OpenFOAM®

Simulation of the first stages of a turbofan using OpenFOAM

Degree: Aerospace Engineering

Course: Application of Open-Source CFD to Engineering Problems

Delivery date: 09-12-2016

Students:

Herrán Albelda, Fernando

Martínez Viol, Víctor

Morata Carranza, David

Contents

1 Introduction

During the development of the course *'Application of Open-Source CFD to engineering problems'* we have learned the basics of how to use and solve real-world cases and situations related with fluid mechanics using OpenFOAM, an open source CFD tool. The first days of the course, several possible projects were presented and we had to choose one of them and make a report. After agreeing with the professor, we decided to simulate an option that was not on that list. Since we are really interested in propulsion, we thought that it was a good idea to try to simulate the flow inside the first stages of compression of a turbofan engine.

These type of engines are the most used propulsion system in the aerospace industry. It presents several advantages to other systems such as the turbohelix or the turbojet; for example, this kind of engine takes advantage of the flow that goes through the fan (that cannot be more compressed or heated given that the combustion chamber has a limited volume) and comes out through the rear nozzle. This results in a higher thrust for the same amount of fuel that is burned. Thus, it is no surprise that state-of-the-art planes such as the Airbus 380 or the Boeing 747 use this kind of propulsion system.

In order to do a realistic simulation, we have been gathering lots of information of this type of engines and their typical working conditions. Also, we have been searching for information about how could we solve a geometry that is rotating. Several simulations and comparisons will be presented in this report to analyze the validity of the results obtained.

It will be presented in this report how to use the Multiple Reference Frame utility (MRF) as well as the boundary conditions, the mesh generation and the solver that has been used for the case. Additionally, several hypotheses will be considered in order to alleviate the computation time (given that this project has been simulated in a laptop).

2 Case

2.1 Description of the case

The aim of the current project is to analyze the airflow inside the first stages of a turbofan. The turbofan consists of an initial stage where a fan is placed. This first stage increases the pressure of the air that goes through it and, as it can be seen in 2.1, a part of the incoming airflow goes to the low pressure compression stage (main or primary flow) and the other goes to the conducts (secondary flow). As the secondary flow advances through the conduct, the main flow enters the engine core where it goes through the low pressure and high pressure compression stages. By increasing the pressure of the air, the density also increases; thus, a higher mass flow can be mixed with the fuel and burned in the combustion chamber. After this process, the gases go through the turbines, interchanging energy with them (pressure and velocity, mainly), and they arrive at the nozzle, where the air is accelerated, and a thrust force is obtained.

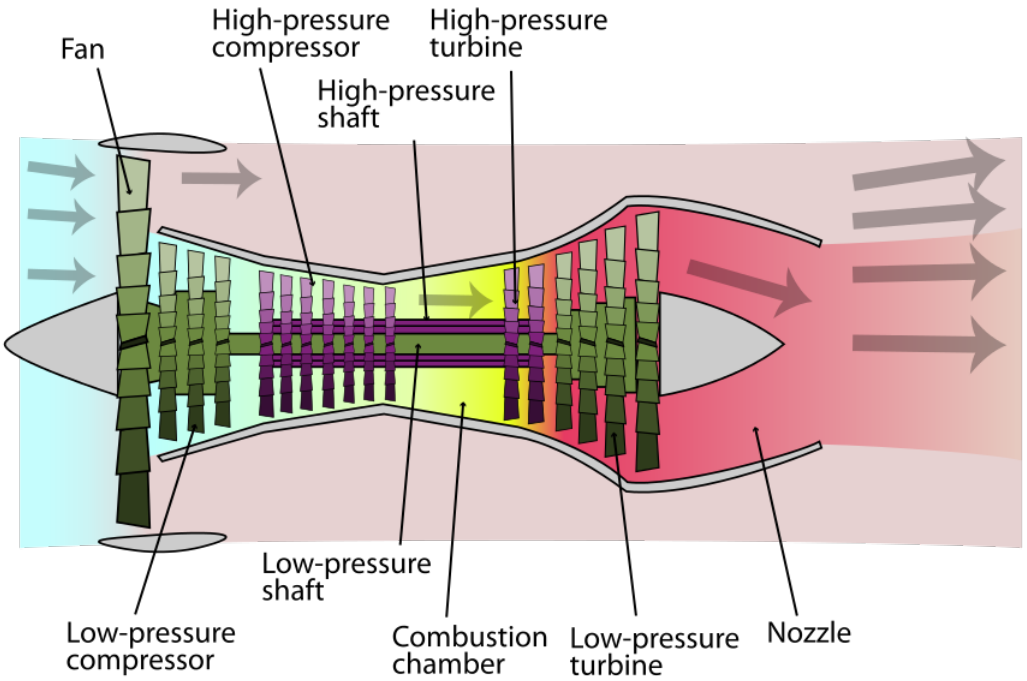

Figure 2.1: Parts of a turbofan

To simulate the flow in the first stages of the turbofan, we have downloaded the following turbine model from *GrabCad*, shown in 2.2. This model is pretty similar with the one shown above (2.1) and the simulation will take place between the back side of the fan and the back side of the second compressor. There are several realistic and potentially functional models on *GrabCad*; however, this model has been selected because it did not present incompatibilities with *SolidWorks* and *Salome*. Given that all the parts of the model have to be clearly differentiated in order to export them as *STL* files, we could not use an assembly.

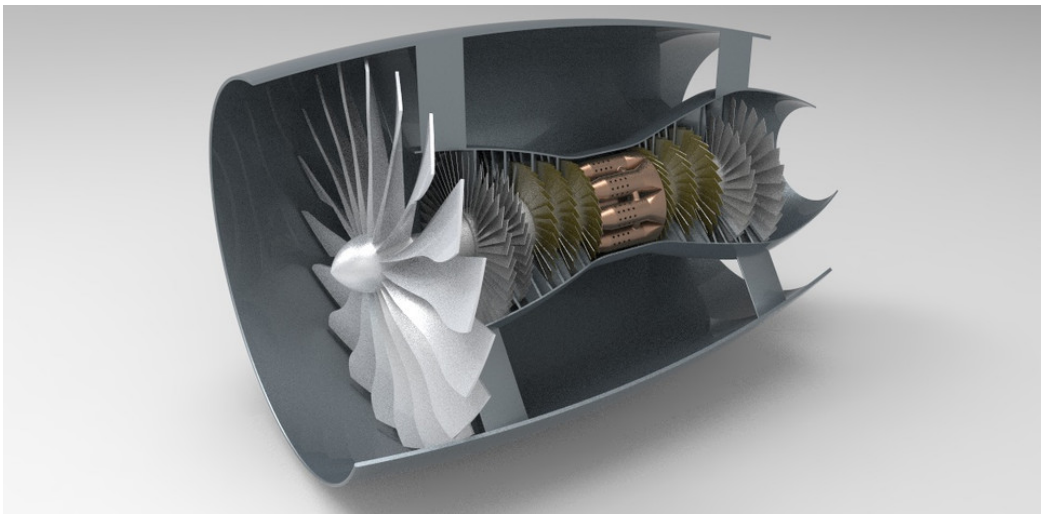

Figure 2.2: Model used

2.2 Hypotheses

Given that we are not able to use a supercomputer and the computational power that is available to us is very limited since we are using our own personal computers to run this simulation, several hypotheses have to be made to alleviate the calculation time, as mentioned before. Some of them can be assumed and some other will make the case a non-realistic project. These hypotheses are presented below.

- Viscous flow
- Incompressible flow

- Newtonian flow
- Stationary flow
- Laminar flow

Clearly, the flow is not incompressible in reality; the aim of the turbofan is to compress the air to seek for a more efficient combustion. However, all of the compressible solvers are really difficult to use and the computational time is also higher given that we have a high number of control volumes (as discussed in the next section) due to a really complex geometry. Thus, although it will not be a completely real flow, we will be also able to see how the low pressure section of the engine increases the pressure of the flow and how the velocity field changes as it goes through the turbine.

The simulation will take into account that it is a tridimensional, that it behaves as a newtonian fluid and it will be run under stationary flow conditions (that is: the velocity in the inlet is always uniform and has a fixed value). With these hypothesis, we can begin to discuss the geometry of the *blockMesh* as well as the refined mesh and the boundary conditions and final results within the next sections.

3 Pre-processing

3.1 Mesh generation

3.1.1 Selection of the tutorial

The first thing that needs to be done is the selection of the tutorial that will work as a base to run the simulation of the case. Among all the tutorials that can be found on the *OpenFOAM* folder, we have selected the **FALTA!** for several reasons.

3.1.2 blockMesh

The definition of the *blockMeshDict* is the first part that needs to be modified. The *blockMesh* must contain the geometry that has to be simulated and we must have an idea of the vertex of the parallelogram that will contain the first stages of the turbofan. In order to do that, the geometry has to be opened using either *Salome* or *Paraview*. Then the axes must be showed and write the points down on the *blockMesh* file. This modification is presented below.

```
vertices
(
    (0.77 0   0)
    (1.35 0   0)
    (1.35 2.3 0)
    (0.77 2.3 0)
    (0.77 0   2.3)
    (1.35 0   2.3)
    (1.35 2.3 2.3)
    (0.77 2.3 2.3)
);
```

It can be clearly seen that the domain of the mesh is a 0.58x2.3x2.3m rectangular prism.

Once the boundaries of the *blockMesh* are defined, the number of cells that it will have has to be set. It has to be taken into account that a very dense

mesh will not be efficient when simulating the case given that we will use the *snappyHexMesh* later and it will be over densified. On the other hand, a very coarse mesh will not be efficient either because additional divisions will have to be set when generating the refined mesh and the computational time might grow. Thus, a solution between a mesh with a very high number of cells and a very low number of cells has to be attained.

This basic mesh has been divided every $0.05m$. It means that we have done 12 divisions in the x direction, 46 on the y direction and 46 more on the z direction.

```
blocks
(
    hex (0 1 2 3 4 5 6 7) (12 46 46) simpleGrading (1 1 1)
);
```

Finally, the differents faces of the mesh must be defined depending on if they are the inlet and outlet faces or the lateral faces.To do this, the vertices are numbered according to their appearance in the *blockMeshDict* and the faces are defined by those numbers. Since, the vertex numeration has been kept the same as the one that comes as default in every tutorial case; it is easier to define the inlet face of the *blockMesh* as well as the outlet face that will be used to define the boundary conditions.

```
boundary
(
    frontAndBack
    {
        type patch;
        faces
        (
            (3 7 6 2)
            (1 5 4 0)
            (0 3 2 1)
            (4 5 6 7)
        );
    }
    inlet
```

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

The *blockMesh* file is presented below. This is the final file that has been used to generate the *blockMesh* and it is included in the *.zip* file attached to this report. Only the parameters mentioned above have been modified; the rest is equal to the tutorial case that has been selected.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  4.0                                   |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}

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

```
convertToMeters 1;

vertices
(
    (0.77 0   0)
    (1.35 0   0)
    (1.35 2.3 0)
    (0.77 2.3 0)
    (0.77 0   2.3)
    (1.35 0   2.3)
    (1.35 2.3 2.3)
    (0.77 2.3 2.3)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (12 46 46) simpleGrading (1 1 1)
);

edges
(
);

boundary
(
    frontAndBack
    {
        type patch;
        faces
        (
            (3 7 6 2)
            (1 5 4 0)
            (0 3 2 1)
            (4 5 6 7)
        );
    }
    inlet
    {
        type patch;
        faces
        (
```

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

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

The log obtained when the *blockMesh* has been generated is presented below. As it can be seen, no errors were found during the computation of this basic mesh. We can see that the number of cells is relatively high (**POSAR N. CELLS**) but perfectly suitable to proceed with the refinement of the mesh. Additionally, a caption of the basic mesh is presented in 3.1.

```
Writing polyMesh
--------
Mesh Information
--------
boundingBox:  (0.77 0 0) (1.35 2.3 2.3)
nPoints:  28717
nCells:  25392
nFaces:  79396
nInternalFaces:  72956
--------
Patches
--------
patch 0 (start:  72956 size:  2208) name:  frontAndBack
patch 1 (start:  75164 size:  2116) name:  inlet
patch 2 (start:  77280 size:  2116) name:  outlet

End
```

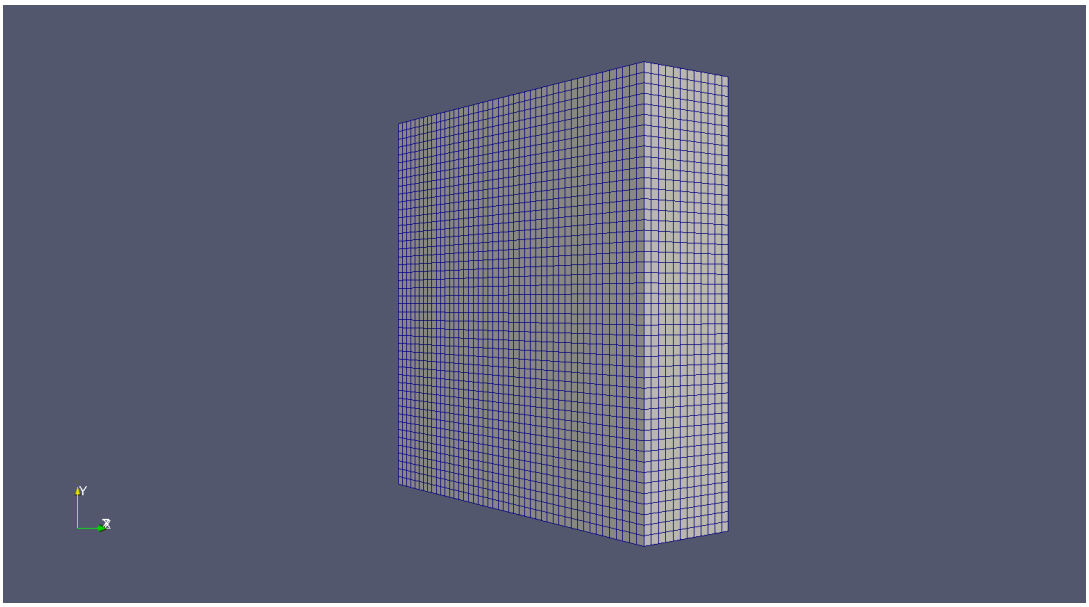

Figure 3.1: blockMeshcaption

3.1.3 Mesh refinement

To refine the mesh, the *snappyHexMesh* utility is used (included in *OpenFOAM*) and several parameters have to be modified in order to obtain a dense mesh that is suitable for the simulation of this complex geometry. It has to be considered that a particular geometry is has a relative velocity as well; this is, the rotor is rotating, and so the first and second stages of the Low Pressure Compression of the turbofan engine, while the nacelle and the combustor are static.

In the *snappyHexMeshDict* several parameters have to be modified. First, the *snappyHexMesh* must be aware of the geometries that it has to take into account. As it can be seen below, the files *Fan.stl*, *HPSpool.stl*, *LPSpool.stl* and *NacelleStator.stl* have been included here.

```
geometry
{
    box1x1x1
    {
        type searchableBox;
        min (1.5 1 -0.5);
        max (3.5 2 0.5);
    }

    Fan.stl
    {
        type triSurfaceMesh;
        regions
        {
        }
    }
    HPSpool.stl
    {
        type triSurfaceMesh;
        regions
        {
        }
    }
    LPSpool.stl
```

```
    {
        type triSurfaceMesh;
        regions
        {
        }
    }
    NacelleStator.stl
    {
        type triSurfaceMesh;
        regions
        {
        }
    }
};
```

Next, the number of control volumes has to be limited to ensure that the laptop is capable of running a simulation. This number has been limited to two million cells, which is a pretty high number and the following lines have to be modified.

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

The next step is to define the refinement required of the mesh for the different geometries that will be simulated. It can be clearly seen in this section that we have included the same *STL* files that we did before. The higher the level of the refinement, the denser the mesh will be and the better it will resemble to the real geometry. But the limitation here is the computational power so, the maximum refinement number cannot be as high as we would like to. Thus, depending on the complexity of the geometry, several minimum (the first number) and maximum (the second number) refinement leves have been defined.

```
    // Surface based refinement
```

```
    // ~~~~~~~~~~~~~~~~~~~~~~

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

    refinementSurfaces
    {
        Fan.stl
        {
            // Surface-wise min and max refinement level
            level (3 5);

            // Optional region-wise level specification
            regions
            {
            }
            patchInfo
            {
                type patch;
                inGroups (meshedPatches);
            }
}
        HPSpool.stl
        {
            // Surface-wise min and max refinement level
            level (3 5);

            // Optional region-wise level specification
            regions
            {
            }
            patchInfo
            {
                type patch;
                inGroups (meshedPatches);
            }
}
        LPSpool.stl
```

```
        {
            // Surface-wise min and max refinement level
            level (6 8);
    cellZone rotor; cellZoneInside inside;

            // Optional region-wise level specification
            regions
            {
            }
            patchInfo
            {
                type patch;
                inGroups (meshedPatches);
            }
}
        NacelleStator.stl
        {
            // Surface-wise min and max refinement level
            level (4 6);

            // Optional region-wise level specification
            regions
            {
            }
            patchInfo
            {
                type patch;
                inGroups (meshedPatches);
            }
}
    }
```

It is worth mentioning that a zone has been defined within the *LPSpool.stl* of the surface refinement from above. The MFR options must have a particular set of cellzones defined that will rotate. So, in order to make these first stages of the compressor to rotate, this has to be defined.

```
        LPSpool.stl
        {
            // Surface-wise min and max refinement level
```

```
            level (6 8);
    cellZone rotor; cellZoneInside inside;
[...] %MODIFIED ^
```

The next step is to select a point within the mesh. So, the *locationInMesh* has to be modified with the x, y, and z-coordinates of a point with that feature.

```
    // After refinement patches get added for all refinementSurfaces and
    // all cells intersecting the surfaces get put into these patches. The
    // section reachable from the locationInMesh is kept.
    // NOTE: This point should never be on a face, always inside a cell, even
    // after refinement.
    locationInMesh (.97443222 1.40534444343 1.24221211);
```

Finally, the *surfaceFeatureExtract* has been used. What this option does it to refine even more the mesh near the points that have complex geometries such as the edges of the blades. This is particularly useful for the turbofan given that it has a high number of blades and a twisting geometry that will resemble more to the reality when using this option. Thus, the following lines within the *snappyHexMeshDict* must be modified.

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

    // Specifies a level for any cell intersected by explicitly provided
    // edges.
    // This is a featureEdgeMesh, read from constant/triSurface for now.
    // Specify 'levels' in the same way as the 'distance' mode in the
    // refinementRegions (see below). The old specification
    //      level   2;
    // is equivalent to
    //      levels  ((0 2));

    features
    (
        {
           file "NacelleStator.eMesh";
           level 5;
```

```
        //    levels ((0.0 2) (1.0 3));
        }
        {
           file "LPSpool.eMesh";
           level 7;
        //    levels ((0.0 2) (1.0 3));
        }
    );
```

All of the other parameters of the *snappyHexMeshDict* have not been modified.

3.1.4 Comparison of different types of mesh

A comparison between the coarse mesh and the refined mesh generated is presented below. The mesh that has been used to simulate the case is the refined mesh (for obvious reasons). Also, it has to be kept in mind that the coarser the mesh, the worse that the geometry will be (this is, it will not resemble to the real geometry), as can be seen below.

3.1.4.1 Coarse mesh

The number of control volumes in the mesh is 40, which is a very small number given the size of the turbofan. It can be seen in 3.2 and 3.3 that this mesh cannot be used to run a simulation.

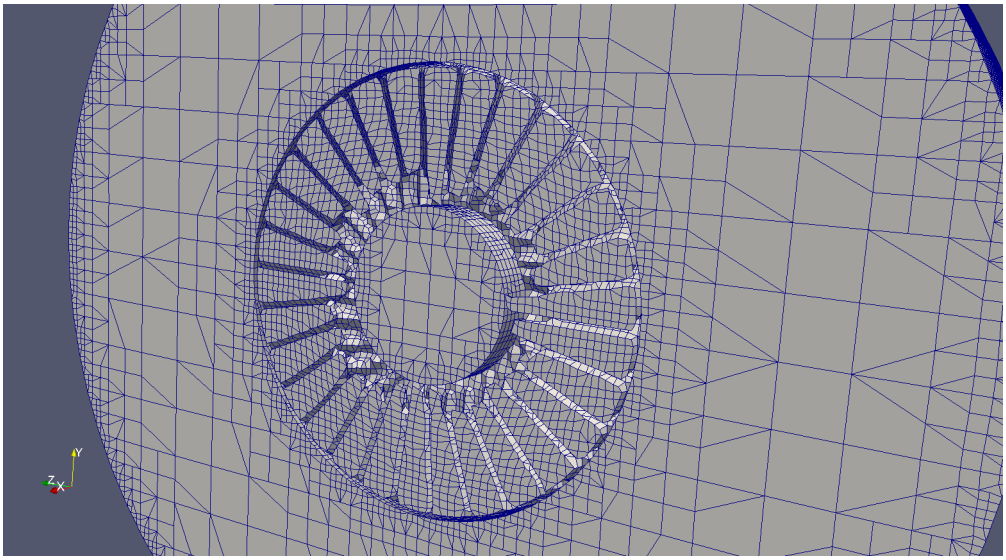

Figure 3.2: Detail of the coarse mesh

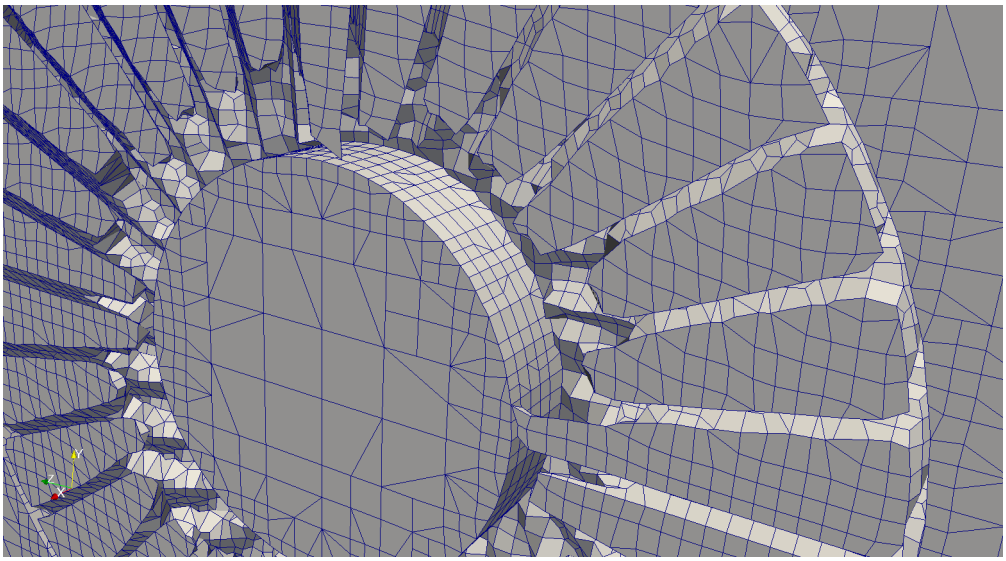

Figure 3.3: Detail of the coarse mesh

3.1.4.2 Standard mesh

Next, it is presented the standard mesh. It has a higher number of cells compared to the coarse mesh, but it cannot be used either given that the size of the turbine is pretty big. The figures 3.4 and 3.5 are a proof of that.

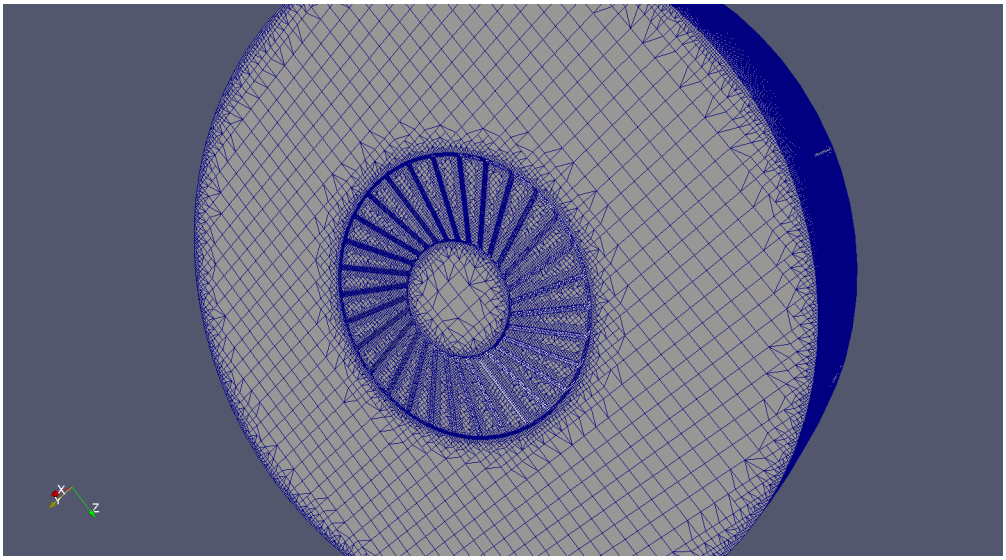

Figure 3.4: Detail of the standard mesh

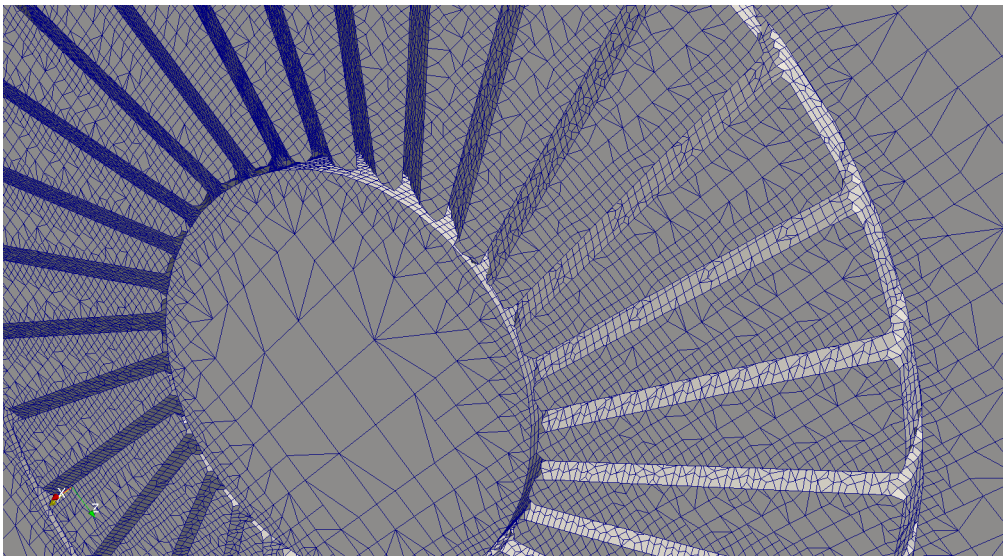

Figure 3.5: Detail of the standard mesh

3.1.4.3 Dense mesh

Finally, the dense mesh is presented. This is by far the best mesh that has been generated for the case. It can be clearly seen on 3.6 and 3.7 that it can be suitable to run the simulation of the first stages of this turbofan.

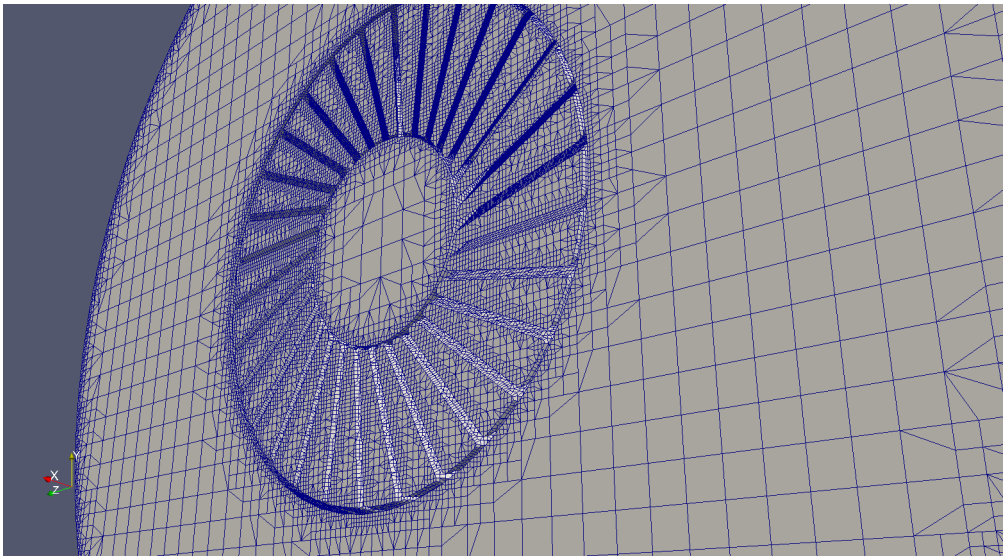

Figure 3.6: Detail of the dense mesh

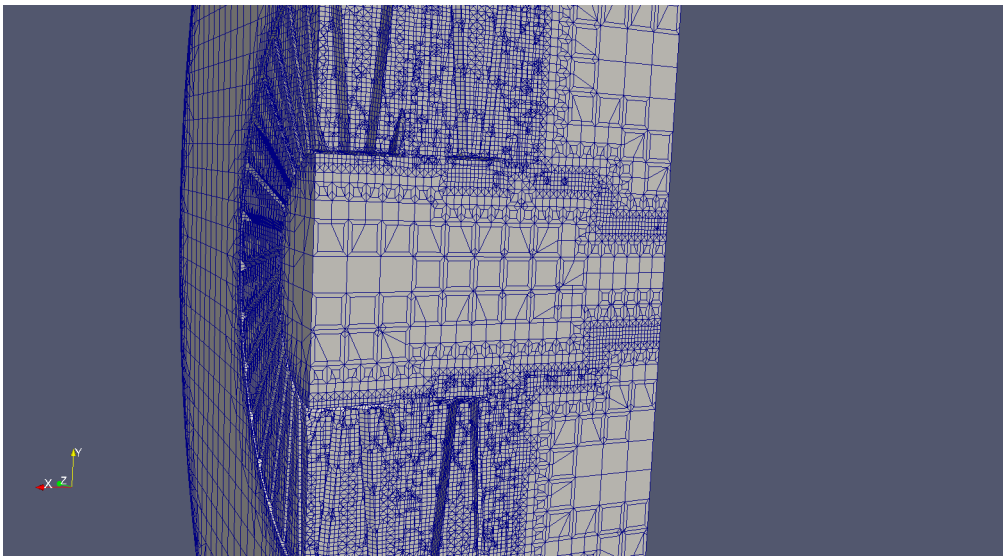

Figure 3.7: Detail of the dense mesh

To end with this section, the *log* file that has been obtained when the mesh has been defined with the *snappyHexMesh* utility is presented below. It has to be taken into account that it is a huge file since the number of iterations is really big and only the final part of the *log* file is shown.

```
Snapped mesh : cells:1626270  faces:5479413  points:2344680
Cells per refinement level:
    0 8592
    1 24456
    2 126790
    3 918976
    4 299109
    5 118950
    6 129397
Writing mesh to time constant
Wrote mesh in = 97.48 s.
Mesh snapped in = 1121.77 s.
Checking final mesh ...
Checking faces in error :
    non-orthogonality > 65  degrees                        : 0
    faces with face pyramid volume < 1e-13                 : 0
    faces with face-decomposition tet quality < 1e-15      : 0
    faces with concavity > 80  degrees                     : 0
    faces with skewness > 4   (internal) or 20  (boundary) : 0
    faces with interpolation weights (0..1)  < 0.05        : 0
    faces with volume ratio of neighbour cells < 0.01      : 0
    faces with face twist < 0.02                           : 0
    faces on cells with determinant < 0.001                : 0
Finished meshing without any errors
Finished meshing in = 1705 s.
End
```

3.2 Rotation

An important consideration for this project is that the rotor is spinning at a high speed and this condition has to be somehow communicated to *OpenFOAM*. To work with this condition the Multiple Frame Reference (MFR) method has to be used. The method itself is based on adding a 'source' to the momentum equation in a previously defined zone named 'rotor'. How to do that has been shown in the previous section.

The **constant/MRFProperties** file has to be modified with the appropriate parameters for the case. Since the axis of *OpenFOAM* are based on the right hand rule, the rotation axis must be defined as follows: **CAMBIAR PER ARA!** $(1, 0, 0)$. From information that can be found only, it has been chosen 5000 rpm as the angular velocity of the rotor. It should be noted that the units used must be those of the International System, so the 5000 rpm must be converted to rad/s. Furthermore, an origin point has to be indicated in the axis of rotation; in this case, the origin point is as follows: (01.16753541.15542633).

Finally, it has to be indicated wheter some geometries are rotating or not. Thus, the next parameter has to be modified:

```
    nonRotatingPatches (
NacelleStator.stl
);
```

So, the file **constant/MRFProperties** for the current case is the following:

```
/*---------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  4.0                                   |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
```

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

MRF1
{
    cellZone    rotor;
    active      yes;

    // Fixed patches (by default they 'move' with the MRF zone)
    nonRotatingPatches (
NacelleStator.stl
);

    origin    (0 1.1675354 1.15542633);
    axis      (1 0 0);
    omega     523.5987756;
}

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

It can clearly be seen that the modification of any parameter is pretty straigh-forward in the file since there is no possible confusion with the names.

3.3 Boundary conditions

Some initial conditions and values have to be defined in order to run the simulation. The initial velocity field and pressure are especially important since these inputs are the starting values of the simulation. So, it is clear that the values at the *inlet*, which was defined within the previous sections, have to be defined. Additionally, the values at the *outlet* of the geometry have to be defined as well, but the can be, unlike the *inlet*, not fixed values. Thus, the following modifications have to be made in the *0.orig/U*.

```
internalField   uniform (0 0 0);

boundaryField
{
[...]
    inlet
    {
        type            fixedValue;
value uniform (30 0 0);
    }

    outlet
    {
        type            zeroGradient;
    }
}
[...]
```

These conditions for the pressure have to be modified as well in the *0.orig/p* file.

```
internalField   uniform 0;

boundaryField
{
[...]
    inlet
    {
        type            fixedValue;
    value     uniform 0;
```

```
    }

    outlet
    {
        type            zeroGradient;
    }
}
[...]
```

Given that we have more solids that will be simulated (the Nacelle and the rotor), their conditions for the velocity and the pressure have to be defined as well. Since we have a newtonian flow, the condition for the velocity in these solids will be the *noSlip* condition. For the pressure, the condition will be the *zeroGradient* condition.

Finally, both files are presented below. These are the files that have been used to run the simulation for this case.

Velocity

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  4.0                                   |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

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

internalField   uniform (0 0 0);
```

```
boundaryField
{
    LPSpool.stl
    {
        type            noSlip;
    }

    NacelleStator.stl
    {
        type            noSlip;
    }

    inlet
    {
        type            fixedValue;
value uniform (30 0 0);
    }

    outlet
    {
        type            zeroGradient;
    }
}

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

Pressure

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  4.0                                   |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
```

```
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    LPSpool.stl
    {
        type            zeroGradient;
    }

    NacelleStator.stl
    {
        type            zeroGradient;
    }

    inlet
    {
        type            fixedValue;
    value     uniform 0;
    }

    outlet
    {
        type            zeroGradient;
    }
}

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

3.4 Properties of the flow

The properties of the flow have to be defined taking into account the hypotheses made for the case. Given that the flow is air, the kinematic viscosity *nu* (ν) is $1.5e^{-5}m^2/s$. Also, as mentioned previously, the simulation will work with a Newtonian flow and this has to be defined here as well.

To change all these flow parameters and properties, the *constant/transportProperties* file has to be modified as follows. There is no need to dedicate more time to this file since everything is really clear within it.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  4.0                                   |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

transportModel  Newtonian;

nu              [0 2 -1 0 0 0 0] 1.5e-05;

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

Another hypotheses that has been made in order to alleviate the computation time is that the flow is laminar, although it would not be a very realistic flow in this case. To impose this condition, a modification has to be made in *constant/turbulenceProperties* file.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  4.0                                   |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      turbulenceProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

simulationType laminar;

RAS
{
    RASModel            kOmegaSST;

    turbulence          on;

    printCoeffs         on;
}

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

Since we are only interested in the topology of the flow of this file, the other parameters have not been modified.

3.5 ControlDict

This file is really important for the simulation since it contains the timesteps that it will pursue. So, to begin, the initial and final times of the simulation have to be defined. In this case, the *startTime* is 0s and the *endTime* has been set to 20 s. No more simulation time is needed since it has been assumed that the flow is stationary. These changes can be made in the *system/controlDict* file.

NOTA: DEPEN DEL SOLVER! Si no fem servir SIMPLEFOAM s'he de tornar a COPIAR!

```
application     simpleFoam;

startFrom       latestTime;

startTime       0;

stopAt          endTime;

endTime         20;
```

The next thing that has to be done is to define the time step for the simulation as well as the write interval (this is; the number of time that must pass in order to make a file for the simulation in the current simulation time).

```
deltaT          1;

writeControl    timeStep;

writeInterval   5;

purgeWrite      0;

writeFormat     binary;

writePrecision  6;
```

```
writeCompression uncompressed;

timeFormat      general;

timePrecision   6;
```

Since it is really clear what the other variables do, they will not be explained here. Finally, the file that has been used to run the simulation is presented below.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  4.0                                   |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

application     simpleFoam;

startFrom       latestTime;

startTime       0;

stopAt          endTime;

endTime         20;

deltaT          1;
```

```
writeControl    timeStep;

writeInterval   5;

purgeWrite      0;

writeFormat     binary;

writePrecision  6;

writeCompression uncompressed;

timeFormat      general;

timePrecision   6;

runTimeModifiable true;

/*functions
{
    #include "streamLines"
    #include "wallBoundedStreamLines"
    #include "cuttingPlane"
    #include "forceCoeffs"
}

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

3.6 fvSchemes and fvSolution

The fvSchemes dictionary contains the numerical schemes that they are used in the simulation and the fvSolution contains the resolution methods and the tolerances for each equation.

In the hypotheses section we have considered that the flow is stationary. To impose this condition we must modify the **system/fvSchemes** file, the section of In *ddtSchemes*, which is the temporal integration scheme. We must write steadyState. The fvSolution dictionary has not been modified.

The

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  4.0                                   |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

ddtSchemes
{
    default         steadyState;
}

gradSchemes
{
    default         Gauss linear;
    grad(U)         cellLimited Gauss linear 1;
}
```

```
divSchemes
{
    default         none;
    div(phi,U)      bounded Gauss linearUpwindV grad(U);
    div(phi,k)      bounded Gauss upwind;
    div(phi,omega)  bounded Gauss upwind;
    div((nuEff*dev2(T(grad(U))))) Gauss linear;
}

laplacianSchemes
{
    default         Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}

wallDist
{
    method meshWave;
}


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


/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  4.0                                   |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
```

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{
    p
    {
        solver           GAMG;
        tolerance        1e-7;
        relTol           0.01;
        smoother         GaussSeidel;
    }

    Phi
    {
        $p;
    }

    U
    {
        solver           smoothSolver;
        smoother         GaussSeidel;
        tolerance        1e-8;
        relTol           0.1;
        nSweeps          1;
    }

    k
    {
        solver           smoothSolver;
        smoother         GaussSeidel;
        tolerance        1e-8;
        relTol           0.1;
        nSweeps          1;
    }
```

```
    omega
    {
        solver          smoothSolver;
        smoother        GaussSeidel;
        tolerance       1e-8;
        relTol          0.1;
        nSweeps         1;
    }
}

SIMPLE
{
    nNonOrthogonalCorrectors 0;
    consistent yes;
}

potentialFlow
{
    nNonOrthogonalCorrectors 10;
}

relaxationFactors
{
    equations
    {
        U               0.9;
        k               0.7;
        omega           0.7;
    }
}

cache
{
    grad(U);
}

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

3.7 Running the application

4 Simulation of the turbofan

EMPTY

5 Post-process

EMPTY