Supplement

High level programming in OpenFOAM®
Building blocks

Roadmap

1. **Programming in OpenFOAM®. Building blocks.**
2. Implementing boundary conditions using high level programming
3. Modifying applications – Highlights
4. Implementing an application from scratch
5. Adding the scalar transport equation to icoFoam

Programming in OpenFOAM®. Building blocks

- In the directory **`$WM_PROJECT_DIR/applications/test`**, you will find the source code of several test cases that show the usage of most of the OpenFOAM® classes.
- We highly encourage you to take a look at these test cases and try to understand how to use the classes.
- We will use these basic test cases to understand the following base classes: tensors, fields, mesh, and basic discretization.
- For your convenience, we already copied the directory **`$WM_PROJECT_DIR/applications/test`** into the directory **`$PTOFC/programming_playground/test`**

Programming in OpenFOAM®. Building blocks

- During this session we will study the building blocks to write basic programs in OpenFOAM®:
    - First, we will start by taking a look at the algebra of tensors in OpenFOAM®.
    - Then, we will take a look at how to generate tensor fields from tensors.
    - Next, we will learn how to access mesh information.
    - Finally we will see how to discretize a model equation and solve the linear system of equations using OpenFOAM® classes and templates.
    - And of course, we are going to program a little bit in C++. But do not be afraid, after all this is not a C++ course.

- Remember, all OpenFOAM® components are implemented in library form for easy re-use.
- OpenFOAM® encourage code re-use. So basically, we are going to take something that already exist, and we are going to modify it to fix our needs.
- We like to call this method CPAC (copy-paste-adapt-compile).

Programming in OpenFOAM®. Building blocks

Basic tensor classes in OpenFOAM®

- OpenFOAM® represents scalars, vectors and matrices as tensor fields. A zero rank tensor is a scalar, a first rank tensor is a vector and a second rank tensor is a matrix.
- OpenFOAM® contains a C++ class library named **primitive** (`$FOAM_SRC/OpenFOAM/primitives/`). In this library, you will find the classes for the tensor mathematics.
- In the following table, we show the basic tensor classes available in OpenFOAM®, with their respective access functions.

Tensor Rank	Common name	Basic class	Access function
0	Scalar	`scalar`	
1	Vector	`vector`	`x(), y(), z()`
2	Tensor	`tensor`	`xx(), xy(), xz()` …

Programming in OpenFOAM®. Building blocks

Basic tensor classes in OpenFOAM®

- In OpenFOAM®, the second rank tensor (or matrix)

$$\mathbf{T} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

can be declared in the following way

```
tensor    T(1, 2, 3, 4, 5, 6, 7, 8, 9);
```

- We can access the component T_{13} or T_{xz} using the **xz ( )** access function,

Programming in OpenFOAM®. Building blocks

Basic tensor classes in OpenFOAM®

- For instance, the following statement,

```
Info << "Txz = " << T.xz ( ) << endl;
```

$$\mathbf{T} = \begin{pmatrix} 1 & 2 & \circled{3} \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

- Will generate the following screen output,

```
$> Txz = 3
```

- Notice that to output information to the screen in OpenFOAM®, we use the function **Info** instead of the function **cout** (used in standard C++).
- The function **cout** will work fine, but it will give you problems when running in parallel.

Programming in OpenFOAM®. Building blocks

Algebraic tensor operations in OpenFOAM®

- Tensor operations operate on the entire tensor entity.
- OpenFOAM® syntax closely mimics the syntax used in written mathematics, using descriptive functions (*e.g.* mag) or symbolic operators (*e.g.* +).
- OpenFOAM® also follow the standard rules of linear algebra when working with tensors.
- Some of the algebraic tensor operations are listed in the following table (where **a** and **b** are vectors, s is a scalar, and **T** is a tensor).

Operation	**Remarks**	**Mathematical description**	**OpenFOAM® description**
Addition		**a + b**	`a + b`
Scalar multiplication		s**a**	`s * a`
Outer product	rank a, b >=1	**ab**	`a * b`
Inner product	rank a, b >=1	**a.b**	`a & b`
Double inner product	rank a, b >=2	**a:b**	`a && b`
Magnitude		|**a**|	`mag(a)`
Determinant		det **T**	`det(T)`

You can find a complete list of all operators in the programmer's guide

Programming in OpenFOAM®. Building blocks

Dimensional units in OpenFOAM®

- As we already know, OpenFOAM® is fully dimensional.
- Dimensional checking is implemented as a safeguard against implementing a meaningless operation.
- OpenFOAM® encourages the user to attach dimensional units to any tensor and it will perform dimension checking of any tensor operation.
- You can find the dimensional classes in the directory
  `$FOAM_SRC/OpenFOAM/dimensionedTypes/`
- The dimensions can be hardwired directly in the source code or can be defined in the input dictionaries.
- From this point on, we will be attaching dimensions to all the tensors.

Programming in OpenFOAM®. Building blocks

Dimensional units in OpenFOAM®

- Units are defined using the **dimensionSet** class tensor, with its units defined using the **dimensioned<Type>** template class, the **<Type>** being scalar, vector, tensor, etc. The **dimensioned<Type>** stores the variable name, the dimensions and the tensor values.
- For example, a tensor with dimensions is declare in the following way:

```
1   dimensionedTensor sigma
2   (
3      “sigma”,
4      dimensionSet(1, -1, -2, 0, 0, 0, 0),
5      tensor(10e6,0,0,0,10e6,0,0,0,10e6)
6   );
```

$$\equiv \sigma = \begin{pmatrix} 10^6 & 0 & 0 \\ 0 & 10^6 & 0 \\ 0 & 0 & 10^6 \end{pmatrix}$$

- In line 1 we create the object **sigma**.
- In line 4, we use the class **dimensonSet** to attach units to the object **sigma**.
- In line 5, we set the input values of the tensor **sigma**.

Programming in OpenFOAM®. Building blocks

Units correspondence in dimensionSet

- The units of the class **dimensionSet** are defined as follows

```
dimensionSet (kg, m, s, K, mol, A, cd)
```

- Therefore, the tensor **sigma**,

```
dimensionedTensor sigma
(
   "sigma",
   dimensionSet(1, -1, -2, 0, 0, 0, 0),
   tensor(10e6,0,0,0,10e6,0,0,0,10e6)
);
```

$$\equiv \sigma = \begin{pmatrix} 10^6 & 0 & 0 \\ 0 & 10^6 & 0 \\ 0 & 0 & 10^6 \end{pmatrix}$$

- Has pressure units or $kg\ m^{-1}\ s^{-2}$

Programming in OpenFOAM®. Building blocks

Dimensional units examples

- To attach dimensions to any tensor, you need to access dimensional units class.
- To do so, just add the header file *dimensionedTensor.H* to your program.

```
#include "dimensionedTensor.H"
...
...
...
dimensionedTensor sigma
(
    "sigma",
    dimensionSet(1, -1, -2, 0, 0, 0, 0),
     tensor(1e6,0,0,0,1e6,0,0,0,1e6)
);
Info<< "Sigma: " << sigma << endl;
...
...
...
```

- The output of the previous program should look like this:

```
sigma sigma [1 -1 -2 0 0 0 0] (1e+06 0 0 0 1e+06 0 0 0 1e+06)
```

Programming in OpenFOAM®. Building blocks

Dimensional units examples

- As for base tensors, you can access the information of dimensioned tensors.
- For example, to access the name, dimensions, and values of a dimensioned tensor, you can proceed as follows:

```
Info << “Sigma name: ” << sigma.name ( ) << endl;
Info << “Sigma dimensions: ” << sigma.dimensions ( ) << endl;
Info << “Sigma value: ” << sigma.value ( ) << endl;
```

- To extract a value of a dimensioned tensor, you can proceed as follows:

```
Info<< "Sigma yy (22) value: " << sigma.value().yy() << endl;
```

- Note that the **value()** member function first converts the expression to a tensor, which has a **yy()** member function.
- The **dimensionedTensor** class does not have a **yy()** member function, so it is not possible to directly get its value by using **sigma.yy()**.

Programming in OpenFOAM®. Building blocks

OpenFOAM® lists and fields

- OpenFOAM® frequently needs to store sets of data and perform mathematical operations.
- OpenFOAM® provides an array template class **List<Type>**, making it possible to create a list of any object of class **Type** that inherits the functions of the **Type**. For example a **List** of **vector** is **List<vector>**.
- Lists of the tensor classes are defined in OpenFOAM® by the template class **Field<Type>**.
- For better code legibility, all instances of **Field<Type>**, e.g. **Field<vector>**, are renamed using **typedef** declarations as **scalarField**, **vectorField**, **tensorField**, **symmTensorField**, **tensorThirdField** and **symmTensorThirdField**.
- You can find the field classes in the directory **`$FOAM_SRC/OpenFOAM/fields/Fields`**.
- Algebraic operations can be performed between fields, subject to obvious restrictions such as the fields having the same number of elements.
- OpenFOAM® also supports operations between a field and a zero-rank tensor, e.g. all values of a **Field U** can be multiplied by the **scalar** 2 by simple coding the following line, **U = 2.0 * U**.

Programming in OpenFOAM®. Building blocks

Construction of a tensor field in OpenFOAM®

- To create fields, you need to access the tensor class.
- To do so, just add the header file *tensorField.H* to your program. This class inherit all the tensor algebra.

```
#include "tensorField.H"
...
...
...
tensorField tf1(2, tensor::one);
Info<< "tf1: " << tf1 << endl;
tf1[0] = tensor(1, 2, 3, 4, 5, 6, 7, 8, 9);
Info<< "tf1: " << tf1 << endl;
Info<< "2.0*tf1: " << 2.0*tf1 << endl;
...
...
...
```

- In this example, we created a list of two tensor fields (**tf1**), and both tensors are initialized to one.
- We can access components on the list using the access operator [ ].

Programming in OpenFOAM®. Building blocks

Example of use of tensor and field classes

- In the directory **`$PTOFC/programming_playground/my_tensor`** you will find a tensor class example.
- The original example is located in the directory **`$HOME/OpenFOAM/OpenFOAM-8/applications/test`**. Feel free to compare the files to spot the differences.
- Before compiling the file, let us recall how applications are structure,

```
working_directory/
├── applicationName.C
├── header-files.H
└── Make
    ├── files
    └── options
```

- *`applicationName.C`*: is the actual source code of the application.
- *`header_files.H`*: header files required to compile the application.

Programming in OpenFOAM®. Building blocks

Example of use of tensor and field classes

- Before compiling the file, let us recall how applications are structure.

```
working_directory/
├── applicationName.C
├── header-files.H
└── Make
    ├── files
    └── options
```

- The **`Make`** directory contains compilation instructions.
  - *`files`*: names all the source files `(.C)`, it specifies the name of the new application and the location of the output file.
  - *`options`*: specifies directories to search for include files and libraries to link the solver against.
- At the end of the file *`files`*, you will find the following line of code,
  EXE = $(FOAM_USER_APPBIN)/my_Test-tensor
- This is telling the compiler to name your application **my_Test-tensor** and to copy the executable in the directory **$FOAM_USER_APPBIN**.
- <u>To avoid conflicts between applications, always remember to give a proper name and a location to your programs and libraries.</u>

Programming in OpenFOAM®. Building blocks

Example of use of tensor and field classes

- Let us now compile the tensor class example. Type in the terminal:

```
1.  $> cd $PTOFC/programming_playground/my_tensor
2.  $> wmake
3.  $> my_Test-tensor
```

- In step 2, we used `wmake` (distributed with OpenFOAM®) to compile the source code.
- The name of the executable will be **my_Test-tensor** and it will be located in the directory **`$FOAM_USER_APPBIN`** (as specified in the file *`Make/files`*)
- At this point, take a look at the output and study the file *`Test-tensor.C`*. Try to understand what we have done.
- After all, is not that difficult. Right?

Programming in OpenFOAM®. Building blocks

- At this point, we are a little bit familiar with tensor, fields, and lists in OpenFOAM®.
- They are the base to building applications in OpenFOAM®.
- Let us now take a look at the whole solution process:
  - Creation of the tensors.
  - Mesh assembly.
  - Fields creation.
  - Equation discretization.
- All by using OpenFOAM® classes and template classes

Programming in OpenFOAM®. Building blocks

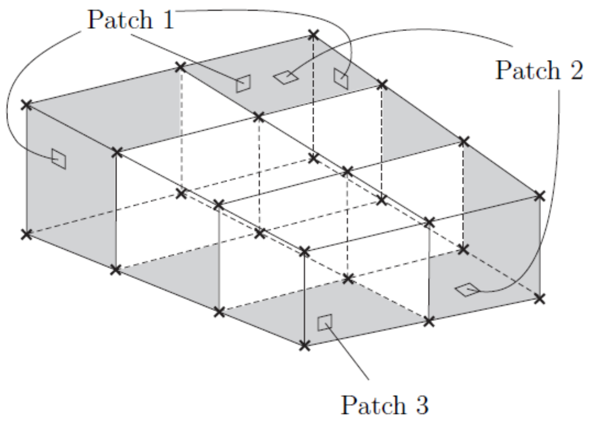
Discretization of a tensor field in OpenFOAM®

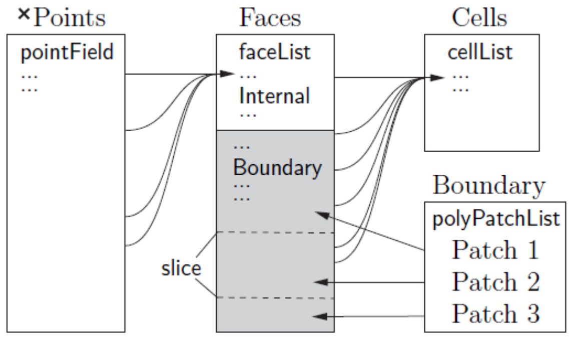
- The discretization is done using the FVM (Finite Volume Method).
- The cells are contiguous, *i.e.*, they do not overlap and completely fill the domain.
- Dependent variables and other properties are stored at the cell centroid.
- No limitations on the number of faces bounding each cell.
- No restriction on the alignment of each face.
- The mesh class **polyMesh** is used to construct the polyhedral mesh using the minimum information required.
- You can find the **polyMesh** classes in the directory **`$FOAM_SRC/OpenFOAM/meshes`**
- The **fvMesh** class extends the **polyMesh** class to include additional data needed for the FVM discretization.
- You can find the **fvMesh** classes in the directory **`$FOAM_SRC/src/finiteVolume/fvMesh`**

Programming in OpenFOAM®. Building blocks

Discretization of a tensor field in OpenFOAM®

- The template class **geometricField** relates a tensor field to a **fvMesh**.
- Using typedef declarations **geometricField** is renamed to **volField** (cell center), **surfaceField** (cell faces), and **pointField** (cell vertices).
- You can find the **geometricField** classes in the directory **`$FOAM_SRC/OpenFOAM/fields/GeometricFields`**.
- The template class **geometricField** stores internal fields, boundary fields, mesh information, dimensions, old values and previous iteration values.
- A **geometricField** inherits all the tensor algebra of its corresponding field, has dimension checking, and can be subjected to specific discretization procedures.
- Let us now access the mesh information of a simple case.





Programming in OpenFOAM®. Building blocks

Data stored in the `fvMesh` class

Class	Description	Symbol	Access function
`volScalarField`	Cell volumes	V	`V()`
`surfaceVectorField`	Face area vector	$\mathbf{S}_f$	`Sf()`
`surfaceScalarField`	Face area magnitude	$\lvert\mathbf{S}_f\rvert$	`magSf()`
`volVectorField`	Cell centres	$\mathbf{C}$	`C()`
`surfaceVectorField`	Face centres	$\mathbf{C}_f$	`Cf()`
`surfaceScalarField`	Face fluxes	ϕ_g	`Phi()`

Programming in OpenFOAM®. Building blocks

Accessing fields defined in a mesh

- To access fields defined at cell centers of the mesh you need to use the class **volField**.
- The class **volField** can be accessed by adding the header *volFields.H* to your program.

```
volScalarField p
(
    IOobject
    (
        "p",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
Info<< p << endl;
Info<< p.boundaryField()[0] << endl;
```

Create scalar volField p

Assign and initialization of scalar volField to the mesh

Output some information

Programming in OpenFOAM®. Building blocks

Accessing fields using for loops

- To access fields using **for** loops, we can use OpenFOAM® macro **forAll**, as follows,

Outputs name of patch

```
forAll(mesh.boundaryMesh(), patchI)
Info << "Patch " << patchI << ": " << mesh.boundary()[patchI].name() << " with "
     << mesh.boundary()[patchI].Cf().size() << " faces. Starts at total face "
     << mesh.boundary()[patchI].start() << endl;
```

Outputs size of patch (number of faces)

- In the previous statement **mesh.boundaryMesh()** is the size of the loop, and **patchI** is the iterator. The iterator always starts from zero.
- The **forAll** loop is equivalent to the standard **for** loop in C++.

```
for (int i = 0; i < mesh.boundaryMesh().size(); i++)
Info << "Patch " << i << ": " << mesh.boundary()[i].name() << " with "
     << mesh.boundary()[i].Cf().size() << " faces. Starts at total face "
     << mesh.boundary()[i].start() << endl;
```

Outputs starting face of patch

- Notice that we used as iterator **`i`** instead of **`patchI`**, this does not make any difference.

Programming in OpenFOAM®. Building blocks

Equation discretization in OpenFOAM®

- At this stage, OpenFOAM® converts the PDEs into a set of linear algebraic equations, **A x = b**, where **x** and **b** are **volFields** (**geometricField**).
- **A** is a **fvMatrix**, which is created by the discretization of a **geometricField** and inherits the algebra of its corresponding field, and it supports many of the standard algebraic matrix operations.
- The **fvm** (**finiteVolumeMethod**) and **fvc** (**finiteVolumeCalculus**) classes contain static functions for the differential operators and discretize any **geometricField**.
- **fvm** returns a **fvMatrix**, and **fvc** returns a **geometricField**.
- In the directories **`$FOAM_SRC/finiteVolume/finiteVolume/fvc`** and **`$FOAM_SRC/finiteVolume/finiteVolume/fvm`** you will find the respective classes.
- Remember, the PDEs or ODEs we want to solve involve derivatives of tensor fields with respect to time and space. What we re doing at this point, is applying the finite volume classes to the fields, and assembling a linear system.

Programming in OpenFOAM®. Building blocks

Discretization of the basic PDE terms in OpenFOAM®

The list is not complete

Term description	Mathematical expression	`fvm::` `fvc::`
Laplacian	$\nabla^2\phi \quad , \nabla \cdot \Gamma \nabla \phi$	`laplacian(phi)` `laplacian(Gamma, phi)`
Time derivative	$\dfrac{\partial \phi}{\partial t} \quad , \quad \dfrac{\partial \rho \phi}{\partial t}$	`ddt(phi)` `ddt(rho,phi)`
Convection	$\nabla \cdot (\psi) \ , \ \nabla \cdot (\psi \phi)$	`div(psi,scheme)` `div(psi,phi)`
Source	$\rho \phi$	`Sp(rho,phi)` `SuSp(rho,phi)`

ϕ `vol<type>Field` ρ `scalar, volScalarField` ψ `surfaceScalarField`

Programming in OpenFOAM®. Building blocks

Discretization of the basic PDE terms in OpenFOAM®

- To discretize the fields in a valid mesh, we need to access the finite volume class. This class can be accessed by adding the header *fvCFD.H* to your program.
- To discretize the scalar transport equation in a mesh, we can proceed as follows,

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

Assemble and solve linear system arising form the discretization

Discretize equations

- Remember, you will need to first create the mesh, and initialize the variables and constants. That is, all the previous steps.
- Finally, everything we have done so far inherits all parallel directives. There is no need for specific parallel programming.

Programming in OpenFOAM®. Building blocks

Discretization of the basic PDE terms in OpenFOAM®

- The previous discretization is equivalent to,

```
fvScalarMatrix TEqn
(
          fvm::ddt(T)
        +    fvm::div(phi,T)
        -    fvm::laplacian(DT,T)
);

Teqn.solve();
```

Creates object TEqn that contains the coefficient matrix arising from the discretization

Discretize equations

Solve the linear system Teqn

- Here, **fvScalarMatrix** contains the matrix derived from the discretization of the model equation.
- **fvScalarMatrix** is used for scalar fields and **fvVectorMatrix** is used for vector fields.
- This syntax is more general, since it allows the easy addition of terms to the model equations.

Programming in OpenFOAM®. Building blocks

Discretization of the basic PDE terms in OpenFOAM®

- At this point, OpenFOAM® assembles and solves the following linear system,

$$
\begin{bmatrix}
a_{11} & a_{12} & & \ddots & & & & \\
a_{21} & a_{22} & a_{23} & & \ddots & & & \\
 & \ddots & \ddots & \ddots & & \ddots & & \\
\ddots & & \ddots & \ddots & \ddots & & \ddots & \\
 & a_S & & a_W & a_P & a_E & & a_N \\
 & & \ddots & & \ddots & \ddots & \ddots & \\
 & & & \ddots & & \ddots & \ddots & \ddots \\
 & & & & \ddots & & \ddots & a_{PP}
\end{bmatrix}
\times
\begin{bmatrix}
 \\ x_S \\ \\ x_W \\ x_P \\ x_E \\ x_N \\ \\
\end{bmatrix}
=
\begin{bmatrix}
 \\ b_S \\ \\ b_W \\ b_P \\ b_E \\ b_N \\ \\
\end{bmatrix}
$$

Coefficient Matrix (sparse, square)
The coefficients depend on geometrical quantities, fluid properties and non-linear equations

Unknow quantity

Boundary conditions and source terms

Programming in OpenFOAM®. Building blocks

Example of use of tensor and field classes

- Let us study a **fvMesh** example. First let us compile the program `my_Test-mesh`. Type in the terminal,

```
1. $> cd $PTOFC/programming_playground/my_mesh/
2. $> wmake
```

- To access the mesh information, we need to use this program in a valid mesh.

```
1. $> cd $PTOFC/programming_playground/my_mesh/cavity
2. $> blockMesh
3. $> my_Test-mesh
```

- At this point, take a look at the output and study the file *`Test-mesh.C`*. Try to understand what we have done.
- FYI, the original example is located in the directory `$PTOFC/programming_playground/test/mesh`.

Programming in OpenFOAM®. Building blocks

A few OpenFOAM® programming references

- You can access the API documentation in the following link, https://cpp.openfoam.org/v5/
- You can access the coding style guide in the following link, https://openfoam.org/dev/coding-style-guide/
- You can report programming issues in the following link, https://bugs.openfoam.org/rules.php
- You can access openfoamwiki coding guide in the following link, http://openfoamwiki.net/index.php/OpenFOAM_guide
- You can access the user guide in the following link, https://cfd.direct/openfoam/user-guide/
- You can read the OpenFOAM® Programmer's guide in the following link (it seems that this guide is not supported anymore), http://foam.sourceforge.net/docs/Guides-a4/ProgrammersGuide.pdf

A few good C++ references

- **The C++ Programming Language.** B. Stroustrup. 2013, Addison-Wesley.
- **The C++ Standard Library.** N. Josuttis. 2012, Addison-Wesley.
- **C++ for Engineers and Scientists.** G. J. Bronson. 2012, Cengage Learning.
- **Sams Teach Yourself C++ in One Hour a Day.** J. Liberty, B. Jones. 2004, Sams Publishing.
- **C++ Primer.** S. Lippman, J. Lajoie, B. Moo. 2012, Addison-Wesley.
- http://www.cplusplus.com/
- http://www.learncpp.com/
- http://www.cprogramming.com/
- http://www.tutorialspoint.com/cplusplus/
- http://stackoverflow.com/

Roadmap

1. ~~Programming in OpenFOAM®. Building blocks.~~
2. **Implementing boundary conditions using high level programming**
3. Modifying applications – Highlights
4. Implementing an application from scratch
5. Adding the scalar transport equation to icoFoam

Implementing boundary conditions using high level programming

- Hereafter we will work with high level programming, this is the hard part of programming in OpenFOAM®.
- High level programming requires some knowledge on C++ and OpenFOAM® API library.
- Before doing high level programming, we highly recommend you to try with **codeStream**, most of the time it will work.
- We will implement the parabolic profile, so you can compare this implementation with **codeStream** ad **codedFixedValue** BCs.
- When we program boundary conditions, we are building a new library that can be linked with any solver. To compile the library, we use the command `wmake` (distributed with OpenFOAM®).
- At this point, you can work in any directory, but we recommend you to work in your OpenFOAM® user directory, type in the terminal,

1.
```
$> cd $WM_PROJECT_USER_DIR/run
```

Implementing boundary conditions using high level programming

- Let us create the basic structure to write the new boundary condition, type in the terminal,

```
1. | $> foamNewBC -f -v myParabolicVelocity
2. | $> cd myParabolicVelocity
```

- The utility `foamNewBC`, will create the directory structure and all the files needed to write your own boundary conditions.
- We are setting the structure for a fixed (the option `-f`) velocity (the option `-v`), boundary condition, and we name our boundary condition `ParabolicVelocity` .
- If you want to get more information on how to use `foamNewBC`, type in the terminal,

```
1. | $> foamNewBC -help
```

Implementing boundary conditions using high level programming

Directory structure of the new boundary condition

```
./myParabolicVelocity
├── Make
│   ├── files
│   └── options
├── myParabolicVelocityFvPatchVectorField.C
└── myParabolicVelocityFvPatchVectorField.H
```

The directory contains the source code of the boundary condition.

- *myParabolicVelocityFvPatchVectorField.C*: is the actual source code of the application. This file contains the definition of the classes and functions.
- *myParabolicVelocityFvPatchVectorField.H*: header files required to compile the application. This file contains variables, functions and classes declarations.
- The **Make** directory contains compilation instructions.
  - *Make/files*: names all the source files (*.C*), it specifies the boundary condition library name and location of the output file.
  - *Make/options*: specifies directories to search for include files and libraries to link the solver against.

Implementing boundary conditions using high level programming

The header file (.H)

- Let us start to do some modifications. Open the header file using your favorite text editor (we use gedit).

```
96    //- Single valued scalar quantity, e.g. a coefficient
97    scalar scalarData_;
98
99    //- Single valued Type quantity, e.g. reference pressure pRefValue_
100   //  Other options include vector, tensor
101   vector data_;
102
103   //- Field of Types, typically defined across patch faces
104   //  e.g. total pressure p0_.  Other options include vectorField
105   vectorField fieldData_;
106
107   //- Type specified as a function of time for time-varying BCs
108   autoPtr<Function1<vector>> timeVsData_;
109
110   //- Word entry, e.g. pName_ for name of the pressure field on database
111   word wordData_;
112
113   //- Label, e.g. patch index, current time index
114   label labelData_;
115
116   //- Boolean for true/false, e.g. specify if flow rate is volumetric_
117   bool boolData_;
118
119
120   // Private Member Functions
121
122   //- Return current time
123   scalar t() const;
```

- In lines 96-123 different types of private data are declared.
- These are the variables we will use for the implementation of the new BC.
- In our implementation we need to use vectors and scalars, therefore we can keep the lines 97 and 101.
- We can delete lines 103-117, as we do not need those datatypes.
- Also, as we will use two vectors in our implementation, we can duplicate line 101.
- You can leave the rest of the file as they are.

Implementing boundary conditions using high level programming

The header file (.H)

- At this point, your header file should looks like this one,

```
96    //- Single valued scalar quantity, e.g. a coefficient
97    scalar scalarData_;
98
99    //- Single valued Type quantity, e.g. reference pressure pRefValue_
100   //  Other options include vector, tensor
101   vector data_;
102   vector data_;
```

- Change the name of **scalarData_** to **maxValue_** (line 97).
- Change the names of the two vectors **data_** (lines 101-102). Name the first one **n_** and the last one **y_**.

```
96    //- Single valued scalar quantity, e.g. a coefficient
97    scalar maxValue_;
98
99    //- Single valued Type quantity, e.g. reference pressure pRefValue_
100   //  Other options include vector, tensor
101   vector n_;
102   vector y_;
```

It is recommended to initialize them in the same order as you declare them in the header file

- We just declared the variables that we will use. You can now save and close the file.

Implementing boundary conditions using high level programming

The source file (.C)

- Let us start to modify the source file. Open the source file with your favorite editor.
- Lines 34-37 refers to a private function definition. This function allows us to access simulation time. Since in our implementation we do not need to use time, we can safely remove these lines.

```
34      Foam::scalar Foam::myParabolicVelocityFvPatchVectorField::t() const
35      {
36          return db().time().timeOutputValue();
37      }
```

- Let us compile the library to see what errors we get. Type in the terminal,

```
$> wmake
```

- You will get a lot of errors.
- Since we deleted the datatypes **fieldData**, **timeVsData**, **wordData**, **labelData** and **boolData** in the header file, we need to delete them as well in the C file. Otherwise the compiler complains.

Implementing boundary conditions using high level programming

The source file (.C)

- At this point, let us erase all the occurrences of the datatypes **fieldData**, **timeVsData**, **wordData**, **labelData**, and **boolData**.
- Locate line 38,

```
38   Foam::myParabolicVelocityFvPatchVectorField::
     ...
     ...
     ...
```

- Using this line as your reference location in the source code, follow these steps,
  - Erase the following lines in incremental order (be sure to erase only the lines that contain the words **fieldData**, **timeVsData**, **wordData**, **labelData** and **boolData**):
    48-52, 63-67, 90-94, 102-106, 115-119, 172-175.
  - Erase the following lines (they contain the word **fieldData**), 126, 140, 151-153.
  - Replace all the occurrences of the word **scalarData** with **maxValue** (11 occurrences).

Implementing boundary conditions using high level programming

The source file (.C)

- Duplicate all the lines where the word **data** appears (6 lines), change the word **data** to **n** in the first line, and to **y** in the second line, erase the comma in the last line. For example,

Original statements

```
fixedValueFvPatchVectorField(p, iF),
maxValue_(0.0),
data_(Zero),
data_(Zero),
```

Modified statements

```
fixedValueFvPatchVectorField(p, iF),
maxValue_(0.0),
n_(Zero),
y_(Zero)
```

$\longleftarrow$ Remember to erase the comma

The source file (.C)

- We are almost done; we just defined all the datatypes. Now we need to implement the actual boundary condition.
- Look for line 147 (**updateCoeffs()** member function), and add the following statements,

```
147    void Foam::myParabolicVelocityFvPatchVectorField::updateCoeffs()
148    {
149        if (updated())
150        {
151            return;
152        }
153
154        boundBox bb(patch().patch().localPoints(), true);
155
156        vector ctr = 0.5*(bb.max() + bb.min());
157
158        const vectorField& c = patch().Cf();
159
160        scalarField coord = 2*((c - ctr) & y_)/((bb.max() - bb.min()) & y_);
```

Add these lines

The actual implementation of the BC is always done in this class

Find patch bounds (minimum and maximum points)

Coordinates of patch midpoint

Access patch face centers

$$U_{max}\left(1.0 - \frac{(y-c)^{\cancel{2}}}{r^{\cancel{2}}}\right)$$

Computes scalar field to be used for defining the parabolic profile

Implementing boundary conditions using high level programming

The source file (.C)

- Add the following statement in line 164,

```
162     fixedValueFvPatchVectorField::operator==
163     (
164         n_*maxValue_*(1.0 - sqr(coord))
165     );
166
```

The access function operator== is used to assign the values to the boundary patches

Our boundary condition

$$U_{max}\left(1.0 - \frac{(y-c)^2}{r^2}\right)$$

- The last step before compiling the new BC is to erase a few commas.
- Look for lines 48, 64, 92, 105, 119, and erase the comma at the end of each line.
- At this point we have a valid library where we implemented a new BC.
- Finally, you can go back to the header file (*.H) and document your boundary condition implementation.
    - You can add the comments in the header of the file (lines 1-73).

Implementing boundary conditions using high level programming

The source file (.C)

- At this point we have a valid library where we have implemented a new BC.
- Try to compile it, we should not get any error (maybe one warning). Type in the terminal,

```
1. $> wmake
```

- If you are feeling lazy, or if you can not fix the compilation errors, you will find the source code in the directory,
    - **`$PTOFC/101programming/src/myParabolicVelocity`**

Implementing boundary conditions using high level programming

The source file (.C)

- Before moving forward, let us comment a little bit the source file.
- First at all, there are five classes constructors and each of them have a specific task.
- In our implementation we did not use all the classes, we only use the first two classes.
- The first class is related to the initialization of the variables.
- The second class is related to reading the input dictionaries.
- We will not comment on the other classes as it is out of the scope of this example (they deal with input tables, mapping, and things like that).
- The implementation of the boundary condition is always done using the **updateCoeffs()** member function.
- When we compile the source code, it will compile a library with the name specified in the file *`Make/file`*. In this case, the name of the library is **libmyParabolicVelocity**.
- The library will be located in the directory **$(FOAM_USER_LIBBIN)**, as specified in the file *`Make/file`*.

Implementing boundary conditions using high level programming

The source file (.C)

- The first class is related to the initialization of the variables declared in the header file.
- In line 47 we initialize **maxValue** with the value of zero. The vectors **n** and **y** are initialized as a zero vector by default or (0, 0, 0).
- It is not a good idea to initialize these vectors as zero vectors by default. Let us use as default initialization (1, 0, 0) for vector **n** and (0,1,0) for vector **y**.

```
38   Foam::myParabolicVelocityFvPatchVectorField::
39   myParabolicVelocityFvPatchVectorField
40   (
41       const fvPatch& p,
42       const DimensionedField<vector, volMesh>& iF
43   )
44   :
45       fixedValueFvPatchVectorField(p, iF),
46       maxValue_(0.0),
47       n_(Zero),
48       y_(Zero)
49   {
50   }
```

Change to n_(1,0,0)

Change to y_(0,1,0)

Implementing boundary conditions using high level programming

The source file (.C)

- The second class is used to read the input dictionary.
- Here we are reading the values defined by the user in the dictionary *U*.
- The function **lookup** will search the specific keyword in the input file.

```
Foam::myParabolicVelocityFvPatchVectorField::
myParabolicVelocityFvPatchVectorField
(
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchVectorField(p, iF),
    maxValue_(readScalar(dict.lookup("maxValue"))),
    n_(pTraits<vector>(dict.lookup("n"))),
    y_(pTraits<vector>(dict.lookup("y")))
{


    fixedValueFvPatchVectorField::evaluate();

}
```

dict.lookup will look for these keywords in the input dictionary

Implementing boundary conditions using high level programming

The source file (.C)

- Since we do not want the vectors **n** and **y** to be zero vectors, we add the following sanity check starting form line 67.
- These statements check if the given **n** and **y** vectors in the input dictionary is zero or not.
- If any of the vectors are zero it gives the fatal error and terminate the program.
- On the other hand, if everything is ok it will normalize **n** and **y** (since in our implementation they are direction vectors).

```
66
67        if (mag(n_) < SMALL || mag(y_) < SMALL)
68        {
69             FatalErrorIn("parabolicVelocityFvPatchVectorField(dict)")
70                 << "n or y given with zero size not correct"
71                 << abort(FatalError);
72        }
73
74        n_ /= mag(n_);      //This is equivalent to n_ = n_/mag(n_)
75        y_ /= mag(y_);      //This is equivalent to y_ = y_/(mag(y_)
76
77        fixedValueFvPatchVectorField::evaluate();
78
```

Add these statements

Implementing boundary conditions using high level programming

The source file (.C)

- At this point, we are ready to go.
- Save the files and recompile. Type in the terminal,

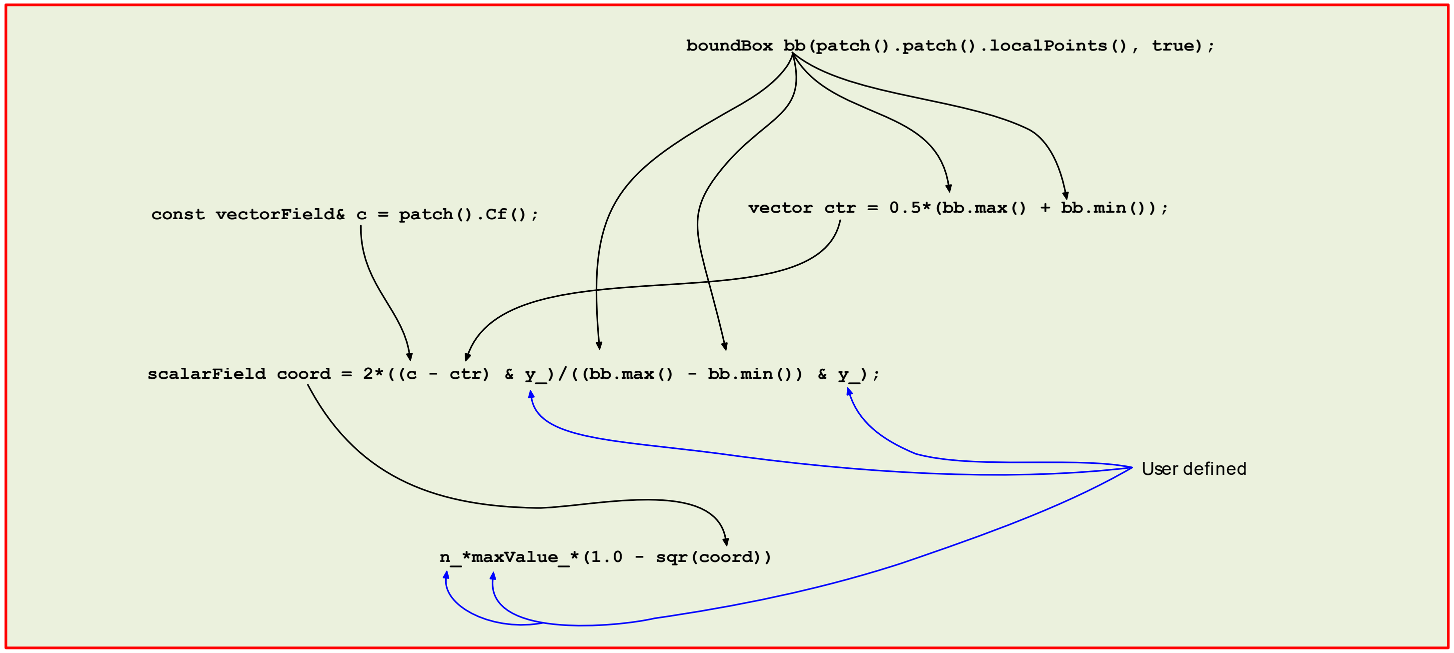
1.
```
$> wmake
```

- We should not get any error (maybe one warning).
- At this point we have a valid library that can be linked with any solver.
- If you get compilation errors, read the screen and try to sort it out, the compiler is always telling you what is the problem.
- If you are feeling lazy, or if you can not fix the compilation errors, you will find the source code in the directory,
    - **`$PTOFC/101programming/src/myParabolicVelocity`**

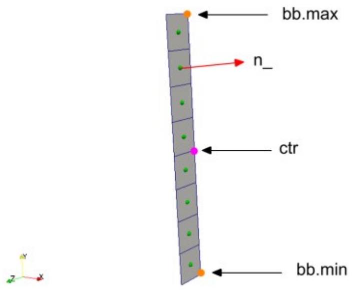
Implementing boundary conditions using high level programming

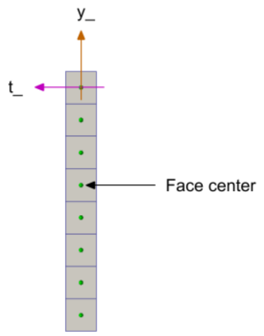
The source file (.C)

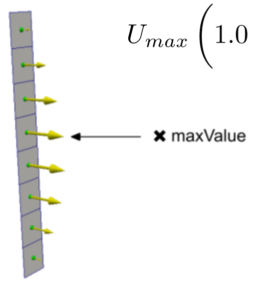
- Before using the new BC, let us take a look at the logic behind the implementation.



```
boundBox bb(patch().patch().localPoints(), true);

const vectorField& c = patch().Cf();

vector ctr = 0.5*(bb.max() + bb.min());

scalarField coord = 2*((c - ctr) & y_)/((bb.max() - bb.min()) & y_);

n_*maxValue_*(1.0 - sqr(coord))
```







$$U_{max}\left(1.0 - \frac{(y-c)^2}{r^2}\right)$$

Implementing boundary conditions using high level programming

Running the case

- This case is ready to run, the input files are located in the directory **`$PTOFC/101programming/src/case_elbow2d`**
- Go to the case directory,

1.
```
$> cd $PTOFC/101programming/src/case_elbow2d
```

- Open the file *`0/U`*, and look for the definition of the new BC **velocity-inlet-5**,

```
velocity-inlet-5
{
    type          myParabolicVelocity;

    maxValue 2.0;
    n        (1 0 0);
    y        (0 1 0);
}
```

Name of the boundary condition

User defined values
max value, n, y
If you set n or y to (0 0 0), the solver will abort execution

Implementing boundary conditions using high level programming

Running the case

- We also need to tell the application that we want to use the library we just compiled.
- To do so, we need to add the new library in the dictionary file *controlDict*,

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

libs ("libmyParabolicVelocity.so");

application     icoFoam;
```

Name of the library
You can add as many libraries as you like

- The solver will dynamically link the library.
- At this point, we are ready to launch the simulation.

Running the case

- This case is ready to run, the input files are located in the directory **`$PTOFC/101programming/src/case_elbow2d`**
- To run the case, type in the terminal,

```
1. $> foamCleanTutorials
2. $> fluentMeshToFoam ../../../meshes_and_geometries/fluent_elbow2d_1/ascii.msh
3. $> icoFoam | tee log.solver
4. $> paraFoam
```

- At this point, you can compare the three implementations (**codeStream**, **codedFixedValue** and high-level programming).
- All of them will give the same outcome.

Implementing boundary conditions using high level programming

Adding some verbosity to the BC implementation

- Let us add some outputs to the BC.
- After the member function **updateCoeffs** (line 177), add the following lines,

```
177   fixedValueFvPatchVectorField::updateCoeffs();
178
179   Info << endl << "Face centers (c):" << endl;
180   Info << c << endl;
181   Info << endl << "Patch center (ctr):" << endl;
182   Info << ctr << endl;
183   Info << endl << "Patch (c - ctr):" << endl;
184   Info << c - ctr << endl;
185   Info << endl << "Patch max bound (bb.max):" << endl;
186   Info <<  bb.max() << endl;
187   Info << endl << "Patch min bound (bb.max):" << endl;
188   Info << bb.min() << endl;
189   Info << endl << "Patch coord ( 2*((c - ctr) & y_)/((bb.max() - bb.min()) & y_) ):" << endl;
190   Info << coord << endl;
191   Info << endl << "Patch ( 1.0 - sqr(coord)) :" << endl;
192   Info << n_*maxValue_*(1.0 - sqr(coord))<< endl;
193   Info << endl << "Loop for c, BC assigment << endl;
194   forAll(c, faceI)
195   {
196       Info << c[faceI] << "      " << n_*maxValue_*(1.0 - sqr(coord[faceI])) << endl;
197   }
198
199
```

- Recompile, rerun the simulation, look at the output, and do the math.

Implementing boundary conditions using high level programming

- In the directory **`$PTOFC/101programming/src/myParabolicVelocityMod`**, you will find an implementation of this boundary condition with conditional switches and screen output information.
- Try to figure out how this BC works.

- **Do you take the challenge?**
  - Starting from this boundary condition, try to implement a paraboloid BC.
  - If you are feeling lazy or at any point do you get lost, in the directory `$PTOFC/101programming/src/myParaboloidVelocity` you will find a working implementation of the paraboloid profile.
  - Open the source code and try to understand what we did (pretty much similar to the previous case).
  - In the directory **`$PTOFC/101programming/src/case_elbow3d`** you will find a case ready to use.

Roadmap

1. ~~Programming in OpenFOAM®. Building blocks.~~
2. ~~Implementing boundary conditions using high level programming~~
3. **Modifying applications – Highlights**
4. Implementing an application from scratch
5. Adding the scalar transport equation to icoFoam

Modifying applications – Highlights

- Implementing a new application from scratch in OpenFOAM® (or any other high-level programming library), can be an incredible daunting task.
- OpenFOAM® comes with many solvers, and as it is today, you do not need to implement new solvers from scratch.
- Of course, if your goal is to write a new solver, you will need to deal with programming. What you usually do, is take an existing solver and modify it.
- But in case that you would like to take the road of implementing new applications from scratch, we are going to give you the basic building blocks.
- We are also going to show how to add basic modifications to existing solvers.
- We want to remind you that this requires some knowledge on C++ and OpenFOAM® API library.
- Also, you need to understand the FVM, and be familiar with the basic algebra of tensors.
- Some common sense is also helpful.

Roadmap

1. ~~Programming in OpenFOAM®. Building blocks.~~
2. ~~Implementing boundary conditions using high level programming~~
3. ~~Modifying applications – Highlights~~
4. **Implementing an application from scratch**
5. Adding the scalar transport equation to icoFoam

Implementing an application from scratch

- Let us do a little bit of high-level programming, this is the hard part of working with OpenFOAM®.
- At this point, you can work in any directory. But we recommend you to work in your OpenFOAM® user directory, type in the terminal,

```
1.  $> cd $WM_PROJECT_USER_DIR/run
```

- To create the basic structure of a new application, type in the terminal,

```
1.  $> foamNewApp scratchFoam
2.  $> cd scratchFoam
```

- The utility `foamNewApp`, will create the directory structure and all the files needed to create the new application from scratch. The name of the application is **scratchFoam**.
- If you want to get more information on how to use `foamNewApp`, type in the terminal,

```
1.  $> foamNewApp -help
```

Implementing an application from scratch

Directory structure of the new boundary condition

```
scratchFoam/
├── createFields.H   ←   Does not exist, we will create it later
├── scratchFoam.C
└── Make
    ├── files
    └── options
```

The **scratchFoam** directory contains the source code of the solver.

- *scratchFoam.C*: contains the starting point to implement the new application.
- *createFields.H*: in this file we declare all the field variables and initializes the solution. This file does not exist at this point, we will create it later.
- The **Make** directory contains compilation instructions.
    - *Make/files*: names all the source files `(.C)`, it specifies the name of the solver and location of the output file.
    - *Make/options*: specifies directories to search for include files and libraries to link the solver against.
- To compile the new application, we use the command `wmake`.

Implementing an application from scratch

- Open the file `scratchFoam.C` using your favorite text editor, we will use gedit.
- At this point you should have this file that does not do anything. We need to add the statements to create a working applications.
- This is the starting point for new applications.

```

#include "fvCFD.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

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

    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

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

    Info<< "End\n" << endl;

    return 0;
}

```

This header is extremely important, it will add all the class declarations needed to access mesh, fields, tensor algebra, fvm/fvc operators, time, parallel communication, linear algebra, and so on.

Implementing an application from scratch

- Stating from line 31, add the following statements.
- We are going to use the PISO control options, even if we do not have to deal with velocity-pressure coupling.

```
30
31    #include "fvCFD.H"

32    #include "pisoControl.H"  ←  Solution control using PISO class
33
34    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
35
36    int main(int argc, char *argv[])
37    {
38        #include "setRootCase.H"   ←  Set directory structure

39        #include "createTime.H"    ←  Create time (object runtime)

41        #include "createMesh.H"    ←  Create time (object mesh)

42        #include "createFields.H"  ←  Initialize fields
                                        This source file does not exist yet, we need to create it

43        #include "CourantNo.H"     ←  Calculates and outputs the Courant Number

44        #include "initContinuityErrs.H"  ←  Declare and initialize the cumulative continuity error

49        pisoControl piso(mesh);    ←  Assign PISO controls to object mesh.  Creates object piso.
                                        Alternatively, you can use the header file createControl.H

51        Info<< "\nStarting time loop\n" << endl;  ←  Output some information
```

Implementing an application from scratch

- We are going to use the PISO control options, even if we do not have to deal with velocity-pressure coupling.

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

            #include "CourantNo.H"

            while (piso.correct())
            {
              while (piso.correctNonOrthogonal())
              {
                   fvScalarMatrix Teqn

                   (
                          fvm::ddt(T)
                          + fvm::div(phi, T)
                          - fvm::laplacian(DT, T)
                   );

                   TEqn.solve();
              }
            }
```

Time loop

Calculates and outputs the Courant Number

PISO options (correct loop)

PISO options (non orthogonal corrections loop)

Create object TEqn.
fvScalarMatrix is a scalar instance of fvMatrix

Model equation (convection-diffusion)
We need to create the scalar field T, vector field U (used in phi or face fluxes), and the constant DT.
We will declare these variables in the createFields.H header file.
In the dictionary fvSchemes, you will need to define how to compute the differential operators, that is,
- ddt(T)
- div(phi, T)
- laplacian(DT, T)

You will need to define the linear solver for T in the dictionary fvSolution

$$\frac{\partial T}{\partial t} + \nabla \cdot (\phi T) - \nabla \cdot (\Gamma \nabla T) = 0$$

Solve TEqn
At this point the object TEqn holds the solution.

Implementing an application from scratch

- We are going to use the PISO control options, even if we do not have to deal with velocity-pressure coupling.

```
        #include "continuityErrs.H"

        runTime.write();

    }

    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

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

    Info<< "End\n" << endl;

    return 0;
}
```

`#include "continuityErrs.H"` ← Computes continuity errors

`runTime.write();` ← Write the solution in the runtime folder
It will write the data requested in the file createFields.H

`}` ← At this point we are outside of the time loop

`Info<< nl << "ExecutionTime = " ...` ← Write CPU time at the end of the time loop.
If you want to compute the CPU time of each iteration, add the same statement inside the time loop

`Info<< "End\n" << endl;` ← Output this message

`return 0;` ← End of the program (exit status).
If everything went fine, the program should return 0.
To now the return value, type in the terminal,
$> echo $?

Implementing an application from scratch

- Let us create the file *createFields.H*, type in the terminal,

1. `$> touch createFields.H`

- Now open the file with your favorite editor, and start to add the following information,

```
    Info<< "Reading field T\n" << endl;

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

Create scalar field T

Create object for input/output operations

Name of the dictionary file to read/write

runtime directory

Object registry

Read the dictionary in the runtime directory (MUST_READ, and write the value in the runtime directory (AUTO_WRITE).
If you do not want to write the value, use the option NO_WRITE

Link object to mesh

Implementing an application from scratch

- Remember, in the file *createFields.H*, we declare all the variables (or fields) that we will use (U and T in this case).
- The dimensions of the fields are defined in the input dictionaries, you also have the option to define the dimensions in the source code.
- You can also define the fields directly in the source file *scratchFoam.C*, but it is good practice to do it in the header. This improves code readability.

```
    Info<< "Reading field U\n" << endl;

        volVectorField U          ← Create vector field U
        (
            IOobject
            (
                "U",              ← Name of the dictionary file to read/write
                runTime.timeName(),
                mesh,
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
            ),
            mesh
        );

```

Implementing an application from scratch

- We also need to declare the constant DT, that is read from the dictionary *transportProperties*.
- The dimensions are defined in the input dictionary.

```
    Info<< "Reading transportProperties\n" << endl;

        IOdictionary transportProperties
        (
            IOobject
            (
                "transportProperties",
                runTime.constant(),
                mesh,
                IOobject::MUST_READ_IF_MODIFIED,
                IOobject::NO_WRITE
            )
        );


        Info<< "Reading diffusivity DT\n" << endl;

        dimensionedScalar DT
        (
            transportProperties.lookup("DT")
        );

        #include "createPhi.H"

```

Create object transportProperties used to read data

Name of the input dictionary

Location of the input dictionary, in this case is located in the directory constant

Re-read data if it is modified

Do not write anything in the dictionary

Create scalar DT (diffusion coefficient)

Access value of DT in the object transportProperties

Creates and initializes the relative face-flux field phi.

Implementing an application from scratch

- At this point, we are ready to compile. Type in the terminal,

```
1. $> wmake
```

- If everything went fine, you should have a working solver named `scratchFoam`.
- If you are feeling lazy or you can not fix the compilation errors, you will find the source code in the directory,

    - **`$PTOFC/101programming/applications/solvers/scratchFoam`**

- You will find a case ready to run in the directory,

`$PTOFC/101programming/applications/solvers/scratchFoam/test_case`

- At this point, we are all familiar with the convection-diffusion equation and OpenFOAM®, so you know how to run the case. Do your magic.

Implementing an application from scratch

- Let us now add a little bit more complexity, a non-uniform initialization of the scalar field T.
- Remember **codeStream**? Well, we just need to proceed in a similar way.
- As you will see, initializing directly in the source code of the solver is more intrusive than using **codeStream** in the input dicitionaries.
- It also requires recompiling the application.
- Add the following statements to the *createFields.H* file, recompile and run again the test case.

```

forAll(T, i)
{
    const scalar x = mesh.C()[i][0];
    const scalar y = mesh.C()[i][1];
    const scalar z = mesh.C()[i][2];

    if ( 0.3 < x && x < 0.7)
        {
            T[i] = 1.;
        }
}
T.write();
```

We add the initialization of T after the its declaration

Access cell center coordinates.
In this case y and z coordinates are not used.

Conditional structure

Write field T. As the file createFields.H is outside the time loop the value is saved in the time directory 0

Implementing an application from scratch

- Let us compute a few extra fields. We are going to compute the gradient, divergence, and Laplacian of T.
- We are going to compute these fields in an explicit way, that is, after finding the solution of T.
- Therefore we are going to use the operator **fvc**.
- Add the following statements to the source code of the solver (*scratchFoam.C*),

```
         }

        #include "continuityErrs.H"
        #include "write.H"
        runTime.write();

    }
```

Add this header file

The file is located in the directory
$PTOFC/101programming/applications/solvers/scratchFoam
In this file we declare and define the new variables, take a look at it

- Recompile the solver and rerun the test case.
- The solver will complain, try to figure out what is the problem (you are missing some information in the *fvSchemes* dictionary).

Implementing an application from scratch

- Let us talk about the file *write.H*,

```
1     volVectorField gradT(fvc::grad(T));
2
3     volVectorField gradT_vector
4     (
5           IOobject
6           (
7                 "gradT",
8                 runTime.timeName(),
9                 mesh,
10                IOobject::NO_READ,
11                IOobject::AUTO_WRITE
12          ),
13          gradT
14    );
15
      ...
56
57    volScalarField divGradT
58    (
59          IOobject
60          (
61                "divGradT",
62                runTime.timeName(),
63                mesh,
64                IOobject::NO_READ,
65                IOobject::AUTO_WRITE
66          ),
67          fvc::div(gradT)
68    );
69    ...
```

Compute gradient of T.
fvc is the explicit operator, it will compute the requested value using the solution of T

Save vector field in output dictionary gradT

Compute divergence of gradT.
The output of this operation is a scalar field.
In this case we compute the quantity inside the scalar field declaration (line 67).
We use the fvc operator because the solution of gradT is already known.

In the dictionary fvSchemes, you will need to tell the solver how to do the interpolation of the term div(grad(T))

Roadmap

1. ~~Programming in OpenFOAM®. Building blocks.~~
2. ~~Implementing boundary conditions using high level programming~~
3. ~~Modifying applications – Highlights~~
4. ~~Implementing an application from scratch~~
5. **Adding the scalar transport equation to icoFoam**

Adding the scalar transport equation to icoFoam

- Let us modify a solver, we will work with **icoFoam**.
- We will add a passive scalar (convection-diffusion equation).
- At this point, you can work in any directory. But we recommend you to work in your OpenFOAM® user directory, type in the terminal,

```
1. $> cd $WM_PROJECT_USER_DIR/run
```

- Let us clone the original solver, type in the terminal,

```
1. $> cp -r $FOAM_APP/solvers/incompressible/icoFoam/ my_icoFoam
2. $> cd my_icoFoam
```

- At this point, we are ready to modify the solver.

Adding the scalar transport equation to icoFoam

- Open the file *icoFoam.C* using your favorite editor and add the new equation in lines 115-120,

```
111                 U = HbyA - rAU*fvc::grad(p);
112                 U.correctBoundaryConditions();
113             }
114

116             solve
117             (
118                 fvm::ddt(S1)
119                 + fvm::div(phi, S1)
120                 - fvm::laplacian(DT, S1)
121             );

122
123
124             runTime.write();
```

Scalar transport equation.
The name of the scalar is S1.
We need to declare it in the createFields.H file.
We also need to read the coefficient DT.

In the dictionary fvSchemes, you will need to define how to compute the differential operators, that is,

ddt(S1)
div(phi, S1)
laplacian(DT, S1)

You will need to define the linear solver for S1 in the dictionary fvSolution

- As the passive scalar equation depends on the vector field U, we need to add this equation after solving U.

Adding the scalar transport equation to icoFoam

- Open the file *createFields.H* using your favorite editor and add the following lines at the beginning of the file,

```

    volScalarField S1
    (
            IOobject
            (
                    "S1",
                    runTime.timeName(),
                    mesh,
                    IOobject::MUST_READ,
                    IOobject::AUTO_WRITE
            ),
            mesh
    );

    Info<< "Reading diffusionProperties\n" << endl;

    IOdictionary diffusionProperties
    (
            IOobject
            (
                    "diffusionProperties",
                    runTime.constant(),
                    mesh,
                    IOobject::MUST_READ_IF_MODIFIED,
                    IOobject::NO_WRITE
            )
    );

    Info<< "Reading diffusivity DT\n" << endl;
    dimensionedScalar DT
    (
            diffusionProperties.lookup("DT")
    );
```

Declaration of scalar field S1.
The solver will read the input file S1 (BC and IC).
You will need to create the file S1 in the time directory 0.

Declaration of input/output dictionary file.
The name of the dictionary is diffusionProperties and is located in the directory constant.

Read DT value from the dictionary diffusionProperties.

Adding the scalar transport equation to icoFoam

- Those are all the modifications we need to do.
- But before compiling the new solver, we need to modify the compilation instructions.
- Using your favorite editor, open the file *`Make/files`*,

Original file

```
icoFoam.C

EXE = $(FOAM_APPBIN)/icoFoam
```

Modified file

```
icoFoam.C

EXE = $(FOAM_USER_APPBIN)/my_icoFoam
```

`icoFoam.C` ← Name of the input file

`my_icoFoam` ← Name of the executable. To avoid conflicts with the original installation, we give a different name to the executable

`$(FOAM_USER_APPBIN)` ← Location of the executable. To avoid conflicts with the original installation, we install the executable in the user's personal directory

Adding the scalar transport equation to icoFoam

- At this point we are ready to compile, type in the terminal,

```
1. | $> wmake
```

- If everything went fine, you should have a working solver named `my_icoFoam`.

- If you are feeling lazy or you can not fix the compilation errors, you will find the source code in the directory,

    - **`$PTOFC/101programming/applications/solvers/my_icoFoam`**

- You will find a case ready to run in the directory,

`$PTOFC/101programming/applications/solvers/my_icoFoam/test_case`

Adding the scalar transport equation to icoFoam

Running the case

- This case is ready to run, the input files are located in the directory **`$PTOFC/101programming/applications/solvers/my_icoFoam/test_case`**
- To run the case, type in the terminal,

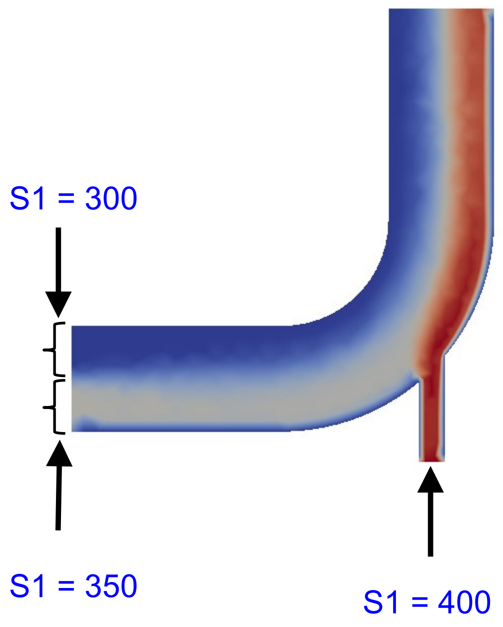
```
1. $> foamCleanTutorials
2. $> fluentMeshToFoam ../../../../../meshes_and_geometries/fluent_elbow2d_1/ascii.msh
3. $> my_icoFoam | tee log
4. $> paraFoam
```

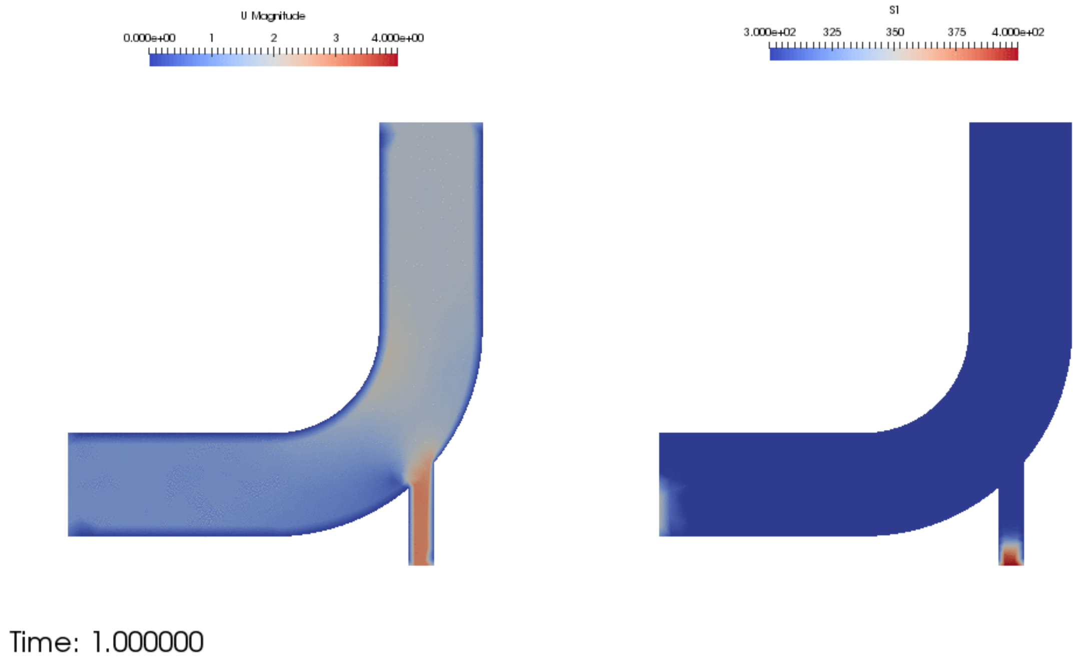
- Remember, you will need to create the file *`0/S1`* (boundary conditions and initial conditions for the new scalar).
- You will also need to create the input dictionary *`constant/diffusionProperties`*, from this dictionary we will read the diffusion coefficient value.
- Finally, remember to update the files *`system/fvSchemes`* and *`system/fvSolution`* to take into account the new equation.

Adding the scalar transport equation to icoFoam

Running the case

- If everything went fine, you should get something like this





S1 inlet values

Visualization of velocity magnitude and passive scalar S1
www.wolfdynamics.com/wiki/BCIC/2delbow_S1