Ingo Jahn, Rowan J. Gollan, and Peter A. Jacobs

# Guide to foamMesh

*using Eilmer geometry elements to build OpenFOAM meshes.*

April 4, 2025

Technical Report 2018/9
School of Mechanical & Mining Engineering
The University of Queensland

**Abstract**

foamMesh is an extension of the geometry package developed for the Eilmer4 compressible-flow simulation program. It allows gas flow and solid domains, generated using the Eilmer4 geometry package, to be converted into grids suitable for OpenFOAM simulations.

The generation of OpenFOAM grids is achieved by adding an extra step to the Lua scripts, used to convert the 2D and 3D grids of finite volume cells into corresponding foam meshes. At the same stage labels are assigned to the outward facing edges in the (x,y)-plane for 2D grids, or outward facing patches for 3D grids, which allow definition of the OpenFoam boundary conditions.

Eilmer4 is available as source code from https://github.com/gdtk-uq/gdtk and is related to the larger collection of compressible flow simulation codes found at https://gdtk.uqcloud.net/.

**Acknowledgment**

This document borrows heavily from the Eilmer4 Geometry User Guide [1].

# Contents


**1 Introduction** **1**
1.1 Some advice . . . . . 1

**2 Compiling** **3**

**3 Converting Structured Grids to FoamBlocks** **5**
3.1 Create appropriate OpenFOAM case . . . . . 5
3.2 Global Settings . . . . . 5
3.3 FoamBlock constructor . . . . . 6
3.4 OpenFOAM boundary conditions . . . . . 6
3.5 Building the grid . . . . . 8
3.6 Some debugging help . . . . . 8

**4 Examples** **9**
4.1 2D (planar) examples . . . . . 9
4.2 3D examples . . . . . 16

**References** **25**

**A Useful Linux Commands for OpenFOAM** **27**

**B Make your own debugging cube** **29**

# 1

# Introduction

foamMesh is a standalone program, based on the geometry engine of the Eilmer geometry package [2]. Structured grids are generated using the same Lua descriptors used in the Eilmer geometry package and are converted into unstructured grids to suit OpenFOAM. At the same time labels are assigned to external edges (in 2D) or faces (in 3D) to allow boundary conditions to be assigned using the OpenFOAM approach. This report is a companion to the geometry user-guide [2], which gives details on the generations of the actual structured meshes.

The tool supports the conversion of multi-block structured 2D meshes (planar and axi-symmetric) and structured 3D meshes. Some support is provided to assist with setting up the initial OpenFOAM boundary conditions, however final adjustment is left to the user.

## 1.1 Some advice

Before describing the details of using foamMesh (and using it to convert geometric elements that you will use to build a description of your flow domain), we would like to offer some advice on the process of building that description. The process is one of programming the geometry-building program to construct an encoded description of your flow domain. With this in mind, we advise the following procedure:

1. Start with a rough sketch of your flow domain on paper, labeling key features.
2. Start small, building a script to describe a very simple element from your full domain.
3. Process this script with foamMesh and view the resulting grid using paraFoam (disable / de-select field data prior to import to acoid crashes associated with data not being available).
4. View this artifact to check that it is what you wanted, debugging as required.
5. Proceed in small steps to complete your domain description.

We believe that this procedure will result in a far more satisfactory experience than coding your entire description in one pass. You might be lucky, but chances are that your mistakes will overpower your luck.

# 2

# Compiling

When building the Eilmer4 code collection from source, foamMesh is not built by default. To test if foamMesh is available, try `foamMesh --help`. If you get the foamMesh help instructions, foamMesh is ready to run.

Assuming you are able to successfully build the core solver, compiling foamMesh is a straightforward task. foamMesh is compiled from the `/gdtk/src/geom/` directory using the `make install` command.

# 3

# Converting Structured Grids to FoamBlocks

## 3.1 Create appropriate OpenFOAM case

For foamMesh to work correctly it must be executed from within an OpenFOAM `case` directory. An apropriate case directory can either be generated manually, or more conveniently by copying an existing OpenFOAM tutorial and removing unnecessary files. The resulting case directory should, at a minimum, contain the follwing sub-directories and files (based on OpenFOAM 5.x):

- case/ ← Case directory, can have any name
  - 0/ ← Directory for initial conditions. Can be empty!
  - constant/
    - Polymesh/ ← Delete initially to remove remnants of previous grid.
    - blockMeshDict ← Delete if it exists
    - thermoPhysicalProperties ← Keep, but not essential
    - turbulenceProperties ← Keep, but not essential
  - system/
    - controlDict ← Keep!
    - fVSolutions ← Keep!
    - fVSchemes ← Keep!

Once the above directory structure exists, the `job.lua` file should be placed in the `case/` directory and foamMesh should be executed from this directory.

## 3.2 Global Settings

The following optional global settings can be used to adjust the type of mesh and outputs generated by foamMesh. It is suggested to set these global settings close to the start of the `job.lua` file for added clarity.

| Variable | Default | Options and Definition |
|---|---|---|
| `turbulence_model` | none | none/S-A/k-epsilon - creates initial conditions in 0_temp for corresponding turbulence model |
| `axisymmetric` | false | true/false - flag to determine if planar or axi-symmetric 2D grid is generated. |
| `dtheta` | 0.01 rad | Set angle between front and rear face for axi-symmetric simulations. |
| `dz` | 0.2 m | Sets thickness (in z-direction) of domain for planar 2-D simulations. |

The last three entries are only used when passing a 2D grid object to foamBlock. Based on these settings, foamMesh will automatically create the appropriate 3D mesh, one cell thick in the z-direction (or $\theta$-direction for axi-symmetric) to suit 2D simulations in OpenFOAM.

## 3.3 FoamBlock constructor

When using foamMesh to generate grids for OpenFOAM, the grid generation starts by following the same approach as outlined in the Eilmer 4 Geometry Package documentation [1].

1. Geometry components are defined, such as points and paths.
2. Paths are joined to make parametric surfaces (or volumes for 3D).
3. Structured grid objects are created from the surfaces (or volumes in 3D). At this stage clustering of grids is performed.

To create a grid suitable for OpenFOAM the next step involves assigning labels to the faces of these blocks that will be outwards facing in the final combined grid. Effectively these are the faces where boundary conditions need to be prescribed. This is done using the `FoamBlock` constructor. The code to combine the two grid objects `grid_left` and `grid_right`, as shown in Fig. 3.1, would be:

```
1 blk_left  = FoamBlock:new{grid=grid_left,
2             bndry_labels={south=''w-00'', north=''w-01'', west=''i-00''}}
3 blk_right = FoamBlock:new{grid=grid_left,
4             bndry_labels={south=''w-00'', north=''w-01'', east=''o-00''}}
```

Here, the `FoamBlock` constructor takes the respective grids and then uses the `bndry_labels` table to assign labels to the outwards facing edges, with names `north`, `south`, `west`, and `east` respectively. For 2D grids foamMesh automatically assigns labels to the top and bottom faces. However, for 3D simulations labels also have to be specified for these faces.

## 3.4 OpenFOAM boundary conditions

At the most fundamental level OpenFOAM differentiates between the following types of boundary conditions: `wall` corresponding to solid surfaces, `empty` and `wedge`

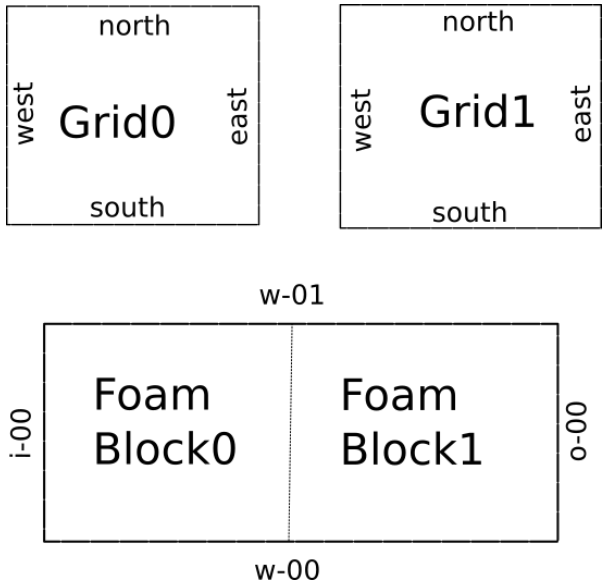



Figure 3.1: Combining grid elements to form a combined mesh. Note, boundary conditions are only defined for boundaries that will be outwards facing after mesh has been combined.

used as the front and rear faces in 2-D planar and 2-D axi-symmetric simulations, `symmetry` for symmetry, and `patch` for everything else. Getting these types right is essential in order to get a correct simulation. Incorrect setting of `wall` or `patch` may result in turbulence models incorrectly calculating wall distance, which may result in low or excessive dissipation of turbulence properties.

foamMesh uses a number of standard labels to assign boundary conditions to the different faces. By re-using the same label, edges (or faces in 3D) from different blocks can be assigned to the same boundary condition. The boundary labels are constructed from a prefix (e.g. w-), which is appended by a two digit number. The available boundary labels and the corresponding boundary type that are assigned automatically, are:

| Label | Type | Description and default settings for boundary conditions. |
|---|---|---|
| w-00 ... w-NN | `wall` | Walls, will be initialised with no-slip wall and appropriate wall functions |
| i-00 ... i-NN | `patch` | Inlets, will be initialised as inlet with prescribed velocity and zeroGradient for pressure |
| o-00 ... o-NN | `patch` | Outlets, will be initialised as outlets with zeroGradient for velocity and prescribed pressure |
| s-00 ... s-NN | `symmetry` | Symmetry planes, will be initialised as symmetry boundary conditions |
| p-00 ... p-NN | `patch` | Generic patch, will be left blank |
| n/a | `empty` | Empty, automatically assigned to front and rear faces for planar 2D geometries |
| n/a | `wedge` | Wedge, automatically assigned to front and rear faced for axi-symmetric 2D geometries |

If required, the boundary types and labels can be modified after mesh generation, by

editing `/case/constant/polyMesh/boundaries`.

When executing foamMesh, a new directory `0_temp/` is created. This contains the template boundary conditions. It is up to the user to copy these across to the `0/` directory and to adjust the boundary conditions to have the correct settings.

## 3.5 Building the grid

The actual OpenFOAM grid is built by executing the command

`foamMesh --job=job` from within the `case/` directory.

Once the command has been executed the on-screen display will report the progress of the mesh joining process before reporting on the different tasks required to combine the individual blocks into a single OpenFOAM mesh. At the final stage checkMesh is executed to assess the quality of the resulting mesh. The checkMesh report will also provide a list containing the number of faces and cooresponding patch names that have been assigned for the mesh. This list should be in agreement with the labels you have assigned to the faces of your foamBlocks. If there are faces with the label `unassigned`, this is an indicator that some outwards facing labels were not supplied in the respective `bndry_labels` table. Boundary conditions defined in `/0` need to include all these ouwards facing patches.

Before viewing the mesh, e.g. in paraFoam, it is essential that the initial fields and boundary conditions corresponding to actual boundary labels are set in `0/`. If this is not done, paraFoam will crash when loading the mesh for visualisation.

## 3.6 Some debugging help

As the code gets exercised, we have been able to identify reoccurring errors or omissions that cause the grid construction to crash. As much as possible we have tried to add self-explanatory error messages to assist you in correcting the code. Below is a summary of these error messages and some suggestions on how to resolve them.

| Error Message | Cause and Suggested action |
|---|---|
| Mesh fails to build | Check that OpenFOAM has been loaded. |
| Number of blocks defined: 0 | No foam blocks have been defined in your lua script. Check that you have defined at least one block. |
| Oops, seem to already have a cell on left | You are trying to join two identical grids, or you are trying to join two grids to one edge of an exiting grid. |
| bad argument # 1 to 'pairs' (table expected got nil) | One of your foamBlocks is missing the `bndry_labels={}` entry. |
| WARNING after checkMesh about unassigned boundaries | This means you have an outwards facing boundary that has not been given a label. The table shows block and corresponding direction. The resulting mesh can still be viewed in paraFoam. |

# 4

# Examples

The following is a brief list of examples of how foamMesh can be employed to use the syntax from the Eilmer4 geometry package to generate corresponding meshes and simulations in OpenFOAM[3]. In the examples we will largely skip the grid generation part, but more information is available in the Geometry User-Guide [1]. The examples have been tested with OpenFOAM 4.x and 5.x, older/different versions some adjustments, in particular when setting boundary conditions is needed.

## 4.1 2D (planar) examples

### 4.1.1 Lid-driven cavity flow

An example of a clipped cavity, as found in Sections 2.1.9 and 2.1.10 of the OpenFOAM manual [4]. You will see that the geometry generation follows the same approach as would be used for an Eilmer simulation, albeit the section configuring the simulation is skipped.

After defining the points and lines, these are used to construct parametric surfaces in the form of `CoonsPatch`. Next, structured grids are added to allow the surfaces to be discretised. In the final step the `FoamBlock` constructur is used to assign the appropriate boundary conditions to the final grid. Done!

To build the mesh one would then run:

```
foamMesh --job=cavity-clipped --verbosity=2
```

adding the `--verbosity=2` option adds some additional outputs, which can be useful for debugging.

```
1  -- An example of a clipped cavity, as found in
2  -- Sections 2.1.9 and 2.1.10 of the OpenFOAM manual.
3  --
4  --             BC=w-01
5  --          f-----g-----h
6  --          |  b3 |  b4 |
7  --  BC=w-00 |     |     | BC=w-00
8  --          c-----d-----e
9  --          |  b0 |
10 --          |     |
11 --          a-----b
12 --          BC=w-00
13 --
14 -- Authors: IJ and RJG
15 -- Date: 2017-06-29
16
17 -- Global settings go first
18 axisymmetric = false
19 turbulence_model = "S-A" -- other option is: "k-epsilon"
20
21 -- Corners of blocks
22 a = Vector3:new{x=0.0, y=0.0}
23 b = Vector3:new{x=0.6, y=0.0}
24 c = Vector3:new{x=0.0, y=0.4}
25 d = Vector3:new{x=0.6, y=0.4}
26 e = Vector3:new{x=1.0, y=0.4}
27 f = Vector3:new{x=0.0, y=1.0}
28 g = Vector3:new{x=0.6, y=1.0}
29 h = Vector3:new{x=1.0, y=1.0}
30
31 -- Lines connecting blocks.
32 ab = Line:new{p0=a, p1=b} -- horizontal line (lowest level)
33 cd = Line:new{p0=c, p1=d}; de = Line:new{p0=d, p1=e} -- horizontal lines (mid level
34 fg = Line:new{p0=f, p1=g}; gh = Line:new{p0=g, p1=h} -- horizontal lines (top level
35 ac = Line:new{p0=a, p1=c}; cf = Line:new{p0=c, p1=f} -- vertical lines (left)
36 bd = Line:new{p0=b, p1=d}; dg = Line:new{p0=d, p1=g} -- vertical lines (mid)
37 eh = Line:new{p0=e, p1=h} -- vertical line (right)
38
39 -- Define patches (which are parametric surfaces, no discretisation at this point.)
```

```
quad0 = CoonsPatch:new{north=cd, east=bd, south=ab, west=ac}
quad1 = CoonsPatch:new{north=fg, east=dg, south=cd, west=cf}
quad2 = CoonsPatch:new{north=gh, east=eh, south=de, west=dg}

-- Define grids. Here's where discretisation is added to a Patch
nx0cells = 12; nx1cells = 8;
ny0cells = 8; ny1cells = 12
grid0 = StructuredGrid:new{psurface=quad0, niv=nx0cells+1, njv=ny0cells+1}
grid1 = StructuredGrid:new{psurface=quad1, niv=nx0cells+1, njv=ny1cells+1}
grid2 = StructuredGrid:new{psurface=quad2, niv=nx1cells+1, njv=ny1cells+1}

-- Lastly, define the blocks.
blk0 = FoamBlock:new{grid=grid0,
                     bndry_labels={west="w-01", south="w-01", east="w-01"}}
blk1 = FoamBlock:new{grid=grid1,
                     bndry_labels={west="w-01", north="w-00"}}
blk2 = FoamBlock:new{grid=grid2,
                     bndry_labels={south="w-01", east="w-01", north="w-00"}}
```

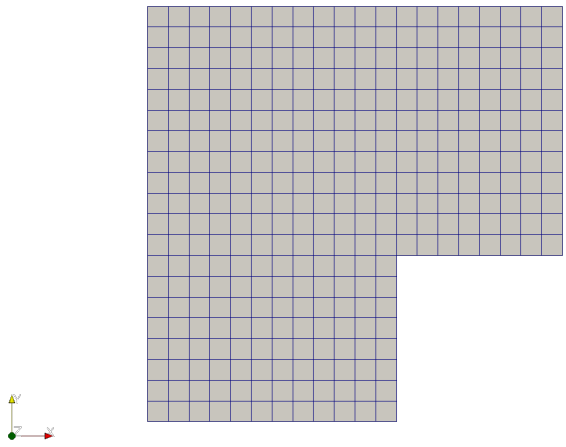


Figure 4.1: Mesh generated for clipped cavity.

### 4.1.2 Axi-symmetric Convergent-Divergent Nozzle

An example of an axi-symmetric convergent-divergent nozzle.

Note, due to current limitations of foamMesh, it is not possible to generate a multi-block mesh with points along the x-axis. A current work-around is to define the points along the centreline to have a very small off-set in the y-direction. For example, $1 \times 10^{-6}\,\mathrm{m}$ (or appropriately lower for smaller geometries) and to define the inward facing edge of the fluid domain as a slip wall.

For some axi-symmetric geometries, checkMesh will identify that the front and rear wedge faces are not planar. This is caused by the precision employed by OpenFOAM. To eliminate this error, edit `/system/controlDict` and increase `writePrecision`.

```
1  -- Mesh for optimisation of convergent-divergent nozzle
2  -- Author: Ingo Jahn
3  -- last modified: 08/04/2017
4
5  axisymmetric = true
6  --########################################
7  --# Create Geometry                      ###
8  --########################################
9  --     +-----------+--\   rt  /--+--------+  R
10 --     |           |   \-+---/   |        |
11 --     |   A0      | A1  |  A2   |   A3   |
12 --     |           |     |       |        |
13 --     +-----------+-----+-------+--------+  r=0
14 --
15 --
16 --     a0---------a1--\        /--a3------a4
17 --     |           |   \-a2---/   |        |
18 --     |   A0      | A1  |   A2   |   A3   |  nr
19 --     |           |     |        |        |
20 --     b0---------b1----b2--------b3------b4
21 --     x0         x1    x2        x3      x4
22 --          n0        n1     n3      n3
23 --
24 -- Bezier Curve Control Points (define as fractions of corner points)
25 --
26 --     a1---c0                      (fixed at R)
27 --
28 --                c1                c1_rf (free to move in r and x)
29 --
30 --                         c2---a2   (fixed at rt)
31 --       c0_xf   c1_xf    c2_xf
32
33 -- ###################
34 -- Input variables to parmetrically define nozzle
35 -- ###################
36 -- Geometric parameters
37 x0 = -0.2
38 x1 = -0.10
39 x3 = 0.1
40 x4 = 0.2
41 R = 0.1
42 Rc = 1e-6
43 x2 = 0.0  -- = (1-OP[0])*x1 + OP[0]*x3
```

```
Rt = 0.05  -- = (1-OP[1])*Rc + OP[1]*R
-- Bezier Curve Control points for a1a2 and a2a3
c0_xf = 0.2 --= OP[2]
c1_xf = 0.5 --= OP[3]
c1_rf = 0.5 --= OP[4]
c2_xf = 0.8 --= OP[5]
-- Bezier Curve Control points for a2a3
d0_xf = 0.2 --= OP[6]
d1_xf = 0.5 --= OP[7]
d1_rf = 0.5 --= OP[8]
d2_xf = 0.8 --= OP[9]

-- Define fixed points
a0 = Vector3:new{x=x0, y=R}
a1 = Vector3:new{x=x1, y=R}
a2 = Vector3:new{x=x2, y=Rt}
a3 = Vector3:new{x=x3, y=R}
a4 = Vector3:new{x=x4, y=R}

b0 = Vector3:new{x=x0, y=Rc}
b1 = Vector3:new{x=x1, y=Rc}
b2 = Vector3:new{x=x2, y=Rc}
b3 = Vector3:new{x=x3, y=Rc}
b4 = Vector3:new{x=x4, y=Rc}

-- define Bezier control points
c0 = Vector3:new{x=((1-c0_xf)*x1+c0_xf*x2), y=R}
c1 = Vector3:new{x=((1-c1_xf)*x1+c1_xf*x2), y=((1-c1_rf)*Rt+c1_rf*R)}
c2 = Vector3:new{x=((1-c2_xf)*x1+c2_xf*x2), y=Rt}
d0 = Vector3:new{x=((1-d0_xf)*x2+d0_xf*x3), y=Rt}
d1 = Vector3:new{x=((1-d1_xf)*x2+d1_xf*x3), y=((1-d1_rf)*Rt+c1_rf*R)}
d2 = Vector3:new{x=((1-d2_xf)*x2+d2_xf*x3), y=R}

-- create Bezier Curves
a1a2 = Bezier:new{points={a1,c0,c1,c2,a2}}
a2a3 = Bezier:new{points={a2,d0,d1,d2,a3}}

-- Define patch (which are parametric surfaces, no discretisation at this point.)
surf = {}
surf[0] = CoonsPatch:new{p00=b0, p10=b1, p11=a1, p01=a0}
surf[1] = CoonsPatch:new{north=a1a2, south=Line:new{p0=b1, p1=b2},
           west=Line:new{p0=b1, p1=a1}, east=Line:new{p0=b2, p1=a2} }
surf[2] = CoonsPatch:new{north=a2a3, south=Line:new{p0=b2, p1=b3},
           west=Line:new{p0=b2, p1=a2}, east=Line:new{p0=b3, p1=a3} }
surf[3] = CoonsPatch:new{p00=b3, p10=b4, p11=a4, p01=a3}

--     a0---------a1--\        /--a3------a4
--     |           |   \-a2---/   |         |
--     |   A0      | A1  |   A2   |   A3    |  nr
--     |           |     |        |         |
--     b0---------b1----b2--------b3------b4
--     x0         x1    x2        x3      x4
--          n0        n1     n3      n3

-- Define 2D grid on patch, clustering can be added if desired
n0=20; n1=30; n2=30; n3=20
nr=20
```

```
101
102 cfr = RobertsFunction:new{end0=false, end1=true, beta=1.05}
103 cf0 = RobertsFunction:new{end0=false, end1=true, beta=1.12}
104 cf1 = RobertsFunction:new{end0=true, end1=false, beta=1.12}
105
106 --cfr = None --RobertsFunction:new{end0=true, end1=true, beta=1.05}
107 grid = {}
108 grid[0] = StructuredGrid:new{psurface=surf[0], niv=n0, njv=nr,
109                 cfList={east=cfr,west=cfr,north=cf0,south=cf0} }
110 grid[1] = StructuredGrid:new{psurface=surf[1], niv=n1, njv=nr,
111                cfList={east=cfr,west=cfr} }
112 grid[2] = StructuredGrid:new{psurface=surf[2], niv=n2, njv=nr,
113                 cfList={east=cfr,west=cfr} }
114 grid[3] = StructuredGrid:new{psurface=surf[3], niv=n3, njv=nr,
115                 cfList={east=cfr,west=cfr,north=cf1,south=cf1} }
116
117 -- Define OpenFoam block (a "grid" with labels)
118 block = {}
119 block[0] = FoamBlock:new{grid=grid[0],
120                     bndry_labels={west="i-00", north="w-00", south="s-00"}}
121 block[1] = FoamBlock:new{grid=grid[1],
122              bndry_labels={north="w-00", south="s-00"}}
123 block[2] = FoamBlock:new{grid=grid[2],
124              bndry_labels={north="w-00", south="s-00"}}
125 block[3] = FoamBlock:new{grid=grid[3],
126              bndry_labels={north="w-00", south="s-00", east="o-00"}}
```

After ruuning `foamMesh --job=con-div-nozzle` you should get the mesh shown in Fig. 4.2. In the current form this shape is not very exciting, but with appropriate optimisation tools that parametrically adjust the nozzle shape, this can be turned into an optimal Mach 2.4 nozzle as shown in Fig. 4.3 (Courtesy of Jianhui Qi).

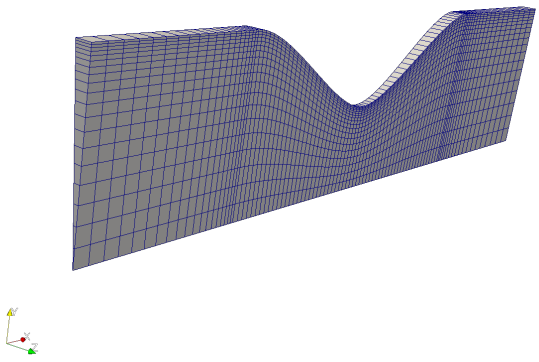
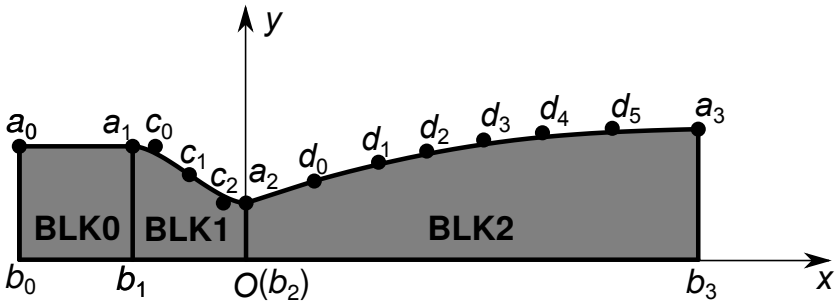
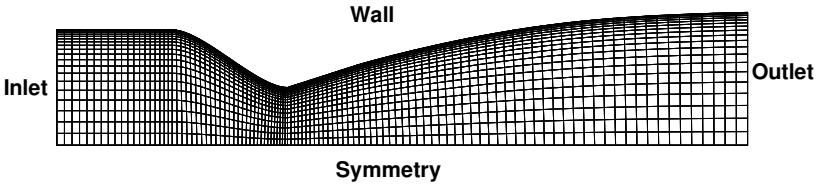


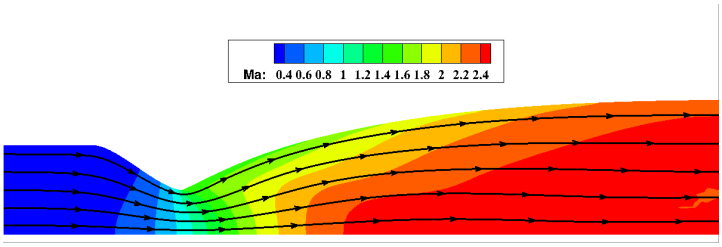
Figure 4.2: Mesh generated for axi-symmetric nozzle. Note the wedge shaped domain, the angle $\Delta\theta$ has been amplified for clarity.




(a) Parameterised nozzle geometry with additional Bezier curve control points.




(b) Mesh and geometry of nozzle after parametric refinement.




(c) Streamlines and contours of Mach number showing unifom and parallel flow.

Figure 4.3: Optimised nozzle shape obtained by linking geometry definition to optimiser, that automatically adjusts Bezier curve control points to achieve optimum performance. (Courtesy of Jianhui Qi)

## 4.2 3D examples

The time required to create a 3D grid grows exponentially compared to a 2D grid. This is largely due to increased complexity, e.g. a block is suddenly defined by twelve edges, rather than just four as in the 2D case. Before attempting to build a 3D grid, make sure you have one or more debugging cubes (available in appendix B). They are an essential tool when trying to work out how blocks connect in 3D. Also, sketch your domain and consider different strategies on how to assemble your grid. For example you might be able to build a 2D grid first and then extrude this to three dimensions... Be warned, even experienced meshers can sink hours into a simple 3D grid!

### 4.2.1 NACA00xx aerofoil 3D

For this example we will make use of the LuaFnPath capability to create a script that will automatically create a high quality mesh for a $NACA00xx$ series aerofoil. Here $xx$ can be any two digit number that defines the wing thickness. To create the grid we first make the simplification that the aerofoil has a sharp trailing edge. This allows simpler grid generation using a C-mesh as we don't have to include treatment of the trailing edge. To generate the 3-D grid for the half wing we start by developing a 2-D grid that sits around an aerofoil cross-section. Using the C-meshing approach, we can mesh the aerofoil by wrapping multiple blocks around the aerofoil, so that the south edge always sits on the aerofoil surface.

To create a nice grid generation file, we start by adding a simple sketch that shows how we arrange our blocks and by defining the parametric settings that define the thickness of the NACA aerofoil, the chord length, the span, the distance to the far-field boundary, and a fraction parameter that sets the position of nodes `a1` and `a3`.

```
1  -- Script for the generation a NACA00xx aerofoil mesh in 3-D.
2  -- This example uses a mix of advanced mesh generation tools,
3  -- so don't be too alarmed if it initially looks intimidating.
4  --
5  -- Make sure you attempt the other meshing tutorials first.
6  --
7  -- Author: Ingo Jahn
8  -- last modified: 18/05/2018
9
10 --#########################################
11 --# Create Geometry                      ###
12 --#########################################
13 --
14 --            ------f3--------N-------t0---N--t1
15 --          /       |                 |        |
16 --         N        |        blk1     |  blk0  |
17 --       /  blk2   -a3----\           /        E
18 --      /         /XXXXXXXX--------- /     S   |
19 --     f2-------a2XXX NACA FOIL XXXXa0--------a4
20 --      \         \XXXXXXXX--------- \     N   |
21 --       \  blk3   -a1----/           \        E
22 --        N         |        blk4     |  blk5  |
23 --          \       |                 |        |
24 --            ------f1--------N-------b0---S--b1
25
26
27 -- ###################
28 -- Input variables to parmetrically define the mesh
29 -- ###################
30 -- Geometric parameters
31 turbulenceModel = 'S-A'
32 thickness = 0.24  -- thickness (in percent) of NACA00XX aerofoil
33 c = 0.2 -- (m) chord length of the wing
34 S = 1.0 -- (m) span of the wing
35 L = 3*c -- (m) distance to far-field boundary
36 frac = 0.4 -- fraction to define position of point a1 and a3 along foil
```

Next we have to define a line that describes the aerfoil surface and the far-field boundary around the front of the wing. As NACA profiles are described by a polynominal, this is done most easily using the `luaFnPath` object. This is achieved by first defining a function `foil(t)`, which can be evaluated to return the `x` and `y` locations along the wing outline. To ensure the line direction suits our blocking structure we start on the lower side of the aerofoil and go in the clockwise direction.

To get the far-field boundary we create another function, called `farField(t)`. This approach allows us to construct the grid in such a way that all grid lines depart from the wing surface in a perpendicular direction. To achieve this we first find a point along the wing surface by evaluating the `foil(t)` function. Next we numerically differentiate the function ( $\frac{dy}{dx} = \frac{y(t+\Delta t)-y(t-\Delta t)}{x(t+\Delta t)-x(t-\Delta t)}$) to obtain the local gradient. Once we know the gradient, we can use this to calculate the local surface normal vector, which is the direction we want grid-lines to point. Finally, we use this normal vector to create the corresponding point along the far-field boundary. As trans-finite-interpolation creates straight grid lines between two opposing lines (as long as the other two lines are straight) this will create a very nice grid.

Once both functions are defined, we can create the `luaFnPath`s and also normalise the lines, so that an equal grid-spacing with respect to arc length along the line is generated.

```
1  -- ####################
2  -- Define Lines that create block boundaries
3  -- ####################
4
5  -- we use a function to decribe the outline of the aerofoil
6  function foil(t)
7     -- function that return x/y positon along profiles as a function
8     -- of paramter t, which start at t=0 at bottom rear, t=0.5 at leading
9     -- edge and and finishes as t=1 at top rear.
10    -- calculate y from NACA polynominal with last value adjusted to close trailing
11    LE=0.005
12    if t < 0.5-LE then -- do lower edge
13       x = 1.-2*t
14       y = -5*thickness * ( 0.2969*x^0.5 - 0.1260*x - 0.3516*x^2 + 0.2843*x^3 - 0.10
15    elseif t > 0.5+LE then -- do upper edge
16       x = (t-0.5)*2
17       y = 5*thickness * ( 0.2969*x^0.5 - 0.1260*x - 0.3516*x^2 + 0.2843*x^3 - 0.103
18    else
19       -- to get good clustering at the leading edge we can approximate the tip by a
20       r = 1.1019 *thickness^2
21       xL = 1.-2*(0.5-LE)
22       yL = -5*thickness * ( 0.2969*xL^0.5 - 0.1260*xL - 0.3516*xL^2 + 0.2843*xL^3 -
23       theta_start = math.atan(yL/(r-xL))
24       t = (t-(0.5-LE))/LE -- re-dscretise t to suit circle
25       y = r *     math.sin(theta_start + t* math.abs(theta_start))
26       x = r *(1 - math.cos(theta_start + t*math.abs(theta_start)))
27    end
28    return {x=x, y=y}
29 end
30 --and finally we can create a paths and normalise them with respect to path length
31 foil_path = LuaFnPath:new{luaFnName="foil"}
32 foil_path_norm = ArcLengthParameterizedPath:new{underlying_path=foil_path}
33
```

```
-- to get the far-field bounding boundary, we are going to do some construction
function farField(t)
   -- find a position along the wing surface
   --F = foil(t) -- we can do this by simply evaluating the foil() function
   F = foil_path_norm(t)  -- here F is a table with entries F.x and F.y
   --print("F,",F.x,F.y)
   -- find the local gradient by numerical differentiation
   delta_t = 0.001
   if t < delta_t then
      tm = t; tp = t+delta_t
   elseif t > 1-delta_t then
      tm = t-delta_t; tp = t
   else
      tm = t-delta_t; tp = t+delta_t
   end
   Fp = foil_path_norm(tp)
   Fm = foil_path_norm(tm)
   delta_x = Fp.x-Fm.x
   delta_y = Fp.y-Fm.y
   -- for best mesh quality we want grid lines to be perpendicular to the
   -- wall. This can be achieved by placing points on the farField line
   -- along a line perpendicular to the surface. Lets find the unit vector
   -- for this perpendicular line.
   X = -delta_y; Y = delta_x
   X = X / (delta_x^2 + delta_y^2)^0.5; Y = Y / (delta_x^2 + delta_y^2)^0.5;
   -- Now find the corresponding far field point by vector addition.
   xval = F.x + X*L
   yval = F.y + Y*L
   return {x=xval, y=yval}
end
-- and to cretae the corresponding path
farField_path_norm = LuaFnPath:new{luaFnName="farField"}
```

To allow generation of the different surfaces we required for a 2-D grid we first directly evaluate the lines to create points that sit on the lines (e.g. `b0`, `a0`, `f1`, ...) and sub-divide the two lines using the `SubRangedPath` function. The remaining points for the downstream blocks are created by relative construction. Using the lines and points we can create the CoonsPatches required for the 2-D grid.

```
-- Having these two paths defined, we can subivide these to create lines used
-- for mesh construction and also to find the points along the lines.
b0 = farField_path_norm(0.)  -- we can simply evaluate the path to get coordinates
f1 = farField_path_norm(frac)
f2 = farField_path_norm(0.5)
f3 = farField_path_norm(1.-frac)
t0 = farField_path_norm(1.)

a0 = foil_path_norm(0.)
a1 = foil_path_norm(frac)
a2 = foil_path_norm(0.5)
a3 = foil_path_norm(1.-frac)

b0f1 = SubRangedPath:new{underlying_path=farField_path_norm, t0=0., t1=frac}
f1f2 = SubRangedPath:new{underlying_path=farField_path_norm, t0=frac, t1=0.5}
f2f3 = SubRangedPath:new{underlying_path=farField_path_norm, t0=0.5, t1=1.-frac}
```

```
f3t0 = SubRangedPath:new{underlying_path=farField_path_norm, t0=1.-frac, t1=1.}
a0a1 = SubRangedPath:new{underlying_path=foil_path_norm, t0=0., t1=frac}
a1a2 = SubRangedPath:new{underlying_path=foil_path_norm, t0=frac, t1=0.5}
a2a3 = SubRangedPath:new{underlying_path=foil_path_norm, t0=0.5, t1=1.-frac}
a3a0 = SubRangedPath:new{underlying_path=foil_path_norm, t0=1.-frac, t1=1.}

-- the remaining points we can define relative to the others
t1 = Vector3:new{x=t0.x+L, y=t0.y}
b1 = Vector3:new{x=b0.x+L, y=b0.y}
a4 = Vector3:new{x=a0.x+L, y=a0.y}

-- Define patch (which are parametric surfaces, no discretisation at this point.)
surf = {}
surf[0] = CoonsPatch:new{p00=a0, p10=a4, p11=t1, p01=t0}
surf[1] = CoonsPatch:new{north=f3t0, south=a3a0,
           west=Line:new{p0=a3, p1=f3}, east=Line:new{p0=a0, p1=t0} }
surf[2] = CoonsPatch:new{north=f2f3, south=a2a3,
           west=Line:new{p0=a2, p1=f2}, east=Line:new{p0=a3, p1=f3} }
surf[3] = CoonsPatch:new{north=f1f2, south=a1a2,
           west=Line:new{p0=a1, p1=f1}, east=Line:new{p0=a2, p1=f2} }
surf[4] = CoonsPatch:new{north=b0f1, south=a0a1,
           west=Line:new{p0=a0, p1=b0}, east=Line:new{p0=a1, p1=f1} }
surf[5] = CoonsPatch:new{p00=b0, p10=b1, p11=a4, p01=a0}

-- We could stop here if we were to build a 2-D mesh. But for a 3-D mesh
-- an extra surface is needed to go on the end of the wing.
-- Note all mesh sections beyond the wing tips have numbers starting with 1x
surf[16] = AOPatch:new{north=a2a3, south=ReversedPath:new{underlying_path=a0a1},
           west=a1a2, east=ReversedPath:new{underlying_path=a3a0} }
```

If we wanted to run a 2-D simulation we could proceed from here to define grid objects and then `FoamBlock`s. However, for a 3-D grid we need to define a further surface, `surf[16]`, which corresponds to the wing tip. Creation of the 3-D grid is conducted in two steps using the `SweptSurfaceVolume` function. First we extrude surfaces $0$ to $5$ using a vector with length $S$ in the z-direction. This vector must coincide with the respective `p00` corner for each surface. Next for the volumes that sit beyond the wing tip ($10$ to $16$, we repeat the same process, but we use two vectors going to $S$ and $S + L$ respectively.

```
-- To create the 3-D mesh we will extrude the surfaces along a vector.
-- This vector will go from the respective p00 points and extrude in the
-- +z direction.
-- Volumes that sit around the wing
volume = {}
volume[0] = SweptSurfaceVolume:new{face0123=surf[0], edge04=Line:new{p0=a0,
                p1=Vector3:new{x=a0.x, y=a0.y, z=a0.z+S}} }
volume[1] = SweptSurfaceVolume:new{face0123=surf[1], edge04=Line:new{p0=a3,
                p1=Vector3:new{x=a3.x, y=a3.y, z=a3.z+S}} }
volume[2] = SweptSurfaceVolume:new{face0123=surf[2], edge04=Line:new{p0=a2,
                p1=Vector3:new{x=a2.x, y=a2.y, z=a2.z+S}} }
volume[3] = SweptSurfaceVolume:new{face0123=surf[3], edge04=Line:new{p0=a1,
                p1=Vector3:new{x=a1.x, y=a1.y, z=a1.z+S}} }
volume[4] = SweptSurfaceVolume:new{face0123=surf[4], edge04=Line:new{p0=a0,
                p1=Vector3:new{x=a0.x, y=a0.y, z=a0.z+S}} }
volume[5] = SweptSurfaceVolume:new{face0123=surf[5], edge04=Line:new{p0=b0,
```

```
                p1=Vector3:new{x=b0.x, y=b0.y, z=b0.z+S}} }

-- Volumes that sit beyond wing tip. Here we set the edge04 to start at y = L
volume[10] = SweptSurfaceVolume:new{face0123=surf[0],
    edge04=Line:new{p0=Vector3:new{x=a0.x, y=a0.y, z=a0.z+S},
        p1=Vector3:new{x=a0.x, y=a0.y, z=a0.z+S+L}} }
volume[11] = SweptSurfaceVolume:new{face0123=surf[1],
    edge04=Line:new{p0=Vector3:new{x=a3.x, y=a3.y, z=a3.z+S},
        p1=Vector3:new{x=a3.x, y=a3.y, z=a3.z+S+L}} }
volume[12] = SweptSurfaceVolume:new{face0123=surf[2],
    edge04=Line:new{p0=Vector3:new{x=a2.x, y=a2.y, z=a2.z+S},
        p1=Vector3:new{x=a2.x, y=a2.y, z=a2.z+S+L}} }
volume[13] = SweptSurfaceVolume:new{face0123=surf[3],
    edge04=Line:new{p0=Vector3:new{x=a1.x, y=a1.y, z=a1.z+S},
        p1=Vector3:new{x=a1.x, y=a1.y, z=a1.z+S+L}} }
volume[14] = SweptSurfaceVolume:new{face0123=surf[4],
    edge04=Line:new{p0=Vector3:new{x=a0.x, y=a0.y, z=a0.z+S},
        p1=Vector3:new{x=a0.x, y=a0.y, z=a0.z+S+L}} }
volume[15] = SweptSurfaceVolume:new{face0123=surf[5],
    edge04=Line:new{p0=Vector3:new{x=b0.x, y=b0.y, z=b0.z+S},
        p1=Vector3:new{x=b0.x, y=b0.y, z=b0.z+S+L}} }
volume[16] = SweptSurfaceVolume:new{face0123=surf[16],
    edge04=Line:new{p0=Vector3:new{x=a1.x, y=a1.y, z=a1.z+S},
        p1=Vector3:new{x=a1.x, y=a1.y, z=a1.z+S+L}} }
```

At this stage we have defined the volume of the entire fluid domain. To finalise the grid we need to define the number of cells for each volume, and also apply some clustering. This is identical to the 2-D case, albeit the parametric volume is passed to `pvolume`, there is a `nkv` that needs to be defined, and there are now twelve edges that can be clustered.

```

--            ------f3--------N-------t0---N--t1
--          /       |                 |        |
--         N        |        blk1     |  blk0  |  ny0
--       /  blk2   -a3----\           /        E
--      /         /XXXXXXXX--------- /     S   |
--     f2-------a2XXX NACA FOIL XXXXa0--------a4
--      \         \XXXXXXXX--------- \     N   |
--       \  blk3   -a1----/           \        E
--         N        |        blk4     |  blk5  |  ny0
--     nx0  \       |                 |        |
--            ------f1--------N-------b0---S--b1
--                            nx0          nx1
--
-- Define number of cells in each block
nx0=61; nx1=30
ny0=40
nz0=50; nz1=20

-- set up refining function
N_refine = 1
nx0 = math.ceil(N_refine*nx0); nx1 = math.ceil(N_refine*nx1)
ny0 = math.ceil(N_refine*ny0)
nz0 = math.ceil(N_refine*nz0); nz1 = math.ceil(N_refine*nz1)
```

```
25
26 -- Define Custer Functions.
27 cfr0 = RobertsFunction:new{end0=true, end1=false, beta=1.02}
28 cfr1 = RobertsFunction:new{end0=false, end1=true, beta=1.02}
29 cfz0 = RobertsFunction:new{end0=false, end1=true, beta=1.08} -- z-direction on wing
30 cfz1 = RobertsFunction:new{end0=true, end1=false, beta=1.05} -- z-direction in far-
31 cfx0 = RobertsFunction:new{end0=true, end1=false, beta=1.03}
32
33 -- Now we can define the grid!!!  (Hope you have debugging cube)
34 -- To get an optimum grid you should add some clustering in the wing tangential dir
35 grid = {}
36 grid[0] = StructuredGrid:new{pvolume=volume[0], niv=nx1, njv=ny0, nkv = nz0,
37             cfList={edge04=cfz0, edge15=cfz0, edge26=cfz0, edge37=cfz0,
38                     edge56=cfr0, edge12=cfr0, edge03=cfr0, edge47=cfr0,
39                     edge01=cfx0, edge32=cfx0, edge76=cfx0, edge45=cfx0} }
40 grid[1] = StructuredGrid:new{pvolume=volume[1], niv=nx0, njv=ny0, nkv = nz0,
41             cfList={edge04=cfz0, edge15=cfz0, edge26=cfz0, edge37=cfz0,
42                     edge56=cfr0, edge12=cfr0, edge03=cfr0, edge47=cfr0} }
43 grid[2] = StructuredGrid:new{pvolume=volume[2], niv=nx0, njv=ny0, nkv = nz0,
44             cfList={edge04=cfz0, edge15=cfz0, edge26=cfz0, edge37=cfz0,
45                     edge56=cfr0, edge12=cfr0, edge03=cfr0, edge47=cfr0} }
46 grid[3] = StructuredGrid:new{pvolume=volume[3], niv=nx0, njv=ny0, nkv = nz0,
47             cfList={edge04=cfz0, edge15=cfz0, edge26=cfz0, edge37=cfz0,
48                     edge56=cfr0, edge12=cfr0, edge03=cfr0, edge47=cfr0} }
49 grid[4] = StructuredGrid:new{pvolume=volume[4], niv=nx0, njv=ny0, nkv = nz0,
50             cfList={edge04=cfz0, edge15=cfz0, edge26=cfz0, edge37=cfz0,
51                     edge56=cfr0, edge12=cfr0, edge03=cfr0, edge47=cfr0} }
52 grid[5] = StructuredGrid:new{pvolume=volume[5], niv=nx1, njv=ny0, nkv = nz0,
53             cfList={edge04=cfz0, edge15=cfz0, edge26=cfz0, edge37=cfz0,
54                     edge56=cfr1, edge12=cfr1, edge03=cfr1, edge47=cfr1,
55                     edge01=cfx0, edge32=cfx0, edge76=cfx0, edge45=cfx0} }
56 grid[10] = StructuredGrid:new{pvolume=volume[10], niv=nx1, njv=ny0, nkv = nz1,
57             cfList={edge04=cfz1, edge15=cfz1, edge26=cfz1, edge37=cfz1,
58                     edge56=cfr0, edge12=cfr0, edge03=cfr0, edge47=cfr0,
59                     edge01=cfx0, edge32=cfx0, edge76=cfx0, edge45=cfx0} }
60 grid[11] = StructuredGrid:new{pvolume=volume[11], niv=nx0, njv=ny0, nkv = nz1,
61             cfList={edge04=cfz1, edge15=cfz1, edge26=cfz1, edge37=cfz1,
62                     edge56=cfr0, edge12=cfr0, edge03=cfr0, edge47=cfr0} }
63 grid[12] = StructuredGrid:new{pvolume=volume[12], niv=nx0, njv=ny0, nkv = nz1,
64             cfList={edge04=cfz1, edge15=cfz1, edge26=cfz1, edge37=cfz1,
65                     edge56=cfr0, edge12=cfr0, edge03=cfr0, edge47=cfr0} }
66 grid[13] = StructuredGrid:new{pvolume=volume[13], niv=nx0, njv=ny0, nkv = nz1,
67             cfList={edge04=cfz1, edge15=cfz1, edge26=cfz1, edge37=cfz1,
68                     edge56=cfr0, edge12=cfr0, edge03=cfr0, edge47=cfr0} }
69 grid[14] = StructuredGrid:new{pvolume=volume[14], niv=nx0, njv=ny0, nkv = nz1,
70             cfList={edge04=cfz1, edge15=cfz1, edge26=cfz1, edge37=cfz1,
71                     edge56=cfr0, edge12=cfr0, edge03=cfr0, edge47=cfr0} }
72 grid[15] = StructuredGrid:new{pvolume=volume[15], niv=nx1, njv=ny0, nkv = nz1,
73             cfList={edge04=cfz1, edge15=cfz1, edge26=cfz1, edge37=cfz1,
74                     edge56=cfr1, edge12=cfr1, edge03=cfr1, edge47=cfr1,
75                     edge01=cfx0, edge32=cfx0, edge76=cfx0, edge45=cfx0} }
76 grid[16] = StructuredGrid:new{pvolume=volume[16], niv=nx0, njv=nx0, nkv = nz1,
77             cfList={edge04=cfz1, edge15=cfz1, edge26=cfz1, edge37=cfz1} }
```

At the final stage we apply `FoamBlock` again, which is used to set the boundary labels used by OpenFOAM. Here the additional keywords `bottom` and `top` need to

considered as we have a 3-D simulation. For the `bottom`, which can be assumed to be the centre of the wing, we are using a symmetry plane. For `top` the outside face we simply add `i-00`, which can be set as a bi-directional inlet.

```
-- Define OpenFoam block (a "grid" with labels)
block = {}
block[0] = FoamBlock:new{grid=grid[0],
                     bndry_labels={north="i-00", east="o-00", bottom="s-00"}}
block[1] = FoamBlock:new{grid=grid[1],
                     bndry_labels={north="i-00", south="w-00", bottom="s-00"}}
block[2] = FoamBlock:new{grid=grid[2],
                     bndry_labels={north="i-00", south="w-00", bottom="s-00"}}
block[3] = FoamBlock:new{grid=grid[3],
                     bndry_labels={north="i-00", south="w-00", bottom="s-00"}}
block[4] = FoamBlock:new{grid=grid[4],
                     bndry_labels={north="i-00", south="w-00", bottom="s-00"}}
block[5] = FoamBlock:new{grid=grid[5],
                     bndry_labels={south="i-00", east="o-00", bottom="s-00"}}
block[10] = FoamBlock:new{grid=grid[10],
                     bndry_labels={north="i-00", east="o-00", top="i-00"}}
block[11] = FoamBlock:new{grid=grid[11],
                     bndry_labels={north="i-00", top="i-00"}}
block[12] = FoamBlock:new{grid=grid[12],
                     bndry_labels={north="i-00", top="i-00"}}
block[13] = FoamBlock:new{grid=grid[13],
                     bndry_labels={north="i-00", top="i-00"}}
block[14] = FoamBlock:new{grid=grid[14],
                     bndry_labels={north="i-00", top="i-00"}}
block[15] = FoamBlock:new{grid=grid[15],
                     bndry_labels={south="i-00", east="o-00", top="i-00"}}
block[16] = FoamBlock:new{grid=grid[16],
                     bndry_labels={top="i-00", bottom="w-01"}}
```

If all of the above has been done correctly the resulting mesh is shown in Fig. 4.4. With the standard settings this mesh has 858 696 hexahedral cells, and as you can see there is still a lot of scope for improvement to enhance mesh quality...

Enjoy simulating this and reviewing the effects of the wing tip vortices. (Note: Computing the grid will take up to 30 min and simulation can be most of a day.)

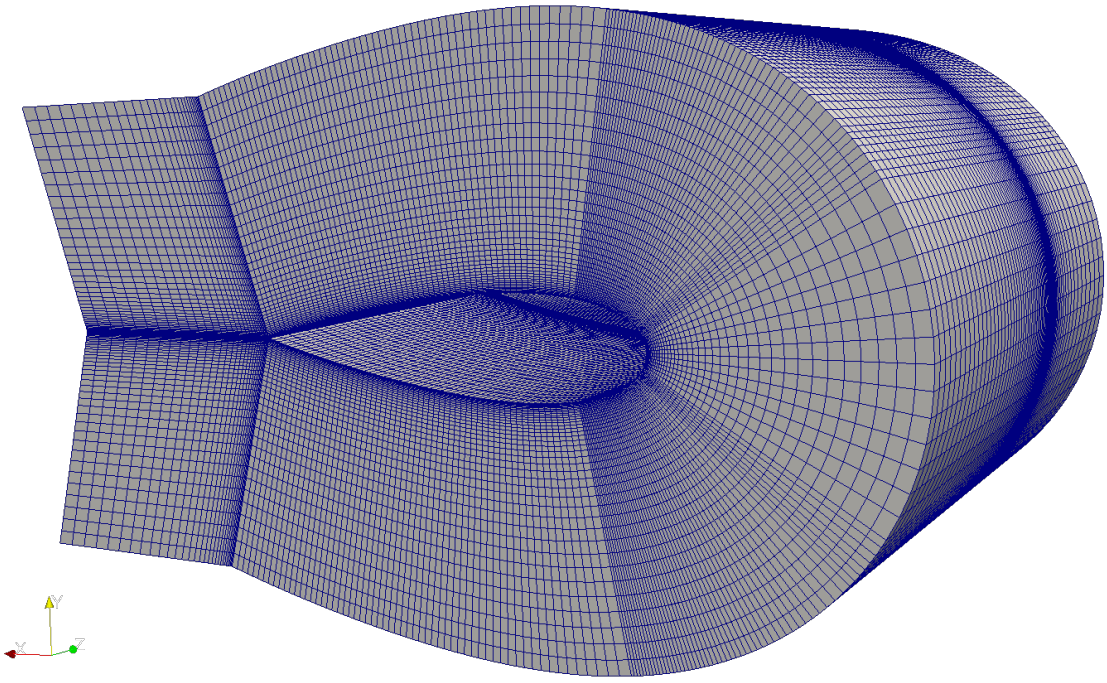


Figure 4.4: Mesh generated for a NACA00xx aerofoil.

# References


[1] Peter A. Jacobs, Rowan J. Gollan, and Ingo Jahn. The Eilmer 4.0 flow simulation program: Guide to the geometry package, for construction of flow paths. School of Mechanical and Mining Engineering Technical Report 2017/25, The University of Queensland, Brisbane, Australia, February 2018.

[2] Peter A. Jacobs and Rowan J. Gollan. The user's guide to the Dlang geometry package for use with the Eilmer4 flow simulation program. School of Mechanical and Mining Engineering Technical Report 2016/19, The University of Queensland, Brisbane, Australia, October 2016.

[3] H. Weller, C. Greenshields, and C. de Rouvray. *OpenFOAM*. The OpenFOAM Foundation Ltd., London, United Kingdom, 5.0 edition, 2018.

[4] The OpenFOAM Foundation. *OpenFOAM v5 User Guide*.

# A

# Useful Linux Commands for OpenFOAM

A short compilation of useful commands to help you manage your OpenFOAM simulations:

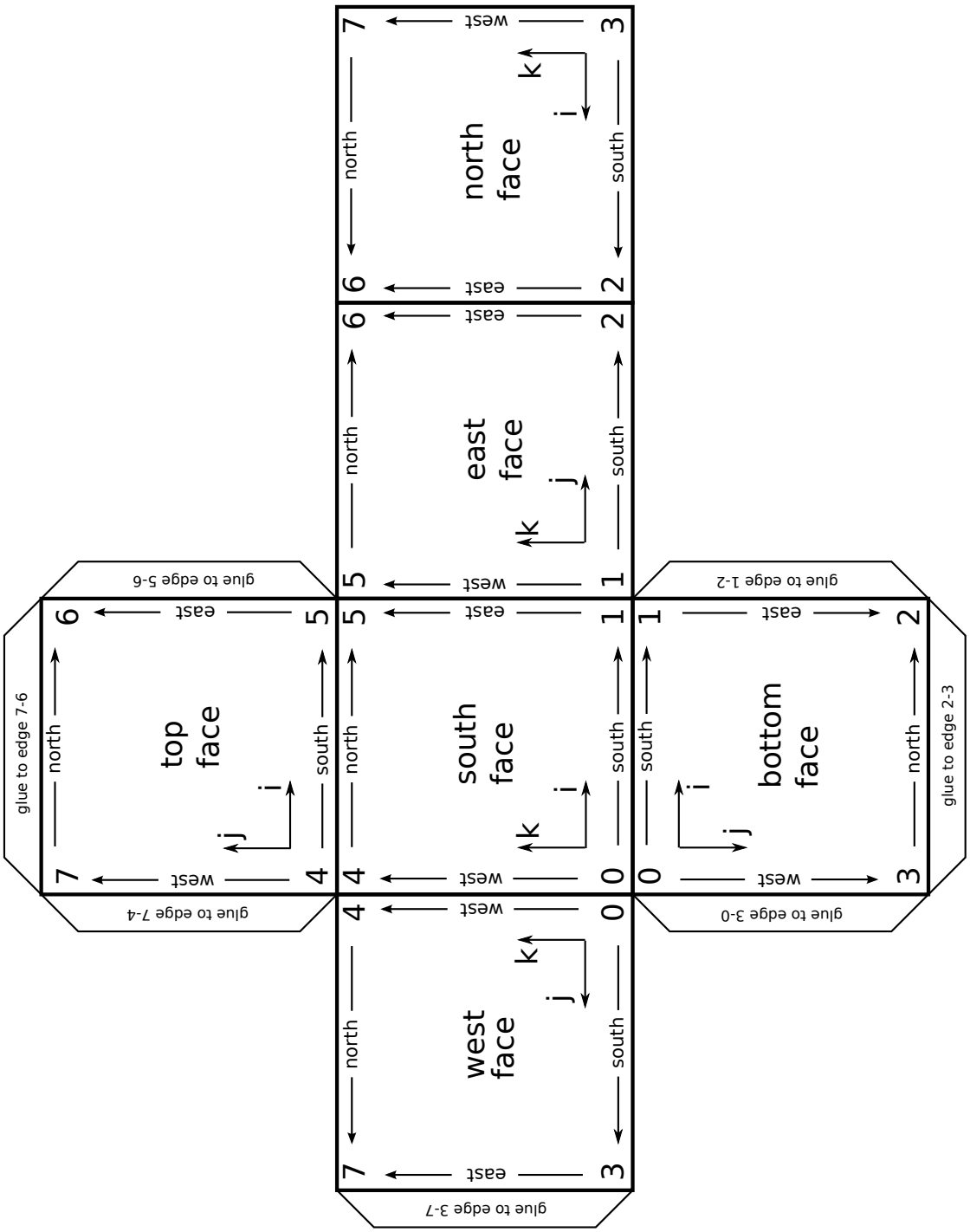
| Command | Description |
|---|---|
| `<command> -help` | Display the usage instructions for a given command. |
| `cp -r $FOAM_TUTORIAL /pathtoFolder .` | Will copy contents of tutorial selected by `$FOAM_TUTORIAL/pathtoFolder` into the current directory. (Dont forget the `.` ) |
| `checkMesh` | Run the build im mesh checking tool to get information about your mesh. |
| `foamListTime -rm` | Removes all time directories apart from the `0` directory. |
| `decomposePar` | Will use information in `decomposeDict` to split your mesh for parallel processing. |
| `mpirun -np <nProcs> <foamExec> -parallel > log &` | To run an application in parallel. `nPocs` must equal number of domains created using `decomposePar` and `foamExec` is the solver command you plan to use (e.g. `simpleFoam`). |
| `reconstructPar` | Will reassemble mesh and fields so that you can open a parallel simulation. |
| `mapFields` | Tool to copy solutions from one mesh to another. |
| `machNo` | Calculates Mach number from velocity field (only works with compressible solvers). |
| `?` | Something else useful... |

There are many more useful commands available online. As a first port of call, look at the OpenFOAM online documentation https://cfd.direct/openfoam/user-guide/.

# B

# Make your own debugging cube

Cut out the development on the reverse of this page, fold along all of the edges and stick the your own cube together. A pair of cubes is very handy for sorting out the specification of connections between structured-grid blocks.


north face
east face
south face
west face
top face
bottom face
glue to edge 5-6
glue to edge 1-2
glue to edge 7-6
glue to edge 2-3
glue to edge 7-4
glue to edge 3-0
glue to edge 3-7
north
south
east
west
i
j
k