OpenFOAM Programming – the basic classes

Prof Gavin Tabor

Friday 25th May 2018

OpenFOAM : Overview

OpenFOAM is an Open Source CCM (predominantly CFD) code based on 2nd order FVM on arbitrary unstructured (polyhedral cell) meshes. Full range of modelling (turbulence, combustion, multiphase) and solution algorithms.

- Several independent versions and developments (-dev, pyFoam)
- Extensive user community :
  - 13th OpenFOAM Workshop: JT University Shanghai
- Academic and commercial usage.

OpenFOAM comes with extensive pre-written solvers – can still be used as a “black box” CFD tool. However, strictly, OpenFOAM is not a CFD code – it is an open source C++ library of classes for writing CFD codes.

CCP-WSI a Collaborative Computational Project in Wave Structure Interaction

UNIVERSITY OF EXETER

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 code new models, solvers and utilities for CFD.

Open Source software development encourages code sharing and information exchange. OpenFOAM's structure and use of C++ is ideal for this and provides a common platform for CFD research.

At the highest level, provides a framework for significant development projects.

Taxonomy of OpenFOAM use :

- **User** Use pre-written apps "as is" – free at point of use.
- **Modeller** Use OpenFOAM modelling language to develop new models, apps
- **Guru** Develops base code (!)

Course requirements and overview

Starting point – I assume :

- You know how to run OpenFOAM simulations
- You have some programming experience – C++ (not essential), C, Python, Java . . .

Course will cover : top level syntax; reading code; modifying existing apps (and coding your own); run time selection and introducing new model classes.

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

All variables need to be initialised before use (good practice) – no null constructors.

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

Higher level syntax : Burgers equation

OpenFOAM provides additional classes which aim to provide 'pseudo-mathematical' syntax at top level. Eg. Burgers equation :

1d Burgers equation :

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}$$

3d version (conservative form) :

$$\frac{\partial \underline{u}}{\partial t} + \frac{1}{2}\nabla.(\underline{u}\ \underline{u}) = \nu\nabla^2\underline{u}$$

Implemented in OpenFOAM as :

```
fvVectorMatrix UEqn
(
    fvm::ddt(U)
  + 0.5*fvm::div(phi, U)
  - fvm::laplacian(nu, U)
);

UEqn.solve();
```

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

Explanation

```
fvVectorMatrix UEqn
(
    fvm::ddt(U)
  + 0.5*fvm::div(phi, U)
  - fvm::laplacian(nu, U)
);

UEqn.solve();
```

- `U` is a `volVectorField` defined on a mesh – a discretised representation of the field variable $\underline{u}$
- `fvm::ddt` etc. construct entries in matrix equation of form

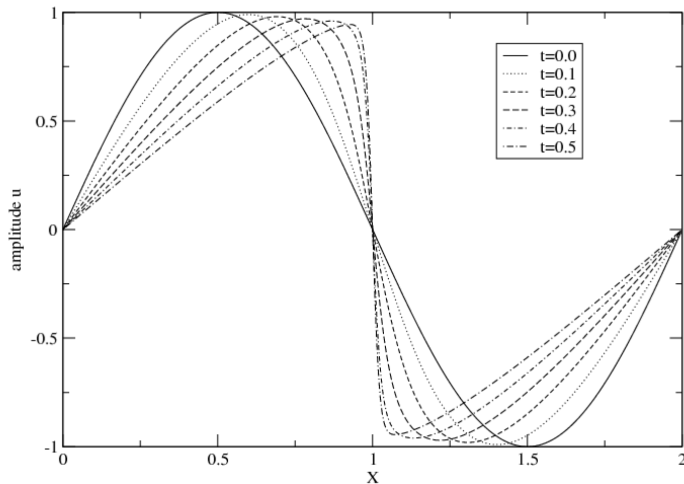
$$\mathcal{M}y = q$$

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

Enclose in loop and iterate :

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

        #include "CourantNo.H"

        fvVectorMatrix UEqn
        (
            fvm::ddt(U)
          + 0.5*fvm::div(phi, U)
          - fvm::laplacian(nu, U)
        );

        UEqn.solve();

        U.correctBoundaryConditions();

        phi = (fvc::interpolate(U) & mesh.Sf());

        runTime.write();

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

Test case : 1-d sine wave



Turbulent Kinetic Energy equation

Equation solved as part of the standard $k - \epsilon$ model :

$$\frac{\partial k}{\partial t} + \nabla . \underline{u} k - \nabla . \left[\left(\nu + \nu_t \right) \nabla k \right] = \nu_t \left[\frac{1}{2} \left(\nabla \underline{u} + \nabla \underline{u}^T \right) \right]^2 - \frac{\epsilon}{k}$$

Implemented as :

```
solve
(
    fvm::ddt(k)
  + fvm::div(phi, k)
  - fvm::laplacian(nu()+nut,k)
  == nut*magSqr(symm(fvc::grad(U)))
  - fvm::Sp(epsilon/k,k)
);
```

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 `preprocessor` 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.

CCP-WSI a Collaborative Computational Project in Wave Structure Interaction
UNIVERSITY OF EXETER

Memory access

Most languages provide facilities to manipulate storage in memory – typically through an `array` of variables – a table of variables that can be accessed using an index

C++ provides 3 mechanisms; arrays, pointers and references.

An array is an indexed block of variables of a specific type;

```
int values[5];
```

creates an array of integers numbered 0 . . . 4.
We can access the individual components using the `[]` syntax;

```
values[2] = 20;
```

Arrays of all types are possible; `int, double, char` etc.

OpenFOAM data structures

OpenFOAM also provides a number of classes for creating and addressing memory; these include : `List`, `ptrList`, `SLList`.

These are *templates* which can be declared for any storage type. Commonly used versions are given specific names, eg.

```
List<label> someList;
```

is the same as

```
labelList someList;
```

CCP-WSI
a Collaborative Computational Project in Wave Structure Interaction
UNIVERSITY OF EXETER

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`

CCP-WSI a Collaborative Computational Project in Wave Structure Interaction

UNIVERSITY OF EXETER

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.

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).

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",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ
    ),
    mesh
);
```

- `volVectorField` read in from disk
- *Must* be read
- Associated with `runTime` database

Example – writing field

```
volScalarField magU
(
    IOobject
    (
        "magU",
         runTime.timeName(),
         mesh,
         IOobject::NO_READ,
         IOobject::AUTO_WRITE
     ),
     ::mag(U)
 );

magU.write();
```

- Construct `mag(U)` object – `volScalarField`
- *No* read – construct using `::mag()` function
- Then write it out.

CCP-WSI a Collaborative Computational Project in Wave Structure Interaction
UNIVERSITY OF EXETER

Example – magU

```
#include "fvCFD.H"

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

#   include "setRootCase.H"

#   include "createTime.H"

    instantList Times = timeSelector::select0(runTime, args);

#   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

CCP-WSI a Collaborative Computational Project in Wave Structure Interaction
UNIVERSITY OF EXETER

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

Compilation

Example – `magU`

```
<grtabor@emps-copland>ls
Make   magU.C   magU.dep
```

Directory contains

- File `magU.C`
- Dependency file `magU.dep`
- Directory `Make`

To compile, type `wmake`

Derivatives

Navier-Stokes equations ;

$$\nabla . \underline{u} = 0$$

$$\frac{\partial \underline{u}}{\partial t} + \nabla . \underline{u}\ \underline{u} = -\frac{1}{\rho}\nabla p + \nu \nabla^2 \underline{u}$$

Several types of derivative here :

$$\frac{\partial}{\partial t} \quad , \quad \frac{\partial^2}{\partial t^2}$$ time derivatives

$$\nabla p = \underline{i}\frac{\partial p}{\partial x} + \underline{j}\frac{\partial p}{\partial y} + \underline{k}\frac{\partial p}{\partial z}$$ Gradient

CCP-WSI a Collaborative Computational Project in Wave Structure Interaction
UNIVERSITY OF EXETER

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \qquad \text{Laplacian}$$

$$\nabla.\underline{u}\,\underline{u} \qquad \text{Transport term}$$

Each has its own peculiarities when being evaluated.

OpenFOAM is a *finite volume* code – integrate equations over volume of computational cell as first step in solution. Spatial derivatives constructed via Gauss' theorem (fluxes at cell faces)

Operators can be evaluated using known values – *explicit* evaluation – or future values (matrix solve) – *implicit* evaluation.

`fvc::`, `fvm::` operators

Evaluation of these terms through named functions. Need to distinguish implicit and explicit versions of mathematically similar terms. The functions are grouped into *namespaces*.

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.

CCP-WSI a Collaborative Computational Project in Wave Structure Interaction

UNIVERSITY OF EXETER

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 $\frac{\partial}{\partial t}(\rho \frac{\partial A}{\partial t})$
`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; $\nabla \times V_A$

CCP-WSI
a Collaborative Computational Project in Wave Structure Interaction
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`

CCP-WSI a Collaborative Computational Project in Wave Structure Interaction

UNIVERSITY OF EXETER