FSB

OpenFOAM: Programming Tutorial

Hrvoje Jasak

h.jasak@wikki.co.uk

Wikki Ltd, United Kingdom and

FSB, University of Zagreb, Croatia

7-9th June 2007

Outline

Objective

- Illustrate basic steps in creating custom applications, adding on-the-fly post-processing and extending capabilities of the library

Background Information

- Organise your work with OpenFOAM
- Programming guidelines: enforcing consistent style
- Debugging OpenFOAM with gdb
- Release and debug version; environment variables and porting

Practical Exercises

- Walk through a simple solver: `scalarTransportFoam`
- Scalar transport, swirl test: non-uniform initial field, using field algebra
- On-the-fly post-processing in the solver
- Manipulating boundary values
- Reading control data from a dictionary
- Walk through a simple utility: `magU`

OpenFOAM Work Space

FSB

Organise Your Work with OpenFOAM

- OpenFOAM is a library of tools, not a monolithic single-executable
- Most changes do not require surgery on the library level: code is developed in local work space for results and custom executables
- In most cases, groups of users rely on a central installation. If changes are necessary, take out only files to be changed
- Environment variables and library structure control the location of the library, external packages (*e.g.* gcc, Paraview) and work space
- For model development, start by copying a model and changing its name: library functionality is unaffected

Local Work Space

- **Run directory**: `$FOAM_RUN`. Ready-to-run cases and results, test loop *etc.* May contain case-specific setup tools, solvers and utilities
- **Local work space**: `/home/hjasak/OpenFOAM/hjasak-1.4`. Contains applications, libraries and personal library and executable space

Creating Your Applications

Customising Solver and Utilities

- Source code in OpenFOAM serves 2 functions
  - Efficient and customised top-level solver for class of physics. Ready to run in a manner of commercial CFD/CCM software
  - Example of OpenFOAM classes and library functionality in use
- Modifications can be simple, *e.g.* additional post-processing or customised solver output or a completely new model or solver

Creating Your Applications

1. Find appropriate code in OpenFOAM which is closest to the new use or provides a starting point
2. Copy into local work space and rename
3. Change file name and location of library/executable: Make/files
4. Environment variables point to local work space applications and libraries: `$FOAM_PROJECT_USER_DIR`, `$FOAM_USER_APPBIN` and `$FOAM_USER_LIBBIN`
5. Change the code to fit your needs

Example: `scalarTransportFoam`

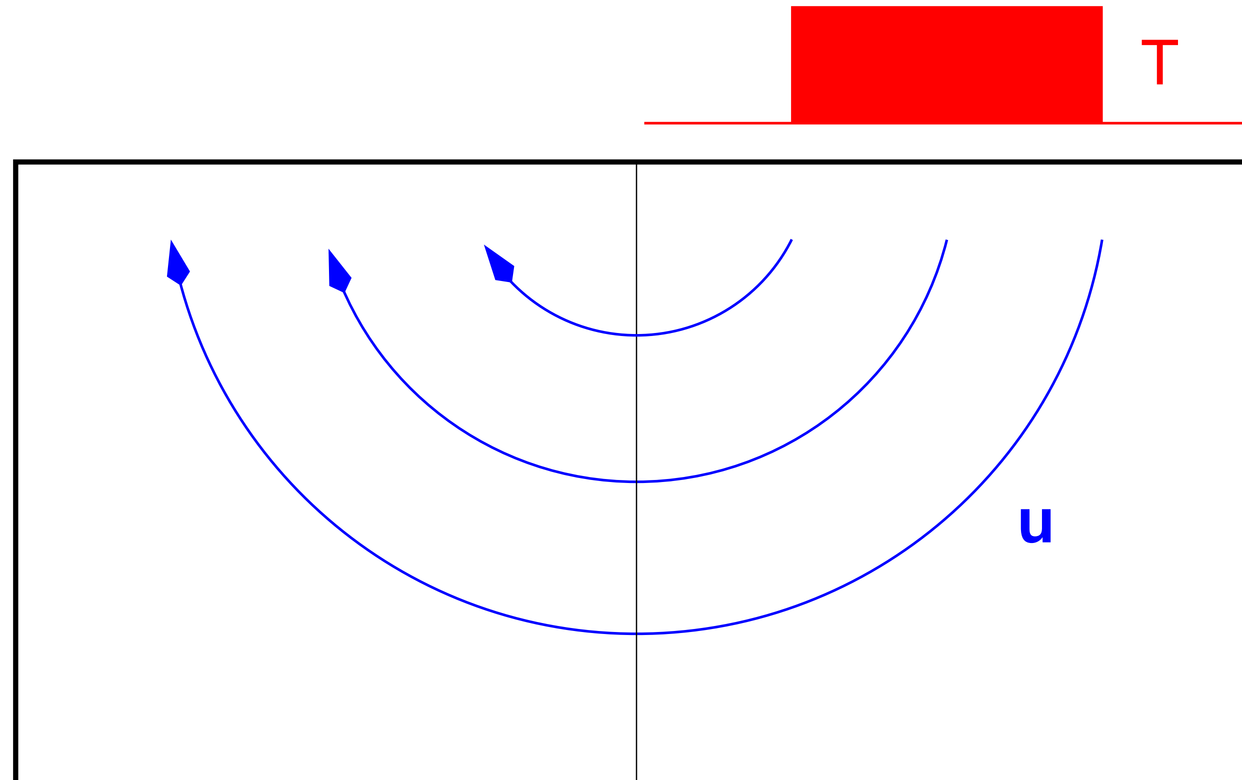
- Manipulating input/output; changing inlet condition; adding a probe

Walk Through a Simple Solver

Solver Walk-Through: `scalarTransportFoam`

- Types of files
  - **Header files**
    - Located before the entry line of the executable
      `int main(int argc, char *argv[])`
    - Contain various class definitions
    - Grouped together for easier use
  - **Include files**
    - Often repeated code snippets, *e.g.* mesh creation, Courant number calculation and similar
    - Held centrally for easier maintenance
    - Enforce consistent naming between executables, *e.g.* `mesh`, `runTime`
  - **Local implementation files**
    - Main code, named consistently with executable
    - `createFields.H`

Swirl Test on `scalarTransportFoam`



- Setting up initial velocity field
- Forcing assignment on boundary conditions
- Types of boundary conditions

Scalar Transport: Swirl Test

Initial Condition Utility

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

// Do cells
const volVectorField& centres = mesh.C();

point origin(1, 1, 0.05);
vector axis(0, 0, -1);

U = axis ^ (centres.internalField() - origin);
U.write();
```

Scalar Transport: Swirl Test

Data Sampling and Additional Output

- Write out sample data
- Locate a cell and a boundary face

Manipulating Boundary Conditions

- Manipulating boundary value from top-level code
- Time-dependent boundary value
- Implementing a boundary condition as a class

New Boundary Condition

Run-Time Selection Table Functionality

- In many cases, OpenFOAM provides functionality selectable at run-time which needs to be changed for the purpose. Example: viscosity model; ramped fixed value boundary conditions
- New functionality should be run-time selectable (like implemented models)
- . . . but should not interfere with existing code! There is no need to change existing library functionality unless we have found bugs
- For the new choice to become available, it needs to be instantiated and linked with the executable.

Boundary Condition: Ramped Fixed Value

- Find closest similar boundary condition: `oscillatingFixedValue`
- Copy, rename, change input/output and functionality. Follow existing code patterns
- Compile and link executable; consider relocating into a library
- Beware of the `defaultFvPatchField` problem: verify code with print statements

Walk Through a Post-Utility

FSB

Utility Walk-Through: `magU`

- Go through a set of results and calculate a scalar field of velocity magnitude from existing velocity field `U`
- In `applications/utilities/postProcessing/velocityField/magU`
- Some code snippets:

```
    // Get times list
    instantList Times = runTime.times();

    // set startTime and endTime depending on -time and -latestTime
#   include "checkTimeOptions.H"

    runTime.setTime(Times[startTime], startTime);

#   include "createMesh.H"

    for (label i=startTime; i<endTime; i++)
    {
        runTime.setTime(Times[i], i);

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

Walk Through a Post-Utility

Utility Walk-Through: `magU`

```
if (Uheader.headerOk())
{
    mesh.readUpdate();
    volVectorField U(Uheader, mesh);
    volScalarField magU
    (
        IOobject
        (
            "magU",
            runTime.timeName(),
            mesh,
            IOobject::NO_READ
        ),
        mag(U)
    );
    magU.write();
}
else
{
    Info<< "    No U" << endl;
}
```

Programming Guidelines

OpenFOAM And Object-Orientation

- OpenFOAM library tools are strictly object-oriented: trying hard to weed out the hacks, tricks and work-arounds
- Adhering to standard is critical for quality software development in C++: ISO/IEC 14882-2003 incorporating the latest Addendum notes

Writing C in C++

- C++ compiler supports the complete C syntax: writing procedural programming in C is very tempting for beginners
- Object Orientation represents a paradigm shift: the way the problem is approached needs to be changed, not just the programming language. This is not easy
- Some benefits of C++ (like data protection and avoiding code duplication) may seem a bit esoteric, but they represent a real qualitative advantage
    1. Work to understand why C++ forces you to do things
    2. Adhere to the style even if not completely obvious: ask questions, discuss
    3. Play games: minimum amount of code to check for debugging :-)
    4. Analyse and rewrite your own work: more understanding leads to better code
    5. Try porting or regularly use multiple compilers
    6. Do not tolerate warning messages: they are really errors!

Enforcing Consistent Style

Writing Software In OpenFOAM Style

- OpenFOAM library tools are strictly object-oriented; top-level codes are more in functional style, unless implementation is wrapped into model libraries
- OpenFOAM uses ALL features of C++ to the maximum benefit: you will need to learn it. Also, the code is an example of **good C++**: study and understand it

Enforcing Consistent Style

- Source code style in OpenFOAM is remarkably consistent:
  - Code separation into files
  - Comment and indentation style
  - Approach to common problems, e.g. I/O, construction of objects, stream support, handling function parameters, const and non-const access
  - Blank lines, no trailing whitespace, no spaces around brackets
- **Using file stubs**: `foamNew` script
  - `foamNew H exampleClass`: new header file
  - `foamNew C exampleClass`: new implementation file
  - `foamNew I exampleClass`: new inline function file
  - `foamNew IO exampleClass`: new IO section file
  - `foamNew App exampleClass`: new application file

Debugging OpenFOAM

Build and Debug Libraries

- Release build optimised for speed of execution; Debug build provides additional run-time checking and detailed trace-back capability
- Using trace-back on failure
  - `gdb icoFoam`: start debugger on `icoFoam` executable
  - `r <root> <case>`: perform the run from the debugger
  - `where` provides full trace-back with function names, file and line numbers
  - Similar tricks for debugging parallel runs: attach gdb to a running process
- Debug switches
  - Each set of classes or class hierarchy provides own debug stream
  - ...but complete flow of messages would be overwhelming!
  - Choosing debug message source: $HOME/.OpenFOAM-1.4/controlDict

OpenFOAM Environment

Environment Variables and Porting

- Software was developed on multiple platforms and ported regularly: better quality and adherence to standard
- Switching environment must be made easy: source single dot-file
- All tools, compiler versions and paths can be controlled with environment variables
- **Environment variables**
  - Environment setting support one installation on multiple machines
  - User environment: $HOME/.OpenFOAM-1.4/cshrc. Copied from OpenFOAM installation for user adjustment
  - OpenFOAM tools: OpenFOAM-1.4/.cshrc
    - Standard layout, *e.g.* FOAM_SRC, FOAM_RUN
    - Compiler and library settings, communications library *etc.*
- Additional setting
  - `FOAM_ABORT`: behaviour on abort
  - `FOAM_SIGFPE`: handling floating point exceptions
  - `FOAM_SETNAN`: set all memory to invalid on initialisation

Summary

FSB

OpenFOAM Programming

- OpenFOAM is a good and complete example of use of object orientation and C++
- Code layout designed for multiple users sharing a central installation and developing tools in local workspace
- Consistent style and some programming guidelines available through file stubs: `foamNew` script for new code layout
- Most (good) development starts from existing code and extends its capabilities
- Porting and multiple platform support handled through environment variables