Implementing boundary conditions using high level programming

- Hereafter we will work with high level programming, this is the hard part of programming in OpenFOAM®.
- High level programming requires some knowledge on C++ and OpenFOAM® API library.
- Before doing high level programming, we highly recommend you to try with **codeStream**, most of the time it will work.
- We will implement the parabolic profile, so you can compare this implementation with **codeStream** ad **codedFixedValue** BCs.
- When we program boundary conditions, we are actually building a new library that can be linked with any solver. To compile the library, we use the command `wmake` (distributed with OpenFOAM®).
- At this point, you can work in any directory. But we recommend you to work in your OpenFOAM® user directory, type in the terminal,

```
1. $> cd $WM_PROJECT_USER_DIR/run
```

Implementing boundary conditions using high level programming

- Let us create the basic structure to write the new boundary condition, type in the terminal,

```
1. $> foamNewBC -f -v myParabolicVelocity
2. $> cd myParabolicVelocity
```

- The utility `foamNewBC`, will create the directory structure and all the files needed to write your own boundary conditions.
- We are setting the structure for a fixed (the option `-f`) velocity (the option `-v`), boundary condition, and we name our boundary condition `ParabolicVelocity`.
- If you want to get more information on how to use `foamNewBC`, type in the terminal,

```
1. $> foamNewBC -help
```

Implementing boundary conditions using high level programming

Directory structure of the new boundary condition

```
./myParabolicVelocity
├── Make
│   ├── files
│   └── options
├── myParabolicVelocityFvPatchVectorField.C
└── myParabolicVelocityFvPatchVectorField.H
```

The directory contains the source code of the boundary condition.

- *myParabolicVelocityFvPatchVectorField.C*: is the actual source code of the application. This file contains the definition of the classes and functions.
- *myParabolicVelocityFvPatchVectorField.H*: header files required to compile the application. This file contains variables, functions and classes declarations.
- The **Make** directory contains compilation instructions.
  - *Make/files*: names all the source files (*.C*), it specifies the boundary condition library name and location of the output file.
  - *Make/options*: specifies directories to search for include files and libraries to link the solver against.

Implementing boundary conditions using high level programming

The header file (.H)

- Let us start to do some modifications. Open the header file using your favorite text editor (we use gedit).

```
99      //- Single valued scalar quantity, e.g. a coefficient
100     scalar scalarData_;
101
102     //- Single valued Type quantity, e.g. reference pressure pRefValue_
103     //  Other options include vector, tensor
104     vector data_;
105
106     //- Field of Types, typically defined across patch faces
107     //  e.g. total pressure p0_.  Other options include vectorField
108     vectorField fieldData_;
109
110     //- Type specified as a function of time for time-varying BCs
111     autoPtr<Function1<vector>> timeVsData_;
112
113     //- Word entry, e.g. pName_ for name of the pressure field on database
114     word wordData_;
115
116     //- Label, e.g. patch index, current time index
117     label labelData_;
118
119     //- Boolean for true/false, e.g. specify if flow rate is volumetric_
120     bool boolData_;
121
122
123     // Private Member Functions
124
125     //- Return current time
126     scalar t() const;
```

- In lines 99-126 different types of private data are declared.
- These are the variables we will use for the implementation of the new BC.
- In our implementation we need to use vectors and scalars, therefore we can keep the lines 100 and 104.
- We can delete lines 106-120, as we do not need those datatypes.
- Also, as we will use two vectors in our implementation, we can duplicate line 104.
- You can leave the rest of the file as it is.

Implementing boundary conditions using high level programming

The header file (.H)

- At this point, your header file should looks like this one,

```
99    //- Single valued scalar quantity, e.g. a coefficient
100   scalar scalarData_;
101
102   //- Single valued Type quantity, e.g. reference pressure pRefValue_
103   //  Other options include vector, tensor
104   vector data_;
105   vector data_;
```

- Change the name of **scalarData**_ to **maxValue**_ (line 100).
- Change the names of the two vectors **data**_ (lines 104-105). Name the first one **n**_ and the last one **y**_.

```
99    //- Single valued scalar quantity, e.g. a coefficient
100   scalar maxValue_;
101
102   //- Single valued Type quantity, e.g. reference pressure pRefValue_
103   //  Other options include vector, tensor
104   vector n_;
105   vector y_;
```

It is recommended to initialize them in the same order as you declare them in the header file

- You can now save and close the file.

Implementing boundary conditions using high level programming

The source file (.C)

- Let us start to modify the source file. Open the source file with your favorite editor.
- Lines 34-37 refers to a private function definition. This function allows us to access simulation time. Since in our implementation we do not need to use time, we can safely remove these lines.

```
34      Foam::scalar Foam::myParabolicVelocityFvPatchVectorField::t() const
35      {
36          return db().time().timeOutputValue();
37      }
```

- Let us compile the library to see what errors we get. Type in the terminal,

1. `$> wmake`

- You will get a lot of errors.
- Since we deleted the datatypes **fieldData**, **timeVsData**, **wordData**, **labelData** and **boolData** in the header file, we need to delete them as well in the C file. Otherwise the compiler complains.

Implementing boundary conditions using high level programming

The source file (.C)

- At this point, let us erase all the occurrences of the datatypes **fieldData**, **timeVsData**, **wordData**, **labelData**, and **boolData**.
- Locate line 38,

```
38   Foam::myParabolicVelocityFvPatchVectorField::
     ...
     ...
     ...
```

- Using this line as your reference location in the source code, follow these steps,
  - Erase the following lines in incremental order (be sure to erase only the lines that contain the words **fieldData**, **timeVsData**, **wordData**, **labelData** and **boolData**): 48-52, 64-68, 92-96, 105-109, 119-123, 177-180.
  - Erase the following lines (they contain the word **fieldData**), 131, 146, 159-161.
  - Replace all the occurrences of the word **scalarData** with **maxValue** (11 occurrences).

Implementing boundary conditions using high level programming

The source file (.C)

- Duplicate all the lines where the word **data** appears (6 lines), change the word **data** to **n** in the first line, and to **y** in the second line, erase the comma in the last line. For example,

Original statements

```
45   fixedValueFvPatchVectorField(p, iF),
46   maxValue_(0.0),
47   data_(Zero),
48   data_(Zero),
```

Modified statements

```
45   fixedValueFvPatchVectorField(p, iF),
46   maxValue_(0.0),
47   n_(Zero),
48   y_(Zero)
```

Remember to erase the comma

Implementing boundary conditions using high level programming

The source file (.C)

- We are almost done, we just defined all the datatypes. Now we need to implement the actual boundary condition.
- Starting in line 156, add the following statements,

```
150    void Foam::myParabolicVelocityFvPatchVectorField::updateCoeffs()
151    {
152        if (updated())
153        {
154            return;
155        }
156    boundBox bb(patch().patch().localPoints(), true);
157
158    vector ctr = 0.5*(bb.max() + bb.min());
159
160    const vectorField& c = patch().Cf();
161
162    scalarField coord = 2*((c - ctr) & y_)/((bb.max() - bb.min()) & y_);
163
```

The actual implementation of the BC is always done in this class

Add these lines

Find patch bounds (minimum and maximum points)

Coordinates of patch midpoint

Access patch face centers

Computes scalar field to be used for defining the parabolic profile

$$U_{max}\left(1.0 - \frac{(y-c)^2}{r^2}\right)$$

Implementing boundary conditions using high level programming

The source file (.C)

- Add the following statement in line 166,

```
164     fixedValueFvPatchVectorField::operator==
165     (
166         n_*maxValue_*(1.0 - sqr(coord))
167     );
168
```

The access function operator== is used to assign the values to the boundary patches

Our boundary condition

$$U_{max}\left(1.0 - \frac{(y-c)^2}{r^2}\right)$$

- At this point we have a valid library where we implemented a new BC.
- Try to compile it, we should not get any error (maybe one warning). Type in the terminal,

1. `$> wmake`

- If you are feeling lazy or you can not fix the compilation errors, you will find the source code in the directory,
    - **$PTOFC/101programming/src/myParabolicVelocity**

Implementing boundary conditions using high level programming

The source file (.C)

- Before moving forward, let us comment a little bit the source file.
- First at all, there are five classes constructors and each of them have a specific task.
- In our implementation we do not use all the classes, we only use the first two classes.
- The first class is related to the initialization of the variables.
- The second class is related to reading the input dictionaries.
- We will not comment on the other classes as it is out of the scope of this example (they deal with input tables, mapping, and things like that).
- The implementation of the boundary condition is always done using the **updateCoeffs()** member function.
- When we compile the source code, it will compile a library with the name specified in the file *Make/file*. In this case, the name of the library is **libmyParabolicVelocity**.
- The library will be located in the directory **$(FOAM_USER_LIBBIN)**, as specified in the file *Make/file*.

Implementing boundary conditions using high level programming

The source file (.C)

- The first class is related to the initialization of the variables declared in the header file.
- In line 47 we initialize **maxValue** with the value of zero. The vectors **n** and **y** are initialized as a zero vector by default or (0, 0, 0).
- It is not a good idea to initialize these vectors as zero vectors by default. Let us use as default initialization (1, 0, 0) for vector **n** and (0,1,0) for vector **y**.

```
38   Foam::myParabolicVelocityFvPatchVectorField::
39   myParabolicVelocityFvPatchVectorField
40   (
41       const fvPatch& p,
42       const DimensionedField<vector, volMesh>& iF
43   )
44   :
45       fixedValueFvPatchVectorField(p, iF),
46       maxValue_(0.0),
47       n_(Zero),
48       y_(Zero)
49   {
50   }
```

Change to n_(1,0,0)

Change to y_(0,1,0)

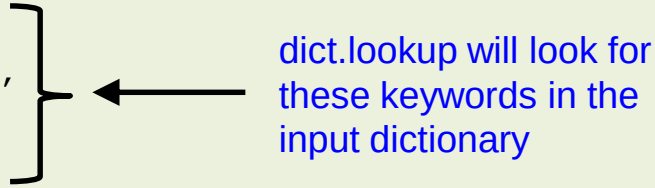
Implementing boundary conditions using high level programming

The source file (.C)

- The second class is used to read the input dictionary.
- Here we are reading the values defined by the user in the dictionary `U`.
- The function **lookup** will search the specific keyword in the input file.

```
53   Foam::myParabolicVelocityFvPatchVectorField::
54   myParabolicVelocityFvPatchVectorField
55   (
56       const fvPatch& p,
57       const DimensionedField<vector, volMesh>& iF,
58       const dictionary& dict
59   )
60   :
61       fixedValueFvPatchVectorField(p, iF),
62       maxValue_(readScalar(dict.lookup("maxValue"))),
63       n_(pTraits<vector>(dict.lookup("n"))),
64       y_(pTraits<vector>(dict.lookup("y")))
65   {
66
67
68       fixedValueFvPatchVectorField::evaluate();

77   }
```



Implementing boundary conditions using high level programming

The source file (.C)

- Since we do not want the vectors **n** and **y** to be zero vectors, we add the following sanity check from lines 67-75.
- These statements check if the given **n** and **y** vectors in the input dictionary is zero or not.
- If any of the vectors are zero it gives the fatal error and terminate the program.
- On the other hand, if everything is ok it will normalize **n** and **y** (since in our implementation they are direction vectors).

```
66
67        if (mag(n_) < SMALL || mag(y_) < SMALL)
68        {
69             FatalErrorIn("parabolicVelocityFvPatchVectorField(dict)")
70                 << "n or y given with zero size not correct"
71                 << abort(FatalError);
72        }
73
74        n_ /= mag(n_);      //This is equivalent to n_ = n_/mag(n_)
75        y_ /= mag(y_);      //This is equivalent to y_ = y_/(mag(y_)
76
77        fixedValueFvPatchVectorField::evaluate();
78
```

Add these statements

Implementing boundary conditions using high level programming

The source file (.C)

- At this point, we are ready to go.
- Save files and recompile. Type in the terminal,

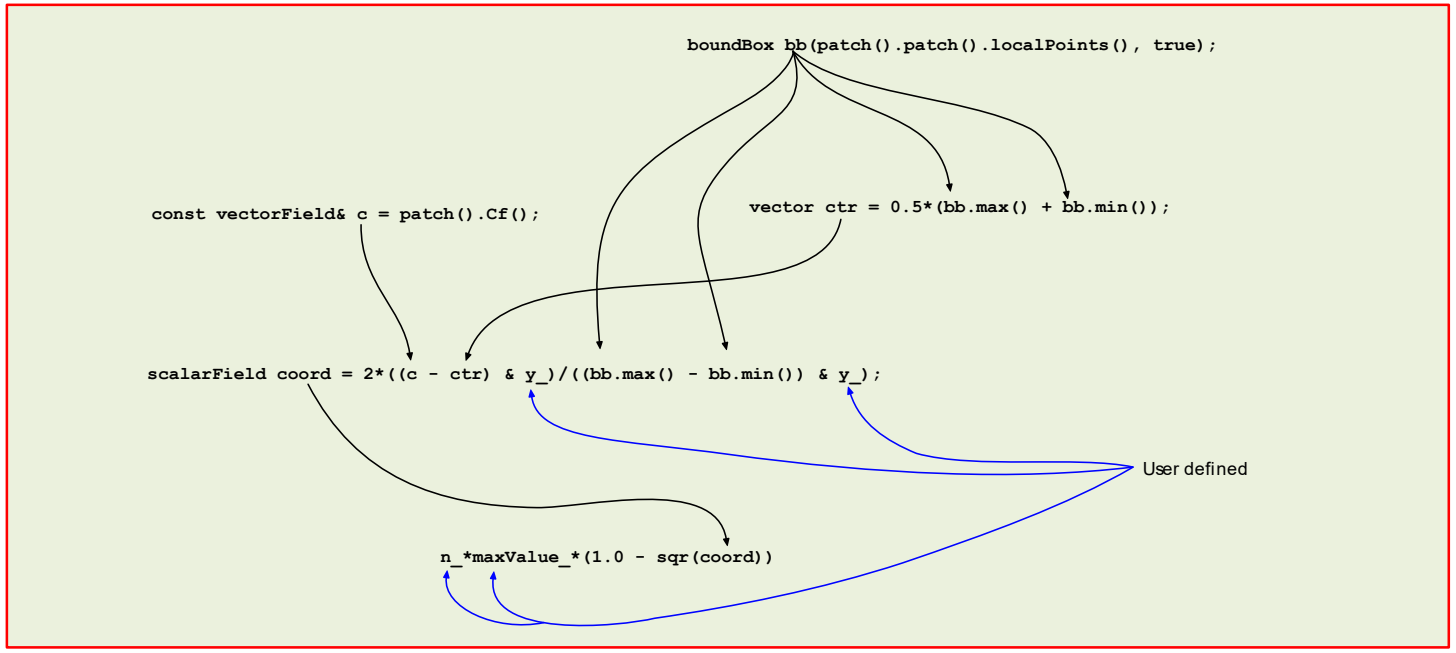
```
1. $> wmake
```

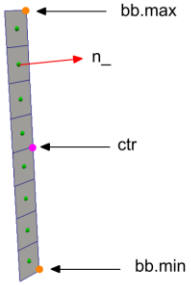
- We should not get any error (maybe one warning).
- At this point we have a valid library that can be linked with any solver.
- If you get compilation errors read the screen and try to sort it out, the compiler is always telling you what is the problem.
- If you are feeling lazy or you can not fix the compilation errors, you will find the source code in the directory,
    - **`$PTOFC/101programming/src/myParabolicVelocity`**

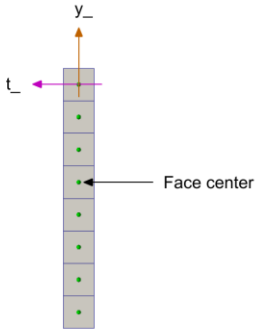
Implementing boundary conditions using high level programming

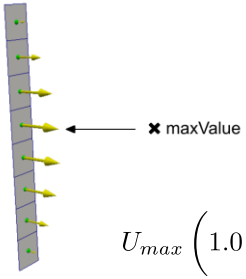
The source file (.C)

- Before using the new BC, let us take a look at the logic behind the implementation.









$$U_{max}\left(1.0 - \frac{(y-c)^2}{r^2}\right)$$

Implementing boundary conditions using high level programming

Running the case

- This case is ready to run, the input files are located in the directory **`$PTOFC/101programming/src/case_elbow2d`**
- Go to the case directory,

1. `$> cd $PTOFC/101programming/src/case_elbow2d`

- Open the file *0/U*, and look for the definition of the new BC **velocity-inlet-5**,

```
velocity-inlet-5
{
    type            myParabolicVelocity;

    maxValue 2.0;
    n          (1 0 0);
    y          (0 1 0);
}
```

Name of the boundary condition

User defined values
max value, n, y
If you set n or y to (0 0 0), the solver will abort execution

Implementing boundary conditions using high level programming

Running the case

- We also need to tell the application that we want to use the library we just compiled.
- To do so, we need to add the new library in the dictionary file *controlDict*,

```
15   // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
16
17   libs ("libmyParabolicVelocity.so");
18
19   application     icoFoam;
```

Name of the library
You can add as many libraries as you like

- The solver will dynamically link the library.
- At this point, we are ready to launch the simulation.

Implementing boundary conditions using high level programming

Running the case

- This case is ready to run, the input files are located in the directory **$PTOFC/101programming/src/case_elbow2d**
- To run the case, type in the terminal,

1. `$> foamCleanTutorials`
2. `$> fluentMeshToFoam ../../../meshes_and_geometries/fluent_elbow2d_1/ascii.msh`
3. `$> icoFoam | tee log`
4. `$> paraFoam`

- At this point, you can compare the three implementations (**codeStream**, **codedFixedValue** and high level programming).
- All of them will give the same outcome.

Implementing boundary conditions using high level programming

Adding some verbosity to the BC implementation

- Let us add some outputs to the BC.
- Starting at line 181 (after the function **updateCoeffs**), add the following lines,

```
179   fixedValueFvPatchVectorField::updateCoeffs();
180
181   Info << endl << "Face centers (c):" << endl;
182   Info << c << endl;
183   Info << endl << "Patch center (ctr):" << endl;
184   Info << ctr << endl;
185   Info << endl << "Patch (c - ctr):" << endl;
186   Info << c - ctr << endl;
187   Info << endl << "Patch max bound (bb.max):" << endl;
188   Info <<  bb.max() << endl;
189   Info << endl << "Patch min bound (bb.max):" << endl;
190   Info << bb.min() << endl;
191   Info << endl << "Patch coord ( 2*((c - ctr) & y_)/((bb.max() - bb.min()) & y_) ):" << endl;
192   Info << coord << endl;
193   Info << endl << "Patch ( 1.0 - sqr(coord)) :" << endl;
194   Info << n_*maxValue_*(1.0 - sqr(coord))<< endl;
195   Info << endl << "Loop for c, BC assigment << endl;
196   forAll(c, faceI)
197   {
198       Info << c[faceI] << "     " << n_*maxValue_*(1.0 - sqr(coord[faceI])) << endl;
199   }
200
201
```

- Recompile, rerun the simulation, look at the output, and do the math.

Implementing boundary conditions using high level programming

Do you take the challenge?

- Starting from this boundary condition, try to implement a paraboloid BC.
- If you are feeling lazy or at any point do you get lost, in the directory `$PTOFC/101programming/src/myParaboloid` you will find a working implementation of the paraboloid profile.
- Open the source code and try to understand what we did (pretty much similar to the previous case).
- In the directory **`$PTOFC/101programming/src/case_elbow3d`** you will find a case ready to use.