# Introduction to Open Source CFD for Water Resource and Recovery Facilities

**Getting Started with OpenFOAM**

nelson.marques@fsdynamics.pt; bruno.santos@fsdynamics.pt

30th September – 1st October 2017

## Progress

<table>
<thead>
<tr>
<th>TIME</th>
<th>TOPIC</th>
<th>INSTRUCTOR AND AFFILIATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:30 to 9:15</td>
<td>Welcome and Introduction to CFD for WRRF</td>
<td>Randal Samstag, Civil and Sanitary Engineer</td>
</tr>
<tr>
<td>9:15 to 10:00</td>
<td>Good Modeling Practice for CFD</td>
<td>Edward Wicklein, Carollo Engineers</td>
</tr>
<tr>
<td>10:00 to 10:30</td>
<td>Break</td>
<td></td>
</tr>
<tr>
<td>10:30 to 12:00</td>
<td>Introduction to CFD using Open FOAM</td>
<td>Nelson Marques, blueCAPE</td>
</tr>
<tr>
<td>12:00 to 1:30</td>
<td>Lunch</td>
<td></td>
</tr>
<tr>
<td>1:30 to 3:00</td>
<td>Getting Started with OpenFOAM: Example Case (Parshall Flume) - Setup, Meshing, Pre Processing, Simulation and Post Processing</td>
<td>Nelson Marques, blueCAPE</td>
</tr>
<tr>
<td>3:00 to 3:30</td>
<td>Break</td>
<td>Edward Wicklein, Carollo Engineers</td>
</tr>
<tr>
<td>3:30 to 4:00</td>
<td>CFD for flow splitting</td>
<td></td>
</tr>
<tr>
<td>4:00 to 5:00</td>
<td>OpenFOAM case: Flow Splitting</td>
<td>Nelson Marques, blueCAPE</td>
</tr>
</tbody>
</table>
Pre-processing

Meshing

Section Contents

1. Available Meshers
2. blockMesh
3. snappyHexMesh
   • Surface preparation and import
   • Background mesh
   • Mesh parameters
   • Visualisation
Available Meshers (1/4)

Meshers available from within OpenFOAM:

- **blockMesh**: [openfoamwiki.net/index.php/BlockMesh](http://openfoamwiki.net/index.php/BlockMesh)
- **snappyHexMesh**: [openfoamwiki.net/index.php/SnappyHexMesh](http://openfoamwiki.net/index.php/SnappyHexMesh)
- **foamyHexMesh**: [openfoam.org/release/2-3-0/foamyhexmesh/](http://openfoam.org/release/2-3-0/foamyhexmesh/)
- **foamyQuadMesh**: 2D version of **foamyHexMesh**.
- **polyDualMesh**: this is mainly a mesh conversion tool, aimed at converting tetrahedral meshes into polyhedral meshes.

Available Meshers (2/4)

Known open-source GUIs for OpenFOAM’s meshers:

- **SwiftBlock**: [openfoamwiki.net/index.php/Contrib/SwiftBlock](http://openfoamwiki.net/index.php/Contrib/SwiftBlock)
- **SwiftSnap**: [openfoamwiki.net/index.php/Contrib/SwiftSnap](http://openfoamwiki.net/index.php/Contrib/SwiftSnap)
- **Helyx-OS**: [engys.github.io/HELYX-OS/](http://engys.github.io/HELYX-OS/)
Available Meshers (3/4)

Other open-source meshers compatible with OpenFOAM:

- **cfMesh**: [sourceforge.net/projects/cfmesh/](https://sourceforge.net/projects/cfmesh/)
- **Netgen**: [sourceforge.net/projects/netgen-mesher/](https://sourceforge.net/projects/netgen-mesher/)
- **TetGen**: [wias-berlin.de/software/tetgen/](https://wias-berlin.de/software/tetgen/)
- **Salome**: [www.salome-platform.org](http://www.salome-platform.org)
  - Tutorials can be found here: [www.caelinux.org/wiki](http://www.caelinux.org/wiki)
- **terrainBlockMesher**: [github.com/jonasIWES/terrainBlockMesher](https://github.com/jonasIWES/terrainBlockMesher)
- **extBlockMesh**: [www.etudes-ng.net/home/development/extBlockMesh](https://www.etudes-ng.net/home/development/extBlockMesh)

Available Meshers (4/4)

Commercial meshers compatible with OpenFOAM:

- **ANSA**: [www.beta-cae.com/ansa.htm](http://www.beta-cae.com/ansa.htm)
- **Pointwise**: [www.pointwise.com](http://www.pointwise.com)
- And most of the ones presented here: [openfoamwiki.net/index.php/GUI](http://openfoamwiki.net/index.php/GUI)

Commercial meshers, not 100% compatible:

- Fluent and related ANSYS software, such as T-Grid
- Star-CD and Star-CCM+ by CD-adapco
blockMesh (1/13)

Designed for creating a mesh from various sets of blocks, this mesher is very powerful but can quickly get very complicated to use.

We will showcase its basic use with a simple backward-facing step:
• Length of initial channel section: \(3.8 \, H\)
• Height of the initial channel section: \(4 \, H\)
• Length of the final channel: \(30 \, H\)
  • Total length: \(33.8 \, H\)
• Height of the step: \(H\)
  • Which implies that the height of the final channel section is: \(5 \, H\)
• The characteristic height \(H\) to be used will be 1 meter.
blockMesh (3/13)
Why all those reference points?
Because we will create 3 blocks, namely:
• ADEB
• EFGI
• BEIC

Even though the geometry is defined solely in 2D, OpenFOAM needs the 3rd Dimension just the same, which means that the list of points will be doubled, one for the front, another for the back.

blockMesh (4/13)
To make it easier to create these blocks, we use a few strategies:
• Define the reference X positions for: AD, BEF and CIG
• Define the reference Y positions for: ABC, DEI and FG
• A list of the indexes associated to each point in the front, as well as a list of the indexes for the points in the back.
In practice, our “blockMeshDict” will look like this:

```plaintext
convertToMeters 1.0;

// positions ABCDEFGI
ADx     0.0;
BEFx    3.8;
CIGx    33.8;
ABCy    5.0;
DEIy    1.0;
FGy     0.0;
Aa      0;
Ba      1;
Ca      2;
Da      3;
Ea      4;
... 

Fa     5;
Ga     6;
Ia     7;
Ab     8;
Bb     9;
Cb     10;
Db     11;
Eb     12;
Fb     13;
Gb     14;
Ib     15;
```

List of vertices (front & back):

```plaintext
vertices
{
    //Z=0
    ($ADx $ABCy 0.0) //A, 0
    ($BEFx $ABCy 0.0) //B, 1
    ($CIGx $ABCy 0.0) //C, 2
    ($ADx $DEIy 0.0) //D, 3
    ($BEFx $DEIy 0.0) //E, 4
    ($BEFx $FGy 0.0) //F, 5
    ($CIGx $FGy 0.0) //G, 6
    ($CIGx $DEIy 0.0) //I, 7
    ... 
    //Z=0.1
    ($ADx $ABCy 0.1) //A, 8
    ($BEFx $ABCy 0.1) //B, 9
    ($CIGx $ABCy 0.1) //C, 10
    ($ADx $DEIy 0.1) //D, 11
    ($BEFx $DEIy 0.1) //E, 12
    ($BEFx $FGy 0.1) //F, 13
    ($CIGx $FGy 0.1) //G, 14
    ($CIGx $DEIy 0.1) //I, 15
    ... 
};
```
blockMesh (7/13)

blocks
{
    // ADEB
    hex ($Aa $Da $Ea $Ba $Ab $Db $Eb $Bb) (1 1 1) simpleGrading (1 1 1)
    // EFGI
    hex ($Ea $Fa $Ga $Ia $Eb $Fb $Gb $Ib) (1 1 1) simpleGrading (1 1 1)
    // BEIC
    hex ($Ba $Ea $Ia $Ca $Bb $Eb $Ib $Cb) (1 1 1) simpleGrading (1 1 1)
};

blockMesh (8/13)

boundary
{
    inlet
    {
        type patch;
        faces
        {
            ($Aa $Da $Db $Ab)
        }
    }
    outlet
    {
        type patch;
        faces
        {
            ($Ca $Ia $Ib $Cb)
            ($Ia $Ga $Gb $Ib)
        }
    }
    upperWall
    {
        type wall;
        faces
        {
            ($Aa $Ba $Bb $Ab)
            ($Ba $Ca $Cb $Bb)
        }
    }
    lowerWall
    {
        type wall;
        faces
        {
            ($Da $Ea $Eb $Db)
            ($Ea $Fa $Fb $Eb)
            ($Fa $Ga $Gb $Fb)
        }
    }
}
blockMesh (9/13)

Geometrical boundaries (2/2):

Reminder: the vertices should be defined counter-clockwise and in the same order for the front and back.

```plaintext
frontAndBack
{
  type empty;
  faces
  {
    ($Aa$ Da $Ea$ $Ba$)
    ($Ba$ $Ea$ $Ia$ $Ca$)
    ($Ea$ $Fa$ $Ga$ $Ia$)
    ($Ab$ $Db$ $Eb$ $Bb$)
    ($Bb$ $Eb$ $Ib$ $Cb$)
    ($Eb$ $Fb$ $Gb$ $Ib$)
  }
}
```

blockMesh (10/13)

Last but not least, the “edges” list and “mergePatchPairs”:

```plaintext
edges
{
);
mergePatchPairs
{
);
```

Where:

- **edges**: for providing a list of edge modifiers, e.g.:
  
  ```plaintext
  arc 0 5 (0.469846 0.17101 -0.5)
  ```

- **mergePatchPairs**: for merging patches, e.g. if we had two geometrical boundaries that we wanted to *stitch together*. 
Workflow:
1. We use a tutorial case as a basis, for example "basic/potentialFoam/pitzDaily".
2. Modify the file "system/blockMeshDict".
3. Run blockMesh.
4. If all goes well, run paraFoam.
5. What we will see in ParaView is something like this:

```
//ADEB
hex ($Aa $Da $Ea $Ba $Ab $Db $Eb $Bb) (20 19 1) simpleGrading (1 1 1)

//EFGI
hex ($Ea $Fa $Ga $Ia $Eb $Fb $Gb $Ib) (5 150 1) simpleGrading (1 1 1)

//BEIC
hex ($Ba $Ea $Ia $Ca $Bb $Eb $Ib $Cb) (20 150 1) simpleGrading (1 1 1)
```

Will result in this:
blockMesh (13/13)

The grading over each direction depends in the order of the vertices:

hex ($Ba$ $Ea$ $Ia$ $Ca$ $Bb$ $Eb$ $Ib$ $Cb$) (20 150 1)...

This mesher acts as a chiseller, given it will work on an initial mesh:

1. Castellation:
   - Refines the selected edges, surfaces and volumes. All of the selected cells will be divided evenly.
   - Remove unwanted cells, namely the ones inside or outside of the provided geometrical models.

2. Snapping:
   - The cells near the surfaces of the geometrical models will be cut and/or snapped onto those surfaces.

3. Layer addition:
   - The internal mesh near the surfaces is compacted and prismatic cells are added between the internal mesh and the surfaces.

snappyHexMesh (1/49)
Examples of each stage (1/2):

Initial (background) mesh

Castellated mesh

Examples of each stage (2/2):

Snapped mesh

Before and after layer addition (section-cut view)
Surface preparation and import (1/6):
1. Fix the geometry before exporting:
   1. Keep it simple
   2. Remember which side you will mesh
   3. If it is too complex, try something simpler first
   4. Know your physics and solvers
   5. Know the format you are exporting to
   6. Know your units (preferably SI)
   7. Know the location of your geometry
2. Export the geometry to a suitable format
3. Check the resulting discretized geometry

Surface preparation and import (2/6):
• OpenFOAM mostly deals with geometric models in STL and Wavefront OBJ files, where both must already be tessellated and preferably in ASCII format.
• Both file formats can handle the identification of separate groups of triangles as patches. For example, in STL, this means that there can be several solids in a single STL file.
snappyHexMesh (6/49)

Surface preparation and import (3/6):

Example of an STL:

```plaintext
solid backWall
  facet normal 1 0 0
  outer loop
     vertex 5.75 0.687 -0.65
     vertex 5.75 -0.667 -0.188
     vertex 5.75 -0.669 -0.257
  endloop
  endfacet
endsolid backWall

solid inlet
...
endsolid inlet
```

Example of an OBJ:

```plaintext
# Wavefront OBJ file
# Regions:
# 0  frt-fairing_001_1
# 1  windshield_002_2
...
# points : 132871
# triangles : 331653
...
v -0.00568945 0.0242072 1.6e-08
...
v 1.74458 -0.00375868 1.01589
  g frt-fairing_001_1
  f 64491 64463 65119
  ...
  f 17054 16748 17078
  g windshield_002_2
  f 66424 67098 66986
  ...
```

snappyHexMesh (7/49)

Surface preparation and import (4/6):

Once the file is in STL or OBJ format, we can check how OpenFOAM will interpret, by running:

```
surfaceCheck path/to/the_file.stl
```

It will give us a lot of information, such as:

- Bounding box
- Regions (= solids)
- If any illegal triangles were found
- An histogram of the quality of the triangles
- Lengths of the triangle edges and nearness of vertices
- If the geometry is closed and identifies parts if not closed
Surface preparation and import (5/6):
Place the exported geometry file in the folder “constant/triSurface”.

Move/translate the whole geometry so that the center of the geometry coincides with the origin of the referential.

This can be done with surfaceTransformPoints, e.g.:

```
surfaceTransformPoints -translate '(-4.25 0.687 -0.55)' \constant/triSurface/halfParshall.org.stl \constant/triSurface/halfParshall.stl
```

Surface preparation and import (6/6):
Configure the dictionary file “system/surfaceFeatureExtractDict”, for extracting feature edges from the geometry, e.g.:

```
"halfParshall.stl"
{
    extractionMethod extractFromSurface;
    extractFromSurfaceCoeffs
    {
        includedAngle 150;
    }
    writeFeatureEdgeMesh no;
}
```

Then simply run:

```
surfaceFeatureExtract
```
snappyHexMesh (10/49)

Background mesh (1/6):
Also known as “initial mesh” or “base mesh”, this is the initial mesh that will be used by snappyHexMesh.
Which means that:
• The initial mesh should be similar to the geometry;
• Or at least good reference points from the geometry must be part of this initial mesh.
Otherwise, things like this happen. The image shown is for level 0, i.e. initial mesh.

snappyHexMesh (11/49)

Background mesh (2/6):
Although we can try to fix it with increasing the refinement in the castellation step with snappyHexMesh:

Level 1 refinement

Level 2 refinement
snappyHexMesh (12/49)

Background mesh (3/6):
The usual is a single block done with blockMesh, e.g.:

```plaintext
convertToMeters 1;

vertices
{
    (-5.75 -0.68683 -0.65)
    (5.75 -0.68683 -0.65)
    (5.75 0.687 -0.65)
    (-5.75 0.687 -0.65)
    (-5.75 -0.68683 0.65)
    (5.75 -0.68683 0.65)
    (5.75 0.687 0.65)
    (-5.75 0.687 0.65)
};

patches
{
    patch maxX
    ( (1 2 6 5) )
    patch minX
    ( (0 4 7 3) )
    patch maxY
    ( (3 7 6 2) )
    patch minY
    ( (1 5 4 0) )
    patch maxZ
    ( (4 5 6 7) )
    patch minZ
    ( (0 3 2 1) )
};

blocks
{
    hex (0 1 2 3 4 5 6 7)
    (115 14 13)
    simpleGrading (1 1 1)
};
```

snappyHexMesh (13/49)

Background mesh (4/6):
Here is the result in our example:
As shown, this is a very good initial mesh for our model given the several reference points that are present in this initial mesh.
The detail is that we need an additional layer outside of this mesh.
snappyHexMesh (14/49)

Background mesh (5/6):
The additional layer outside of the mesh shown is needed for ensuring that snappyHexMesh is able to properly correlate this initial mesh with the geometrical model.

One way is to do the math and extend the vertices defined in “blockMeshDict” file, along with 2 more cells on all direction.

The other is to rely on extrudeMesh to do this for us.

snappyHexMesh (15/49)

Background mesh (6/6):
Example file “system/extrudeMeshDict” from our case:

```plaintext
constructFrom mesh;
sourceCase ".";
sourcePatches(maxX minX maxY minY maxZ minZ);
extrudeModel linearNormal;
nLayers 1;
expansionRatio 1.0;
linearNormalCoeffs {
  thickness 0.02;
}
mergeFaces false;
mergeTol 0;
```
snappyHexMesh (16/49)

Mesh parameters (1/32):
The file "system/snappyHexMeshDict" has all of the necessary settings for **snappyHexMesh** to manipulate the background mesh.

The main reference file is present in the application’s source code folder, whose location is shown with this command:

```
echo $FOAM_UTILITIES/mesh/generation/snappyHexMesh
```

Other examples can be found with this command:

```
find $FOAM_TUTORIALS –name snappyHexMeshDict
```

snappyHexMesh (17/49)

Mesh parameters (2/32):
The structure of the file is as follows:

- Initial parameters which control what steps to perform.
- "geometry" block, where we list the geometrical entities we want to either mesh onto or use as refinement references.
- "castellatedMeshControls" block, for the castellation step.
- "snapControls" block, for the snapping step.
- "addLayersControls" block, for the layer adding step.
- "meshQualityControls" block, for quality control parameters that are used during the snapping and layer adding steps.
- Last parameters are for debugging and point tolerance.
Mesh parameters (3/32) – run steps selection:
There are only 3 options for this section of the dictionary file:

- `castellatedMesh true;`
- `snap true;`
- `addLayers false;`

Setting each one to true or false will tell `snappyHexMesh` to proceed with each step.

Keep in mind that each one of these 3 steps has associated a block of settings for each, as listed in the previous slide.

Mesh parameters (4/32) – Geometry definitions (1/5):
The geometry definitions are defined within this block:

```
geometry
{
    //...
};
```

Inside this block the user should add as many geometry objects as needed, where each block is identified as follows:

```
object_file_name.extension
{
    type the_type_of_object;
    //settings for this object
}
```
From our example case, we have two geometries:

```
"halfParshall.stl"
{
  type triSurfaceMesh;
  regions
  {
    backWall
    {
      name backWall;
    }
    top
    {
      name top;
    }
  }
}
```

- **Type** "triSurfaceMesh" is the one used for external model files.
- As the designation implies, this model must be in a file format that uses a triangle discretization of the surfaces.
- We're using STL, as it's the easiest one to generate and manipulate.

(continues on next slide...)
snappyHexMesh (22/49)

Mesh parameters (7/32) – Geometry definitions (4/5):
- The regions block is where we can rename the solids provided in the STL and give them the names we want.
- These renamed names will later be used for defining the patches that make up the surface mesh of our final mesh.
- This is also used because without this renaming step, the default names assigned by snappyHexMesh would likely be something like this:
  - halfParshall_backWall
  - halfParshall_bottomWall
  - halfParshall_sideWall

snappyHexMesh (23/49)

Mesh parameters (8/32) – Geometry definitions (5/5):
Other internal geometrical entities can be used, some examples:
- searchableBox – defines a box by bounding points (which we used in our example case);
- searchableSphere – defines a sphere by center and radius;
- searchableCylinder – defines a cylinder by height vector and radius;
- searchablePlate – defines a plate by origin and span;
- searchablePlane (planeType PointAndNormal) – defines a plane by point and normal vector;
- searchablePlane (planeType 3Points) – defines a box by plane by three points.
snappyHexMesh (24/49)

Mesh parameters (9/32) – Castellation controls (1/11):

First we need to understand how this step handles refinement levels:

1. Initial mesh is the level of refinement 0 (zero).

2. Each level of refinement indicates the multiple of 2 for dividing cells. In other words:
   - level 1: a cell from the initial mesh is split into 2 parts on all major directions (X, Y, Z), in its own referential. In other words, each cell will be subdivided into 8 smaller cells.
   - level 2: cell split into 4 parts over X,Y,Z → will be subdivided into 64 smaller cells.
   - level 3: split into 8 parts over X,Y,Z → 512 smaller cells.

snappyHexMesh (25/49)

Mesh parameters (10/32) – Castellation controls (2/11):

3. Refinement levels are not cumulative:
   - if two or more overlapping zones and/or surfaces are set to different levels of resolution, it's only the greatest value that will be used.
   - For example:
Mesh parameters (11/32) – Castellation controls (3/11):

Choosing which cells to refine is done based on the three major types of geometries in 3D space:

- **Lines**: Provided as feature edges in OpenFOAM's the file format “.eMesh”.
- **Surfaces**: More specifically, all of the surfaces from the geometries defined in the “geometry” block.
- **Volumes**: Can refine inside or outside of closed shells from the “geometry” block; or distance based for any geometric entity from the “geometry” block.

Mesh parameters (12/32) – Castellation controls (4/11):

In block "castellatedMeshControls", we focus on:

- "maxLocalCells" is used when running in parallel. This is the guideline on when it should transfer excess cells to other processor sub-domains, assuming the others are less populated.
- "maxGlobalCells" is the maximum number of cells that **snappyHexMesh** will allow to be generated in the refinement step. A rule of thumb is that mesh generation can take up somewhere between 1 and 2 GB of RAM for each one million cells.
Mesh parameters (13/32) – Castellation controls (5/11):

- “features” provides a list of nameless blocks that define the feature edge files to be used for refinement and later on for snapping.
- For each nameless block, we can define the file to be used and the associated refinement level.
- The advantages of using feature edges as references for the refinement/castellation step depend on the geometry at hand.
- Best to define the refinement level for each file to be 0; otherwise, there is a risk of having refined cells in very impractical locations.

From our example case, this is the “features” block:

```
features
{
  file "halfParshall.eMesh";
  level 0;
}
```

This is what would happen with level 1
snappyHexMesh (30/49)

Mesh parameters (15/32) – Castellation controls (7/11):

- “refinementSurfaces” block provides the refinement settings for the surfaces on the geometries defined in the geometry block.
- The structure to be followed is similar to the one used in the geometry block.
- For each *solid* (named region) we can define two sets of refinement levels, namely the minimum and the maximum level.
- **Note:** It is good to remember that a more uniform mesh is usually preferable to a mesh with many mesh level refinement transitions.

snappyHexMesh (31/49)

Mesh parameters (16/32) – Castellation controls (8/11):

From the example case – “refinementSurfaces” block:

```plaintext
refinementSurfaces
{
    "halfParshall.stl"
    {
        level (0 0);
        regions
        {
            backWall
            {
                level (0 0);
            }
            bottomWall
            {
                level (0 0);
            }
        }
        symmetry
        {
            level (0 0);
        }
        top
        {
            level (0 0);
        }
    }
    outlet
    {
        level (0 0);
    }
}
```
Mesh parameters (17/32) – Castellation controls (9/11):

- “resolveFeatureAngle” is in essence what defines the smallest angle between two surfaces that we can use to consider if there is a sharp edge between the two.
- “refinementRegions” already mentioned regarding refinement volumes, which are based on the geometries defined in “geometry”.

From our example case:

```plaintext
refinementRegions
{
    refinementBox
    {
        mode inside;
        levels ((1e-15 2));
    }
}
```

The first value, “1e-15”, is meant to be used only for the “distance” mode.

Mesh parameters (18/32) – Castellation controls (10/11):

- “locationInMesh” will tell the mesher that a particular point is inside or outside of the closed surface. Use with care!
- If the point is inside, it will preserve only the mesh inside the geometry.
- If the point is outside, it will preserve only the mesh outside of the geometry.
- Examples:

```
Good points
```

bad points
snappyHexMesh (34/49)

Mesh parameters (19/32) – Castellation controls (11/11):

How is “locationInMesh” used?

It’s used for drawing lines between this reference point and the centers of cells and faces, to ascertain which cells are inside or outside of the mesh.

Example for when the point outside of the geometry.

snappyHexMesh (35/49)

Mesh parameters (20/32) – Snapping controls (1/5):

The block for this snapping/morphing step starts and ends like this:

```
snapControls
{
    //...
}
```

As for the specific parameters for this block, it is all well explained in the comments already available in OpenFOAM’s example, therefore we’ll address how each parameter can affect the snapping process.
snappyHexMesh (36/49)

Mesh parameters (21/32) – Snapping controls (2/5):

- “nSmoothPatch” is the number of iterations for smoothing. When set to 0, stays true to the initial mesh shape; when set too high, will result in a surface mesh that will resemble a seawater blowfish.
- “tolerance” is the relative distance for cell edge length for snapping points.
  - Common values are 1.0 and 2.0.
  - If values are too high, it risks snapping vertexes on the mesh that have nothing to do with the nearest surface.
  - If the values are too low, snapping might never occur.

snappyHexMesh (37/49)

Mesh parameters (22/32) – Snapping controls (3/5):

- “nSolveIter” is the number of iterations for adjusting the mesh.
  - Set to 0 if the surface of the base mesh is parallel to the surfaces of the final mesh.
  - Other integer values above zero can improve the resulting mesh.
- “nRelaxIter” is pretty much a must-use.
  - Default value of 5 usually does a very good job.
  - More iterations can make it slower than needed.
snappyHexMesh (38/49)

Mesh parameters (23/32) – Snapping controls (4/5):

- “nFeatureSnapIter” is the number of iterations for snapping to feature edges.
  - The default value of 10 is usually the best value.
  - Too low can result in an incomplete morph.
  - Too high can lead to strange mesh distortions.
- “implicitFeatureSnap” when set to true, you don't need the “.eMesh” files.
- “explicitFeatureSnap” when set to true, it will use the “.eMesh” files listed in the features block at “castellatedMeshControls”.

snappyHexMesh (39/49)

Mesh parameters (24/32) – Snapping controls (5/5):

- “multiRegionFeatureSnap” when set to true, it will only work if “explicitFeatureSnap” is also set to true.
  - Only useful for multi-region meshing, namely on both inside and outside.
  - Using this is risky, because it will enforce mesh quality controls for both sides of the mesh, namely inside and outside, which can result in crooked looking mesh.

When set to false

![Image](image1)

When set to true

![Image](image2)
Mesh parameters (25/32) – Layer addition controls (1/7):
The importance of adding layers is related to the requirements needed for the wall treatment models being used for the simulation.

The block for this layer addition step starts and ends like this:

```plaintext
addLayersControls
{
    //...
}
```

Mesh parameters (26/32) – Layer addition controls (2/7):
The more relevant layer adding parameters are:

- “relativeSizes” when set to true, will use dimensioning relative to the cell edges where the layers will be added.
- “layers” block lists all patches (boundary surfaces) that should have layers added or not to them. It can even allow per-patch definition of the layer adding parameters.
- “expansionRatio” is the factor of how each layer relates to the previous one added.
- “finalLayerThickness” is the first reference layer size, which is the thickness of the layer farthest from the original surface.
snappyHexMesh (42/49)

Mesh parameters (27/32) – Layer addition controls (3/7):

• “minThickness” is the smallest desired thickness at the middle axis of a layer’s cell.

• “nBufferCellsNoExtrude” is the number of cells on the mesh before adding layers, relative to the border of a patch. For example, with a value of 1, only the centre cells on the mesh below would get layers added:

![Surface mesh of the patch on which layers will be added.]

Example of the “layers” block:

```plaintext
layers
{
  backWall
  {
    nSurfaceLayers 1;
  }
  bottomWall
  {
    nSurfaceLayers 1;
  }
...}
```

```
inlet
{
  nSurfaceLayers 1;
  // Per patch layer data
  expansionRatio 1.0;
  finalLayerThickness 2.0;
  minThickness 0.1;
}
...
```

snappyHexMesh (43/49)

Mesh parameters (28/32) – Layer addition controls (4/7):

Example of the “layers” block:
Mesh parameters (29/32) – Layer addition controls (5/7):

Usual sources of problems for not being able to add layers with \textit{snappyHexMesh}:

- The mesh resulting from the snapping step has quality issues.
  - For example, the image below demonstrates that layers cannot be added near the shown distorted cells.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{example_mesh_distortion}
\caption{Example mesh distortion causing layer addition issues.}
\end{figure}

Mesh parameters (30/32) – Layer addition controls (6/7):

- Complex surfaces identified as the same \textit{solid} in STL, can result in calculations of where the layer should start and where it should end.
  - For example, if a complex profile such as the “bottomWall” in our example case, having all surfaces at the bottom catalogued as being part of “bottomWall”, results in this layer adding flaw:

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{example_surface_identification}
\caption{Example surface identification causing layer addition flaw.}
\end{figure}
Mesh parameters (31/32) – Layer addition controls (7/7):

• Layer sizes can be relative or absolute.
  • If the mesh resulting from the snapping step gives rise to very small
cells where cuts had to be done, then very thin layers can appear
where the cells became smaller, when using relative sizing.
  • Absolute sizing can fix this issue, although it requires a clear notion
of the geometrical sizes desired for the added layers.
  • When layers are set to be too large, it can result in very distorted
internal mesh, given that the layers are added by compressing the
internal mesh, not by cutting the cells near the surface.

Mesh parameters (32/32):

As for the remaining controls:

• Mesh quality controls are usually well calibrated in the tutorial cases
and rarely need to be changed.
  • Nonetheless, changing them should be done with simple small test
cases, for diagnosing if they will affect your mesh or not.
• The “debug” flag(s) is(are) usually only needed for diagnosing in which
exact mesh operation things went wrong.
• The “mergeTolerance” rarely needs to be changed from the default
“1E-6”. This is relative to the bounding box of the whole mesh.
**snappyHexMesh (48/49)**

**Visualisation (1/2):**

When it comes to visualizing the mesh, the 3 important rules are:

1. Make sure you load the correct time step.
   - This is because when **snappyHexMesh** is executed without the option "-overwrite", there will be a time snapshot for each meshing step.
2. Make sure you uncheck the option "Decompose polyhedra" (shown in the next slide).
3. Make sure you use the filter "Extract Cells by Region".
   - This is because the "Slice" filter will cut the cells by decomposing the cut cells into triangles.

**snappyHexMesh (49/49)**

**Visualisation (2/2):**

If the option "Decompose polyhedra" is checked (shown at the bottom of the image), the result is that polyhedral cells will be decomposed into tetrahedral cells.

**Note:** Also uncheck "Cache mesh", if you want to be able to simply click on the "Refresh" button for seeing a newly generated mesh.
Simulation

Meshing et al

Section Contents

1. Relevant solvers
2. Boundary conditions
   - Manipulation of fields and domains
   - Turbulence models
3. Convective term discretization
4. Diffusive term discretization
5. Linear system solvers
6. Parallel runs
Relevant solvers (1/3)

- OpenFOAM has a number of solvers available: openfoam.org/features/
- With OpenFOAM, users select solvers rather than models, as in commercial CFD codes.
- There are basic modelling functionalities which permeate several solvers at once. For example: solvers that allow for turbulence modelling generally allow users to select one model from the vast available range; the same is true for the specification of thermophysical properties.
- New solvers can be easily (relatively speaking) developed if necessary, since OpenFOAM is fundamentally a tool for solving partial differential equations.

Relevant solvers (2/3)
## Relevant solvers (3/3)

<table>
<thead>
<tr>
<th>Solver</th>
<th>Base model equations</th>
<th>Other physical models</th>
<th>Turbulence</th>
<th>Time domain</th>
<th>Fluid properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>interFoam</code></td>
<td>Navier-Stokes</td>
<td>VOF</td>
<td>All incompressible flow models</td>
<td>Transient</td>
<td>Constant</td>
</tr>
<tr>
<td><code>buoyantBoussinesqPimpleFoam</code></td>
<td>Navier-Stokes</td>
<td>Energy equation, with radiation</td>
<td>All incompressible flow models</td>
<td>Transient</td>
<td></td>
</tr>
<tr>
<td><code>driftFluxFoam</code></td>
<td>Navier-Stokes</td>
<td>Settling Scalar(s)</td>
<td>Incompressible flow models</td>
<td>Transient</td>
<td>Varying</td>
</tr>
</tbody>
</table>

### Boundary conditions (1/21)

We will show:

1. How to define boundary conditions for the fields at $t = 0s$
2. How to initialize the internal fields
3. How to set-up the turbulence models and initial values

The case overview and respective boundary conditions are quickly revised in the next slide:
Boundary conditions (2/21)

- Case name: halfParshall
- Boundary conditions:
  - Inlet: 375 kg/s
  - Bottom floor and side wall: no-slip
  - Outlet surfaces: pressure outlet
  - Symmetry plane surface: symmetry
- Fluid properties:
  - Water:
    - Density: 999 kg/m$^3$
    - Dynamic Viscosity: 1.15E-3 Pa.s
  - Air:
    - Density: 1.18 kg/m$^3$
    - Dynamic Viscosity: 1.855E-5 Pa.s

Boundary conditions (3/21)

In summary, the example case has:
- 6 field files, which are initially defined in the folder 0.orig:
  - U – velocity field
  - p_rgh – pressure field, without the hydrostatic term
  - alpha.water – phase fraction field
    - 1 = 100% water
    - 0 = 100% not water (air in our case)
  - epsilon – turbulent dissipation rate field
  - k – turbulent kinetic energy field
  - nut – turbulent dynamic viscosity field
Boundary conditions (4/21)

- 4 major groups of boundary conditions:
  1. *Inlet* – assigned to the “inlet” surface
     - Velocity set to a predefined value.
     - Pressure set to zero gradient or fixed flux.
     - Phase fraction set to 1.
     - Turbulence fields $k$ and $\epsilon$ set with appropriate values (addressed in the respective subtopic section).
     - $\nu_t$ set to calculated.

(continues in the next slide...)

Boundary conditions (5/21)

2. *Outlet* – assigned to the “outlet” and “top” surfaces
   - Velocity set to zero gradient and/or a condition that disables recirculation.
   - Pressure ($p_{rgh}$) set to 0 Pa.
   - Turbulence fields $k$ and $\epsilon$ set to zero gradient and a condition that disables recirculation.
   - $\nu_t$ set to calculated.

(continues in the next slide...)
Boundary conditions (6/21)

3. **Wall** – assigned to the “backWall”, “bottomWall”, “sideWall” surfaces
   - Velocity set to no-slip, i.e. 0 m/s.
   - Pressure set to zero gradient or fixed flux.
   - Turbulence fields $k$, $\epsilon$ and $\nu$ set use wall treatments.

4. **Symmetry** – assigned to the “symmetry” surface
   - Boundary set to “symmetry” on all fields.

Boundary conditions (7/21)

Main parameters in a field file:
- **dimensions** – the units for the field:
  - Mass - kilogram
  - Length - metre
  - Time - second
  - Temperature - Kelvin
  - Quantity - mole
  - Current - ampere
  - Luminous intensity – candela
- **internalField** – the value list for the internal field
- **boundaryField** – the list of boundary conditions

Example:

\[ [0 1 1 0 0 0] = \text{m/s} \]
Boundary conditions (8/21)

For example, the $U$ field file roughly looks like this:

```plaintext
dimensions [0 1 -1 0 0 0];
internalField uniform (0.0 0.0 0.0);
boundaryField
{
    backWall
    {
        type fixedValue;
        value uniform (0.0 0.0 0.0);
    }
    bottomWall
    {
        type fixedValue;
        value uniform (0.0 0.0 0.0);
    }
    ...
}
```

Boundary conditions (9/21)

Figuring out what are the corresponding boundary conditions is usually done with the following strategies:

1. Looking into the tutorial cases that OpenFOAM has.
2. Checking the OpenFOAM User Guide, section “5.2 Boundaries”.
3. Looking at the complete list of boundary conditions, available in the Doxygen generated code documentation: [cpp.openfoam.org/v4/](http://cpp.openfoam.org/v4/)
   - In the section “Using OpenFOAM” are 3 links:
     - [FunctionObjects](http://cpp.openfoam.org/v4/) namespace [Foam::functionObjects](http://cpp.openfoam.org/v4/)
     - [Boundary Conditions](http://cpp.openfoam.org/v4/)
Boundary conditions (10/21)

The next slides show the boundary conditions used in the example case.

Firstly, however, a small side note about Regular Expressions:

• These are search patterns that OpenFOAM supports in several dictionary files.

• For example:
  • “(backWall|bottomWall|sideWall)” → refers to the 3 patch names backWall, bottomWall and sideWall.
  • “procBoundary.*” → refers to all patch names that start with “procBoundary”.

For more details: [en.wikipedia.org/wiki/Regular_expression](http://en.wikipedia.org/wiki/Regular_expression)

Boundary conditions (11/21)

Inlet group (1/2):

U:
```plaintext
inlet
{
    type flowRateInletVelocity;
    massFlowRate 375;
    rho rho;
    rhoInlet 999.0;
}
```

alpha.water:
```plaintext
inlet
{
    type fixedValue;
    value uniform 1.0;
}
```

p_rgh:
```plaintext
inlet
{
    type fixedFluxPressure;
    value uniform 0.0;
}
```

nut:
```plaintext
inlet
{
    type calculated;
    value uniform 0.0;
}
```
Boundary conditions (12/21)

Inlet group (2/2):

**epsilon:**
inlet
{
  type turbulentMixingLengthDissipationRateInlet;
  mixingLength 0.2;
  value uniform 1.665138;
}

**k:**
inlet
{
  type turbulentIntensityKineticEnergyInlet;
  intensity 0.5;
  value uniform 3.778352;
}

Boundary conditions (13/21)

Outlet group:

**U:**
outlet
{
  type pressureInletOutletVelocity;
  value uniform (0.0 0.0 0.0);
}

**alpha.water:**
outlet
{
  type inletOutlet;
  inletValue uniform 0;
  value uniform 0;
}

**nut:**
outlet
{
  type calculated;
  value uniform 0.0;
}

**p_rgh:**
outlet
{
  type fixedValue;
  value uniform 0.0;
}

**k, epsilon:**
outlet
{
  type zeroGradient;
}
Boundary conditions (14/21)

Wall group (1/2):

**U:**

```
"(backWall|bottomWall|sideWall)"
{
  type fixedValue;
  value uniform (0.0 0.0 0.0);
}
```

**p_rgh:**

```
"(backWall|bottomWall|sideWall)"
{
  type fixedFluxPressure;
  value uniform 0.0;
}
```

**alpha.water:**

```
"(backWall|bottomWall|sideWall)"
{
  type zeroGradient;
}
```

**nut:**

```
"(backWall|bottomWall|sideWall)"
{
  type nutkWallFunction;
  value uniform 0.0;
}
```

Boundary conditions (15/21)

Wall group (2/2):

**epsilon:**

```
"(backWall|bottomWall|sideWall)"
{
  type epsilonWallFunction;
  value uniform 1.665138;
}
```

**k:**

```
"(backWall|bottomWall|sideWall)"
{
  type kqRWallFunction;
  value uniform 3.778352;
}
```
Boundary conditions (16/21)

Symmetry group, for all 6 fields:

```
symmetry
{
    type symmetry;
}
```

Special group, interfaces between domains, in all 6 fields:

```
"procBoundary.*"
{
    type processor;
    value uniform init_value_or_vector;
}
```

Boundary conditions (17/21)

**Manipulation of fields and domains (1/2):**

This was not used in our example case but the idea is simple:

What if we need to initialize a part of the internal field and/or fixed value patches with a value specific only to a group of cells or faces?

This is where `setFields` comes into play. This utility will use the settings given in the dictionary file “system/setFieldsDict”, for assigning values to each desired field.

Example in the next slide.
Boundary conditions (18/21)

Manipulation of fields and domains (2/2):

defaultFieldValues
(
  volScalarFieldValue alpha.water 0
);

regions
(
  // Set cell values
  // (does zerogradient on boundaries)
  boxToCell
  
  box (-2.0 -2.0 -1) (11.0 1.0 0.2);
  fieldValues
  
  volScalarFieldValue alpha.water 1
  
);  // Set patch values (using ==)

boxToFace
{
  box (-2.0 -2.0 -1) (11.0 1.0 0.2);
  fieldValues
  
  volScalarFieldValue alpha.water 1

)

Note: The selection box is meant to include the cell centres and/or face centres, for selecting the respective cells and faces.

Boundary conditions (19/21)

Turbulence Models (1/3):

Two categories of files have to be taken into account:

• “constant/turbulenceProperties” – for defining the major group of turbulence modelling to be used and the respective options settings for that model.

• In the time folders, we then have the fields associated to the turbulence model we want to use.

  • Example in our “0.orig” folder are: k, epsilon and nut
Boundary conditions (20/21)

Turbulence Models (2/3):

Content of “constant/turbulenceProperties”:

simulationType RAS;
RAS
{
  RASModel kEpsilon;
  turbulence on;
  printCoeffs on;
}

Want laminar flow modelling?

simulationType laminar;

Boundary conditions (21/21)

Turbulence Models (3/3):

The last critical detail for turbulence models is:

What initial values should we use and what values at the inlets?

This depends on your simulation, but a few guidelines exist, e.g. online:

- [www.cfd-online.com/Wiki/Turbulence_free-stream_boundary_conditions](http://www.cfd-online.com/Wiki/Turbulence_free-stream_boundary_conditions)
- [support.esi-cfd.com/esi-users/turb_parameters/](http://support.esi-cfd.com/esi-users/turb_parameters/)

The bottom line is that you will have to test which values are suited to your simulation.
Convective term discretization (1/4)

Convective term refers to the divergence operator $\nabla \cdot$.

The OpenFOAM Programmer’s Guide goes into more details about this in the subsection “2.4.2 The convection term”, in which the following expression can be found:

\[
\int_V \nabla \cdot (\rho U \phi) \, dV = \int_S \mathbf{n} \cdot (\rho U \phi) = \sum_f S_f \cdot (\rho U) \phi_f = \sum_f F \phi_f
\]

Which shows how the convection term is integrated and linearized. We will be addressing on this topic how we can control this term.

Convective term discretization (2/4)

The settings for the divergence schemes in the file “system/fvSchemes”, namely in the block “divSchemes”.

From our example case:

```plaintext
divSchemes
{
    default none;
    div(rhoPhi,U) Gauss upwind;
    div(phi, alpha) Gauss upwind;
    div(phirb, alpha) Gauss upwind;
    div(phi, k) Gauss upwind;
    div(phi, epsilon) Gauss upwind;
    div((muEff*dev(T(grad(U)))) Gauss linear;
}
```

This is currently defined to be mostly of first order discretization, i.e. upwind.
Convective term discretization (3/4)

How do we know which terms we need?

We either:

1. Let the solver complain when they are not present.
2. Or we look at the source code. For example:

   cat $FOAM_SOLVERS/incompressible/simpleFoam/UEqn.H

we can see this:

   // Momentum predictor
   tmp<fvVectorMatrix> UEqn
   {
     fvm::div(phi, U)
     + turbulence->divDevReff(U)
     ==
     fvOptions(U)
   };

   Refers to this entry:
   div(phi,U)

Convective term discretization (4/4)

How can we increase accuracy in our example case?

- Use an intermediate scheme between linear and upwind:
  
  div(rhoPhi,U) Gauss linearUpwind grad(U);

- Special limiter, great for VOF-related fields:
  
  div(phi,alpha) Gauss vanLeer;

- Second order scheme:
  
  div(phirb,alpha) Gauss linear;

- Problems with running in parallel with linearUpwind?
  
  div(phi,epsilon) Gauss linearUpwind limitedGrad;

along with the gradSchemes block having this entry:

  limitedGrad cellLimited Gauss linear 1;
Diffusive term discretization (1/4)

Diffusive term refers to the Laplace operator $\nabla^2$

The OpenFOAM Programmer’s Guide goes into more details about this in the subsection “2.4.1 The Laplacian term”, in which the following expression can be found:

$$
\int_V \nabla \cdot (\Gamma \nabla \phi) \, dV = \int_S \nabla \cdot (\Gamma \nabla \phi) = \sum_f \Gamma_f s_f \cdot (\nabla \phi)_f
$$

Which shows how the diffusion term is integrated and linearized.

We will be addressing on this topic how we can control this term.

Diffusive term discretization (2/4)

The settings for the Laplacian schemes in the file “system/fvSchemes”, in the block “laplacianSchemes”.

From our example case:

```cpp
laplacianSchemes
{
    default Gauss linear corrected;
}
```

Meaning:

• The same setting is used for all Laplacian terms.
• Second order or above should always be used.
• corrected option is the surface normal gradient scheme to be used.
Diffusive term discretization (3/4)

Other examples from OpenFOAM's tutorials:

- Gauss linear limited corrected 0.5;
- Gauss linear limited corrected 0.333;

These usually depend on how orthogonal or non-orthogonal the mesh cells are.

- Gauss linear \textit{orthogonal};
- Gauss linear \textit{corrected};
- Gauss linear \textit{uncorrected};

Diffusive term discretization (4/4)

Laplacian schemes depend on the surface normal gradient discretization (block “snGradSchemes”).

Therefore we have to take into account how the normal of a face relates to the centers of the cells that share said face.

Specifically, an orthogonal mesh has cell centers aligned with the centers and normals of the faces shared by those cells.
The end result of the discretization process are linear systems of equations:

$$Ax = b$$

Where:
- the bold forms designate tensor quantities,
- uppercase letters stand for matrices,
- lowercase for vectors.

These equations will appear for every conservation law and at every outer iteration. Depending on the cases, their approximate solution can easily reach 75% of the overall CPU time.

In order to configure the solver entry for each named block, we need to assess the type of equation that will be discretised and that will give rise the matrix form $Ax = b$, where:

- $A$ is the coefficient matrix that correlates the values between the centres, i.e., our unknowns;
- $x$ is the vector that represents the values at the cell centres for which we are solving the linear system;
- $b$ keeps the source terms for each respective cell.
Linear system solvers (3/35)

As a result of this construct, the following types of equations will exist and the respective matrix solvers should be used:

- The equation for the pressure field, i.e., continuity equation, gives rise to a symmetric matrix, hence it should use solvers devised for this type of matrices.
- All other equations give rise to a usually non-symmetric matrix due to convection, which is why we cannot use solvers that are meant for symmetric matrix equations.

Linear system solvers (4/35)

Symmetric $A$ matrix

\[
\begin{bmatrix}
  i & kk & 0 & \cdots & 0 & 0 & 0 \\
  kk & j & hh & \cdots & 0 & 0 & 0 \\
  0 & hh & k & \ddots & \cdots & 0 & 0 \\
  \vdots & \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\
  0 & 0 & \ddots & \ddots & gg & 0 & \\
  0 & 0 & 0 & \cdots & gg & r & ll \\
  0 & 0 & 0 & \cdots & 0 & ll & s
\end{bmatrix}
\]

Asymmetric $A$ matrix

\[
\begin{bmatrix}
  i & hj & qw & \cdots & 0 & 0 & 0 \\
  kz & j & kq & \cdots & 0 & 0 & 0 \\
  we & le & k & \ddots & \cdots & 0 & 0 \\
  \vdots & \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\
  0 & 0 & \ddots & \ddots & \ddots & \ddots & ae & 0 \\
  0 & 0 & 0 & \cdots & ax & r & qu \\
  0 & 0 & 0 & \cdots & 0 & lw & s
\end{bmatrix}
\]
Linear system solvers (5/35)

The file “system/fvSolution” must be used to select whatever method is found appropriate to solve.

Here are a few expressions that we will be using:

- **outer iteration** – this usually refers to one step in time, if transient, or to a sweep of all transport equations which are being solved.

- **matrix solver iteration** – In the next slide you will see at the end of each line the information “No Iterations”, which refers to the number of iterations it took to solve the respective linear system of equations.

**Note:** More details are available in section "4.5 Solution and algorithm control" in the OpenFOAM User Guide

Linear system solvers (6/35)

This is an example output for an **outer iteration**:

```
Time = 2
DILUPBiCG: Solving for Ux, Initial residual = 0.33783062, Final residual = 0.022319355, No Iterations 2
DILUPBiCG: Solving for Uy, Initial residual = 0.16839243, Final residual = 0.0037979544, No Iterations 2
DILUPBiCG: Solving for Uz, Initial residual = 0.17283387, Final residual = 0.016074394, No Iterations 1
DILUPBiCG: Solving for h, Initial residual = 0.97900298, Final residual = 0.020167345, No Iterations 1
GAMG: Solving for p, Initial residual = 0.90628728, Final residual = 0.0041221651, No Iterations 12
time step continuity errors : sum local = 0.00046531931, global = 1.4428116e-006, cumulative = -3.528428e-006
rho max/min : 1.1274839 0.72937754
DILUPBiCG: Solving for epsilon, Initial residual = 0.99931859, Final residual = 4.869335e-005, No Iterations 1
DILUPBiCG: Solving for k, Initial residual = 0.82482828, Final residual = 9.5040752e-006, No Iterations 1
ExecutionTime = 2.509 s ClockTime = 2 s
```

Zooming-in on one equation:

```
DILUPBiCG: Solving for Ux, Initial residual = 0.33783062, Final residual = 0.022319355, No Iterations 2
```
Linear system solvers (7/35)

system/fvSolution:
This dictionary file was designed to handle the settings for the linear equation solvers and the algorithms to be used by a solver application, e.g. interFoam.

Starting with the linear equation solvers, these are configured inside this block list:

```plaintext
solvers
{
    ...
}
```

It’s in here that we will be configuring the matrix solvers.

---

Linear system solvers (8/35)

Starting with our example case, for configuring the linear equation solvers for the fields named “alpha.water”, we are using the following settings:

```plaintext
"alpha.water.*"
{
    nAlphaCorr 2;
    nAlphaSubCycles 1;
    cAlpha 1;
    MULESCorr yes;
    nLimiterIter 3;
    solver smoothSolver;
    smoother symGaussSeidel;
    tolerance 1e-6;
    relTol 0;
}
```

Specific for the linear equations related to the phase fraction equations.
Linear system solvers (9/35)

Keep in mind that each solver has its own settings, where the first tier of possible options for a matrix solver refers to one of two major possible types of settings:

• **preconditioner** is needed for solvers that rely on a preconditioning strategy to speed up their iterative process.
  
  • For more details on what preconditioning is, see: en.wikipedia.org/wiki/Preconditioner

• **smoother** is designed to smooth-out numerical issues that usually arise from ill-formed matrices and strongly uneven intermediate solutions for the matrix equation.


Linear system solvers (10/35)

There are 3 other parameters that are common to most of the matrix solvers. Let us look at an example output from an outer iteration:

DILUPBiCG: Solving for Ux, Initial residual = 0.33783062, Final residual = 0.022319355, No Iterations 2

The residual is essentially the result from this expression:

\[ \text{residual} = \text{sum}(\text{abs}(b - Ax)) \]

The reported residual values are normalized values from this equation, in order to keep values between 0.0 and 1.0 for an easier interpretation of how good or bad the residuals are.
Linear system solvers (11/35)

The 3 major parameters we need to control:

- **tolerance** – this is the minimum residual value we want to achieve at the end of the iterations of the matrix solver. In other words, if the Final Residual falls below this value, the matrix solver stops iterating. Default value is $1e^{-6}$.

- **relTol** – this relative tolerance refers to whether the residual for the current iteration is lesser than **relTol** times the Initial Residual. Default value is $0.0$ (= off).

- **maxIter** – maximum number of iterations for the matrix solver to perform, regardless of the convergence status. Anti-infinite loop counter-measure. Default value is $1000$.

One tolerance will suffice to stop the matrix solver:

- In order to only define the relative tolerance:
  
  ```
tolerance 0.0;
relTol 0.01;
  ```

- In order to only define the (absolute) tolerance:
  
  ```
tolerance 1e-06;
relTol 0.0;
  ```

- In order to allow the maximum number of iterations to be reached:
  
  ```
tolerance 0.0;
relTol 0.0;
  ```
Linear system solvers (13/35)

Which values should you use?
It all depends on:
• the problem you are solving;
• how accurate you want it to be, while weighing:
  • it is usually never possible to reach the exact solution for a matrix equation...
  • and even if it is, the solution might be useless if an outer iteration is still needed for balancing the results over all equations.

Therefore, this is usually something that can be adjusted after reaching good solutions for your cases.

Linear system solvers (14/35)

A few good reference values are as follows:
• For the pressure field, make sure you have a tighter control, such as:
  
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>tolerance</td>
<td>1e-06</td>
</tr>
<tr>
<td>relTol</td>
<td>0.001</td>
</tr>
<tr>
<td>maxIter</td>
<td>250</td>
</tr>
</tbody>
</table>

• For all other equations, you can loosen up a bit the control, since most other equations will be affected by the pressure field in the next major iteration:
  
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>tolerance</td>
<td>1e-05</td>
</tr>
<tr>
<td>relTol</td>
<td>0.01</td>
</tr>
<tr>
<td>maxIter</td>
<td>100</td>
</tr>
</tbody>
</table>
Linear system solvers (15/35)

Which matrix solvers should we use and in which situations? The answer strongly depends:

• on the simulation being performed;
• on whether the run in serial or in parallel.

Therefore, don't assume that there is a fool proof way of selecting the solvers and respective preconditioners or smoothers.

Nonetheless, it is possible to present some guidelines:

1. When in doubt, use the values in the tutorials which are the most similar to your problem.

(continues…)

Linear system solvers (16/35)

2. If you need the matrix solving steps to be as fast/efficient as possible, a good combination is to use:

• For the pressure equations, use “GAMG”. Example:

```cpp
p {
    solver GAMG;
    smoother GaussSeidel;
    cacheAgglomeration true;
    nCellsInCoarsestLevel 10;
    agglomerator faceAreaPair;
    mergeLevels 1;
    tolerance 1e-06;
    relTol 0.001;
    maxIter 250;
}
```

(continues…)
Linear system solvers (17/35)

- GAMG can be somewhere 2 and 5 times faster than using PCG+DIC.
- There is a downside to using GAMG: it is only efficient enough if you properly calibrate its parameters. For example:
  - “nCellsInCoarsestLevel” can depend on the number of cells your case has, but make sure to test the values for a few major iterations first, before using the value in a real simulation scenario.
- The PBiCG solver is usually more efficient for running in parallel for all other equations (non-pressure). Although in some cases, GAMG is faster than PBiCG.

(continues…)

Linear system solvers (18/35)

3. The PCG solver can in many cases be better than GAMG for the pressure equations, therefore, you should always double-check which one is best for your simulation.

4. In some cases, the FDIC preconditioner may prove to be more efficient than DIC and give results faster (symmetric matrices only).

5. GAMG can also be used as a preconditioner for working cooperatively with a matrix solver, but isn’t very common. But again, it can be and should be tested for your own cases.

(continues…)
Linear system solvers (19/35)

6. A choice of smoother (specific matrix solvers only), may depend strongly on your case. A few examples:
   - GaussSeidel is commonly used in conjunction with GAMG, since it can offer a direct resolution for each major block.
   - DIC and DILU can also be used as smoothers, but they can prove to be more efficient if used in conjunction with GaussSeidel, namely by using the variants DICGaussSeidel and DILUGaussSeidel.

(continues…)

Linear system solvers (20/35)

7. “smoothSolver” as a matrix solver can prove to be more efficient, if a preconditioner has issues due to numerical spikes. A smoother will instead try to solve the equation directly, while sort-of not fretting over imperfections in the achieved solutions. Configuration example:

```java
U{
    solver smoothSolver;
    smoother GaussSeidel;
    tolerance 1e-8;
    relTol 0.1;
    nSweeps 1;
}
```
Linear system solvers (21/35)

Continuing with the settings we have on our example case:

```plaintext
pcorr
{
    solver        GAMG;
tolerance      1e-5;
relTol          0.001;
smoother       GaussSeidel;
nPreSweeps     0;
nPostSweeps    2;
cacheAgglomeration on;
agglomerator   faceAreaPair;
nCellsInCoarsestLevel 10;
mergeLevels    1;
}
```

```plaintext
$p_rgh$
{
    tolerance      1e-07;
relTol          0.05;
}
```

```plaintext
$p_{rghFinal}$
{
    $pcorr$;
    relTol          0;
}
```

Linear system solvers (22/35)

And for all of the other fields:

```plaintext
"(U|k|epsilon).*"
{
    solver        smoothSolver;
    smoother      symGaussSeidel;
tolerance      1e-06;
relTol          0;
minIter         1;
}
```
Algorithm configurations

The most common algorithms implemented in OpenFOAM:

- PISO – Pressure-Implicit Split-Operator
- SIMPLE – Semi-Implicit Method for Pressure-Linked Equations
- PIMPLE – it’s a PISO-SIMPLE hybrid

In our case example, we use interFoam with these settings:

```plaintext
PIMPLE
{
    momentumPredictor no;
    nOuterCorrectors 1;
    nCorrectors 3;
    nNonOrthogonalCorrectors 1;
}
```

In general, the algorithms have one or more of the following parameters:

```plaintext
nOuterCorrectors 0;
nCorrectors 0;
nNonOrthogonalCorrectors 0;
turbOnFinalIterOnly no;
momentumPredictor yes;
transonic no;
residualControl
{
    p 1e-2;
    U 1e-3;
    "(k|epsilon|omega)" 1e-3;
}
pRefCell 0;
pRefPoint (0 0 0);
pRefValue 0;
```
Linear system solvers (25/35)

Description for each parameter:
• `nOuterCorrectors` – number of iterations that should be used for the external loop of the algorithm (not to be confused with "outer loop iteration" we've been referring to for the time step).
• `nCorrectors` – number of iterations that should be used for the internal loop of the algorithm.
• `nNonOrthogonalCorrectors` – number of iterations for attempting to correct the effect that non-orthogonal cells have on the solution of the problem. Rule of thumb:
  • 0 for a fully orthogonal mesh;
  • 20 iterations for the most non-orthogonal meshes;
  • 1-3 iterations if there are a few non-orthogonal cells.

Linear system solvers (26/35)

• `turbOnFinalIterOnly` – this flag allows us to postpone the calculation of the turbulence fields to the last iteration.
• `momentumPredictor` – Not all solvers use this parameter. Those that do support this parameter, will not solve the momentum equation ($U$) if this parameter is set to “no”.
• `transonic` – Only solvers that have implementations for sonic flow will support this flag.
• `residualControl` - This is a named block gives the ability to add an additional stopping criteria for the Initial Residual of each one of the listed equations.
Linear system solvers (27/35)

*Side note*

The following three parameters fall in a new topic: having a location in the domain with a fixed reference pressure value.

This is necessary whenever all of pressure boundary conditions do not have a fixed value, i.e. if they are all defined as zero gradient or a similar boundary condition.

For such a situation, we must avoid having an incomplete definition of the pressure equation, therefore we can rely on the following parameters for defining a specific location in the mesh that has a fixed and pre-defined value of pressure.

- **pRefCell** or **pRefPoint** - For selecting a location in the mesh where the cell centre is used for pressure reference. Keep in mind that:
  - **pRefCell** refers to the cell ID on the mesh.
  - **pRefPoint** ensures us that the provided position is used for selecting the cell.
  - **Note**: **pRefCell** takes precedence over **pRefPoint**.
- **pRefValue** - This is the pressure value, which should be defined with the same units as the pressure fields.
Linear system solvers (29/35)

Almost complete example for SIMPLE:

SIMPLE
{

    nNonOrthogonalCorrectors 0;

    momentumPredictor yes;
    transonic no;

    residualControl
    {
        p 1e-2;
        U 1e-3;
        "(k|epsilon|omega)" 1e-3;
    }

    //pRefCell 0;
    pRefPoint (0 0 0);
    pRefValue 0;
}

Depends on the solver

Linear system solvers (30/35)

Almost complete example for PISO:

PISO
{

    nCorrectors 2;
    nNonOrthogonalCorrectors 0;

    momentumPredictor yes;

    pRefCell 0;
    pRefPoint (0 0 0);
    pRefValue 0;
}

Depends on the solver
Linear system solvers (31/35)

Almost complete example for PIMPLE:

```
PIMPLE
{
    nOuterCorrectors 1;
    nCorrectors 2;
    nNonOrthogonalCorrectors 0;
    turbOnFinalIterOnly no;
    momentumPredictor yes;
    transonic no;
    residualControl
    {
        p 1e-2;
        U 1e-3;
        "(k|epsilon|omega)" 1e-3;
    }  //pRefCell 0;
    pRefPoint (0 0 0);
    pRefValue 0;
}
```

Relaxation factors

Technically, OpenFOAM uses under-relaxation factors, because the values are between 0.0 and 1.0.

These help the outer iterative convergence process, making it more robust while simultaneously increasing the likelihood that we can reach a numerical solution for our problem. A poor choice of values can delay and even prevent convergence.

This is further explained in the OpenFOAM User Guide, subsection “4.5.2 Solution under-relaxation”.

Linear system solvers (32/35)
Linear system solvers (33/35)

1. If the value 0.0 is given, then the solution is kept unchanged between each outer iteration.

2. If 1.0 is given, then the under-relaxation does not take place at all.

3. As we reduce the factor from 1.0 towards 0.0, we increase the impact of the under-relaxation. Examples:
   a) 0.9 – 90% of current solution of outer iteration is preserved and 10% previous outer iteration.
   b) 0.1 – the current solution has a very small impact in the final solution after this relaxation step, making the flow results to evolve slower with every outer iteration.

Linear system solvers (34/35)

Relaxation factors: transient vs steady-state

• In most cases, the relaxation factors for transient simulations are either simply set to 1.0 or not at all.

• This is because PISO and PIMPLE algorithms perform time accurate simulations, where the time step essentially does what the relaxation factor is used for in steady-state simulations.

• Nonetheless, PIMPLE is a hybrid algorithm, therefore, it can also rely on the relaxation factors for the SIMPLE loop within the PIMPLE algorithm. Therefore, those relaxation factors are still applicable.
Linear system solvers (35/35)

From our example case it doesn’t need much for interFoam (PIMPLE):

```plaintext
relaxationFactors
{
    fields
    {
    }
    equations
    {
        ".*" 1;
    }
}
```

When compared to the tutorial “incompressible/simpleFoam/motorBike”:

```plaintext
relaxationFactors
{
    fields
    {
        p 0.3;
    }
    equations
    {
        U 0.7;
        k 0.7;
        omega 0.7;
    }
}
```

Parallel runs (1/9)

We will address the following topics:
1. Domain decomposition
2. Domain balancing
3. Running in parallel
4. Domain reconstruction
Parallel runs (2/9)

Domain decomposition (1/2)

Application: decomposePar

Dictionary: system/decomposeParDict

Relevant parameters:

- **numberOfSubdomains** is the number for sub-domains, i.e. how many processors for running in parallel.
- **method** is for choosing the algorithm for decomposing the domain. The easiest to use is the *scotch* option.
- **scotchCoeffs** is the block relative to the method *scotch*, which doesn’t even need be present, since its internal parameters are for advanced users.

Example of “system/decomposeParDict”:

```plaintext
method scotch;
scotchCoeffs {
    //writeGraph true;
    //strategy "b";
}
```

Example uses:

```
decomposePar
decomposePar -help
decomposePar -force
decomposePar -cellDist
decomposePar -noZero -fields -time 10
```

Parallel runs (3/9)

Domain decomposition (2/2)

Example of “system/decomposeParDict”:

```plaintext
method hierarchical;
hierarchicalCoeffs {
    n (1 2 1);
delta 0.001;
order xyz;
}
```

OR

```plaintext
method scotch;
scotchCoeffs {
    //writeGraph true;
    //strategy "b";
}
```
Parallel runs (4/9)

Domain balancing

There are essentially two types of domain balancing:

1. Complete sub-domain redistribution, by using `redistributePar`, which can help balance the number of cells per processor.
   - Requires “system/decomposeParDict”.
   - Can run in parallel.

2. Reordering the connections between cells, by using `renumberMesh`, which will improve the configuration of the equations in matrix form, for an optimum memory access.
   - Can run in parallel.

Parallel runs (5/9)

Running in parallel (1/4)

The common denominator is that the “-parallel” option must be used. For example, if we run this command:

```
simpleFoam -help
```

We will see this line:

```
-parallel run in parallel
```

Therefore, for running in parallel, the simplest command would be:

```
mpirun -np 2 simpleFoam -parallel
```

where the “-np” means that the number on the right is the number of processors to be used, i.e. 2.
Parallel runs (6/9)

Running in parallel (2/4)
The use of `mpirun` is not a standard on all platforms, e.g. clusters can use dedicated job schedulers and use dedicated scripts.

OpenFOAM has another two ways for running in parallel:

- `foamJob` is a script that comes in handy for running any utility and it has the ability to either run in serial or in parallel.
- `runParallel` is a function-script that is accessible only when we source the script `RunFunctions`. This is why this function is only seen inside the `Allrun` scripts that are present in OpenFOAM's tutorials.

Parallel runs (7/9)

Running in parallel (3/4) – Examples for `foamJob`:

Run in parallel as a background job:

```
foamJob -p simpleFoam
```

Run in parallel and show on-screen the output:

```
foamJob -p -s simpleFoam
```

For more details:

```
foamJob -help
```

**Note:** Application output is saved into the file named “log”.
Parallel runs (8/9)

Running in parallel (4/4) – Details for runParallel:

Will only work once this command is used (or similar):

```bash
source $WM_PROJECT_DIR/bin/tools/RunFunctions
```

commonly found in the Allrun scripts.

Usage structure:

```bash
runParallel app_name number_of_cores app_arguments
```

Example:

```bash
runParallel snappyHexMesh 4 -overwrite
```

Parallel runs (9/9)

Domain reconstruction

As mentioned before, there are two types of domain reconstruction:

1. When the mesh was generated in parallel, we need to reconstruct it with `reconstructParMesh`.

2. When the mesh is the same before and after decomposing/reconstructing, then `reconstructPar` should be used for reconstructing the time snapshots.

Both can only be executed in serial mode (not in parallel).

More details with the "-help" option, e.g.:

```bash
reconstructPar -help
```
Thank you for your time.

Next: 30 min break!