Writing New Boundary Conditions in OpenFOAM

UK FOAM/OpenFOAM User Day

Prof Gavin Tabor

18th April 2016

Aim of todays session...

Let's create a new boundary condition in OpenFOAM!!

We could do this in several ways – eg *swak4foam* – but I'm going to code it in C++. Why?

- To learn some more programming
- To learn how OpenFOAM uses *polymorphism*
- To learn something of the structure of classes in OpenFOAM

Background knowledge

I'm going to assume :

- Some familiarity with C++ (or equivalent; C, Python)
- Some knowledge of OpenFOAM as a user

OpenFOAM can best be treated as a special programming language for writing CFD codes. Programming in OpenFOAM should not be seen as scary or risky, but can be quite achievable.

More details – see *How to write OpenFOAM Applications and Get On in CFD* (1st UK OF Users Day 2014)

Classes in OpenFOAM

C++ is structured around *Classes* – data plus algorithms – can be thought of as new user-defined variable types. In OpenFOAM these are used to create new variable types such as :

- higher level data types – eg. `dimensionedScalar`
- FVM meshes (`fvMesh`)
- fields of scalars, vectors and 2nd rank tensors
- matrices and their solution

with which we can write solvers and utilities for doing CFD.

Some other things are also classes in OpenFOAM – turbulence models, viscosity models, *functionObjects*...

More about Classes

C++ also enables us to define *relationships* between classes – in particular, to derive one class from another. We use this to extend the functionality of a class, and to group classes together.

E.g: All turbulence models are related – take $\underline{U}$, return the Reynolds Stress. In OF each turbulence model is its own class, but all are *derived* from the same *base class*, thus linking them.

The base class is *virtual* – defines what functions have to be implemented in the derived classes (the class interface)

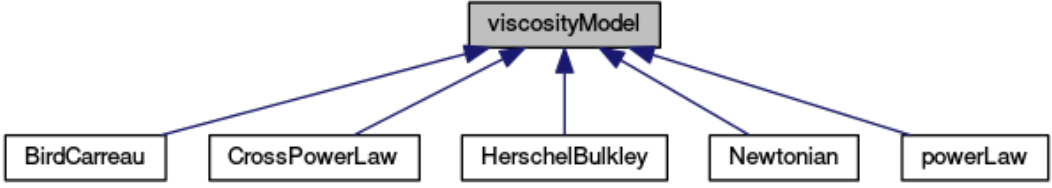
Polymorphism + runtime selection

This also means that turbulence models are interchangable and thus can be selected at runtime!

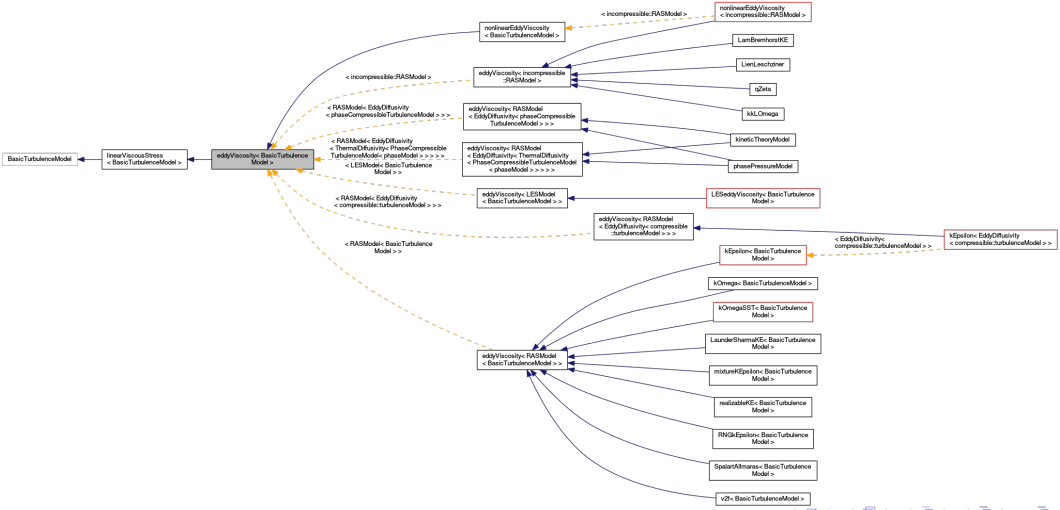
```
fvVectorMatrix UEqn
(
    fvm::ddt(U)
  + fvm::div(phi, U)
  + turbulence->divDevReff(U)
);
```

If we want to create a new turbulence model (viscous model, B.C etc), just derive it from the base class and it can plug in alongside any other model. OF even has run time 'hooks' in *controlDict* which mean the code can be added at runtime.

Examples – *viscosityModels*



Examples – *RASModels*



Parabolic Inlet

Laminar flow in a pipe gives a parabolic profile – lets implement a new b.c. for this :

$$\underline{u} = \underline{\hat{n}} \cdot u_m \left(1 - \frac{y^2}{R^2}\right)$$

Process :

1. Identify an existing B.C. to modify
2. Copy across to user working directory
3. Rename files/classes
4. Re-write class functions
5. Set up library compilation and compile
6. Link in runtime and test

Boundary Conditions

B.C. are in *src/finiteVolume/fields/fvPatchFields*; subdirectories *basic*, *constraint*, *derived*, *fvPatchField*

- *fvPatchField* is the (virtual) base class
- *basic* contains intermediate classes; in particular *fixedValue*, *fixedGradient*, *zeroGradient*, *mixed*
- *derived* contains the actual useable classes. *cylindricalInletVelocity* (derived from *fixedValue*) looks suitable!

Initial steps

So :

1. Copy the directory across to the user directory
2. Rename files *cylindricalInletVelocity* $\rightarrow$ *parabolicInletVelocity* (.C, .H files)
3. Change *cylindrical*$\rightarrow$*parabolic* throughout
4. Set up *Make* directory with *files* and *options*
5. Check that it compiles – *wmake libso*

files :

```
parabolicInletVelocityFvPatchVectorField.C

LIB = $(FOAM_USER_LIBBIN)/libnewBC
```

options :

```
EXE_INC = \
    -I$(LIB_SRC)/triSurface/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/finiteVolume/lnInclude

LIB_LIBS = \
    -lOpenFOAM \
    -ltriSurface \
    -lmeshTools \
    -lfiniteVolume
```

Changing the Code

C++ classes contain data, class functions. For *cylindricalInletVelocity* class functions are : various *constructors* (complicated), *updateCoeffs()*, *write(Ostream&)*.

```
class parabolicInletVelocityFvPatchVectorField
:
    public fixedValueFvPatchVectorField
{
    // Private data

        //- Axial velocity
        const scalar maxVelocity_;

        //- Central point
        const vector centre_;

        //- Axis
        const vector axis_;

        //- Radius
        const scalar R_;

public:

   //- Runtime type information
   TypeName("parabolicInletVelocity");
```

Private data : we need vectors for the centre of the inlet and an axis direction (already there) and scalars for the maximum velocity and the pipe radius.

Also need *TypeName* – will become the name of the B.C at run time

Constructor functions

This gets set to zero for a null constructor :

```
Foam::
parabolicInletVelocityFvPatchVectorField::
parabolicInletVelocityFvPatchVectorField
(
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF
)
:
    fixedValueFvPatchField<vector>(p, iF),
    maxVelocity_(0),
    centre_(pTraits<vector>::zero),
    axis_(pTraits<vector>::zero),
    R_(0)
{}
```

```
Foam::
parabolicInletVelocityFvPatchVectorField::
parabolicInletVelocityFvPatchVectorField
(
    const parabolicInletVelocityFvPatchVectorField& ptf,
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF,
    const fvPatchFieldMapper& mapper
)
:
    fixedValueFvPatchField<vector>(ptf, p, iF, mapper),
    maxVelocity_(ptf.maxVelocity_),
    centre_(ptf.centre_),
    axis_(ptf.axis_),
    R_(ptf.R_)
{}
```

. . . and copied across for a copy construct

Read in . . .

We want to read in the actual values from the velocity file – a *dictionary* :

```
Foam::
parabolicInletVelocityFvPatchVectorField::
parabolicInletVelocityFvPatchVectorField
(
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchField<vector>(p, iF, dict),
    maxVelocity_(readScalar(dict.lookup("maxVelocity"))),
    centre_(dict.lookup("centre")),
    axis_(dict.lookup("axis")),
    R_(readScalar(dict.lookup("radius")))
{}
```

. . . and write out

Write out through the *write(Ostream& os)* function :

```
void Foam::parabolicInletVelocityFvPatchVectorField::write(Ostream& os) const
{
    fvPatchField<vector>::write(os);
    os.writeKeyword("maxVelocity") << maxVelocity_ <<
        token::END_STATEMENT << nl;
    os.writeKeyword("centre") << centre_ << token::END_STATEMENT << nl;
    os.writeKeyword("axis") << axis_ << token::END_STATEMENT << nl;
    os.writeKeyword("radius") << R_ <<
        token::END_STATEMENT << nl;
    writeEntry("value", os);
}
```

updateCoeffs()

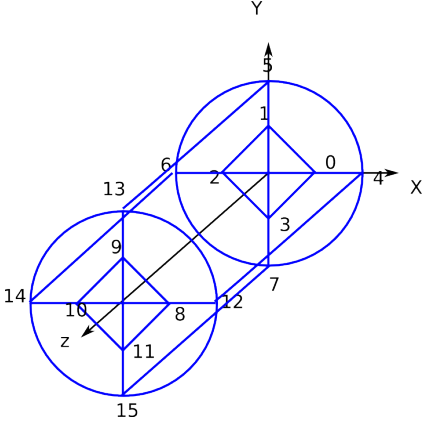
This is the actual code setting the boundary conditions

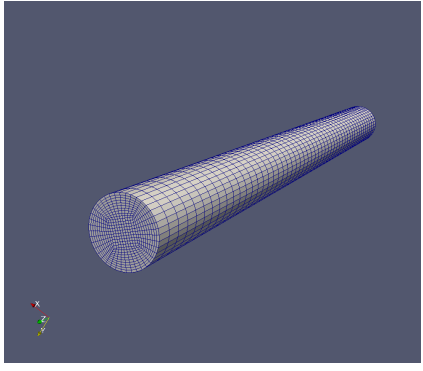
- Again; we can modify what is already there!
- OpenFOAM syntax makes this easier
- (Note call to updateCoeffs() in parent class)

```
void Foam::parabolicInletVelocityFvPatchVectorField::updateCoeffs()
{
    if (updated())
    {
        return;
    }

    vector hatAxis = axis_/mag(axis_);

    const scalarField r(mag(patch().Cf() - centre_));

    operator==(hatAxis*maxVelocity_*(1.0 - (r*r)/(R_*R_)));

    fixedValueFvPatchField<vector>::updateCoeffs();
}
```

Pipe flow case

Set up a test case – flow in a circular pipe. 5 block *blockMesh* demonstrating curved boundaries (circle arcs) and m4 script variables





B.C syntax

If we look at the "Read in ..." constructor, see that we need to specify

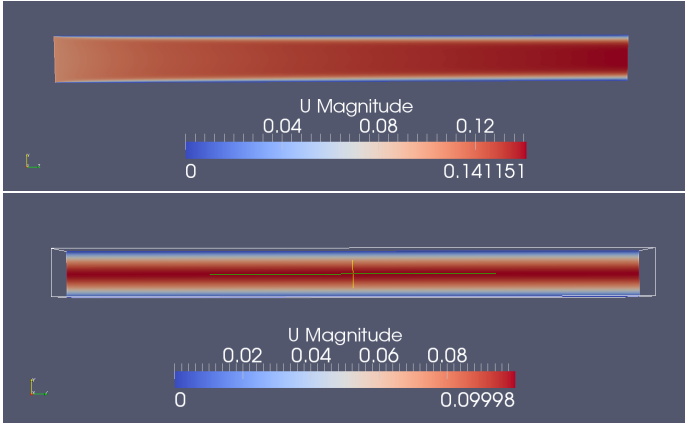
- maxVelocity
- *centre* (a vector)
- *axis* (another vector)
- *radius*

Also need a dummy "*value*"

```
inlet
{
    type            parabolicInletVelocity;
    axis            (0 0 1);
    centre          (0 0 0);
    maxVelocity     30;
    radius          10;
    value           (0 0 0);
}
```

Results

Replace inlet with new condition



Conclusions

- OpenFOAM uses classes to represent a range of different constructs (turb models, viscous models, b.c. etc)
- Class inheretance + virtual base class gives relationship between these
- Also allows run time selectivity
- (Fairly) easy to create a new class – derive from base class, compile to .so, hook in at runtime (*controlDict*)