TME205 OpenSource CFD: An OpenFOAM tutorial

A.Berce
Solid and Fluid Mechanics
Chalmers University of Technology, Göteborg, Sweden
Revieved by: M.Hammas and J.Andric

November 3, 2010

Contents

Abstract

This is the report of the project-course *TME205 - CFD with OpenSource software, 2010* given by *Håkan Nilsson, Applied Mechanics, Chalmers*. The goal of the project is to get an understanding of *C++* programming in OpenFOAM. I would like to thank the administrators *Håkan Nilsson* and *Jelena Andric* for all the support during the development. Also, I would like to thank the reviewers, *Martin Hammas* and *Jelena Andric*, for increasing the quality of this report through good feedback and critique. Going into this project I had no experience of C++ and it has been a very worthwhile course.

This report will focus on one solver and two utilities. The solver, *icoFoam*, is an incompressible, laminar, transient CFD solver and the utilities are *icoErrorEstimate* and *refineMesh*. The first utility is used to estimate the error of the incompressible momentum equation and the second one is used to refine the mesh according to some specifications. The aim is to eventually combine these three applications into a solver that automatically refines the computational domain in the areas where the magnitude of the error is largest.

The applications are first studied in detail and short tutorials of the usage is presented. In these tutorials the applications are used independently of each other to get an understanding of their functions. While studying the first three sections in this report the reader should keep the purpose of this project in mind and try to think about how the different functions can be integrated. Then in *Section* 4 the author suggests a solution to the formulated problem.

The developments has been done for *OpenFOAM 1.7.x*.

1 icoFoam

Here the Transient CFD solver for incompressible, laminar flow of Newtonian fluids *icoFoam* is described in detail. The main code in `icoFoam.C` and a reasonable amount of sub codes are gone through in a systematic manner. This is a simple application that solves the incompressible momentum equation once and a pressure equation a prescribed number of times per timestep.

```
#include "fvCFD.H"
```

Code 1: Standard in OpenFOAM codes

Code 1 includes all the standard header files and sets up the basic OpenFOAM environment. It is not something to consider any closer. After entering the main function a few other header files are included according to *Code* 2

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

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

Code 2: Header files

The `setRootCase.H` controls that `icoFoam` is called in the correct directory, namely the case directory. If that is false it will break the process and return a `FOAM FATAL IO ERROR` message. The second include statement `createTime.H` which defines the `runTime` and prints a message "Create time" to the prompt.

The third header file called `createMesh.H` creates the mesh using the `fvMesh` class. `createFields.H` reads the initial conditions specified in the `0`-directory and connect them to the mesh. The final header `initContinuityErrs.H` declares and initializes the cumulative continuity error.

```
Info<< "\nStarting time loop\n" << endl;
```

Code 3: Information to the user

Code 3 tells the user that the program is now entering the time loop. This loop runs according to *Code* 4

```
while (runTime.loop())
```

Code 4: Time loop

In *Code* 4 `runTime.loop()` is a boolean and the while-loop is run as long as it has a true argument. Later it will be seen that when the `endTime` is reached this boolean will be set to false.

The first line inside the time-loop is an `Info` statement which prints the current time object called `runTime.timeName()` to the prompt, the syntax for this is shown in *Code* 5.

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

Code 5: Runtime info

Then two more header files are included; one that makes a dictionary called `piso` and reads from it, and one that calculates the Courant number on all internal cells and prints it to the prompt. Now the first equation is defined; the momentum equation. The equation is stated in regular math notation in *Equation* 1

$$\frac{\partial U}{\partial t} + \nabla \cdot (\phi U) - \nabla \cdot (\nu \nabla U) = -\nabla p \tag{1}$$

In *Code* 6 the OpenFOAM syntax of *Equation* 1 is stated. It can clearly be seen that `ddt` corresponds to a time-derivative, `div` to a divergence, in this sence the *OpenFOAM* -syntax is quite close to regular mathematical notation.

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

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

Code 6: OpenFOAM syntax of U-equation

The solve command runs several files that together solve the system of equations, examples of functions used are `fvMatrixSolve.C` and `gvcGrad.C`.

Next the PISO-loop (*Pressure Implicit with Splitting of Operators*) is implemented. This equation is iterated inside the time-loop and the number of iterations is specified in `fvSolution` in the system directory of the case. In *Code* 7 the entire PISO-loop is stated.

```
 67         for (int corr=0; corr<nCorr; corr++)
 68         {
 69             volScalarField rUA = 1.0/UEqn.A();
 70
 71             U = rUA*UEqn.H();
 72             phi = (fvc::interpolate(U) & mesh.Sf())
 73                 + fvc::ddtPhiCorr(rUA, U, phi);
 74             adjustPhi(phi, U, p);
 75             //Debugger info
 76           Info << "Debugger info: rUA.dimensions() "<< rUA.dimensions() <<endl;
 77           Info << "Debugger info: UEqn.dimensions()"<< UEqn.dimensions()<<endl;
 78           Info << "Debugger info: U.dimensions()   "<< U.dimensions()   <<endl;
 79
 80
 81             for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
 82             {
 83                 fvScalarMatrix pEqn
 84                 (
 85                     fvm::laplacian(rUA, p) == fvc::div(phi)
 86                 );
 87
 88                 pEqn.setReference(pRefCell, pRefValue);
 89                 pEqn.solve();
 90
 91                 if (nonOrth == nNonOrthCorr)
 92                 {
 93                     phi -= pEqn.flux();
 94                 }
 95             }
 96
 97             #include "continuityErrs.H"
 98
 99             U -= rUA*fvc::grad(p);
100             U.correctBoundaryConditions();
101         }
```

Code 7: The PISO loop

Note that a few extra lines of code called `//Debugger info` are included. This will print the dimensions of the three different variables. In *Code* 8 the extra part of the resulting `logSolve` file when running `icoFoamDebugger |tee log_solve` is stated. Prior to running this simulation a new solver was compiled, the only difference between the original `icoFoam` and the revised one is in these extra information outputs.

```
Debugger info: rUA.dimensions()       [0 0 1 0 0 0 0]
Debugger info: UEqn.dimensions()      [0 4 -2 0 0 0 0]
Debugger info: U.dimensions()         [0 1 -1 0 0 0 0]
```

Code 8: Extra output from PISO-loop

The dimensions shows that rUA has units $[s]$, UEq $[m^4/s^2]$ etc. according to *Table* 1

No.	Property	Unit	Symbol
1	Mass	kilogram	k
2	Length	meter	m
3	Time	second	s
4	Temperature	kelvin	K
5	Quantity	moles	mole
6	Current	ampere	A
7	Luminous Intensity	candela	cd

Table 1: Dimensions in OpenFOAM

Adding similar expressions as the above is a way of debugging the program and it is quite good for a beginner since it lets the user interact with the program and get some feedback on the results of the added code.

The piso loop (*Code* 7) can be summed up into the following steps:

67-68 : Initialize loop, iterate `nCorr` times
69-71 : Calculate a_p (diagonal coefficient) and then U
72-74 : Calculate and adjust the flux
75-80 : Extra code added to get some extra output
81-95 : Define and solve the pressure equation, repeat `nNonOrthCorr` times
91-94: Correct the flux
96-98 : Calculate continuity error and output to prompt
99-100: Perform momentum corrector step

2 icoErrorEstimate

This post processing utility is simple to use. Its purpose is to estimate the error of each time in a given solution. As implied by its name it is used on the solutions created by the solver `icoFoam`. It loops through all the time directories in the case directory, calculates the error and saves it as a new object. To illustrate the usage of `icoErrorEstimate` the following tutorial has been composed.

2.1 Error estimation on the cavity case

First the cavity tutorial is run as usual according to *Code* 9.

```
run
cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavity/ .
cd cavity
blockMesh
icoFoam |tee logSolve
```

Code 9: Running the cavity case

Now the regular results can be viewed in *paraFoam* . Lets use the utility to estimate the error, this is done simply by the syntax `icoErrorEstimate` inside the case directory. The results can now be viewed in *paraFoam* (see *Figure* 1). The errors are printed to file as a field variable similar to the velocity- and pressure fields, and hence it appears in the *Object Inspector*. The user is now able to visualize the error to see where in the domain the solution is flawed.

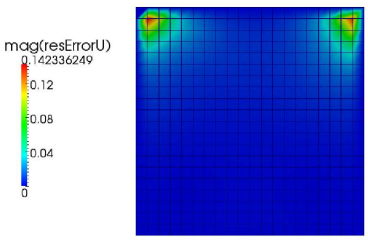


Figure 1: Magnitude of error in the cavity case

In the next subsection a deeper look into the code that we just ran is presented.

2.2 A look into icoErrorEstimate

As said, `icoErrorEstimate` loops through the time directories of a solution and calculates the error for each time. However, one could question how the error is calculated. Lets have a look in `icoErrorEstimate.C` by typing

```
vim $FOAM_UTILITIES/errorEstimation/icoErrorEstimate/icoErrorEstimate.C
```

The first piece of code that catches the eye is `timeSelector::addOptions();`. This calls on a function inside `timeSelector.C` located in `$FOAM_SRC/OpenFOAM/db/Time/` shown in *Code* 10

```
118 void Foam::timeSelector::addOptions
119 (
120     const bool constant,
121     const bool zeroTime
122 )
123 {
124     if (constant)
125     {
126         argList::validOptions.insert("constant", "");
127     }
128     if (zeroTime)
129     {
130         argList::validOptions.insert("zeroTime", "");
131     }
132     argList::validOptions.insert("noZero", "");
133     argList::validOptions.insert("time", "ranges");
134     argList::validOptions.insert("latestTime", "");
135 }
```

Code 10: timeSelector::addOptions()

As can be seen this member function in class `timeSelector` takes two boolean arguments, since none of them are specified when calling upon the function they will both be `false` and the if-statements are not entered. The three following `argList::validOptions.insert` are executed and their purpose is to add arguments to a list which defines which times will be treated.

Next a function in the same class as *Code* 10 is called. This takes `runTime` and `args` as input and returns a list of the time-directories. The function is given in *Code* 11. It can also be seen that if the directory does not contain any time directories an error message will be returned and the utility aborted.

```
230 Foam::List<Foam::instant> Foam::timeSelector::select0
231 (
232     Time& runTime,
233     const argList& args
234 )
235 {
236     instantList timeDirs = timeSelector::select(runTime.times(), args);
237
238     if (timeDirs.empty())
239     {
240         FatalErrorIn(args.executable())
241             << "No times selected"
242             << exit(FatalError);
243     }
244
245     runTime.setTime(timeDirs[0], 0);
246
247     return timeDirs;
248 }
```

Code 11: timeSelector::select0()

After setting up the rules for the calculations the mesh is defined as discussed in *Section* 1 with `# include "createMesh.H"` and some information is displayed to the user through the `Info` statement. Before entering the loop through the time directories a dictionary containing the transport properties is defined. This simply reads the file called `transportProperties` located in the `constant` directory. Then `nu` is read from the newly created dictionary. Now lets enter the loop through the time directories. The whole loop is given in *Code* 12

```
 69     forAll(timeDirs, timeI)
 70     {
 71         runTime.setTime(timeDirs[timeI], timeI);
 72
 73         Info<< "Time = " << runTime.timeName() << endl;
 74
 75         mesh.readUpdate();
 76
 77         IOobject pHeader
 78         (
 79             "p",
 80             runTime.timeName(),
 81             mesh,
 82             IOobject::MUST_READ
 83         );
 84
 85         IOobject Uheader
 86         (
 87             "U",
 88             runTime.timeName(),
 89             mesh,
 90             IOobject::MUST_READ
 91         );
 92
 93         if (pHeader.headerOk() && Uheader.headerOk())
 94         {
 95             Info << "Reading p" << endl;
 96             volScalarField p(pHeader, mesh);
 97
 98             Info << "Reading U" << endl;
 99             volVectorField U(Uheader, mesh);
100
101 #           include "createPhi.H"
102
103             errorEstimate<vector> ee
104             (
105                 resError::div(phi, U)
106               - resError::laplacian(nu, U)
107              ==
108                -fvc::grad(p)
109             );
110
111             volVectorField e = ee.error();
112             e.write();
113             mag(e)().write();
114         }
115         else
116         {
117             Info<< "    No p or U" << endl;
118         }
119
120         Info<< endl;
121     }
```

Code 12: icoErrorEstimate.C - Loop over time directories

The `forAll`-statement shows that the loop will go through all the times in `timeDirs`, hence all the time directories. `timeI` is the counter. Then the `runTime` variable is set accordingly. An Info statement is displayed to the user and then the loop calls upon a function in the mesh class as `mesh.readUpdate();`. Let's look deeper into that.

The function is a member function defined in `polyMesh.H` and there it states that the function updates the mesh according to the mesh files saved in time directories. Since, in the cavity case, we now have the same mesh throughout the time span the `mesh.readUpdate();` will do nothing.

Moving further in the loop we see that two objects are defined; `pHeader` and `Uheader`. Note that the `MUST_READ` is activated which tells *OpenFOAM* to read the files in the time directories called `"p"` and `"U"`. If they exist an Info statement is prompted to the user and the objects `U` and `p` are declared.

The actual error calculation is now about to start, the *OpenFOAM* -syntax of this can be seen on line 103-109 in *Code* 12. Note the similarity to the `UEqn` in the `icoFoam` solver, *Code* 6. Instead of solving the equation through matrices (as in `icoFoam`) the error estimation is done by use of the `resError`-class which calculates the residual error for each cell (see files in directory `$FOAM_SRC/errorEstimation/errorEstimate`). Then the error and the magnitude of the error are stored in each time directory in the same manner as a regular variable (the files are called `resErrorU` and `mag(resErrorU)`)

3 refineMesh

This sections treats the *refineMesh* utility, first the usage of it and then the source code. The aim is to get a good understanding of what is happening when a refinement is made to be able to translate the vital parts of this utility into a dynamic mesh refinement.

3.1 Using refineMesh utility tutorial

This will be done on the cavity-case. We will first solve on an original coarse mesh and then two different refinement options will be presented; refining the whole mesh AND refining a section of the mesh. Let's start with copying the cavity case and solving it on the original mesh according to *Code* 9 in *Section* 2.1. Now, standing in the cavity-directory, do the following.

```
run
rm -rf cavity
cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavity .
mv cavity cavityOrig
cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavity .
mv cavity cavityRefineWhole
cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavity .
mv cavity cavityRefineSet
cd cavityOrig
blockMesh
icoFoam
```

Code 13: Set up refine tutorial

3.1.1 Refine whole mesh

This is a quite easy way to refine the mesh. The standard is to split once in the x- and once in the y-direction (for a 2D case). We shall now refine the mesh, map the results of the original mesh and run icoFoam again.

```
run
cd cavityRefineWhole
rm -rf 0.*
blockMesh |tee logMesh
refineMesh -overwrite
```

Code 14: Refine whole mesh

The overwrite flag tells the utility to replace the existing polyMesh with the refined one. If this flag would not be added a new time directory named after the timestep would be created, this new time directory would only contain the refined polyMesh. Now lets map the original results to the fine mesh. First we need to make sure that the new solution will start at the `endTime` of the previous, hence change the `startTime` in the controlDict to 0.5 and the `endTime` to 0.7. Since the cell length is now half of the previous the timestep should be lowered so that the solver remains stable (it should not transport properties more than one cell-length per timestep, Courant number limitation). Set `deltaT` to 0.0025. One more important note is that the refineMesh has now created a polyMesh directory (containing a cellMap) inside the 0 directory, this needs to be removed to be able to use mapFields. Now lets map the results from the original mesh. Run *Code* 15 standing in the cavityRefineWhole-directory.

```
rm -r 0/polyMesh
mapFields ../cavityOrig -sourceTime latestTime -consistent
icoFoam
```

Code 15: Map and run refineWhole case

The results can now be viewed in *paraFoam* .

3.1.2 Refine set

This part of the tutorial is slightly more complicated. We are going to select a part of the mesh and refine it. This is done by defining a `cellSet` and calling `refineMesh` with a dictionary flag. Go into the `cavityRefineSet`-directory and copy the dictionary needed.

```
run
cd cavityRefineSet
cp $FOAM_UTILITIES/mesh/manipulation/refineMesh/refineMeshDict system/
vim system/refineMeshDict
```

Code 16: refineMeshDict

Inside the refineMeshDict several things are specified. The `cellSet` we are going to refine will be called `c0` and the directions to refine should be `tan1` and `tan2` which are the x- and y-directions. Note that you need to modify this in the `refineMeshDict` because a third direction is specified under "`// List of directions to refine`" in the original file.

Now the `cellSet` should be created. There is a utility for this but now we will create it by hand. First create the `sets`-directory

```
mkdir constant/polyMesh/sets/
vim constant/polyMesh/sets/c0
```

Code 17: Create cellSet

Now the set will be specified. The cells chosen will be the 40 top cells of the domain (two rows closest to the lid). Copy *Code* 18 into `c0`.

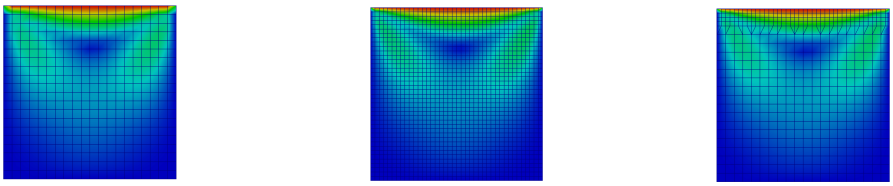
```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  1.7.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       cellSet;
    location    "constant/polyMesh/sets";
    object      c0;
}

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
40 //number of cells to refine
(
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
)
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

Code 18: cellSet c0 to refine

Make sure that `blockMesh` has been executed in the `cavityRefineSet`-directory, otherwise there would be no mesh to refine. Now run `refineMesh -dict -overwrite` to create the new mesh. Have a look at the new mesh in *paraFoam* before mapping the results. Next lets map the results from `cavityOrig` on the new mesh. In `system/controlDict`, set the `startTime` to 0.5, the `endTime` to 0.7, `deltaT` to 0.0025 and repeat *Code* 15

This concludes this part of the tutorial. View and compare the results in *paraFoam* and note that this refinement is not preferred to graded meshes because of the fact that the area ratio in the interface between the refined and non-refined cells is inevitably 4. This is not good because the finite volume schemes in CFD-solvers uses cell length to interpolate the variables in the cells. The rapid change in cell-size could lead to a small discontinuity in the domain.



(a) cavityOrig (b) cavityRefineWhole (c) cavityRefineSet

Figure 2: Comparison of different refinements

3.2 A look into the source code of refineMesh

The tutorial in the previous section requires no knowledge of the source code of the refineMesh utility. In this section a not too deep study of the source code of this utility is presented. Main focus will be put on `refineMesh.C` since this is the main program of this utility. This file can be found in `$FOAM_UTILITIES/mesh/manipulation/refineMesh/` which is the directory of the studied utility.

This is a quite long and complicated function so we will focus on the main program. Main starts at line 291 in `refineMesh.C` and before that functions for checking edges, axis etc. are declared. The first few lines of code in the main program is stated in *Code* 19

```
291 int main(int argc, char *argv[])
292 {
293     Foam::argList::validOptions.insert("dict", "");
294     Foam::argList::validOptions.insert("overwrite", "");
295
296 #   include "setRootCase.H"
297 #   include "createTime.H"
298     runTime.functionObjects().off();
299 #   include "createPolyMesh.H"
300     const word oldInstance = mesh.pointsInstance();
301
302     printEdgeStats(mesh);
303
304
305     //
306     // Read/construct control dictionary
307     //
308
309     bool readDict = args.optionFound("dict");
310     bool overwrite = args.optionFound("overwrite");
311
312     // List of cells to refine
313     labelList refCells;
314
315     // Dictionary to control refinement
316     dictionary refineDict;
```

Code 19: First part of main in refineMesh.C

The first part of *Code* 19 makes a list of the arguments on which the function is called upon. In the beginning of *Section* 3 these arguments have been specified in a few different ways. Then some includements are made which purposes are to check if the function is called from the correct case-directory and creating the time object. On line 300 a quite interesting function is called, its purpose is to create a string that has the name of the directory where the `points` file is located, hence the path to the `polyMesh`-directory. *OpenFOAM* now knows where the `polyMesh` is located.

The `printEdgeStats`-function is defined in the header of `refineMesh.C` and when called with the argument `mesh` it will print out some statistics of the mesh. In the next step a dictionary that will control the refinement is to be created/read depending on if `refineMesh` was called with the argument `-dict` or not.

The `-overwrite` argument specifies if the `constant/polyMesh`-directory should be replaced by the new mesh or not. `refCells` is a list of all the cells, unless a `cellSet` is specified in the `refineMeshDict` (Note: This will be manipulated to contain cells with large error in *Section* 4).

Code 20 will be executed if the `-dict` flag was used when calling `refineMesh`. It will set up a local dictionary called `refineDict` that will ultimately be used in the actual refinement. If the boolean `readDict` is false the code in the `else`-box will be executed instead. This part is treated in *Section* 4 as it will be used to create a special dictionary for the dynamic mesh refinement. In the original utility, however, it will use the standard settings to refine all the cells in the

domain.

```
318     if (readDict)
319     {
320         Info<< "Refining according to refineMeshDict" << nl << endl;
321
322         refineDict =
323             IOdictionary
324             (
325                 IOobject
326                 (
327                     "refineMeshDict",
328                     runTime.system(),
329                     mesh,
330                     IOobject::MUST_READ,
331                     IOobject::NO_WRITE
332                 )
333             );
334
335         word setName(refineDict.lookup("set"));
336
337         cellSet cells(mesh, setName);
338
339         Pout<< "Read " << cells.size() << " cells from cellSet "
340             << cells.instance()/cells.local()/cells.name()
341             << endl << endl;
342
343         refCells = cells.toc();
344     }
345     else
346 { Dictionary will be created from scratch }
```

Code 20: If -dict flag is used

After setting up the `refineDict` it will be used to define the refinement. The program then decides if the old mesh should be overwritten or if the new mesh should be put in a new time-directory with name one timestep larger than the current one. The `overwrite`-boolean is used to accomplish this according to *Code* 21

```
415     string oldTimeName(runTime.timeName());
416
417     if (!overwrite)
418     {
419         runTime++;
420     }
```

Code 21: Overwrite old polyMesh or not

Now all the necessary specifications are set and the code is ready to split the cells. This is done by calling a function in the class `Foam::multiDirRefinement` and the input arguments are `mesh`,

`refCells` and `refineDict`. The third one was specified in *Code* 20. The syntax of calling the refinement function is shown in *Code* 22. This piece of code also writes out the mesh in the (by boolean `overwrite`) specified directory. This report will not present the code inside the function `multiDirRefinement` since the later modification will only modify the arguments which this function is using. If the reader wishes to examine the contents of `multiDirRefinement` the source code can be found in `$FOAM_SRC/dynamicMesh/meshCut/meshModifiers/multiDirRefinement`

```
423     // Multi-directional refinement (does multiple iterations)
424     multiDirRefinement multiRef(mesh, refCells, refineDict);
425
426
427     // Write resulting mesh
428     if (overwrite)
429     {
430         mesh.setInstance(oldInstance);
431     }
432     mesh.write();
```

Code 22: Calling the refinement function

The rest of the code in `refineMesh.C` is devoted to storing information. `LabelLists` of added cells, a `cellSet` containing the new cells and a map-list is created.

Enough understanding has now bee gained to make an attempt to create an application which integrates parts of the treated utilities to a solver that automatically refines the mesh in the areas where the magnitude of the error is largest.

4 icoFoamErrorRefine

This section is divided into a few subsections; firstly the demands on the solver are specified in *Section* 4.1, the developments are made in *Section* 4.2, then a short tutorial of how to implement the code on the cavity case is presented in *Section* 4.3. Finally a discussion concerning the modifications and further developments is presented in *Section* 4.4.

In [1] an adaptive method of mesh refinement and coarsening based on the error has been done so the application about to be developed in this tutorial will not contribute to anything new in the world of *OpenFOAM* . The aim is instead to get a deeper understanding of the built-in functions of the three treated applications and try to blend them into a new application. Also, to learn about developing and debugging.

4.1 Goals

The application to be developed should be applicable to an arbitrary 2D mesh created by the `blockMesh` utility. The outline of the, so far imaginary, application is specified by the flowchart in *Figure* 3.

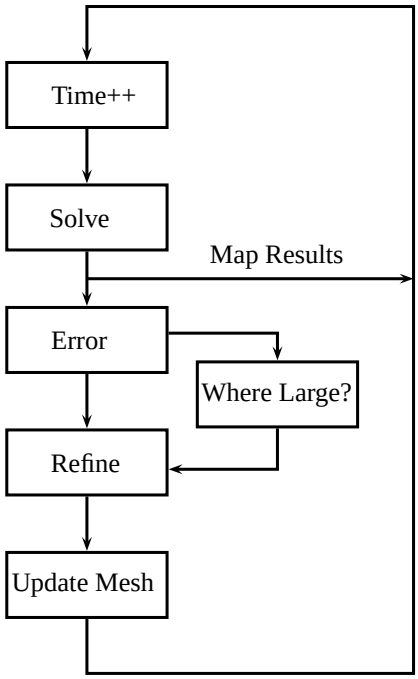


Figure 3: Flowchart icoFoamDynamicRefine

Described in words this application should firstly solve on an original mesh using the original `icoFoam`. Then with parts of the `icoErrorEstimate`-utility it will estimate the error in each cell and then we will add some own developed code to find the cells with the largest errors. Here one of two options should be applicable, either the solver finds the 10 cells with the largest errors OR it finds all cells with error larger than a specified value `errTol`.

Identify and put the labels of the cells to be refined in a list for the refinement utility to use. Then we specify how the refinement shall be done and perform it using parts of the refineMesh

utility. Finally the program should map the results from the previous mesh to the refined one and repeat the process.

The refinement should be done once every `writeInterval` specified in `system/controlDict`.

4.2 Developments

This section will be formed as a tutorial. The reader should be able to study this chapter, follow the steps and end up with a new functional solver. The solver will be based on the `icoFoam` application, so lets copy that into the user applications directory , see *Code* 23.

```
run
cd ../applications/
mkdir -p solvers/incompressible/icoFoamErrorRefine
cd solvers/incompressible/icoFoamErrorRefine
cp -r $FOAM_SOLVERS/incompressible/icoFoam/* .
rm *.dep
rm -r Make/linux*
mv icoFoam.C icoFoamErrorRefine.C
sed -i s/"icoFoam"/"icoFoamErrorRefine"/g Make/files
sed -i s/"FOAM_APPBIN"/"FOAM_USER_APPBIN"/g Make/files
```

Code 23: Copy icoFoam

In createFields, add *Code* 24 on line 48 (after the volVectorField U is created).

```
    //*********NEW*****************
    volVectorField err
    (
        IOobject
        (
            "err",
            runTime.timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh
    );
    // ****************************/
```

Code 24: Add to createFields.C

We are going to create an additional mesh, called `newMesh`, inside the solver. Hence we need to use a modified version of `createMesh.H`. Lets copy it into the `icoFoamErrorRefine`-directory, follow *Code* 25

```
cp $FOAM_SRC/OpenFOAM/include/createMesh.H .
mv createMesh.H createNewMesh.H
sed -i s/"mesh"/"newMesh"/g createNewMesh.H
```

Code 25: Copy createMesh.H

Now we will start to modify `icoFoamErrorRefine.C` . First, we need a few extra header files for the additional features. Hence, add *Code* 26 on line 33 (just after `#include "fvCFD.H"`) inside `icoFoamErrorRefine.C`.

```
// Error Estimation
#include "errorEstimate.H"
#include "resError.H"

// Mesh refinement
#include "multiDirRefinement.H" // For the actual refinement
#include <vector>               // For creating standard vectors

// Map Mesh
//#include "fvMesh.H"
//#include "IOobjectList.H"
//#include "meshToMesh.H"
//#include "MapVolFields.H"
//#include "MapConsistentVolFields.H"
//#include "UnMapped.H"
//#include "processorFvPatch.H"
//#include "mapLagrangian.H"
```

Code 26: Extra headers in icoFoamErrorRefine.C

We will also add an extra function for the mapping in the header of `icoFoamErrorRefine.C`. Hence, add *Code* 27 on line 47 (before main). This function, called `mapConsistentMesh`, is a part of the `mapFields`-utility whos source code can be found in
`$FOAM_UTILITIES/preProcessing/mapFields/`
It takes two fvMesh-objects and creates interpolation schemes, finds all field-variables (like `U` and `p`) by searching through the case-directories that the two fvMesh-objects belong to. The difference from the `mapFields`-utility and the `icoErrorRefine`-solver is the original `mapFields`-utility maps from one case to another and `icoErrorRefine` should map inside one case. The two last lines in *Code* 27 define two object that will be used to define the refinement. Two different refinement methods can be used. The first alternative refines all cells that have an error-magnitude larger than `errTol` and the second alternative refines the 10 cells who have the largest error-magnitude. If the boolean `refine10` is true the second alternative is used. If it is false the first alternative is used.

```
// Define mapConsistentMesh
/*void mapConsistentMesh
(
    const fvMesh& meshSource,
    const fvMesh& meshTarget
)
{
    // Create the interpolation scheme
    meshToMesh meshToMeshInterp(meshSource, meshTarget);

        Info<< nl
        << "Consistently creating and mapping fields for time "
        << meshSource.time().timeName() << nl << endl;

    {
        // Search for list of objects for this time
        Foam::IOobjectList objects(meshSource, meshSource.time().timeName());
        // Map volFields
        // ~~~~~~~~~~~~~
        MapConsistentVolFields<scalar>(objects, meshToMeshInterp);
        MapConsistentVolFields<vector>(objects, meshToMeshInterp);
        MapConsistentVolFields<sphericalTensor>(objects, meshToMeshInterp);
        MapConsistentVolFields<symmTensor>(objects, meshToMeshInterp);
        MapConsistentVolFields<tensor>(objects, meshToMeshInterp);
    }

    {
        // Search for list of target objects for this time
        IOobjectList objects(meshTarget, meshTarget.time().timeName());

        // Mark surfaceFields as unmapped
        // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        UnMapped<surfaceScalarField>(objects);
        UnMapped<surfaceVectorField>(objects);
        UnMapped<surfaceSphericalTensorField>(objects);
        UnMapped<surfaceSymmTensorField>(objects);
        UnMapped<surfaceTensorField>(objects);

        // Mark pointFields as unmapped
        // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        UnMapped<pointScalarField>(objects);
        UnMapped<pointVectorField>(objects);
        UnMapped<pointSphericalTensorField>(objects);
        UnMapped<pointSymmTensorField>(objects);
        UnMapped<pointTensorField>(objects);
    }

    mapLagrangian(meshToMeshInterp);
}
*/
// Extra objects for setting up refinement
static const scalar errTol = 1E-5;
static bool refine10 = true;
```

Code 27: Function mapConsistentMesh

Now we enter main. In the initial includes, add

```
#include "createNewMesh.H" //Create object newMesh
```

just after the regular `createMesh.H`. The original code from `icoFoam` is not alternated and the next modification will be done before the end of the time-loop (after the Info statements after `runTime.write()`). Add *Code* 28 in that position, this code is based on the icoErrorEstimate-utility (see `icoErrorEstimate.C`). Notice the if-statement at the top. This means that the error will only be calculated when the code uses `runTime.write()`, hence every `writeInterval` timestep.

```
    //******** Error Estimation *********
    if(runTime.write())
    {
        errorEstimate<vector> ee
        (
            resError::div(phi, U)
          - resError::laplacian(nu, U)
         ==
           -fvc::grad(p)
        );

        volVectorField err = ee.error();
        err.write();
        mag(err)().write();
```

Code 28: Estimate the error

Now it is time to define the refinement. In the same manner as the refineMesh-utility we define a list of cells to refine and a dictionary that controls the refinement. Lets start with finding out which cells to refine, loop through all cells and pick out the 10 cells with the largest error-magnitude OR pick all cells with error-magnitude larger than the specified error tolerance `errTol`. Add *Code* 29 after the recently added *Code* 28

```
const cellList& cells = mesh.cells();   //List to loop through
labelList refCells;                     //List of cells to refine
refCells.clear();                       //Clear the list

if(refine10)
{
    std::vector<double> refCellsVekt(10,0);
    forAll(cells,cellI)
    {
        int n = 0;
        while(n < 10)
        {
//Info << "cellI = " << cellI << endl;
//Info << "n = " << n << endl;
//Info << "mag(err[cellI]) = " << mag(err[cellI]) << endl;
//Info << "mag(err[refCellsVekt[n]]) = " << mag(err[refCellsVekt[n]]) << endl;
            if(mag(err[cellI]) > mag(err[refCellsVekt[n]]))
            {
                refCellsVekt[n]=cellI;
                //Info << "****************************************BREAKING"<<endl;
                break;
            }
            n++;
            if(n==10)
            {
                //Info << "****************************************"<<endl;
            }
        }
    }
    // refCellVekt should now contain the cellnumbers of the 10
    // cells that have the largest error
    // Now put those cellnumbers in the refCells list
    forAll(refCellsVekt,i)
    {
        refCells.resize(i,refCellsVekt[i]);
    }
    Info << "10 cells with largest mag(err):" << refCells << endl;
}
else
{
    int nRef = 0;
    forAll(cells,cellI)      //Loop through cells
    {
        if (mag(err[cellI]) > errTol)
        {
            //Add to list
            nRef++;
            refCells.resize(nRef,cellI);
        }
    }
}
```

Code 29: Loop and find cellnumbers with largest errors

Note the Info statements in *Code* 29, this is a way of visualizing what is happening while the code is run. They are left as comments for now but if uncommented they will, when running the application, print out extra information that makes the code in *Code* 29 easier to understand.

It is now time to refine the cells that have just been selected. The refinement dictionary will first be defined in the same manner as the refineMesh-utility and then the function `multiDirRefinement` will be used to do the cell-splitting. Note that we are still in the `if(runTime.write())`-statement. Add *Code* 30 directly after *Code* 29.

```
//******** Cell Refinement **********
// Since this is a pretty long function i would in this section
// like to call upon a function file that refines the mesh.
// That file should be similar to refineMesh.C

// Define refinement dictionary
dictionary refineDict; //Declare

dictionary coeffsDict;
coeffsDict.add("tan1", vector(1, 0, 0));
coeffsDict.add("tan2", vector(0, 1, 0));

wordList directions(2);
directions[0] = "tan1";
directions[1] = "tan2";

refineDict.add("directions", directions); // Add directions to the dictionary
refineDict.add("useHexTopology", "false");// Use standard cutter
refineDict.add("coordinateSystem", "global");

refineDict.add("globalCoeffs", coeffsDict);

refineDict.add("geometricCut", "false");
refineDict.add("writeMesh", "false");

// Multi-directional refinement (does multiple iterations)
Info << "Entering Refinement" << endl;
multiDirRefinement multiRef(newMesh, refCells, refineDict);
Info << "Done with Refinement" << endl;
Info << "Writing new mesh with " << refCells.size() << " refinements" << endl;

mesh.write();
newMesh.write();
```

Code 30: Mesh refinement

The last step is to map the result from `mesh` to `newMesh`. This part has not been completed yet, the code that follows from here on should be added to `icoFoamErrorRefine` but it should be left commented. The reader is urged to try compiling and running the solver with some of the comments uncommented and study the errormessages to complete this application. Add *Code* 31 after *Code* 30. The final `}` finishes the if-statement mentioned earlier.

```
        // Map fields
        //mapConsistentMesh(mesh,newMesh);
        }

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

Code 31: Mapping

Save and quit `icoFoamErrorRefine`.

We are about to compile the program, but before doing that we need to add a few references in the `Make/options`-file. Make sure it has the same appearance as *Code* 32

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/errorEstimation/lnInclude \
    -I$(LIB_SRC)/dynamicMesh/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/lagrangian/basic/lnInclude \
    -I$(LIB_SRC)/sampling/lnInclude
EXE_LIBS = \
    -lmeshTools \
    -lfiniteVolume \
    -ltopoChangerFvMesh \
    -lsampling
```

Code 32: Make/options

Compile by executing `wmake` in the `icoFoamErrorRefine`-directory.

4.3 Solve cavity case with icoFoamErrorRefine

Now it is time to try the new code out to see what it really does. We will solve the `cavity` case with the new solver. Note that the mapping has not been implemented yet so we should not expect the results to be satisfactory. Start with copying the `cavity` case and renaming it to `cavityErrorRefine` according to *Code* 33. We also need to have an object called `err` in the `0`-directory so we simply copy the `U`-file and make some minor changes inside of it. Then run the case and have a look in *paraFoam* . Note that all volume-fields needs to be unchecked from the Object inspector in *paraFoam* since the field variables are not connected to the refined mesh.

```
run
cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavity .
mv cavity cavityErrorRefine
cd cavityErrorRefine
cp 0/U 0/err
sed -i s/"U"/"err"/g 0/err
sed -i s/"(1 0 0)"/"(0 0 0)"/g 0/err

blockMesh |tee logBlockMesh
icoFoamErrorRefine |tee logIcoFoamErrorRefine
```

Code 33: run the cavityErrorRefine case

4.4 Discussion

The goal of the solver has not been reached. The final step of mapping the results calculated on the old mesh to the new mesh was never taken due to lack of time. The resulting solver will now instead solve on the original mesh for all timesteps and refine a sister-mesh called `newMesh` once every `writeInterval`. This will lead to the same cells being refined over and over again instead of the result improving in these areas.

However, this work leaves space for further development in form of making the mapping work. Also an adaptive coarsening could be implemented with the aim of having an error field that is as smooth as possible.

References

[1] Jasak, H. and Gosman, A. D. *'AUTOMATIC RESOLUTION CONTROL FOR THE FINITEVOLUME METHOD, PART 2: ADAPTIVE MESH REFINEMENT AND COARSENING', Numerical Heat Transfer, Part B: Fundamentals, 38: 3, 257 271, URL: http://dx.doi.org/10.1080/10407790050192762*. 2000.