2017 Online Training – Advanced session

Advanced meshing using OpenFOAM® technology: cfMesh

wolf dynamics

Copyright and disclaimer

This offering is not approved or endorsed by OpenCFD Limited, the producer of the OpenFOAM software and owner of the OPENFOAM® and OpenCFD® trade marks.

Wolf Dynamics makes no warranty, express or implied, about the completeness, accuracy, reliability, suitability, or usefulness of the information disclosed in this training material. This training material is intended to provide general information only. Any reliance the final user place on this training material is therefore strictly at his/her own risk. Under no circumstances and under no legal theory shall Wolf Dynamics be liable for any loss, damage or injury, arising directly or indirectly from the use or misuse of the information contained in this training material.

Revision 1-2017

Before we begin

On the training material

- This training is based on OpenFOAM 4.x and cfMesh 1.1.2
- In the USB key you will also find all the training material in a compressed file.
- You can extract the training material wherever you want. From now on, this directory will become:
    - **`$TM`**
- To uncompress the training material go to the directory where you copied it and then type in the terminal,
    - `$> tar -zxvf file_name.tar.gz`
- In each tutorial directory there is a README.FIRST file. In this file you will find general comments and the instructions of how to run the case

Conventions used

- The following typographical conventions are used in this training material:

  - `Courier new`

    Indicates Linux commands that should be typed literally by the user in the terminal
  - **`Courier new bold`**

    Indicates directories
  - *`Courier new italic`*

    Indicates human readable files or ascii files
  - **Arial bold**

    Indicates program elements such as variables, function names, classes, databases, data types, environment variables, statements and keywords. They also highlight important information.
  - Arial underline in blue

    Indicates URLs and email addresses

Conventions used

- The following typographical conventions are used in this training material:

  - Large code listing, ascii files listing, and screen outputs can be written in a square box, as follows:

```
#include <iostream>
using namespace std;

// main() is where program execution begins.  It is the main function.
// Every program in c++ must have this main function declared

int main ()
{
        cout << "Hello world";                    //prints Hello world
        return 0;                                 //returns nothing
}
```

  - To improve readability, the text might be colored.
  - The font can be `Courier new` or **Arial bold**.
  - And when required, the line number will be shown.

Conventions used

- The following typographical conventions are used in this training material:

 This icon indicates a warning or a caution

 This icon indicates a tip, suggestion, or a general note

 This icon indicates that more information is available in the referred location

 This icon indicates a folder or directory

 This icon indicates an ascii file

$> This symbol indicates that a Linux command should be typed literally by the user in the terminal

 This icon indicates that the figure is an animation (animated gif)

Roadmap

1. Mesh quality assessment in CFD
2. Mesh generation using cfMesh
3. The cylinder tutorial
4. The 2D airfoil tutorial
5. The static mixer tutorial
6. The Ahmed body tutorial
7. The mixing elbow comparison
8. The moving quadcopter tutorial

Roadmap

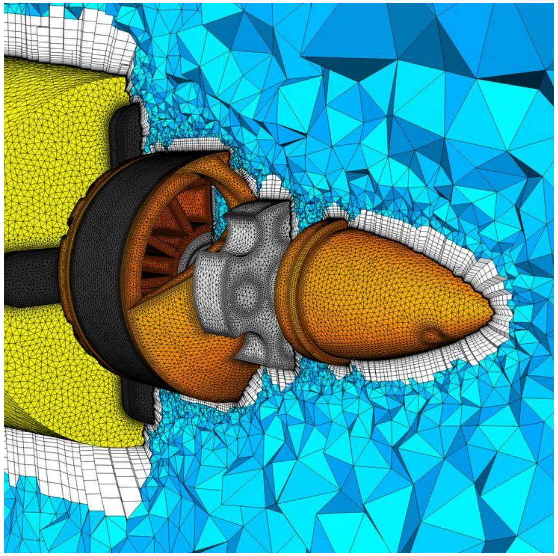
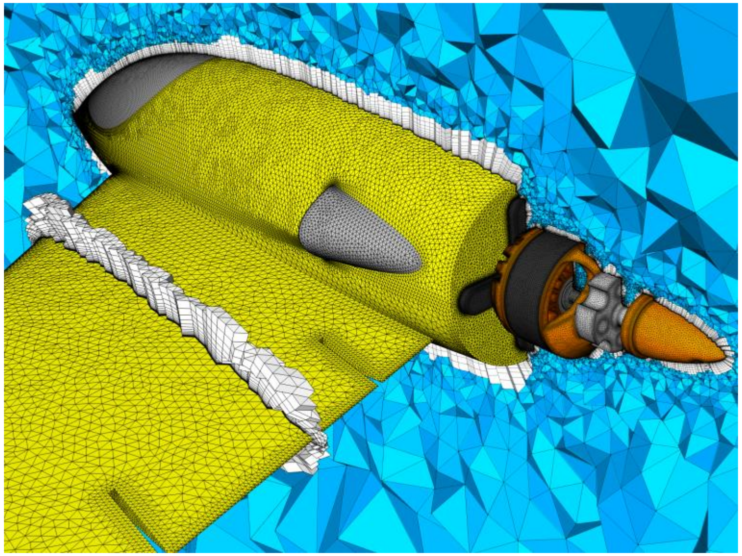
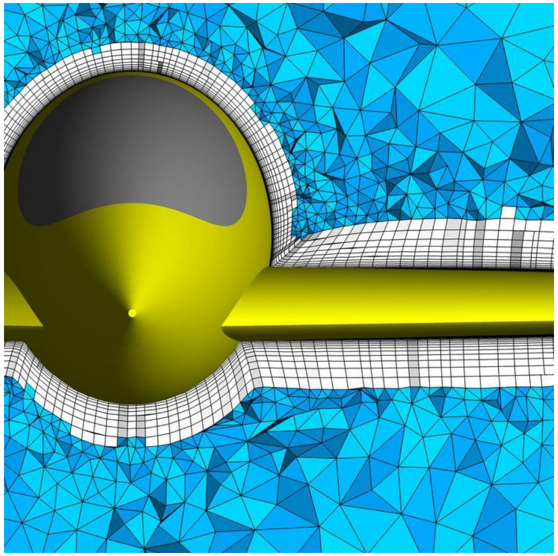
1. **Mesh quality assessment in CFD**
2. Mesh generation using cfMesh
3. The cylinder tutorial
4. The 2D airfoil tutorial
5. The static mixer tutorial
6. The Ahmed body tutorial
7. The mixing elbow comparison
8. The moving quadcopter tutorial

Mesh quality assessment in CFD

- In CFD, the mesh is everything.
- So try to always get good quality meshes.
- With that in mind, remember to always check the quality of the mesh.
- Do not go into the solution stage unless you have an acceptable/good mesh.
- So, what is a good mesh?

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.

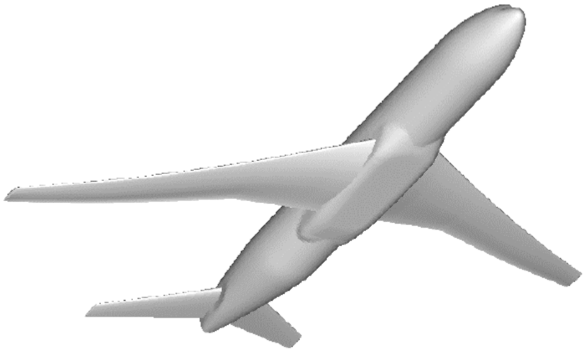




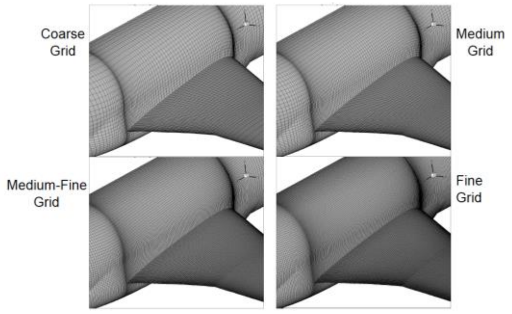
22rd IMR Meshing Maestro Contest Winner
Travis Carrigan, John Chawner and Carolyn Woeber. Pointwise.
http://imr.sandia.gov/22imr/MeshingContest.html

What is a good mesh?

- The user can rely on grid dependency studies, but they are time consuming and expensive.

	Coarse mesh	Medium mesh	Fine mesh	Extra fine mesh
Cells	≈ 3 500 000	≈ 11 000 000	≈ 36 000 000	≈ 105 000 000
C_D	0.0282	0.0270	0.0268	0.0269
C_M	-0.0488	-0.0451	-0.0391	-0.0391





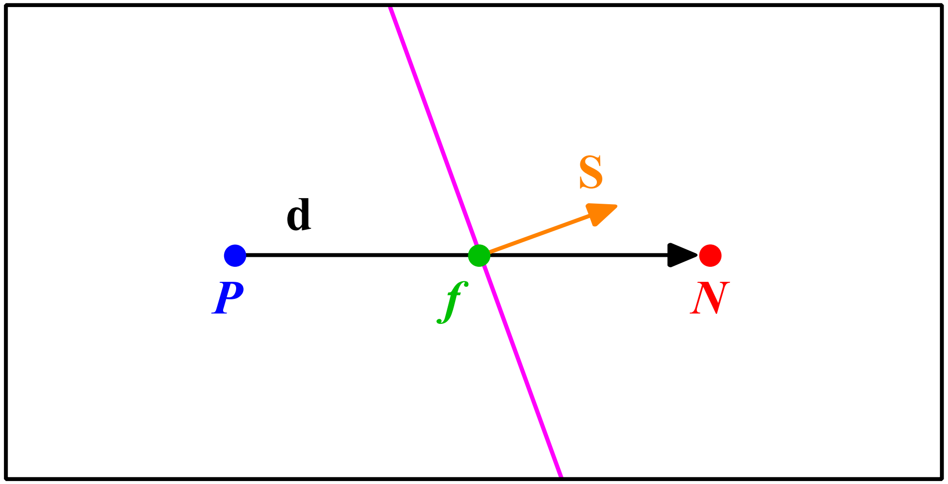
Results from AIAA CFD Drag Prediction Workshop

What is a good mesh?

- 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 mesh quality.
- Have in mind that there are many more mesh quality metrics out there, some of them are not very easy to interpret (*e.g.,* jacobian matrix, determinant, flatness, equivalence, condition number, and so on).
- It seems that it is much easier diagnosing bad meshes than good meshes.

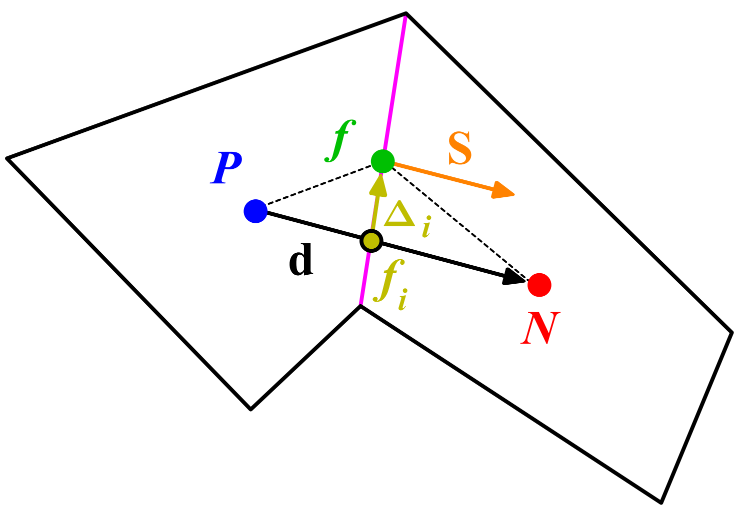
Mesh quality metrics. Mesh orthogonality

- 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°.
- Affects the gradient of the face center ***f***.
- It adds diffusion to the solution.
- It mainly affects the diffusive terms.



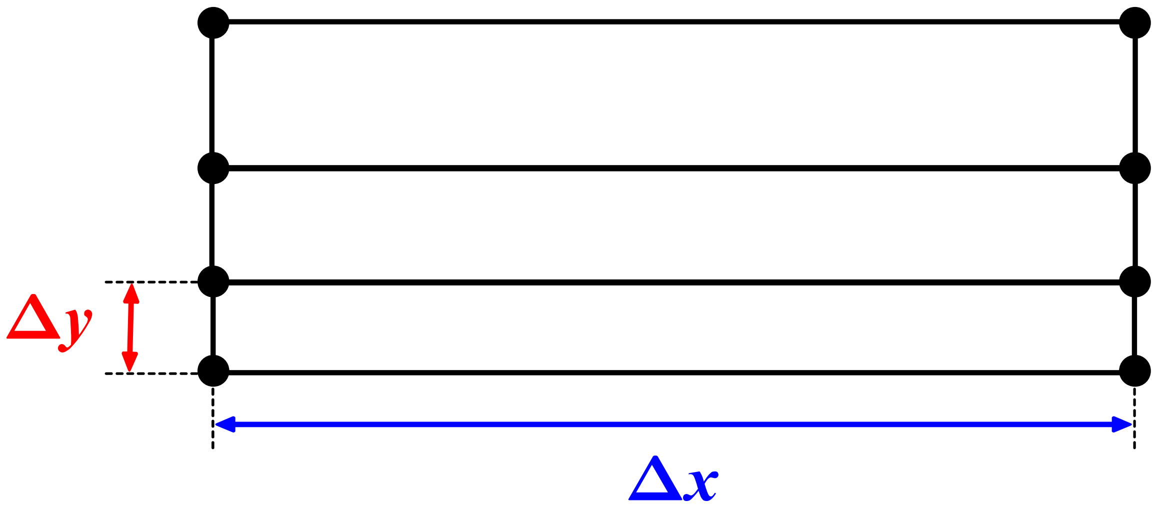
Mesh quality metrics. Mesh skewness

- Skewness 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*** .
- Affects the interpolation of the cell centered quantities to the face center ***f***.
- It adds diffusion to the solution.
- It affects the convective terms.



Mesh quality metrics. Mesh aspect ratio AR

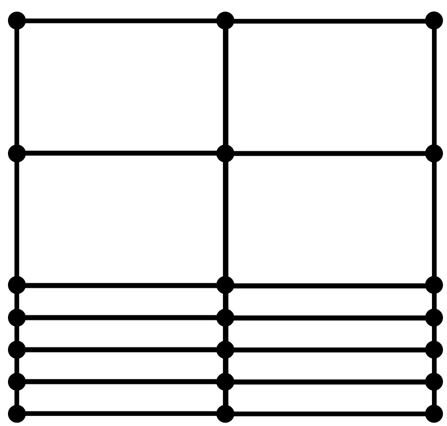
- Mesh aspect ratio AR is the ratio between the longest side Δx and the shortest side Δy .
- Large AR are ok if gradients in the largest direction are small.
- High AR smear gradients.



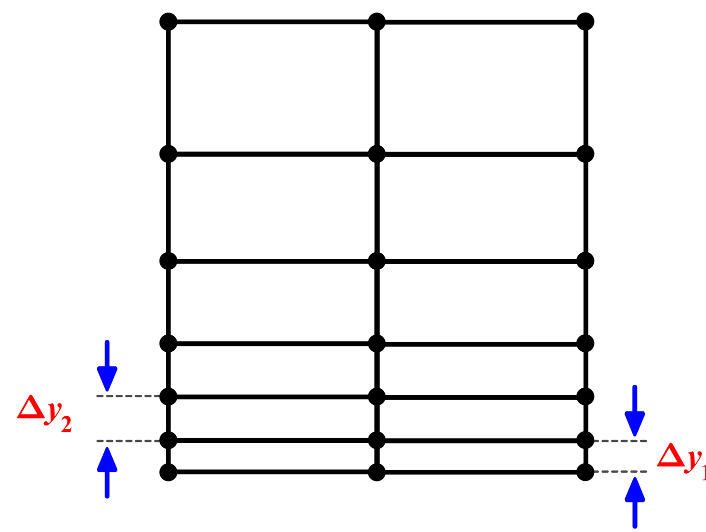
Mesh quality metrics. Smoothness

- 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%:

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



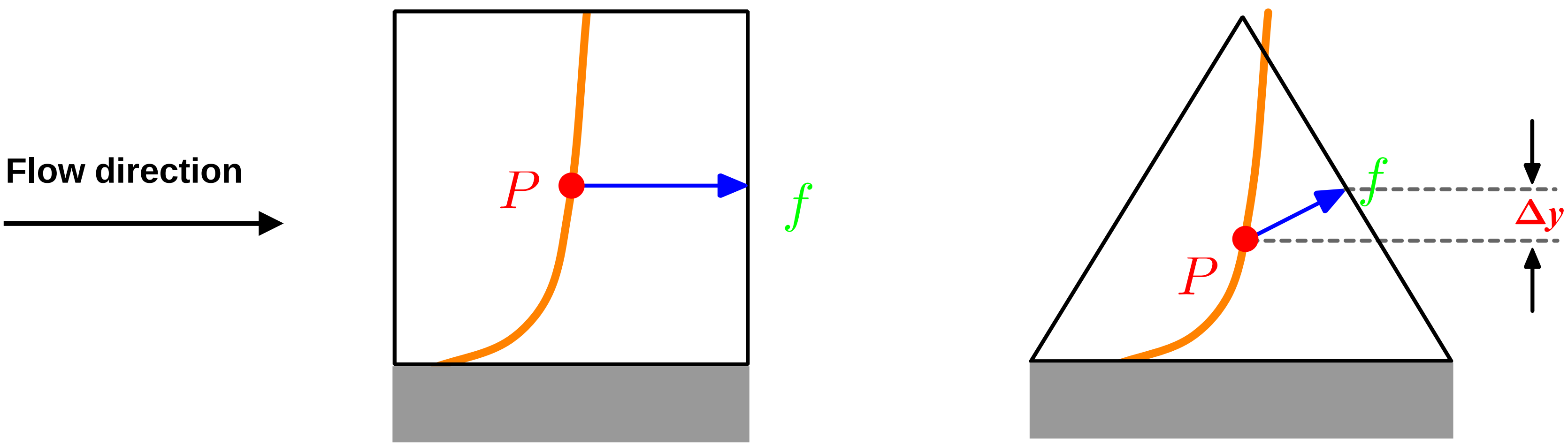
Steep transition



Smooth transition

Element type close to the walls - Cell/Flow alignment

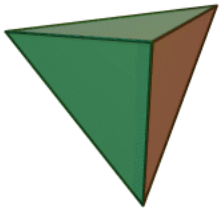
- Hexes, prisms, and quadrilaterals can be stretched easily to resolve boundary layers without losing quality.
- Triangular and tetrahedral meshes have inherently larger truncation error.
- Less truncation error when faces aligned with flow direction and gradients.



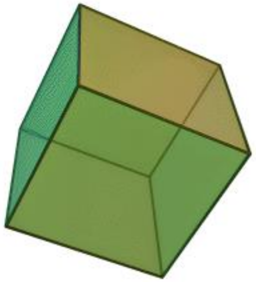
$$\phi_f = \phi_P + \cancel{\frac{\partial \phi}{\partial y}} \Delta y + \cancel{\mathcal{O}(\Delta y^2)}$$

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

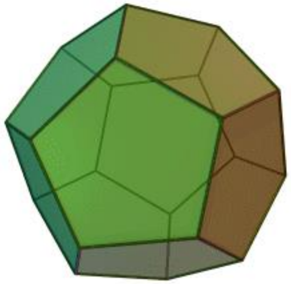
What cell type do I use?



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



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

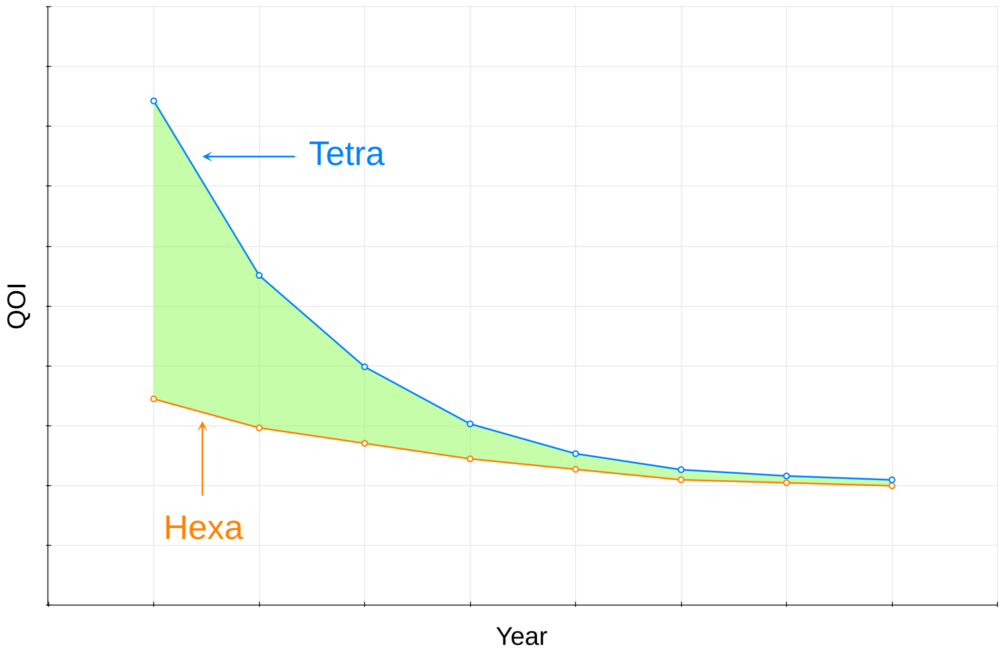


http://www.wolfdynamics.com/wiki/cells/ani_poly.gif

- Each cell type has its very own properties when it comes to approximating the gradients and interpolating the fluxes.
- Generally speaking, hexahedrons meshes will give more accurate solutions under certain conditions.
- But for complex flows without dominant flow direction, quad and hex meshes loose their advantages.
- Polyhedral cells approximates better the gradients, but skewness can be a problem on these cells. The more faces polyhedral cells have, the more likely your solution will become oscillatory due to skewness.
- Also, it is quite difficult to control the growth rate and volumetric refinement on polyhedral cells.
- Among all cell types tetrahedron have the minimum number of faces, so gradient are less accurate. However, they can be easily adapted to any kind of geometry.
- On tetra meshes, the growth rate can be controlled relatively easy and they can be easily adapted using volumetric refinement with conforming cells (cells with faces that share only two neighbors, so there is no need to split the fluxes across the faces).
- 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.

Striving for quality

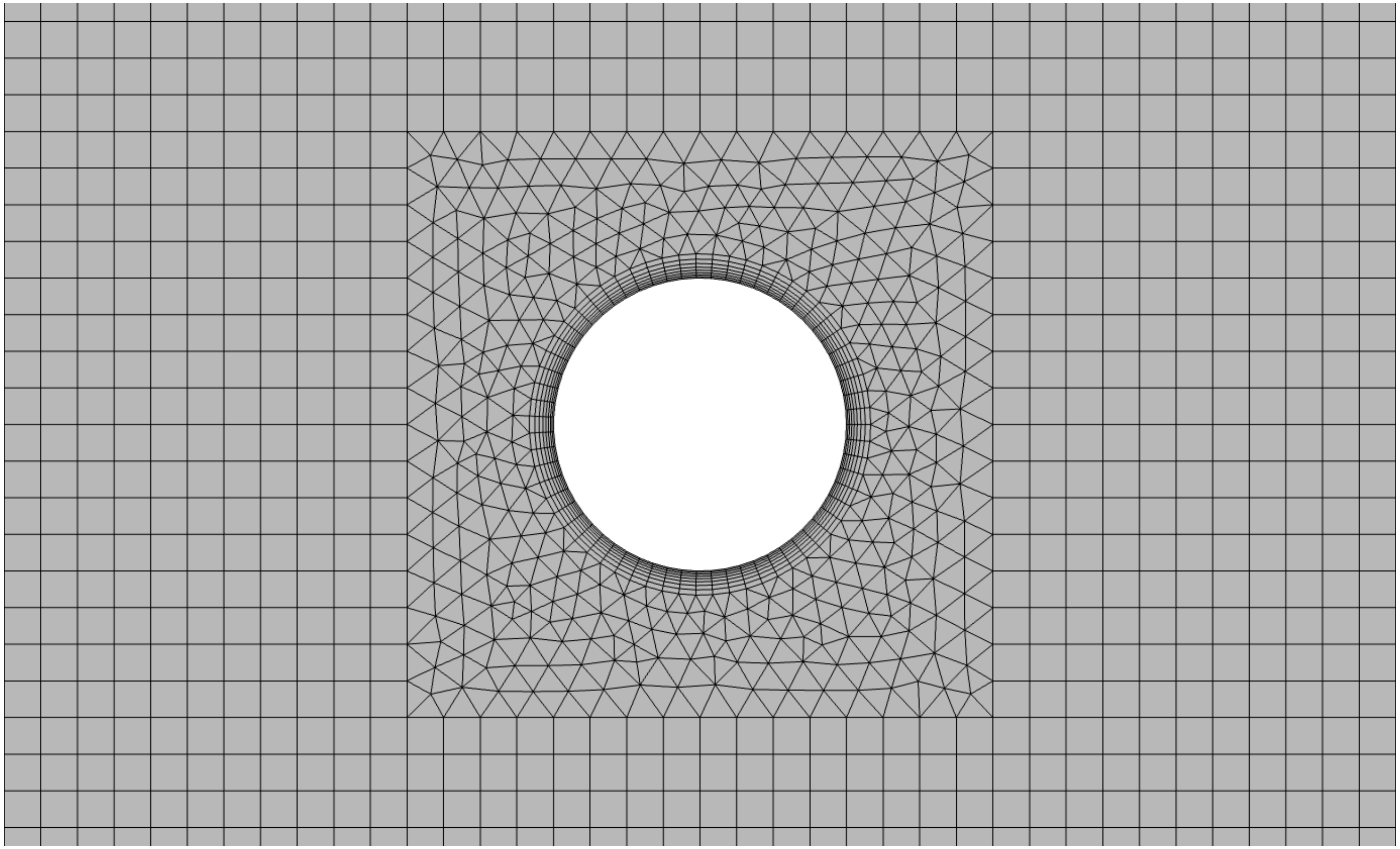
- For the same cell count, hexahedral meshes will give more accurate solutions, especially if the grid lines are aligned with the flow.
- But this does not mean that tetrahedral meshes are not good, by carefully choosing the numerical scheme you can get the same level of accuracy as in hexahedral meshes.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

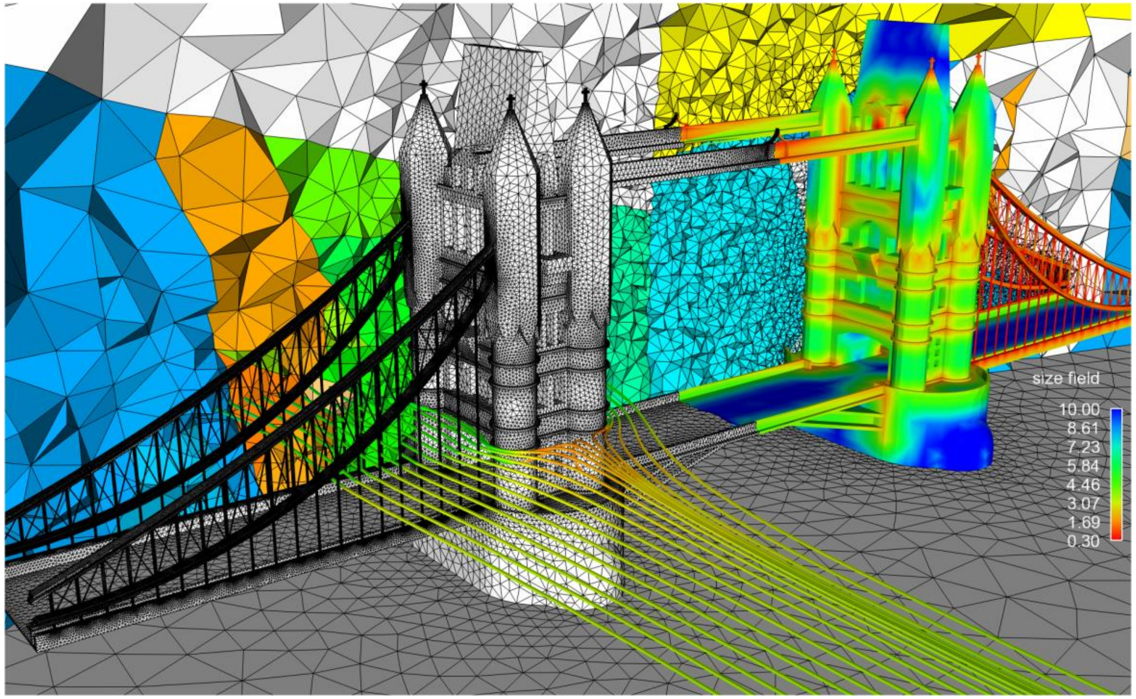
Striving for quality

- And by the way, you can combine all cell types to get a hybrid 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

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.
- Triangular and tetrahedral meshes have inherently larger truncation error.
- 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.
- But attention, a finer mesh does not mean a good or better mesh.
- To keep 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 adjacent 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.

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.
- Plan your meshing approach.
- 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.

Mesh quality assessment in CFD

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. Guerrero – WD

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 used 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;
```

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.

Checking mesh quality in OpenFOAM®

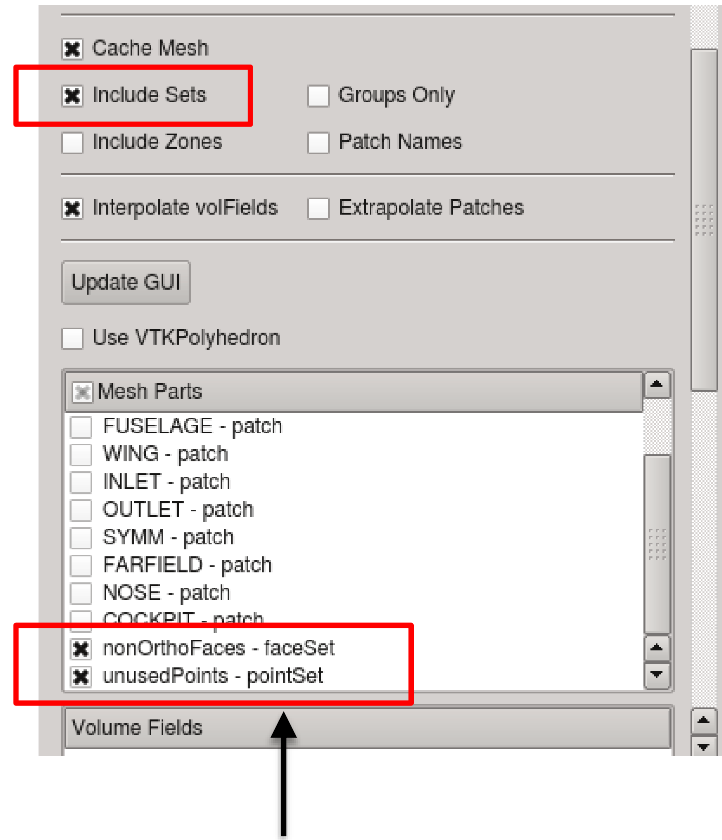
- To check the mesh quality and validity, OpenFOAM® comes with the utility `checkMesh`.
- To use this utility, just type in the terminal `checkMesh`, and read the screen output.
- `checkMesh` will look for/check for:
    - Mesh stats and overall number of cells of each type.
    - Check topology (boundary 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/`**

Checking mesh quality in OpenFOAM®

- 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` or you can use the utility `foamToVTK`.
- The utility `foamToVTK` converts the failed sets to VTK format.

Visualizing the failed sets in OpenFOAM®

- To visualize the failed sets directly within `paraFoam` you can proceed as follows.
  - Use the utility `checkMesh` to check the mesh quality.
  - If there are problems in the mesh, `checkMesh` will automatically save the sets in the directory **`constant/polyMesh/sets`**
  - The following are a few of the possible faulty sets `checkMesh` can detect: highAspectRatioCells, nonOrthoFaces, wrongOrientedFaces, skewFaces, unusedPoints.
  - In `paraFoam`, simply select the option **Include Sets** and then select the sets you want to visualize.
  - Just to be clear, this method only works with `paraFoam`. It does not work in paraview.



Failed sets

Roadmap

1. ~~Mesh quality assessment in CFD~~
2. **Mesh generation using cfMesh**
3. The cylinder tutorial
4. The 2D airfoil tutorial
5. The static mixer tutorial
6. The Ahmed body tutorial
7. The mixing elbow comparison
8. The moving quadcopter tutorial

Mesh generation using cfMesh

- cfMesh (now at version **1.1.2**) is a library for automatic mesh generation built on top of OpenFOAM®.
- Two versions of cfMesh are available, an opensource and a commercial version (cfMeshPRO).
- Both versions have the same capabilities when it comes to mesh generation using the command line interface or CLI.
- cfMeshPRO has some additional features like a graphical user interface, automatic refinements, advanced boundary layer options, topology controls, and others.
- For more information on cfMesh, please refer to the official website:

http://cfmesh.com/

Mesh generation using cfMesh

- cfMesh supports both 3D and 2D meshes.
- cfMesh comes with the following meshers or meshing algorithms:
    - `cartesianMesh`: generates hex dominant 3D meshes
    - `cartesian2DMesh`: generates quad dominant 2D meshes
    - `tetMesh`: generates tetra dominant 3D meshes
    - `pMesh`: generates polyhedral dominant 3D meshes
- By default cfMesh runs in parallel using all threads available in the system.
- Contrary to snappyHexMesh, there is no need to decompose the domain before meshing.
- If you need to limit the amount of cores to use, you can set the following environment variable:

```
$> export OMP_NUM_THREADS=2
```

cfMesh workflow and input dictionary

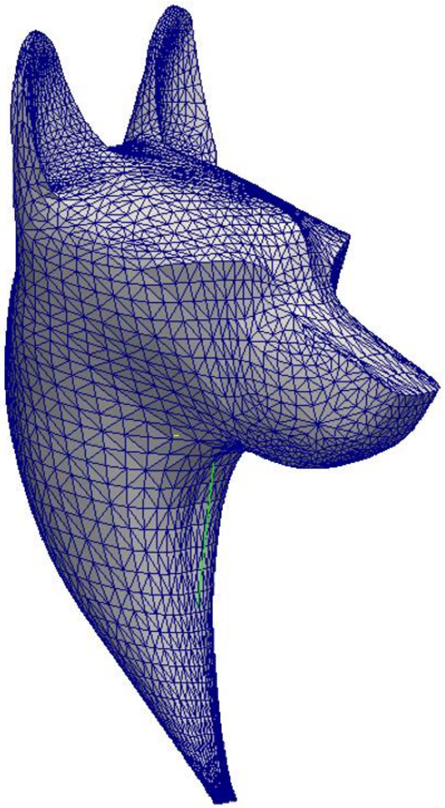
- The meshing algorithm starts the meshing process by creating a so-called mesh template from the input geometry and the user-specified settings.
- The template is later on adjusted to match the input geometry. The process of fitting the template to the input geometry is designed to be tolerant to poor quality input data, which does not need to be watertight.
- However, a good quality input geometry model is always recommended to obtain an optimal body-fitted surface mesh.
- cfMesh uses one single mesh configuration file, the *`meshDict`* dictionary.
- This mesh configuration file is located in the **`system`** directory.

Mesh generation using cfMesh

- If you are already familiar with snappyHexMesh, you will find the following similarities/differences between both meshers:

Similarities	Differences
• Text input files (dictionaries). • Geometry is provided as a STL file. • Global and local parameters to control mesh refinement. • Lines, surfaces, and volumes refinement. • Boundary layer meshing. • The quality of the mesh is check using checkMesh.	• No need to generate a background mesh • The input STL file must contain the enclosure of the domain. • Meshing is done in one single step. • No need to use surfaceFeatureExtract. • Boundary layer meshing is very reliable. • It is super fast.

Mesh generation using cfMesh



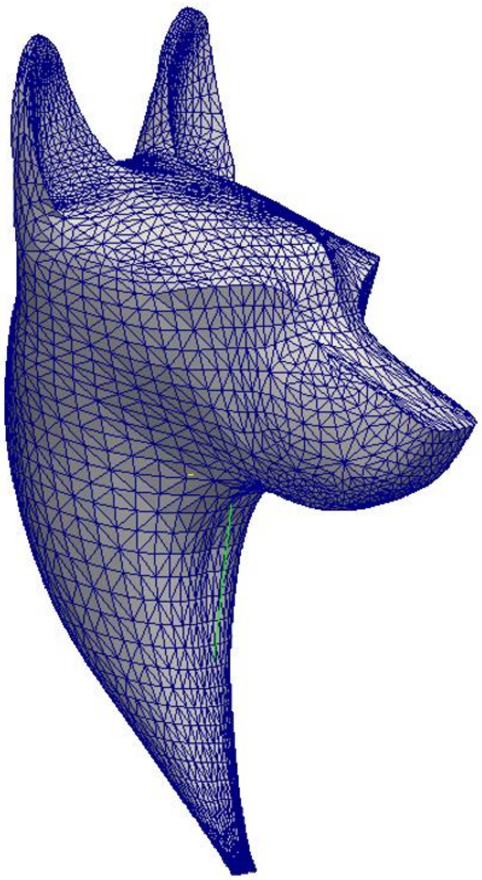
- Let us explain cfMesh workflow by meshing this geometry.
- The objective is to mesh a rectangular region surrounding an object described by a STL surface.

Mesh generation using cfMesh

- The cfMesh input file *meshDict* is located in the directory **system**,

```
surfaceFile "…";
maxCellSize …;
boundaryCellSize …;

objectRefinements{
…
}

localRefinement{
…
}

surfaceMeshRefinement{
…
}

boundaryLayers{
…
}

renameBoundary{
…
}
```

Location of the input geometry file

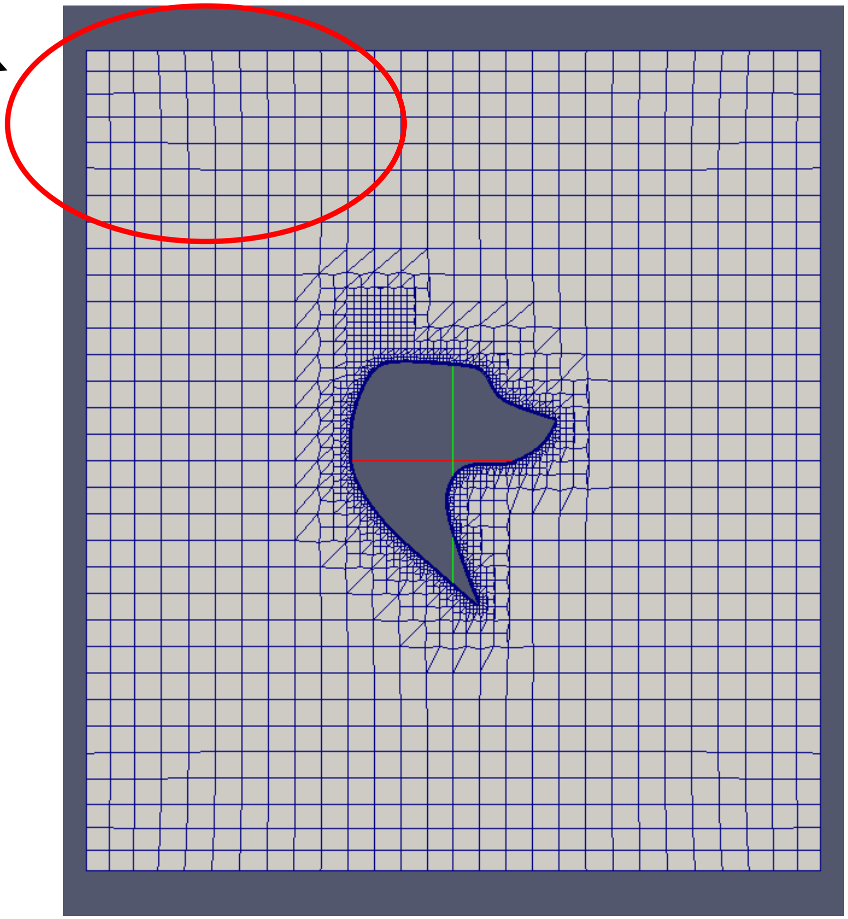


Mesh generation using cfMesh

- The cfMesh input file *meshDict* is located in the directory **system**,

```
surfaceFile "...";
maxCellSize ...;
boundaryCellSize ...;

objectRefinements{
...
}

localRefinement{
...
}

surfaceMeshRefinement{
...
}

boundaryLayers{
...
}

renameBoundary{
...
}
```

Global mesh refinement options

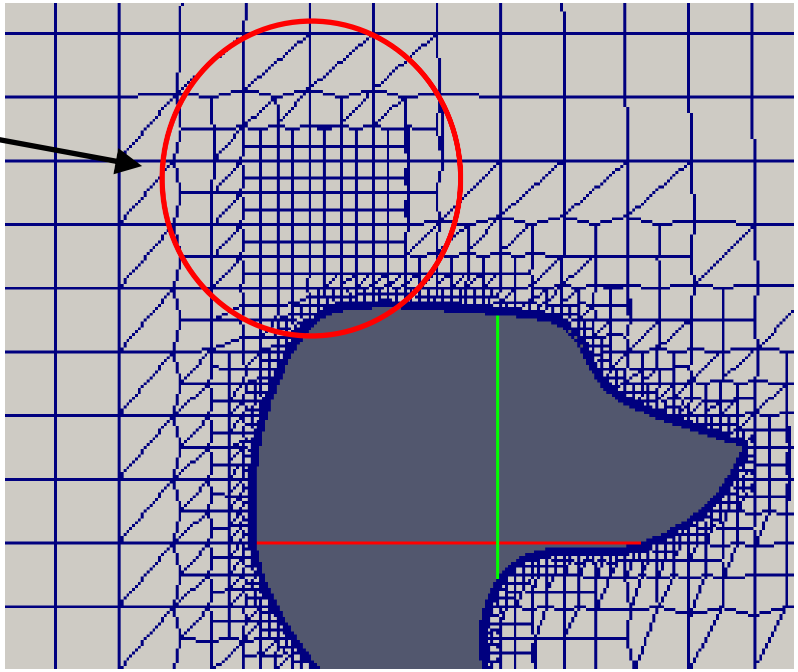


Mesh generation using cfMesh

- The cfMesh input file *meshDict* is located in the directory **system**,

```
surfaceFile "…";
maxCellSize …;
boundaryCellSize …;

objectRefinements{
…
}

localRefinement{
…
}

surfaceMeshRefinement{
…
}

boundaryLayers{
…
}

renameBoundary{
…
}
```

Set the refinement level of given mesh zones

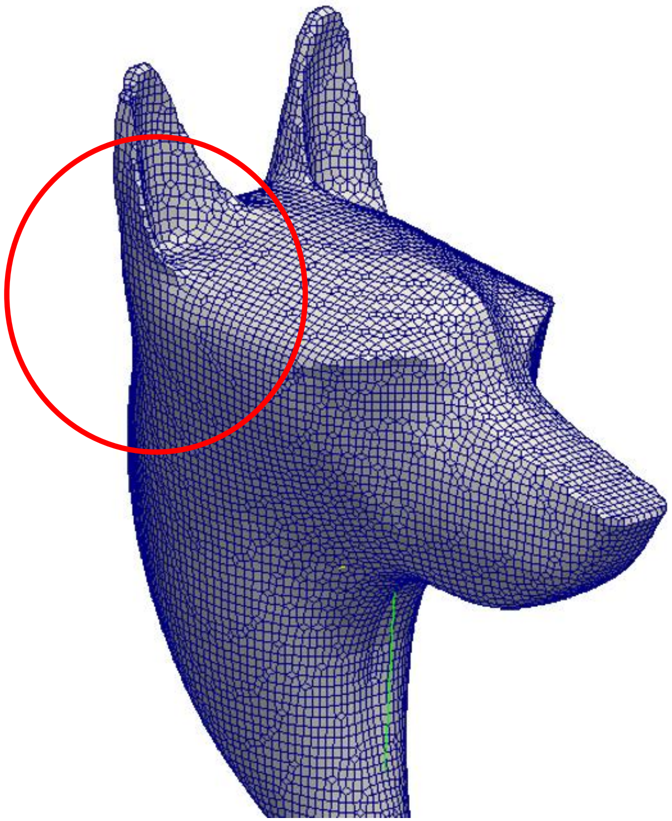


Mesh generation using cfMesh

- The cfMesh input file *meshDict* is located in the directory **system**,

```
surfaceFile "…";
maxCellSize …;
boundaryCellSize …;

objectRefinements{
…
}

localRefinement{
…
}

surfaceMeshRefinement{
…
}

boundaryLayers{
…
}

renameBoundary{
…
}
```

Set the refinement of a patch as defined by the geometry file



Mesh generation using cfMesh

- The cfMesh input file *meshDict* is located in the directory **system**,

```
surfaceFile "…";
maxCellSize …;
boundaryCellSize …;

objectRefinements{
…
}

localRefinement{
…
}

surfaceMeshRefinement{
…
}

boundaryLayers{
…
}

renameBoundary{
…
}
```

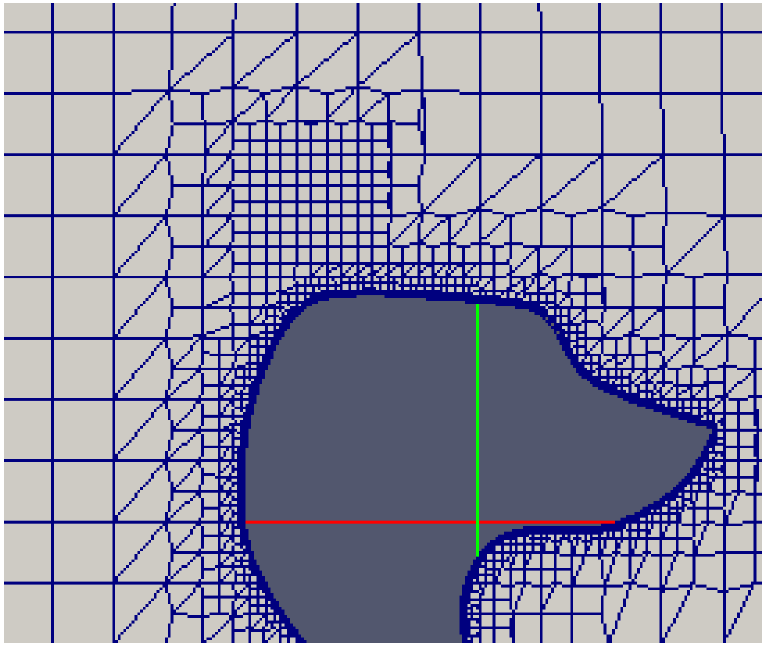
Allows the use of surface meshes provided by additional surface files as refinement zones in the mesh

Mesh generation using cfMesh

- The cfMesh input file *meshDict* is located in the directory **system**,

```
surfaceFile "…";
maxCellSize …;
boundaryCellSize …;

objectRefinements{
…
}

localRefinement{
…
}

surfaceMeshRefinement{
…
}

boundaryLayers{
…
}

renameBoundary{
…
}
```

Set the boundary layers refinement



Mesh generation using cfMesh

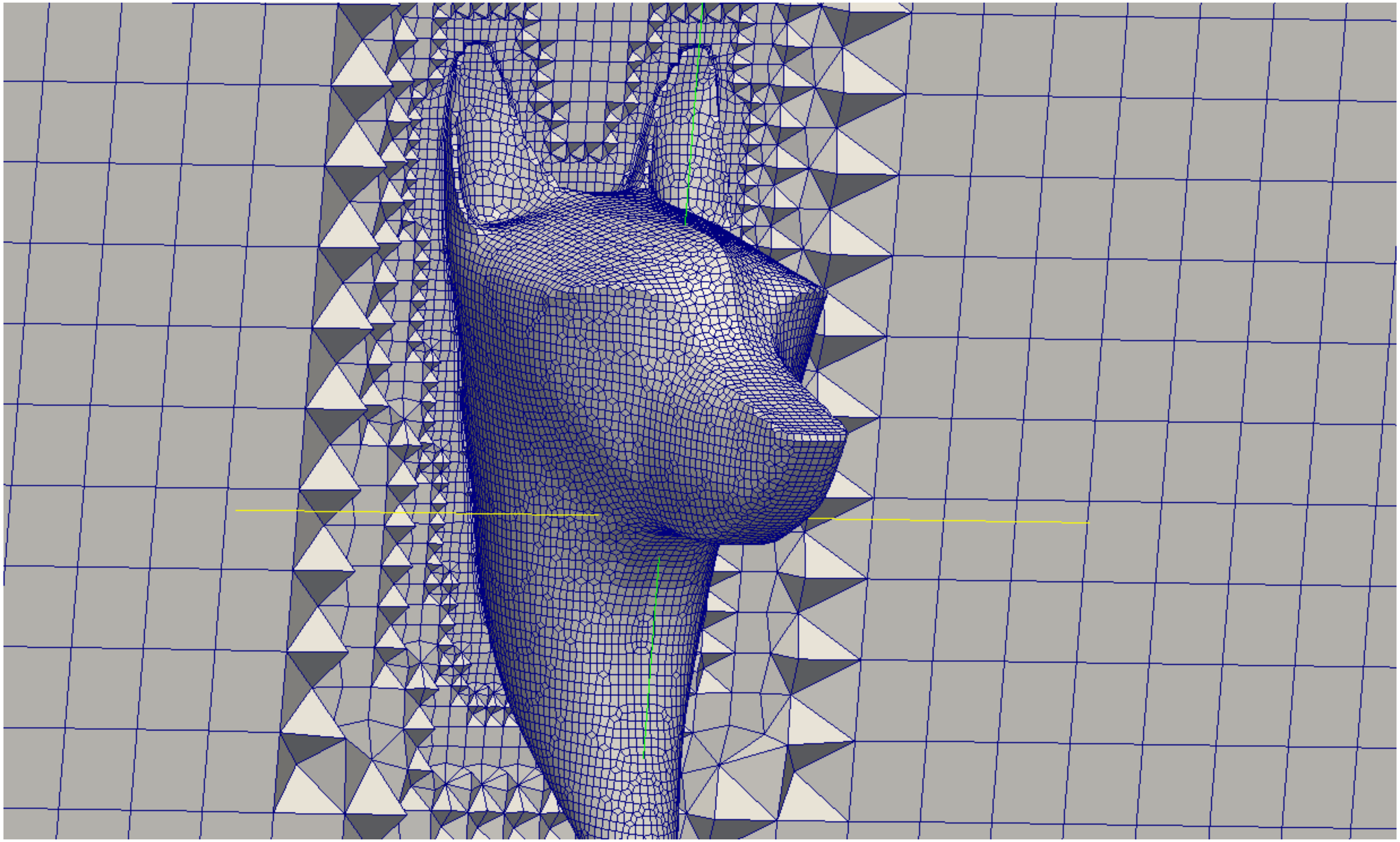
- The cfMesh input file *meshDict* is located in the directory **system**,

```
surfaceFile "…";
maxCellSize …;
boundaryCellSize …;

objectRefinements{
…
}

localRefinement{
…
}

surfaceMeshRefinement{
…
}

boundaryLayers{
…
}

renameBoundary{
…
}
```

Rename the patches and specify their Type according to the user needs.

Mesh generation using cfMesh

- With a fairly simple input dictionary you can easily obtain high quality meshes.
- You will find this example in the directory `cfMesh/tutorials/CF_extra1_wolf`



Mesh generation using cfMesh

- The minimum information required in the input dictionary *system/meshDict* is:
  - **surfaceFile**: the location of the input geometry
  - **maxCellSize**: the maximum cell size
- The volume that will be meshed is the volume enclosed in the supplied geometry.
- The preferred surface format of cfMesh is fms. This proprietary format stores patches, subsets, and feature edges in a single file.
- The fms surface file is typically generated from a STL file using the following cfMesh utility:

```
$> surfaceFeatereEdsges <.stl file>  <.fms file>
```

which also detects the geometry features for edges refinement.

- You can inspect the feature edges by using the following cfMesh utility:

```
$> FMSToSurface <input fms> <surface file> -exportFeatureEdges
```

where the option -exportFeatureEdges writes feature edges to a vtk file.

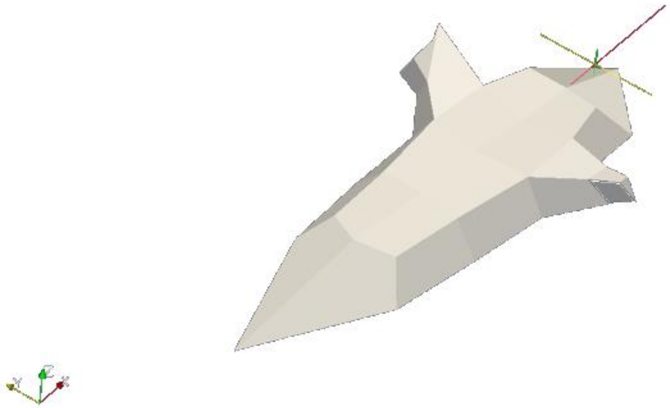
Mesh generation using cfMesh

- In the input dictionary, the following keywords control:
  - **boundaryCellSize**: specifies the size of all boundary cells (global option).
  - **boundaryCellSizeRefinementThickness**: specifies at which distance the **boundaryCellSize** option is still active (global option).
  - **minCellSize**: it sets the minimum cell size.
  - **localRefinement**: it overwrites the **boundaryCellSize** option for a particular patch (named as in the input surface). Can be specified thought the **cellSize** or the additionalRefinementLevels **keywords**.

```
localRefinement
{
    solid
    {
          //cellSize 1;
          additionalRefinementLevels 4;
    }
}
```

Mesh generation using cfMesh

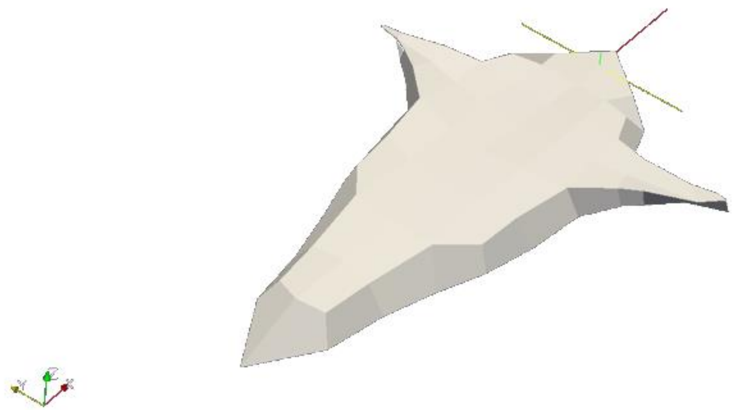
additionalRefinementLevels 0



Number of cells	Maximum non-orthogonality	Maximum skewness
4348	62	6.0

Mesh generation using cfMesh

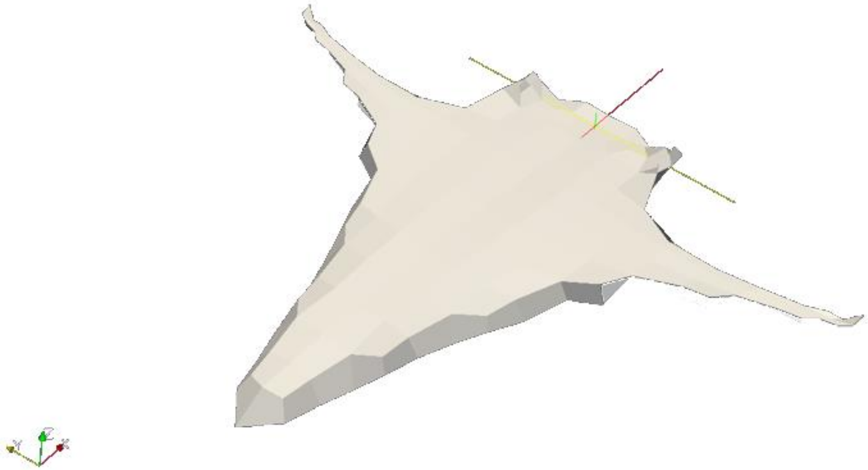
additionalRefinementLevels 1



Number of cells	Maximum non-orthogonality	Maximum skewness
4714	62	4.7

Mesh generation using cfMesh

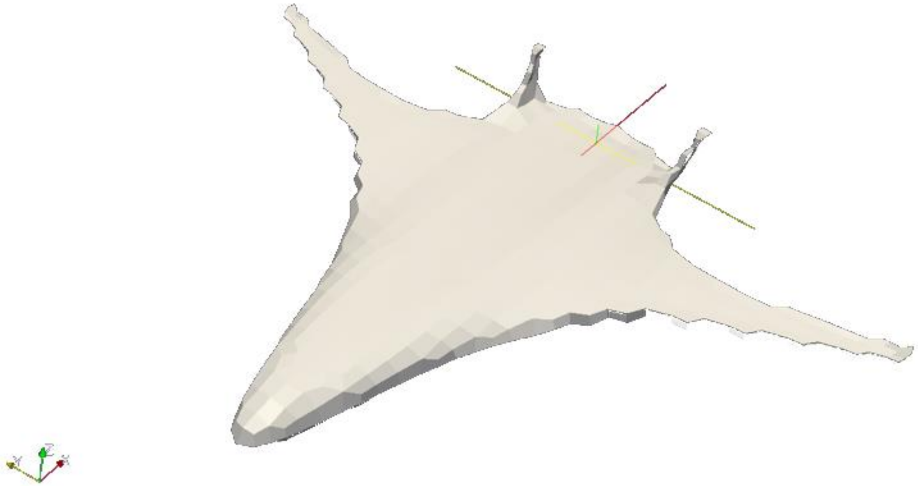
additionalRefinementLevels 2



Number of cells	Maximum non-orthogonality	Maximum skewness
5818	45	4.1

Mesh generation using cfMesh

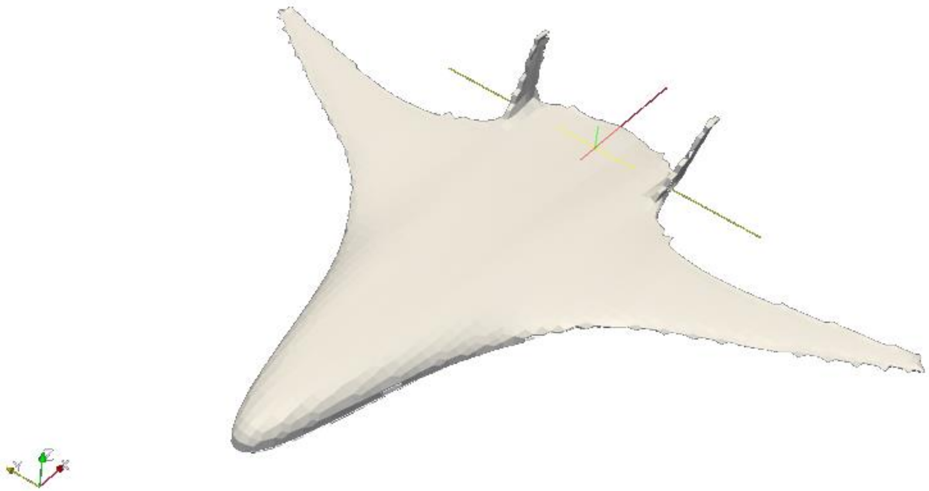
additionalRefinementLevels 3



Number of cells	Maximum non-orthogonality	Maximum skewness
9262	43	3.3

Mesh generation using cfMesh

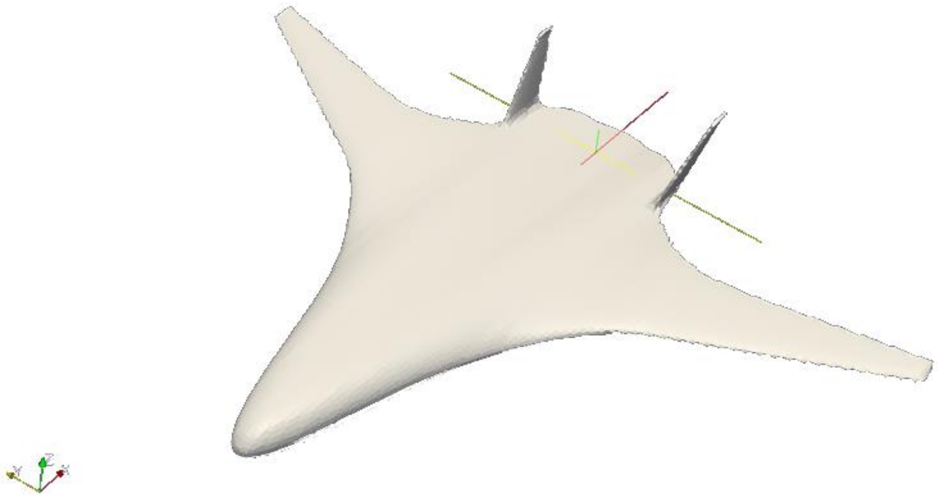
additionalRefinementLevels 4



Number of cells	Maximum non-orthogonality	Maximum skewness
20298	52	3.4

Mesh generation using cfMesh

additionalRefinementLevels 5

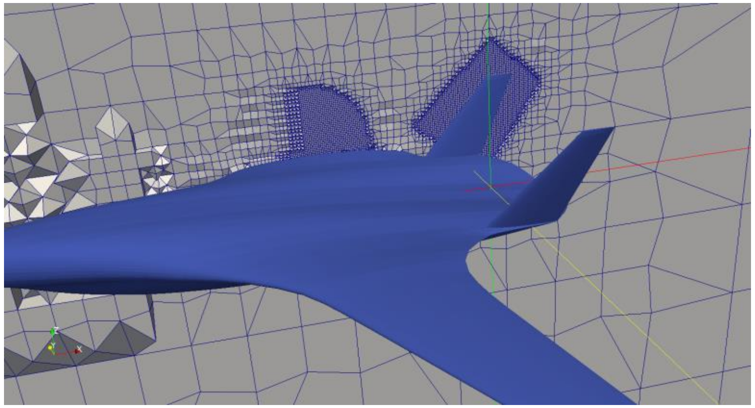


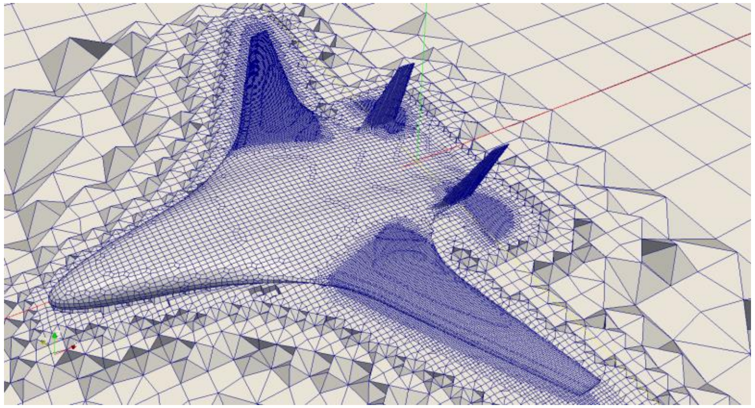
Number of cells	Maximum non-orthogonality	Maximum skewness
51342	52	2.9

Mesh generation using cfMesh

- In the input dictionary, the following keywords control:
  - **objectRefinement**: it specifies refinement zones inside the volume (lines, spheres, boxes, and truncated cones). No support for STL so far.

```
left_wing
{
        type cone;
        p0 (27.5 -10 2);
        p1 (37.5 -37.5 2);
        radius0 5;
        radius1 2;
        cellSize 0.5;
        refinemntThickness 1;
}
```





Mesh generation using cfMesh

- In the input dictionary, the following keywords control:
  - **keepCellsIntersectingBoundary**: global option to keep cells in the template mesh which are intersected by the boundary (default value is false).
  - **keepCellsIntersectingPatches**: local option that overwrites **keepCellsIntersectingBoundary** on specified patches.
  - **removeCellsIntersectingBoundary**: local option that overwrites **keepCellsIntersectingBoundary** on specified patches.
  - **boundaryLayers**: controls the boundary layer parameters in all patches
    - **nLayers**: global option that controls the number of layers which will be grow from all the patches.
    - **thicknessRatio**: global option that controls the growth rate of the inflation layer.
    - **maxFirstLayerThickness**: global option that controls the thickness of the first layer.

```
boundaryLayers
{
        nLayers     3;
        thicknessRatio  1.2;
        maxFirstLayerThickness 0.5;
        allowDiscontinuity 1;
}
```

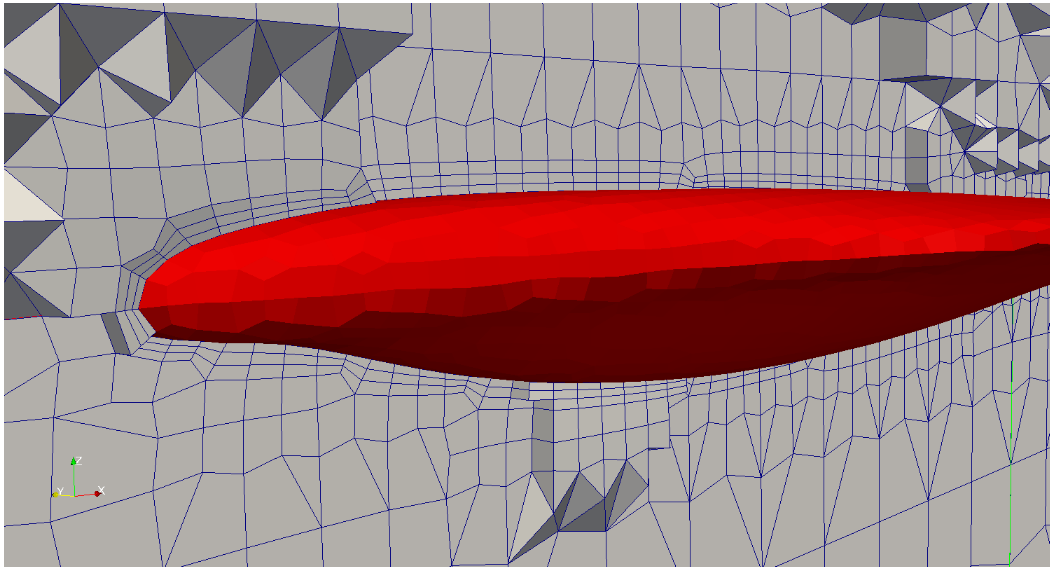
Mesh generation using cfMesh

- In the input dictionary, the following keywords control:
  - **patchBoundaryLayers**: local option that specifies the local properties of the boundary layer for individual patches according to the names given in the input file. The keyword **allowDiscontinuity**, ensures that the number of layers required in a patch does not spread to the other patches in the same layer (1 is on and 0 is off).

```
patchBoundaryLayers
{
        sur2
        {
                nLayers        4;
                thicknessRatio        1.2;
                maxFirstLayerThickness 5;
                allowDiscontinuity 1;
        }

        sur3
        {
                nLayers        2;
                thicknessRatio        1.2;
                maxFirstLayerThickness 5;
        allowDiscontinuity 1;
        }
}
```

Mesh generation using cfMesh

Boundary layer meshing



nLayers	thicknessRatio	maxFirstLayerThickness
3	1.2	0.5

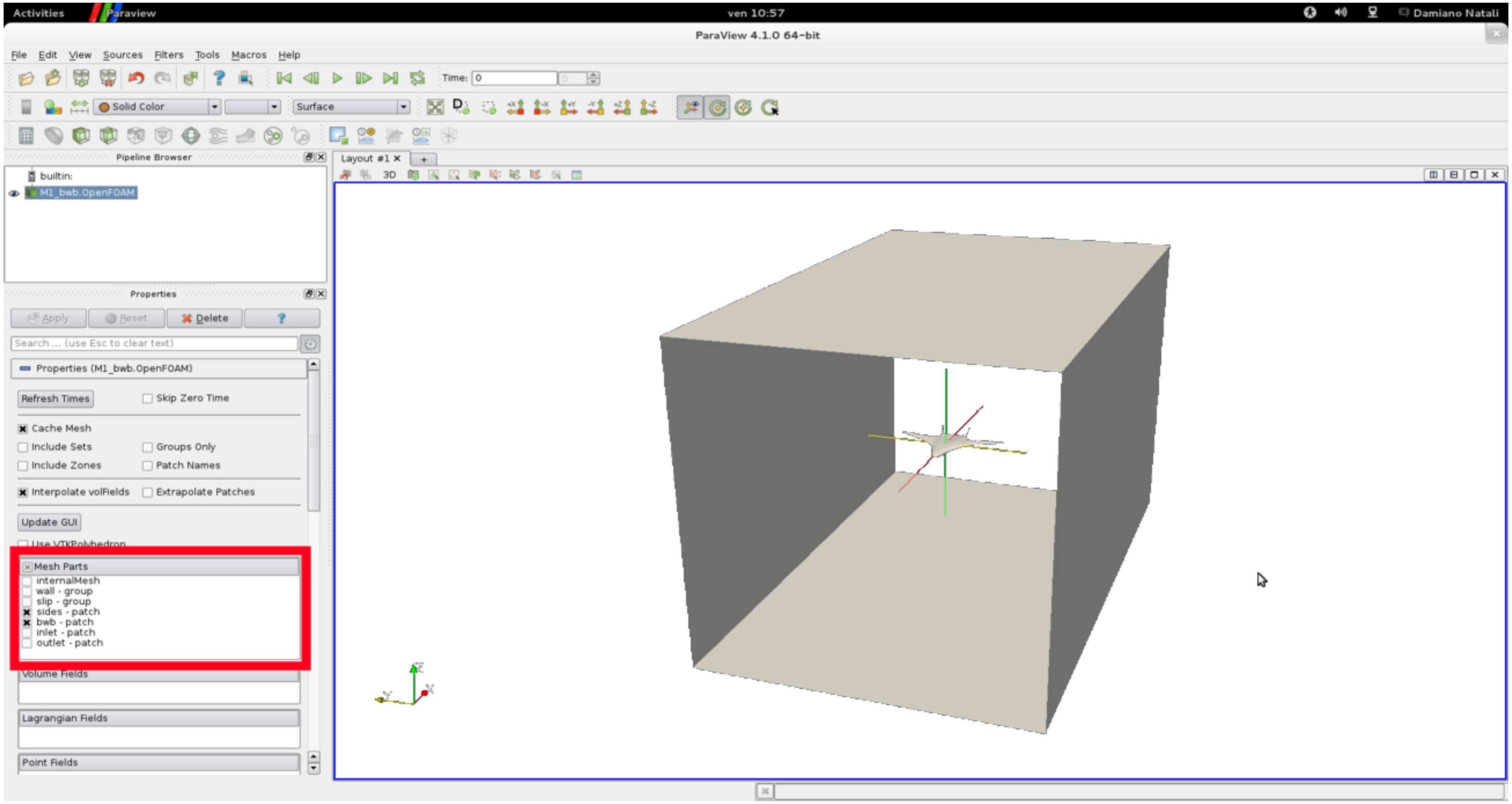
Mesh generation using cfMesh

- In the input dictionary, the following keywords control:
  - **renameBoundary**: this option overwrite both the name and the type of the patches in the file *constant/polyMesh/boundary*

```
renameBoundary
{
        defaultName myWalls;
        defaultType wall;

        newPatchNames
        {
                "sur.*"
                {
                        newName sides;
                        type slip;
                }
        }
}
```

Mesh generation using cfMesh

Boundary patches names



Mesh generation using cfMesh

- The two previous cases are located in the following directories:
  - **$TM/CFMESH/c_extra1_wolf**
  - **$TM/CFMESH/c_extra2_bwb**

- To run the cases go the case directory and type in the terminal:

```
1. $> foamCleanTutorials
2. $> cp -rp system/meshDict.org system/meshDict
3. $> cartesianMesh
4. $> checkMesh
5. $> paraFoam
```

Mesh generation using cfMesh

- The standard installation of cfMesh contains several complementary utilities to perform some geometry and mesh manipulation operations. It is worth mentioning the following:

    - **`checkSurfaceMesh:`** performs basic topology and geometric checks on the input surface mesh. It reports potential problems that could affect the quality of the mesh.

    - **`FMSToSurface:`** this utility converts the data in a fms file into several files which can be imported into ParaView.

    - **`FMSToVTK:`** converts a fms file into vtk format.

    - **`improveMeshQuality:`** it applies a smoother to the mesh in order to improve the overall quality. The number of iterations is controllable via optional parameters.

Mesh generation using cfMesh

- The standard installation of cfMesh contains several complementary utilities to perform some geometry and mesh manipulation operations. It is worth mentioning the following:

  - `mergeSurfacePatches:` this utility allow the user to specify the patches in the surface mesh which shall be merge together.

  - `scaleMesh:` it scales the mesh by a given factor.

  - `surfaceToFMS:` it converts a common surface triangulation (STL) into fms format.

  - `surfaceFeatureEdges:` it is used for extracting feature edges (sharp angles). If the output is a **fms** file, the extracted edges are stored as feature edges. Otherwise, it generates patches bounded by the selected feature edges.

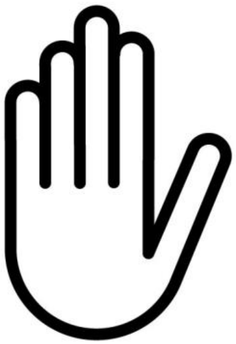
Roadmap

1. ~~Mesh quality assessment in CFD~~
2. ~~Mesh generation using cfMesh~~
3. **The cylinder tutorial**
4. The 2D airfoil tutorial
5. The static mixer tutorial
6. The Ahmed body tutorial
7. The mixing elbow comparison
8. The moving quadcopter tutorial

Cylinder tutorial

- Meshing with cfMesh.
- Meshing tutorial 1. The 3D Cylinder (external mesh).

```
$TM/CFMESH/c1_cyl/
```



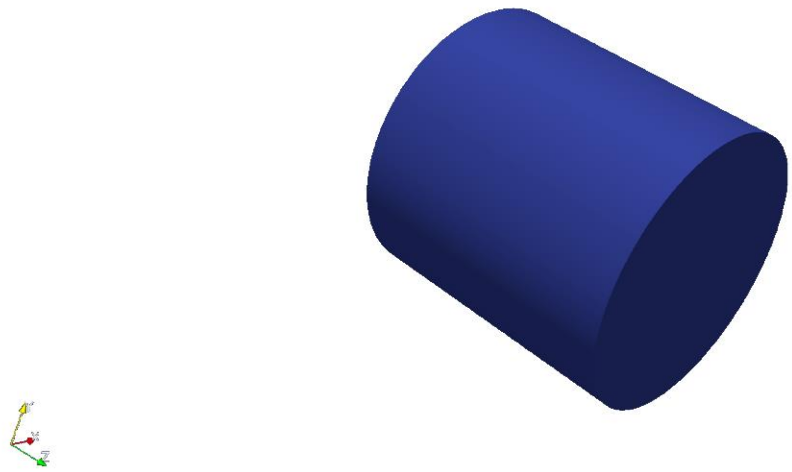
- From this point on, please follow me.
- We are all going to work at the same pace.
- Remember, `$TM` is pointing to the path where you unpacked the tutorials.

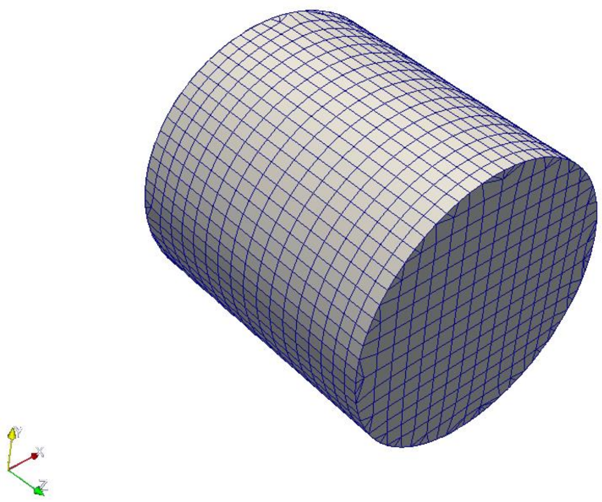
Cylinder tutorial

- The `geo/cylinder.stl` file to be used contains the original cylinder STL that can be visualized with Paraview

- The STL file is composed by one single surface for the whole cylinder geometry.

- This is also recognizable in Paraview that, by default, colors the cylinder with a single color (blue in this case).





Cylinder tutorial

- In order to create an external aero-dynamic mesh we need to define the computational bounding box.

- The geometry file provided in this tutorial does not contain the bounding box.

- Conversely to `blockMesh`, **cfMesh** dictionary does not include the definition of the bounding box patches in its dictionary.

- In the next slides, we will learn how to use complementary STL utilities included in OpenFOAM and **cfMesh** to perform the necessary geometry manipulations.

- As a first step, we have to consolidate our understanding of the STL data structure.

Cylinder tutorial

- An STL file (STereo Lithography interface format or Standard Triangulation Language) is a simple list of triangles called **facets** with an outward normal versor (needed to recognize what is inside and what is outside),

```
facet normal -1 0 0
        outer loop
                vertex -1.5 1.5 -1.5
                vertex -1.5 -1.5 -1.5
                vertex -1.5 -1.5 1.5
        endloop
endfacet
```

- The union of different facets form a **solid** that, after the mesh generation, will be associated with a patch.

Cylinder tutorial

```
solid solid1
        facet normal -1 0 0
                outer loop
                        vertex -1.5 1.5 -1.5
                        vertex -1.5 -1.5 -1.5
                        vertex -1.5 -1.5 1.5
                endloop
        endfacet
...
        facet normal -1 0 0
                outer loop
                        ...
                endloop
        endfacet
endsolid solid1

solid solid2
        facet normal 0 1 0
                outer loop
                        ...
                endloop
        endfacet
...
        facet normal 0 1 0
                outer loop
                        ...
                endloop
        endfacet
endsolid solid2
```

Facet 1

Facet N

Facet 1

Facet M

solid1

solid2

Cylinder tutorial

- An STL file can be stored in two different formats:
  - ASCII (human readable)
  - Binary format (machine format)

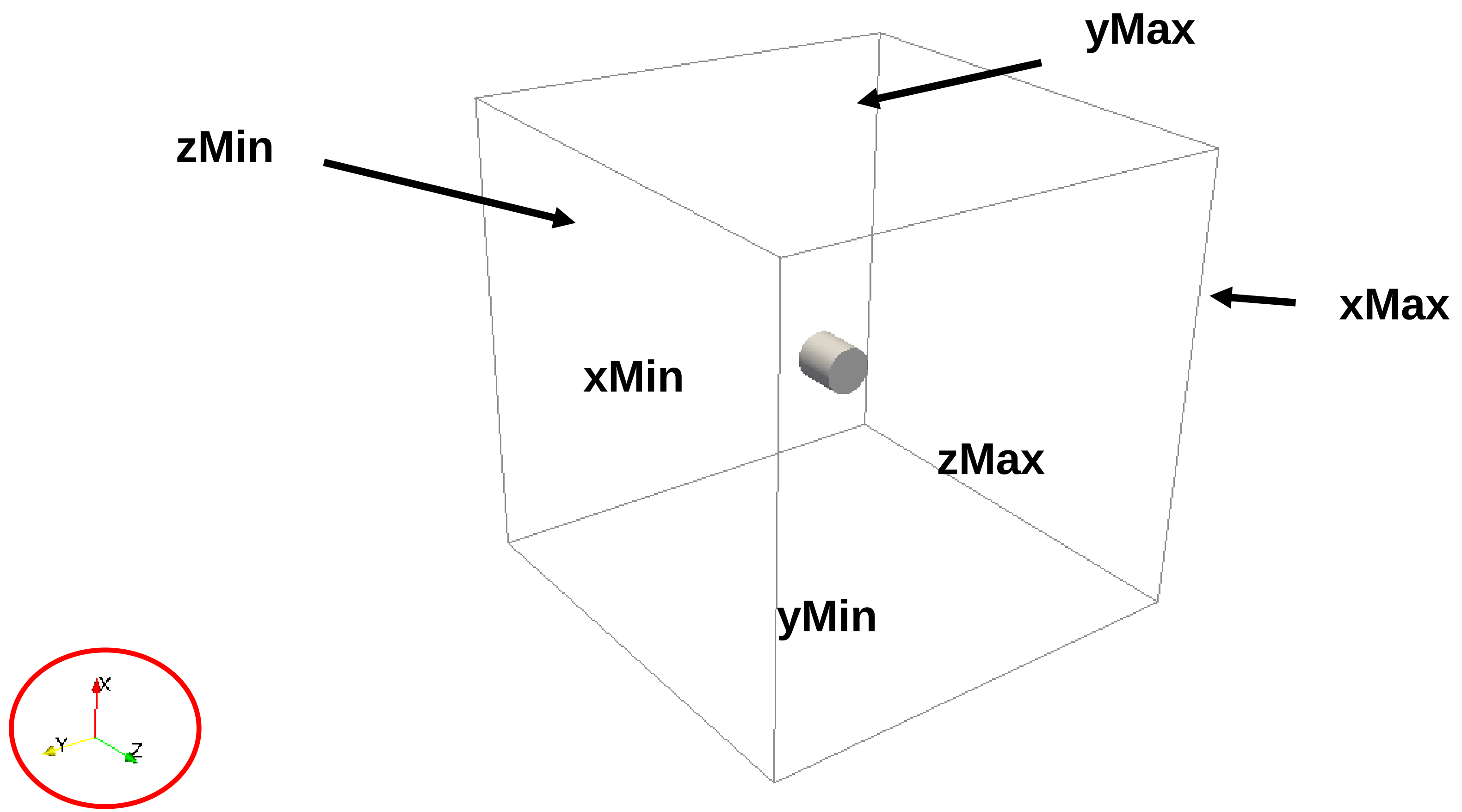
- Binary format is lighter (usually ~ 1/3 of hard drive space) but cannot be edited via a text editor.

- Now, let us define a proper bounding box by taking advantage of the `surfaceGenerateBoundingBox` cfMesh utility as follows:
  - `$> surfaceGenerateBoundingBox <input stl file> <output stl file> xNeg xPos yNeg yPos zNeg zPos`

- The `xNeg`, `xPos`, `yNeg`, `yPos`, `zNeg` and `zPos` arguments are the distances from the STL geometry surfaces and **must be expressed with non-negative values.**

Cylinder tutorial

- The output STL files now contains six new surfaces named with the following convention:



Cylinder tutorial

- We have seen how the new STL **solids** are automatically created and defined by means of the `surfaceGenerateBoundingBox` utility.

- The respective mesh patches can be easily renamed in the *`system/meshDict`* dictionary as in the following example …

```
renameBoundary
{
        newPatchNames
        {
                xMin
                {
                        newName inlet;
                        type patch;
                }
                xMax
                {
                        newName outlet;
                        type patch;
                                }

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

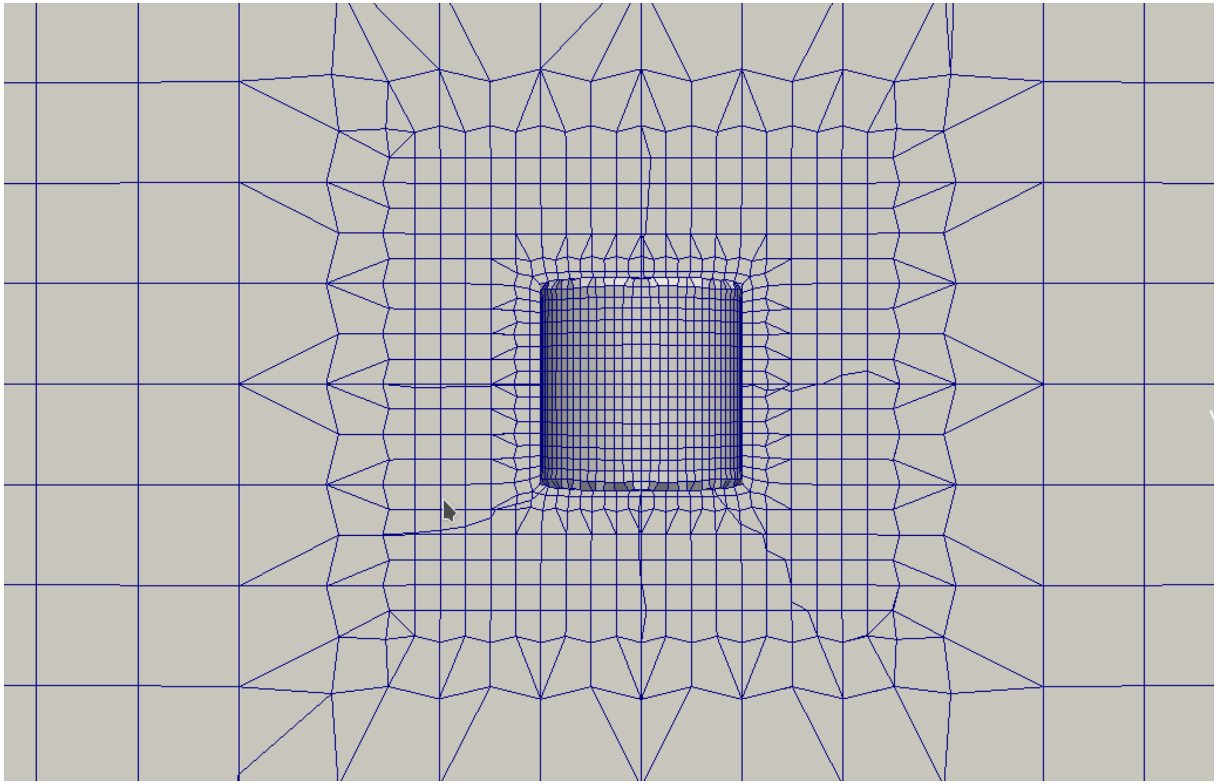
Cylinder tutorial

- Now we are ready to perform an initial mesh. The following commands can be provided in the terminal shell:

1. `$> foamCleanTutorials`
2. `$> cp -rp system/meshDict.org system/meshDict`
3. `$> surfaceGenerateBoundingBox geo/cylinder.stl constant/triSurface/boxCylinder.stl 9 9 9 9 9 9`
4. `$> ls -l geo/`
5. `$> cartesianMesh`
6. `$> checkMesh`
7. `$> paraFoam`

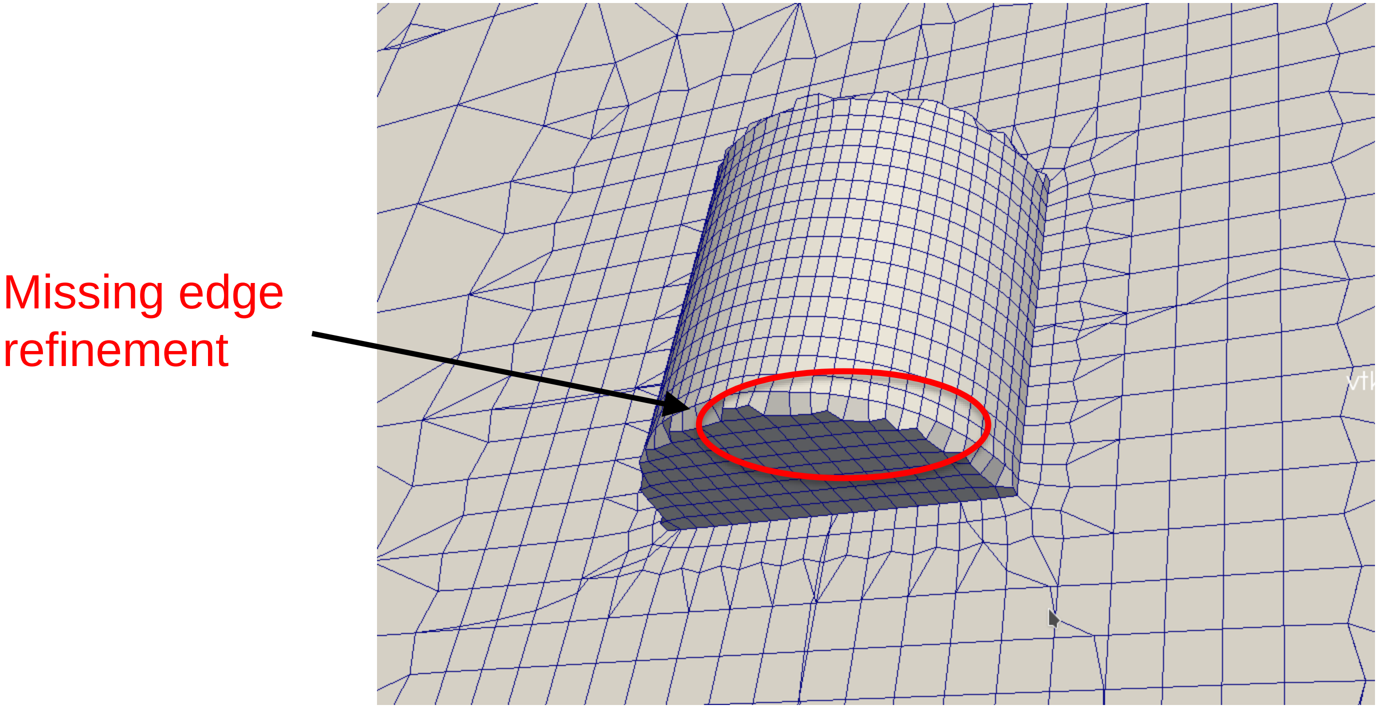
Cylinder tutorial

- To check the result of this initial mesh we can use Paraview and highlight the spatial discretization by means of the “Surface with Edge” view. The internal region of the mesh is usually inspected by using the Slice filter.



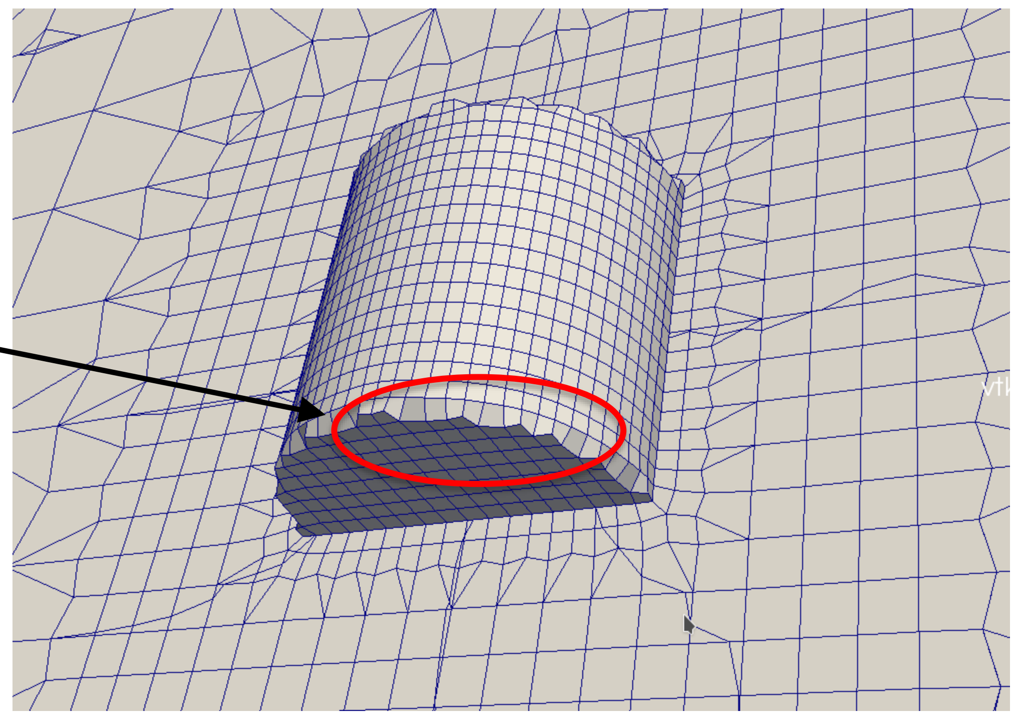
Cylinder tutorial

- The edges of the geometries must be treated accordingly when calculating the computational grid.
- Conversely to `blockMesh + snappyHexMesh`, **cfMesh** has no explicit edge refinement controls in its dictionary.
- That is, **when working with cfMesh we have to pay a lot of attention in preparing the STL geometry file accordingly to obtain optimal mesh refinements.**



Cylinder tutorial

- Our strategy is to split the original cylinder solid into different solids inside the STL file.
- This will improve the capabilities of cfMesh in identifying edges for refinement.
- Please remember that in the STL language the word solid means a group of surface triangles.



Missing edge refinement

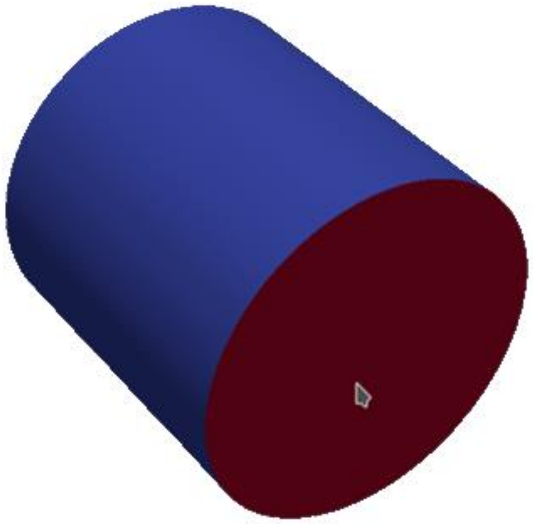
Cylinder tutorial

- With the aim of obtaining a new STL file composed by multiple **solids** (surfaces) we will use an additional **cfMesh** utility: `surfaceFeatureEdges`

- The utility needs the following arguments:

```
$> surfaceFeatureEdges <input> <output> -angle <sFE_angle>
```

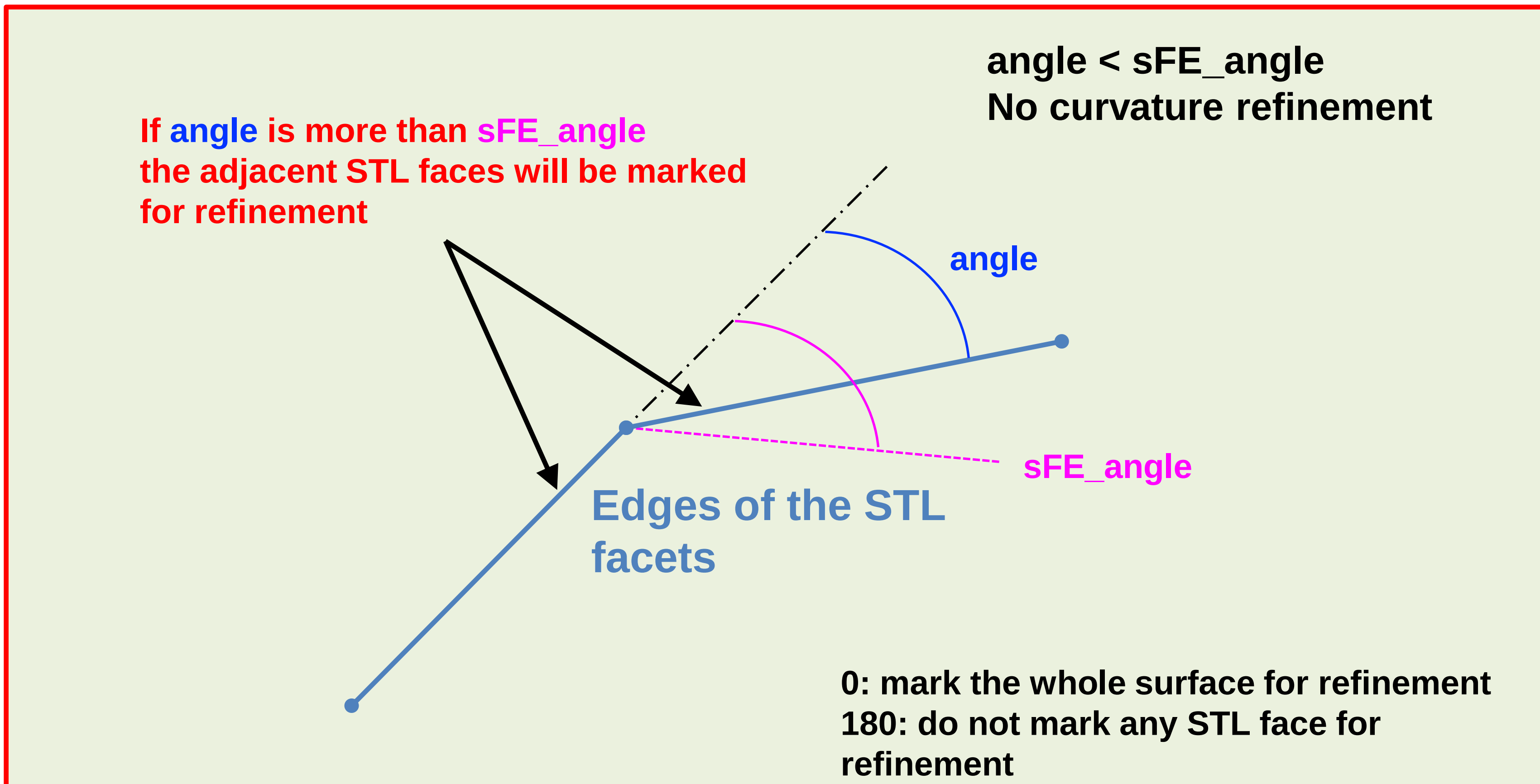
- The command `surfaceFeatureEdges` will read the `<input>` .STL file (in ASCII or binary form) and produce an ASCII `<output>` .STL (or .FMS) file in which every couple of facets that form an angle greater than `<sFE_angle>` will be split in different solids.



- To choose the right angle we suggest to open the original STL in paraview and to use the **Feature Edge** filter to check the edge identification by varying angle value.

How does **surfaceFeatureEdges** works?

```
$> surfaceFeatureEdges <input> <output> -angle <sFE_angle>
```

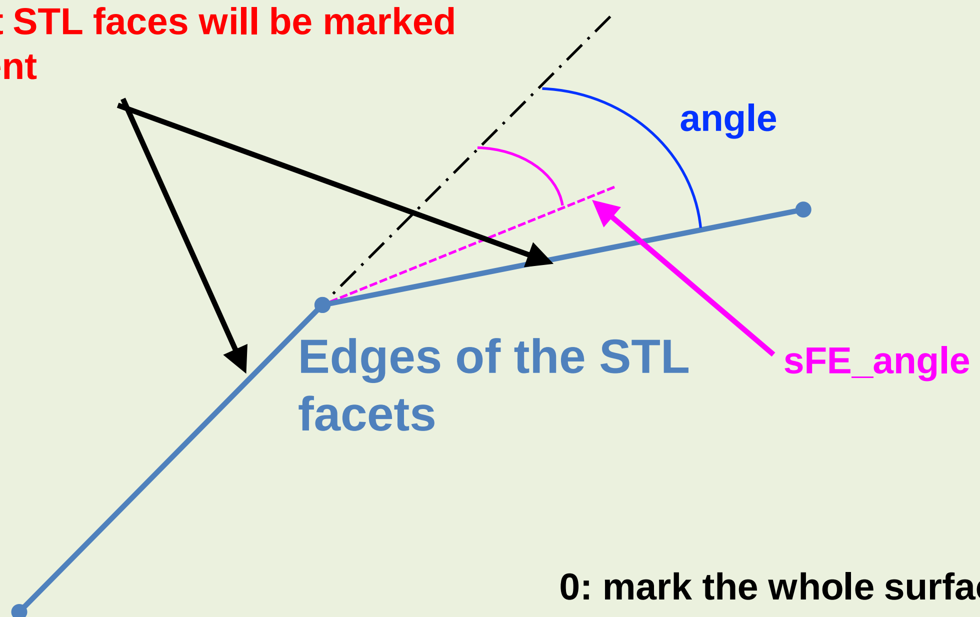


How does **surfaceFeatureEdges** works?

$> surfaceFeatureEdges <input> <output> **-angle <sFE_angle>**

angle > sFE_angle
Curvature refinement

If angle is more than sFE_angle the adjacent STL faces will be marked for refinement



0: mark the whole surface for refinement
180: do not mark any STL face for refinement

Cylinder tutorial

- And remember, when it comes to define boundary refinements with incremental names (e.g. , parentSolid_1, parentSolid_2, etc) we can simply refer to all of them by using regular expressions inside the `system/meshDict` dictionary:

```
"parentSolid_.*"
{
    cellSize 0.001;
}
```

- You may also want to group all patches of your input .stl in a unique patch. You can do it within the **renameBoundary** sub-dictionary `system/meshDict`.

```
renameBoundary
{
        newPatchNames
        {
                "parentSolid_.*"
                {
                        newName newSolid;
                        type wall;
                }
        }
}
```

Cylinder tutorial

- Let us use the utility `surfaceFeatureEdges`, type in the terminal window:

  ```
  $> surfaceFeatureEdges geo/cylinder.stl geo/cylinderSplit.stl
  -angle 90
  ```

  `surfaceFeatureEdges` will subdivide your original STL **solid** into several STL **solids** depending on the angle between the single faces as we have seen before.

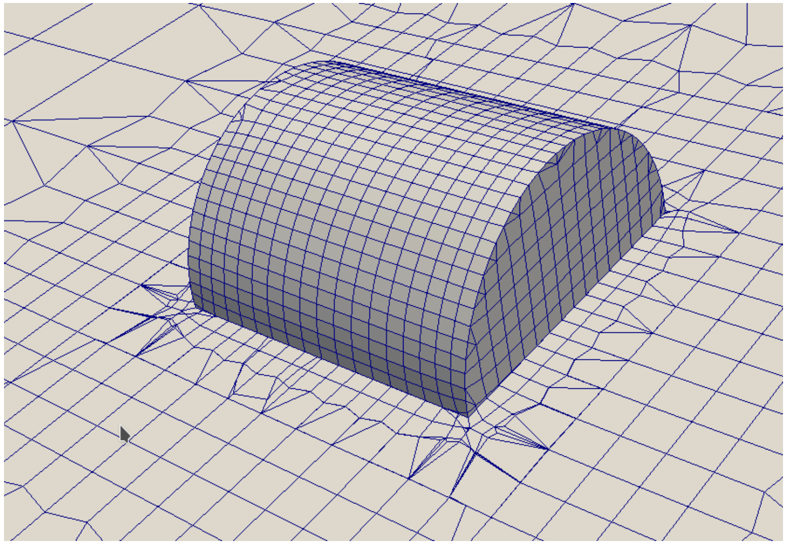
- The **solids** that share the same parent **solid** will have a common naming syntax in the form of **parent_solid_name+i** where **i** is an incremental number starting from 0.

- Now, we are ready to generate the mesh with `cartesianMesh`

Cylinder tutorial

- To compute the final mesh, type in the terminal:

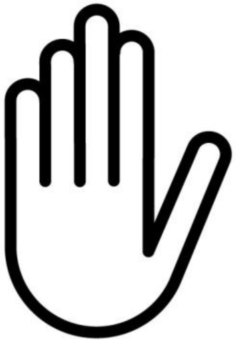
1. `$> foamCleanTutorials`
2. `$> cp -rp system/meshDict.org system/meshDict`
3. `$> surfaceFeatureEdges geo/cylinder.stl geo/cylinderSplit.stl  -angle 90`
4. `$> surfaceGenerateBoundingBox geo/cylinderSplit.stl constant/triSurface/boxCylinder.stl 9 9 9 9 9 9`
5. `$> cartesianMesh`
6. `$> checkMesh`
7. `$> paraFoam`



Cylinder tutorial

- Meshing with cfMesh.
- Meshing tutorial 2. The 3D Cylinder with boundary layer refinements.

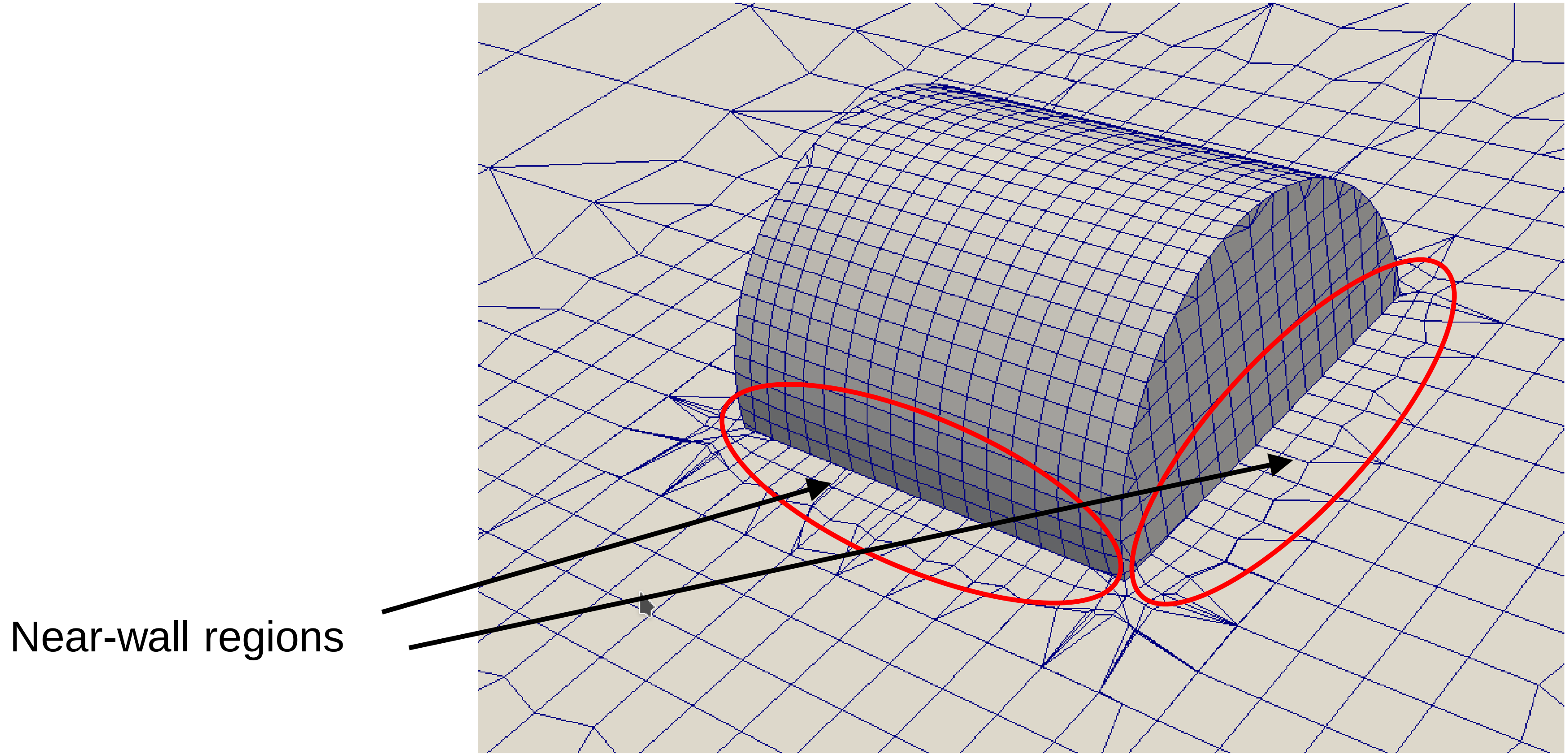
`$TM/CFMESH/c2_cyl_bl/`



- From this point on, please follow me.
- We are all going to work at the same pace.
- Remember, `$TM` is pointing to the path where you unpacked the tutorials.

Cylinder tutorial

- In many applications we have to perform an additional volume refinement close to the wall boundaries according to the near-wall treatment adopted in your CFD case (a topic addressed in the Turbulence modelling lesson).



Cylinder tutorial

- To achieve the desired boundary layer refinement, we set the **boundaryLayers** settings in the `system/meshDict` dictionary

- **cfMesh** requires to specify the number of layers, the growth ratio and the maximum thickness of the first layer.

- The boundary layer refinement settings can be **global** or **local**

- In order to provide patch-specific refinements we can use the **patchBoundaryLayers** sub-dictionary

```
boundaryLayers
{
    patchBoundaryLayers
    {
    "cyl_.*"
        {
            nLayers 3;
            thicknessRatio  1.2;
            maxFirstLayerThickness 0.06;
            allowDiscontinuity 1;
        }
    }
}
```

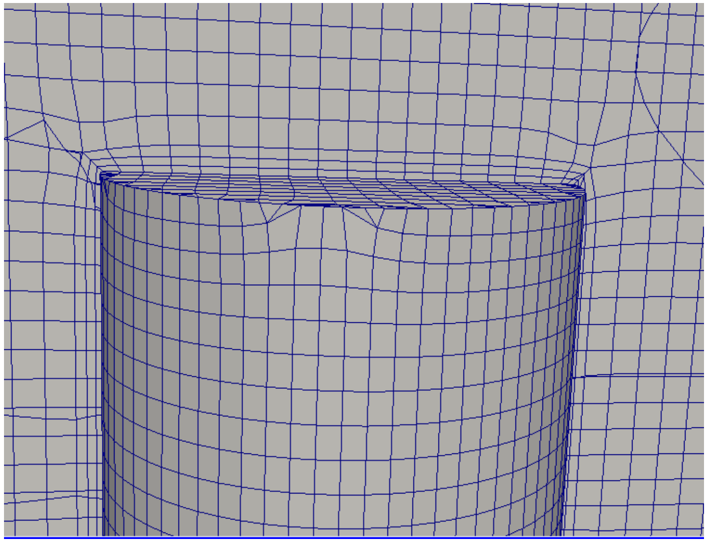
The **allowDiscontinuity** option ensures that the number of layers required for a patch shall not spread to other patches in the same layer

Cylinder tutorial

- To achieve the desired boundary layer refinement, we set the **boundaryLayers** settings in the `system/meshDict` dictionary

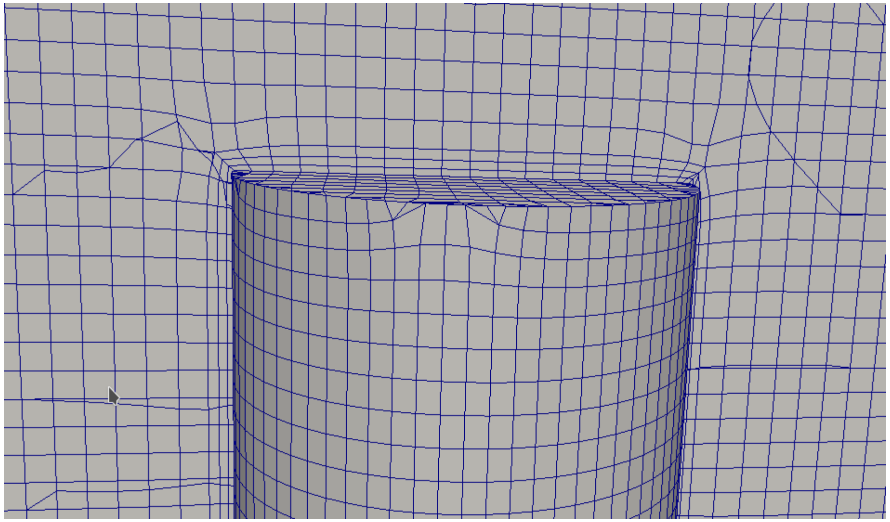
- **cfMesh** requires to specify the number of layers, the growth ratio and the maximum thickness of the first layer.

```
boundaryLayers
{
        patchBoundaryLayers
        {
        "cyl_.*"
                {
                        nLayers 3;
                        thicknessRatio  1.2;
                        maxFirstLayerThickness 0.06;
                        allowDiscontinuity 1;
                }
        }
}
```

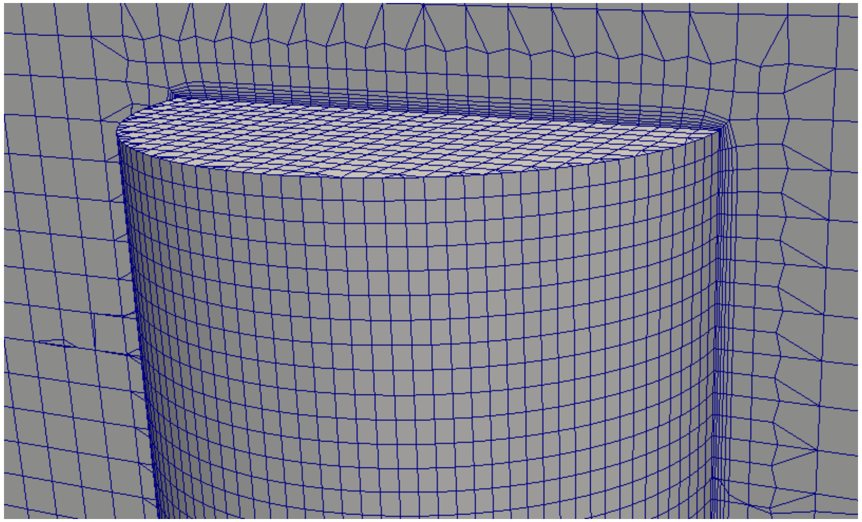


Cylinder tutorial

- Sometimes we need a higher level of refinement in the near-wall region. We suggest to play around with the boundary layers settings and to check the resulting mesh quality with the `checkMesh` utility and `paraFoam`.



nLayers 3;
thicknessRatio 1.2;
maxFirstLayerThickness 0.06;
allowDiscontinuity 1;



nLayers 6;
thicknessRatio 1.2;
maxFirstLayerThickness 0.03;
allowDiscontinuity 1;

Cylinder tutorial

- In the terminal window we provide the following commands:

1. `$> foamCleanTutorials`
2. `$> cp -rp system/meshDict.org system/meshDict`
3. `$> surfaceFeatureEdges geo/cylinder.stl geo/cylinderSplit.stl -angle 90`
4. `$> surfaceGenerateBoundingBox geo/cylinderSplit.stl constant/triSurface/boxCylinder.stl 9 9 9 9 9 9`
5. `$> surfaceCheck`
6. `$> cartesianMesh`
7. `$> checkMesh`
8. `$> paraFoam`

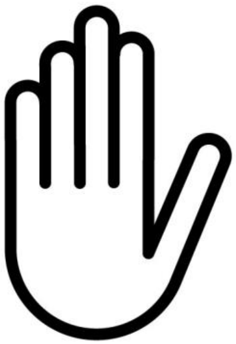
Roadmap

1. ~~Mesh quality assessment in CFD~~
2. ~~Mesh generation using cfMesh~~
3. ~~The cylinder tutorial~~
4. **The 2D airfoil tutorial**
5. The static mixer tutorial
6. The Ahmed body tutorial
7. The mixing elbow comparison
8. The moving quadcopter tutorial

Cylinder tutorial

- Meshing with cfMesh.
- Meshing tutorial 3. The 2D airfoil (external mesh)

```
$TM/CFMESH/c3_2Dairfoil
```



- From this point on, please follow me.
- We are all going to work at the same pace.
- Remember, `$TM` is pointing to the path where you unpacked the tutorials.

2D airfoil tutorial

- To generate this mesh, type in the terminal:

1. `$> foamCleanTutorials`
2. `$> surfaceGenerateBoundingBox geo/naca0012.stl geo/naca2D.stl 10 20 10 10 0 0`
3. `$> cartesian2DMesh`
4. `$> checkMesh`
5. `$> paraFoam`

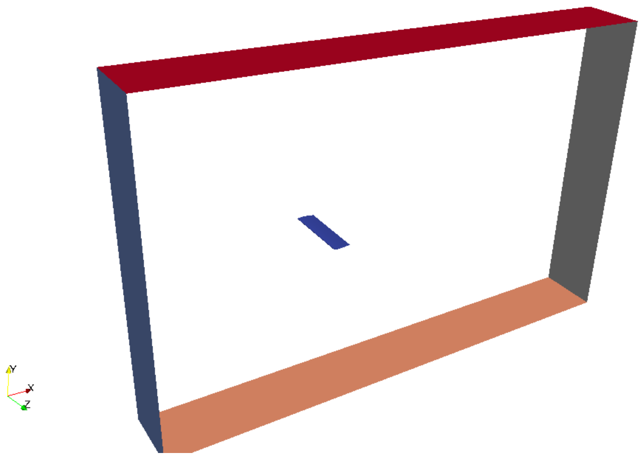
2D airfoil tutorial

- To generate 2D meshes with cfMesh we use the `cartesian2DMesh` utility that performs spatial discretizations in the x-y plane by using hexahedral cells.

- The z direction will be ignored by OpenFOAM solvers since is not subdivided in multiple cells.

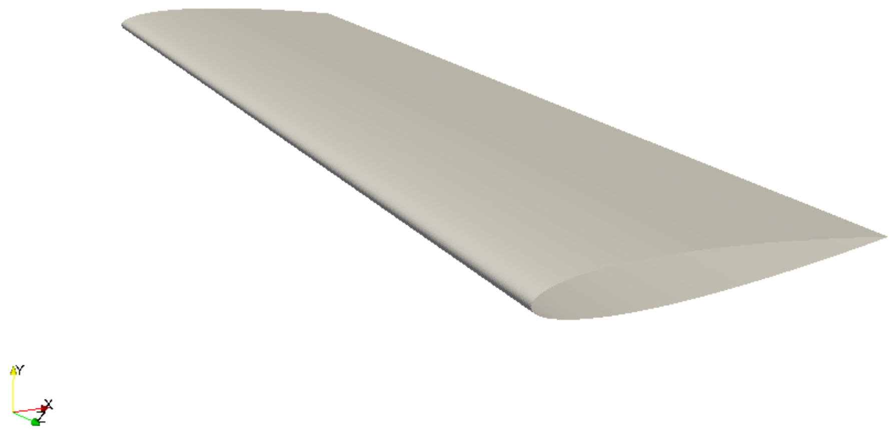
- `cartesian2DMesh` reads the *`system/meshDict`* dictionary. The information provided in the dictionary are the same illustrated for the `cartesianMesh` cases.

- An important requirement of `cartesian2DMesh` is to input geometries in a form of a ribbon in the x-y plane and extruded in the z direction. As usual, stl and fms formats are valid options for the geometry input file.



2D airfoil tutorial

- This tutorial starts from the *`geo/naca0012.stl`* airfoil geometry file.

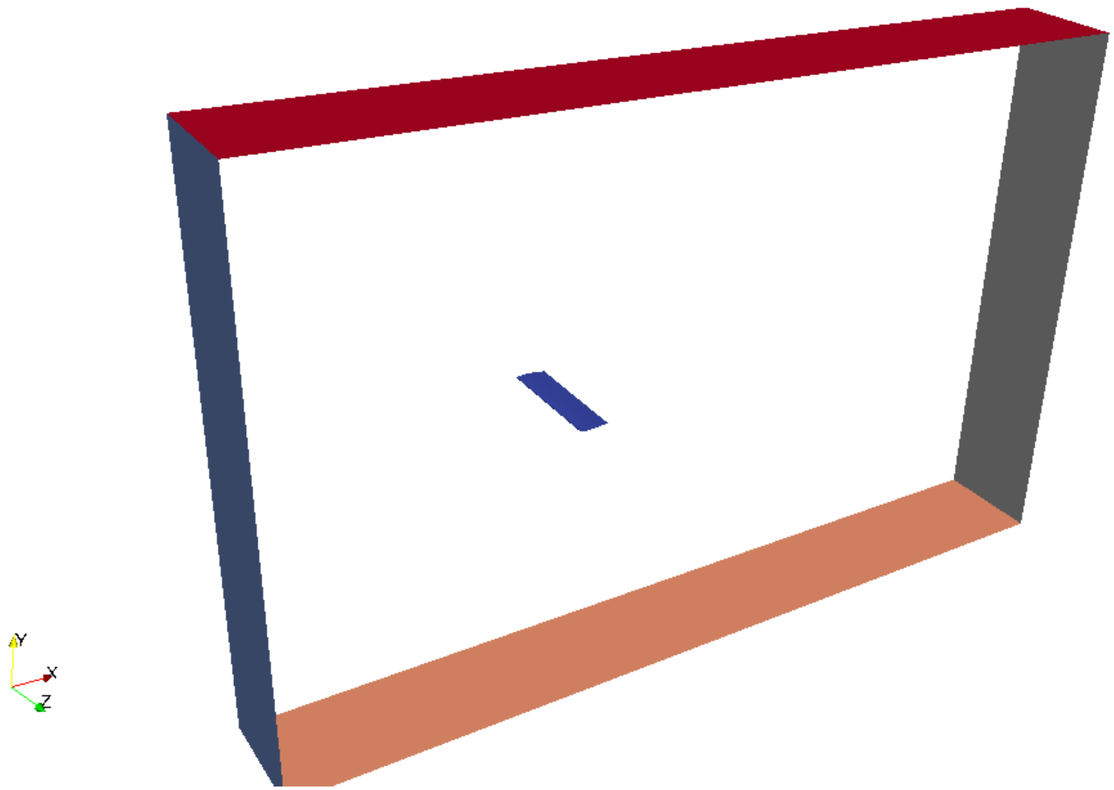


- Similarily to the previous case, we need to build a 2D bounding box around the airfoil spatial domain in order to set-up this external aero-dynamics case.

2D airfoil tutorial

- In order to generate the bounding box we use the `surfaceGenerateBoundingBox` utility as follows:

```
$> surfaceGenerateBoundingBox geo/naca0012.stl geo/naca2D.stl
10 20 10 10 0 0
```



2D airfoil tutorial

- The *system/meshDict* dictionary contains the following instructions:

```
…
maxCellSize     0.25; //[m]
surfaceFile     "geo/naca2D.stl";
objectRefinements
{
        wake
        {
                cellSize 0.05; // [m]
                type    box; // or box or sphere or line
                centre      (11 0 0);
                lengthX     24;
                lengthY     1.6;
                lengthZ     1;
        }
}
```

We are defining a rectangular regions that will be refined with A 0.05 m size

2D airfoil tutorial

- The *system/meshDict* dictionary contains the following instructions:

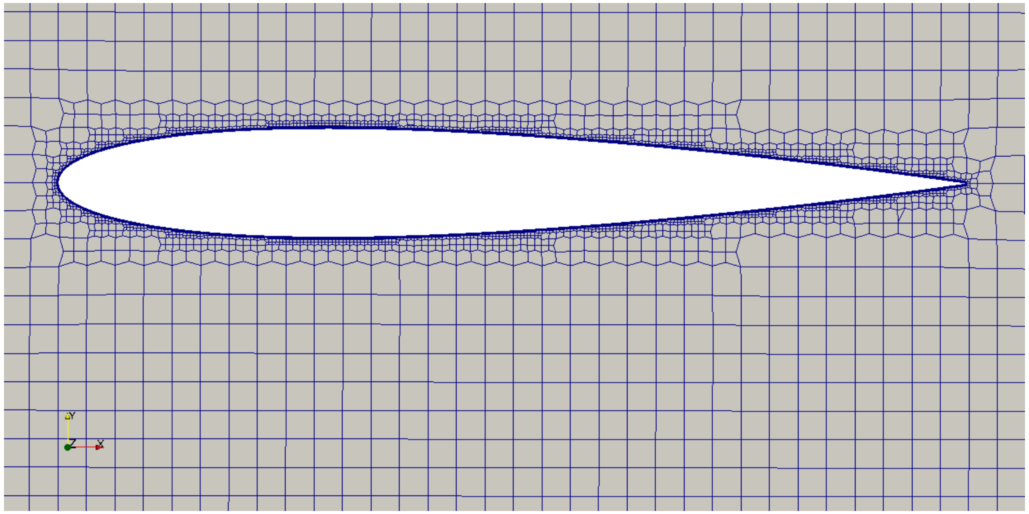
```
localRefinement
{
        patch0
        {
                additionalRefinementLevels 6;
        }
}
boundaryLayers
{
        patchBoundaryLayers
        {
                patch0
                {
                        nLayers         5;
                        thicknessRatio  1.2;
                        maxFirstLayerThickness 1;
                        allowDiscontinuity 1;
                }
        }
}
```

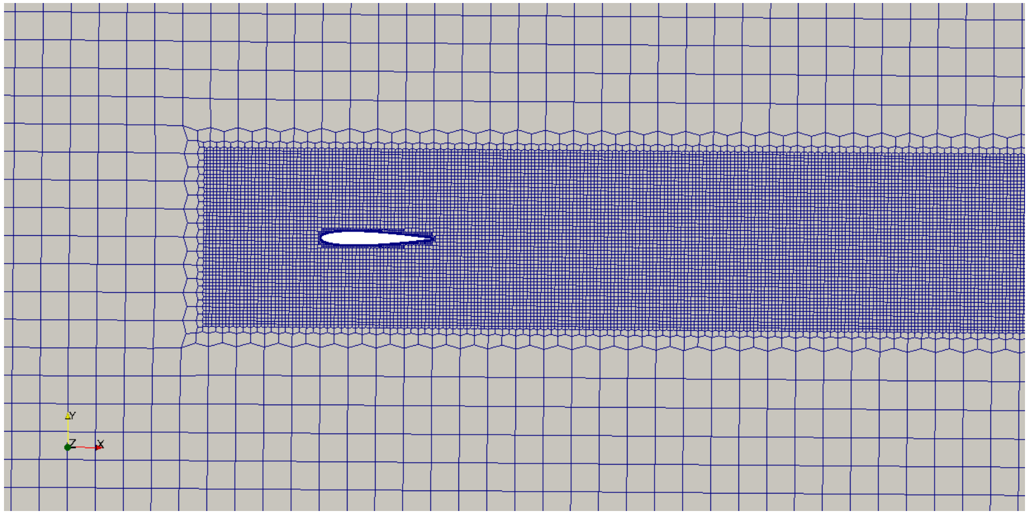
Refinement along specific patches defined in the input geometry file

Local boundary layers refinement

2D airfoil tutorial

- Visualizing the mesh with **paraview**:



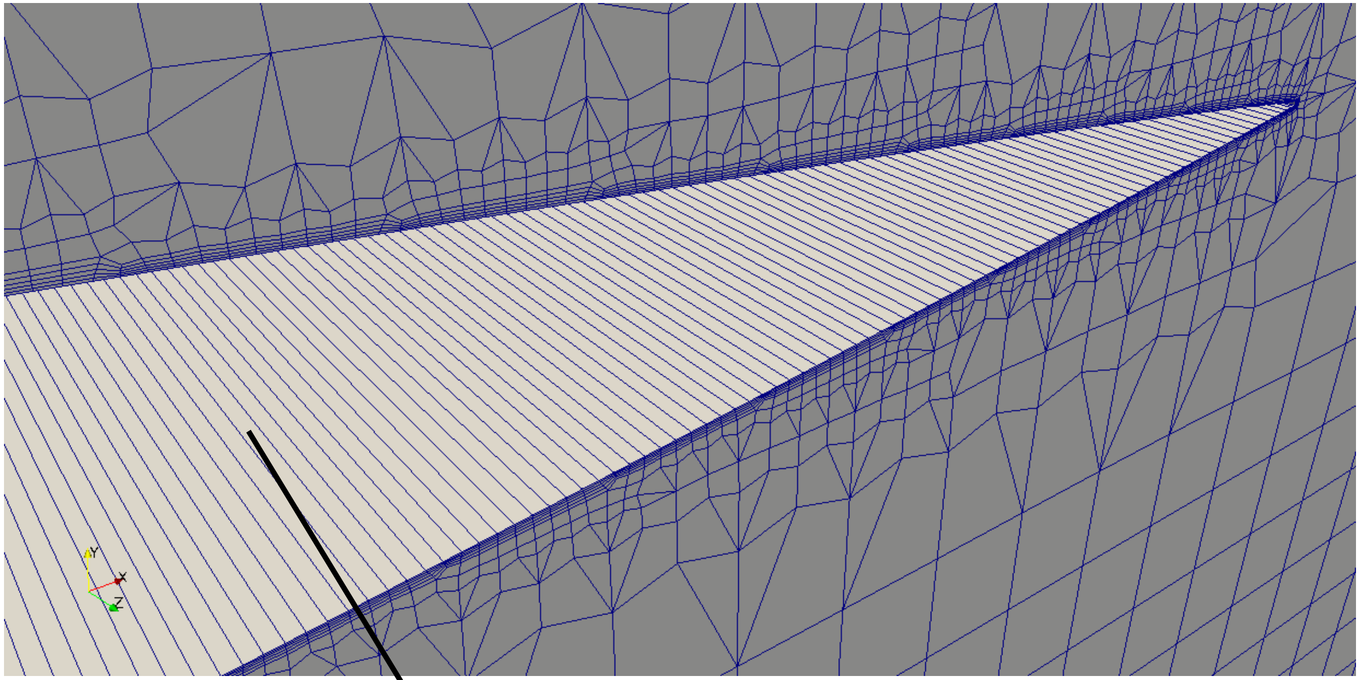


In order to perform a 2D visualization of the airfoil mesh it may be useful to load only the **bottomEmptyFaces** mesh part.

The result is represented in the left figures.

2D airfoil tutorial

- Visualizing the mesh with **paraview**:



Note that the actual internal mesh has a third dimension (along the coordinate z) that does not present any cells subdivision.

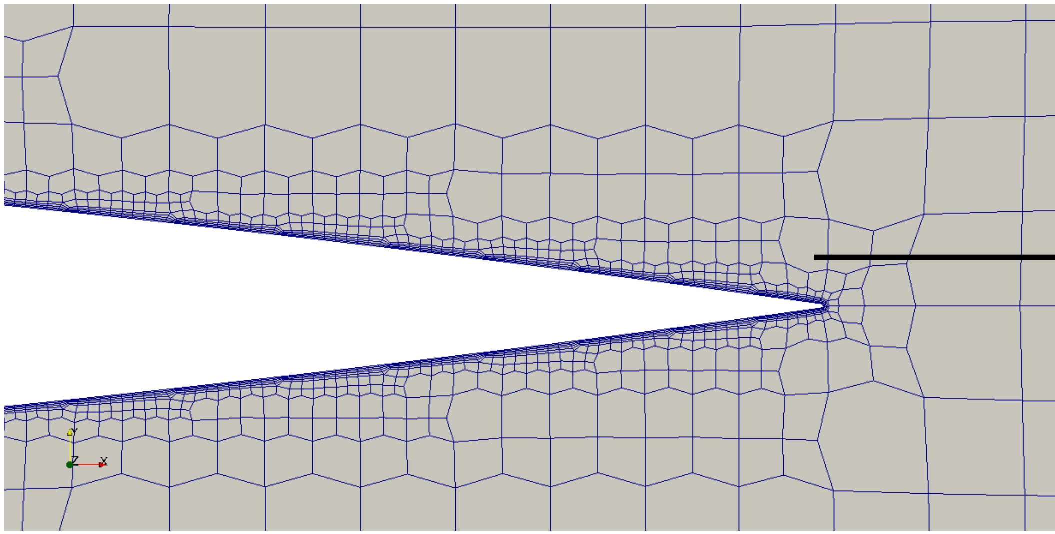
This is the standard OpenFOAM way to set-up a bi-dimensional case.

2D airfoil tutorial

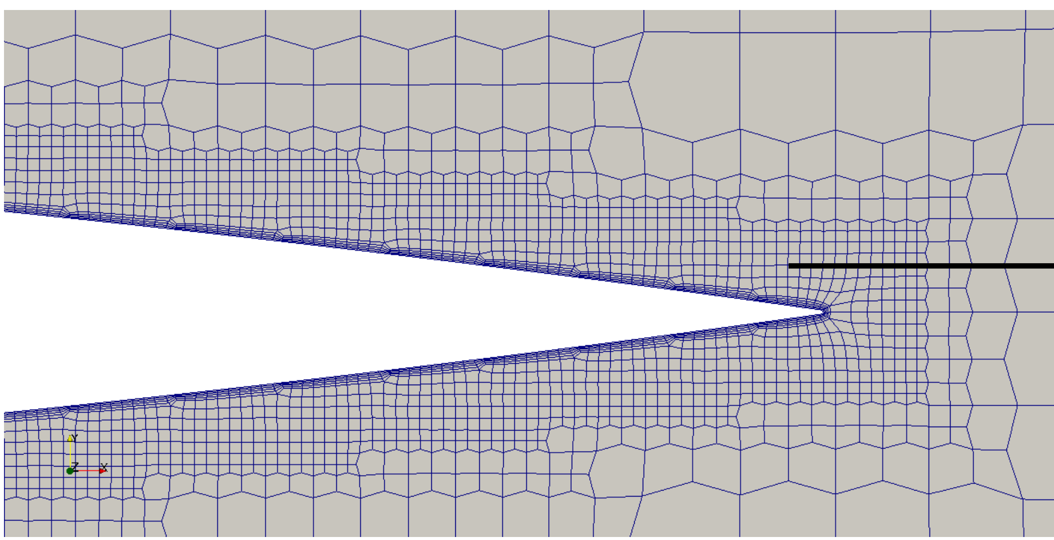
- In case it is necessary to enlarge the airfoil patch refinement region, a solution is to use the **refinementThickness** instruction in the **localRefinement** section of the cfMesh dictionary as follows:

```
localRefinement
{
        patch0
        {
                additionalRefinementLevels 6;
                refinementThickness 0.02; //cm
        }

}
```

2D airfoil tutorial



no refinementThickness;



refinementThickness 0.02;

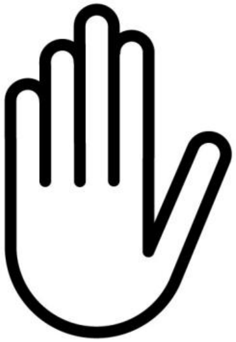
Roadmap

1. ~~Mesh quality assessment in CFD~~
2. ~~Mesh generation using cfMesh~~
3. ~~The cylinder tutorial~~
4. ~~The 2D airfoil tutorial~~
5. **The static mixer tutorial**
6. The Ahmed body tutorial
7. The mixing elbow comparison
8. The moving quadcopter tutorial

Static mixer tank tutorial

- Meshing with cfMesh.
- Meshing tutorial 4. The 3D static mixer tank and geometry surfaces manipulation

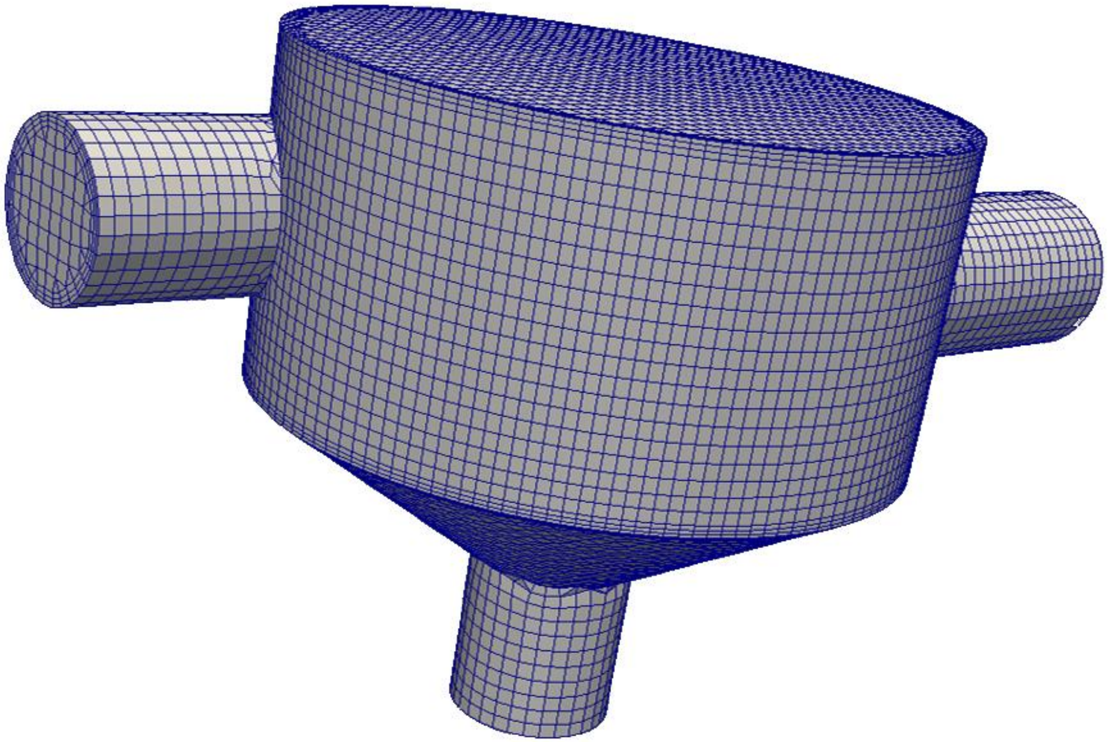
```
$TM/CFMESH/c4_staticMixer
```



- From this point on, please follow me.
- We are all going to work at the same pace.
- Remember, `$TM` is pointing to the path where you unpacked the tutorials.

Static mixer tank tutorial

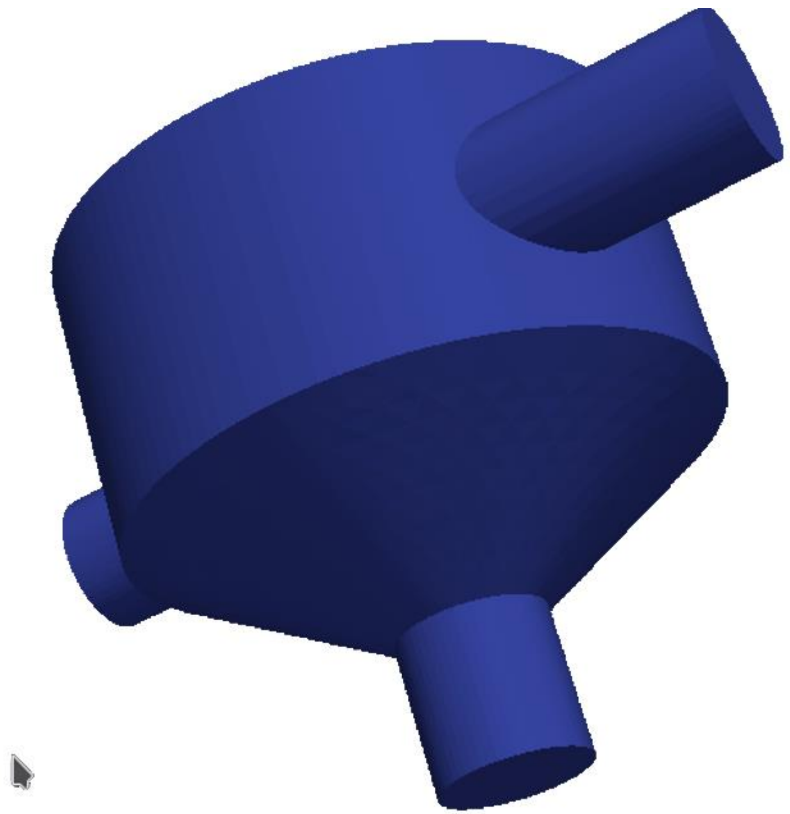
At the end of this tutorial we will be able to mesh a static mixer tank like this:



Static mixer tank tutorial

For this internal aero-dynamic problem we have a STL geometry composed by one *solid.* Try now to check the geometry by using paraview.

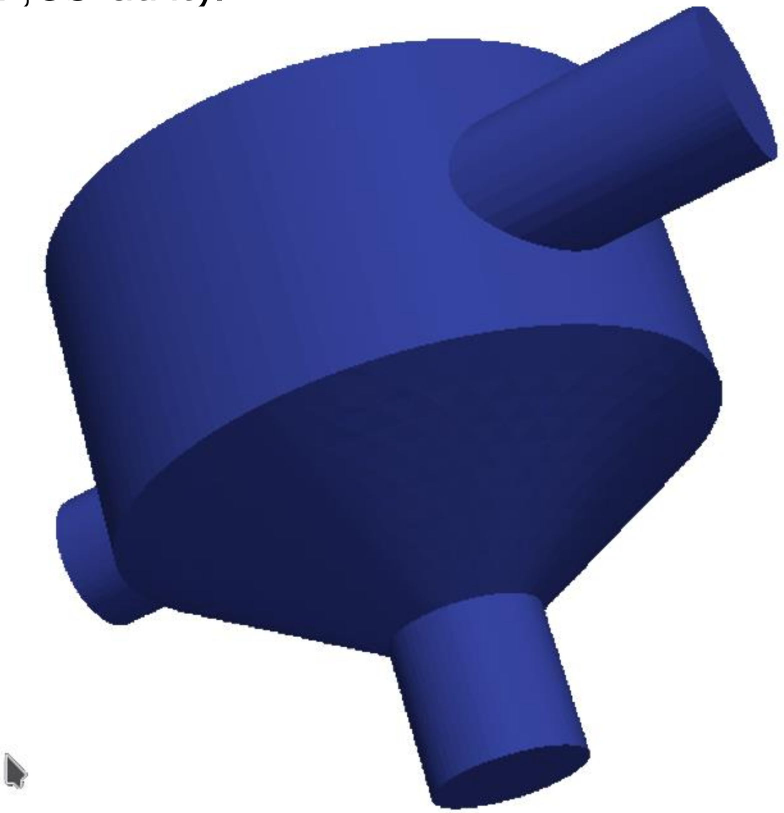
```
$> paraview geo/staticMixer.stl
```



Static mixer tank tutorial

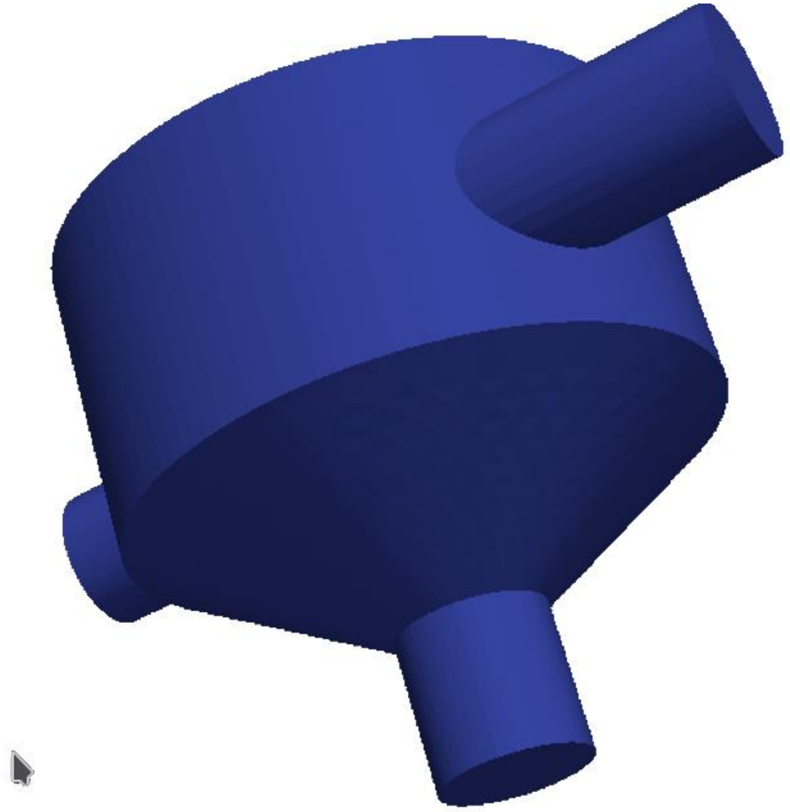
We will provide different names for the wall surfaces and inlet/outlet patches inside the STL file. In this manner we will be able to specify local meshing operations with cfMesh.

Again, the idea is to split the STL *solid* into multiple *solids* by using `surfaceFeatureEdges` utility.



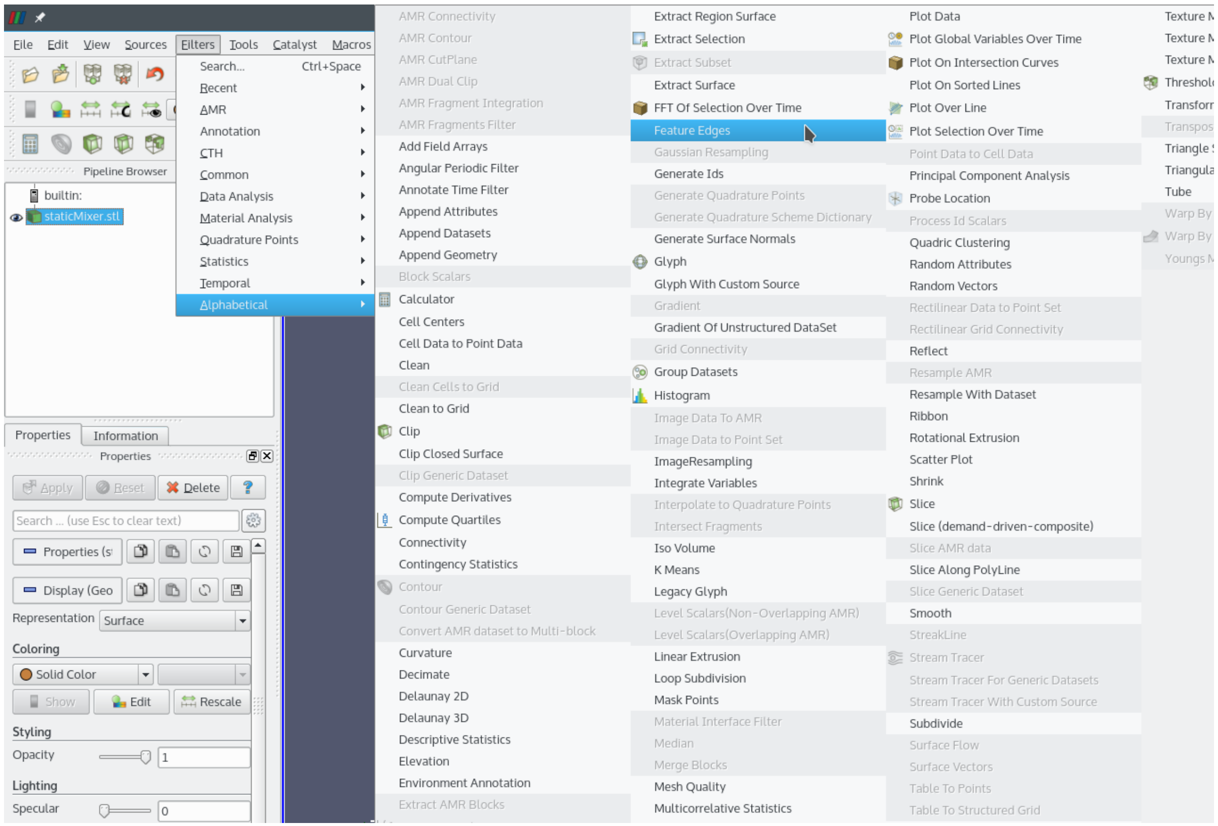
Static mixer tank tutorial

As seen during the cylinder tutorial, another fundamental advantage of using the STL splitting technique is to help cfMesh in recognizing the mixing tank feature edges for refinements.



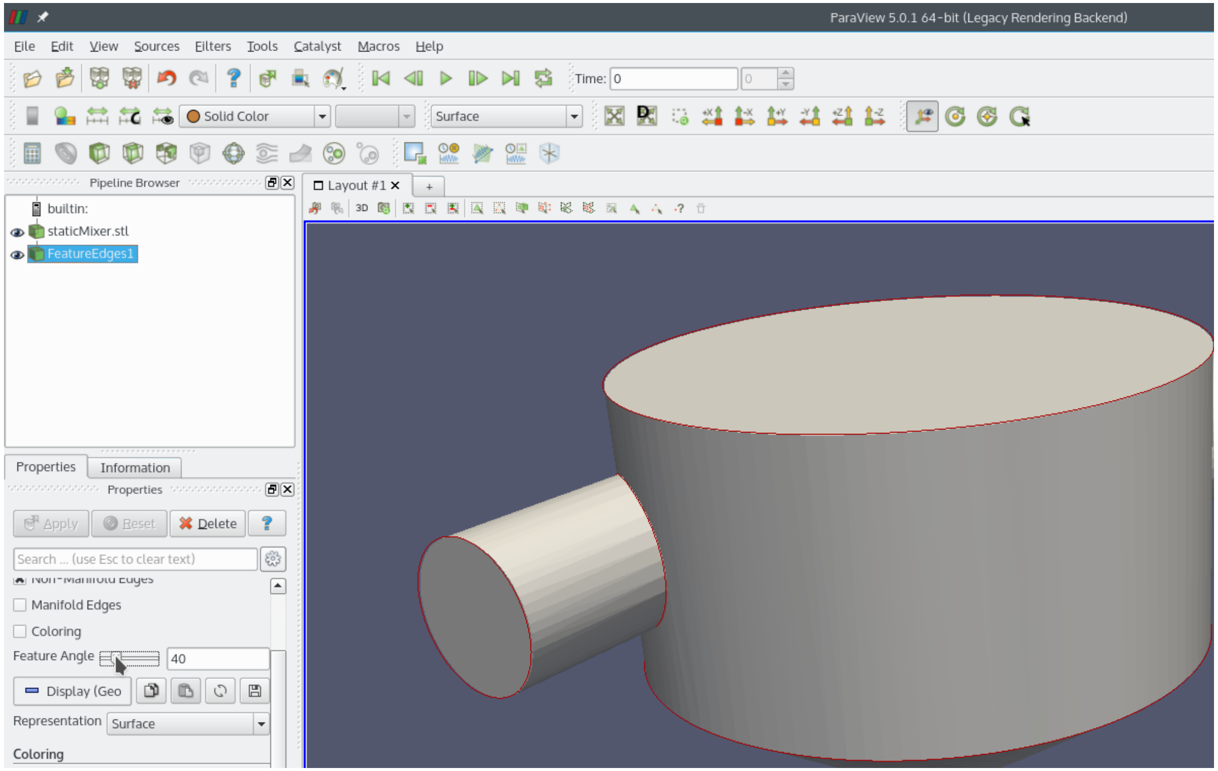
Static mixer tank tutorial

Before doing that, we suggest to open the *geo/staticMixer.stl* geometry in paraview and to select the Feature Edge filter.



Static mixer tank tutorial

Playing around with the **Feature Angle** toggle will help you in identifying the correct angle value to be used in the `surfaceFeatureEdges` operation. According to the **Feature Angle** level, different edges will be highlighted in the static mixed viewer.



Static mixer tank tutorial

After the solid split operation, we obtain an STL file composed by several *solids* automatically renamed with an incremental number.

In order to handle the different surface parts in cfMesh (or snappyHexMesh) it is convenient to rename them directly in the STL ASCII file.

```
solid solid_0
  facet normal 0 0 1
    outer loop
       vertex 1.43357 1.39459 2
       ...
    endloop
  endfacet
endsolid solid_0
solid solid_1
  facet normal 0 0 1
    outer loop
    ...
```

Static mixer tank tutorial

We can easily deal with an high number of **solids** to be renamed by using the Linux stream text editor utility as follows:

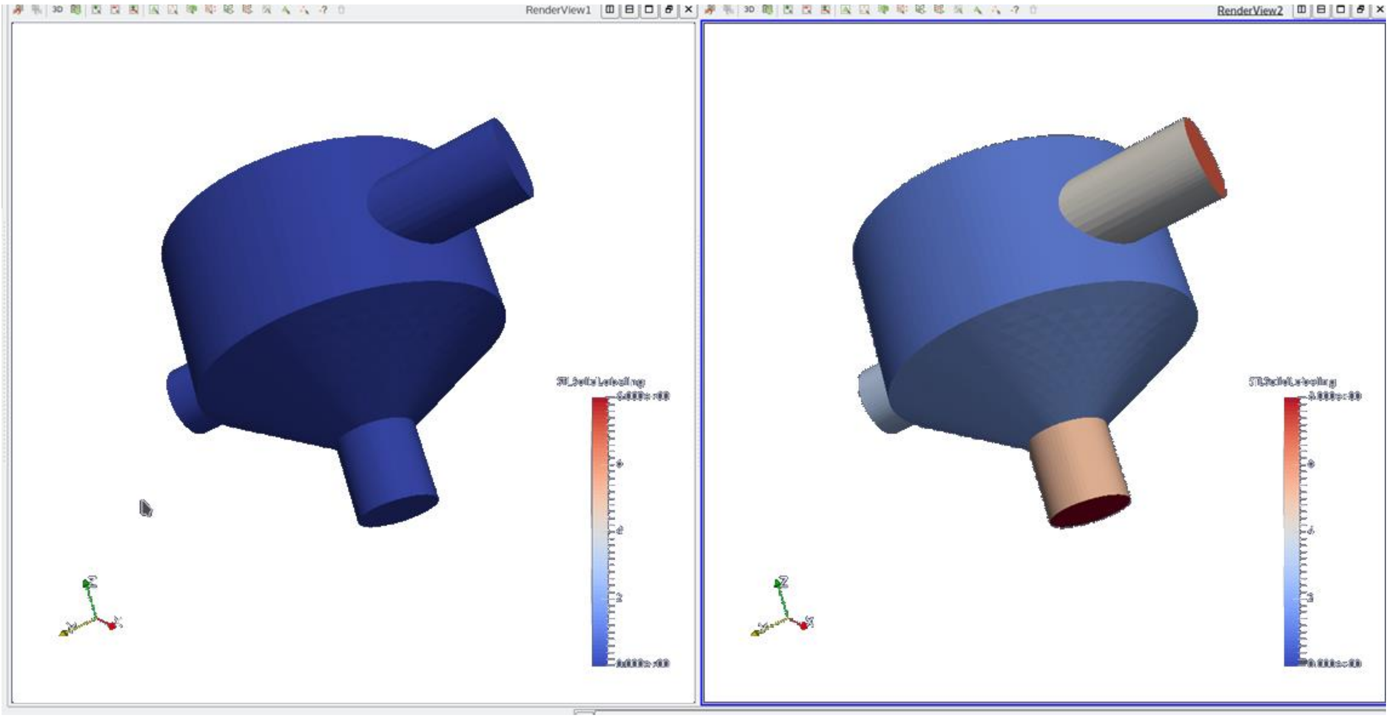
```
$> sed -i 's/<string_to_replace>/<new_string>/g' <file_name>
```

Remember, this operation is possible only if your .stl geometry has been exported as an ASCII file.

The results of this kind of surface manipulation can be easily monitored by taking advantage of the OpenFOAM `surfaceCheck` utility:

```
$> surfaceCheck <file_name>
```

This is the result of our geometry manipulation:



Static mixer tank tutorial

In the terminal window we provide the following commands:

1. `$> foamCleanTutorials`
2. `$> cp -rp` **`system/meshDict.org system/meshDict`**
3. `$> export OMP_NUM_THREADS=2`
4. `$> surfaceFeatureEdges -angle 40` **`geo/staticMixer.stl geo/smsplit.stl`**
5. `$> paraFoam`

Static mixer tank tutorial

In the terminal window we provide the following commands:

```
6.  $> sed -i 's/solid_0/wall_0/g' geo/smsplit.stl
7.  $> sed -i 's/solid_1/wall_1/g' geo/smsplit.stl
8.  $> sed -i 's/solid_2/wall_2/g' geo/smsplit.stl
9.  $> sed -i 's/solid_3/wall_3/g' geo/smsplit.stl
10. $> sed -i 's/solid_4/wall_4/g' geo/smsplit.stl
11. $> sed -i 's/solid_5/wall_5/g' geo/smsplit.stl
12. $> sed -i 's/solid_6/inlet_1/g' geo/smsplit.stl
13. $> sed -i 's/solid_7/inlet_2/g' geo/smsplit.stl
14. $> sed -i 's/solid_8/outlet/g' geo/smsplit.stl
```

Static mixer tank tutorial

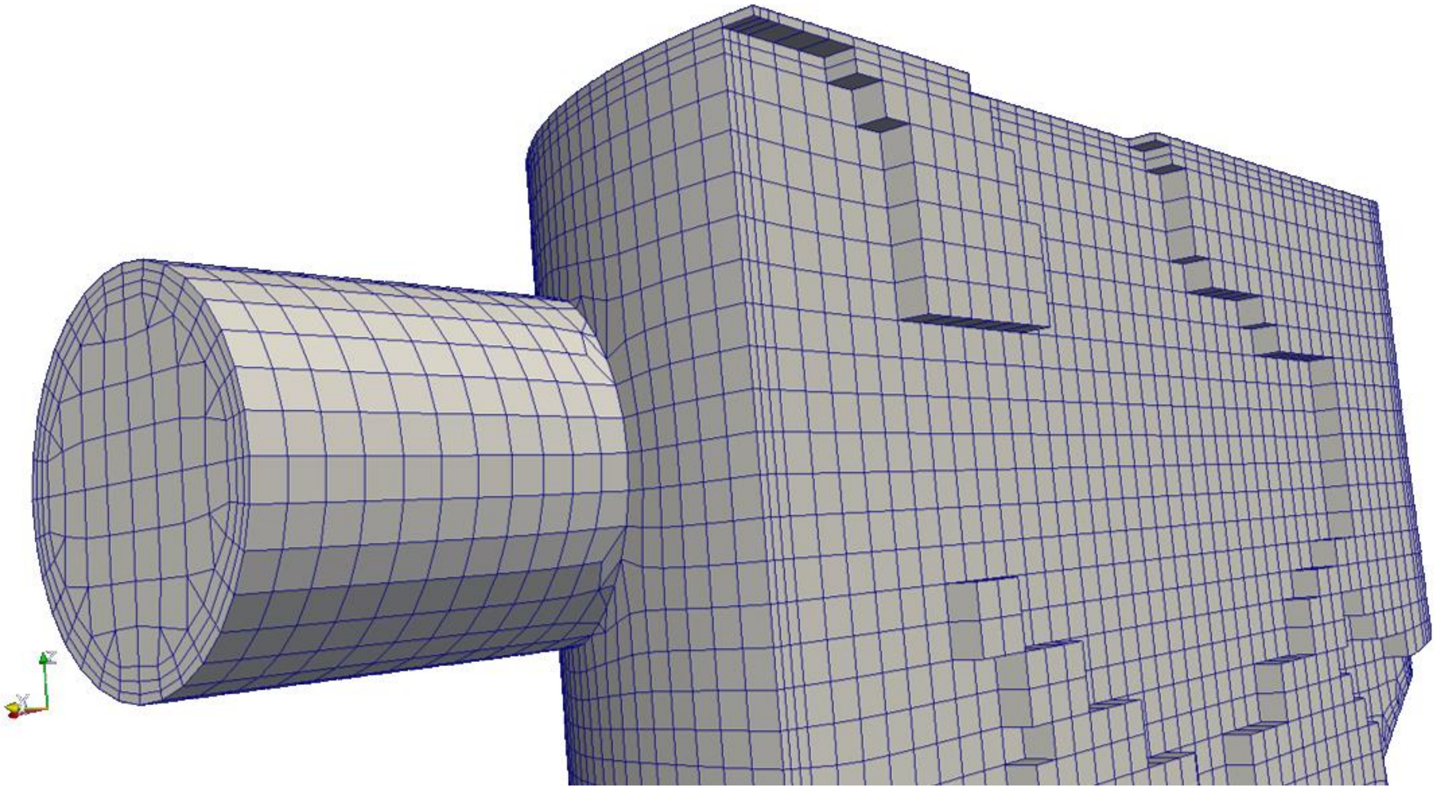
And finally:

15. `$> surfaceCheck` **`geo/smsplit.stl`**
16. `$> cartesianMesh`
17. `$> checkMesh`

The `export OMP_NUM_THREADS=2` (step 3) command instruct cfMesh to perform the mesh computations in a parallel mode over 2 different threads. This parameter can be changed arbitrarily in order to exploit the computing capabilities of your hardware.

Static mixer tank tutorial

The **Clip** and **Slice** paraview filters may help you in supervise the internal spatial discretization and the result of wall refinements



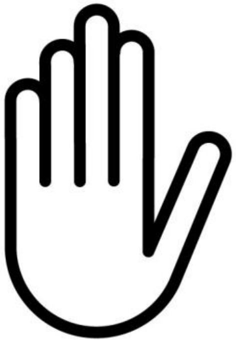
Roadmap

1. ~~Mesh quality assessment in CFD~~
2. ~~Mesh generation using cfMesh~~
3. ~~The cylinder tutorial~~
4. ~~The 2D airfoil tutorial~~
5. ~~The static mixer tutorial~~
6. **The Ahmed body tutorial**
7. The mixing elbow comparison
8. The moving quadcopter tutorial

Ahmed body tutorial

- Meshing with cfMesh.
- Meshing tutorial 5. Dealing with edge features and the 3D Ahmed body (external mesh)

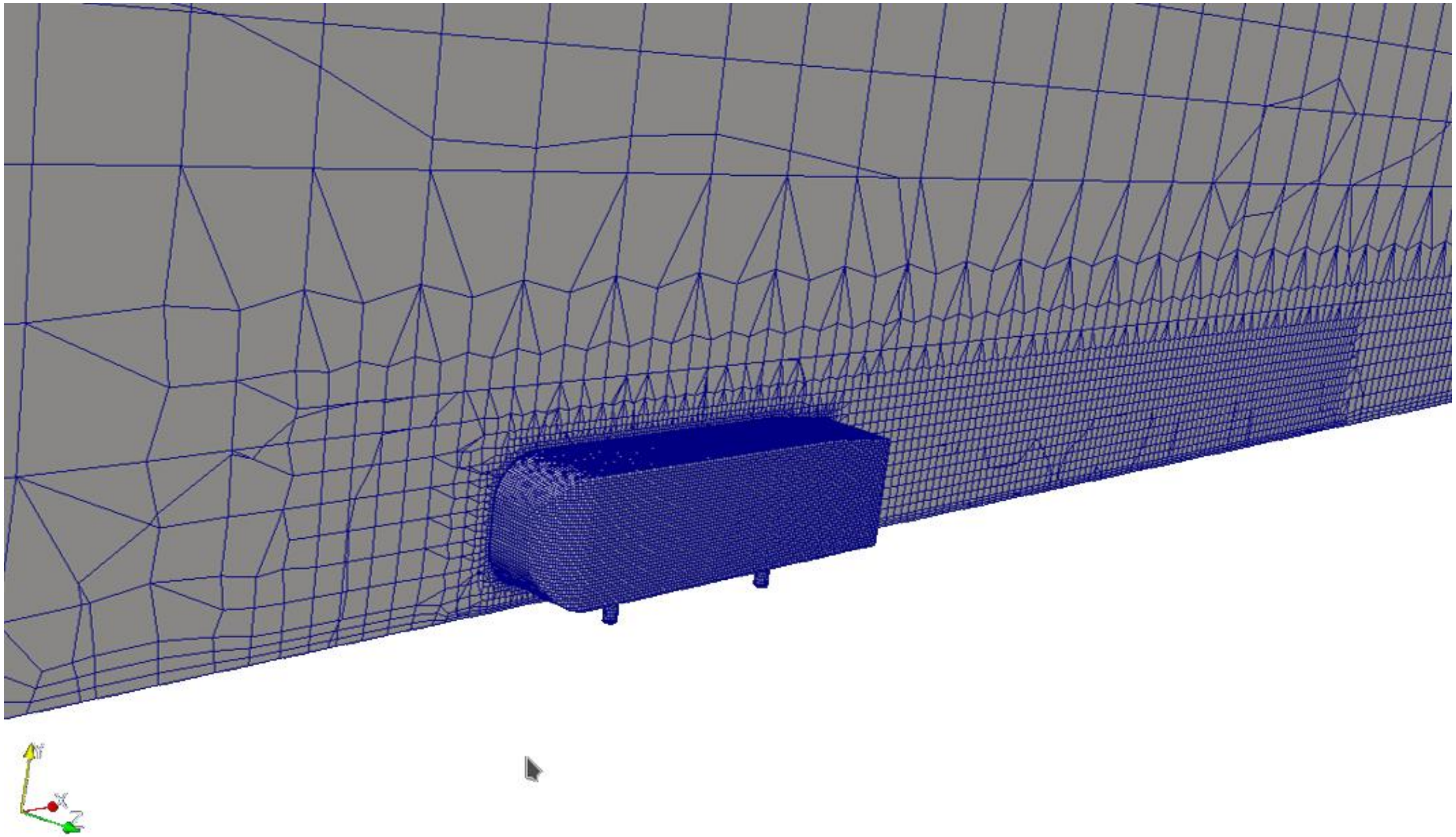
```
$TM/CFMESH/c5_ahmed
```



- From this point on, please follow me.
- We are all going to work at the same pace.
- Remember, `$TM` is pointing to the path where you unpacked the tutorials.

Ahmed body tutorial

At the end of this tutorial the final mesh should look like this



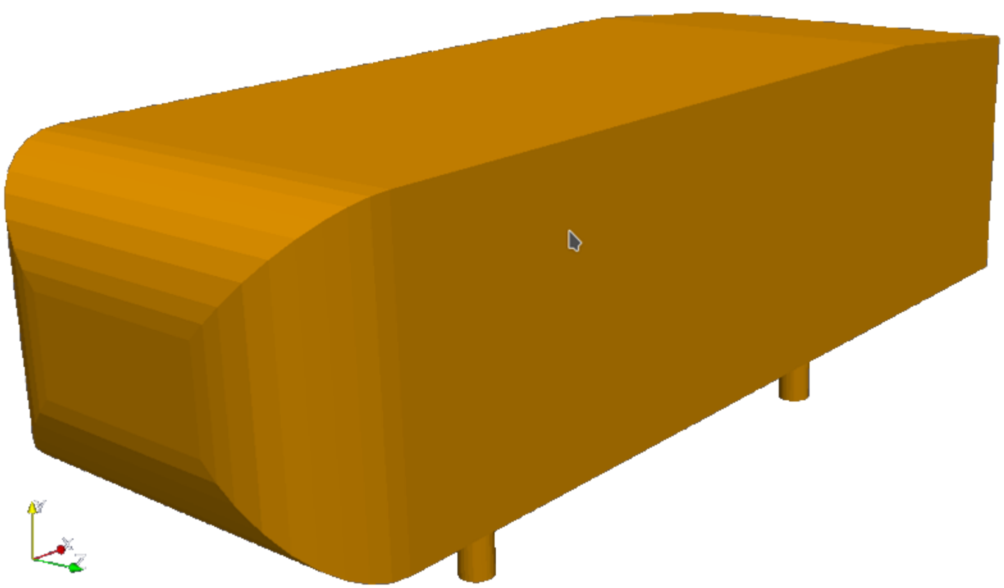
Ahmed body tutorial

The Ahmed body tutorial can be run by following these steps:

1. `$> foamCleanTutorials`
2. `$> cp system/meshDict.stl system/meshDict`
3. `$> surfaceGenerateBoundingBox constant/triSurface/abscale.stl abb.stl 4 6 1 1 0 2`
4. `$> surfaceFeatureEdges -angle 80  abb.stl constant/triSurface/abb.stl`
5. `$> surfaceCheck constant/triSurface/abb.stl`
6. `$> cartesianMesh`
7. `$> checkMesh`

Ahmed body tutorial

We start from our `constant/triSurface/ab.stl` surface file generated during the solid modelling lesson by using the Salome suite.

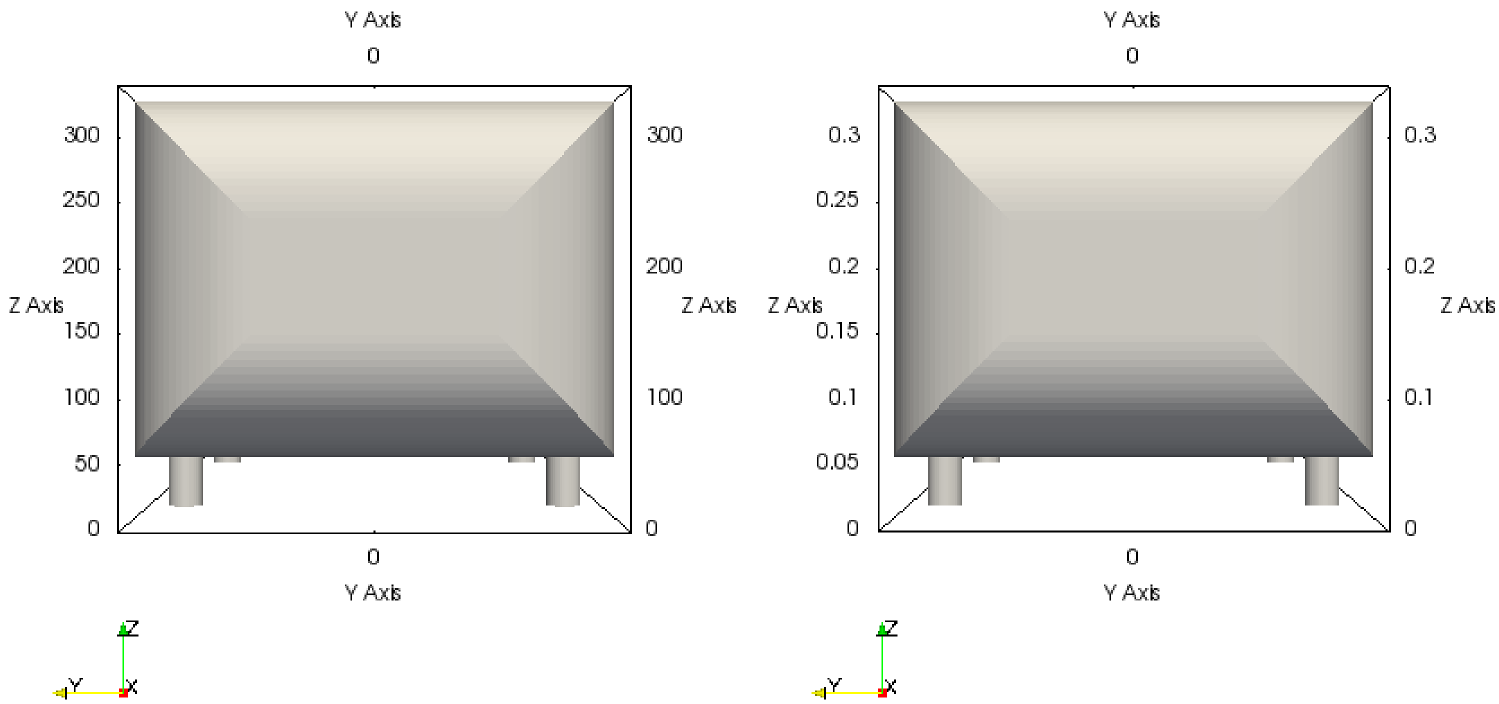


Please, remember that our CAD model was made in mm units. In this case, we will use the `surfaceTransformPoints` utility to scale our geometry in meter units.

`surfaceTransformPoints` can handle several STL transformations, check the help option to get a quick overview of its capabilities.

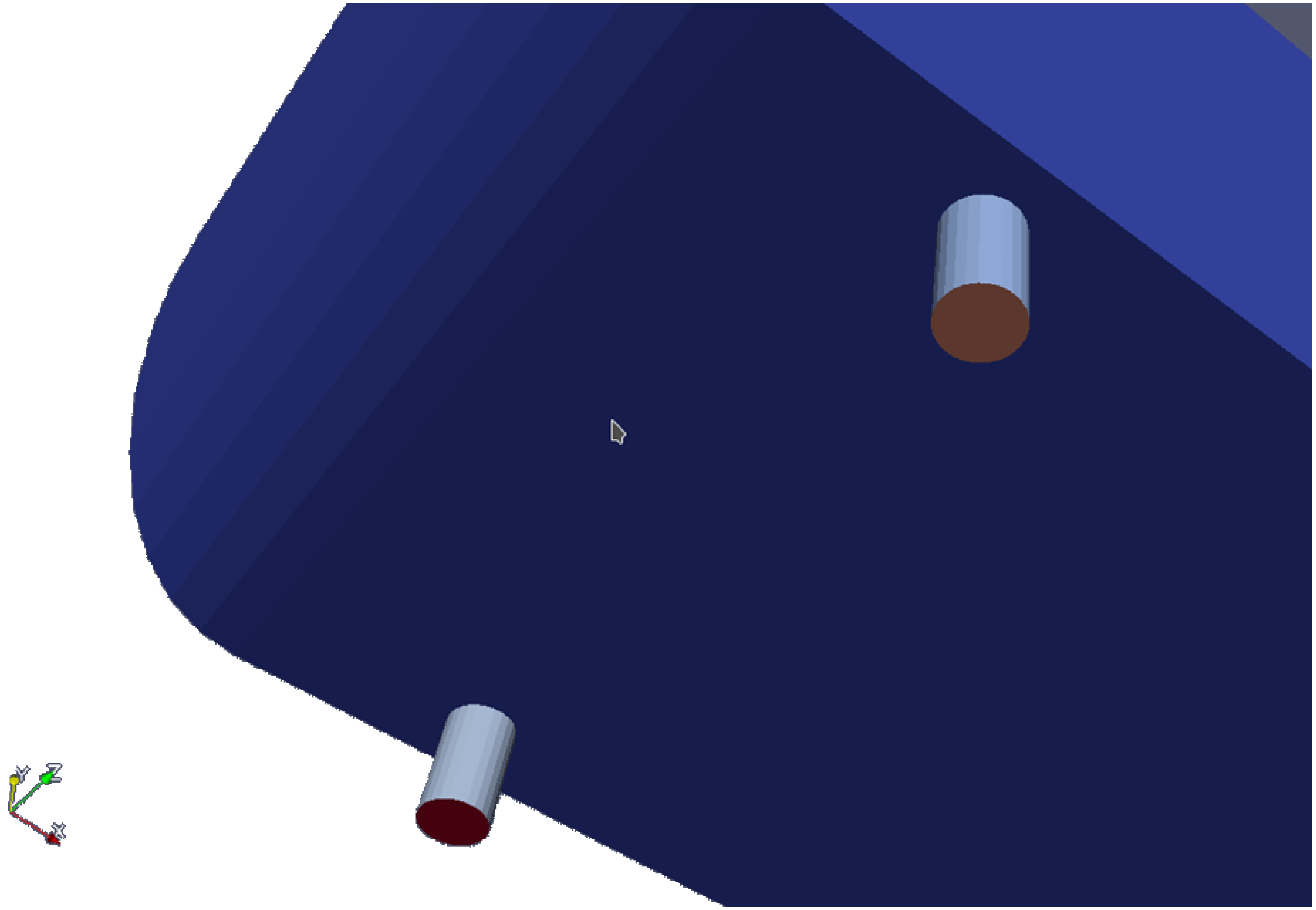
In our case we may entry the following command in the Linux shell:

```
surfaceTransformPoints -scale '(0.001 0.001 0.001)'
constant/triSurface/ab.stl constant/trisurface/abscale.stl
```



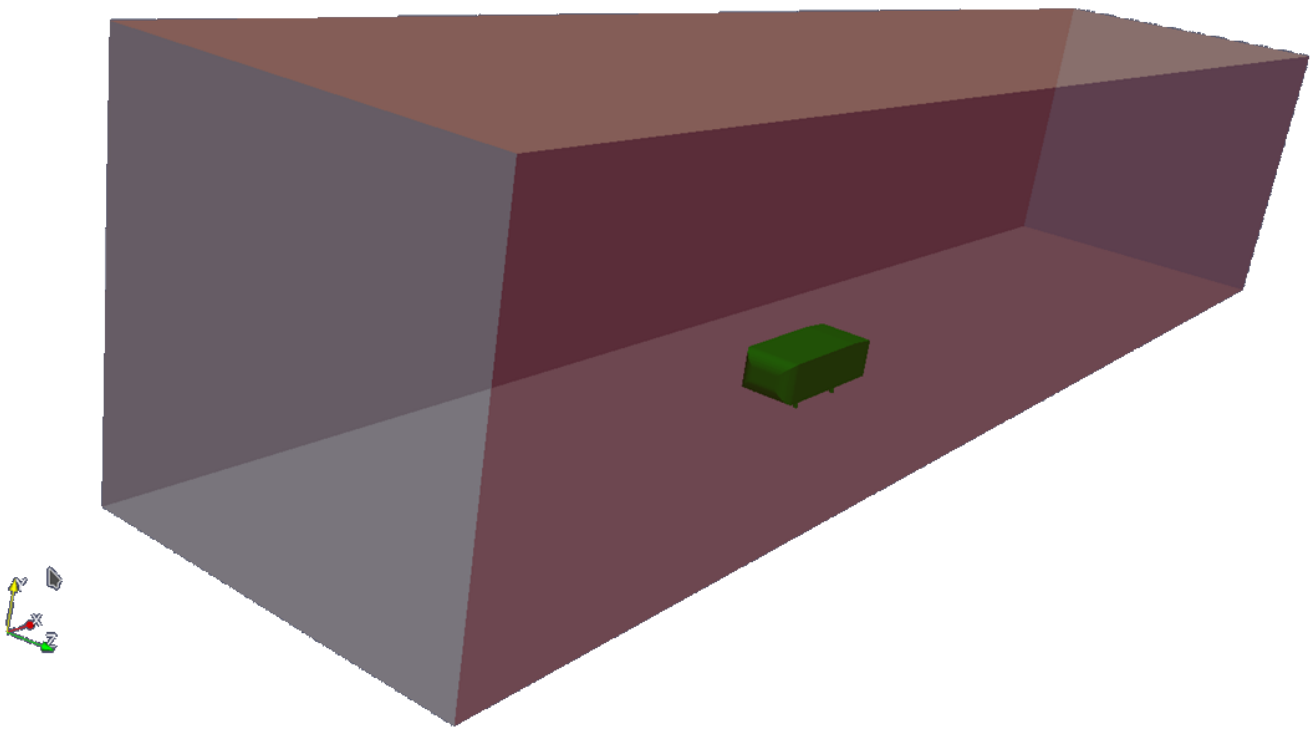
Ahmed body tutorial

Do not forget to split the STL solid into multiple parts with `surfaceFeatureEdges` in order to improve the edge reinfements made by cfMesh.



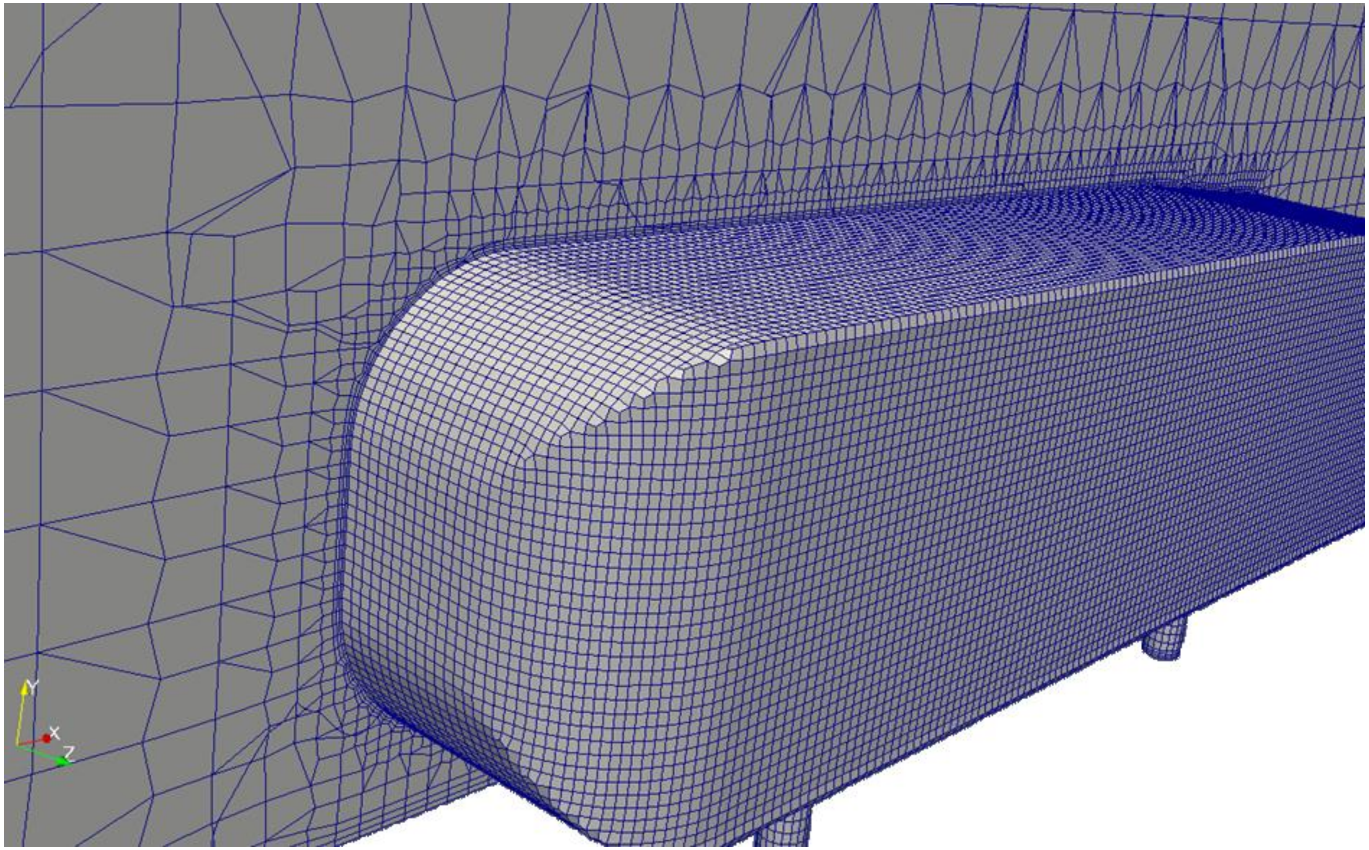
Ahmed body tutorial

It is also necessary do add the bounding box surfaces by using `surfaceGenerateBoundingBox` for the external aero-dynamics



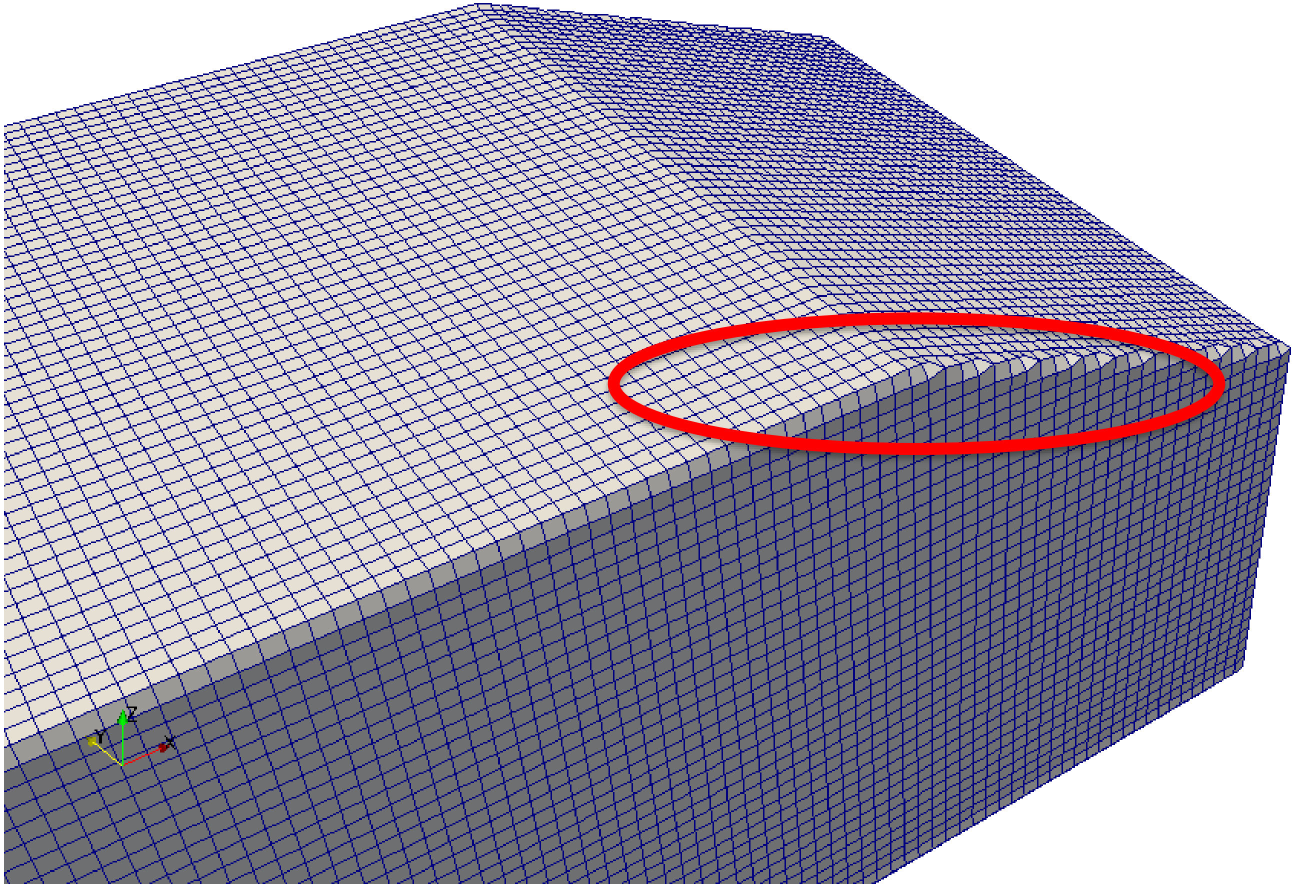
Ahmed body tutorial

Our tutorial performs boundary layer refinement close to the Ahmed body walls. The choice of the depth and the number of refinements of the boundary layer refinement depends on the CFD model to be performed.



Ahmed body tutorial

However, in this case it is difficult to perform a precise body-fitted mesh since the Ahmed body surface can not be completely splitted in all the faces that compose the main body. This affects the quality of the edges as showed in this slide.



Ahmed body tutorial

To solve this kind of problem it is recommended to work with a fms surface file instead of a standard stl geometry.
Let us take a look at the content of a fms file:

```
16
(
OpenSCAD_Model_0
empty

OpenSCAD_Model_1
empty

OpenSCAD_Model_2
empty
…

zMin_14
empty

zMax_15
empty
)
```

The first section of the fms file contains the declaration of the patch number, name and type.

Please note that the order of the patches declaration is important and must be consistent with the following sections of the file.

Ahmed body tutorial

To solve this kind of problem it is recommended to work with a fms surface file instead of a standard stl geometry.
Let us take a look at the content of a fms file:

```
920
(
(0 -0.1945 0.05)
(0 0.1945 0.05)
(0 -0.1945 0.262072)
(0 0.1945 0.262072)
(-0.358428 -0.157113 0)
(-0.358428 0.157113 0)
(-0.358053 -0.157978 0)
(-0.358053 0.157978 0)
(-0.358855 -0.156274 0)
(-0.358855 0.156274 0)
(-0.357734 -0.158865 0)
…
(6 -1.1945 0)
(6 1.1945 0)
(6 1.1945 2.338)
(6 -1.1945 2.338)
)
```

In the second section we find the list of the surface mesh points (vertex of the triangles).

The points coordinates correspond with the original stl triangles definition. That is, the fms file contains the same triangular surface mesh of the original stl file.

Ahmed body tutorial

To solve this kind of problem it is recommended to work with a fms surface file instead of a standard stl geometry.
Let us take a look at the content of a fms file:

```
1832
(
((873 888 862) 0)
((862 888 883) 0)
((862 883 889) 0)
((862 889 874) 0)
((888 873 871) 0)
((888 871 892) 0)
((873 874 872) 0)
((873 872 871) 0)
…
((913 918 917) 13)
((913 914 918) 13)
((912 913 917) 14)
((912 917 916) 14)
((915 918 914) 15)
((915 919 918) 15)
)
```

The third section is composed by the declaration of the triangles. The triangles are identified by the vertexes positional identifier as defined in the previous section.

In addition, each triangle is accompanied with the patch identification number.

Please note that the order of the patches and vertexes affects the triangle definitions.

Ahmed body tutorial

To solve this kind of problem it is recommended to work with a fms surface file instead of a standard stl geometry.
Let us take a look at the content of a fms file:

```
918
(
(0 1)
(0 2)
(0 807)
(1 3)
(1 806)
(2 3)
(2 404)
…
(916 917)
(916 919)
(917 918)
(918 919)
)

0()
0()
```

The fourth section is optional and is composed by the declaration edges.

The edges are identified by the triangles vertexes ID.

The last two sections are optional and define subsets of points and facets.

Ahmed body tutorial

The fms surface can be easily generated starting from an existing stl (in our case *constant/triSurface/abb.stl*) by using the cfMesh conversion utility `surfaceToFMS`.

```
$> surfaceToFMS <inputFile.stl>
```

After having converted the *constant/triSurface/abb.stl* geometry the new file will be stored in the same directory.

The fms geometry file can contain patch names definition, the triangles vertex coordinates, the declaration of the lines connecting the vertexes and geometrys edges marked for refinement.

You can inspect the feature edges by using:

```
$>  FMSToSurface <input fms> <surface file> -exportFeatureEdges
```

where -exportFeatureEdges writes feature edges in a vtk file

Ahmed body tutorial

Another option for generating fms geometries is to use again the *surfaceFeatureEdge* utility by adopting the .fms extension in the output file name.

In this manner we will generate a fms file that includes the geometry edges information. In our case, we can apply this method by using the splitted *constant/triSurface/abb.stl* file as input as follows:

```
$> surfaceFeatureEdges -angle 1 constant/triSurface/abb.stl
                    constant/triSurface/abb.fms
```

We can inspect the feature edges by using:

```
$> FMSToSurface <inputFile.fms> <surface file name>
                -exportFeatureEdges
```

where -exportFeatureEdges writes feature edges in a vtk file.

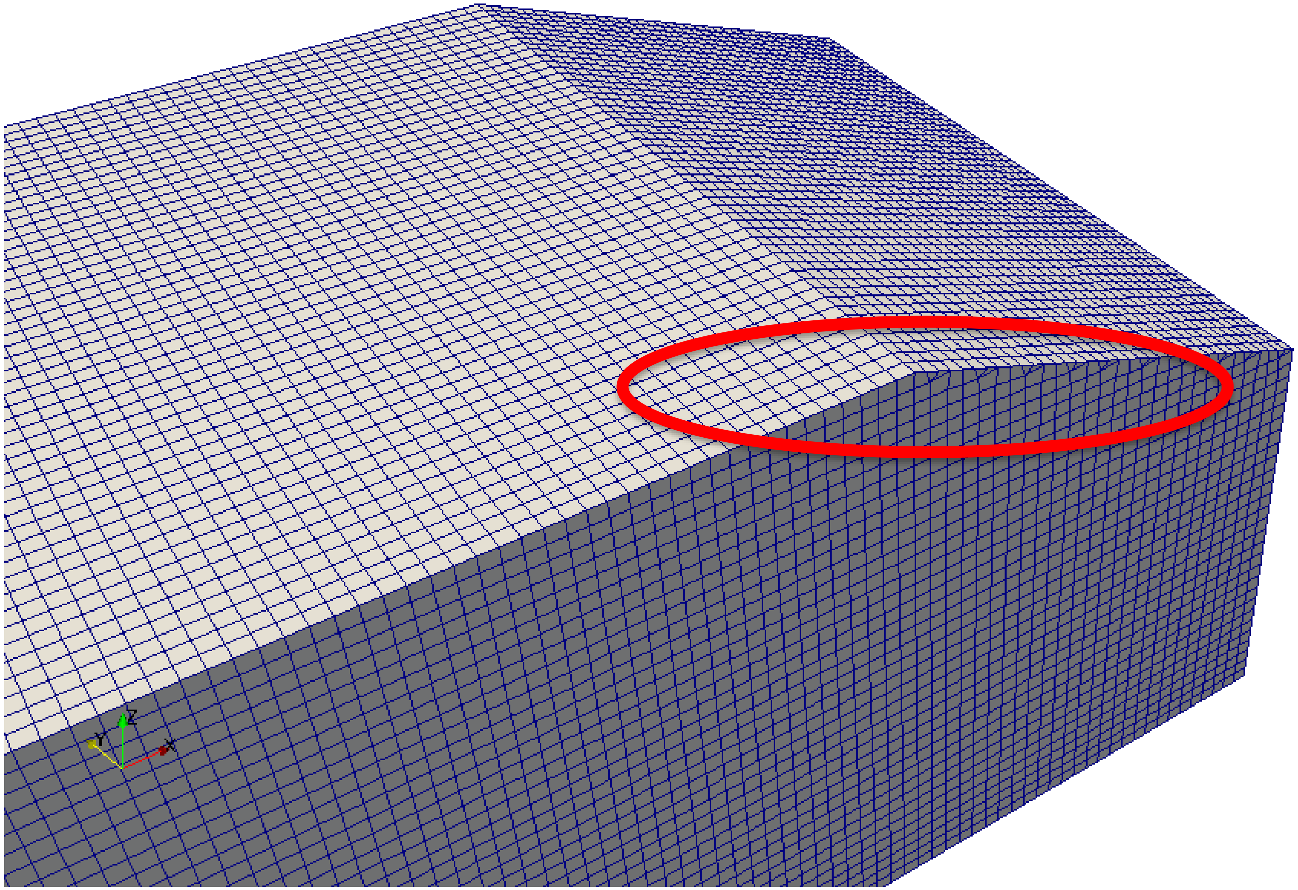
Ahmed body tutorial

To generate the mesh starting from the fms surface file we modify our original workflow in the following manner:

1. `$> foamCleanTutorials`
2. `$> cp system/meshDict.stl system/meshDict`
3. `$> surfaceGenerateBoundingBox constant/triSurface/abscale.stl abb.stl 4 6 1 1 0 2`
4. `$> surfaceFeatureEdges -angle 80  abb.stl constant/triSurface/abb.stl`
5. `$> surfaceFeatureEdges -angle 1 constant/triSurface/abb.stl constant/triSurface/abb.fms`
6. `$> cp system/meshDict.fms system/meshDict`
7. `$> cartesianMesh`

Ahmed body tutorial

The result is an improved edges surface mesh. This is particularly evident when looking at the 90° angles of the object main body.



Ahmed body tutorial

In our tutorial we included a steady RANS simulation setup for a fully turbulent case $Re=10^6$ to be solved with a SST k-Omega model.

In this case, we want to apply a boundary layer refinement that results in a maxium wall unit value $y^+ < 300$ in the Ahmed body surface.

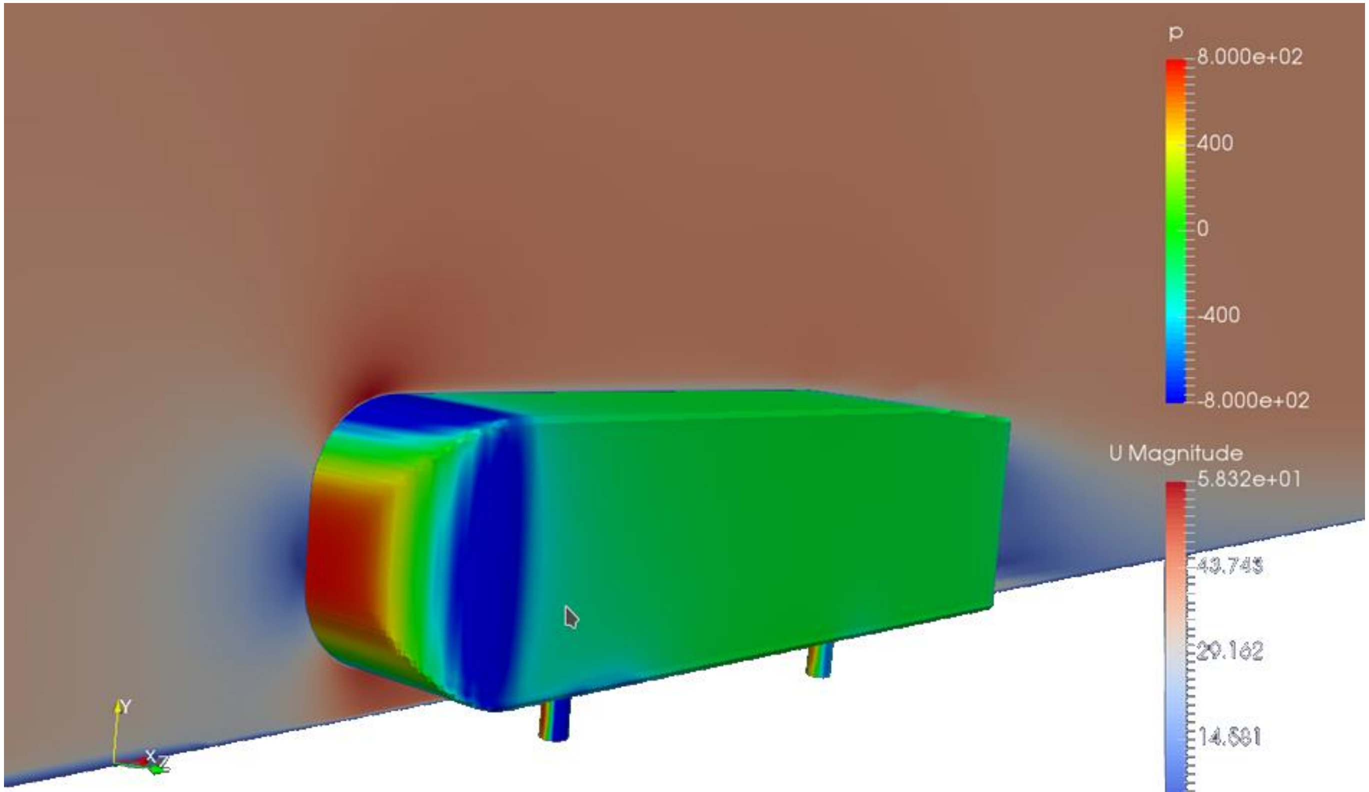
Topics like the boundary layer refinement, turbulence modelling and parallel processing will be addressed in a specific module.

In the terminal window type:

1. `$> decomposePar`
2. `$> mpirun -np 4 simpleFoam -parallel > log.yPlus`
3. `$> reconstructPar -latestTime`
4. `$> rm -rf processor*`
5. `$> simpleFoam -postProcess -func yPlus > log.yPlus`

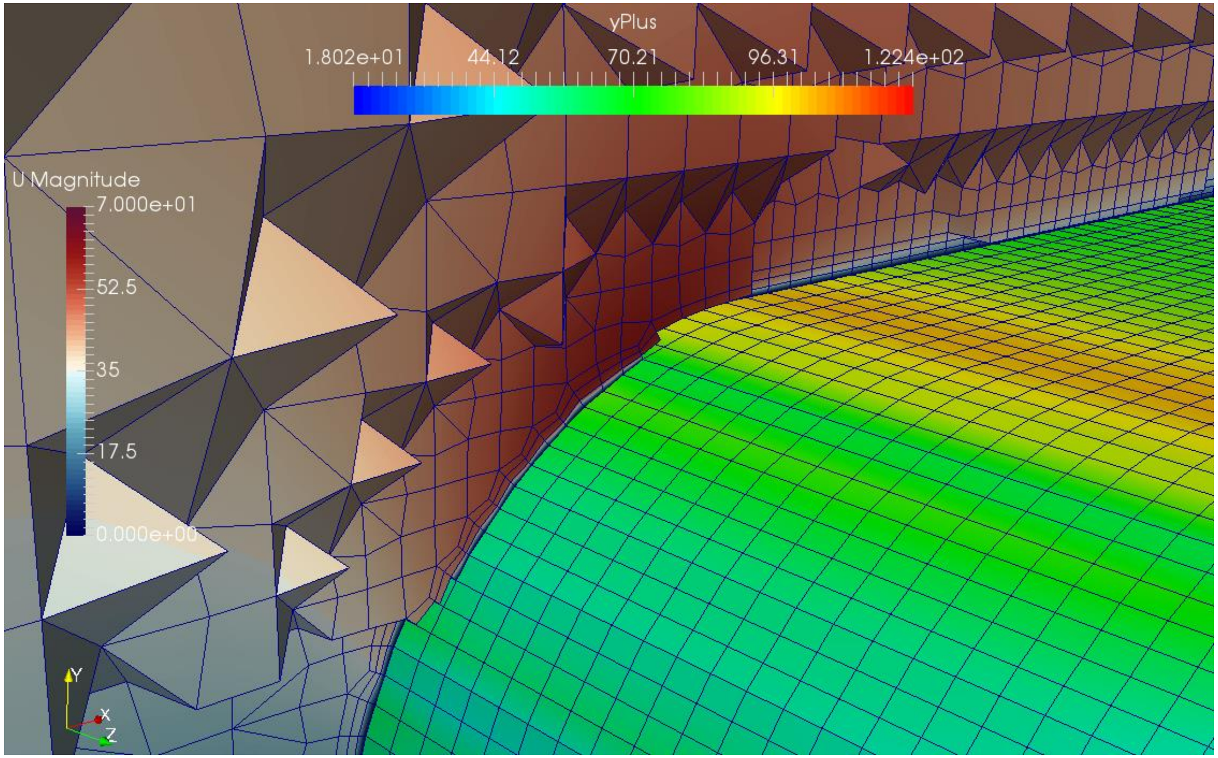
Ahmed body tutorial

The velocity magnitude and the pressure fields can be easily represented in paraview.



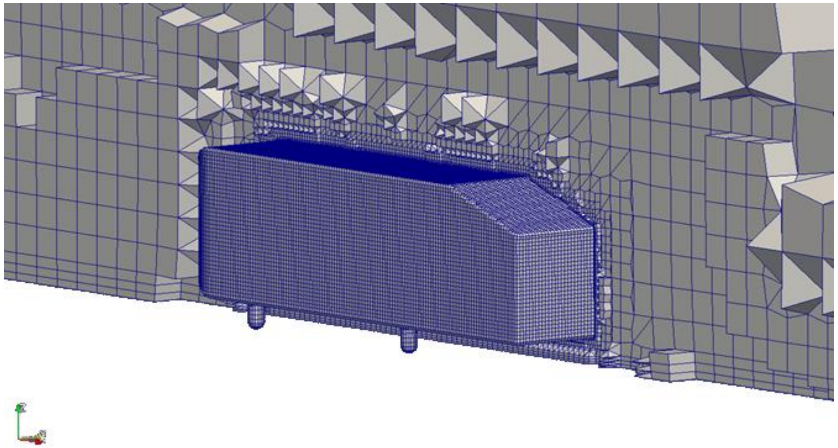
Ahmed body tutorial

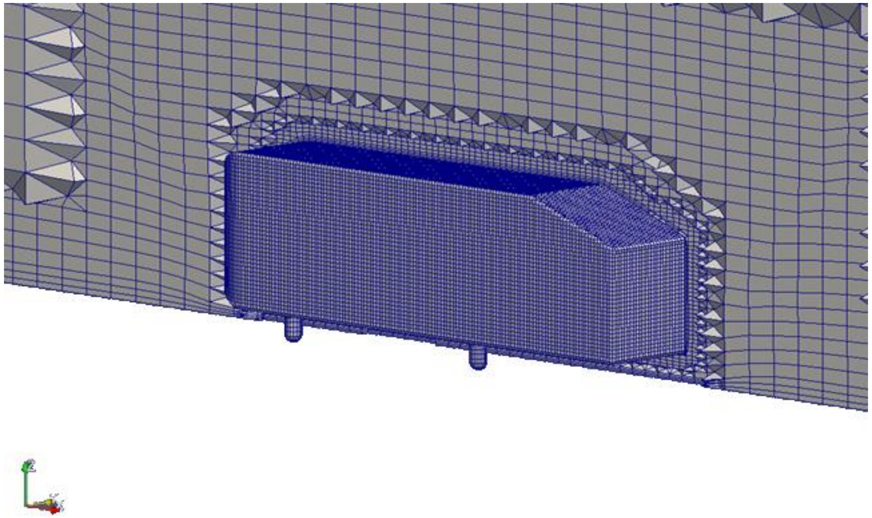
The execution of the RANS simulation included in this tutorial will provide a maximum $y^+ = 122$ around the Ahmed body surface. We suggest to try different mesh refinements and to check the effects on the wall unit.



Ahmed body tutorial

- You will find the tutorial in the **`CFMESH/c5_ahmed`** folder.
- The cfMesh case is accompanied by a `blockMesh + snappyHexMesh` set-up. Both the set-ups share a similar number of cell and refiniment level.
- Try to compare the mesh quality and the execution time.





Roadmap

1. ~~Mesh quality assessment in CFD~~
2. ~~Mesh generation using cfMesh~~
3. ~~The cylinder tutorial~~
4. ~~The 2D airfoil tutorial~~
5. ~~The static mixer tutorial~~
6. ~~The Ahmed body tutorial~~
7. **The mixing elbow comparison**
8. The moving quadcopter tutorial

The mixing elbow comparison

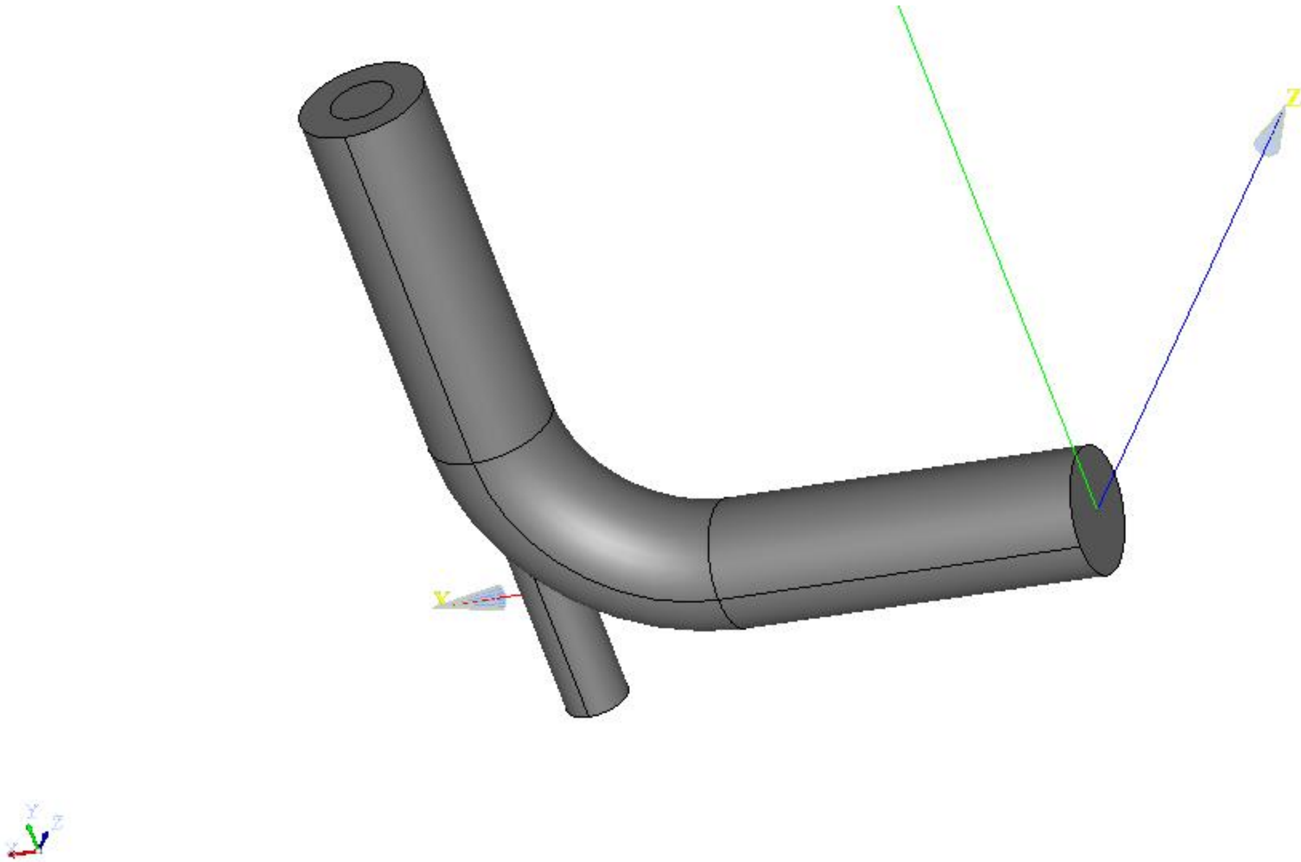
- Meshing with cfMesh.
- Meshing tutorial 6. Comparison of different open source meshing techniques

`$TM/CFMESH/c6_mixingElbow`

- From this point on, please follow me.
- We are all going to work at the same pace.
- Remember, `$TM` is pointing to the path where you unpacked the tutorials.

The mixing elbow comparison

We start from the mixing elbow geometry file created during the solid modelling lessons.



The mixing elbow comparison

In this tutorial we are going to mesh the same geometry with 5 mesh generators

1. Tetrahedral NETGEN, tetra-dominant mesher from salome
2. `blockMesh` + `snappyHexMesh`, the combination of a background mesh plus castellation, snap and layer addition
3. `cartesianMesh`, hexa-dominant mesher from cfMesh
4. `tetMesh`, tetra-dominant mesher from cfMesh
5. `polyDualMesh`, conversion of a tetrahedral mesh to its dual mesh

The mixing elbow comparison

So, go into the main folder

```
$> cd $TM/CFMESH/c6_mixingElbow
```

Choose a sub-folder according to the mesher you want to test and run

```
$> foamCleanTutorials
```

to clean the case.

The mixing elbow comparison

The first case is the salome tetra mesh, you can open salome and view the *mixingElbow.hdf* project. Then export the .unv file in OpenFOAM with

```
$> ideasUnvToFoam mixingElbow.unv
```

To compute a `snappyHexMesh` mesh, go to the **`snappy`** directory and type:

```
$> blockMesh
$> snappyHexMesh -overwrite -noFunctionObjects
```

To compute a cartesian mesh, go to the **`cartestianMesh`** directory and type:

```
$> cartesianMesh
```

To compute a tetrahedral mesh, go to the **`tetMesh`** directory and type:

```
$> tetMesh
```

It is important to note that `tetMesh` is part of the **cfMesh** suite and it uses the *`system/meshDict`* in the same way of `cartesianMesh`. The content of this dictionary remains the same.

The mixing elbow comparison

Lastly, we compute a polyhedral mesh starting from an existing tetrahedral grid. The strategy of `polyDualMesh` is to compute the dual elements of the base tetrahedral grid in order to obtain cell elements characterized by multiple faces.

To start with this case we can use a tetrahedral mesh generated by means of `tetMesh` or Salome.

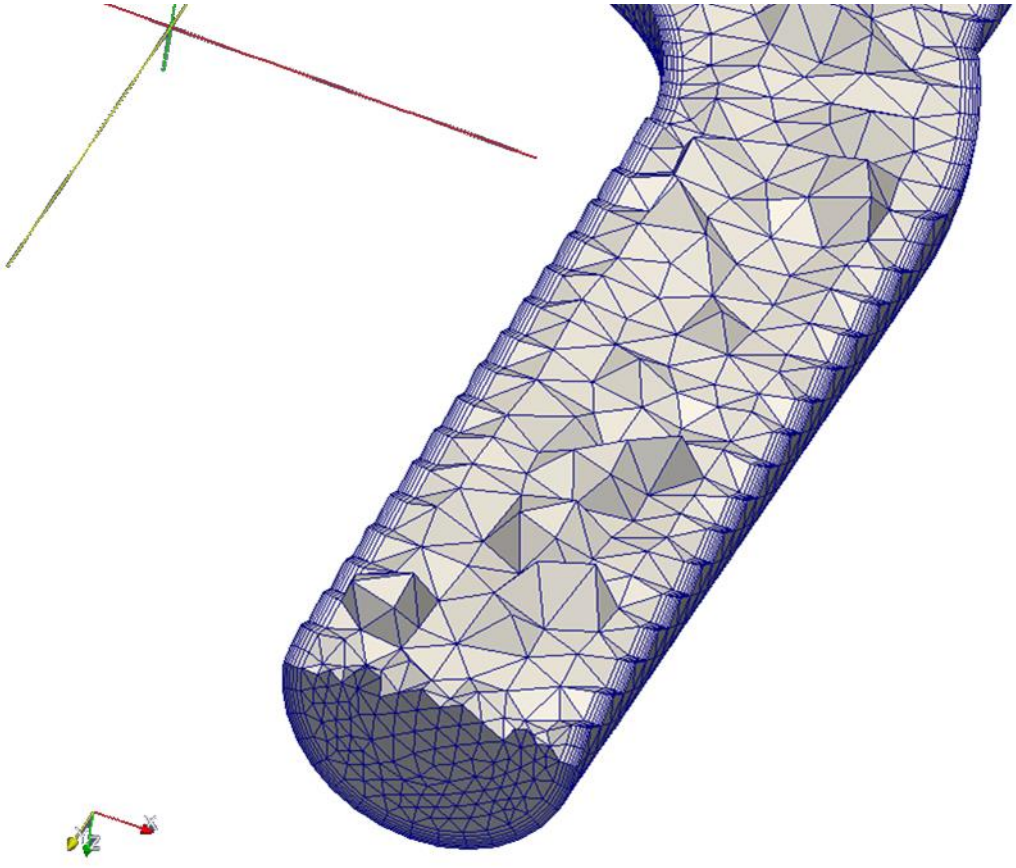
Go to the **polyDual** directory and type:

```
$> polyDualMesh
```

The mixing elbow comparison

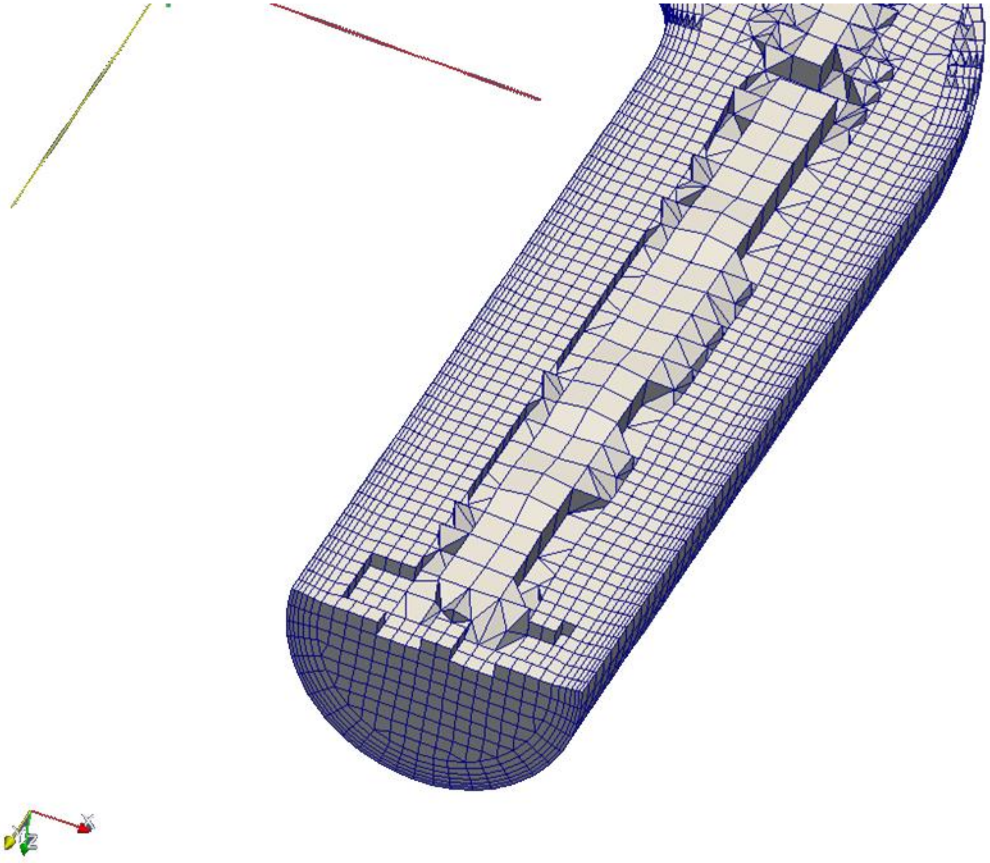
NETGEN

By following correctly the instructions you should have get something like this



BlockMesh + SnappyHexMesh

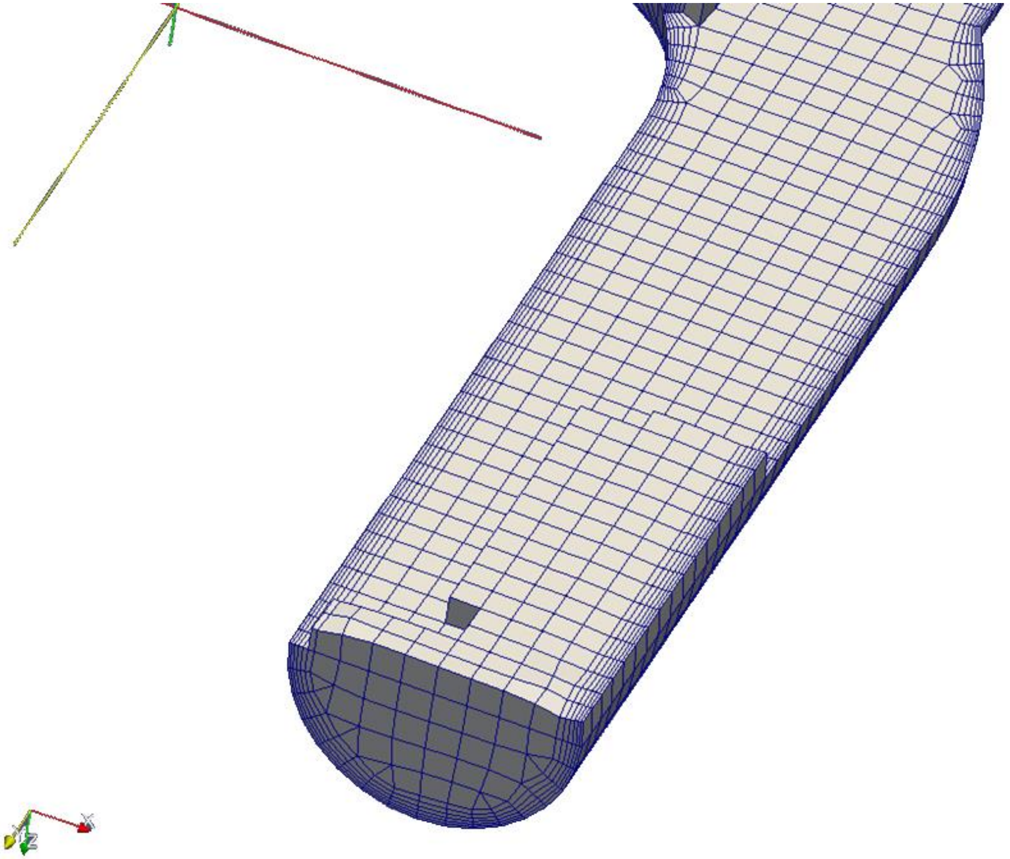
By following correctly the instructions you should have get something like this



The mixing elbow comparison

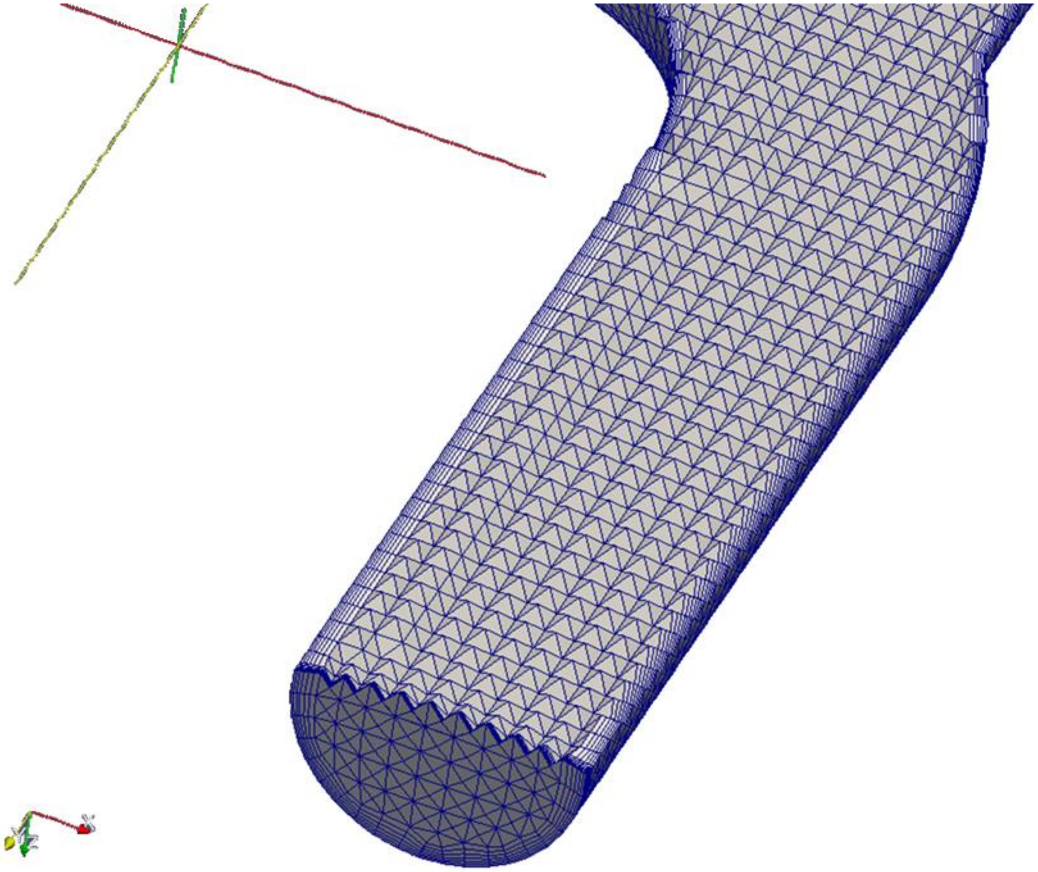
cfMesh - cartesian

By following correctly the instructions you should have get something like this



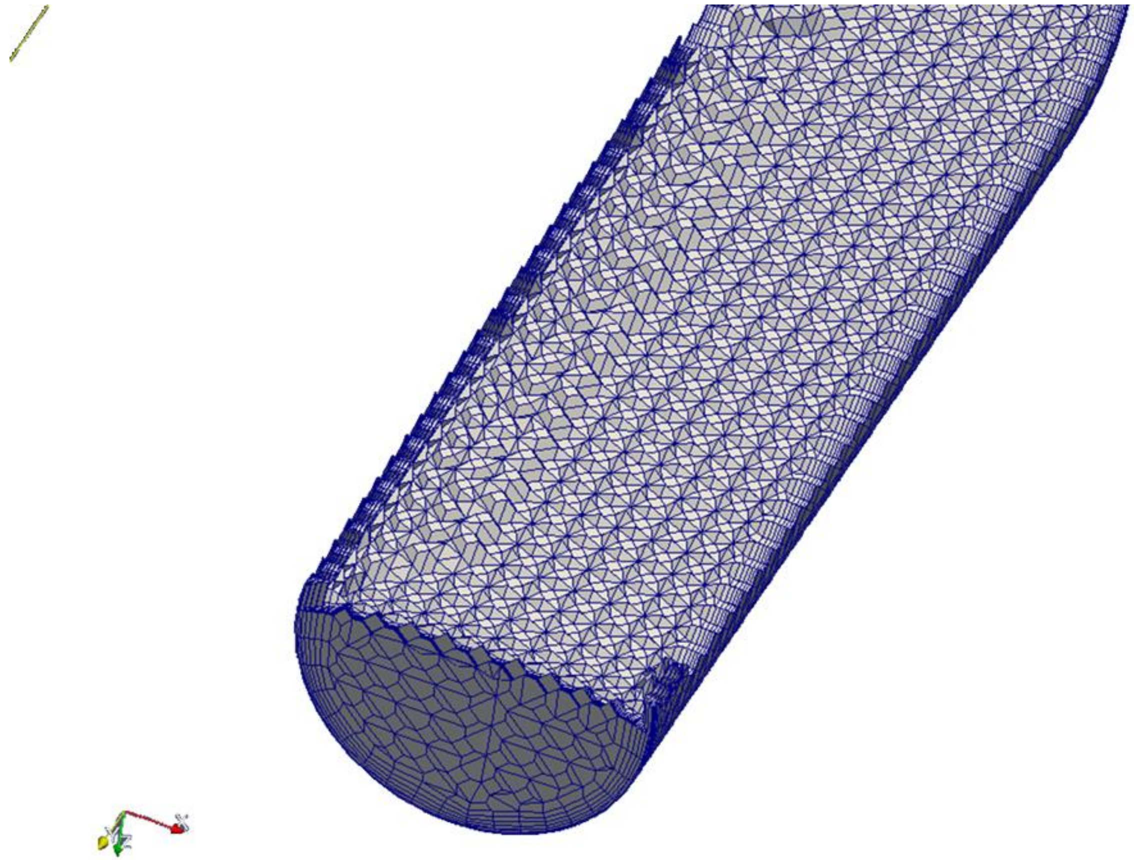
cfMesh - tetrahedral

By following correctly the instructions you should have get something like this



cfMesh - polyhedral

By following correctly the instructions you should have get something like this



The mixing elbow comparison

Meshes elements type budget

	NETGEN (Salome)	blockMesh + snappyHM	cfMesh cartesian	cfMesh tetrahedral	polyDualMesh
Hexahedrons	-	89%	98.8%	2%	29%
Polyhedron	-	8%	-	-	71%
Wedges	-	-	-	-	-
Prisms	66%	3%	0.2%	42%	-
Pyramids	-	-	0.4%	-	-
Tetrahedrons	33%	-	0.3%	55%	
Tet-wedges	-	-	-	-	-

The mixing elbow comparison

Meshes quality

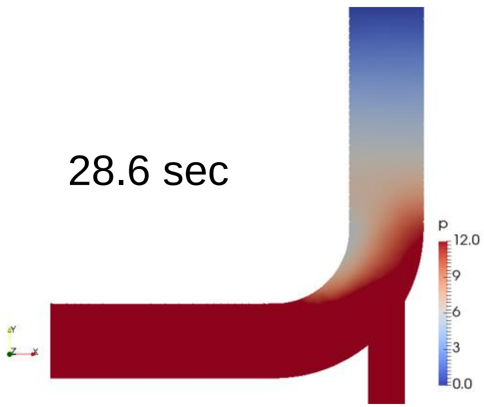
	Number of cells	Non-orthogonality	Max. skewness	Execution time (sec)
`NETGEN (Salome)`	54k	Max. 72 Avg. 12.1	3.3	3
`blockMesh + snappyHM`	55k	Max. 61 Avg.7.1	1.6	5
`cfMesh cartesian`	48k	Max 59 Avg. 6.7	1.6	3
`cfMesh tetrahedral`	158k	Max. 69 Avg. 16	1.7	8
`polyDualMesh`	55k	Max. 76 Avg. 15	2.2	3 (in addition to tetMesh)

The mixing elbow comparison

Pressure fields after 0.5 sec (icoFoam)

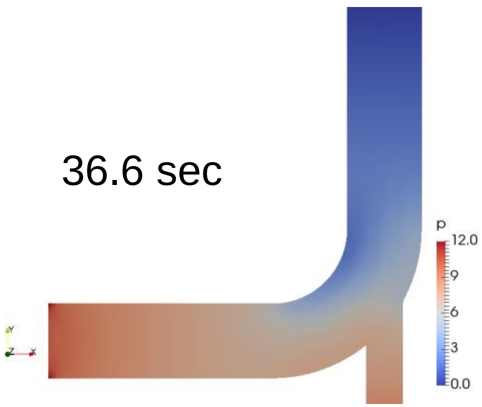
NETGEN

28.6 sec



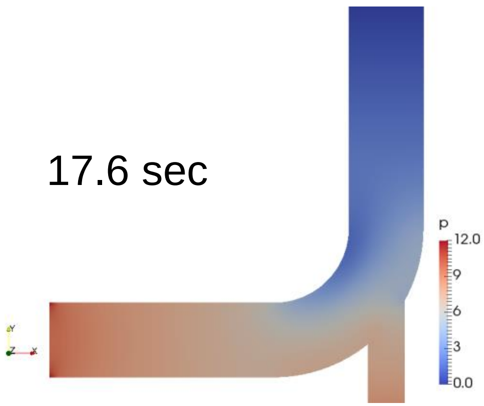
blockMesh + snappyHexMesh

36.6 sec



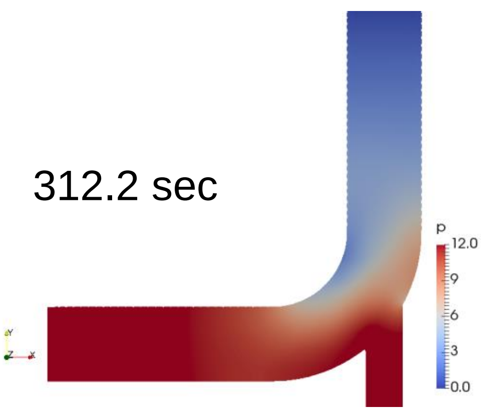
cartesianMesh

17.6 sec



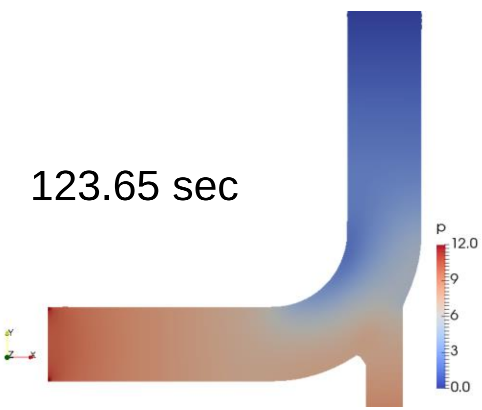
tetMesh

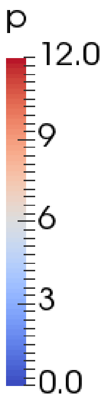
312.2 sec



polyDualMesh

123.65 sec



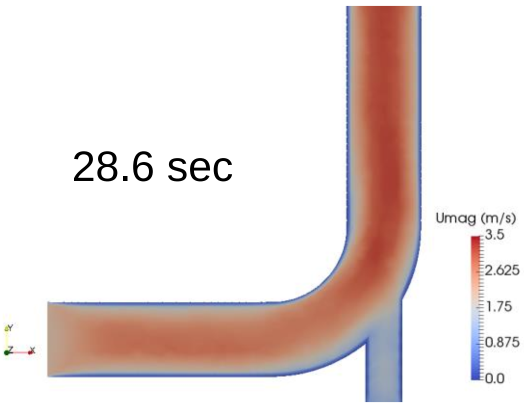


The mixing elbow comparison

Velocity fields after 0.5 sec (icoFoam)

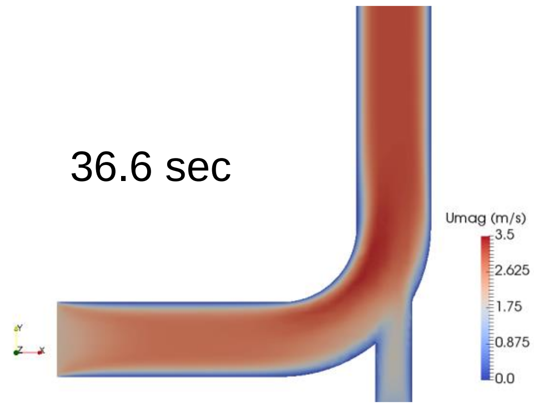
`NETGEN`

28.6 sec



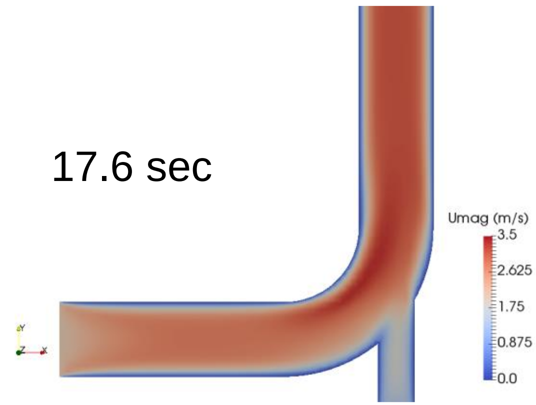
`blockMesh + snappyHexMesh`

36.6 sec



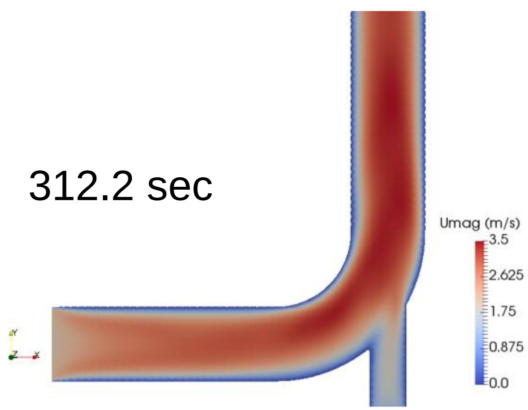
`cartesianMesh`

17.6 sec



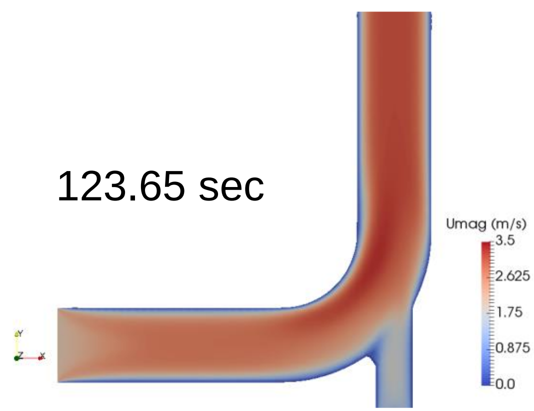
`tetMesh`

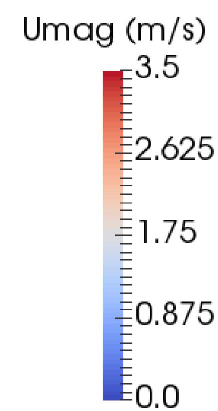
312.2 sec



`polyDualMesh`

123.65 sec





The mixing elbow comparison

cfMesh comes with an additional automatic mesher that performs a 3D spatial discretization using polyhedral cells: `pMesh`.

Similarly to `tetMesh`, the dictionary that governs the computation of the polyhedral grid is always *`system/meshDict`* and the settings are the same illustrated for `cartesianMesh`.

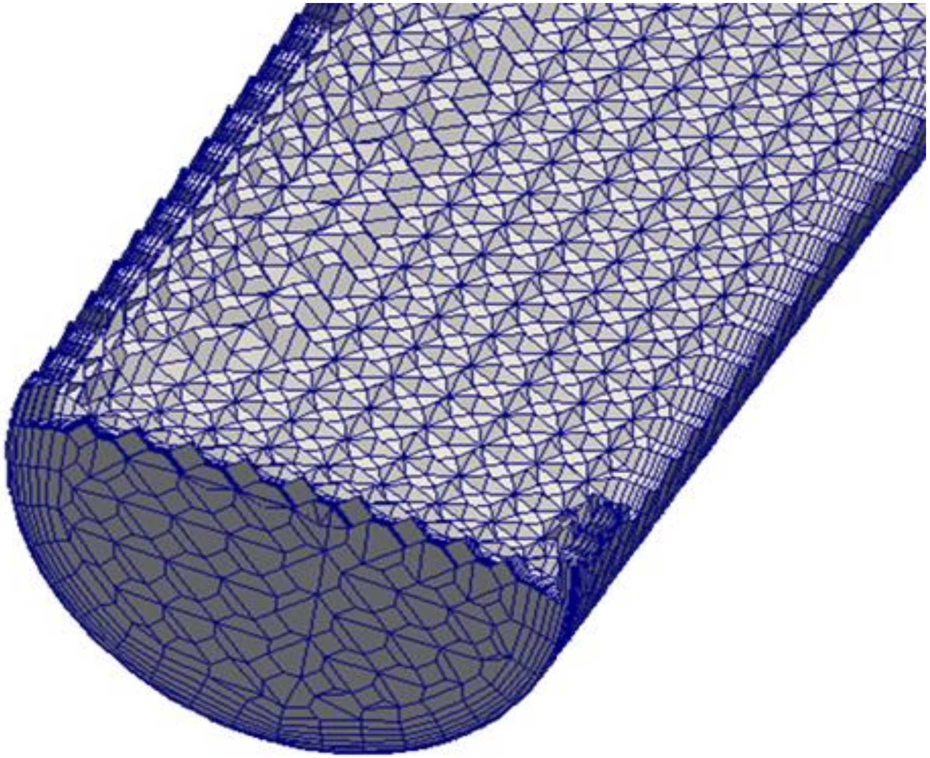
Go to the **`$TM/CFMESH/c6_mixingElbow/pMesh`** sub-folder to test the performance of `pMesh` against the other meshing tools showed in this tutorial.

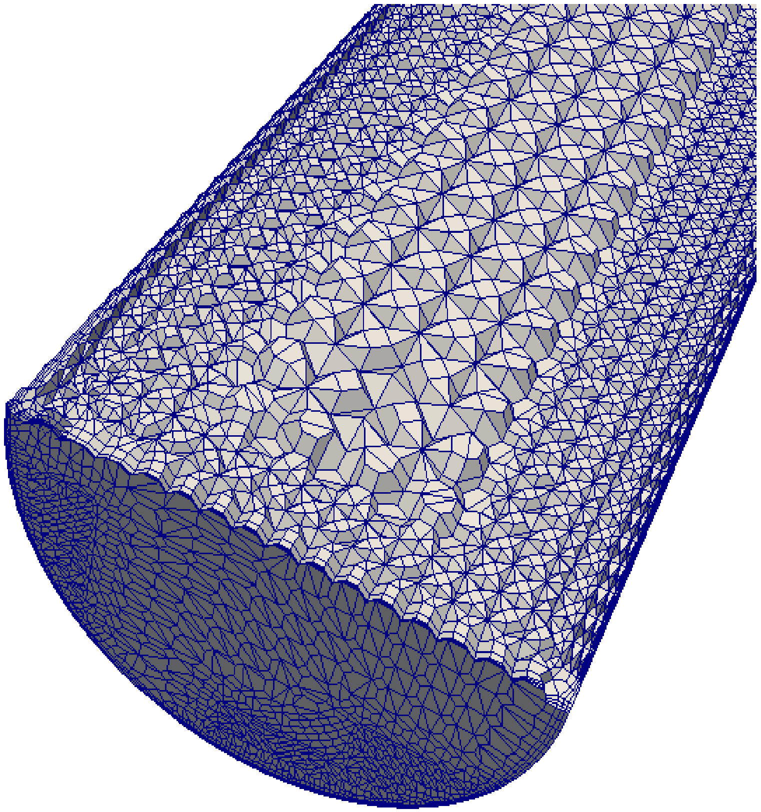
In the following slide we show a brief comparison between the result of different `pMesh` settings and `polyDualMesh`.

The mixing elbow comparison

polyDualMesh

pMesh





N. Of cells: 55k
Mesh generation time: 11 sec
Quality check failures: 5 non-ortho faces

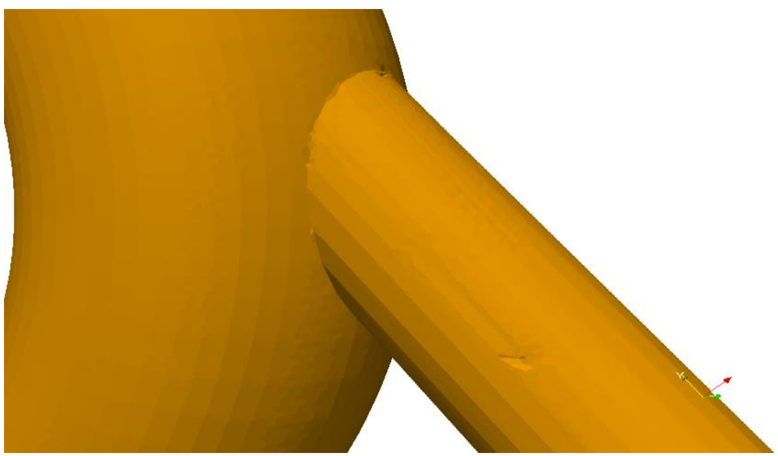
N. Of cells: 270k
Mesh generation time: 606 sec
Quality check failures: 163 non-ortho faces

The mixing elbow comparison

Several attempts to perform a polyhedral mesh by means of the `pMesh` utility showed poor performance with respect to `polyDualMesh` in terms of execution time, geometrical quality metrics and body-fitting.

The largest part of the mesh computation time is due to the iterative process of the boundary meshing iterations that has to balance the overall mesh quality with the goodness of the geometry fitting.

N. Of cells: 270k
Mesh generation time: 606 sec
Boundary cells size: 0.1 m

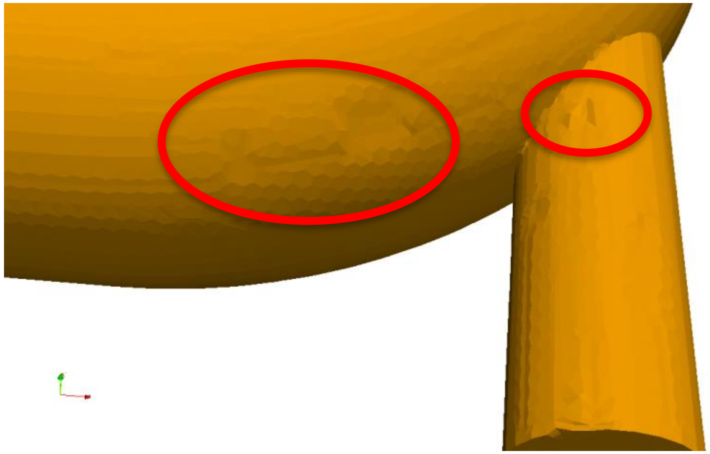


N. Of cells: 440k
Mesh generation time: 204 sec
Boundary cells size: 0.075 m

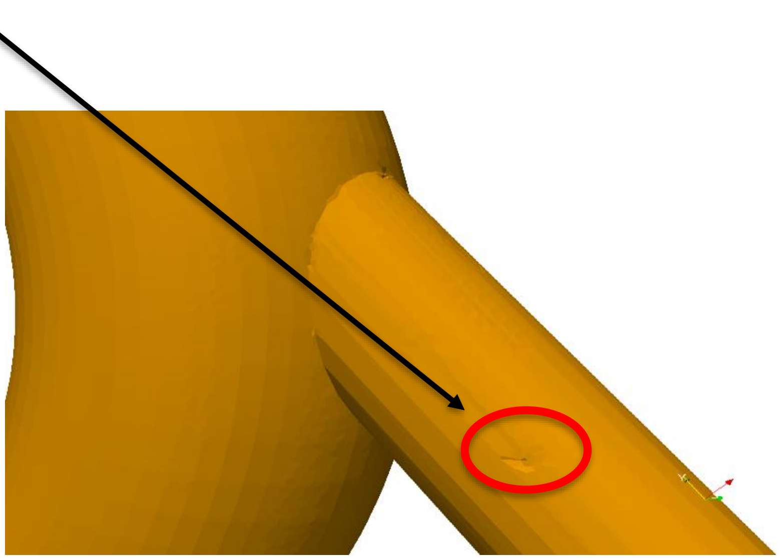
The mixing elbow comparison

To improve the quality of the surface elements computed by the current version of `pMesh` (**cfMesh** release 1.1.2), it may be necessary to reduce the size of the local refinement with the downside of increasing the total number of cells.

Yet, the result is not always satisfactory.



N. Of cells: 270k
Mesh generation time: 606 sec
Boundary cells size: 0.1 m



N. Of cells: 440k
Mesh generation time: 204 sec
Boundary cells size: 0.075 m

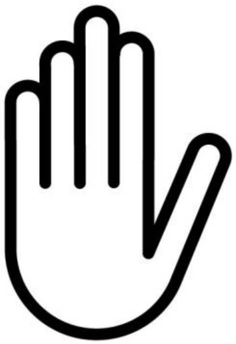
Roadmap

1. ~~Mesh quality assessment in CFD~~
2. ~~Mesh generation using cfMesh~~
3. ~~The cylinder tutorial~~
4. ~~The 2D airfoil tutorial~~
5. ~~The static mixer tutorial~~
6. ~~The Ahmed body tutorial~~
7. ~~The mixing elbow comparison~~
8. **The moving quadcopter tutorial**

Moving quad tutorial – dynamic mesh

- Meshing with cfMesh.
- Meshing tutorial 7. The moving quadcopter tutorial (dynamic mesh)

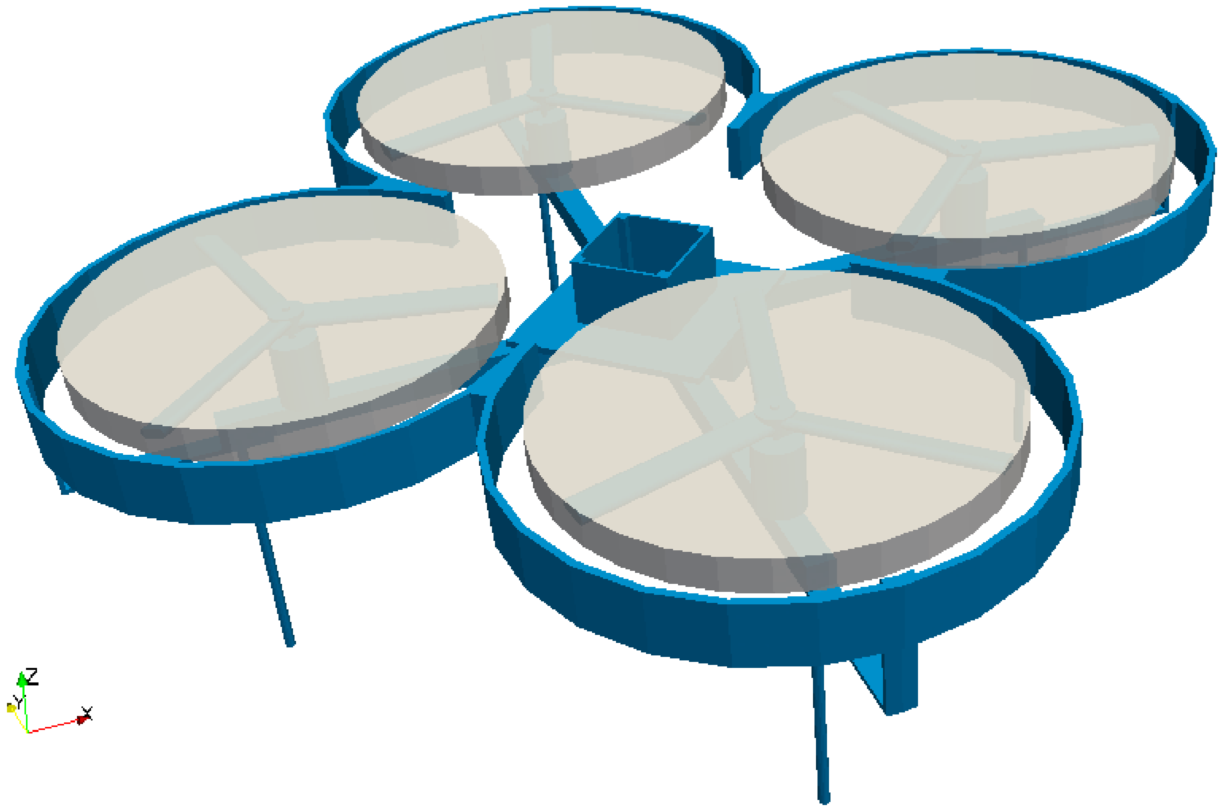
`$TM/CFMESH/c7_quad`



- From this point on, please follow me.
- We are all going to work at the same pace.
- Remember, `$TM` is pointing to the path where you unpacked the tutorials.

Moving quad tutorial – dynamic mesh

The result of the tutorial is a dynamic mesh that looks like this:



Moving quad tutorial – dynamic mesh

The quadcopter model is composed by four different rotors that spins around their rotation axis.

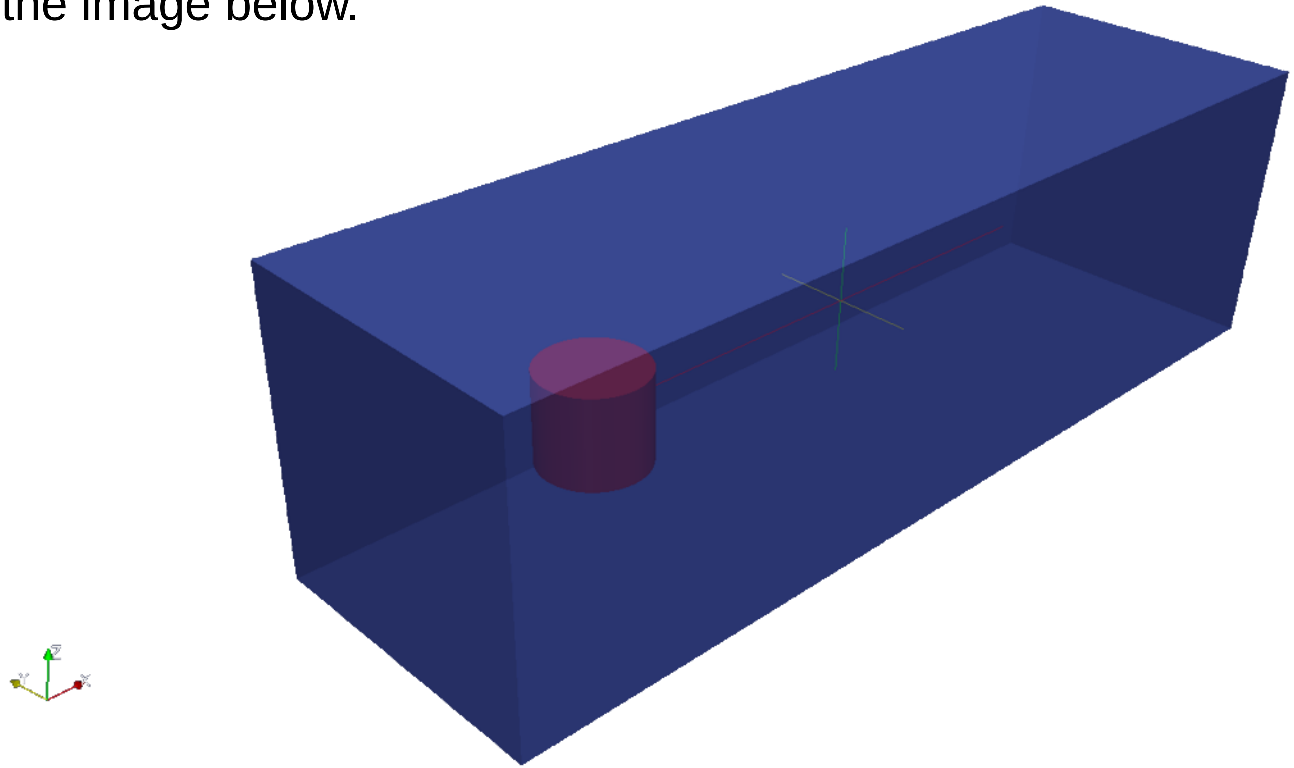
OpenFOAMOpenFOAM® has the capability to handle this problem in two ways:

1) sliding interfaces, i.e. one part of the mesh will move effectively with respect to the other one,

2) moving (or multiple) reference frame (MRF), i.e. the mesh will not move and the rotation will be simulated thorugh the addition of a volume force representing both centripetal and Coriolis forces.

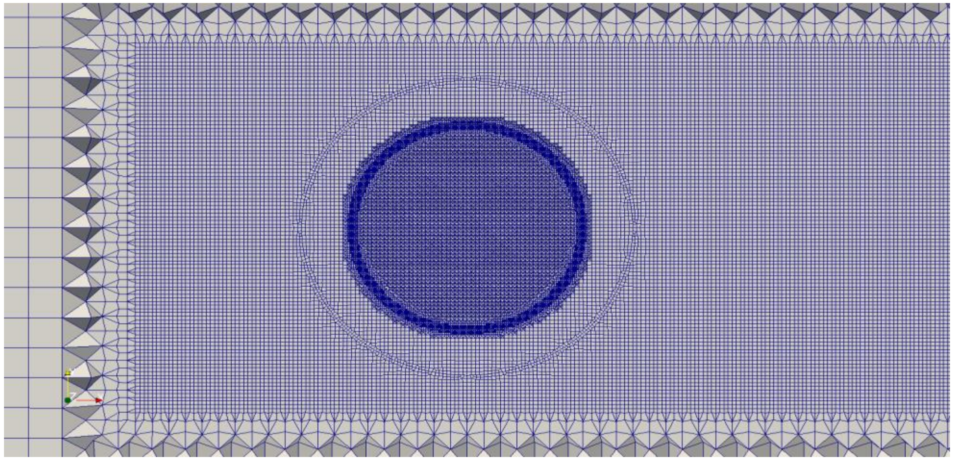
Moving quad tutorial – dynamic mesh

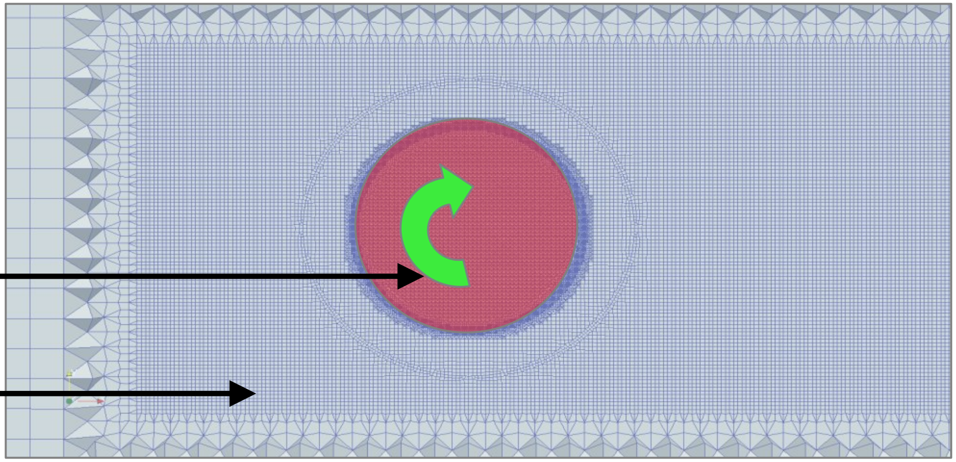
- The mesh that we are going to produce is suitable for both simulations.
- The strategy is to subdivide the domain into the fixed spatial grid and four different moving (dynamic) parts around the rotor regions.
- The overall mesh will be composed by the sum of the separate parts, similarly to the image below.



Moving quad tutorial – dynamic mesh

- The images on the right show the simple case of a mesh with a single dynamic region.
- The fixed and the rotating meshes will be computed separately in different case folders.
- The overall mesh is obtained by means of the `mergeMeshes` utility.





Moving quad tutorial – dynamic mesh

You will find this tutorial in the **CFMESH/c7_quad** folder.

In the terminal window type:

1. $> cd **CFMESH/c7_moving_quad**
2. $> ls

You will see 5 folders:

- **rotorXmax**
- **rotorXmin**
- **rotorYmax**
- **rotorYmin**
- **total**

where the first 4 folders contain the independent meshes for all rotors, while the **total** folder contains the mesh for the main body.
Later, it will be update to contain the sum of all 5 parts.

Moving quad tutorial – dynamic mesh

The different mesh parts must be created separately as follows:

```
1.  $> cd total/
2.  $> foamCleanTutorials
3.  $> cartesianMesh
4.  $> cd ../rotorXmin/
5.  $> foamCleanTutorials
6.  $> cartesianMesh
```

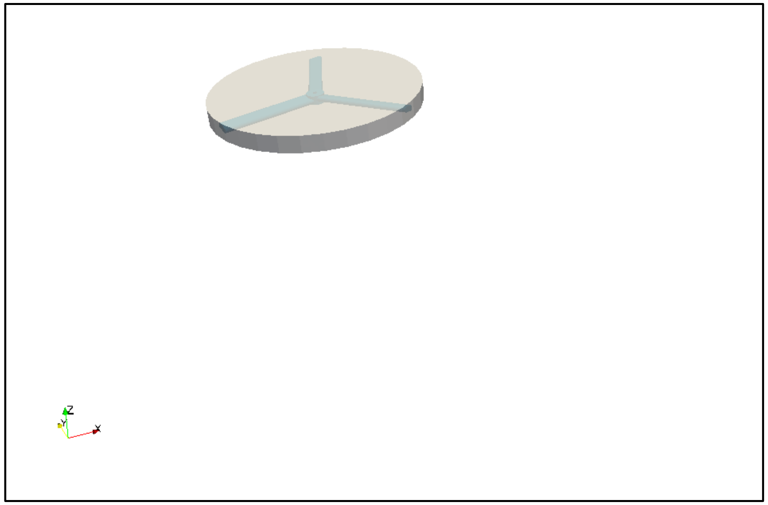
Moving quad tutorial – dynamic mesh

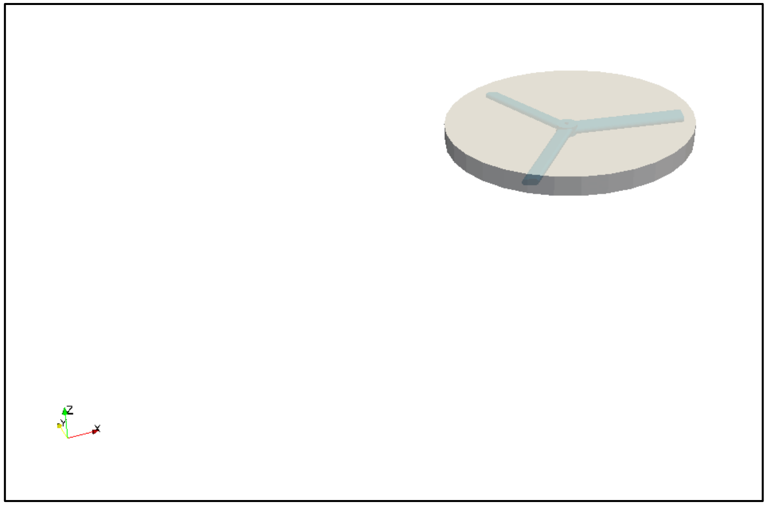
The different mesh parts must be created separately as follows:

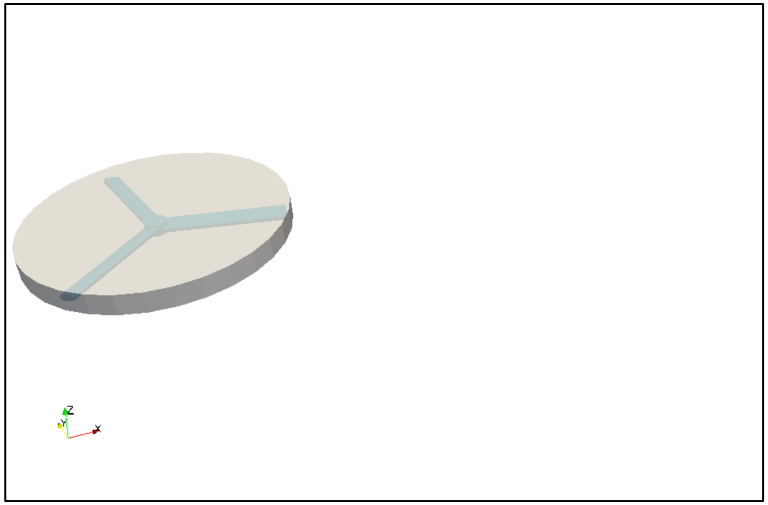
7. $> cd ../**rotorXmax/**
8. $> foamCleanTutorials
9. $> cartesianMesh
10. $> cd ../**rotor/Ymin/**
11. $> foamCleanTutorials
12. $> cartesianMesh
13. $> cd ../**rotorYmax/**
14. $> foamCleanTutorials
15. $> cartesianMesh

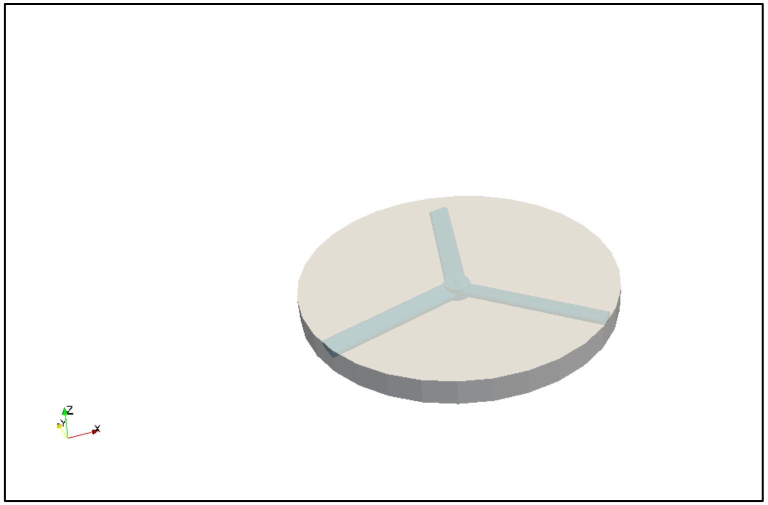
Moving quad tutorial – dynamic mesh

This is are the patches of the four rotor sub-meshes visaulized separately in Paraview.









Moving quad tutorial – dynamic mesh

Now go back to the **`total`** folder and merge all the meshes

```
1.  $> cd ../total/
2.  $> mergeMeshes -overwrite . ../rotorXmin/
3.  $> mergeMeshes -overwrite . ../rotorXmax/
4.  $> mergeMeshes -overwrite . ../rotorYmin/
5.  $> mergeMeshes -overwrite . ../rotorYmax/
```

It is important to note that the interface elements of the various merged parts are not conformal and disconnected.

The approach proposed in this tutorial takes advantage of an advanced method to deal with non-conformal patches in order to create a working dynamic mesh: the Arbitrary Mesh Interface.

Moving quad tutorial – dynamic mesh

The Arbitrary Mesh Interface (AMI) s a technique that allows simulation across disconnected, but adjacent, mesh domains. The domains can be stationary or move relative to one another.

By default, AMI operates by projecting one of the patches' geometry onto the other.

For the quadcopter tutorial, we create the AMI interface between the two sides of each interface cylinder.

This operation can be done by hand or by using the utility `changeDictionary`, which needs the changeDictionaryDict file in the **`system`** folder.

6. `$> gedit system/changeDictionaryDict`
7. `$> changeDictionary`

Moving quad tutorial – dynamic mesh

The **`system/changeDictionaryDict`** dictionary has the following structure:

```
dictionaryReplacement
{
    boundary
    {
        cylinderXmaxOuter
        {
            type           cyclicAMI;
            neighbourPatch  cylinderXmaxInner;
            transform       noOrdering;
        }
        ...
        cylinderXmaxInner
        {
            type           cyclicAMI;
            neighbourPatch  cylinderXmaxOuter;
            transform       noOrdering;
        }
        ...
    }
}
```

Connectivity information

Connectivity information

In our case we specify the boundaries that must be connected and the patch type (in our case **cyclicAMI**).

Moving quad tutorial – dynamic mesh

Now let us run the `checkMesh` utility in order to check the quality of the mesh and the number of unconnected regions.
This operation will also create the sets that we need to build the dynamic mesh

8. **`$> checkMesh`**

To create zones from the `checkMesh` sets:

9. `$> setsToZone`

The rotation coefficients are set in the **`constant/dynamicMeshDict`** file.

10. `$> gedit` **`constant/dynamicMeshDict`**

Moving quad tutorial – dynamic mesh

The **`constant/dynamicMeshDict`** dictionary has the following structure:

```
...
multiSolidBodyMotionFvMeshCoeffs
{
        region1 //Xmin
        {
                solidBodyMotionFunction rotatingMotion;
                rotatingMotionCoeffs
                {
                        origin   (-0.22490 0.01289 0);
                        axis             (0 0 1);
                        omega     constant -0.1;
                }
        }
        region2 //Xmax
        {
                ...
        }
}
```

Dynamic region name

Origin and axis of rotation

Angular velocity

In other words, we have set which region will be subject to the motion (in our case a rotation) and the motion parameters.

Moving quad tutorial – dynamic mesh

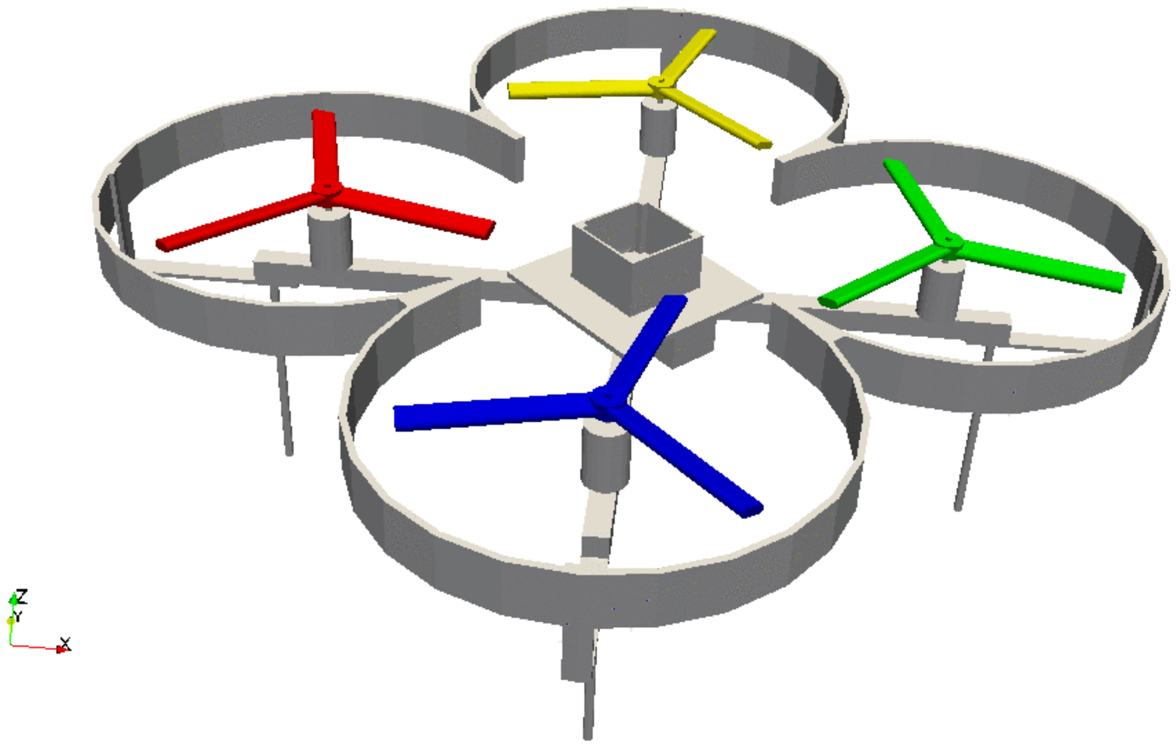
It is now possible to preview the motion of the dynamic mesh with:

```
11. $> moveDynamicMesh
12. $> paraFoam
```

- The `moveDynamicMesh` utility created several time steps associated with a specific spatial grid for the overall mesh.
- The `moveDynamicMesh` utility can also be executed with the `–checkAMI` option to check the weights of the AMI patches in the standard output.
- In this manner we can also visualize the weights values in Paraview by means of the newly created VTK files.

Moving quad tutorial – dynamic mesh

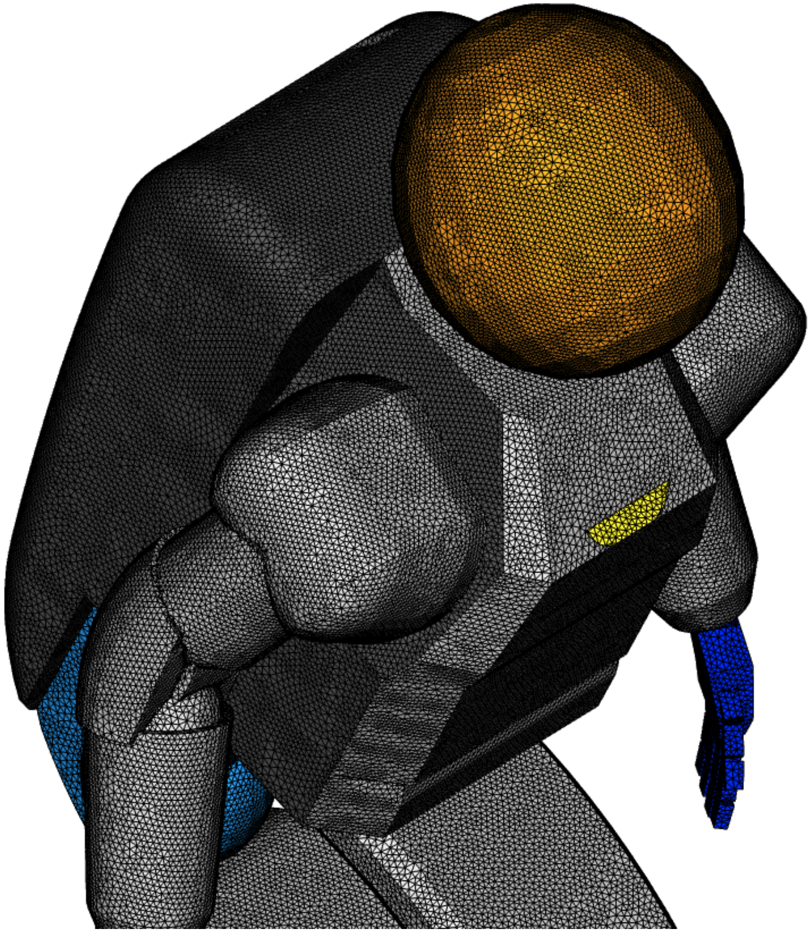
Loading the `total` case in Paraview will give you the possibility of visualizing the drone body and the rotors surface and the sub-meshes interface as follows:

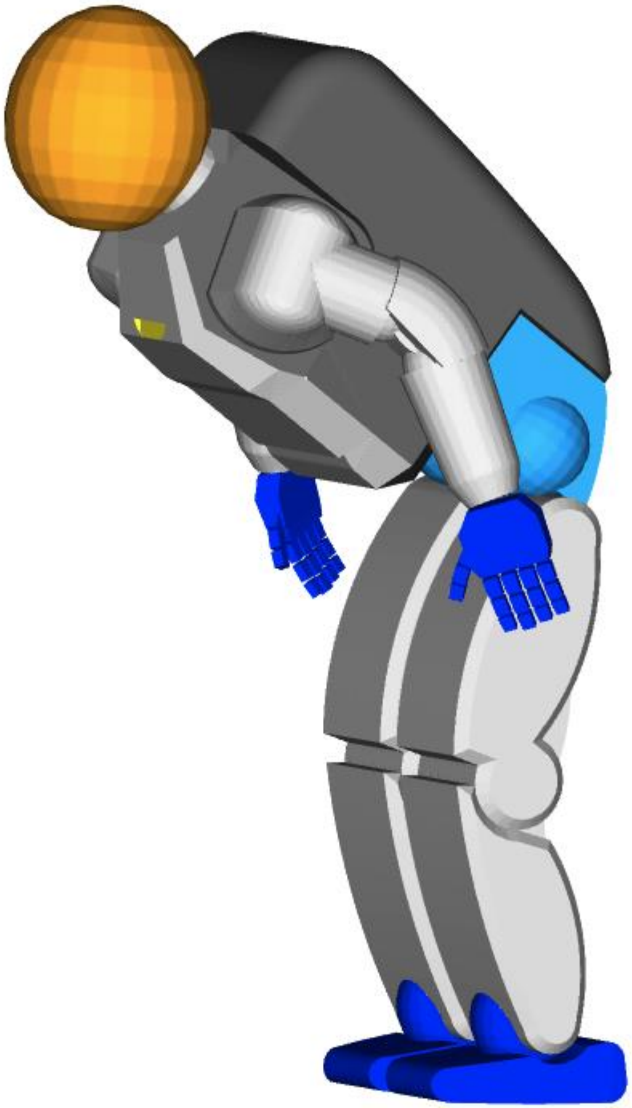


http://www.wolfdynamics.com/wiki/meshing/quadcopter.gif

Play around with the time cursor of Paraview to move the mesh.

Thank you for your attention





Thank you for your attention

- We hope you have found this training useful and we hope to see you in one of our advanced training sessions:
    - OpenFOAM® – Multiphase flows
    - OpenFOAM® – Naval applications
    - OpenFOAM® – Turbulence Modeling
    - OpenFOAM® – Compressible flows, heat transfer, and conjugate heat transfer
    - OpenFOAM® – Advanced meshing
    - DAKOTA – Optimization methods and code coupling
    - Python – Programming, data visualization, and exploratory data analysis
    - Python and R – Data science and big data
    - ParaView – Advanced scientific visualization and python scripting
    - And many more available on request
- Besides consulting services, we also offer '**Mentoring Days**' which are days of one-on-one coaching and mentoring on your specific problem.
- For more information, ask your trainer, or visit our website http://www.wolfdynamics.com/

Contact information

www.wolfdynamics.com

info@wolfdynamics.com

wolf dynamics

multiphysics simulations,
optimization & data analytics