How to write OpenFOAM Apps and Get On in CFD

10th OpenFOAM Workshop, U. Michigan, Ann Arbor

Dr Gavin Tabor

30th June 2015

UNIVERSITY OF EXETER

Basics of Programming in OpenFOAM

OpenFOAM can best be treated as a special programming language for writing CFD codes. Much of this language is inherited from C (basic I/O, base variable types, loops, function calls) but the developers have used C++'s object-orientated features to add classes (trans : additional variable types) to manipulate :

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

With these features we can write solvers and utilities for doing CFD.

UNIVERSITY OF EXETER

Header files

Sections of code in C++ programs often coded in separate files – compiler reads in all the files and processes them one by one.

Often it is useful to group certain types of code lines together – eg. all function prototypes as a *header file*.

C++ provides a `preprcessor` which can be used to include files into other files :

```
#include ‘‘myHeaderFile.H’’
```

OpenFOAM uses this more widely to separate sections of code which are widely used. Eg. `CourantNo.H` is a file containing several lines of code for calculating the Courant number – widely used.

Base types

C++ implements some obvious basic types;

- int, float, double, char

OpenFOAM adds some additional classes;

- label, scalar
- dimensionedScalar, vector, dimensionedVector etc...
- storage classes
- GeometricField (volScalarField etc)
- Time, database, IOobject. fvMatrix

UNIVERSITY OF EXETER

These can be used in the same way as built-in types. E.g. OpenFOAM implements a complex number class `complex`. Users can declare a complex number :

```
complex myVariable(5.0,2.0);
```

access functions associated with the class :

```
Info << myVariable.Re() << endl;
```

and perform algebraic operations

```
sum = myVariable + myVariable2;
```

where appropriate

Output

C++ output is through *streams* – objects to which output can be redirected using the << operation.

OpenFOAM implements its own versions of these for IO (screen); `Info` object :

```
Info<< "Time = " << runTime.timeName() << nl << endl;
```

Communication to disk (files, dictionaries, fields etc) controlled by `IOobject`

Example – magU

```
#include "fvCFD.H"

int main(int argc, char *argv[])
{
#   include "addTimeOptions.H"

#   include "setRootCase.H"

#   include "createTime.H"

    instantList Times = runTime.times();

#   include "createMesh.H"
```

- A program must contain at least one block called `main`
- OpenFOAM uses `#include` to store commonly-used code sections
- `runTime` is a variable of OpenFOAM class `Time` – used to control the timestepping through the code

```
    for(label i=0; i<runTime.size(); i++)
    {
        runTime.setTime(Times[i],i);
        Info << "Time: " << runTime.value() << endl
        volVectorField U
        (
             IOobject
             (
                 "U",
                 Times[i].name(),
                 runTime,
                 IOobject::MUST_READ
             ),
             mesh
         );
         volScalarField magU
         (
              IOobject
              (
                 "magU",
                 Times[i].name(),
                 runTime,
                 IOobject::NO_READ,
                 IOobject::AUTO_WRITE
              ),
              ::mag(U)
         );
         magU.write();
    }
    return 0;
}
```

- Loop over all possible times
- Read in a `volVectorField U`
- Construct a `volScalarField magU`
- and write it out.

fvMesh

Geometric (mesh) information held in `fvMesh` object. Basic mesh information held includes points, faces, cells and boundaries (eg. points as `pointField`, cells as `cellList`).

Read in from `constant/polyMesh`

Additional information such as cell centres, face centres available.

Addressing information also held (eg. edgeFaces – all edges belonging to a face).

`fvMesh` also responsible for mesh changes due to mesh motion.

UNIVERSITY OF EXETER

GeometricField

Fields of objects (vectors, scalars, tensors etc) defined at each point on the mesh :

`volScalarField` field of scalars (eg. pressure)

`volVectorField` field of vectors (eg. velocity)

`volTensorField` field of 2nd rank tensors (eg. stress)

Each field also has dimensions associated – automatic dimension checking – and boundary conditions.

Access functions provided for boundary and internal values; also previous timestep data (where appropriate).

Algebraic operations defined (+,-,*,/, etc).

UNIVERSITY OF EXETER

IOobject and objectRegistry

OpenFOAM maintains an object registry of entities (such as dictionaries, fields etc.) which are to be read in or written out.

`IOobject` defines the I/O attributes of entities managed by the object registry.

- When the object is created or asked to read : `MUST_READ`, `READ_IF_PRESENT`, `NO_READ`
- When the object is destroyed or asked to write : `AUTO_WRITE`, `NO_WRITE`

Top-level object registry associated with `Time` class – controls time during OpenFOAM simulations

- Usually declared as a variable `runTime`
- Provides a list of saved times `runTime.times()`
- Provides other info; timestep, time names etc.

Example – reading dictionary

```
Info<< "Reading transportProperties" << endl;

IOdictionary transportProperties
(
    IOobject
    (
        "transportProperties",
        runTime.constant(),
        mesh,
        IOobject::MUST_READ,
        IOobject::NO_WRITE
    )
);

dimensionedScalar nu
(
    transportProperties.lookup("nu")
);
```

- Create `IOdictionary` object to interface with `transportProperties` file
- File read in at creation
- Then look up ‘‘`nu`’’ in the dictionary

Example – reading field

```
volVectorField U
(
    IOobject
    (
        "U",
        Times[i].name(),
        runTime,
        IOobject::MUST_READ
    ),
    mesh
);

volScalarField magU
(
    IOobject
    (
        "magU",
        Times[i].name(),
        runTime,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    ::mag(U)
);

magU.write();
```

- `volVectorField` read in from disk
- *Must* be read
- Associated with `runTime` database
- Construct `mag(U)` object – `volScalarField`
- Then write it out.

Higher-level programming

OpenFOAM provides additional classes which aim to provide 'pseudo-mathematical' syntax at top level. Example; heat transfer equation

$$\frac{\partial T}{\partial t} = \kappa \nabla^2 T$$

expressed as

```
solve(fvm::ddt(T) == kappa*fvc::laplacian(T))
```

Explanation :

- `T` (and `kappa`?) are `volScalarFields` defined on a mesh – a discretised representation of the field variables T (κ?)

- `fvc::laplacian` evaluates ∇^2 based on known values of `T` – a differential operator
- `fvm::ddt` constructs a matrix equation of the form

$$\mathcal{M}y = q$$

  representing the time derivative. $\mathcal{M}$, q are known, so this can be inverted to advance one step
- `solve()` performs this inversion to solve for one step

This syntax makes it very easy to understand (and debug) the code. Classes also include other features to prevent mistakes (eg. automatic dimension checking).

fvc::, fvm:: operators

Need to evaluate derivatives of these fields; $\frac{\partial}{\partial t}$, ∇. etc. This is done by finite differencing on the mesh, or by evaluating fluxes on the cell faces. The functions are grouped into *namespaces*.

Two types of derivatives; explict and implicit :

Explicit Evaluate derivative based on known `GeometricField` values – functions grouped into namespace `fvc::`

Implicit Evaluate derivative based on unknown values. This creates a matrix equation

$$\mathcal{M}x = q$$

which has to be inverted to obtain the solution. Functions grouped into namespace `fvm::` – generate `fvMatrix` object which can be inverted.

Explicit evaluation and `fvc::`

These functions perform an explicit evaluation of derivatives (i.e. based on known values). All functions return appropriate `geometricField` :

Operation	Description
`fvc::ddt(A)`	time derivative $\frac{\partial A}{\partial t}$ (A can be scalar, vector or tensor field)
`fvc::ddt(rho,A)`	density-weighted time derivative $\frac{\partial \rho A}{\partial t}$ (ρ can be any scalar field in fact).
`fvc::d2dt2(rho,A)`	Second time derivative $\frac{\partial}{\partial t}\left(\rho \frac{\partial A}{\partial t}\right)$
`fvc::grad(A)`	Gradient ∇A – result is a `volVectorField`. This can also be applied to a `volVectorField` to give a `volTensorField`
`fvc::div(VA)`	Divergence of vector field V_A – returns a `volScalarField`
`fvc::laplacian(A)`	Laplacian of A; $\nabla^2 A$
`fvc::laplacian(s,A)`	Laplacian of A; $\nabla \cdot (s \nabla A)$
`fvc::curl(VA)`	Curl of V_A; $\nabla \times V_A$

UNIVERSITY OF EXETER

Implicit evaluation and `fvm::`

These construct the matrix equation

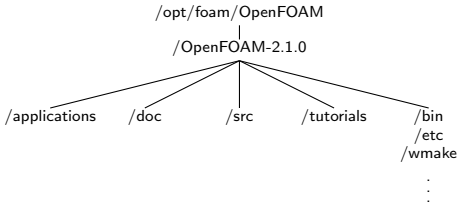
$$\mathcal{M}x = q$$

These functions return `fvMatrix` object. Function names are the same as before (but with `fvm::` rather than `fvc::`).

Some additional functions :

Operation	Description
`fvm::div(phi,A)`	Divergence of field A (can be a vector or tensor as well as a scalar field) explicitly using the flux ϕ to evaluate this.
`fvm::Sp(rho,A)`	Implicit evaluation of the source term in the equation.
`fvm::SuSp(rho,A)`	Implicit/Explicit source term in equation, depending on sign of `rho`

Compilation



Easiest to modify code from core library

Applications (apps) in `/applications` subdirectory

Copy to user directory! (alongside user case directory)

App directory structure

Example – `icoFoam`

```
<grtabor@emps-copland>ls
createFields.H Make   icoFoam.C   icoFoam.dep
```

Directory contains

- File `icoFoam.C`
- Other files (`.H`, `.C`)
- Directory `Make`

To compile, type `wmake`

UNIVERSITY OF EXETER

Compilation

wmake is a make system – directs the compiler to compile specific files in particular ways. Controlled through files in Make :

- files – specifies which user-written files to compile and what to call the result
- options – specifies additional header files/libraries to be included in the compilation.

files :

```
icoFoam.C
EXE = $(FOAM_APPBIN)/icoFoam
```

1. Need to change $(FOAM_APPBIN) to $(FOAM_USER_APPBIN)
2. Probably need to change executable name!
3. Might need to change name of .C file

Example – Boussinesq approximation

For buoyancy-driven flow we often make use of the *B*oussinesq approximation : air modelled as incompressible with a body force proportional to $\Delta\theta$. Can we implement this into `icoFoam`?

Need to solve standard heat conduction equation :

$$\frac{\partial \theta}{\partial t} + \nabla.(\underline{u}\theta) = \frac{\kappa}{\rho_0 C_V}\nabla^2\theta$$

and alter the momentum equation

$$\frac{\partial \underline{u}}{\partial t} + \nabla.\underline{u}\,\underline{u} = -\nabla p + \nu\nabla^2\underline{u} - \beta\underline{g}(\theta_0 - \theta)$$

to accommodate this.

Standard icoFoam

```
int main(int argc, char *argv[])
{
    #include "setRootCase.H"

    #include "createTime.H"
    #include "createMesh.H"
    #include "createFields.H"
    #include "initContinuityErrs.H"

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

        #include "readPISOControls.H"
        #include "CourantNo.H"

        fvVectorMatrix UEqn
        (
            fvm::ddt(U)
          + fvm::div(phi, U)
          - fvm::laplacian(nu, U)
        );
        solve(UEqn == -fvc::grad(p));
```

Include files – `createFields.H`

Solve

$$a_p U_p = H(U) - \nabla p$$

to find U_p – Momentum predictor

```
        for (int corr=0; corr<nCorr; corr++)
        {
            volScalarField rUA = 1.0/UEqn.A();

            U = rUA*UEqn.H();
            phi = (fvc::interpolate(U) \& mesh.Sf())
                + fvc::ddtPhiCorr(rUA, U, phi);

            adjustPhi(phi, U, p);

                fvScalarMatrix pEqn
                (
                    fvm::laplacian(rUA, p)
                 == fvc::div(phi)
                );

                pEqn.setReference(pRefCell, pRefValue);
                pEqn.solve();

#           include "continuityErrs.H"

            U -= rUA*fvc::grad(p);
            U.correctBoundaryConditions();
        }
}
```

Enter pressure loop – set up variables

Solve

$$\nabla . \left(\frac{1}{a_p} \nabla p \right) = \sum_f \underline{S} . \left(\frac{H(U)}{a_p} \right)_f$$

to find p – Pressure corrector

Update flux using

$$F = \underline{S} . \left[\left(\frac{H(U)}{a_p} \right)_f - \left(\frac{1}{a_p} \right)_f (\nabla p)_f \right]$$

boussinesqFoam

Modify this in the following way :

1. Open `createFields.H` and read in the various properties :

```
    dimensionedScalar kappa
    (
        transportProperties.lookup("kappa")
    );
```

   (similar lines for `rho0`, `Cv`, `theta0` and `beta`). Also worth introducing `hCoeff` :

```
    dimensionedScalar hCoeff = kappa/(rho0*Cv);
```

2. Introduce gravitational accelleration $\underline{g}$; read in from the same dictionary, but is a `dimensionedVector` rather than a `dimensionedScalar`.
3. Create a temperature field `theta` as a `volScalarField` and read it in. This is very similar to the pressure field, so make a copy of this and modify accordingly.

(...cont)

4. Modify the momentum equation (`UEqn`) to add the term

```
          + beta*g*(theta0-theta)
```

5. End of the PISO loop – create and solve the temperature equation :

```
fvScalarMatrix tempEqn
(
    fvm::ddt(theta)
  + fvm::div(phi,theta)
  - fvm::laplacian(hCoeff,theta)
);

tempEqn.solve();
```

6. Compile this using `wmake` (rename executable as `boussinesqFoam`)

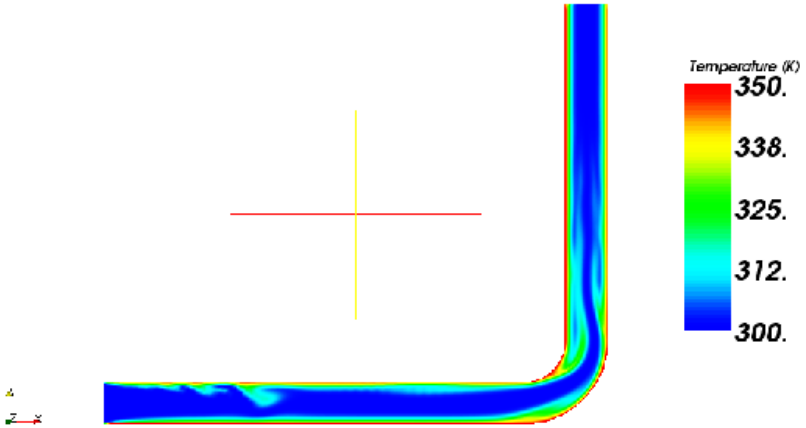
Case

We need to modify a case to function with `boussinesqFoam`. Use a pipe flow problem – modify as follows;

1. Create a `theta` file in the `0` timestep directory. This is best done by creating a copy of `U` and editing it. Don't forget to change the dimensions of `theta` as well.
2. Introduce the physical parameters. `boussinesqFoam` looks for the thermophysical constants in `transportProperties`; check that these are in there and that the values are correct.
3. The differencing schemes need to be specified for the `theta` equation. These are in `fvSchemes`; check that they are appropriate.
4. Finally, `solvers` in `fvSolution` needs an entry for the `theta` equation. Again, this has been provided, but you should check that it is correct.

UNIVERSITY OF EXETER

Results



Example 2 – Casson model

The Casson model is a non-Newtonian viscosity model – used for chocolate, blood, etc. Can we implement in OpenFOAM?

Stress-strain relation for a fluid

$$\tau = \rho\nu\dot{\gamma} \qquad \text{where} \qquad \dot{\gamma} = \frac{1}{2}\left(\nabla\underline{u} + \nabla\underline{u}^T\right) \qquad \text{is rate of strain tensor}$$

$\nu = const$ is a Newtonian fluid. $\nu = \nu(\dot{\gamma}, \ldots)$ is non-Newtonian.

Casson model :

$$\nu(J_2) = \frac{\left[\left(\eta^2 J_2\right)^{1/4} + \sqrt{\frac{\tau_y}{2}}\right]^2}{\rho\sqrt{J_2}} \qquad \text{where} \qquad J_2 = \|\frac{1}{2}\left(\nabla\underline{u} + \nabla\underline{u}^T\right)\|^2$$

Implementation

To implement this convert the viscosity `nu` from `dimensionedScalar` into `volScalarField`. In `createFields` we create an appropriate `volScalarField` :

```
Info << "Reading␣field␣nu" << nl << endl;
volScalarField nu
(
    IOobject
    (
        "nu",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
```

and remove the original definition of `nu` as a `dimensionedScalar`.

Also need to read in Casson model coefficients in `createFields.H`.

Calculate the value of `nu` somewhere within the iterative loop :

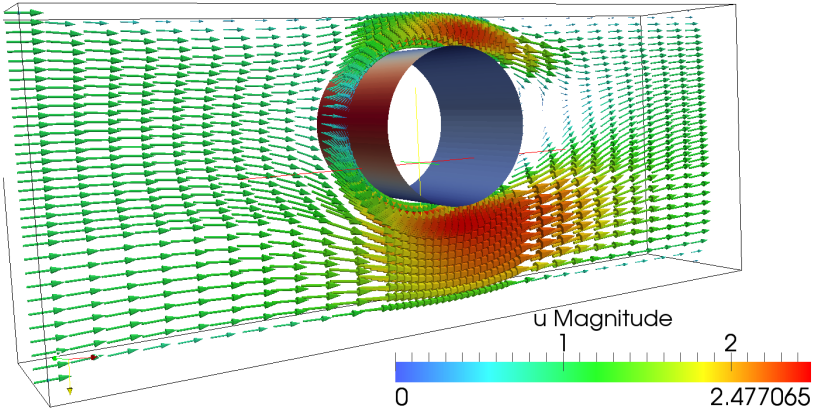
```
volScalarField J2 = 0.5*magSqr(symm(fvc::grad(U))) +
  dimensionedScalar("tiny",dimensionSet(0,0,-2,0,0,0,0),0.0001);

nu = sqr(pow(sqr(eta)*J2,0.25) + pow(tau_y/2,0.5))/(rho*sqrt(J2));
```

Note :

- J2 created locally – not being saved
- Introduce “tiny” to avoid division by zero

Results : offsetCylinder case from tutorials



Run time selectivity

`nonNewtonianIcoFoam` allows selection of which viscosity model we want. How does this work, and can we hang our new model into this framework?

Need to understand how classes work. OOP is about more than designing new language types – it allows us to define relationships between classes.

Two possible ways to program the `complex` class. A complex number can be represented by real and imaginary variables :

```
class complex
{
        //- Real and imaginary parts of the complex number
        scalar re, im;
        .....
```

This is *encapsulation*, a "has-a" relationship

Inheritance

Alternatively, recognise that a complex number is a point on a 2-d plane, with particular extra properties (phase angle, functions such as log...).

If we had an existing class `point` we could *extend* this to add extra features :

```
class complex : public point
{
      extra parts go here!!
```

This is *inheritance*, a "is-a" relationship. The new class *extends* the definition of the old one.

Interface vs. Implementation

In practice, the users don't need to know *how* the complex number is represented (the *implementation*) – just what functions they can use. This is the *interface* – defined by the class definition.

We can take this further and define a *virtual base class*, which is just the interface with *no* implementation. Any class derived from this has to define how the various functions work, but will thus have the same interface, and so be interchangeable.

All non-Newtonian viscosity models have to return a value for ν. If we derive them all from a virtual base class, this will force them to have the same interface, so they can be accessed from a list – *run time selection*.

Polymorphism

This is an example of a concept known as *polymorphism* :
"One of the key features of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class. *Polymorphism* is the art of taking advantage of this simple but powerful and versatile feature."

All viscosity models in OpenFOAM are derived from a base class `viscosityModel`. This defines a run time selector function and virtual functions `nu()` and `correct()`

Classes stored in

`/opt/OpenFOAM-2.1.0/src/transportModels/incompressible/viscosityModels`

and sub-directories

Implementing the Casson model

Easiest (again) to copy existing model – eg. `powerLaw`

1. Copy `powerLaw` sub-directory to home directory
2. Rename the files `Casson.H` and `Casson.C`; and also change all instances of `powerLaw` to `Casson` inside the files.
3. Change over private data to hold the Casson model coefficients; re-implement constructor, `nu()` and `correct()` functions

The make system will compile libraries as well – command `wmake libso`. Again; this uses information from a directory `Make`. Copy the one from `viscosityModels`, and modify :

4. `files` needs to read in `Casson.C` and write to a library `libChocolateFlowModels` in the user directory
5. `options` needs to reference the original `transportModels` library.

files, options

```
<grtabor@sibelius>more files
Casson.C
LIB = $(FOAM_USER_LIBBIN)/libChocolateFlowModels
```

```
<grtabor@sibelius>more options
EXE_INC = \
    -I.. \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/transportModels/incompressible/lnInclude

LIB_LIBS = \
    -lfiniteVolume \
    -lincompressibleTransportModels
```

Alterations to case files

Finally need to alter the `offsetCylinder` case files :

6. Include the line

```
libs ( "libChocolateFlowModels.so" );
```

in `controlDict`

7. Specify the coefficients in `transportProperties` :

```
transportModel Casson;
CassonCoeffs
{
    eta       eta [ 1 -1 -1 0 0 0 0 ] 4.86;
    tau_y     tau_y [ 1 -1 -2 0 0 0 0 ] 14.38;
    rho       rho [1 -3 0 0 0 0 0] 1200;
}
```

`offsetCylinder` case should now run with `nonNewtonianIcoFoam`!!

Summary

Take-home message(s) :

- OpenFOAM programming should not be seen as scary or risky!!
- Think “MatLab for CFD”.
- Easy to read code; modify existing apps; implement transport equations.

Happy to circulate tutorial file on programming OpenFOAM : email: g.r.tabor@ex.ac.uk

UNIVERSITY OF EXETER