

Appendix A - OpenFOAM Setup

The objective of this Appendix is to provide a guide to the meshing and execution processes that were applied on the numerical investigation of clean and iced propeller performance in this study.

The framework of the propeller simulation was based on the OpenFOAM `pimpleFoam` propeller tutorial, Antham (2016), and Gagliarde (2020).

A.1 Meshing

The OpenFOAM package has an excellent tool for meshing complex geometries called `snappyHexMesh`. It is very powerful to meshing a wide range of complex geometries. However it lacks in mesh refinement quality when a fine refinement is desired at some mesh regions such as an airfoil leading-edge.

Contrastingly, `cfMesh` is a much easier tool to handle and requires minimum user inputs. The software was designed to generate the mesh with just a few inputs, differently than `snappyHexMesh`. `cfMesh` takes much less time to generate a mesh than `snappyHexMesh` and enables to address refinement to the desired regions more adequately.

A.1.1 Domain

The mesh domain consists in two regions: a rotating region, called `rotor`, which is a small cylinder that contains the propeller geometry; and a static cylinder, called `stator`, which encloses the rotor region and represents the domain of the propeller flow, as shown in Figure A.1. The Arbitrary Mesh Interface (AMI) was used to couple the `rotor` and `stator` patches which share the same boundaries at their interface.

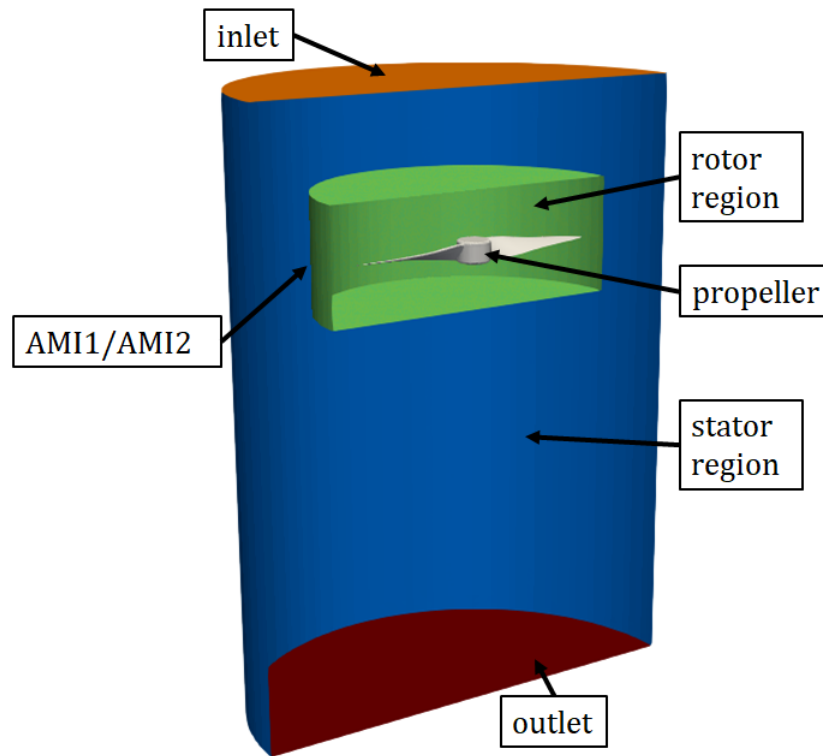


FIGURE A.1 – Propeller mesh domain.

A.1.2 Geometry

The geometry of the domain elements was generated with SALOME 9.6.0, which is a free CAD and meshing software that allows creating the solids, defining the patches and boundaries, as well as export them as STL files.

The mesh domain is composed by two solids: `innerCylinderSmall` and `outerCylinder`, which are shown in Figure A.2a. The cylinders were named after the OpenFOAM propeller tutorial. The SALOME `Cylinder` feature was used to create them. According to the geometry dependency tree in Figure A.2b, the cylinders were rotated and translated to be positioned along the y-axis, and such that `innerCylinderSmall` was inside `outerCylinder` close to the inlet face. The origin of the domain lies in the center of the propeller.

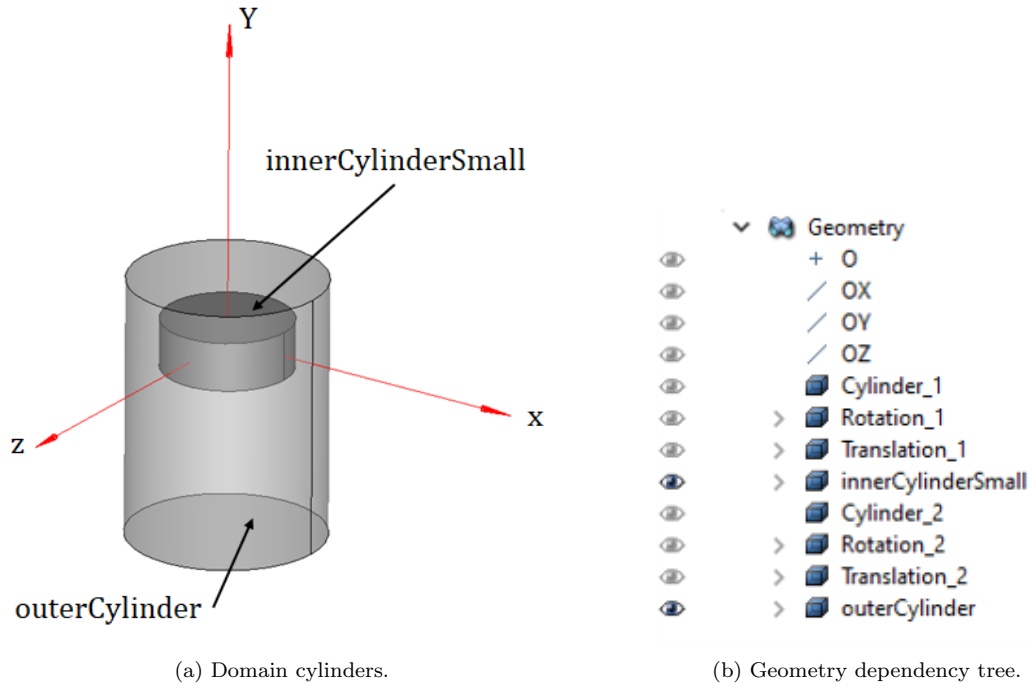


FIGURE A.2 – SALOME cylinders generation.

The propeller solid can either be meshed as a single solid, as it can be divided in regions of interest to which different levels of refinement can be attributed to. Both the **snappyHexMesh** and **cfMesh** meshers showed limitations on the refinement of the propeller leading-edge, trailing-edge and outboard blade sections using only one refinement level for the whole propeller solid. Hence, in this study, the propeller was divided in 6 regions, as shown in Figure A.3, to address a more adequate and dedicated refinement to regions below:

- **hub** - The propeller hub.
- **ibdLeadingEdge** - The inboard blade leading-edge.
- **obdLeadingEdge** - The outboard blade leading-edge.
- **ibdTrailingEdge** - The inboard blade trailing-edge.
- **obdTrailingEdge** - The outboard blade trailing-edge.
- **mainBox** - The blades upper and lower surfaces.

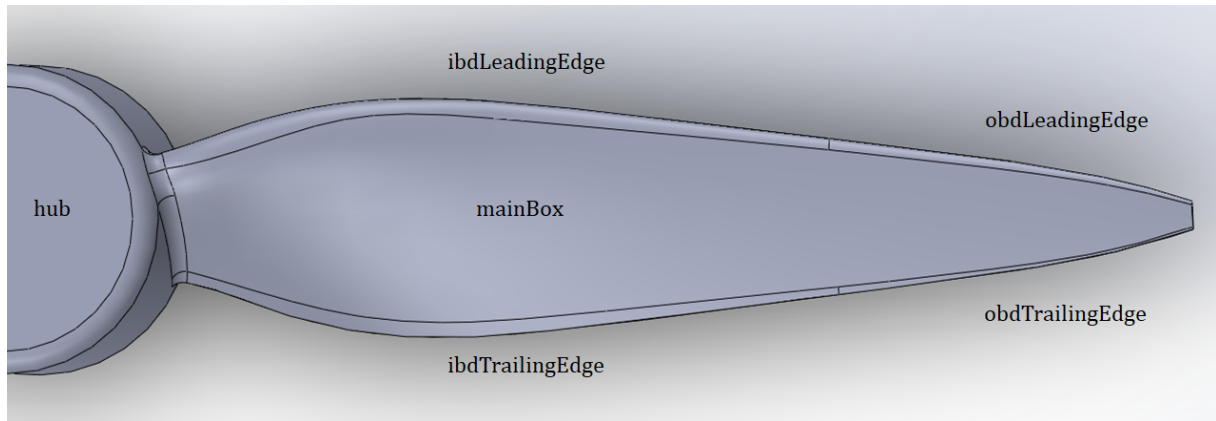


FIGURE A.3 – Blade parts.

The **stator** and **rotor** meshes are created separately in two different folders and then are combined, as discussed in the next sections.

A.1.3 Stator

The patch **innerCylinderSmall** must be present in both the **stator** and **rotor** regions so they can be later combined to form the AMI1 and AMI2 interface patches.

In the **stator** region, the patch is a copy of **innerCylinderSmall.stl** and was named as **innerCylinderSmall_slave.stl**. The solids **innerCylinderSmall_slave.stl** and **outerCylinder.stl** must be placed in the simulation base directory and shall be combined into one solid **combined.stl**. Feature edges are then extracted with **surfaceFeatureEdges**, and the mesh can be generated with **cfMesh cartesianMesh** mesher. The **stator** mesh configurations are available at [GitHub](#).

```
cat outerCylinder.stl innerCylinderSmall_slave.stl > combined.stl
surfaceFeatureEdges combined.stl combined.fms -angle 5
cartesianMesh
```

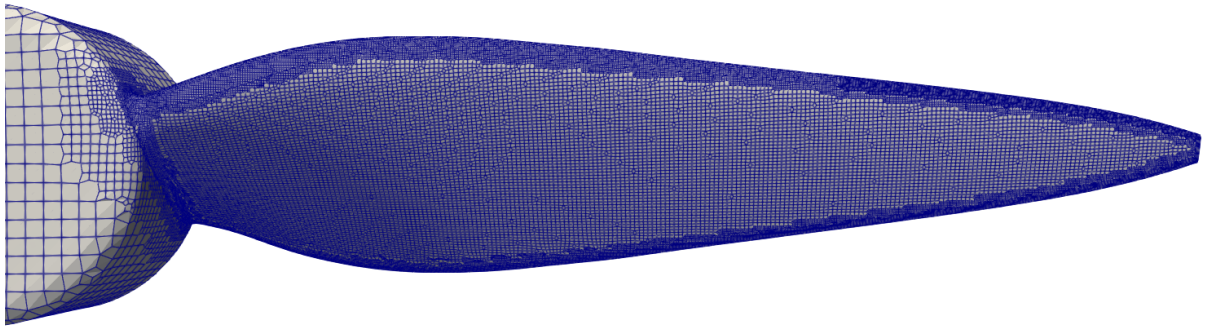
The advantage of **cfMesh cartesianMesh** is that it already runs in parallel and does not need the user intervention to it.

A.1.4 Rotor

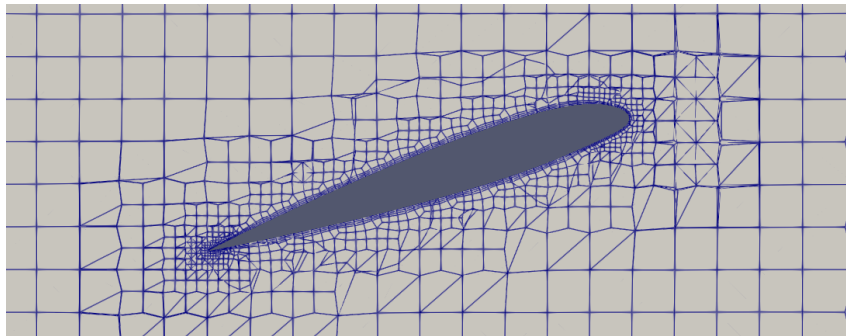
The propeller individual solids and **innerCylinderSmall** must be placed in the **rotor** directory. The propeller parts are combined into a single **prop.stl** and then combined to the **innerCylinderSmall**. The same procedure of **stator** for generating the mesh is

repeated. The `rotor` mesh configurations are available at GitHub. Figure A.4a shows how the different levels of refinement are distributed over the blade surface. The inboard leading-edge and trailing-edge are more refined than the upper and lower blade surfaces, as well as the outboard edges are further more refined than the inboard sections.

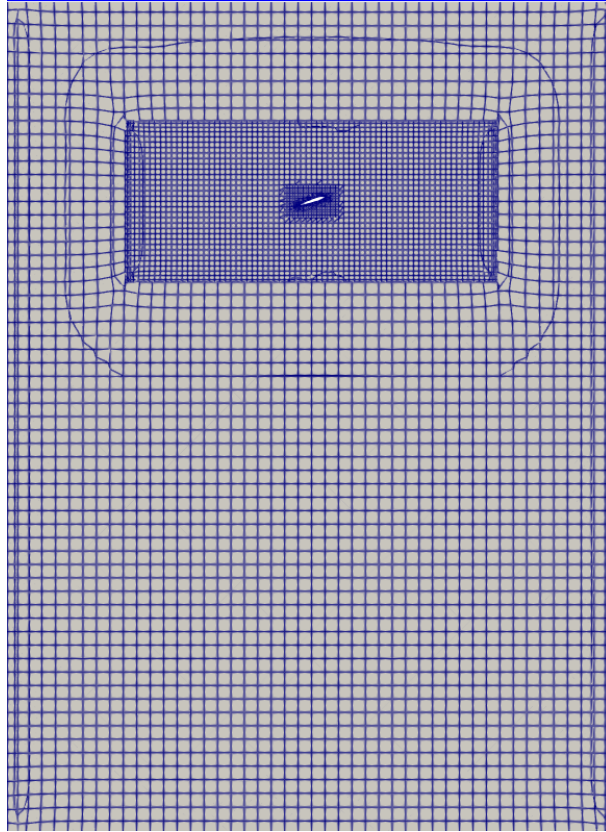
```
cat hub.STL ibdLeadingEdge.stl ibdTrailingEdge.stl obdLeadingEdge.stl ...  
... obdTrailingEdge.stl mainBox.stl > prop.stl  
cat prop.stl innerCylinderSmall.stl > combined.stl  
surfaceFeatureEdges combined.stl combined.fms -angle 5  
cartesianMesh
```



(a) Blade surface refinement.



(b) Blade cross-section mesh.



(c) Volumetric mesh.

FIGURE A.4 – Mesh visualization.

A.1.5 Combining the Meshes

The separate meshes must be combined into a single domain with the `mergeMeshes` application. The command shall be executed in the `stator` mesh directory so that it becomes the `region0` and `rotor` the `region1`. The resulting mesh is shown in Figure A.4c.

```
cd stator
mergeMeshes . ../rotor -overwrite
```

A.1.6 Create patches

Some of the patches that are used in the simulation boundary conditions such as the inlet and outlet, and the the AMI1 and AMI2, which are used in the MRF method, are created here. `system/createInletOutletSets.topoSetDict` creates the `inlet` and `outlet` patch faces. And the `system/createPatchDict` creates the `inlet` and `outlet` patch from the faces, as well as creates the AMI1 and AMI2 patches from `innerCylinderSmall` and `innerCylinderSmall_slave`, respectively. The files are available at [GitHub](#).

```
topoSet -dict system/createInletOutletSets.topoSetDict
createPatch -overwrite
```

A.1.7 MRF

Although the AMI patches were created, it is still required to define the rotating cell region that will be used by the MRF method in the `constant/MRFProperties` file. The `system/createAMIFaces.topoSetDict` is used then to create the rotating cell zone from the `region1` cell set. `checkMesh` must be executed before in order to the mesh regions be created properly within the mesh files. This process can be carried out either by using the `topoSet` or the `setSet` applications. `topoSet` is preferred since it can be automatized. The files are available at [GitHub](#).

```
checkMesh
topoSet -dict system/createAMIFaces.topoSetDict

or

checkMesh
setSet
cellZoneSet rotor new setToCellZone region1 quit
```

The `constant/MRFProperties` file is where the rotating cell zone, and the simulation propeller rotation speed, axis and origin are defined. The files are available at [GitHub](#).

```
MRF1
{
    cellZone    rotor;
    active      yes;

    // Fixed patches (by default they 'move' with the MRF zone)
    nonRotatingPatches (AMI1 AMI2);

    origin      (0 0 0);
    axis         (0 1 0);
    omega        314.16;    // [rad/s]  314.16 rad/s = 3000 rpm
}
```

A.2 Simulation

Given that the mesh was properly generated, the simulation boundary and initial conditions, turbulence model, and simulation parameter must be configured first before simulating the case.

A.2.1 Turbulence Model

The turbulence model that is applied in the simulation is configured in the `constant/turbulenceProperties`. The `spalartAllmaras` model was used in this study.

```
simulationType  RAS;

RAS
{
    RASModel      SpalartAllmaras;
    //RASModel     kEpsilon;
    //RASModel     kOmegaSST;

    turbulence     on;

    printCoeffs    on;
}
```

A.2.2 Boundary Conditions

The initial and boundary conditions of each simulation variable must be configured in the files the `0/` directory. In each of these files the boundary conditions of each patch must be configured. The flow velocity must be zero at the propeller in order to meet the boundary-layer no slip condition at the surface. The inlet speed, pressure, and temperature of the inlet and volumetric domain is also configured.

The turbulence models are represented by a set of equations that are solved along with the flow governing equations. New equations also increase the number of simulation variables, and boundary conditions must also be configured for these variables. The NASA Modeling Resource provides a good guide on how to set up the initial values boundary condition for these variables.

A.2.3 Execution

The compressible steady-state `rhoSimpleFoam` solver was chosen for the propeller simulation. If the reader is interested in the propeller unsteady flow, an unsteady solver, such as `rhoPimpleFoam`, should be applied along with the Dynamic Mesh approach, since the MRF method can only be used in steady-state. The Dynamic Mesh approach is implemented in the `pimpleFoam` propeller tutorial with the incompressible `pimpleFoam` solver. The solver can be executed by simply typing:

```
rhoSimpleFoam
```

The simulation convergence criteria is achieved when both the residuals, and the forces and moments converges simultaneously, as shown in Figure A.5.

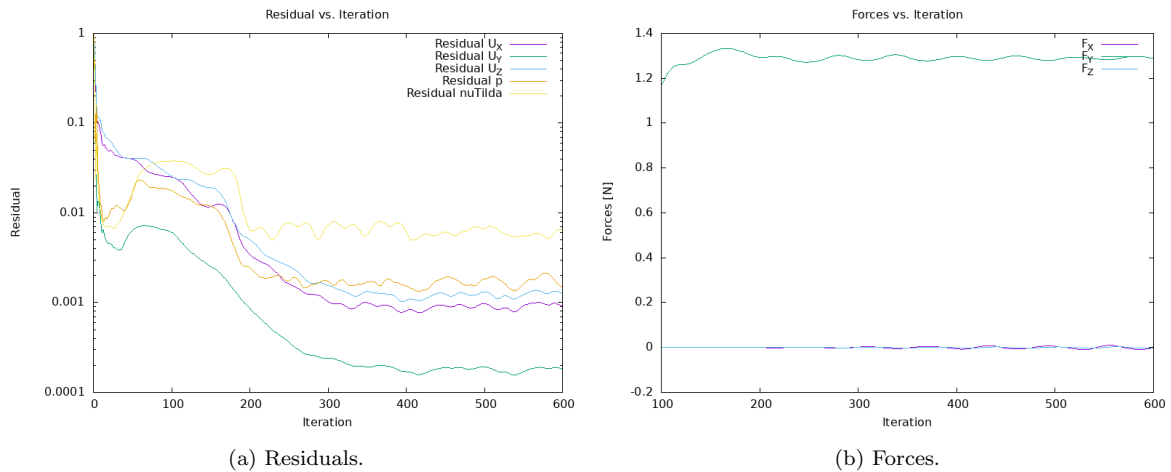


FIGURE A.5 – Simulation convergence.

A.2.3.1 Parallel Computation

However, serial computation takes quite a long time to run and parallel computation is required to increase the simulation speed. Parallel computation requires the mesh to be divided in multiple domains that each processor will execute separately. The `system/decomposeParDict` is responsible for splitting the mesh. The solver can then be executed in parallel with `mpirun`. The number of processors must be the same as configured in the `system/decomposeParDict`.

```
decomposePar
mpirun -np <number-of-processors> rhoSimpleFoam -parallel
reconstructPar
```

A.2.3.2 Batch Computation

This study was interested in the propeller dynamic performance. Thus, the propeller simulation had to be carried out at multiple points so that a performance curve could be obtained. The OpenFOAM simulation set up is very time-consuming and automation is required improve simulation time.

This automation can be made with basically any programming language script. Nevertheless, the PyFoam Python library was developed exclusively to handle the OpenFOAM environment and provides multiple tools and resources dedicated to its characteristics.

The `paramVariation.py` script, available at GitHub, enables the variation of the propeller RPM, inlet speed, among other parameters. The script copies a base folder that contains the mesh, initial and boundary conditions and simulation scripts, and renames it according to the run number and velocity. Then, it changes the boundary conditions and runs the case. It also uses the previous simulation point solution as initial condition to next, which considerably saves simulation time, since the flowfield is already developed and the solver has just to update it.

A.2.3.3 Post Processing

The post processing consists in reading the solver output files of each run and organizing the forces and moments, for each velocity and rotation speed, in output files. The `parseResults.py` script was used on the post processing of the simulation data.

These output files are then used to plotting the results, as observed in Section 5.2 figures. Paraview software was also used on the flow visualization by plotting surface shearlines and pressure flowfield.