AM119: One last tutorial

Prof. Trask

May 2, 2016

1 Generating a custom boundary condition

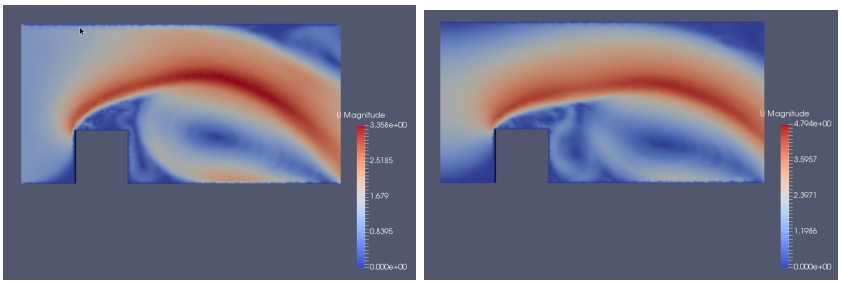
OpenFoam comes with a selection of general purpose boundary conditions that are good enough for most applications. In this tutorial we'll address the situation of what to do when you can't find something in the source code that covers your specific application. For some of you, this will be necessary for your final project. For others, this will be a useful exercise in understanding how to navigate the source code.

2 Objective

So far we've seen how to apply a uniform velocity inlet boundary condition. A more physical boundary condition for channel flow is to impose a parabolic velocity profile. For this exercise, we'll modify the flow past a square homework to handle the inlet velocity profile:

$$u(y) = y(y - H)$$

where H is the height of the channel. The figure below demonstrates the difference in the resulting velocity field for the two cases.



3 Idea

Rather than attempt to understand the code and write a boundary condition from scratch, we will find a boundary condition similar to what we need that's already in the code and alter it to suit our needs. When starting to work with a big library, this paradigm is a good first way to get oriented. Eventually of

course, you will want to read through the library, but this is an effective way to get a feel for the idioms of how the library is designed.

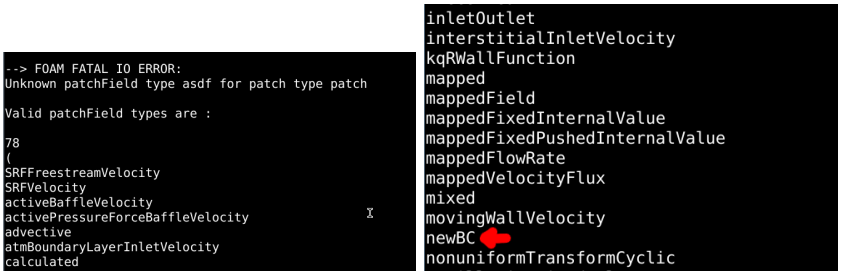
4 Copying a standard BC into a custom BC

To start with, lets take a look at the part of the source code related to boundary conditions. By entering *cd $FOAM_SRC/finiteVolume/fields/fvPatchFields/* you will see the directory containing everything related to boundary conditions. You can find some details about boundary conditions in the OpenFoam users guide (See section 5.1-5.2 of `http://foam.sourceforge.net/docs/Guides-a4/UserGuide.pdf`). Similar to the turbulence models from last week, you can find some base class boundary conditions in the *fvPatchFields/basic* subdirectory. These specify how to apply Dirichlet boundary conditions (*fixedValue*), Neumann boundary conditions (*fixedGradient*), Robin conditions (*mixed*), and some others. Since what we'll want to do is apply a Dirichlet boundary condition, take a minute to look through the source code for *fixedValue*. You'll find a bunch of constructors/destructors and a function called updateCoeffs - this is the function where the boundary condition is applied. We'll now go into the *fvPatchFields/derived*. This is a collection of classes that inherit from the basic boundary conditions. What we'd like to do is find something that inherits from the *fixedValue* type and does something similar to our objective. The *pressureInletVelocity* boundary condition meets those criteria - it's a boundary condition meant to be applied at a velocity inlet where the pressure is specified. So far we've only considered flows driven by a fixed velocity profile, but many applications require a given pressure drop across a channel, and this boundary conditions applies a uniform velocity consistent with the pressure drop.

First off, we're going to copy this into one of our solver directories so we can start altering it. Entering *pwd* will print the current directory of where this boundary condition is in the code. Next, go into our solvers directory, enter the subdirectory of your Navier-Stokes solver, and create a folder called *newBC*. Copy the header and C-file from the pressureInletVelocity directory into this case, and change their names to whatever you'd like to call your new condition (for example, I changed *pressureInletVelocityFvPatchVectorField.C* and *pressureInletVelocityFvPatchVectorField.H* to *newBCFvPatchVectorField.C* and *newBCFvPatchVectorField.H*). We'll now go into each of these files and replace everywhere that says *pressureInletVelocity* to *newBC*. (You'll want to find the find and replace command in whatever text editor you're using or this will get tedious). At this point, you should have a brand new boundary condition that behaves exactly like the pressureInletVelocity condition, but has a new name. This is pretty much the identical process to when we copied old solvers as a framework to hang our new solvers on.

To compile this, you'll need to add a line pointing to *BC/newBCFvPatchVectorField.C* in your *Make/files* file. After running *wclean* and *wmake*, you should have a fully functional boundary condition. If you rerun your previous homework, you should see your new boundary condition pop up in the list of available

options.



Now we have everything implemented and run-time selectable- the only thing left to do is make it apply the BC that we actually want. Go back into your solver directory and pop open *BC/newBCFvPatchVectorField.C*. We're going to want to alter the updateCoeffs function. Again, we don't really know OpenFoam well enough to go through and write this from scratch. Instead we can refer back to the source code and try to find another boundary condition that does something similar. What we want is something that applies some Dirichlet condition that is a function of some spatial variables. The *prghTotalPressure* BC does exactly this - it sets the pressure according to a hydrostatic pressure formula $p = p_0 - (\rho g z - \rho g z_{ref})$. We want to find a way to extract the face coordinates, so that we can use it in our new BC.

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

    const scalarField& rhop = patch().lookupPatchField<volScalarField, scalar>
    (
        rhoName_
    );

    const uniformDimensionedVectorField& g =
        db().lookupObject<uniformDimensionedVectorField>("g");

    const uniformDimensionedScalarField& hRef =
        db().lookupObject<uniformDimensionedScalarField>("hRef");

    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();
}
```

From their *updateCoeffs* function we can see that the function *patch().Cf()* returns the face coordinate. In what follows, you can see how I similarly copied this over to generate our custom BC.

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

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

    //Get patch face center location
    const vectorField& Xc = patch().Cf();

    //Location of bottom of patch and height of channel
    vector Xmin(0.0,0.0,0.0);
    scalar H(3.0);

    //Compute distance from bottom of patch
    vectorField dy = Xc - Xmin;

    //Get component of vectorField along patch
    scalarField dyz = dy & vector(0.0,1.0,0.0);

    //Compute velocity profile according to formula
    vectorField Uprofile = vector(1.0,0.0,0.0)*dyz*(H - dyz);

    //Compute boundary condition using formula for velocity profile
    operator==( Uprofile );

    fixedValueFvPatchVectorField::updateCoeffs();
}
```

In this case I hard-coded our case details into the source code - this is a bit of a hack but good enough for the purposes of our final projects. If we were developing this code so that it could be used by the general public, we would alter the constructor so that $Xmin$ and H could be read in from the initial condition directory. You can see the difficulty - if the geometry of the case changes, you will need to go back to the source code, change the parameters, and recompile the code.

At this point, after another *wclean* and *wmake* you should be done - with a fancy new boundary condition that does just what we set out to do.