OpenFOAM programming tutorial

Tommaso Lucchini

Department of Energy
Politecnico di Milano

Outline

- Overview of the OpenFOAM structure
- A look at icoFoam
- Customizing an application
- Implementing a transport equation in a new application
- Customizing a boundary condition
- General information

Structure of OpenFOAM

The OpenFOAM code is structures as follows (type `foam` and then `ls`).

- applications: source files of all the executables:
  - solvers
  - utilities
  - bin
  - test
- bin: basic executable scripts.
- doc: pdf and Doxygen documentation.
  - Doxygen
  - Guides-a4
- lib: compiled libraries.
- src: source library files.
- test: library test source files.
- tutorials: tutorial cases.
- wmake: compiler settings.

Structure of OpenFOAM

Navigating the source code

- Some useful commands to navigate inside the OpenFOAM sources:
  - *app* = `$WM_PROJECT_DIR/applications`
  - *sol* = `$WM_PROJECT_DIR/applications/solvers`
  - *util* = `$WM_PROJECT_DIR/applications/utilities`
  - *src* = `$WM_PROJECT_DIR/src`

- Environment variables:
  - `$FOAM_APP = $WM_PROJECT_DIR/applications`
  - `$FOAM_SOLVERS = $WM_PROJECT_DIR/applications/solvers`
  - `$FOAM_UTILITIES = $WM_PROJECT_DIR/applications/utilities`
  - `$FOAM_SRC = $WM_PROJECT_DIR/src`

- OpenFOAM source code serves two 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

Walk through a simple solver

Solver walk-through: `icoFoam`

- 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 the executable
    - `createFields.H`

Walk through icoFoam

File organization

`sol` → `cd incompressible` → `cd icoFoam`

- The `icoFoam` directory consists of what follows (type `ls`):

  `createFields.H  FoamX/  icoFoam.C  icoFoam.dep  Make/`
- The `FoamX` directory is for pre-processing.
- The `Make` directory contains instructions for the `wmake` compilation command.
- `icoFoam.C` is the main file, while `createFields.H` is included by `icoFoam.C`.
- The file `fvCFD.H`,included by `icoFoam.C`, contains all the class definitions which are needed by `icoFoam`. See the file `Make/options` to understand where `fvCFD.H` is included from:
  - `$FOAM_SRC/finiteVolume/lnInclude/fvCFD.H`, symbolic link to: `$FOAM_SRC/finiteVolume/cfdTools/general/include/fvCFD.H`
- Use the command `find PATH -iname "*LETTERSINFILENAME*"` to find where in `PATH` a file name containing `LETTERSFILENAME` in its file name is located.

  Example: `find $WM_PROJECT_DIR -iname "*fvCFD.H*"`

Walk through icoFoam

A look into fvCFD.H

```
#ifndef fvCFD_H
#define fvCFD_H

#include "parRun.H"

#include "Time.H"
#include "fvMesh.H"
#include "fvc.H"
#include "fvMatrices.H"
#include "fvm.H"
#include "linear.H"
#include "calculatedFvPatchFields.H"
#include "fixedValueFvPatchFields.H"
#include "adjustPhi.H"
#include "findRefCell.H"
#include "mathematicalConstants.H"
```

```
#include "OSspecific.H"
#include "argList.H"

#ifndef namespaceFoam
#define namespaceFoam
    using namespace Foam;
#endif

#endif
```

The inclusion files before `main` are all the class definitions required by `icoFoam`. Have a look into the source files to understand what these classes do.

Walk through icoFoam

A look into icoFoam.C, case setup and variable initialization

- `icoFoam` starts with

  ```
  int main(int argc, char *argv[])
  ```

  where `int argc` and `char *argv[]` are the number of parameters and the actual parameters used when running `icoFoam`.

- The case is initialized by:

  ```
  #   include "setRootCase.H"

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

  where all the included files except `createFields.H` are in `$FOAM_SRC/finiteVolume/lnInclude`.

- `createFields.H` is located in the `icoFoam` directory. It initializes all the variables used in `icoFoam`. Have a look inside it and see how variables are created.

Walk through icoFoam

A look into icoFoam.C, time-loop code

- The time-loop starts by:
  `for (runTime++; !runTime.end(); runTime++)`
  and the rest is done at each time-step

- The `fvSolution` subdictionary `PISO` is read, and the Courant Number is calculated and written to the screen by (use the `find` command):

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

- The momentum equations are defined and a velocity predictor is solved by:

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

        solve(UEqn == -fvc::grad(p));
```

Walk through icoFoam

A look into icoFoam.C, the PISO loop

- A PISO corrector loop is initialized by:

  `for (int corr=0; corr<nCorr; corr++)`

- The PISO algorithm uses these member functions:
  - `A()` returns the central coefficients of an `fvVectorMatrix`
  - `H()` returns the H operation source of an `fvVectorMatrix`
  - `Sf()` returns cell face area vector of an `fvMesh`
  - `flux()` returns the face flux field from an `fvScalarMatrix`
  - `correctBoundaryConditions()` corrects the boundary fields of a `volVectorField`
- Identify the object types (classes) and use the OpenFOAM Doxygen (`http://foam.sourceforge.net/doc/Doxygen/html`) to better understand them what they do

Walk through icoFoam

A look into icoFoam.C, write statements

- At the end of `icoFoam` there are some write statements

```
runTime.write();

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

- `write()` makes sure that all the variables that were defined as an `IOobject` with `IOobject::AUTO_WRITE` are written to the time directory according to the settings in the `$FOAM_CASE/system/controlDict` file.
- `elapsedCPUTime()` is the elapsed CPU time.
- `elapsedClockTime()` is the elapsed wall clock time.

OpenFOAM work space

General information

- 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
- 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 workspace:
  - **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**: `~/OpenFOAM/tommaso-1.5-dev/`. Contains applications, libraries and personal library and executable space.

Creating your OpenFOAM 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

myIcoFoam

Creating the new application directory, setting up Make/files, compiling

- The applications are located in `$WM_PROJECT_DIR/applications`
  - `cd $WM_PROJECT_DIR/applications/solvers/incompressible`
- Copy the `icoFoam` solver and put it in the `$WM_PROJECT_USER_DIR/applications` directory
  - `cp -r icoFoam $WM_PROJECT_DIR/applications`
- Rename the directory and the source file name, clean all the dependancies and
  - `mv icoFoam myIcoFoam`
  - `cd icoFoam`
  - `mv icoFoam.C myIcoFoam.C`
  - `wclean`
- Go the the `Make` directory and change `files` as follows:

```
myIcoFoam.C
EXE = $(FOAM_USER_APPBIN)/myIcoFoam
```

- Now compile the application with `wmake` in the `myIcoFoam` directory. `rehash` if necessary.

Creating your OpenFOAM applications

Example:

- Creating the application `icoScalarTransportFoam`. It is an incompressible solver with a scalar transport equation (species mass fraction, temperature, `...`).
- To do this, we need to create a new application based on the `icoFoam` code.

icoScalarTransportFoam

Creating the new application directory, setting up Make/files

- The applications are located in $WM_PROJECT_DIR/applications
  - cd $WM_PROJECT_DIR/applications/solvers/incompressible
- Copy the icoFoam solver and put it in the $WM_PROJECT_USER_DIR/applications directory
  - cp -r icoFoam $WM_PROJECT_DIR/applications
- Rename the directory and the source file name, clean all the dependancies and
  - mv icoFoam icoScalarTransportFoam
  - cd icoFoam
  - mv icoFoam.C icoScalarTransporFoam.C
  - wclean
- Go the the Make directory and change files as follows:

```
icoScalarTransportFoam.C
EXE = $(FOAM_USER_APPBIN)/icoScalarTransportFoam
```

`icoScalarTransportFoam`

Physical/numerical model modeling

- We want to solve the following transport equation for the scalar field T
- It is an unsteady, convection-diffusion transport equation. ν is the kinematic viscosity.

$$\frac{\partial T}{\partial t} + \nabla \cdot (\mathbf{U}T) - \nabla \cdot (\nu \nabla T) = 0 \tag{1}$$

- What to do:
  - Create the geometric field T in the `createFields.H` file
  - Solve the transport equation for T in the `icoScalarTransportFoam.C` file.

icoScalarTransportFoam

Creating the field T

- Modify `createFields.H` adding this `volScalarField` constructor before `#include "createPhi.H"`:

```
Info<< "Reading field T\n" << endl;
volScalarField T
(
    IOobject
    (
        "T",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
```

icoScalarTransportFoam

Creating the field T

- We have created a `volScalarField` object called `T`.
- `T` is created by reading a file (`IOobject::MUST_READ`) called `T` in the `runTime.timeName()` directory. At the beginning of the simulation, `runTime.timename()` is the `startTime` value specified in the `controlDict` file.
- `T` will be automatically written (`IOobject::AUTO_WRITE`) in the `runTime.timeName()` directory according to what is specified in the `controlDict` file of the case.
- `T` is defined on the computational mesh (`mesh` object):
  - It has as many internal values (`internalField`) as the number of mesh cells
  - It needs as many boundary conditions (`boundaryField`) as the mesh boundaries specified in the `constant/polyMesh/boundary` file of the case.

icoScalarTransportFoam

Solving the transport equation for T

- Create a new empty file, `TEqn.H`:
  - `echo > TEqn.H`
- Include it in `icoScalarTransportFoam.C` at the beginning of the PISO loop:

```
        for (int corr=0; corr<nCorr; corr++)
        {

#           include "TEqn.H"

            volScalarField rUA = 1.0/UEqn.A();
```

- Now we will implement the scalar transport equation for T in `icoScalarTransportFoam`...

icoScalarTransportFoam

Solving the transport equation for T

- This the transport equation:

$$\frac{\partial T}{\partial t} + \nabla \cdot (\mathbf{U}T) - \nabla \cdot (\nu \nabla T) = 0$$

- This is how we implement and solve it in `TEqn.H`

```
solve
(
    fvm::ddt(T)
  + fvm::div(phi, T)
  - fvm::laplacian(nu, T)
);
```

- Now compile the application with `wmake` in the `icoScalarTransportFoam` directory. `rehash` if necessary.

icoScalarTransportFoam

icoScalarTransportFoam: setting up the case

- Copy the `cavity` tutorial case in your `$FOAM_RUN` directory and rename it
  - `cp -r $FOAM_TUTORIALS/icoFoam/cavity $FOAM_RUN`
  - `mv cavity cavityScalarTransport`
- Introduce the field `T` in `cavityScalarTransport/0` directory:
  - `cp p T`

icoScalarTransportFoam

Running the application - case setup - startTime

- Modify T as follows:

```
dimensions      [0 0 0 0 0 0 0];
internalField   uniform 0;
boundaryField
{
    movingWall
    {
        type            fixedValue;
        value           uniform 1;
    }
    fixedWalls
    {
        type            fixedValue;
        value           uniform 0;
    }
    frontAndBack
    {
        type            empty;
    }
}
```

icoScalarTransportFoam

Running the application - case setup - system/fvSchemes

- Modify the subdictionary `divSchemes`, introducing the discretization scheme for `div(phi,T)`

```
divSchemes
{
    default         none;
    div(phi,U)      Gauss linear;
    div(phi,T)      Gauss linear;
}
```

- Modify the subdictionary `laplacianSchemes`, introducing the discretization scheme for `laplacian(nu,T)`

```
laplacianSchemes
{
    default         none;
    laplacian(nu,U) Gauss linear corrected;
    laplacian((1|A(U)),p) Gauss linear corrected;
    laplacian(nu,T) Gauss linear corrected;
}
```

icoScalarTransportFoam

Running the application - case setup - system/fvSolution

- Introduce the settings for T in the solvers subdictionary

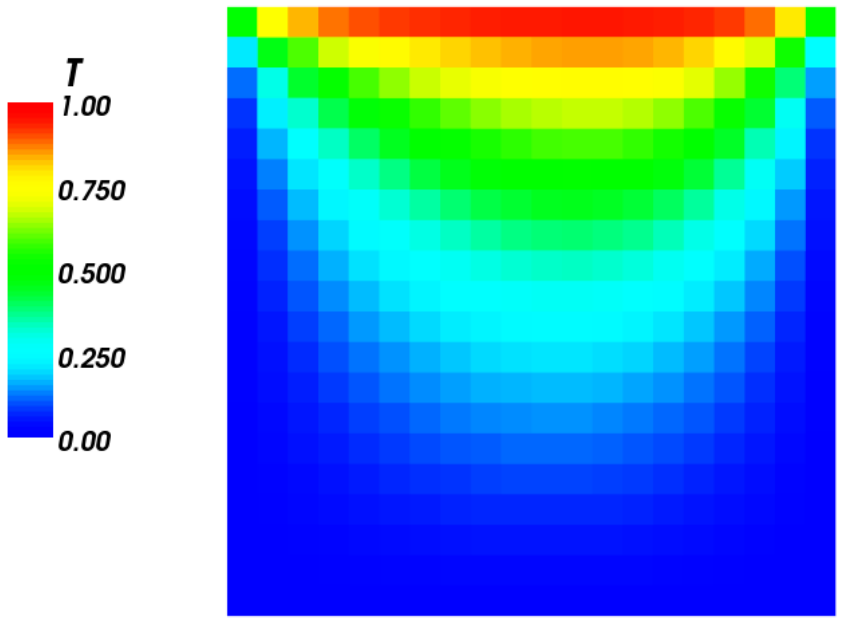
```
T PBiCG
{
    preconditioner
    {
        type         DILU;
    }

    minIter          0;
    maxIter          500;
    tolerance        1e-05;
    relTol           0;
};
```

icoScalarTransportFoam

icoScalarTransportFoam: post-processing

- Run the case
  - `icoScalarTransportFoam -case cavityScalarTranport`
- Nice picture:



Implementing a new boundary condition

General information

Run-Time Selection Table Functionality

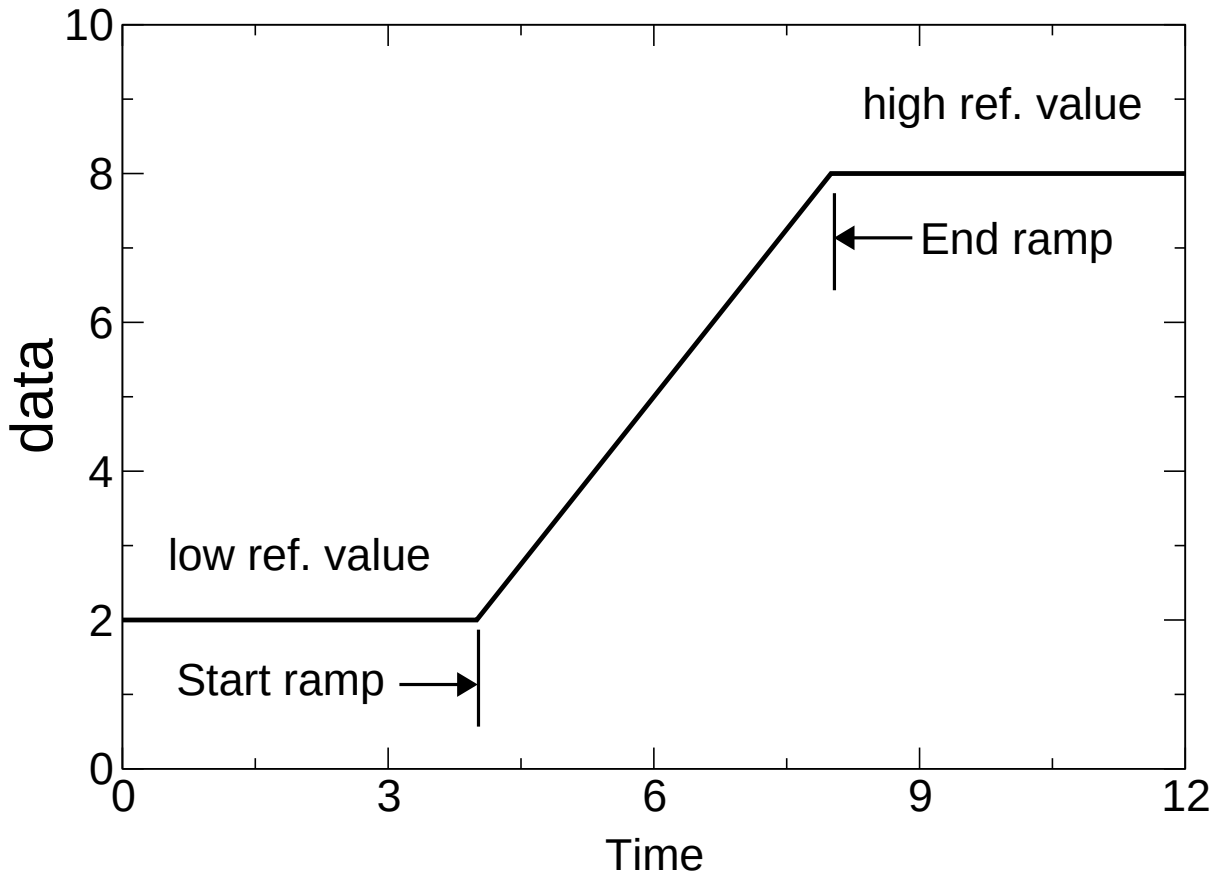
- In many cases, OpenFOAM provides functionality selectable at run-time which needs to be changed for the purpose. Example: viscosity model; ramped fixed value boundary conditions
- New functionality should be run-time selectable (like implemented models)
- . . . but should not interfere with existing 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

Implementing a new boundary condition

What `rampedFixedValue` should do



Implementing a new boundary condition

In a new application `icoFoamRamped`

- `cp $FOAM_SOLVERS/compressible/icoFoam \`
  `$FOAM_USER_DIR/applications/icoFoamRamped`
- Copy the content of `$FOAM_SRC/fields/fvPatchFields/derived/oscillatingFixedValue/` to `$WM_PROJECT_USER_DIR/applications/icoFoamRamped/`
- Change the file names

```
mv oscillatingFixedValueFvPatchField.C     rampedFixedValueFvPatchField.C
mv oscillatingFixedValueFvPatchField.H     rampedFixedValueFvPatchField.H
mv oscillatingFixedValueFvPatchFields.C    rampedFixedValueFvPatchFields.C
mv oscillatingFixedValueFvPatchFields.H    rampedFixedValueFvPatchFields.H
```

- `wclean`

Implementing a new boundary condition

`rampedFixedValueFvPatchField.H`

- Template class, contains the class definition for the generic objects.
- Replace the string `oscillating` with the string `ramped` (use the *replace* function of any text editor with the *case sensitive* option. This has the following effects:
  - The new class begins with

    ```
    #ifndef rampedFixedValueFvPatchField_H
    #define rampedFixedValueFvPatchField_H
    ```
  - Class declaration

    ```
    template<class Type>
    class rampedFixedValueFvPatchField
    ```
  - Objects we need:
    - Reference value low bound → `Field<Type> refValueLow_;`
    - Reference value high bound → `Field<Type> refValueHigh_;`
    - Ramp start time → `scalar startRamp_;`
    - Ramp end time → `scalar endRamp_;`
    - Current time index → `label curTimeIndex_;`

Implementing a new boundary condition

`rampedFixedValueFvPatchField.H`

- All the constructors

```
//- Construct from patch and internal field
rampedFixedValueFvPatchField
(
    const fvPatch&,
    const DimensionedField<Type, volMesh>&
);
// other constructors
        //- Construct from patch, internal field and dictionary
        //- Construct by mapping given rampedFixedValueFvPatchField
        //  onto a new patch
        //- Construct as copy
        //- Construct and return a clone
        //- Construct as copy setting internal field reference
        //- Construct and return a clone setting internal field reference
```

- Private member function to evaluate the boundary condition: `currentScale()`
- Provide member functions to access them (const/non const)

```
    //- Return the ref value
    Field<Type>& refValueHigh()
    {
        return refValueHigh_;
    }
```

Implementing a new boundary condition

`rampedFixedValueFvPatchField.H`

- Other member functions:
  - Mapping

```
//- Map (and resize as needed) from self given a mapping object
virtual void autoMap
(
    const fvPatchFieldMapper&
);

//- Reverse map the given fvPatchField onto this fvPatchField
virtual void rmap
(
    const fvPatchField<Type>&,
    const labelList&
);
```

  - Evaluation of the boundary condition

```
virtual void updateCoeffs();
```

  - Write to file:

```
virtual void write(Ostream&) const;
```

Implementing a new boundary condition

`rampedFixedValueFvPatchField.C`

- Contains the class implementation:
  - Constructors
  - Private member functions:
    - Access (if not defined in the `.H` file)
    - Map
    - Evaluation
    - Write

Implementing a new boundary condition

`rampedFixedValueFvPatchField.C` - Constructors

```
template<class Type>
rampedFixedValueFvPatchField<Type>::rampedFixedValueFvPatchField
(
    const fvPatch& p,
    const Field<Type>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchField<Type>(p, iF),
    refValueLow_("refValueLow", dict, p.size()),
    refValueHigh_("refValueHigh", dict, p.size()),
    startRamp_(readScalar(dict.lookup("startRamp"))),
    endRamp_(readScalar(dict.lookup("endRamp"))),
    curTimeIndex_(-1)
{
    Info << "Hello from ramp! startRamp: " << startRamp_
        << " endRamp: " << endRamp_ << endl;

    if (dict.found("value"))
    {
        fixedValueFvPatchField<Type>::operator==
        (
            Field<Type>("value", dict, p.size())
        );
    }
    else
    {
        fixedValueFvPatchField<Type>::operator==
        (
            refValueLow_ + (refValueHigh_ - refValueLow_)*currentScale()
        );
    }
}
```

Implementing a new boundary condition

`rampedFixedValueFvPatchField.C` - Private member function

- currentScale() is used to evaluate the boundary condition. It is the ramp fraction at time `t`:

```
template<class Type>
scalar rampedFixedValueFvPatchField<Type>::currentScale() const
{
    return
        min
        (
            1.0,
            max
            (
                (this->db().time().value() - startRamp_)/
                (endRamp_ - startRamp_),
                0.0
            )
        );
}
```

Implementing a new boundary condition

`rampedFixedValueFvPatchField.C - updateCoeffs()`

- updateCoeffs(): evaluates the boundary conditions

```
// Update the coefficients associated with the patch field
template<class Type>
void rampedFixedValueFvPatchField<Type>::updateCoeffs()
{
    if (this->updated())
    {
        return;
    }

    if (curTimeIndex_ != this->db().time().timeIndex())
    {
        Field<Type>& patchField = *this;

        patchField =
            refValueLow_
          + (refValueHigh_ - refValueLow_)*currentScale();

        curTimeIndex_ = this->db().time().timeIndex();
    }

    fixedValueFvPatchField<Type>::updateCoeffs();
}
```

Implementing a new boundary condition

`rampedFixedValueFvPatchField.C - write(Ostream& os)`

- This function writes to a file `os` the boundary condition values. Useful when the simulation is restarted from the latest time.

```
template<class Type>
void rampedFixedValueFvPatchField<Type>::write(Ostream& os) const
{
    fvPatchField<Type>::write(os);
    refValueLow_.writeEntry("refValueLow", os);
    refValueHigh_.writeEntry("refValueHigh", os);
    os.writeKeyword("startRamp")
        << startRamp_ << token::END_STATEMENT << nl;
    os.writeKeyword("endRamp")
        << endRamp_ << token::END_STATEMENT << nl;
    this->writeEntry("value", os);
}
```

Implementing a new boundary condition

`rampedFixedValueFvPatchFields.H`

- The generic `rampedFixedValueFvPatchField<Type>` class becomes specific for `scalar`, `vector`, `tensor`, ... by using the command:

  `makePatchTypeFieldTypedefs(rampedFixedValue)`

- This function is defined in `$FOAM_SRC/finiteVolume/fvPatchField.H` and it uses `typedef` for this purpose:

```
typedef rampedFixedValueFvPatchField<scalar> rampedFixedValueFvPatchScalarField;
typedef rampedFixedValueFvPatchField<vector> rampedFixedValueFvPatchVectorField;
typedef rampedFixedValueFvPatchField<tensor> rampedFixedValueFvPatchTensorField;
```

Implementing a new boundary condition

`rampedFixedValueFvPatchFields.C`

- It adds to the `runTimeSelectionTable` the new boundary conditions created in `rampedFixedValueFvPatchFields.H`, by calling the function:
  `makePatchFields(rampedFixedValue);`

- In this way, the new boundary condition can be used for `volScalarField`, `volVectorField`, `volTensorField`, ... just typing in the field file:

```
boundaryField // example for a volScalarField
{
    // some patches
    // ....
    inlet
    {
        type            rampedFixedValue;
        refValueLow     uniform 10;
        refValueHigh    uniform 20;
        startRamp       20;
        endRamp         50;
    }
}
```

Implementing a new boundary condition

In the solver, modification of `Make/files`

- The `Make/files` should be modified as follows:

```
icoFoamRamped.C
rampedFixedValueFvPatchFields.C

EXE = $(FOAM_USER_APPBIN)/icoFoamRamped
```

- `wmake`

- In this way, the new boundary condition can be *only* used by the `icoFoamRamped` application.

Implementing a new boundary condition

In a dynamic library

- If all the user-defined boundary conditions were put in a library, they will be available to all the solvers
- Create in the `$WM_PROJECT_USER_DIR` the directory `myBCs`
- Copy in that directory all the `rampedFixedValue*` files
- Create the `Make` directory, with two empty files inside: `files` and `options`
  - `Make/files`

```
rampedFixedValueFvPatchFields.C

LIB = $(FOAM_USER_LIBBIN)/libMyBCs
```

  - `Make/options`

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

EXE_LIBS = \
    -lfiniteVolume
```

  - Compile the library in the `$WM_PROJECT_USER_DIR/myBCs` with the command

```
wmake libso
```

Implementing a new boundary condition

In a dynamic library, to be used by the solvers

- The boundary condition will not be recognized by any of the original OpenFOAM solvers unless we tell OpenFOAM that the library exists. In OpenFOAM-1.5 this is done by adding a line in the `system/controlDict` file:

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

  i.e. the library must be added for each case that will use it, but no re-compilation is needed for any solver. `libMyBCs.so` is found using the `LD_LIBRARY_PATH` environment variable, and if you followed the instructions on how to set up OpenFOAM and compile the boundary condition this should work automatically.

- You can now set up the case as we did earlier and run it using the original `icoFoam` solver. `icoFoam` does not need to be recompiled, since `libMyBCs.so` is linked at run-time using dlopen.

- Example. Solve the cavity tutorial with the user defined library of boundary conditions.

Implementing a turbulence model

General information

- Creating a new turbulence model (based on the $k-\varepsilon$ model) that can be used by all the existing OpenFOAM applications.
- A user library, called `myTurbulenceModels` will be created. It will be included run-time as for the ramped fixed value boundary condition.
- The turbulence model will be tested on the pitzDaily tutorial case of the `simpleFoam` application.
- A `RASModel` object is created in the `createFields.H` file of the `simpleFoam` application:

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

- At the end of the PISO Loop, the function `turbulence->correct()` will be called. This function solves the transport equation of the turbulence fields ($k, \varepsilon, \omega, \ldots$) and updates the turbulence viscosity field (`turbulence->muEff()`).

Implementing a turbulence model

A short look to the `incompressible/RASModel` library

- Type `cd $FOAM_SRC/turbulenceModels/` and then type `ls`:
  `LES RAS`
- The `RAS/incompressible` directory contains:
  - Boundary conditions for k and ε fields at the inlet located in the `derivedFvPatchFields` directory
  - Different turbulence models that can be used by incompressible RANS solvers (`kEpsilon, kOmega, laminar, LaunderSharmaKE...`)
  - Implementation of the wall functions (`wallFunctions`)
- Create a new directory called `myTurbulenceModels` located in `~/OpenFOAM/root-1.5-dev/applications`
- Copy the `kEpsilon` model directory into `~/OpenFOAM/root-1.5-dev/applications/myTurbulenceModels`
- Rename it `mykEpsilon`. Rename the files in the directory:

```
mv kEpsilon.H mykEpsilon.H
mv kEpsilon.C mykEpsilon.C
```

- Replace the word `kEpsilon` with `mykEpsilon` in the `.C` and `.H` files.

Implementing a turbulence model

A look to `mykEpsilon.H`

- The class `mykEpsilon` is derived from the `incompressible/RASModel` class.
- Class members:
  - Model constants (C_μ,C_1, C_2, α_ϵ).
  - Fields: k, ε, ν_t (turbulence viscosity).
  - Typename to be run-time selectable.
- Constructors, destructors
- Class functions
  - Access: reference to the class members.
  - Fields: effective diffusivity for k, effective diffusivity for ε, Reynolds stress tensor, source term for the momentum equation
  - Edit: the `correct()` function solves the transport equations for k and ε| and updates the ν_t field accordingly.

Implementing a turbulence model

A look to `mykEpsilon::correct()`

```
void mykEpsilon::correct()
{
    transportModel_.correct();

    if (!turbulence_)
    {
        return;
    }

    RASModel::correct();

    volScalarField G = nut_*2*magSqr(symm(fvc::grad(U_)));

#   include "wallFunctionsI.H"
```

- The kinematic viscosity field is updated when the `transportModel_.correct()` function is called. Have a look to `$FOAM_SRC/transportModels/incompressible/viscosityModels` to see the transport models available for incompressible flows.
- The `correct()` function of the base class is called to update the `nearWallDist` field if the mesh changes (motion/topological change).
- The `G` field is updated and then the `wallFunctionsI.H` file updates the `G` and ε fields at the wall boundary cells.

Implementing a turbulence model

A look to `mykEpsilon::correct()`

```
    // Dissipation equation
    tmp<fvScalarMatrix> epsEqn
    (
        fvm::ddt(epsilon_)
      + fvm::div(phi_, epsilon_)
      + fvm::SuSp(-fvc::div(phi_), epsilon_)
      - fvm::laplacian(DepsilonEff(), epsilon_)
     ==
        C1_*G*epsilon_/k_
      - fvm::Sp(C2_*epsilon_/k_, epsilon_)
    );

    epsEqn().relax();

#   include "wallDissipationI.H"

    solve(epsEqn);
    bound(epsilon_, epsilon0_);
```

- The transport equation for ϵ is firstly constructed and then relaxed.
- In the `wallDissipationI.H` file the boundary values of the ε field are forced to the ones calculated in the `wallFunctions.H` file previously.
- The equation is then solved and, eventually, bounded.

Implementing a turbulence model

A look to `mykEpsilon::correct()`

```
    // Turbulent kinetic energy equation
    tmp<fvScalarMatrix> kEqn
    (
        fvm::ddt(k_)
      + fvm::div(phi_, k_)
      - fvm::Sp(fvc::div(phi_), k_)
      - fvm::laplacian(DkEff(), k_)
     ==
        G
      - fvm::Sp(epsilon_/k_, k_)
    );

    kEqn().relax();
    solve(kEqn);
    bound(k_, k0_);


    // Re-calculate viscosity
    nut_ = Cmu_*sqr(k_)/epsilon_;

#   include "wallViscosityI.H"
```

- The same procedure (equation definition, relax, solving and bounding) is also used fo the k field.
- The turbulent viscosity is updated.
- And finally `wallViscosityI.H` calculates the turbulence viscosity at the wall boundary cells.

Implementing a turbulence model

Library implementation

- The `Make` directory contains:
  - `files`:

```
mykEpsilon/mykEpsilon.C

LIB = $(FOAM_USER_LIBBIN)/libmyTurbulenceModels
```

  - `options`:

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/transportModels \
    -I$(LIB_SRC)/turbulenceModels/RAS/incompressible/lnInclude \
    -I$(LIB_SRC)/transportModels/incompressible/lnInclude

LIB_LIBS = \
    -lfiniteVolume \
    -lmeshTools \
    -lincompressibleRASModels \
    -lincompressibleTransportModels
```

Implementing a turbulence model

Running the case with the `mykEpsilon` model

- Add the following lines after the constructor of the `mykEpsilon` turbulence model:

```
Info << "hello mykEpsilon!!!!!!!" << endl;
```

- Copy the `pitzDaily` tutorial case to your run directory and rename it as `pitzDailyMykEpsilon`.

```
cp -r $FOAM_TUTORIALS/simpleFoam/pitzDaily pitzDailyMykEpsilon
```

- Modify the `pitzDailyMykEpsilon/constant/RASProperties`
- Specify in the `constant/RASProperties` file of the case that the `mykEpsilon` turbulence model must be used:

```
RASModel            mykEpsilon;
```

- Rename the sub-dictionary called `kEpsilonCoeffs` to `mykEpsilonCoeffs`
- Add the following line to the `system/controlDict` file of the case:

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

- Run the case...

Some 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/OpenFOAM-1.5/etc/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/OpenFOAM-1.5/etc/cshrc`. Copied from OpenFOAM installation for user adjustment
    - OpenFOAM tools: `OpenFOAM-1.5-dev/settings.sh`; `OpenFOAM-1.5-dev/aliases.sh`
    - 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

OpenFOAM environment

- 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