundary conditions that are derived from the basic Dirichlet, Neumann, and Robin boundary conditions. These derived boundary conditions are simply specializations of the basic types.

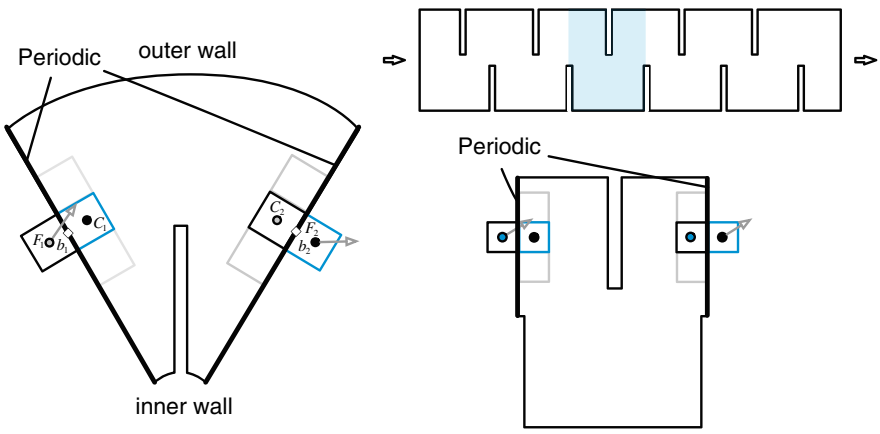

Fig. 18.1 A schematic of a periodic boundary condition

18.2 Boundary Condition Customization

To write a new boundary condition it is essential to understand the role of five main functions, namely, *updateCoeffs, valueInternalCoeffs, valueBoundaryCoeffs, gradientInternalCoeffs,* and *gradientBoundaryCoeffs.*

updateCoeffs: This member function is responsible for the explicit update of the values at the boundary face centers. The function is called whenever the patch field values need to be updated iteratively. For example in the *totalTemperature FvPatchScalarField* class, *updateCoeffs* is used to compute, as shown in Listing 18.3, the static temperature from the specified Total Temperature value according to the relation $T = T_0 - 0.5(\gamma - 1)U^2/(\gamma R)$.

```
void Foam::totalTemperatureFvPatchScalarField::updateCoeffs()
{
    if (updated())
    {
        return;
    }
    const fvPatchVectorField& Up =
        patch().lookupPatchField<volVectorField, vector>(UName_);

    const fvPatchField<scalar>& psip =
        patch().lookupPatchField<volScalarField, scalar>(psiName_);

    scalar gM1ByG = (gamma_ - 1.0)/gamma_;
    operator==

    (
        T0_/(1.0 + 0.5*psip*gM1ByG*magSqr(Up))
    );

    fixedValueFvPatchScalarField::updateCoeffs();
}
```

Listing 18.3 Script used to iteratively update the static temperature from total temperature using the *updateCoeffs* function

In Listing 18.3 the *operator* == assigns the computed value to the static temperature.

Another example is setting a mean value to an entire patch. The idea is to impose an average value at all faces of a boundary patch based on the average of the values at the centroids of their associated boundary cells and a specified mean value. The required *updateCoeffs* function is written as shown in Listing 18.4.

```
template<class Type>
void fixedMeanFvPatchField<Type>::updateCoeffs()
{
    if (this->updated())
    {
        return;
    }

    Field<Type> newValues(this->patchInternalField());
    Type meanValuePsi = gSum(this->patch().magSf()*newValues)
       /gSum(this->patch().magSf());
    newValues += (meanValue_ - meanValuePsi);
    this->operator==(newValues);
    fixedValueFvPatchField<Type>::updateCoeffs();
}
```

Listing 18.4 Another example of using the function *updateCoeffs* to update boundary values

Note that the function in this case is a template. Therefore it can be used in conjunction with a variety of types such as scalars, vectors, or tensors.

With *updateCoeffs()* the values at the face centroids are set explicitly as for a Dirichlet condition. However the other functions, *valueInternalCoeffs*, *valueBoundaryCoeffs*, *gradientInternalCoeffs*, and *gradientBoundaryCoeffs* are used to linearize the boundary condition in order to complement the *updateCoeffs()* function. The *valueInternalCoeffs* and *valueBoundaryCoeffs* are generally used to linearize the boundary condition for a divergence operator since for that operator the value at the patch faces is needed. On the other hand, the *gradientInternalCoeffs* and *gradientBoundaryCoeffs* are used to linearize the boundary condition for a laplace type operator since for that operator the gradient at the patch faces is needed. This is summarized in Table 18.1.

To clarify the above, the implementations of two boundary conditions are considered. The first is the Neuman (zeroFlux) boundary condition, while the second is the Dirichlet (specifiedValue) boundary condition.

For a Neumann (zeroFlux) boundary condition and assuming an orthogonal mesh, the value of the boundary patch is equal to the value of the boundary element since the normal gradient should be equal to zero. For the divergence term where the boundary patch value will be needed, the specified value at the boundary patch is written in term of the *valueInternalCoeffs* and *valueBoundaryCoeffs*, which represent the linearization of the boundary element value and its non-linearizable

Table 18.1 A summary of the coefficients used to linearize boundary conditions

	Diagonal Coeff	Source term
Divergence	valueInternalCoeffs	valueBoundaryCoeffs
Laplacian	gradientInternalCoeffs	gradientBoundaryCoeffs

part, respectively. For example, the boundary value of a zero gradient boundary condition can be written as

$$\begin{aligned}\phi_b &= \mathit{FluxCb}\,\phi_C + \mathit{FluxVb} \\ &= \mathit{valueInternalCoeffs}\,\phi_C + \mathit{valueBoundaryCoeffs} \\ &= 1\,\phi_C + 0\end{aligned} \tag{18.1}$$

The syntax used to implement the zeroFlux condition for the divergence operator is shown in Listing 18.5.

```
template<class Type>
tmp<Field<Type> > zeroGradientFvPatchField<Type>::valueInternalCoeffs
(
    const tmp<scalarField>&
) const
{
    return tmp<Field<Type> >
    (
        new Field<Type>(this->size(), pTraits<Type>::one)
    );
}
template<class Type>
tmp<Field<Type> > zeroGradientFvPatchField<Type>::valueBoundaryCoeffs
(
    const tmp<scalarField>&
) const
{
    return tmp<Field<Type> >
    (
        new Field<Type>(this->size(), pTraits<Type>::zero)
    );
}
```

Listing 18.5 Implementation of the zeroFlux boundary condition for the divergence operator

For the laplacian operator, the gradient at the boundary is needed. In this case the gradient at the boundary patch is set to zero (zeroFlux). This is done through the use of the gradient linearization, which is written as

$$\begin{aligned}\nabla\phi_b &= \mathit{gradientInternalCoeffs}\,\phi_C + \mathit{gradientBoundaryCoeffs} \\ &= 0\,\phi_C + 0\end{aligned} \tag{18.2}$$

where now the *gradientInternalCoeffs* is the linearized coefficient and *gradientBoundaryCoeffs* is the non-linearized component of the gradient as shown in Listing 18.6.

```
template<class Type>
tmp<Field<Type> >
zeroGradientFvPatchField<Type>::gradientInternalCoeffs() const
{
    return tmp<Field<Type> >
    (
        new Field<Type>(this->size(), pTraits<Type>::zero)
    );
}


template<class Type>
tmp<Field<Type> >
zeroGradientFvPatchField<Type>::gradientBoundaryCoeffs() const
{
    return tmp<Field<Type> >
    (
        new Field<Type>(this->size(), pTraits<Type>::zero)
    );
}
```

Listing 18.6 Implementation of the zeroFlux boundary condition for the laplacian operator

Table 18.2 A summary of the coefficients for a zeroFlux boundary condition

Neumann (zero order)	Diagonal Coeff	Source term
Divergence	Value(1)	0
Laplacian	0	0

A summary of the value of the coefficients used to implement a zeroFlux boundary condition is given in Table 18.2.

In the above, the value at the boundary is set equal to the value of the boundary element since a zero flux condition is specified.

For a Dirichlet boundary condition, the contribution to the matrix of coefficients will be just a source term on the right hand side of the equations. In this case the boundary condition does not alter the diagonal. The *valueInternalCoeffs* and *valueBoundaryCoeffs* are defined inside OpenFOAM® as

$$\begin{aligned} \phi_b &= FluxCb\,\phi_C + FluxVb \\ &= valueInternalCoeffs\,\phi_C + valueBoundaryCoeffs \\ &= 0\,\phi_C + \phi_{specified} \end{aligned} \tag{18.3}$$

For the laplacian operator, the gradient at the boundary is based on the Dirichlet value. In this case the gradient at the boundary patch is set again through the use of the gradient linearization, which is written as

Table 18.3 A summary of the coefficients for a Dirichlet boundary condition

Dirichlet	Diagonal Coeff	Source term
Divergence	0	Boundary value
Laplacian	Delta	Boundary value and delta

$$\nabla\phi_b = gradientInternalCoeffs\,\phi_C + gradientBoundaryCoeffs$$
$$= \frac{-\phi_C + \phi_b}{d} = (-\phi_C + \phi_b)delta = -\phi_C\,delta + \phi_b delta$$

A summary of the coefficient values is given in Table 18.3, while the template code is shown in Listing 18.7.

```
template<class Type>
tmp<Field<Type> > fixedValueFvPatchField<Type>::valueInternalCoeffs
(
    const tmp<scalarField>&
) const
{
    return tmp<Field<Type> >
    (
        new Field<Type>(this->size(), pTraits<Type>::zero)
    );
}
template<class Type>
tmp<Field<Type> > fixedValueFvPatchField<Type>::valueBoundaryCoeffs
(
    const tmp<scalarField>&
) const
{
    return *this;
}
…
template<class Type>
tmp<Field<Type> >
fixedValueFvPatchField<Type>::gradientInternalCoeffs() const
{
    return -pTraits<Type>::one*this->patch().deltaCoeffs();
}

template<class Type>
tmp<Field<Type> >
fixedValueFvPatchField<Type>::gradientBoundaryCoeffs() const
{
    return this->patch().deltaCoeffs()*(*this);
}
```

Listing 18.7 Syntax used for the implementation of the Dirichlet boundary condition

18.3 Development of a New BC: No Slip Wall Condition

The task is to define a new boundary condition type for the proper implementation of the no slip condition in the context of the finite volume discretization in OpenFOAM® (Fig. 18.2).

The no slip condition is a fundamental boundary condition in solving flow problems. According to Newton's law, the shear stress experienced by a viscous fluid flowing past a wall is proportional to the normal gradient of the velocity parallel to the wall and is expressed mathematically as

$$\tau_{wall} = -\mu \frac{\partial \mathbf{v}_{||}}{\partial (d_{\perp})_C} \simeq -\frac{\mu}{(d_{\perp})_C} \begin{bmatrix} (1-n_x^2) & -n_y n_x \\ -n_y n_x & (1-n_y^2) \end{bmatrix} \begin{bmatrix} u_C \\ v_C \end{bmatrix} \tag{18.4}$$

where C refers to values at the centroid of the boundary cell. This equation suggests that the no slip condition is anisotropic because it is function only of the velocity component parallel to the wall and the normal distance to the wall. For the case when the wall is aligned with the x-axis, only the x-component of velocity is expected to affect the shear stress value. In fact from Eq. (18.4) the shear stress equation becomes

$$\tau_{wall} = -\mu \frac{\partial \mathbf{v}_{||}}{\partial (d_{\perp})_C} \simeq -\frac{\mu}{(d_{\perp})_C} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} u_C \\ v_C \end{bmatrix} \tag{18.5}$$

where it is evident that only the diagonal coefficient of the x-component of velocity has to be injected into the matrix.

A common simplification for this boundary condition, used in OpenFOAM®, is its treatment as a Dirichlet boundary condition, i.e., a *fixedValue* (*0 0 0*) boundary condition.

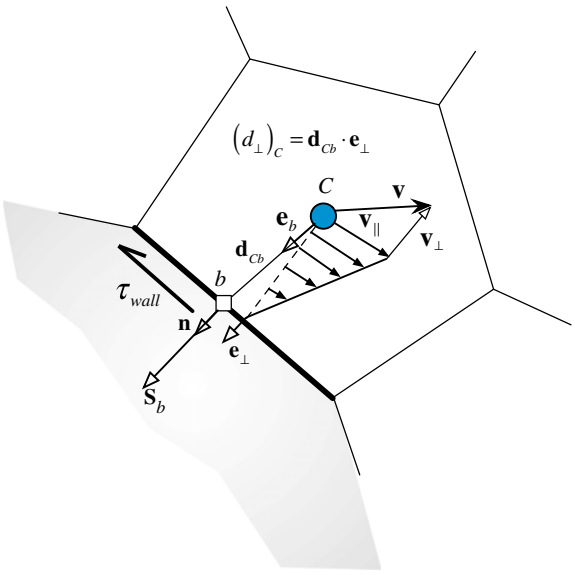

Fig. 18.2 No-slip wall shear stress

Using a Dirichlet condition for the implementation of the no-slip condition at a wall introduces an important error. This error is due to treating the shear stress distribution as isotropic with its value influenced by the velocity component (of the internal cells) normal to the wall. In fact using a Dirichlet boundary condition for the case when the wall is parallel to the x-velocity component, results in a shear stress evaluated as

$$\tau_{wall} = -\mu \frac{\partial \mathbf{v}_{||}}{\partial (d_{\perp})_C} \simeq -\mu \frac{\partial \mathbf{v}}{\partial (d_{\perp})_C} \simeq -\frac{\mu}{(d_{\perp})_C} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_C \\ v_C \end{bmatrix} \tag{18.6}$$

which, when compared with Eq. (18.2), clearly demonstrates the unphysical dependence of the wall shear stress on the y-component of velocity.

In the following, the proper implementation of the no slip wall condition in OpenFOAM® is described. The suggested implementation should always be used instead of the simplified assumption of a Dirichlet boundary condition for momentum discretization at a no-slip wall.

As stated above, the definition of a new boundary condition necessitates redefining the proper functions. A good starting point is to use an existing boundary class, copying it, and modifying it with the new algorithms. For this purpose, the *fixedValue class* can be used and altered for vector type. The .H file defining the new virtual no slip wall class is depicted in Listing 18.8.

```
namespace Foam
{
/*---------------------------------------------------------------------------*
\
                Class noSlipWallFvPatchVectorField Declaration
\*---------------------------------------------------------------------------
*/
class noSlipWallFvPatchVectorField
:
    public fixedValueFvPatchField<vector>
{
protected:

    // Protected data
public:
    //- Runtime type information
    TypeName("noSlipWall");
.
.
.
//- Return the matrix diagonal coefficients corresponding to the
//  evaluation of the gradient of this patchField
virtual tmp<Field<vector> > gradientInternalCoeffs() const;

//- Return the matrix source coefficients corresponding to the
//  evaluation of the gradient of this patchField
virtual tmp<Field<vector> > gradientBoundaryCoeffs() const;
.
.
```

Listing 18.8 Synopsis of the .H file for the declaration of the noSlipWallFvPatchVectorField class

As noticed, just the *gradientCoeffs* function is redefined, since in the *fixedValue* class the *valueCoeffs* and the *updateCoeffs* functions are already correctly implemented.

Once the declarations of the functions are ready, implementation proceeds with the modifications to the .C file. The starting point is Eq. (18.1) but now written in three dimensions as

$$\begin{aligned}\tau_{wall} &= -\mu \frac{\partial \mathbf{v}_{||}}{\partial (d_{\perp})_C} \\ &\simeq \frac{\mu}{(d_{\perp})_C}\mathbf{v}_{||wall} - \frac{\mu}{(d_{\perp})_C}\begin{bmatrix} (1-n_x^2) & -n_y n_x & -n_z n_x \\ -n_y n_x & (1-n_y^2) & -n_z n_y \\ -n_z n_x & -n_z n_y & (1-n_z^2) \end{bmatrix}\begin{bmatrix} u_C \\ v_C \\ w_C \end{bmatrix}\end{aligned} \tag{18.7}$$

Equation (18.7) indicates that all elements in the matrix have implicit contributions because they depend on the internal cell value and have to be implemented with the *gradientInternalCoeffs*. The other term is the tangential component of the wall velocity and it depends on the face value only. In this case it is necessary to be described through the function *gradientBoundaryCoeffs*. Moreover, a closer look at the implicit contribution suggests storing its mixed terms on the right hand side of the equation to be treated explicitly since the momentum equations are solved in a segregated fashion. Thus Eq. (18.7) is rewritten as

$$\begin{aligned}\tau_{wall} \simeq &-\frac{\mu}{(d_{\perp})_C}\underbrace{\begin{bmatrix} (1-n_x^2) & 0 & 0 \\ 0 & (1-n_y^2) & 0 \\ 0 & 0 & (1-n_z^2) \end{bmatrix}}_{gradientInternalCoeffs}\begin{bmatrix} u_C \\ v_C \\ w_C \end{bmatrix} \\ &-\frac{\mu}{(d_{\perp})_C}\underbrace{\begin{bmatrix} 0 & -n_y n_x & -n_z n_x \\ -n_y n_x & 0 & -n_z n_y \\ -n_z n_x & -n_z n_y & 0 \end{bmatrix}}_{gradientBoundaryCoeffs}\begin{bmatrix} u_C \\ v_C \\ w_C \end{bmatrix} + \underbrace{\frac{\mu}{(d_{\perp})_C}\mathbf{v}_{||wall}}_{gradientBoundaryCoeffs}\end{aligned} \tag{18.8}$$

Based on the above formulation, the *gradienInternalCoeffs* function reads as shown in Listing 18.9.

```
// * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * *
* * * //
tmp<Field<vector> >
noSlipWallFvPatchVectorField::gradientInternalCoeffs() const
{
    vectorField normal = this->patch().nf();
    vectorField impCoeff(this->size(),pTraits<vector>::zero);
    forAll(impCoeff,faceI)
    {
       impCoeff[faceI][0] = (scalar(1.0) - pow(normal[faceI][0],2));
       impCoeff[faceI][1] = (scalar(1.0) - pow(normal[faceI][1],2));
       impCoeff[faceI][2] = (scalar(1.0) - pow(normal[faceI][2],2));
    }

    return -impCoeff*this->patch().deltaCoeffs();
}
```

Listing 18.9 Script used for calculating the implicit contribution

On the other hand, the calculation of the *gradientBoundaryCoeffs* is as shown in Listing 18.10.

```
tmp<Field<vector> >
noSlipWallFvPatchVectorField::gradientBoundaryCoeffs() const
{
    vectorField normal = this->patch().nf();
    vectorField expCoeff(this->size(),pTraits<vector>::zero);
    vectorField boundTanField = *this - ((*this) & normal) * normal;
    vectorField intField = this->patchInternalField();
    forAll(expCoeff,faceI)
    {
        expCoeff[faceI][0] = boundTanField[faceI][0]
                            +normal[faceI][0]*normal[faceI]
                             [1]*intField[faceI][1]
                            +normal[faceI][2]*normal[faceI]
                             [0]*intField[faceI][2];
        expCoeff[faceI][1] = boundTanField[faceI][1]
                            +normal[faceI][1]*normal[faceI]
                             [0]*intField[faceI][0]
                            +normal[faceI][2]*normal[faceI]
                             [1]*intField[faceI][2];
        expCoeff[faceI][2] = boundTanField[faceI][2]
                            +normal[faceI][2]*normal[faceI]
                             [0]*intField[faceI][0]
                            +normal[faceI][1]*normal[faceI]
                             [2]*intField[faceI][1];
    }

    return this->patch().deltaCoeffs()*expCoeff;
}
```

Listing 18.10 Script used for calculating the explicit contribution

The new boundary condition can now be used with any wall by just defining the patch type as *noSlipWall*, as described in Listing 18.11.

```
wall
{
    type            noSlipWall;
    value           uniform (0 0 0);
}
```

Listing 18.11 Script needed to use the no-slip boundary condition at runtime

18.4 The No-Slip Boundary Condition in uFVM

Adding a new boundary condition in uFVM requires that the boundary be implemented for each term to which it can be applied, thus is not as modular as in OpenFOAM®. Still it can be quite straightforward as demonstrated for the no-slip boundary condition.

It is important to emphasize that the no-slip boundary condition is somewhat a hybrid condition where a flux (the shear stress) has to be computed while ensuring that the boundary velocities are set to specified values, in this case zero. So it relates somewhat to the Dirichlet and Neumann conditions.

In uFVM information about boundary conditions are stored in a patch-based structure composed of the following four arrays:

1. thePatchFlux.FLUXC1f : the linearization coefficient for the owner cell
2. thePatchFlux.FLUXC2f : the linearization coefficient for the neighbor cell
3. thePatchFlux.FLUXVf: the non-linearized part
4. thePatchFlux.FLUXTf: the total flux at the face

in such a manner that the boundary flux is expressed as

$$\text{thePathFlux.FLUXTf} = \text{thePathFlux.FLUXC1f phi\_owner} + \text{thePathFlux.FLUXC2 phi\_boundary} + \text{thePathFlux.FLUXVf} \tag{18.9}$$

Equation (18.9) shows how the total flux at any internal face is linearized in terms of the owner and neighbor elements sharing the face, except that for a patch the neighbor node is basically the boundary node. For a boundary face, where there is no neighbor element defined, thePatchFlux.FLUXC2f is always set to zero.

For the no slip condition where the total flux (i.e., the shear stress) depends on the change of the velocity component parallel to the wall, the expressions of the various contributions become

$$\begin{aligned}
\text{theFluxes.FLUXC1f(iBFaces)} &= \text{area.} * \text{TM.} * (1 - \text{dot}(\text{nc}', \text{nc}')');\\
\text{theFluxes.FLUXC2f(iBFaces)} &= 0;\\
\text{theFluxes.FLUXVf(iBFaces)} &= \text{Fc} - \text{area.} * \text{TM.} * (1 - \text{dot}(\text{nc}', \text{nc}')'). * \text{velc(iOwners)};\\
\text{theFluxes.FLUXTf(iBFaces)} &= \text{theFluxes.FLUXC1f(iBFaces)}. * \text{velc(iOwners)}\\
&\quad + \text{theFluxes.FLUXC2f(iBFaces)}. * \text{velc(iBElements)}\\
&\quad + \text{theFluxes.FLUXVf(iBFaces)};
\end{aligned} \tag{18.10}$$

where F_c is the actual shear flux computed at the patch wall faces as

$$F_c = -\mu \frac{\mathbf{v} \cdot \mathbf{e}_{\parallel}}{d_{\perp}} \tag{18.11}$$

The complete code for the implementation of the boundary condition is shown in Listing 18.12.

```
function theFluxes =
cfdAssembleStressTermWallNoSlipBC(iPatch,theFluxes,theEquationName,the
Term,iComponent)

theMesh = cfdGetMesh;
theFluidTag = cfdGetFluidTag(theEquationName);

theBoundary = theMesh.boundaries(iPatch);
numberOfElements = theMesh.numberOfElements;
numberOfInteriorFaces = theMesh.numberOfInteriorFaces;
numberOfBFaces = theBoundary.numberOfBFaces;

%
iFaceStart = theBoundary.startFace;
iFaceEnd = iFaceStart+numberOfBFaces-1;
iBFaces = iFaceStart:iFaceEnd;
%
iElementStart = numberOfElements+iFaceStart-numberOfInteriorFaces;
iElementEnd = iElementStart+numberOfBFaces-1;
iBElements = iElementStart:iElementEnd;

%
%
% specifiy the Term Field
%
if(isempty(theTerm.variableName))
    theTermFieldName = theEquationName;
else
    theTermFieldName = theTerm.variableName;
end
theTermField = cfdGetMeshField(theTermFieldName);
velc = theTermField.phi(:,iComponent);

%phiGradient = theTermField.phiGradient(:,:,iComponent);
```

Listing 18.12 Script used for the implementation of the no-slip boundary condition in uFVM

```
if(iComponent==1)
    e = [1;0;0];
elseif(iComponent==2)
    e = [0;1;0];
elseif(iComponent==3)
    e = [0;0;1];
end
%
% specifiy the Term Coefficient Field
%
theTermCoefficientField = cfdGetMeshField(['Viscosity' theFluidTag]);
visc = theTermCoefficientField.phi;

%---------------------- End Term Info ----------------------%
%
%
geodiff = [theMesh.faces(iBFaces).geoDiff]';
Tf = [theMesh.faces(iBFaces).T]';
area = [theMesh.faces(iBFaces).area]';
n = [theMesh.faces(iBFaces).Sf]'./[area area area];
iOwners = [theMesh.faces(iBFaces).iOwner]';

TM=cfdGetUCoef(iPatch,theFluidTag);

%/////////////////////////////////////////////////////////
vel = theTermField.phi(iOwners,:);
velb =  theTermField.phi(iBElements,:);

mag = dot(vel',n')';

vel_n = [mag mag mag].*n;
vel_t = vel - vel_n;

magb = dot(velb',n')';
velb_n = [magb magb magb].*n;
velb_t = velb - velb_n;

vel_wall = velb_t;
nc = n*e;

Fc = -area.*TM.*((vel_t - vel_wall)*e);
%/////////////////////////////////////////////////////////
theFluxes.FLUXC1f(iBFaces) =   area.*TM.*(1- dot(nc',nc')');
theFluxes.FLUXC2f(iBFaces) = 0;
theFluxes.FLUXVf(iBFaces)  = - Fc - area.*TM.*(1-
dot(nc',nc')').*velc(iOwners);
theFluxes.FLUXTf(iBFaces)  = theFluxes.FLUXC1f(iBFaces) .*
velc(iOwners) + theFluxes.FLUXC2f(iBFaces) .* velc(iBElements) +
theFluxes.FLUXVf(iBFaces);

end
```

Listing 18.12 (continued)

18.5 Closure

The chapter discussed the implementation of boundary conditions in OpenFOAM®. It also presented the needed steps for implementing new boundary conditions in OpenFOAM® by detailing the various required stages for properly adding a no-slip boundary condition. A brief discussion of boundary conditions in uFVM was also presented. The next chapter is devoted to detailing the steps needed to solve a turbulent flow problem in OpenFOAM®.

Reference

1. OpenFOAM (2015) Version 2.3.x. http://www.openfoam.org