Implementing an application from scratch

- 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.

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, this does not do anything. We need to add the statements to create a working applications.
- This is the starting point for new applications.

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.

```

#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;
}

```

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"
33
34   // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
35
36   int main(int argc, char *argv[])
37   {
38       #include "setRootCase.H"
39       #include "createTime.H"
40       #include "createMesh.H"
41       #include "createFields.H"
42       #include "CourantNo.H"
43       #include "initContinuityErrs.H"
44
45       pisoControl piso(mesh);
46
47       Info<< "\nStarting time loop\n" << endl;
48
```

- Line 32: Solution control using PISO class
- Line 38: Set directory structure
- Line 39: Create time (object runtime)
- Line 40: Create time (object mesh)
- Line 41: Initialize fields. This source file does not exist yet, we need to create it
- Line 42: Calculates and outputs the Courant Number
- Line 43: Declare and initialize the cumulative continuity error
- Line 45: Assign PISO controls to object mesh. Creates object piso. Alternatively, you can use the header file createControl.H
- Line 47: 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)

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

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;
}
```

Computes continuity errors

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

Output this message

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 $?

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

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 a 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))