5.2.2 Basic boundary conditions

Boundary conditions are specified in field files, *e.g.* *p*, *U*, in time directories as described in section 4.2.8. An example pressure field file, *p*, is shown below for the rhoPimpleFoam case corresponding to the *boundary* file presented in section 5.2.1.

```
17  dimensions      [1 -1 -2 0 0 0 0];
18
19  internalField   uniform 1;
20
21  boundaryField
22  {
23      inlet
24      {
25          type            fixedValue;
26          value           uniform 1;
27      }
28
29      outlet
30      {
31          type            waveTransmissive;
32          field           p;
33          psi             thermo:psi;
34          gamma           1.4;
35          fieldInf        1;
36          lInf            3;
37          value           uniform 1;
38      }
39
40      bottom
41      {
42          type            symmetryPlane;
43      }
44
45      top
46      {
47          type            symmetryPlane;
48      }
49
50      obstacle
51      {
52          type            zeroGradient;
53      }
54
55      defaultFaces
56      {
57          type            empty;
58      }
59  }
60
61  // ************************************************************************* //
```

Every patch includes a `type` entry that specifies the type of boundary condition. They range from a basic `fixedValue` condition applied to the `inlet`, to a complex `waveTransmissive` condition applied to the `outlet`. The patches with non-generic types, *e.g.* symmetryPlane, defined in *boundary*, use consistent boundary condition types in the *p* file.

The main basic boundary condition types available in OpenFOAM are summarised below using a patch field named $\mathbf{Q}$. This is not a complete list; for all types see *$FOAM_SRC/finiteVolume/fields/fvPatchFields/basic*.

- fixedValue: value of $\mathbf{Q}$ is specified by `value`.
- fixedGradient: normal gradient of $\mathbf{Q}$ ($\partial\mathbf{Q}/\partial n$) is specified by `gradient`.
- zeroGradient: normal gradient of $\mathbf{Q}$ is zero.
- calculated: patch field $\mathbf{Q}$ calculated from other patch fields.

- mixed: mixed fixedValue/ fixedGradient condition depending on `valueFraction` ($0 \leq$ `valueFraction` ≤ 1) where

$$\texttt{valueFraction} = \begin{cases} 1 & \text{corresponds to } \mathbf{Q} = \texttt{refValue}, \\ 0 & \text{corresponds to } \partial\mathbf{Q}/\partial n = \texttt{refGradient}. \end{cases} \tag{5.1}$$

- directionMixed: mixed condition with tensorial `valueFraction`, to allow different conditions in normal and tangential directions of a vector patch field, *e.g.* fixedValue in the tangential direction, zeroGradient in the normal direction.

5.2.3 Derived types

There are numerous more complex boundary conditions derived from the basic conditions. For example, many complex conditions are derived from fixedValue, where the value is calculated by a function of other patch fields, time, geometric information, *etc.* Some other conditions derived from mixed/directionMixed switch between fixedValue and fixedGradient (usually zeroGradient).

There are a number of ways the user can list the available boundary conditions in OpenFOAM, with the -listScalarBCs and -listVectorBCs utility being the quickest. The boundary conditions for scalar fields and vector fields, respectively, can be listed for a given solver, *e.g.* simpleFoam, as follows.

```
simpleFoam -listScalarBCs -listVectorBCs
```

These produce long lists which the user can scan through. If the user wants more information of a particular condition, they can run the foamInfo script which provides a description of the boundary condition and lists example cases where it is used. For example, for the totalPressure boundary condition, run the following.

```
foamInfo totalPressure
```

In the following sections we will highlight some particular important, commonly used boundary conditions.

5.2.3.1 The inlet/outlet condition

The inletOutlet condition is one derived from mixed, which switches between zeroGradient when the fluid flows out of the domain at a patch face, and fixedValue, when the fluid is flowing into the domain. For inflow, the inlet value is specified by an `inletValue` entry. A good example of its use can be seen in the damBreak tutorial, where it is applied to the phase fraction on the upper `atmosphere` boundary. Where there is outflow, the condition is well posed, where there is inflow, the phase fraction is fixed with a value of 0, corresponding to 100% air.

```
17  dimensions      [0 0 0 0 0 0 0];
18
19  internalField   uniform 0;
20
21  boundaryField
22  {
23      leftWall
```

```
    {
        type            zeroGradient;
    }

    rightWall
    {
        type            zeroGradient;
    }

    lowerWall
    {
        type            zeroGradient;
    }

    atmosphere
    {
        type            inletOutlet;
        inletValue      uniform 0;
        value           uniform 0;
    }

    defaultFaces
    {
        type            empty;
    }
}

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

5.2.3.2 Entrainment boundary conditions

The combination of the totalPressure condition on pressure and pressureInletOutletVelocity on velocity is extremely common for patches where some inflow occurs and the inlet flow velocity is not known. That includes the `atmosphere` boundary in the damBreak tutorial, inlet conditions where only pressure is known, outlets where flow reversal may occur, and where flow in entrained, *e.g.* on boundaries surrounding a jet through a nozzle.

The totalPressure condition specifies

$$p = \begin{cases} p_0 & \text{for outflow} \\ p_0 - \frac{1}{2}|\mathbf{U}^2| & \text{for inflow (incompressible, subsonic)} \end{cases} \tag{5.2}$$

where the user specifies p_0 through the p0 keyword. The pressureInletOutletVelocity condition specifies zeroGradient at all times, except on the tangential component which is set to fixedValue for inflow, with the tangentialVelocity defaulting to 0.

The idea behind this combination is that the condition is a standard combination in the case of outflow, but for inflow the normal velocity is allowed to find its own value. Under these circumstances, a rapid rise in velocity presents a risk of instability, but the rise is moderated by the reduction of inlet pressure, and hence driving pressure gradient, as the inflow velocity increases.

The specification of these boundary conditions in the *U* and *p_rgh* files, in the damBreak case, are shown below.

```

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

internalField   uniform (0 0 0);

boundaryField
{
    leftWall
    {
        type            noSlip;
    }
    rightWall
```

```
29      {
30          type            noSlip;
31      }
32      lowerWall
33      {
34          type            noSlip;
35      }
36      atmosphere
37      {
38          type            pressureInletOutletVelocity;
39          value           uniform (0 0 0);
40      }
41      defaultFaces
42      {
43          type            empty;
44      }
45  }
46
47
48  // ************************************************************************* //
```

```
17  dimensions      [1 -1 -2 0 0 0 0];
18
19  internalField   uniform 0;
20
21  boundaryField
22  {
23      leftWall
24      {
25          type            fixedFluxPressure;
26          value           uniform 0;
27      }
28
29      rightWall
30      {
31          type            fixedFluxPressure;
32          value           uniform 0;
33      }
34
35      lowerWall
36      {
37          type            fixedFluxPressure;
38          value           uniform 0;
39      }
40
41      atmosphere
42      {
43          type            totalPressure;
44          p0              uniform 0;
45      }
46
47      defaultFaces
48      {
49          type            empty;
50      }
51  }
52
53  // ************************************************************************* //
```

5.2.3.3 Fixed flux pressure

In the above example, it can be seen that all the wall boundaries use a boundary condition named fixedFluxPressure. This boundary condition is used for pressure in situations where zeroGradient is generally used, but where body forces such as gravity and surface tension are present in the solution equations. The condition adjusts the gradient accordingly.

5.2.3.4 Time-varying boundary conditions

There are several boundary conditions for which some input parameters are specified by a function of time (using Function1 functionality) class. They can be searched by the following command.

```
find $FOAM_SRC/finiteVolume/fields/fvPatchFields -type f | \
    xargs grep -l Function1 | xargs dirname | sort -u
```

They include conditions such as uniformFixedValue, which is a fixedValue condition which applies a single value which is a function of time through a `uniformValue` keyword entry.

The Function1 is specified by a keyword following the `uniformValue` entry, followed by parameters that relate to the particular function. The Function1 options are list below.

- `constant`: constant value.
- `table`: inline list of `(time value)` pairs; interpolates values linearly between times.
- `tableFile`: as above, but with data supplied in a separate file.
- `csvFile`: time-value data supplied in a file in CSV format.
- `square`: square-wave function.
- `sine`: sine function.
- `one` and `zero`: constant one and zero values.
- `polynomial`: polynomial function using a list `(coeff exponent)` pairs.
- `coded`: function specified by user coding.
- `scale`: scales a given `value` function by a scalar `scale` function; both entries can be themselves Function1; `scale` function is often a ramp function (below), with `value` controlling the ramp value.
- `linearRamp`, `quadraticRamp`, `halfCosineRamp`, `quarterCosineRamp` and `quarterSineRamp`: monotonic ramp functions which ramp from 0 to 1 over specified duration.
- `reverseRamp`: reverses the values of a ramp function, e.g. from 1 to 0.

Examples or a time-varying inlet for a scalar are shown below.

```
inlet
{
    type         uniformFixedValue;
    uniformValue constant 2;
}

inlet
{
    type         uniformFixedValue;
    uniformValue table ((0 0) (10 2));
}

inlet
{
    type         uniformFixedValue;
    uniformValue polynomial ((1 0) (2 2)); // = 1*t^0 + 2*t^2
}

inlet
{
    type         uniformFixedValue;
```

```
    uniformValue
    {
        type             tableFile;
        file             "dataTable.txt";
    }
}

inlet
{
    type         uniformFixedValue;
    uniformValue
    {
        type             csvFile;
        nHeaderLine      4;              // number of header lines
        refColumn        0;              // time column index
        componentColumns (1);            // data column index
        separator        ",";            // optional (defaults to ",")
        mergeSeparators  no;             // merge multiple separators
        file             "dataTable.csv";
   }
}

inlet
{
    type         uniformFixedValue;
    uniformValue
    {
        type             square;
        frequency        10;
        amplitude        1;
        scale            2;  // Scale factor for wave
        level            1;  // Offset
    }
}

inlet
{
    type         uniformFixedValue;
    uniformValue
    {
        type             sine;
        frequency        10;
        amplitude        1;
        scale            2;  // Scale factor for wave
        level            1;  // Offset
    }
}

input  // ramp from 0 -> 2, from t = 0 -> 0.4
{
    type         uniformFixedValue;
    uniformValue
    {
        type             scale;
        scale            linearRamp;
        start            0;
        duration         0.4;
        value            2;
    }
}

input  // ramp from 2 -> 0, from t = 0 -> 0.4
{
    type         uniformFixedValue;
    uniformValue
    {
        type             scale;
        scale            reverseRamp;
        ramp             linearRamp;
        start            0;
        duration         0.4;
```

```
        value           2;
    }
}

inlet
{
    type            uniformFixedValue;
    uniformValue    coded;
    name            pulse;
    codeInclude
    #{
        #include "mathematicalConstants.H"
    #};

    code
    #{
        return scalar
        (
            0.5*(1 - cos(constant::mathematical::twoPi*min(x/0.3, 1)))
        );
    #};
}
```

5.3 Mesh generation with the blockMesh utility

This section describes the mesh generation utility, blockMesh, supplied with OpenFOAM. The blockMesh utility creates parametric meshes with grading and curved edges.

The mesh is generated from a dictionary file named *blockMeshDict* located in the *system* (or *constant/polyMesh*) directory of a case. blockMesh reads this dictionary, generates the mesh and writes out the mesh data to *points* and *faces*, *cells* and *boundary* files in the same directory.

The principle behind blockMesh is to decompose the domain geometry into a set of 1 or more three dimensional, hexahedral blocks. Edges of the blocks can be straight lines, arcs or splines. The mesh is ostensibly specified as a number of cells in each direction of the block, sufficient information for blockMesh to generate the mesh data.

Each block of the geometry is defined by 8 vertices, one at each corner of a hexahedron. The vertices are written in a list so that each vertex can be accessed using its label, remembering that OpenFOAM always uses the C++ convention that the first element of the list has label '0'. An example block is shown in Figure 5.3 with each vertex numbered according to the list. The edge connecting vertices 1 and 5 is curved to remind the reader that curved edges can be specified in blockMesh.

It is possible to generate blocks with less than 8 vertices by collapsing one or more pairs of vertices on top of each other, as described in section 5.3.5.

Each block has a local coordinate system (x_1, x_2, x_3) that must be right-handed. A right-handed set of axes is defined such that to an observer looking down the Oz axis, with O nearest them, the arc from a point on the Ox axis to a point on the Oy axis is in a clockwise sense.

The local coordinate system is defined by the order in which the vertices are presented in the block definition according to:

- the axis origin is the first entry in the block definition, vertex 0 in our example;
- the x_1 direction is described by moving from vertex 0 to vertex 1;
- the x_2 direction is described by moving from vertex 1 to vertex 2;