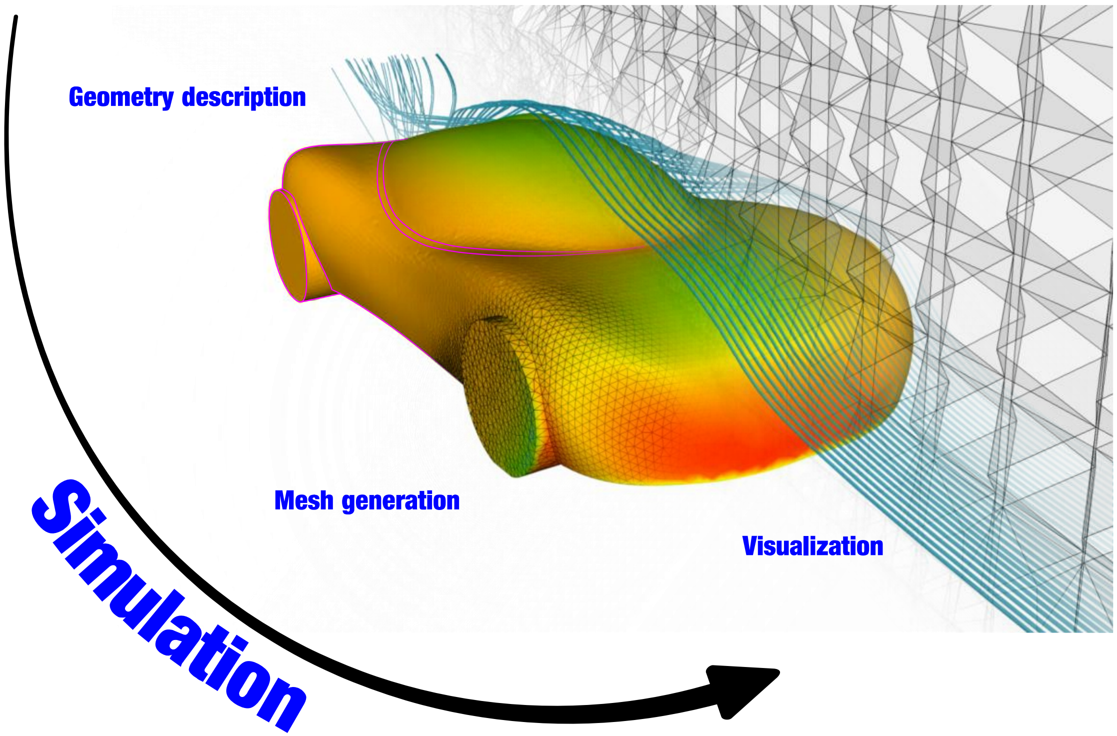
Module 3

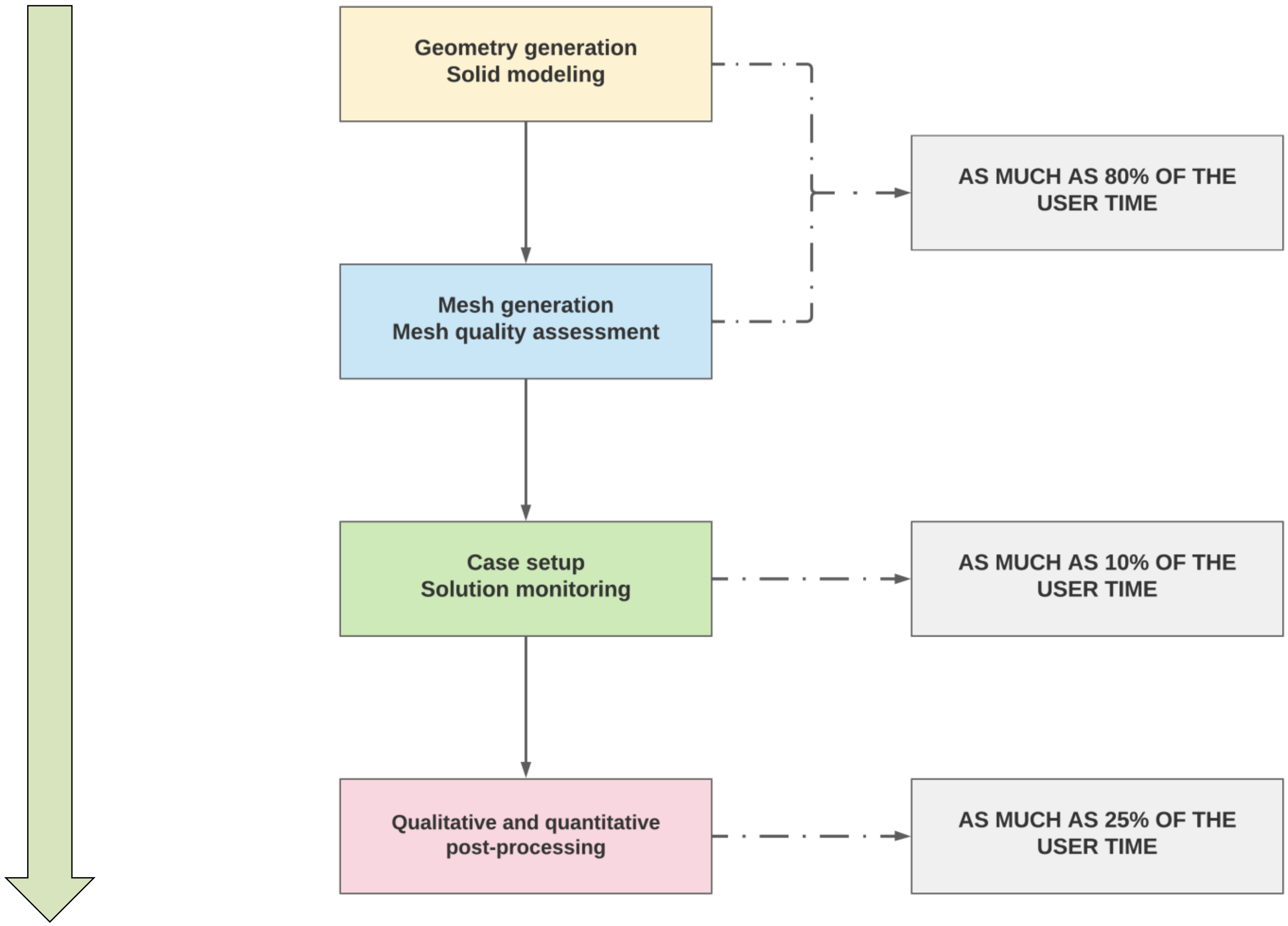
Meshing preliminaries – Mesh quality assessment – Meshing in OpenFOAM®

Before we begin



- The starting point of every CFD workflow is the geometry.
- Then we proceed to generate the mesh and assign the boundaries surface patches.
- Mesh quality and mesh size depend on the underlying geometry.
- And the quality and convergence rate of the solution highly depend on the mesh.
- So, try to do your best when generating the geometry and the mesh.
- After we have a valid mesh, we proceed to the case setup, and we launch/monitor the simulation.
- At the end, we do the post-processing (quantitative and qualitative).

Before we begin



- The percentages shown are based on personal experience.
- The percentages do not add to 100% because the overload changes from case to case.
- During this course we are going to address solid modeling and meshing.

Before we begin

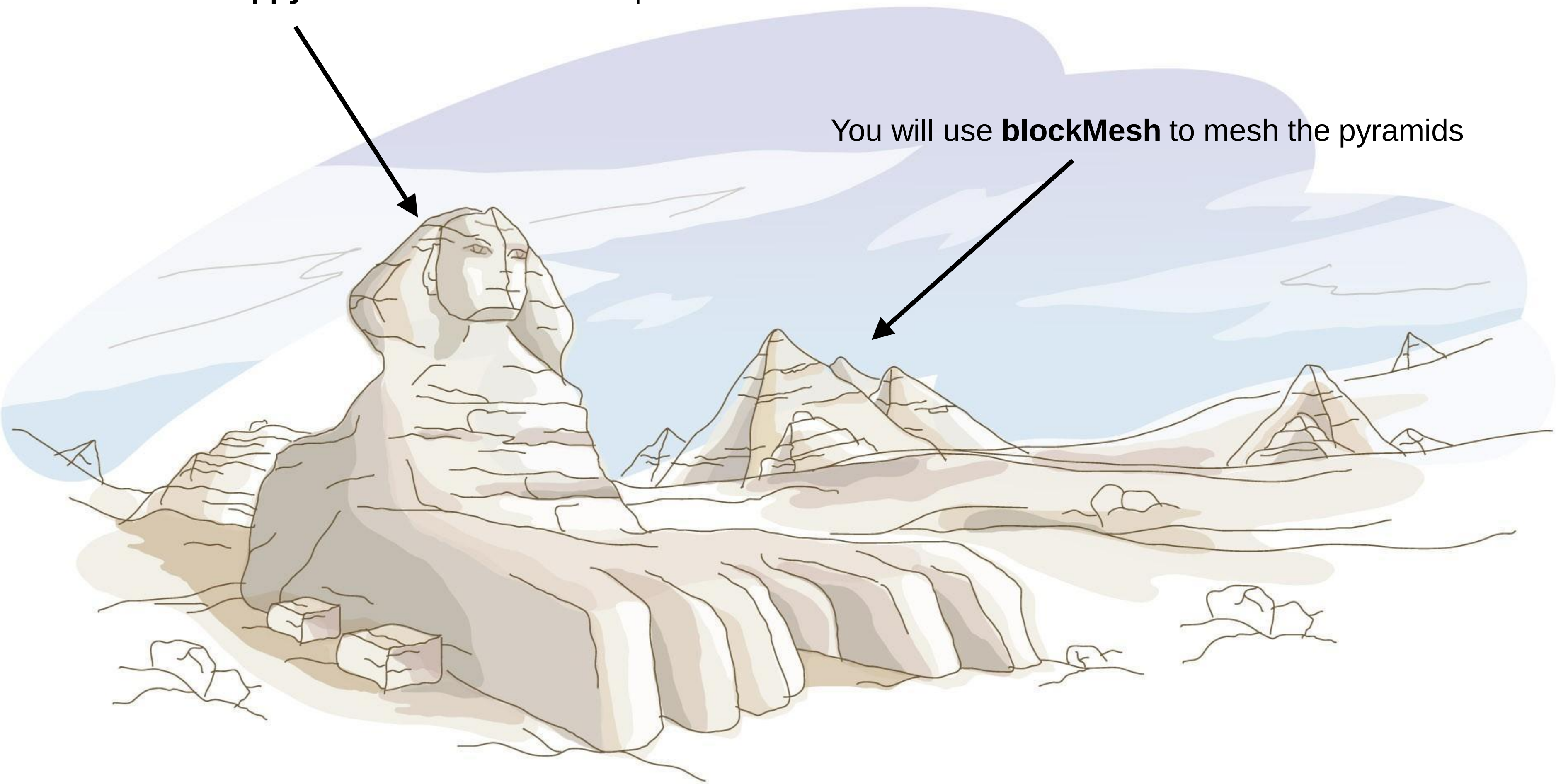
- OpenFOAM® comes with the following meshing applications:
  - `blockMesh`
  - `snappyHexMesh`
  - `foamyHexMesh`
  - `foamyQuadMesh`
- We are going to work with `blockMesh` and `snappyHexMesh`.
- **blockMesh** is a multi-block mesh generator.
- **snappyHexMesh** is an automatic split hex mesher, refines and snaps to surface.
- If you are not comfortable using OpenFOAM® meshing applications, you can use an external mesher.
- OpenFOAM® comes with many mesh conversion utilities. Many popular meshing formats are supported. To name a few: gambit, cfx, fluent, gmsh, ideas, netgen, plot3d, starccm, VTK.
- In this module, we are going to address how to mesh using OpenFOAM® technology, how to convert meshes to OpenFOAM® format, and how to assess mesh quality in OpenFOAM®.

Before we begin

By the end of this module, you will realize that

You will use **snappyHexMesh** to mesh the sphinx

You will use **blockMesh** to mesh the pyramids

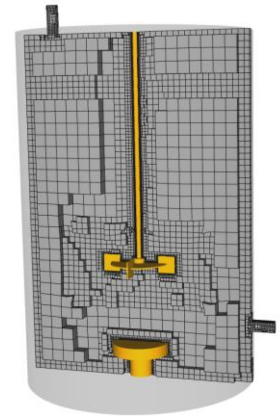


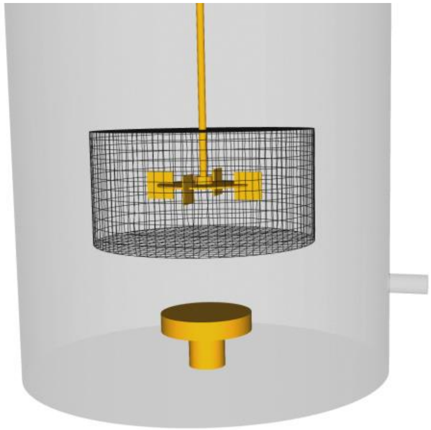
Roadmap

1. **Meshing preliminaries**
2. What is a good mesh?
3. Mesh quality assessment in OpenFOAM®
4. Mesh generation using blockMesh.
5. Mesh generation using snappyHexMesh.
6. snappyHexMesh guided tutorials.
7. Mesh conversion
8. Geometry and mesh manipulation utilities

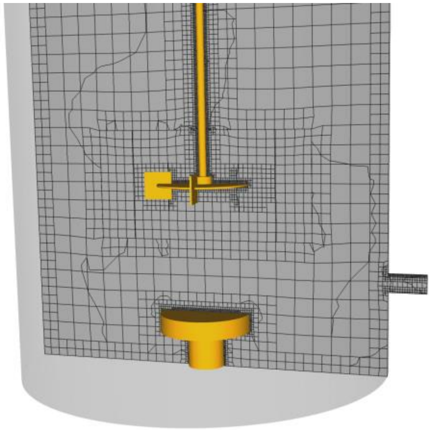
Meshing preliminaries

- Mesh generation or domain discretization consist in dividing the physical domain into a finite number of discrete regions, called control volumes or cells in which the solution is sought





www.wolfdynamics.com/wiki/moving/ani1.gif



www.wolfdynamics.com/wiki/moving/ani2.gif

Meshing preliminaries

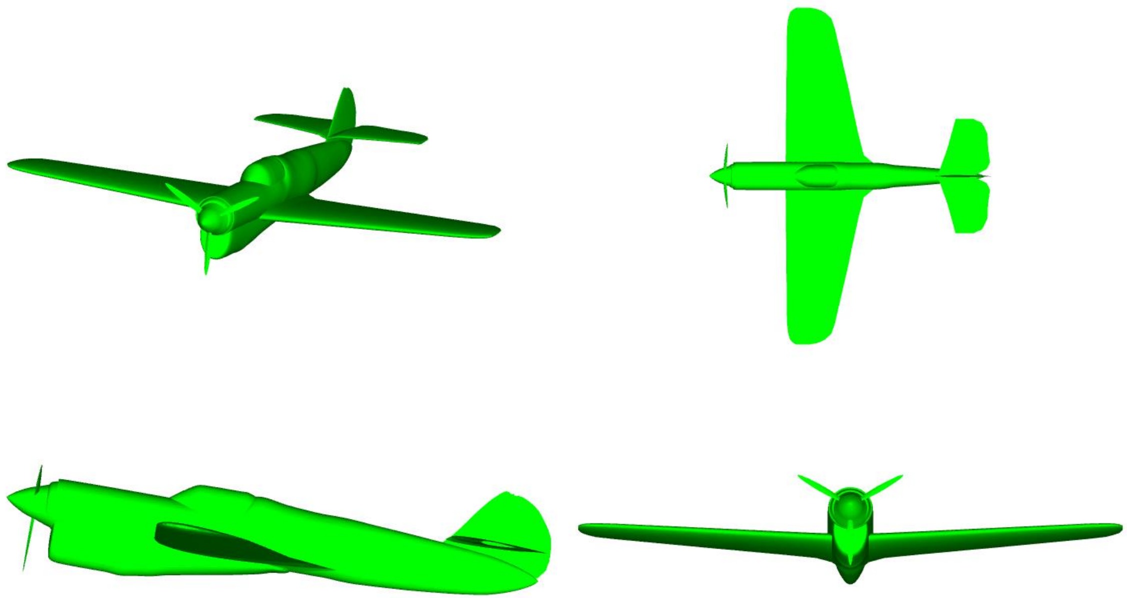
Mesh generation process

- Generally speaking, when generating the mesh, we follow these three simple steps:
  - **Geometry generation:** we first generate the geometry that we are going to feed into the meshing tool.
  - **Mesh generation:** the mesh can be internal or external. We also define surface and volume refinement regions. We can also add inflation layers to better resolve the boundary layer. During the mesh generation process we also check the mesh quality.
  - **Definition of boundary surfaces:** in this step we define physical surfaces where we are going to apply the boundary conditions. If you do not define these individual surfaces, you will have one single surface and it will not be possible to apply different boundary conditions.

Meshing preliminaries

Geometry generation - Input geometry

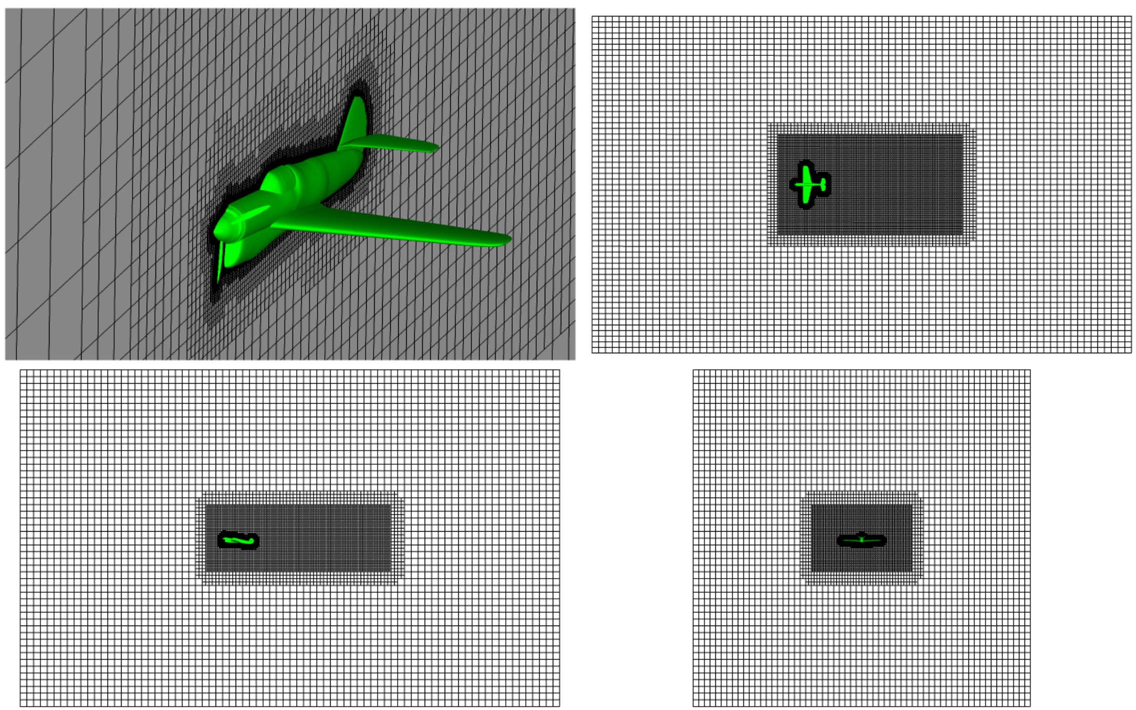
- The geometry must be watertight.
- Remember, the quality of the mesh and hence the quality of the solution greatly depends on the geometry. So always do your best when creating the geometry.



Meshing preliminaries

Mesh generation

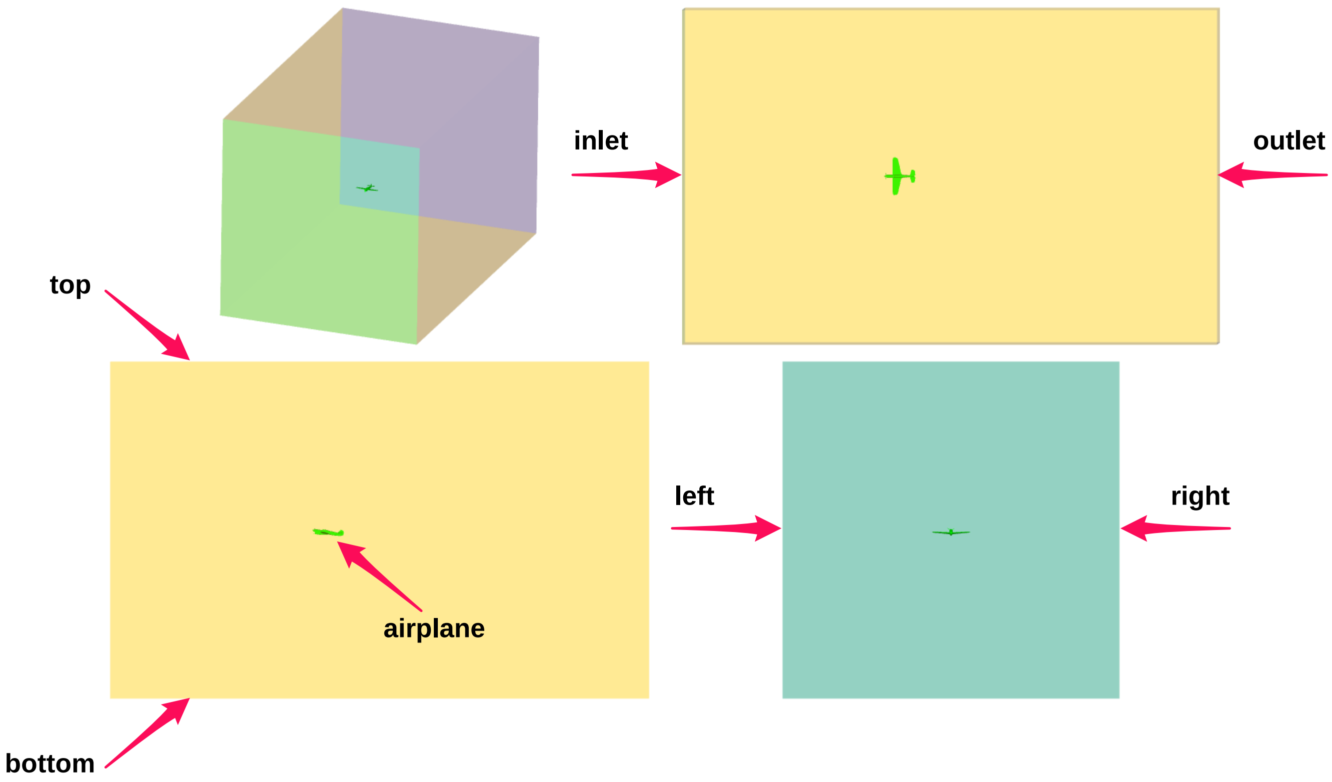
- If we are interested in external aerodynamics, we define a physical domain and we mesh the region around the body.
- If we are interested in internal aerodynamics, we simply mesh the internal volume of the geometry.
- To resolve better the flow features, we can add surface and volume refinement.
- always check the mesh quality.



Meshing preliminaries

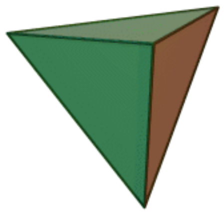
Definition of boundary surfaces (patches)

- In order to assign boundary conditions, we need to create boundary surfaces (patches) where we are going to apply the boundary values.
- The boundary surfaces (patches) are created at meshing time.
- In OpenFOAM®, you will find this information in the *boundary* dictionary file which is located in the directory **`constant/polyMesh`**. This dictionary is created automatically at meshing time.

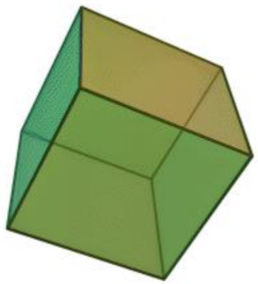


Meshing preliminaries

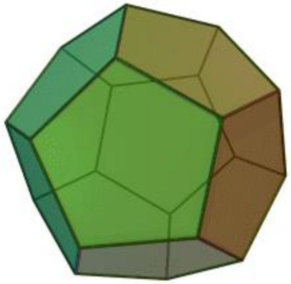
What cell type should I use?



http://www.wolfdynamics.com/wiki/cells/ani_tetra.gif



http://www.wolfdynamics.com/wiki/cells/ani_hexa.gif



http://www.wolfdynamics.com/wiki/cells/ani_poly.gif

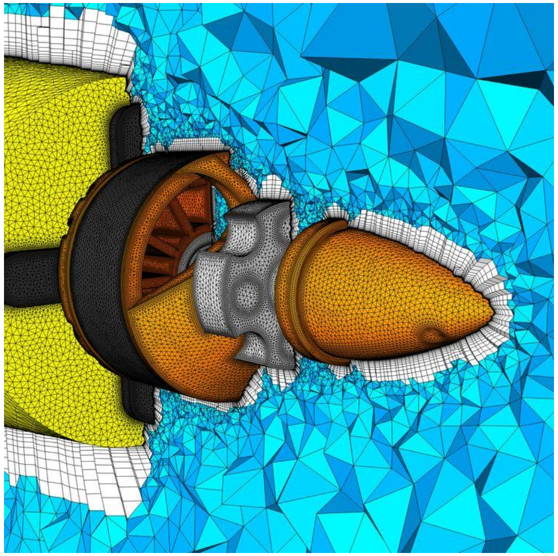
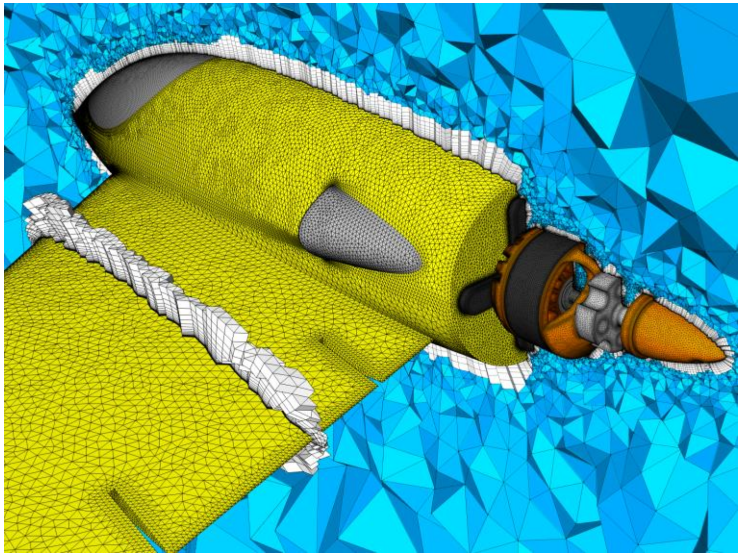
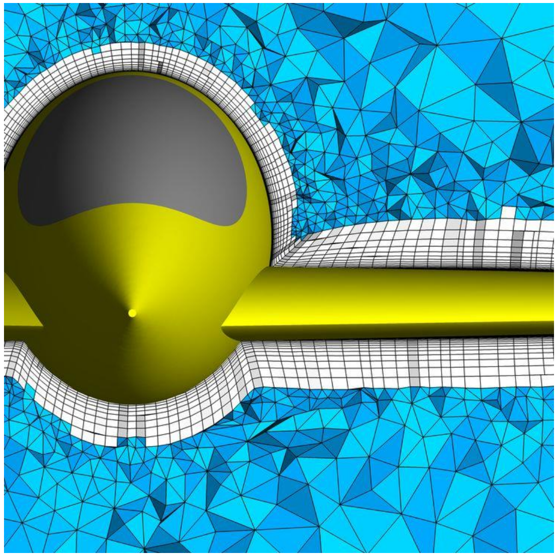
- In the meshing world, there are many cell types. Just to name a few: tetrahedrons, pyramids, hexahedrons, prisms, polyhedral.
- Each cell type has its very own properties when it comes to approximating the gradients and fluxes, we are going to talk about this later when we deal with the FVM.
- Generally speaking, hexahedral cells will give more accurate solutions under certain conditions.
- However, this does not mean that tetra/poly cells are not good.
- What cell type do I use? It is up to you; at the end of the day the overall quality of the final mesh should be acceptable, and your mesh should resolve the physics

Roadmap

1. ~~Meshing preliminaries~~
2. **What is a good mesh?**
3. Mesh quality assessment in OpenFOAM®
4. Mesh generation using blockMesh.
5. Mesh generation using snappyHexMesh.
6. snappyHexMesh guided tutorials.
7. Mesh conversion
8. Geometry and mesh manipulation utilities

What is a good mesh?

- There is no written theory when it comes to mesh generation and mesh quality assessment.
- Basically, the whole process depends on user experience and trial-and-error (it is an iterative process).
- A standard rule of thumb is that the elements shape and distribution should be pleasing to the eye.

22rd IMR Meshing Maestro Contest Winner
Travis Carrigan, John Chawner and Carolyn Woeber. Pointwise.
http://imr.sandia.gov/22imr/MeshingContest.html

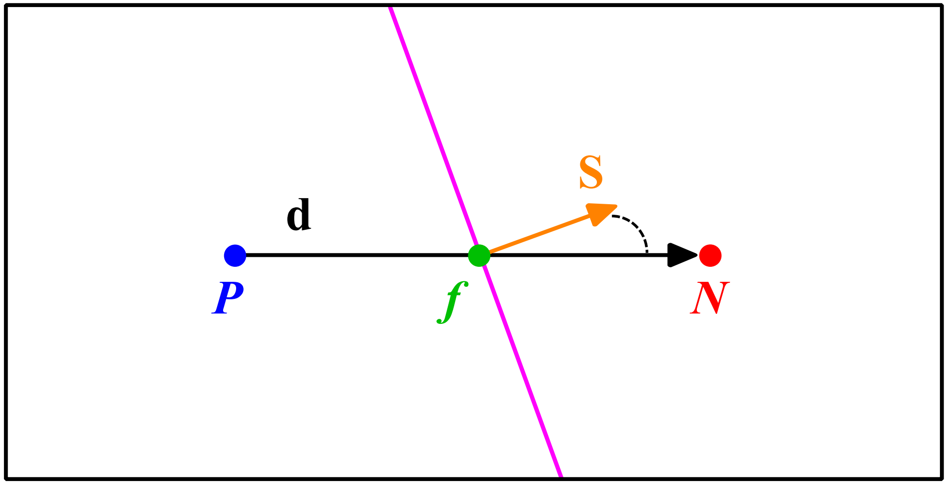
What is a good mesh?

- In a sounder way, the user can rely in mesh metrics.
- However, no single standard benchmark or metric exists that can effectively assess the quality of a mesh, but the user can rely on suggested best practices.
- Hereafter, we will present the most common mesh quality metrics:
    - Orthogonality.
    - Skewness.
    - Aspect Ratio.
    - Smoothness.
- After generating the mesh, we measure these quality metrics, and we use them to assess the goodness of the mesh.
- Have in mind that there are many more mesh quality metrics out there, and some of them are not very easy to interpret (*e.g.*, jacobian matrix, determinant, flatness, equivalence, condition number, and so on).
- <u>It seems that it is much easier diagnosing bad meshes than good meshes.</u>

What is a good mesh?

Mesh quality metrics. <u>Mesh orthogonality</u>

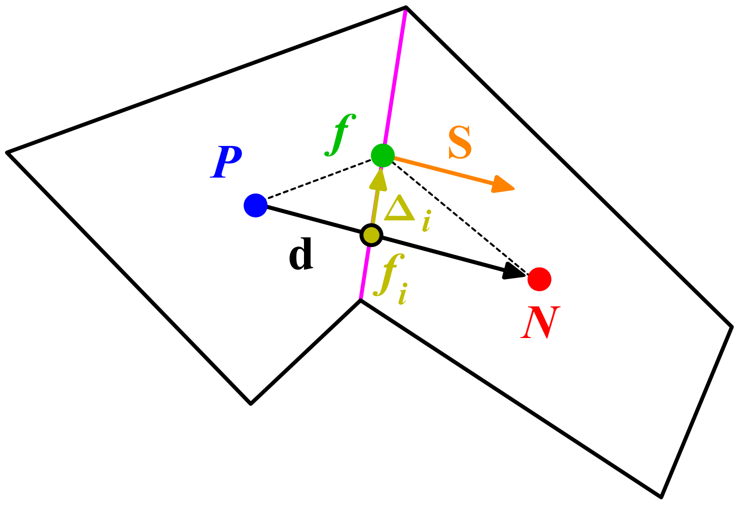
- Mesh orthogonality is the angular deviation of the vector **S** (located at the face center ***f***) from the vector **d** connecting the two cell centers **P** and **N**. In this case is 20°.
- It mainly affects the Laplacian (diffusive) terms and gradient terms at the face center ***f***.
- It adds numerical diffusion to the solution.



What is a good mesh?

Mesh quality metrics. <u>Mesh skewness</u>

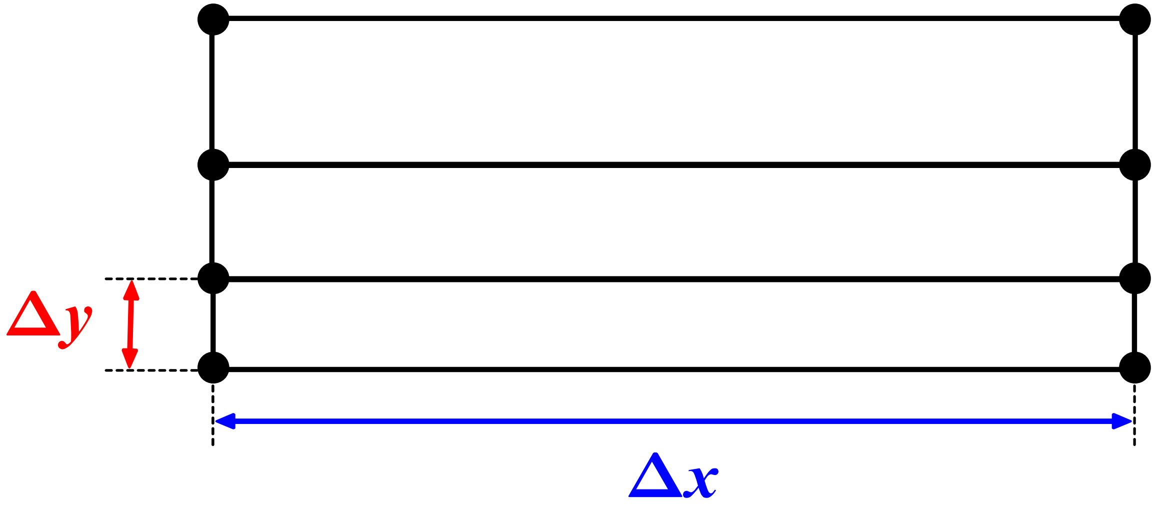
- Skewness (also known as non-conjunctionality) is the deviation of the vector **d** that connects the two cells **P** and **N**, from the face center ***f***.
- The deviation vector is represented with Δ and f_i is the point where the vector **d** intersects the face ***f*** .
- It affects the interpolation of the cell centered quantities at the face center ***f***.
- It affects the convective and diffusive terms (but to a lesser extend when compared to the orthogonality).
- It adds numerical diffusion and wiggles to the solution.



What is a good mesh?

Mesh quality metrics. <u>Mesh aspect ratio AR</u>

- Mesh aspect ratio AR is the ratio between the longest side Δx and the shortest side Δy.
- Large AR are ok if gradients in the largest direction are small.
- High AR smear gradients.
- Large AR add numerical diffusion to the solution.
- In RANS/URANS simulation large AR are acceptable.
- Instead, in SRS simulations (DES and LES), large AR can add too much numerical dissipation.

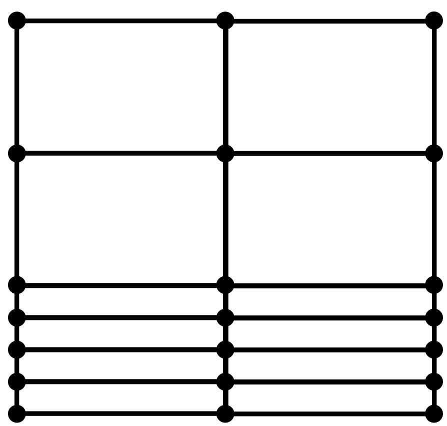


What is a good mesh?

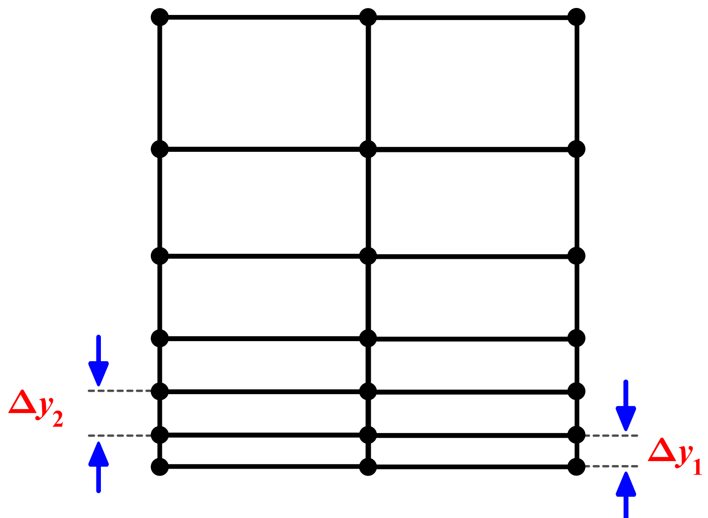
Mesh quality metrics. Smoothness

- Smoothness, also known as expansion rate, growth factor or uniformity, defines the transition in size between contiguous cells.
- Large transition ratios between cells add diffusion to the solution.
- Ideally, the maximum change in mesh spacing should be less than 20%:

$$\frac{\Delta y_2}{\Delta y_1} \leq 1.2$$



Steep transition

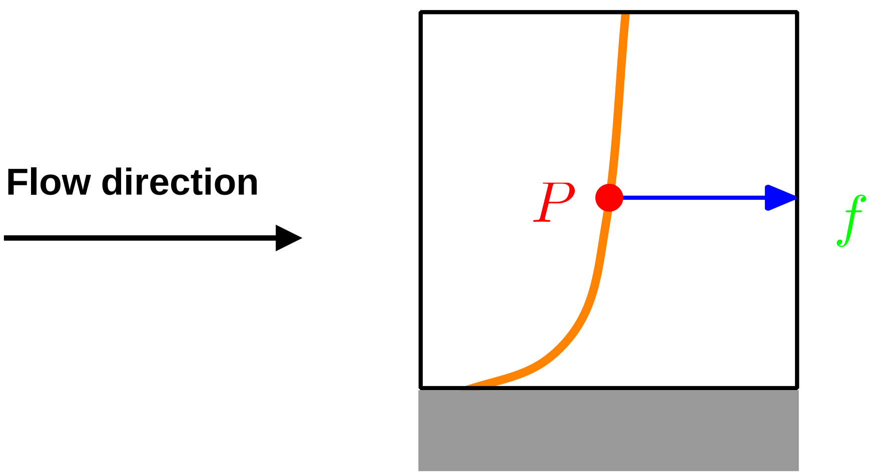


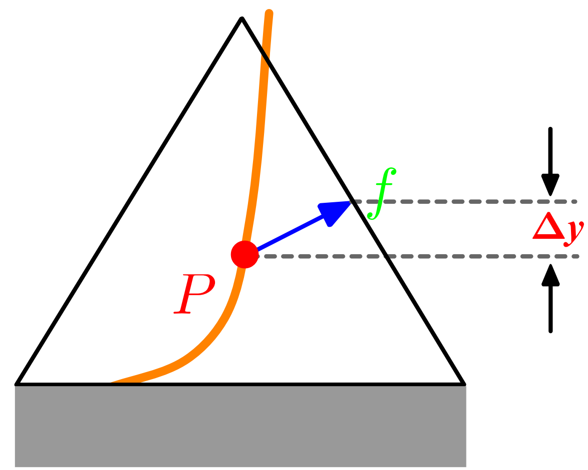
Smooth transition

What is a good mesh?

Mesh quality metrics. <u>Element type close to the walls - Cell/Flow alignment</u>

- Hexes, prisms, and quadrilaterals can be stretched easily to resolve boundary layers without losing quality.
- Triangular and tetrahedral meshes have inherently larger truncation error.
- Less truncation error when faces aligned with flow direction and gradients.





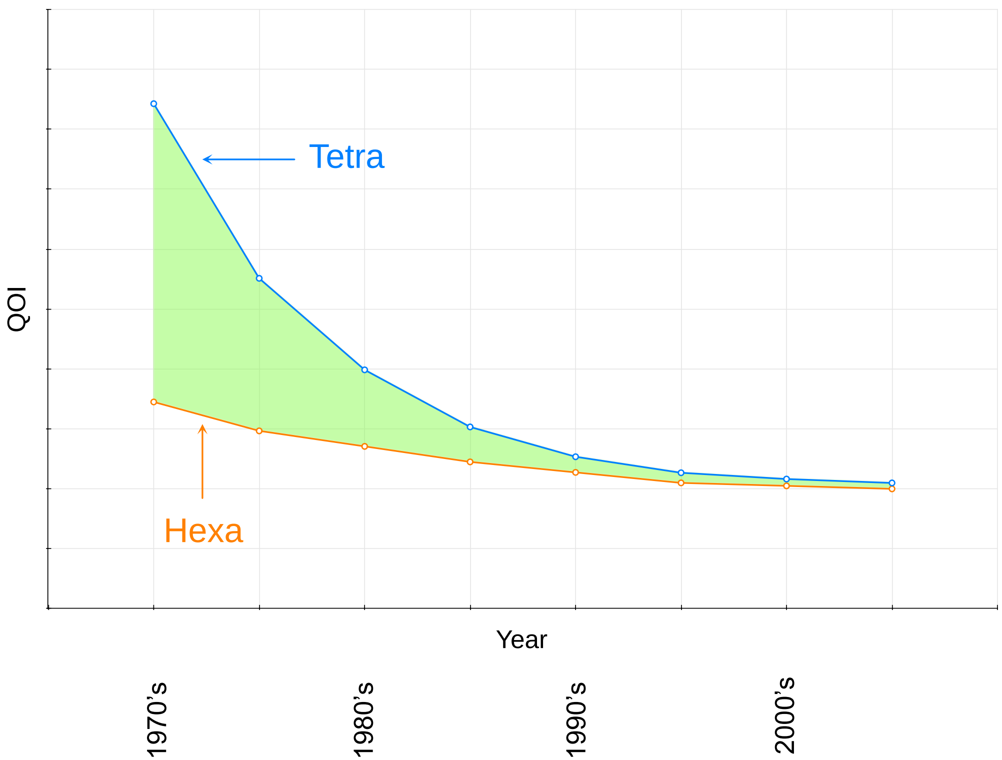
$$\phi_f = \phi_P + \cancel{\frac{\partial \phi}{\partial y}} \Delta y + \mathcal{O}(\cancel{\Delta y^2})$$

$$\phi_f = \phi_P + \frac{\partial \phi}{\partial y} \Delta y + \mathcal{O}(\Delta y^2)$$

What is a good mesh?

Striving for quality

- For the same cell count, hexahedral meshes will give more accurate solutions, especially if the grid lines are aligned with the flow.
- But this does not mean that tetrahedral meshes are not good, by carefully choosing the numerical scheme you can get the same level of accuracy as in hexahedral meshes.
- The problem with tetrahedral meshes is mainly related to the way gradients are computed.

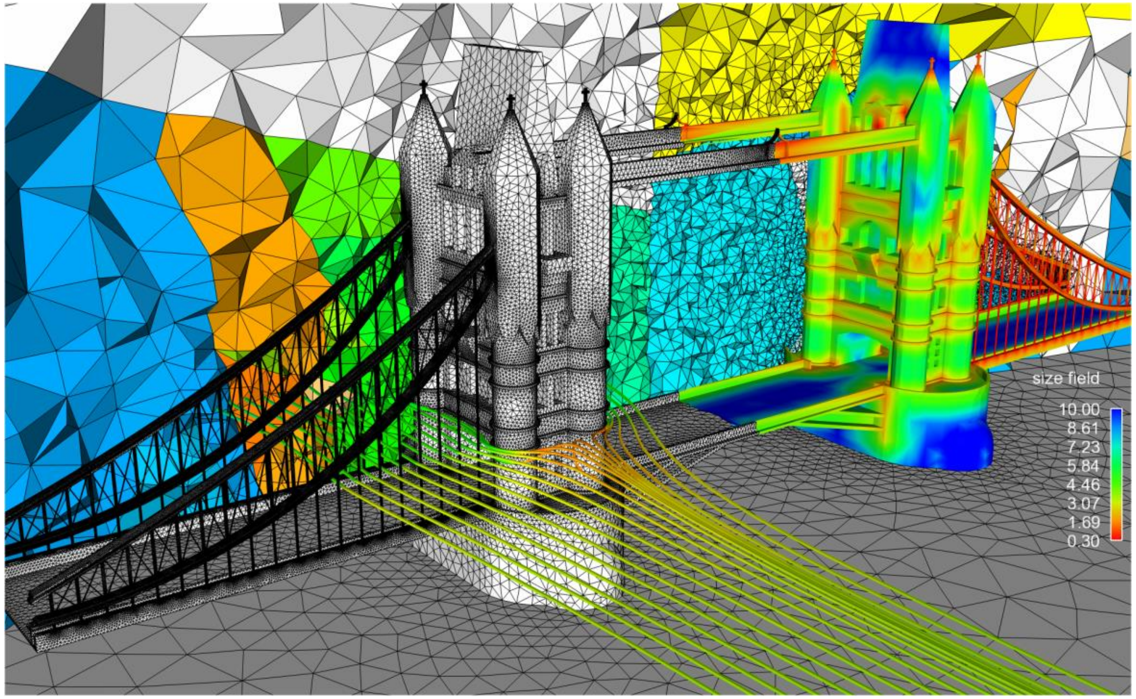


- In the early years of CFD, there was a huge gap between the outcome of tetra and hex meshes.
- But with time and thanks to developments in numerical methods and computer science (software and hardware), today all cell types give the same results.

What is a good mesh?

Striving for quality

- The mesh density should be high enough to capture all relevant flow features.
- In areas where the solution change slowly, you can use larger elements.
- A good mesh does not rely in the fact that the more cells we use the better the solution.



23rd IMR Meshing Maestro Contest Winner
Zhoufang Xiao , Jianjing Zheng, Dawei Zhao, Lijuan Zeng, Jianjun Chen, Yao Zheng
Center for Engineering & Scientific Computation, Zhejiang University, China.
http://www.sandia.gov/imr/MeshingContest.html

What is a good mesh?

Striving for quality

- Hexes, prisms, and quadrilaterals can be easily aligned with the flow.
- They can also be stretched to resolve boundary layers without losing much quality.
- Triangular and tetrahedral meshes can easily be adapted to any kind of geometry. The mesh generation process is almost automatic.
- Tetrahedral meshes normally need more computing resources during the solution stage. But this can be easily offset by the time saved during the mesh generation stage.
- Increasing the cells count will likely improve the solution accuracy, but at the cost of a higher computational cost. However, a finer mesh does not mean a better mesh.
- To keep the cell count low, use non-uniform meshes to cluster cells only where they are needed. Use local refinements and solution adaption to further refine only on selected areas.
- In boundary layers, quads, hexes, and prisms/wedges cells are preferred over triangles, tetrahedrons, or pyramids.
- If you are not using wall functions (turbulence modeling), the mesh next to the walls should be fine enough to resolve the boundary layer flow. Have in mind that this will rocket the cell count and increase the computing time.

What is a good mesh?

Striving for quality

- Use hexahedral meshes whenever is possible, specially if high accuracy in predicting forces is your goal (drag prediction) or for turbo machinery applications.
- For complex flows without dominant flow direction, quad and hex meshes loose their advantages.
- Keep orthogonality, skewness, and aspect ratio to a minimum.
- Change in cell size should be smooth.
- Always check the mesh quality. Remember, one single cell can cause divergence or give you inaccurate results.
- When you strive for quality, you avoid the GIGO syndrome (garbage in, garbage out).
- Just to end for good the mesh quality talk:
    - A good mesh is a mesh that serves your project objectives.
    - So, as long as your results are physically realistic, reliable and accurate; your mesh is good.
    - Know your physics and generate a mesh able to resolve the physics involve, without over-doing.

What is a good mesh?

A good mesh might not lead to the ideal solution, but a bad mesh will always lead to a bad solution.

P. Baker – Pointwise

Who owns the mesh, owns the solution.

H. Jasak – Wikki Ltd.

Avoid the GIGO syndrome (Garbage In – Garbage Out).
As I am a really positive guy I prefer to say,
good mesh – good results.

J. G. – WD

Roadmap

1. ~~Meshing preliminaries~~
2. ~~What is a good mesh?~~
3. **Mesh quality assessment in OpenFOAM®**
4. Mesh generation using blockMesh.
5. Mesh generation using snappyHexMesh.
6. snappyHexMesh guided tutorials.
7. Mesh conversion
8. Geometry and mesh manipulation utilities

Mesh quality assessment in OpenFOAM®

Mesh quality metrics in OpenFOAM

- In the file *primitiveMeshCheck.C* located in the directory **$WM_PROJECT_DIR/src/OpenFOAM/meshes/primitiveMesh/primitiveMeshCheck/** you will find the quality metrics hardwired in OpenFOAM. Their maximum (or minimum) values are defined as follows:

```
Foam::scalar Foam::primitiveMesh::closedThreshold_  = 1.0e-6;
Foam::scalar Foam::primitiveMesh::aspectThreshold_  = 1000;
Foam::scalar Foam::primitiveMesh::nonOrthThreshold_ = 70;    // deg
Foam::scalar Foam::primitiveMesh::skewThreshold_    = 4;
Foam::scalar Foam::primitiveMesh::planarCosAngle_   = 1.0e-6;
```

- You will be able to run simulations with mesh quality errors such as high skewness, high aspect ratio, and high non-orthogonality.
- But remember, they will affect the solution accuracy, might give you strange results, and eventually can made the solver blow-up.
- Have in mind that if you have bad quality meshes, you will need to adapt the numerics to deal with this kind of meshes. We will give you our recipe later when we deal with the numerics.

Mesh quality assessment in OpenFOAM®

Mesh quality metrics in OpenFOAM

- In the file *primitiveMeshCheck.C* located in the directory **$WM_PROJECT_DIR/src/OpenFOAM/meshes/primitiveMesh/primitiveMeshCheck/** you will find the quality metrics hardwired in OpenFOAM. Their maximum (or minimum) values are defined as follows:

```
Foam::scalar Foam::primitiveMesh::closedThreshold_  = 1.0e-6;
Foam::scalar Foam::primitiveMesh::aspectThreshold_  = 1000;
Foam::scalar Foam::primitiveMesh::nonOrthThreshold_ = 70;    // deg
Foam::scalar Foam::primitiveMesh::skewThreshold_    = 4;
Foam::scalar Foam::primitiveMesh::planarCosAngle_   = 1.0e-6;
```

- You should avoid as much as possible non-orthogonality values close to 90. This is an indication that you have zero-volume cells.
- In overall, large aspect ratios do not represent a problem. It is just an indication that you have very fine meshes (which is the case when you are resolving the boundary layer).
- The default quality metrics in OpenFOAM seems to be a little bit conservative.

Mesh quality assessment in OpenFOAM®

Mesh quality metrics in OpenFOAM

- Our own personal quality metrics maximum values are:
    - Non-orthogonality = 80
    - Skewness = 8

- If we get values higher than these, we inspect the mesh and depending on the physics involved and the number and location of the bad quality cells/faces, we decide to redo the mesh or proceed with the simulation.

- If we proceed with the simulation, we choose a numerical scheme able to reduce the numerical errors introduced due to the low-quality cells/faces.

Mesh quality assessment in OpenFOAM®

Checking the mesh quality in OpenFOAM®

- To check the mesh quality and validity, OpenFOAM® comes with the utility `checkMesh`.
- To use this utility, just type in the terminal `checkMesh`, and read the screen output.
- `checkMesh` will look for/check for:
  - Mesh stats and overall number of cells of each type.
  - Check topology (boundary conditions definitions).
  - Check geometry and mesh quality (bounding box, cell volumes, skewness, orthogonality, aspect ratio, and so on).

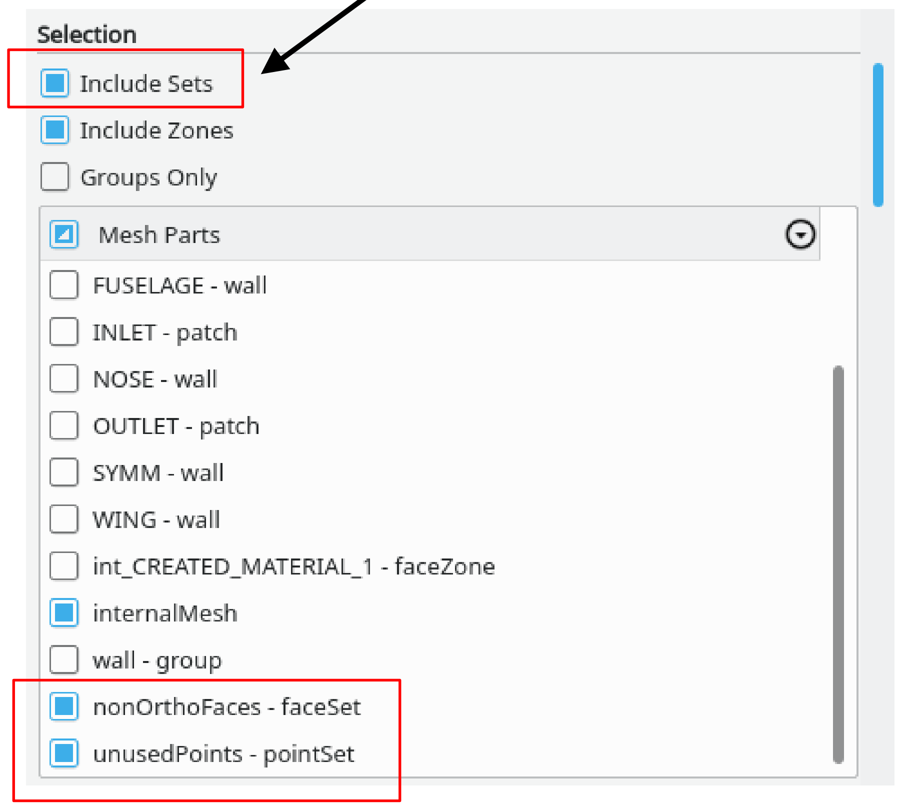
- If for any reason `checkMesh` finds errors, it will give you a message and it will tell you what check failed.
- It will also write a set with the faulty cells, faces, and/or points.
- These sets are saved in the directory **`constant/polyMesh/sets/`**
- Mesh topology and patch topology errors must be repaired.
- You will be able to run with mesh quality errors such as skewness, aspect ratio, minimum face area, and non-orthogonality.
- But remember, they will severely tamper the solution accuracy, might give you strange results, and eventually can made the solver blow-up.
- Unfortunately, `checkMesh` does not repair these errors.
- You will need to check the geometry for possible errors and generate a new mesh.
- You can visualize the failed sets directly in `paraFoam`.
- You can also convert the failed sets into VTK format by using the utility `foamToVTK`.

Mesh quality assessment in OpenFOAM®

Visualizing the failed sets in OpenFOAM®

- You can load the failed sets directly within `paraFoam`.
- Remember, you will need to create the sets. To do so, just run the `checkMesh` utility.
- If there are problems in the mesh, `checkMesh` will automatically save the sets in the directory **`constant/polyMesh/sets`**
- In `paraFoam`, simply select the option **Include Sets** and then select the sets you want to visualize.
- This method only works when using the wrapper `paraFoam`.
- If you are using paraview or a different scientific visualization application, you will need to convert the failed sets to VTK format or an alternative format.
- Also, when working with large meshes we prefer to convert the faulty sets to VTK format.
- To convert the faulty sets to VTK format you can use the utility `foamToVTK`.

Check this box to include sets



Failed sets

Mesh quality assessment in OpenFOAM®

Visualizing the failed sets in OpenFOAM®

- To convert the failed faces/cells/points to VTK format, you can proceed as follows:

  - `$> foamToVTK -set_type name_of_sets`

  where **set_type** is the type of sets (faceSet, cellSet, pointSet, surfaceFields) and **name_of_sets** is the name of the set located in the directory **`constant/polyMesh/sets`** (highAspectRatioCells, nonOrthoFaces, wrongOrientedFaces, skewFaces, unusedPoints, and so on).

- At the end, `foamToVTK` will create a directory named **`VTK`**, where you will find the failed faces/cells/points in VTK format.
- At this point you can use `paraview/paraFoam` or any scientific visualization application to open the VTK files and visualize the failed sets.

Checking mesh quality in OpenFOAM®

- Sample `checkMesh` output,

```
Mesh stats
    points:           81812
    faces:            902132
    internal faces:   871012
    cells:            443286
    faces per cell:   4
    boundary patches: 9
    point zones:      0
    face zones:       1
    cell zones:       1

Overall number of cells of each type:
    hexahedra:     0
    prisms:        0
    wedges:        0
    pyramids:      0
    tet wedges:    0
    tetrahedra:    443286
    polyhedra:     0

Checking topology...
    Boundary definition OK.
    Cell to face addressing OK.
 ***Unused points found in the mesh, number unused by faces: 16 number unused by cells: 16
  <<Writing 16 unused points to set unusedPoints
    Upper triangular ordering OK.
    Face vertices OK.
    Number of regions: 1 (OK).
```

Mesh stats

Number of each type of cells

Checking mesh topology

Unused points found in the mesh
In this case they do not harm the solution
They can be removed using topoSet and subsetMesh

Checking mesh quality in OpenFOAM®

- Sample `checkMesh` output,

```
Checking patch topology for multiply connected surfaces...
    Patch                   Faces     Points    Surface topology
    FAIRING                 1267      727       ok (non-closed singly connected)
    FUSELAGE                3243      1774      ok (non-closed singly connected)
    WING                    15313     7706      ok (non-closed singly connected)
    INLET                   272       160       ok (non-closed singly connected)
    OUTLET                  272       160       ok (non-closed singly connected)
    SYMM                    6280      3324      ok (non-closed singly connected)
    FARFIELD                3136      1645      ok (non-closed singly connected)
    NOSE                    76        49        ok (non-closed singly connected)
    COCKPIT                 1261      670       ok (non-closed singly connected)

Checking geometry...
    Overall domain bounding box (-15000 -7621.0713 -7396.4536) (30048.969 0 7446.8442)
    Mesh has 3 geometric (non-empty/wedge) directions (1 1 1)
    Mesh has 3 solution (non-empty) directions (1 1 1)
    Boundary openness (-4.2298633e-18 8.0240802e-16 4.013988e-16) OK.
    Max cell openness = 4.8098963e-16 OK.
    Max aspect ratio = 29.575835 OK.
    Minimum face area = 0.0066721253. Maximum face area = 1037224.8.  Face area magnitudes OK.
    Min volume = 0.00050536842. Max volume = 3.2500889e+08.  Total volume = 5.0960139e+12.  Cell volumes OK.
    Mesh non-orthogonality Max: 86.939754 average: 17.939523
   *Number of severely non-orthogonal (> 70 degrees) faces: 3168.
    Non-orthogonality check OK.
  <<Writing 3168 non-orthogonal faces to set nonOrthoFaces
    Face pyramids OK.
    Max skewness = 2.5719979 OK.
    Coupled point location match (average 0) OK.

Failed 1 mesh checks.

End
```

Boundary patches

Mesh bounding box

Aspect ratio

High non-orthogonality
But we still can run the simulation

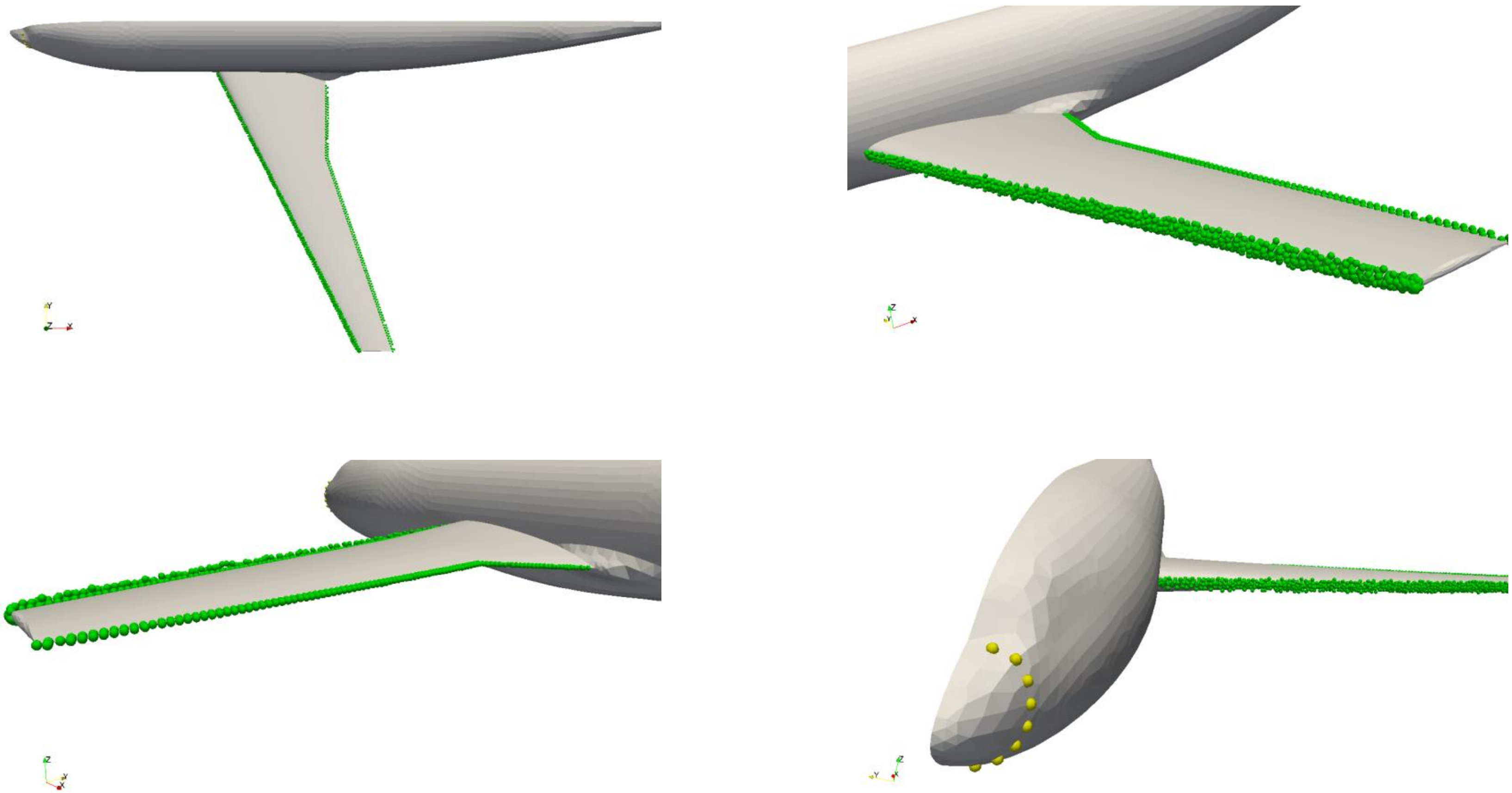
Skewness

The fact that one check failed does not mean that you can not run the simulation

Mesh quality assessment in OpenFOAM®

Visualization of faulty sets in paraFoam

- You will find this case ready to use in the directory,
  `$PTOFC/mesh_quality_manipulation/M1_wingbody`
- To run the case, just follow the instructions in the *`README.FIRST`* files.



Non orthogonal faces (green spheres) and unused points (yellow spheres)

Roadmap

1. ~~Meshing preliminaries~~
2. ~~What is a good mesh?~~
3. ~~Mesh quality assessment in OpenFOAM®~~
4. **Mesh generation using blockMesh.**
5. Mesh generation using snappyHexMesh.
6. snappyHexMesh guided tutorials.
7. Mesh conversion
8. Geometry and mesh manipulation utilities

blockMesh

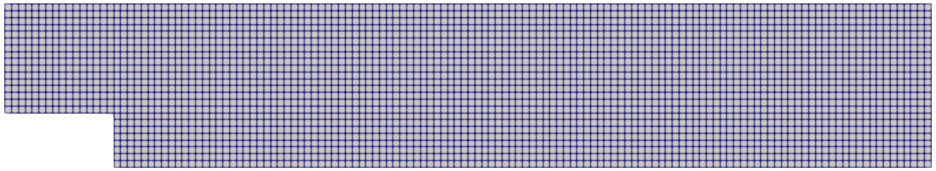
- *"blockMesh is a multi-block mesh generator."*
- For simple geometries, the mesh generation utility `blockMesh` can be used.
- The mesh is generated from a dictionary file named *`blockMeshDict`* located in the **`system`** directory.
- This meshing tool generates high quality meshes.
- It is the tool to use for very simple geometries. As the complexity of the geometry increases, the effort and time required to setup the dictionary increases a lot.
- Usually, the background mesh used with `snappyHexMesh` consist of a single rectangular block; therefore, `blockMesh` can be used with no problem.
- It is highly recommended to create a template of the dictionary *`blockMeshDict`* that you can change according to the dimensions of your domain.
- You can also use m4 or Python scripting to automate the whole process.

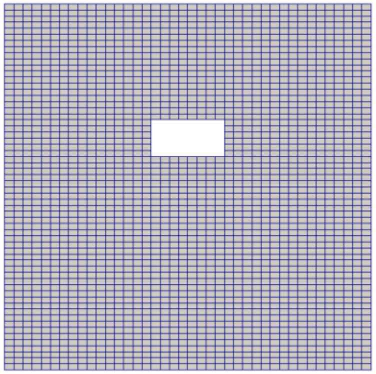
blockMesh

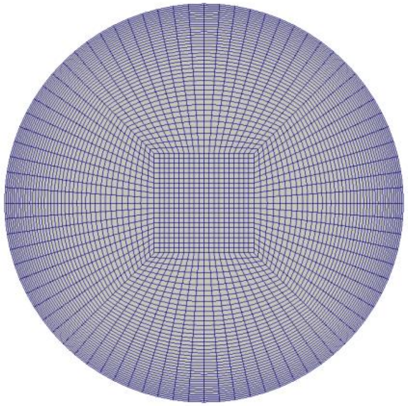
- *"blockMesh is a multi-block mesh generator."*
- For simple geometries, the mesh generation utility `blockMesh` can be used.
- The mesh is generated from a dictionary file named *`blockMeshDict`*, which is located in the directory **`system`**.
- If you are using OpenFOAM 2.4.x (or older versions) this dictionary is located in the **`constant/polyMesh`** directory.
- The *`blockMeshDict`* dictionary can be easily parameterize.
- The meshing tool generates high quality meshes; it is the tool to use for very simple geometries. As the complexity of the geometry increases, the effort and time required to setup the dictionary increases a lot.
- Usually, the background mesh used with `snappyHexMesh` consist of a single rectangular block, therefore `blockMesh` can be used with no problem.
- It is highly recommended to create a template of the dictionary *`blockMeshDict`* that you can change according to the dimensions of your domain.
- You can also use m4 or Python scripting to automate the whole process.

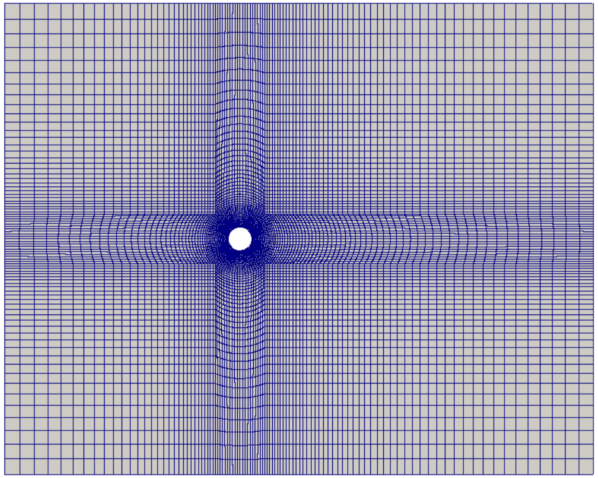
blockMesh

- These are a few meshes that you can generate using `blockMesh`. As you can see, they are not very complex.
- However, generating the blocking topology requires some effort.



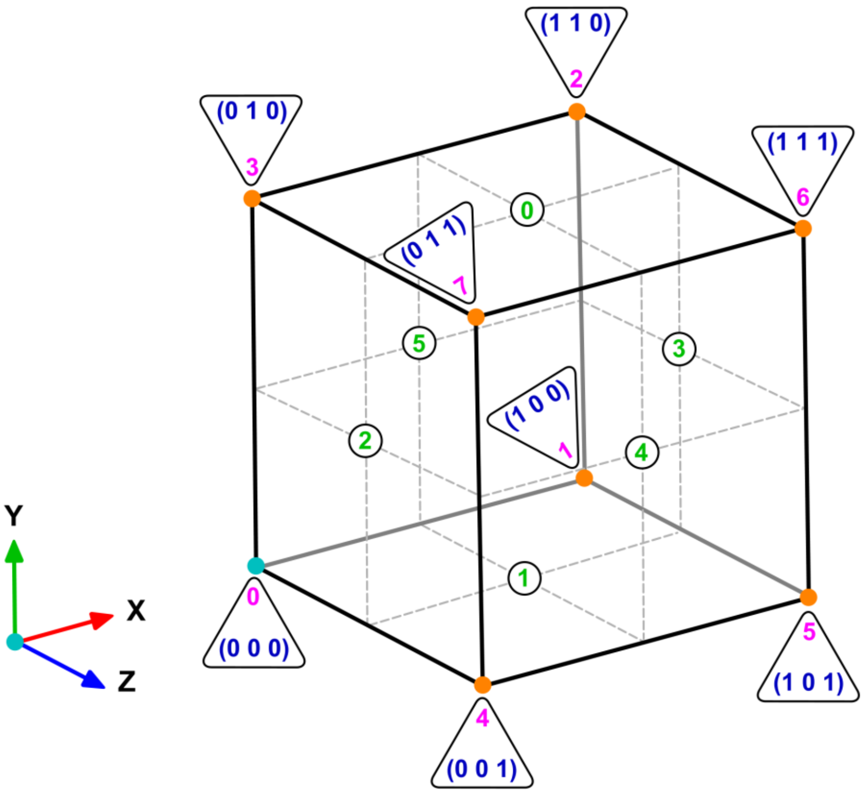






Mesh generation using blockMesh

blockMesh workflow



VERTICES

0	0	0	0
1	1	0	0
2	1	1	0
3	0	1	0
4	0	0	1
5	1	0	1
6	1	1	1
7	0	1	1

BLOCK (HEX)

0 1 2 3 4 5 6 7

FACES

0	3	7	6	2
1	1	5	4	0
2	0	4	7	3
3	2	6	5	1
4	0	3	2	1
5	4	5	6	7

FACE INDEX IS NOT IMPORTANT

- To generate a mesh with `blockMesh`, you will need to define the vertices, block connectivity and number of cells in each direction.
- To assign boundary patches, you will need to define the faces connectivity

blockMesh guided tutorials

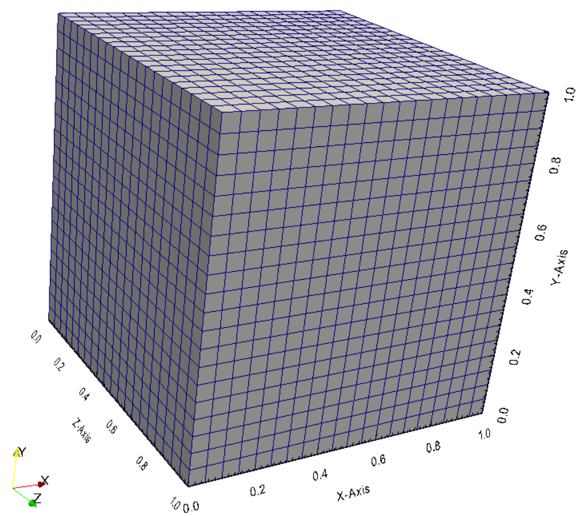
- Meshing with blockMesh – Case 1.
- We will use the square cavity case.
- You will find this case in the directory:

```
$PTOFC/101BLOCKMESH/C1
```

- In the case directory, you will find a few scripts with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on.
- These scripts can be used to run the case automatically by typing in the terminal, for example,
  - `$> sh run_solver`
- These scripts are human-readable, and we highly recommend you open them, get familiar with the steps, and type the commands in the terminal. In this way, you will get used with the command line interface and OpenFOAM commands.
- If you are already comfortable with OpenFOAM, run the cases automatically using these scripts.
- In the case directory, you will also find the `README.FIRST` file. In this file, you will find some additional comments.

What are we going to do?

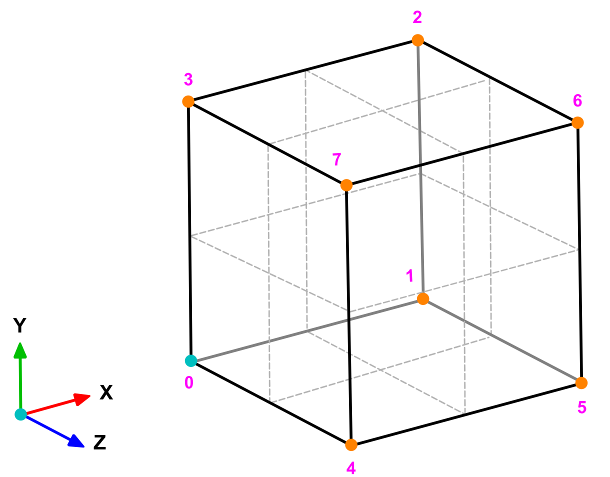
- We will use this simple case to take a close look at a *blockMeshDict* dictionary.
- We will study all sections in the *blockMeshDict* dictionary.
- We will introduce two features useful for parameterization, namely, macro syntax and inline calculations.
- You can use this dictionary as a *blockMeshDict* template that you can change automatically according to the dimensions of your domain and the desired cell spacing.



The *blockMeshDict* dictionary.

```
17      convertToMeters 1;
18
19      xmin 0;
20      xmax 1;
21      ymin 0;
22      ymax 1;
23      zmin 0;
24      zmax 1;
25
30      deltax 0.05;
31      deltay 0.05;
32      deltaz 0.05;
33
34      lx #calc "$xmax - $xmin";
35      ly #calc "$ymax - $ymin";
36      lz #calc "$zmax - $zmin";
37
38      xcells #calc "round(($lx)/($deltax))";
39      ycells #calc "round(($ly)/($deltay))";
40      zcells #calc "round(($lz)/($deltaz))";
41
42      vertices
43      (
44      //BLOCK 0
45          ($xmin  $ymin  $zmin)      //0
46          ($xmax  $ymin  $zmin)      //1
47          ($xmax  $ymax  $zmin)      //2
48          ($xmin  $ymax  $zmin)      //3
49          ($xmin  $ymin  $zmax)      //4
50          ($xmax  $ymin  $zmax)      //5
51          ($xmax  $ymax  $zmax)      //6
52          ($xmin  $ymax  $zmax)      //7
64      );
```

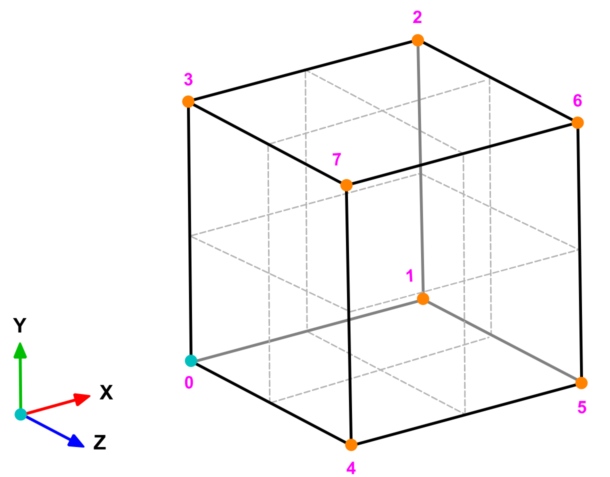
- The keyword **convertToMeters** (line 17), is a scaling factor. In this case we do not scale the dimensions.
- In lines 19-24 we declare some variables using macro syntax notation. With macro syntax, we first declare the variables and their values (lines 19-24), and then we can use the variables by adding the symbol **$** to the variable name (lines 45-52).
- In lines 30-32 we use macro syntax to declare another set of variables that will be used later.
- Macro syntax is a very convenient way to parameterize dictionaries.



The *blockMeshDict* dictionary.

```
17     convertToMeters 1;
18
19     xmin 0;
20     xmax 1;
21     ymin 0;
22     ymax 1;
23     zmin 0;
24     zmax 1;
25
30     deltax 0.05;
31     deltay 0.05;
32     deltaz 0.05;
33
34     lx #calc "$xmax - $xmin";
35     ly #calc "$ymax - $ymin";
36     lz #calc "$zmax - $zmin";
37
38     xcells #calc "round(($lx)/($deltax))";
39     ycells #calc "round(($ly)/($deltay))";
40     zcells #calc "round(($lz)/($deltaz))";
41
42     vertices
43     (
44     //BLOCK 0
45         ($xmin  $ymin  $zmin)      //0
46         ($xmax  $ymin  $zmin)      //1
47         ($xmax  $ymax  $zmin)      //2
48         ($xmin  $ymax  $zmin)      //3
49         ($xmin  $ymin  $zmax)      //4
50         ($xmax  $ymin  $zmax)      //5
51         ($xmax  $ymax  $zmax)      //6
52         ($xmin  $ymax  $zmax)      //7
64     );
```

- In lines 34-40 we are doing inline calculations using the directive **#calc**.
- Basically, we are programming directly in the dictionary. OpenFOAM® will compile this function as it reads it.
- With inline calculations and **codeStream** you can access many OpenFOAM® functions from the dictionaries.
- Inline calculations and **codeStream** are very convenient ways to parameterize dictionaries and program directly on the dictionaries.



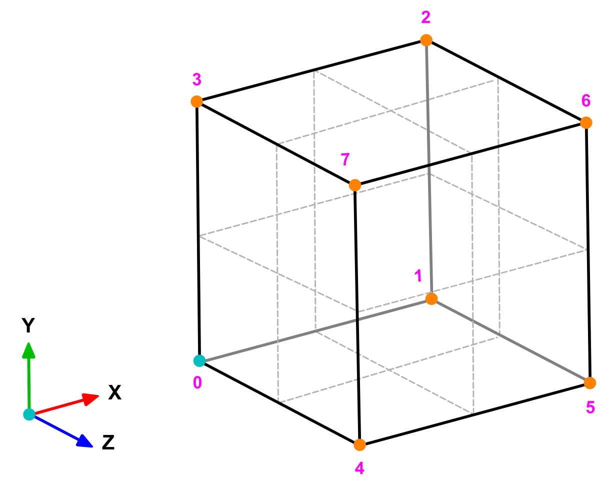
The *blockMeshDict* dictionary.

```
17      convertToMeters 1;
18
19      xmin 0;
20      xmax 1;
21      ymin 0;
22      ymax 1;
23      zmin 0;
24      zmax 1;
25
30      deltax 0.05;
31      deltay 0.05;
32      deltaz 0.05;
33
34      lx #calc "$xmax - $xmin";
35      ly #calc "$ymax - $ymin";
36      lz #calc "$zmax - $zmin";
37
38      xcells #calc "round(($lx)/($deltax))";
39      ycells #calc "round(($ly)/($deltay))";
40      zcells #calc "round(($lz)/($deltaz))";
41
42      vertices
43      (
44      //BLOCK 0
45          ($xmin  $ymin  $zmin)       //0
46          ($xmax  $ymin  $zmin)       //1
47          ($xmax  $ymax  $zmin)       //2
48          ($xmin  $ymax  $zmin)       //3
49          ($xmin  $ymin  $zmax)       //4
50          ($xmax  $ymin  $zmax)       //5
51          ($xmax  $ymax  $zmax)       //6
52          ($xmin  $ymax  $zmax)       //7
64      );
```

- To do inline calculations using the directive **#calc**, we proceed as follows (we will use line 35 as example):

ly #calc "$ymax - $ymin";

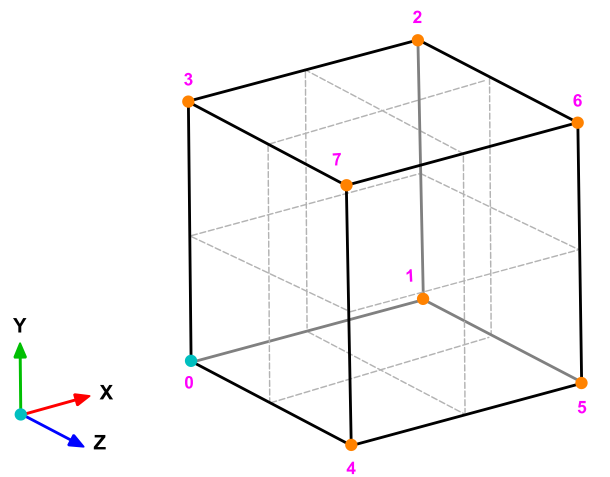
- We first give a name to the new variable (**ly**), we then tell OpenFOAM® that we want to do an inline calculation (**#calc**), and then we do the inline calculation (**"$ymax-$ymin";)**). Notice that the operation must be between double quotation marks.



The *blockMeshDict* dictionary.

```
17      convertToMeters 1;
18
19      xmin 0;
20      xmax 1;
21      ymin 0;
22      ymax 1;
23      zmin 0;
24      zmax 1;
25
30      deltax 0.05;
31      deltay 0.05;
32      deltaz 0.05;
33
34      lx #calc "$xmax - $xmin";
35      ly #calc "$ymax - $ymin";
36      lz #calc "$zmax - $zmin";
37
38      xcells #calc "round(($lx)/($deltax))";
39      ycells #calc "round(($ly)/($deltay))";
40      zcells #calc "round(($lz)/($deltaz))";
41
42      vertices
43      (
44      //BLOCK 0
45          ($xmin  $ymin  $zmin)      //0
46          ($xmax  $ymin  $zmin)      //1
47          ($xmax  $ymax  $zmin)      //2
48          ($xmin  $ymax  $zmin)      //3
49          ($xmin  $ymin  $zmax)      //4
50          ($xmax  $ymin  $zmax)      //5
51          ($xmax  $ymax  $zmax)      //6
52          ($xmin  $ymax  $zmax)      //7
64      );
```

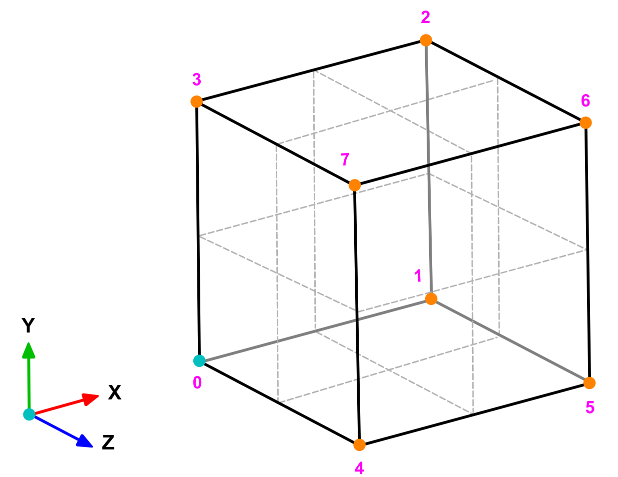
- In lines lines 34-36, we use inline calculations to compute the length in each direction.
- Then we compute the number of cells to be used in each direction (lines 38-40).
- To compute the number of cells we use as cell spacing the values declared in lines 30-32.
- By proceeding in this way, we can compute automatically the number of cells needed in each direction according to the desired cell spacing.



The *blockMeshDict* dictionary.

```
17      convertToMeters 1;
18
19      xmin 0;
20      xmax 1;
21      ymin 0;
22      ymax 1;
23      zmin 0;
24      zmax 1;
25
30      deltax 0.05;
31      deltay 0.05;
32      deltaz 0.05;
33
34      lx #calc "$xmax - $xmin";
35      ly #calc "$ymax - $ymin";
36      lz #calc "$zmax - $zmin";
37
38      xcells #calc "round(($lx)/($deltax))";
39      ycells #calc "round(($ly)/($deltay))";
40      zcells #calc "round(($lz)/($deltaz))";
41
42      vertices
43      (
44      //BLOCK 0
45          ($xmin  $ymin  $zmin)      //0
46          ($xmax  $ymin  $zmin)      //1
47          ($xmax  $ymax  $zmin)      //2
48          ($xmin  $ymax  $zmin)      //3
49          ($xmin  $ymin  $zmax)      //4
50          ($xmax  $ymin  $zmax)      //5
51          ($xmax  $ymax  $zmax)      //6
52          ($xmin  $ymax  $zmax)      //7
64      );
```

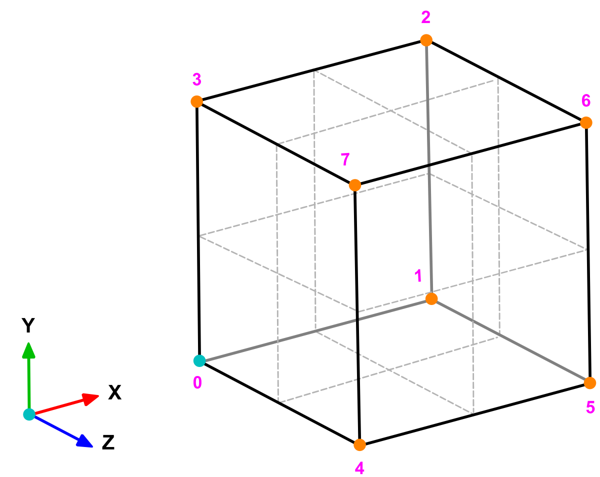
- In the vertices section (lines 42-64), we define the vertex coordinates of the geometry.
- In this case, there are eight vertices defining a 3D block.
- Remember, OpenFOAM® always uses 3D meshes, even if the simulation is 2D. For 2D meshes, you only add one cell in the third dimension.
- Notice that the vertex numbering starts from 0 (as the counters in c++). This numbering applies for blocks as well.



The *blockMeshDict* dictionary.

- In lines 66-69, we define the block topology, **hex** means that it is a structured hexahedral block. In this case, we are generating a rectangular mesh.
- In line 68, **(0 1 2 3 4 5 6 7)** are the vertices used to define the block (and yes, the order is important). Each hex block is defined by eight vertices, in sequential order. Where the first vertex in the list represents the origin of the coordinate system (vertex **0** in this case).
- **($xcells $ycells $zcells)** is the number of mesh cells in each direction (**X Y Z**). Notice that we are using macro syntax, and we compute the values using inline calculations.
- **simpleGrading (1 1 1)** is the grading or mesh stretching in each direction (**X Y Z**), in this case the mesh is uniform. We will deal with mesh grading/stretching in the next case.

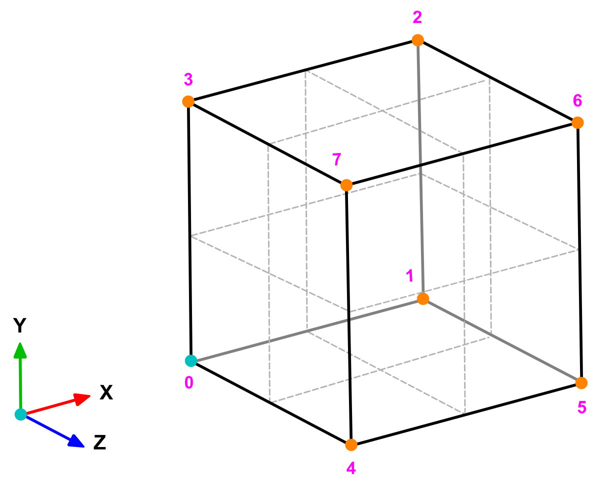
```
66      blocks
67      (
68          hex (0 1 2 3 4 5 6 7) ($xcells $ycells $zcells) simpleGrading (1 1 1)
69      );
70
71      edges
72      (
73
74      );
```



The *blockMeshDict* dictionary.

- Let us talk about the block ordering **hex (0 1 2 3 4 5 6 7)**, which is extremely important.
- **hex** blocks are defined by eight vertices in sequential order. Where the first vertex in the list represents the origin of the coordinate system (vertex 0 in this case).
- Starting from this vertex, we construct the block topology. So, in this case, the first part of the block is made up by vertices 0 1 2 3 and the second part of the block is made up by vertices 4 5 6 7 (notice that we start from vertex 4 which is the projection in the **Z**-direction of vertex 0).
- In this case, the vertices are ordered in such a way that if we look at the screen/paper (-z direction), the vertices rotate counter-clockwise.
- If you add a second block, you must identify the first vertex and starting from it, you should construct the block topology. In this case, you will need to merges faces, you will find more information about merging face in the supplement lectures.

```
blocks
(
    hex (0 1 2 3 4 5 6 7) ($xcells $ycells $zcells) simpleGrading (1 1 1)
);

edges
(

);
```

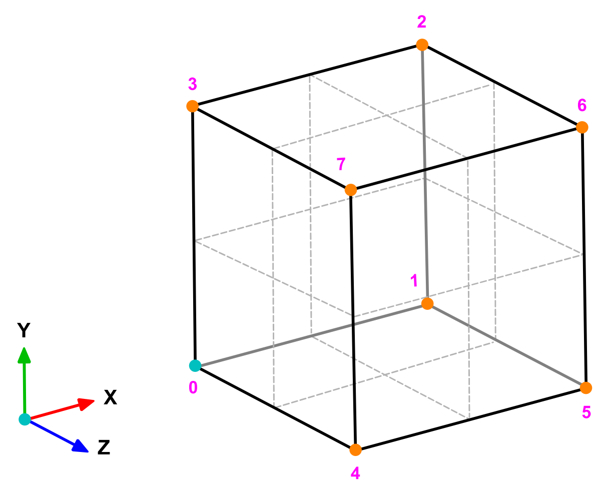


blockMesh guided tutorials

The *blockMeshDict* dictionary.

- In lines 71-74, we define the edges.
- Edges, are constructed from the vertices definition.
- Each edge joining two vertices is assumed to be straight by default.
- The user can specify any edge to be curved by entries in the section **edges**.
- Possible options are Bspline, arc, line, polyline, project, projectCurve, spline.
- For example, to define an arc we first define the vertices to be connected to form an edge and then we give an interpolation point.
- To define a polyline, we first define the vertices to be connected to form an edge and then we give a list of the coordinates of the interpolation points.
- In this case and as we do not specify anything, all edges are assumed to be straight lines.

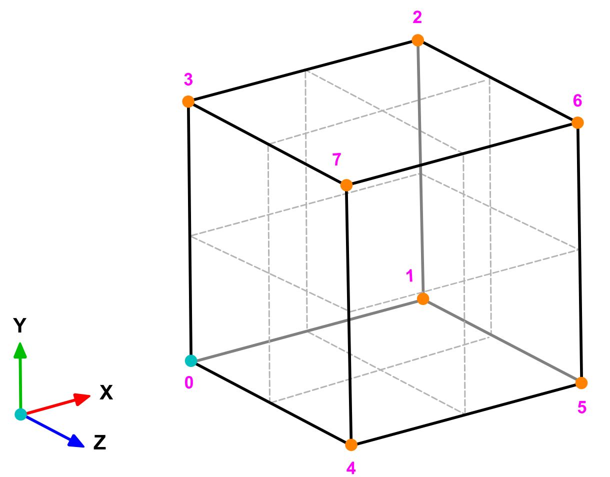
```
66      blocks
67      (
68          hex (0 1 2 3 4 5 6 7) ($xcells $ycells $zcells) simpleGrading (1 1 1)
69      );
70
71      edges
72      (
73
74      );
```



The *blockMeshDict* dictionary.

```
boundary
(
    minX
    {
        type wall;
        faces
        (
            (0 4 7 3)
        );
    }
    maxX
    {
        type wall;
        faces
        (
            (2 6 5 1)
        );
    }
    minY
    {
        type wall;
         faces
        (
            (0 1 5 4)
        );
    }
    maxY
    {
        type wall;
        faces
        (
            (3 7 6 2)
        );
    }
```

- In the section **boundary**, we define all the patches where we want to apply boundary conditions.
- This step is of paramount importance, because if we do not define the surface patches, we will not be able to apply the boundary conditions to individual surface patches.

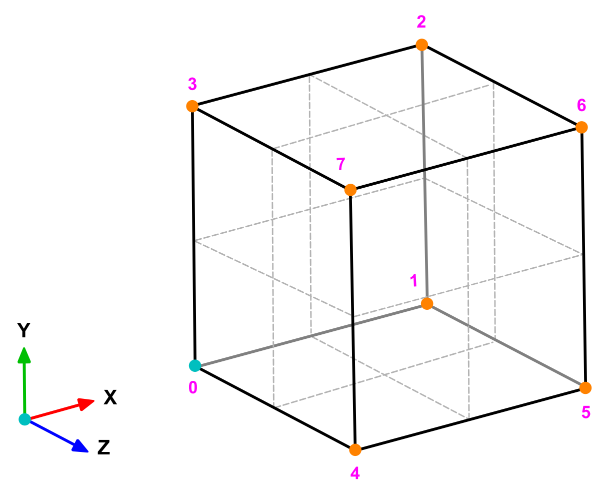


blockMesh guided tutorials

The *blockMeshDict* dictionary.

```
76      boundary
77      (
79          minX
80          {
81              type wall;
82              faces
83              (
84                  (0 4 7 3)
85              );
86          }
88          maxX
89          {
90              type wall;
91              faces
92              (
93                  (2 6 5 1)
94              );
95          }
97          minY
98          {
99              type wall;
100              faces
101             (
102                 (0 1 5 4)
103             );
104         }
106         maxY
107         {
108             type wall;
109             faces
110             (
111                 (3 7 6 2)
112             );
113         }
```

- In lines 79-86 we define a boundary patch.
- In line 79 we define the patch name **minX** (the name is given by the user).
- In line 81 we give a **base type** to the surface patch. In this case **wall** (do not worry we are going to talk about this later).
- In line 84 we give the connectivity list of the vertices that made up the surface patch or face, that is, **(0 4 7 3)**.
- Have in mind that the vertices need to be neighbors and it does not matter if the ordering is clockwise or counterclockwise.

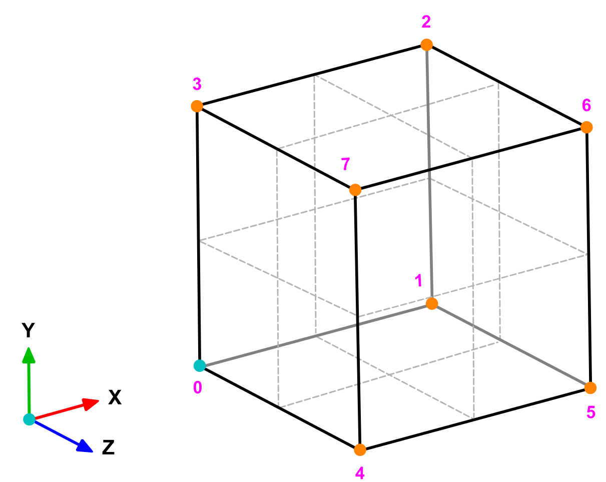


blockMesh guided tutorials

The *blockMeshDict* dictionary.

```
76      boundary
77      (
79          minX
80          {
81              type wall;
82              faces
83              (
84                  (0 4 7 3)
85              );
86          }
88          maxX
89          {
90              type wall;
91              faces
92              (
93                  (2 6 5 1)
94              );
95          }
97          minY
98          {
99              type wall;
100              faces
101             (
102                 (0 1 5 4)
103             );
104         }
106         maxY
107         {
108             type wall;
109             faces
110             (
111                 (3 7 6 2)
112             );
113         }
```

- Have in mind that the vertices need to be neighbors and it does not matter if the ordering is clockwise or counterclockwise.
- Remember, faces are defined by a list of 4 vertex numbers, e.g., **(3 7 6 2)**.
- In lines 88-95 we define the patch **maxX**.
- In lines 97-104 we define the patch **minY**.
- In lines 106-113 we define the patch **maxY**.



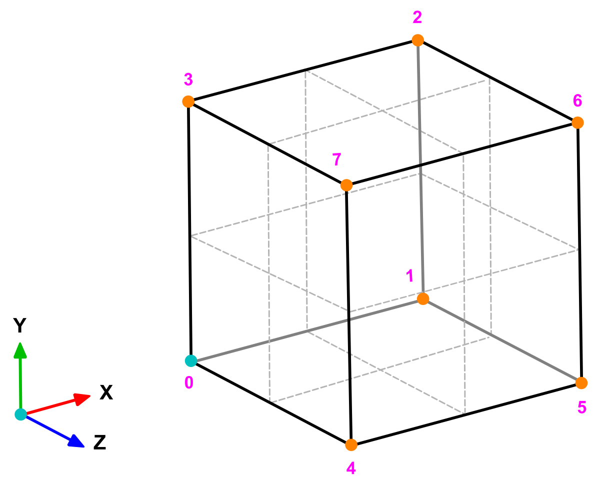
blockMesh guided tutorials

The *blockMeshDict* dictionary.

```
115        minZ
116        {
117            type wall;
118            faces
119            (
120                (0 3 2 1)
121            );
122        }
124        maxZ
125        {
126            type wall;
127            faces
128            (
129                (4 5 6 7)
130            );
131        }
132    );
133
134    mergePatchPairs
135    (
136
137    );
```

- In lines 115-122 we define the patch **minZ**.
- In lines 124-132 we define the patch **maxZ**.
- You can also group many faces into one patch, for example, instead of creating the patches **minZ** and **maxZ**, you can group them into a single patch named **backAndFront**, as follows,

```
backAndFront
{
    type wall;
    faces
    (
        (4 5 6 7)
        (0 3 2 1)
    );
}
```

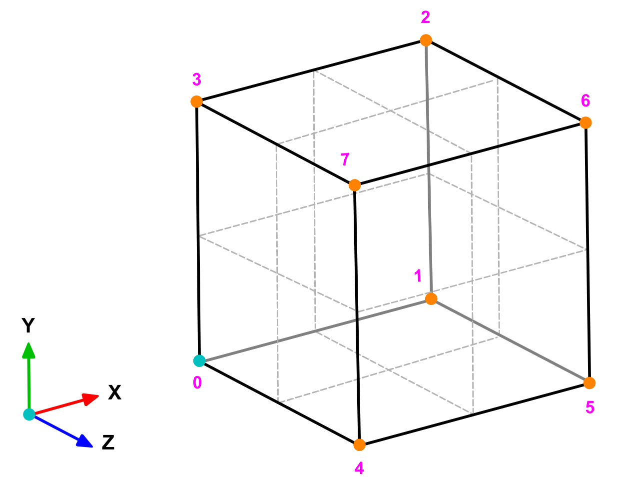


blockMesh guided tutorials

The *blockMeshDict* dictionary.

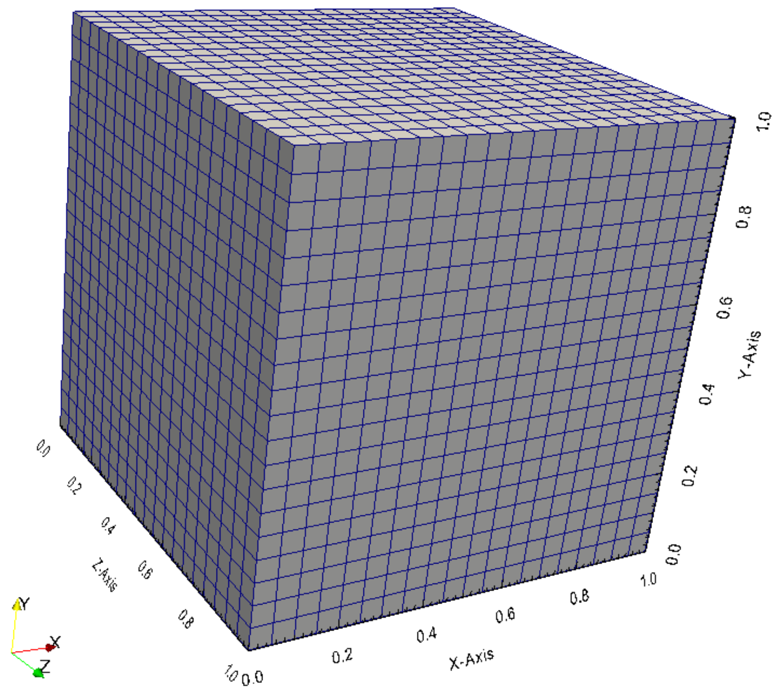
```
115         minZ
116         {
117             type wall;
118             faces
119             (
120                 (0 3 2 1)
121             );
122         }
124         maxZ
125         {
126             type wall;
127             faces
128             (
129                 (4 5 6 7)
130             );
131         }
132     );
133
134     mergePatchPairs
135     (
136
137     );
```

- We can merge blocks in the section **mergePatchPairs** (lines 134-137).
- The block patches to be merged must be first defined in the **boundary** list, `blockMesh` then connect the two blocks.
- In this case, as we have one single block there is no need to merge patches.



The *blockMeshDict* dictionary.

- To sum up, the *blockMeshDict* dictionary generates a single block with:
  - **X**/**Y**/**Z** dimensions: **1.0**/**1.0**/**1.0**
  - As the cell spacing in all directions is defined as **0.05**, it will use the following number of cells in the **X**, **Y** and **Z** directions: **20** x **20** x **20** cells.
  - One single **hex** block with straight lines.
  - Six patches of base type **wall**, namely, **left**, **right**, **top**, **bottom**, **front** and **back**.
- The information regarding the patch **base type** and patch **name** is saved in the file *boundary*. Feel free to modify this file to fit your needs.
- Remember to use the utility `checkMesh` to check the quality of the mesh and look for topological errors.
- Topological errors must be repaired.
- If you are interested in visualizing the actual block topology, you can use `paraFoam` as follows,
  - `$> paraFoam -block`



blockMesh guided tutorials

The *constant/polyMesh/boundary* dictionary

```
6
(
    minX
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       22800;
    }
    maxX
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       23200;
    }
    minY
    {
        type            empty;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       23600;
    }
    maxY
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       24000;
    }
    minZ
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       24400;
    }
    maxZ
    {
        type            empty;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       24800;
    }
)
```

- First of all, this file is automatically generated after you create the mesh, or you convert it from a third-party format.
- In this file, the geometrical information related to the **base type** patch of each boundary of the domain is specified.
- The **base type** boundary condition is the actual surface patch where we are going to apply a **primitive type** boundary condition (or numerical boundary condition).
- The **primitive type** boundary condition assign a field value to the surface patch.
- You define the **numerical type** patch (or the value of the boundary condition), in the directory `0` or time directories.
- The **name** and **base type** of the patches was defined in the dictionary *blockMeshDict* in the section **boundary**.
- You can change the **name** if you do not like it. Do not use strange symbols or white spaces.
- You can also change the **base type**. For instance, you can change the type of the patch **minX** from **wall** to **patch**.

The *constant/polyMesh/boundary* dictionary

```
6
(
    minX
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       22800;
    }
    maxX
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       23200;
    }
    minY
    {
        type            empty;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       23600;
    }
    maxY
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       24000;
    }
    minZ
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       24400;
    }
    maxZ
    {
        type            empty;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       24800;
    }
)
```

- If you do not define the boundary patches in the dictionary *blockMeshDict*, they are grouped automatically in a default group named **defaultFaces** of type **empty**.
- For instance, if you do not assign a **base type** to the patch **front**, it will be grouped as follows:

```
defaultFaces
{
        type                    empty;
        inGroups                1(empty);
        nFaces                  400;
        startFace               24800;
}
```

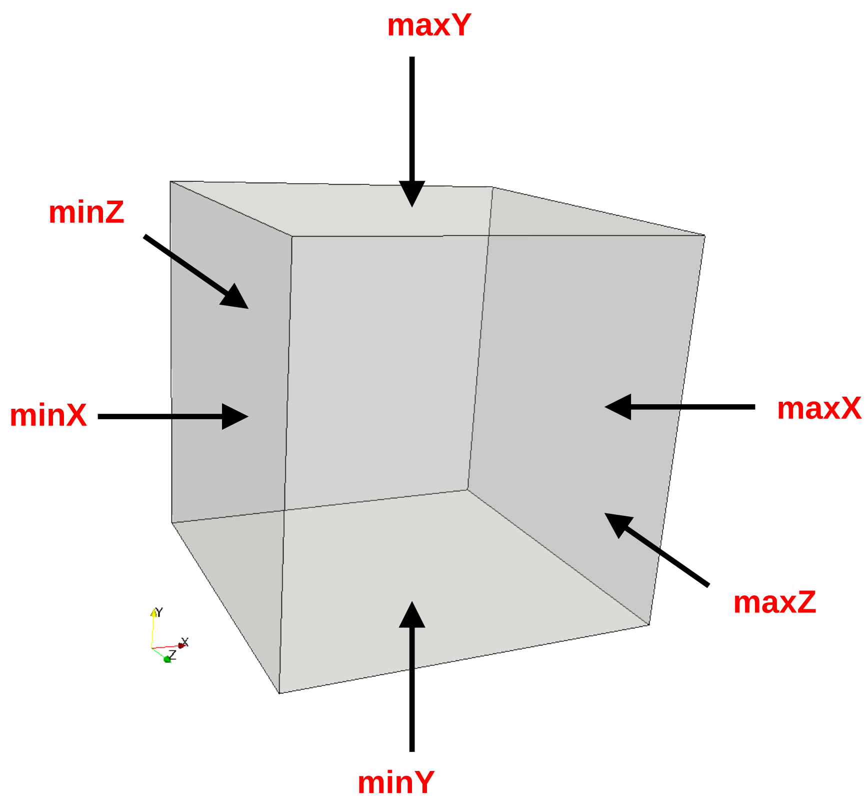
- Remember, you can manually change the name and type.

The *constant/polyMesh/boundary* dictionary

```
6
(
    minX
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       22800;
    }
    maxX
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       23200;
    }
    minY
    {
        type            empty;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       23600;
    }
    maxY
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       24000;
    }
    minZ
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       24400;
    }
    maxZ
    {
        type            empty;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       24800;
    }
)
```

Number of surface patches

In the list bellow there must be 6 patches definition.



The *constant/polyMesh/boundary* dictionary

```
6
(
    minX
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       22800;
    }
    maxX
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       23200;
    }
    minY
    {
        type            empty;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       23600;
    }
    maxY
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       24000;
    }
    minZ
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       24400;
    }
    maxZ
    {
        type            empty;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       24800;
    }
)
```

Name and type of the surface patches

- The **name** and **base type** of the patch is given by the user.
- In this case the **name** and **base type** was assigned in the dictionary *blockMeshDict*.
- You can change the **name** if you do not like it. Do not use strange symbols or white spaces.
- You can also change the **base type**.
- For instance, you can change the type of the patch **minX** from **wall** to **patch**.

The *constant/polyMesh/boundary* dictionary

```
6
(
    minX
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       22800;
    }
    maxX
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       23200;
    }
    minY
    {
        type            empty;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       23600;
    }
    maxY
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       24000;
    }
    minZ
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       24400;
    }
    maxZ
    {
        type            empty;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       24800;
    }
)
```

inGroups keyword

- This is optional.
- You can erase this information safely.
- It is used to group patches during visualization in ParaView/paraFoam. If you open this mesh in paraFoam you will see that there are two groups, namely: **wall** and **empty**.
- As usual, you can change the name.
- If you want to put a surface patch in two groups, you can proceed as follows:

2(wall wall1)

In this case the surface patch belongs to the group **wall** (which can have another patch) and the group **wall1**

blockMesh guided tutorials

The *constant/polyMesh/boundary* dictionary

```
6
(
    minX
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       22800;
    }
    maxX
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       23200;
    }
    minY
    {
        type            empty;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       23600;
    }
    maxY
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       24000;
    }
    minZ
    {
        type            wall;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       24400;
    }
    maxZ
    {
        type            empty;
        inGroups        List<word> 1(wall);
        nFaces          400;
        startFace       24800;
    }
)
```

nFaces and startFace keywords

- Unless you know what are you doing, **you do not need to change this information.**
- Basically, this is telling you the starting face and ending face of the patch.
- This information is created automatically when generating the mesh or converting the mesh.

blockMesh guided tutorials

Running the case

- To generate the mesh, in the terminal window type:

1.
```
$> foamCleanTutorials
```
2.
```
$> blockMesh
```
3.
```
$> checkMesh
```
4.
```
$> paraFoam
```

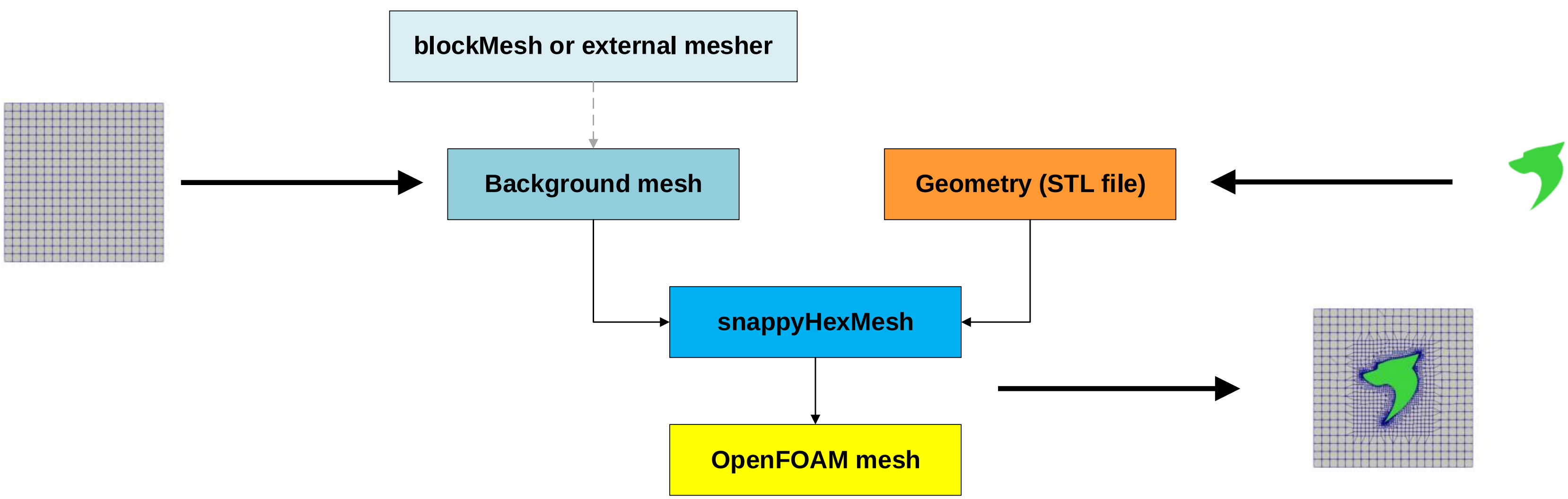
- If you want to visualize the blocking topology, type in the terminal

1.
```
$> paraFoam -block
```

- <u>You can run the rest of the cases following the same steps.</u>

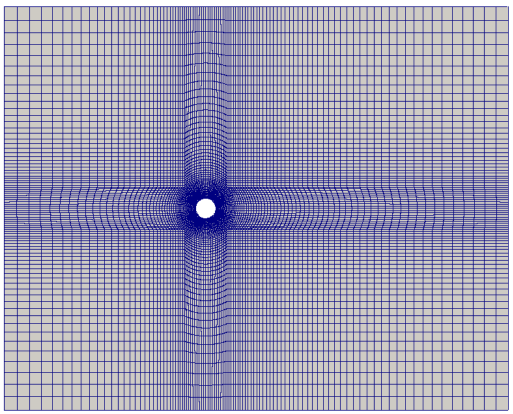
Final remarks on blockMesh

- For the moment, we will limit the use of `blockMesh` to single-block mesh topologies, which are used to run some simple cases.
- Also, single-block meshes are the starting point for `snappyHexMesh`, as shown in the diagram below.
- So, it is extremely important to master this simple mesh topologies.

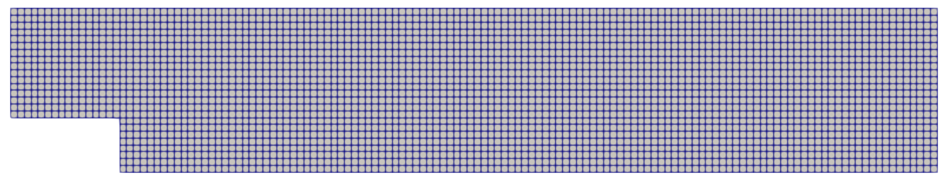


Final remarks on blockMesh

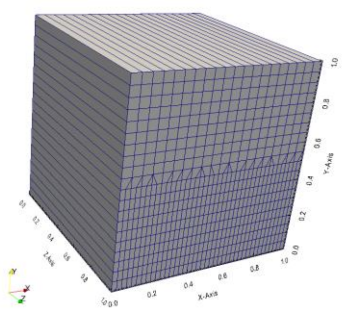
- Have in mind that you can do more elaborated meshes, however, it requires careful setup of the input file.
- It is tricky to generate multi-block meshes with curve edges and stretching.
- With the training material, you will find a set of supplement slides where we explain how to create multi-block meshes, how to add stretching, and how to define curve edges.



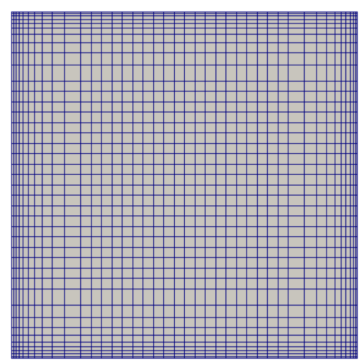
Multi-block mesh with curved edges and multi-stretching



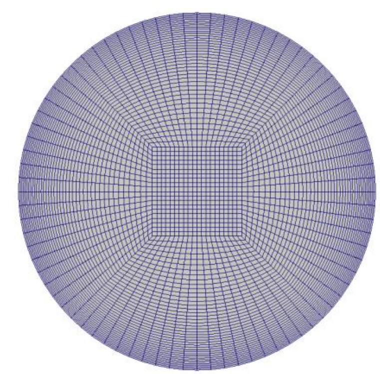
Multi-block mesh with face matching



Multi-block mesh with face merging



Single-block mesh with multi-stretching



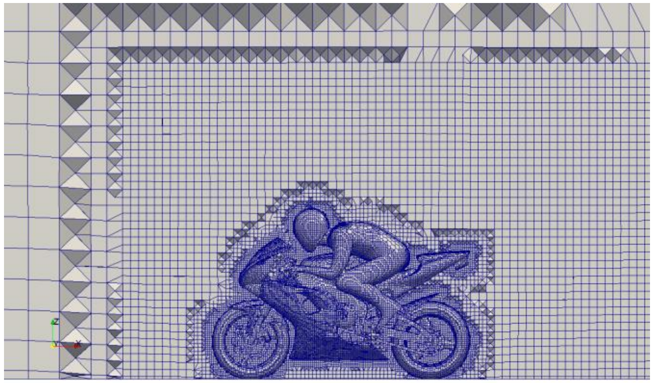
Multi-block mesh with curved edges and multi-stretching

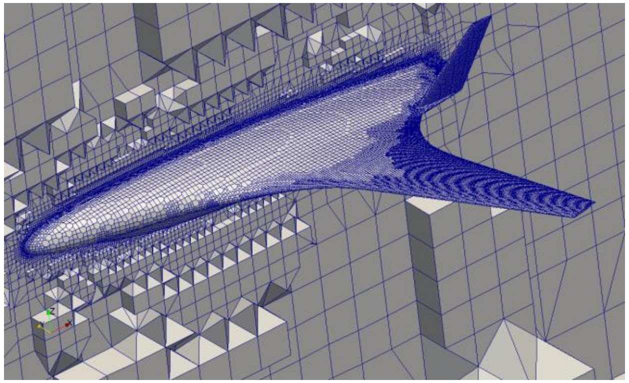
Roadmap

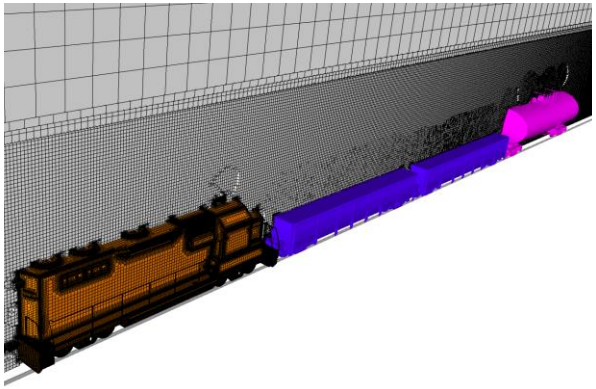
1. ~~Meshing preliminaries~~
2. ~~What is a good mesh?~~
3. ~~Mesh quality assessment in OpenFOAM®~~
4. ~~Mesh generation using blockMesh.~~
5. **Mesh generation using snappyHexMesh.**
6. snappyHexMesh guided tutorials.
7. Mesh conversion
8. Geometry and mesh manipulation utilities

snappyHexMesh

- *"Automatic split hex mesher. Refines and snaps to surface."*
- For complex geometries, the mesh generation utility `snappyHexMesh` can be used.
- The `snappyHexMesh` utility generates 3D meshes containing hexahedra and split-hexahedra from a triangulated surface geometry in Stereolithography (STL) format.
- The mesh is generated from a dictionary file named *`snappyHexMeshDict`* located in the system directory and a triangulated surface geometry file located in the directory **`constant/triSurface`**.



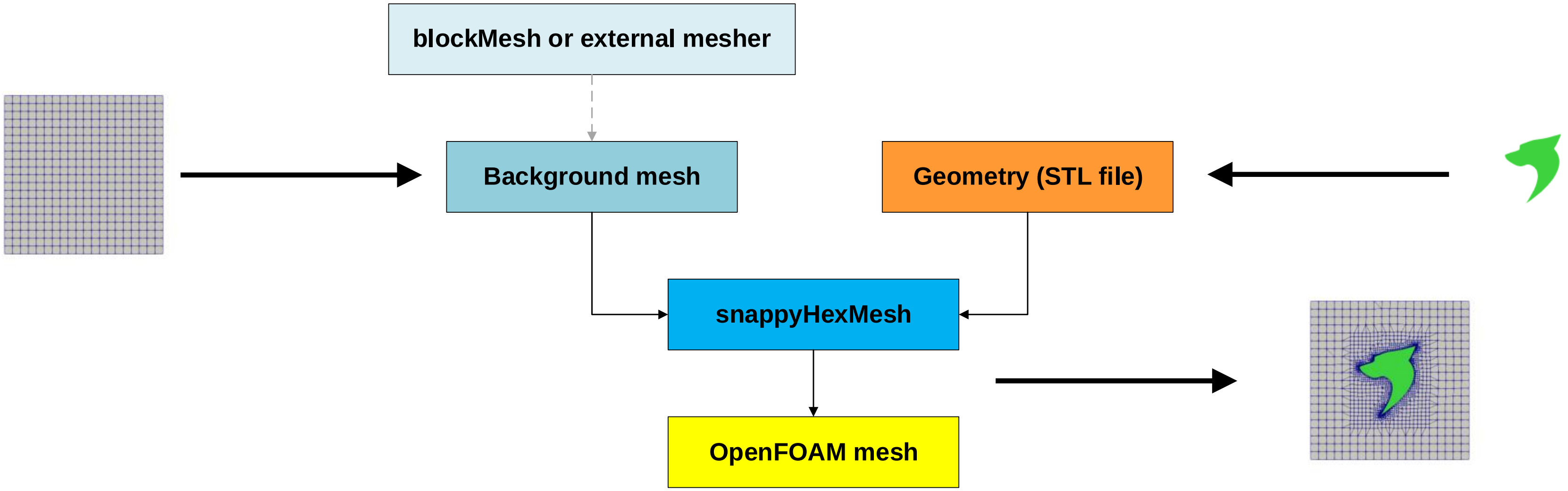




Mesh generation using snappyHexMesh

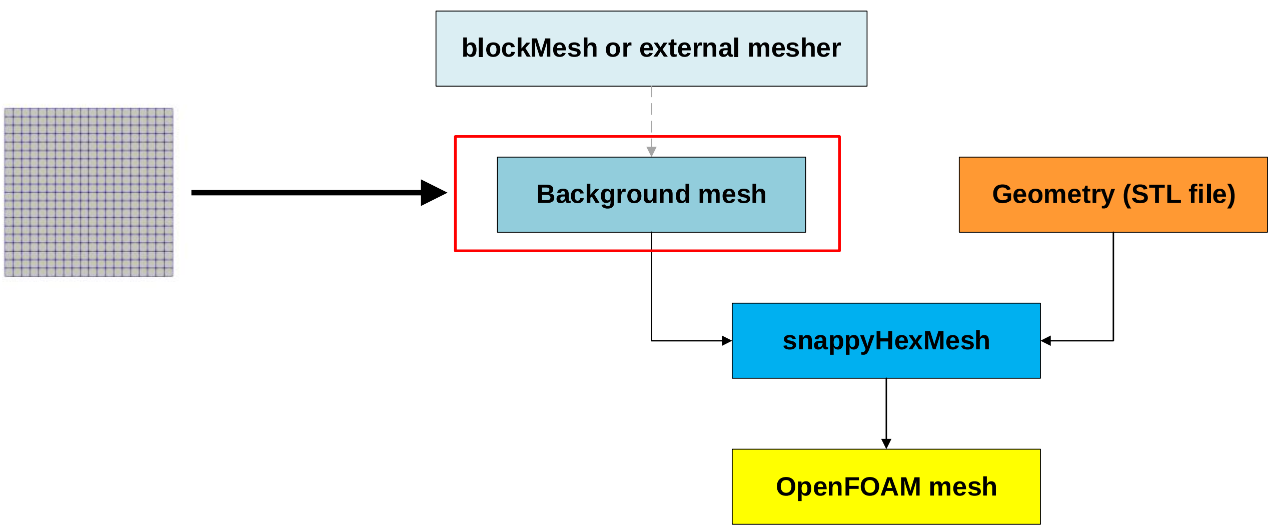
snappyHexMesh workflow

- To generate a mesh with snappyHexMesh we proceed as follows:
    - Generation of a background or base mesh.
    - Geometry definition.
    - Generation of a castellated mesh or cartesian mesh.
    - Generation of a snapped mesh or body fitted mesh.
    - Addition of layers close to the surfaces or boundary layer meshing.
    - Check/enforce mesh quality.



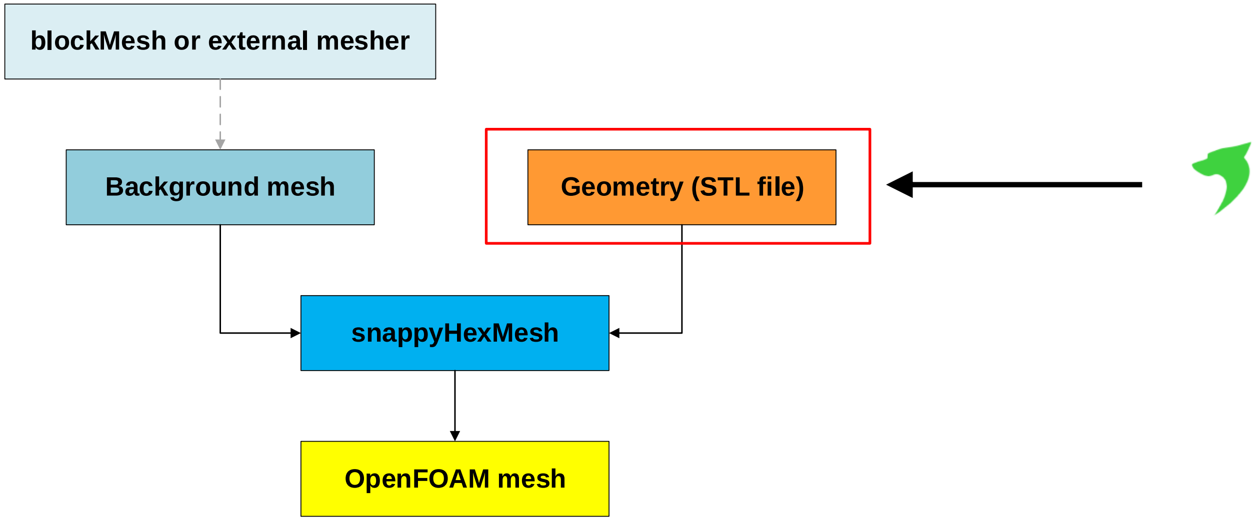
snappyHexMesh workflow – Background mesh

- The background or base mesh can be generated using blockMesh or an external mesher.
- The following criteria must be observed when creating the background mesh:
  - The mesh must consist purely of hexes.
  - The cell aspect ratio should be approximately 1, at least near the STL surface.
  - There must be at least one intersection of a cell edge with the STL surface.
    - However, the more cells that intersect the STL, the better (this means fine background meshes).
  - It is extremely recommended to align the background mesh with the STL surface. However, most of the times this not trivial.



snappyHexMesh workflow – Geometry (STL file)

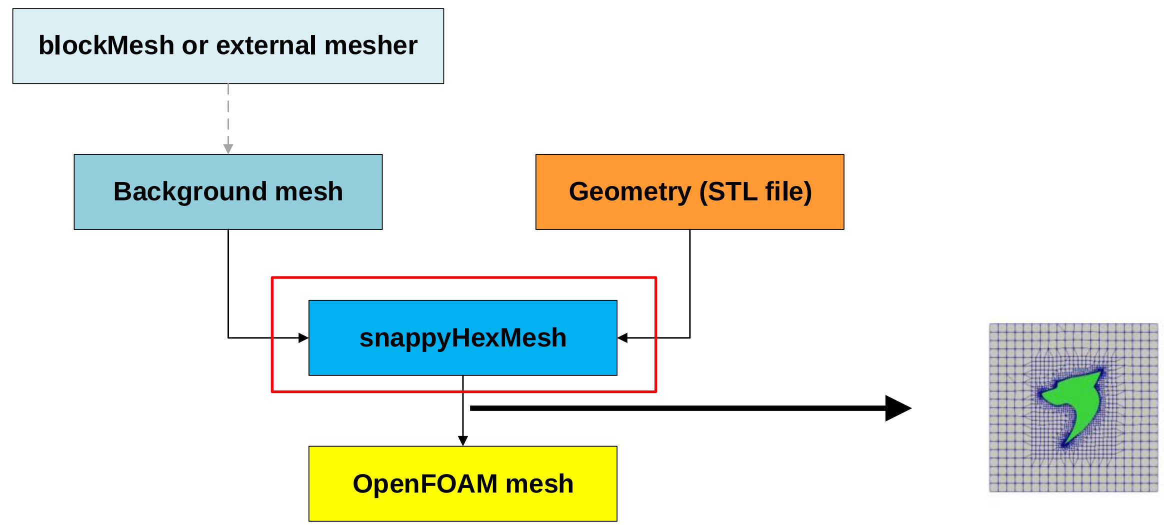
- The STL geometry can be obtained from any geometry modeling tool.
- The STL file can be made up of a single surface describing the geometry, or multiple surfaces that describe the geometry.
- In the case of a STL file with multiple surfaces, we can use local refinement in each individual surface.
  - This gives us more control when generating the mesh.
- The STL geometry is always located in the directory **`constant/triSurface`**



Mesh generation using snappyHexMesh

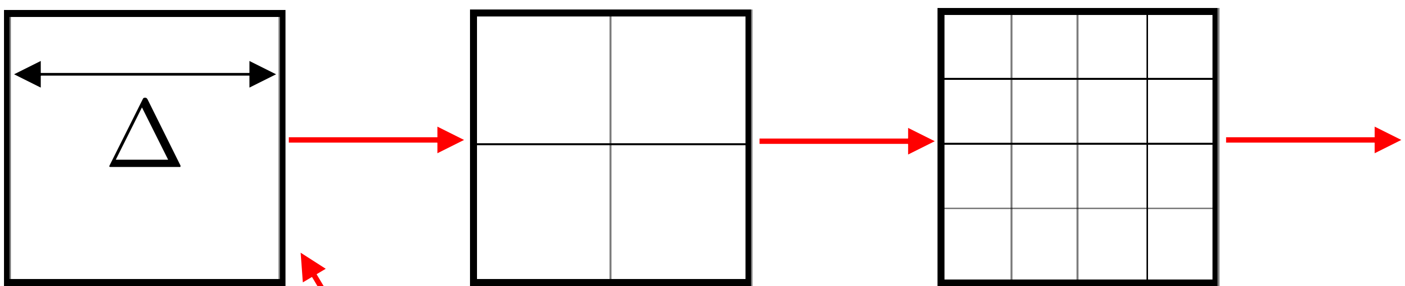
snappyHexMesh workflow

- The meshing utility `snappyHexMesh` reads the dictionary *`snappyHexMeshDict`* located in the directory system.
- The `snappyHexMesh` meshing utility generates the mesh in three steps: castellation, snapping, and boundary layer meshing.
  - All these steps are controlled by the dictionary *`snappyHexMeshDict`*.
- The final mesh is always located in the directory **`constant/polyMesh`**



snappyHexMesh workflow – Cell splitting

- All the volume and surface refinement is done in reference to the background or base mesh.
- snappyHexMesh works by splitting hexahedral cells.



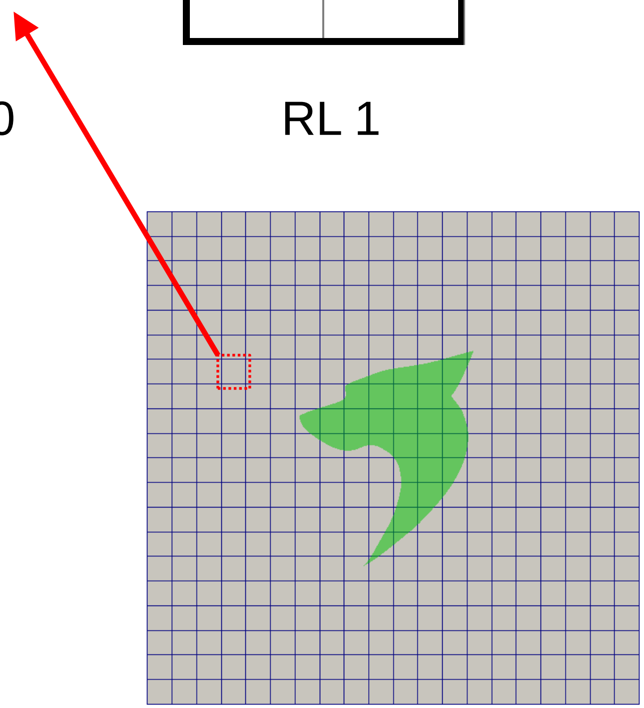
and so on …

Base cell – RL 0

RL 1

RL 2

* RL = refinement level



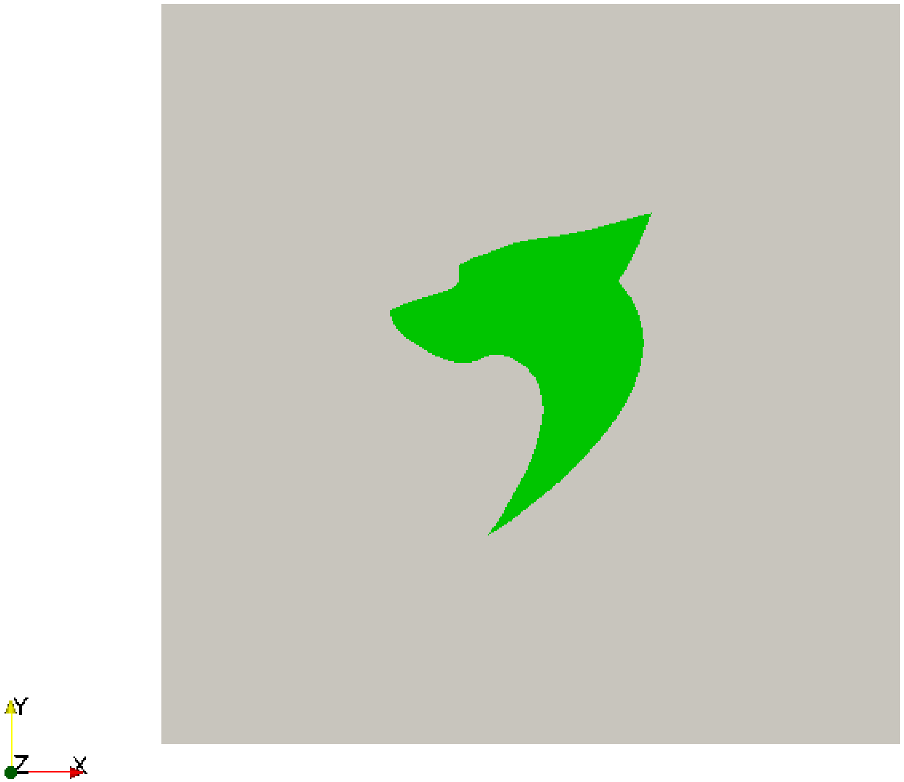
$$\text{Edge size} = ES = \frac{\Delta}{2^n}$$

Note:

- In 2D each quad is subdivided in 4 quads.
- In 3D each hex is subdivided in 8 hexes.

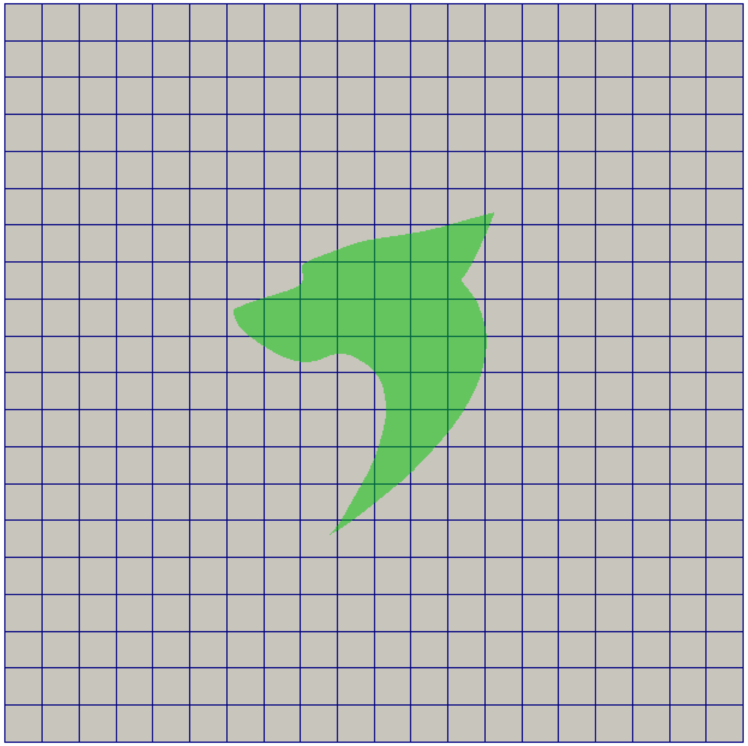
Mesh generation using snappyHexMesh

snappyHexMesh workflow



- The process of generating a mesh using `snappyHexMesh` will be described using this figure.
- The objective is to mesh a rectangular shaped region (shaded grey in the figure) surrounding an object described by a STL surface (shaded green in the figure).
- This is an external mesh (*e.g.*, for external aerodynamics).
- You can also generate an internal mesh (*e.g.*, flow in a pipe).

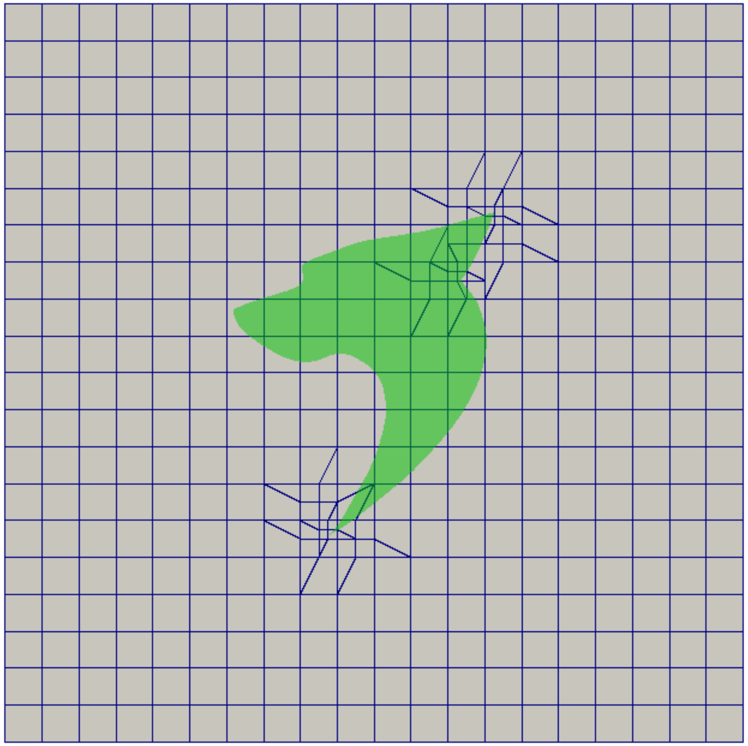
snappyHexMesh workflow



Step 1. Creating the background hexahedral mesh

- Before `snappyHexMesh` is executed the user must create a background mesh of hexahedral cells that fills the entire region as shown in the figure. This can be done by using `blockMesh` or any other mesher.
- The following criteria must be observed when creating the background mesh:
  - The mesh must consist purely of hexes. That is, around the surfaces and volume regions where you are planning to add refinement, the mesh must consist of pure hexes
  - The cell aspect ratio should be approximately 1, at least near the STL surface.
  - There must be at least one intersection of a cell edge with the STL surface.

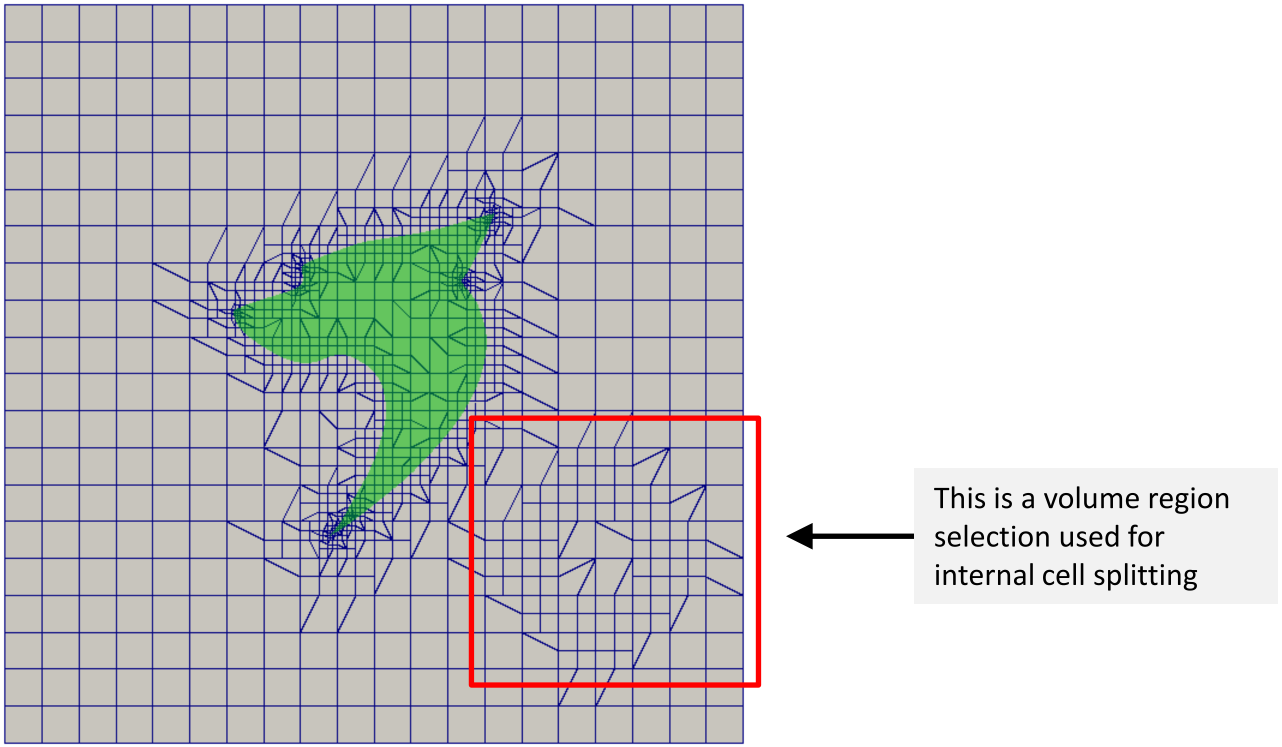
snappyHexMesh workflow



Step 2. Cell splitting at feature edges

- Cell splitting is performed according to the specification supplied by the user in the **castellatedMeshControls** sub-dictionary in the *snappyHexMeshDict* dictionary.
- The splitting process begins with cells being selected according to specified edge features as illustrated in the figure.
- The feature edges can be extracted from the STL geometry file using the utility `surfaceFeatures`.
- The feature edges can also be extracted using paraview.

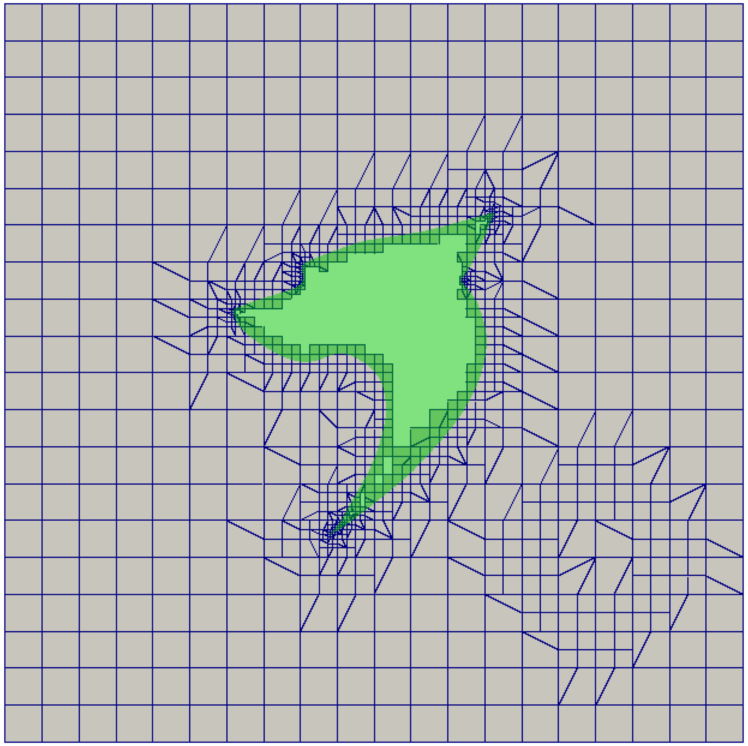
snappyHexMesh workflow



Step 3. Cell splitting at surfaces

- Following feature edges refinement, cells are selected for splitting in the locality of specified surfaces as illustrated in the figure.
- The surface refinement (splitting) is performed according to the specification supplied by the user in the **refinementSurfaces** in the **castellatedMeshControls** sub-dictionary in the *snappyHexMeshDict* dictionary.
- Notice that we added additional internal cells splitting (the region within the red square).
    - This new cell region can be used to define a source term, or it can be put into motion.

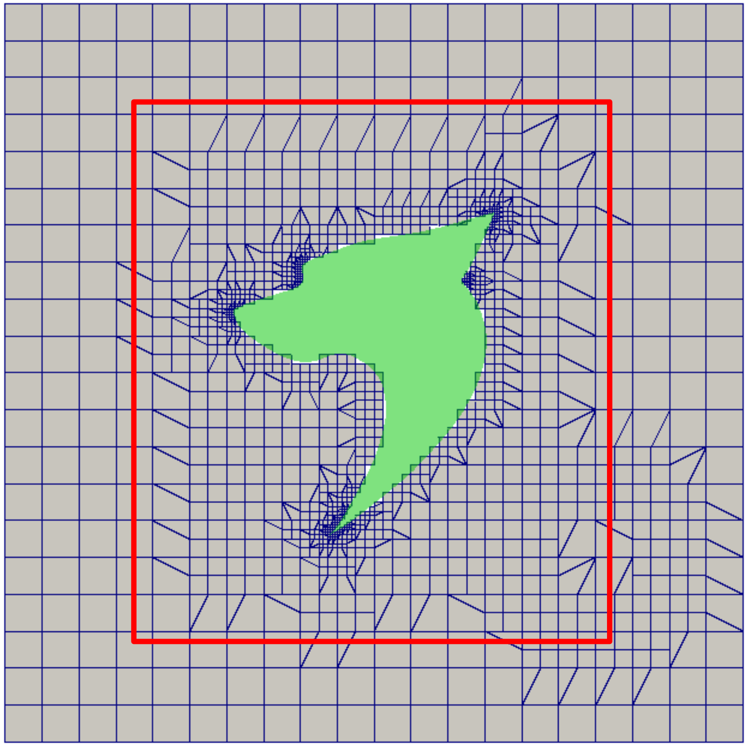
snappyHexMesh workflow



Step 4. Cell removal

- Once the feature edges and surface splitting is complete, a process of cell removal begins.
- The region in which cells are retained are simply identified by a location point within the region, specified by the **locationInMesh** keyword in the **castellatedMeshControls** sub-dictionary in the *snappyHexMeshDict* dictionary.
- Cells are retained if, approximately speaking, 50% or more of their volume lies within the region.
- Be careful to put the **locationInMesh** point in pure hexahedral regions. Do no put it in transition regions as you will get into problems.

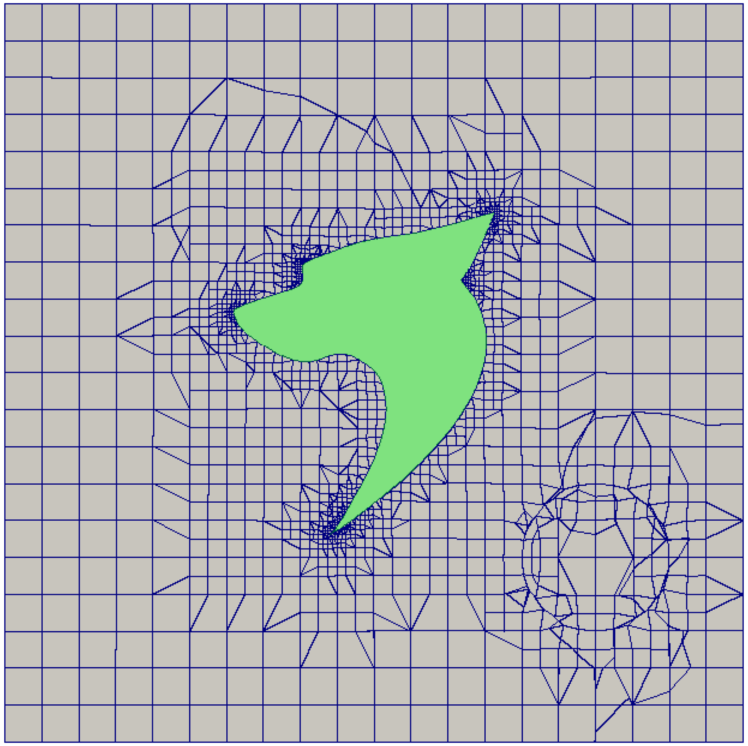
snappyHexMesh workflow



Step 5. Cell splitting in specified regions

- Those cells that lie within one or more specified volume regions can be further split by a region (in the figure, the region within the red rectangle).
- The information related to the refinement of the volume regions is supplied by the user in the **refinementRegions** block in the **castellatedMeshControls** sub-dictionary in the *snappyHexMeshDict* dictionary.
- This is a valid castellated or cartesian mesh that can be used for a simulation.

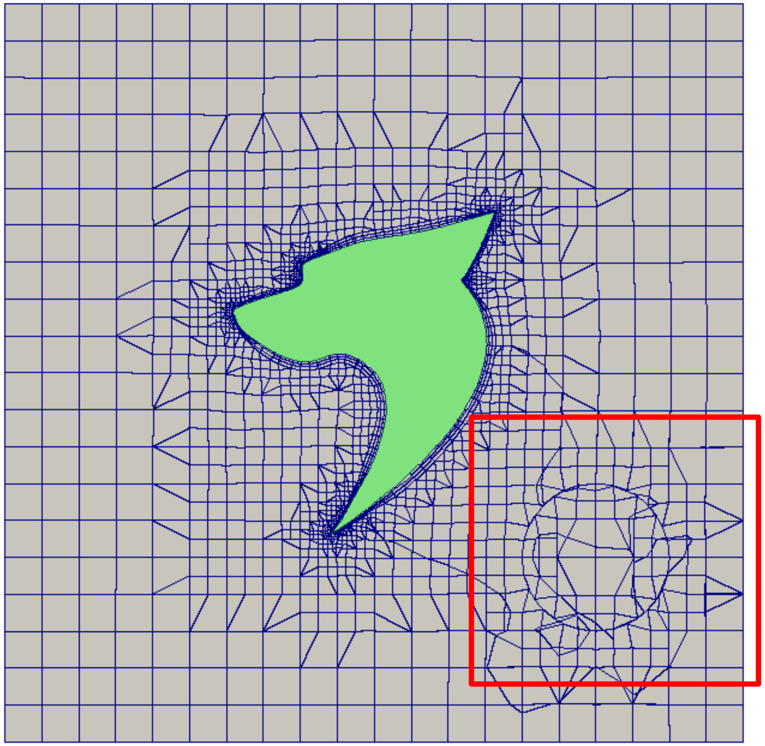
snappyHexMesh workflow



Step 6. Snapping to surfaces

- After deleting the cells in the region specified and refining the volume mesh, the points are snapped on the surface to create a conforming mesh.
- The snapping is controlled by the user supplied information in the **snapControls** sub-dictionary in *snappyHexMeshDict*.
- Sometimes, the recommended **snapControls** options are not enough and you will need to adjust the values to get a good mesh, so it is advisable to save the intermediate steps with a high writing precision (*controlDict*).
- This is a valid snapped or body fitted mesh that can be used for a simulation.

snappyHexMesh workflow



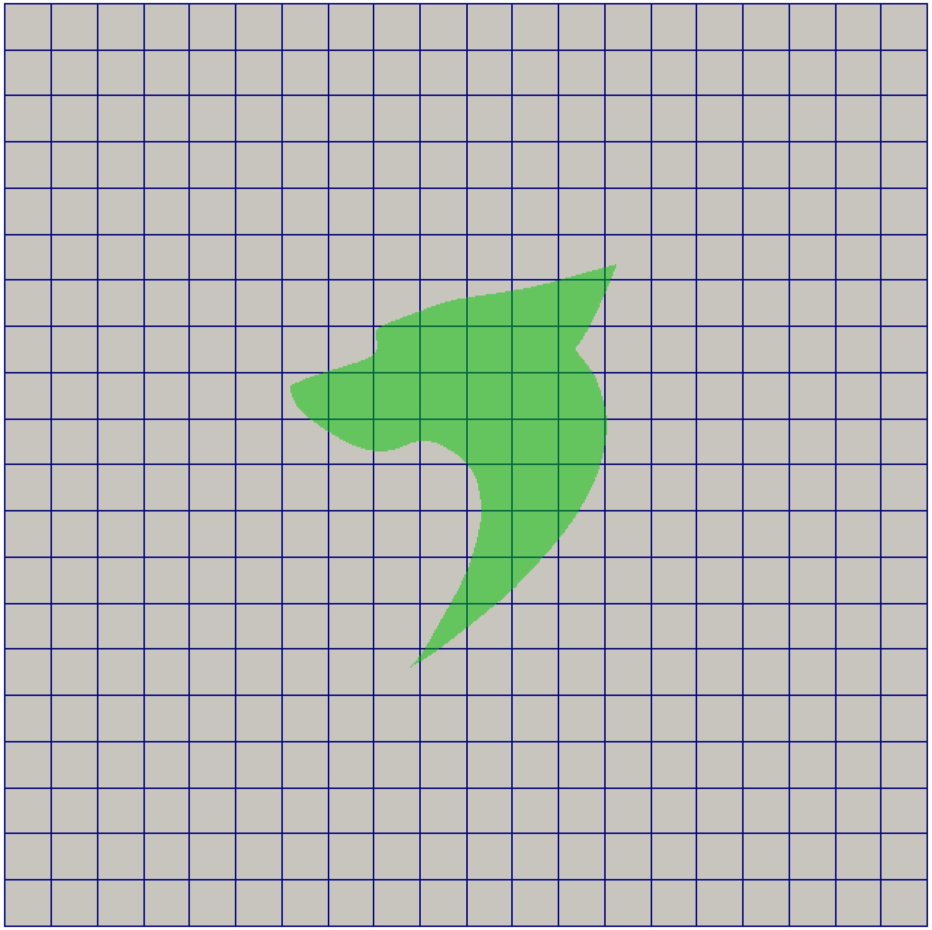
This is a volume region selection used for internal cell splitting

Step 7. Mesh layers

- The mesh output from the snapping stage may be suitable for simulation, although it can produce some irregular cells along boundary surfaces.
- There is an optional stage of the meshing process which introduces boundary layer meshing in selected parts of the mesh.
- This information is supplied by the user in the **addLayersControls** sub-dictionary in the *snappyHexMeshDict* dictionary.
- This is the final step of the mesh generation process using `snappyHexMesh`.
- This is a valid body fitted mesh with boundary layer meshing, that can be used for a simulation.

snappyHexMesh in action

www.wolfdynamics.com/http://www.wolfdynamics.com/training/meshing/image4.gif
/shm/ani.gif



- Let us study the `snappyHexMesh` dictionary in details.
- We are going to work with the case we just saw in action.
- You will find this case in the directory:

```
$PTOFC/101SHM_basic/M101_WD
```

Let us explore the snappyHexMeshDict dictionary.

The dictionary *snappyHexMeshDict* consists of five main sections:

- **geometry**
  Definition of geometry entities to be used for meshing.

- **castellatedMeshControls**
  Definition of feature, surface and volume mesh refinement. Definition of mesh location point. All the mesh refinement is done in this step.

- **snapControls**
  Definition of surface mesh snapping and advanced parameters.

- **addLayersControls**
  Definition of boundary layer meshing and advanced parameters. Only prismatic elements are added in this step, there is no refinement of the surface or volume mesh.

- **meshQualityControls**
  Definition of mesh quality metrics

Mesh generation using snappyHexMesh

Let us explore the snappyHexMeshDict dictionary.

```
castellatedMesh true;          //or false
snap true;                     //or false
addLayers true;                //or false

        geometry
        {
                ...
                ...
        }

        castellatedMeshControls
        {
                ...
                ...
        }

        snapControls
        {
                ...
                ...
        }

        addLayersControls
        {
                ...
                ...
        }

        meshQualityControls
        {
                ...
                ...
        }
```

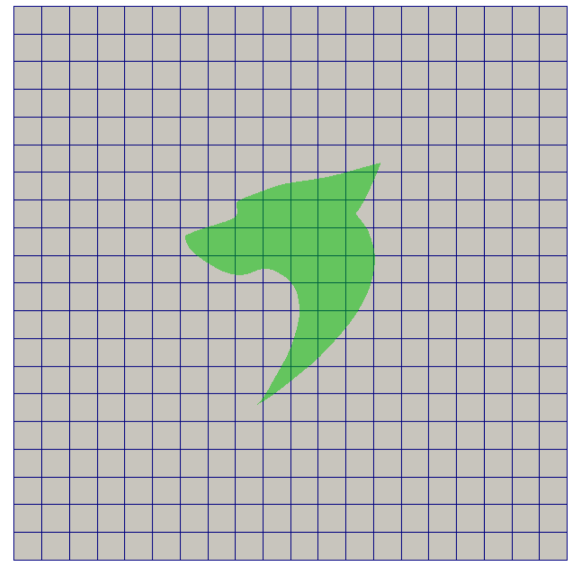
geometry ← **Definition of geometry entities to be used for meshing**

castellatedMeshControls ← **Definition of feature, surface and volume mesh refinement**

snapControls ← **Definition of surface mesh snapping and advanced parameters**

addLayersControls ← **Definition of boundary layer meshing and advanced parameters**

meshQualityControls ← **Definition of mesh quality metrics**



- Open the dictionary *snappyHexMeshDict* with your favorite text editor (we will use gedit).
- The snappyHexMesh dictionary is made up of five sections, namely: **geometry**, **castellatedMeshControls**, **snapControls**, **addLayersControls** and **meshQualityControls**. Each section controls a step of the meshing process.
- In the first three lines we can turn off and turn on the different meshing steps. For example, if we want to generate a body fitted mesh with no boundary layer we should proceed as follows:

castellatedMesh true;
snap true;
addLayers false;

Mesh generation using snappyHexMesh

Let us explore the snappyHexMeshDict dictionary.

```
castellatedMesh true;           //or false
snap true;                      //or false
addLayers true;                 //or false


        geometry
        {
                ...
                ...
        }

        castellatedMeshControls
        {
                ...
                ...
        }

        snapControls
        {
                ...
                ...
        }

        addLayersControls
        {
                ...
                ...
        }

        meshQualityControls
        {
                ...
                ...
        }
```

It can be located In a separated file

- Have in mind that there are more than 60 parameters to control in `snappyHexMeshDict` dictionary.
- Adding the fact that there is no native GUI, it can be quite tricky to control the mesh generation process.
- Nevertheless, `snappyHexMesh` generates very good hexa dominant meshes.
- Hereafter, we will only comment on the most important parameters.
- The parameters that you will find in the *`snappyHexMeshDict`* dictionaries distributed with the tutorials, in our opinion are robust and will work most of the times.

Mesh generation using snappyHexMesh

Let us explore the snappyHexMeshDict dictionary.

Geometry controls section

```
geometry
{

    wolfExtruded.stl
    {
        type triSurfaceMesh;
        name wolf;

        regions
        {
            wolflocal
            {
                name wolf_wall;
            }
        }
    }

    box
    {
        type searchableBox;
        min (-100.0 -120.0 -50.0 );
        max (100.0 120.0 150.0 );
    }

    sphere
    {
        type searchableSphere;
        centre (120.0 -100.0 50.0 );
        radius 40.0;
    }

}
```

- wolfExtruded.stl: STL file to read
- name wolf;: Name of the surface inside snappyHexMesh
- regions: Use this option if you have a STL with multiple patches defined
- wolflocal: This is the name of the region or surface patch in the STL
- name wolf_wall;: User-defined patch name. This is the final name of the patch
- box: Name of geometrical entity
- sphere: Name of geometrical entity
- type searchableSphere;: Note 1

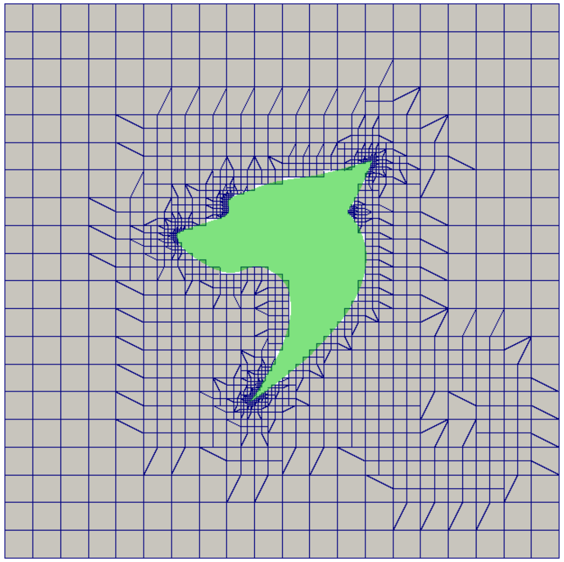
- In this section we read in the STL geometry. Remember, the input geometry is always located in the directory `constant/triSurface`
- We can also define geometrical entities that can be used to refine the mesh, create regions, or generate baffles.
- You can add multiple STL files.
- If you do not give a name to the surface, it will take the name of the STL file.
- The geometrical entities are created inside `snappyHexMesh`.

Note 1:
If you want to know what geometrical entities are available, just misspelled something in the **type** keyword.

Let us explore the snappyHexMeshDict dictionary.

```
castellatedMeshControls
{
        //Refinement parameters
        maxLocalCells  100000;
        maxGlobalCells 2000000;          <---- Note 1
        nCellsBetweenLevels 3;
                ...
                ...

        //Explicit feature edge refinement
        features                         <---- Dictionary block
        (
                ...
                ...
        );

        //Surface based refinement
        refinementSurfaces               <---- Dictionary block
        {
                ...
                ...
        }

        //Region-wise refinement
        refinementRegions                <---- Dictionary block
        {
                ...
                ...
        }

        //Mesh selection
        locationInMesh (-100.0 0.0 50.0 );   <---- Note 2
}
```

Castellated mesh controls section



- In the **castellatedMeshControls** section, we define the global refinement parameters, explicit feature edge refinement, surface-based refinement, region-wise refinement and the material point.
- **In this step, we are generating the castellated mesh.**

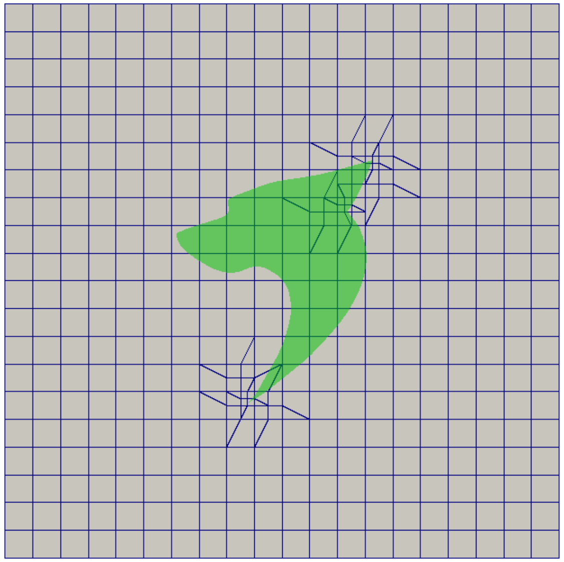
Note 1:
Maximum number of cells in the domain. If the mesher reach this number, it will not add more cells.

Note 2:
The material point indicates where we want to create the mesh, that is, inside or outside the body to be meshed.

Let us explore the snappyHexMeshDict dictionary.

```
castellatedMeshControls
{

        // Refinement parameters
        maxLocalCells  100000;
        maxGlobalCells 2000000;
        minRefinementCells 0;
        maxLoadUnbalance 0.10;
        nCellsBetweenLevels 3;          <-- Note 1

        //Local curvature and
        //feature angle refinement
        resolveFeatureAngle 30;         <-- Note 2

        planarAngle 30;

        allowFreeStandingZoneFaces true;


        //Explicit feature edge refinement
        features     <-- Dictionary block
        (
                {
                        file "wolfExtruded.eMesh";   <-- Note 3
                        level 2;
                }
        );

        ...
        ...
        ...

}
```

Castellated mesh controls section



Note 1:
This parameter controls the transition between cell refinement levels.

Note 2:
This parameter controls the local curvature refinement. The higher the value, the less features it captures. For example, if you use a value of 100 it will not add refinement in high curvature areas. It also controls edge feature snapping; high values will not resolve sharp angles in surface intersections.

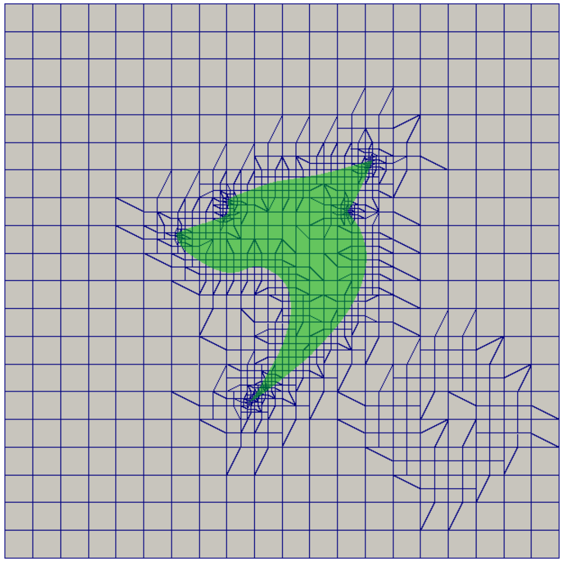
Note 3:
This file is automatically created when you use the utility `surfaceFeatures`. The file is located in the directory **`constant/triSurface`**

Mesh generation using snappyHexMesh

Let us explore the snappyHexMeshDict dictionary.

```
castellatedMeshControls
{

        ...
        ...
        ...

        //Surface based refinement
        refinementSurfaces                 <-- Dictionary block
        {

                //wolf was defined in the geometry section
                wolf                       <-- Note 1
                {

                        level (1 1);        //Global refinement

                        regions            <-- Note 2
                        {

                                wolflocal  <-- Note 3
                                {
                                        level (2 4);     <-- Local refinement

                                        patchInfo
                                        {
                                                type wall;    <-- Note 4

                                        }
                                }
                        }
                }
        ...
        ...
        }
```

Castellated mesh controls section



Note 1:
The surface wolf was defined in the geometry section.

Note 2:
The region **wolflocal** was defined in the geometry section.

Note 3:
Named region in the STL file. This refinement is local.
To use the surface refinement in the regions, the local regions must exist in STL file. We created a pointer to this region in the **geometry** section.

Note 4:
You can only define patches of type wall or patch.

Mesh generation using snappyHexMesh

Let us explore the snappyHexMeshDict dictionary.

```
castellatedMeshControls
{

    //Surface based refinement
    refinementSurfaces
    {

        …
        …
        …
            //This surface or geometrical entity
            //was defined in geometry section
            sphere
            {
                level (1 1);

                faceZone face_inner;
                cellZone cell_inner;

                mode inside;

                //faceType internal;

            }

    }

    …
    …
}
```

Dictionary block

Note 1

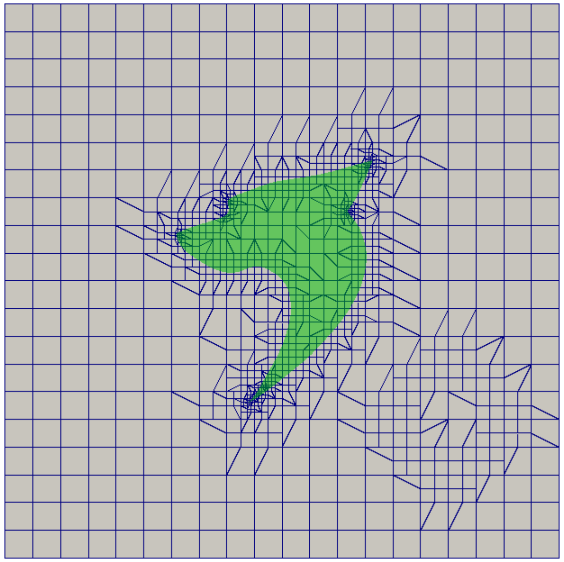
Name of faceZone

Name of cellZone

Create inner cellZone

Create internal faces from faceZone
Uncomment to create the internal faceZone

Castellated mesh controls section



Note 1:
Optional specification of what to do with **faceZone** faces:

internal: keep them as internal faces (default)
baffle: create baffles from them. This gives more freedom in mesh motion
boundary: create free-standing boundary faces (baffles but without the shared points)

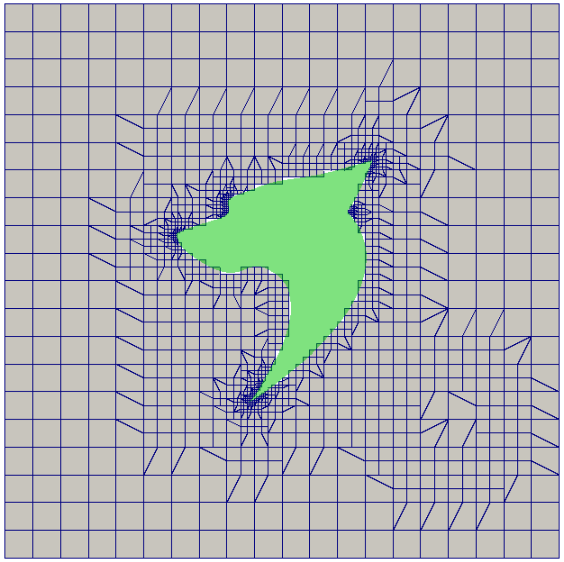
e.g., **faceType** internal;

Mesh generation using snappyHexMesh

Let us explore the snappyHexMeshDict dictionary.

```
castellatedMeshControls
{

        ...
        ...
        ...

        //Region-wise refinement
        refinementRegions                    <- Dictionary block
        {

                //This region or geometrical entity
                //was defined in the geometry section

                box                          <- Note 1
                {
                        mode inside;
                        levels  (( 1 1 ));
                }

        }


        //Mesh selection
        locationInMesh (-100.0 0.0 50.0 );

}
```

Castellated mesh controls section



Note 1:

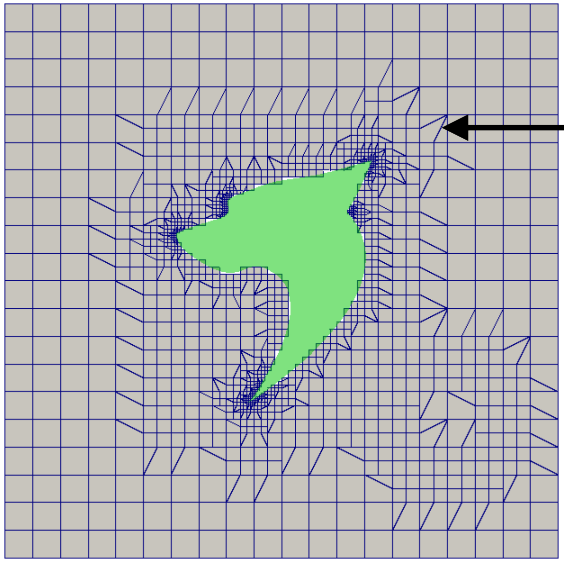
- This region or geometrical entity was created in the **geometry** section.
- You can use open or close geometries.
- You can use STL files.
    - But you cannot use regions defined in the STL.

Let us explore the snappyHexMeshDict dictionary.

```
castellatedMeshControls
{

        ...
        ...
        ...

        //Region-wise refinement
        refinementRegions
        {

                //This region or geometrical entity
                //was defined in the geometry section

                box
                {
                        mode inside;
                        levels  (( 1 1 ));
                }

        }


        //Mesh selection
        locationInMesh (-100.0 0.0 50.0 );

}
```

Dictionary block

Castellated mesh controls section



Do not put the locationInMesh point in this region because the cells are not hexahedral

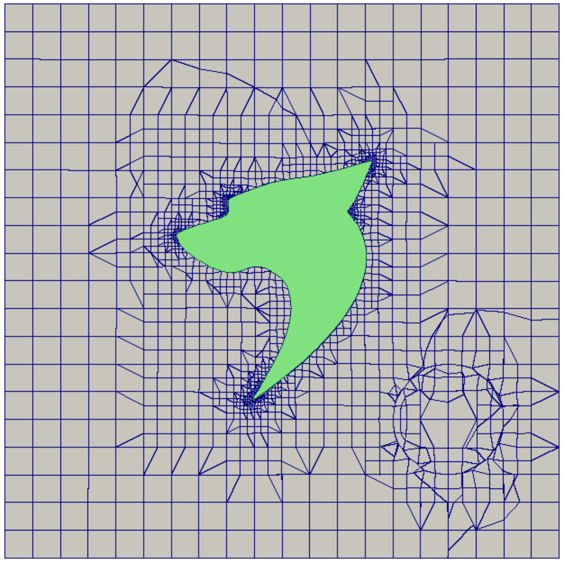
- **This point defines where do you want the mesh.**
- **According to the point locations, the mesh can be internal or external.**
  - **If the point is inside the STL, it is an internal mesh.**
  - **If the point is inside the background mesh and outside the STL it is an external mesh.**
- **Put this point in pure hexahedral regions. Do no put it in transition regions as it will give you problems.**

- At this point we have a valid mesh (cartesian)

Let us explore the snappyHexMeshDict dictionary.

```
snapControls
{

    //Number of patch smoothing iterations
    //before finding correspondence to surface
    nSmoothPatch 3;

    tolerance 2.0;

    //- Number of mesh displacement relaxation
    //iterations.
    nSolveIter 30;          <---- Note 1

    //- Maximum number of snapping relaxation
    //iterations. Should stop before upon
    //reaching a correct mesh.
    nRelaxIter 5;           <---- Note 2

    // Feature snapping

        //Number of feature edge snapping iterations.
        nFeatureSnapIter 10;           <---- Note 3

        //Detect (geometric only) features by
        //sampling the surface (default=false).
        implicitFeatureSnap false;

        // Use castellatedMeshControls::features
        // (default = true)
        explicitFeatureSnap true;

        multiRegionFeatureSnap false;

}
```

Snap mesh controls section



Note 1:
The higher the value the better the body fitted mesh. The recommended value is 30. If you are having problems with the mesh quality (related to the snapping step), try to increase this value to 100. Have in mind that this will increase the meshing time.

Note 2:
Increase this value to improve the quality of the body fitted mesh.

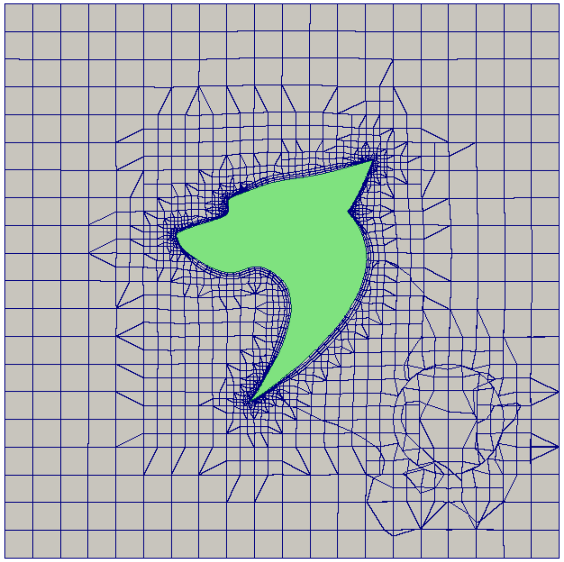
Note 3:
Increase this value to improve the quality of the edge features.

- **In this step, we are generating the body fitted mesh.**

Let us explore the snappyHexMeshDict dictionary.

```
addLayersControls
{
        //Global parameters
        relativeSizes true;
        expansionRatio 1.2;
        finalLayerThickness 0.5;
        minThickness 0.01;

        layers                                  Note 1
        {
                wolf_wall                       Note 2
                {
                        nSurfaceLayers 3;
                        //Local parameters
                        //expansionRatio        1.3;
                        //finalLayerThickness   0.3;
                        //minThickness          0.1;
                }
        }
        // Advanced settings
        nGrow 0;
        featureAngle 130;                       Note 3
        maxFaceThicknessRatio 0.5;
        nSmoothSurfaceNormals 1;
        nSmoothThickness 10;
        minMedianAxisAngle 90;
        maxThicknessToMedialRatio 0.3;
        nSmoothNormals 3;
        slipFeatureAngle 30;
        nRelaxIter 5;
        nBufferCellsNoExtrude 0;
        nLayerIter 50;
        nRelaxedIter 20;
}
```

Boundary layer mesh controls section



Note 1:
In this section we select the patches where we want to add the layers. We can add multiple patches (if they exist).

Note 2:
This patch was created in the **geometry** section.

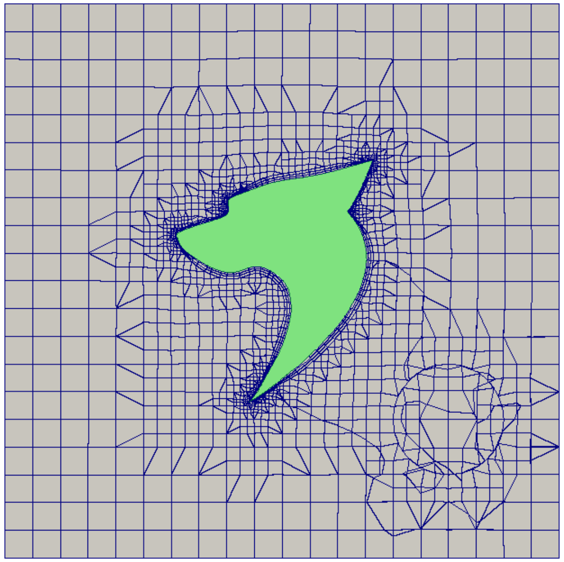
Note 3:
Specification of feature angle above which layers are collapsed automatically.

- **In this step, we are generating the boundary layer mesh.**

Let us explore the snappyHexMeshDict dictionary.

```
meshQualityControls
{
        maxNonOrtho 75;                     <-- Note 1

        maxBoundarySkewness 20;

        maxInternalSkewness 4;              <-- Note 2

        maxConcave 80;

        minVol 1E-13;

        //minTetQuality  1e-15;
        minTetQuality -1e+30;

        minArea -1;

        minTwist 0.02;

        minDeterminant 0.001;

        minFaceWeight 0.05;

        minVolRatio 0.01;

        minTriangleTwist -1;

        minFlatness 0.5;

        nSmoothScale 4;

        errorReduction 0.75;
}
```

Mesh quality controls section



Note 1:
Maximum non-orthogonality angle.

Note 2:
Maximum skewness angle.

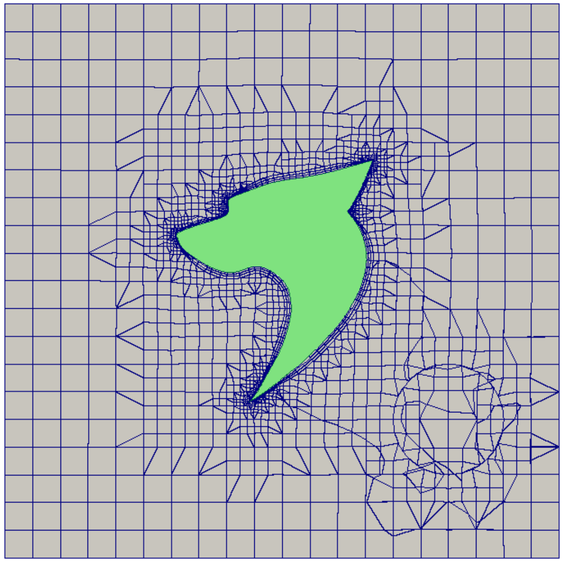
- During the mesh generation process, the mesh quality is continuously monitored.
- The mesher `snappyHexMesh` will try to generate a mesh using the mesh quality parameters defined by the user.
- If a mesh motion or topology change introduces a poor quality cell or face the motion or topology change is undone to revert the mesh back to a previously valid error free state.

Mesh generation using snappyHexMesh

Let us explore the snappyHexMeshDict dictionary.

```
debugFlags
(
        // write intermediate meshes
        mesh

        // write current mesh intersections as .obj files
        intersections

        // write information about explicit feature edge
        // refinement
        featureSeeds

        // write attraction as .obj files
        attraction

        // write information about layers
        layerInfo
);


writeFlags
(
        // write volScalarField with cellLevel for
        // postprocessing
        scalarLevels

        // write cellSets, faceSets of faces in layer
        layerSets

        // write volScalarField for layer coverage
        layerFields
);
```

Mesh debug and write controls sections



- At the end of the dictionary, you will find the sections: **debugFlags** and **writeFlags**
- By default, they are commented. If you uncomment them, you will enable debug information.
- **debugFlags** and **writeFlags** will produce a lot of outputs that you can use to post process and troubleshoot the different steps of the meshing process.

Let us explore the snappyHexMeshDict dictionary.

- Of all entries in the `snappyHexMeshDict` dictionary, probably the most important ones are those related to the snap control (**snapControls** section).
- Remember, there are no default values, so you will need to play around to find the best parameters, which at the same time are likely to be problem dependent.
- We recommend you use the following values,

Recommended values

```
snapControls
{
        nSmoothPatch            3;
        tolerance               2.0;
        nSolveIter              30;
        nRelaxIter              5;
        nFeatureSnapIter        10;
        implicitFeatureSnap false;
        explicitFeatureSnap true;
}
```

Improved values

```
snapControls
{
        nSmoothPatch            3;
        tolerance               2.0;
        nSolveIter              100;
        nRelaxIter              20;
        nFeatureSnapIter        100;
        implicitFeatureSnap false;
        explicitFeatureSnap true;
}
```

- A word of caution, these values are based on our experience and do not represent best standard practices when generating the mesh using snappyHexMesh.

Let us explore the snappyHexMeshDict dictionary.

- If the recommended values do not generate a good mesh, try to use the improved values.
- However, do not immediately use the advised maximum values as this will considerably increase the meshing time.
- Instead, starting from the initial recommended values, you can double the reference values.
- Usually, after one doubling of the parameters you will fix most of the issues.
- You can keep doubling until reaching the advised maximum values.
- If after reaching the advised maximum values you are still getting meshing problems, it is advised to increase the surface mesh refinement or the background mesh resolution.

Improved values (after one doubling iteration)

```
snapControls
{
        nSmoothPatch          3;
        tolerance             2.0;
        nSolveIter            50;
        nRelaxIter            10;
        nFeatureSnapIter      20;
        implicitFeatureSnap false;
        explicitFeatureSnap true;
}
```

Improved values (advised maximum values)

```
snapControls
{
        nSmoothPatch          3;
        tolerance             2.0;
        nSolveIter            100;
        nRelaxIter            20;
        nFeatureSnapIter      100;
        implicitFeatureSnap false;
        explicitFeatureSnap true;
}
```

Let us explore the snappyHexMeshDict dictionary.

- Another important entry in the `snappyHexMeshDict` dictionary, is the **resolveFeatureAngle** in the **castellatedMeshControls** section.
- The parameter **resolveFeatureAngle** controls the local curvature refinement.
- The higher the value, the less features it captures. For example, if you use a value of 100 it will not add refinement in high curvature areas.
- This parameter also influence edge feature snapping. As for surface curvature, high values will not resolve sharp angles in surface intersections.
- Usually, a value of 30 is a good choice. If you want to resolve more feature, simply reduce this value.

Recommended starting value

```
castellatedMeshControls
{
        ...
        ...
        ...

        resolveFeatureAngle 30;

        ...
        ...
        ...
}
```

Let us explore the snappyHexMeshDict dictionary.

- Regarding the inflation layer parameters (**addLayersControls**), in our experience the most important parameters are **featureAngle** and **featureAngle**.
- To set these values, you can follow the same guidelines as the ones we defined for **snapControls**.
- It is important to stress that we are referring to the control parameters related to the mesh quality and iterative relaxation.
- The parameters related to the inflation layer thickness are much more important.
- We will demonstrate this using an excel worksheet.

Recommended values

```
addLayersControls
{
    …

    featureAngle              130;
    maxFaceThicknessRatio     0.5;

    …
}
```

Improved values (advised maximum values)

```
addLayersControls
{
    …

    featureAngle              330;
    maxFaceThicknessRatio     1;

    …
}
```

Mesh generation using snappyHexMesh

Let us generate the mesh of the wolf dynamics logo.

- This tutorial is located in the directory:
  - **`$PTOFC/101SHM_basic/M101_WD`**

- In this case we are going to generate a body fitted mesh with boundary layer. This is an external mesh.
- Before generating the mesh take a look at the dictionaries and files that will be used.
- These are the dictionaries and files that will be used.
  - *`system/snappyHexMeshDict`*
  - *`system/surfaceFeaturesDict`*
  - *`system/meshQualityDict`*
  - *`system/blockMeshDict`*
  - *`constant/triSurface/wolfExtruded.stl`*
  - *`constant/triSurface/wolfExtruded.eMesh`*

- The file *`wolfExtruded.eMesh`* is generated after using the utility `surfaceFeatures`, which reads the dictionary *`surfaceFeaturesDict`*.

Mesh generation using snappyHexMesh

Let us generate the mesh of the wolf dynamics logo.

- To generate the mesh, in the terminal window type:

1. `$> foamCleanTutorials`
2. `$> blockMesh`
3. `$> surfaceFeatures`
4. `$> snappyHexMesh`
5. `$> checkMesh -latestTime`

- To visualize the mesh, in the terminal window type:
  - `$> paraFoam`

- Remember to use the VCR controls in paraView/paraFoam to visualize the mesh intermediate steps.

Mesh generation using snappyHexMesh

Let us generate the mesh of the wolf dynamics logo.

- In the case directory you will find the time folders **`1`**, **`2`**, and **`3`**, which contain the castellated mesh, snapped mesh and boundary layer mesh respectively.
- In this case, `snappyHexMesh` automatically saved the intermediate steps.
- Before running the simulation, remember to transfer the solution from the latest mesh to the directory **`constant/polyMesh`**, in the terminal type:

1. `$> cp 3/polyMesh/* constant/polyMesh`
2. `$> rm -rf 1`
3. `$> rm -rf 2`
4. `$> rm -rf 3`
5. `$> checkMesh -latestTime`

Mesh generation using snappyHexMesh

Let us generate the mesh of the wolf dynamics logo.

- If you want to avoid the additional steps of transferring the final mesh to the directory **`constant/polyMesh`** by not saving the intermediate steps, you can proceed as follows:

  - `$> snappyHexMesh -overwrite`

- When you proceed in this way, `snappyHexMesh` automatically saves the final mesh in the directory **`constant/polyMesh`**.
- Have in mind that you will not be able to visualize the intermediate steps.
- Also, you will not be able to restart the meshing process from a saved state (castellated or snapped mesh).
- Unless it is strictly necessary, from this point on we will not save the intermediate steps.

The *constant/polyMesh/boundary* file

- At this point, we have a valid mesh to run a simulation.
- Have in mind that before running the simulation you will need to set the boundary and initial conditions in the directory **0**.
- Let us talk about the *constant/polyMesh/boundary* file,
  - First of all, this file is automatically generated after you create the mesh, or you convert it from a third-party format.
  - In this file, the geometrical information related to the **base type** patch of each boundary of the domain is specified.
  - The **base type** boundary condition is the actual surface patch where we are going to apply a **numerical type** boundary condition.
  - The **numerical type** boundary condition assign a field value to the surface patch (**base type**).
  - You define the **numerical type** patch (or the value of the boundary condition), in the directory **0** or time directories.
  - The **name** and **base type** of the patches was defined in the dictionaries *blockMeshDict* and *snappyHexMeshDict*.
  - You can change the **name** if you do not like it. Do not use strange symbols or white spaces.
  - You can also change the **base type**. For instance, you can change the type of the patch **maxY** from **wall** to **patch**.

The *constant/polyMesh/boundary* file

- At this point, we have a valid mesh to run a simulation.
- Have in mind that before running the simulation you will need to set the boundary and initial conditions in the directory **0**.
- The **name** and **base type** information of the boundary patches is saved in the file *constant/polyMesh/boundary*.
- Remember, the **base type** (patch type defined in the file *constant/polyMesh/boundary*) and the **numerical type** of the boundary conditions (patch type defined in the fields dictionary in the directory **0**), must be compatible.
- You also need to use the same naming convention. That is, the name of the patches defined in the file *constant/polyMesh/boundary* and the name of the patches defined in the files inside the directory **0**, must be the same.

Mesh generation using snappyHexMesh

The *constant/polyMesh/boundary* file

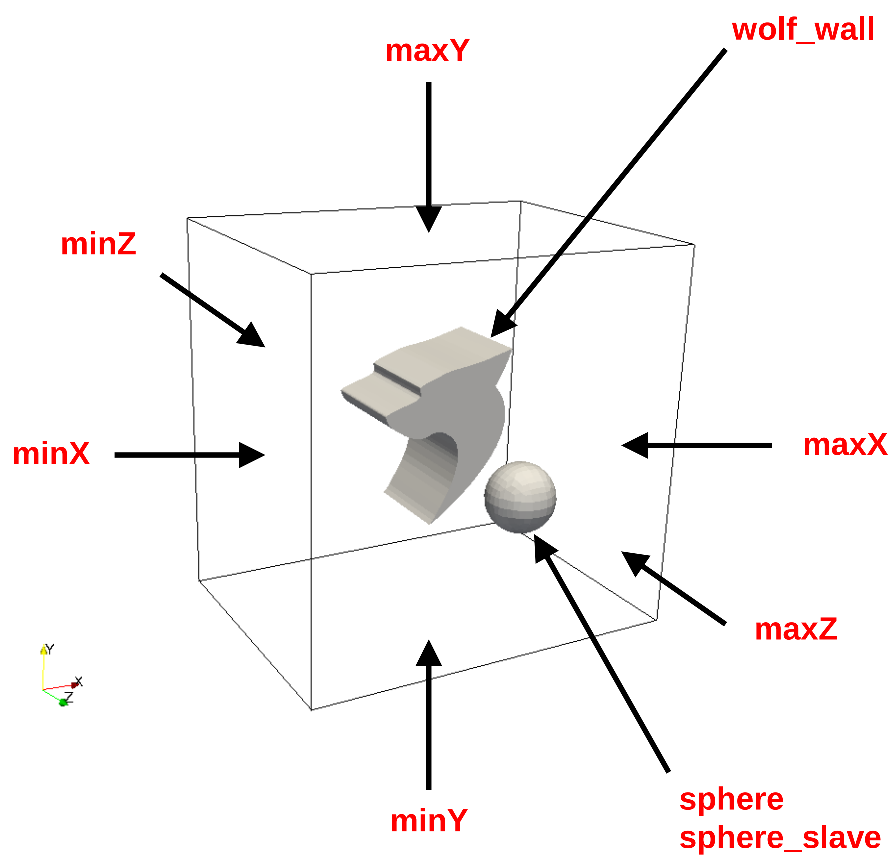
- First of all, this file is automatically generated after you create the mesh, or you convert it from a third-party format.
- In this file, the geometrical information related to the **base type** patch of each boundary of the domain is specified.
- The **base type** boundary condition is the actual surface patch where we are going to apply a **numerical type** boundary condition (or numerical boundary condition).
- The **numerical type** boundary condition assign a field value to the surface patch (**base type**).
- You define the **numerical type** patch (or the value of the boundary condition), in the directory **0** or time directories.
- The **name** and **base type** of the patches was defined in the dictionaries *blockMeshDict* and *snappyHexMeshDict*.
- You can change the **name** if you do not like it. Do not use strange symbols or white spaces.
- You can also change the **base type**. For instance, you can change the type of the patch **maxY** from **wall** to **patch**.

Mesh generation using snappyHexMesh

The *constant/polyMesh/boundary* file

```
9
(
    minX
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       466399;
    }
    maxX
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       466799;
    }
    minY
    {
        type            empty;
        inGroups        1(wall);
        nFaces          400;
        startFace       467199;
    }
    maxY
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       467599;
    }
    minZ
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       467999;
    }
```

Number of surface patches
In the list bellow there must be 9 patches definition.



The *constant/polyMesh/boundary* file

```
9
(
    minX
    {
        type            wall;
        nFaces          400;
        startFace       499702;
    }
    maxX
    {
        type            wall;
        nFaces          400;
        startFace       500102;
    }
    minY
    {
        type            empty;
        nFaces          400;
        startFace       500502;
    }
    maxY
    {
        type            wall;
        nFaces          400;
        startFace       500902;
    }
    minZ
    {
        type            wall;
        nFaces          400;
        startFace       501302;
    }
```

Name

Type

nFaces
startFace

Name and type of the surface patches

- The **name** and **base type** of the patch is given by the user.
- In this case the **name** and **base type** was assigned in the dictionaries *blockMeshDict* and *snappyHexMeshDict*.
- You can change the **name** if you do not like it. Do not use strange symbols or white spaces.
- You can also change the **base type**. For instance, you can change the type of the patch **maxY** from **wall** to **patch.**

nFaces and startFace keywords

- Unless you know what are you doing, **you do not need to change this information.**
- Basically, this is telling you the starting face and ending face of the patch.
- This information is created automatically when generating the mesh or converting the mesh.

Mesh generation using snappyHexMesh

The *constant/polyMesh/boundary* file

```
        maxZ
        {
                type            wall;
                nFaces          400;
                startFace       500902;
        }
        wolf
        {
                type            wall;
                inGroups        1(wall);
                nFaces          20419;
                startFace       502102;
        }
        sphere
        {
                type            empty;
                inGroups        1(wall);
                nFaces          368;
                startFace       522521;
        }
        sphere_slave
        {
                type            wall;
                inGroups        1(wall);
                nFaces          368;
                startFace       522889;
        }
)
```

Name

Type

nFaces
startFace

Name and type of the surface patches

- The **name** and **base type** of the patch is given by the user.
- In this case the **name** and **base type** was assigned in the dictionaries *blockMeshDict* and *snappyHexMeshDict*.
- You can change the **name** if you do not like it. Do not use strange symbols or white spaces.
- You can also change the **base type**. For instance, you can change the type of the patch **maxY** from **wall** to **patch**.

nFaces and startFace keywords

- Unless you know what are you doing, **you do not need to change this information.**
- Basically, this is telling you the starting face and ending face of the patch.
- This information is created automatically when generating the mesh or converting the mesh.

Cleaning the case directory

- When generating the mesh using OpenFOAM®, it is extremely important to start from a clean case directory.

- To clean all the case directory, in the terminal type:

    - `$> foamCleanTutorials`

- To only erase the mesh information, in the terminal type:

    - `$> foamCleanPolyMesh`

- If you are planning to start the meshing from a previous saved state, you do not need to clean the case directory.

- Before proceeding to compute the solution, remember to always check the quality of the mesh.

Roadmap

1. ~~Meshing preliminaries~~
2. ~~What is a good mesh?~~
3. ~~Mesh quality assessment in OpenFOAM®~~
4. ~~Mesh generation using blockMesh.~~
5. ~~Mesh generation using snappyHexMesh.~~
6. **snappyHexMesh guided tutorials.**
7. Mesh conversion
8. Geometry and mesh manipulation utilities

snappyHexMesh guided tutorials

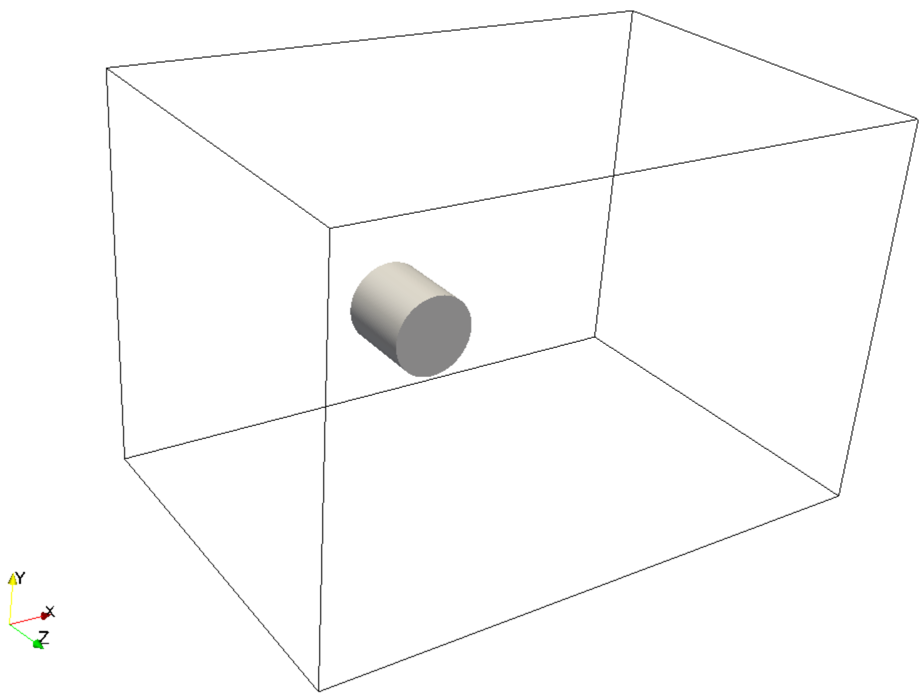
- Meshing with snappyHexMesh – Case 1.
- 3D cylinder with feature edge refinement (external mesh).
- You will find this case in the directory:

```
$PTOFC/101SHM_basic/M1_cyl/C1
```

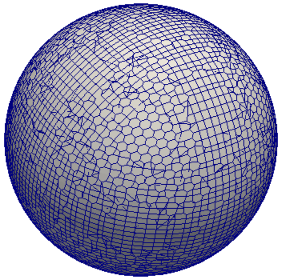
- In the case directory, you will find a few scripts with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on.
- These scripts can be used to run the case automatically by typing in the terminal, for example,
  - `$> sh run_solver`
- These scripts are human-readable, and we highly recommend you open them, get familiar with the steps, and type the commands in the terminal. In this way, you will get used with the command line interface and OpenFOAM commands.
- If you are already comfortable with OpenFOAM, run the cases automatically using these scripts.
- In the case directory, you will also find the `README.FIRST` file. In this file, you will find some additional comments.

snappyHexMesh guided tutorials

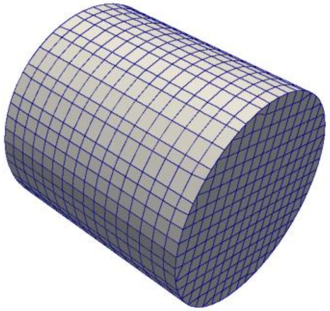
- Our first case will be a mesh around a cylinder.
- This is a simple geometry, but we will use it to study all the meshing steps and introduce a few advanced features.
- This case is located in the directory **`$PTOFC/101SHM_basic/M1cyl`**



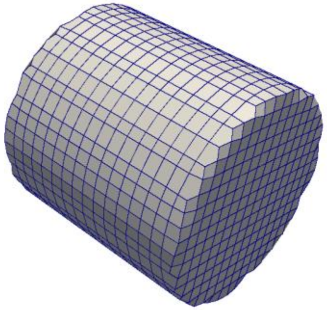
3D Cylinder with edge refinement.



Sphere with no edge refinement



Cylinder with edge refinement



Cylinder with no edge refinement

- If the geometry does not have sharp angles, you do not need to do this extra step.
- In the end, it is up to you to decide if you want to resolve the sharp angles.
- However, it is extremely recommended to resolve sharp angles (if they exist).
- In the left figure there is no need to use edge refinement as there are no sharp angles.
- In the mid figure we used edge refinement to resolve the sharp angles.
- In the right figure we did not use edge refinement, therefore we did not resolve well the sharp angles.

3D Cylinder with edge refinement.

- How do we control curvature refinement and enable edge refinement?
- In the file *snappyHexMeshDict*, look for the following entries:

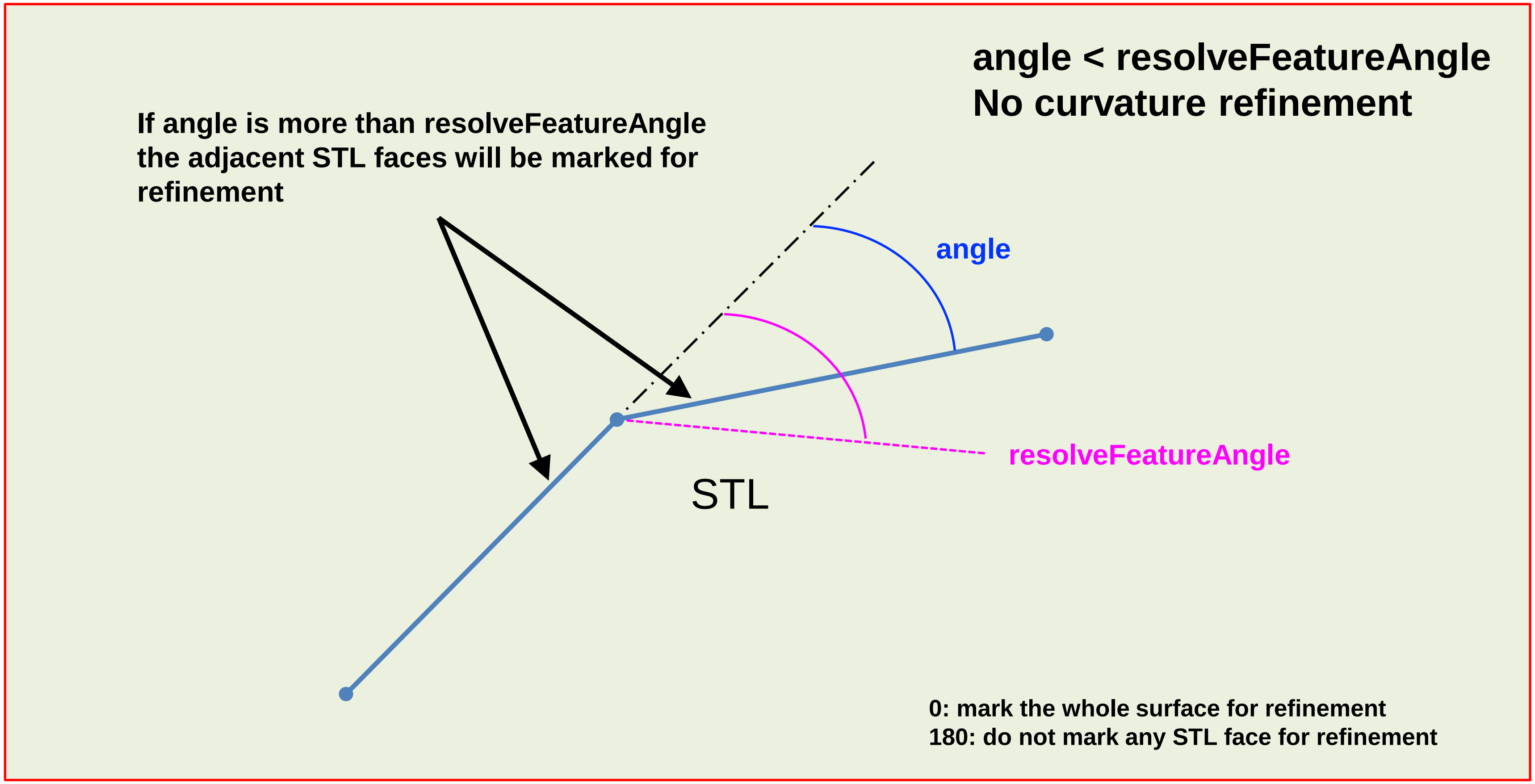
```
castellatedMeshControls
{

        ...
        ...
        ...

        //Local curvature and
        //feature angle refinement
        resolveFeatureAngle 30;

        ...
        ...
        ...

        //Explicit feature edge refinement
        features
        (
                {
                        file "surfacemesh.eMesh";
                        level 0;
                }
        );

        ...
        ...
        ...

}
```

To control curvature refinement

To enable and control edge refinement level

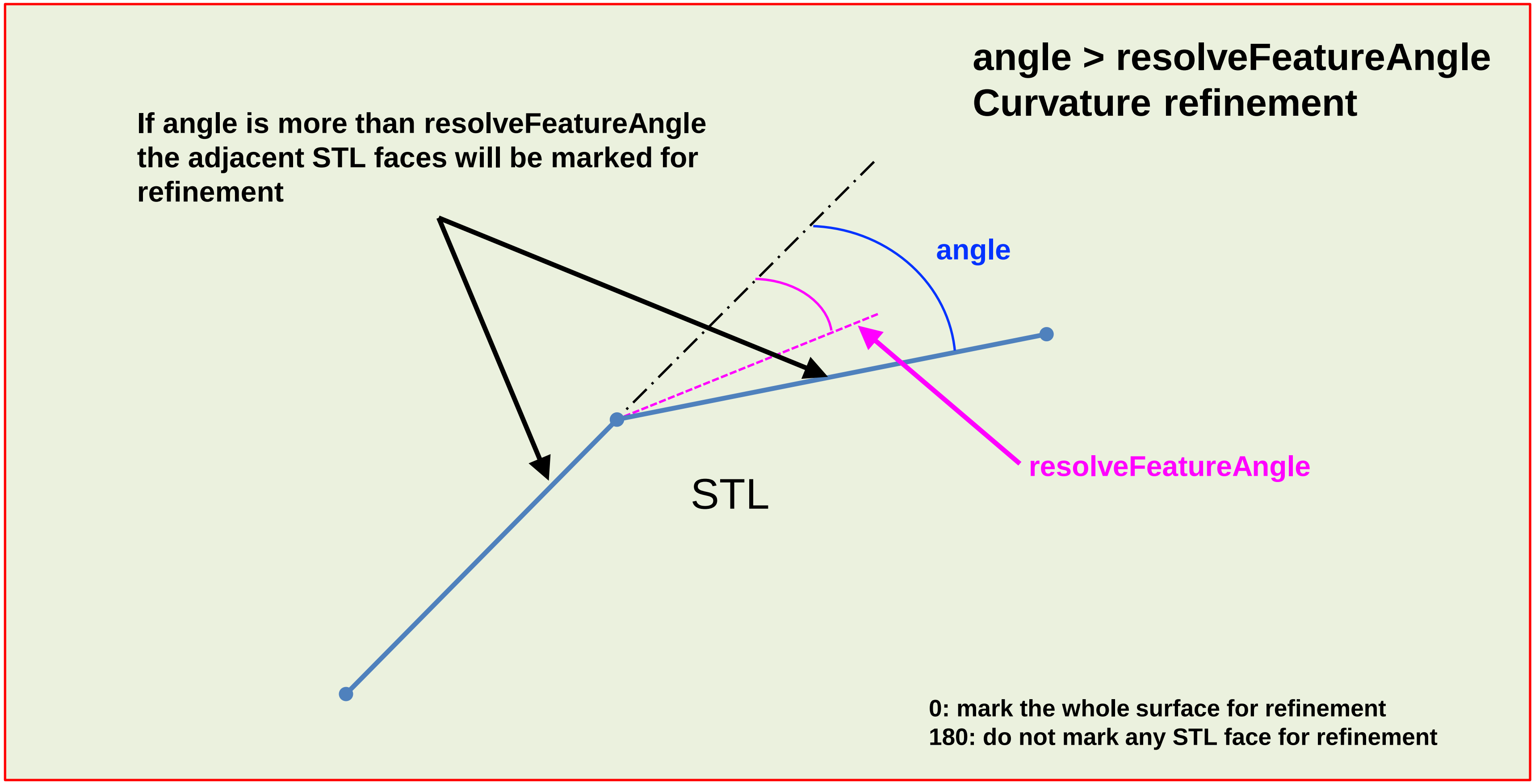
3D Cylinder with edge refinement.

How **resolveFeatureAngle** works?



3D Cylinder with edge refinement.

How **resolveFeatureAngle** works?



3D Cylinder with edge refinement.

- How do we control surface refinement?
- In the file `snappyHexMeshDict`, look for the following entry:

```
castellatedMeshControls
{
        ...
        ...
        ...

    //Surface based refinement
    refinementSurfaces
    {
        banana_stlSurface
        {
            level (2 4);
        }
    }

        ...
        ...
        ...

}
```

To control surface refinement.
The first digit controls the global surface refinement level, and the second digit controls the curvature refinement level, according to the angle set in the entry resolveFeatureAngle

3D Cylinder with edge refinement.

- How do we create refinement regions?
- In the file *snappyHexMeshDict*, look for the following entry:

```
geometry
{
        ...
        ...
        ...

        refinementBox                 Name of refinement region
        {
                type searchableBox;           Geometrical entity type.
                min  ( -2 -2 -2);             This is the zone where we
                max (  2   2  2);             want to apply the refinement
        }
                                              Dimensions of geometrical entity
        ...
        ...
        ...
};
```

3D Cylinder with edge refinement.

- How do we create refinement regions?
- In the file *snappyHexMeshDict*, look for the following entry:

```
castellatedMeshControls
{
    ...
    ...
    ...

    refinementRegions
    {
        refinementBox
        {
            mode inside;
            levels ((1   1));
        }
    }

    ...
    ...
    ...
}
```

Name of the region created in the geometry section

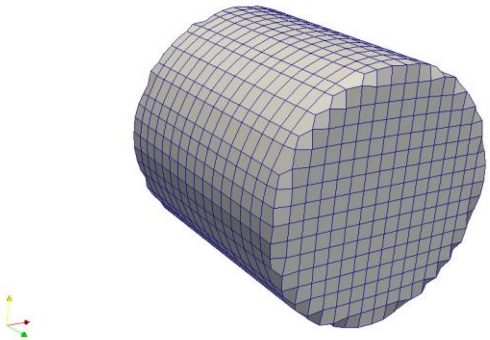
Type of refinement (inside, outside, or distance mode)

Max refinement level

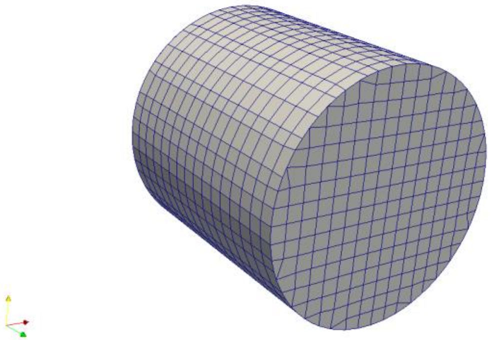
Min refinement level
If distance mode is used, it represents the distance normal to the wall (in both direction)

3D Cylinder with edge refinement.

Effect of various parameters on edge capturing and surface refinement



Explicit feature edge refinement level 0
resolveFeatureAngle 110
Surface based refinement level (2 2)

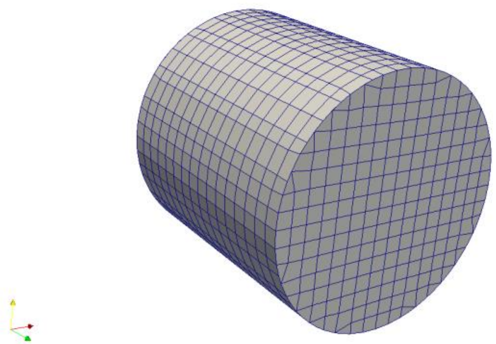


Explicit feature edge refinement level 0
resolveFeatureAngle 60
Surface based refinement level (2 2)

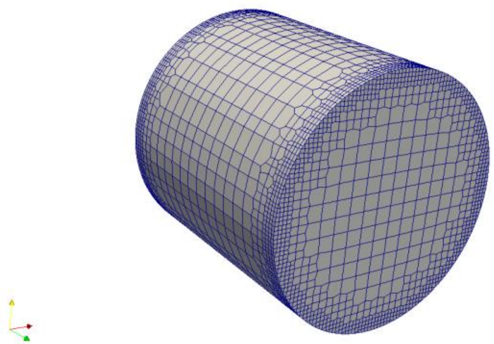
- To control edges capturing you can decrease the value of **resolveFeatureAngle**.
- Be careful, this parameter also controls curvature refinement, so if you choose a low value you also will be adding a lot of refinement on the surface.

3D Cylinder with edge refinement.

Effect of various parameters on edge capturing and surface refinement



Explicit feature edge refinement level 0
resolveFeatureAngle 60
Surface based refinement level (2 2)

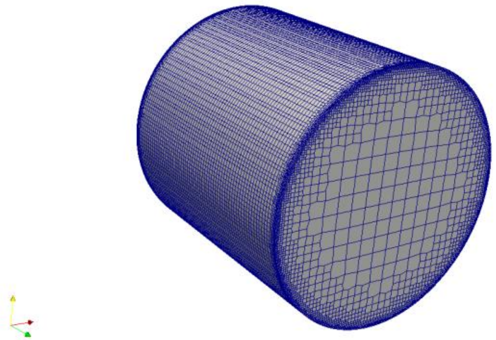


Explicit feature edge refinement level 4
resolveFeatureAngle 60
Surface based refinement level (2 2)

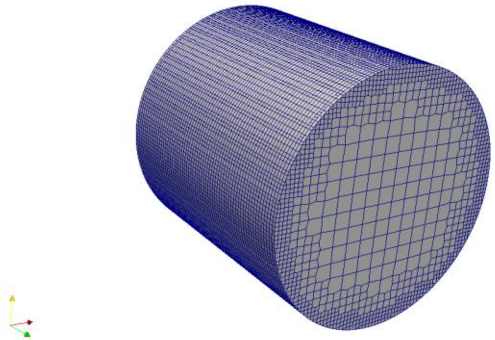
- To control edges refinement level, you can change the value of the explicit feature edge refinement level.

3D Cylinder with edge refinement.

Effect of various parameters on edge capturing and surface refinement



Explicit feature edge refinement level 6
resolveFeatureAngle 5
Surface based refinement level (2 4)

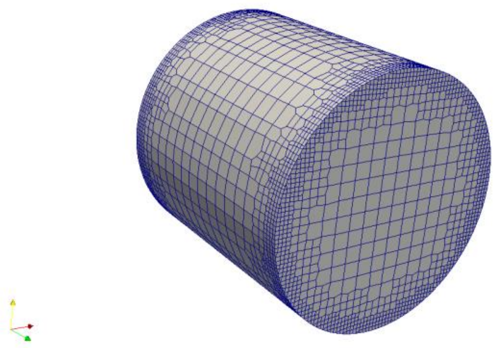


Explicit feature edge refinement level 0
resolveFeatureAngle 5
Surface based refinement level (2 4)

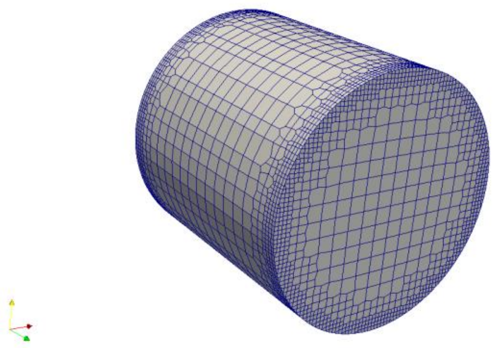
- To control edges refinement level, you can change the value of the explicit feature edge refinement level.

3D Cylinder with edge refinement.

Effect of various parameters on edge capturing and surface refinement



Explicit feature edge refinement level 0
resolveFeatureAngle 60
Surface based refinement level (2 4)

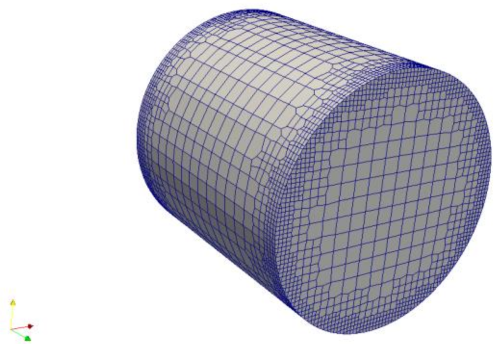


Explicit feature edge refinement level 4
resolveFeatureAngle 60
Surface based refinement level (2 2)

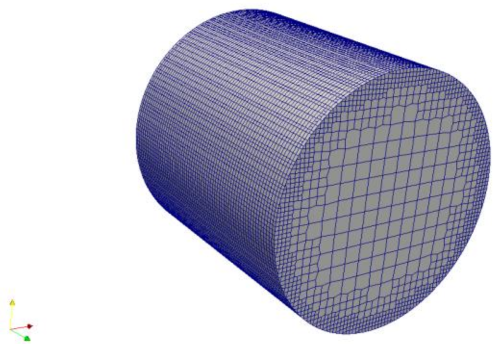
- To control surface refinement level, you can change the value of the surface-based refinement level.
- The first digit controls the global surface refinement level, and the second digit controls the curvature refinement level.

3D Cylinder with edge refinement.

Effect of various parameters on edge capturing and surface refinement



Explicit feature edge refinement level 0
resolveFeatureAngle 60
Surface based refinement level (2 4)



Explicit feature edge refinement level 0
resolveFeatureAngle 5
Surface based refinement level (2 4)

- To control surface refinement due to curvature together with control-based surface refinement level, you can change the value of **resolveFeatureAngle**, and surface-based refinement level

3D Cylinder with edge refinement.

- Let us explore the dictionary *surfaceFeaturesDict* used by the utility `surfaceFeatures`.
- This utility will extract surface features (sharp angles) according to an angle criterion (**includedAngle**).

```
surfaces ("surfacemesh.stl")

includedAngle           150;

subsetFeatures
{
        nonManifoldEdges       yes;

        openEdges              yes;
}

writeObj        yes;
```

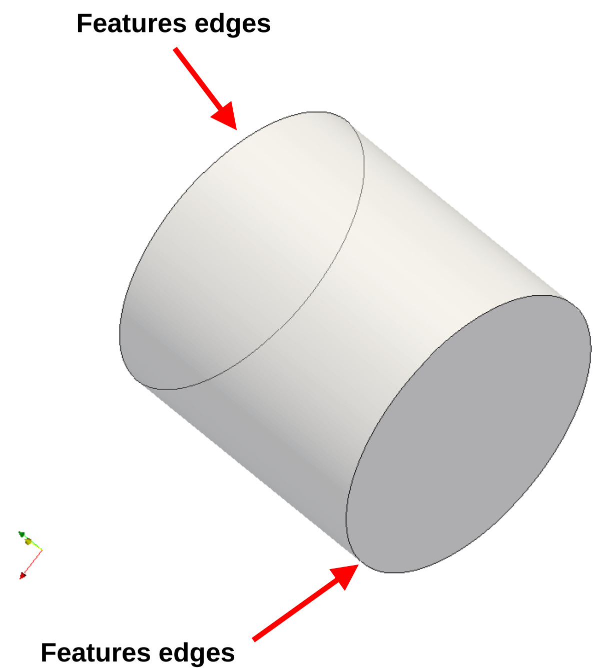
Name of the STL. The STL file is located in constant/triSurface

Angle criterion to extract features

Keep non-manifold edges (edges with more that 2 connected faces)

Keep open edges (edges with 1 connected face)

If you want to save the .obj files



3D Cylinder with edge refinement.

- Let us explore the dictionary *surfaceFeaturesDict* used by the utility `surfaceFeatures`.
- This utility will extract surface features (sharp angles) according to an angle criterion (**includedAngle**).

```
surfaces ("surfacemesh.stl")

includedAngle          150;

subsetFeatures
{
        nonManifoldEdges     yes;

        openEdges            yes;
}

writeObj        yes;
```

Name of the STL. The STL file is located in constant/triSurface

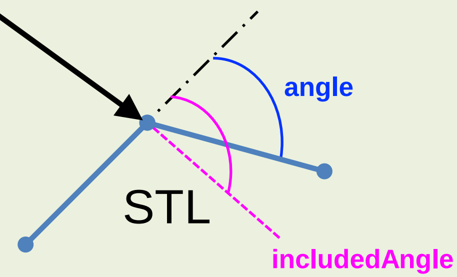
Angle criterion to extract features

Keep non-manifold edges (edges with more that 2 connected faces)

Keep open edges (edges with 1 connected face)

If you want to save the .obj files

If angle is less than includedAngle this feature will be marked



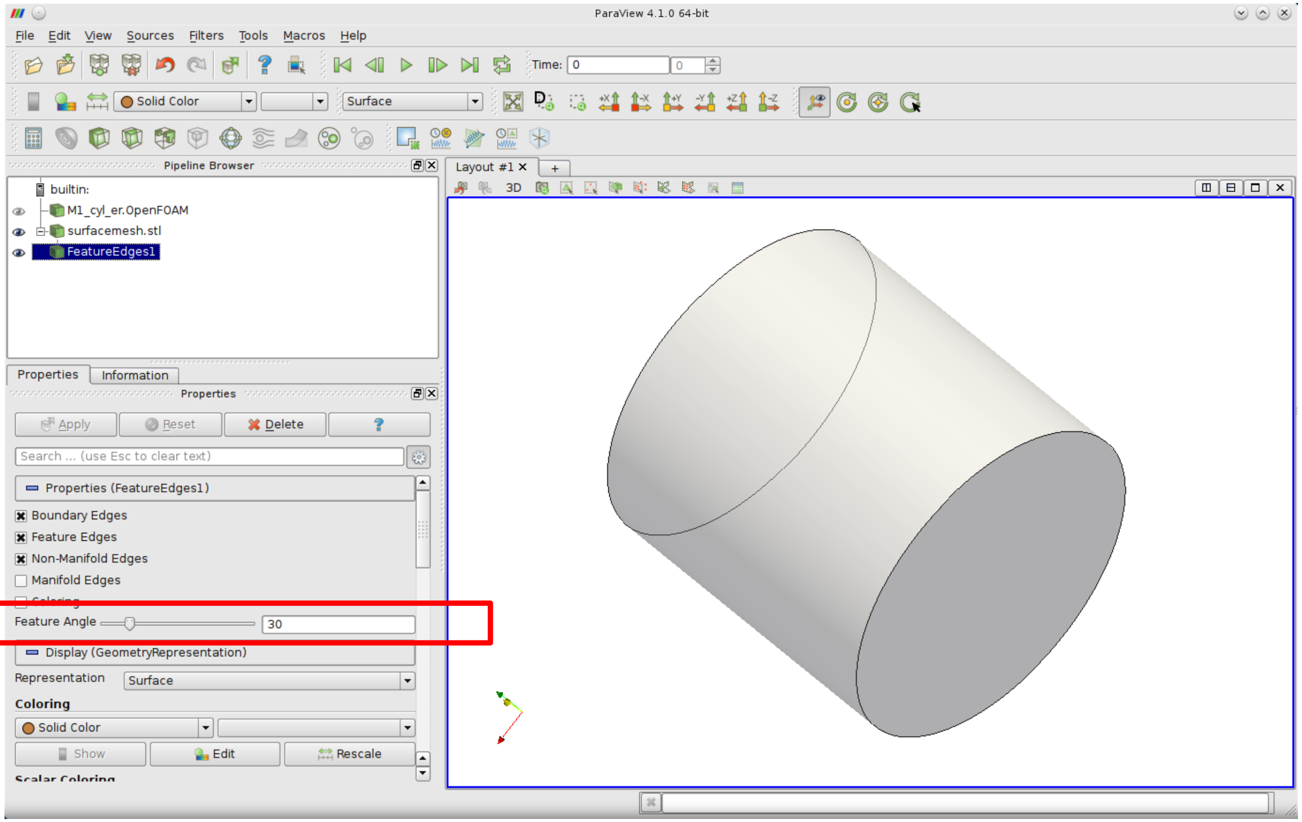
Mark edges whose adjacent surface normals are at an angle less than includedAngle

0: selects no edges
180: selects all edge

snappyHexMesh guided tutorials

3D Cylinder with edge refinement.

- If you want to have a visual representation of the feature edges, you can use paraview/paraFoam.
- Just look for the filter `Feature Edges`.
- Have in mind that the angle you need to define in paraview/paraFoam is the complement of the angle defined in the dictionary *`surfaceFeaturesDict`*



3D Cylinder with edge refinement.

- In this case we are going to generate a body fitted mesh with edge refinement. This is an external mesh.
- These are the dictionaries and files that will be used.
  - *system/snappyHexMeshDict*
  - *system/surfaceFeaturesDict*
  - *system/meshQualityDict*
  - *system/blockMeshDict*
  - *constant/triSurface/surfacemesh.stl*
  - *constant/triSurface/surfacemesh.eMesh*

- The file *surfacemesh.eMesh* is generated after using the utility `surfaceFeatures`, which reads the dictionary *surfaceFeaturesDict*.
- The utility `surfaceFeatures`, will save a set of *.obj files with the captured edges. These files are located in the directory **`constant/extendedFeatureEdgeMesh`**. You can use paraview to visualize the *.obj files.

3D Cylinder with edge refinement.

- Let us generate the mesh, in the terminal window type:

```
1.  $> foamCleanTutorials
2.  $> surfaceFeatures
3.  $> blockMesh
4.  $> snappyHexMesh  -overwrite
5.  $> checkMesh -latestTime
6.  $> paraFoam
```

- In step 2 we extract the sharp angles from the geometry.
- In step 3 we generate the background mesh.
- In step 4 we generate the body fitted mesh. Have in mind that as we use the option `–overwrite`, we are not saving the intermediate steps.
- In step 5 we check the mesh quality.

snappyHexMesh guided tutorials

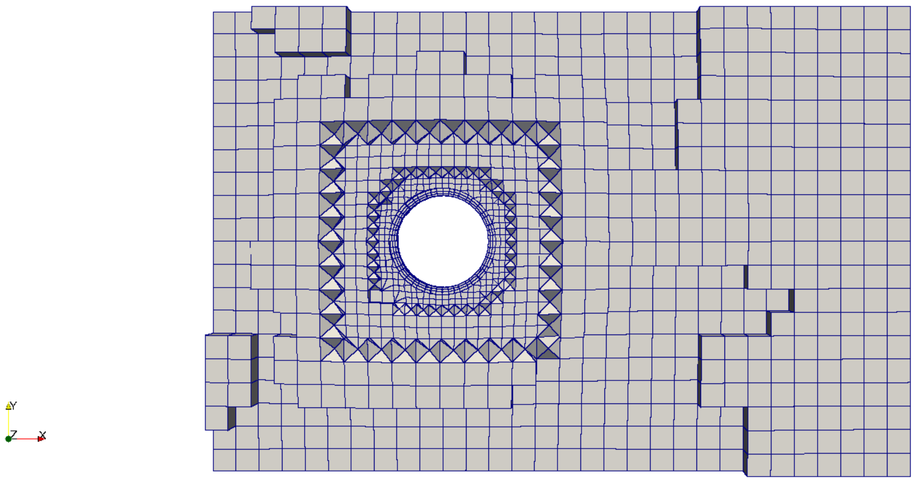
- Meshing with snappyHexMesh – Case 2.
- 3D cylinder with feature edge refinement and boundary layer (external mesh).
- You will find this case in the directory:

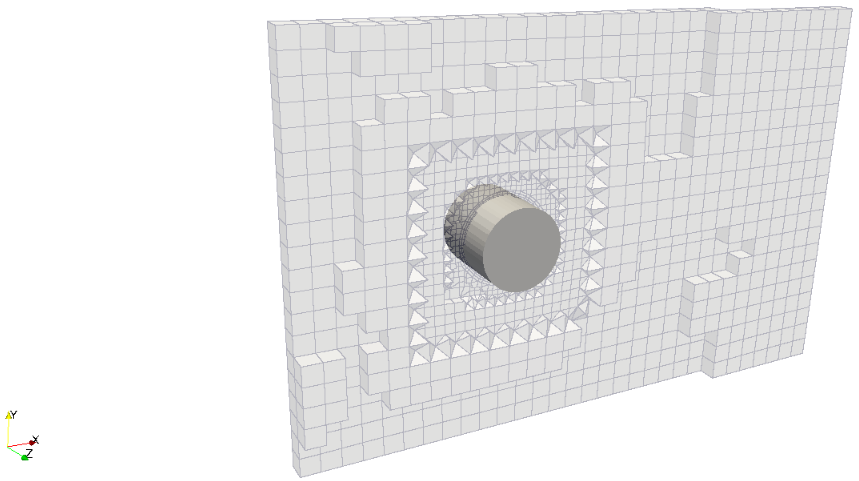
```
$PTOFC/101SHM_basic/M1_cyl/C2
```

- In the case directory, you will find a few scripts with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on.
- These scripts can be used to run the case automatically by typing in the terminal, for example,
  - `$> sh run_solver`
- These scripts are human-readable, and we highly recommend you open them, get familiar with the steps, and type the commands in the terminal. In this way, you will get used with the command line interface and OpenFOAM commands.
- If you are already comfortable with OpenFOAM, run the cases automatically using these scripts.
- In the case directory, you will also find the `README.FIRST` file. In this file, you will find some additional comments.

snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer.





Your final mesh should look like this one

3D Cylinder with edge refinement and boundary layer.

- How do we enable boundary layer?
- In the file *snappyHexMeshDict*, look for the following entry:

```
castellatedMesh true;        //or false
snap true;                   //or false
addLayers true;              //or false

    ...
    ...
    ...
```

Set this parameter to true if you want to enable boundary layer meshing

3D Cylinder with edge refinement and boundary layer.

- How do we enable boundary layer?
- In the file *snappyHexMeshDict*, look for the section **addLayersControls**:

```
addLayersControls
{

    //Global parameters
    relativeSizes true;
    expansionRatio 1.2;
    finalLayerThickness 0.5;
    minThickness 0.1;

    layers
    {
        banana_stlSurface
        {
            nSurfaceLayers 3;
        }
    }

    // Advanced settings

    ...
    ...
    ...

}
```

These options control how the boundary layer mesh grows from the surface into the domain. Possible combinations are:

- First layer thickness (**firstLayerThickness**) and overall thickness (**thickness**).
- First layer thickness (**firstLayerThickness**) and expansion ratio (**expansionRatio**).
- Final layer thickness (**finalLayerThickness**) and expansion ratio (**expansionRatio**).
- Final layer thickness (**finalLayerThickness**) and overall thickness (**thickness**).
- Overall thickness (**thickness**) and expansion ratio (**expansionRatio**).
- The option **minThickness** controls the minimum thickness of the layers.

3D Cylinder with edge refinement and boundary layer.

- How do we enable boundary layer?
- In the file `snappyHexMeshDict`, look for the section **addLayersControls**:

```
addLayersControls
{

    //Global parameters
    relativeSizes true;
    expansionRatio 1.2;
    finalLayerThickness 0.5;
    minThickness 0.1;

    layers
    {
        banana_stlSurface
        {
            nSurfaceLayers 3;
        }
    }

    // Advanced settings

    ...
    ...
    ...

}
```

Name of the surface or user-defined patch where you want to add the boundary layer mesh.

Number of layers to add.

In this sub/dictionary you can also set local per patch parameters. The local options overwrite the global parameters.

You can use the same parameters as the one you use in the global parameters, except for relativeSizes.

3D Cylinder with edge refinement and boundary layer.

- How do we control boundary layer collapsing?
- In the file *snappyHexMeshDict*, look for the section **addLayersControls**:

```
addLayersControls
{

    ...
    ...
    ...

    // Advanced settings
    nGrow 0;
    featureAngle 130;


    maxFaceThicknessRatio    0.5;
    ...
    ...
    ...

}
```

Increase this value to avoid BL collapsing
The maximum value should not be greater than 330

Increase this value to avoid BL collapsing
The maximum value should not be greater than 1

Let us explore the snappyHexMeshDict dictionary.

- Regarding the inflation layer parameters (**addLayersControls**), in our experience the most important parameters are **featureAngle** and **featureAngle**.
- To set these values, you can follow the same guidelines as the ones we defined for **snapControls**.
- It is important to stress that we are referring to the control parameters related to the mesh quality and iterative relaxation.
- The parameters related to the inflation layer thickness are much more important.
- We will demonstrate this using an excel worksheet.

Recommended values

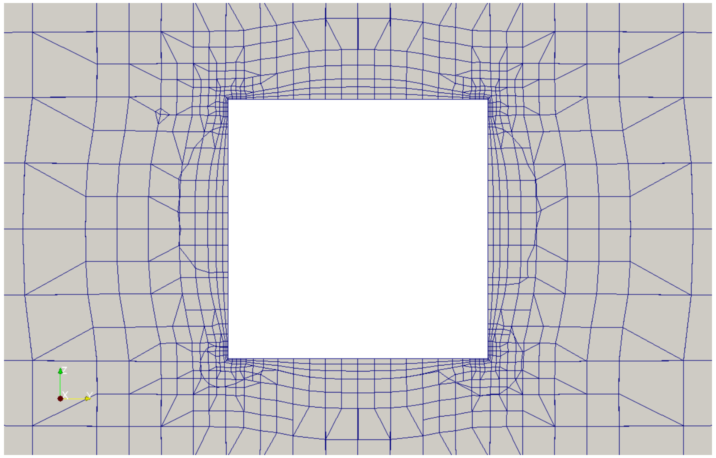
```
addLayersControls
{
    ...

    featureAngle                130;
    maxFaceThicknessRatio       0.5;


    ...
}
```

Improved values (advised maximum values)

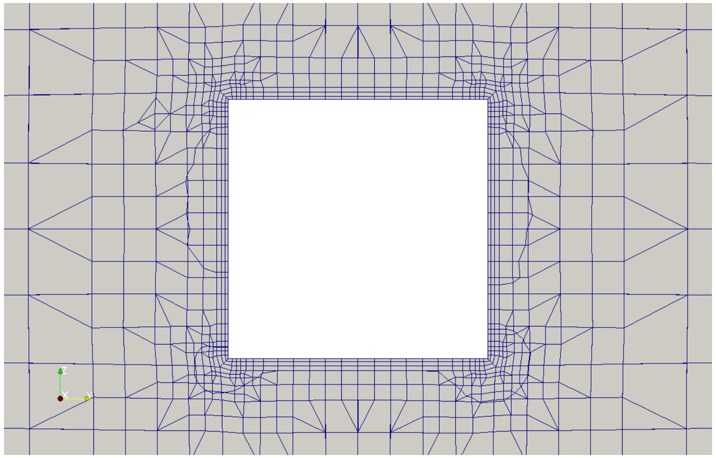
```
addLayersControls
{
    ...

    featureAngle                330;
    maxFaceThicknessRatio       1;


    ...
}
```

3D Cylinder with edge refinement and boundary layer.

Effect of different parameters on the boundary layer meshing



relativeSizes true
expansionRatio 1.2
finalLayerThickness 0.5
minThickness 0.1
featureAngle 130
nSurfaceLayers 3
Surface based refinement level (2 4)

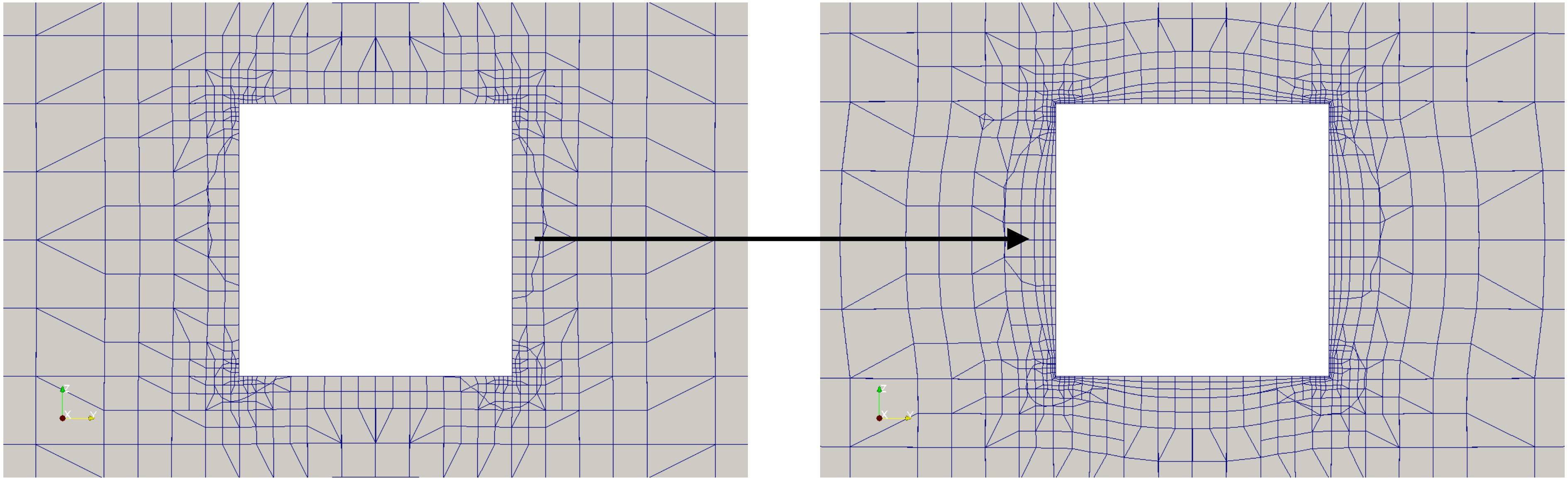


relativeSizes false
expansionRatio 1.2
firstLayerThickness 0.025
minThickness 0.01
featureAngle 130
nSurfaceLayers 3
Surface based refinement level (2 4)

- The option **finalLayerThickness** controls the thickness of the final layer, whereas the option **minThickness** controls the minimum thickness of the first layer.
- The actual thickness of the layers depends on the keyword **relativeSizes**.

3D Cylinder with edge refinement and boundary layer.

Effect of different parameters on the boundary layer meshing

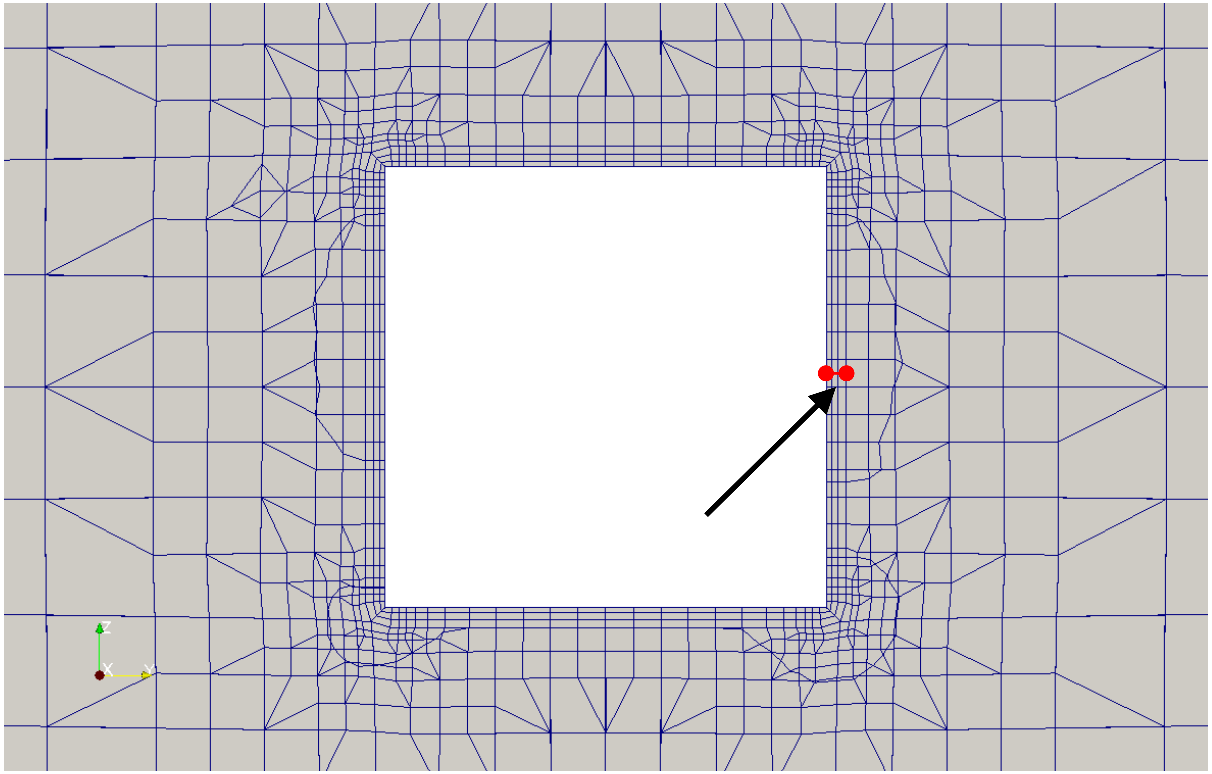


- When the option **relativeSizes** is true, the boundary layer meshing is done relative to the size of the cells next to the surface.
- This option requires less user intervention but can not guarantee a uniform boundary layer.
- Also, it is quite difficult to set a desired thickness of the first layer.

snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer.

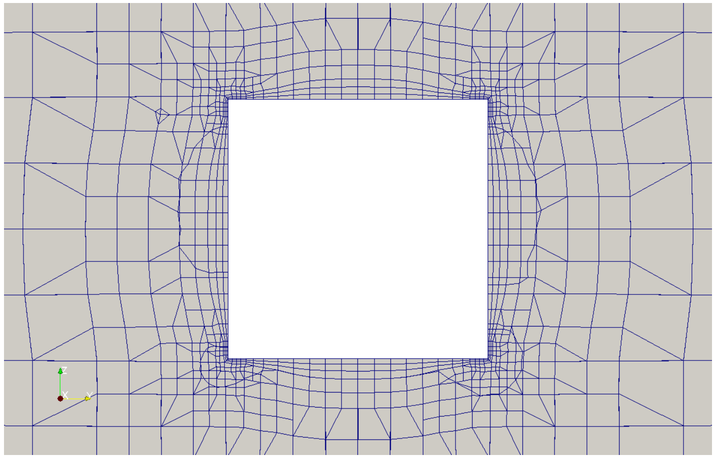
Effect of different parameters on the boundary layer meshing



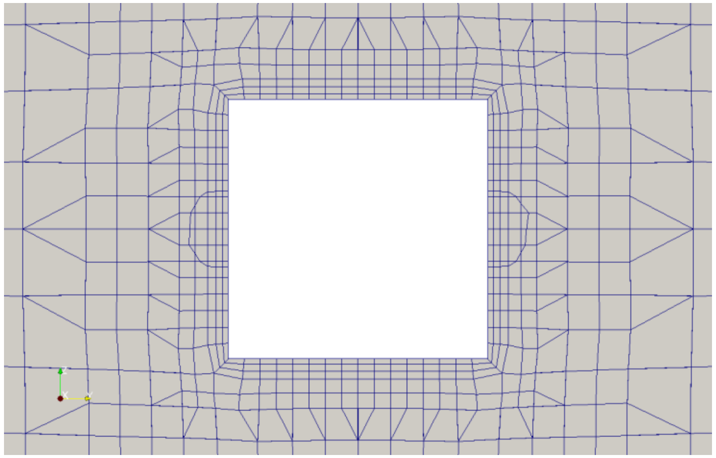
- When the option **relativeSizes** is false, we give the actual thickness of the layers.
- This option requires a lot user intervention, but it guarantees a uniform boundary layer and the desired layer thickness.

3D Cylinder with edge refinement and boundary layer.

Effect of different parameters on the boundary layer meshing



relativeSizes true
expansionRatio 1.2
finalLayerThickness 0.5
minThickness 0.1
featureAngle 130
nSurfaceLayers 3
Surface based refinement level (2 4)

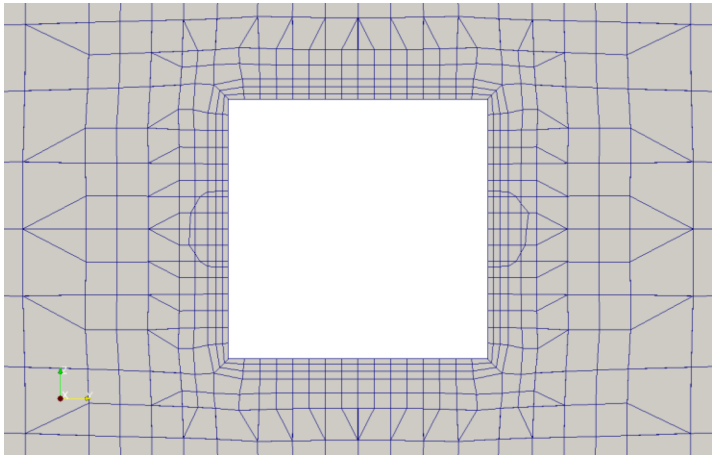


relativeSizes true
expansionRatio 1.2
finalLayerThickness 0.5
minThickness 0.1
featureAngle 130
nSurfaceLayers 3
Surface based refinement level (2 2)

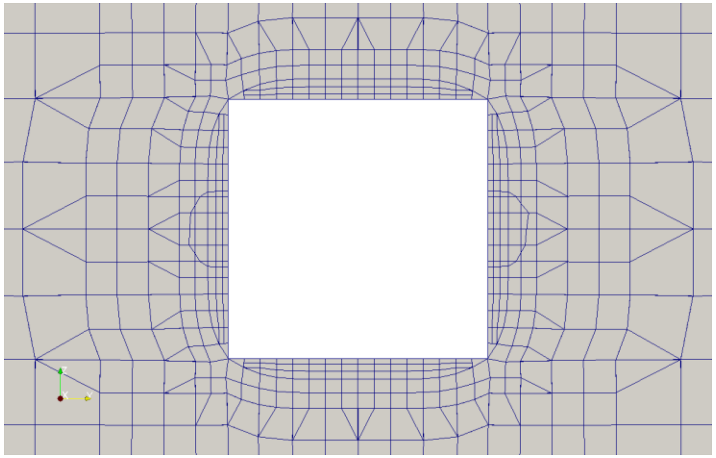
- When the option **relativeSizes** is true and in order to have a uniform boundary layer, we need to have a uniform surface refinement.
- Nevertheless, we still do not have control on the desired thickness of the first layer.

3D Cylinder with edge refinement and boundary layer.

Effect of different parameters on the boundary layer meshing



relativeSizes true
expansionRatio 1.2
finalLayerThickness 0.5
minThickness 0.1
featureAngle 130
nSurfaceLayers 3
Surface based refinement level (2 2)

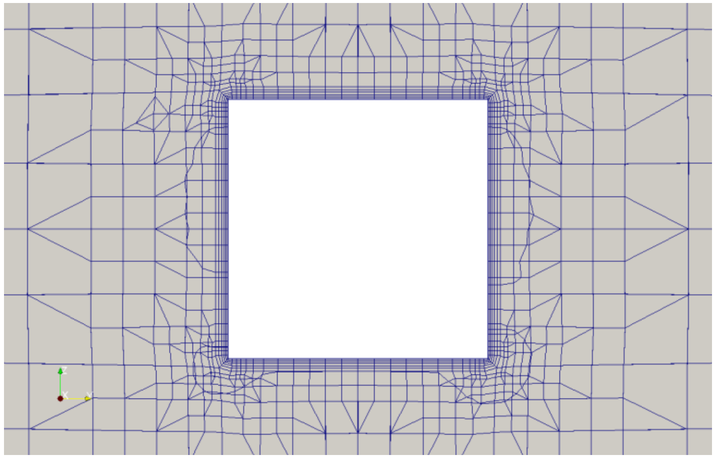


relativeSizes true
expansionRatio 1.2
finalLayerThickness 0.5
minThickness 0.1
featureAngle 30
nSurfaceLayers 3
Surface based refinement level (2 2)

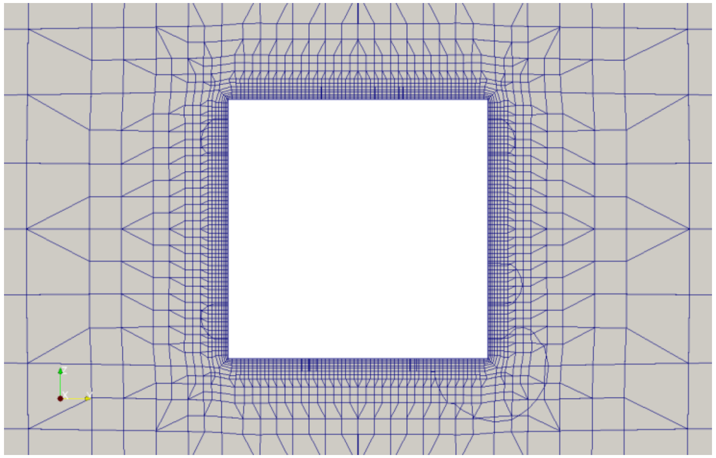
- To avoid boundary layer collapsing close to the corners, we can increase the value of the boundary layer parameter **featureAngle**.

3D Cylinder with edge refinement and boundary layer.

Effect of different parameters on the boundary layer meshing



relativeSizes false
nSurfaceLayers 6



relativeSizes false
nSurfaceLayers 6
Refinement region at the stl surface:
mode distance;
levels ((0.05 4))

- The disadvantage of setting **relativeSizes** to false, is that it is difficult to control the expansion ratio from the boundary layer meshing to the far mesh.
- To control this transition, we can add a refinement region at the surface with distance mode.

snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer.

- To generate the mesh, in the terminal window type:

1. `$> foamCleanTutorials`
2. `$> surfaceFeatures`
3. `$> blockMesh`
4. `$> snappyHexMesh -overwrite`
5. `$> checkMesh –latestTime`
6. `$> paraFoam`

3D Cylinder with edge refinement and boundary layer.

- At the end of the meshing process, you will get the following information regarding the boundary layer meshing:

```
patch                  faces      layers     overall    thickness
                                             [m]        [%]
-----                  -----      ------     ---        ---
banana_stlSurface      4696       3          0.0569     95.9

Layer mesh : cells:48577     faces:157942     points:61552
```

- This is a general summary of the boundary layer meshing.
- Pay particular attention to the overall and thickness information.
- Overall is roughly speaking the thickness of the whole boundary layer.
- Thickness is the percentage of the patch that has been covered with the boundary layer mesh.
- A thickness of 100% means that the whole patch has been covered (a perfect BL mesh).

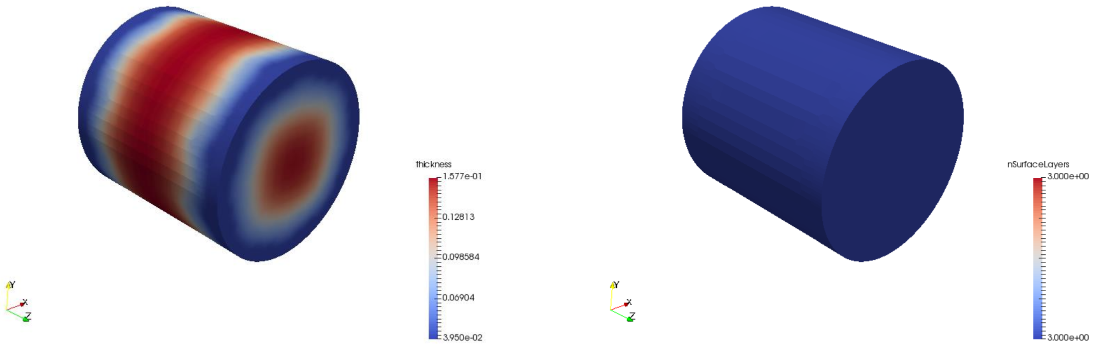
3D Cylinder with edge refinement and boundary layer.

- If you want to visualize the boundary layer thickness, you can enable **writeFlags** in the *snappyhexMeshDict* dictionary,

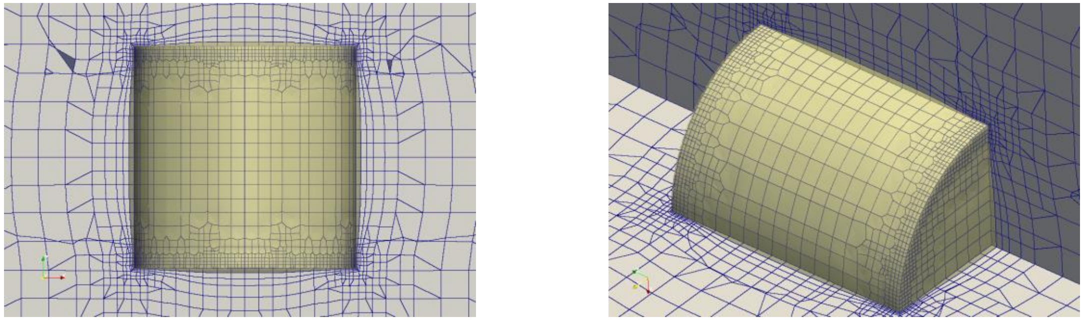
```
...
...
...

writeFlags
(
        scalarLevels;     // write volScalarField with cellLevel for postprocessing
        layerSets;        // write cellSets, faceSets of faces in layer
        layerFields;      // write volScalarField for layer coverage
);

...
...
...
```

3D Cylinder with edge refinement and boundary layer.

- Then you can use paraview/paraFoam to visualize the boundary layer coverage.



Boundary layer thickness and number of layers



The yellow surface represent the BL coverage

snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer.

- After creating the mesh and if you do not like the inflation layer or you want to try different layer parameters, you do not need to start the meshing process from scratch.
- To restart the meshing process from a saved state you need to save the intermediate steps (castellation and snapping), and then create the inflation layers starting from the snapped mesh.
- That is, do not use the option `snappyHexMesh -overwrite`.
- Also, in the dictionary *`controlDict`* remember to set the entry `startFrom` to `latestTime` or the time directory where the snapped mesh is saved (in this case **2**).
- Before restarting the meshing, you will need to turn off the castellation and snapping options and turn on the boundary layer options in the *`snappyHexMeshDict`* dictionary.

3D Cylinder with edge refinement and boundary layer.

- Remember, before restarting the meshing you will need to modify the *snappyHexMeshDict* dictionary as follows:

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

- At this point, you can restart the meshing process by typing in the terminal,
    - `$> snappyHexMesh`

- By the way, you can restart the boundary layer mesh from a previous mesh with a boundary layer.
- So, in theory, you an add one layer at a time, this will give you more control, but it will require more manual work and some scripting.

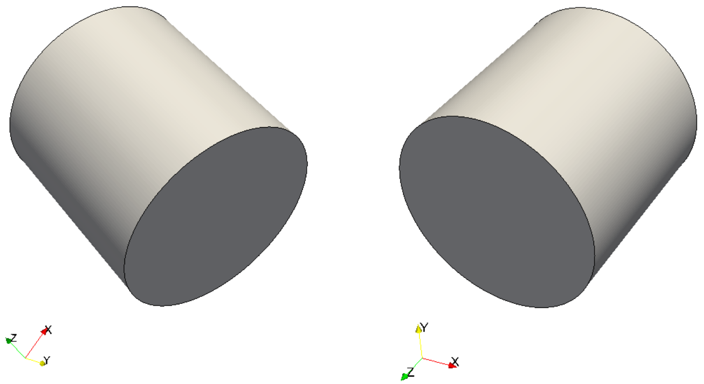
snappyHexMesh guided tutorials

- Meshing with snappyHexMesh – Case 3.
- 3D cylinder with feature edge refinement and boundary layer using a STL with multiple surfaces (external mesh).
- You will find this case in the directory:

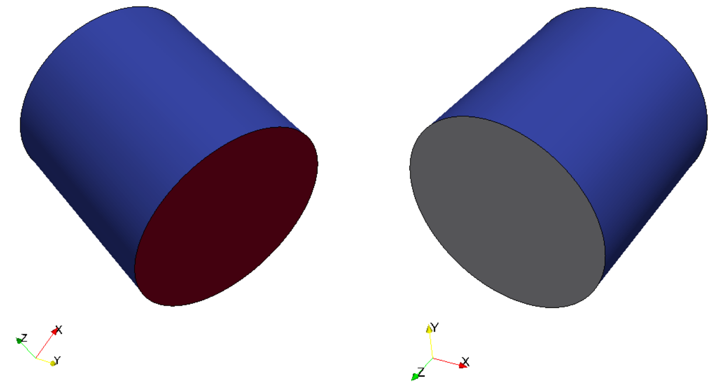
```
$PTOFC/101SHM_basic/M1_cyl/C3
```

- In the case directory, you will find a few scripts with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on.
- These scripts can be used to run the case automatically by typing in the terminal, for example,
  - `$> sh run_solver`
- These scripts are human-readable, and we highly recommend you open them, get familiar with the steps, and type the commands in the terminal. In this way, you will get used with the command line interface and OpenFOAM commands.
- If you are already comfortable with OpenFOAM, run the cases automatically using these scripts.
- In the case directory, you will also find the `README.FIRST` file. In this file, you will find some additional comments.

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.



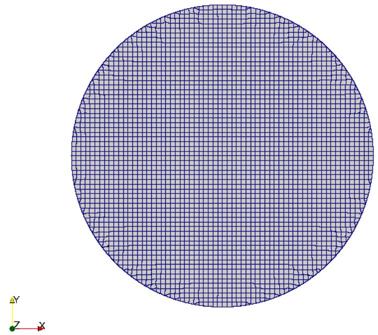
STL visualization with a single surface using paraview (the single surface in represented with a single color)

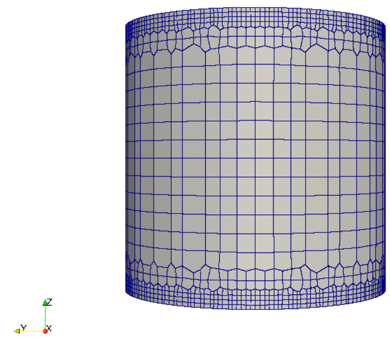


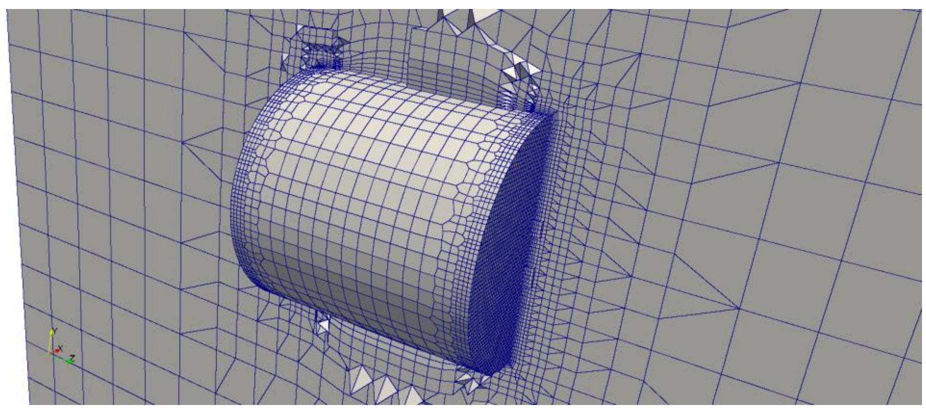
STL visualization with multiple surfaces using paraview (each color corresponds to a different surface)

- When you use a STL with multiple surfaces, you have more control over the meshing process.
- By default, STL files are made up of one single surface.
- If you want to create the multiple surfaces you will need to do it in the solid modeler.
- Alternatively, you can split the STL manually or using the utility `surfaceAutoPatch`.
- Loading multiple STLs is equivalent to using a STL with multiple surfaces.

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.







- When you use a STL with multiple surfaces, you have more control over the meshing process.
- In this case, we were able to use different refinement parameters in the lateral and central surface patches of the cylinder.

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.

- How do we assign different names to different surface patches?
- In the file *snappyHexMeshDict*, look for the following entry:

```
geometry
{
        surfacemesh.stl
        {
                type triSurfaceMesh;
                name stlSurface;

                regions
                {
                        patch0                  Named region in the STL file
                        {
                                name surface0;  User-defined patch name
                                                This is the name you need to use when setting the
                                                boundary layer meshing
                        }
                        patch1
                        {
                                name surface1;
                        }
                        patch2
                        {
                                name surface2;
                        }
                }
        }
        ...
        ...
        ...
}
```

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.

- How do we refine user defined surface patches?
- In the file *snappyHexMeshDict*, look for the following entry:

```
castellatedMeshControls
{
        ...
        ...
        ...
        refinementSurfaces
        {
                level (2 2);          Global refinement level
                regions
                {
                        patch0        Local surface patch
                        {
                                level (2 2);     Local refinement level
                                patchInfo
                                {
                                        type wall;      Type of the patch.
                                                        This information is optional
                                }
                        }
                        ...
                        ...
                        ...
                }
        }
        ...
        ...
        ...
}
```

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.

- How do we control curvature refinement on surface patches?
- In the file *snappyHexMeshDict*, look for the following entry:

```
castellatedMeshControls
{
        ...
        ...
        ...
        refinementSurfaces
        {
                level (2 2);                    <- Global refinement level
                regions
                {
                        patch0                  <- Local surface patch
                        {
                                level (2 4);    <- Local curvature refinement (in red)
                                patchInfo
                                {
                                        type wall;
                                }
                        }
                        ...
                        ...
                        ...
                }
        }
        ...
        ...
        ...
}
```

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.

- How do we control curvature refinement on surface patches?
- In the file *snappyHexMeshDict*, look for the following entry:

```
castellatedMeshControls
{

    ...
    ...
    ...

    //Local curvature and
    //feature angle refinement
    resolveFeatureAngle 60;

    ...
    ...
    ...

}
```

The default value is 30.
Using a higher value will capture less features.

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.

- How do we control boundary layer meshing on the surface patches?
- In the file `snappyHexMeshDict`, look for the following entry:

```
addLayersControls
{

    //Global parameters
    relativeSizes true;
    expansionRatio 1.2;
    finalLayerThickness 0.5;
    minThickness 0.1;
    layers
    {
        "surface.*"
        {
            nSurfaceLayers 5;
        }
        surface0
        {
            nSurfaceLayers 3;
            expansionRatio 1.0;
            finalLayerThickness 0.25;
            minThickness 0.1;
        }
    }

    //Advanced settings
    ...
    ...
    ...
}
```

Global BL parameters

POSIX wildcards are permitted

Local surface patch

Local BL parameters

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.

- Let us first create the STL file with multiple surfaces.
- In the directory **geo**, you will find the original STL file.
- In the terminal type:

```
1. $> cd geo
2. $> surfaceAutoPatch geo.stl output.stl 130
3. $> cp output.stl ../constant/triSurface/surfacemesh.stl
4. $> cd ..
5. $> paraview
```

- The utility `surfaceAutoPatch` will read the original STL file (*geo.stl*), and it will find the patches using an angle criterion of 130 (similar to the angle criterion used with the utility `surfaceFeatures`). It writes the new STL geometry in the file *output.stl*.
- By the way, it is better to create the STL file with multiple surfaces directly in the solid modeler.
- FYI, there is an equivalent utility for meshes, `autoPatch`. So, if you forgot to define the patches, this utility will automatically find the patches according to an angle criterion.

snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.

- If you open the file *`output.stl`*, you will notice that there are three surfaces defined in the STL file. The different surfaces are defined in by the following sections:

solid patch0

… Surface patch 1

endsolid patch0

solid patch1

… Surface patch 2

endsolid patch1

solid patch2

… Surface patch 3

endsolid patch2

- The name of the solid sections are automatically given by the utility `surfaceAutoPatch`.
- The convention is as follows: patch0, patch1, patch2, … patchN.
- If you do not like the names, you can change them directly in the STL file.

snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.

- The new STL file is already in the **`constant/triSurface`** directory.
- To generate the mesh, in the terminal window type:

```
1.  $> foamCleanTutorials
2.  $> surfaceFeatures
3.  $> blockMesh
4.  $> snappyHexMesh -overwrite
5.  $> checkMesh –latestTime
```

- To visualize the mesh, in the terminal window type:

```
6.  $> paraFoam
```

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.

- This case is ready to run using the solver `simpleFoam`. But before running, you will need to set the boundary and initial conditions.
- You will need to manually modify the file *`constant/polyMesh/boundary`*
- Remember:
  - **Base type** boundary conditions are defined in the file *`boundary`* located in the directory **`constant/polyMesh`**.
  - **Numerical type** boundary conditions are defined in the field variables files located in the directory **0** or the time directory from which you want to start the simulation (e.g., *`U`*, *`p`*).
  - The name of the base type boundary conditions and numerical type boundary conditions needs to be the same.
  - Also, the base type boundary condition needs to be compatible with the numerical type boundary condition.

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.

- This case is ready to run with `simpleFoam`.
- To run the case (mesh and simulation), type in the terminal,

1.
```
$> sh run_all.sh
```

- Feel free to open the files `run_mesh.sh` (meshing steps) and `run_solver.sh` (simulation steps) to get an idea of all steps used.
- The most critical step is to give the right name and type to the boundary patches, this is done in the file `boundary` and the input files located in the directory **0** (boundary conditions and initial conditions).

snappyHexMesh guided tutorials

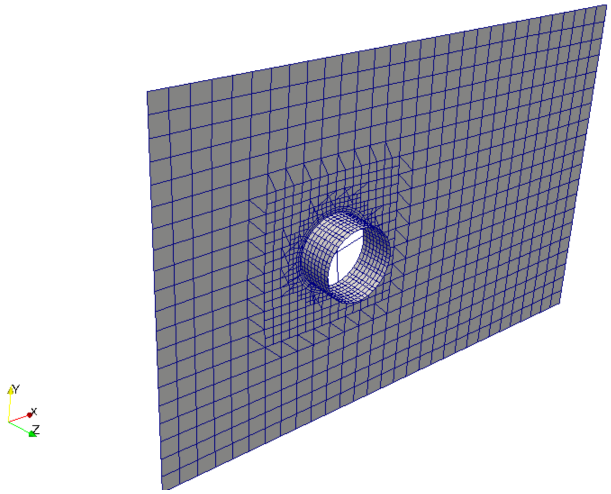
- Meshing with snappyHexMesh – Case 4.
- 2D cylinder (external mesh)
- You will find this case in the directory:

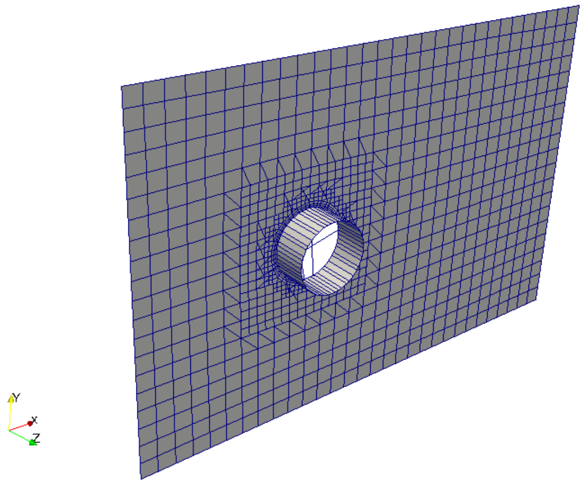
```
$PTOFC/101SHM_basic/M1_cyl/C4
```

- In the case directory, you will find a few scripts with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on.
- These scripts can be used to run the case automatically by typing in the terminal, for example,
  - `$> sh run_solver`
- These scripts are human-readable, and we highly recommend you open them, get familiar with the steps, and type the commands in the terminal. In this way, you will get used with the command line interface and OpenFOAM commands.
- If you are already comfortable with OpenFOAM, run the cases automatically using these scripts.
- In the case directory, you will also find the `README.FIRST` file. In this file, you will find some additional comments.

snappyHexMesh guided tutorials

2D Cylinder



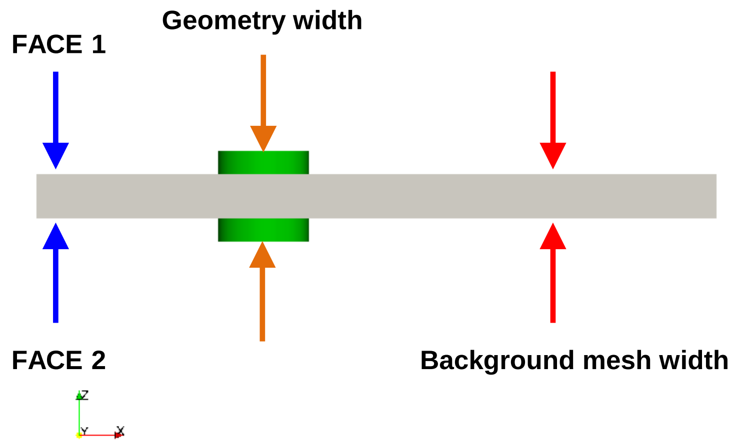


From **3D** → To **2D**

- To generate a 2D mesh using `snappyHexMesh`, we need to start from a 3D. After all, `snappyHexMesh` is a 3D mesher.
- To generate a 2D mesh (and after generating the 3D mesh), we use the utility `extrudeMesh`.
- The utility `extrudeMesh` works by projecting a face into a mirror face.
- Therefore, the faces need to parallel.

2D Cylinder

The utility `extrudeMesh` works by projecting FACE 1 into FACE 2. Therefore, the faces need to be parallel.



- At most, the input geometry and the background mesh need to have the same width.
- If the input geometry is larger than the background mesh, it will be automatically cut by the faces of the background mesh.
- In this case, the input geometry will be cut by the two lateral patches of the background mesh.
- If you want to take advantage of symmetry in 3D, you can cut the geometry in half using one of the faces of the background mesh.
- When dealing with 2D
- Extracting the features edges is optional for the 2D geometry extremes, but it is recommended if there are internal edges that you want to resolve.

2D Cylinder

- How do we create the 2D mesh?
- After generating the 3D mesh, we use the utility `extrudeMesh`.
- This utility reads the *`extrudeMeshDict`*,

```
constructFrom patch;

sourceCase "."
sourcePatches (minZ);

exposedPatchName maxZ;

extrudeModel linearNormal

nLayers 1;

linearNormalCoeffs
{
        thickness 1;
}

mergeFaces false;
```

Name of source patch

Name of the mirror patch

Number of layers to use in the linear extrusion.
As this is a 2D case we must use 1 layer

Thickness of the extrusion.
It is highly recommended to use a value of 1

2D Cylinder

- To generate the mesh, in the terminal window type:

```
1.  $> foamCleanTutorials
2.  $> blockMesh
3.  $> snappyHexMesh -overwrite
4.  $> extrudeMesh
5.  $> checkMesh -latestTime
6.  $> paraFoam
```

- Remember, the utility `extrudeMesh` (step 4) reads the dictionary *`extrudeMeshDict`*, which is located in the directory **`system`**.
- Also remember to set the empty patches in the dictionary *`boundary`* and in the boundary conditions.

Exercises

- To get a feeling of the surface refinement, try to change the value of the surface refinement in the dictionary *snappyHexMeshDict*.
- In the dictionary *snappyHexMeshDict*, change the value of **nCellsBetweenLevels** and **resolveFeatureAngle**. What difference do you see in the output?
- Use paraview to get a visual representation of the feature angles.
- In the dictionary *snappyHexMeshDict*, try to add curvature-based refinement.
- In the dictionary *snappyHexMeshDict*, in the section **addLayersControls** change the value of **featureAngle**. Use a value of 60 and 160 and compare the boundary layer meshing.
- To control the boundary layer collapsing, try to use a uniform surface refinement. For this you have two options, set surface level refinement to a uniform value or adding distance region refinement at the wall.
- To control the boundary layer collapsing, try to use absolute sizes when creating the boundary layer mesh.
- To get a feeling of region refinement, try to change the value of the local refinement in the dictionary *snappyHexMeshDict*. What difference do you see in the output?
- Try to use local inflation layers in the regions defined.
- In the dictionary *extrudeMeshDict*, change the value of **nLayers** and **thickness**.
- In the dictionary *extrudeMeshDict*, try to change the **extrudeModel**.

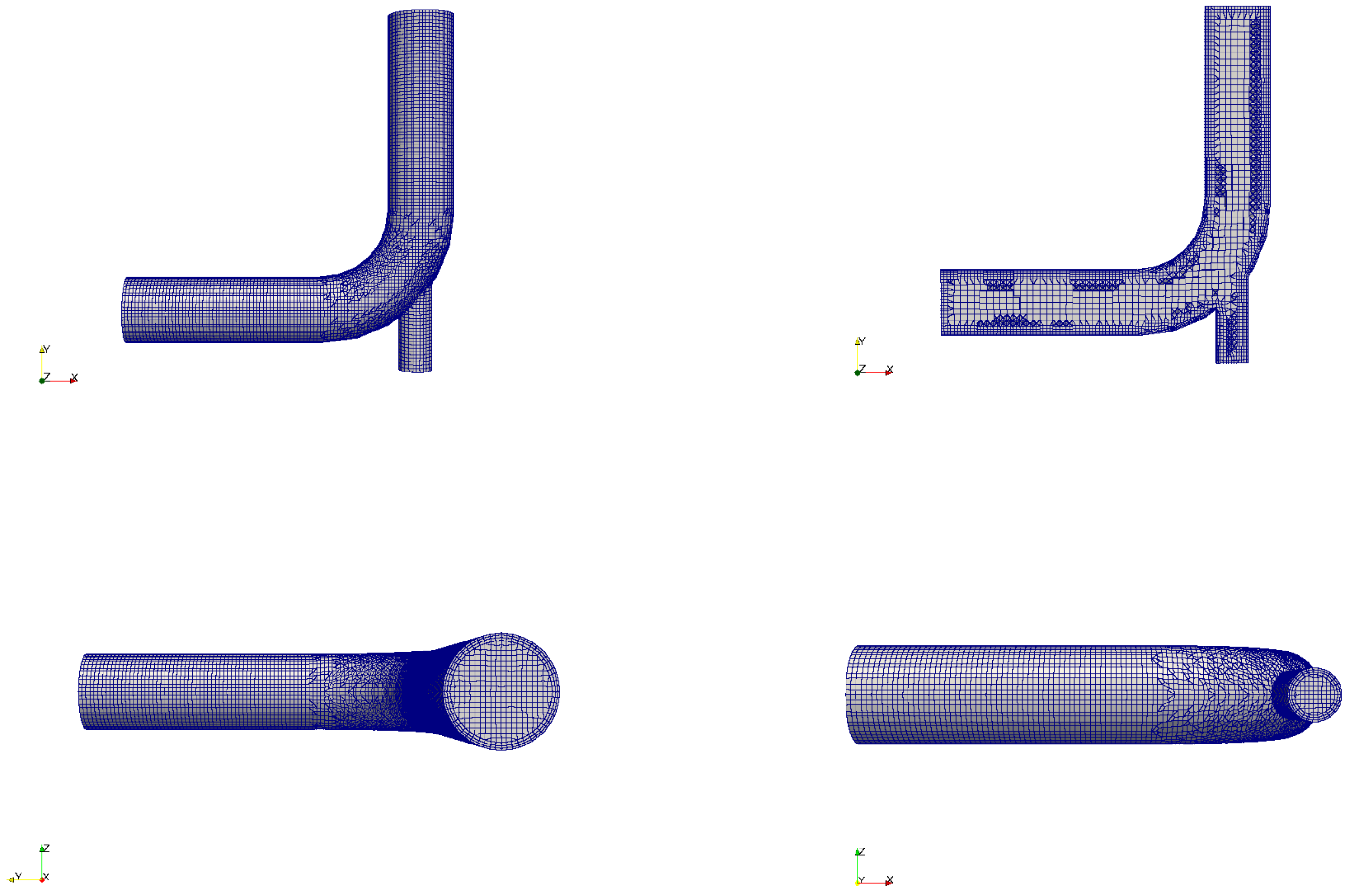
snappyHexMesh guided tutorials

- Meshing with snappyHexMesh – Case 5.
- Mixing elbow (internal mesh)
- You will find this case in the directory:

```
$PTOFC/101SHM_basic/M2_mixing_elbow
```

- In the case directory, you will find a few scripts with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on.
- These scripts can be used to run the case automatically by typing in the terminal, for example,
  - `$> sh run_solver`
- These scripts are human-readable, and we highly recommend you open them, get familiar with the steps, and type the commands in the terminal. In this way, you will get used with the command line interface and OpenFOAM commands.
- If you are already comfortable with OpenFOAM, run the cases automatically using these scripts.
- In the case directory, you will also find the `README.FIRST` file. In this file, you will find some additional comments.

Mixing elbow.



Your final mesh should look like this one

Mixing elbow.

- How do we control surface refinement using region refinement?
- In the file *snappyHexMeshDict*, look for the following entry:

```
castellatedMeshControls
{
        ...
        ...
        ...

        refinementRegions
        {
                mixing_elbow                    Name of surface
                {
                        mode distance;          Refinement using distance mode
                        levels ((1e-4 1));
                }
        }
                          Distance from          Refinement level
                          the surface patch
        ...
        ...
        ...
}
```

Mixing elbow.

- In this case we are going to generate a body fitted mesh with edge refinement and boundary layer meshing.
- This is an internal mesh.
- These are the dictionaries and files that will be used.
    - *system/snappyHexMeshDict*
    - *system/surfaceFeaturesDict*
    - *system/meshQualityDict*
    - *system/blockMeshDict*
    - *constant/triSurface/surfacemesh_multi.stl*
    - *constant/triSurface/surfacemesh_multi.eMesh*

- The file *surfacemesh_multi.eMesh* is generated after using the utility `surfaceFeatures`, which reads the dictionary *surfaceFeaturesDict*.

Mixing elbow.

- At this point, we are going to work in parallel (but you can work in serial as well).
- To generate the mesh, in the terminal window type:

```
1.  $> foamCleanTutorials
2.  $> surfaceFeatures
3.  $> blockMesh
4.  $> decomposePar
5.  $> mpirun -np 4 snappyHexMesh -parallel -overwrite
6.  $> mpirun -np 4 checkMesh -parallel -latestTime
7.  $> reconstructParMesh -constant
8.  $> paraFoam
```

- Meshing in parallel is no different from meshing in serial. So, from now on feel free to run in parallel.
- Have in mind that `blockMesh` does not run in parallel.

Mixing elbow.

- So, what did we do?
    - Step 4: we distribute the mesh among the processors we want to use.
    - Step 5 and 6: we run in parallel.
    - Step 7: we put back together the decomposed mesh. Notice that to reconstruct a parallel mesh we use the command `reconstructParMesh`.
    - Step 8: we visualize the reconstructed mesh.

- Remember, to reconstruct parallel meshes we use the command,
    - `$> reconstructParMesh`
- Also remember to use the option `–constant` or `–time <time_folder>`.
- If you are dealing with adaptive mesh refinement (AMR), you will need to first reconstruct the parallel mesh with,
    - `$> reconstructParMesh`
- After reconstructing the mesh, you will need to reconstruct the fields using,
    - `$> reconstructPar`

Mixing elbow.

- Notice that the utility `blockMesh` does not run in parallel.
- Remember to set the keyword **numberOfSubdomains** in the dictionary *decomposeParDict* equal to the number of processors you want to use.
- In this case, as we are using 4 processors with mpirun, **numberOfSubdomains** needs to be equal to 4.
- To run the simulation and after reconstructing the mesh, you will need to transfer the boundary and initial conditions information to the decomposed mesh as follows,
  - `$> decomposePar -fields`
- Or you can force to decompose everything as follows,
  - `$> decomposePar -force`

Mixing elbow.

- After running `checkMesh`, you will get the following information regarding the patch names:

```
               Patch     Faces    Points                  Surface topology
 mixing_elbow_inlet1      1264      1297   ok (non-closed singly connected)
                pipe     38884     41118   ok (non-closed singly connected)
 mixing_elbow_inlet2       314       337   ok (non-closed singly connected)
 mixing_elbow_outlet      1264      1297   ok (non-closed singly connected)
```

- Sometimes you can get empty patches.

```
               Patch     Faces    Points                  Surface topology
                minX         0         0                        ok (empty)
                maxX         0         0                        ok (empty)
                minY         0         0                        ok (empty)
                maxY         0         0                        ok (empty)
                minZ         0         0                        ok (empty)
                maxZ         0         0                        ok (empty)
 mixing_elbow_inlet1      1264      1297   ok (non-closed singly connected)
                pipe     38884     41118   ok (non-closed singly connected)
 mixing_elbow_inlet2       314       337   ok (non-closed singly connected)
 mixing_elbow_outlet      1264      1297   ok (non-closed singly connected)
```

Mixing elbow.

- Empty patches are no problem, they remain from the background mesh.
- To erase the empty patches, you can do it manually (you will need to modify the file *boundary*), or you can use the utility `createPatch` as follows (the utility runs in parallel):
    - `$> createPatch -overwrite`

- The surface patch **pipe** was created in the geometry section of the dictionary *snappyHexMeshDict*.
- The patches **mixing_elbow_outlet**, **mixing_elbow_inlet1** and **mixing_elbow_inlet2** were created automatically by `snappyHexMesh`.
- You have the choice of giving the names of the patches yourself or letting `snappyHexMesh` assign the names automatically.
- Remember, when creating the boundary layer mesh, these are the names you need to use to assign the layers.

Mixing elbow.

- The mesh used in the previous case was a STL with multiple surfaces.
- In you do not create the regions in the geometry section of the dictionary *snappyHexMeshDict*, snappyHexMesh will automatically assign the names of the surface patches as follows:

- mixing_elbow_outlet
- mixing_elbow_inlet1
- mixing_elbow_inlet2

system/surfaceFeaturesDict

```
...
...

geometry
{
        surfacemesh.stl
        {
                type triSurfaceMesh;
                name mixing_elbow;
                regions
                {
                        pipe            NOTE 1
                        {
NOTE 2                          name pipe;
                        }
                }
        }
};

...
...
```

NOTE 1:
This is the name of the surface patch (or solid) in the STL file.

NOTE 2:
User-defined patch name (it can be different from the name in the STL file). This is the final name of the patch.
This is the name to be used when generating the inflation layer

Mixing elbow.

- The mesh used in the previous case was a STL with multiple surfaces.
- In you do not create the regions in the geometry section of the dictionary *snappyHexMeshDict*, `snappyHexMesh` will automatically assign the names of the surface patches as follows:

- mixing_elbow_outlet
- mixing_elbow_inlet1
- mixing_elbow_inlet2

constant/triSurface/surfacemesh.stl

```
solid outlet
...
...
...
solid outlet

solid inlet1
...
...
...
solid inlet1

solid inlet2
...
...
...
solid inlet2
```

Mixing elbow.

- If you do not want to use a single STL with multiple surfaces, you can load multiple surfaces in the dictionary *snappyHexMeshDict*.

```
geometry
{
        geo1.stl
        {
                type triSurfaceMesh;
                name geo1;
                regions
                {
                        geo1_region
                        {
                                name my_patch;
                        }
                }
         }

        geo2.stl
        {
                type triSurfaceMesh;
                name geo2;
         }

        geo3.stl
        {
                type triSurfaceMesh;
                name geo3;
         }
}
```

This is the name of the region or surface patch in the STL file

User-defined patch name. This is the final name of the patch.

Mixing elbow.

- The mesh used in the previous case was a STL with multiple surfaces.
- In the directory **`geometry`**, you fill find the file *`allss.stl`*, this STL has one single surface.
- Try to use this STL file to generate the mesh.
- You will notice that the final mesh has only one patch, namely **mixing_elbow** (or whatever name you chose).
- Also, it is not possible to have local control on the mesh refinement and boundary layer meshing.
- You will also face the conundrum that as there is only one surface patch, it is not possible to assign boundary conditions.

Mixing elbow.

- To solve the problem of the single surface patch, you can use the utility `autoPatch`. To do so, you can proceed as follows:
    - `$> autoPatch 60 -overwrite`
- The option `-overwrite`, will copy the new mesh in the directory **`constant/polyMesh`**.
- The utility `autoPatch` will use an angle criterion to find the patches, and will assign the name **auto0**, **auto1**, **auto2** and **auto3** to the new patches.
- The angle criterion is similar to that of the utility `surfaceFeatures`.
- The only difference is that it uses the complement of the angle. So, the smaller the angle the more patches it will find.
- The naming convention is **autoN**, where N is the patch number.
- Remember, `autoPatch` will manipulate the mesh located in the directory **`constant/polyMesh`**.
- FYI, `autoPatch` does not un in parallel.

Mixing elbow.

- To restart this case from the latest saved solution and do only the boundary layer meshing, modify the dictionary *snappyHexMeshDict* as follows:

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

- To generate the mesh restarting from the snapped mesh (or latest time), in the terminal window type :
  - `$> snappyHexMesh`
- Remember to set in the dictionary *controlDict* the entry **startFrom** to **latestTime** or the time directory where the snapped mesh is saved (in this case **2**).
- At this point, you can work in serial or parallel.

Exercises

- To get a feeling of the **includedAngle** value, try to change the value in the dictionary *surfaceFeaturesDict*.
- Remember the higher the **includedAngle** value, the more features you will capture.
- In the dictionary *snappyHexMeshDict*, change the value of **resolveFeatureAngle** (try to use a lower value), and check the mesh quality in the intersection between both pipes.
- In the **castellatedMeshControls** section, try to disable or modify the distance refinement of the **mixing_elbow** region (**refinementRegions**).
- What difference do you see in the output?
- Starting from the body fitted mesh, add 3 inflation layers at the walls (save the intermediate step).
- Try to add local surface refinement at the surface patch inlet2 (look at the STL file *constant/triSurface/surfacemesh_multi.stl*).
- Using paraview, extract the feature edge at the joint section of the pipes. Then, try to add local refinement at this feature edge.
- Try to add curvature refinement at the feature edge extracted from the surface patch inlet1.

Exercises

- Use the STL file with a single surface (*`surfacemesh_single.stl`*) and generate the same mesh, do not add inflation layers.
    - Use the utility `autopatch` to split the mesh in different surface patches. To get a feeling on how to use this utility, use different angle values. Try to get four surface patches.
    - After splitting the mesh in four surface patches, rename the boundary patches using the utility `createPatch`.
    - After renaming the boundary patches, change the type of each one using the utility `foamDictionary`.
    - Starting from the body fitted mesh, add 3 inflation layers at the walls (do not save the intermediate step).
    - **Hints: if you do not know how to use the utilities createPatch and foamDictionary, look at the script file run_mesh_single_surface.sh**

- After generating the mesh, setup a simple incompressible simulation (with no turbulence model).
    - Set the inlet velocity to 1 at both inlet patches and use a dynamic viscosity value equal to 0.01. Run the simulation in steady and unsteady mode.

snappyHexMesh guided tutorials

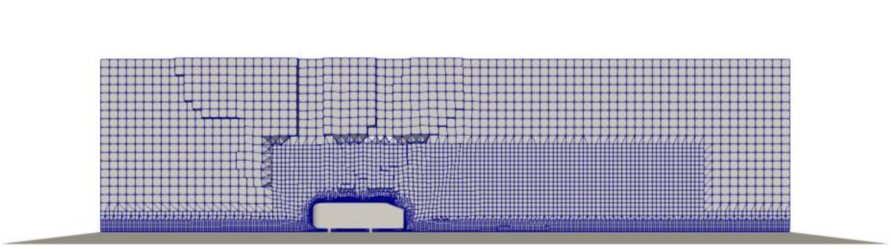
- Meshing with snappyHexMesh – Case 6.
- Ahmed body (external mesh).
- You will find this case in the directory:

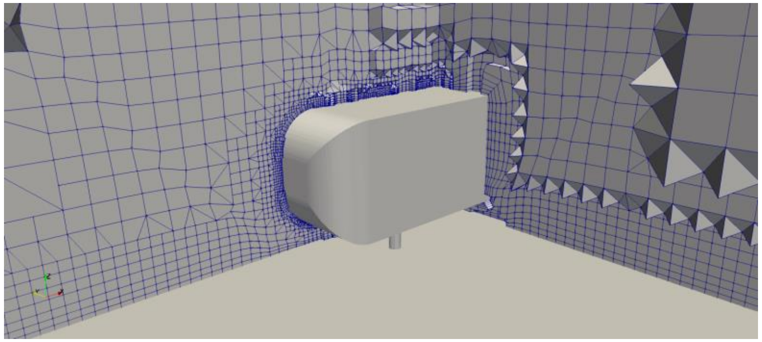
```
$PTOFC/101SHM_basic/M4_ahmed
```

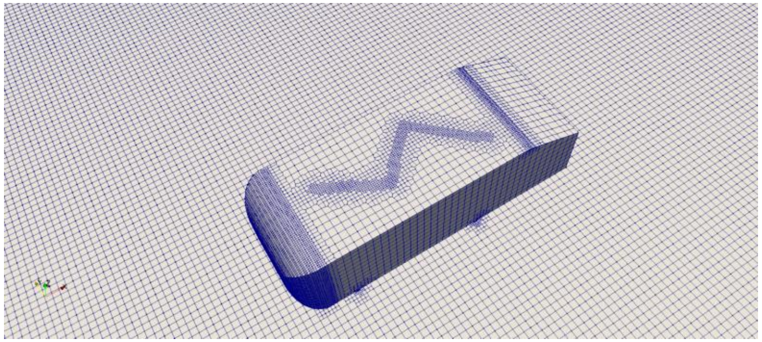
- In the case directory, you will find a few scripts with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on.
- These scripts can be used to run the case automatically by typing in the terminal, for example,
  - `$> sh run_solver`
- These scripts are human-readable, and we highly recommend you open them, get familiar with the steps, and type the commands in the terminal. In this way, you will get used with the command line interface and OpenFOAM commands.
- If you are already comfortable with OpenFOAM, run the cases automatically using these scripts.
- In the case directory, you will also find the `README.FIRST` file. In this file, you will find some additional comments.

snappyHexMesh guided tutorials

Ahmed body







- At this point, we all have a clear idea of how `snappyHexMesh` works.
- So let us go free styling and play around with this case.

Ahmed body

- In our YouTube channel you will find many instructional videos.
- You can find our YouTube channel in the following link: https://www.youtube.com/channel/UCNNBm3KxVS1rGeCVUU1p61g
- You can also find a playlist dedicated to this case.
- The playlist is titled: CFD workflow tutorial using open-source tools.
- You can find the playlist at this link: https://www.youtube.com/playlist?list=PLoI86R1JVvv-EN7BsoyomcWJIPaVPXaHO
- In these videos, we show a few extra features and some tips and tricks to take the most out of `snappyHexMesh`.

Ahmed body

- The dictionaries *snappyHexMeshDict* and *blockMeshDict* used in this case are very clean and ready to use.
- Feel free to use them as your templates
- Our best advice is not to get lost in all the options available in the dictionary *snappyHexMeshDict*.
- Most of the times the recommended options will work fine.
- That being said, you only need to follow the following steps:
    - Read in the geometries.
    - Set the feature edges and surface refinement levels.
    - Set region refinement (if required).
    - Choose in which surfaces you want to add the boundary layers.
    - Choose how many layers you want to add and follow the guidelines given during the lectures so you can get good inflation layers..

Ahmed body

- Final advice:
  - Use your solid modeling tool or paraFoam/paraview to get visual references.
  - Instead of using large surface refinement values, it is better to have finer background meshes.
    - Usually, surface refinement larger than 4 will give problems with the boundary layer.
    - To avoid very large background meshes, you can use mesh stretching or local refinement to concentrate more cells in the region close to the STL surface.
  - If you want to generate the boundary layer mesh, do it at the end using the restarting method.
  - If you are working with very complicated geometry, add one layer at a time.
  - Always check the quality of your mesh.

Exercises

- As an exercise, try to setup the boundary conditions and run the case with an inlet velocity of 30 m/s.
  - Run with simpleFoam for no more the 300 iterations.
  - Do not use turbulence model.
  - Use air standard properties.
  - Monitor the forces on the body.

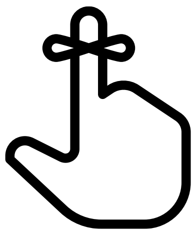
Roadmap

1. ~~Meshing preliminaries~~
2. ~~What is a good mesh?~~
3. ~~Mesh quality assessment in OpenFOAM®~~
4. ~~Mesh generation using blockMesh.~~
5. ~~Mesh generation using snappyHexMesh.~~
6. ~~snappyHexMesh guided tutorials.~~
7. **Mesh conversion**
8. Geometry and mesh manipulation utilities

Mesh conversion

- OpenFOAM® gives users a lot of flexibility when it comes to meshing.
- You are not constrained to use OpenFOAM® meshing tools.
- To convert a mesh generated with a third-party software to OpenFOAM® format, you can use the OpenFOAM® mesh conversion utilities.
- If your format is not supported, you can write your own conversion tool.
- By the way, many of the commercially available meshers can save the mesh in OpenFOAM® format or in a compatible format.
- In the directory **`$PTOFC/mesh_conversion_sandbox`** you will find a few meshes generated using the most popular third-party mesh generation applications.
- Feel free to play with these meshes.

- Remember to always check the file *`boundary`* after converting the mesh.
- You will need to change the **name** and **type** of the surface patches according to what you to do.
- When possible, save the mesh in ASCII format in the third-party meshing tools.
- Also, to convert the mesh you need to be in the top level of the case directory, and you need to give to the conversion utility the path (absolute or relative) of the mesh to be converted.



Mesh conversion

- In the directory **`$FOAM_UTILITIES/mesh/conversion`** you will find the source code for the mesh conversion utilities:

  - **ansysToFoam**
  - **cfx4ToFoam**
  - **datToFoam**
  - **fluent3DMeshToFoam**
  - **fluentMeshToFoam**
  - **foamMeshToFluent**
  - **foamToStarMesh**
  - **foamToSurface**
  - **gambitToFoam**
  - **gmshToFoam**
  - **ideasUnvToFoam**
  - **kivaToFoam**
  - **mshToFoam**
  - **netgenNeutralToFoam**
  - **Optional/ccm26ToFoam**
  - **plot3dToFoam**
  - **sammToFoam**
  - **star3ToFoam**
  - **star4ToFoam**
  - **tetgenToFoam**
  - **vtkUnstructuredToFoam**
  - **writeMeshObj**

- Take your time and read the instructions/comments contained in the source code of the mesh conversion utilities so you can understand how to use these powerful tools.

Mesh conversion

- Let us convert to OpenFOAM® format a mesh generated using Salome.
- You will find this case in the directory:

```
$PTOFC/mesh_conversion_sandbox/M1_mixing_elbow_salome
```

- In the case directory, you will find a few scripts with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on.
- These scripts can be used to run the case automatically by typing in the terminal, for example,
  - `$> sh run_solver`
- These scripts are human-readable, and we highly recommend you open them, get familiar with the steps, and type the commands in the terminal. In this way, you will get used with the command line interface and OpenFOAM commands.
- If you are already comfortable with OpenFOAM, run the cases automatically using these scripts.
- In the case directory, you will also find the `README.FIRST` file. In this file, you will find some additional comments.

Mesh conversion

Case 1. Mixing elbow (internal mesh).

- Remember to export the mesh in UNV format in Salome.
- Then use the utility `ideasUnvToFoam` to convert the mesh to OpenFOAM® native format.
- In the terminal window type:

1. `$> foamCleanTutorials`
2. `$> foamCleanPolyMesh`
3. `$> ideasUnvToFoam ../../meshes_and_geometries/salome_elbow3d/Mesh_1.unv`
4. `$> checkMesh`
5. `$> paraFoam`

- Remember to always check the file *boundary* after converting the mesh.
- To convert the mesh, you need to be in the top level of the case directory, and you need to give the path (absolute or relative) of the mesh to be converted.

Mesh conversion

Case 1. Mixing elbow (internal mesh).

- `ideasUnvToFoam` output.

```
Create time

Processing tag:2411
Starting reading points at line 3.
Read 31136 points.

Processing tag:2412
Starting reading cells at line 62278.
First occurrence of element type 11 for cell 1 at line 62279
First occurrence of element type 41 for cell 361 at line 63359
First occurrence of element type 111 for cell 20933 at line 104503
Read 151064 cells and 20572 boundary faces.

Processing tag:2467
Starting reading patches at line 406633.
For group 1 named pipe trying to read 19778 patch face indices.
For group 2 named inlet1 trying to read 358 patch face indices.
For group 3 named inlet2 trying to read 78 patch face indices.
For group 4 named outlet trying to read 358 patch face indices.

Sorting boundary faces according to group (patch)
0: pipe is patch
1: inlet1 is patch
2: inlet2 is patch
3: outlet is patch

Constructing mesh with non-default patches of size:
    pipe        19778
    inlet1      358
    inlet2      78
    outlet      358

End
```

Internal cells and boundary faces read

Boundary patches detected

Mesh conversion

Case 1. Mixing elbow (internal mesh).

- `checkMesh` output.

```
Mesh stats
    points:           31136
    faces:            312414
    internal faces:   291842
    cells:            151064
    faces per cell:   4
    boundary patches: 4
    point zones:      0
    face zones:       0
    cell zones:       0

Overall number of cells of each type:
    hexahedra:     0
    prisms:        0
    wedges:        0
    pyramids:      0
    tet wedges:    0
    tetrahedra:    151064
    polyhedra:     0

Checking topology...
    Boundary definition OK.
    Cell to face addressing OK.
    Point usage OK.
    Upper triangular ordering OK.
    Face vertices OK.
    Number of regions: 1 (OK).
```

Mesh conversion

Case 1. Mixing elbow (internal mesh).

- `checkMesh` output.

```
 Checking patch topology for multiply connected surfaces...
    Patch               Faces    Points   Surface topology
    pipe                19778    9938     ok (non-closed singly connected)
    inlet1              358      200      ok (non-closed singly connected)
    inlet2              78       50       ok (non-closed singly connected)
    outlet              358      200      ok (non-closed singly connected)

Checking geometry...
    Overall domain bounding box (0 -0.414214 -0.5) (5 5 0.5)
    Mesh has 3 geometric (non-empty/wedge) directions (1 1 1)
    Mesh has 3 solution (non-empty) directions (1 1 1)
    Boundary openness (-1.0302e-17 -6.17232e-17 -1.77089e-16) OK.
    Max cell openness = 2.32045e-16 OK.
    Max aspect ratio = 4.67245 OK.
    Minimum face area = 0.000286852. Maximum face area = 0.010949.  Face area magnitudes OK.
    Min volume = 2.74496e-06. Max volume = 0.00035228.  Total volume = 6.75221.  Cell volumes OK.
    Mesh non-orthogonality Max: 54.2178 average: 15.1295
    Non-orthogonality check OK.
    Face pyramids OK.
    Max skewness = 0.649359 OK.
    Coupled point location match (average 0) OK.

Mesh OK.

End
```

Everything is OK

Mesh conversion

Case 1. Mixing elbow (internal mesh).

- The *boundary* file.

```
4          Number of boundary patches
(
    pipe
    {
        type            patch;
        nFaces          19778;
        startFace       291842;
    }
    inlet1
    {
        type            patch;
        nFaces          358;
        startFace       311620;
    }
    inlet2
    {
        type            patch;
        nFaces          78;
        startFace       311978;
    }
    outlet
    {
        type            patch;
        nFaces          358;
        startFace       312056;
    }
)
```

Name of the boundary patches

- In this case, the utility recognized the name of the boundary patches.
- If you do not like the names feel free to change them.
- Remember, do not use spaces of strange symbols.

Mesh conversion

Case 1. Mixing elbow (internal mesh).

- The *boundary* file.

```
4
(
    pipe
    {
        type            patch;
        nFaces          19778;
        startFace       291842;
    }
    inlet1
    {
        type            patch;
        nFaces          358;
        startFace       311620;
    }
    inlet2
    {
        type            patch;
        nFaces          78;
        startFace       311978;
    }
    outlet
    {
        type            patch;
        nFaces          358;
        startFace       312056;
    }
)
```

Base type of the boundary patches

- In this case, the utility automatically assigned the **base type patch** to all boundary patches.
- Feel free to change the **base type** according to your needs.
- In this case, it will be wise to change the **base type** of patch **pipe** to **wall**.

Mesh conversion

Exercises

- Remember, you can change the name and type of the boundary patches manually, but as we want to do things automatically, we will use the utilities `createPatch` and `foamDictionary`
  - After converting the mesh to OpenFOAM® format, rename the boundary patches using the utility `createPatch`.
  - After converting the mesh to OpenFOAM® format, change the type of each boundary patch using the utility `foamDictionary`.
- After converting the mesh to OpenFOAM® format, add 5 inflation layers at the walls and save the intermediate step.
- Check the mesh quality before and after adding the inflation layers.
- After converting/generating the mesh, setup a simple incompressible simulation (with no turbulence model).
  - Set the inlet velocity to 1 at both inlet patches and use a dynamic viscosity value equal to 0.01.
  - Run the simulation in steady and unsteady mode.

Roadmap

1. ~~Meshing preliminaries~~
2. ~~What is a good mesh?~~
3. ~~Mesh quality assessment in OpenFOAM®~~
4. ~~Mesh generation using blockMesh.~~
5. ~~Mesh generation using snappyHexMesh.~~
6. ~~snappyHexMesh guided tutorials.~~
7. ~~Mesh conversion~~
8. **Geometry and mesh manipulation utilities**

Geometry and mesh manipulation utilities

- First of all, by mesh manipulation we mean modifying a valid OpenFOAM® mesh.
- These modifications can be scaling, rotation, translation, mirroring, topological changes, mesh refinement and so on.
- In the directory **`$FOAM_UTILITIES/mesh/manipulation`** you will find the mesh manipulation utilities. Just to name a few:

  - **autoPatch**
  - **checkMesh**
  - **createBaffles**
  - **mergeMeshes**
  - **splitBaffles**
  - **mirrorMesh**
  - **polyDualMesh**
  - **refineMesh**
  - **renumberMesh**
  - **rotateMesh**
  - **setsToZones**
  - **splitMesh**
  - **splitMeshRegions**
  - **stitchMesh**
  - **subsetMesh**
  - **topoSet**
  - **transformPoints**
  - **mergeBaffles**

Geometry and mesh manipulation utilities

- In the directory **`$FOAM_UTILITIES/mesh/manipulation`** you will find the following mesh manipulation utilities.
- Inside each utility directory you will find a *`*.C`* file with the same name as the directory. This is the main file, where you will find the top-level source code and a short description of the utility.
- For instance, in the directory **`checkMesh`**, you will find the file *`checkMesh.C`*, which is the source code of the utility `checkMesh`. In the source code you will find the following description:

```
Description
    Checks validity of a mesh.

Usage
    \b checkMesh [OPTION]

    Options:
      - \par -allGeometry
        Checks all (including non finite-volume specific) geometry

      - \par -allTopology
        Checks all (including non finite-volume specific) addressing

      - \par -meshQuality
        Checks against user defined (in \a system/meshQualityDict) quality
        settings

      - \par -region \<name\>
        Specify an alternative mesh region.

      - \par -writeSets \<surfaceFormat\>
        Reconstruct all cellSets and faceSets geometry and write to
        postProcessing directory according to surfaceFormat
        (e.g. vtk or ensight). Additionally reconstructs all pointSets and
        writes as vtk format.
```

Geometry and mesh manipulation utilities

- In OpenFOAM® it is also possible to manipulate the geometries in STL format.
- These modifications can be scaling, rotation, translation, mirroring, topological changes, normal orientation, and so on.
- In the directory **`$FOAM_UTILITIES/surface`** you will find the mesh manipulation utilities. Just to name a few:

- **surfaceAdd**
- **surfaceAutoPatch**
- **surfaceBooleanFeatures**
- **surfaceCheck**
- **surfaceConvert**
- **surfaceFeatureConvert**
- **surfaceFeatures**
- **surfaceInertia**
- **surfaceMeshConvert**
- **surfaceMeshExport**
- **surfaceMeshTriangulate**
- **surfaceOrient**
- **surfaceSplitByPatch**
- **surfaceSubset**
- **surfaceToPatch**
- **surfaceTransformPoints**

Geometry and mesh manipulation utilities

- In the directory **`$FOAM_UTILITIES/surface`** you will find the following surface manipulation utilities.
- Inside each utility directory you will find a *`*.C`* file with the same name as the directory. This is the main file, where you will find the top-level source code and a short description of the utility.
- For instance, in the directory **`surfaceTransformPoints`**, you will find the file *`surfaceTransformPoints.C`*, which is the source code of the utility `surfaceTransformPoints`. In the source code you will find the following description:

```
Description
    Transform (translate, rotate, scale) a surface.

Usage
    \b surfaceTransformPoints "\<transformations\>" \<input\> \<output\>
    Supported transformations:
      - \par translate=<translation vector>
        Translational transformation by given vector
      - \par rotate=(\<n1 vector\> \<n2 vector\>)
        Rotational transformation from unit vector n1 to n2
      - \par Rx=\<angle [deg] about x-axis\>
        Rotational transformation by given angle about x-axis
      - \par Ry=\<angle [deg] about y-axis\>
        Rotational transformation by given angle about y-axis
      - \par Rz=\<angle [deg] about z-axis\>
        Rotational transformation by given angle about z-axis
      - \par Ra=\<axis vector\> \<angle [deg] about axis\>
        Rotational transformation by given angle about given axis
      - \par scale=\<x-y-z scaling vector\>
        Anisotropic scaling by the given vector in the x, y, z
        coordinate directions
```

Geometry and mesh manipulation utilities

- Let us do some surface manipulation.
- For this we will use the ahmed body tutorial.
- You will find this case in the directory:

```
$PTOFC/mesh_quality_manipulation/M5_ahmed_body_transform
```

- In the case directory, you will find a few scripts with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on.
- These scripts can be used to run the case automatically by typing in the terminal, for example,
  - `$> sh run_solver`
- These scripts are human-readable, and we highly recommend you open them, get familiar with the steps, and type the commands in the terminal. In this way, you will get used with the command line interface and OpenFOAM commands.
- If you are already comfortable with OpenFOAM, run the cases automatically using these scripts.
- In the case directory, you will also find the `README.FIRST` file. In this file, you will find some additional comments.

Geometry and mesh manipulation utilities

Geometry manipulation in OpenFOAM®

- We will now manipulate a STL geometry. In the terminal type:

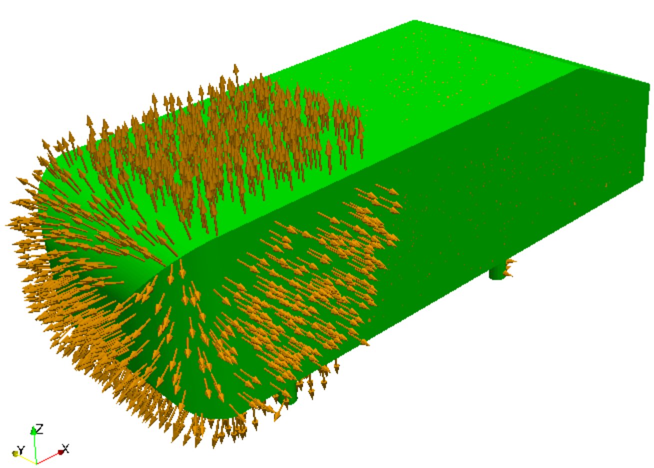
1.
```
$> foamCleanTutorials
```
2.
```
$> surfaceMeshInfo ./constant/triSurface/ahmed_body.stl
```
3.
```
$> surfaceCheck ./constant/triSurface/ahmed_body.stl
```
4.
```
$> surfaceTransformPoints Rx=15
   ./constant/triSurface/ahmed_body.stl rotated.stl
```
5.
```
$> surfaceTransformPoints translate='(0 0.12 0)'
   ./constant/triSurface/ahmed_body.stl   translated.stl
```
6.
```
$> surfaceTransformPoints scale='(0.9 1.1 1.3)'
    ./constant/triSurface/ahmed_body.stl   scaled.stl
```
7.
```
$> surfaceInertia -density 2700 ./constant/triSurface/ahmed_body.stl
```
8.
```
$> surfaceOrient ./constant/triSurface/ahmed_body_wrong_normals.stl
   out.stl '(1e10 1e10 1e10)'
```

Geometry manipulation in OpenFOAM®

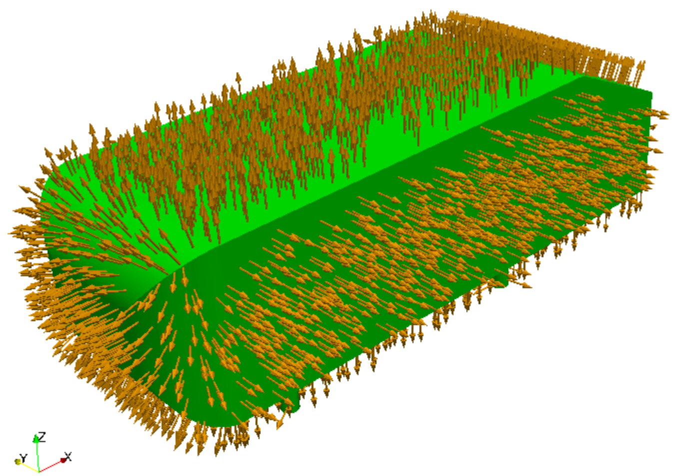
- In step 2 we use the utility `surfaceMeshInfo` to get general information about the STL (such as number of faces and so on).
- In step 3 we use the utility `surfaceCheck` to check the STL file.
- In step 4 we use the utility `surfaceTransformPoints` to rotate the STL 15 degrees about the X axis. We read in the STL *./constant/triSurface/ahmed_body.stl* and we write out the STL *rotated.stl*
- In step 5 we use the utility `surfaceTransformPoints` to translate the STL. We read in the STL *./constant/triSurface/ahmed_body.stl* and we write out the STL *translated.stl*
- In step 6 we use the utility `surfaceTransformPoints` to scale the STL. We read in the STL *./constant/triSurface/ahmed_body.stl* and we write out the STL *scaled.stl*
- In step 7 we use the utility `surfaceInertia` to compute the inertia of the STL. We read in the STL *./constant/triSurface/ahmed_body.stl*. Notice that we need to give a reference density value.
- In step 8 we use the utility `surfaceOrient` to orient the normals of the STL in the same way. We read in the STL `./constant/triSurface/ahmed_body_wrong_normals.stl` and we write out the STL *out.stl*. Notice that we give an outside point or `‘(1e10 1e10 1e10)’`, if this point is outside the STL all normals will be oriented outwards, if the point is inside the STL all normals will be oriented inwards.

Geometry manipulation in OpenFOAM®

- Pay particular attention to step 8.
- We already have seen that snappyHexMesh computes surface angles using the surface normals as a reference, so it is extremely important to have the normals oriented in the same way and preferably outwards.



ahmed_body_wrong_normals.stl



STL after orienting all normals in the same direction.

Geometry manipulation in OpenFOAM®

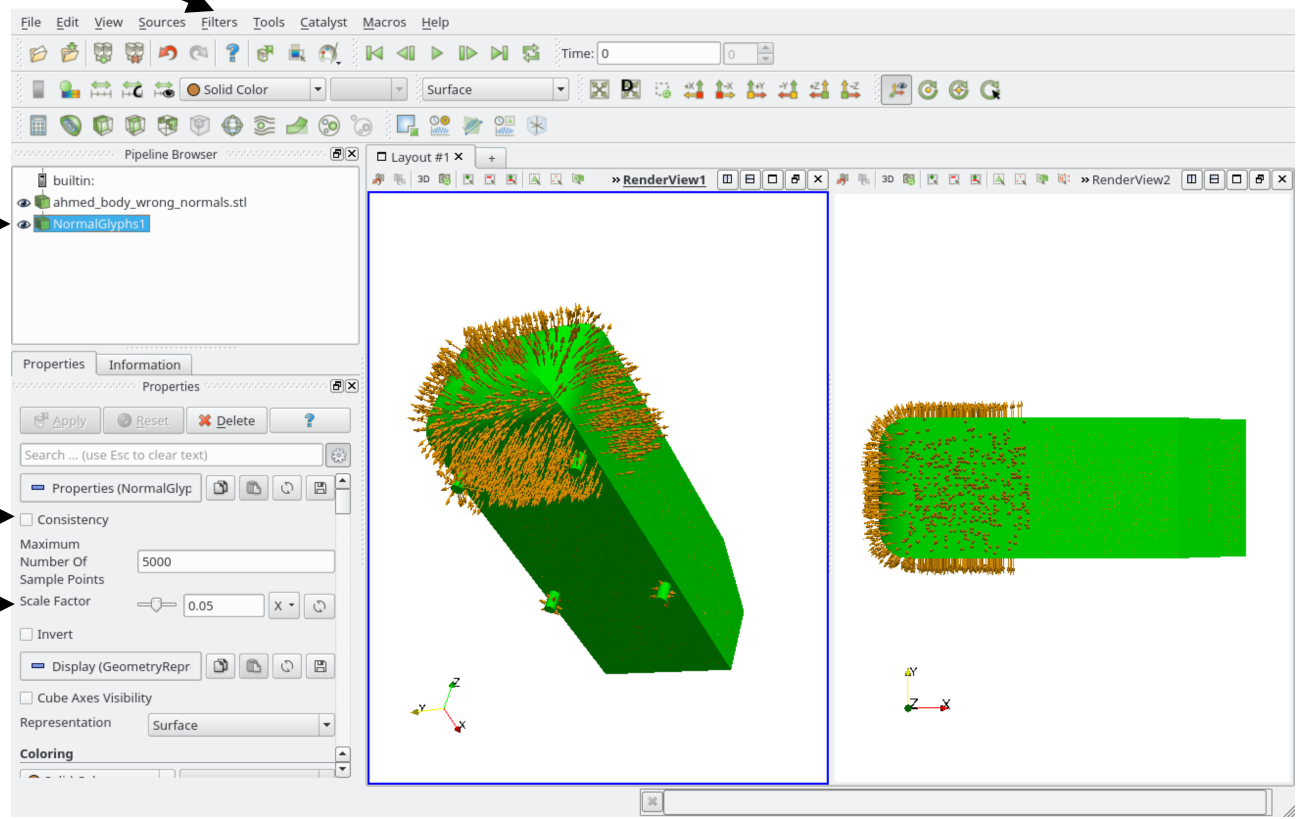
- To plot the normals in paraview/paraFoam you can use the filter `Normal Glyphs`

Select the `Normal Glyphs` from the filter menu

Apply the `Normal Glyphs` filter to the STL

Uncheck this option

Scale the vectors to fit the screen



Geometry and mesh manipulation utilities

- Let us do some mesh manipulation.
- For this we will use the 2D cylinder tutorial.
- You will find this case in the directory:

```
$PTOFC/mesh_quality_manipulation/M7_cylinder_transform
```

- In the case directory, you will find a few scripts with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on.
- These scripts can be used to run the case automatically by typing in the terminal, for example,
  - `$> sh run_solver`
- These scripts are human-readable, and we highly recommend you open them, get familiar with the steps, and type the commands in the terminal. In this way, you will get used with the command line interface and OpenFOAM commands.
- If you are already comfortable with OpenFOAM, run the cases automatically using these scripts.
- In the case directory, you will also find the `README.FIRST` file. In this file, you will find some additional comments.

Mesh manipulation in OpenFOAM®

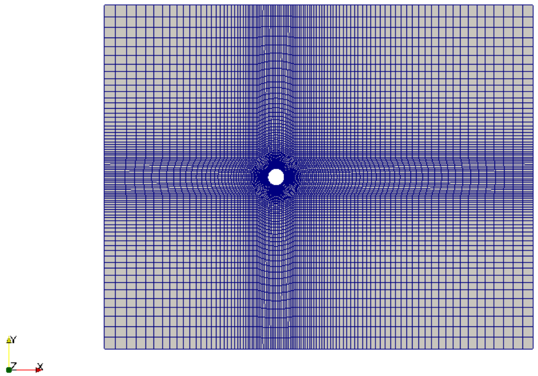
- We will now manipulate a mesh. In the terminal type:

1. `$> foamCleanTutorials`
2. `$> blockMesh`
3. `$> transformPoints 'Rz=90'`
4. `$> transformPoints 'scale=(0.01 0.01 0.01)'`
5. `$> transformPoints 'translate=(0 0 1)'`
6. `$> createPatch -noFunctionObjects -overwrite`
7. `$> checkMesh`
8. `$> paraFoam`

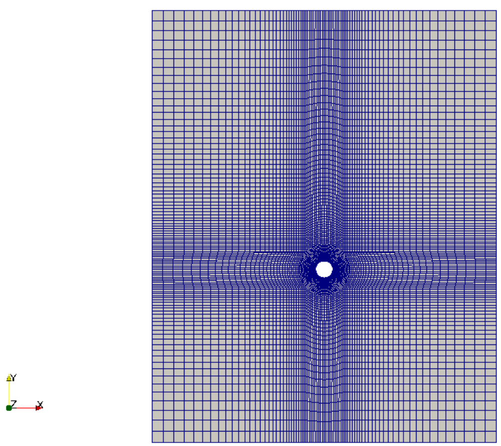
- In step 3 we use the utility `transformPoints` to rotate the mesh. We rotate the mesh by 90° about the **Z** axis.
- In step 4 we use the utility `transformPoints` to scale the mesh. We scale the mesh by a factor of `'(0.01 0.01 0.01)'`.
- In step 5 we use the utility `transformPoints` to translate the mesh. We translate the mesh by the vector `'(0 0 1)'`.
- In step 6 we use the utility `createPatch` to rename the patches of the mesh. This utility reads the dictionary `system/createPatchDict`. Instead of using the utility `createPatch` we could have modified the *`boundary`* file directly.
- This case is ready to run using the solver `buoyantBoussinesqPimpleFoam`.

Geometry and mesh manipulation utilities

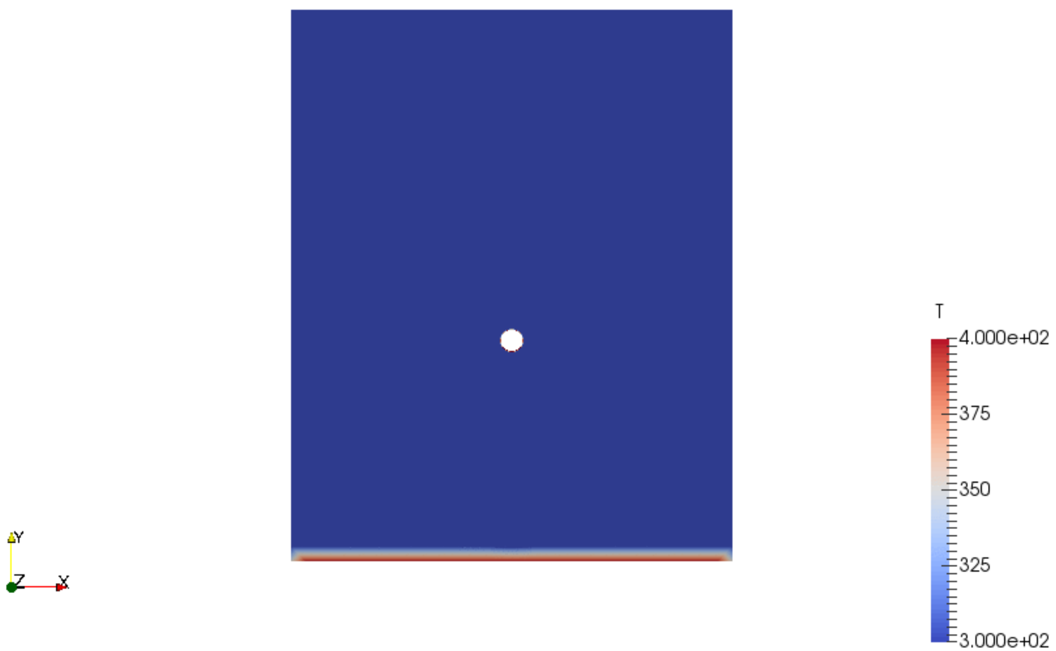
Mesh manipulation in OpenFOAM®



Original mesh



Transformed mesh



After renaming the patches and transforming the mesh, we can use it to conduct this buoyant flow simulation

www.wolfdynamics.com/wiki/heated_cyl/ani1.gif