Supplement 1

Meshing with blockMesh

Meshing with blockMesh

- Let us take a close look to a `blockMeshDict` dictionary.
- We will use the square cavity case.
- You will find this case in the directory:

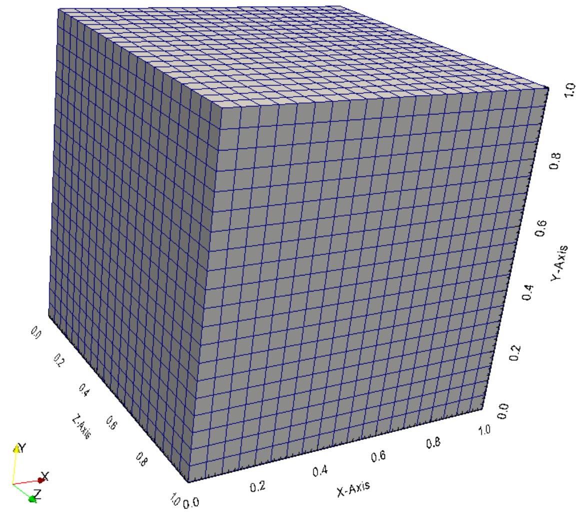
`$BM/101BLOCKMESH/C1`

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

Meshing with blockMesh

What are we going to do?

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

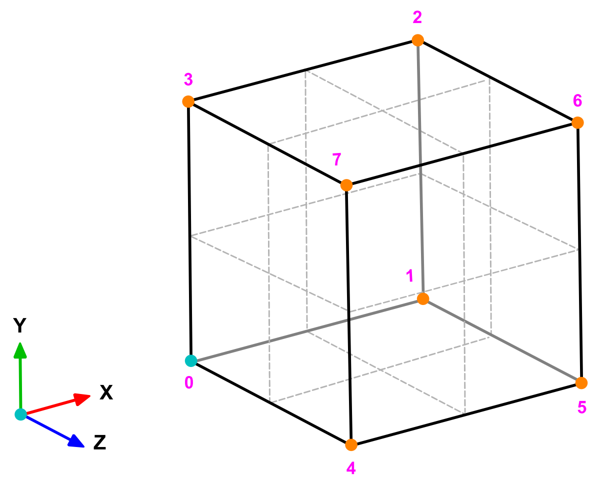


Meshing with blockMesh

The *blockMeshDict* dictionary.

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

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

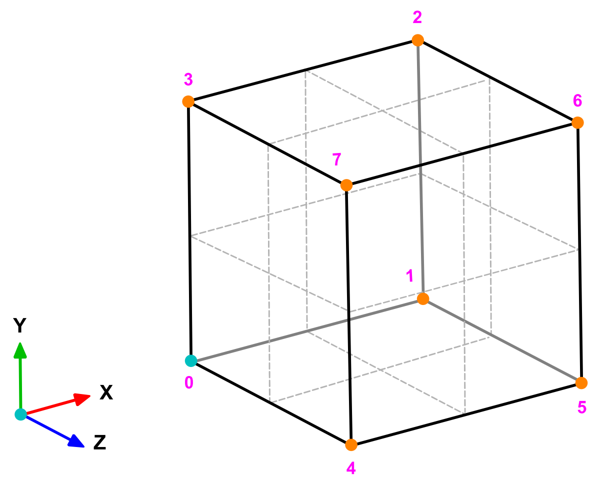


Meshing with blockMesh

The *blockMeshDict* dictionary.

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

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



Meshing with blockMesh

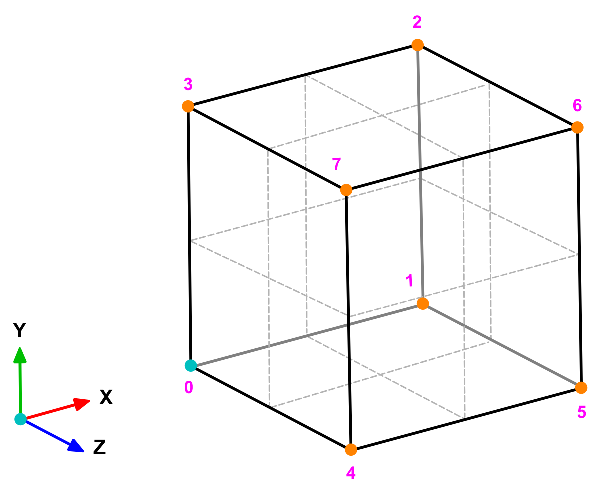
The *blockMeshDict* dictionary.

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

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

ly #calc "$ymax - $ymin";

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

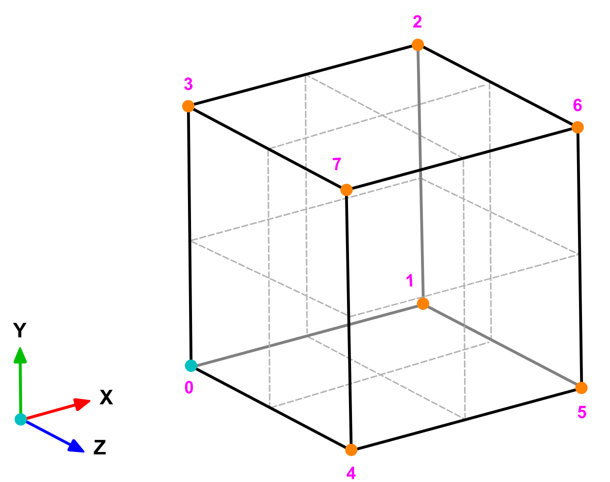


Meshing with blockMesh

The *blockMeshDict* dictionary.

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

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

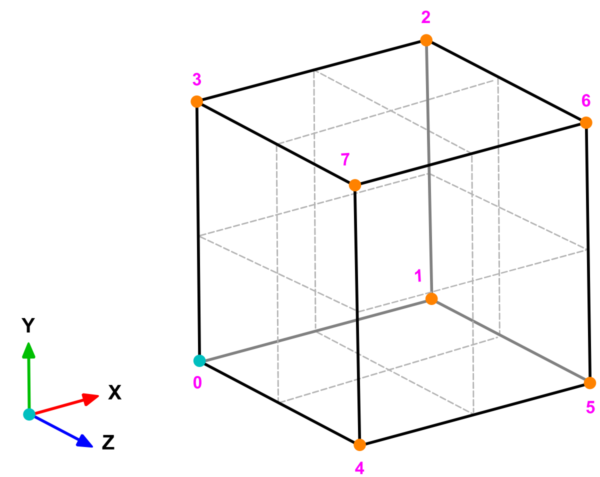


Meshing with blockMesh

The *blockMeshDict* dictionary.

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

- By the way, as this dictionary is designed for blocks with positive vertices coordinates, there is a small catch in the way we compute the length (lines 34-36) and the number of cells (lines 38-40).
  - What will happen if **xmin** is negative?
  - What will happen if **xcells** is negative?
  - What will happen if **xcells** is a float with decimals?
  - Can you find a solution to these small problems?

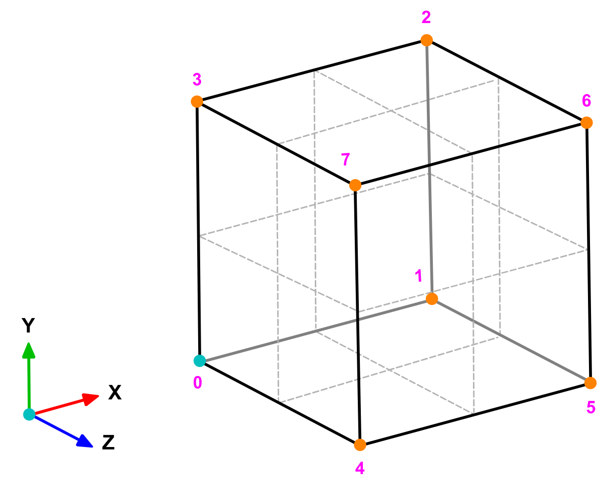


Meshing with blockMesh

The *blockMeshDict* dictionary.

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

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

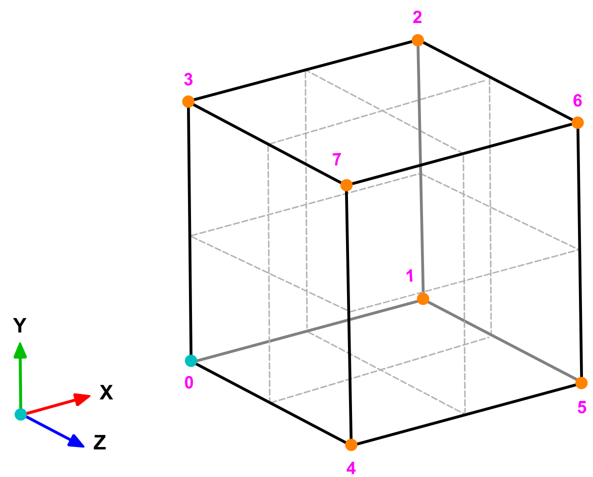


Meshing with blockMesh

The *blockMeshDict* dictionary.

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

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

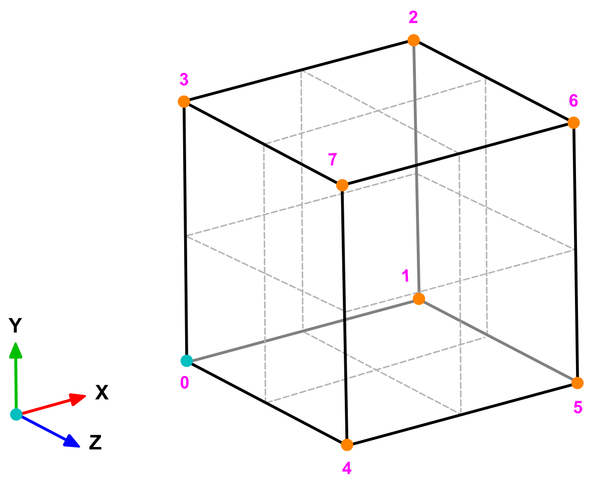


Meshing with blockMesh

The *blockMeshDict* dictionary.

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

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

edges
(

);
```

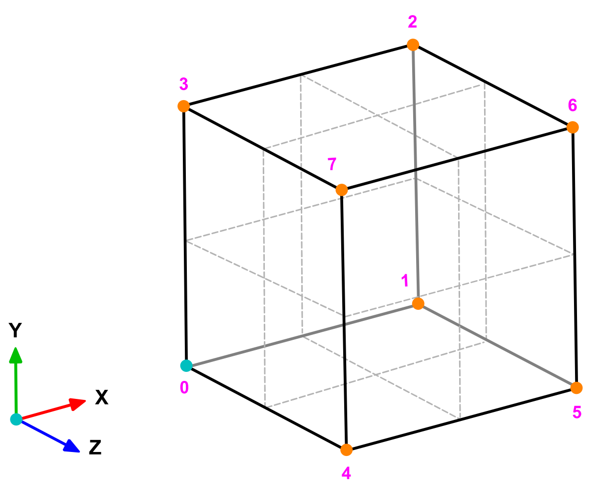


Meshing with blockMesh

The *blockMeshDict* dictionary.

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

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

edges
(

);
```

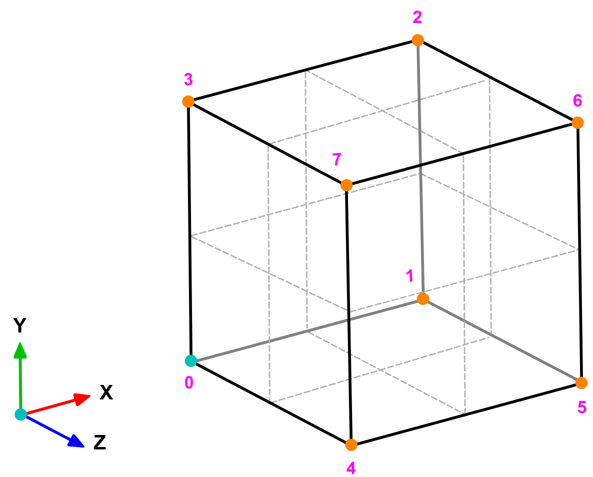


Meshing with blockMesh

The *blockMeshDict* dictionary.

```
boundary
(
    top
    {
        type wall;
        faces
        (
            (3 7 6 2)
        );
    }
    left
    {
        type wall;
        faces
        (
            (0 4 7 3)
        );
    }
    right
    {
        type wall;
        faces
        (
            (2 6 5 1)
        );
    }
    bottom
    {
        type wall;
        faces
        (
            (0 1 5 4)
        );
    }
```

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

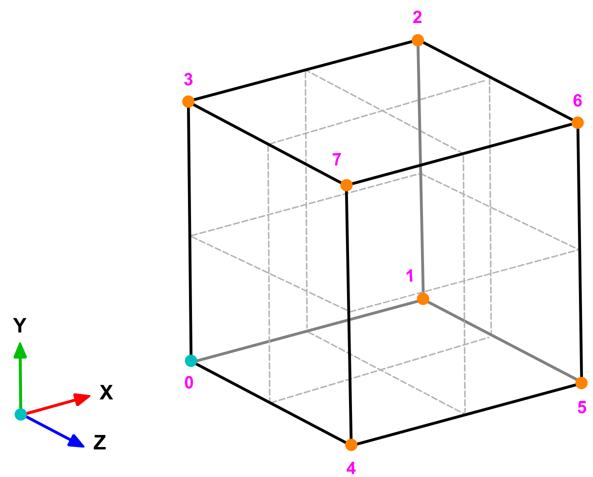


Meshing with blockMesh

The *blockMeshDict* dictionary.

```
76      boundary
77      (
78          top
79          {
80              type wall;
81              faces
82              (
83                  (3 7 6 2)
84              );
85          }
86          left
87          {
88              type wall;
89              faces
90              (
91                  (0 4 7 3)
92              );
93          }
94          right
95          {
96              type wall;
97              faces
98              (
99                  (2 6 5 1)
100             );
101         }
102         bottom
103         {
104             type wall;
105             faces
106             (
107                 (0 1 5 4)
108             );
109         }
```

- In lines 78-85 we define a boundary patch.
- In line 78 we define the patch name **top** (the name is given by the user).
- In line 80 we give a **base type** to the surface patch. In this case **wall** (do not worry we are going to talk about this later).
- In line 83 we give the connectivity list of the vertices that made up the surface patch or face, that is, **(3 7 6 2)**.
- Have in mind that the vertices need to be neighbors and it does not matter if the ordering is clockwise or counterclockwise.

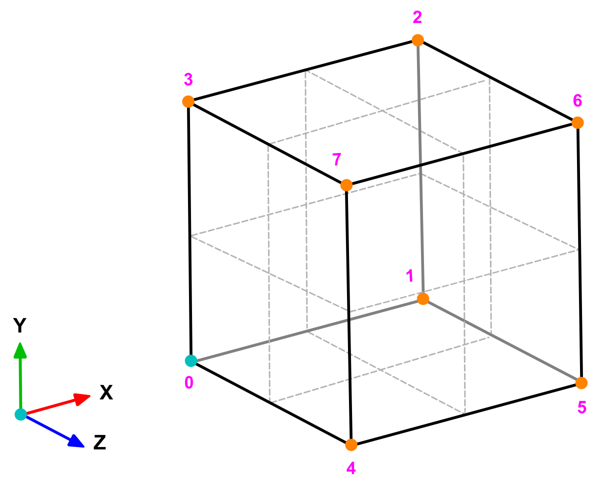


Meshing with blockMesh

The *blockMeshDict* dictionary.

```
76      boundary
77      (
78          top
79          {
80              type wall;
81              faces
82              (
83                  (3 7 6 2)
84              );
85          }
86          left
87          {
88              type wall;
89              faces
90              (
91                  (0 4 7 3)
92              );
93          }
94          right
95          {
96              type wall;
97              faces
98              (
99                  (2 6 5 1)
100             );
101         }
102         bottom
103         {
104             type wall;
105             faces
106             (
107                 (0 1 5 4)
108             );
109         }
```

- Have in mind that the vertices need to be neighbors and it does not matter if the ordering is clockwise or counterclockwise.
- Remember, faces are defined by a list of 4 vertex numbers, e.g., **(3 7 6 2)**.
- In lines 86-93 we define the patch **left**.
- In lines 94-101 we define the patch **right**.
- In lines 102-109 we define the patch **bottom**.



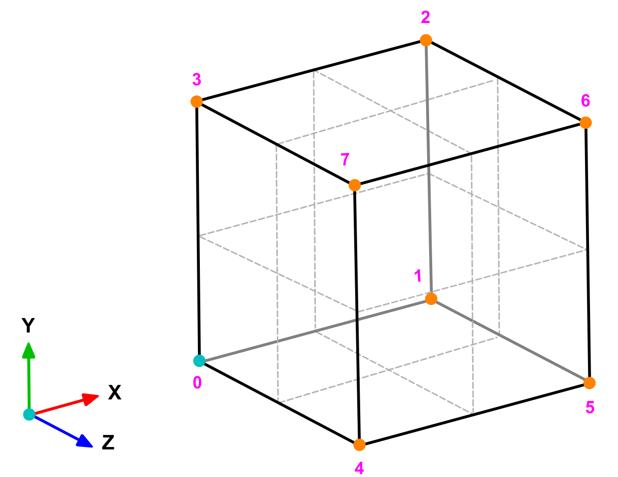
Meshing with blockMesh

The *blockMeshDict* dictionary.

```
110        front
111        {
112            type wall;
113            faces
114            (
115                (4 5 6 7)
116            );
117        }
118        back
119        {
120            type wall;
121            faces
122            (
123                (0 3 2 1)
124            );
125        }
126    );
127
128    mergePatchPairs
129    (
130
131    );
```

- In lines 110-117 we define the patch **front**.
- In lines 118-125 we define the patch **back**.
- You can also group many faces into one patch, for example, instead of creating the patches **front** and **back**, you can group them into a single patch named **backAndFront**, as follows,

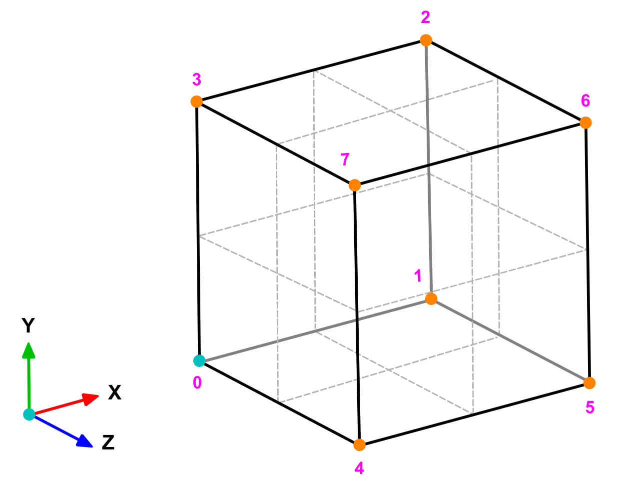
```
backAndFront
{
    type wall;
    faces
    (
        (4 5 6 7)
        (0 3 2 1)
    );
}
```



The *blockMeshDict* dictionary.

```
110         front
111         {
112             type wall;
113             faces
114             (
115                 (4 5 6 7)
116             );
117         }
118         back
119         {
120             type wall;
121             faces
122             (
123                 (0 3 2 1)
124             );
125         }
126     );
127
128     mergePatchPairs
129     (
130
131     );
```

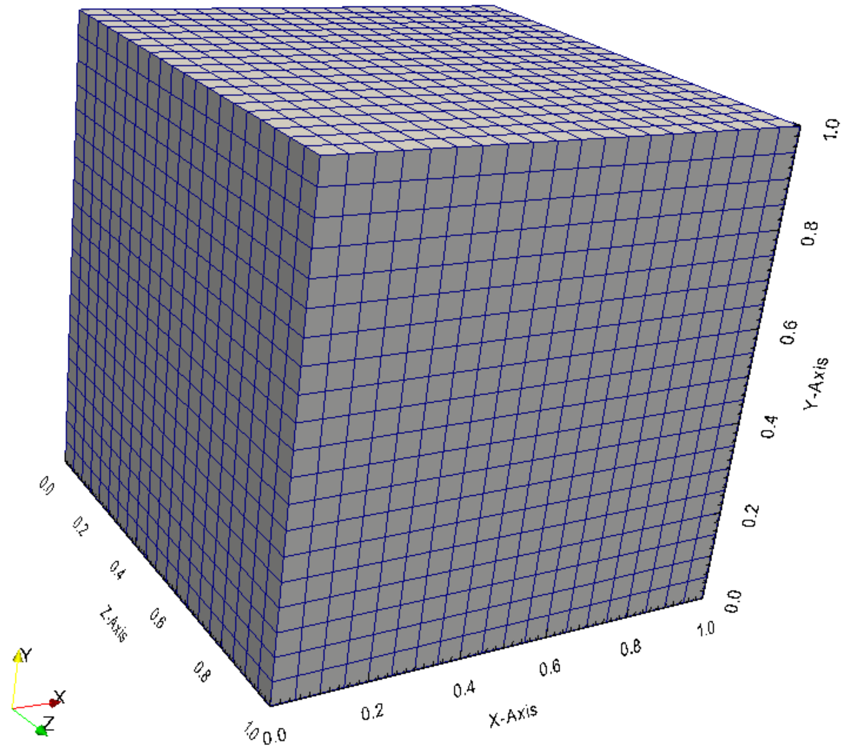
- We can merge blocks in the section **mergePatchPairs** (lines 128-131).
- The block patches to be merged must be first defined in the **boundary** list, `blockMesh` then connect the two blocks.
- In this case, as we have one single block there is no need to merge patches.



Meshing with blockMesh

The *blockMeshDict* dictionary.

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



Meshing with blockMesh

The *constant/polyMesh/boundary* dictionary

```
6
(
    top
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       22800;
    }
    left
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       23200;
    }
    right
    {
        type            empty;
        inGroups        1(wall);
        nFaces          400;
        startFace       23600;
    }
    bottom
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24000;
    }
    front
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24400;
    }
    back
    {
        type            empty;
        inGroups        1(wall);
        nFaces          400;
        startFace       24800;
    }
)
```

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

Meshing with blockMesh

The *constant/polyMesh/boundary* dictionary

```
6
(
    top
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       22800;
    }
    left
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       23200;
    }
    right
    {
        type            empty;
        inGroups        1(wall);
        nFaces          400;
        startFace       23600;
    }
    bottom
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24000;
    }
    front
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24400;
    }
    back
    {
        type            empty;
        inGroups        1(wall);
        nFaces          400;
        startFace       24800;
    }
)
```

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

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

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

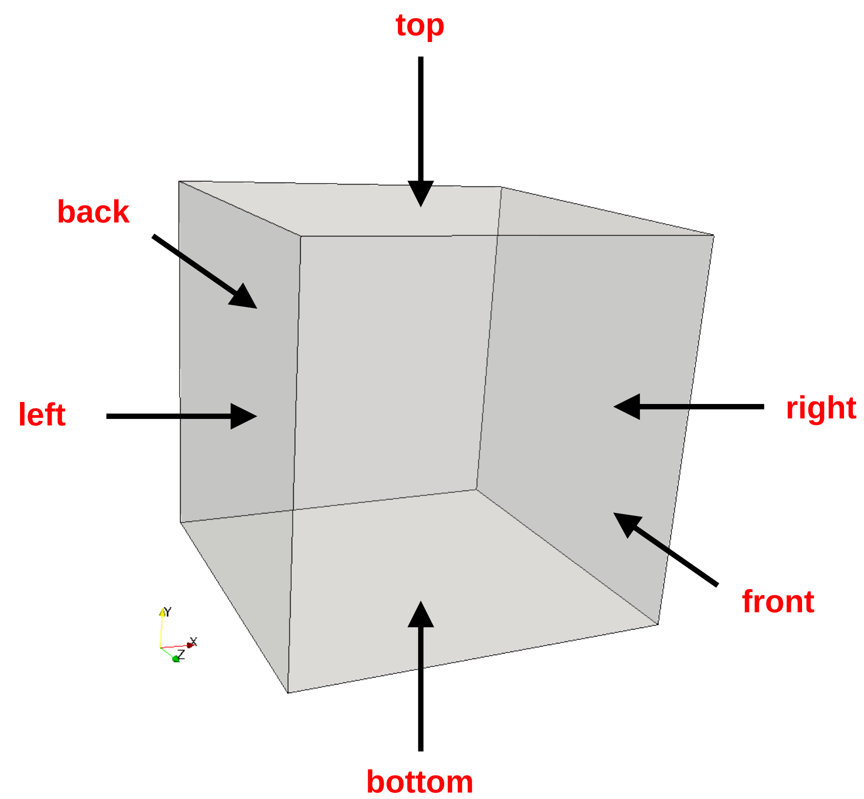
Meshing with blockMesh

The *constant/polyMesh/boundary* dictionary

```
6
(
    top
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       22800;
    }
    left
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       23200;
    }
    right
    {
        type            empty;
        inGroups        1(wall);
        nFaces          400;
        startFace       23600;
    }
    bottom
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24000;
    }
    front
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24400;
    }
    back
    {
        type            empty;
        inGroups        1(wall);
        nFaces          400;
        startFace       24800;
    }
)
```

Number of surface patches

In the list bellow there must be 6 patches definition.



Meshing with blockMesh

The *constant/polyMesh/boundary* dictionary

```
6
(
    top
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       22800;
    }
    left
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       23200;
    }
    right
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       23600;
    }
    bottom
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24000;
    }
    front
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24400;
    }
    back
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24800;
    }
)
```

Name and type of the surface patches

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

Meshing with blockMesh

The *constant/polyMesh/boundary* dictionary

```
6
(
    top
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       22800;
    }
    left
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       23200;
    }
    right
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       23600;
    }
    bottom
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24000;
    }
    front
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24400;
    }
    back
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24800;
    }
)
```

inGroups keyword

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

2(wall wall1)

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

Meshing with blockMesh

The *constant/polyMesh/boundary* dictionary

```
6
(
    top
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       22800;
    }
    left
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       23200;
    }
    right
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       23600;
    }
    bottom
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24000;
    }
    front
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24400;
    }
    back
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24800;
    }
)
```

nFaces and startFace keywords

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

Meshing with blockMesh

Running the case

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

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

Meshing with blockMesh

- Let us take a close look to a *`blockMeshDict`* dictionary to study how to use mesh grading.
- We will use the square cavity case.
- You will find this case in the directory:

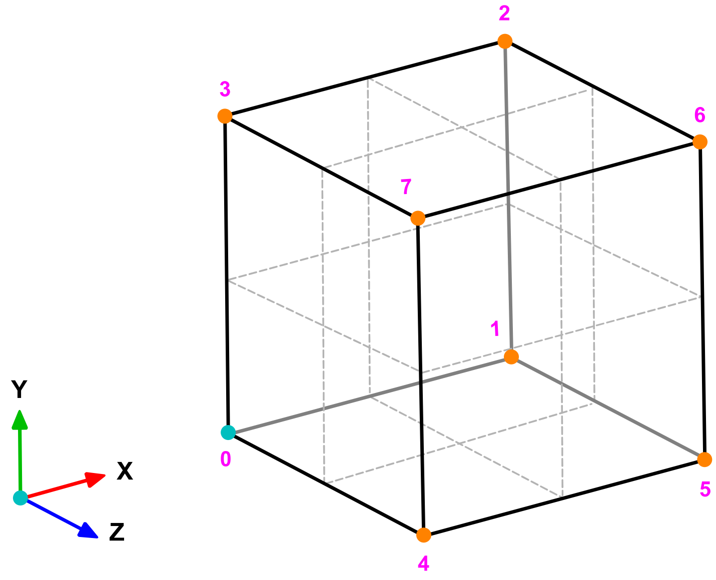
$BM/101BLOCKMESH/C2

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

Meshing with blockMesh

What are we going to do?

- We will use this case to study how to change mesh grading (growth rate).
- You can use this dictionary as a *blockMeshDict* template that you can change automatically according to the dimensions of your domain and the desired cell spacing and growth rate.

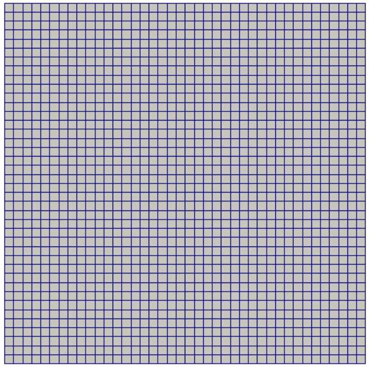


Meshing with blockMesh

The *blockMeshDict* dictionary.

No grading

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



Mesh grading

- To control mesh grading, we use the **simpleGrading** keyword.
- Setting the values to **(1 1 1)** means no grading (uniform mesh).
- A value different from **1** will add grading to the edge, that is, it will cluster more cells towards one extreme of the block.
- Let us take a look at a 2D mesh.

Meshing with blockMesh

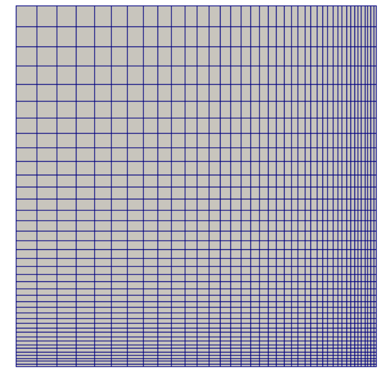
The *blockMeshDict* dictionary.

Unidirectional grading

Stretching in the Y direction (edge 0-3)

```
blocks
(
    hex (0 1 2 3 4 5 6 7) ($xcells $ycells $zcells) simpleGrading (0.125 8 1)
);
```

Stretching in the X direction (edge 0-1)

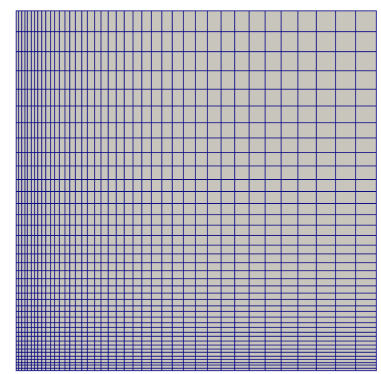


Unidirectional grading

Stretching in the Y direction (edge 0-3)

```
blocks
(
    hex (0 1 2 3 4 5 6 7) ($xcells $ycells $zcells) simpleGrading (8 8 1)
);
```

Stretching in the X direction (edge 0-1)



Meshing with blockMesh

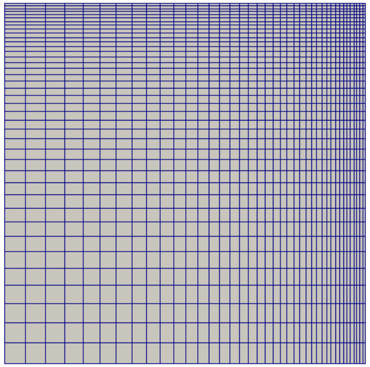
The *blockMeshDict* dictionary.

Unidirectional grading

Stretching in the Y direction (edge 0-3)

```
blocks
(
    hex (0 1 2 3 4 5 6 7) ($xcells $ycells $zcells) simpleGrading (0.125 0.125 1)
);
```

Stretching in the X direction (edge 0-1)

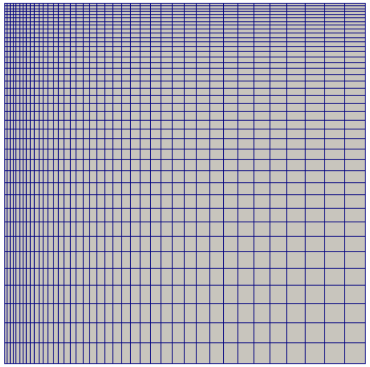


Unidirectional grading

Stretching in the Y direction (edge 0-3)

```
blocks
(
    hex (0 1 2 3 4 5 6 7) ($xcells $ycells $zcells) simpleGrading (8 0.125 1)
);
```

Stretching in the X direction (edge 0-1)

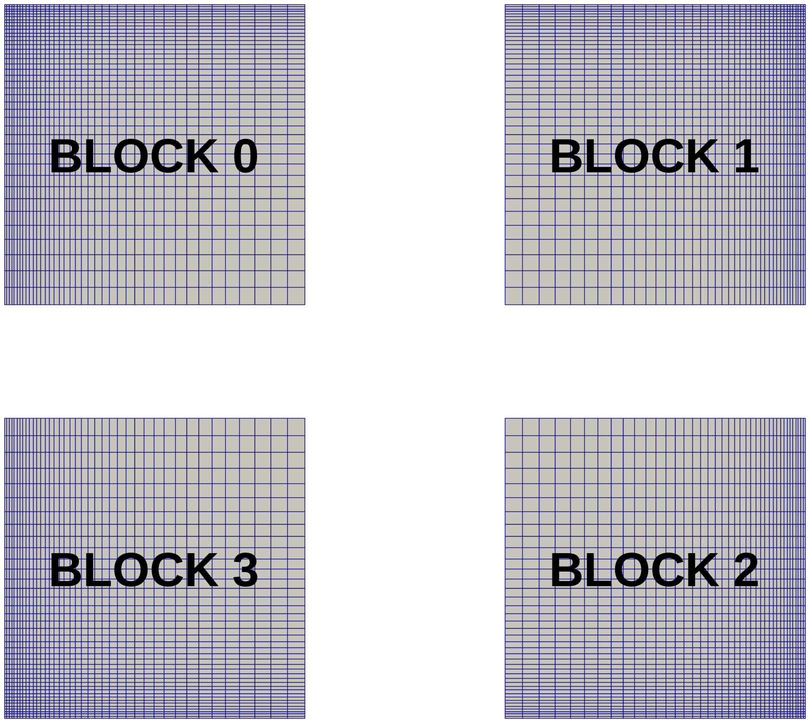


Meshing with blockMesh

The *blockMeshDict* dictionary.

Multi-grading of a block

- Using a single grading to describe mesh grading permits only one-way grading of the block.
- For example, to mesh the square cavity with grading towards all the walls requires four blocks, each one with different grading.
- To reduce complexity and effort we can use multi-grading to control grading within separate divisions of a single block, rather than have to define several blocks with one grading per block.



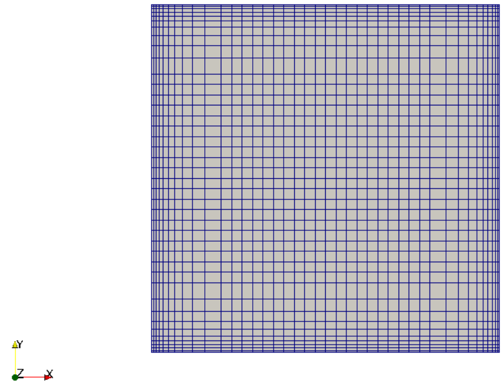
Meshing with blockMesh

The *blockMeshDict* dictionary.

```
64      blocks
65      (
67
68          hex (0 1 2 3 4 5 6 7)   ($xcells $ycells $zcells)
69          simpleGrading
70          (
71              // x-direction expansion ratio
73              (
74                  (0.2 0.25 8)                        //Division 1
75                  (0.6 0.50 1)                        //Division 2
76                  (0.2 0.25 0.125)                    //Division 3
77              )
78
79              // y-direction expansion ratio
81              (
82                  (0.2 0.25 8)
83                  (0.6 0.5 1)
84                  (0.2 0.25 0.125)
85              )
86
87              // z-direction expansion ratio
88              1        //no expansion ratio
89          )
90
91      );
```

Multi-grading of a block

- Let us use multi-grading in the **X**-direction (lines 73-77).
- First, we split the block into 3 divisions in the **X**-direction representing 20% or **0.2** (division 1), 60% or **0.6** (division 2), and 20% or **0.2** (division 3) of the block length.
- Then, we assign 25% (**0.25**) of the total cells in the **X**-direction in divisions 1 and 3, and the remaining 50% (**0.50**) in division 2.
- Finally, we apply a grading of **8** in division 1, a grading of **1** (uniform mesh) in division 2, and a grading of(1/8) in division 3.



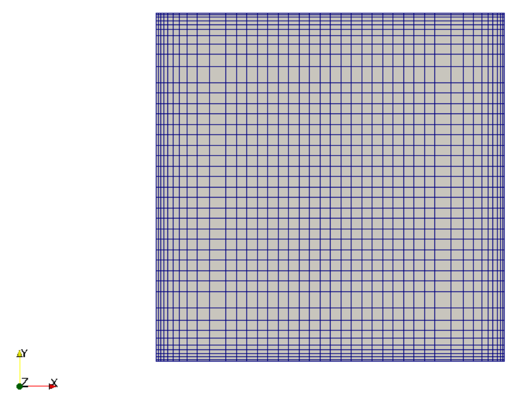
Meshing with blockMesh

The *blockMeshDict* dictionary.

```
64    blocks
65    (
67
68        hex (0 1 2 3 4 5 6 7)  ($xcells $ycells $zcells)
69        simpleGrading
70        (
71            // x-direction expansion ratio
73            (
74                (0.2 0.25 8)                //Division 1
75                (0.6 0.50 1)                //Division 2
76                (0.2 0.25 0.125)            //Division 3
77            )
78
79            // y-direction expansion ratio
81            (
82                (0.2 0.25 8)
83                (0.6 0.5 1)
84                (0.2 0.25 0.125)
85            )
86
87            // z-direction expansion ratio
88            1        //no expansion ratio
89        )
90
91    );
```

Multi-grading of a block

- Let us use multi-grading in the **Y**-direction (lines 81-85).
- First, we split the block into 3 divisions in the **Y**-direction representing 20% or **0.2** (division 1), 60% or **0.6** (division 2), and 20% or **0.2** (division 3) of the block length.
- Then, we assign 25% (**0.25**) of the total cells in the **Y**-direction in divisions 1 and 3, and the remaining 50% (**0.50**) in division 2.
- Finally, we apply a grading of **8** in division 1, a grading of **1** (uniform mesh) in division 2, and a grading of(1/8) in division 3.



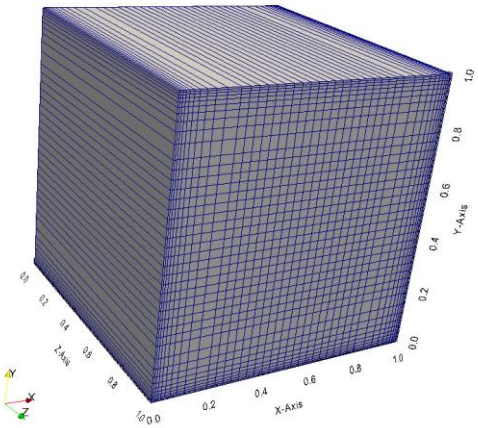
Meshing with blockMesh

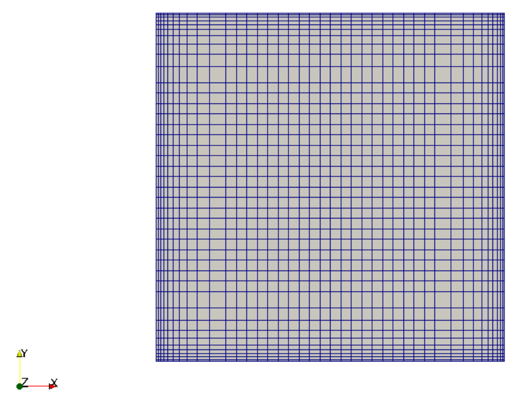
The *blockMeshDict* dictionary.

```
64    blocks
65    (
67
68        hex (0 1 2 3 4 5 6 7)  ($xcells $ycells $zcells)
69        simpleGrading
70        (
71            // x-direction expansion ratio
73            (
74                (0.2 0.25 8)                //Division 1
75                (0.6 0.50 1)                //Division 2
76                (0.2 0.25 0.125)            //Division 3
77            )
78
79            // y-direction expansion ratio
81            (
82                (0.2 0.25 8)
83                (0.6 0.5 1)
84                (0.2 0.25 0.125)
85            )
86
87           // z-direction expansion ratio
88            1      //no expansion ratio
89        )
90
91    );
```

Multi-grading of a block

- Finally, as the mesh is 2D, we do not need to add grading in the **Z**-direction (line 88).





Meshing with blockMesh

Running the case

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

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

Meshing with blockMesh

- Let us take a close look to a `blockMeshDict` dictionary to study how to create multiple blocks.
- We will use the square cavity case.
- You will find this case in the directory:

$BM/101BLOCKMESH/C3

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

Meshing with blockMesh

What are we going to do?

- We will use this case to take a close look at a `blockMeshDict` dictionary.
- We will study how to work with multiple blocks.
- When working with multiples blocks, we need to deal with the common face between blocks. If we do not connect these blocks, `blockMesh` will create a boundary patch and we will need to assign a boundary condition to this patch.
- When we connect the blocks, `blockMesh` will create an internal face (therefore we do not need to assign a boundary condition to the face).
- There are two ways to connect blocks, using face matching and face merging.
- Hereafter we are going to study **face merging**.

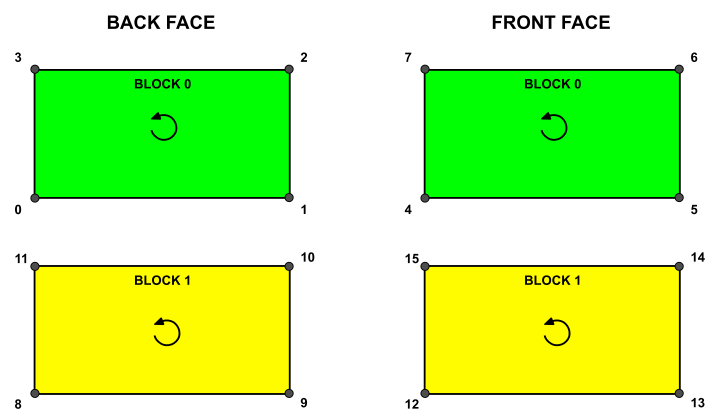
Meshing with blockMesh

The *blockMeshDict* dictionary.

```
19      xmin 0;
20      xmax 1;
21      ymin 0.5;
22      ymax 1;
23      zmin 0;
24      zmax 1;
25
26      ymin2 0;
27      ymax2 0.5;
28
29      xcells 20;
30      ycells 10;
31      zcells 1;
39
45      vertices
46      (
47      //BLOCK 0
48          ($xmin  $ymin  $zmin)      //0
49          ($xmax  $ymin  $zmin)      //1
50          ($xmax  $ymax  $zmin)      //2
51          ($xmin  $ymax  $zmin)      //3
52          ($xmin  $ymin  $zmax)      //4
53          ($xmax  $ymin  $zmax)      //5
54          ($xmax  $ymax  $zmax)      //6
55          ($xmin  $ymax  $zmax)      //7
56
57      //BLOCK 1
58          ($xmin  $ymin2  $zmin)     //8
59          ($xmax  $ymin2  $zmin)     //9
60          ($xmax  $ymax2  $zmin)     //10
61          ($xmin  $ymax2  $zmin)     //11
62          ($xmin  $ymin2  $zmax)     //12
63          ($xmax  $ymin2  $zmax)     //13
64          ($xmax  $ymax2  $zmax)     //14
65          ($xmin  $ymax2  $zmax)     //15
76      );
```

Multiple blocks – Face merging

- To do a mesh with multiple blocks we proceed in the same as we have done so far.
- When using face merging, we need to define all the vertices that made up each block.
- In lines 19-27 we use macro syntax to declare the variables that we will use to define the vertices.
- In lines 29-31 we use macro syntax to define the number of cells in each direction. As this is a 2D case there is only one cell in the **Z**-direction.
- In lines 45-76 we use macro syntax to define the vertices that made up each block.



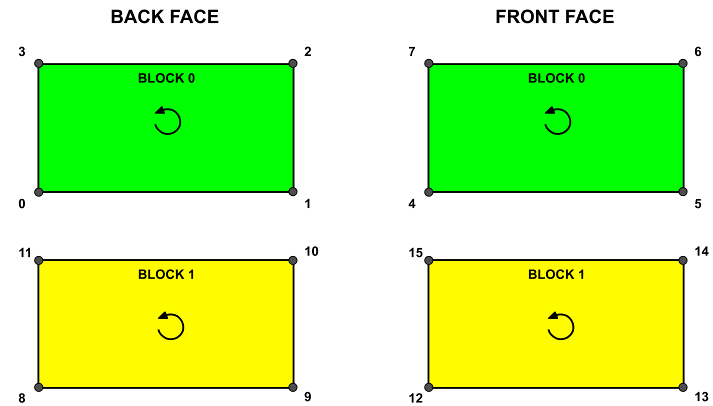
Meshing with blockMesh

The *blockMeshDict* dictionary.

```
78      blocks
79      (
80          hex (0 1 2 3 4 5 6 7) ($xcells $ycells $zcells) simpleGrading (1 1 1)                     //BLOCK 0
81          hex (8 9 10 11 12 13 14 15) ($xcells $ycells $zcells) simpleGrading (1 1 1)               //BLOCK 1
82          //hex (8 9 10 11 12 13 14 15) (40 $ycells $zcells) simpleGrading (1 1 1)
83      );
```

Multiple blocks – Face merging

- In lines 78-83, we define the blocks.
- In line 80, **(0 1 2 3 4 5 6 7)** are the vertices used to define **block 0** (the top block).
- Remember, the first vertex in the list represents the origin of the coordinate system (vertex **0** in this case). Starting from this vertex, we construct the block topology. So, in this case, the first part of the block is made up by vertices **0 1 2 3** and the second part of the block is made up by vertices **4 5 6 7** (notice that we start from vertex **4** which is the projection in the **Z**-direction of vertex **0**).
- **($xcells $ycells $zcells)** is the number of mesh cells in each direction (**X Y Z**). Notice that we are using macro syntax.
- **simpleGrading (1 1 1)** is the grading or mesh stretching in each direction (**X Y Z**), in this case the mesh is uniform.



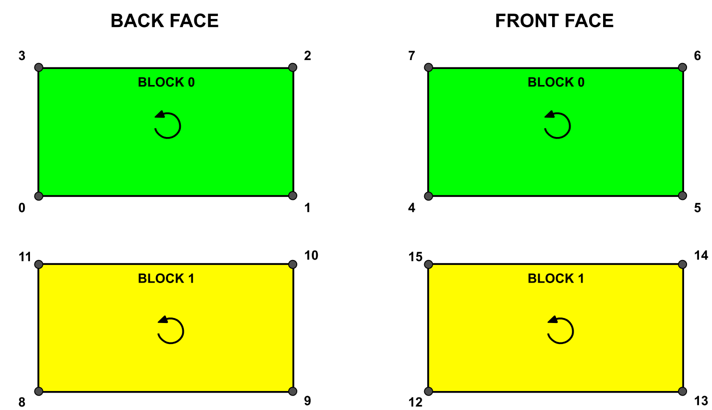
Meshing with blockMesh

The *blockMeshDict* dictionary.

```
78     blocks
79     (
80         hex (0 1 2 3 4 5 6 7) ($xcells $ycells $zcells) simpleGrading (1 1 1)           //BLOCK 0
81         hex (8 9 10 11 12 13 14 15) ($xcells $ycells $zcells) simpleGrading (1 1 1)     //BLOCK 1
82         //hex (8 9 10 11 12 13 14 15) (40 $ycells $zcells) simpleGrading (1 1 1)
83     );
```

Multiple blocks – Face merging

- In line 81, **(8 9 10 11 12 13 14 15)** are the vertices used to define **block 1** (the bottom block).
- The first vertex in the list represents the origin of the coordinate system (vertex 8 in this case). Starting from this vertex, we construct the block topology. So, in this case, the first part of the block is made up by vertices 8 9 10 11 and the second part of the block is made up by vertices 12 13 14 15 (notice that we start from vertex 12 which is the projection in the **Z**-direction of vertex 8).
- **($xcells $ycells $zcells)** is the number of mesh cells in each direction (**X Y Z**). Notice that we are using macro syntax.
- **simpleGrading (1 1 1)** is the grading or mesh stretching in each direction (**X Y Z**), in this case the mesh is uniform.



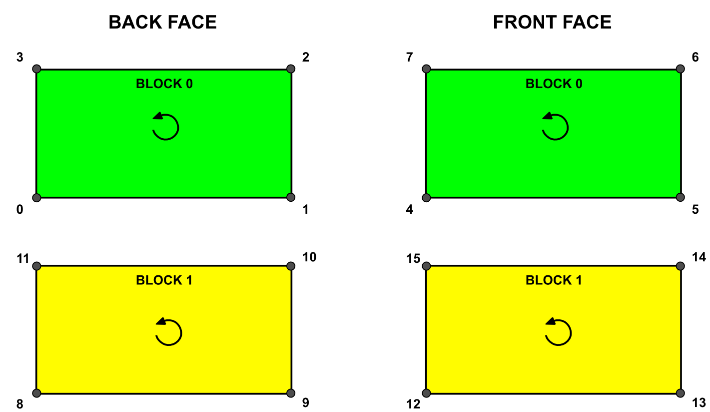
Meshing with blockMesh

The *blockMeshDict* dictionary.

```
91      boundary
92      (
93          movingWall
94          {
95              type wall;
96              faces
97              (
98                  (3 7 6 2)
99              );
100         }
101         fixedWalls
102          {
103              type wall;
104              faces
105             (
106                 (0 4 7 3)
107                 (2 6 5 1)
109                 (11 15 12 8)
110                 (10 14 13 9)
111                 (8 9 13 12)
112             );
113         }

131         back
132         {
133             type empty;
134             faces
135             (
136                 (0 3 2 1)
137                 (8 11 10 9)
138             );
139         }



148         }
```

Multiple blocks – Face merging

- In lines 91-148 we define the boundary patches of the domain.
- We are defining the external patches.



Meshing with blockMesh

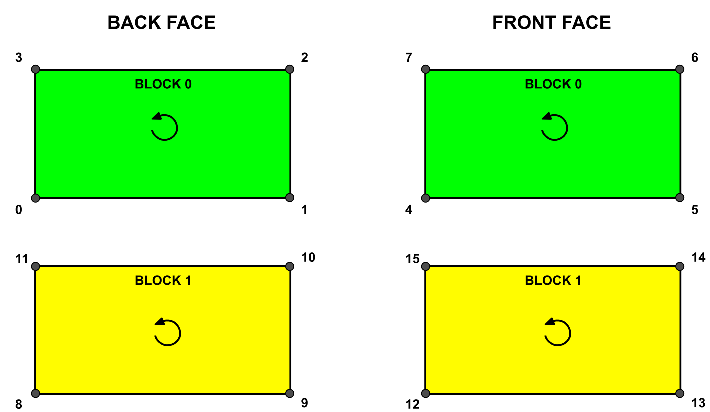
The *blockMeshDict* dictionary.

```
140         front
141         {
142             type empty;
143             faces
144             (
145                 (4 5 6 7)
146                 (12 15 14 13)
147             );
148         }

152         interface1
153         {
154             type wall;
155             faces
156             (
157                 (0 1 5 4)
158             );
159         }
160
161         interface2
162         {
163             type wall;
164             faces
165             (
166                 (11 10 14 15)
167             );
168         }
169     );
170
171     mergePatchPairs
172     (
173         (interface1 interface2)
174     );
```

Master Slave

Multiple blocks – Face merging

- In lines 152-168 we define the boundary patches common to each block (interfaces).
- In this case we need to use **mergePatchPairs** to create an internal face, otherwise OpenFOAM® will see this patch as a boundary patch.
- To merge patches, we need to define them in the section **boundary** of the *blockMeshDict* dictionary.
- In line 173 we merge the patches. The first entry corresponds to the master patch and the second entry is the slave patch.

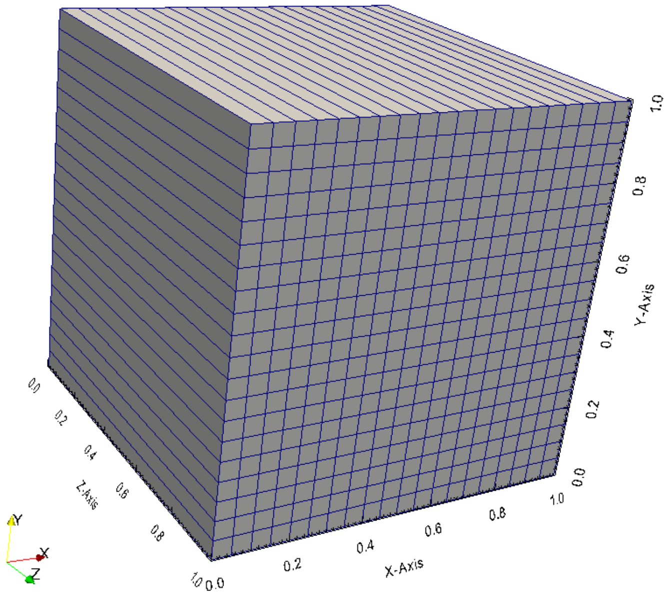


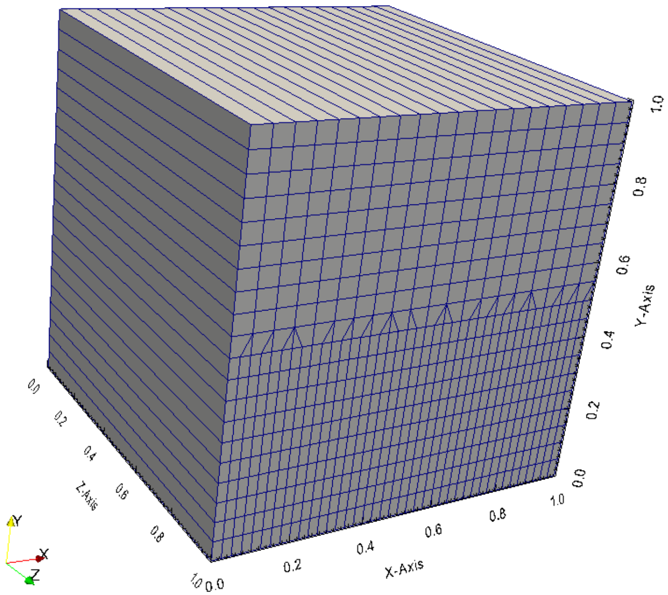
Meshing with blockMesh

The `blockMeshDict` dictionary.

Multiple blocks – Face merging

- The advantage of using **face merging** instead of **face matching**, is that we can use blocks with different grading and number of cells.
- If the blocks are different, `blockMesh` will modify the block that owns the slave patch in order to have a conforming mesh.
- The block that owns the master patch remains unchanged.





Meshing with blockMesh

Running the case

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

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

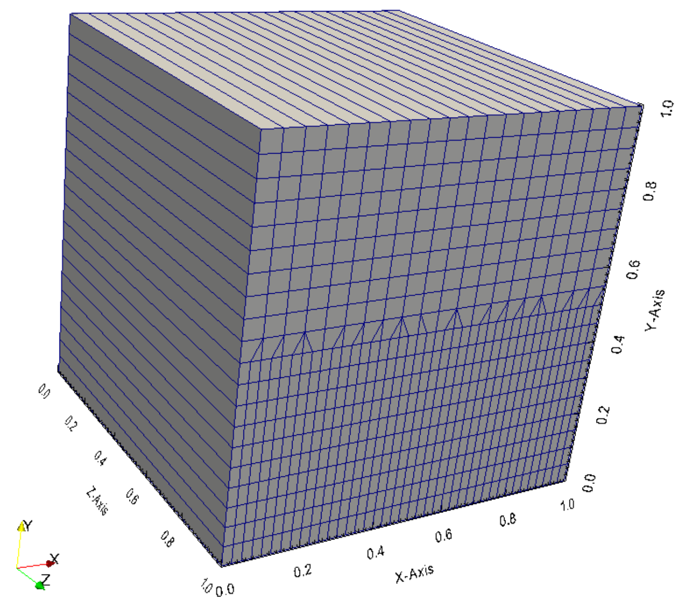
Meshing with blockMesh

The *blockMeshDict* dictionary.

```
78      blocks
79      (
80          hex (0 1 2 3 4 5 6 7) ($xcells $ycells $zcells) simpleGrading (1 1 1)                  //BLOCK 0
81          //hex (8 9 10 11 12 13 14 15) ($xcells $ycells $zcells) simpleGrading (1 1 1)
82          hex (8 9 10 11 12 13 14 15) (40 $ycells $zcells) simpleGrading (1 1 1)                  //BLOCK 1
83      );
```

Multiple blocks – Face merging

- To have different blocks, we changed the number of cells in the **X**-direction of the bottom block (line 82).
- The definition of the block topology remains unchanged, i.e., **(8 9 10 11 12 13 14 15).**
- Also, the grading does not change. If you want, you can use a non-uniform grading.
- Have in mind that the mesh will no longer be 2D because `blockMesh` will add cells to make the blocks conforming. To get the 2D mesh, you will need to use the utility `extrudeMesh`, which reads the *extrudeMeshDict* dictionary.
- Type in the terminal,
  - `$> extrudeMesh`



Meshing with blockMesh

The `extrudeMeshDict` dictionary.

- The utility `extrudeMesh` will create a 2D mesh by projecting the source patch into the exposed patch.
- To create the 2D mesh, you will need to use 1 layer (**nLayers**).
- It is also recommended to set the extrusion **thickness** to 1.

```
constructFrom patch;

sourceCase ".";

sourcePatches (back);

exposedPatchName front;

extrudeModel linearNormal

nLayers 1;

linearNormalCoeffs
{
    thickness 1;
}

mergeFaces false;
```

Name of source patch

Name of the mirror patch

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

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

Meshing with blockMesh

- Let us take a close look to a `blockMeshDict` dictionary to study how to create multiple blocks.
- We will use the square cavity case.
- You will find this case in the directory:

$BM/101BLOCKMESH/C4

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

Meshing with blockMesh

What are we going to do?

- We will use this case to take a close look at a *`blockMeshDict`* dictionary.
- We will study how to work with multiple blocks.
- When working with multiples blocks, we need to deal with the common face between blocks. If we do not connect these blocks, `blockMesh` will create a boundary patch and we will need to assign a boundary condition.
- When we connect the blocks, `blockMesh` will create an internal face (therefore we do not need to assign a boundary condition to the face).
- There are two ways to connect blocks, using face matching and face merging.
- Hereafter we are going to study **face matching**.

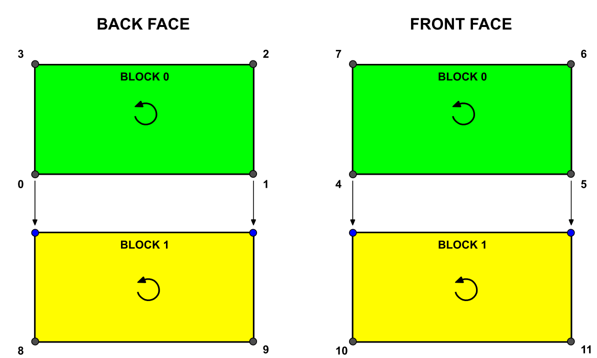
Meshing with blockMesh

The *blockMeshDict* dictionary.

```
19      xmin 0;
20      xmax 1;
21      ymin 0.5;
22      ymax 1;
23      zmin 0;
24      zmax 1;
25
26      ymin2 0;
27      ymax2 0.5;
28
29      xcells 20;
30      ycells 10;
31      zcells 1;
39
45      vertices
46      (
47      //BLOCK 0
48          ($xmin  $ymin  $zmin)       //0
49          ($xmax  $ymin  $zmin)       //1
50          ($xmaax  $ymax  $zmin)      //2
51          ($xmin  $ymax  $zmin)       //3
52          ($xmin  $ymin  $zmax)       //4
53          ($xmax  $ymin  $zmax)       //5
54          ($xmax  $ymax  $zmax)       //6
55          ($xmin  $ymax  $zmax)       //7
56
57      //BLOCK 1
58          ($xmin  $ymin2  $zmin)      //8
59          ($xmax  $ymin2  $zmin)      //9
60          ($xmin  $ymin2  $zmax)      //10
61          ($xmax  $ymin2  $zmax)      //11
73      );

143     mergePatchPairs
144     (
145
146     );
```

Multiple blocks – Face matching

- To do a mesh with multiple blocks we proceed in the same way as we have done so far.
- When using **face matching,** we do not need to define all the vertices that made up each block.
- For the common face between blocks, we only need to define one set of vertices.
- In lines 19-27 we use macro syntax to declare the variables that we will use to define the vertices.
- In lines 29-31 we use macro syntax to define the number of cells in each direction. As this is a 2D case there is only one cell in the **Z**-direction.

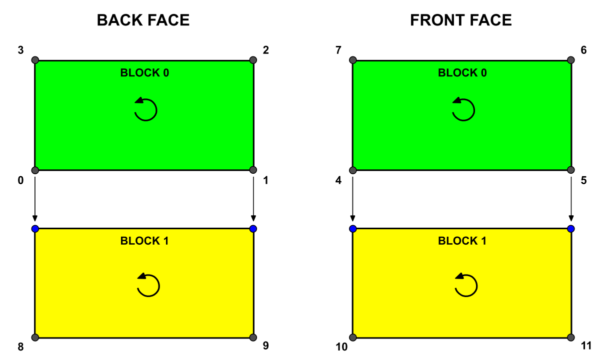


The *blockMeshDict* dictionary.

```
19    xmin 0;
20    xmax 1;
21    ymin 0.5;
22    ymax 1;
23    zmin 0;
24    zmax 1;
25
26    ymin2 0;
27    ymax2 0.5;
28
29    xcells 20;
30    ycells 10;
31    zcells 1;
39
45    vertices
46    (
47    //BLOCK 0
48        ($xmin  $ymin  $zmin)      //0
49        ($xmax  $ymin  $zmin)      //1
50        ($xmaax  $ymax  $zmin)     //2
51        ($xmin  $ymax  $zmin)      //3
52        ($xmin  $ymin  $zmax)      //4
53        ($xmax  $ymin  $zmax)      //5
54        ($xmax  $ymax  $zmax)      //6
55        ($xmin  $ymax  $zmax)      //7
56
57    //BLOCK 1
58        ($xmin  $ymin2  $zmin)     //8
59        ($xmax  $ymin2  $zmin)     //9
60        ($xmin  $ymin2  $zmax)     //10
61        ($xmax  $ymin2  $zmax)     //11
73    );

143   mergePatchPairs
144   (
145
146   );
```

Multiple blocks – Face matching

- In lines 45-73 we use macro syntax to define the vertices that made up each block.
- In lines 48-55 we define the vertices that made up the top block.
- In lines 58-61 we define the vertices that made up the bottom block. Notice that we are only defining the new vertices (**8 9 10 11**).
- The vertices (**0 1 4 5**), that are common between the block are not redefined.

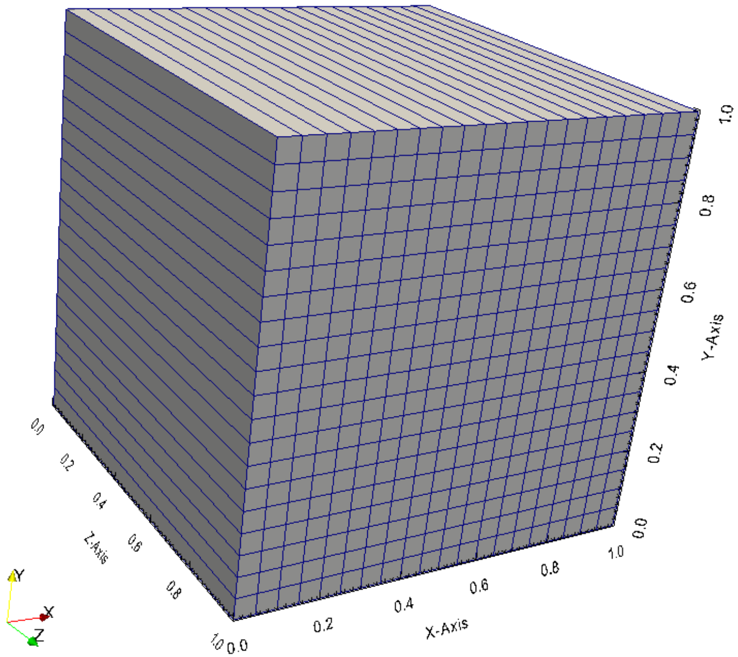


Meshing with blockMesh

The *blockMeshDict* dictionary.

Multiple blocks – Face matching

- Have in mind that the blocks need to be identical, that is, same number of cells and same grading.
- If the blocks are different, `blockMesh` will not generate the mesh.
- You do not need to define the common patches in the section **boundary** of the *blockMeshDict* dictionary.
- Finally, we do not need to define the patches in the keyword **mergePatchPairs** as *blockMesh* will automatically merge the common faces.



Meshing with blockMesh

Running the case

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

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

Meshing with blockMesh

- Let us take a close look to a `blockMeshDict` dictionary to study how to create non-straight edges.
- We will use the square cavity case.
- You will find this case in the directory:

`$BM/101BLOCKMESH/C5`

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

Meshing with blockMesh

What are we going to do?

- We will use this case to take a close look at a *`blockMeshDict`* dictionary.
- We will study how to create non straight edges.
- Possible options are; arc, spline, polyline, Bspline, line.
- Edges are constructed from the vertices definition.
- Each edge joining two vertices is assumed to be straight by default.
- The user can specify any edge to be curved by entries in the section **edges**.
- For example, to define an arc we first define the vertices to be connected to form an edge and then we give an interpolation point.
- To define a polyline, we first define the vertices to be connected to form an edge and then we give a list of the coordinates of the interpolation points.
- Let us study how to create curved edges using the square cavity case with **face merging**.

Meshing with blockMesh

The *blockMeshDict* dictionary.

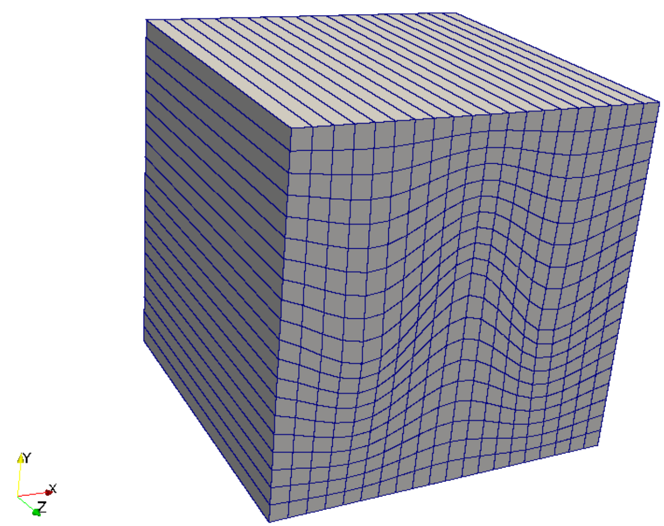
Interpolation method

Vertices to connect

Interpolation points

```
75      edges
76      (
112         spline 0 1
113         (
114             (0.25 0.4 0)
115             (0.5 0.6 0)
116             (0.75 0.4 0)
117         )
118         spline 4 5
119         (
120             (0.25 0.4 1)
121             (0.5 0.6 1)
122             (0.75 0.4 1)
123         )
124
125         spline 11 10
126         (
127             (0.25 0.4 0)
128             (0.5 0.6 0)
129             (0.75 0.4 0)
130         )
131         spline 15 14
132         (
133             (0.25 0.4 1)
134             (0.5 0.6 1)
135             (0.75 0.4 1)
136         )
138     );
```

- In lines 75-138 we define spline edges.
- As we are using **face merging**, we need to define the splines in each common patch.
- To define a spline, we first define the vertices to be connected to form an edge and then we give a list of the coordinates of the interpolation points.
- In lines 112-123 we define the splines belonging to block 0.
- In lines 125-136 we define the splines belonging to block 1.



Meshing with blockMesh

The `blockMeshDict` dictionary.

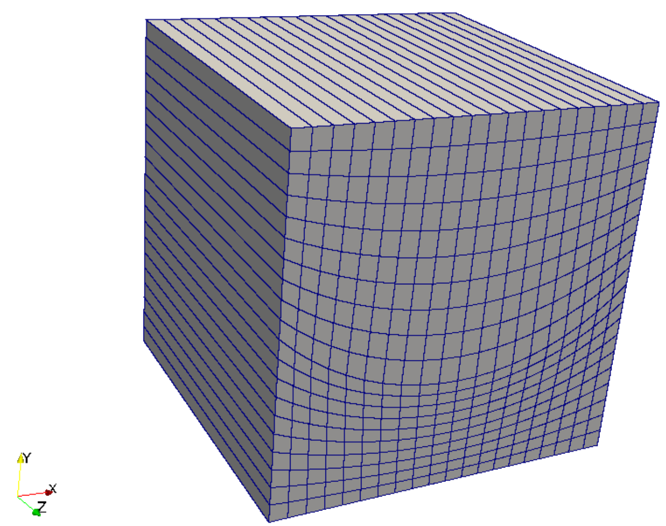
Interpolation method

Interpolation points

```
75    edges
76    (

78        arc 0 1    (0.5 0.3 0)
79        arc 4 5    (0.5 0.3 1)
80        arc 11 10 (0.5 0.3 0)
81        arc 15 14 (0.5 0.3 1)

138   );
```

Vertices to connect

- In lines 78-81 we define arc edges.
- As we are using **face merging**, we need to define the arcs in each common patch.
- To define an arc, we first define the vertices to be connected to form an edge and then we give an interpolation point.
- In lines 78-79 we define the arcs belonging to block 0.
- In lines 80-781we define the arcs belonging to block 1.



Meshing with blockMesh

Running the case

- Choose any of the previous cases.
- To generate the mesh, in the terminal window type:

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

Meshing with blockMesh

- Let us take a close look to a `blockMeshDict` dictionary to study how to create an O-grid mesh.
- We will use the square cavity case.
- You will find this case in the directory:

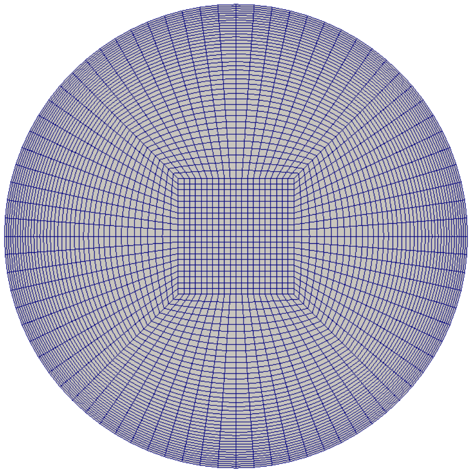
$BM/101BLOCKMESH/C6

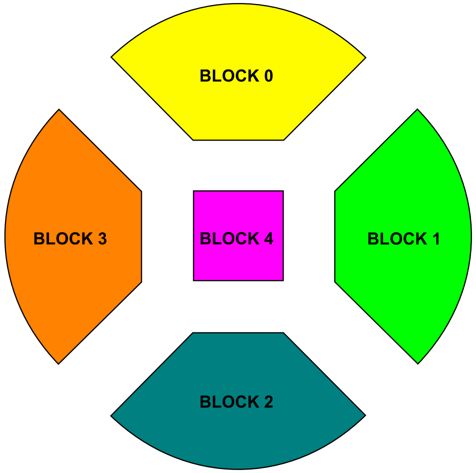
- In the case directory, you will find a few scripts with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on.
- These scripts can be used to run the case automatically by typing in the terminal, for example,
  - `$> sh run_solver`
- These scripts are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

Meshing with blockMesh

What are we going to do?

- We will use this case to take a close look at a `blockMeshDict` dictionary.
- We will create a 3D pipe using an O-grid topology.
- To create the O-grid topology we will use five blocks.
- At a first glance, this seems to be an easy task, but it requires some work to layout the topology.
- We will use **face matching**.



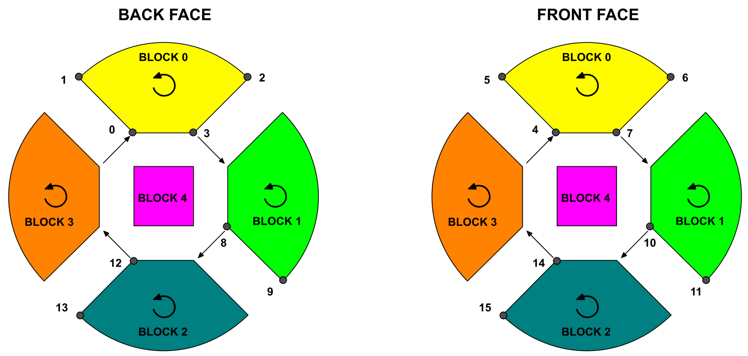


Meshing with blockMesh

The *blockMeshDict* dictionary.

```
17      convertToMeters 0.025;
18
19      vertices
20      (
21          //block0 vertices
22          (-0.25 0.25 0)                //0
23          (-0.707106 0.707106 0)        //1
24          (0.707106 0.707106 0)         //2
25          (0.25 0.25 0)                 //3
26          (-0.25 0.25 100)              //4
27          (-0.707106 0.707106 100)      //5
28          (0.707106 0.707106 100)       //6
29          (0.25 0.25 100)               //7
30
31          //block1 new vertices
32          (0.25 -0.25 0)                //8
33          (0.707106 -0.707106 0)        //9
34          (0.25 -0.25 100)              //10
35          (0.707106 -0.707106 100)      //11
36
37          //block3 new vertices
38          (-0.25 -0.25 0)               //12
39          (-0.707106 -0.707106 0)       //13
40          (-0.25 -0.25 100)             //14
41          (-0.707106 -0.707106 100)     //15
42      );
43
44
45      xcells 20;
46      ycells 40;
47      zcells 60;
48
49      xcells1 20;
50      ycells1 20;
51      zcells1 60;
52
53      stretch 0.25;
```

- In this case we use and scaling factor of 0.025 (line 17).
- We can also scale the mesh using the mesh utility `transformPoints`.
- In lines 19-42 we define the coordinates of all the vertices. Remember, we are using **face matching**.
- In lines 45-53 we use macro syntax to declare a set of variables that will be used later.

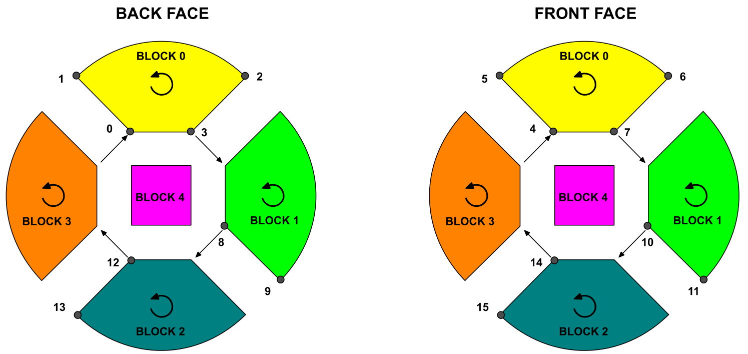


Meshing with blockMesh

The *blockMeshDict* dictionary.

```
54
55     blocks
56     (
57         //block0
58         hex (0 3 2 1 4 7 6 5)              ($xcells $ycells $zcells)      simpleGrading (1 $stretch 1)
59         //block1
60         hex (3 8 9 2 7 10 11 6)            ($xcells $ycells $zcells)      simpleGrading (1 $stretch 1)
61         //block2
62         hex (8 12 13 9 10 14 15 11)        ($xcells $ycells $zcells)      simpleGrading (1 $stretch 1)
63         //block3
64         hex (12 0 1 13 14 4 5 15)          ($xcells $ycells $zcells)      simpleGrading (1 $stretch 1)
65         //block4
66         hex (0 12 8 3 4 14 10 7)           ($xcells1 $ycells1 $zcells1)   simpleGrading (1 1 1)
67     );
```

- In lines 55-67, we define all the blocks that made up the O-grid topology.
- Notice that we are creating five blocks.
- We also define the number of cells of each block and the grading.
- As we are using **face matching**, the grading and number of cells in the common faces need to be the same.

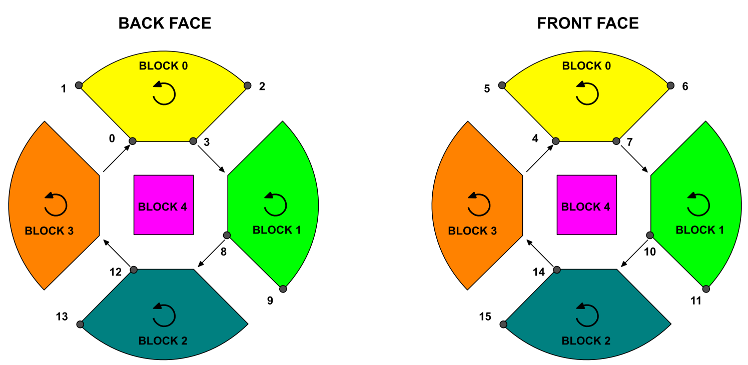


Meshing with blockMesh

The *blockMeshDict* dictionary.

```
69      edges
70      (
71          //block0 arc
72          arc 1 2   (0 1 0)
73          arc 5 6   (0 1 100)
74
75          //block1 arc
76          arc 2 9   (1 0 0)
77          arc 6 11  (1 0 100)
78
79          //block2 arc
80          arc 9 13   (0 -1 0)
81          arc 11 15  (0 -1 100)
82
83          //block3 arc
84          arc 1 13   (-1 0 0)
85          arc 5 15   (-1 0 100)
86      );
87
88      boundary
89      (
90
91          inlet
92          {
93              type patch;
94              faces
95              (
96                  (0 1 2 3)
97                  (2 3 8 9)
98                  (8 9 13 12)
99                  (13 12 0 1)
100                 (0 3 8 12)
101             );
102         }
103
```

- In lines 69-86 we define arc edges.
- In lines 88-129 we define the boundary patches.
- In lines 91-102 we define the patch **inlet**. Notice that this boundary patch has five faces.

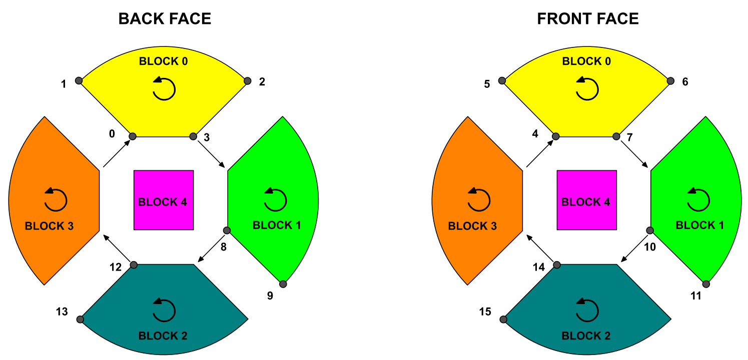


Meshing with blockMesh

The *blockMeshDict* dictionary.

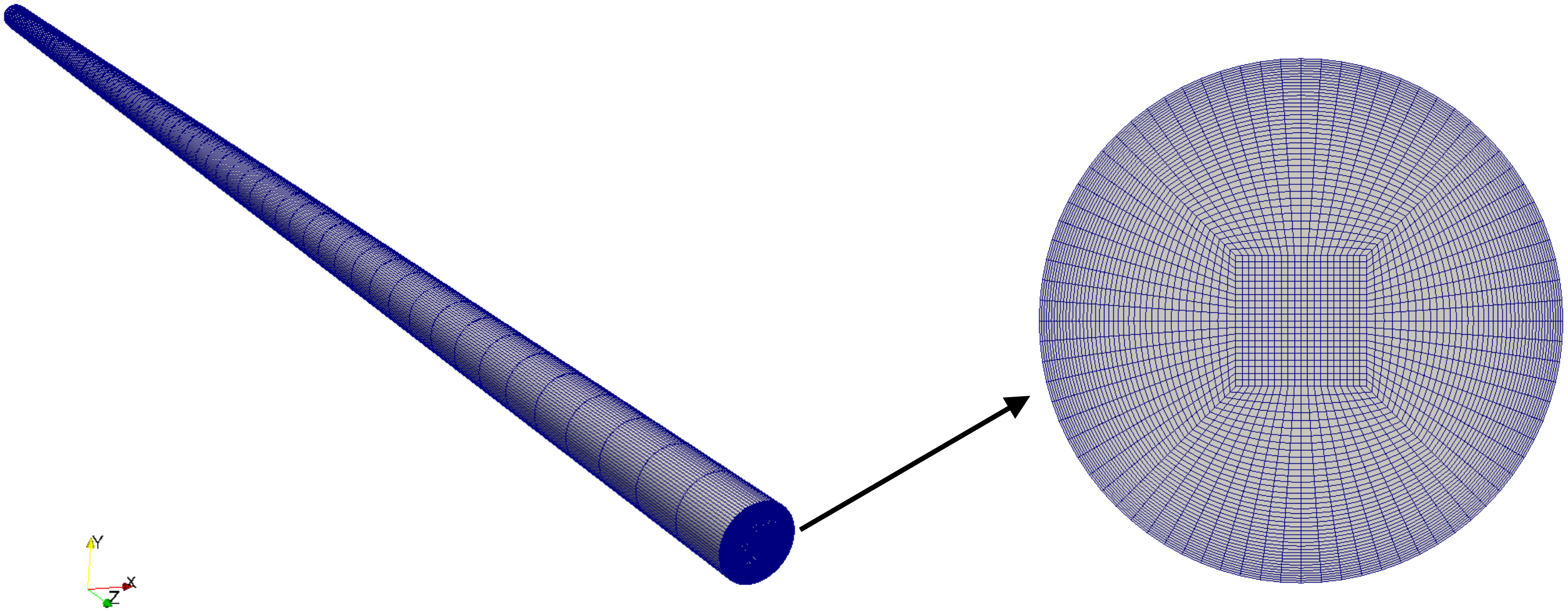
```
104        outlet
105        {
106            type patch;
107            faces
108            (
109                (4 5 6 7)
110                (6 7 10 11)
111                (15 11 10 14)
112                (15 14 4 5)
113                (4 7 10 14)
114            );
115        }
116
117        pipe
118        {
119            type wall;
120            faces
121            (
122                (1 5 6 2)
123                (2 6 11 9)
124                (9 11 15 13)
125                (15 13 5 1)
126            );
127        }
128
129    );
130
131    mergePatchPairs
132    (
133    );
```

- In lines 88-129 we define the boundary patches.
- In lines 104-115 we define the patch **outlet**. Notice that this boundary patch has five faces.
- In lines 117-127 we define the patch **pipe**. Notice that this boundary patch has four faces.
- In this case we do not use face merging (lines 131-133).



Meshing with blockMesh

The *`blockMeshDict`* **dictionary.**



3D pipe

O-grid topology (outlet patch)

Meshing with blockMesh

Running the case

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

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