A doctoral skill course at University College London
Established with a UCl Research-led initiative Award

Course Notes:

UCL OpenFOAM Course 2019

funded by the UCL Doctoral Skills Development Programme

Authors:
Luofeng Huang
Daniela Benites
Shiyu Lyu
Tom Smith
Minghao Li
Yeru Shang
Christian Klettner

Department of Mechanical Engineering
University College London (UCL)

October 2019

Preface

OpenFOAM is an opensource simulation tool originally developed by Jasak et al. [1] and Weller at al. [2]. It has been successfully applied to various research areas, including Engineering, Physics, Chemistry and Biology. By employing OpenFOAM, we can simulate the desired practical processes using a computer, followed by the post-processing utility to analyse the results. Pictures and videos can be generated to assist illustration. As the development of IT technique, OpenFOAM has become a powerful skill for a researcher/engineer to grasp.

Compared with other commercial simulation tools, people usually think that OpenFOAM is more difficult to learn. The main difficulty includes:

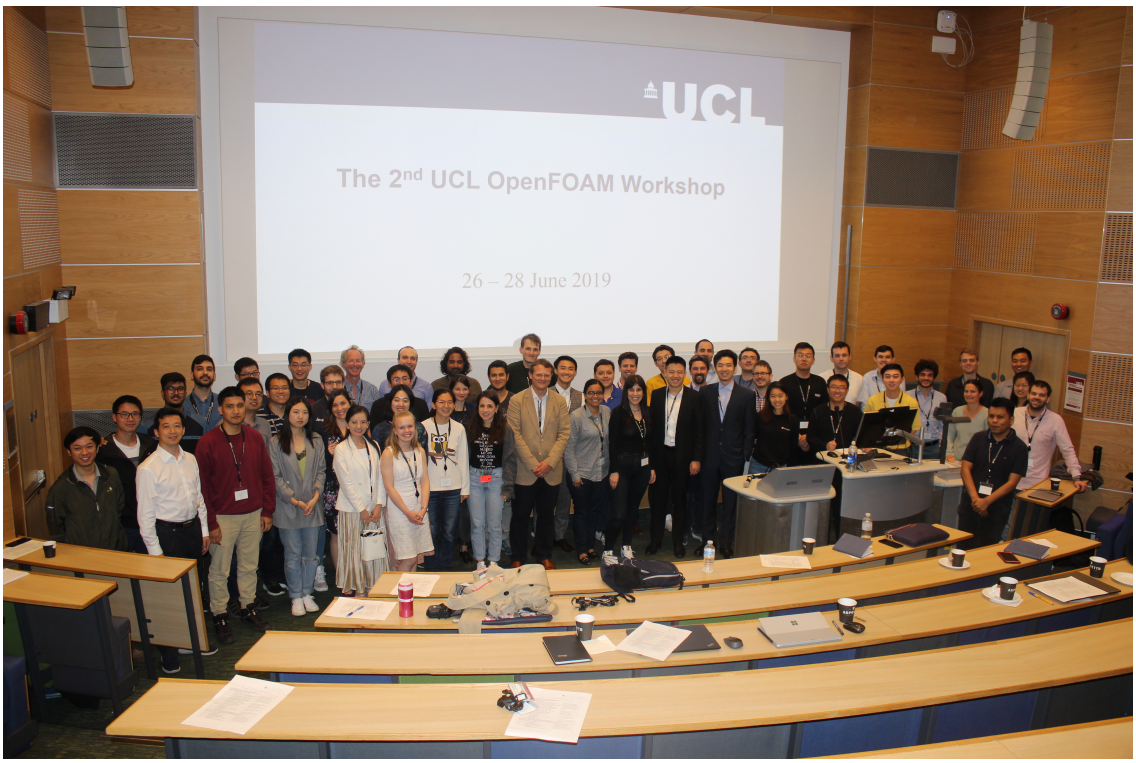
- A beginner is not familiar with its operation environment, Linux. It can be very tricky even trying installing OpenFOAM.
- To understand OpenFOAM codes requires a foundation in Computational Fluid Dynamics (CFD) and C++.
- Online study materials are not comprehensive, and on-site courses are in scarcity.

Nevertheless, benefited from its open-source nature, every single line of OpenFOAM code is accessible, which means you are able to see and manipulate what is actually running behind the simulation. By using OpenFOAM, you will gain a deep understanding on CFD theories and approached, as well as how these methods are implemented. You are free to conduct any modifications upon the codes, which allows you to perform innovations.

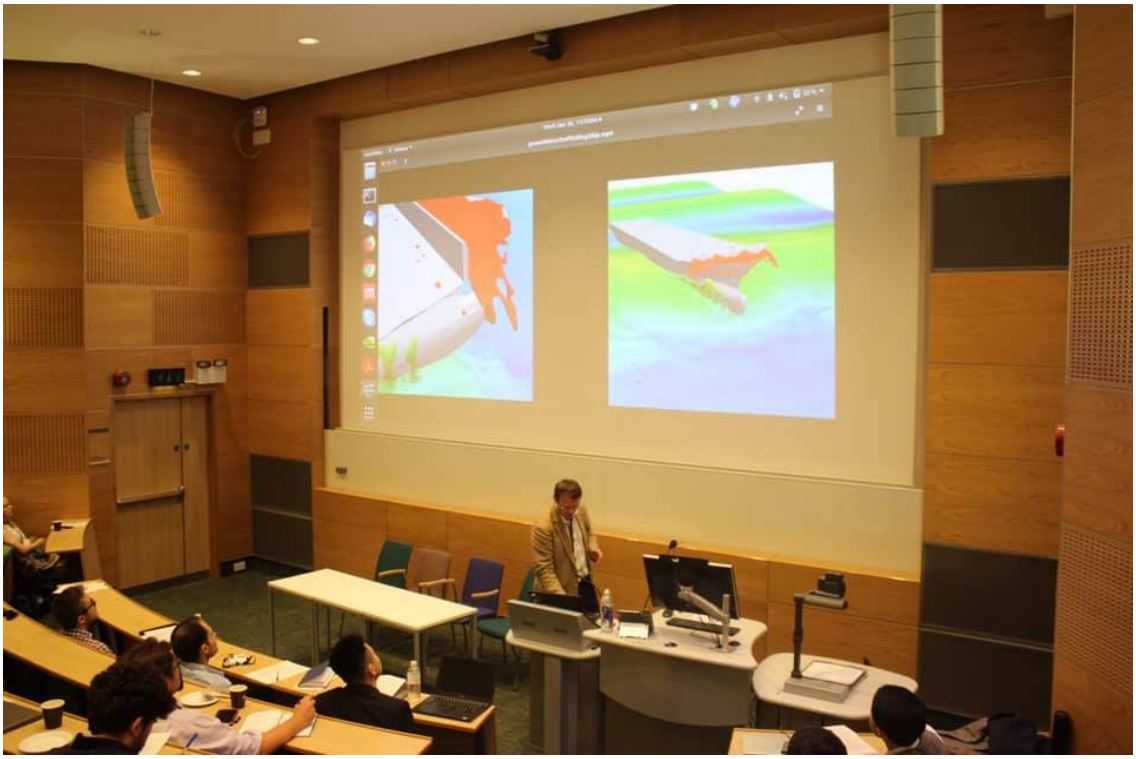
The UCL OpenFOAM Course was initiated by the Department of Mechanical Engineering, totally free and registered under UCL doctoral school. It aims to popularise OpenFOAM among research students and help beginners to get through the initial painful stage dealing with the unfamiliar operation environment, also an excellent chance to exchange simulation skills and generate collaborations. In 2019, the course was held during 26-28 June, with our lecturers and 55 students attended. It was fantastic to see so many conversations getting started, and to feel that our UK/London community is getting stronger. We received very positive feedback, and more importantly, strong interests from worldwide users who wanted but could not join us in London.

Thereby, this document is published online to demonstrate what we have taught. We hope this will be helpful for a wider audience. In Chapter 1-4, we present step-by-step guideline for installing/using/understanding OpenFOAM; subsequently, our Appendixes provides advanced tutorials for various purposes.

Special thanks go to Professor Hrvoje Jasak, the founder of OpenFOAM, for providing invaluable supports on the course's delivery. Now He has also commenced to be a visiting professor at UCL to facilitate OpenFOAM-based teaching/research. Moreover, the course was inspired by the OSCFD course being taught by Håkan Nilsson at Chalmers University of Technology, Sweden (`http://www.tfd.chalmers.se/~hani/kurser/OS_CFD/`).



Group photo



Professor Hrvoje Jasak is lecturing

Contents

Chapter 1

OpenFOAM basis

1.1 Installation

1.1.1 Linux

OpenFOAM can be installed on Linux, macOS and Windows, but Linux has the best compatibility and covers all the functions of OpenFOAM. Therefore, it is recommended to install Linux to operate OpenFOAM.

The easiest way to install Linux system is to install a virtual machine on top of your original operation system (e.g. Windows). An instruction to install a virtual linux system is given as follows (based on the notes provided by Professor Håakan Nilsson).

1. Download the official version VirtualBox (both Platform Packages and Extension Pack), from `https://www.virtualbox.org/wiki/Downloads`. Install it.

2. After installing VirtualBox, start it and:
   Create new virtual disk:
   Click on New
   Name: Ubuntu 18.04 LTS
   Operating System: Linux
   Version: Ubuntu 64-bit (in case you can only select 32-bit here, see `http://www.fixedbyvonnie.com/2014/11/virtualbox-showing-32-bit-guest-versions-64-bit-host-os/#.WvHgFYgvxPY`)
   Base memory: 2048MB (minimum 2GB required for 64-bit operating system)

3. Create a virtual hard drive now
   VDI
   Dynamically allocated
   Set a limit on hard drive storage to 50GB
   Click on Create
   (Settings can be changed later)

4. Download Ubuntu 18.04 from `http://releases.ubuntu.com/18.04/`,
   click `64-bit PC (AMD64) desktop image`, then you will get an iso file (1.9GB).

5. Double-click on the virtual disk you just created in VirtualBox. In the pop-up window, click on the icon "Choose a virtual optical disk file", and browse to the Ubuntu iso-file you just downloaded.
   Ubuntu should be initialised in a window named "Ubuntu 18.04 LTS [Running] Oracle VM VirtualBox".
   Click on Install Ubuntu and select your specific settings

It is recommended that you tick: Download updates while installing, and Install third-party software
Restart when asked (just click on the button that pops up). You may need to press Enter at some point.
(When it asks you if it should erase the disk, don't worry - it refers to the virtual disk you have just created in VirtualBox, not your Windows disk.)

6. After the installation is done. You will enter an Linux Desktop (Ubuntu).

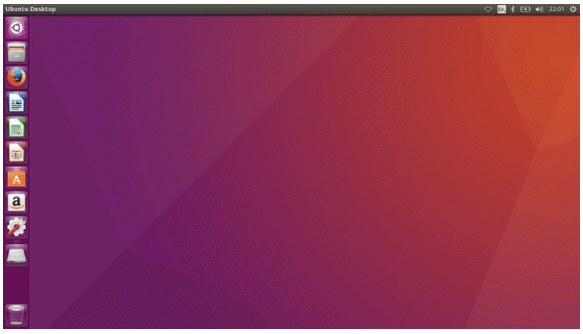


Figure 1.1: Linux desktop of Ubuntu

1.1.2 OpenFOAM

In a Linux system, you are ready to install OpenFOAM. OpenFOAM has different versions, and it keeps updating. Here we give the example of how to install OpenFOAM-v1806. (most of functions are the same in different versions)
To install and utilise OpenFOAM, it is essential to use a Terminal. By press "Ctrl+Alt+T", a terminal window will occur, where you can type your commands.

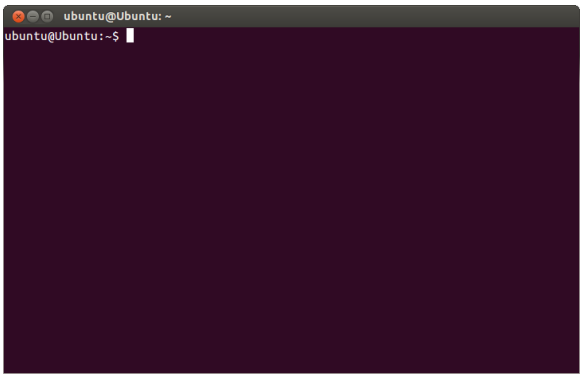


Figure 1.2: A terminal in Ubuntu

1. In your terminal, copy-paste-enter the following commands (one by one, you may need to enter your password at some points to give permission). Note the short cuts for copy and paste here are "ctrl+shift+c" and "ctrl+shift+v" respectively.

   `mkdir $HOME/OpenFOAM`
   `cd $HOME/OpenFOAM`
   `sudo apt install git` (Type password and ENTER, then later you need to 'Y)
   `git clone https://develop.openfoam.com/Development/OpenFOAM-plus.git` (This step requires a user name and password from your registration. Thus, go to: `http://openfoam.`

com/code/repositories.php and register an account)
mv OpenFOAM-plus OpenFOAM-v1806
At the following lines you should include the brackets "(" and ")". The commands executed between the brackets are in a subshell, and after the line you remain in the same directory as before the command. (cd $HOME/OpenFOAM/OpenFOAM-v1806 && git fetch)
(cd $HOME/OpenFOAM/OpenFOAM-v1806 && git checkout maintenance-v1806)
(cd OpenFOAM-v1806 && git submodule init)
wget https://sourceforge.net/projects/openfoamplus/files/v1806/ThirdParty-v1806.tgz
md5sum ThirdParty-v1806.tgz #Should give 3c06cb20d08ab564b70f9df5186ec936
tar xvf ThirdParty-v1806.tgz
rm ThirdParty-v1806.tgz

2. At this point you have two directories: (a) $HOME/OpenFOAM/OpenFOAM-v1806 and (b)$HOME/OpenFOAM/ThirdParty-v1806. The first one contains all the source files to compile OpenFOAM, and the second one contains instructions on how to compile third-party packages, if necessary.
   We start by installing some required packages:
   sudo -i (Type root password for your Ubuntu installation. Makes you root user, so that you can run apt-get directly and without entering password. The following five lines all start with "apt-get". Copy each line separately. You may have to type Y at some point.)
   apt-get install build-essential flex bison cmake zlib1g-dev libboost-system-dev \
   libboost-thread-dev
   apt-get install libopenmpi-dev openmpi-bin gnuplot libreadline-dev libncurses-dev libxt-dev
   apt-get install qt5-default libqt5x11extras5-dev libqt5help5 qtdeclarative5-dev \
   qttools5-dev libqtwebkit-dev
   apt-get install freeglut3-dev libqt5opengl5-dev texinfo
   apt-get install libscotch-dev libcgal-dev
   exit (Stop being root user)

3. We set up an alias for activating the OpenFOAM-v1806 environment:
   echo "alias OFv1806='. $HOME/OpenFOAM/OpenFOAM-v1806/etc/bashrc'" >> $HOME/.bashrc
   (everytime you open a new terminal and intend to use OpenFOAM, you have to execute command "OFv1806" to initialise it.)

4. Now we are ready to compile. To make sure that we get the correct environment we close down the terminal window(s) and open a new one by "Ctrl+Alt+T". OFv1806
   foam
   export WM_NCOMPPROCS=4(To use all four cores on my virtual machine while compiling)
   ./Allwmake(This will take several hours. So just do something else and wait it to finish)

5. After the compilation of OpenFOAM-v1806, we will install ParaView, which is the post-processing utility to view/process our computational results.
   sudo apt-get install paraview

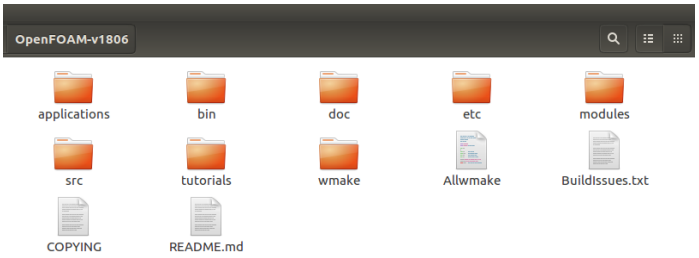
6. Test the installation, open a new terminal:
   OFv1806
   mkdir -p $FOAM_RUN
   run
   cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavity/cavity .
   cd cavity

`blockMesh`
`icoFoam`
`paraFoam` (Click on Apply, a square should appear in the window.)

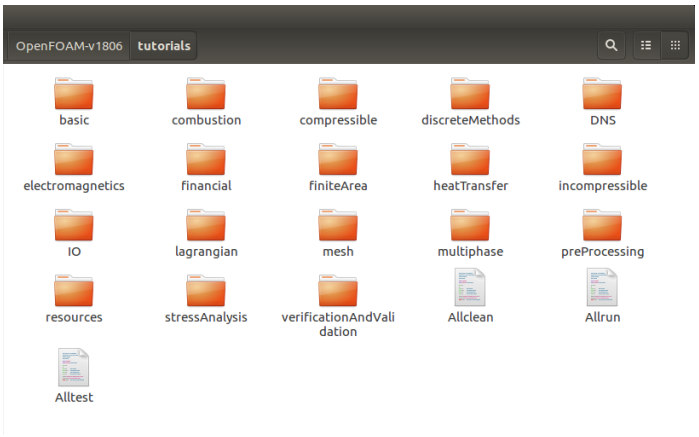
Congratulations. Your OpenFOAM-v1806 has been set up OK!

1.2 OpenFOAM case setup

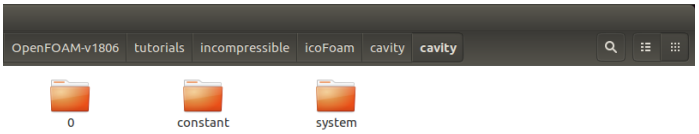
Once installed, OpenFOAM files are saved at the directory: OpenFOAM/OpenFOAM-v1806



There are many ready-to-use cases stored under the folder "tutorials", which are fantastic to start with. Your own cases can always be modified based on these tutorials, so you should never write a case from scratch.
Note: when modifying a tutorial, you should first copy it to your own directory, which was created in the last step of the installation; thus, the original case setting can be preserved.



Let's enter a tutorial case to see how a basic OpenFOAM case looks like,
go to: tutorials/incompressible/icoFoam/cavity/cavity



A basic OpenFOAM case consists of three separate directories:

1. The "0" time folder: the boundary and initial conditions for each of the variables in question, e.g. pressure, velocity.
2. The "constant" folder: the properties of the fluid in question, e.g. viscosity, density.
3. The "system" folder: how will we solve the case, including space discretization, time discretization, solver (what governing equations to use) and numerical scheme/solution.

1.2.1 Introductory CFD

Before further learning of OpenFOAM, this is a short section to prepare a little CFD knowledge as background.
The basic idea to solve an OpenFOAM simulation includes two parts, space discretization and time discretization. Space discretization is to devide a domain of interest into a number of non-overlapping cells, known as mesh; time discretization is to split a process into a number of timesteps.
When we want to learn a process, we mesh the domain by a number of cells and get the solution of each single cell, which forms the solution of the whole space domain. This is similar to a film: a single frame consists of many elements; each element stores its own information, so a higher cell/element) number means you get a clearer image.
Timestep is simpler. When the time of a process you want to learn is certain, e.g. 10s. 100-timesteps means you want to solve the results per 0.1s; then 1000-timesteps means 0.01s.
To sum up, increasing cell number or timestep number means solving a simulation with a higher resolution, while the computational time will increase accordingly.

Then, for a single cell and a timestep, we can solve the governing equations to obtain the parametres that are of interest, e.g. the Navier-Stokes equations to solve pressure (P) and velocity (v).

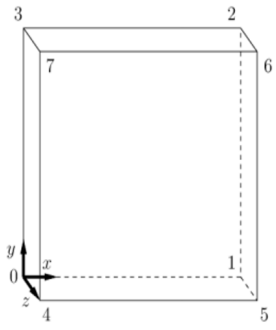
$$\nabla \cdot v = 0 \tag{1.1}$$

$$\frac{\partial(\rho v)}{\partial t} + \nabla \cdot (\rho v v) - \nabla \cdot \tau = -\nabla p + \rho g \tag{1.2}$$

1.2.2 Mesh

The generation of mesh is dictated by a file called `blockMeshDict`, stored under the `system` directory.
Open the `blockMeshDict` of the cavity tutorial, we can see the content as in the picture below. (after "//" are annotations)

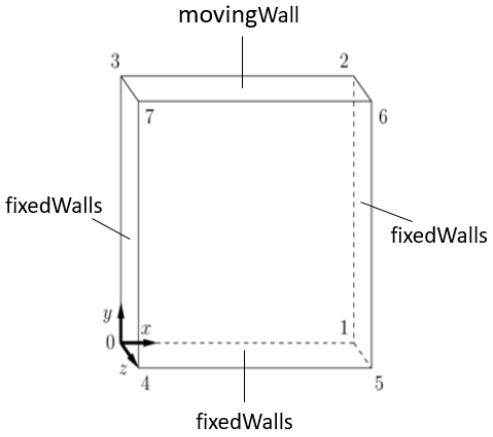
```
vertices
(
    (0 0 0)//0
    (1 0 0)//1
    (1 1 0)//2
    (0 1 0)//3
    (0 0 0.1)//4
    (1 0 0.1)//5
    (1 1 0.1)//6
    (0 1 0.1)//7
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (20 20 1) simpleGrading (1 1 1)
);
```



OpenFOAM always operates in a three-dimensional Cartesian coordinate system. In `blockMeshDict`, first we need to define some vertices:
Once the vertices are defined, they are numbered in order (start with 0, so there are 8 points in total, numbered 0-7).
The vertices 0-7 can form a hexahedron, which is defined in the `blocks` part, and this is our computational domain.
`(20 20 1)` means the hexahedron will be meshed with 20 cells in the X direction, 20 cells in the Y direction and 1 cell in the Z direction (in a two-dimensional problem, we only put one cell in the inactive direction).
Therefore, this domain will be divided into 20*20*1 = 400 cells.
The `simplegrading (1 1 1)` means the sizes of the mesh are uniform on all X Y Z directions. Changing the value here can make the cell size gradually increase along a direction. For example, simplegrading (10 1 1) means the length of the last cell is 10 times of that of the first cell along the X direction.

Every four vertices can form a face, which can be classified in the `Boundary` part:

```
boundary
(
    movingWall
    {
        type wall;
        faces
        (
            (3 7 6 2)
        );
    }
    fixedWalls
    {
        type wall;
        faces
        (
            (0 4 7 3)
            (2 6 5 1)
            (1 5 4 0)
        );
    }
    frontAndBack
    {
        type empty;
        faces
        (
            (0 3 2 1)
            (4 5 6 7)
        );
    }
);
```



1.2.3 Timestep and solver

The `system/controlDict` codes are shown as follows, in which you can modify the solver, runtime, time-step size, how often the results are stored, etc..
The name of a OpenFOAM solver ends with Foam, and the prefixion depends on the solver type. For example:
`icoFoam solves the incompressible laminar Navier-Stokes equations using the PISO algorithm;`
`interFoam is a solver for 2 incompressible fluids;`
`fsiFoam is used for Fluid-Structure Interaction (FSI) problems;`

```
application     icoFoam;//the solver to use

startFrom       startTime;

startTime       0;

stopAt          endTime; //define the total run time

endTime         0.5;

deltaT          0.005;//time-step size

writeControl    timeStep;

writeInterval   20; //record the results every 20 time steps

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable true;
```

1.2.4 Initial and boundary conditions

So far, we introduced the setting of how to solve a case. Now we are going to discuss how to define a problem physically, which is known as initial condition (IC) and boundary conditions (BC). IC is straightforward to understand - it defines the initial configurations of a simulation, i.e. what you want the case to be when t = 0. BC defines how the simulation being pushed forward per timestep. Open the `0` folder, there are two files called P and U, which defines the IC and BC for presure and velocity respectively.
Here we use the U file to give an example. As there are three kinds of boundary defined in `blockMeshDict`, movingWall, fixedWall and frontAndBack. The U file shows the `movingWall` boundary always have a constant velocity of (1 0 0), and the fixedWall does not have a velocity. "empty" BC is usually used on the inactive boundaries of a two-dimensional problem, which can be understand as: no solution is required.

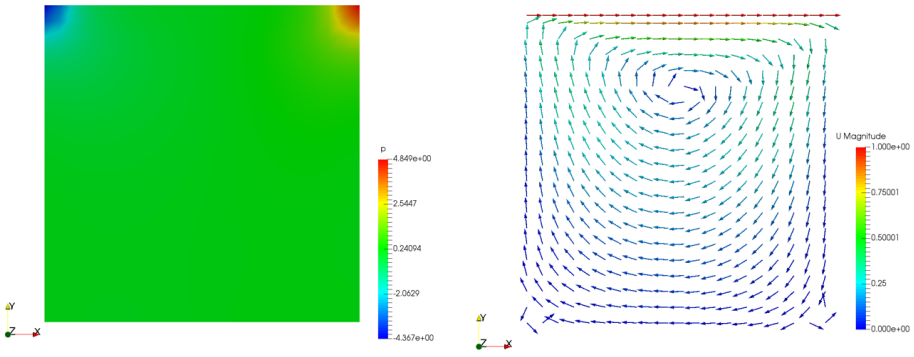
Thus, the case is set up as:
Fluid is initialised as static in a box.
The upper wall of the box start moving towards the right at a velocity of 1 m/s, while the other walls keep fixed. The moving wall will change the status of the fluid inside, as the fluid has viscosity (see `constant/transportProperties`).
If you run the simulation, OpenFOAM will tell you what would happen in such a box.

```
dimensions      [0 1 -1 0 0 0];
//dimensions define the unit of the value.
// (kg m s K mol A cd)
// So here the unit of the velocity is m1*s-1

internalField   uniform (0 0 0); //initial value of internalfield (except the boundaries)

boundaryField
{
    movingWall
    {
        type            fixedValue;
        value           uniform (1 0 0);
    }

    fixedWalls
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }

    frontAndBack
    {
        type            empty;
    }
}
```

1.3 Simulation

To run this simulation you need to execute the following steps from your terminal:

1. as mentioned earlier, always copy the case to your user directory before running or modifying, so as to keep the original file:
   `cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavity/cavity $FOAM_RUN`
   `cd $FOAM_RUN/cavity`
2. build the mesh:
   `blockMesh`
3. launch the solver:
   `icoFoam`
4. post-processing (OpenFOAM is installed with an opensource post-processing tool - paraview)
   `paraFoam`
5. now you have entered paraview and you can view the computational results (the field of P and U for each time step)



More details please see: `https://www.openfoam.com/documentation/tutorial-guide/tutorialse2.php#x6-60002.1`

Chapter 2

Ship flow simulation

To study how to simulate ship flow in OpenFOAM, we use a case of DTCHull as an example in this chapter, which is stored under `tutorials/multiPhase/interFoam/ras/DTCHull`.

2.1 Case introduction

A schematic diagram of the case is shown in Figure 2.1. The flow field is a multi-phase field, with air in the upper part and water in the lower part. The ship model was initialised as floating on the water surface according to the designed draft, and it is fixed. A constant velocity (U) against the ship bow was set to the water, so there is a relative velocity between the ship and water. Thus, the simulation is equivalent to a ship advancing in calm water. To be conformed to an open ocean environment, the computational domain was modelling as infinite, i.e. no boundary wall was set at the sides or the bottom. The resistance of the ship is of interest in this work, which is calculated as the water force against the bow direction.

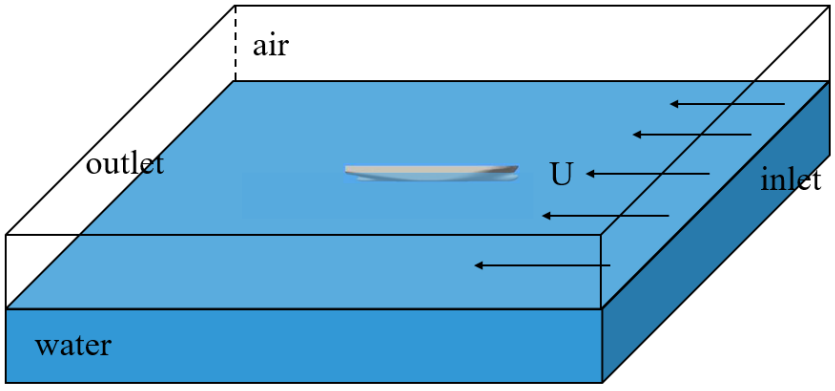


Figure 2.1: Schematic diagram of the simulation case. The hull is floating on the water surface and subjected to flowing water of a constant velocity (U).

2.2 Free-surface modelling

As the fluid domain is a two-phase mixture of air and water, the Volume of Fluid (VOF) method [3] is used to model the free surface. The VOF method introduces a passive scalar α, which denotes the fractional volume of a cell occupied by a specific phase. In this model, a value of $\alpha = 1$ corresponds

to a cell full of water and a value of $\alpha = 0$ indicates a cell full of air. Thus, the free surface, which is a mix of these two phases, is formed by the cells with $0 < \alpha < 1$. The α value was solved by its transport equation as expressed in Equation 2.1, and further the local density (ρ) and viscosity (μ) were determined according to the corresponding α, as Equation 2.2 and 2.3. Fluid properties of both fluids are set in `constant/transportProperties`.

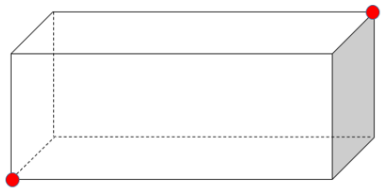
$$\frac{\partial \alpha}{\partial t} + \nabla(\overline{\mathbf{v}}\alpha) = 0 \tag{2.1}$$

$$\rho = \alpha\rho_{water} + (1 - \alpha)\rho_{air} \tag{2.2}$$

$$\mu = \alpha\mu_{water} + (1 - \alpha)\mu_{air} \tag{2.3}$$

The multi-phase model is defined in `system/setFieldDict`, as:

```
defaultFieldValues
(
    volScalarFieldValue alpha.water 0// default alpha value
);

regions
(
    //set cell values
    boxToCell
    {
        box (-999 -999 -999) (999 999 0.244);
       //two opposite vertexes of the cuboid

        fieldValues
        (
            volScalarFieldValue alpha.water 1
        );
    }

    // Set patch values
    boxToFace
    {
        box (-999 -999 -999) (999 999 0.244);

        fieldValues
        (
            volScalarFieldValue alpha.water 1
        );
    }
);
```



It defines a specific area where $\alpha = 1$, and the other area defaults $\alpha = 0$. Here it says the cells of $Z < 0.244$ m is water and $Z > 0.244$ m is air, so the free surface of this case is at Z = 0.244 m.

2.3 Boundary conditions

The BCs of this case is to generate a water flow of steady velocity, which is explained separately as `0/U` and `0/P`.

U:

- Inlet: `fixedValue` is a vbasic BC and easy to understand. Here it gives the inlet boundary a constant speed of (-1.668 0 0) towards the outlet direction.
- Outlet: `outletPhaseMeanVelocity` adjusts the velocity for the given phase to achieve the specified mean thus causing the phase-fraction to adjust according to the mass flow rate. By applying this boundary condition, the mass flow rate of the water phase in the outlet boundary is set to equal to the inlet mass rate, so that the water volume in the computational domain can keep constant. This BC can avoid the waterline continuously increase/decrease, which is typically used as the outlet condition for a towing-tank ship simulation in order to maintain the outlet water level as the same as the inlet.

- Atmosphere: `pressureInletOutletVelocity` - here we specific the tangential velocity across this face to equal (-1.668 0 0).

```
Umean 1.668;
mUmean -1.668;

dimensions      [0 1 -1 0 0 0 0];

internalField   uniform ($mUmean 0 0);

boundaryField
{
    //- Set patchGroups for constraint patches
    #includeEtc "caseDicts/setConstraintTypes"

    inlet
    {
        type            fixedValue;
        value           $internalField;
    }

    outlet
    {
        type            outletPhaseMeanVelocity;
        alpha           alpha.water;
        Umean           $Umean;
        value           $internalField;
    }

    atmosphere
    {
        type            pressureInletOutletVelocity;
        tangentialVelocity $internalField;
        value           uniform (0 0 0);
    }
```

P:

- Inlet: `fixedFluxPressure` is used to set the pressure gradient to be specified by the velocity boundary condition, so that the fixed water velocity can be matched.
- Outlet: `zeroGradient` means the gradient of pressure equals to zero, typically used as outlet pressure BC.
- Atmosphere: `totalPressure` calculates the pressure from a specified total pressure p0 and local velocity U. (Here we define the atmosphere pressure to be 0 pascal)

```
dimensions      [1 -1 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    //- Set patchGroups for constraint patches
    #includeEtc "caseDicts/setConstraintTypes"

    inlet
    {
        type            fixedFluxPressure;
        value           $internalField;
    }

    outlet
    {
        type            zeroGradient;
    }

    atmosphere
    {
        type            totalPressure;
        p0              uniform 0;
        U               U;
        phi             phi;
        rho             rho;
        psi             none;
        gamma           1;
        value           $internalField;
    }
```

2.4 Mesh around the ship

2.4.1 Local mesh refinement

Sometimes we need high-quality meshes in certain areas of a computational domain. For example, in this case we want the mesh to be refined around the ship and in the free surface area. OpenFOAM provides a utility called `refineMesh`, by which we can specify an area and just refine the mesh inside. It is controlled by `system/refineMeshDict`:

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

set             c0;//the area to be refined, defined using topoSet

coordinateSystem global;

globalCoeffs
{
    tan1            (1 0 0);//refine in x direction
    tan2            (0 1 0);//refine in y direction
}

patchLocalCoeffs
{
    patch           outside;
    tan1            (1 0 0);
}

directions      ( tan1 tan2 );

useHexTopology  no;

geometricCut    yes;

writeMesh       no;

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

Here we only refine X and Y directions, since Z direction has already been set to gradually increase towards waterline in `blockMeshDict`. `refineMesh` cuts every grid in the area to be 2 in X direction and 2 in Y direction, so one grid becomes four grids. If applied to all three directions, `refineMesh` will cut one grid into eight. See more on `https://openfoamwiki.net/index.php/RefineMesh`.

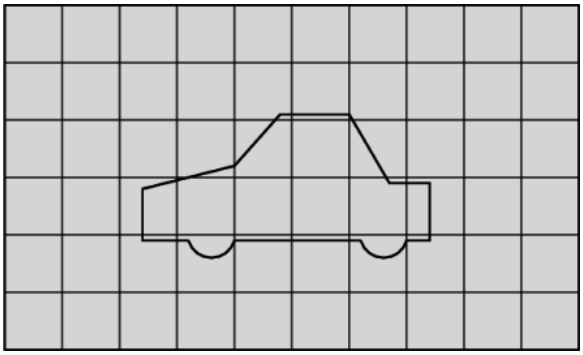
2.4.2 SnappyHexMesh

As a hull geometry is complex, blockMesh is incapable to build the mesh around the hull. Here we introduce another OpenFOAM mesh tool snappyHexMesh to generate high-quality grids fitting around the hull surface. See more in Appendix B.

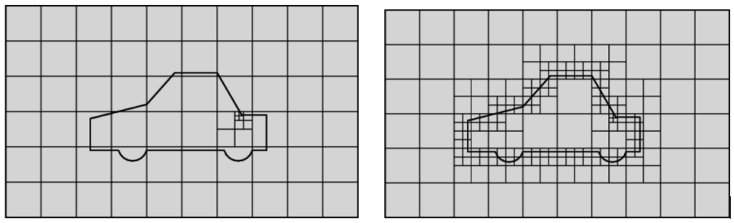
There are three main steps in SnappyHexMesh:
1. castellatedMesh
2. snap
3. addLayers

An example is given below to illustrate the process: (details see `https://cfd.direct/openfoam/user-guide/v6-snappyhexmesh/`)

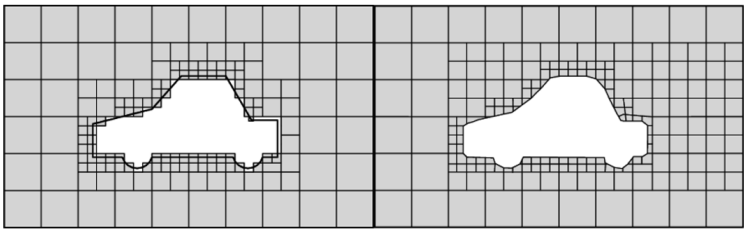
This is a geometry in a domain after blockMesh:



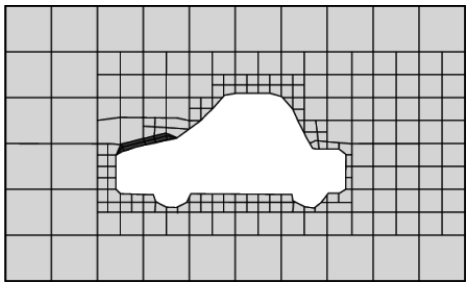
castellatedMesh: the edges of the geometry are detected, and then the meshes that intersect with the edges are refined.



snap: the cells inside the geometry are removed, and then the boundary is smoothed.



addLayers: shrink the existing mesh from the boundary and insert layers of cells



In the DTCHull case, the snappyHexMesh is governed by `system/snappyHexMeshDict`. First of all, we need to store the geometry file of the hull under `constant/triSurface`, and it needs to be in the form of STL.

Load the STL file of the hull and define the hull surface as `wall`:

```
// Which of the steps to run
castellatedMesh true;
snap            true;
addLayers       true;


// Geometry. Definition of all surfaces. All surfaces are of class
// searchableSurface.
// Surfaces are used
// - to specify refinement for any mesh cell intersecting it
// - to specify refinement for any mesh cell inside/outside/near
// - to 'snap' the mesh boundary to the surface
geometry
{
    DTC-scaled.stl
    {
        type triSurfaceMesh;
        name hull;

        patchInfo
        {
            type wall;
        }
    }
};
```

Now we open the system/snappyHexMeshDict file, where we can define the settings about how to be refined the mesh around the ship.
Step 1, castellatedMesh:

```
castellatedMeshControls
{

    // Refinement parameters
    // ~~~~~~~~~~~~~~~~~~~~~

    // If local number of cells is >= maxLocalCells on any processor
    // switches from from refinement followed by balancing
    // (current method) to (weighted) balancing before refinement.
    maxLocalCells 100000;

    // Overall cell limit (approximately). Refinement will stop immediately
    // upon reaching this number so a refinement level might not complete.
    // Note that this is the number of cells before removing the part which
    // is not 'visible' from the keepPoint. The final number of cells might
    // actually be a lot less.
    maxGlobalCells 2000000;

    // The surface refinement loop might spend lots of iterations refining just a
    // few cells. This setting will cause refinement to stop if <= minimumRefine
    // are selected for refinement. Note: it will at least do one iteration
    // (unless the number of cells to refine is 0)
    minRefinementCells 0;

    // Number of buffer layers between different levels.
    // 1 means normal 2:1 refinement restriction, larger means slower
    // refinement.
    nCellsBetweenLevels 3;



    // Explicit feature edge refinement
    // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    // Specifies a level for any cell intersected by its edges.
    // This is a featureEdgeMesh, read from constant/triSurface for now.
    features
    (
         {
             file "DTC-scaled.eMesh";
             level 0;
         }
    );



    // Surface based refinement
    // ~~~~~~~~~~~~~~~~~~~~~~~~

    // Specifies two levels for every surface. The first is the minimum level,
    // every cell intersecting a surface gets refined up to the minimum level.
    // The second level is the maximum level. Cells that 'see' multiple
    // intersections where the intersections make an
    // angle > resolveFeatureAngle get refined up to the maximum level.

    refinementSurfaces
    {
        hull
        {
            // Surface-wise min and max refinement level
            level (0 0);
        }
    }
```

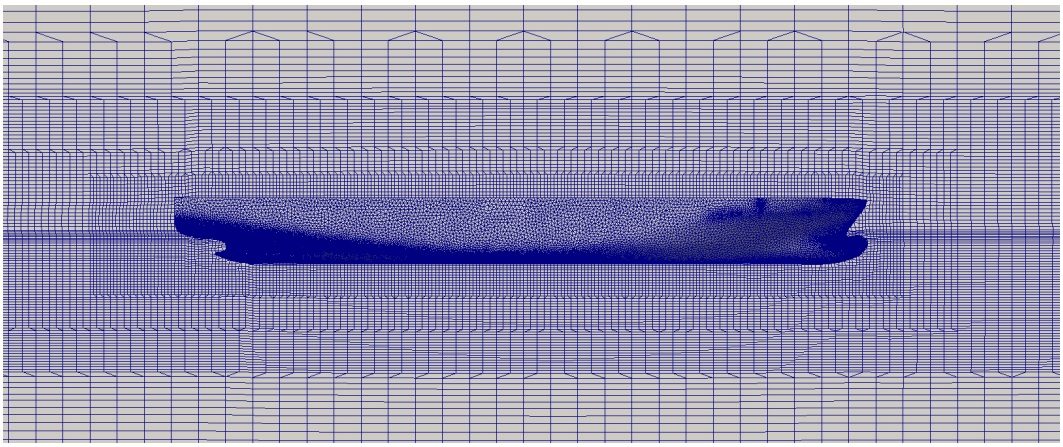
Step 2, snap:

```
// Settings for the snapping.
snapControls
{
    //- Number of patch smoothing iterations before finding correspondence
    //  to surface
    nSmoothPatch 3;

    //- Relative distance for points to be attracted by surface feature point
    //  or edge. True distance is this factor times local
    //  maximum edge length.
    //    tolerance 4.0;
    tolerance 1.0;

    //- Number of mesh displacement relaxation iterations.
    nSolveIter 100;

    //- Maximum number of snapping relaxation iterations. Should stop
    //  before upon reaching a correct mesh.
    nRelaxIter 5;

    nFeatureSnapIter 10;
}
```

Step 3, addLayers:

```
// Settings for the layer addition.
addLayersControls
{
    // Are the thickness parameters below relative to the undistorted
    // size of the refined cell outside layer (true) or absolute sizes (false).
    relativeSizes true;

    // Per final patch (so not geometry!) the layer information
    layers
    {
        hull
        {
            nSurfaceLayers 3;

        }
    }

    // Expansion factor for layer mesh
    expansionRatio 1.5;


    // Wanted thickness of final added cell layer. If multiple layers
    // is the thickness of the layer furthest away from the wall.
    // Relative to undistorted size of cell outside layer.
    // See relativeSizes parameter.
    finalLayerThickness 0.7;

    // Minimum thickness of cell layer. If for any reason layer
    // cannot be above minThickness do not add layer.
    // See relativeSizes parameter.
    minThickness 0.25;
```

The generated mesh around the ship is shown below.



2.5 Ship resistance

Based on the simulation, the fluid force on the ship can be calculated as an integration of surrounding fluid mesh over the hull surface:

$$\mathbf{F_h} = \int (-\overline{P}\mathbf{n} + \overline{\tau} \cdot \mathbf{n}) dS \tag{2.4}$$

where $\overline{p}$ denotes the pressure, $\overline{\tau} = \mu[\nabla\overline{\mathbf{v}} + (\nabla\overline{\mathbf{v}})^T]$ is the viscous term.

OpenFOAM uses a `functionObject-forces` to calculate ship resistance, which is added in the end of the `controlDict` file. This functionObject outputs the pressure force and shear force acting on a specific body after each timestep, and stores the result as a function of time.

```
functions
{
    forces
    {
        type forces;
        functionObjectLibs ( "libforces.so" );
        patches (hull);
        rhoInf  998.8;
        rhoName rho;
        UName   U;
        log     on;
        outputControl   timeStep;
        outputInterval  1;
        CofR (2.929541 0 0.2);
    }
}
```

The result of this functionObject is stored under `postprocessing/forces/0`, where you can find a data file. Open the file in Excel you can see the forces as a function of time, then the ship resistance can be calculated as the sum of pressure force and shear force on the X direction.
As the ship is a symmetry geometry, we can only calculated resistance of half the ship (by applying `symmetryPlane` boundary condition at Midship), which saves around half of the computational time (only half mesh required). With this method used, we just need to manually double the resistance result.

	A	B	C	D	E	F	G
1	Forces						
2	CofR						
3	Time	pressure (x	y	z)	shear (x	y	z)
4	8.93E-04	2.40E+03	2.68E+02	-6.207609e+02	-1.86E+00	-3.38E-03	5.399121e-02)
5	3.11E-03	5.47E+02	4.94E+03	1.494884e+04)	-1.86E+00	5.56E-03	3.587385e-02)
6	8.64E-03	1.63E+01	2.03E+03	5.161612e+03)	-1.85E+00	8.90E-03	2.490339e-02)
7	2.24E-02	-4.37E+01	1.66E+03	3.915681e+03)	-1.85E+00	5.71E-03	2.670788e-02)
8	5.65E-02	-3.98E+01	1.68E+03	3.982493e+03)	-1.90E+00	-9.44E-04	2.622622e-02)
9	1.06E-01	-3.66E+01	1.70E+03	4.039720e+03)	-2.02E+00	-4.48E-03	2.155079e-02)
10	1.56E-01	-3.55E+01	1.70E+03	4.054497e+03)	-2.15E+00	-4.69E-03	1.454954e-02)
11	2.06E-01	-3.45E+01	1.70E+03	4.059923e+03)	-2.29E+00	-2.86E-03	7.560070e-03)
12	2.56E-01	-3.31E+01	1.70E+03	4.062940e+03)	-2.44E+00	9.47E-04	1.421342e-03)
13	3.06E-01	-3.12E+01	1.70E+03	4.063855e+03)	-2.59E+00	6.28E-03	-4.685995e-04)
14	3.56E-01	-2.92E+01	1.69E+03	4.061544e+03)	-2.75E+00	1.14E-02	-2.578994e-04)
15	4.06E-01	-2.72E+01	1.68E+03	4.056567e+03)	-2.92E+00	1.60E-02	2.017698e-03)
16	4.56E-01	-2.54E+01	1.67E+03	4.049447e+03)	-3.08E+00	1.88E-02	6.423719e-03)
17	5.06E-01	-2.38E+01	1.66E+03	4.041002e+03)	-3.25E+00	2.01E-02	1.181973e-02)
18	5.56E-01	-2.24E+01	1.66E+03	4.031769e+03)	-3.42E+00	1.97E-02	1.768184e-02)
19	6.06E-01	-2.10E+01	1.65E+03	4.022407e+03)	-3.58E+00	1.77E-02	2.365543e-02)
20	6.56E-01	-1.97E+01	1.65E+03	4.013425e+03)	-3.74E+00	1.50E-02	2.934392e-02)
21	7.06E-01	-1.85E+01	1.65E+03	4.005153e+03)	-3.89E+00	1.22E-02	3.480134e-02)
22	7.56E-01	-1.71E+01	1.65E+03	3.997773e+03)	-4.04E+00	9.51E-03	4.028272e-02)
23	8.05E-01	-1.57E+01	1.65E+03	3.991510e+03)	-4.19E+00	6.97E-03	4.599752e-02)
24	8.55E-01	-1.42E+01	1.65E+03	3.986573e+03)	-4.32E+00	3.65E-03	5.226067e-02)
25	9.05E-01	-1.28E+01	1.65E+03	3.982927e+03)	-4.46E+00	-1.07E-04	5.815231e-02)
26	9.55E-01	-1.14E+01	1.65E+03	3.980704e+03)	-4.59E+00	-4.21E-03	6.353141e-02)
27	1.01E+00	-1.00E+01	1.66E+03	3.979698e+03)	-4.72E+00	-8.39E-03	6.813555e-02)
28	1.06E+00	-8.63E+00	1.66E+03	3.979827e+03)	-4.83E+00	-1.21E-02	7.174886e-02)
29	1.11E+00	-7.30E+00	1.66E+03	3.980870e+03)	-4.94E+00	-1.54E-02	7.466380e-02)
30	1.16E+00	-6.03E+00	1.67E+03	3.982807e+03)	-5.04E+00	-1.84E-02	7.740367e-02)
31	1.20E+00	-4.81E+00	1.67E+03	3.985480e+03)	-5.13E+00	-2.11E-02	8.019280e-02)
32	1.25E+00	-3.66E+00	1.68E+03	3.988790e+03)	-5.22E+00	-2.33E-02	8.319755e-02)
33	1.30E+00	-2.56E+00	1.68E+03	3.992599e+03)	-5.29E+00	-2.53E-02	8.659922e-02)
34	1.35E+00	-1.52E+00	1.68E+03	3.996945e+03)	-5.37E+00	-2.74E-02	9.023966e-02)
35	1.40E+00	-5.45E-01	1.69E+03	4.001636e+03)	-5.43E+00	-2.95E-02	9.381670e-02)
36	1.45E+00	3.99E-01	1.69E+03	4.006693e+03)	-5.49E+00	-3.15E-02	9.703953e-02)
37	1.50E+00	1.29E+00	1.70E+03	4.011995e+03)	-5.55E+00	-3.37E-02	9.979226e-02)
38	1.55E+00	2.14E+00	1.70E+03	4.017463e+03)	-5.60E+00	-3.59E-02	1.020715e-01)

The ship resistance at six different velocities (velocity can be varied in the `0/U` file) was calculated and plotted in Figure 2.2, as a function of the simulation time. It can be seen that the resistance value oscillates and the oscillation amplitude reduces over time. When the resistance curve eventually approaches a steady harmonic state, the computational resistance was taken as the average value of the crest and trough.
A comparison between the computational results (CFD) and the corresponding experimental data (Exp.) [4] is given in Figure 2.3, where good agreement can be found. For all the six velocity conditions, the computational results are slightly smaller than the experimental values and the deviations are less than 5%. Therefore, it is concluded the applied numerical approach can accurately predict the total resistance of a ship in open water.

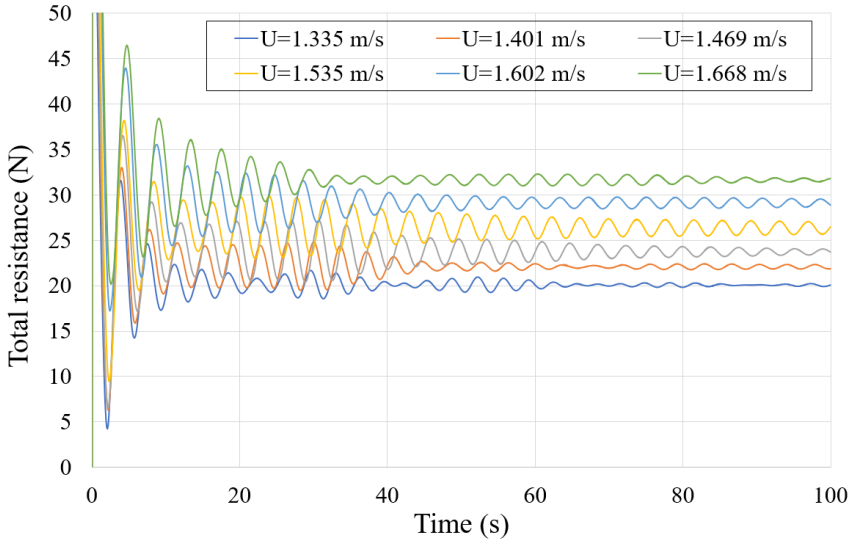


Figure 2.2: Total ship resistance at different velocities (U) over simulation time.

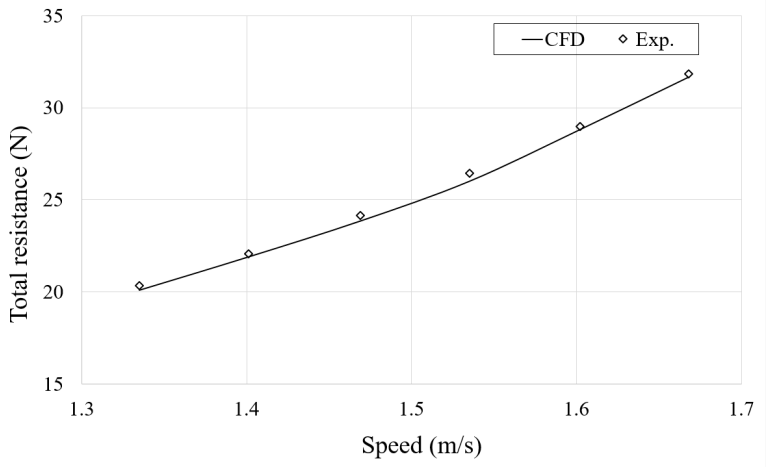


Figure 2.3: Comparison between computational ship resistance and the corresponding experimental data [4].

Chapter 3

Wave simulation

To generate ocean surface waves in a Numerical Wave Tank (NWT), two branches of method are generally used: (a) mimic a physically wave generator (paddle/piston) in the NWT (b) manually modify numerical solutions to achieve theoritical wave profiles. In OpenFOAM, there are two mature NWT tools based on the branch (b), namely: `waves2Foam` [5] using a relaxation zone method, and `ihFoam` [6] using a boundary control method. Both tools are introduced as follows.

3.1 Waves2Foam

3.1.1 Installation

In order to use the `waves2Foam` toolbox, we need to install it for the current version of OpenFOAM (based on its manual [7]):

1. Before proceeding with the installation of `waves2Foam`, we must include the dependencies (or verify they are already within the Ubuntu environment):
   - GNU Scientific Library (GSL) (https://astrointro.wordpress.com/2017/05/17/ installing-gnu-scientific-library-gsl-in-ubuntu-16-04-and-compiling -codes/) (you should get the latest version)
   - Subversion (SVN) (Execute in a terminal: `sudo apt install subversion`)
   - git (Execute in a terminal: `sudo apt install git`)
   - gfortran (Execute in a terminal: `sudo apt install gfortran`)
2. Download `waves2Foam`, execute in the terminal:

   ```
   svn co http://svn.code.sf.net/p/openfoam-extend/svn/trunk/
   Breeder_1.6/other/waves2Foam
   ```

3. Establish directories for `waves2Foam`:

   ```
   cd OpenFOAM/
   cd YOUR USER DIRECTORY (name-v1812)
   mkdir -p applications
   cd applications
   mkdir -p utilities
   ```

4. Find the `waves2Foam` folder in your Downloads (or Home) section, extract and copy it into the Utilities directory you just created (OpenFOAM/YOUR NAME-v1812/applications/utilities).
5. Compile `waves2Foam` with your OpenFOAM, open a terminal (take a while to complete):

```
cd OpenFOAM/YOUR NAME-v1812/applications/utilities/waves2Foam
./Allwmake
```

Now, `waves2Foam` should have been set up OK. We are ready to try one of the tutorials and verify that the toolbox is running smoothly. The manual of `waves2Foam` is available at ResearchGate, search "waves2Foam manual".
To update `waves2Foam`, use:

```
cd OpenFOAM/YOUR NAME-v1812/applications/utilities/waves2Foam
svn update
./Allwmake
```

To verify that the toolbox is correctly installed we are going to run a tutorial case, `waveFlume`. Further information regarding this and other tutorials available are described in the official manual. Check that your terminal is still within the `waves2Foam` folder and execute:

```
cd tutorials/waveFoam/
cp -r waveFlume waveFlume_Example
cd waveFlume_Example
./Allrun
```

This tutorial will run for 10 to 20 minutes (if you want, you can reduce the 20 s of simulation inside the `controlDict file`) and you can check its progress inside the folderâs window. After the execution of the tutorial finishes, check that the post-processor is working as well, proceed with:
`nano foam.foam`
An edit file will be opened within the terminal window, type any letters (example XYZ), and save the file by using CTRL+O, followed by exiting with CTRL+X. A dummy filed was created inside our tutorial. Then, to open ParaView:
`paraview foam.foam &`
Click "Apply", select the option of visualisation of alpha.water and click âPlayâ button. You will see the generation of the waves inside the NWT. The red section is the water, whilst the blue one is the air. By playing the animation of this simulation, we can see that waves have been effectively generated and propagated in the model.
Next, we will understand the theories behind `waves2Foam`: how this toolbox generates the free surface waves, how we build our domain, which is the input data for the waves as to understand what output results we can get depending on the case.

3.1.2 Numerical Wave Tank

The study case used in this tutorial is the implementation of a two-dimensional Numerical Wave Tank (NWT), and its dimensions are 25.0 x 0.1 x 1.6 meters (Length x Breadth x Height). The water level is set at 1.2 meters. For the waves generation the waves are regular and correspond to the Stokes' first order wave theory (or Airy's Wave). The wave height used is 0.1 m, the wave period 2.0 s, whereas the wavelength is 5.5 m.

Theoretical background: relaxation zones

In physics, wave is actually the distribution of the free surface. To model the loads of waves on a solid body, a desired wave field should be generated in the CFD model, which is known as a numerical wave tank.
In OpenFOAM, the `waves2foam` toolkit can be used to model a numerical wave tank. Its installation instruction and theoretical details can be found in the manual [7].
The waves2foam toolkit uses a technique known as relaxation zone [8] to facilitate the modelling of a numerical wave tank. Commonly, two relaxation zones are set at the inlet and outlet of the domain, as the schematic diagram shown in Figure 3.1. These two zones can effectively help generate and absorb surface waves respectively. A relaxation zone can also be set to other shapes, e.g. cylindrical

rather than rectangular.

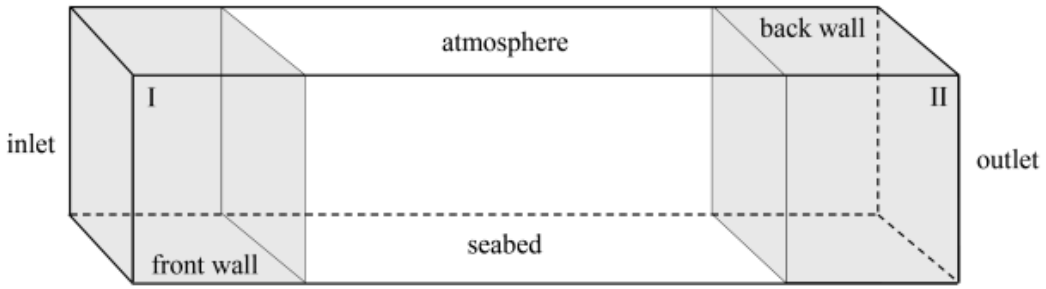


Figure 3.1: Schematic diagram for the inlet and outlet relaxation zones (grey) in a numerical wave tank [9].)

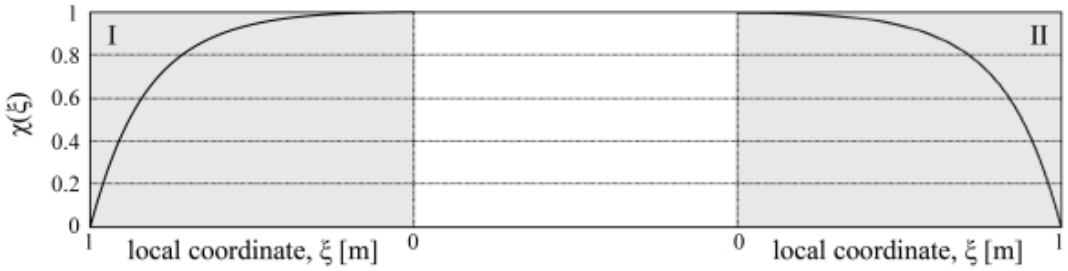


Figure 3.2: The value of spatial weighting factor, $\chi(\xi)$, as a function of local coordinate[9].)

The relaxation zones can be employed to prevent wave reflection from the outlet boundary and also to prevent internally reflected waves, e.g. the waves reflected by internal structure to influence the wave generation at the inlet boundary. In the relaxation zone method, a spatial weighting factor χ is introduced as:

$$\chi(\xi) = 1 - \frac{exp(\xi^{\beta} - 1)}{exp(1) - 1} \tag{3.1}$$

where ξ is the local coordinate in the relaxation zone, which equals to 1 at the outer end and 0 at the inner end of the relaxation zone, as indicated in Figure 3.2. The shape factor β can be defined arbitrarily. Then a local value ϕ is dependent on χ as:

$$\phi = \chi\phi_{computed} + (1 - \chi)\phi_{target} \tag{3.2}$$

where ϕ_{target} is the target solution such as `U` or α, and $\phi_{computed}$ is the numerically computed value, obtained from the Navier-Stokes and VOF equations. Thus, the relaxation zone can obtain an adjusted ϕ over each timestep, thereby minimizing the interference caused by wave reflection. The target wave parameters are set according to wave theories through a file named `waveProperties.input`, which will be introduced as below.
Based on the theory above, Figure 3.3 below shows the schematic of this tutorial case: the minimum lengths of each zone depend on the wave length (L) considered.

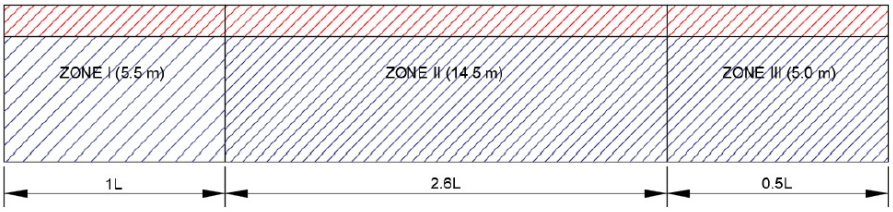


Figure 3.3: Layout of the NWT using waves2Foam: ZONE I - Inlet Relaxation Zone (Wave Generation); ZONE II - Section for the analysis; ZONE III - Outlet Relaxation Zone.

Case setup and running

Now, in order to build our tutorial case, we are starting by copying the tutorial `waveFlume` and rename it as `waveFlume_NWT1`:

```
run
cd ../applications/utilities/waves2foam/tutorials/waveFoam/
cp -r waveFlume waveFlume_NWT1
```

Mesh

For the mesh generation we are going to use the same option as it was done in the previous cavity tutorial in 1.2.2, `blockMeshDict`, which produces a block-structured hexahedral mesh. The only difference is that our domain here involves the two sections of the NWT: one for the water and the other for the atmosphere. Below, the vertices show that our NWT has dimensions of 25.0 x 1.6 x 0.1m (the short distance -one-cell- in the z-direction indicates a 2D case), the depth of the tank is 1.2 m. The next sections of the `blockMeshDict` are the same as explained in section 1.2.2.

```
convertToMeters 1;

vertices
(
    (  0 -1.20 0 )
    ( 25 -1.20 0 )
    (  0  0.40 0 )
    ( 25  0.40 0 )

    (  0 -1.20 0.1 )
    ( 25 -1.20 0.1 )
    (  0  0.40 0.1 )
    ( 25  0.40 0.1 )
);

blocks
(
    hex (0 1 3 2 4 5 7 6) ( 1250 80 1 )
    simpleGrading (1 1 1)
);
```

```
    empty frontBack
    (
        (0 1 3 2)
        (4 5 7 6)
    )
    wall fixedPlate
    (
    )
);

mergePatchPairs
(
);
```

Waves

For setting up the wave parameters, inside the constant folder, we will open the file called `waveProperties.input`. Inside this file we can include the average depth, the wave period, the wave height, the relaxation zones and the wave theory to be applied (for this case we are using the first order wave theory, named here `firstStokes`). In the first part of the `inletCoeffs` section the parameters of the wave will be written:

```
inletCoeffs
{
    // Wave type to be used at boundary "inlet" and in relaxation zone "inlet"
    waveType    stokesFirst;

    // Ramp time of 2 s
    Tsoft       2;

    // Water depth at the boundary and in the relaxation zone
    depth       1.20;

    // Wave period
    period      2.00;

    // Phase shift in the wave
    phi         0.000000;

    // Wave number vector, k.
    direction  (1.0 0.0 0.0);

    // Wave height
    height      0.1;
```

And in the second part would be the ones for the relaxation zones. More information on the parameters used here can be found in Section 2.7 of the waves2foam manual [7].

```
    // Specifications on the relaxation zone shape and relaxation scheme
    relaxationZone
    {
        relaxationScheme Spatial;
        relaxationShape  Rectangular;
        beachType        Empty;

        relaxType   INLET;
        startX      (0 0.0 -1);
        endX        (5.5 0.0  1);
        orientation      (1.0 0.0 0.0);
    }
};

outletCoeffs
{
    waveType    potentialCurrent;
    U           (0 0 0);
    Tsoft       2;

    relaxationZone
    {
        relaxationScheme Spatial;
        relaxationShape  Rectangular;
        beachType        Empty;

        relaxType   OUTLET;
        startX      (20 0.0 -1);
        endX        (25 0.0  1);
        orientation      (1.0 0.0 0.0);
    }
};


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

Boundary Conditions

Below are the boundary conditions applied in each patch of the NWT and which are located in the 0.org folder:

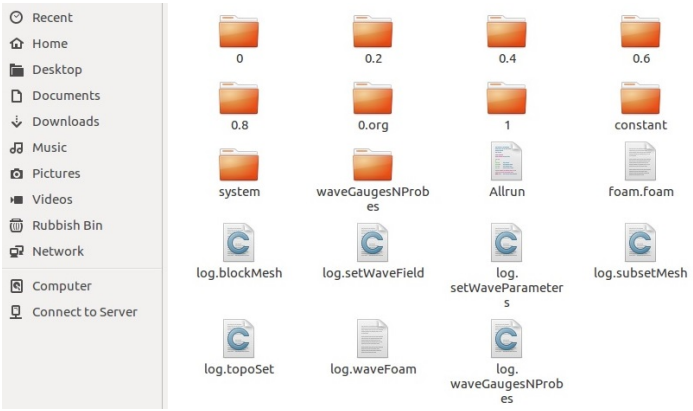
Boundary	alpha1	p-rgh/pd	U
inlet	waveAlpha	zeroGradient	waveVelocity
outlet	zeroGradient	zeroGradient	fixedValue
bottom	zeroGradient	zeroGradient	fixedValue
atmosphere	inletOutlet	totalPressure	pressureInletOutletVelocity
frontBack	empty	empty	empty

Table 3.1: **Boundary Conditions**

Simulation
Now to run the model we execute the file `./Allrun`:

```
run
cd ../applications/utilities/waves2foam/tutorials/waveFoam/waveFlume_NWT1
./Allrun
```

We should have new folders of each time step as well as the log files of each application in our example window as it is shown below.



Post-processing: free surface elevation

First we create the dummy file called foam.foam and then open it in Paraview, as in the previous examples.

Now, we are going to see the results of the wave surface elevation measured at the centre of the NWT. For this:

1. Select `Filters>Alphabetical>Contour` option within Paraview, in the "Contour By" drop window the `alphawater` field is selected to a value of 0.5 (in the `Isosurfaces` section) and click "Apply".

2. Select the `Filters>Alphabetical>Slice` option within Paraview. The origin is set to the middle of the tank at water level: 12.5 x 0.05 x 1.2 and click "Apply".

3. Select the "Split Horizontal" option on the right hand side of the view window (next to **RenderView1**), select the SpreadSheet View and select one of the points showed for that Slice.

4. Select `Filters>Alphabetical>Plot Selection Over Time` and click on the "Apply" button, which will produce a picture as Figure 3.4.

5. you can also save the file with the extension .csv and open it using Excel.

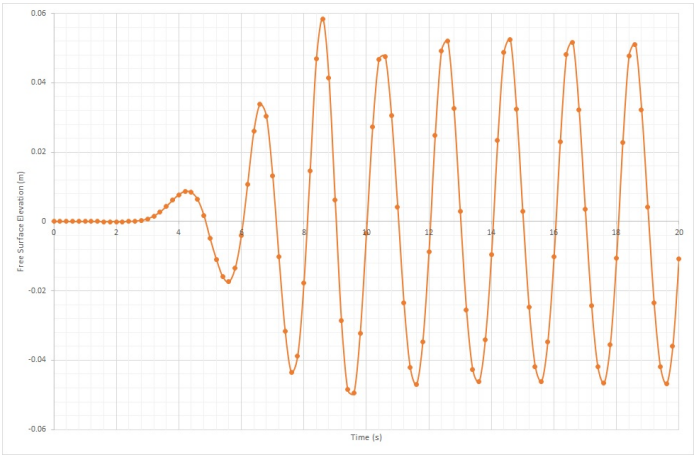


Figure 3.4: Free Surface Elevation measured at 12.5 m from the wave-maker.

3.2 ihFoam

There is no need to install ihFoam as it is already within OpenFOAM-v1806. Same tutorial case used with waves2foam will be perfoamed using ihFoam.

Boundary control method

In the technique used in ihFoam, the values of the velocity fields and the free surface elevation are corrected at each time-step at the inlet patch according to the wave theory applied. This correction is done by comparing the numerical measured value against the theoretical value, and, depending on which one is greater, the initial values of U (velocity vector field) and alpha (phase fraction factor used in the VOF method [3]) in the boundary are corrected. This static boundary wave generator is combined with active wave absorption, and by thus, dissipation zones are not needed, and unnecessary water level increase is avoided. The methodology is detailed in [10; 11]. The pressure is calculated within the numerical model whilst the values of the velocity fields and the free surface elevation are corrected in the wave generation patch according to the wave theory applied. The free surface is measured at each time step and compared to the theoretical value, and, depending which one is greater, corrections are done at the patch and the values of U and alpha are updated to these corrections.

Case setup and running

Now, in order to build our tutorial case, we are starting by copying the tutorial `waveFlume` and rename it as `waveFlume_NWT1`:

```
run
cp -r $FOAM_TUTORIALS/multiphase/interFoam/laminar/waveExampleStokesI waveFlume_NWT2
```

Mesh

For the mesh generation we are going to use the same option as the previous cases, `blockMeshDict`, which produces a block-structured hexahedral mesh. In here, the domain of the NWT is set as atmosphere only and the water will be included using the `setFields` application. Below, the vertices show that our NWT has dimensions of 25.0 x 0.1 x 1.2 m (the short distance -one-cell- in the z-direction indicates a 2D case). The next sections of the `blockMeshDict` are the same as explained in section 1.2.2.

```
scale   1;

vertices
(
    ( 0.0 0.0 0.0)
    ( 25.0 0.0 0.0)
    ( 25.0 0.1 0.0)
    ( 0.0 0.1 0.0)
    ( 0.0 0.0 1.6)
    ( 25.0 0.0 1.6)
    ( 25.0 0.1 1.6)
    ( 0.0 0.1 1.6)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (1250 1 80) simpleGrading (1 1 1)
);
```

Now, to set the water depth of the NWT the water volume is initiated using `setFields`, for this, in the `setFieldsDict` we set the water depth to 1.2 m:

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

defaultFieldValues
(
    volScalarFieldValue alpha.water 0
);

regions
(
    boxToCell
    {
        box (0 0 0) (25.0 1.0 1.2);
        fieldValues
        (
            volScalarFieldValue alpha.water 1
        );
    }
);

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

Waves

For setting up the wave parameters we open the file `waveProperties` in the constant folder. Inside this file we can include the average depth, the wave period, the wave height and the wave theory to be applied (for this case we are using the first order wave theory, named here `StokesI`):

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

inlet
{
    alpha           alpha.water;

    waveModel       StokesI;

    nPaddle         1;

    waveHeight      0.1;

    waveAngle       0.0;

    rampTime        2.0;

    activeAbsorption yes;

    wavePeriod      2.0;
}

outlet
{
    alpha           alpha.water;

    waveModel       shallowWaterAbsorption;

    nPaddle         1;
}


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

Boundary Conditions

Below are the boundary conditions applied in each patch of the NWT and which are located in the 0.org folder:

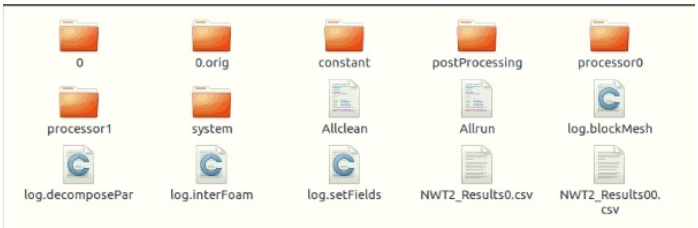
Boundary	alpha.water	p-rgh/pd	U
inlet	waveAlpha	zeroGradient	waveVelocity
outlet	zeroGradient	zeroGradient	waveVelocity
bottom	zeroGradient	zeroGradient	fixedValue
atmosphere	inletOutlet	totalPressure	pressureInletOutletVelocity
sides	empty	empty	empty

Table 3.2: **Boundary Conditions**

Simulation
Now to run the model we execute the file `./Allrun`:

```
run
cd waveFlume_NWT2
./Allrun
```

We should have now the processors employed to run this case (2) as well as the log files of each application in our example window as it is shown below. To divide the domain into smaller sub-domains into which the solutions are being solved decrease the total time of simulation, this is called Parallelisation, which is addressed in another Chapter of these guidelines.



Post-processing
Same as in Page 23

3.3 Wave pass through a fixed plate

The study case used in this tutorial includes a Numerical Wave Tank (NWT) with a fixed vertical plate located near its mid-section. In the Figure below are shown the dimensions of the plate as well as the ones of the NWT. As for the characteristics of the regular wave used for the example, the length is 5.5 m, the height is 0.1 m and the wave period is 2.0 s; the conditions are considered for intermediate water depth.

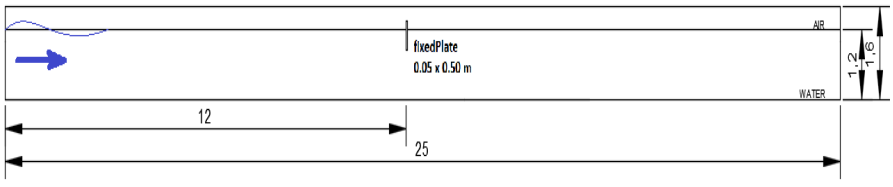


Figure 3.5: Dimensions of the NWT and fixed vertical plate

For the wave generation at the inlet and wave absorption at the outlet a relaxation zone technique in waves2Foam is used. The details of the wave generation have been introduced in the former section. Now, in order to build our example case, we are starting by copying the tutorial waveFlume and rename it as waveFlume_Example:

run
cd ../applications/utilities/waves2foam/tutorials/waveFoam/
cp -r waveFlume waveFlume_Example

Mesh Generation
For the mesh generation of NWT, we are going to use the same option as the one in the former case of waveFlume_NWT1, blockMeshDict, which produces a block-structured hexahedral mesh. The only difference is that here a patch for an object (a fixed plate) inside the NWT needs to be included.

```
convertToMeters 1;

vertices
(
    (  0 -1.20 0 )
    ( 25 -1.20 0 )
    (  0  0.40 0 )
    ( 25  0.40 0 )

    (  0 -1.20 0.1 )
    ( 25 -1.20 0.1 )
    (  0  0.40 0.1 )
    ( 25  0.40 0.1 )
);

blocks
(
    hex (0 1 3 2 4 5 7 6) ( 1250 80 1 )
    simpleGrading (1 1 1)
);
```

```
    empty frontBack
    (
        (0 1 3 2)
        (4 5 7 6)
    )
    wall fixedPlate
    (
    )
);

mergePatchPairs
(
);
```

In the case of the fixed object inside the NWT, we are going to create the patch using the topoSetDict as it is shown below. This file must be in the folder called system. Please download it from the folder Wave Generation in the Dropbox link shared for the course. In the case of this patch, the geometry of the object is a rectangle (with the boxToCell selection, but other options are available, check https://openfoamwiki.net/index.php/TopoSet). Then, the dimensions of the plate would be 0.05 x 0.50 x 0.1 m. What this will do is to create a void geometry whose sides will be acting as walls, in the limits we set of the bounding box.

```
    object      topoSetDict;
}

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

actions
(
    {
        name    c0;
        type    cellSet;
        action  new;
        source  boxToCell;
        sourceInfo
        {
            box (12.00 -0.35 0.0) (12.05 0.15 0.1);
        }
    }

    {
        name    c0;
        type    cellSet;
        action  invert;
    }
);

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

Wave Generation For setting up the wave parameters, inside the constant folder, we will open waveProperties.input file. Inside this file we can include the average depth, the wave period, the wave height, the relaxation zones dimensions and characteristics and the wave theory applied to its generation. For this case, we are using the first order wave theory, named here stokesFirst, which is same as the former waveFlum_NWT1 tutorial.
Boundary Conditions In the case of the BCs of the cell faces we must include the created fixedPlate patch in each one of the files of the 0.org folder, as it is highlighted below:

Boundary name	alpha1	p-rgh/pd	U
inlet	waveAlpha	zeroGradient	waveVelocity
outlet	zeroGradient	zeroGradient	fixedValue
bottom	zeroGradient	zeroGradient	fixedValue
atmosphere	inletOutlet	totalPressure	pressureInletOutletVelocity
frontBack	empty	empty	empty
fixedPlate	zeroGradient	fixedFluxPressure	fixedValue

Simulation First, we need to enter through the window terminal to the folder waveFlume_Example:

run
cd ../applications/utilities/waves2foam/tutorials/waveFoam
/waveFlume_Example
gedit Allrun

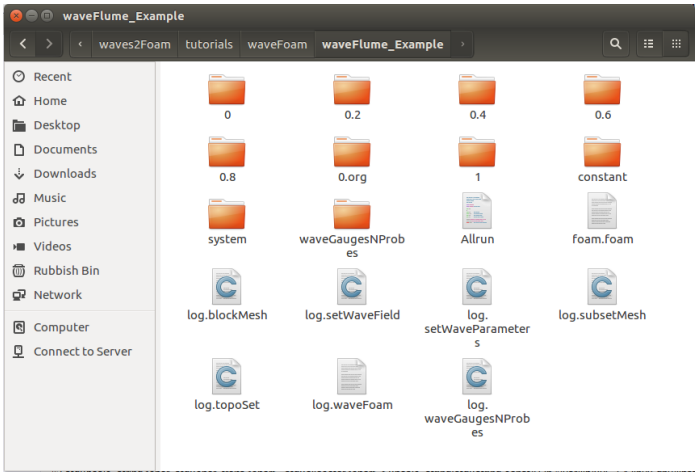
The text file to execute the simulation of the model will be opened. Below the line of meshing we are going to include the lines for running the topoSetDict:

runApplication topoSet
runApplication subsetMesh -overwrite c0 -patch fixedPlate

After this, we save and close the file and go back to the window terminal. We execute the command:

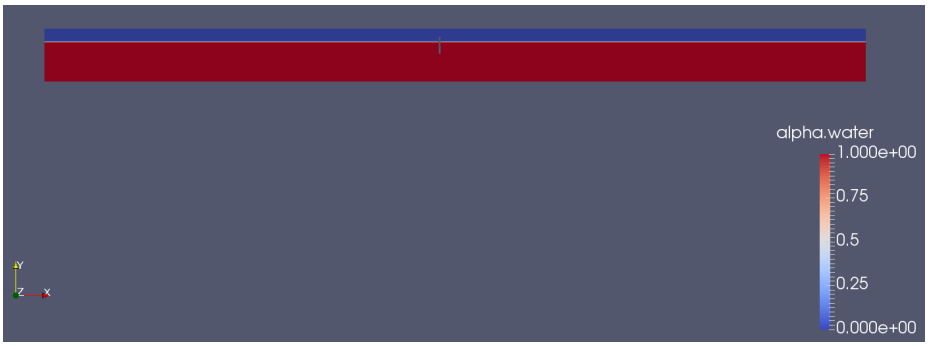
./Allrun

We should have new folders of each time step as well the log files of each application in our example window as it is shown below.

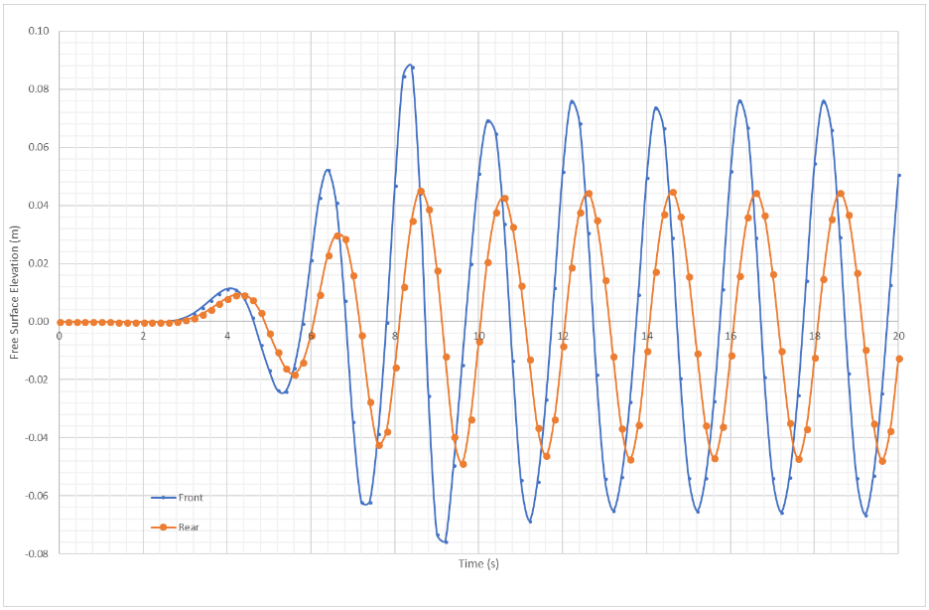


Post-Processing

First create the dummy file foam.foam and then open it in Paraview, as in the previous examples. This is the image you will get:



Now, we are going to see some of the results of the free surface elevation obtained and measured in the front and rear of the fixed plate (See Page 23).



Chapter 4

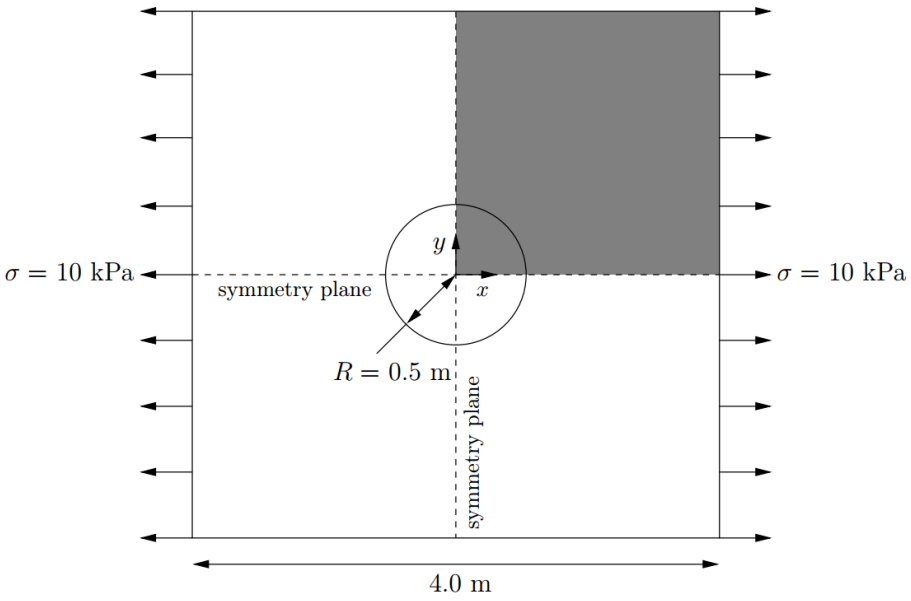
Fluid-Structure Interaction

In previous simulations, fluid-induced solid deformation is negligible, and the solid is commonly assumed as rigid, so we only obtain the solution of the fluid field. However, when considerable solid deformation occurs, FSI approach is required to solve both fluid and solid mechanics.

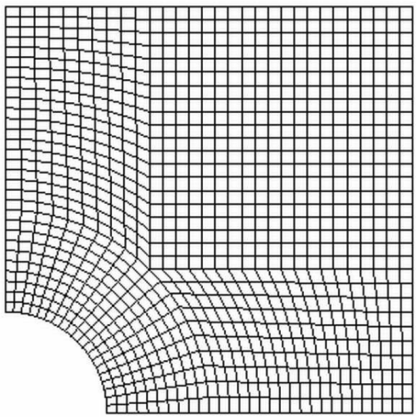
4.1 Preparation: stress-analysis

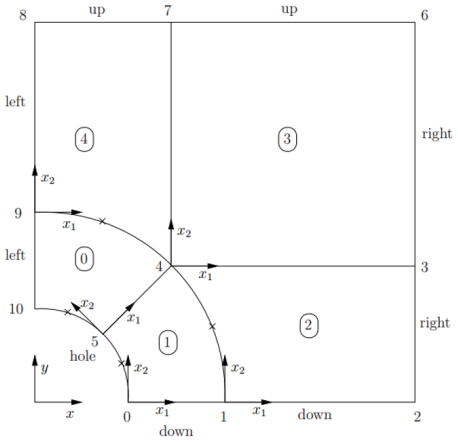
To solve an FSI problem with OpenFOAM, this section will present how to solve solid mechanics in OpenFOAM, as a preparation. The tutorial case we are using is plateHole, stored under `tutorials/stressAnalysis/solidDisplacementFoam/plateHole`.

The case conducts the structural analysis of a square plate with a circular hole at its centre. It is loaded with a uniform traction of $\sigma = 10$ kPa over its left and right faces



According to the symmetry, only one quarter of the plate was taken into calculation, to save computational costs. The mesh and boundary conditions of the case were defined as below.






```
vertices
(
    (0.5 0 0)
    (1 0 0)
    (2 0 0)
    (2 0.707107 0)
    (0.707107 0.707107 0)
    (0.353553 0.353553 0)
    (2 2 0)
    (0.707107 2 0)
    (0 2 0)
    (0 1 0)
    (0 0.5 0)
    (0.5 0 0.5)
    (1 0 0.5)
    (2 0 0.5)
    (2 0.707107 0.5)
    (0.707107 0.707107 0.5)
    (0.353553 0.353553 0.5)
    (2 2 0.5)
    (0.707107 2 0.5)
    (0 2 0.5)
    (0 1 0.5)
    (0 0.5 0.5)
);

blocks
(
    hex (5 4 9 10 16 15 20 21) (10 10 1) simpleGrading (1 1 1)
    hex (0 1 4 5 11 12 15 16) (10 10 1) simpleGrading (1 1 1)
    hex (1 2 3 4 12 13 14 15) (20 10 1) simpleGrading (1 1 1)
    hex (4 3 6 7 15 14 17 18) (20 20 1) simpleGrading (1 1 1)
    hex (9 4 7 8 20 15 18 19) (10 20 1) simpleGrading (1 1 1)
);

edges
(
    arc 0 5 (0.469846 0.17101 0)
    arc 5 10 (0.17101 0.469846 0)
    arc 1 4 (0.939693 0.34202 0)
    arc 4 9 (0.34202 0.939693 0)
    arc 11 16 (0.469846 0.17101 0.5)
    arc 16 21 (0.17101 0.469846 0.5)
    arc 12 15 (0.939693 0.34202 0.5)
    arc 15 20 (0.34202 0.939693 0.5)
);
```

```
dimensions      [0 1 0 0 0 0 0];

internalField   uniform (0 0 0);

boundaryField
{
    left
    {
        type            symmetryPlane;
    }
    right
    {
        type            tractionDisplacement;
        traction        uniform (10000 0 0);
        pressure        uniform 0;
        value           uniform (0 0 0);
    }
    down
    {
        type            symmetryPlane;
    }
    up
    {
        type            tractionDisplacement;
        traction        uniform (0 0 0);
        pressure        uniform 0;
        value           uniform (0 0 0);
    }
    hole
    {
        type            tractionDisplacement;
        traction        uniform (0 0 0);
        pressure        uniform 0;
        value           uniform (0 0 0);
    }
    frontAndBack
    {
        type            empty;
    }
}
```

The mechanical properties of the solid is defined in `mechanicalProperties`, including density, Poisson's ratio and Young's modulus.

```
rho//density
{
    type        uniform;
    value       7854;
}

nu//Poisson's ratio
{
    type        uniform;
    value       0.3;
}

E//Young's modulus
{
    type        uniform;
    value       2e+11;
}

planeStress     yes;//yes for 2D, no for 3D
```

The results are presented as the displacement of each cell (D). By varying the Young's modulus (E) of the plate, it can be observed (with the filter `wrap by vector`) that considerable deformation occurs with a small Young's modulus applied.
More details please see: `https://www.openfoam.com/documentation/tutorial-guide/tutorialse9.php#x16-830005.1`.

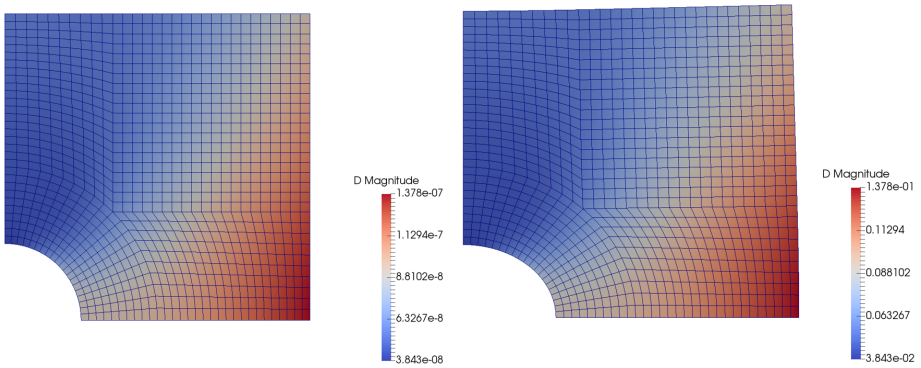


Figure 4.1: Structural response of the plate. Left: E = 2e11 Pa; Right: E = 2e5 Pa.

In the above case, the load on the solid body was applied by boundary conditions, i.e. a specific traction value was set on the solid boundaries. However, in an FSI problem, the load on the solid body is usually unknown. We need to solve the fluid field first, get the fluid load on the solid, and then the solid solver can solve the solid mechanism. In OpenFOAM, such a process can be preformed via fsiFoam, an opensource FSI solver developed by Tukovic et al. [12; 13]. Next, we will introduce how to install and use fsiFoam.

4.2 Installation(FSI)

fsiFoam needs to be installed on foam-extend, which is the extended/advanced version of openFOAM that developed by active contributors. To conduct FSI simulations, this section first provides instructions on installing foam-extend and fsiFoam.

4.2.1 foam-extend

foam-extend is a different version to the original openFOAM. First, we need to revise the bashrc file (the commands that will automatically execute when you open a terminal):
1. open the bashrc file by: go to the home direcotory, ctrl+H to show hiden files, open `.bashrc`
2. in the file, REMOVE the line for initialising openFOAM
`source /opt/openfoam4/etc/bashrc`
AND ADD
`alias of40='source ../../opt/openfoam4/etc/bashrc'`
3. ADD one line under the last line for initialising foam-extend:
`alias fe40='source $HOME/foam/foam-extend-4.0/etc/bashrc'`

Thus, when you open a new terminal, type `of40` to use openFoam 4.0, type `fe40` to use foam-extend 4.0. Otherwise, these two versions will interrupt each other.

The steps to install foam-extend 4.0 are outlined as below. (based on the notes given by Prof Hakan Nillson)

1. Install dependences, Open new terminal window and execute the following commands. (line-by-line, since you have to type your password on the "sudo" lines, and also agree to do the

installations by typing Y)

```
sudo apt-get install git-core build-essential binutils-dev cmake flex
sudo apt-get install zlib1g-dev qt4-dev-tools libqt4-dev libncurses5-dev libiberty-dev
sudo apt-get install libxt-dev rpm mercurial graphviz python python-dev
sudo apt-get install openmpi-bin libopenmpi-dev
sudo apt-get install paraview
```

2. Download it

```
mkdir $HOME/foam
cd $HOME/foam
git clone git://git.code.sf.net/p/foam-extend/foam-extend-4.0 foam-extend-4.0
```

3. Some changes to the installation procedure to save time and disk:

```
echo "export WM_THIRD_PARTY_USE_BISON_27=1" > etc/prefs.sh
echo "export WM_MPLIB=SYSTEMOPENMPI" >> etc/prefs.sh
echo "export OPENMPI_DIR=/usr" >> etc/prefs.sh
echo "export OPENMPI_BIN_DIR=\$OPENMPI_DIR/bin" >> etc/prefs.sh
```

4. Complie

`source etc/bashrc`

`./Allwmake.firstInstall` (this step takes hours)

5. Make a user derectory

`mkdir -p $FOAM_RUN`

4.2.2 fsiFoam

1. Download the package (oepn a terminal and use `fe40` to initialise foam-extend)

```
mkdir -p $WM_PROJECT_USER_DIR
cd $WM_PROJECT_USER_DIR
wget https://openfoamwiki.net/images/d/d6/Fsi_40.tar.gz
tar -xzf Fsi_40.tar.gz
```

2. Compile

```
cd FluidSolidInteraction/src
./Allwmake
```

3. Then, before we try running the tutorial cases, we need to fix a few dependencies (note, between "do" and "done", the two "sed" commands are both end with "item" in the second line):

```
cd ..
find run -name options | while read item
do
sed -i -e 's=$(WM_PROJECT_DIR)/applications/solvers/FSI=$(WM_PROJECT_USER_DIR)/
FluidSolidInteraction/src=' $item
sed -i -e 's=$(WM_THIRD_PARTY_DIR)/packages/eigen3=$(WM_PROJECT_USER_DIR)/
FluidSolidInteraction/src/ThirdParty/eigen3=' $item
done
```

4. Fix some bugs

Open `$WM_PROJECT_DIR/src/finiteVolume/finiteVolume/fvSchemes/fvSchemes.C`,

uncomment the following lines (they should be commended, when you open the file).

```
382         if (dict.found("fluxRequired"))
383         {
384             fluxRequired_ = dict.subDict("fluxRequired");
385
386             if
387             (
388                 fluxRequired_.found("default")
389              && word(fluxRequired_.lookup("default")) != "none"
390             )
391             {
392                 defaultFluxRequired_ = Switch(fluxRequired_.lookup("default"));
393             }
394         }
```

Open `$WM_PROJECT_DIR/src/finiteVolume/fvMatrices/fvMatrix/fvMatrix.C`, comment the following lines.

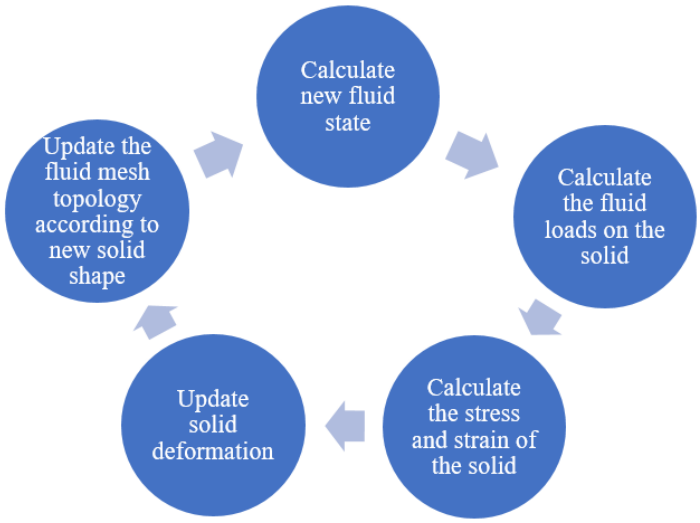
```
1044    //if (!psi_.mesh().schemesDict().fluxRequired(psi_.name()))
1045    // {
1046    //     FatalErrorIn("fvMatrix<Type>::flux()")
1047    //         << "flux requested but " << psi_.name()
1048    //         << " not specified in the fluxRequired sub-dictionary"
1049    //         " of fvSchemes."
1050    //         //<< abort(FatalError);
1051    //}
```

and do:

```
cd $WM_PROJECT_DIR/src/finiteVolume
wmake libso
```

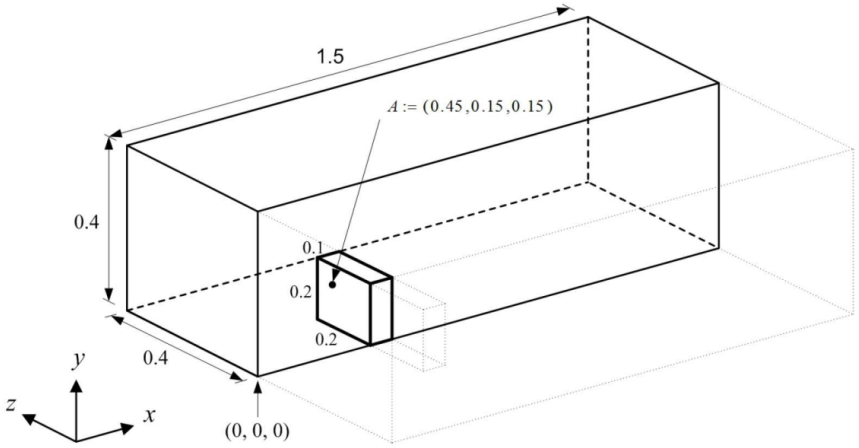
4.3 FSI approach

In fsiFoam, a partitioned algorithm is used to solve the FSI problem, which solves the fluid and solid mechanism separately and links them together via the fluid-solid interface. In other words, fsiFoam employs a fluid solver (e.g. icoFoam) to obtain the fluid field, and employs a solid solver (e.g. solidDisplacementFoam) to solve the solid mechanism. Besides, an FSI scheme is used to link the fluid solution and solid solution. The process is illustrated as below.



4.4 FSI tutorial

Here we use a tutorial to demonstrate how to conduct an FSI simulation. The case is stored under `USER-DIRECTORY/FluidSolidInteraction/run/fsiFoam/beamInCrossFlow`
The case consists of an elastic thick plate attached to the bottom surface of a rectangular channel. The geometry of the spatial domain is given below. An incompressible viscous fluid enters the channel from the left-hand side with a velocity that gradually increases [13].



4.4.1 Case structure

Since the fluid and solid parts are solved separately, they are defined in two folders. As shown in the tree diagram below, the FSI case mainly consists of two parts: `fluid` and `solid`. Each of them has its own `0`, `constant` and `system` directory, as a common OpenFOAM case, and the "createZones" and "setBatch" define the fluid/solid interface. The "makeLinks", "makeSerialLinks" and "removeSerialLinks" files manage the link between the fluid and solid. The "Allrun" and "Allclean" files are located in the fluid part, and the "AllrunPar" file is used for parallel computation. During the process of an FSI simulation, only the fluid part needs to be executed, because the solid part will be called automatically over each timestep.

```
[A FSI CASE]
|-- fluid
|--  |-- 0
|--  |-- Allclean
|--  |-- Allrun
|--  |-- AllrunPar
|--  |-- constant
|--  |-- createZones
|--  |-- setBatch
|--  |-- system
|-- makeLinks
|-- makeSerialLinks
|-- removeSerialLinks
|-- solid
    |-- 0
    |--constant
    |-- createZones
    |-- setBatch
    |-- system
```

4.4.2 Mesh

The mesh of an FSI case also contains two parts, i.e. fluid mesh and solid mesh, defined separately in their `constant/polyMesh/blockMeshDict`. The fluid mesh and solid mesh must match each other through their `interface` and compose a whole computational domain, as shown in Figure 4.2.
An `interface` boundary condition has to be defined in both the fluid and solid mesh. Moreover, the interface in fluid mesh and the interface in solid mesh have to be at a same location. Through the interface, the loads of fluid on solid is outputted to the solid solver and converted into the displacement of the solid surface.

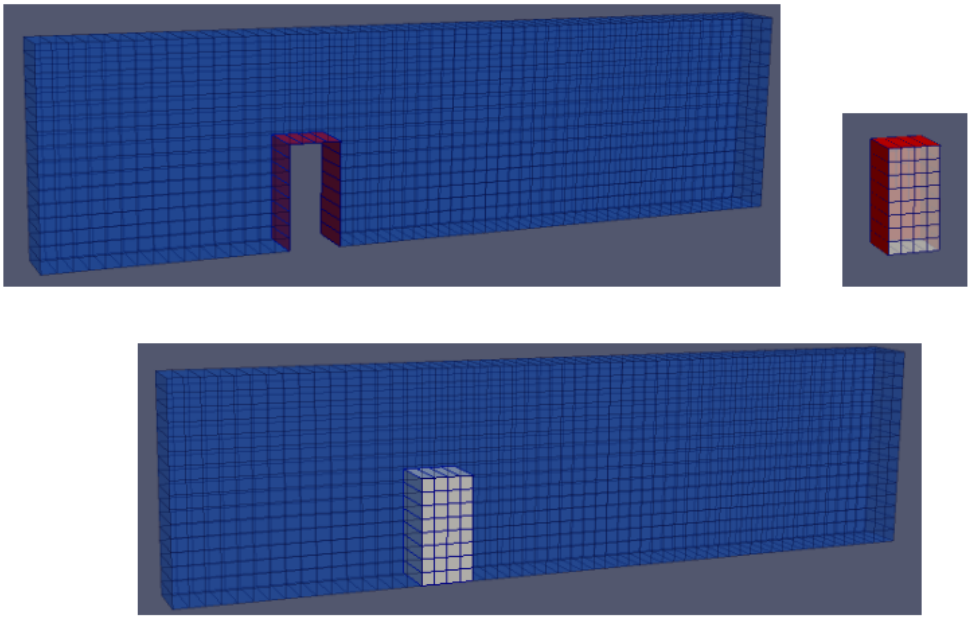


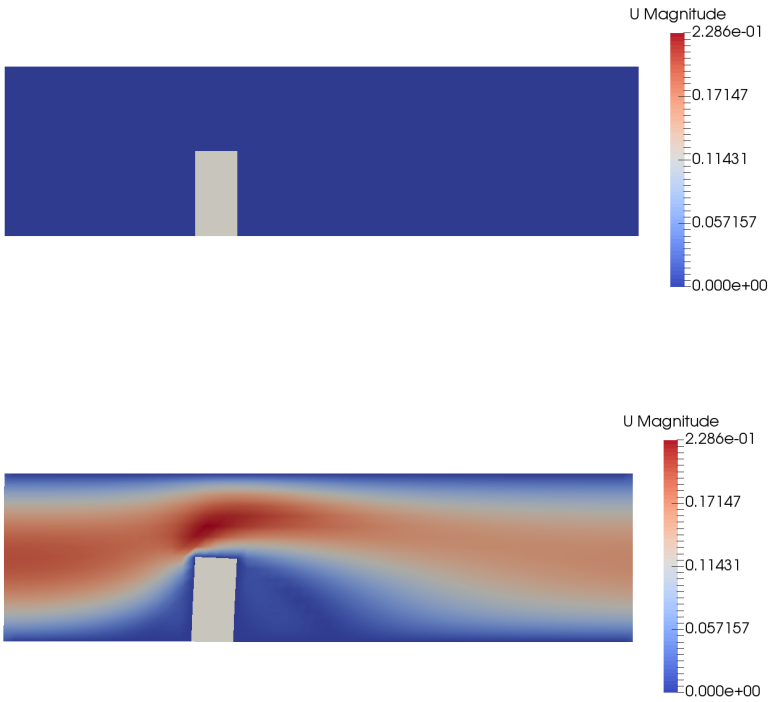
Figure 4.2: Fluid mesh, solid mesh and the integral mesh. (Blue: fluid field; Red: interface; Grey: Solid field)

4.4.3 Simulation

Running the case with `./Allrun`
(If you are using this case for the first time, you need to do:)

```
sed -i s/tcsh/sh/g *Links
./removeSerialLinks fluid solid
./makeSerialLinks fluid solid
cd fluid
./Allclean
./Allrun
```

Use paraview to view the results and it can be seen that obvious deformation of the solid body has been simulated (upper: t=0s; lower: t=10s).

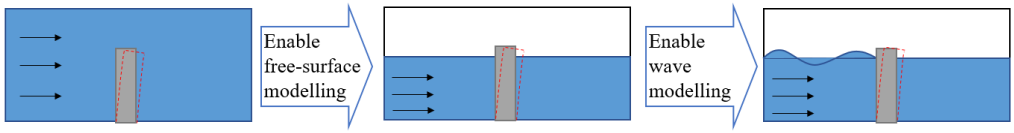


4.5 Wave-induced FSI problems

Last section introduced how to conduct an FSI simulation via fsiFoam. However, the fsiFoam package that we can download from public domain has not included a multiphase library, which means it cannot be used to simulate the FSI problems containing free-surface modelling. In order to investigate the FSI problems in maritime/ocean field, e.g. [14], this section will develop the code of fsiFoam. Specifically, we will compose three solvers: fsiFoam, interDyMFoam and waves2Foam.

4.5.1 Code development

The development route of this work is shown below, where the original FSI package will first be extended into two-phase (fsiFoam+interDyMFoam) and then be endowed the capability of generating a target wave field (fsiFoam+interDyMFoam+waves2Foam).



(a) the deformation of a beam induced by single-phase flow

(b) the deformation of a beam induced by two-phase flow

(c) the deformation of a beam induced by wave loads

To extend the FSI package from single-phase to two-phase, essentially it is to build a new free-surface fluid library beside other existing fluid libraries, e.g. `pisoFluid, icoFluid`. The fluid libraries of FSI package are stored under `src/fluidSolidInteraction/fluidSolvers/`. For this purpose, the code of `interDyMFoam` will be transplanted as a new two-phase fluid library under the `fluidSolvers` directory, which will be named as "interFluid".
The coupling of waves2Foam with FSI package is divided into two parts: on the one hand, to build a new solver that can call both waves2Foam package and FSI package, named `waveFsiFoam`; on the other hand, to include necessary wave objects into `interFluid`,named `waveInterFluid`.

Detailed steps can be found at [15]:
`http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2017/LuofengHuang/2017_OSCFD_Report_Luofeng.pdf`

4.5.2 Tutorial

Here we provide a tutorial of how to use the developed solver, `waveFsiFoam`, to run a wave-induced FSI case: beamInWave.
First of all, we need to modify the case file of "beamInCrossFlow" as follows.

Boundary conditions

Compared with the original "beamInCrossFlow" case , the following revises are required under `fluid/0/` :
1. Copy the `alpha1.org` file from the tutorial case "damBreakWithoutObstacle" (within the installation of foam-extend 4.0), so that the VOF method will divide the domain into two phase.
2. Rename the pressure field from `p` to `pd`; Revise its outlet boundary into a `zeroGradient` condition and its top boundary into a `totalPressure` (typically used to model the atmosphere) condition.
3. To generate waves at the inlet boundary, the inlet boundary conditions of volume fraction `alpha1.org` and velocity `U` are set as `waveAlpha` and `waveVelocity` respectively, which are the boundary conditions installed with the `waves2Foam` package.

```
For alpha1.org
        inlet
        {
         type waveAlpha;
         refValue uniform 0;
         refGrad uniform 0;
         valueFraction uniform 1;
         value uniform 0;
        }
For U
        inlet
        {
         type waveVelocity;
         refValue uniform ( 0 0 0 );
         refGradient uniform ( 0 0 0 );
         valueFraction uniform 1;
         value uniform ( 0 0 0 );
        }
```

Constant

Compared with the original "beamInCrossFlow" case, the following revises are required under `fluid/constant/` :
1. In `fluidProperties`, set the "fluidslover" as `waveInterFluid` (this step calls the new fluid library developed in this study), also change the following "fluidcoeffs" value into `waveInterFluidCoeffs`.
2. Replace the `transportProperties` file by that of the tutorial case "damBreakWithoutObstacle".
3. Copy `waveProperties.input` from the toturial case "waveFlume", as well as the `g` file and `RASproperties` file. Adjust the "sealevel" and other wave parameters according to the geometry.

System and Allrun

Compared with the original "beamInCrossFlow" case, the following revises are required under `fluid/system/` :
1. In `controlDict`, change the "application" value into `waveFsiFoam` (this step calls the developed new solver) and comment the previous object functions.
2. Replace the `fvSchemes` file and `fvSolution` file by those of the tutorial case "damBreakWithoutObstacle", and revise the pressure field of the two files from `p` to `pd`.

To run the case by the command "./Allrun", the `fluid/Allrun` file needs to be revised as below:

- Allrun:line 21-30

```
cd fluid
cp 0/alpha1.org 0/alpha1


runApplication setWaveParameters
runApplication setWaveField

runApplication $application

# ------------------------------------------------------------------ end-of-file
```

The utility `setWaveParameters` is a pre-processing utility, which computes all the necessary wave parameters based on physical meaningful properties, e.g. `setWaveParametes` converts information on water depth and wave period into a wave number for first order Stokes wave theory. In this step, it will load the `constant/waveProperties.input` and output the processed data into a new file, `constant/waveProperties` .
The utility `setWaveField` is used to set the initial conditions according to a user defined wave theory, which is defined by the keyword "initializationName" in the file `waveProperties.input` (see Section 2.3).
The last step calls the solver `waveFsiFoam`, as defined by "getAppapplication" in `fluid/controlDict`. Thus, the new solver `waveFsiFoam` will solve the case.

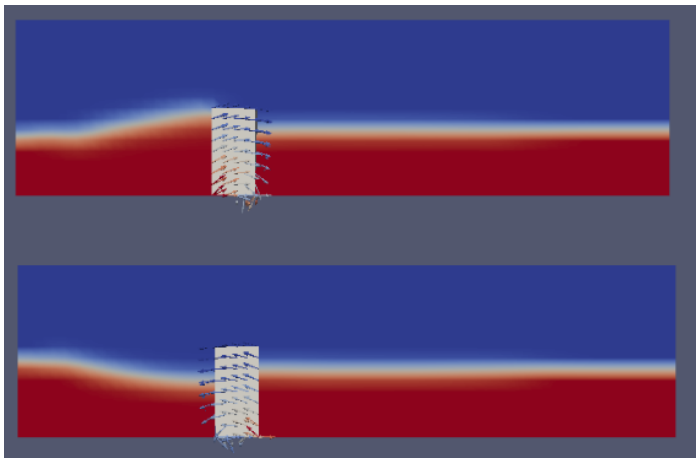


Figure 4.3: The interaction of the beam with incoming waves: when the beam is hit by a wave crest, it deforms forward (above); when the beam is hit by a wave trough, it deforms backward (below).

Reference

[1] Jasak H, Jemcov A, Tukovic Z, et al. OpenFOAM: A C++ library for complex physics simulations. In: International workshop on coupled methods in numerical dynamics. vol. 1000. IUC Dubrovnik, Croatia; 2007. p. 1–20.

[2] Weller HG, Tabor G, Jasak H, Fureby C. A tensorial approach to computational continuum mechanics using object-oriented techniques. Computers in physics. 1998;12(6):620–631.

[3] Hirt CW, Nichols BD. Volume of fluid (VOF) method for the dynamics of free boundaries. Journal of computational physics. 1981;39(1):201–225.

[4] Moctar Oe, Shigunov V, Zorn T. Duisburg Test Case: Post-panamax container ship for benchmarking. Ship Technology Research. 2012;59(3):50–64.

[5] Jacobsen NG, Fuhrman DR, Fredsøe J. A Wave Generation Toolbox for the Open-Source CFD Library: OpenFoam®. International Journal for Numerical Methods in Fluids. 2012;70(9):1073–1088.

[6] Higuera P, Lara JL, Losada IJ. Realistic wave generation and active wave absorption for Navier–Stokes models: Application to OpenFOAM®. Coastal Engineering. 2013;71:102–118.

[7] Jacobsen N. waves2Foam Manual. Deltares, The Netherlands. 2017;.

[8] Mayer S, Garapon A, Sørensen LS. A fractional step method for unsteady free-surface flow with applications to non-linear wave dynamics. International Journal for Numerical Methods in Fluids. 1998;28(2):293–315.

[9] Bruinsma N. Validation and application of a fully nonlinear numerical wave tank (Master Thesis, TU Delft). 2016;.

[10] Higuera P, Lara JL, Losada IJ. Three-dimensional interaction of waves and porous coastal structures using OpenFOAM®. Part I: formulation and validation. Coastal Engineering. 2014;83:243–258.

[11] Higuera P, Lara JL, Losada IJ. Three-dimensional interaction of waves and porous coastal structures using OpenFOAM®. Part II: Application. Coastal Engineering. 2014;83:259–270.

[12] Tukovic Z, Cardiff P, Karac A, Jasak H, Ivankovic A. OpenFOAM library for fluid structure interaction. In: 9th OpenFOAM Workshop. vol. 2014; 2014. .

[13] Tukovic Z, Karac A, Cardiff P, Jasak H, Ivankovic A. OpenFOAM finite volume solver for fluid-solid interaction. Tractions of FAMENA. 2018;.

[14] Huang L, Ren K, Li M, Tuković Ž, Cardiff P, Thomas G. Fluid-structure interaction of a large ice sheet in waves. Ocean Engineering. 2019;182:102–111.

[15] Huang L. An opensource solver for wave-induced FSI problems. In Proceedings of CFD with-OpenSource Software; 2018.

Appendix A

Coding tutorial

Chapter 1

Basic Coding and Compilation

OpenFOAM is written in the object-oriented C++ language. Users of C++ will be familiar with many concepts such as inheritance and polymorphism, concepts which are used extensively in OpenFOAM to reduce code duplication and improve efficiency.

However, much of the core functionality of C++ has been overloaded in OpenFOAM to allow for the code to be written in a more mathematical way. We shall see many examples of this over the course of this introduction.

OpenFOAM uses the compiler **wmake** which is similar to cmake. wmake comes as part of OpenFOAM and it is highly recommended that this is used for all functions, utilities and libraries to avoid any errors in compilation and interpretation.

In this chapter, the basic elements of programming and compilation are introduced. This is done through a number of examples which introduce some of the different data types and member functions that can be used.

1.1 A Simple Example

The first example introduces some of the basic concepts using the C standard library. We shall write a simple program, compile it using wmake, and then execute it.

1.1.1 A Simple Example with C++

The program take 2 inputs, performs some simple arithmetic and then display the outputs the terminal. Firstly, create a new folder in the working directory:

$ mkdir example1
$ cd example1

now open a text editor to write the code for example1.C. This is shown below:

```
#include <iostream> // header file that contains input/output functionality

int main() // declare main function
{
    using namespace std;

    int a=5;    // declare an integer
    double pi=3.141593; // declare a double

    cout << "Some basic arithmetic... \n";
    cout << a << " + " << pi << " = " << a+pi << "\n";
    cout << a << " * " << pi << " = " << a*pi << "\n";

    return 0; // main was declared as an integer so return a dummy integer value at the end

}
```

Figure 1.1: Code for example 1

To compile the code, it is necessary to create a directory called **Make** which must contain 2 files, **files** and **options**. OpenFOAM has created a function that will do this for us:

$ wmakeFilesAndOptions

This creates the necessary folder and files and pre-populates them. **files** contains the list of programs to compile and also contains the destination for the compiled code. **options** contains information about dependencies. Before we compile the code, make the following change to **files**:

```

example1.C

EXE = $(FOAM_USER_APPBIN)/example1
```

Figure 1.2: Contents of Make/files for example 1

This makes sure that the code we create is stored in a different place to the OpenFOAM source code.

Execute the code by typing the name into the prompt:

$ example1

1.1.2 A Simple Example Using OpenFOAM

We will now consider the same example, but using a different set of class definitions. Standard object types such as *int* and *double* are now replaced with *label* and *scalar*, which are OpenFOAM classes.

Make a new folder, called **example2**. Go into this folder and create the following code, called **example2.C**.

```
#include "IOstreams.H" // header file for OpenFOAM I/O streaming

using namespace Foam;

int main()
{
    label a=5; // label is the OpenFOAM version of integer
    scalar pi=3.141593; // similar to double

    // Info is similar to cout, but also works in parallel applications
    Info << "Some basic arithmetic... " << endl;
    Info << a << " + " << pi << " = " << a+pi << endl;
    Info << a << " * " << pi << " = " << a*pi << endl;

    return 0;
}
```

Figure 1.3: Code for example 2

Run the **wmakeFilesAndOptions** and change the location of the executable to FOAM_USER_APPBIN as before. Compile the code.

Running this, we note that it is exactly the same as the previous example. However, there are significant benefits to using OpenFOAM's core classes instead of those from the C standard library. In particular, we shall see in the next section that by using the OpenFOAM class definitions we are able to perform vector and tensor algebra and calculus in a more intuitive manner.

1.2 OpenFOAM Core Classes

1.2.1 Vectors and Tensors

In the previous section, the two classes *label* and *scalar* were introduced as alternatives to *int* and *double*. By using OpenFOAM's core classes, it is possible to perform vector and tensor algebra in a more intuitive, mathematical way. Both the *vector* and *tensor* classes are only valid for 3 dimensional data, and are primarily for data of the type $\phi_i = \phi_i(x, y, z)$.

The following code includes a number of examples that show how this can be used.

To begin with, create a new folder called **example3** and create the code below

and compile using the steps described previously.

```
#include "IOstreams.H"
#include "fieldTypes.H" // header file containing vector, tensor, etc.

using namespace Foam;

int main()
{

    scalar a=5.2;   // define a scalar
    vector m(1, 2, 3);  // define a 3x1 vector
    vector n(0.5, 0.25, 1);
    tensor P(1, 2, 3, 4, 5, 6, 7, 8, 9); // define a 3x3 tensor

    Info << "vector multiplied by a scalar:" << endl;
    Info << a << "*" << m << " = " << a*m << endl;

    Info << "sum of 2 vectors: " << endl;
    Info << m << " + " << n << " = " << m+n << endl;

    Info << "dot product of 2 vectors:" << endl;
    Info << m << " & " << n << " = " << (m & n) << endl;

    Info << "Tensor times a vector:" << endl;
    Info << P << " & " << m << " = " << (P & m) << endl;

    return 0;

}
```

Figure 1.4: Code for example 3

Note the use of brackets when combined with the & operator. This is necessary because in C++ the << operator takes precedence over the & operator.

Compile this code and execute.

1.2.2 Lists and Fields

For storing and processing large vectors or arrays of data, we use the *List* class. This is actually a template class, and it inherits different functionality depending on the class of data that make up the list. For example, we might have a list of n scalars, which would simply be a $n \times 1$ list of scalar values. Or, we might have a list of velocity vectors of the form:

(u_1, v_1, w_1)
(u_2, v_2, w_2)
...
(u_n, v_n, w_n)

Let's consider an example of how the *List* class can be used. The following example produces a finite sequence and stores the results of each iteration in a list. As well as using the basic class, we shall also make use of some of the class member functions. The code will be used to evaluate:

$$\sum_{n=0}^{\infty} \frac{1}{2^n} = 2 \tag{1.1}$$

Because the sum is over an infinite n, we shall introduce a convergence criterion to stop the calculation once a certain level of convergence has been reached. This also provides the opportunity to introduce two of the member functions for the class, namely *size* and *last*.

Begin by creating a new folder, called **example4** and write out the code below in a text file with the same name. Then follow the compilation steps from the previous examples, remembering to change the compilation location in **files**.

```
#include "IOstreams.H"
#include "List.H" // need to include header file for List
#include <math.h>

using namespace Foam;

int main()
{

    List<scalar> myList(1); // declare a list with 1 scalar value
    myList[0] = 1; // set the first value equal to 1
    scalar total = myList[0];
    scalar convergenceCriterion = 1E-5;
    scalar convergenceValue = 1;
    label n = 1;
    scalar exponent = -1;

    while (convergenceValue > convergenceCriterion)
    {
        scalar newValue = pow((pow(2,n)),exponent); // calculate new value
        myList.append(newValue); // increases the list length by one and adds the newValue
        total += myList.last(); // compound assignment, same as total = total +
myList.last()
        n = n+1;

        convergenceValue = myList[myList.size() - 2] - myList.last(); // check convergence
        // N.B. in C++, indexing of lists, vectors, etc. starts at 0, not 1!

    }

    Info << "myList = " << myList << endl;
    Info << "Sum = " << total << endl;

    return 0;
}
```

Figure 1.5: Code for example 4

An extension of the *List* class is Field. *Field* inherits all of the functionality of *List*, but it also includes field algebra. We shall see examples of this in the next chapter.

Chapter 2

Creating Utilities and Data Access

One of the most important elements of programming within OpenFOAM is data access. We need this to read information about the mesh and to access field data. In this chapter, we are going to create a utility to calculate the volume ratios of neighbouring cells within a mesh. High volume ratios can lead errors and instabilities in simulations, particularly for LES and DNS.

The example introduces a number of important concepts, including the template for developing an application, accessing data and the use a number of member functions for different class types.

2.1 Creating a New Application or Utility

To begin, we are going to use an OpenFOAM utlity that creates a template for the application:

$ foamNewApp volumeRatioCheck

This creates a new directory and populates it with a template code and the Make directory. Go into this directory and compile the code to check there are no errors.

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Copyright (C) 2019 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Application
    volumeRatioCheck

Description

\*---------------------------------------------------------------------------*/

#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;
}
```

Figure 2.1: Template code for volumeRatioCheck

2.2 Volume Ratio Check

The first line of code, shown in figure 2.1, adds a header file **fvCFD.H**. This header actually contains a number of other header files, which is done to reduce code duplication, save time and make the code easier to read. We can see which header files are contained in this by looking at the source code, shown in figure 2.2.

The code shown in the following figures is split into 3 sections. The first contains a number of header files that are necessary for the code to run. A number of access functions are also used to access the necessary mesh data.

The second part, see figure 2.4, contains the loop that calculates the volume

```
#ifndef fvCFD_H
#define fvCFD_H

#include "parRun.H"

#include "Time.H"
#include "fvMesh.H"
#include "fvc.H"
#include "fvMatrices.H"
#include "fvm.H"
#include "linear.H"
#include "uniformDimensionedFields.H"
#include "calculatedFvPatchFields.H"
#include "extrapolatedCalculatedFvPatchFields.H"
#include "fixedValueFvPatchFields.H"
#include "zeroGradientFvPatchFields.H"
#include "fixedFluxPressureFvPatchScalarField.H"
#include "constrainHbyA.H"
#include "constrainPressure.H"
#include "adjustPhi.H"
#include "findRefCell.H"
#include "IOMRFZoneList.H"
#include "constants.H"

#include "OSspecific.H"
#include "argList.H"
#include "timeSelector.H"

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

#endif
```

Figure 2.2: Code for fvCFD.H

ratio of each cell with it's neighbours. These are stored in a list using the **append** function, which was introduced earlier.

The third part of the code prints the maximum volume ratio to the screen, together with the run-time of the application.

Modify the template code using the figures above, recompile and check there are no compilation errors.

We are now ready to test the utility to see how it works. To do this, copy the pitzDaily tutorial into your working directory and run the blockMesh utility to create the mesh:

$ run
$ cp -r FOAM_TUTORIALS/incompressible/simpleFoam/pitzDaily .
$ cd pitzDaily
$ blockMesh
$ volumeRatioCheck

The utility runs and returns the largest volume ratio to the screen.

```
31 #include "fvCFD.H" // contains lots of other header files
32
33 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
34
35 int main(int argc, char *argv[]) // allows for a number of input arguments
36 {
37     #include "setRootCase.H"
38     #include "createTime.H" // creates time information
39
40     // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
41
42     // This header allows us to access mesh information
43     #include "createMesh.H"
44
45     // Access function tells us number of mesh elements
46     const scalar c = mesh.C().size();
47     Info << "Number of cells = " << c << endl;
48
49     // obtain neighbour cells for all cells
50     const labelListList& neighbour = mesh.cellCells();
51
52     List<scalar> ratios(0); // declare an empty scalar list
53     scalar volumeRatio = 0;
```

Figure 2.3: Code for volumeRatioCheck (lines 31-53)

```
55     forAll(neighbour, celli)
56     {
57
58         // for each cell, obtain neighbour cells
59         List<label> n = neighbour[celli];
60
61         // calculate the volume ratio of the cell with its neighbour cells
62         const scalar cellVolume = mesh.V()[celli];
63
64         // for each neighbour cell, calculate volume ratio
65         forAll(n, i)
66         {
67             label neighbourIndex = n[i];
68             scalar neighbourVolume = mesh.V()[neighbourIndex];
69
70             // want volume ratio to be greater than 1
71             if (neighbourVolume >= cellVolume)
72             {
73                 volumeRatio = neighbourVolume/cellVolume;
74             }
75             else if (neighbourVolume < cellVolume)
76             {
77                 volumeRatio = cellVolume/neighbourVolume;
78             }
79
80             // update list
81             ratios.append(volumeRatio);
82
83         }
84
85     }
```

Figure 2.4: Code for volumeRatioCheck (lines 55-85)

For this example, the utility runs quickly but that may not be the case for meshes with more elements. If you try to run to the utility for a larger mesh, you will find that it takes significantly longer to run. The reason for this can be found on line 81 of the code. The **append** member function for the *list* class, whilst convenient, is very inefficient. This is because the function actually creates a new list with n+1 elements and then copies the old list into it before adding the new data. Thus, as the list size increases, more and more data

```
87    Info << "Maximum volume ratio = " << max(ratios) << endl;
88
89    // By default, OpenFOAM puts in this piece of code which prints the
90    // run-time to the screen
91    Info<< nl << "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
92        << "  ClockTime = " << runTime.elapsedClockTime() << " s"
93        << nl << endl;
94
95    Info<< "End\n" << endl;
96
97    return 0;
98 }
```

Figure 2.5: Code for volumeRatioCheck (lines 87-98)

has to be copied from the old list to the new one, which becomes increasingly expensive.

To get around this problem, it is better to allocate the memory beforehand. The simplest way to do this is simply to declare a list that will be large enough to contain all of the necessary values. We can do this by making use of a different class constructor for *list*, which allows us to specify a size and populate it with zeros. This particular constructor requires two inputs, which must be of type *label* and *scalar*. The figures below, 2.6 and 2.7 show how we can modify the code to make use of this.

```
52    label len = 10*mesh.C().size();
53    scalar initial = 0;
54    List<scalar> ratios(len, initial); // allocate memory for list
55    label counter = 0;
```

Figure 2.6: Code modification for volumeRatioCheck (lines 52-55)

We can then assign the value from each iteration to the correct location in the list:

```
83            // update list
84            ratios[counter] = volumeRatio;
85            counter += 1; // compound assigment
```

Figure 2.7: Code modification for volumeRatioCheck (lines 83-85)

Reompile the code and re-run the application in the pitzDaily directory.

2.3 Creating a Dictionary for Data Input

In many cases, we want to supply utilities and applications with information to tell them how to run and what to calculate. One way of doing this is through a **dictionary**. This is simply a text file that contains a number of inputs that can be read by the code.

To illustrate this, we shall introduce a maximum volume ratio criterion to our application. As well as calculating the maximum ratio, the code will also tell

us how many cell volume ratios exceed a certain criterion. This criterion will be set by the user by way of a dictionary.

The new code for the application is shown in figures 2.8, 2.9 and 2.10.

```
31 #include "fvCFD.H" // contains lots of other header files
32
33 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
34
35 int main(int argc, char *argv[]) // allows for a number of input arguments
36 {
37     #include "setRootCase.H"
38     #include "createTime.H" // creates time information
39
40     // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
41
42     #include "createMesh.H"
43
44
45     // Create IO object using information from dictionary
46     IOdictionary volumeRatioDictionary
47     (
48         IOobject
49         (
50             "volumeRatioDictionary", // name of file/object
51             runTime.system(), // file lives in system directory
52             mesh, // associate with  mesh
53             IOobject::MUST_READ, // read file
54             IOobject::NO_WRITE // code does not modify anything
55         )
56     );
57
58     // Access function tells us number of mesh elements
59     const scalar c = mesh.C().size();
60     Info << "Number of cells = " << c << endl;
61
62     // obtain neighbour cells for all cells
63     const labelListList& neighbour = mesh.cellCells();
64
65     label len = 10*mesh.C().size();
66     scalar initial = 0;
67     List<scalar> ratios(len, initial); // allocate memory for list
68     label counter = 0;
69     scalar volumeRatio = 0;
70
71     label nFail = 0; // declare counter for number of cells that exceed criterion
72
73     // read in criterion from diction
74     scalar maxRatio(readScalar(volumeRatioDictionary.lookup("maxRatio")));
75
```

Figure 2.8: New code volumeRatioCheck (lines 31-75)

Lines 46 to 56 now contain the definition for the I/O (input/output) object of class *IOdictionary*. Line 74 reads in the specific information from the dictionary using both the lookup function and the readScalar function.

An additional *if* statement is now used to check if a cell volume ratio exceeds the criterion. A counter is used to determine the total number of failed cases. This is then printed to the screen.

Compile this new code as before.

In order to run the code, we now need to create a dictionary called **volumeRatioDictionary**. This needs to go in the **system** directory. This is shown in

```
76      forAll(neighbour, celli)
77      {
78
79          // for each cell, obtain neighbour cells
80          List<label> n = neighbour[celli];
81
82          // calculate the volume ratio of the cell with its neighbour cells
83          const scalar cellVolume = mesh.V()[celli];
84
85          // for each neighbour cell, calculate volume ratio
86          forAll(n, i)
87          {
88              label neighbourIndex = n[i];
89              scalar neighbourVolume = mesh.V()[neighbourIndex];
90
91              // want volume ratio to be greater than 1
92              if (neighbourVolume >= cellVolume)
93              {
94                  volumeRatio = neighbourVolume/cellVolume;
95              }
96              else if (neighbourVolume < cellVolume)
97              {
98                  volumeRatio = cellVolume/neighbourVolume;
99              }
100
101         // check if ratio exceeds criterion
102         if (volumeRatio > maxRatio)
103         {
104         nFail += 1; // increase fail counter by 1
105         }
106
107             // update list
108             ratios[counter] = volumeRatio;
109         counter += 1; // compound assigment
110
111         }
112
113     }
```

Figure 2.9: New code volumeRatioCheck (lines 76-113)

```
115     Info << "Maximum volume ratio = " << max(ratios) << endl;
116     Info << "Number of cell volume ratios exceeding " << maxRatio << " = " << nFail <<
    endl;
117     // By default, OpenFOAM puts in this piece of code which prints the
118     // run-time to the screen
119     Info<< nl << "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
120         << "  ClockTime = " << runTime.elapsedClockTime() << " s"
121         << nl << endl;
122
123     Info<< "End\n" << endl;
124
125     return 0;
126 }
127
```

Figure 2.10: New code volumeRatioCheck (lines 115-127)

figure 2.11.

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  6
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      volumeRatioDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

maxRatio  20;

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

Figure 2.11: Dictionary for volumeCheckDict

Chapter 3

Solver Development

OpenFOAM has a large number of solvers for a wide range of applications, but there may be times when you wish to add something to an existing solver, or implement a new one based on new research. In this chapter, we are going to add a scalar transport equation for temperature to the **icoFoam** solver, which is models an unsteady incompressible laminar flow.

By utilising the existing functionality of OpenFOAM and its unique architecture, implementing this model is surprisingly straightforward and requires only limited modifications to the existing code.

The temperature will be modelled as a conserved passive scalar. That is, it will not influence the pressure or velocity of the flow and so is only applicable to problems where the temperature changes are small. One-way coupling from momentum to temperature is accounted for through the convection term in the temperature equation:

$$\frac{\partial T}{\partial t} + \nabla \cdot (\mathbf{U}T) - \nabla \cdot D_T \nabla T = 0 \tag{3.1}$$

The modified solver will be called **icoThermalFoam** and its development is described in the following section.

3.1 icoThermalFoam

Begin by copying the source code for the icoFoam solver to your working directory

$ run

$ cp -r $FOAM_SOLVERS/incompressible/icoFoam icoThermalFoam $ cd icoThermalFoam
$ wclean
$ mv icoFoam.C icoThermalFoam.C

The **wclean** function removes the files generated during the compilation that were copied from the original folder. In the **Make/files** file, change icoFoam to icoThermalFoam. This file should now look as shown in figure 3.1.

```
icoThermalFoam.C

EXE = $(FOAM_USER_APPBIN)/icoThermalFoam
```

Figure 3.1: files for icoThermalFoam

The icoFoam solver uses the PISO algorithm to solve for the coupled pressure-velocity field. The main solver file, icoThermalFoam.C, contains the momentum prediction equation followed by the PISO loop. Open the .C file with a text editor and add the equation as shown in figure 3.2. This should be added below the PISO loop as it does not form part of the pressure-velocity coupling.

```
        // Insert the temperature equation
        solve
        (
            fvm::ddt(T) + fvm::div(phi, T) == fvm::laplacian(DT, T)
        );

        runTime.write();

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

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

    return 0;
}
```

Figure 3.2: Changes to icoThermaFoam.C

We also need to modify the **createFields.H** header file. This is included in the main code near to the start and reads in the field data and also the transport properties. We need to make two changes to this file to include the temperature field and also the thermal diffusivity, D_T.

```
// Add dimensioned scalar for thermal diffusivity
// This will be read from transportProperties in constant
dimensionedScalar DT
(
    "DT",
    transportProperties.lookup("DT")
);
```

Figure 3.3: Inclusion of thermal diffusivity in createFields.H

```
// Add a new volume scalar field for T
Info<< "Reading field T\n" << endl;
volScalarField T
(
    IOobject
    (
        "T",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
```

Figure 3.4: Inclusion of temperature field in createFields.H

3.2 Testing the Solver

We now want to test the solver to check it behaves as expected. This will be done using the cavityClipped tutorial. Begin by copying this into your working directory.

$ run
$ cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavityClipped .
$ cd cavityClipped
$ blockMesh

The cavityClipped tutorial has a horizontal velocity imposed on the top boundary which induces a circulating flow field in the cavity below. We are going to include a temperature differential on the boundaries where the top boundary has a higher temperature than the wall boundaries.

Before we run the solver, we must add the initial and boundary conditions for the temperature and add the thermal diffusivity to the transport properties. This is shown in figures 3.5 and 3.6.

We also need to add details to the **fvSchemes** and **fvSolution** files to tell OpenFOAM how to discretise and solve the new equation. The changes to these files are shown in figures 3.7 and 3.8.

Finally, we need to change the control dictionary so that the correct solver is called. This done by changing **icoFoam** to **icoThermalFoam** in the **system/controlDict** file.

Run the solver and load the results in paraview:

$ icoThermalFoam
$ paraFoam

The temperature field now appears alongside the pressure and velocity fields and so post-processing can be carried out in the usual manner.

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  6
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    location    "0";
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 0 0 1 0 0 0];

internalField   uniform 300;

boundaryField
{
    lid
    {
        type            fixedValue;
        value           uniform 400;
    }
    fixedWalls
    {
        type            fixedValue;
        value           uniform 300;
    }
    frontAndBack
    {
        type            empty;
    }
}


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

Figure 3.5: Initial and boundary conditions for T in the 0 directory

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

nu  [0 2 -1 0 0 0 0] 0.01;

DT  [0 2 -1 0 0 0 0] 0.001;


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

Figure 3.6: Transport properties

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

Figure 3.7: Modification of fvSchemes

```
34    "(U|T)"
35    {
36        solver          smoothSolver;
37        smoother        symGaussSeidel;
38        tolerance       1e-05;
39        relTol          0;
40    }
```

Figure 3.8: Modifications of fvSolution

Appendix B

Tutorial: Flow passes a motorbike

Chapter 1

Introduction

In this tutorial, we are going to work through the motorbike tutorial to model the steady-state, incompressible turbulent flow over a motorbike with a rider.

The following elements will be considered:

- Meshing: Creation of initial isotopic block mesh domain followed by the generation of an unstructured mesh including boundary layer elements using snappyHexMesh;
- Turbulence modelling: Selection of turbulence model and consideration of wall functions;
- Boundary and initial conditions;
- Numerical Methods including discretisation schemes and solvers
- Post-processing using function objects and Paraview

The principle parameters of the simulation are:

- Simulation type: steady state, turbulent RANS
- Turbulence: 2-equation $k - \omega SST$ model with wall functions
- Motorbike geometry bounding box is approx $2.0 \times 0.8 \times 1.3\ m$
- Flow velocity: $|\mathbf{U}| = 20\ ms^{-1}$
- Kinematic viscosity: $\nu = 1.5 \times 10^{-5}\ m^2 s^{-1}$
- Characteristic length: $L \approx 0.5\ m$

- Reynolds number: $Re = 6.7 \times 10^5$

Begin by downloading the tutorial files into your working directory.

Chapter 2

Geometry and Meshing

2.1 Domain Generation

To create the mesh around the motorbike, we first need to create the domain, i.e. the bounding box, that will be meshed. This, together with an isotropic block mesh are created using the **blockMesh** utility. This utility, as with other, has a dictionary located in the system directory that tells it what to do and how to run. Start by opening the dictionary:

$ gedit system/blockMeshDict

Creating the block mesh is done by typing the utility name into the prompt.

$ blockMesh

By default, the utility will look for a dictionary in the system directory called **blockMeshDict** but users can specify other files as well. This is done by typing:

$ blockMesh -dict *nameOfDictionary*

This is true of most utilities including **topoSet**, **snappyHexMesh**, **surfaceFeatureExtract**, etc. All have a default dictionary which has the utility name followed by **dict**.

2.2 Identifying Features and Edges

The second meshing step uses the **surfaceFeatureExtract** utility, which can be used to explicity determine feature edges. This is important for complex

geometries as edges often have to be refined to a higher level that the surrounding mesh. You can open the dictionary for this utility by typing

$ gedit system/surfaceFeatureExtractDict

Run this utility by typing:

$ surfaceFeatureExtract

2.3 Unstructured Grid Generation

The third step is to create the unstructured mesh around the motorbike. To do this, we are going to use the **snappyHexMesh** utility, which is OpenFOAM's unstructured mesh generator. This consists of three steps:

1. Castellate - inserts geometry and performs refinement of edges, surfaces, volumetric regions, etc;
2. Snapping - Mesh is snapped to surface. Displacement/smoothing iterations carried out;
3. Layer Addition - Layers at walls. Existing mesh is "shrunk" back from wall to allow for layers to be inserted.

The dictionary for **snappyHexMesh** contains a large number of comments that explain each step. Open this up:

$ gedit system/snappyHexMeshDict

For the layer insertion, we need to estimate the required distance of the first grid point from the wall, Δy. For a complex geometry, this is difficult to do *a priori* and so it is common to use a flat-plate formula to estimate the required size and then check this as part of the post-processing. There are many possible formulas for the prediction of the skin friction. We shall use the Prandtl 1/7th power law:

$$C_f = \frac{0.027}{Re^{1/7}} \tag{2.1}$$

The friction velocity can then be determined using

$$u_\tau = |\mathbf{U}|\sqrt{C_f/2} \tag{2.2}$$

This can be used to determine the distance of the first grid point from the wall using

$$\Delta y = \frac{y^+ \nu}{u_\tau} \tag{2.3}$$

For our case, we choose a target $y^+ = 200$, which is consistent with a wall function approach. Therefore, we have that $\Delta y \approx 3\ mm$. The isotropic mesh generated at the start generates cells of size $1\ m \times 1\ m \times 1\ m$ and the level of refinement on the surface is 5-6 levels. Therefore, the average cell length near the wall, prior to snapping, is approximately 0.02 m. This allows us to set the require relative cell size for the layer additions. In practice, it is important to consider the following when creating this type of mesh:

- Height of first grid point, (Δy)
- Number of layers to be added (30 or more required for wall-resolved approaches)
- Growth ratio of layers (should ideally be no more than 1.3)
- Change in size from boundary layer mesh to outer mesh. This needs to be as smooth as possible to avoid interpolation errors

It should be noted that due to the time constraints of this course, the geometry and mesh used in this tutorial do not satisfy these criteria. We shall see this later on during the post-processing.

We now wish to generate the unstructured mesh. By default, the utility saves each of the three stages separately and stores them in sequential time directories. For this tutorial, we don't want this to happen so we add the following flag when calling the utility:

$ snappyHexMesh -overwrite

Each stage of the process is visualised in the following images, showing the geometry, castellated and snapped meshes.

snappyHexMesh will only add layers where they satisfy the quality criteria defined in the **meshQualityDict** located in the system directory. For coarse meshes and complex geometries, this can lead to the coverage of the layers being very poor. We can visualise this in Paraview, and this is shown in the figure below:

It can be seen in figure 2.2 that the layer coverage is quite poor, at around 45%. This is primarily due to the coarseness of the mesh, but also the coarseness

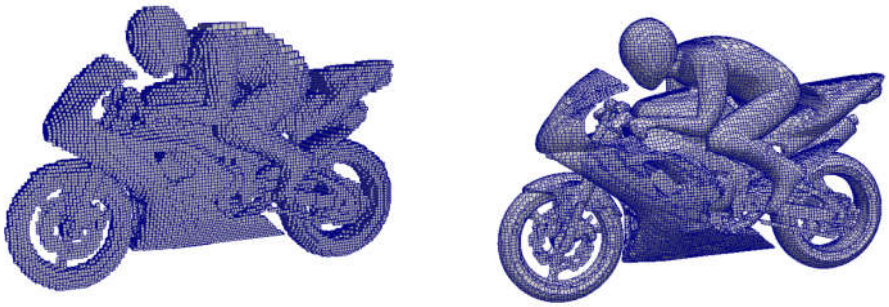


Figure 2.1: Castellated (left) and snapped (right) Mesh.

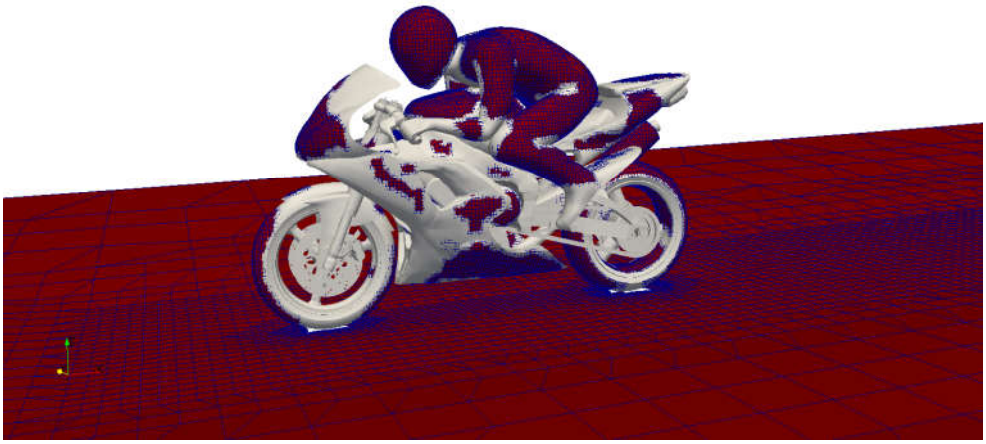


Figure 2.2: Layer addition. Surface where layers added shown in red.

of the original geometry. A high quality mesh with good layer coverage for a geometry of this complexity would require many millions of elements.

2.4 Mesh Quality

It can be difficult to define a "good" mesh in terms of quality. There are many different metrics that can be applied and generating very high quality meshes around complex geometries can be very time-consuming. It is therefore important to understand what the mesh quality metrics are and how they relate to the mathematics. The required mesh quality also depends on the type of simulation you are running. For examlpe, methods such as large eddy simulation are far more sensitive to mesh quality than RANS simulations.

The mesh quality can be assessed using the **checkMesh** utility. This uses a number of different criteria to assess the quality of a mesh, and three of the more important criteria are described here.

2.4.1 Non-orthogonality

The non-orthogonality of a face is determined by the angle made between the line connecting the 2 nodes either side of the face and and the face normal. This is illustrated in 2.3. Non-orthogonality primarily affects the diffusion terms and should be kept as low as possible. The default maximum in OpenFOAM is $\theta_{NO} < 65°$. Values higher than this are likely to introduce instabilities into the simulation.

It is important to note that mesh non-orthogonality will not be improved by isotropic mesh refinement as this does not alter the angles. Instead, modifying the actual design of the mesh will be required, for example by changing the refinement regions or improving the mesh grading. It is also possible to "correct" the non-orthogonality through the discretisation scheme. This will be shown later.

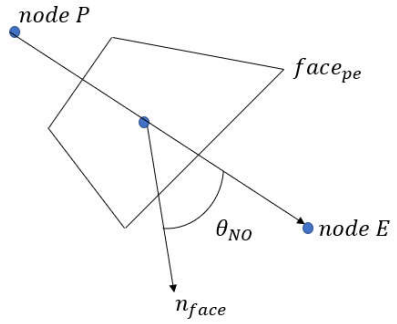


Figure 2.3: Face non-orthogonality defined as the angle between $\overrightarrow{PE}$ and n_{face}

2.4.2 Skewness

Another common measure of mesh quality is cell skewness. To understand this, it is important to remember that the finite volume discretisation requires the interpolation of field data from nodes to faces. If the cell skewness is small (¡ 1), then the line between 2 adjacent nodes will pass through the face co-owned by those 2 nodes. If the skewness is greater than 1, then the line does not intersect the face. This is demonstrated in figure 2.4

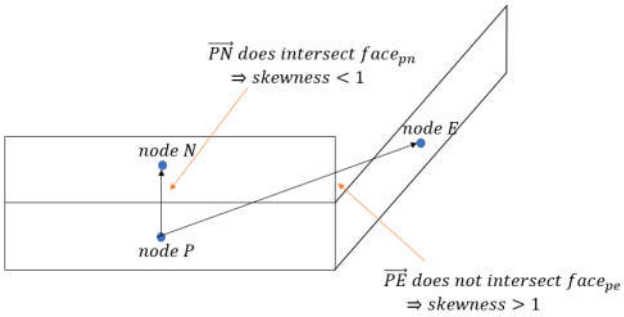


Figure 2.4: Illustration of cell skewness

In OpenFOAM, it is recommended that skewness should be below 4. Higher skewness does not necessariy lead to instabilities but it does reduce the accuracy of the simulation. For unsteady turbulent-resolving simulations (e.g. LES and DNS), the skewness must be kept as low as possible (ideally less than 1 across the domain).

2.4.3 Aspect Ratio

The aspect ratio is a measure of the squareness of a cell. For a 3D cell of arbitrary type, it is defined in OpenFOAM as

$$\sigma_{ar} = \sum_{f} \frac{|S_f|}{6V^{2/3}} \tag{2.4}$$

For a cube, $\sigma_{ar} = 1$. High aspect ratio cells make it more difficult for the matrix equations more difficult to solve and can lead to convergence issues. Skewness often results from high aspect ration cells, as can be seen in figure 2.4. The default maximum aspect ratio in OpenFOAM is 1000. However, it is advisable to keep well below the limit, particularly in regions of high pressure gradients.

Chapter 3

Turbulence Modelling

This tutorial uses a RANS approach for the turbulence modelling. Therefore, all scales of turbulent motion are modelled, as opposed to resolved. There are a number of turbulence models available in OpenFOAM from simple 1-equation models up to more complex models.

In this tutorial we shall use the 2 equation $k - \omega$ SST model which is a popular and well-validated model for this type of flow and is used widely in both industry and academia.

The model consists of 2 transport equations for the turbulent kinetic energy, k and the specific turbulent dissipation rate, ω.

When using this turbulence model, either a wall-resolved or a wall-modelled approach can be taken. For the wall-resolved approach, it is necessary to have a near wall resolution of $y^+ \leq 1$. For higher Reynolds number cases, this can result either in very high near-wall cell counts, or very high aspect ratios. Alternatively, wall functions can be used to model the behaviour of the inner-most part of the boundary layer. When this approach is used, the y^+ should be greater than 30 but less than 300, which places it within the fully turbulent part of the boundary layer.

In OpenFOAM, the turbulence model is defined in the **turbulenceProperties** file in the constant directory. The boundary conditions and wall functions are defined in the relevant files within the 0 directory, which is covered in the following section.

$ gedit constant/turbulenceProperties

Chapter 4

Boundary and Initial Conditions

In OpenFOAM, all initial and boundary conditions must be explicitly defined. Users of commercial CFD codes may be familiar with simply defining a "velocity inlet" or "pressure outlet" and providing a turbulence intensity. Mathematically, it is necessary to provide boundary conditions for every flow variable on every boundary. Commercial codes simplify this process by automatically defining many of the boundary conditions for you, whereas OpenFOAM does not. Whilst adding to the overall complexity of the problem, this does give the user far more flexibility and it also aids with trouble-shooting and can help to identify sources of divergence or lack of convergence, which all CFD users will experience.

The boundary conditions used in this tutorial are shown 4.1.

	p	**U**	k	ω	ν
inlet	zeroGradient	fixedValue	fixedValue	fixedValue	calculated
outlet	fixedValue	inletOutlet	inletOutlet	inletOutlet	calculated
motorBike	zeroGradient	fixedValue	kqRWall-Function	omegaWall-Function	nutkWall-Function
lowerWall	zeroGradient	fixedValue	kqRWall-Function	omegaWall-Function	nutkWall-Function
sideWalls, upperWall	slip	slip	slip	slip	calculated

Table 4.1: Boundary condition types for each variable and boundary

We need to define what the values of the turbulent quantities k and ω are at the boundaries and what their initial values are. A suitable value for k can be derived from the turbulence intensity:

$$k = \frac{3}{2}(|\mathbf{U}|I)^2 \tag{4.1}$$

The value of omega can be determined from the value of k and from a length-scale. This can be difficult to know given that turbulence, in reality, exists over a wide range of scales. For this example, we shall assume that the length-scale is of the order of 0.5m, which is commensurate with the lengthscales of the different surfaces that make up the geometry. In practice, simulations of external aerodynamic flows should not be overly sensitive to this. Using this lengthscale, the inlet and initial value of omega is

$$\omega = \frac{C_\mu^{0.75} k^{1.5}}{L} \tag{4.2}$$

where $C_\mu = 0.09$.

Despite the turbulent viscosity, ν_t not being solved for, it is necessary to define it at boundaries in OpenFOAM. For this purpose, we use the *calculated* boundary condition which tells OpenFOAM that it will be calculated by the turbulence model.

The boundary conditions and initial conditions are contained within the 0 directory. For this tutorial, the initial conditions have been placed in a separate file which is then read into each of the field variable files using the #include directive. The initial conditions and the boundary conditions file for velocity are shown below:

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  plus                                  |
|   \\  /    A nd           | Web:      www.OpenFOAM.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/

flowVelocity        (20 0 0);
pressure            0;
turbulentKE         0.24;
turbulentOmega      1.78;

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

Figure 4.1: Initial conditions file

You may notice that, for some boundary conditions, a value is assigned where the value should actually be calculated, either by the turbulence model or the wall function. The reason we assign a value is because Paraview needs something to read in for the 0 time-step. Otherwise, it will give an error message and may crash. When the solver is run, these dummy values will be over-written by the appropriate values.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  plus                                  |
|   \\  /    A nd           | Web:      www.OpenFOAM.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    location    "0";
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

// copies in the lines from the initial conditions file
#include        "include/initialConditions"

// all flow variables are dimensional
dimensions      [0 1 -1 0 0 0 0];

// This is the initial condition for the flow. The solver will update this as it goes...
internalField   uniform $flowVelocity;

boundaryField
{
    inlet
    {
        type  fixedValue;
        value $internalField; // $ sign used to reference a value
    }

    outlet
    {
        type            inletOutlet;
        inletValue      uniform (0 0 0);
        value           $internalField;
    }

    lowerWall
    {
        type            fixedValue;
        value           uniform $flowVelocity;
    }

    motorBikeGroup
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }

    upperWall
    {
        type slip;
    }

    frontAndBack
    {
        type slip;
    }
}


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

Figure 4.2: Boundary conditions file for velocity

Chapter 5

Numerical Methods

Now that we have defined the domain, mesh and boundary conditions, it is necessary to decide how to discretise and solve the equations for mass, momentum and the turbulence quantities. This is done in 2 files which live in the system directory, namely **fvSchemes** and **fvSolution**.

5.1 Discretisation schemes

OpenFOAM offers a much wider range of discretisation schemes than most commercial CFD codes. However, it is likely that you will only ever need a small number of those available.

The discretisation schemes are set in the **fvSchemes** file in the system directory. This is split into a number of sub-dictionaries:

- ddtSchemes: Time derivatives
- divSchemes: Divergence terms, including the all-important momentum convection term
- gradSchemes: Schemes for gradient terms
- laplacianSchemes: Schemes for Laplacian terms

It is possible to define a specific scheme for the discretisation of every term in every equation, but this is not usually necessary. Instead, it is common to use assign the same scheme for all terms of the same type.

In this tutorial, the simulation is steady-state and so we use the **steadyState** dummy scheme for the time derivatives. The second-order upwind scheme

linearUpwind is used for the discretisation of the momentum convection term and the first-order **upwind** scheme is used for the turbulent convection terms. The first-order upwind scheme should generally be avoided due to it's poor accuracy and highly dissipative properties. In particular, it should never be used for the discretisation of the momentum convection term. It's dissipative nature does help with stability however, and it can be used at the start of a simulation to better allow for the flow field to move away from un-physical initial conditions.

The gradient schemes are set as linear which is the second-order central scheme. This is also used for the Laplacian terms, although we use the **linear corrected** scheme here. The correction is to account for the non-orthogonality in the mesh, which as discussed earlier, affects the diffusion (Laplacian) terms.

5.2 Solvers

The solvers, as defined in **fvSolution** are responsible for solving the discretised matrix equations at each time-step.

In this tutorial, The geometric multi-grid solver (GAMG) is used for the pressure equation and smooth solvers are used for the momentum and scalar equations.

The SIMPLE-Consistent method is used with relaxation factors applied to the pressure, velocity and turbulent fields to improve the stability of the simulation.

Chapter 6

Running the Simulation

The simulation is controlled from the **controlDict** in the system directory. This defines what solver is to be used as well as information about the time-step, write control and any function objects.

We shall record information about the forces using a forceCoeffs function object. This is shown in figure 6.1. This will output the forces and coefficients to a text file for post-processing.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  plus                                  |
|   \\  /    A nd           | Web:      www.OpenFOAM.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/

forceCoeffs1
{
    type            forceCoeffs;

    libs            ("libforces.so");

    writeControl    timeStep;
    timeInterval    1;

    log             yes;

    patches         (motorBikeGroup);
    rho             rhoInf;       // Indicates incompressible
    rhoInf          1;            // Redundant for incompressible
    liftDir         (0 0 1);
    dragDir         (1 0 0);
    CofR            (0.72 0 0);  // Axle midpoint on ground
    pitchAxis       (0 1 0);
    magUInf         20;
    lRef            1.42;         // Wheelbase length
    Aref            0.75;         // Estimated

}


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

Figure 6.1: Function object for recording force data

To run the simulation, type the solver name into the prompt:

$ simpleFoam

To run in parallel, the domain must first be decomposed. The decomposition is handled by the utilit **decomposePar** and a dictionary is used to tell the utility how to carry it out. The scotch algorithm is selected in this tutorial and the domain will be split into 4 parts. To run the simulation in parallel:

$ decomposePar
$ mpirun -np 4 simpleFoam -parallel

When the simulation has finished, the domain can then be reconstructed using the **reconstructPar** utility:

$ reconstructPar -time latestTime

We can now use Paraview to visualise the results.

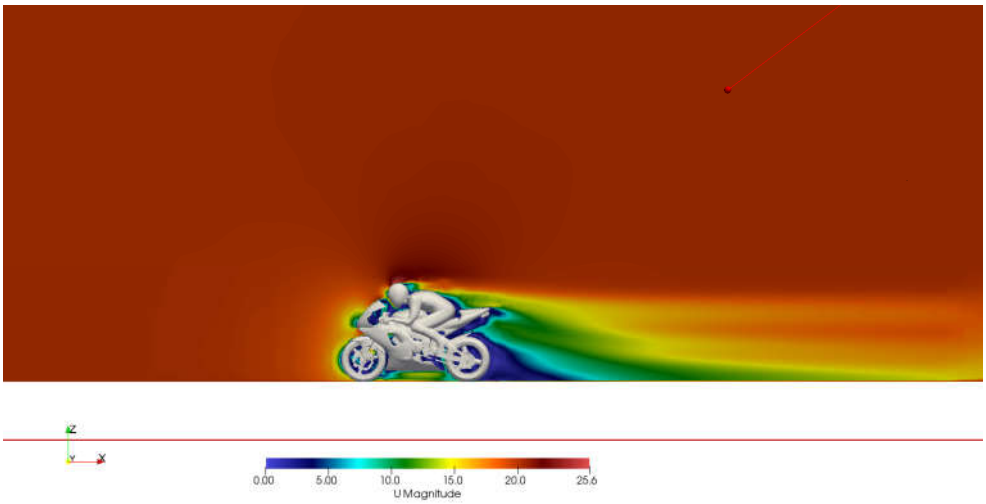


Figure 6.2: Velocity contour plane

Appendix C

Tutorial: Flow passes a cylinder

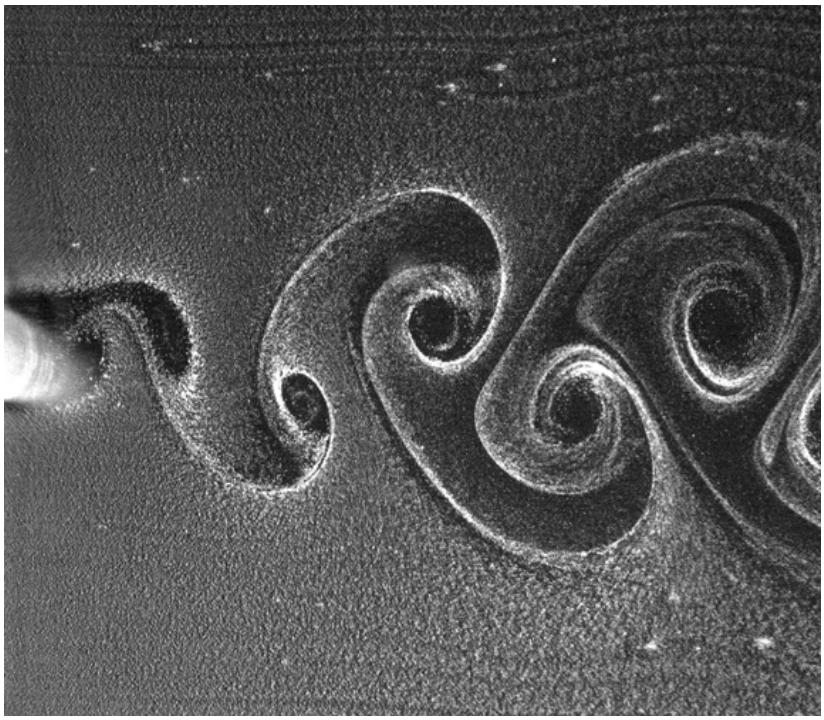


Figure 1: A typical von Karman vortex street behind a cylinder where hydrodgen bubbles were used to highlight the flowfield (unpublished).

1. Introduction

In many engineering applications it is necessary to analyse a stationary or moving object which is exposed to a uniform or turbulent flow, including automobile and plane aerodynamics, ship and submarine hydrodynamics, flow past oil riser pipes, flow through and around cities etc. The case that will be considered is the two-dimensional laminar flow past a cylinder which exhibits many interesting flow features (see figure 1), that are also present in more complex applications, including:

- boundary layer formation
- flow separation
- vortex shedding

In this tutorial the learning outcomes are:

- extract the the drag and lift coefficients on the cylinder
- pressure and friction coefficients on a bluff body
- take out velocity data in the flow domain

using the following Openfoam utilities

- forces object
- sample
- wallGradU
- probes

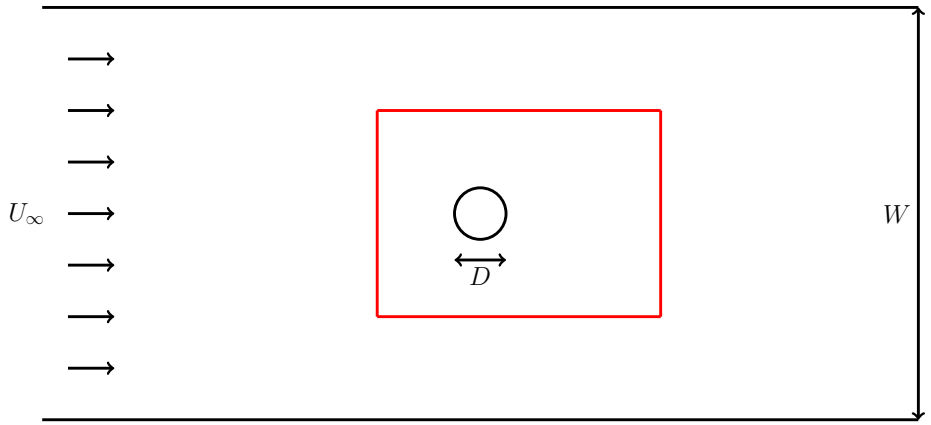


Figure 2: Schematic of the computational domain of width W to investigate the flow past a cylinder (of diameter D). The inlet boundary condition is $\boldsymbol{u} = \{U_\infty, 0\}$. The side walls have the no flux condition and zero shear condition. The cylinder boundary condition has the no flux and no-slip conditions ($\boldsymbol{u} = \boldsymbol{0}$) applied it.

1.1. *Problem statement*

A schematic of the problem is shown in figure 2, where a cylinder (diameter D) is placed $20D$ from the inlet in a domain of width $W = 40D$. The inlet boundary condition is $\boldsymbol{u} = \{u, v\} = \{U_\infty, 0\}$. The side walls have the no flux condition and zero shear condition. The cylinder boundary condition has the no flux and no-slip conditions ($\boldsymbol{u} = \boldsymbol{0}$) applied it and the normal gradient of the pressure is zero. The non-dimensional number that controls this physical system is the Reynolds number, $Re = U_\infty D/\nu$ where ν is the kinematic viscosity. θ is taken from the front stagnation point.

1.2. *Definition of diagnostics*

The force on a body is defined as

$$\boldsymbol{F} = \int_S (p\boldsymbol{I} - \boldsymbol{\tau}) \cdot \hat{\boldsymbol{n}} \mathrm{d}S,$$

where S is the surface of the body, p is the pressure, $\boldsymbol{I}$ is the identity matrix, $\boldsymbol{\tau}$ is the viscous stress tensor and $\hat{\boldsymbol{n}}$ is the normal pointing out of the fluid domain (Batchelor 1957). Note, a bold $\boldsymbol{F}$ indicates that it is a vector quantity, while the pressure is a scalar and therefore is not bold.

The mean and root mean square of a time series $\xi(t)$ are defined as

$$\frac{\bar{\xi}}{D} = \frac{1}{T_2 - T_1} \int_{T_1}^{T_2} \xi \mathrm{d}t, \quad \frac{\xi'}{D} = \sqrt{\frac{1}{T_2 - T_1} \int_{T_1}^{T_2} |\xi - \bar{\xi}|^2 \mathrm{d}t}. \tag{1.1}$$

In this work the time series ($\tau = tU_\infty/D$) of interest are the drag and lift coefficient which are the force coefficients in the streamwise and cross-stream direction, defined as

$$C_D(\tau) = \frac{F_D(\tau)}{1/2\rho U_\infty^2 A}, \quad C_L(\tau) = \frac{F_L(\tau)}{1/2\rho U_\infty^2 A}.$$

where A is a reference area. The force on a body is composed of a pressure and shear stress component, therefore it is insightful to also consider the pressure coefficient and

friction coefficient on the surface of the body. The pressure coefficient is defined as

$$C_p = \frac{p - p_\infty}{1/2\rho U_\infty^2},$$

where p_∞ is the free stream pressure. The potential flow solution for the pressure coefficient on the surface of the cylinder is given by

$$C_p = 2\cos(2\theta) - 1,$$

which provides a point of comparison. The friction coefficient is defined as

$$C_f = \frac{\tau_w}{1/2\rho U_\infty^2},$$

where the wall shear stress τ_w is

$$\tau_w = \mu \frac{\partial u_s}{\partial n}, \tag{1.2}$$

where $u_s = -u\sin\theta + v\cos\theta$ is the tangential velocity and n is the normal. Therefore,

$$\frac{\partial u_s}{\partial n} = -\frac{\partial u}{\partial n}\sin\theta + \frac{\partial v}{\partial n}\cos\theta. \tag{1.3}$$

This diagnostic is also of interest as flow separation occurs at $\tau_w = 0$.

The following section will go through the important aspects of the all the casefiles in the three folders '0', 'constant' and 'system'. Additionally, at the end, is a brief description of the utility 'WallGradU'.

1.3.1. *0*

U

The inlet is set to $\boldsymbol{u} = \{U_\infty, 0\}$ where $U_\infty = 1$. The no slip condition is applied to the cylinder. As two-dimensional simulations are carried out the boundaries 'frontandback' are set to 'empty'. The top and bottom are taken to be symmetry planes (for these boundaries the normal and normal gradient of $\boldsymbol{u}$ are set to zero).

p

On the cylinder the boundary condition is that the normal gradient of the pressure is zero. For the inlet and outlet the pressure is set to the freestream value. Again, as two-dimensional simulations are carried out the boundaries 'frontandback' are set to 'empty'. The top and bottom are taken to be symmetry planes (for these boundaries the normal and normal gradient of p are set to zero).

1.3.2. *constant*

RASProperties

The Reynolds averaged stress Model (RASModel) is set to laminar as $Re = 100$.

transportProperties

The diameter of the cylinder is $D = 0.1$ and the inlet flow is set to $U_\infty = 1$. Therefore to have $Re = 100$, the kinematic viscosity is set to $\nu = 0.001$.

blockMesh

The mesh is already present in 'constant' folder and is defined in the file 'constant/polyMesh/blockMesh'. The no-slip condition on the cylinder surface results in the formation of a boundary layer adjacent to the cylinder where the flow varies rapidly from zero to the freestream (in the radial direction). As the Reynolds number in the boundary layer must be $O(1)$, an estimate of the thickness δ can be made:

$$\delta \sim DRe^{-1/2}.$$

As a rule of thumb it is advised to have ten elements across the boundary layer to resolve it and so, the above estimate is helpful in determining the smallest element size. This rapid variation is only in the radial direction, the variation in the azimuthal direction is more gradual and so the finer resolution is only required in the radial direction.

Note in the 'blockMesh' file that the cylinder is defined as 'type wall'. This ensures that it is possible to use the 'wallGradU' utility later on.

1.3.3. *system*

controlDict

The standard aspects of the file 'controlDict' have already been covered in a previous tutorial. The following code takes out the force on the body (denoted by cylinder in the blockMesh file). The 'outputInterval' determines how often the force is written to file and can be varied if long simulations are being carried out. In setting our Reynolds number the kinematic viscosity was set. Here we see that one needs to define the density ρ_∞, which is set to unity such that $\mu = \nu$. The 'CoR' is the centre of rotation and needs to be set appropriately for calculating moments on the cylinder.

```
forces_cylinder_1
{
type forces;
 functionObjectLibs ("libforces.so");
 outputControl    timeStep;
 outputInterval 1;
patches (cylinder);
  rhoName rhoInf;
pName p;
 UName U;
  rhoInf 1;
CofR (0 0 0);  ;
}
```

Forces are collated in the files in 'postProcessing/forces$_cylinder_1$/0/$forces.dat'.The format of the output is the follo$

$$Time \quad \underbrace{F_{px} \quad F_{py} \quad F_{pz}}_{Pressure} \quad \underbrace{F_{\nu x} \quad F_{\nu y} \quad F_{\nu z}}_{Viscous} \quad \underbrace{F_{porx} \quad F_{pory} \quad F_{porz}}_{Porous}$$

$$\underbrace{M_{px} \quad M_{py} \quad M_{pz}}_{Pressure} \quad \underbrace{M_{\nu x} \quad M_{\nu y} \quad M_{\nu z}}_{Viscous} \quad \underbrace{M_{porx} \quad M_{pory} \quad M_{porz}}_{Porous}$$

The following code takes out the velocity and pressure at all the locations under 'probeLocations' every tenth time step. Care must be taken here as if many points are chosen and long simulations are run, the data that is collected can be massive.

```
probes1
{

type probes;

functionObjectLibs ("libsampling.so");

//dictionary probesDict;

writeInterval 10;

region region0;

probeLocations
```

```
6
(
(1.5 -1 0)
...
(1.5 1 0)
);

fields
(
p
U
);

}
```

It is also possible to use the utility probes as a postprocessing tool by running the command 'postProcess -func probes'.

sampleDict

The 'sample' utility (called by printing 'sample' in the command line in the main directory) allows one to extract data from the time files. Two different types of data can be extracted - sets and surfaces. In this tutorial only the latter will be covered. Below is an extract from the file 'system/sampleDict', which indicates that the pressure should be taken out at the cylinder surface.

```
surfaces
(
   cylinder
   {
type patch;
patches (cylinder);
   }
);


  fields
    (
     p
    );
```

The files are output in the folder 'postProcessing/surfaces/TIMES'. The output from this file

```
# p  FACE_DATA 160
#  x  y  z  p
0.0346477 0.0360357 0 0
   ...
```

where the values printed are described on the second line. In this case it is the Cartesian coordinate of the cell centre (x, y, z) and the pressure at that point.

'wallGradU'

The wallGradU utility (called by printing 'wallGradU' in the command line in the main directory) can be used to calculate velocity gradient normal to the surface of the cylinder. This will only be calculated on surfaces which were defined as 'type wall' in the 'blockMesh' file. The files are output in the main directory for that time (i.e. '0' etc.). A section of the output for this

```
 outlet
    {
        type            calculated;
        value           uniform (0 0 0);
    }

cylinder
    {
        type            calculated;
        value           nonuniform List<vector>
160
(
(-8.82272 8.60517 -2.908e-16)
...
```

In this case, there are 160 values of the gradient of the velocity which corresponds to the number of cells on the cylinder surface. The three values shown are:

$$\frac{\partial u}{\partial n}, \quad \frac{\partial v}{\partial n}, \quad \frac{\partial w}{\partial n}.$$

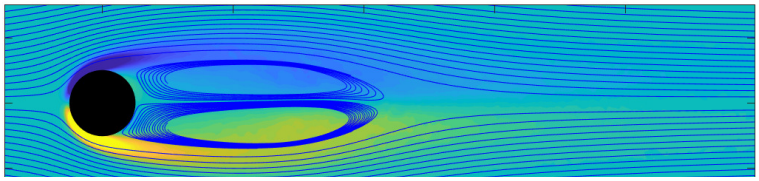
1.4. *Tutorial steps*

The impulsively started flow past a cylinder at $Re = 100$ results in a wake bubble forming behind the cylinder, which can be seen in figure 3. This will eventually go unstable and then the flow will proceed into a quasi-steady regime of von Karman vortex shedding, shown in figure 3.

As the simulation takes some time to run, there are two time files in the directory, namely '0' and '100'. The simulation can therefore be started at 0, to capture the transient and the quasi-steady state or can be started after $\tau = 100$ to only analyse the periodic vortex shedding. This can be controlled in the file '/system/controlDict' through the parameter 'startTime' and stopAt'. For this tutorial we will only consider the periodic regime and so the 'startTime' and 'stopAt' are set to 100 and 105 respectively. The choice of end time has been determined by looking at the forces on the cylinder which shows that a time period of 5 is approximately 15 periods which is sufficiently long for velocity averaging. Note: for turbulent flows one might require much longer time series for averaging. The commands to run the code are:

```
icoFoam
wallGradU
sample
```

These commands show that we can differentiate between runtime diagnostics and post-processing diagnostics. In the current case, the runtime diagnostics are (i) the forces on the cylinder and (ii) the velocity probes, whereas postprocessing diagnostics are cylinder



(a)

(b)

Figure 3: Vorticity field at $\tau = 15$ showing a symmetrical recirculation region (highlighted by the streamlines) behind the cylinder which results in zero lift. This will eventually go unstable $\tau \approx 50$, which result in a von Karman vortex street behind the cylinder, which can be seen in (b). Vorticity is shown in (a) and (b), ranging from -0.2 (blue) to zero (green) to 0.2 (yellow).

surface pressure and velocity gradient measurements. To plot out the diagnostics, Matlab is used and the code has been provided in the folder 'Analysis' to plot out the data. Forces files are in in the folder 'postProcessing/forces$_c ylinder$1/0/'. Once you have run the code from '100' to '105', there will be a new folder in here, '100', which will have the force data for that run. Copy/paste the 'forces.dat' file into your 'Analysis/Forces' folder and run the following command in Matlab

```
plot_forces
```

The file 'wallGradU' can be found in the time folders in the main directory. Copy/paste this into the folder 'Analysis/Cf'. Additionally you will need to copy/paste the file 'postProcessing/surfaces/0/pcylinder.raw' into 'Analysis/Cf' and run the following command in Matlab

```
c_f_plot
```

Copy/paste the file 'postProcessing/surfaces/0/pcylinder.raw' into 'Analysis/Cp' and run the following command in Matlab

```
c_p_plot
```

Sample outputs for the time series of the drag and lift coefficient are shown in figure 4 and additionally the pressure and friction coefficient are shown in figure 5.

Note: as the probes generate a lot of data, this functionality should be turned off if the simulation is started from the beginning.

Things to consider in the future:

• Mesh independence study - try varying the size of the mesh in the wake to fully resolve the shed vortices from the cylinder. Does this affect the drag and lift coefficients?

• Calculate the mean, rms and Strouhal number of the drag and lift coefficients

References

Batchelor, G.K. (1957) An introduction to fluid dynamics. *Cambridge University Press.*

Dimopoulous, H.G. & Hanratty, T.J. (1968) Velocity gradients at the wall for flow around a cylinder for Reynolds number between 60 and 360. *J. Fluid Mech.* **33** 303-319.

Homann, F. (1936) Influence of higher viscosity on flow around cylinder. *Forsch. Gebiete Ingenieur.* **17**

Rajani, B.N., Kandasamy, A. & Majumdar, S. (2009) Numerical simulation of laminar flow past a circular cylinder. *App. Math. Mod.* **33** 1228-1247.

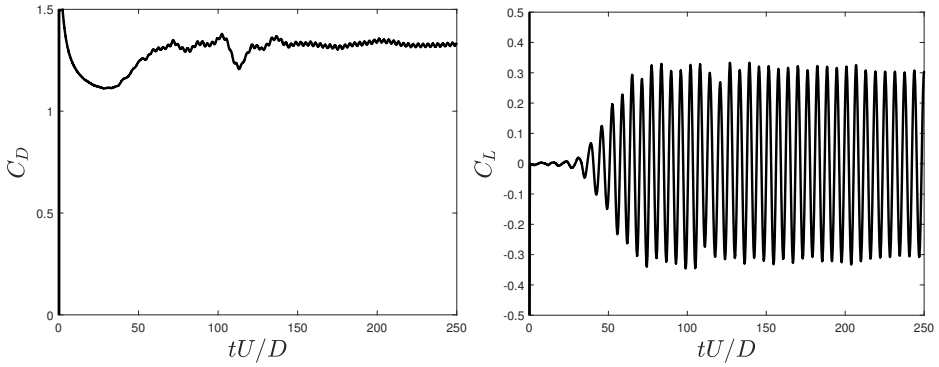


Figure 4: Time series of the (a) drag and (b) lift coefficient.

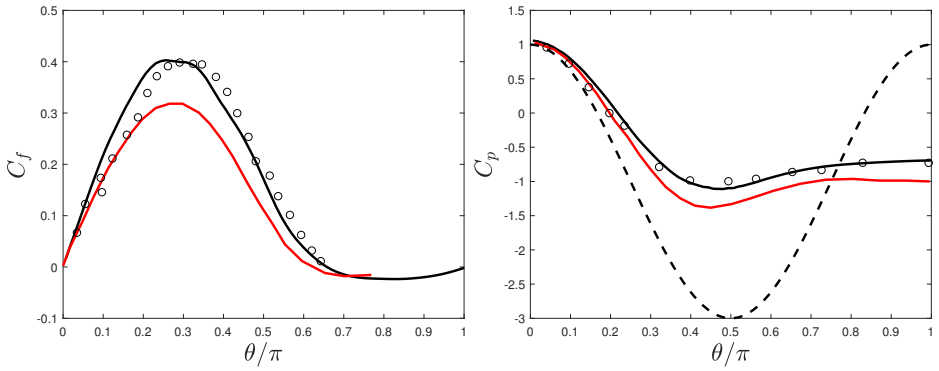


Figure 5: Surface profiles of the mean (a) friction and (b) pressure coefficient. In (a) the present simulations (black line), the experiments by Dimopoulous & Hanratty (1968) ($\circ$) (Re = 104) and the numerical simulations of Rajani *et al.* (2009) (red line) are shown. In (b) the present simulations (black line), the experiments by Homann (1936) ($\circ$) (Re = 107), the numerical simulations of Rajani *et al.* (2009) (red line) and the analytical solution for a potential flow (dashed line) are shown. The front stagnation point is at zero radians.

Williamson, C.H.K. (1996) Vortex dynamics in the cylinder wakes. *Ann. Rev. Fluid Mech.* **2**8 477-539.

Zdravkovich, M. (1997) Flow around circular cylinders. *Oxford Science Publication.*

Appendix D

Tutorial: Develop a turbulent model

Chapter 1

Introduction

The main aim of this tutorial is to describe how to implement a new Two-equation Sub-grid Scale (SGS) model. The new model is claimed by its developer to be self-adaptive and be able to model turbulence on any arbitrary mesh density, i.e. from Reynolds-averaged Navier-Stokes (RANS) to Large Eddy Simulation (LES) and even to Direct Numerical Simulation (DNS) [3]. The tutorial also demonstrates how to implement wall damping function as a warm-up exercise.

The main content is listed below:

1. This tutorial starts with a theoretical background of turbulence modelling, including different methods of turbulence modelling, the commonly used eddy viscosity model concept and the Two-equations SGS model being implemented. A tour of the templated turbulence model library in OpenFOAM 1706 will also be given with focus on most related existing models in order for the reader to familiarise themselves with these codes.

2. The Chapter 3 will be dedicated to show how exactly the damping function and the new model are being implemented by a step by step guide.

3. The Chapter 4 will prepare the pitzDaily test case to show the result of this implemented wall damping function and the new model by using ParaView.

4. The tutorial will end with conclusion and further development.

Chapter 2

Background

2.1 Turbulence Modelling

Computational Fluid Dynamics (CFD) has a indispensable role in engineering application and research activities. It can simulate real flow conditions, heat transfers and other phenomena with a reasonable accuracy but much lower cost than conducting an experiment. However CFD still has many bottlenecks, one among which is its capability to model turbulent flow, the most common flow type in the real engineering situation.

Turbulence, the three-dimensional, random and complex state of a fluid with wide range of length scales, is one of the most challenging problems in fluid dynamics, yet having great significance in practical engineering applications. Consequently, numerous scientists have invested a great deal of effort in the observation, description and understanding of turbulent flows. It was found out that by applying the conservation of mass, momentum and energy, governing equation of fluid flow can be derived. If using Newtonian model for viscous stresses, the governing equation will lead to Navier-Stokes Equations (NSEs) [4]. If NSEs is solved on a spatial grid that is fine enough to solve the Kolmogorov length scale with time step sizes that sufficiently small to resolve the fastest fluctuation, all flow characteristics can be captured, including mean flow and turbulence. This method is known as Direct Numerical Simulation (DNS). But the computational cost of DNS is prohibitably high and is not used in real engineering applications.

Due to the limitation of computing resources, the attention of early CFD research was on the mean flow and modeling the effect of turbulence on the mean flow. This lead to a method called Reynolds-averaged Navier-Stokes (RANS). This method conducts a time or ensemble average on NSEs and the extra term created due the averaging process is modelled by so-called RANS turbulence models. RANS has been widely applied in industrial flow computations due to its modest computing resources requirement and reasonable accuracy. However, it is widely recognised that RANS fails to provide satisfactory accuracy in flow with separation, reattachment and noise, etc. It is mainly due to RANS represents all turbulent energy by modelling, whereas turbulence plays an dominant role in such flow conditions. Here comes the Large Eddy Simulation (LES), it can offer a solution to balance the high computational cost of DNS and the low accuracy of RANS. LES spatially filters the NSEs and directly resolves the governing equations for large eddies (larger than filter size[1]) and their turbulent energy, while it uses a Sub-grid Scale (SGS) model to simulate the effect of small eddies (smaller than filter size) on the mean flow and large eddies. Computing resource requirement of LES is greater than RANS but is much less than DNS. Thanks to the rapid increase of computing power, LES has started to be applied on complex geometries.

Recently, hybrid turbulence models which combine RANS and LES characteristics in various way have attracted lots of research attentions. Detached Eddy Simulation (DES) is one of the examples. DES uses RANS formula to solve the flow field close to the wall whereas use LES to solve large (detached) eddies away from the wall. Some other hybrid models solve eddy viscosity

[1]For implicit LES, filter size is mesh size.

by both RANS and LES and blend them according to certain parameters, normally being mesh size. Scale-adaptive Simulation (SAS) model, such as k-ω SST SAS model invited by Menter and co-workers [1], adds an extra production term in the ω equation which will increase the production of ω when detecting the unsteadiness, thereby decrease the turbulence viscosity in order to provide RANS with LES content in unsteady regions without any blend factor.

The Two-equation SGS model to be implemented in the tutorial is also claimed to be self adapting. But unlike k-ω SST SAS model which is based on RANS model and resolve more turbulence if mesh resolution allows to do so, this model is based on SGS model and can resolve as much energy as possible for any mesh resolution, i.e. it changes characters across RANS, LES and even DNS depending on flow situation and mesh density [3].

2.2 Eddy Viscosity Model

By far, most of the turbulence models implemented in OpenFOAM are based on eddy viscosity theory, such as Spalart-Allmaras, k-ϵ series and k-ω series RANS models, and Smagorinsky and k-equation SGS model for LES, including the one being implemented. Therefore, it is crucial to understand eddy viscosity theory before starting any implementation or modification of this kind of turbulence models.

By applying a spatial filter of uniform width, Δ, on incompressible, Newtonian flows with constant thermo-physical properties, one can obtain the governing equations for the Large Eddy Simulation of such a flow as

$$\nabla \cdot \bar{\boldsymbol{U}} = 0 \tag{2.1a}$$

$$\frac{\partial \bar{\boldsymbol{U}}}{\partial t} + \nabla \cdot (\bar{\boldsymbol{U}}\bar{\boldsymbol{U}}) = -\nabla \bar{p} + \nabla \cdot \nu(\nabla \bar{\boldsymbol{U}} + \nabla \bar{\boldsymbol{U}}^T) - \nabla \cdot \tau \tag{2.1b}$$

Where overbar indicates spatial filtering process, $\bar{U}$ is filtered instantaneous velocity (time-averaged velocity in RANS), $\overline{p}$ is filtered instantaneous pressure (time-averaged velocity in RANS) divided by the constant density, ν is kinematic viscosity, τ is the SGS stress tensor (Reynolds stress in RANS) which has to be modelled to close the system.

It is well known that in the Newton's law of viscosity for incompressible flow

$$\tau_{Newtonian} = 2\nu S = \nu(\nabla \boldsymbol{U} + \nabla \boldsymbol{U}^T) \tag{2.2}$$

where S is the rate of deformation of fluid elements. It has been found that the turbulent stresses increase as the mean rate of deformation increases. Boussinesq hypothesis proposed that the Reynolds stress in RANS is proportional to the mean rates of deformation. In SGS model, this theory is interpreted as SGS stresses are proportional to the instantaneous rates of deformation, i.e.

$$\tau = -2\nu_t \overline{S} + \frac{1}{3} tr(\tau)\boldsymbol{I} = -\nu_t(\nabla \overline{\boldsymbol{U}} + \nabla \overline{\boldsymbol{U}}^T) + \frac{1}{3} tr(\tau)\boldsymbol{I} \tag{2.3}$$

where ν_t is the SGS eddy viscosity (or turbulence viscosity in RANS) and I is Kronecker Delta.

On dimensional grounds, it is assumable that ν_t can be expressed as a product of a SGS velocity scale, ϑ, and a SGS length scale, ℓ, as

$$\nu_t = C\vartheta\ell \tag{2.4}$$

where C is a dimensionless constant.

Therefore, the turbulence model based on Eddy Viscosity theory is to find appropriate equations for ϑ and ℓ by either algebraic relation or transport equations, and to use them to obtain ν_t in order to close the filtered NSEs.

2.3 Inagi Wall Damping Function

The wall damping function which will be used as a warm-up exercise is proposed by Inagi et al. [2]. It is only applicable to SGS models that contain k-equation. As can be seen in later section that the standard k-equation SGS model evaluates kinematic eddy viscosity via

$$\nu_t = C_k \Delta \sqrt{k_{sgs}} \tag{2.5}$$

while wall damping function introduce a parameter F_{wY} into the equation above, i.e.

$$\nu_t = F_{wY} C_k \Delta \sqrt{k_{sgs}} \tag{2.6}$$

where

$$F_{wY} = \frac{1}{1 + \Delta \sqrt{2|S_{ij}|^2}/C_T \sqrt{k_{sgs}}} \tag{2.7}$$

and $C_T = 10.0$.

2.4 Two-equation SGS model

The new Two-equation SGS model proposed by Perot and Gadebusch [3] reads

$$\frac{\partial k_{sgs}}{\partial t} + \nabla \cdot (k_{sgs} \bar{\boldsymbol{U}}) = \nabla \cdot \left[\left(\nu + \frac{\nu_t}{\sigma_k} \right) \nabla k_{sgs} \right] + \alpha P - \varepsilon_{sgs} \tag{2.8a}$$

$$\frac{\partial \varepsilon_{sgs}}{\partial t} + \nabla \cdot (\varepsilon_{sgs} \bar{\boldsymbol{U}}) = \nabla \cdot \left[\left(\nu + \frac{\nu_t}{\sigma_\varepsilon} \right) \nabla \varepsilon_{sgs} \right] + \frac{\varepsilon_{sgs}}{k_{sgs}} \left[C_{\varepsilon 1} P - C_{\varepsilon 2} \varepsilon_{sgs} \right] \tag{2.8b}$$

where k_{sgs} is the SGS turbulent kinetic energy, ε_{sgs} is the SGS turbulent kinetic energy dissipation rate, $P = \nu_t (\nabla \overline{\boldsymbol{U}} + \nabla \overline{\boldsymbol{U}}^T) \nabla \overline{\boldsymbol{U}}$ is the production of SGS turbulent kinetic energy. The parameter $C_{\varepsilon 2} = (11/6) f + (25/Re_T) f^2$, where $Re_T = k_{sgs}^2 / \nu \varepsilon_{sgs}$ is the local turbulent Reynolds number and function $f = (Re_T/30[\sqrt{1 + 60/Re_T} - 1]$. The constants in this model are

$$C_{\varepsilon 1} = 1.55; \quad \sigma_k = 1.0; \quad \sigma_\varepsilon = 1.2 \tag{2.9}$$

The energy transfer (backscatter) variable, α, is

$$\alpha = 1.5 \left\{ 1 - C^* \left(\frac{k_{sgs}}{k_{sgs} + k_r} \right)^2 \left[\left(\frac{\Delta x_i}{\sqrt{k_r}} \frac{\partial \sqrt{k_r}}{\partial x_i} \right)^2 + 0.11 \right]^{-1} \right\} \tag{2.10}$$

Where the empirically determined constant $C^* = 0.28$, and

$$\frac{\left(\Delta x_i \frac{\partial \sqrt{k_r}}{\partial x_i} \right)^2}{k_r} = \frac{\left\{ \left(\Delta x \frac{\partial \sqrt{k_r}}{\partial x} \right)^2 + \left(\Delta y \frac{\partial \sqrt{k_r}}{\partial y} \right)^2 + \left(\Delta z \frac{\partial \sqrt{k_r}}{\partial z} \right)^2 \right\}}{k_r} \tag{2.11}$$

is a dimensionless measure of the gradient of resolved turbulent kinetic energy.

The eddy viscosity is then evaluated as

$$\nu_t = C_\mu \frac{k_{sgs}^2}{\varepsilon_{sgs}} \left(\frac{k_{sgs}}{k_{sgs} + k_r} \right) \tag{2.12}$$

where $C_\mu = 0.18$ and $k_r = 0.5 \times U'^2$ is the resolved turbulent kinetic energy.
Finally, the SGS stresses are constructed as

$$\tau = -\alpha \nu_t (\nabla \overline{\boldsymbol{U}} + \nabla \overline{\boldsymbol{U}}^T) + \frac{2}{3} k_{sgs} \boldsymbol{I} \tag{2.13}$$

2.5 Turbulence Model Library

Turbulence model library in OpenFOAM 1706 is a templated library, located at `$FOAM_SRC/TurbulenceModels`. The templated Turbulence model class contains many sub-classes. How is the specific turbulence model selected during the run time is well documented in the lecture of Prof. Nilsson [2]. This section will take reader to a tour from the perspective of specific turbulence model.

A good practice of implementing new piece of code in OpenFOAM is to firstly find class with a similar function and then modify the functionality based on it. The model being implemented is a SGS model but belongs to k-ε series. A detailed look at the available SGS model at `$FOAM_SRC/TurbulenceModels/turbulenceModels/LES` shows the there is no k-ε based SGS model, the directory contains only the k-equation SGS model. Whereas k-ε series model only appears in `$FOAM_SRC/TurbulenceModels/turbulenceModels/RAS` which contains RANS models. Therefore, current model will be a combination of k-equation SGS model and k-ε RANS model.

2.5.1 The Standard k-ε Model in OpenFOAM 1706

The standard k-ε RANS model implemented in OpenFOAM 1706 reads

$$\frac{\partial k}{\partial t} + \nabla \cdot (k\bar{\boldsymbol{U}}) = \nabla \cdot \left[\left(\nu + \frac{\nu_t}{\sigma_k} \right) \nabla k \right] + P - \varepsilon \tag{2.14a}$$

$$\frac{\partial \varepsilon}{\partial t} + \nabla \cdot (\varepsilon\bar{\boldsymbol{U}}) = \nabla \cdot \left[\left(\nu + \frac{\nu_t}{\sigma_\varepsilon} \right) \nabla \varepsilon \right] + \frac{\varepsilon}{k} \left[C_{\varepsilon 1} P - C_{\varepsilon 2} \varepsilon \right] \tag{2.14b}$$

$$\nu_t = C_\mu \frac{\sqrt{k}}{\varepsilon} \tag{2.14c}$$

where k is the turbulent kinetic energy, ε is its dissipation rate. The constants of this model are

$$C_\mu = 0.09; \quad C_{\varepsilon 1} = 1.44; \quad C_{\varepsilon 2} = 1.92; \quad \sigma_k = 1.0; \quad \sigma_\varepsilon = 1.3 \tag{2.15}$$

This model is implemented in the directory `$FOAM_SRC/TurbulenceModels/turbulenceModels/RAS/kEpsilon` by two files, i.e. `kEpsilon.C` and `kEpsilon.H`. The H file declares the class, any member data and functions, and the C file contains the detailed implementation of the model as (line 254 to line 295)

```
// Dissipation equation
tmp<fvScalarMatrix> epsEqn
(
    fvm::ddt(alpha, rho, epsilon_)
  + fvm::div(alphaRhoPhi, epsilon_)
  - fvm::laplacian(alpha*rho*DepsilonEff(), epsilon_)
 ==
    C1_*alpha()*rho()*G*epsilon_()/k_()
  - fvm::SuSp(((2.0/3.0)*C1_ - C3_)*alpha()*rho()*divU, epsilon_)
  - fvm::Sp(C2_*alpha()*rho()*epsilon_()/k_(), epsilon_)
  + epsilonSource()
  + fvOptions(alpha, rho, epsilon_)
);

epsEqn.ref().relax();
fvOptions.constrain(epsEqn.ref());
epsEqn.ref().boundaryManipulate(epsilon_.boundaryFieldRef());
solve(epsEqn);
```

[2] Link: https://pingpong.chalmers.se/public/courseId/8331/lang-en/publicPage.do?item=3855255

```
fvOptions.correct(epsilon_);
bound(epsilon_, this->epsilonMin_);

// Turbulent kinetic energy equation
tmp<fvScalarMatrix> kEqn
(
    fvm::ddt(alpha, rho, k_)
  + fvm::div(alphaRhoPhi, k_)
  - fvm::laplacian(alpha*rho*DkEff(), k_)
 ==
    alpha()*rho()*G
  - fvm::SuSp((2.0/3.0)*alpha()*rho()*divU, k_)
  - fvm::Sp(alpha()*rho()*epsilon_()/k_(), k_)
  + kSource()
  + fvOptions(alpha, rho, k_)
);

kEqn.ref().relax();
fvOptions.constrain(kEqn.ref());
solve(kEqn);
fvOptions.correct(k_);
bound(k_, this->kMin_);

correctNut();
```

And the effective diffusivity for k and ε are calculated in H file as (line 161 to line 185)

```
    //- Return the effective diffusivity for k
    tmp<volScalarField> DkEff() const
    {
        return tmp<volScalarField>
        (
            new volScalarField
            (
                "DkEff",
                (this->nut_/sigmak_ + this->nu())
            )
        );
    }

    //- Return the effective diffusivity for epsilon
    tmp<volScalarField> DepsilonEff() const
    {
        return tmp<volScalarField>
        (
            new volScalarField
            (
                "DepsilonEff",
                (this->nut_/sigmaEps_ + this->nu())
            )
        );
    }
```

Then the turbulent kinematic viscosity, ν_t, is evaluated in C file as

```
template<class BasicTurbulenceModel>
void kEpsilon<BasicTurbulenceModel>::correctNut()
{
    this->nut_ = Cmu_*sqr(k_)/epsilon_;
    this->nut_.correctBoundaryConditions();
    fv::options::New(this->mesh_).correct(this->nut_);

    BasicTurbulenceModel::correctNut();
}
```

2.5.2 The Standard k-equation SGS Model in OpenFOAM 1706

The standard k-equation SGS model implemented in OpenFOAM 1706 reads

$$\frac{\partial k_{sgs}}{\partial t} + \nabla \cdot (k_{sgs}\bar{\boldsymbol{U}}) = \nabla \cdot \left[\left(\frac{\nu + \nu_t}{\sigma_k} \right) \nabla k_{sgs} \right] + P - \varepsilon_{sgs} \tag{2.16a}$$

$$\nu_t = C_k \Delta \sqrt{k_{sgs}} \tag{2.16b}$$

where k is the SGS turbulent kinetic energy, $\varepsilon_{sgs} = C_e k^{3/2}/\Delta$ is its dissipation rate. The constant of this model are

$$C_k = 0.094; \quad C_\varepsilon = 1.048; \quad \sigma_k = 1.0; \tag{2.17}$$

This model is implemented in the directory
`$FOAM_SRC/TurbulenceModels/turbulenceModels/LES/kEqn` by two files,
i.e. `kEqn.C` and `kEqn.H`. The H file declares the class, any member data and functions, the C file contains the detailed implementation of the model as (line 186 to line 205)

```
tmp<fvScalarMatrix> kEqn
(
    fvm::ddt(alpha, rho, k_)
  + fvm::div(alphaRhoPhi, k_)
  - fvm::laplacian(alpha*rho*DkEff(), k_)
 ==
    alpha*rho*G
  - fvm::SuSp((2.0/3.0)*alpha*rho*divU, k_)
  - fvm::Sp(this->Ce_*alpha*rho*sqrt(k_)/this->delta(), k_)
  + kSource()
  + fvOptions(alpha, rho, k_)
);

kEqn.ref().relax();
fvOptions.constrain(kEqn.ref());
solve(kEqn);
fvOptions.correct(k_);
bound(k_, this->kMin_);

correctNut();
```

And the effective diffusivity for k_{sgs} is calculated in H file as (line 152 to line 159)

```
//- Return the effective diffusivity for k
tmp<volScalarField> DkEff() const
{
    return tmp<volScalarField>
```

```
            (
                new volScalarField("DkEff", this->nut_ + this->nu())
            );
        }
```

Then the turbulent kinematic viscosity, ν_t, is evaluated in C file as

```
template<class BasicTurbulenceModel>
void kEqn<BasicTurbulenceModel>::correctNut()
{
    this->nut_ = Ck_*sqrt(k_)*this->delta();
    this->nut_.correctBoundaryConditions();
    fv::options::New(this->mesh_).correct(this->nut_);

    BasicTurbulenceModel::correctNut();
}
```

It is observed that the k equations for k-equation SGS and k-ε RANS model are almost the same, except for the evaluation of ε. Therefore, although RANS and LES have fundamentally different mathematical theory, the unclosed governing equations solved on a computer are identical![3] This, theoretically, indicates there could be an universal model which can solve turbulent energy over whole spectrum, i.e. across RANS, LES and DNS.

[3]subject to the spatial filtering operation is implicit for LES.

Chapter 3

Model Implementation

3.1 Warm-up Exercise

The implementation of Inagi wall damping function will be treated as a warm-up exercise before implementing the new SGS model.

As the Turbulence model classes are templated in OpenFOAM 1706, one has to copy the entire Turbulence model directory into the user directory rather than only coping certain existing models as for previous version, such as OpenFOAM 2.3.0. So some preparatory work is needed as below:

```
of+ // reader may use OF1706+
mkdir -p $FOAM_RUN // make sure having user and run directory
foam
cp -r --parents src/TurbulenceModels $WM_PROJECT_USER_DIR
cd $WM_PROJECT_USER_DIR/src/TurbulenceModels
```

Find all the Make directories, change the location of compiled files to relevant user directory, and compile:

```
find . -name Make
sed -i s/FOAM_LIBBIN/FOAM_USER_LIBBIN/g ./*/Make/files
./Allwmake
```

Make sure the following three new shared-object files in
`$WM_PROJECT_USER_DIR/platforms/linux64GccDPInt32Opt/lib`:

```
libcompressibleTurbulenceModels.so
libincompressibleTurbulenceModels.so
libturbulenceModels.so
```

Note: The 3 files may also be in `linux64GccDPInt64Opt`, depends on how was it built.
Copy the kEqn SGS model and change the class name to kEqnInagi and conduct all necessary process to compile:

```
cd turbulenceModels/LES
cp -r kEqn kEqnInagi
cd kEqnInagi
mv kEqn.H kEqnInagi.H
mv kEqn.C kEqnInagi.C
sed -i 's/kEqn/kEqnInagi/g *
sed -i 's/"OpenFoam Foundation"/"Your Name"/g *
```

Then open `turbulentTransportModels.C`:

```
cd $WM_PROJECT_USER_DIR/src/TurbulenceModels
vi incompressible/turbulentTransportModels/turbulentTransportModels.C
```

add the following lines:

```
#include "kEqnInagi.H"
makeLESModel(kEqnInagi);
```

under the lines for kEqn model:

```
#include "kEqn.H"
makeLESModel(kEqn);
```

save and close the file, then update lnInclude directory and compile:

```
wmakeLnInclude -u ../../../turbulenceModels
./../../../Allwmake
```

Then open `kEqnInagi.C` by:

```
vi turbulenceModels/LES/kEqnInagi/kEqnInagi.C
```

and replace

```
    this->nut_ = Ck_*sqrt(k_)*this->delta();
```

with

```
    dimensionedScalar verySmall
    (
            "verySmall",
            dimensionSet (0, 1, -1, 0, 0, 0, 0),
            VSMALL
    );
    this->nut_ = 10*Ck_*k_*this->delta()/
    (10.0*sqrt(k_) + this->delta()*sqrt(2*magSqr(symm(fvc::grad(this->U_)))) + verySmall);
```

Then save and close the file, update `turbulentTransportModels.C` and recompile:

```
touch incompressible/turbulentTransportModels/turbulentTransportModels.C
./../../../Allwmake
```

So far the Inagi wall damping function has been implemented, its preliminary result will be shown together with the new SGS model in later Chapter.

3.2 Preparatory Work

Same as the warm-up exercise, some preparatory work is needed before implementing the model: Copy the kEqn SGS model and change the class name to kEpsilonSAS and conduct all necessary process to compile:

```
cd $WM_PROJECT_USER_DIR/src/TurbulenceModels
cd turbulenceModels/LES
cp -r kEqn kEpsilonSAS
cd kEpsilonSAS
mv kEqn.H kEpsilonSAS.H
mv kEqn.C kEpsilonSAS.C
sed -i 's/kEqn/kEpsilonSAS/g *
sed -i 's/"OpenFoam Foundation"/"Your Name"/g *
```

Then open `turbulentTransportModels.C` via:

```
cd $WM_PROJECT_USER_DIR/src/TurbulenceModels
vi incompressible/turbulentTransportModels/turbulentTransportModels.C
```

add following lines:

```
#include "kEpsilonSAS.H"
makeLESModel(kEpsilonSAS);
```

under lines for kEqn model:

```
#include "kEqn.H"
makeLESModel(kEqn);
```

save and close the file, then update lnInclude directory and compile:

```
wmakeLnInclude ../../../turbulenceModels
./../../../Allwmake
```

Following the preparatory work, the new k-ε SGS model can now be implemented by adding the functionality from k-ε RANS model to the existing k-equation SGS model with extra modifications.

It is a good practice to modify the code by adopting the step-by-step approach, starting from constants and existing functionality, then gradually reach the complicated functions. Every major step will be tested on cases before moving to next step. It is the methodology adopted when implementing the new model. However, it will be lengthy if all the steps are shown here. Therefore only the differences between kEqn SGS model and the final implementation of k-ε SGS model are provided.

3.3 Detailed Implementation

3.3.1 H file

Under `// Fields` list:

add

```
volScalarField epsilon_;
volVectorField UMean_;
volScalarField kR_;
volScalarField dimlessGradkR_;
volScalarField alfa_;
volScalarField nutByAlfa_;
```

under

```
 volScalarField k_;
```

Under `// Model constants` list:

replace

```
dimensionedScalar Ck_;
```

with

```
dimensionedScalar Cnu_;
dimensionedScalar Ce1_;
dimensionedScalar sigmaK_;
dimensionedScalar sigmaEps_;
```

Under `// Protected Member Functions` list:

add

```
virtual tmp<fvScalarMatrix> epsilonSource() const;
```

under

```
virtual tmp<fvScalarMatrix> kSource() const;
```

Under `// Member Functions` list:

replace

```
//- Return sub-grid dissipation rate
virtual tmp<volScalarField> epsilon() const;

//- Return the effective diffusivity for k
tmp<volScalarField> DkEff() const
{
    return tmp<volScalarField>
    (
        new volScalarField("DkEff", this->nut_ + this->nu())
    );
}
```

with

```
//- Return sub-grid dissipation rate
virtual tmp<volScalarField> epsilon() const
{
    return epsilon_; // added
}

//- Return the effective diffusivity for k
tmp<volScalarField> DkEff() const
{
    return tmp<volScalarField>
    (
        new volScalarField("DkEff", (nutByAlfa_ + this->nu())/sigmaK_)
    );
}

//- Return the effective diffusivity for epsilon
tmp<volScalarField> DepsilonEff() const
{
    return tmp<volScalarField>
    (
        new volScalarField("DepsilonEff", (this->nutByAlfa_ +
        this->nu())/sigmaEps_)
    );
}
```

3.3.2 C file

Under `// Protected Member Functions` list:

within the function of

```
template<class BasicTurbulenceModel>
void kEpsilonSAS<BasicTurbulenceModel>::correctNut()
```

replace

```
    this->nut_ = Ck_*sqrt(k_)*this->delta();
```

with

```
    kR_ = 0.5*magSqr(UMean_ - this->U_);
    volVectorField gradSqrtkR = fvc::grad(sqrt(kR_));

    surfaceVectorField fV = this->mesh_.Sf();
    volScalarField surfaceSumX = 0.5*fvc::surfaceSum(mag(fV.component(0)));
    volScalarField surfaceSumY = 0.5*fvc::surfaceSum(mag(fV.component(1)));
    volScalarField surfaceSumZ = 0.5*fvc::surfaceSum(mag(fV.component(2)));

    volScalarField cv
    (
            IOobject
            (
                    "cv",
                    this->runTime_.timeName(),
                    this->mesh_,
                    IOobject::NO_READ,
                    IOobject::AUTO_WRITE
            ),
            this->mesh_,
            dimensionedScalar("zero",dimVolume,0.0)
    );

    cv.ref() = this->mesh_.V();

    dimensionedScalar surfaceMin
    (
            "surfaceMin",
            dimensionSet (0, 2, 0, 0, 0, 0, 0),
            VSMALL
    );

    dimlessGradkR_ =
    (
     sqr(cv / (surfaceSumX + surfaceMin) * gradSqrtkR.component(0)) +
     sqr(cv / (surfaceSumY + surfaceMin) * gradSqrtkR.component(1)) +
     sqr(cv / (surfaceSumZ + surfaceMin) * gradSqrtkR.component(2))
    )
    /
    (kR_ + this->kMin_);

    alfa_ =
    1.5*
       (
         1.0 -
         0.28 * sqr(k_ / (k_ + kR_ + this->kMin_)) /
         (dimlessGradkR_ + 0.11)
       );
```

```
    // Calculate sgs nut
    nutByAlfa_ = Cnu_*sqr(k_)/(epsilon_+this->epsilonMin_)*
    (k_/(k_+kR_+this->kMin_));
    this->nut_ = alfa_*nutByAlfa_;
```

also add

```
template<class BasicTurbulenceModel>
tmp<fvScalarMatrix> kEpsilonSAS<BasicTurbulenceModel>::epsilonSource() const
{
    return tmp<fvScalarMatrix>
    (
        new fvScalarMatrix
        (
            epsilon_,
            dimVolume*this->rho_.dimensions()*epsilon_.dimensions()
            /dimTime
        )
    );
}
```

under

```
template<class BasicTurbulenceModel>
tmp<fvScalarMatrix> kEqn<BasicTurbulenceModel>::kSource() const
{
    return tmp<fvScalarMatrix>
    (
        new fvScalarMatrix
        (
            k_,
            dimVolume*this->rho_.dimensions()*k_.dimensions()
            /dimTime
        )
    );
}
```

Within `Constructors`:

replace

```
    Ck_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "Ck",
            this->coeffDict_,
            0.094
        )
    )
```

with

```
    epsilon_
    (
        IOobject
```

```
        (
            IOobject::groupName("epsilon", this->U_.group()),
            this->runTime_.timeName(),
            this->mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        this->mesh_
    ),

    UMean_
    (
        IOobject
        (
            IOobject::groupName("UMean", this->U_.group()),
            this->runTime_.timeName(),
            this->mesh_,
            IOobject::MUST_READ,
            IOobject::NO_WRITE
        ),
        this->mesh_
    ),

    kR_
    (
        IOobject
        (
            IOobject::groupName("kR", this->U_.group()),
            this->runTime_.timeName(),
            this->mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        this->mesh_,
        dimensionedScalar("kR",this->k_.dimensions(),SMALL)
    ),

    dimlessGradkR_
    (
            IOobject
            (
                    "dimlessGradkR",
                    this->runTime_.timeName(),
                    this->mesh_,
                    IOobject::NO_READ,
                    IOobject::AUTO_WRITE
            ),
            this->mesh_,
            dimensionedScalar("dimlessGradkR",dimless,SMALL)
    ),

    alfa_
    (
        IOobject
```

```
        (
            IOobject::groupName("alfa", this->U_.group()),
            this->runTime_.timeName(),
            this->mesh_,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        this->mesh_,
        dimensionedScalar("alfa",dimless,SMALL)
    ),

    nutByAlfa_
    (
        IOobject
        (
            IOobject::groupName("nutByAlfa", this->U_.group()),
            this->runTime_.timeName(),
            this->mesh_,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        this->mesh_,
        dimensionedScalar("nutByAlfa",this->nut_.dimensions(),SMALL)
    ),

    Cnu_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "Cnu",
            this->coeffDict_,
            0.18
        )
    ),

    Ce1_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "Ce1",
            this->coeffDict_,
            1.55
        )
    ),

    sigmaK_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "sigmaK",
            this->coeffDict_,
            1.0
        )
    ),
```

```
    sigmaEps_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "sigmaEps",
            this->coeffDict_,
            1.2
        )
    )
```

Also add

```
    bound(epsilon_, this->epsilonMin_);
```

under

```
    bound(k_, this->kMin_);
```

Within the function of

```
template<class BasicTurbulenceModel>
bool kEqn<BasicTurbulenceModel>::read()
```

replace

```
        Ck_.readIfPresent(this->coeffDict());
```

with

```
        Cnu_.readIfPresent(this->coeffDict());
        Ce1_.readIfPresent(this->coeffDict());
        sigmaK_.readIfPresent(this->coeffDict());
        sigmaEps_.readIfPresent(this->coeffDict());
```

Remove the function below:

```
template<class BasicTurbulenceModel>
tmp<volScalarField> kEpsilonSAS<BasicTurbulenceModel>::epsilon() const
{
    return tmp<volScalarField>
    (
        new volScalarField
        (
            IOobject
            (
                IOobject::groupName("epsilon", this->U_.group()),
                this->runTime_.timeName(),
                this->mesh_,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            this->Ce_*k()*sqrt(k())/this->delta()
        )
    );
}
```

Within the function of:

```
template<class BasicTurbulenceModel>
void kEpsilonSAS<BasicTurbulenceModel>::correct()
```

add

```
    Info << "This is kEpsilonSAS" << endl;
```

before

```
    // Local references
```

Also add

```
    // Calculate Ce2 dynamically
    tmp<volScalarField> ReT(sqr(k_)/(this->nu()*(epsilon_+this->epsilonMin_)));
    tmp<volScalarField> f(sqrt(sqr(ReT()/30)+ReT()/15)-ReT()/30);
    volScalarField Ce2(11/6*f()+25/(ReT()+SMALL)*sqr(f()));
    ReT.clear();
    f.clear();

    volScalarField epsilon(epsilon_ + this->epsilonMin_);
    volScalarField k(k_ + this->kMin_); // added

    // Dissipation equation // added
    tmp<fvScalarMatrix> epsEqn
    (
        fvm::ddt(alpha, rho, epsilon_)
      + fvm::div(alphaRhoPhi, epsilon_)
      - fvm::laplacian(alpha*rho*DepsilonEff(), epsilon_)
     ==
        Ce1_*alpha*rho*G*epsilon_/k
      - fvm::SuSp(((2.0/3.0)*Ce1_)*alpha*rho*divU, epsilon_)
      - fvm::Sp(Ce2*alpha*rho*epsilon_/k, epsilon_)
      + epsilonSource()
      + fvOptions(alpha, rho, epsilon_)
    );

    epsEqn.ref().relax();
    fvOptions.constrain(epsEqn.ref());
    //epsEqn.ref().boundaryManipulate(epsilon_.boundaryFieldRef());
    solve(epsEqn);
    fvOptions.correct(epsilon_);
    bound(epsilon_, this->epsilonMin_);
```

before

```
    tmp<fvScalarMatrix> kEqn
```

Within the function of

```
    tmp<fvScalarMatrix> kEqn
```

replace

```
        alpha*rho*G
```

with

```
        alpha*rho*G*alfa_
```

also replace

```
- fvm::Sp(this->Ce_*alpha*rho*sqrt(k_)/this->delta(), k_)
```

with

```
- fvm::Sp(alpha*rho*epsilon_/k, k_)
```

Then recompile:

```
cd $WM_PROJECT_USER_DIR/src/TurbulenceModels
touch incompressible/turbulentTransportModels/turbulentTransportModels.C
./Allwmake
```

Chapter 4

Tutorial Setup

This chapter covers the necessary setup needed to get the pitzDaily case running with the Inagi wall damping function and the new turbulence model. The original case details can be found in following directories respectively:

`$FOAM_TUTORIALS/incompressible/pisoFoam/LES/pitzDaily`

As mentioned in Prerequisites, the readers are presumed to have some knowledge on pitzDaily tutorial cases. So the basic introduction of the case, such as geometry will not be presented here.

4.1 k-equation with Inagi wall damping function

Copy the pitzDaily tutorial to the run directory:

```
run
rm -r pitzDaily
cp -r $FOAM_TUTORIALS/incompressible/pisoFoam/LES/pitzDaily \
$FOAM_RUN/pitzDailyKInagi
cd $FOAM_RUN/pitzDailyKInagi
```

The file structure of the pitzDaily case is similar to other OpenFOAM tutorials which contain normal `/0`, `/constant` and `/system` directories. As usual, in `/system` directory, one can find `controlDict` for write and time control, `blockMeshDict` for mesh setup, `fvSchemes` for discretisation method, and `fvSolution` for solver control. In `/constant` directory, one can find `transportProperties` for viscosity and `turbulenceProperties` for turbulence models. As the Wall damping function is hard-coded into the SGS model `kEqnInagi`, so the damping function can be applied by directly replacing the SGS model in `/constant/turbulenceProperties`, i.e.:
replace

```
    LESModel        dynamicKEqn;
```

with

```
    LESModel        kEqInagi;
```

Then, just type following command to run the case:

```
blockMesh
pisoFoam >& log&
```

4.2 Two-equation SGS Model

4.2.1 Getting Started

Same as the case setup for Inagi wall damping function, the pitzDaily tutorial should be copied to the run directory first:

```
run
rm -r pitzDaily
cp -r $FOAM_TUTORIALS/incompressible/pisoFoam/LES/pitzDaily \
$FOAM_RUN/pitzDailyKESAS
cd $FOAM_RUN/pitzDailyKESAS
```

4.2.2 Changes in 0 directory

As this case is only for demonstration, so extra parameters, such as `nuTilda` and `s`, which are not solved in current turbulence model should be removed. However, the `U`, `p`, `nuSgs` and `k` stay the same as in existing tutorial.

The new SGS model will solve ε-equation which requires `epsilon` dictionary to be provided in `/0` directory as an initial condition. So copy `/0/k` to `/0/epsilon`, then change `object` from `k` to `epsilon`, and make remaining `epsilon` file as:

```
dimensions      [0 2 -3 0 0 0 0];

internalField   uniform 79e-5;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform 79e-5;
    }

    outlet
    {
        type            zeroGradient;
    }

    upperWall
    {
        type            fixedValue;
        value           uniform 0;
    }

    lowerWall
    {
        type            fixedValue;
        value           uniform 0;
    }

    frontAndBack
    {
        type            empty;
    }
}
```

`$FOAM_TUTORIALS` has `pitzDaily` case for k-equation SGS model and k-ε RANS model. The initial k value for current case stays the same as the tutorial case for k-equation SGS model. It is found out that the `k` value in `/0` directory for the SGS model is much smaller than the case for RANS model. So the ratio of ε between the RANS and LES cases is kept as the same ratio of `k` between existing RANS and LES cases.

4.2.3 Changes in system directory

In `/system/controlDict`:
comment out everything except `fieldAverage1`. Then within the `fieldAverage1` only keep `mean on;` for `U`, and turn others off.

In `/system/fvSchemes`:
add

```
div(phi,epsilon)      Gauss limitedLinear 1;
```

under

```
div(phi,k)      Gauss limitedLinear 1;
```

In `/system/fvSolution`:
replace

```
"(U|k|B|nuTilda|s)"
```

with

```
"(U|k|epsilon|B|nuTilda|s)"
```

4.2.4 Changes in constant directory

Keep the `/constant/transportProperties` unchanged.

In `/constant/turbulenceProperties`:
replace

```
LESModel        dynamicKEqn;
```

with

```
LESModel        kEpsilonSAS;
```

also replace

```
delta        cubeRootVol;
```

to

```
delta        vanDriest;
```

4.2.5 Running the code

Just type following command to run the case:

```
blockMesh
pisoFoam >& log&
```

4.3 Post-processing in ParaView

It is reported that the new model can lead to divergence, whereas removing the energy backscatter term in k-equation can improve the stability. The real cause for divergence is subject to further investigation. The result shown in the test cases is based on the code which is without the energy backscatter term in k-equation.

A preliminary comparison between k-equation with Inagi wall damping function, k-ε SGS model, k-equation SGS model with `cubeRootVol` Δ and conventional k-ε RANS model is conducted. pitzDaily tutorial case located in `$FOAM_TUTORIALS` is used as the test case.

The inlet velocity are set to be $U = 10m/s$ with turbulence intensity of 2%, 1% and 1% at x, y and z direction respectively. Initial condition for pressure, p, eddy viscosity, ν_t, k, and ε value are kept as same as in the original tutorial cases. The result are analysed at $t = 0.3s$.

Fig. 4.1 shows the instantaneous velocity magnitude for all SGS models, with the top one being the k-equation with Inagi wall damping function, middle one being the new model, and the bottom one being the k-equation SGS model with `cubeRootVol` Δ. The figure indicates that the new model can produce a "LES-like" velocity profile as k-equation SGS model does. In addition, the Inagi wall damping function shows a different velocity profile from k-equation SGS model with `cubeRootVol` Δ, indicating the modification works.

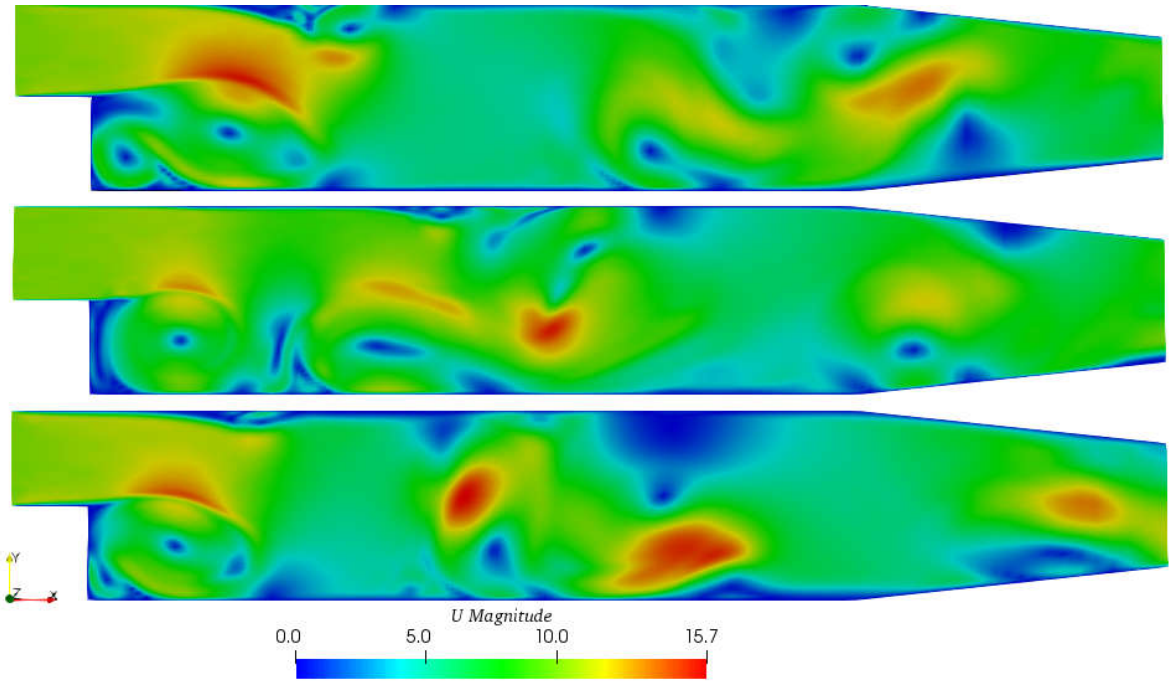


Figure 4.1: Instantaneous velocity magnitude for k-equation with Inagi wall damping function (top), k-ε SGS model (middle), and k-equation SGS model with normal Δ (bottom).

Fig. 4.2 shows the time-averaged velocity magnitude for the three SGS models plus k-ε RANS model. It is found that the time-averaged velocity profile of the three SGS models have some similarities, but different from conventional k-ε RANS model. However the RANS model cannot predict the unsteadiness for the current case.

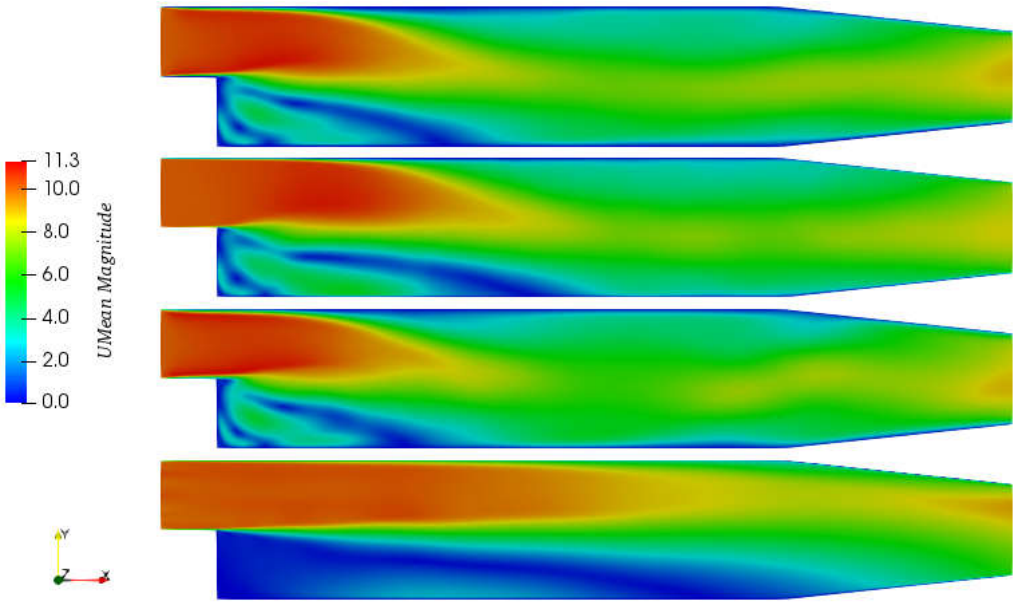


Figure 4.2: Time-averaged velocity magnitude for k-equation SGS model with Inagi wall damping function (top), k-ε SGS model (second from top), k-equation SGS model with normal Δ (second from bottom) and k-ε RANS model (bottom).

Chapter 5

Conclusion and Future Work

A wall damping function and new k-ε based Two-equation SGS model have been implemented and preliminary tested on 2D pitzDaily case. However the preliminary test shows that the new model can lead to divergence, the stability can be improved by removing the energy backscatter term from k-equation. The real cause for divergence is subject to further investigation. After removing the backscatter term, the wall dumping function and the new model are compared with conventional k-equation SGS model and conventional k-ε RANS model on the pitzDaily case . The results indicate that same as the conventional k-equation model, the new model can resolve turbulence and produce an unsteady solution. In addition the Inagi wall damping function also shows effects on the result indicating the implementation is working. As expected, the conventional k-ε RANS model is not able to capture the unsteadiness.

In terms of the future work, it is important to find out the cause of divergence. Then corresponding improvement will be done followed by extensive tests on channel flow, backward facing step and other more complicated scenarios.

Bibliography

[1] Lars Davidson. "Evaluation of the SST-SAS model: channel flow, asymmetric diffuser and axi-symmetric hill". In: (2006).

[2] Masahide Inagaki, Tsuguo Kondoh, and Yasutaka Nagano. "A mixed-time-scale SGS model with fixed model-parameters for practical LES". In: 127 (Jan. 2005).

[3] J. Blair Perot and Jason Gadebusch. "A self-adapting turbulence model for flow simulation at any mesh resolution". In: *Physics of Fluids* 19.11 (2007), p. 115105. DOI: 10.1063/1.2780195. eprint: https://doi.org/10.1063/1.2780195. URL: https://doi.org/10.1063/1.2780195.

[4] H.K. Versteeg and W. Malalasekera. *An Introduction to Computational Fluid Dynamics: The Finite Volume Method*. Pearson Education Limited, 2007. ISBN: 9780131274983. URL: https://books.google.se/books?id=RvBZ-UMpGzIC.

Appendix E

Tutorial: Parallelisation and HPC

Chapter 1

Parallel running

In order to decrease the time to run a computational simulation, the numerical domain can be divided into sub-domains, in which the governing equations will be solved for the fluid pressure and velocity, this is called parallelisation. This procedure is done using the OpenFOAM tools: `decomposePar` (for the domain partition in sub-domains) and `reconstrucPar` (for building/connecting the whole domain from the sub-domains).

The domain decomposition performed in OpenFOAM is done by breaking the geometrical and associated fields to smalls sub-domains and assigning each one to a processor. This option is already available in OpenFOAM for most of the solvers/utilities, but you do need MPI (message passing interface) library installed (`mpirun` command) and for this you can check the Wiki installation site (https://openfoamwiki.net/index.php/Main_Page) of the programme version you desire to use. The utilities that are not supported to run in parallel are those used for the parallelisation, such as `decomposePar` and `reconstructPar`, and the mesh generator `blockMesh`.

1.1 Cores/Processors availability

Before being able to do a parallelised simulation, you need to know how many processors/-cores you can use or have available in your computer, and for this you type the following line in the terminal window:

```
lscpu
```

You'll see a similar description to the following:

In which you can determine the number of physical cores/processors you can use to decompose your case in, which in the current case would be 16 processors. The process of parallelisation is done by:

1. Decomposing the domain into sub-domains
2. Running the simulation in parallel using openMPI

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                16
On-line CPU(s) list:   0-15
Thread(s) per core:    1
Core(s) per socket:    8
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Model name:            Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
Stepping:              2
CPU MHz:               2120.812
CPU max MHz:           3200.0000
CPU min MHz:           1200.0000
BogoMIPS:              4799.75
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              20480K
NUMA node0 CPU(s):     0-7
NUMA node1 CPU(s):     8-15
```

-> Virtual cores available

Physical cores = Cores/s x socket
= 8 x 2 = 16

captionNumber of processors

3. Reconstructing the whole domain

1.2 Domain decomposition

Now for the domain decomposition using `decomposePar` command, the file `decomposeParDict` within the system folder needs to be set up. The number of processors used in each direction is defined in this dictionary along with the decomposition method available: `hierarchical, manual, metis, multilevel, none, scotch, simple` and `structured` (more information of each of the methods can be checked at the OpenFOAM page: https://www.openfoam.com/documentation/guides/latest/doc/openfoam-guide-parallel.html. From which the most used are the `simple`, `scotch` and `hierarchical`. According to the method used, different specifications should be done, an example for these three methods is later shown in the case study (damBreak tutorial).

The information in each processor folder (`processorN`) will contain the mesh information, boundary conditions, initial conditions and solution of each sub-domain (N = 0, 1, 2,... `numberOfSubdomains\verb`).

1.3 Running the case

When you run in parallel, the `Allrun` file should be modified to include the following line (`<NPROCS>` = `numberOfSubdomains`):

```
mpirun \{np <NPROCS> <application/utility> \{parallel
```

Or, depending in the foam functions are already directed in the header, you can use:

```
runParallel $(getApplication)
```

1.4 Reconstructing the case

Following, to build the whole domain back to have continuous information and solution, the command `reconstrucPar` is used. And the results are shown as in the previous case, per time-step selected for the interval to be written (`writeInterval`).

1.5 Tutorial case parallel: damBreak

This is a two-dimensional multiphase tutorial case for incompressible fluids (water and air) available in most of OpenFOAM versions. The dimensions of the domain is of 0.584 x 0.584 meters, in which a water column is "dropped" when the simulation begins, at time t = 0 (remember all cases in OpenFOAM are dealt as three-dimensional, considering a unit cell in the z-direction), as seen in the figure below.

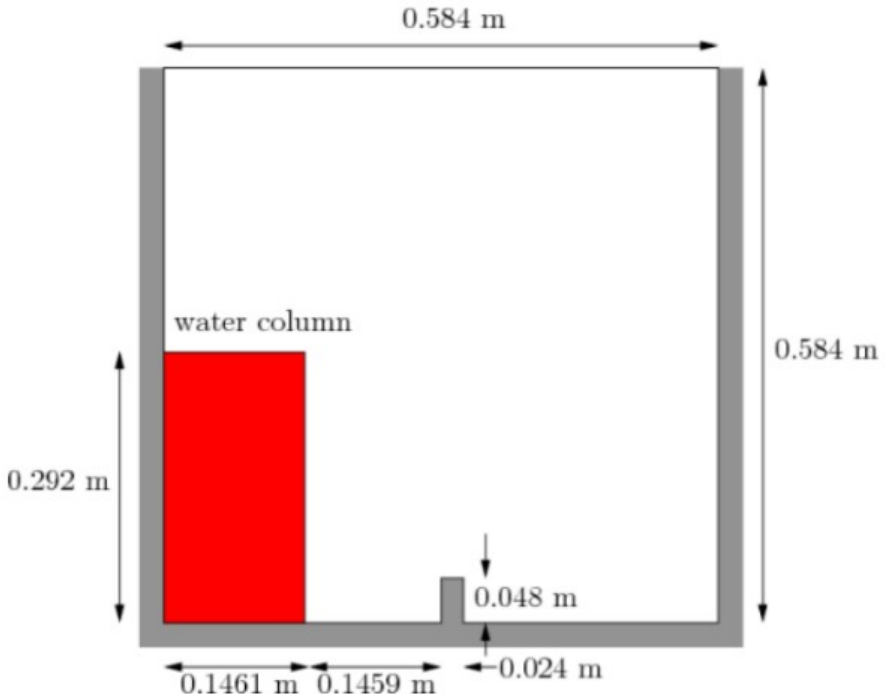


Figure 1.1: damBreak case geometry (figure taken from https://cfd.direct/openfoam/user-guide/v6-dambreak/)

The total simulation time is 1s, whilst the time-step, `deltaT`, is 0.001 s and the `writeInterval` is 0.05 s. The solver used for this multiphase case is `interFoam`. The cell discretisation in this case is of 46 x 50 x 1 cells in the x, y and z-directions, respectively (cell size of 0.012 x 0.011 x 0.100 m).

Additional relevant setup data, such as boundary and initial conditions, can be checked following previous material given during the workshop.

Now to do the `damBreak` (or any case) we must edit or include the `decomposeParDict` file

in the system folder, first we copy the tutorial case in our run folder by:

```
run
cp -r $FOAM_TUTORIALS/multiphase/interFoam/laminar/damBreak damBreak_Example
cd damBreak_Example
```

For this case, the simulation is executed using 1, 2, 4, 8 and 16 processors, and, applying the simple, hierarchical and scotch decomposition methods, in order to compare the different execution times the cases take and select the appropriate amount required.

To edit the file, we type the following in the command window:

```
cd system
nano decomposeParDict
```

For 8 processors the domain will be sub-divided in 4 pieces in the x-direction and 2 in the z-direction, whilst for the 16 processors, it is a 4 x 4 system. In the case of the 4 processors, the domain is sub-divided in 2 pieces in the x- and y-directions, as for the 2 processors case it is a 2 x 1 system.

An example of how to set the 8 processors case for the hierarchical, simple and scotch methods is done below (but for this tutorial we only include the information required for the hierarchical method):

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

numberOfSubdomains 8;

method          hierarchical;

hierarchicalCoeffs
{
    n               (4 2 1);
    delta           0.001;
    order           xyz;
}

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

Figure 1.2: Hierarchical setup case

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

numberOfSubdomains 8;

method          simple;

simpleCoeffs
{
    n               (4 2 1); // number of subdomains
    delta           0.001; // cell skew factor
}

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

Figure 1.3: Simple setup case

Now, we need to edit the `Allrun` file as follows:

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

numberOfSubdomains 8;

method          scotch;

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

Figure 1.4: Scotch setup case

```
cd ..
nano Allrun
```

To the `Allrun` file we add:

```
runApplication decomposePar
runParallel $(getApplication)
runApplication reconstructPar
```

To run `Allrun` file, we put in the terminal window:

```
./Allrun
```

After the run is performed, we should have the following files in our folder (or very similar).

```
0      0.25   0.5    0.75   1           log.blockMesh      processor0   processor5
0.05   0.3    0.55   0.8    Allclean    log.decomposePar   processor1   processor6
0.1    0.35   0.6    0.85   Allrun      log.interFoam      processor2   processor7
0.15   0.4    0.65   0.9    constant    log.reconstrucPar  processor3   system
0.2    0.45   0.7    0.95   foam.foam   log.setFields      processor4
```

Figure 1.5: List of Folders and Results

We create the dummy file for Paraview visualisation foam.foam by:

```
nano foam.foam
```

Write "XYZ" followed by "CTRL+O" and "CTRL+X" to save it. The visualisation and post-processing can be done using Paraview, for which the case should be selected to be the reconstructed one (select Reconstructed Case in the dropdown menu from Case Type in the Properties window on the left).

```
paraview foam.foam
```

The post-processing tools are the same as learned before during the workshop tutorials. Now, we are going to evaluate the amount of processors that perform better for each decomposition method (taking the simulation time from the `log.interFoam` files of each cases). It can be seen in the table below that with 16 processors, the time taken to run the cases increases considerably, therefore, using all of them, may not be a good idea for this specific case.

Refining the mesh by its half it takes longer execution time (around 1600 s). Below is a comparison of both cases at 0.30 s. As it was seen previously in the workshop, you should also do a convergence mesh study.

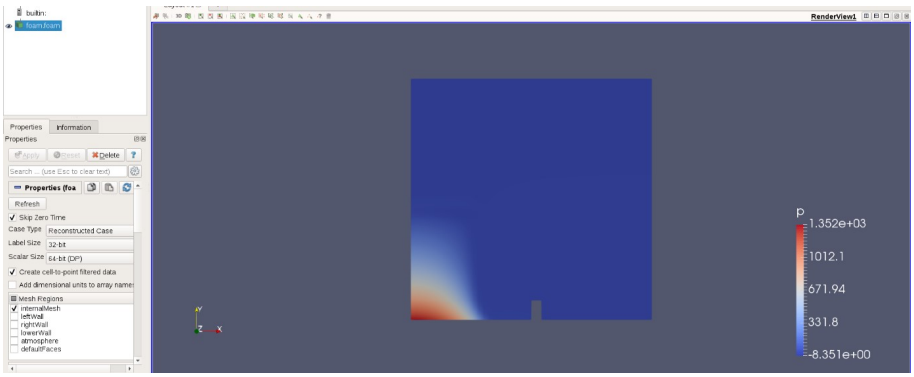


Figure 1.6: Paraview window

	Hierarchical	Scotch	Simple		
Nr. Of Processors	**Clock time (s) Mesh 1**			**Mesh 1**	**Mesh 2**
1	6	6	6	46x50x1	92x100x1
2 (2 x 1)	5	4	4	46x50x1	92x100x1
4 (2 x 2)	4	4	4	46x50x1	92x100x1
8 (4 x 2)	4	4	4	46x50x1	92x100x1
16 (4 x 4)	8	9	6	46x50x1	92x100x1

Figure 1.7: Simulation time

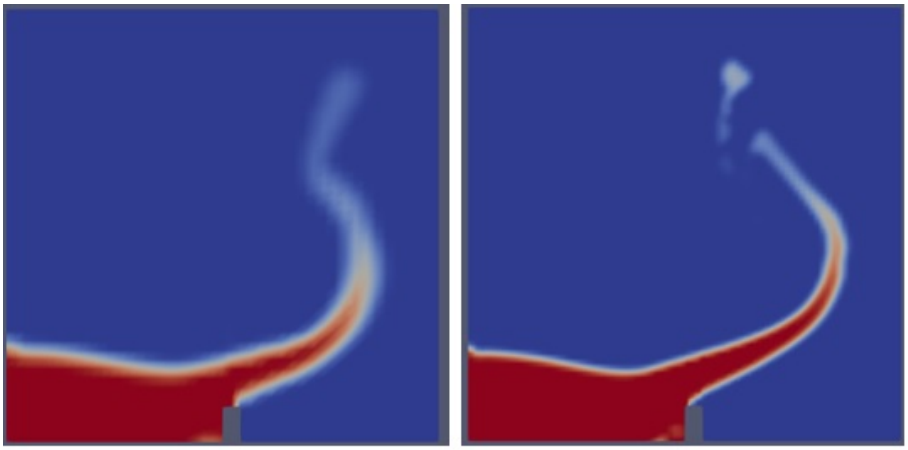


Figure 1.8: Alpha water view (original and refined cases)

Chapter 2

High Performance Computing (HPC) usage, based on UCL Computing Systems

The UCL High Performance Computing (HPC) manages three HPC systems or platforms available for researchers at UCL, these are Legion, Grace and Myriad, which you can choose according to your computational requirements (the technical specifications of each of the clusters can be found here https://wiki.rc.ucl.ac.uk/wiki/RC_Systems). These clusters run a software stack based upon Red Hat Enterprise Linux 7 and Son of Grid Engine and can be used by UNIX-like operating system users. Grace for example has more than 684 nodes, each node has 16 cores in total that can be used. The Research Centre (RC) at UCL are the ones in charge of giving the maintenance and addressing the concerns of the users and they have drop-in sessions every two Tuesdays at the main campus. The first step to use the HPC is to require access to the RC department, in which you state the technical requirements and other specifications related to the research itself. For this, you need to fill the form available in https://wiki.rc.ucl.ac.uk/wiki/Account_Services.

2.1 Access

Once you have the user access to one of the cluster, you can enter from the terminal window by typing (the examples given here are for the Grace cluster):

```
ssh <your_UCL_user_id>@grace.rc.ucl.ac.uk
```

Or, when requiring the graphic tools available for post-processing (using Paraview)

```
ssh -X <your_UCL_user_id>@grace.rc.ucl.ac.uk
```

Other way is to use PuTTY for Windows users and adding Exceed for the graphic utility, which is the way will be presented during the workshop. To access outside UCL you need to have IS VPN service accessed, and it will allow you to work with all the options mentioned above (UNIX and Windows). More information on this, can be found in

https://wiki.rc.ucl.ac.uk/wiki/Accessing_RC_Systems.

2.2 Modules

There are modules that are already loaded for all the users, which are the basic ones, to see them you can type this in the HPC terminal window:

```
module list
```

```
[<userid>@login06 ~]$ module list
Currently Loaded Modulefiles:
 1) gcc-libs/4.9.2                8) screen/4.2.1                 15) tmux/2.2
 2) cmake/3.2.1                   9) gerun                        16) mrxvt/0.5.4
 3) flex/2.5.39                  10) nano/2.4.2                   17) userscripts/1.3.0
 4) git/2.10.2                   11) nedit/5.6-aug15              18) rcps-core/1.0.0
 5) apr/1.5.2                    12) dos2unix/7.3                 19) compilers/intel/2017/update1
 6) apr-util/1.5.4               13) giflib/5.1.1                 20) mpi/intel/2017/update1/intel
 7) subversion/1.8.13            14) emacs/24.5                   21) default-modules/2017
```

Figure 2.1: Modules loaded by default

In which can already be recognised some, that are required for running OpenFOAM, such as subversion, gerun and mpi, and other known utilities such as nano and nedit. To check the programmes that are available, we type (in which we obtain a quite large list, in which OpenFOAM is included):

```
module available
```

```
openfoam/2.3.1/intel-2015-update2
openfoam/2.4.0/intel-2017-update1
openfoamplus/v1706/gnu-4.9.2
openmx/3.8.3
p7zip/15.09/gnu-4.9.2
pandoc/1.19.2.1
parallel/20181122
paraview/5.3.0
```

Figure 2.2: Modules Available (section of the list)

Among the software that are available are openfoam 2.4.0 and openfoamplus v1706. The platforms are Linux based, therefore similar commands to those learnt in this course can be used in their terminal windows. In order to know which utilities are required for OpenFOAM and Paraview, we type:

```
module show openfoamplus
```

```
/shared/ucl/apps/modulefiles/applications/openfoamplus/v1706/gnu-4.9.2: Location

module-whatis    Adds OpenFOAMplus v1706 to your environment OpenFOAM Version
conflict         openfoam
conflict         openfoamplus
prereq   gcc-libs
prereq   compilers/gnu/4.9.2
prereq   mpi/openmpi/1.10.1/gnu-4.9.2
setenv           BOOST_ARCH_PATH /shared/ucl/apps/openfoamplus/v1706/gnu-4.9.2/ThirdParty-v1706/platforms/linux64Gcc/boost_1_64_0
setenv           CGAL_ARCH_PATH /shared/ucl/apps/openfoamplus/v1706/gnu-4.9.2/ThirdParty-v1706/platforms/linux64Gcc/CGAL-4.9.1
setenv           FFTW_ARCH_PATH /shared/ucl/apps/openfoamplus/v1706/gnu-4.9.2/ThirdParty-v1706/platforms/linux64Gcc/fftw-3.3.6-pl1
setenv           FOAM_APP /shared/ucl/apps/openfoamplus/v1706/gnu-4.9.2/OpenFOAM-v1706/applications
setenv           FOAM_APPBIN /shared/ucl/apps/openfoamplus/v1706/gnu-4.9.2/OpenFOAM-v1706/platforms/linux64GccDPInt32Opt/bin
```

Required modules to be loaded for OF to work

Additional information for shortcuts

Figure 2.3: Details of openfoamplus module

```
module show paraview
```

```
/shared/ucl/apps/modulefiles/applications/paraview/5.3.0:

module-whatis    This module adds the ParaView 5.3.0 binaries to your environment. ParaView is an open-source,
prereq   gcc-libs
prereq   llvm/3.9.1
prereq   mesa/13.0.6/gnu-4.9.2
prereq   xorg-utils
conflict         paraview
prepend-path     PATH /shared/ucl/apps/paraview/5.3.0/gnu-4.9.2/ParaView-5.3.0-Qt5-OpenGL2-MPI-Linux-64bit/bin
```

Figure 2.4: Details of Paraview module

Now, in order to load these modules (for both programmes) we type:

```
module unload compilers
module unload mpi
module load gcc-libs
module load compilers/gnu/4.9.2
module load mpi/openmpi/1.10.1/gnu-4.9.2
module load openfoamplus/v1706/gnu-4.9.2
module load llvm/3.9.1
module load mesa/13.0.6/gnu-4.9.2
module load xorg-utils
module load paraview/5.3.0
```

2.3 HPC tutorial case: damBreak

Now, we are copying the case of the damBreak to our system, once you have access to the HPC, you will have one folder called output, inside another one called Scratch (which has larger storage, but is not backed up, more information on this can be found in https://wiki.rc.ucl.ac.uk/wiki/Managing_Data_on_RC_Systems), and is in there, where all your cases are going to be stored (and run).

```
cd Scratch/output
cp -r /shared/ucl/apps/openfoamplus/v1706/gnu-4.9.2/OpenFOAM-v1706/tutorials/multiphase/i
cd damBreak_Example2
```

In order to process the case (batch processing), we include the file script.sh, as follows:

```
nano script.sh
```

And to this empty file we include:

Then, we click "CTRL+O" and "CTRL+X" to save it.

Now, we change the number of processors in `decomposeParDict` as follows:

```
nano damBreak/system/decomposeParDict
```

Change `numberOfSubdomains` to 32 and `method` to scotch. Again, we click "CTRL+O"

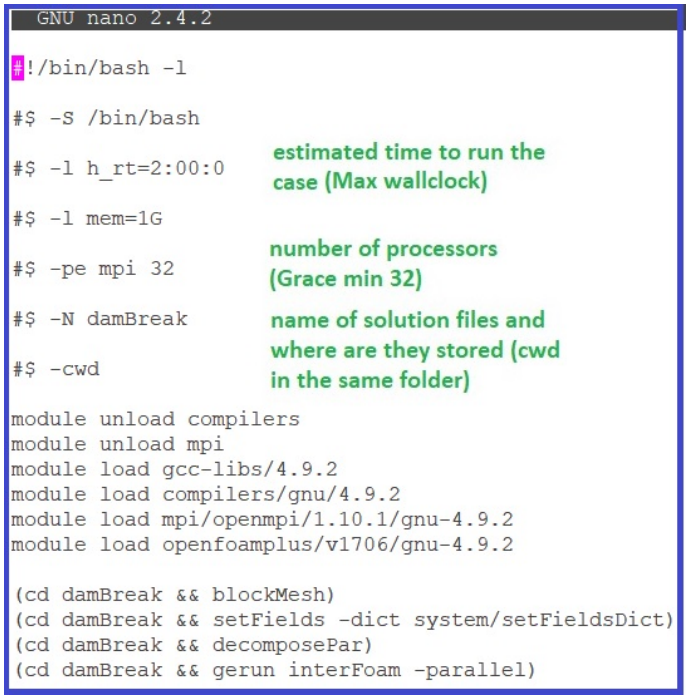


Figure 2.5: Script for HPC Running

and "CTRL+X" to save it. For Grace cluster these are the maximum wallclock times according to the required processors (https://wiki.rc.ucl.ac.uk/wiki/Resource_Allocation):

Wallclock times

Cores	Max wallclock
32-256	48hrs
257-512	24hrs
513-10912	12hrs

Figure 2.6: Wallclock time (Grace Cluster)

Now, to submit the case for the job scheduler, type the command:

```
qsub script.sh
```

To check the status of the case, the following is used (12345 represents the number assigned to the simulation once is submitted):

```
qstat -f -j 12345
```

To erase the simulation while in the batch processing, we type:

```
qdel 12345
```

It is important to know two things, first, that the selection of the cases simulated is not in order of submission (check job scheduler in https://wiki.rc.ucl.ac.uk/wiki/Legion_Scheduler) and second, it is very important to select the correct amount of time and processors to run the simulation because this is an important factor to select the cases to be simulated in the queue. To get and send data from the personal computer to the HPC you can use the following commands (available in https://wiki.rc.ucl.ac.uk/wiki/Managing_Data_on_RC_Systems):

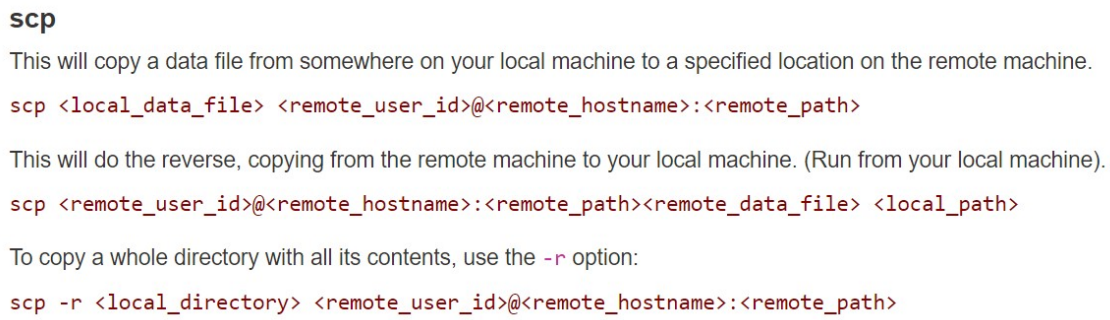


Figure 2.7: Copy-paste commands for interaction HPC-desktop

You can also do changes to the certain solvers/utilities/codes, and compile them normally as you would do in your personal computer, just verify the location with the `module show` option. The information for this section was based on the website of the Research Centre of UCL: (https://www.ucl.ac.uk/research-it-services/research-computing-platforms).

Appendix F

Programme of the 2nd UCL OpenFOAM Workshop

Programme of the 2nd UCL OpenFOAM Workshop

Day 1 (the 26th of June) - Venue: B40 LT, Darwin Building		
9:00 - 9:45 Registration desk opens and audience in seats		
9:50 – 10:00 Opening remarks, Luofeng Huang		
10:00 - 11:00	Keynote speech: OpenFOAM, from 1993 towards 2030	You will meet the founder of OpenFOAM, listening to his stories and future expectation. You will know what has driven OpenFOAM to the position today, what has been achieved, how are academia and industry using it. You will see interesting examples of how OpenFOAM simulates our real life. A round-table discussion will be performed, where four experts will have interesting communications around OpenFOAM. You can also ask questions of your interests and trigger their discussion.
	Prof Hrvoje Jasak, principal developer of OpenFOAM	
11:00 – 11:25	Theme speech 1: *(Chair – Daniela Benites)* Computational fluid dynamics of multi-body problems	
	Dr Christian Klettner, UCL teaching fellow	
11:25 – 11:50	Theme speech 2: *(Chair – Shiyu Lyu)* Large Eddy Simulation of a channel with cavities	
	Dr Yeru Shang, Engineer at Mott MacDonald	
11:50 – 12:30	Round-table discussion *(Chair – Tom Smith)*	
	Prof Hrvoje Jasak, Prof Giles Thomas, Dr Christian Klettner, Dr Yeru Shang.	
12:30 – 12:55	Theme speech 3: *(Chair – Bojan Igrec)* Hydroelastic interaction between waves and ice	
	Mr Luofeng Huang, UCL PhD	
12:55 – 13:00	Group Photo	
13:00 – 14:30	Networking Lunch	
Tutorial Session: (Chair – Daniela Benites)		
14:30 – 15:30	OpenFOAM basis	You will learn how OpenFOAM work and its usage in varied simulations of fluid. You will learn to how to use post-processing to create a digital and precise view of fluid in our life, e.g. air and water.
	Dr Rui Song, University of Liverpool	
15:30 – 16:30	Fluid dynamics 1	
	Dr Christian Klettner, UCL teaching fellow	
16:30 – 17:30	Fluid dynamics 2	
	Mr Tom Smith, UCL PhD	
17:30 – 18:30	Fluid dynamics 3	
	Dr Yeru Shang, Engineer at Mott MacDonald	

Day 2 (the 27th of June) - Venue: Anatomy B15, Medical Sciences and Anatomy		
Tutorial Session: (Chair – Morning/Shiyu Lyu, Afternoon/Daniela Benites)		
10:00 - 11:30	Ship hydrodynamics	Step by step tutorials will be given to teach the applications of OpenFOAM in maritime and civil engineering. You will learn how to model ship advancement, ocean waves; how to solve vibration and deformation of solids, as well as their interactions with fluid.
	Mr Luofeng Huang, UCL PhD	
11:30 – 13:00	Wave modelling	
	Ms Daniela Benites, UCL PhD	
14:00 – 15:30	Fluid-solid Interaction (rigid body)	
	Mr Shiyu Lyu, UCL PhD	
15:30 – 17:30	Solid Mechanics and Fluid-solid Interaction (elastic body)	
	Mr Minghao Li, Engineer at FsDynamics	

Day 3 (the 28th of June) - Venue: G15 Public Cluster, DMS Watson Building		
Tutorial Session: (Chair – Morning/Shiyu Lyu, Afternoon/Luofeng Huang)		
10:00 - 11:00	Programming1: basis	You will learn how to implement extended functions based on OpenFOAM, also how to use high-performance computation (HPC) power to speed up simulations. With these advanced skills, you will be able to simulate amazing turbulence and write your own solver for specific purposes.
	Mr Tom Smith, UCL PhD	
11:30 – 12:30	Programming2: implement functions	
	Mr Tom Smith, UCL PhD	
13:30 – 15:00	Programming3: implement a new solver	
	Mr Minghao Li, Engineer at FsDynamics	
15:00 – 16:00	Turbulence modelling	
	Mr Tom Smith, UCL PhD	
16:00 – 17:30	Parallel simulation and supercomputing	
	Ms Daniela Benites, UCL PhD	

*****_This activity is a Researcher-led Initiative that has been funded by UCL Organisational Development._

Department of Mechanical Engineering

University College London

Local committee:
Luofeng Huang
Daniela Benites
Shiyu Lyu
Tom Smith

Reading recommendations

1. OpenFOAM user guide
   http://foam.sourceforge.net/docs/Guides-a4/OpenFOAMUserGuide-A4.pdf

2. Youtube channel of József Nagy, excellent tutorial-vedios to begin with:
   https://www.youtube.com/channel/UCjdgpuxuAxH9BqheyE82Vvw?&ab_channel=J%C3%B3zsefNagy

3. OpenFOAM tutorials of Victor Pozzobon on Researchgate:
   https://www.researchgate.net/project/OpenFOAM-Tutoring

4. Maric, T., Hopken, J. and Mooney, K., 2014. The OpenFOAM technology primer.
   (first five chapters provide comprehensive OpenFOAM foundation)

5. Moukalled, F., Mangani, L. and Darwish, M., 2016. The finite volume method in computational fluid dynamics.
   (the theoretical foundation of OpenFOAM)

6. "15 days" OpenFOAM online course: https://wiki.openfoam.com/%223_weeks%22_series.

Acknowledgements

Leading the way

Professor Hrvoje Jasak (University of Zagreb)
Professor Giles Thomas (University College London)
Prof Håkan Nilsson (Chalmers University of Technology)
Prof Gavin Tabor (University of Exeter)
Dr Željko Tukovic (University of Zagreb)
Dr Vuko Vukcevic (University of Zagreb)
Dr Philip Cardiff (University College Dublin)

Logistical supports

Rochelle Rowe (UCL Doctoral Skills Team)
Kasia Bronk (UCL Doctoral Skills Team)
Emillia Brzozowska (UCL Doctoral Skills Team)
Sheetal Saujani (UCL Doctoral Skills Team)
Alexandra Grimova (UCL Mechanical Engineering Financial)
Martina Bertazzon (UCL Mechanical Engineering Financial)
Edward drinkwater (UCL Mechanical Engineering Financial)
Luke Kelly (UCL Mechanical Engineering Logistical)

Fighting together

Dimitris Stagonas (Cranfield University)
Rui Song (The university of Liverpool)
Alberto Alberello (The UUniversity of Adelaide)
Filippo Nelli (The University of Melbourne)
Sasan Tavakoli (The University of Melbourne)
Edward Ransley (University of Plymouth)
Pal Schmitt (Queen's University Belfast)
Josh Davidson (Budapest University of T&E)
Yuzhu Li (Technical University of Denmark)
Azam Dolatshah (Swinburne University of Technology)
Peiying Sun (University of Sussex)
Bojan Igrec (UCL)
Thomas Peach (UCL)
Christopher Ryan (UCL)
Andrea GL Rosa (UCL)
Christopher-John Cassar (UCL)
Dian Fan (UCL)
Nathaniel Henman (UCL)
Katherine Wang (UCL)