Short Course on OpenFOAM® development

ENIEF 2014

Juan Marcelo Gimenez [1]
Axel Larreteguy [2]
Santiago Márquez Damián [1,3]
Norberto Nigro [1,3]

[1]Centro de Investigaciones en Mecánica Computacional (CIMEC)
UNL/CONICET, Predio Conicet Litoral Centro, Santa Fe, Argentina

[2]MySLab, Instituto de Tecnología, Universidad Argentina de la Empresa

[3]Facultad de Ingeniería y Ciencias Hídricas
Universidad Nacional del Litoral

Instituto Balseiro - Bariloche, Argentina - September 2014

Disclaimer

This offering is not approved or endorsed by ESI, the producer of the OpenFOAM®software and owner of the OpenFOAM®trade mark.

Outline

1. Object Oriented Programming: Its use in OpenFOAM®
   - Basics on OOP
   - An overview of OpenFOAM®class Diagram
2. Programming Solvers
3. Turbulence Model Implementation
4. Boundary Condition Implementation
5. Adding a control system to an application

Outline

Outline

1. Object Oriented Programming: Its use in OpenFOAM®
   - Basics on OOP
   - An overview of OpenFOAM® class Diagram
2. Programming Solvers
3. Turbulence Model Implementation
4. Boundary Condition Implementation
5. Adding a control system to an application

Outline

Outline

Outline

1. Object Oriented Programming: Its use in OpenFOAM®
   - Basics on OOP
   - An overview of OpenFOAM® class Diagram
2. Programming Solvers
3. Turbulence Model Implementation
4. Boundary Condition Implementation
5. Adding a control system to an application

OpenFOAM®

This section is based on Håkan Nilsson course.

- OpenFOAM® is a C++ library, used primarily to create executables, known as applications. The applications fall into two categories:
  - `solvers`, that are designed to solve a specific continuum mechanics problem. Example: `icoFoam`.
  - `utilities`, that are designed to perform tasks that involve data manipulation. Example: `blockMesh`.
- Special applications for pre- and post-processing are included in OpenFOAM®. Converters to/from other pre- and post-processors are available.
- OpenFOAM® is distributed with a large number of applications, but soon any advanced user will start developing new applications or specific codes for his/ her special needs.
- Programming Code Style: http://www.openfoam.org/contrib/code-style.php

Outline

1. Object Oriented Programming: Its use in OpenFOAM®
   - Basics on OOP
   - An overview of OpenFOAM® class Diagram
2. Programming Solvers
3. Turbulence Model Implementation
4. Boundary Condition Implementation
5. Adding a control system to an application

C++ types, classes and objects

- The *types* (`int`, `double`) can be seen as *classes*, and the variables we assign to a *type* are *objects* of that class (`int a;`)
- Object orientation focuses on the *objects* instead of the functions.
- An *object* belongs to a *class* of objects with the same attributes. The class defines:
  - The construction of the object
  - The destruction of the object
  - Attributes of the object (member data)
  - Methods which manipulate the object (member functions)
- I.e. it is the `int` class that defines how the operator + should work for objects of that class, and how to convert between classes if needed (e.g. $1 + 1.0$ involves a conversion).

C++ class definition

- General description of the structure to define the class with name `myClass` and its public and private member functions and member data.

```
class myClass {
  public:
    //declarations of public member functions and data
  private:
    //declaration of hidden member functions and data
};
```

- `public` attributes are visible from outside the class.
- `private` attributes are only visible within the class.
- If neither public nor private are specified, all attributes will be `private`.
- Declarations of attributes and methods are done just as functions and variables are declared outside a class.
- A standard practise is to encapsulate the attributes as `private` data and set and get them through the class interface provided by methods.

C++ class usage

- Example Class definition:

```
class myClass {
  private:
    int a;
  public:
    inline void set(int a){this->a = a;};
    int get();
};
inline int myClass::get(){return a;};
```

- Example usage: objects, pointers and references.

```
myClass myObj; //object declaration
myObj.set(10);
myClass* p = &myObj; //pointer (memory address)
myClass& r = &myObj; //reference (alias)

cout<<"a: "<<p->a<<endl; // not allowed!!
cout<<"a: "<<p->get()<<endl; // allowed!!
```

C++ organization of classes

- A good programming standard is to organize the class files in pairs, the other one with the class declarations (`.H`), and one with the class definitions (`.C`).
- The class *declaration* file must be included in the files where the class is used, i.e. the class definition file and files that inherits from, or construct objects of that class.
- The compiled *definition* file is statically or dynamically linked to the executable by the compiler.
- Inline functions must be implemented in the class *declaration* file, since they must be inlined without looking at the class *definition* file.

C++ Constructors

- A *constructor* is a special method that is called each time a new object of that class is instanciated.
- Example: `Vector` class from OpenFOAM®

```
// Constructors
//- Construct null (default)
inline Vector();
//- Construct by copy
inline Vector(const Vector<Cmpt>&);
//- Construct given three components
inline Vector(const Cmpt& vx, const Cmpt& vy, const Cmpt& vz);
//- Construct from Istream
inline Vector(Istream&);
```

- The `Vector` will be initialized differently depending on which of these constructors is chosen
- Also there is a *destructor* method: `~Vector()...`

C++ operators

Example:

```
Vector<scalar> a(1.0,2.0,0.0);
Vector<scalar> b(-1.0,1.0,1.0);
a+=b;
Info<<a<<endl;
```

Why work += and << ? Operator overloading

```
Vector<Cmpt>& Vector<Cmpt>::operator+=(Vector<Cmpt>& v){
  this->v_[0]+=v.x();
  this->v_[1]+=v.y();
  this->v_[2]+=v.z();
  return this;
};
...
Ostream& Vector<Cmpt>::operator<<(Vector<Cmpt>& v){ ... }
```

(this implementation differs from real OpenFOAM® implementation)

C++ inheritance

- A class can inherit members from already existing classes extending their funcionality with new members.
- Syntax, when defining the new class:

```
class newClass : public oldClass { ...members... }
```

- `newClass` is now a sub-class to `oldClass`.
- Members of a class can be `public`, `private` or `protected`.
  - `private` members are never visible in a sub-class, while `public` and `protected` are. However, `protected` are only visible in a sub-class (not in other classes).
  - The visibility of the inherited members can be modified in the new class. It is only possible to make the members of a base-class less visible in the sub-class.
  - To combine features, a class may be a sub-class to several base-classes (multiple inheritance).

C++ other features

- C++ `virtual` functions: should be implemented by sub-classes
- C++ `abstract` classes: they have at least one pure virtual method.

  (see $FOAM_SRC/turbulenceModels/incompressible/LES/LESModel/LESModel.H)

  ```
  virtual tmp<volScalarField> nuSgs() const = 0;
  ```

- C++ `templates`:

  ```
  template<class T> Vector{ ... } //declaration
  ...
  Vector<scalar> V; //usage
  ```

- C++ `typedef`:

  ```
  typedef Vector<int> integerVector; //declaration
  ...
  integerVector iV; //usage
  ```

- C++ `namespace`:

  ```
  namespace Foam{ ... declarations ... } //declaration
  ...
  Foam::member() //usage
  ```

Outline

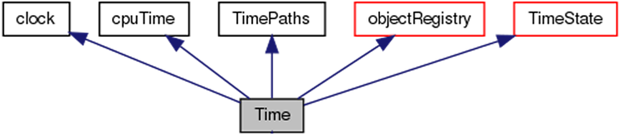
1. Object Oriented Programming: Its use in OpenFOAM®
   - Basics on OOP
   - An overview of OpenFOAM® class Diagram
2. Programming Solvers
3. Turbulence Model Implementation
4. Boundary Condition Implementation
5. Adding a control system to an application

Outline

This section is based on the H. Jasak presentation

- Space and time: polyMesh, fvMesh, Time
- Field algebra: Field, DimensionedField and GeometricField
- Boundary conditions: fvPatchField and derived classes
- Sparse matrices: lduMatrix, fvMatrix and linear solvers
- Finite Volume discretisation: fvc and fvm namespace

Representation of Time



- Class `Time` manages simulation in terms of time-steps: start and end time, delta t
  - `deltaT()`: Return time step
  - `name()`: Return current directory name
  - `operator++()`, `operator+=(scalar)`: Time increments.
  - `write()`: Write to disk the objects.
  - `startTime()`,`endTime()`.
- Time is associated with IO functionality: what and when to write
- User main simulation control through a dictionary: `controlDict` file

Space and time

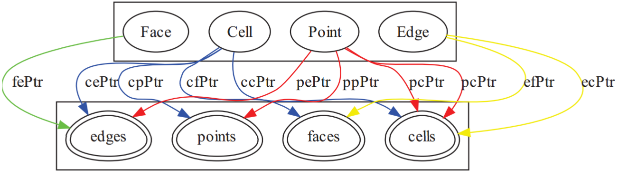
objectRegistry: all IOobjects, including mesh, fields and dictionaries registered in the class `Time`

```
volScalarField p
(
    IOobject
    (
        "p",
        runTime.timeName(),  //directory name
        mesh,
        IOobject::MUST_READ, //read controls
        IOobject::AUTO_WRITE //write controls
    ),
    mesh
);
```

Space and time

Representation of Space

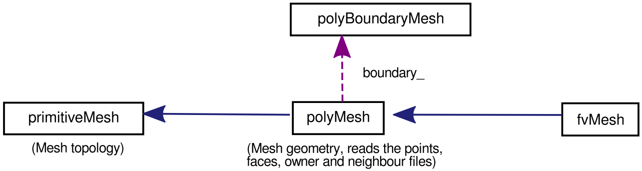
- Computational mesh consists of
  - List of points. Point index is determined from its position in the list
  - List of faces. A face is an ordered list of points (defines face normal)
  - List of cells OR owner-neighbour addressing
  - List of boundary patches, grouping external faces
- Main classes:
  - `primitiveMesh`: Cell-face mesh analysis engine (Figure from Passalacqua, Pal Singh, 2008):



  - `polyMesh`: Mesh consisting of general polyhedral cells: reads points, faces, owner and neighbor files.
  - `polyBoundaryMesh` is a list of polyPatches. It is an attribute of `polyMesh`.

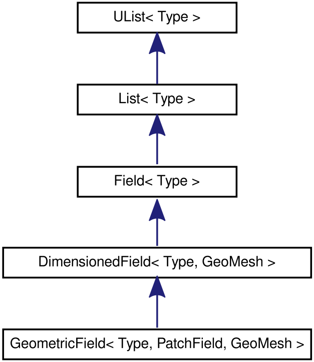
Finite Volume Method

- `polyMesh` class provides mesh data in generic manner: it is used by multiple applications and discretisation methods
- For convenience, each discretisation wraps up primitive mesh functionality to suit its needs: mesh metrics, addressing etc.
- fvMesh: Mesh data needed to do the Finite Volume discretisation
  - `C()`, `V()`, `Sf()`, `magSf()`, `Cf()`: geometrical information
  - `movePoints()`, `updateMesh()`: designed for dynamic meshes



Field Classes: Containers with Algebra

- `UList`: unallocated array pointer and access
  `begin()`, `end()`, `operator[]`
- `List`: allocation + resizing
  `size()`, `resize()`, `clear()`, `append()`
- `Field`: algebra overloaded for scalar, vector, tensor
  `operator+=()`, `operator-=()`, `operator*=()`, `operator/=()`, `T()`
- `DimensionedField`: I/O, dimension set, name, mesh reference
  `dimensions()`, `mesh()`, `operatorXX()` with dimensions check
- `GeometricField`: internal field, boundary conditions, old time
  `internalField()`, `boundaryField()`



Field Classes: Containers with Algebra

- `Field`
  - Simply, a list with algebra, templated on element type
  - Assign unary and binary operators from the element, mapping functionality etc
- `DimensionedField`
  - A field associated with a mesh, with a name and mesh reference
  - Derived from IOobject for input-output and database registration
- `GeometricField`
  - Consists of an internal field (derivation) and a `GeometricBoundaryField`
  - Boundary field is a field of fields or boundary patches
  - Geometric field can be defined on several mesh entities and element types:
    `volScalarField`, `volVectorField`, `surfaceScalarField`, `surfaceTensorField`, `volSymmTensorField`, `tensorAverageIOField`

Finite Volume Boundary Conditions

- Implementation of boundary conditions is a perfect example of a virtual class hierarchy
- Consider the implementation of a boundary condition
  - Evaluate function: calculate new boundary values depending on behaviour: fixed value, zero gradient etc.
  - Enforce boundary type constraint based on matrix coefficients
  - Virtual function interface: run-time polymorphic dispatch
- Base class: `fvPatchField`
  - Derived from a field container
  - Reference to fvPatch: easy data access
  - Reference to internal field
- Types of `fvPatchField`:
  - Basic: fixed value, zero gradient, mixed, coupled, default
  - Constraint: enforced on all fields by the patch: cyclic, empty, processor, symmetry, wedge, GGI
  - Derived: wrapping basic type for physics functionality

GeometricBoundaryField: It has a list of fvPatchFields.

- GeometricField calls its GeometricBoundaryField object

```
correctBoundaryConditions()
{
  this->setUpToDate();
  storeOldTimes();
  boundaryField_.evaluate(); //method of its attribute
}
```

- A loop over each fvPatchField is done

```
evaluate(){
  forAll(*this, patchi)
  {
    this->operator[](patchi).evaluate();
  }
}
```

Sparse Matrix and Solver

Sparse Matrix Class

- Addressing classes:
  - `lduAddressing`: matrix profile
    `upperAddr()`, `lowerAddr()`
  - `lduInterface`: treatment of special B.C: cyclic, parallel and so on.
  - `lduMesh`, `lduPrimitiveMesh`: `lduAddressing`+`lduInterface`
- `lduMatrix`: matrix coefficients (values)
  `upper()`, `lower()`, `diag()`
- Animated .gif

Sparse Matrix and Solver

- Finite Volume matrix class: `fvMatrix`
- Derived from `lduMatrix`, with a reference to the solution field and to the *rhs* vector. It looks like an equation system class: `residual()`, `source()`, `relax()`, `solve()`
- Solver technology: preconditioner, smoother, solver → out of scope
- Matrices are currently always scalar: segregated solver for vector and tensor variables
- It allows matrix assembly at equation level: adding and subtracting matrices
- *Non-standard* matrix functionality in `fvMatrix`:
  - `A()`: return matrix diagonal in FV field form
  - `H()`: vector-matrix multiply with current psi(), using off-diagonal coefficients and rhs
  - `flux()`: consistent evaluation of off-diagonal product in face form.

Finite Volume Discretisation

- Finite Volume Method implemented in 3 parts
  - Surface interpolation: cell-to-face data transfer
  - Finite Volume Calculus (fvc): given a field, create a new field
  - Finite Volume Method (fvm): create a matrix representation of an operator, using FV discretisation
- In both cases, we have static functions with no common data. Thus, `fvc` and `fvm` are implemented as namespaces
- Discretisation involves a number of choices on how to perform identical operations: eg. gradient operator. In all cases, the signature is common

```
volTensorField gradU = fvc::grad(U);
```

- Multiple algorithmic choices of gradient calculation operator: Gauss theorem, least squares fit, limiters etc. implemented to use run-time selection.
- Choice of discretisation controlled by the user on a per-operator basis: `system/fvSschemes`

Dispatch run-Time selection

$$\vec{\nabla}\psi = \frac{1}{V_P}\sum_f \psi_f \vec{S}_f$$

Example Code

```
...
volScalarField p;
surfaceScalarField phi;

//call 1 -> fvcGrad.C -> gaussGrad.C
fvc::grad(phi);

//call 2 -> fvcGrad.C -> gradScheme.C -> gaussGrad.C
fvc::grad(p);
...
```

Call I in fvcGrad.C

```
template<class Type>
tmp
<
    GeometricField
    <
        typename outerProduct<vector, Type>::type, fvPatchField, volMesh
    >
>
grad
(
    const GeometricField<Type, fvsPatchField, surfaceMesh>& ssf
)
{
    return fv::gaussGrad<Type>::gradf(ssf, "grad(" + ssf.name() + ')');
}
```

Call I in gaussGrad.C

```
...
Foam::fv::gaussGrad<Type>::gradf
(
    const GeometricField<Type, fvsPatchField, surfaceMesh>& ssf,
    const word& name
)
{
...
    forAll(owner, facei)
    {
        GradType Sfssf = Sf[facei]*ssf[facei];
        igGrad[owner[facei]] += Sfssf;
        igGrad[neighbour[facei]] -= Sfssf;
    }
    forAll(mesh.boundary(), patchi)
    {
        const labelUList& pFaceCells =
            mesh.boundary()[patchi].faceCells();
        const vectorField& pSf = mesh.Sf().boundaryField()[patchi];
        const fvsPatchField<Type>& pssf = ssf.boundaryField()[patchi];
        forAll(mesh.boundary()[patchi], facei)
        {
            igGrad[pFaceCells[facei]] += pSf[facei]*pssf[facei];
        }
    }
    igGrad /= mesh.V();
    gGrad.correctBoundaryConditions();
    return tgGrad;
}
```

Call II

- fvcGrad.C

```
...
    return fv::gradScheme<Type>::New
    (
        vf.mesh(),
        vf.mesh().gradScheme(name)
    )().grad(vf, name);
```

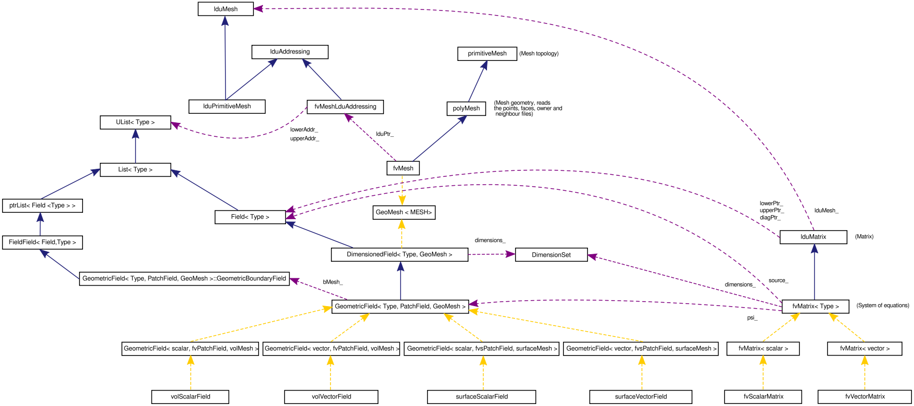
- gradScheme.C

```
    typename IstreamConstructorTable::iterator cstrIter =
        IstreamConstructorTablePtr_->find(schemeName);
...
         return calcGrad(vsf, name);
```

- gaussGrad.C

```
    tmp<GeometricField<GradType, fvPatchField, volMesh> > tgGrad
    (
        gradf(tinterpScheme_().interpolate(vsf), name)
    );
```

Brief Class Diagram



Outline

1. Object Oriented Programming: Its use in OpenFOAM®
   - Basics on OOP
   - An overview of OpenFOAM®class Diagram
2. Programming Solvers
3. Turbulence Model Implementation
4. Boundary Condition Implementation
5. Adding a control system to an application

Compiling OpenFOAM® on debug mode

- If the instalation of OpenFOAM® is in a system directory, login as root (`$ sudo su`)
- Edit the file `$WM_PROJECT_DIR/etc/bashrc` and modify the enviroment variable `WM_COMPILE_OPTION` setting the line:

```
export WM_COMPILE_OPTION=Debug
```

- Reload enviroment variables (or open a new terminal instance)

```
$ . $WM_PROJECT_DIR/etc/bashrc
```

- Recompile

```
$ cd $WM_PROJECT_DIR
$ ./Allwmake
```

A new solver from an existing one

- Create the host folder imitating the tree directory of OpenFOAM®

```
mkdir $WM_PROJECT_USER_DIR/applications/solvers
```

- Copy the most similar solver

```
$ cp -r \
$WM_PROJECT_DIR/applications/solvers/incompressible/pisoFoam \
$WM_PROJECT_USER_DIR/applications/solvers/myPisoFoam
```

- Rename and edit some files

```
$ cd $WM_PROJECT_USER_DIR/applications/solvers/myPisoFoam
$ mv pisoFoam.C myPisoFoam.C
$ sed -i s/pisoFoam/myPisoFoam/g myPisoFoam.C
$ sed -i s/pisoFoam/myPisoFoam/g Make/Files
$ sed -i s/FOAM_APPBIN/FOAM_USER_APPBIN/g Make/Files
```

Using QTcreator

- To use the IDE QTcreator follow the guidelines in http://openfoamwiki.net/index.php/HowTo_Use_OpenFOAM_with_QtCreator
- Once done, we will able to
  - Add new proyects to the IDE (applications, solvers, tests)
  - Code-autocompletion
  - Navigate throught files (definitions, declarations)
  - Compile applications and solvers
  - Execute tests
  - User-friendly debugging

gdbOF

`gdbOF` is a tool attachable to the GNU debugger (`gdb`) that includes macros to debug OpenFOAM®solvers and applications in an easier way.

- Download, installation and user-manual from: http://openfoamwiki.net/index.php/Contrib_gdbOF
- In QTcreator activate:
  `Windows -> Views -> Debugger log`
- Once at breakpoint, write the gdbOF command in the corresponding box.

myPisoFoam

- Starting from the turbulent incompressible flow solver `pisoFoam`, which solves ...

$$\nabla \cdot \mathbf{u} = 0$$

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u}\mathbf{u}) = -\nabla p + \nabla \cdot (\nu \nabla \mathbf{u})$$

- ... and a passive scalar transport equation is added

$$\frac{\partial T}{\partial t} + \nabla \cdot (\mathbf{u} T) = \nabla \cdot (\alpha \nabla T)$$

- where the viscosity and diffusivity are related by $Pr = \nu/\alpha$.

myInterFoam

- Starting from the solver for 2 incompressible fluids using Volume of Fluid (VoF) `interFoam`, which solves ...

$$\nabla \cdot \mathbf{u} = 0$$

$$\frac{\partial \rho \mathbf{u}}{\partial t} + \nabla \cdot (\rho \mathbf{u}\mathbf{u}) = -\nabla p + \nabla \cdot \left(\mu \nabla \mathbf{u} + \nabla^T \mathbf{u}\right) + \rho g + \sigma \kappa \nabla \alpha$$

$$\frac{\partial \alpha}{\partial t} + \nabla \cdot (\alpha \mathbf{u}) = -\nabla \cdot [\alpha (1 - \alpha) \mathbf{v}]$$

- ... where $0 <= \alpha <= 1$ is the volume fraction (a sharp function)
- A widely used strategy is coupling VoF and LevelSet to smooth α and to improve the surface tension forces calculation.
- Level Set function ϕ properties
  - $\phi = 0$ at interface
  - $|\nabla \phi| = 1$
  - $\mathbf{n} = \nabla \phi$
  - $\kappa = -\nabla \cdot \mathbf{n}$

Following [Albadawi et al.,2013]

- Initial value of Level Set function

$$\phi^0 = (2\alpha - 1)\Gamma$$

- Reinitialization equation

$$\frac{\partial \phi^{n+1}}{\partial \tau} = S(\phi^0)(1 - |\nabla \phi^n|)$$

- Volumetric surface tension force

$$\mathbf{F}_\sigma = \sigma \kappa(\phi)\delta(\phi)\nabla\phi$$

- Heaviside smoothed Function (for physical properties)

$$H(\phi) = \begin{cases} 0 & \phi < -\epsilon \\ \frac{1}{2}\left[1 + \frac{\phi}{\epsilon} + \frac{1}{\pi}\sin\frac{\pi\phi}{\epsilon}\right] & |\phi| < \epsilon \\ 1 & \phi > \epsilon \end{cases}$$

Implementation based on the work of Takuya Yamamoto

Outline

1. Object Oriented Programming: Its use in OpenFOAM®
   - Basics on OOP
   - An overview of OpenFOAM® class Diagram
2. Programming Solvers
3. Turbulence Model Implementation
4. Boundary Condition Implementation
5. Adding a control system to an application

Run-time type selection (RTS) mechanism

With the aid of Bruno Santos and Tomislav Maric (Turbulence models and run time selection tables).

Run-time type selection is a mechanism that allows the user of a program to select different variable (object) types at a point of program execution. This mechanism is heavily used in OpenFOAM®to enable the user a high level of flexibility in choosing:

- boundary conditions,
- discretization schemes,
- models (viscosity, turbulence, two-phase property),
- function objects,

and similar elements that build up a numerical simulation –all at run-time. The type selection mechanism can be considered as a black box, that takes the input parameter, and returns a constructed object of an appropriate class (type).

Run-time type selection (RTS) mechanism

Other options to this mechanism could be:

- hard-coding,
- using multiple decision structures (f.e. `switch--case` in C++),

	Advantages	Disadvantages
Hard-coding	Speed	Not-flexible
Decision structures	Speed	Partially flexible
RTS	Highly flexible	Complex

- RTS is based on static variables and methods and the extensive use of pre-processor macros,
- the contructors of the selected objects are taken from predefined tables which are loaded at startup

RTS principal macros

Class names

`defineTypeNameAndDebug(Type, DebugSwitch) (className.H)`

```
//- Define the typeName and debug information
#define defineTypeNameAndDebug(Type, DebugSwitch)                     \
    defineTypeName(Type);                                             \
    defineDebugSwitch(Type, DebugSwitch)

//- Define the typeName
#define defineTypeName(Type)                                          \
    defineTypeNameWithName(Type, Type::typeName_())

//- Define the typeName, with alternative lookup as \a Name
#define defineTypeNameWithName(Type, Name)                            \
    const ::Foam::word Type::typeName(Name)
```

RTS principal macros

Run time selection tables creation. Allocation of table pointer.

`defineRunTimeSelectionTable(baseType,argNames)`
`(runTimeSelectionTables.H)`

```
// external use:
// ~~~~~~~~~~~~~
// define run-time selection table
#define defineRunTimeSelectionTable\
(baseType,argNames)                                                  \
                                                                     \
    defineRunTimeSelectionTablePtr(baseType,argNames);               \
    defineRunTimeSelectionTableConstructor(baseType,argNames);       \
    defineRunTimeSelectionTableDestructor(baseType,argNames)
```

```
// internal use:
// create pointer to hash-table of functions
#define defineRunTimeSelectionTablePtr\
(baseType,argNames)                                                  \
                                                                     \
    /* Define the constructor function table */                      \
    baseType::argNames##ConstructorTable*                            \
        baseType::argNames##ConstructorTablePtr_ = NULL
```

RTS principal macros

Run time selection tables creation. Creation of empty table.

`defineRunTimeSelectionTable(baseType,argNames)` `(runTimeSelectionTables.H)`

```
// internal use:
// constructor aid
#define defineRunTimeSelectionTableConstructor\
(baseType,argNames)                                                     \
                                                                        \
    /* Table constructor called from the table add function */          \
    void baseType::construct##argNames##ConstructorTables()             \
    {                                                                   \
        static bool constructed = false;                                \
        if (!constructed)                                               \
        {                                                               \
            constructed = true;                                         \
            baseType::argNames##ConstructorTablePtr_                    \
                = new baseType::argNames##ConstructorTable;             \
        }                                                               \
    }
```

RTS principal macros

Run time selection tables use. Direct calling of constructors, add of new entries.

`declareRunTimeSelectionTable(autoPtr,baseType,argNames ,argList,parList) (runTimeSelectionTables.H)`

```
    // declareRunTimeSelectionTable is used to create a run-time selection table
    // for a base-class which holds constructor pointers on the table.

// external use:
// ~~~~~~~~~~~~~
// declare a run-time selection:
#define declareRunTimeSelectionTable\
(autoPtr,baseType,argNames,argList,parList)                                    \
                                                                               \
    /* Construct from argList function pointer type */                         \
    typedef autoPtr< baseType > (*argNames##ConstructorPtr)argList;            \
                                                                               \
    /* Construct from argList function table type */                           \
    typedef HashTable< argNames##ConstructorPtr, word, string::hash >          \
        argNames##ConstructorTable;                                            \
                                                                               \
    /* Construct from argList function pointer table pointer */                \
    static argNames##ConstructorTable* argNames##ConstructorTablePtr_;         \
                                                                               \
    /* Table constructor called from the table add function */                 \
    static void construct##argNames##ConstructorTables();                      \
                                                                               \
    /* Table destructor called from the table add function destructor */      \
    static void destroy##argNames##ConstructorTables();                        \
```

RTS principal macros

```
    /* Class to add constructor from argList to table */         \
    template< class baseType##Type >                              \
    class add##argNames##ConstructorToTable                       \
    {                                                             \
    public:                                                       \
                                                                  \
        static autoPtr< baseType > New argList                    \
        {                                                         \
            return autoPtr< baseType >(new baseType##Type parList); \
        }                                                         \
                                                                  \
        add##argNames##ConstructorToTable                         \
        (                                                         \
            const word& lookup = baseType##Type::typeName         \
        )                                                         \
        {                                                         \
            construct##argNames##ConstructorTables();             \
            if (!argNames##ConstructorTablePtr_->insert(lookup, New)) \
            {                                                     \
                std::cerr<< "Duplicate entry " << lookup          \
                    << " in runtime selection table " << #baseType \
                    << std::endl;                                 \
                error::safePrintStack(std::cerr);                 \
            }                                                     \
        }                                                         \
                                                                  \
        ~add##argNames##ConstructorToTable()                      \
        {                                                         \
            destroy##argNames##ConstructorTables();               \
        }                                                         \
    };                                                            \
```

RTS principal macros

Run time selection tables use. Calling of “New” methods, add of new entries.

`declareRunTimeNewSelectionTable(autoPtr,baseType, argNames,argList,parList) (runTimeSelectionTables.H)`

```
    declareRunTimeNewSelectionTable is used to create a run-time selection
    table for a derived-class which holds "New" pointers on the table.

// external use:
// ~~~~~~~~~~~~~
// declare a run-time selection for derived classes:
#define declareRunTimeNewSelectionTable\
(autoPtr,baseType,argNames,argList,parList)                                  \
                                                                             \
    /* Construct from argList function pointer type */                       \
    typedef autoPtr< baseType > (*argNames##ConstructorPtr)argList;          \
                                                                             \
    /* Construct from argList function table type */                         \
    typedef HashTable< argNames##ConstructorPtr, word, string::hash >        \
        argNames##ConstructorTable;                                          \
                                                                             \
    /* Construct from argList function pointer table pointer */              \
    static argNames##ConstructorTable* argNames##ConstructorTablePtr_;       \
                                                                             \
    /* Table constructor called from the table add function */               \
    static void construct##argNames##ConstructorTables();                    \
```

RTS principal macros

```
    /* Table destructor called from the table add function destructor */   \
    static void destroy##argNames##ConstructorTables();                     \
                                                                            \
    /* Class to add constructor from argList to table */                    \
    template< class baseType##Type >                                        \
    class add##argNames##ConstructorToTable                                 \
    {                                                                       \
    public:                                                                 \
                                                                            \
        static autoPtr< baseType > New##baseType argList                    \
        {                                                                   \
            return autoPtr< baseType >(baseType##Type::New parList.ptr());  \
        }                                                                   \
                                                                            \
        add##argNames##ConstructorToTable                                   \
        (                                                                   \
            const word& lookup = baseType##Type::typeName                   \
        )                                                                   \
        {                                                                   \
            construct##argNames##ConstructorTables();                       \
            if                                                              \
            (                                                               \
               !argNames##ConstructorTablePtr_->insert                      \
                (                                                           \
                    lookup,                                                 \
                    New##baseType                                           \
                )                                                           \
            )                                                               \
```

RTS principal macros

Run time selection tables use.

Adding new entries. The actual adding is performed when the `add##thisType##argNames##ConstructorTo##baseType##Table_` object is instantiated via its constructor.

`addToRunTimeSelectionTable(baseType,thisType,argNames)` `(addToRunTimeSelectionTable.H)`

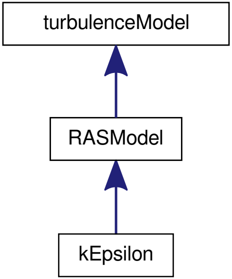
```
//    Macros for easy insertion into run-time selection tables

// add to hash-table of functions with typename as the key
#define addToRunTimeSelectionTable\
(baseType,thisType,argNames)                                          \
                                                                      \
    /* Add the thisType constructor function to the table */          \
    baseType::add##argNames##ConstructorToTable< thisType >           \
        add##thisType##argNames##ConstructorTo##baseType##Table_
```

RTS in turbulence models

The whole mechanism is based on two tables (RAS case):

- Turbulence models = 3(`LESModel, RASModel, laminar`)
- RAS models = 18(`LRR,LamBremhorstKE,LaunderGibsonRSTM, LaunderSharmaKE,LienCubicKE,LienCubicKELowRe, LienLeschzinerLowRe,NonlinearKEShih,RNGkEpsilon, SpalartAllmaras,kEpsilon,kOmega,kOmegaSST, kkLOmega,laminar,qZeta,realizableKE,v2f`)

where their objects have the following hierarchy:



RTS in turbulence models

Definition of Turbulence Models table via static data (creation of empty table) at startup

`defineRunTimeSelectionTable(turbulenceModel, turbulenceModel);` (`turbulenceModel.C`)

```
// * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //

defineTypeNameAndDebug(turbulenceModel, 0);
defineRunTimeSelectionTable(turbulenceModel, turbulenceModel);
```

Definition of RAS Models table via static data (creation of empty table) at startup. Adding of `RASModel` to Turbulence Models table.

`defineRunTimeSelectionTable(turbulenceModel, turbulenceModel);`
`addToRunTimeSelectionTable(turbulenceModel, RASModel, turbulenceModel);`
(RASModel.C)

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

defineTypeNameAndDebug(RASModel, 0);
defineRunTimeSelectionTable(RASModel, dictionary);
addToRunTimeSelectionTable(turbulenceModel, RASModel, turbulenceModel);
```

RTS in turbulence models

Finally each RAS turbulence models adds itself to RAS Models table, f.e. `kEpsilon`

`addToRunTimeSelectionTable(RASModel, kEpsilon, dictionary);` `(kEpsilon.C)`

```
// * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //

defineTypeNameAndDebug(kEpsilon, 0);
addToRunTimeSelectionTable(RASModel, kEpsilon, dictionary);
```

RTS in action, selecting `kEpsilon` in `pimpleFoam`

The selection of turbulence model requires to set first the `constant/turbulenceProperties` user dictionary to select the main type of turbulence model.

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

simulationType RASModel;

// ************************************************************************* //
```

Then, the `constant/RASProperties` user dictionary is set to select the particular RAS turbulence model

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

RASModel        kEpsilon;

turbulence      on;

printCoeffs     on;


// ************************************************************************* //
```

RTS in action, selecting `kEpsilon` in `pimpleFoam`

Once `pimpleFoam` is executed and reaches the following line of the corresponding `createFields.H`:

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

Here the `turbulence` auto pointer is set in order to invoke the methods required from the turbulence model such as `turbulence->correct()` to solve the turbulence models equations or `turbulence->divDevReff(U)` to evaluate the viscous terms of momentum equations.

The `turbulenceModel::New(U, phi, laminarTransport)` method is a selector which reads from `constant/turbulenceProperties` dictionary and searches in the Turbulence Models table, if the required main turbulence models exists continues the calling chain, if not, shows an error by the `stdout`.

RTS in action, selecting kEpsilon in pimpleFoam

turbulenceModel.C

```
autoPtr<turbulenceModel> turbulenceModel::New
(
    const volVectorField& U,
    const surfaceScalarField& phi,
    transportModel& transport,
    const word& turbulenceModelName
)
{
    // get model name, but do not register the dictionary
    // otherwise it is registered in the database twice
    const word modelType
    (
        IOdictionary
        (
            IOobject
            (
                "turbulenceProperties",
                U.time().constant(),
                U.db(),
                IOobject::MUST_READ_IF_MODIFIED,
                IOobject::NO_WRITE,
                false
            )
        ).lookup("simulationType")
     );

     Info<< "Selecting turbulence model type " << modelType << endl;

     turbulenceModelConstructorTable::iterator cstrIter =
         turbulenceModelConstructorTablePtr_->find(modelType);
```

RTS in action, selecting `kEpsilon` in `pimpleFoam`

```
    if (cstrIter == turbulenceModelConstructorTablePtr_->end())
    {
        FatalErrorIn
        (
            "turbulenceModel::New(const volVectorField&, "
            "const surfaceScalarField&, transportModel&, const word&)"
        )   << "Unknown turbulenceModel type "
            << modelType << nl << nl
            << "Valid turbulenceModel types:" << endl
            << turbulenceModelConstructorTablePtr_->sortedToc()
            << exit(FatalError);
    }

    return autoPtr<turbulenceModel>
    (
        cstrIter()(U, phi, transport, turbulenceModelName)
    );
}
```

The `cstrIter()` has a pointer to a `New` method of `RASModel` class as was defined using the `declareRunTimeNewSelectionTable` macro included in `turbulenceModel.H`.

RTS in action, selecting `kEpsilon` in `pimpleFoam`

RASModel.C

```
autoPtr<RASModel> RASModel::New
(
    const volVectorField& U,
    const surfaceScalarField& phi,
    transportModel& transport,
    const word& turbulenceModelName
)
{
    // get model name, but do not register the dictionary
    // otherwise it is registered in the database twice
    const word modelType
    (
        IOdictionary
        (
            IOobject
            (
                "RASProperties",
                U.time().constant(),
                U.db(),
                IOobject::MUST_READ_IF_MODIFIED,
                IOobject::NO_WRITE,
                false
            )
        ).lookup("RASModel")
    );

    Info<< "Selecting RAS turbulence model " << modelType << endl;

    dictionaryConstructorTable::iterator cstrIter =
        dictionaryConstructorTablePtr_->find(modelType);
```

RTS in action, selecting `kEpsilon` in `pimpleFoam`

```
    if (cstrIter == dictionaryConstructorTablePtr_->end())
    {
        FatalErrorIn
        (
            "RASModel::New"
            "("
                "const volVectorField&, "
                "const surfaceScalarField&, "
                "transportModel&, "
                "const word&"
            ")"
        )   << "Unknown RASModel type "
            << modelType << nl << nl
            << "Valid RASModel types:" << endl
            << dictionaryConstructorTablePtr_->sortedToc()
            << exit(FatalError);
    }

    return autoPtr<RASModel>
    (
        cstrIter()(U, phi, transport, turbulenceModelName)
    );
}
```

The `cstrIter()` has a pointer to a true constructor for `kEpsilon` class as was defined using the `declareRunTimeSelectionTable` macro included in `RASModel.H`.

RTS in action, selecting `kEpsilon` in `pimpleFoam`

Once the `kEpsilon` model object is instantiated the calling chain returns to `createFields.H` and the turbulence model is ready to use.

`UEqn.H`

```
// Solve the Momentum equation

tmp<fvVectorMatrix> UEqn
(
    fvm::ddt(U)
  + fvm::div(phi, U)
  + turbulence->divDevReff(U)
 ==
    fvOptions(U)
);
```

`pimpleFoam.C`

```
// --- Pressure-velocity PIMPLE corrector loop
while (pimple.loop())
{
    #include "UEqn.H"

    // --- Pressure corrector loop
    while (pimple.correct())
    {
        #include "pEqn.H"
    }

    if (pimple.turbCorr())
    {
        turbulence->correct();
    }
}
```

Turbulence model implementation study case

Modifications in `kEpsilon` model.

Key concepts:

- Reading parameters in the initialization list;
- implementation of the `correct` method;
- call the `addToRunTimeSelectionTable` macro to add the new turbulence model to the RTS table;
- set the `Make/files` and `Make/options` files.

Remember the `$WM_PROJECT_DIR` and `$WM_PROJECT_USER_DIR` environment variables.

Turbulence model implementation study case

Main steps.

- Copy the orginal `kEpsilon` implementation to the `$WM_PROJECT_USER_DIR`;

```
cp -r --parents src/turbulenceModels/incompressible/RAS/kEpsilon \
    $WM_PROJECT_USER_DIR
```

  (The original directory tree and naming is used.)

- rename directories;

```
cd $WM_PROJECT_USER_DIR/src/turbulenceModels/incompressible/RAS
mv kEpsilon mykEpsilon
```

Turbulence model implementation study case

- create Make/files and Make/options files;

```
mkdir Make
cd Make/
nano files

mykEpsilon.C
LIB = $(FOAM_USER_LIBBIN)/libmyIncompressibleRASModels

nano options

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

This method is intended for only one turbulence model within the library. The other way is to reproduce the `$WM_PROJECT_DIR/.../RAS/Make` directory and files, *add* the new model and generate a library for all the new turbulence models.

Turbulence model implementation study case

- edit filenames and code;

```
cd ..
rm kEpsilon.dep
mv kEpsilon.C mykEpsilon.C
mv kEpsilon.H mykEpsilon.H
sed -i s/kEpsilon/mykEpsilon/g mykEpsilon.C
sed -i s/kEpsilon/mykEpsilon/g mykEpsilon.H
```

- make simple changes in constructor;

```
Info << "Defining my own kEpsilon model" << endl;
```

- compile;

```
wmake libso
```

Turbulence model implementation study case

- run.

```
cd $FOAM_RUN
cp -r $FOAM_TUTORIALS/incompressible/pimpleFoam/pitzDaily
cd pitzDaily
nano system/controlDict

libs ("libmyIncompressibleRASModels.so");

nano constant/RASProperties

RASModel        mykEpsilon;

pimpleFoam
```

The message included in the turbulence model constructor will be displayed by the `stdout`.

For further details see: "How to implement a turbulence model" slides by Håkan Nilsson.

Boundary conditions from a FVM basis

Setting boundary conditions affects the matrix's coefficients and the source term of the linear system. For example, in case of fixed value derived BC's:

- Convective term:

$$\int_\Gamma \vec{v}\,\phi \cdot d\vec{\Gamma} = \sum_f \phi_f \left(\vec{v}_f \cdot \vec{S}_f \right)$$

The contribution to source term is given by the value $-\phi_b \vec{v}_f \cdot \vec{S}_f$ and the contribution to the matrix is zero.

Boundary conditions from a FVM basis

- Diffusive term:

$$\int_{\Gamma} \vec{\nabla} \cdot (\nu \vec{\nabla} \phi)\, d\Omega = \sum_{f} (\nu)_f (\vec{\nabla}\phi)_f \cdot \vec{S}_f$$

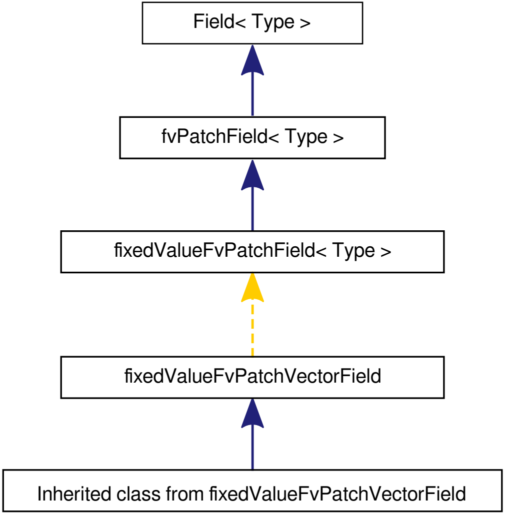
For the case of orthogonal meshes and `Gauss linear` laplacian scheme the contributions to the matrix and source term are given by:

$$(\nu)_f (\vec{\nabla}\phi)_b \cdot \vec{S}_f = (\nu)_f |\vec{S}_f| \frac{\phi_b - \phi_P}{|\vec{d}_n|}$$

with a contribution to the source term of $-(\nu)_f |\vec{S}_f| \frac{\phi_b}{|\vec{d}_n|}$ and to the matrix diagonal of $-(\nu)_f |\vec{S}_f| \frac{1}{|\vec{d}_n|}$.

The fixed value boundary conditions are implemented via `fixedValueFvPatchField` and related classes.

Inheritance diagram for `fixedValueFvPatchField` and related classes



Inheritance in `fixedValueFvPatchField` and related classes

- `fvPatchField< Type >`

  Abstract base class with a fat-interface to all derived classes covering all possible ways in which they might be used.

  –The first level of derivation is to basic patchFields which cover zero-gradient, fixed-gradient, fixed-value and mixed conditions.

  –The next level of derivation covers all the specialised types with specific evaluation proceedures, particularly with respect to specific fields.

Inheritance in `fixedValueFvPatchField` and related classes

- `fixedValueFvPatchField< Type >`

  This boundary condition supplies a fixed value constraint, and is the base class for a number of other boundary conditions.

  –The derived classes could not accept templatization, then a series of `typedef`'s is created automatically for scalar, vector, tensor type and so on. This is achieved by the `makePatchFields` macro

fixedValueFvPatchFields.C

```
// * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //

makePatchFields(fixedValue);

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

Inheritance in fixedValueFvPatchField and related classes

fvPatchField.H

```
#define makePatchFields(type)                                    \
    makeTemplatePatchTypeField                                   \
    (                                                            \
        fvPatchScalarField,                                      \
        type##FvPatchScalarField                                 \
    );                                                           \
    makeTemplatePatchTypeField                                   \
    (                                                            \
        fvPatchVectorField,                                      \
        type##FvPatchVectorField                                 \
    );                                                           \
    makeTemplatePatchTypeField                                   \
    (                                                            \
        fvPatchSphericalTensorField,                             \
        type##FvPatchSphericalTensorField                        \
    );                                                           \
    makeTemplatePatchTypeField                                   \
    (                                                            \
        fvPatchSymmTensorField,                                  \
        type##FvPatchSymmTensorField                             \
    );                                                           \
    makeTemplatePatchTypeField                                   \
    (                                                            \
        fvPatchTensorField,                                      \
        type##FvPatchTensorField                                 \
    );
```

Inheritance in `fixedValueFvPatchField` and related classes

- `fixedValueFvPatchVectorField`

  From this class other vectorial fixed value boundary condition can be inherited.

Induction to some methods required in boundary condition classes

- The case of `fvm::div(phi, U)` term.

  This term requires the evaluation of a divergence in weak form. The implementation for the Gauss case is presented in `gaussConvectionScheme.C` and returns a `tmp<fvMatrix<Type> >`

```
template<class Type>
tmp<fvMatrix<Type> >
gaussConvectionScheme<Type>::fvmDiv
(
    const surfaceScalarField& faceFlux,
    const GeometricField<Type, fvPatchField, volMesh>& vf
) const
{
    tmp<surfaceScalarField> tweights = tinterpScheme_().weights(vf);
    const surfaceScalarField& weights = tweights();

    tmp<fvMatrix<Type> > tfvm
    (
        new fvMatrix<Type>
        (
            vf,
            faceFlux.dimensions()*vf.dimensions()
        )
    );
```

Induction to some methods required in boundary condition classes

```
    fvMatrix<Type>& fvm = tfvm();

    fvm.lower() = -weights.internalField()*faceFlux.internalField();
    fvm.upper() = fvm.lower() + faceFlux.internalField();
    fvm.negSumDiag();

    forAll(vf.boundaryField(), patchI)
    {
        const fvPatchField<Type>& psf = vf.boundaryField()[patchI];
        const fvsPatchScalarField& patchFlux = faceFlux.boundaryField()[patchI];
        const fvsPatchScalarField& pw = weights.boundaryField()[patchI];

        fvm.internalCoeffs()[patchI] = patchFlux*psf.valueInternalCoeffs(pw);
        fvm.boundaryCoeffs()[patchI] = -patchFlux*psf.valueBoundaryCoeffs(pw);
    }

    if (tinterpScheme_().corrected())
    {
        fvm += fvc::surfaceIntegrate(faceFlux*tinterpScheme_().correction(vf));
    }

    return tfvm;
}
```

The contribution to the system matrix is stored in `fvm.internalCoeffs()` and the contribution to the source term is stored in `fvm.boundaryCoeffs()`. The `valueInternalCoeffs()` and `valueBoundaryCoeffs()` are required from the boundary condition classes.

Induction to some methods required in boundary condition classes

fixedValueFvPatchField.C

```
template<class Type>
tmp<Field<Type> > fixedValueFvPatchField<Type>::valueInternalCoeffs
(
    const tmp<scalarField>&
) const
{
    return tmp<Field<Type> >
    (
        new Field<Type>(this->size(), pTraits<Type>::zero)
    );
}


template<class Type>
tmp<Field<Type> > fixedValueFvPatchField<Type>::valueBoundaryCoeffs
(
    const tmp<scalarField>&
) const
{
    return *this;
}
```

As was expected the contribution to the matrix will be zero and the contribution to the source term returns `*this`. Since the `fvPatch` related classes inherit from `Field`, the `this` pointer points to the BC field values. These values are calculated following their definition equations.

Induction to some methods required in boundary condition classes

The fixed values on each face of the boundary patch are evaluated in the boundary condition class constructor by the `evaluate` method.

fvPatchField.C

```
template<class Type>
void Foam::fvPatchField<Type>::evaluate(const Pstream::commsTypes)
{
    if (!updated_)
    {
        updateCoeffs();
    }

    updated_ = false;
    manipulatedMatrix_ = false;
}
```

The `evaluate` execution requires to have implemented the `updateCoeffs` method. This method is boundary condition specific and the center of the efforts in new fixed value boundary conditions implementation.

Boundary condition implementation study case

Parabolic inlet boundary condition from `foam-extend-3.1` (`parabolicVelocityFvPatchVectorField`).

This boundary conditions fixes a parabolic velocity in a boundary with normal `n`, along direction `y` and with a peak velocity of `maxValue`.

Key concepts:

- Reading parameters in the initialization list;
- implementation of the `updateCoeffs` and `write` methods;
- call the `makePatchTypeField` macro to add the new boundary condition to the RTS table;
- set the `Make/files` and `Make/options` files.

`fvPatchField.H`

```
// for non-templated patch fields
#define makePatchTypeField(PatchTypeField, typePatchTypeField)              \
    defineTypeNameAndDebug(typePatchTypeField, 0);                          \
    addToPatchFieldRunTimeSelection(PatchTypeField, typePatchTypeField)
```

Boundary condition implementation study case

Compiling and using

- Compile the new boundary condition using `wmake libso`;
- set the parameter in the `0/U` field;

```
type        parabolicVelocity;
n           (1 0 0);
y           (0 1 0);
maxValue    1;
value       uniform (0 0 0);
```

- load the library declaring it in the `system/controlDict` file: `libs ("parabolicVelocity.so");`

For further details see: "How to implement a new boundary condition" slides by Håkan Nilsson.

Outline

Module: adding a control system to an application

- An excuse to learn how to:
  - modify fixedValue BCs during runtime, and
  - exchange information between processors.
- Base code: scalarTransportFoam
- Base case: pitzDaily

Activities

- Base case
  - Serial run
- Version 1: variable BC
  - Add code for modifying fixedValue BC
  - Serial run
- Version 2: control system (first try)
  - Add control system code
  - Create control system dictionaries
  - Serial run
  - Decompose and run in parallel
- Version 3: control system (revisited)
  - Add interprocess communication
  - Run in parallel

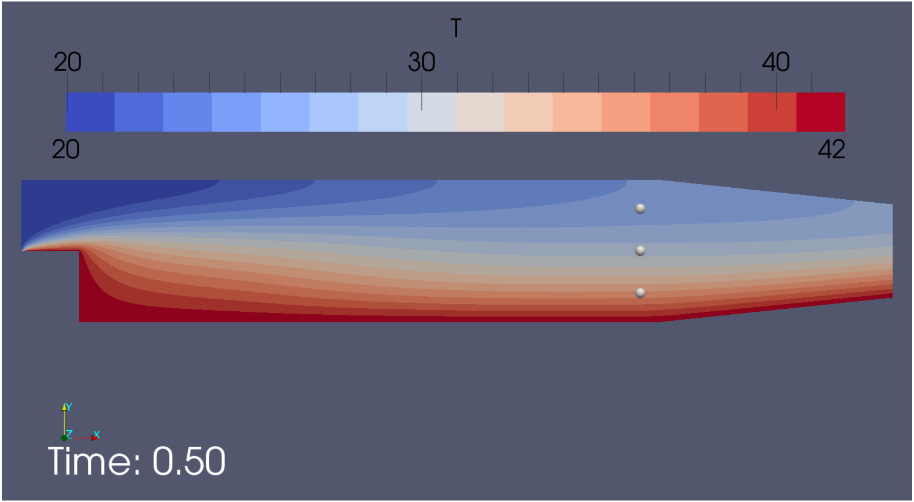
Base code: scalarTransportFoam

```
int main(int argc, char *argv[])
{
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "createFields.H"
//    #include "CScreateSensors.H"
    simpleControl simple(mesh);

    Info<< "\nCalculating scalar transport\n" << endl;

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

Base code: scalarTransportFoam

```
...
while (simple.loop())
    {
        Info<< "Time = " << runTime.timeName() << nl << endl;
// #include "CScontrolActions.H"
        while (simple.correctNonOrthogonal())
        {
            solve
            (
                fvm::ddt(T)
              + fvm::div(phi, T)
              - fvm::laplacian(DT, T)
            );
        }
        runTime.write();
    }
    Info<< "End\n" << endl;
    return 0;
}
```

Base case: pitzDaily (modified)



Version 1: variable BC

- #include "modifyBC.H" inside the time loop to linearly increase the BC value of patch "lowerWall"

```
//file: modifyBC.H
//Identify patchID
label patchID = mesh.boundaryMesh().findPatchID("lowerWall");
//Get pointer to the field data
fixedValueFvPatchScalarField& WallTemperature =
    refCast<fixedValueFvPatchScalarField>(T.boundaryField()[patchID]);
//Copy pointer
scalarField& TemperatureValue  = WallTemperature;
//for all faces in the patch, increase temperature  in 0.001C
forAll (TemperatureValue,i) {
    TemperatureValue[i] += 0.001;
}
```

- Compile and run

Version 2: control system (first try)

- Remove #include "modifyBC.H"
- Add #include "CScreateSensors.H" before the time loop
- Add #include "CScontrolActions.H" inside the time loop
- Compile
- Create dictionary "constant/probeLocations"
- Run serial
- Decompose and run in parallel

"CScreateSensors.H"

```
IOdictionary pLocs
(
  IOobject
  (
    "probeLocations",
    runTime.constant(),
    mesh,
    IOobject::MUST_READ,
    IOobject::NO_WRITE
  )
);
```

. . .

"CScreateSensors.H"

```
...

// Probes
// create pointer to probes
const pointField& probeLocations(pLocs.lookup("probeLocations"));
// build cell ID list
labelList probeCells(probeLocations.size(), -1);
// create array for storing temperature readings
List<double> Tsensor(probeLocations.size(), 0.0);
// locate probe cells and take first reading
forAll(probeLocations, pI)
{
    probeCells[pI] = mesh.findCell(probeLocations[pI]);
    Tsensor[pI] = T[probeCells[pI]];
}

...
```

"CScreateSensors.H"

```
. . .

// Control system parameters
double CS_Tmed; // variable for storing measured mean temperature
dimensionedScalar CS_setPoint       // set point
(    pLocs.lookup("CS_setPoint")    );
dimensionedScalar CS_gain             // gain
(    pLocs.lookup("CS_gain")          );
```

"CScontrolActions.H"

```
forAll(probeLocations, pI)
{
    Tsensor[pI] = T[probeCells[pI]]; // measure temperatures
}

// Calculate mean temperature
CS_Tmed = 0;
forAll(probeLocations, pI)
{
    CS_Tmed = CS_Tmed + (Tsensor[pI]);
}
if (probeLocations.size()>0) {CS_Tmed = CS_Tmed/probeLocations.size();}
.    .    .
```

"CScontrolActions.H"

```
. . .

// Control lowerWall temperature
label patchID = mesh.boundaryMesh().findPatchID("lowerWall");
fixedValueFvPatchScalarField& WallTemperature =
  refCast<fixedValueFvPatchScalarField>(T.boundaryField()[patchID]);
scalarField& TemperatureValue  = WallTemperature;

forAll (TemperatureValue,i) {
  TemperatureValue[i] -= CS_gain.value()*(CS_Tmed-CS_setPoint.value());
}

// Output relevant information (processor 0 only)
if( Pstream::myProcNo() == 0 )
{
  Pout << endl;
  Pout << " *** Control System | mean T: " << CS_Tmed << "oC - ";
  Pout << " target T: "  << CS_setPoint.value() << "oC - ";
  Pout << " lowerWall T: " << TemperatureValue[0] << "oC - ***" << endl << endl;
}
```

"constant/probeDictionary"

```
// FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      probeLocations;
}
probeLocations
(
  ( 0.2  -0.01  0)
  ( 0.2   0.0  0)
  ( 0.2   0.01  0)
);
fields
(
  T
);
CS_setPoint CS_setPoint [0 0 0 0 0 0 0]  30;
CS_gain     CS_gain     [0 0 0 0 0 0 0]  0.01;
```

Version 2: control system (first try)

- Remove #include "modifyBC.H"
- Add #include "CScreateSensors.H" before the time loop
- Add #include "CScontrolActions.H" inside the time loop
- Compile
- Create dictionary "constant/probeLocations"
- Run serial
- Decompose and run in parallel

Version 2: control system (first try)

"CScreateSensors.H"

```
IOdictionary pLocs
(
  IOobject
  (
    "probeLocations",
    runTime.constant(),
    mesh,
    IOobject::MUST_READ,
    IOobject::NO_WRITE
  )
);


. . .
```

"CScreateSensors.H"

```
. . .

// Probes
// create pointer to probes
const pointField& probeLocations(pLocs.lookup("probeLocations"));
// build cell ID list
labelList probeCells(probeLocations.size(), -1);
// create array for storing temperature readings
List<double> Tsensor(probeLocations.size(), 0.0);
// locate probe cells and take first reading
forAll(probeLocations, pI)
{
  probeCells[pI] = mesh.findCell(probeLocations[pI]);
  Tsensor[pI] = T[probeCells[pI]];
}

. . .
```

Version 3: control system (revisited)

- Modify #include "CScreateSensors.H" to include interprocessor communication
- Modify #include "CScontrolActions.H" to include interprocessor communication
- Compile
- Run in parallel

"CScreateSensors.H"

```
:
forAll(probeLocations, pI)
{
probeCells[pI] = mesh.findCell(probeLocations[pI]);
Tsensor[pI] = T[probeCells[pI]];
}
reduce( Tsensor, sumOp<List<double> >() ); // *** communicate temperatures ***
:
```

Version 3: control system (revisited)

"CScontrolActions.H"

```
:
forAll(probeLocations, pI)
{
Tsensor[pI] = T[probeCells[pI]]; // measure temperatures
}
reduce( Tsensor, sumOp<List<double> >() ); // *** communicate temperatures ***
:
```

Thanks for your attention...
Have a nice week and a fruitful conference!