AM119: HW4 and more OpenFOAM tutorials

Prof. Trask

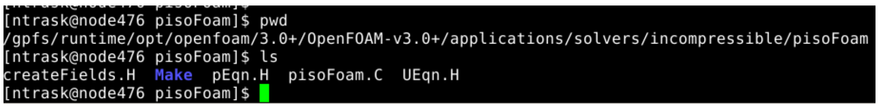
April 25, 2016

1 Final assignment: the $k - \epsilon$ model in OpenFoam

In the last two lectures we've learned a little bit about the ideas behind turbulence modeling. For the final assignment, we'll use the pisoFoam solver that comes with OpenFoam to do some turbulence modeling of flow past a sphere. In order to do this, we'll be taking a tour through some of the standard features of OpenFoam that we haven't touched on yet - now that we have a sense of what makes a solver work.

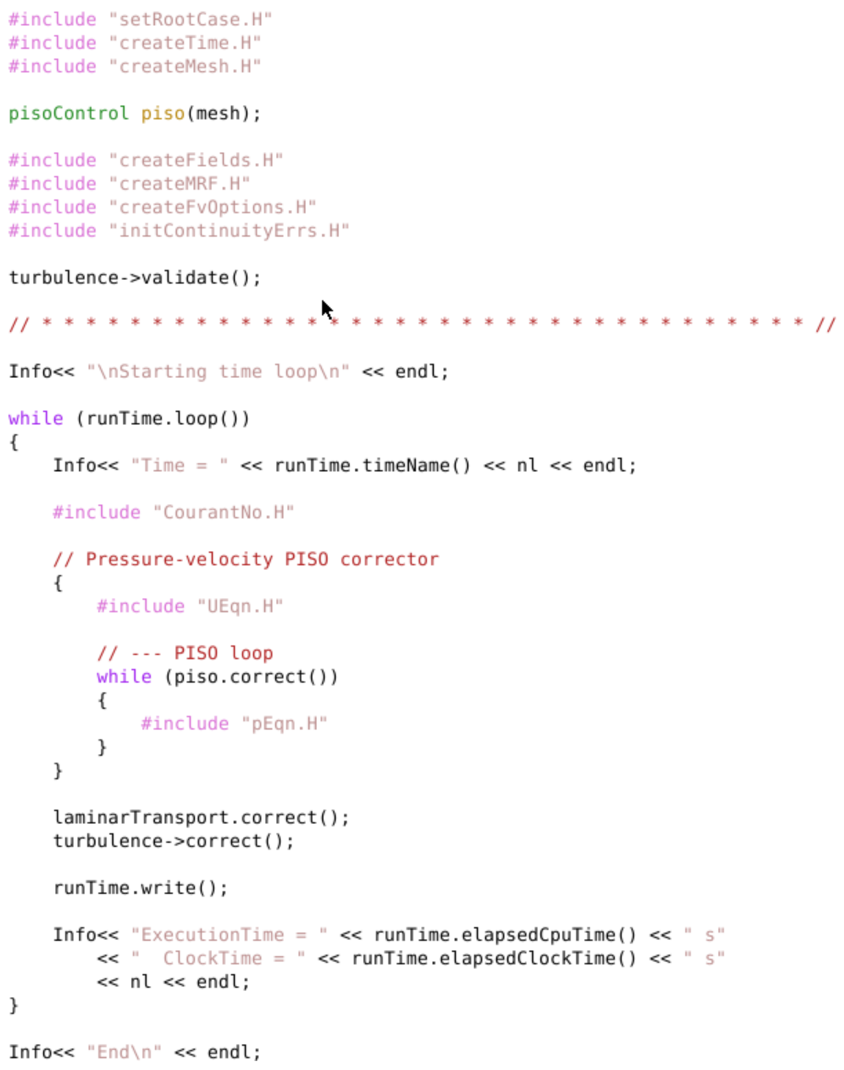
2 The pisoFoam solver

First let's take a look at the solver itself. This solver is essentially identical to our projection method, but has some extra bells and whistles to better handle instability for high Reynolds number flows. To take a look at it, type *sol* to get to the OpenFoam solvers directory. You can then go to *incompressible/pisoFoam* to find a solver directory very similar to the ones that we've been working on.



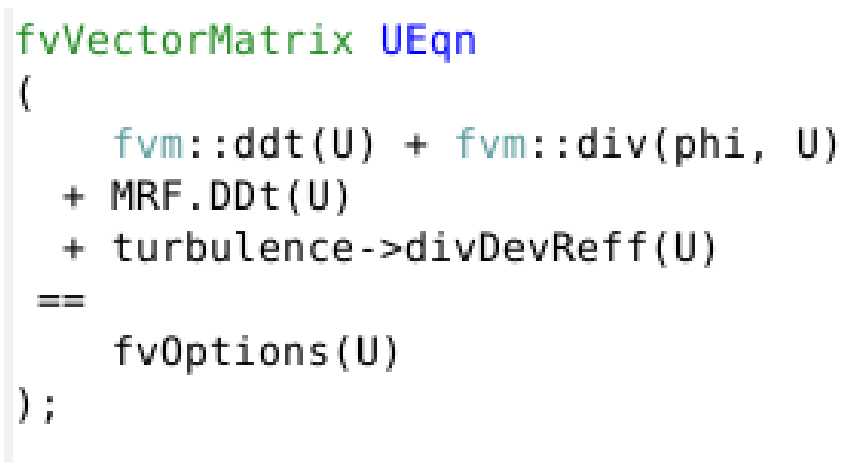
```
[ntrask@node476 pisoFoam]$ pwd
/gpfs/runtime/opt/openfoam/3.0+/OpenFOAM-v3.0+/applications/solvers/incompressible/pisoFoam
[ntrask@node476 pisoFoam]$ ls
createFields.H  Make  pEqn.H  pisoFoam.C  UEqn.H
[ntrask@node476 pisoFoam]$
```

We can see the main solver file *pisoFoam.C*, a *createField.H* file like the ones we usually use, and a couple new files. Let's open up the main file to get a feel for how the solver works



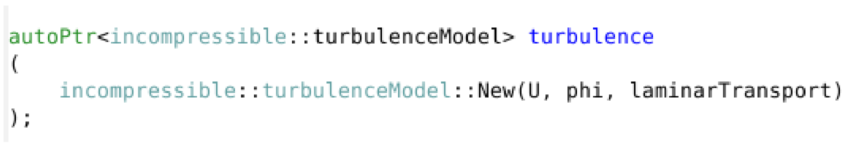
```
#include "setRootCase.H"
#include "createTime.H"
#include "createMesh.H"

pisoControl piso(mesh);

#include "createFields.H"
#include "createMRF.H"
#include "createFvOptions.H"
#include "initContinuityErrs.H"

turbulence->validate();

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

Info<< "\nStarting time loop\n" << endl;

while (runTime.loop())
{
    Info<< "Time = " << runTime.timeName() << nl << endl;

    #include "CourantNo.H"

    // Pressure-velocity PISO corrector
    {
        #include "UEqn.H"

        // --- PISO loop
        while (piso.correct())
        {
            #include "pEqn.H"
        }
    }

    laminarTransport.correct();
    turbulence->correct();

    runTime.write();

    Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
        << "  ClockTime = " << runTime.elapsedClockTime() << " s"
        << nl << endl;
}

Info<< "End\n" << endl;
```

You can see that the code follows a very similar flow to your nsFoam solver. First a provisional velocity is solved (to make the code easily readable, they've stuck this in *UEqn.H*). Then a pressure equation is solver to make this velocity divergence free. This solver applies the so-called *PISO algorithm* in order to more tightly couple the velocity and pressure together. It turns out that for high Reynolds number flows on 3D meshes it is much more challenging to get a stable simulation. If we take a look at *UEqn.H* we can look for how the introduction of a turbulence model alters their code.



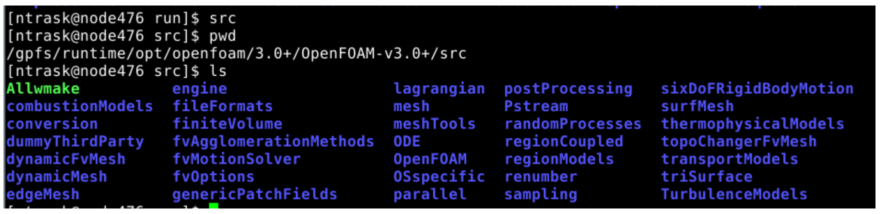
```
fvVectorMatrix UEqn
(
    fvm::ddt(U) + fvm::div(phi, U)
  + MRF.DDt(U)
  + turbulence->divDevReff(U)
 ==
    fvOptions(U)
);
```

There's some extra stuff here to handle some features of OpenFoam that you can ignore (the MRF part and the fvOptions part). The rest of the momentum equation consists of the standard stuff, plus a turbulence term *turbulence-¿divDevReff(U)*. This is the divergence of the Reynolds stress tensor that we've discussed in lecture. If we look back into *createFields.H* we can see where the *turbulence* object is initialized.



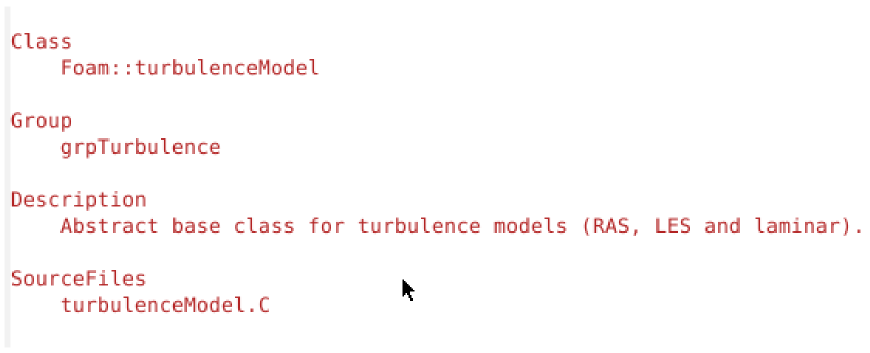
```
autoPtr<incompressible::turbulenceModel> turbulence
(
    incompressible::turbulenceModel::New(U, phi, laminarTransport)
);
```

It turns out the *turbulence* is a pointer to an object of type *incompressible::turbulenceModel*. What we're going to do now is give a crash course in how to chase objects down in the OpenFoam source code so we can see what they actually do - for many of your final projects you will need to do similar things. To do this, write *src* to switch to the OpenFoam source code directory.



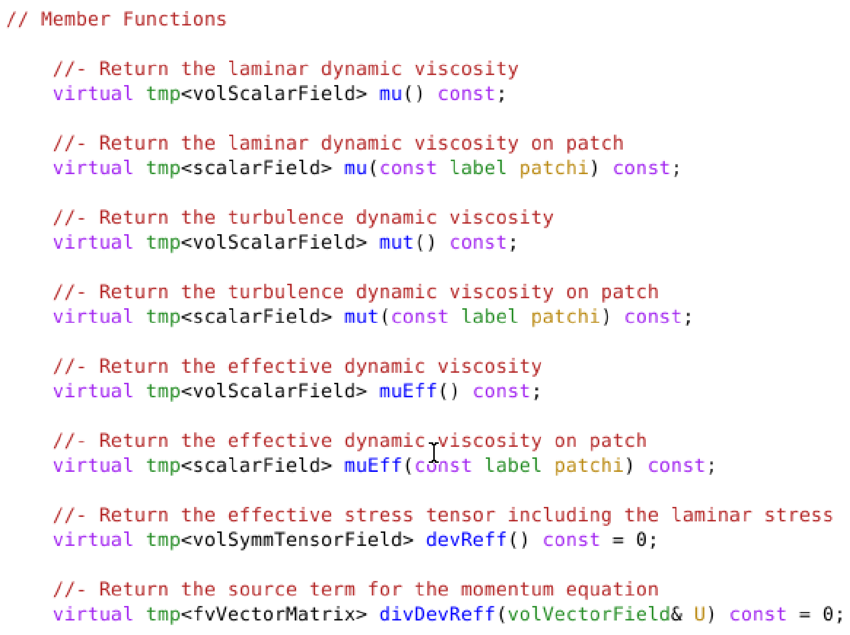
```
[ntrask@node476 run]$ src
[ntrask@node476 src]$ pwd
/gpfs/runtime/opt/openfoam/3.0+/OpenFOAM-v3.0+/src
[ntrask@node476 src]$ ls
Allwmake         engine                  lagrangian  postProcessing    sixDoFRigidBodyMotion
combustionModels fileFormats             mesh        Pstream           surfMesh
conversion       finiteVolume            meshTools   randomProcesses   thermophysicalModels
dummyThirdParty  fvAgglomerationMethods  ODE         regionCoupled     topoChangerFvMesh
dynamicFvMesh    fvMotionSolver          OpenFOAM    regionModels      transportModels
dynamicMesh      fvOptions               OSspecific  renumber          triSurface
edgeMesh         genericPatchFields      parallel    sampling          TurbulenceModels
```

Inside here you can see a number of subdirectories pertaining to the different parts of the OpenFoam library. So far, everything that we've used can be found in the subdirectory *OpenFOAM* (which pertains to stuff like defining dimesionedScalars, matrices, vectors, etc) and the subdirectory *finiteVolume* (which is where all the stuff related to the FV method like boundary conditions, meshes, etc live). For this tutorial, you can see a subdirectory called *TurbulenceModels*. Inside this directory you will find a series of subdirectories related to all the different turbulence models. If you enter the *turbulenceModels* subdirectory you will find a pair of files describing the turbulence model base class. Opening the header, you can see the interface to any turbulence model object will have to have the set of functions necessary (e.g. construct the Reynolds stresses, compute the turbulent kinetic energy, compute the eddy viscosity, etc).



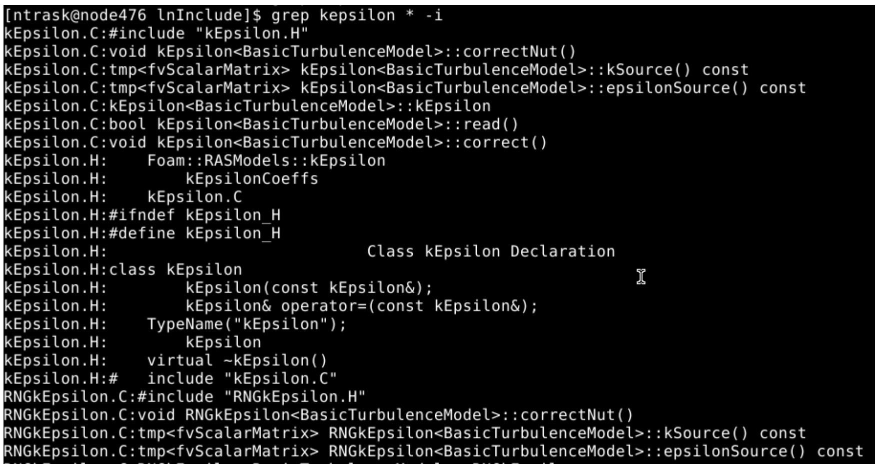
```
Class
    Foam::turbulenceModel

Group
    grpTurbulence

Description
    Abstract base class for turbulence models (RAS, LES and laminar).

SourceFiles
    turbulenceModel.C
```

Now if we back up a directory into *src/TurbulenceModels/incompressible* you will find a file called *incompressibleTurbulenceModel*. If you open up the header for this object you will see from the class description that this object inherits from turbulenceModel and requires some extra functions - specifically the function *divDevReff* that pisoFoam uses that we are looking for. What this interface tells us is that *any* incompressible turbulence model in OpenFoam will have to have a function that returns the divergence of the Reynolds stress tensor.



```
// Member Functions

    //- Return the laminar dynamic viscosity
    virtual tmp<volScalarField> mu() const;

    //- Return the laminar dynamic viscosity on patch
    virtual tmp<scalarField> mu(const label patchi) const;

    //- Return the turbulence dynamic viscosity
    virtual tmp<volScalarField> mut() const;

    //- Return the turbulence dynamic viscosity on patch
    virtual tmp<scalarField> mut(const label patchi) const;

    //- Return the effective dynamic viscosity
    virtual tmp<volScalarField> muEff() const;

    //- Return the effective dynamic viscosity on patch
    virtual tmp<scalarField> muEff(const label patchi) const;

    //- Return the effective stress tensor including the laminar stress
    virtual tmp<volSymmTensorField> devReff() const = 0;

    //- Return the source term for the momentum equation
    virtual tmp<fvVectorMatrix> divDevReff(volVectorField& U) const = 0;
```

OK - so the last thing we want to do is figure out what the $k - \epsilon$ model looks like in OpenFoam. There must be a $k - \epsilon$ object somewhere that inherits from these base classes. Let's go back to the turbulenceModel directory. In there you will find a folder called *lnInclude.* This folder contains *symbolic links* to all of the headers and C-files in the directory. A symbolic link is a bit like a pointer - if you try to open any of these symbolic link files the operating system will re-route you to the file it's pointing to. We're not quite sure what we're looking for, but there should be something related to the $k - \epsilon$ model somewhere. If we search for the phrase *kepsilon* then maybe we'll find something.



```
[ntrask@node476 lnInclude]$ grep kepsilon * -i
kEpsilon.C:#include "kEpsilon.H"
kEpsilon.C:void kEpsilon<BasicTurbulenceModel>::correctNut()
kEpsilon.C:tmp<fvScalarMatrix> kEpsilon<BasicTurbulenceModel>::kSource() const
kEpsilon.C:tmp<fvScalarMatrix> kEpsilon<BasicTurbulenceModel>::epsilonSource() const
kEpsilon.C:kEpsilon<BasicTurbulenceModel>::kEpsilon
kEpsilon.C:bool kEpsilon<BasicTurbulenceModel>::read()
kEpsilon.C:void kEpsilon<BasicTurbulenceModel>::correct()
kEpsilon.H:    Foam::RASModels::kEpsilon
kEpsilon.H:        kEpsilonCoeffs
kEpsilon.H:    kEpsilon.C
kEpsilon.H:#ifndef kEpsilon_H
kEpsilon.H:#define kEpsilon_H
kEpsilon.H:                             Class kEpsilon Declaration
kEpsilon.H:class kEpsilon
kEpsilon.H:        kEpsilon(const kEpsilon&);
kEpsilon.H:        kEpsilon& operator=(const kEpsilon&);
kEpsilon.H:    TypeName("kEpsilon");
kEpsilon.H:        kEpsilon
kEpsilon.H:    virtual ~kEpsilon()
kEpsilon.H:#   include "kEpsilon.C"
RNGkEpsilon.C:#include "RNGkEpsilon.H"
RNGkEpsilon.C:void RNGkEpsilon<BasicTurbulenceModel>::correctNut()
RNGkEpsilon.C:tmp<fvScalarMatrix> RNGkEpsilon<BasicTurbulenceModel>::kSource() const
RNGkEpsilon.C:tmp<fvScalarMatrix> RNGkEpsilon<BasicTurbulenceModel>::epsilonSource() const
```

Sure enough, there's a file called *kEpsilon.H* in the directory. If you enter *ls -lt kEpsilon.H* you will see where that file lives. If we go to that directory, we'll see a header and C-file there describing the object. In the header you'll see declarations of everything that object does, along with some references to journal papers describing the details of the model, the default model coefficients used in this implementation, and we can see that it has a function called correct that solves the $k - \epsilon$ equations. If we open up the C-file, we can take a look at the implementation of the interface and we will finally see how the method is implemented.

Depending upon your final project, the details of turbulence modelling may or may not be relevant. What is important though is that you get a feel for how to dig through the source code. Since OpenFoam has limited documentation (as do most real-life code), this is often the only way to figure out how something works.

Assignment: In the foamCases directory of the course website you will find a case called sphereCase. This is a fully 3D case of a sphere in a channel set up to do some turbulence modeling and compute lift/drag coefficients of the flow. Your final HW assignment will be to replicate a plot of drag coefficient vs Reynolds number similar to the one on the course website. You'll need to specify the inlet conditions on the turbulence model quantities, and adjust the final runTime and transport properties to ensure a converged simulation for a given Reynolds number. You can find a reference for how to set reasonable inflow boundary conditions for turbulence models on the course website.

For this assignment I expect a more detailed presentation - 3 pages describing your process and how you decided that your results were converged, some figures showing qualitatively the difference in the turbulent quantities at different Reynolds numbers, and finally the drag coefficient vs Reynolds number plot.