CCP-WSI Programming Day : Course Notes

Learning Outcomes:

Modifying, writing and compiling OpenFOAM apps, model (shared object) libraries for runtime selection. These notes accompny the code examples which you should have downloaded and unpacked into your home `run` directory. The examples all use v5 from the foundation (but may work with other branches).

WSI.1 Compiling in OpenFOAM

One common task with OpenFOAM is modifying and recompiling code. In OpenFOAM, each individual program is stored in its own separate directory, which can contain .C (code) and .H (header) files which are compiled and joined together to create an executable program. OpenFOAM provides a very convenient mechanism for compiling its programs; just go to the specific code directory and type

```
wmake
```

and the program will be compiled.

[**Q.WSI.1**] Compile `magU`. Go into the code directory, type `wmake`
[**Q.WSI.2**] Try altering `magU` to calculate enstropy, defined as

$$\Sigma = \frac{1}{2}|\nabla \times \underline{u}|^2$$

You can also write a code to calculate vorticity. In both cases you will need to calculate the explicit curl using the function

```
fvc::curl(U)
```

The function `::magSqr` may also be useful.
[**Q.WSI.3**] Compile `burgersFoam`. Go into the code directory, type `wmake`.
[**Q.WSI.4**] `burgers1d` is a case directory for the sine wave `burgersFoam` case. Run `burgersFoam` on this. A `sampleDict` file is set up to sample the values to produce the plots shown in the lecture. Run

```
postProcess -func sampleDict
```

to generate the data.

WSI.2 Compiling a pre-existing code

A good starting point for writing your own apps is to take and modify an existing program. The core distribution contains a selection of pre-written codes (look for the `applications` directory in the OpenFOAM installation directory), and it is well worth looking through to see what is available. Howwever it would be poor practice to modify the codes in the core distribution itself (and you may not have edit permissions to do this anyway). Instead, copy the relevant program directory to your home OpenFOAM directory and modify the copy. To do this requires making some changes, as follows.

Within the program directory is a subdirectory `Make` which stores information to control the compilation process. Specifically, inside this subdirectory are two files; `files` and `options`. `options` specifies various flags to the compiler (for example where to find additional libraries) and need not concern us at this stage. `files` specifies which files need to be compiled, and importantly, where the resulting executable is to be placed. If you look in the `files` file for `icoFoam` you will find the line

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

which specifies that the executable is to be called `icoFoam` and that it is to go in a particular directory (specified by the environment variable `FOAM_APPBIN`). This line will need to be changed in two ways :

1. The directory specified will be `FOAM_APPBIN` – this would need to be changed to `FOAM_USER_APPBIN` as we do not have the correct privilege to write to the main installation (nor is it a good idea to be overwriting the main installation even if it is possible).
2. The name of the executable will probably need to be changed, to avoid confusion. For example we would probably want to have a program called `myIcoFoam` to avoid confusion with `icoFoam` itself! We could rename the files as well, but this is seldom necessary.

[**Q.WSI.5**] In directory `lecture2` is a copy of the core `icoFoam` code. Make a copy of this and change the name to `boussinesqFoam`. Change the filename `icoFoam.C` to `boussinesqFoam`, and make the required changes in `files`. Compile.

WSI.3 Buoyancy and heat transfer

This tutorial models buoyancy effects for a stream of hot air. To do this we must modify `icoFoam` to take account of heat transfer and buoyancy. Heat transfer involves solving the standard heat conduction equation;

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

Changes of temperature create changes of density in the fluid and this generates different gravitational forces, leading to convective motion. A full solution would require solving for a compressible flow, and would need to be done very accurately because the buoyancy effects are driven by only very small variations in density. However, under certain circumstances the Boussinesq approximation can be used.[1] In this, we can neglect density differences in the fluid and treat it as an incompressible fluid, but with a body force proportional to the temperature

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

where β is the coefficient of thermal expansion of the fluid, $\underline{g}$ the gravitational force vector and θ_0 a reference temperature.

A copy of `icoFoam` is contained in the tutorial file, renamed as `boussinesqFoam`, and we will modify it to introduce these additional effects. We need to read in the various coefficients; $\kappa, \rho_0, C_V, \theta_0$ and β. The `transportProperties` dictionary is already opened in the file `createFields.H`, so it makes sense to read the additional properties from here. Open `createFields.H` in emacs and add lines

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

and similar lines for `rho0, Cv, theta0` and `beta`. It is also worth introducing a variable `hCoeff` :

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

We need to introduce the gravitational accelleration $\underline{g}$; this can be read in from the same dictionary, but of course is a `dimensionedVector` rather than a `dimensionedScalar`.

`createFields.H` also creates the dependent variable fields; we need to 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 the lines starting

```
Info<< "Reading field p" << endl;
volScalarField p
⋮
```

and change them to create a field `theta` instead.

[1]This is the Boussinesq approximation in buoyancy, which is different from the Boussinesq approximation for turbulence modelling.

In the file `icoFoam.C` we need to modify the momentum equation to include the extra term, and to create and solve the temperature equation. The momentum equation is `UEqn` ; add the additional term

$$-\beta \underline{g}(\theta_0 - \theta)$$

to this. Add the line

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

At the end of the PISO loop we need to create and solve the temperature equation :

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

tempEqn.solve();
```

Having done this, type `wmake` to compile the code.

Also in `tutorial4` is a case, `bendHeat`, which consists of a duct with cooling water flowing along it. Run `blockMesh` to generate the mesh, and take a look at the details of the `blockMeshDict`; this illustrates how to create curved edges in `blockMesh`.

We need to modify this case to function with `boussinesqFoam`. This requires the following ;

1. Create a `theta` file in the `0` timestep directory. This is best done by creating a copy of `U` and editing it. The inlet conditions for the temperature are $\theta = 300$ K, and the wall temperatures are $\theta = 350$K. 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.

Then run `boussinesqFoam` on the case, and plot the results for two of the resulting timesteps. Note that if there are errors in the input steps (1-3 above) the code will not run; but the resulting error messages are quite informative.

[**Q.WSI.6**] Alter your code `boussinesqFoam` (as compiled in the last section) to include the Boussinesq approximation for buoyancy and heat transfer.
[**Q.WSI.7**] Alter the `bendHeat` case as indicated to include the `theta` field, physical parameters and numerical settings. Run `boussinesqFoam` on this and post-process the results.
[**Q.WSI.8**] Rewrite `boussinesqFoam` to include an averaged temperature field `thetaAv`; for each timestep update this using the current temperature field using the running average formula :

$$\theta_{av}^{n+1} = \frac{n}{n+1}\theta_{av}^{n} + \frac{1}{n+1}\theta^{n+1}$$

Plot this and the temperature profile at the outlet.
[**Q.WSI.9**] From a fresh copy of `icoFoam` try implementing the Casson equations for a non-Newtonian fluid. Identify a suitable test case and try running the resulting code.

WSI.4 Adding new model classes – `chocolate`

In OOP, anything can be a class. So far we have looked at how OpenFOAM uses classes to create a pseudo-mathematical high level notation for tensor fields and matrix classes which mimic conventional PDEs. However, classes can also be used to organise physical models for turbulence, combustion, non-Newtonian viscous models etc. As shown in the lectures, virtual base classes are used to define an interface to a group of models, specifying how all members of the group operate, whilst the details of a specific model are implemented in derived classes (using of course OpenFOAM's high level mathematical language to implement the model equations). As well as reflecting the common structure of (for instance) all turbulence models, this forms the basis for run time selection of these models – through polymorphism, all derived classes behave in the same way and so are interchangeable. These groups of models are compiled in the base code as *shared object libraries* with the file extension `.so`

This enables us to add extra models to the library, provided they are derived from the correct virtual base class. One might think that this would require us to recompile the whole shared object library, but in fact it is possible simply to add the new classes on by compiling a new shared object library which can then be linked in at runtime. The class structure for a model class can be complex, so the best approach is to identify and modify an existing model class. Here we will add a new class `Chocolate` to the group of non-Newtonian viscosity models in OpenFOAM by modifying an existing model `powerLaw`.

In directory `lecture3` there is a code directory `Chocolate`. This was developed by copying across the `powerLaw` directory from the base installation and modifying it; specifically, changing `powerLaw` to `Chocolate` throughout, changing the class data, and replacing the constructor functions, `nu()` and `correct()` within the class. Most importantly, the new code needs to access the original shared object library, so we need to reference `incompressibleTransportModels.so` and the associated header files in `options` :

```
EXE_INC = \
    -I.. \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/transportModels/incompressible/lnInclude

LIB_LIBS = \
    -lfiniteVolume \
    -lincompressibleTransportModels
```

The new shared object library is written to `FOAM_USER_LIBBIN` as specified in `files` :

```
Chocolate.C
LIB = $(FOAM_USER_LIBBIN)/libChocolateFlowModels
```

To make use of this class as part of `nonNewtonianIcoFoam` we need to specify the new shared object library in the case files. To do this, add the line

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

to `controlDict`. You may recognise this entry from running `functionObjects` or other add-on libraries – it directs the code to include the specified shared object library at run time. The `Chocolate` model class expects to find specific coefficients in `transportProperties`, so we have to add these to that dictionary :

```
transportModel Chocolate;
ChocolateCoeffs

    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;
```

With these modifications, `nonNewtonianIcoFoam` is fully equipped to use the new viscosity model.

[**Q.WSI.10**] The directory `Chocolate` (in `lecture3` directory) contains the files for the `libChocolateFlowModels` library, modified from `powerLaw` as outlined here. Compile these using the command

```
wmake libso
```

to create the appropriate library.
[**Q.WSI.11**] In the `lecture3` directory is a copy of the `offsetCylinder` case from the OpenFOAM tutorials. Modify this as described to access the new library (change `controlDict` and add the new entries to `transportProperties`) and run the case. You might like to compare the results with other non-Newtonian models.
[**Q.WSI.12**] Directory `lecture3b` contains a new boundary condition, `parabolicInletVelocity`; modified from `cylindricalInletVelocity` as discussed in the lectures. Try compiling this and running the `ParaPipe` case.

WSI.5 Appendix – some useful class functions

OpenFOAM provides a vast array of classes, and even the key ones (`time, fvMesh` etc) are actually complex hierachies of derivation. The doxygen-derived html descriptions can be useful in understanding how the classes fit together and what class functions are available. These can be generated locally or accessed online (https://cpp.openfoam.org/v5/). However even this information can be quite complex to understand.

Looking at code is another good way to learn or identify useful classes and class functions. In the `lecture3b` directory there is an app `boundaries`, which prints out information about every boundary patch in a mesh (patch names and face centre locations). You might like to look through this and see if you can understand how it works.

Below are some useful class functions and other code fragments used in various of the codes in this course. Note that these entries are intended to be informative rather than pedantically accurate! In particular, a lot of the following functions are `const` or have `const` versions – i.e. they permit access to the information without allowing it to be changed. There are often different ways to achieve the same result, and the key classes here are in fact complex hierachies of composite classes and references. The objective here is to give a little more information to help you read the codes given in the lectures, and provide a starting point to investigate the different classes.

WSI.5.1 Loops

Obviously repeating a set of instructions is a key element of CFD coding – at the top level to iterate to a solution or over timesteps, at lower levels, to work through all cells, or all boundary patches, or whatever. As a high-level language, OpenFOAM inherits all C++ loop structures, including the `while` loop :

```
while (runTime.loop())
{
    // code goes here!
}
```

which is used in quite a few solver apps; the loop repeats until the condition (`runTime.loop()`) becomes false – see below.

Sometimes we want to loop over elements in a list. OpenFOAM adds structures called *iterators*, and a `forAll` loop structure :

```
forAll (Times, i)
{
    // code goes here!
}
```

which cycles through all appropriate elements in the list (here the list of timesteps `Times`) using an integer variable `i`.

WSI.5.2 Class functions – `Time`

Function	Description	Returns
`.times()`	List of times in case directory. Note that this includes `constant` directory; to avoid this you can use `timeSelector::select0(runTime, args);`	`instantList`
`.setTime()`	Set the `time` object to a specific timestep	
`.loop()`	Returns true if not at the end of the list of timesteps	Boolean return
`.startTime()` `.endTime()`	Return start and end times for simulation	`dimensionedScalar`
`.value()`	Since `Time` is basically a list of `dimensionedScalar`'s, this gives the numeric value without the associated information (dimensions etc)	pure number

WSI.5.3 Class functions – `fvMesh` and related

`fvMesh` is derived from `polyMesh` and inherits a lot of functions from there.

Class functions – `fvMesh`		
Function	Description	Returns
`.C()`	Returns cell centres	`volVectorField`
`.Cf()`	Returns face centres as `surfaceVectorField`	`surfaceVectorField`
`.V()`	Returns cell volumes as the internal part of a `volScalarField` (and previous timesteps : `.V0(), .V00()`)	

`.Sf()`
`.magSf()`
	Returns cell face area vectors
	Same but cell face area magnitudes
		`surfaceVectorField`
		`surfaceScalarField`

`.boundary()`	Returns a reference to the list of boundaries	`fvBoundaryMesh`

`fvBoundaryMesh` is itself derived from `fvPatchList`, i.e. is a list of the boundary patches of the mesh which can be indexed by name.

Class functions – `fvBoundaryMesh`		
Function	Description	Returns
`[‘‘patchName’’]`	Accesses named patch	`fvPatch`
Class functions – `fvPatch`		
`.Cf()`	Returns face centres as `surfaceVectorField`	`vectorField`
`.name()`	Name of patch	`word`

WSI.5.4 `geometricField`

`volScalarField`, `volVectorField`, `volTensorField` are core classes which share significant common features – all are fields of tensors of different ranks which know their dimensions and have boundary information. Thus they are implemented as specific versions of a *template class* `geometricField`. Although this is the correct way to do this, it does make understanding the various classes that bit more complicated.

Most of the arithmetic operations for fields are fairly obvious – `+, -, -` etc. behave as you would expect. However to complete the full set of mathematical operations some additional symbols have been pressed into service : in particular &, ˆ. C++ allows these to be overloaded, but it isn't possible to change the order of precedence – thus they don't necessarily behave quite as one would expect in a long expression. If in doubt, add brackets to make the order of evaluation explicit. Most algebraic functions operate as expected (including `det(T)` for the determinant of a `volTensorField`).

Class functions – `geometricField` and related		
Function	Description	Returns
`internalField()`	Access the internal field (i.e. not including boundary information). This is one of the ways to do this.	field of the correct type (`scalarField` etc)
`.boundaryField()`	Access a list of the boundary patches field values	
`+,-`	Addition, Subtraction	Appropriate `geometricField`
`*`	Multiplication by a `dimensionedScalar`	
`&`	Inner product; dot product for two vector fields	
`^`	Cross product for two vector fields	
`*`	Outer product (higher rank tensors)	
`.T()`	Transpose (for `volTensorField`)	

WSI.5.5 Misc issues

The syntax for reading in a `dimensionedScalar` or other OpenFOAM class from a dictionary is as follows :

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

However this mechanism doesn't work for base (C++) variable types. In particular, to read in a simple floating point number, use the access function `readScalar` :

```
scalar R = readScalar(transportProperties.lookup(‘‘radius’’));
```