Cite as: Fahlbeck, J.: Implementation of an incompressible headLossPressure boundary condition. In Proceedings of CFD with OpenSource Software, 2020, Edited by Nilsson. H.,
http://dx.doi.org/10.17196/OS_CFD#YEAR_2020

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Implementation of an incompressible headLossPressure boundary condition

Developed for OpenFOAM-v1912

Author:
Jonathan Fahlbeck
Chalmers University of Technology
fahlbeck@chalmers.se

Peer reviewed by:
Erik Josefsson
Saeed Salehi
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 to 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 17, 2021

Learning outcomes

The main requirements of a tutorial in the course is that it should teach the 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-up cases with pressure driven flow.
- how to account for head losses.
- how velocity and pressure needs to be coupled with boundary conditions.

The theory of it:

- the theory of various pressure boundary conditions.
- the theory of continuity preservation through coupling with other fields.

How it is implemented:

- how a number of pressure boundary conditions are implemented
- how is the coupling between pressure and velocity boundary conditions implemented.

How to modify it:

- how to modify of the available totalPressure boundary condition.
- how to include any number of friction and/or minor losses to calculate the pressure on the patch.
- how to change the input definition to include surface elevation and atmospheric pressure.

Prerequisites

- How to run standard tutorials like the damBreak tutorial in OpenFOAM.
- Fundamentals of Computational Fluid Dynamics.
- How to customise a boundary conditions and do top-level application programming.
- Basic knowledge of head losses in pipes due to friction and minor effects.

Contents

Nomenclature

Acronyms

CFD	Computational Fluid Dynamics

English symbols

A	Area	m^2
d	Diameter	m
d_h	Hydraulic diameter	m
f	Friction loss coefficient	$-$
g	Gravity	$\mathrm{m/s^2}$
H	Head	m
k	Minor loss coefficient	$-$
L	Length	m
n	Normal vector	$-$
p	Pressure	$\mathrm{kg/(m \cdot s^2)}$
p_0	Total pressure	$\mathrm{kg/(m \cdot s^2)}$
Q	Volumetric flow rate	$\mathrm{m^3/s}$
U	Velocity	$\mathrm{m/s}$
Re	Reynolds number	$-$

Greek symbols

ϵ	Surface roughness	m
ν	Fluid kinematic viscosity	$\mathrm{m^2/s}$
ϕ	Volumetric flux	$\mathrm{m^3/(s \cdot m^2)}$
ρ	Fluid density	$\mathrm{kg^3/m^3}$

Superscripts

*	Kinematic

Subscripts

ana	analytical
d	diameter
p	patch
ref	reference
sim	simulated

Chapter 1

Theoretical background

1.1 Introduction

In a numerical framework one always wants to have as small computational domain as possible to be able to focus on the areas of interest. In a pump application for instance, the numerical domain does most often not include the entire piping system which the pump is connected to. This is a valid approximation in cases where one are certain of the losses and flow rates at all time. This is most likely not the case, say that the flow rate is unknown and the only known property is the height elevation between two water surfaces. The flow rate will thus vary depending on how the machine is operated. Pressure losses tend to scale quadratic to a change in flow rate. Hence, it becomes extensively difficult to in forehand know the pressure loss or flow rate for a general case.

An option to overcome this difficulty is to include loss-factors at the boundaries of the computational domain. In this manner the correct pressure loss to a corresponding flow rate can always be computed and updated. This present work focus on the development of an incompressible boundary condition that can take any number of loss-factors caused. The losses can by one-time occurrences in the flow path (minor) or due to friction in pipes. The static pressure on the patch is adjusted due to the computed losses. The flow rate itself is computed by the solver as a balance between available pressure, losses and flow rate. The boundary condition aims to mimic a full system with two open reservoir surfaces, up- and down-stream. It is the height difference between the reservoirs that drives the flow and the losses that limits the flow.

In most fluid applications head or pressure losses are essential to the overall performance of a system or a device. In a general case losses are extensively difficult or even impossible to formulate analytically. But there are certain idealised cases where analytical models exists. One of these cases is pipe flow [1, 2]. There are a number of empirical models to determine the pressure loss in pipes due to friction, bends, valves, etc. These modeling concepts are implemented in the developed `headLossPressure` boundary condition, which this reports focus on.

1.1.1 Static, dynamic, hydrostatic, and total pressure

In fluid mechanics it is common to refer to a number of different kinds of pressures. The most straightforward is the static pressure, which is the actual "pressure" in the fluid, denoted by p. The stresses in the normal direction of a fluid particle is the static pressure [2]. The second pressure is the dynamic pressure. It is a theoretical concept describing what the static pressure caused by motion would become if the fluid is brought isentropicaly to a standstill. The dynamic and static pressure can be combined into a total pressure

$$p_0 = \underbrace{p}_{\text{static}} + \underbrace{\rho \frac{U^2}{2}}_{\text{dynamic}} \quad . \tag{1.1}$$

Here U is the fluid velocity and ρ is the fluid density. The total pressure is commonly referred to as the stagnation pressure.

Lastly we have the hydrostatic pressure, caused by variation in altitude and defined as $\rho g H$. The height difference is denoted by H, and the gravity by g. By combining the static, dynamic, and hydrostatic pressure we arrive to the famous Bernoulli Equation

$$p + \rho \frac{U^2}{2} + \rho g H = \text{Constant along a streamline.} \tag{1.2}$$

The equation states that the energy of the flow is constant along a streamline, and the equation can be used to convert between static, dynamic, and hydrostatic pressures within a system [2]. The Bernoulli equation have one large drawback, it assumes that there are no losses in the flow [1]. To overcome this problem the concept of head or pressure losses are introduced to the Bernoulli equation, discussed in section 1.1.2.

OpenFOAM uses the kinematic pressure for incompressible cases. All developments in present work are made for incompressible cases. The kinematic pressure is simply the pressure divided by the density

$$p^* = \frac{p}{\rho},$$

where p is the pressure and ρ the density.

1.1.2 Pressure losses

To account for pressure losses in a system Equation 1.2 is expanded to include pressure losses as

$$p_1 + \underbrace{\frac{\rho U_1^2}{2}}_{U_1=0} + \rho g H_1 = p_2 + \frac{U_2^2}{2} + \rho g H_2 + \Delta p_{\text{minor}} + \Delta p_{\text{friction}}. \tag{1.3}$$

Here Δp_{minor} is the pressure loss due to one-time occurrences (minor losses) in the flow path and $\Delta p_{\text{friction}}$ is the pressure loss caused by friction in pipes. The equation is written for the system showed in Figure 1.1. A common assumption is that the velocity in a large reservoir is equal to zero, $U_1 = 0$. The height between locations 1 and 2 is H. It is possible to divide the Equation 1.3 by the density and use the kinematic pressure, as done in OpenFOAM. Thus, the Bernoulli equation between the reservoir surface, 1, and the down-stream position, 2, in the system can be written as

$$p_2^* \underbrace{=}_{H_1 - H_2 = H} p_1^* - \frac{U_2^2}{2} + gH - \Delta p^*_{\text{minor}} - \Delta p^*_{\text{friction}}. \tag{1.4}$$

Note that Equation is written in the form of kinematic pressure, $p^* = p/\rho$.

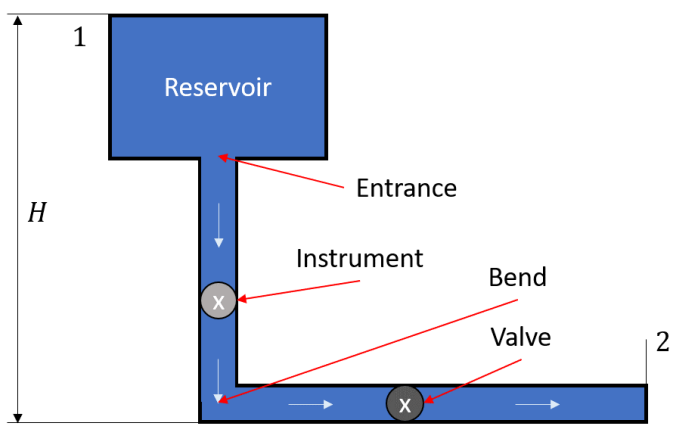


Figure 1.1: Schematic view of a system. The height difference drives the flow between 1 and 2.

The minor losses are caused by changes in the flow path, e.g. bends, valves, expansions, etc. The head loss caused by the changes in the path is expressed as a function of a single loss-factor, k, called the minor loss coefficient. The minor loss coefficient is usually a tabulated value provided by the manufacturer of the device or from tables in textbooks. The calculation of the minor kinematic pressure losses is according to

$$\Delta p^*_{\text{minor}} = \sum_{i=1}^{N} \frac{U_i^2}{2} k_i. \tag{1.5}$$

The calculation of friction losses are similar to the minor losses, but it includes a friction loss coefficient, f, which is cumbersome to obtain. The general formula for calculating the kinematic pressure loss due to friction is given by

$$\Delta p^*_{\text{friction}} = \sum_{j=1}^{N} \frac{U_j^2}{2} \frac{L_j}{d_{h,j}} f_j. \tag{1.6}$$

Here L and d are the length and hydraulic diameter of the pipe, respectively. The friction loss coefficient, f, is readily calculated in a laminar case as

$$f = \frac{64}{\text{Re}_\text{d}}, \tag{1.7}$$

$$\text{Re}_\text{d} = \frac{d_\text{h} U}{\nu}. \tag{1.8}$$

Here Re_d is the Reynolds number based on the hydraulic diameter and ν is the kinematic viscosity. The transition from laminar to turbulent is assumed to occur at $\text{Re} = 2300$ for pipe flow [1]. In a turbulent case the friction coefficient is read from a Moody-chart or calculated using the Colebrook

$$\frac{1}{f^{1/2}} = -2.0 \log_{10} \left(\frac{\epsilon/d}{3.7} + \frac{2.51}{\text{Re}_\text{d} f^{1/2}} \right). \tag{1.9}$$

Here ϵ is the pipe surface roughness. The Colebrook Equation needs to be solved iteratively since it is not an explicit formulation of the friction loss coefficient. A method to obtain a rough estimate of the friction coefficient is to use the more direct Haaland equation

$$f^{1/2} \approx \frac{1}{-1.8 \log_{10} \left(\left(\frac{\epsilon/d}{3.7} \right)^{1.11} + \frac{6.11}{\text{Re}_\text{d}} \right)}. \tag{1.10}$$

To solve the Colebrook Equation 1.9, the Haaland Equation 1.10 can be used as the initial guess for the iteration procedure. The simplest design of an iteration procedure is to use the value of f from a previous iteration to estimate a new value of f. The relative difference between the new and the old value of f can be used to evaluate the convergence of the friction coefficient.

Chapter 2

Boundary conditions and coupling

2.1 fixedValue and zeroGradient

The `fixedValue` condition prescribes a Dirichlet boundary condition, and the `zeroGradient` a homogeneous Neumann condition [3, 4]. The Dirichlet condition does not impose that it must be *a fixed value* in time and space on the patch, but the *values* on the patch are explicitly specified. The homogeneous Neumann condition imposes that the field/variable has a zero gradient normal to the patch, as

$$\left.\frac{\partial \varphi}{\partial \boldsymbol{n}}\right|_{\text{patch}} = 0. \tag{2.1}$$

Here φ is an arbitrary quantity and $\boldsymbol{n}$ is the normal vector of the patch. The `fixedValue` and the `zeroGradient` boundary conditions are located at:

```
$FOAM_SRC/finiteVolume/fields/fvPatchFields/basic
```

The other boundary conditions discussed in this chapter are located at:

```
$FOAM_SRC/finiteVolume/fields/fvPatchFields/derived
```

2.2 totalPressure

The `totalPressure` boundary condition prescribes a static pressure, Dirichlet, condition at the patch. For outgoing flow the pressure at the patch is simply the user-supplied `p0` value. For incoming flow the pressure at the patch is adjusted with the flow velocity. The calculations of the static kinematic pressure on the patch are

$$p^*_{\text{p},i} = p^*_{0\text{p}} - 0.5|U_{\text{p},i}|^2 \qquad \text{for inflow,} \tag{2.2}$$

$$p^*_{\text{p},i} = p^*_{0\text{p}} \qquad \text{for outflow.} \tag{2.3}$$

Here $p^*_{\text{p},i}$ is the kinematic pressure at face i on the patch, $p^*_{0\text{p}}$ is the user supplied total kinematic pressure, and U_i is the velocity magnitude at face i on the patch. Recall that kinematic pressure is the pressure divided by the density. Extracts of the main function, `updateCoeffs`, for the `totalPressure` boundary condition are found below. At line 172 it is shown that the dimension of the pressure determines how the patch calculations should be handled. The two inputs `p0p` and `Up` correspond to the user-supplied total pressure and the velocity at the patch, respectively.

Values used by `totalPressureFvPatchScalarField.C`

```
158 void Foam::totalPressureFvPatchScalarField::updateCoeffs
159 (
160     const scalarField& p0p,
161     const vectorField& Up
162 )
163 {
164     if (updated())
165     {
166         return;
167     }
168
169     const fvsPatchField<scalar>& phip =
170         patch().lookupPatchField<surfaceScalarField, scalar>(phiName_);
171
172     if (internalField().dimensions() == dimPressure)
```

Between lines 211-215 is the calculation for an incompressible case made. The expression in the code equals that showed by Equations 2.2 and 2.3. This is because the function `pos0(phip)` is equal to zero for inflow and one for outflow.

Calculation of patch pressure from `totalPressureFvPatchScalarField.C`

```
211     else if (internalField().dimensions() == dimPressure/dimDensity)
212     {
213         // Incompressible flow
214         operator==(p0p - 0.5*(1.0 - pos0(phip))*magSqr(Up));
215     }
```

At line 240 it is shown that the `totalPressure` boundary condition imports the velocity vector field at the patch via the function `patch().lookupPatchField`.

Input to the main function, from `totalPressureFvPatchScalarField.C`

```
235 void Foam::totalPressureFvPatchScalarField::updateCoeffs()
236 {
237     updateCoeffs
238     (
239         p0(),
240         patch().lookupPatchField<volVectorField, vector>(UName())
241     );
242 }
```

2.2.1 uniformTotalPressure

The `uniformTotalPressure` is similar to the regular `totalPressure` boundary condition. The main difference is that the `uniformTotalPressure` allows a time varying total pressure at the patch via the `Function1` class. The presented codes below are extract from the main function, `updateCoeffs`, for the `uniformTotalPressure` boundary condition. At line 156 it is shown that the value of `p0` is imported, and may vary with time.

Values used by `uniformTotalPressureFvPatchScalarField.C`

```
146 void Foam::uniformTotalPressureFvPatchScalarField::updateCoeffs
147 (
148     const vectorField& Up
149 )
150 {
151     if (updated())
152     {
153         return;
154     }
155
156     scalar p0 = p0_->value(this->db().time().timeOutputValue());
157
```

```
158     const fvsPatchField<scalar>& phip =
159         patch().lookupPatchField<surfaceScalarField, scalar>(phiName_);
160
161     if (internalField().dimensions() == dimPressure)
```

At line 161 it is once again seen that the dimension of the pressure determines what type pressure calculation should be made. From lines 200-204 are the calculation made for an incompressible case. The static pressure calculation on the patch is equal to that mentioned in Section 2.2.

Patch pressure calculation for `uniformTotalPressureFvPatchScalarField.C`

```
200     else if (internalField().dimensions() == dimPressure/dimDensity)
201     {
202         // Incompressible flow
203         operator==(p0 - 0.5*(1.0 - pos0(phip))*magSqr(Up));
204     }
```

Between lines 224-227 are the inputs for the `updateCoeffs` function shown. The boundary condition use `patch().lookupPatchField` to obtain the velocity at the patch.

Velocity input for `uniformTotalPressureFvPatchScalarField.C`

```
224 void Foam::uniformTotalPressureFvPatchScalarField::updateCoeffs()
225 {
226     updateCoeffs(patch().lookupPatchField<volVectorField, vector>(UName_));
227 }
```

2.3 prghPressure

The `prghPressure` is intended for solvers that utilise the hydrostatic pressure for the pressure calculations. The boundary condition prescribes a `fixedValue` on the patch based on the hydrostatic pressure as

$$p_{\mathrm{rgh},i} = p + \rho_i g(H_i - H_{\mathrm{ref}}). \tag{2.4}$$

Here p_{rgh} is the hydrostatic pressure on the patch, p is the user-supplied static pressure, ρ is the fluid density, g is the gravity, H is the elevation of the patch, H_{ref} is the reference elevation. The index i corresponds to face i on the patch. The code that calculates and prescribes the hydrostatic pressure at the patch is presented below. The equivalent code of Equation 2.4 is located at line 168. The variable `p_` is the user-supplied static pressure, `rhop` is the density, `g.value()` is the gravity, `patch().Cf()` is the center of the faces, and `ghRef.value()` is the reference height times the gravity.

Main function in `prghPressureFvPatchScalarField.C`

```
143 void Foam::prghPressureFvPatchScalarField::updateCoeffs()
144 {
145     if (updated())
146     {
147         return;
148     }
149
150     const scalarField& rhop = patch().lookupPatchField<volScalarField, scalar>
151     (
152         rhoName_
153     );
154
155     const uniformDimensionedVectorField& g =
156         meshObjects::gravity::New(db().time());
157
158     const uniformDimensionedScalarField& hRef =
159         db().lookupObject<uniformDimensionedScalarField>("hRef");
160
```

```
    dimensionedScalar ghRef
    (
        mag(g.value()) > SMALL
      ? g & (cmptMag(g.value())/mag(g.value()))*hRef
      : dimensionedScalar("ghRef", g.dimensions()*dimLength, 0)
    );

    operator==(p_ - rhop*((g.value() & patch().Cf()) - ghRef.value()));

    fixedValueFvPatchScalarField::updateCoeffs();
}
```

The `prghPressure` is as a `fixedValue` condition, but for hydrostatic pressure. It should be used in combination with a homogeneous Neumann condition for velocity. This is because it is otherwise not possible to balance the flow velocity with the `prghPressure` and uphold continuity.

2.3.1 prghTotalPressure

The `prghTotalPressure` is a combination of the `totalPressure` and the `prghPressure` boundary conditions. The `prghTotalPressure` can be used at both inlets and outlets, just as the `totalPressure` or the `uniformTotalPressure`.

2.4 Velocity coupling

In OpenFOAM there are a number of velocity boundary conditions named `pressure...Velocity`, where the "dots" indicates the different alternatives. They all have in common that they are used together with a Dirichlet condition for pressure. The basics with the various `pressure...Velocity` boundary conditions is that they are coupled to the pressure boundary condition at the patch via the volumetric face flux. The simplest boundary condition to handle inflow velocity, together with a pressure boundary condition of the `total` type, is the `pressureInletVelocity` condition. This boundary condition is described in Section 2.4.1. There are also a number of boundary conditions ending with `...OutletVelocity`. They are extended versions of a boundary condition named similarly, but without the "`Outlet`" in the name. The extended versions can be used at both inlets and outlets since they allow for reverse flow via utilising a `zeroGradient` for positive flux (outflow). The boundary conditions that do not contain "`Outlet`" in the name should only be used at inlets.

2.4.1 pressureInletVelocity

The `pressureInletVelocity` boundary condition is intend for inflow only. The velocity is obtained from the flux in the normal direction of the face on the patch as

$$\boldsymbol{U}_{\mathrm{p},i} = \boldsymbol{n}_i \frac{\phi_i}{A_i}. \tag{2.5}$$

Here $\boldsymbol{U}_\mathrm{p}$ is the velocity vector at the patch, $\boldsymbol{n}$ is the face normal vector, ϕ is the volumetric face flux, A is the face area, and index i represents face i on the patch.

Below is the main function for the `pressureInletVelocity` boundary condition shown. The volumetric face flux, `phip`, face normal vector, `n`, and face area, `magS`, are read by the boundary condition between lines 112-119. For an incompressible case the patch velocity is calculated, according to Equation 2.5, at line 123.

Main function in `pressureInletVelocityFvPatchVectorField.C`

```
105 void Foam::pressureInletVelocityFvPatchVectorField::updateCoeffs()
106 {
107     if (updated())
108     {
109         return;
```

```
    }

    const surfaceScalarField& phi =
        db().lookupObject<surfaceScalarField>(phiName_);

    const fvsPatchField<scalar>& phip =
        patch().patchField<surfaceScalarField, scalar>(phi);

    tmp<vectorField> n = patch().nf();
    const Field<scalar>& magS = patch().magSf();

    if (phi.dimensions() == dimVelocity*dimArea)
    {
        operator==(n*phip/magS);
    }
    else if (phi.dimensions() == dimDensity*dimVelocity*dimArea)
    {
        const fvPatchField<scalar>& rhop =
            patch().lookupPatchField<volScalarField, scalar>(rhoName_);

        operator==(n*phip/(rhop*magS));
    }
    else
    {
        FatalErrorInFunction
            << "dimensions of phi are not correct"
            << "\n    on patch " << this->patch().name()
            << " of field " << this->internalField().name()
            << " in file " << this->internalField().objectPath()
            << exit(FatalError);
    }

    fixedValueFvPatchVectorField::updateCoeffs();
}
```

Imagine an inflow patch with the `pressureInletVelocity` and `totalPressure` boundary conditions used for velocity and pressure, respectively. The volumetric face flux, ϕ, is used to calculate the patch velocity, $\boldsymbol{U}_{\mathrm{p}}$. The pressure boundary condition use the calculated velocity to adjust the static pressure on the patch. The face flux is later adjusted and correct by the solver algorithm, e.g. `SIMPLE`, to balance flow and pressure. The corrections made in the solver algorithm is not covered in this report.

2.4.2 pressureDirectedInletVelocity

The `pressureDirectedInletVelocity` is a velocity inlet boundary condition with the option to specify the flow inlet direction. The velocity at a face i on the patch is calculated as

$$\boldsymbol{U}_{\mathrm{p},i} = \boldsymbol{n}\frac{\phi_i}{A_{n,i}}. \tag{2.6}$$

Here $\boldsymbol{n}$ is the user-supplied flow direction vector, A_n is the projected face area in the $\boldsymbol{n}$ direction, ϕ is the volumetric flux, and index i represents face i on the patch. A pressure boundary condition of the `total` type later uses the same velocity calculated by this boundary condition to subscribe the static pressure on the patch. Extracts from the definition file of `pressureDirectedInletVelocity` boundary condition is found below. The coded that correspond to equation 2.6 is found at line 155, for an incompressible case.

Extract from the main function in `pressureDirectedInletVelocityFvPatchVectorField.C`

```
    const surfaceScalarField& phi =
        db().lookupObject<surfaceScalarField>(phiName_);

    const fvsPatchField<scalar>& phip =
        patch().patchField<surfaceScalarField, scalar>(phi);
```

```

    tmp<vectorField> n = patch().nf();
    tmp<scalarField> ndmagS = (n & inletDir_)*patch().magSf();

    if (phi.dimensions() == dimVelocity*dimArea)
    {
        operator==(inletDir_*phip/ndmagS);
    }
```

2.4.3 pressureInletUniformVelocity

The `pressureInletUniformVelocity` boundary condition is an inlet velocity condition that generates a uniform velocity profile in the face normal direction. This is accomplished by averaging the face flux and subscribe the same velocity at all faces on the patch as

$$\boldsymbol{U}_{\mathrm{p}} = \boldsymbol{n}_i \frac{\sum_{i=1}^{N} (\boldsymbol{A}_i \cdot \boldsymbol{U}_{\mathrm{p},i})}{\sum_{i=1}^{N} A_i} \tag{2.7}$$

Here N is the number of faces on the patch, and $\boldsymbol{A}$ is the face area vector. Corresponding code from the main function for the `pressureInletUniformVelocity` condition is shown below.

Extract from the main function in `pressureInletUniformVelocityFvPatchVectorField.C`.

```
        operator==(patch().nf()*gSum(patch().Sf() & *this)/gSum(patch().magSf()));
```

It is not mentioned in the description of the boundary condition that it should only be used for a flat patch. The author strongly advise the reader to not use this boundary condition for patch that is not flat. This is because a curved patch has varying face normals.

2.4.4 pressureNormalInletOutletVelocity

The boundary condition is similar to the `pressureInletVelocity`, this extended version also allows for reverse flow via subscribing a `zeroGradient` condition for positive flux. The `Normal` in the name corresponds to that the inflow is only allowed in the face normal direction, as discussed in Section 2.4.1. For the `pressureInletVelocity`, values contribute to the equation system using the `operator==(...)`, no such function is found in the `pressureNormalInletOutletVelocity` source files. The main differences between the main functions of `pressureInletVelocity` and `pressureNormalInletOutletVelocity` is that the latter use `refValue()`, and `valueFraction()` to evaluate the patch velocity with the `mixedFvPatchField` class, see the code below. The `refValue()` shown at line 135 can be compared to the `operator` expression for the `pressureInletVelocity`, since this is where the equation of the boundary condition is formulated. The `valueFraction()` at line 154 keeps track if the flow is inflow or outflow with the `pos0(phip)` function. To update the values at the patch the `mixedFvPatchField` class is used, shown at line 156, as the `pressureNormalInletOutletVelocity` inherit functionality from that class.

Main function in `pressureNormalInletOutletVelocityFvPatchVectorField.C`

```
117 void Foam::pressureNormalInletOutletVelocityFvPatchVectorField::updateCoeffs()
118 {
119     if (updated())
120     {
121         return;
122     }
123
124     const surfaceScalarField& phi =
125         db().lookupObject<surfaceScalarField>(phiName_);
126
127     const fvsPatchField<scalar>& phip =
```

```
        patch().patchField<surfaceScalarField, scalar>(phi);

    tmp<vectorField> n = patch().nf();
    const Field<scalar>& magS = patch().magSf();

    if (phi.dimensions() == dimVelocity*dimArea)
    {
        refValue() = n*phip/magS;
    }
    else if (phi.dimensions() == dimDensity*dimVelocity*dimArea)
    {
        const fvPatchField<scalar>& rhop =
            patch().lookupPatchField<volScalarField, scalar>(rhoName_);

        refValue() = n*phip/(rhop*magS);
    }
    else
    {
        FatalErrorInFunction
            << "dimensions of phi are not correct"
            << "\n    on patch " << this->patch().name()
            << " of field " << this->internalField().name()
            << " in file " << this->internalField().objectPath()
            << exit(FatalError);
    }

    valueFraction() = 1.0 - pos0(phip);

    mixedFvPatchVectorField::updateCoeffs();
}
```

In the `mixedFvPatchField` class the `operator` is found, as shown between lines 158 - 167 in the code below. It is the `valueFraction` that dictates if the `refValue` from the boundary condition should be used or not. The `valueFraction` is equal to one if negative face flux (inflow), otherwise it is zero and a homogeneous Neumann condition is used.

Corresponding `updateCoeffs()` in the `mixedFvPatchField.C` file.

```
151 void Foam::mixedFvPatchField<Type>::evaluate(const Pstream::commsTypes)
152 {
153     if (!this->updated())
154     {
155         this->updateCoeffs();
156     }
157
158     Field<Type>::operator=
159     (
160         valueFraction_*refValue_
161       +
162         (1.0 - valueFraction_)*
163         (
164             this->patchInternalField()
165           + refGrad_/this->patch().deltaCoeffs()
166         )
167     );
168
169     fvPatchField<Type>::evaluate();
170 }
```

The following velocity boundary conditions, with `Outlet` in the name, are only described briefly. The authors intention with this is to highlight the reader that they exist. The implementations are similar to the `pressureNormalInletOutletVelocity`, but with some additional functionalities for the various alternatives.

2.4.5 pressureInletOutletVelocity

This boundary condition is similar to the other `pressure...Velocity` boundary conditions. The difference is that it utilises the velocity field instead of the volumetric flux divided by the face area to subscribe the patch velocity. This boundary condition allows for reverse flow via subscribing a `zeroGradient` condition for positive flux. The user may also supply a tangential velocity. This boundary condition inherit from the `directionMixedFvPatchField` class instead of the `mixedFvPatchField` class, as the `pressureNormalInletOutletVelocity` does. In the code below is an extract of the definition file from the `directionMixedFvPatchField` class. The `transform` function is utilised to formulate both the `normalValue`, at line 164, and the `transformGradValue`, at line 167. The `operator` at line 172 combine the `normalValue` and the `transformGradValue` to set the values or gradients at the patch. At line 167 the velocity vector is obtained via the function call `this->patchInternaField()`. This is because the boundary condition is applied for velocity.

Corresponding `updateCoeffs()` in the `directionMixedFvPatchField.C` file.

```
157 void Foam::directionMixedFvPatchField<Type>::evaluate(const Pstream::commsTypes)
158 {
159     if (!this->updated())
160     {
161         this->updateCoeffs();
162     }
163
164     tmp<Field<Type>> normalValue = transform(valueFraction_, refValue_);
165
166     tmp<Field<Type>> gradValue =
167         this->patchInternalField() + refGrad_/this->patch().deltaCoeffs();
168
169     tmp<Field<Type>> transformGradValue =
170         transform(I - valueFraction_, gradValue);
171
172     Field<Type>::operator=(normalValue + transformGradValue);
173
174     transformFvPatchField<Type>::evaluate();
175 }
```

The `valueFraction` is defined differently for the `pressureInletOutletVelocity` than for the `pressureNormalInletOutletVelocity`. Here it is a `symmTensorField`, defined as:

`valueFraction()` in the `pressureInletOutletVelocityFvPatchVectorField.C` file.

```
180     valueFraction() = neg(phip)*(I - sqr(patch().nf()));
```

The function `neg(phip)` returns one if the face flux is negative (inflow), otherwise the function is zero. In the code, `I` is the identity tensor, and `patch().nf()` is the face normal vector. The boundary condition is made possible since `valueFraction` is a tensor, and the `transform` function is used. Note that `refValue` is zero if the user do not supply a tangential velocity in the boundary file dictionary.

2.4.6 pressureDirectedInletOutletVelocity

This boundary condition is the same as the `pressureDirectedInletVelocity`, but it can handle reverse flows via subscribing a homogeneous Neumann condition for positive volumetric flux (outflow). This boundary condition use the `mixedFvPatchField` class to apply the boundary condition.

2.4.7 pressureInletOutletParSlipVelocity

Similar to `pressureNormalInletOutletVelocity`, but instead of just allowing normal inflow, a slip condition is applied tangentially to the patch. This boundary condition inherit from the `mixedFvPatchField` class.

2.5 Pressure and velocity combinations

Table 2.1 show the possible pressure and velocity boundary condition combinations. The table is derived based on the content stated in this chapter and assumes that the pressure boundary condition is applied. The marker indicates if the velocity boundary condition is suitable, ✓, or unsuitable/impossible, ✗, to use in combination with the pressure boundary condition. It is for example not possible to use `fixedValue` for both pressure and velocity, as the equation system would become over-determined. It is also not possible to use `zeroGradient` for both fields since the equation system is then under-determined. Combinations of `fixedValue` and `zeroGradient` is however possible to use. The combination `totalPressure` and `pressureInletVelocity`, for pressure and velocity respectively, is valid at the inlet. This is due to that the pressure and velocity at the patch balances one another with the user-supplied total pressure, which is not the same as the static pressure. The same combination can not be used at the outlet since the user-supplied "total pressure" is in fact the static pressure in this case. The pressure is hence not adjusted to the velocity, and the velocity is calculated according to the face flux. This combination would actually generate a `fixedValue` for both fields at the outlet with no possibility to conserve the mass. A more appropriate velocity boundary condition at the outlet is the `pressureInletOutletVelocity`, allowing conservation of mass via the usage of a homogeneous Neumann for velocity and a Dirichlet condition for pressure.

Table 2.1: Pressure and velocity boundary condition combinations for inflow, in, and outflow, out. Note that `*` = `pressure`, `**` = `Velocity`, ✓= possible combination, and ✗= impossible combination.

Pressure BC	`fixedValue`		`zeroGradient`		`totalPressure`		`uniformTotalPressure`		`prghPressure`		`prghTotalPressure`	
Velocity BC	in	out	in	out	in	out	in	out	in	out	in	out
`fixedValue`	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
`zeroGradient`	✓	✓	✗	✗	✗	✓	✗	✓	✓	✓	✗	✓
`*DirectedInlet**`	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✓	✗
`*DirectedInletOutlet**`	✗	✓	✗	✗	✓	✓	✓	✓	✗	✓	✓	✓
`*Inlet**`	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✓	✗
`*NormalInletOutlet**`	✗	✓	✗	✗	✓	✓	✓	✓	✗	✓	✓	✓
`*InletOutlet**`	✗	✓	✗	✗	✓	✓	✓	✓	✗	✓	✓	✓
`*InletOutletParSlip**`	✗	✓	✗	✗	✓	✓	✓	✓	✗	✓	✓	✓
`*InletUniform**`	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✓	✗

Chapter 3

The headLossPressure class

3.1 Origin

All the developments and line number references are made for the OpenFOAM-v1912 version, but the author has made sure that the implementations also work for the v2006 release. The newly developed boundary condition, `headLossPressure`, is based on the existing `totalPressure` boundary condition. The location of the `totalPressure` boundary condition is:

`$FOAM_SRC/finiteVolume/fields/fvPatchFields/derived/totalPressure`

The folder includes the declaration file `totalPressureFvPatchScalarField.H` and definition file `totalPressureFvPatchScalarField.C`. The boundary condition is compiled four levels up the directory tree, at the `finiteVolume` directory.

The main function, `updateCoeffs`, in the `totalPressure` boundary condition contain adoptions to handle various compressible cases, but the current developments only focuses on incompressible cases. The `updateCoeffs` function is found below.

The `updateCoeffs` function in file `totalPressureFvPatchScalarField.C`

```
void Foam::totalPressureFvPatchScalarField::updateCoeffs
(
    const scalarField& p0p,
    const vectorField& Up
)
{
    if (updated())
    {
        return;
    }

    const fvsPatchField<scalar>& phip =
        patch().lookupPatchField<surfaceScalarField, scalar>(phiName_);

    if (internalField().dimensions() == dimPressure)
    {
        if (psiName_ == "none")
        {
            // Variable density and low-speed compressible flow

            const fvPatchField<scalar>& rho =
                patch().lookupPatchField<volScalarField, scalar>(rhoName_);

            operator==(p0p - 0.5*rho*(1.0 - pos0(phip))*magSqr(Up));
        }
        else
        {
            // High-speed compressible flow

```

```
187             const fvPatchField<scalar>& psip =
188                 patch().lookupPatchField<volScalarField, scalar>(psiName_);
189
190             if (gamma_ > 1)
191             {
192                 scalar gM1ByG = (gamma_ - 1)/gamma_;
193
194                 operator==
195                 (
196                     p0p
197                    /pow
198                     (
199                         (1.0 + 0.5*psip*gM1ByG*(1.0 - pos0(phip))*magSqr(Up)),
200                         1.0/gM1ByG
201                     )
202                 );
203             }
204             else
205             {
206                 operator==(p0p/(1.0 + 0.5*psip*(1.0 - pos0(phip))*magSqr(Up)));
207             }
208         }
209
210     }
211     else if (internalField().dimensions() == dimPressure/dimDensity)
212     {
213         // Incompressible flow
214         operator==(p0p - 0.5*(1.0 - pos0(phip))*magSqr(Up));
215     }
216     else
217     {
218         FatalErrorInFunction
219             << " Incorrect pressure dimensions " << internalField().dimensions()
220             << nl
221             << "    Should be " << dimPressure
222             << " for compressible/variable density flow" << nl
223             << "    or " << dimPressure/dimDensity
224             << " for incompressible flow," << nl
225             << "    on patch " << this->patch().name()
226             << " of field " << this->internalField().name()
227             << " in file " << this->internalField().objectPath()
228             << exit(FatalError);
229     }
230
231     fixedValueFvPatchScalarField::updateCoeffs();
232 }
```

From lines 172-210 in `totalPressureFvPatchScalarField.C` the different compressible alternatives are handled by the boundary condition. These cases are disregarded in the current development. The focus lies on what occurs before the `if` statement at line 172 and inside the `else if` between lines 211-215. As can been seen at the different `if/else` statements, it is the dimension of the pressure that determines if the case is treated as compressible or incompressible. This is due to the fact that for an incompressible case OpenFOAM utilises the kinematic pressure, $p^* = p/\rho$, with the dimension $\mathrm{m}^2/\mathrm{s}^2$.

3.2 Modifications

In this section it is assumed that OpenFOAM-v1912 is sourced by the user. To make the new implementations as smooth as possible it is advisable to work with an appropriate directory structure. The following commands will help the user to set-up an appropriate file structure.

```
cd $WM_PROJECT_DIR
cp -r --parents src/finiteVolume/fields/fvPatchFields/derived/totalPressure $WM_PROJECT_USER_DIR/
cd $WM_PROJECT_USER_DIR/src/finiteVolume/fields/fvPatchFields/derived/
```

```
foamNewBC -f -s headLossPressure
rm headLossPressure/headLossPressureFvPatchScalarField.*
cp -r totalPressure/totalPressureFvPatchScalarField.* headLossPressure/
rm -r totalPressure/
cd headLossPressure/
mv totalPressureFvPatchScalarField.C headLossPressureFvPatchScalarField.C
mv totalPressureFvPatchScalarField.H headLossPressureFvPatchScalarField.H
sed -i s/'totalPressure'/'headLossPressure'/g headLossPressure*
```

By executing the commands above the user simply copy and rename the existing `totalPressure` boundary condition and make it compatible via the `foamNewBC` function. The folder structure of the new boundary condition is now:

```
headLossPressure/
├── headLossPressureFvPatchScalarField.C
├── headLossPressureFvPatchScalarField.H
└── Make/
    ├── files
    └── options
```

No additional modifications or libraries are required to compile the boundary condition with the `wmake` command at this point, but thus far the `headLossPressure` is just a copy of the `totalPressure`. The modifications made to make the `totalPressure` into the `headLossPressure` boundary condition are described in the following section, from reading to writing data.

3.2.1 Input data

The developed boundary condition requires a number of input variables to function properly. To give the reader an insight in what the outcome of the modifications will look like a compression between typical user-inputs for the `headLossPressure` and the `totalPressure` is found below. In the example input for `headLossPressure`: `patm` is the atmospheric kinematic pressure at a reservoir, `H` is the height of the reservoir surface in relation to the patch, `d` is the hydraulic diameter of the patch, `frictionLossFactor` and `minorLossFactor` are input variables used to calculate the friction and minor-losses.

Typical input `headLossPressure`

```
<patchName>
{
    type            headLossPressure;
    patm            uniform 0;  // m2/s2
    H               10;         // m
    d               1;          // m
    frictionLossFactor
    (
        (($d 2e-05 8) pipe1)
        (($d 2e-05 5) pipe2)
    );
    minorLossFactor
    (
        (($d 0.05) bend)
        (($d 0.01) valve)
    );
}
```

Typical input `totalPressure`

```
<patchName>
{
    type            totalPressure;
    p0              uniform 1e5;
}
```

As can been seen, there are practically no similarities between the user inputs above. To enable the new input, drastic modifications of both the declaration and definition files are required. To start the modifications, it is advised to remove all entities correlated to compressible flow cases, i.e. everything including `rhoName_`, `psiName_`, and `gamma_`. If OpenFOAM-v1912 is used the following commands will clean the `.H` and `.C` files so that they only include the necessary parts from the original `totalPressure` boundary condition.

```
sed -i '149,157d' headLossPressureFvPatchScalarField.H          # Removes unnecessary member data
sed -i '252,289d' headLossPressureFvPatchScalarField.H          # Removes unnecessary member functions
sed -i '/rhoName_/d' headLossPressureFvPatchScalarField.C       # Delete all lines containing rhoName_
sed -i '/psiName_/d' headLossPressureFvPatchScalarField.C       # Delete all lines containing rhoName_
sed -i '/gamma_/d' headLossPressureFvPatchScalarField.C         # Delete all lines containing rhoName_
sed -i '157,190d' headLossPressureFvPatchScalarField.C          # Removes compressible alternatives
sed -i s/'else if'/'if'/g headLossPressureFvPatchScalarField.C  # Change type in updateCoeffs
sed -i s/'p0_'/'patm_'/g headLossPressureFvPatchScalarField.*   # Change name of p0_ to patm_
sed -i s/'"p0"'/'"patm"'/g headLossPressureFvPatchScalarField.* # Change input from p0 to patm
sed -i s/'p0()'/'patm()'/g headLossPressureFvPatchScalarField.* # Change name of p0() to patm()
sed -i s/'p0p'/'patmp'/g headLossPressureFvPatchScalarField.*   # Change name of p0p to patmp
```

An exact view of how the cleaned `headLossPressureFvPatchScalarField` files should look like is found in Appendix A.1.

In the example of user inputs for the new boundary condition a number of new variables are added, namely the head, `H`, hydraulic diameter of the patch, `d`, friction loss-factors. `frictionLossFactor`, and minor loss-factors `minorLossFactor`. These need to be added as private member data in the beginning of the header file as:

Private member data in `headLossPressureFvPatchScalarField.H`

```
150 class headLossPressureFvPatchScalarField
151 :
152     public fixedValueFvPatchScalarField
153 {
154     // Private data
155
156         //- Name of the velocity field
157         word UName_;
158
159         //- Name of the flux transporting the field
160         word phiName_;
161
162         //- Atmospheric pressure
163         scalarField patm_;
164
165         //- Hydraulic diameter of the pipe
166         const scalar d_;
167
168         //- Gravity
169         const scalar g_;
170
171         //- Height
172         const scalar H_;
173
174         //- Tuple2 list of the friction loss factors ((d, epsilon, L) name)
175         typedef Tuple2<vector, word> indexedVector;
176         const List<indexedVector> frictionLossFactor_;
177
178         //- Tuple2 list of the minor loss factors ((d, k) name)
179         typedef Tuple2<vector2D, word> indexedVector2D;
180         const List<indexedVector2D> minorLossFactor_;
```

The variables `H_`, `d_` and `g_` correspond to the user-supplied inputs head, `H`, hydraulic diameter of the patch, `d`, and gravity, `g`. The gravity is not displayed in the "Typical input `headLossPressure`" code since it is an optional entry with $g = 9.81$ m/s^2 as the default value. These member data are added as `const scalar` since they are only scalars and the boundary condition is not allowed to change the values.

The more interesting inputs are the two loss-factors, that are based on `List` constructed by `Tuple2` types. The variable `indexedVector` is a `Tuple2` with the configuration (`vector`, `word`), defined at line 175. To generate the correct form of `frictionLossFactor_` a `const List` is made out of the variable `indexedVector`. This procedure is shown at line 176 and ensures that the `frictionLossFactor_` can take any number of inputs since it is a `List` of `indexedVector`. The boundary condition is not allowed to change the inputs due to the `const` statement. The

`minorLossFactor_` is based on the same philosophy, but with the usage of (`vector2D`, `word`) instead. One may ask why `vectorField` and `vector2DField` are not used for the loss-factors. This is because the output data is written in a binary format when using the `vectorField` class directly. This is of course only when the case is set-up as binary. The boundary condition itself cannot interpret a binary `vectorField` for the loss-factors. Hence the usage of the `Tuple2` class. To enable the `Tuple2` and `vector2D` the corresponding declaration files needs to be added, just after `#include "fixedValueFvPatchFields.H"` in the header file put:

```
#include "vector2DField.H"
#include "Tuple2.H"
```

Once the input variables are declared one can turn the attention towards the definition file. The variables need to be initialised properly and values must be correctly assigned to them. This is taken care of by the first two constructors in the definition file as shown below.

Assigning values to the member data in the `headLossPressureFvPatchScalarField.C`

```
Foam::headLossPressureFvPatchScalarField::headLossPressureFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF
)
:
    fixedValueFvPatchScalarField(p, iF),
    UName_("U"),
    phiName_("phi"),
    patm_(p.size(), Zero),
    d_(Zero),
    g_(9.81),
    H_(Zero),
    frictionLossFactor_(),
    minorLossFactor_()
{}

Foam::headLossPressureFvPatchScalarField::headLossPressureFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchScalarField(p, iF, dict, false),
    UName_(dict.lookupOrDefault<word>("U", "U")),
    phiName_(dict.lookupOrDefault<word>("phi", "phi")),
    patm_("patm", dict, p.size()),
    d_(dict.get<scalar>("d")),
    g_(dict.getOrDefault<scalar>("g",9.81)),
    H_(dict.get<scalar>("H")),
    frictionLossFactor_(dict.lookup("frictionLossFactor")),
    minorLossFactor_(dict.lookup("minorLossFactor"))
{
    if (dict.found("value"))
    {
        fvPatchField<scalar>::operator=
        (
            scalarField("value", dict, p.size())
        );
    }
    else
    {
        fvPatchField<scalar>::operator=(patm_);
    }
}
```

The variables are initialised and values are assigned to them in the second constructor. The hydraulic diameter of the pipe `d_` and the head `H_` need to be specified as the `dict.get<scalar>`

function is used. If the variables are not defined in the boundary dictionary an error message is given and the computation cannot start. A note regarding the patch hydraulic diameter, `d`, by using the value of 0 all the effects caused by minor and friction losses are disabled. This is not recommended since the whole idea with the boundary condition is to calculate and use the losses to modify the pressure on the patch. The gravity `g_` is assigned a default value by the statement `dict.getOrDefault<scalar>`, if the user does not assign the gravity in the boundary file. The `frictionLossFactor_` and `minorLossFactor_` are simply read by the boundary condition, from the boundary file, via the `dict.lookup` function. No default values are assigned to the loss factors, and at least one entry per loss-factor type must be supplied.

To make the boundary condition work properly with other functionalities in OpenFOAM it is required to also add the new variables in the last three constructors in the definition file. It is just a matter of copy, paste, change name, and ensure consistency. An example entry for the third constructor is found below.

Third constructor in `headLossPressureFvPatchScalarField.C`

```
Foam::headLossPressureFvPatchScalarField::headLossPressureFvPatchScalarField
(
    const headLossPressureFvPatchScalarField& ptf,
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const fvPatchFieldMapper& mapper
)
:
    fixedValueFvPatchScalarField(ptf, p, iF, mapper),
    UName_(ptf.UName_),
    phiName_(ptf.phiName_),
    patm_(ptf.patm_, mapper),
    d_(ptf.d_),
    g_(ptf.g_),
    H_(ptf.H_),
    frictionLossFactor_(ptf.frictionLossFactor_),
    minorLossFactor_(ptf.minorLossFactor_)
{}
```

As seen in the code, all the new variables are added in the same manner. For the last two constructors it is exactly the same procedure, but one needs to change from `ptf` to `tppsf`.

If everything is correctly implemented the boundary condition will compile without any errors. It is recommended to compile the boundary condition with `wmake` at this stage to ensure that everything is correctly done. Note that no new functionalities are added to the boundary condition that effects the pressure on the patch yet.

3.2.2 Modifications in the main function

The main function, `updateCoeffs`, in `headLossPressureFvPatchScalarField.C` should at this stage look like the one shown in Appendix A.1, lines 143 - 178. The modifications to the function starts right after the following statement, at the beginning of the function.

```
const fvsPatchField<scalar>& phip =
        patch().lookupPatchField<surfaceScalarField, scalar>(phiName_);
```

The first stage in the new `updateCoeffs` function is to calculate the average velocity at the patch. The calculation is made through summing up the volumetric face fluxes on the patch, `sumPhi`, and dividing by the total area of the patch, `totalArea`. Due to the fact that the `sumPhi` can be negative, the absolute value of the flux is used in the final calculation as

$$U_{\text{avg}} = \frac{|\phi|}{A}.$$

The summation is made through the usage of the `gSum` function, which makes sure that the summation works also for cases that are decomposed for parallel simulation. The code that calculates the average velocity is shown below.

Calculation of the average velocity, `Uavg`, in file `headLossPressureFvPatchScalarField.C`

```
230     // Calculating the Average velocity on the patch, Uavg = phi/A
231     scalar sumPhi = gSum(patch().lookupPatchField<surfaceScalarField, scalar>(phiName_));
232     const scalar& totalArea = gSum(patch().magSf());
233     scalar Uavg;
234     if (sumPhi != 0)
235     {
236         Uavg = mag(sumPhi)/totalArea;
237     }
238     else
239     {
240         Info << "Zero flux on patch" << endl;
241         sumPhi = 1e-20;
242         Uavg = mag(sumPhi)/totalArea;
243     }
```

The variable `sumPhi` is the sum of the flux on the patch and `totalArea` is the area of the patch. The if/else statement in the code is required for the rare, but possible, case of having zero flux at the face. This can for instance happen at the first time step/iteration of a simulation if the simulation is not properly initialised. The variables `sumPhi` and `Uavg` are later used in denominators, hence they are not allowed to equal zero. If there is zero face flux on the patch, a small value is prescribed to the variable `sumPhi` to prevent division by zero.

3.2.2.1 Minor loss factor calculation

Once the average velocity on the patch is known, the pressure drop caused by minor losses is calculated according to Equation 1.5. The code that calculates the minor losses is presented below.

Calculation of the minor pressure drop , `dpMinor`, in `headLossPressureFvPatchScalarField.C`

```
245     // Calculating the Minor losses
246     scalar dpMinor(0);
247     const scalar * dMinor;
248     const scalar * k;
249     for(int i = 0; i < minorLossFactor_.size(); i++)
250     {
251         dMinor = &minorLossFactor_[i].first().component(0);
252         k = &minorLossFactor_[i].first().component(1);
253         dpMinor = *k * sqr(Uavg * sqr(d_ / *dMinor))/2 + dpMinor;
254     }
```

The calculation is made via looping over the input list `minorLossFactor_` and using pointers for the loss factors, `k`, and the diameter of the loss, `dMinor`. Pointers are used to speed up the calculation. At line 253, the actual calculation of `dpMinor` is made and the equivalent mathematical expression is

$$\Delta p_{\mathrm{minor},i} = k_i \frac{\left(U_{\mathrm{avg}} \left(\frac{d_{\mathrm{p}}}{d_i}\right)^2\right)^2}{2} + \Delta p_{\mathrm{minor},i-1}. \tag{3.1}$$

Here the term $\left(\frac{d_{\mathrm{p}}}{d_i}\right)^2$ is correcting the velocity via comparing the hydraulic diameter of the patch, d_{p}, and the diameter specified for the minor loss , d_i. The fact that the volumetric flow rate must be the same is utilised to scale the average velocity at different cross sectional areas, and assuming that the hydraulic diameter determines the area. Both the diameter of the patch and the diameter of the minor loss are inputs read and required by the boundary condition. This originates from the fact that the incompressible flow rate is constant according to

$$Q = U_1 A_1 = U_2 A_2 \Rightarrow U_1 \frac{\pi d_1^2}{4} = U_2 \frac{\pi d_2^2}{4} \Rightarrow U_2 = U_1 \left(\frac{d_1}{d_2}\right)^2. \tag{3.2}$$

3.2.2.2 Friction loss factor calculation

The strategy used to calculate the pressure loss caused by friction is similar to the method for calculating the pressure drop caused by the minor losses. But before digging into the details, the kinematic viscosity is required to calculate the Reynolds number. One way to add this is via letting the user specify it in the boundary file. The viscosity is however already set in the `transportProperties` dictionary, a more appropriate method is to read the viscosity from that dictionary. This is achieved through lines 256 - 259 in `headLossPressureFvPatchScalarField.C`.

Read kinematic viscosity, `nu`, in `headLossPressureFvPatchScalarField.C`

```
256     // Reading Kinematic viscosity from transportProperties
257     const dictionary& transportProperties=
258         db().lookupObject<dictionary>("transportProperties");
259     const dimensionedScalar nuDimScal("nu", dimViscosity, transportProperties);
260     const scalar& nu = nuDimScal.value();
```

With the viscosity available everything is known to continue calculating the friction loss coefficient and the pressure drop due to friction. This is done via looping over the `frictionLossFactor_` and using a function to calculate the friction coefficient, f. The computations are made via the code below.

Calculation of the friction pressure drop, `dpFriction`, in `headLossPressureFvPatchScalarField.C`

```
262     // Calculating the Friction Losses
263     scalar f(0);
264     scalar dpFriction(0);
265     const scalar * dFriction;
266     const scalar * epsilon;
267     const scalar * L;
268     for(int i = 0; i < frictionLossFactor_.size(); i++)
269     {
270         dFriction = &frictionLossFactor_[i].first().component(0);
271         epsilon = &frictionLossFactor_[i].first().component(1);
272         L = &frictionLossFactor_[i].first().component(2);
273
274         f = fFunc(*dFriction,*epsilon,Uavg,nu);
275         dpFriction = f * *L / *dFriction * sqr(Uavg * sqr(d_  / *dFriction))/2 + dpFriction;
276     }
```

Pointers are used once again for the input variables: diameter of the loss, `dFriction`, surface roughness, `epsilon`, and length, `L`, to speed up computations.

The friction factor, `f`, is calculated via the function `fFunc`. To enable this function, it is first declared in the header file, just after the declaration of the `phiName()` functions, as:

Declaration of the `fFunc` in `headLossPressureFvPatchScalarField.H`

```
282             double fFunc
283             (
284                 const scalar&,
285                 const scalar&,
286                 const scalar&,
287                 const scalar&
288             );
```

The function has four input variables diameter of the loss, surface roughness, average velocity on the patch, and the kinematic viscosity. The function is defined in the definition file, before the `updateCoeffs` function.

Definition of the `fFunc` in `headLossPressureFvPatchScalarField.C`

```
165 double Foam::headLossPressureFvPatchScalarField::fFunc
166 (
167     const scalar& dia,
168     const scalar& eps,
169     const scalar& Usim,
```

```
170     const scalar& nu
171 )
172 {
173     const scalar& U = Usim*sqr(d_/dia);
174     const scalar& Re = dia*U/nu;
175
176     scalar f(0);
177
178     if ( Re == 0 )
179     {
180         f = 0;
181     }
182     else if ( Re < 2300 )
183     {
184         f = 64/Re;
185         Info << "Laminar Flow" << endl;
186         Info << "Re = " << Re << endl;
187     }
188     else
189     {
190         const scalar tol = 1e-5;
191         const scalar fInitaL = sqr(1/(-1.8*log10(6.9/Re + pow((eps/dia)/3.7 ,1.11))));
192         scalar fOld = fInitaL;
193
194         for(int i = 0; i < 20; i++)
195         {
196             f = sqr(1/(-2*log10((eps/dia)/3.7 + 2.51/(Re*sqrt(fOld)))));
197
198             if (mag(f - fOld)/fOld <= tol )
199             {
200                 break;
201             }
202             fOld = f;
203         }
204
205         if ( mag(f - fInitaL)/fInitaL >= 0.1 )
206         {
207             Info << "Using f Inital" << endl;
208             f = fInitaL;
209         }
210     }
211
212     return f;
213 }
```

The `fFunc` function first calculates the correct velocity for the pipe section according to Equation 3.2. Then the Reynolds number is computed to check if the flow is laminar or turbulent. There is also a fail-safe defined between lines 178 - 181 to set the friction coefficient to zero if the Reynolds number is zero. This can happen if the user specify the patch or loss diameter to zero. If the Reynolds number is less than 2300 the flow is regarded as laminar and Equation 1.7 is used, see line 188. However, if the flow is turbulent, Equation 1.9 is solved in an iterative manner, with Equation 1.10 as the initial guess. The iteration procedure is seen between line 194 - 203 in `headLossPressureFvPatchScalarField.C` and a flow chart is found in Figure 3.1. The method is based on that the old/initial value of f is used to predict a new value of f. If the relative difference between the old and new value of f is below a certain tolerance, `tol`, the iteration procedure is interrupted. As a final step in the `fFunc` function the iterated value of f is compared to the initial value, if the difference is too large the initial value is used and an error message is given. This is because the equation used to calculate the initial value should only differ from the Colebrook equation by ± 2 % [1], and there is a risk of that the iteration procedure has diverged.

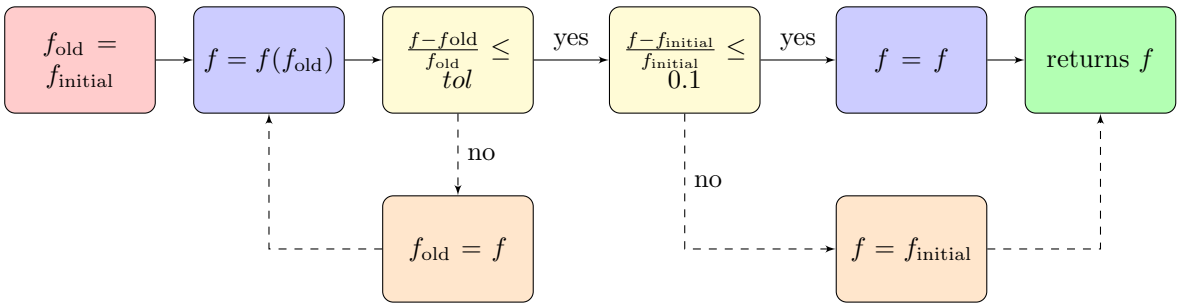


Figure 3.1: Flow chart of iteration procedure to attain the friction loss coefficient f.

Finally, when the friction coefficient has been estimated, the pressure drop due to friction is calculated according, as shown at line 275 in `headLossPressureFvPatchScalarField.C`. The equivalent mathematical expression is

$$\Delta p_{\text{friction},i} = f_i \frac{L_i}{d_i} \frac{\left(U_{\text{avg}} \left(\frac{d_{\text{p}}}{d_i}\right)^2\right)^2}{2} + \Delta p_{\text{minor},i-1}. \tag{3.3}$$

Here f_i is the friction loss-factor, L_i is the length of the loss, and d_i is the hydraulic diameter of the loss.

3.2.2.3 Patch pressure calculation

The pressure calculation on the patch is composed by four terms and the exact formulation is accordingly

$$p_{\text{p},i} = \underbrace{p_{\text{atm,p}}}_{\text{I}} - \underbrace{X_1(\phi_i) 0.5 |U_{\text{p,i}}|^2}_{\text{II}} + \underbrace{X_2(\phi_{\text{sum}})(\Delta p_{\text{friction}} + \Delta p_{\text{minor}})}_{\text{III}} + \underbrace{gH}_{\text{IV}}, \tag{3.4}$$

where the functions X are defined as

$$X_1(\phi_i) = \begin{cases} 1 \text{ if } \phi_i \leq 0 \\ 0 \text{ if } \phi_i > 0 \end{cases} \qquad X_2(\phi_{\text{sum}}) = \begin{cases} -1 \text{ if } \phi_{\text{sum}} < 0 \\ 1 \text{ if } \phi_{\text{sum}} > 0 \end{cases} . \tag{3.5}$$

Here the index i corresponds to face i on the patch. Note that the operator is made for every face, and that the static pressure on the patch will vary with the velocity if it is an inflow patch.

The first term, I, in Equation 3.4 corresponds to the user-supplied `patm` via the boundary dictionary. The second term, II, is the dynamic pressure. The reason for manipulating II with the X_1 expression is to ensure preservation of continuity in combination with a proper velocity boundary condition. The third term, III, is where the pressure losses are added or subtracted. The loss is added for outflow, meaning the patch pressure increases if there are downstream pipes, bends, valves, etc. For a patch with inflow it is the reverse order, the pressure decreases. This is due to the fact that for an inflow case the flow obstacles are upstream, resulting in a lower pressure at the location of the patch in relation to the reservoir surface. Lastly, IV, is the hydrostatic pressure term, where a difference in surface elevation is included. The user-supplied height must be in reference to the patch, and the same reference must be used if this boundary condition is used at both inlet and outlet.

The part of the code that utilises Equation 3.4 is found below. Note in the code that `patmp` is a list the size of patch with the value `patm`. The function `pos0(phip)` returns 1 if the volumetric flux, `phip`, is zero or positive (outflow), otherwise the function returns 0 (inflow).

Patch pressure calculation in headLossPressureFvPatchScalarField.C

```
        // Incompressible flow
        operator==
        (
            patmp - 0.5*(1.0 - pos0(phip))*magSqr(Up)
             + sumPhi/mag(sumPhi)*(dpFriction + dpMinor) + g_*H_
        );
```

If all the implementations in the main function, updateCoeffs, are correctly made the user should at this point be able to compile the boundary condition before moving on to the final section. If the code can not be compiled there is some error made during the modifications of the updateCoeffs function.

3.2.3 Output data

The final step of the boundary condition is to write data to the boundary file. To achieve this the following modifications to the write function are required. The variable g_ will only be written to the file if it is supplied by the user. Otherwise the boundary condition will not write g_ and the default value is used when restarting the simulation. This is achieved by the function writeEntryIfDifferent. The variables d_ and H_ are written to the boundary file as scalars with the writeEntry<scalar> function. The two loss-factors are simply written to the boundary file via writeEntry. The correct form of the write function is found below.

Write data to file headLossPressureFvPatchScalarField.C

```
void Foam::headLossPressureFvPatchScalarField::write(Ostream& os) const
{
    fvPatchScalarField::write(os);
    os.writeEntryIfDifferent<word>("U", "U", UName_);
    os.writeEntryIfDifferent<word>("phi", "phi", phiName_);
    os.writeEntry<scalar>("d", d_);
    os.writeEntryIfDifferent<scalar>("g", 9.81,g_);
    os.writeEntry<scalar>("H", H_);
    os.writeEntry("frictionLossFactor",frictionLossFactor_);
    os.writeEntry("minorLossFactor",minorLossFactor_);
    patm_.writeEntry("patm", os);
    writeEntry("value", os);
}
```

The code is now fully modified and it is time to compile the headLossPressure boundary condition and test its functionality on a suitable test case.

Chapter 4

Test Case

The whole idea of the `headLossPressure` boundary condition is that the boundary condition takes into account total pressure losses that occur outside the computational domain. This is important when evaluating the true capacity of an entire system based on upper and lower reservoirs and with unknown flow rates. A common approach in CFD is to include the important parts of the system and neglect how the flow reached the parts of interest. The test case to verify the functionality of the boundary condition is only a cylinder with the length of 1 m and diameter of 0.1 m. Boundary conditions for pressure and velocity are the `headLossPressure` and the `pressureInletOutletVelocity` at both inlet and outlet.

The purpose is to evaluate if the boundary condition can compute a reasonable flow rate and pressure drop for an entire system. Hence, the cylinder is included in the larger system shown in Figure 4.1, where the part labelled "CFD" is the simulation domain. The inlet is to the left, and the outlet to the right. It is the elevations of the "Upper" and "Lower" reservoirs that drives the flow and the losses in the system limits the flow.

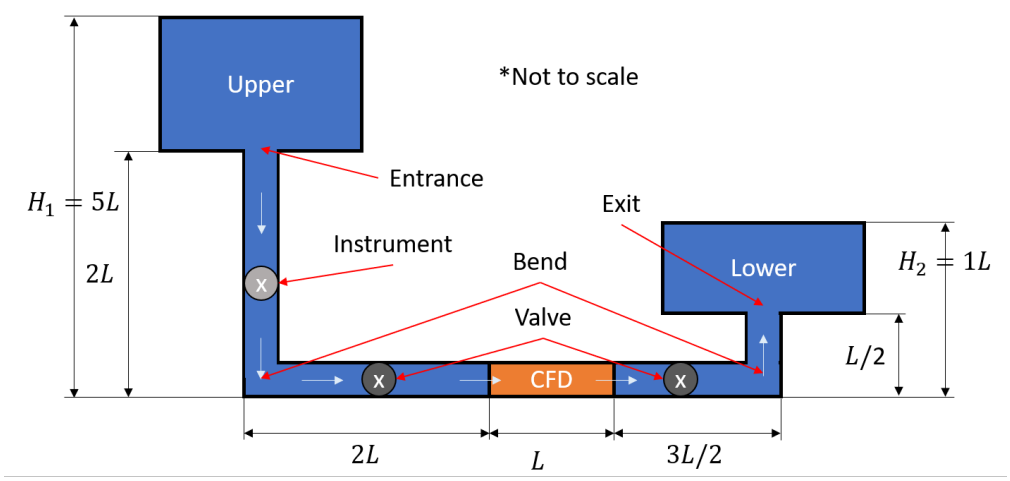


Figure 4.1: Schematic view of the entire system. The flow goes from the Upper to the Lower reservoir with the computational domain marked as CFD, and the reference length $L = 1$ m.

The upstream losses in the system are caused by friction in the pipe, bend, instrument, valve, and entrance. The downstream losses are also due to friction in the pipe, bend, valve, and exit effects. For simplicity the same diameter is used for all losses in the test case. Table 4.1 presents a summary of how the boundary condition is set to operate. Note that the values are arbitrary and do not reflect an exact case.

Table 4.1: Input values in boundary file `p`.

	Inlet		**Outlet**	
$\mathbf{p_{atm}}$	$0\ m^2/s^2$		$0\ m^2/s^2$	
H	5 m		1 m	
d	0.1 m		0.1 m	
Minor Loss-Factor	k		k	
Bend	0.2		0.2	
Valve	0.05		0.05	
Instrument	0.03		-	
Entrance/Exit	0.4		1.0	
Friction Loss-Factor	L [m]	ϵ [m]	L [m]	ϵ [m]
Horizontal pipe	2.0	2.0×10^{-5}	0.5	2.0×10^{-5}
Vertical pipe	2.0	2.0×10^{-5}	0.5	2.0×10^{-5}

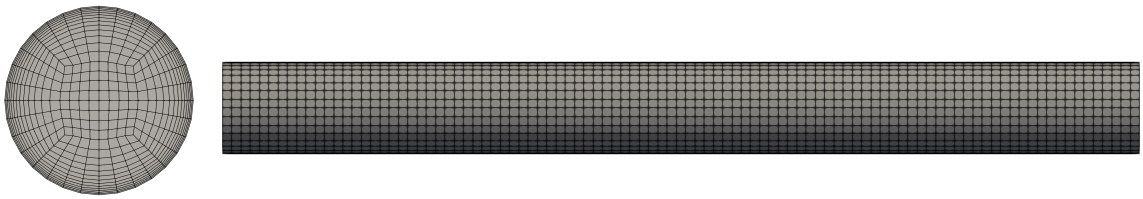
To set up the boundary condition according to Table 4.1, an example `p` boundary dictionary is provided below. Since the diameter is in fact the same for all losses, the `minorLossFactor` and `frictionLossFactor` could have been combined to just include a single input for each type of loss (minor and friction). But for demonstrational purposes all factors are written out in the `p` boundary dictionary.

Part of the boundary dictionary `p` , `epsilon=2e-5`, `dPipe=0.1`.

```
boundaryField
{
    inlet
    {
        type            headLossPressure;
        patm            uniform 0;
        H               5;
        d               0.1;
        minorLossFactor
        (
            (($dPipe 0.2) Bend)
            (($dPipe 0.05) Valve)
            (($dPipe 0.03) Instrument)
            (($dPipe 0.4) Entrance))
        );
        frictionLossFactor
        (
            (($dPipe $epsilon  2.0) Horizontal_pipe)
            (($dPipe $epsilon  2.0) Vertical_pipe)
        );
    }

    outlet
    {
        type            headLossPressure;
        patm            uniform 0;
        H               1;
        d               0.1;
        minorLossFactor
        (
            (($dPipe 0.2) Bend)
            (($dPipe 0.05) Valve)
            (($dPipe 1.0) Exit)
        );
        frictionLossFactor
        (
            (($dPipe $epsilon  0.5) Horizontal_pipe)
            (($dPipe $epsilon  0.5) Vertical_pipe)
        );
    }
```

The computational mesh is block-structured and contains roughly 47,500 cells. To account for turbulence the k-ω SST eddy viscosity model is used with standard inlet conditions and wall functions. The computational domain and mesh is shown in Figure 4.2.



(a) Inlet mesh (b) Mesh side view, inlet at left and outlet at right.

Figure 4.2: Computational domain and mesh. Note that the the figures are not to scale.

To enable the usage of the `headLossPressure` boundary condition the library for the boundary condition needs to be added to the `controlDict` as.

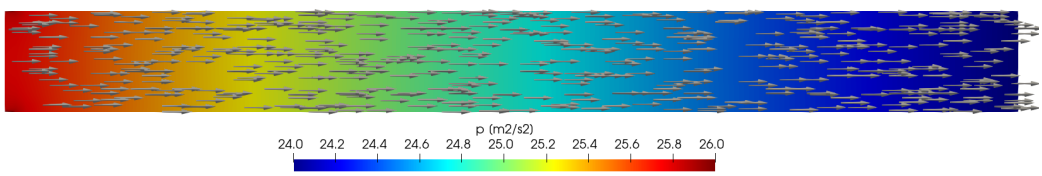
```
libs
(
    "libheadLossPressure.so"
);
```

Further, the test case is evaluated under steady-state and incompressible conditions, thus the `simpleFoam` solver utilised. The convection terms of the momentum equations are discretised using the `linearUpwind` scheme, and the convection terms of the turbulence equations are are discretised with the `limitedLinear` scheme. The computations are set to continue until a convergence of 1×10^{-4} is reached for all variables.

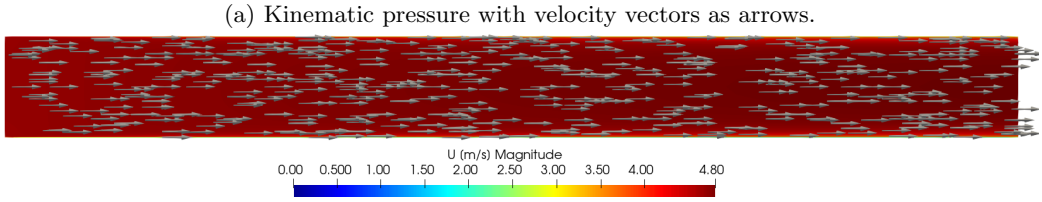
Chapter 5

Results

The simulation converged after roughly 160 iterations in both serial and parallel mode. Results of the computed kinematic pressure and velocity magnitude are found in Figure 5.1. The pressure decreases gradual through the domain, while the velocity is rather constant.



(a) Kinematic pressure with velocity vectors as arrows.



(b) Velocity magnitude with velocity vectors as arrows.

Figure 5.1: Numerical results on a plane cutting the domain.

In the boundary condition, it is specified that the inlet pressure should depend on on the elevation height minus the pressure drop from the losses and the dynamic pressure. It is clear that the pressure loss plays a key role in determining the pressure at the patch, since the inlet pressure is rather far from $gH_{\text{inlet}} = 49\ \text{m}^2/\text{s}^2$. The same goes for the outlet, if the outlet pressure would have been determined by only the hydrostatic pressure, the patch pressure would be $gH_{\text{outlet}} = 9.8\ \text{m}^2/\text{s}^2$. It is clear from Figure 5.1 that the outlet pressure is higher. Remember that for outgoing flow the dynamic pressure is not included, this to ensure that continuity is maintained for the CFD solver.

It is possible to compare the pressure drop predicted by the `headLossPressure` boundary condition and the analytical solution from Equation 1.4, together with loss-factor calculations from Equations 1.5, and 1.6. The comparison is made possible by taking the volumetric flux at the inlet and using this to calculate the velocity for the analytical solution. The two cases are compared, with values are according to Table 4.1, as

$$p^*_{\text{inlet}} = \underbrace{p_{atm}}_{=0} - \frac{U^2}{2} - \Delta p^*_{\text{friction, inlet}} - \Delta p^*_{\text{minor, inlet}} + gH_{\text{inlet}} \tag{5.1}$$

$$p^*_{\text{outlet}} = \underbrace{p_{atm}}_{=0} - \underbrace{\frac{U^2}{2}}_{=0,\ \text{outlet}} + \Delta p^*_{\text{friction, outlet}} + \Delta p^*_{\text{minor, outlet}} + gH_{\text{outlet}} \tag{5.2}$$

$$\Delta p^* = p^*_{\text{inlet}} - p^*_{\text{outlet}}. \tag{5.3}$$

The pressure difference predicted by OpenFOAM is denoted by "sim", and the analytical with "ana". Numerical values of the quantities are

$$\Delta p^*_{\text{sim}} = 1.801\ \text{m}^2/\text{s}^2 \tag{5.4}$$

$$\Delta p^*_{\text{ana}} = 1.799\ \text{m}^2/\text{s}^2 \tag{5.5}$$

$$\delta_{\%} = \left(1 - \frac{\Delta p^*_{\text{ana}}}{\Delta p^*_{\text{sim}}}\right) = 0.1\ \%. \tag{5.6}$$

The percentual difference between the analytical and numerical results is shown in Equation 5.6. It is found that the predictions made by OpenFOAM differs with only 0.1 % from the analytical calculations, it is thus safe to state the implementations made in OpenFOAM are correctly made according to the theory discussed section 1.1.2.

One of the main goals with the `headLossPressure` boundary condition is to evaluate if the correct flow rate can be estimated. Although equation 5.6 show that the loss-factors are correctly implemented. The question still remains if the `headLossPressure` can be used to evaluate the flow rate. The analytical equations are set-up in an iterative manner to evaluate the flow rate as

$$Q_{\text{guess}} \Longrightarrow \begin{cases} U & = \frac{Q_{\text{guess}}}{A} \\ \text{Re}_d & = \frac{Ud}{\nu} \\ \frac{1}{f^{1/2}} & = -2.0 \log_{10}\left(\frac{\epsilon/d}{3.7} + \frac{2.51}{\text{Re}_d f^{1/2}}\right) \\ \Delta p^*_{\text{friction}} & = \frac{fU^2}{2d}(L_{\text{up-stream}} + L_{\text{sim-domain}} + L_{\text{down-stream}}) \\ p^*_{\text{minor}} & = \frac{U^2}{2}\sum_{i=1}^{N} k_i \\ g\Delta H & = \frac{U^2}{2} + \Delta p^*_{\text{friction}} + \Delta p^*_{\text{minor}} \\ \delta_H & = |g\Delta H - g(H_{\text{up-stream}} - H_{\text{down-stream}})|. \end{cases} \tag{5.7}$$

The initial guess of the flow rate, $Q_{\text{guess}} = 0.001\ \text{kg/m}^3$. The flow rate is linearly increased until the term $\delta_H \leq 0.001$. The flow rate estimated by the iteration procedure presented in Equations 5.7 and the CFD simulations are almost identical

$$Q_{\text{sim}} = 0.0351\ \text{m}^3/\text{s},$$
$$Q_{\text{ana}} = 0.0352\ \text{m}^3/\text{s}.$$

The difference is only 0.3 % between the flow rate estimated analytically and numerically.

Chapter 6

Conclusions

In the present work a new pressure `headLossPressure` boundary condition has been developed. The new developments include an incompressible boundary condition that can take into account pressure losses before and after the computational domain and correcting the pressure on the patch caused by friction- and minor losses. The `headLossPressure` can take any number of friction and minor losses so that a whole system can be evaluated. The boundary condition also enables the user to set both elevation height and an atmospheric pressure of an imaginary up- or downstream reservoir surface.

The results presented in Chapter 5 indicate that the boundary condition works as intended. In reference to the specified atmospheric pressure, `patm`, and the reservoir surface height, `H`, a higher pressure is prescribed at a patch with outgoing flow, and a lower pressure for incoming flow. The results showed that the pressure drop predicted by OpenFOAM and the analytical expressions only differ by 0.1 % when using the flow rate estimated by the numerical simulation. The flow rate was also compared analytically and numerically and it differs only by 0.3 %. The boundary condition `headLossPressure` clearly works according the theory.

The analytical expressions from which the boundary condition is derived assumes steady state conditions. It is however the authors intention to at a later stage evaluate the `headLossPressure` under unsteady, and even transient conditions. If the boundary condition can produce accurate results for unsteady cases, this could lead to a much better prediction of the dynamics of an entire system. After all, CFD is in most cases only used for a small part a full system.

Bibliography

[1] F. M. White, *Fluid mechanics*. McGraw-Hill series in mechanical engineering, McGraw-Hill, 2011.

[2] Y. A. Çengel, R. H. Turner, J. M. Cimbala, and M. Kanoğlu, *Fundamentals of thermal-fluid sciences*. McGraw-Hill, 2012.

[3] H. K. Versteeg and W. Malalasekera, *An introduction to computational fluid dynamics : the finite volume method*. Pearson Prentice Hall, 2007.

[4] "OpenFOAM: User Guide v2006." https://www.openfoam.com/documentation/guides/latest/doc/. Accessed: 2020-12-08.

Study questions

1. What is the purpose of modeling head losses?
2. How are head losses calculate due to friction and one time occurrences (minor) in the flow path?
3. What is the purpose of using pressure to drive the flow?
4. What is important when using pressure boundary conditions to drive the flow in OpenFOAM?
5. How can you make the `totalPressure` boundary condition to take any number of inputs?
6. Name two suitable velocity boundary conditions if the `headLossPressure` is used for pressure at the inlet.
7. Is the `headLossPressure` for pressure and `zeroGradient` for velocity a valid combination at outlets?

Appendix A

Code development

A.1 Cleaned header and definitions file

Content of the cleaned headLossPressureFvPatchScalarField.C

```

#ifndef headLossPressureFvPatchScalarField_H
#define headLossPressureFvPatchScalarField_H

#include "fixedValueFvPatchFields.H"

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

namespace Foam
{

/*---------------------------------------------------------------------------*\
              Class headLossPressureFvPatchScalarField Declaration
\*---------------------------------------------------------------------------*/

class headLossPressureFvPatchScalarField
:
    public fixedValueFvPatchScalarField
{
    // Private data

        //- Name of the velocity field
        word UName_;

        //- Name of the flux transporting the field
        word phiName_;


        //- Total pressure
        scalarField patm_;


public:

    //- Runtime type information
    TypeName("headLossPressure");


    // Constructors

        //- Construct from patch and internal field
        headLossPressureFvPatchScalarField
        (
            const fvPatch&,
```

```
            const DimensionedField<scalar, volMesh>&
        );

        //- Construct from patch, internal field and dictionary
        headLossPressureFvPatchScalarField
        (
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
            const dictionary&
        );

        //- Construct by mapping given headLossPressureFvPatchScalarField
        //  onto a new patch
        headLossPressureFvPatchScalarField
        (
            const headLossPressureFvPatchScalarField&,
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
            const fvPatchFieldMapper&
        );

        //- Construct as copy
        headLossPressureFvPatchScalarField
        (
            const headLossPressureFvPatchScalarField&
        );

        //- Construct and return a clone
        virtual tmp<fvPatchScalarField> clone() const
        {
            return tmp<fvPatchScalarField>
            (
                new headLossPressureFvPatchScalarField(*this)
            );
        }

        //- Construct as copy setting internal field reference
        headLossPressureFvPatchScalarField
        (
            const headLossPressureFvPatchScalarField&,
            const DimensionedField<scalar, volMesh>&
        );

        //- Construct and return a clone setting internal field reference
        virtual tmp<fvPatchScalarField> clone
        (
            const DimensionedField<scalar, volMesh>& iF
        ) const
        {
            return tmp<fvPatchScalarField>
            (
                new headLossPressureFvPatchScalarField(*this, iF)
            );
        }


    // Member functions

        // Access

            //- Return the name of the velocity field
            const word& UName() const
            {
                return UName_;
            }

            //- Return reference to the name of the velocity field
            //  to allow adjustment
```

```
            word& UName()
            {
                return UName_;
            }

            //- Return the name of the flux field
            const word& phiName() const
            {
                return phiName_;
            }

            //- Return reference to the name of the flux field
            //  to allow adjustment
            word& phiName()
            {
                return phiName_;
            }

            //- Return the total pressure
            const scalarField& patm() const
            {
                return patm_;
            }

            //- Return reference to the total pressure to allow adjustment
            scalarField& patm()
            {
                return patm_;
            }


        // Mapping functions

            //- Map (and resize as needed) from self given a mapping object
            virtual void autoMap
            (
                const fvPatchFieldMapper&
            );

            //- Reverse map the given fvPatchField onto this fvPatchField
            virtual void rmap
            (
                const fvPatchScalarField&,
                const labelList&
            );


        // Evaluation functions

            //- Inherit updateCoeffs from fixedValueFvPatchScalarField
            using fixedValueFvPatchScalarField::updateCoeffs;

            //- Update the coefficients associated with the patch field
            //  using the given patch total pressure and velocity fields
            virtual void updateCoeffs
            (
                const scalarField& patmp,
                const vectorField& Up
            );

            //- Update the coefficients associated with the patch field
            virtual void updateCoeffs();


        //- Write
        virtual void write(Ostream&) const;
};

```

```

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

} // End namespace Foam

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

#endif
```

Content of the cleaned headLossPressureFvPatchScalarField.C

```
#include "headLossPressureFvPatchScalarField.H"
#include "addToRunTimeSelectionTable.H"
#include "fvPatchFieldMapper.H"
#include "volFields.H"
#include "surfaceFields.H"


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

Foam::headLossPressureFvPatchScalarField::headLossPressureFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF
)
:
    fixedValueFvPatchScalarField(p, iF),
    UName_("U"),
    phiName_("phi"),
    patm_(p.size(), Zero)
{}


Foam::headLossPressureFvPatchScalarField::headLossPressureFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchScalarField(p, iF, dict, false),
    UName_(dict.lookupOrDefault<word>("U", "U")),
    phiName_(dict.lookupOrDefault<word>("phi", "phi")),
    patm_("patm", dict, p.size())
{
    if (dict.found("value"))
    {
        fvPatchField<scalar>::operator=
        (
            scalarField("value", dict, p.size())
        );
    }
    else
    {
        fvPatchField<scalar>::operator=(patm_);
    }
}


Foam::headLossPressureFvPatchScalarField::headLossPressureFvPatchScalarField
(
    const headLossPressureFvPatchScalarField& ptf,
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const fvPatchFieldMapper& mapper
)
:
    fixedValueFvPatchScalarField(ptf, p, iF, mapper),
```

```
    UName_(ptf.UName_),
    phiName_(ptf.phiName_),
    patm_(ptf.patm_, mapper)
{}


Foam::headLossPressureFvPatchScalarField::headLossPressureFvPatchScalarField
(
    const headLossPressureFvPatchScalarField& tppsf
)
:
    fixedValueFvPatchScalarField(tppsf),
    UName_(tppsf.UName_),
    phiName_(tppsf.phiName_),
    patm_(tppsf.patm_)
{}


Foam::headLossPressureFvPatchScalarField::headLossPressureFvPatchScalarField
(
    const headLossPressureFvPatchScalarField& tppsf,
    const DimensionedField<scalar, volMesh>& iF
)
:
    fixedValueFvPatchScalarField(tppsf, iF),
    UName_(tppsf.UName_),
    phiName_(tppsf.phiName_),
    patm_(tppsf.patm_)
{}


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

void Foam::headLossPressureFvPatchScalarField::autoMap
(
    const fvPatchFieldMapper& m
)
{
    fixedValueFvPatchScalarField::autoMap(m);
    patm_.autoMap(m);
}


void Foam::headLossPressureFvPatchScalarField::rmap
(
    const fvPatchScalarField& ptf,
    const labelList& addr
)
{
    fixedValueFvPatchScalarField::rmap(ptf, addr);

    const headLossPressureFvPatchScalarField& tiptf =
        refCast<const headLossPressureFvPatchScalarField>(ptf);

    patm_.rmap(tiptf.patm_, addr);
}


void Foam::headLossPressureFvPatchScalarField::updateCoeffs
(
    const scalarField& patmp,
    const vectorField& Up
)
{
    if (updated())
    {
        return;
    }
```

```

    const fvsPatchField<scalar>& phip =
        patch().lookupPatchField<surfaceScalarField, scalar>(phiName_);

    if (internalField().dimensions() == dimPressure/dimDensity)
    {
        // Incompressible flow
        operator==(patmp - 0.5*(1.0 - pos0(phip))*magSqr(Up));
    }
    else
    {
        FatalErrorInFunction
            << " Incorrect pressure dimensions " << internalField().dimensions()
            << nl
            << "    Should be " << dimPressure
            << " for compressible/variable density flow" << nl
            << "    or " << dimPressure/dimDensity
            << " for incompressible flow," << nl
            << "    on patch " << this->patch().name()
            << " of field " << this->internalField().name()
            << " in file " << this->internalField().objectPath()
            << exit(FatalError);
    }

    fixedValueFvPatchScalarField::updateCoeffs();
}


void Foam::headLossPressureFvPatchScalarField::updateCoeffs()
{
    updateCoeffs
    (
        patm(),
        patch().lookupPatchField<volVectorField, vector>(UName())
    );
}


void Foam::headLossPressureFvPatchScalarField::write(Ostream& os) const
{
    fvPatchScalarField::write(os);
    os.writeEntryIfDifferent<word>("U", "U", UName_);
    os.writeEntryIfDifferent<word>("phi", "phi", phiName_);
    patm_.writeEntry("patm", os);
    writeEntry("value", os);
}


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

namespace Foam
{
    makePatchTypeField
    (
        fvPatchScalarField,
        headLossPressureFvPatchScalarField
    );
}

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

Index