pyHyp

MDO Lab

Feb 20, 2023

CONTENTS

pyHyp is a hyperbolic mesh generator that automatically generates two or three dimensional meshes around simple geometric configurations. The basic idea is to start with an initial surface (or curve) corresponding to the geometry of interest and then *grow* or *extrude* the mesh in successive layers until it reaches a sufficient distance from the original surface. In the process, the entire space surrounding the geometry is meshed.

An overview of the hyperbolic mesh marching method implemented in pyHyp can be found in Section II.A of Secco et al. Most of the theory for pyHyp was taken from Chan and Steger.

Contents:

CHAPTER

ONE

INSTALLATION

1.1 Prerequisites

pyHyp depends heavily on other packages to do much of the underlying "heavy lifting". The following external components are required for pyHyp:

- CGNS Libarary
- PETSc

See the MDO Lab installation guide here for the supported versions and installation instructions.

Note: A working MPI is not strictly required. However, in most cases PETSc should be configured with MPI.

1.2 Compilation

pyHyp follows the standard MDO Lab build procedure. To start, first clone the repo. For stability we recommend checking out a tagged release.

Next, find a configuration file close to your current setup in `config/defaults` and copy it to `config/config.mk`. For example:

```
$ cp config/defaults/config.LINUX_GFORTRAN_OPENMPI.mk config/config.mk
```

If you are a beginner user installing the packages on a Linux desktop, you should use the `config.LINUX_GFORTRAN_OPENMPI.mk` versions of the configuration files. The `config.LINUX_INTEL.mk` versions are usually used on clusters.

Once you have copied the config file, compile pyHyp by running:

```
$ make
```

If everything was successful, the following lines will be printed to the screen (near the end):

```
Testing if module hyp can be imported...
Module hyp was successfully imported.
```

Finally, install the Python interface with:

```
$ pip install .
```

1.3 Testing Your Installation

To test your installation, you can run the regression tests. Running the tests requires additional dependencies. Check if you have these installed by running:

```
$ pip install .[testing]
```

Once you have the necessary dependencies, download the reference mesh files:

```
$ ./reg_tests/ref/get-ref-files.sh
```

Then, in the root directory, run:

```
$ testflo -v
```

CHAPTER

TWO

TUTORIAL

A complete sample script to generate a grid is given below. This particular example is available under *examples/BWB/runBWB.py*.

```
# rst import (start)
import os
from pyhyp import pyHyp

# rst import (end)

baseDir = os.path.dirname(os.path.abspath(__file__))
surfaceFile = os.path.join(baseDir, "bwb.fmt")
volumeFile = os.path.join(baseDir, "bwb.cgns")

options = {
    # ---------------------------
    #        Input Parameters
    # ---------------------------
    "inputFile": surfaceFile,
    "unattachedEdgesAreSymmetry": True,
    "outerFaceBC": "farfield",
    "autoConnect": True,
    "BC": {},
    "families": "wall",
    # ---------------------------
    #        Grid Parameters
    # ---------------------------
    "N": 81,
    "s0": 4e-6,
    "marchDist": 1100.0,
    # ---------------------------
    #   Pseudo Grid Parameters
    # ---------------------------
    "ps0": -1.0,
    "pGridRatio": -1.0,
    "cMax": 2.5,
    # ---------------------------
    #   Smoothing parameters
    # ---------------------------
    "epsE": 1.0,
    "epsI": 2.0,
```

(continues on next page)

(continued from previous page)

```
    "theta": 3.0,
    "volCoef": 0.25,
    "volBlend": 0.0002,
    "volSmoothIter": 150,
}

# rst object
hyp = pyHyp(options=options)
# rst run
hyp.run()
hyp.writeCGNS(volumeFile)
```

Each section of the example is now described.

```
import os
from pyhyp import pyHyp
```

This is the preferred way of importing the *pyHyp* module into python.

The options dictionary is used to provide all run-time options to pyHyp to control the generation of the grid. A description of each option and its general effect on the grid generation process is explained in *Options*.

The next line of code

```
hyp = pyHyp(options=options)
```

generates the pyHyp object.

Note: When exporting a surface mesh from ICEMCFD in plot3d format use the following options:

- Formatted
- Whole
- Double
- No IBLANK Array

Warning: It is essential that the normals of each of the surface patches point in the *OUTWARD* direction, i.e. the marching direction.

The next two lines perform the actual generation and write the resulting grid to a cgns file:

```
hyp.run()
hyp.writeCGNS(volumeFile)
```

The output of the run should look similar to the following:

```
#--------------------#
 Total Nodes:   14105
 Unique Nodes:  13457
 Total Faces:   13376
```

(continues on next page)

(continued from previous page)

```
#--------------------#
 Normal orientation check ...
 Normals are consistent!
 Determining topology ...
 Topology complete.
#--------------------#
Grid Ratio:  1.2532
#--------------------#
#-------------------------------------------------------------------------------------------
↪--------------------------#
# Grid | CPU  | Sub | KSP | nAvg |  Sl  | Sensor | Sensor | Min     | Min     |  deltaS ␣
↪| March    | cMax  | Ratio |
# Lvl  | Time | Its | Its |      |      | Max    | Min    | Quality | Volume  |         ␣
↪| Distance |       | kMax  |
#-------------------------------------------------------------------------------------------
↪--------------------------#
      2    0.1     2    11      0  0.055  1.00006  0.98387  0.38387  0.359E-10  0.314E-
↪05  0.451E-05  0.0011  0.0000
      3    0.2     2    11      0  0.064  1.00010  0.96703  0.35947  0.657E-10  0.493E-
↪05  0.116E-04  0.0018  1.9184
      4    0.3     1    11      0  0.067  1.00012  0.96111  0.35650  0.103E-09  0.618E-
↪05  0.165E-04  0.0022  1.5882
      5    0.4     2    11      0  0.074  1.00024  0.90864  0.35575  0.174E-09  0.787E-
↪05  0.305E-04  0.0035  1.7282
      6    0.4     1    11      0  0.076  1.00031  0.89199  0.35610  0.237E-09  0.987E-
↪05  0.383E-04  0.0035  1.3561
      7    0.5     1    11      0  0.079  1.00037  0.87508  0.35613  0.371E-09  0.124E-
↪04  0.482E-04  0.0044  1.5634
      8    0.5     2    11      0  0.084  1.00083  0.78268  0.35729  0.548E-09  0.155E-
↪04  0.761E-04  0.0069  1.4708
      9    0.6     1    11      0  0.087  1.00108  0.75129  0.36201  0.717E-09  0.194E-
↪04  0.916E-04  0.0069  1.3003
     10    0.6     1    11      0  0.089  1.00124  0.74002  0.36599  0.104E-08  0.243E-
↪04  0.111E-03  0.0086  1.4222
     < iterations skipped for brevity >
     78   33.9    22    29      0  0.904  0.98475  0.97354  0.32556  0.347E+04  0.
↪730E+01  0.562E+03  2.5000  1.2603
     79   36.1    22    29      0  0.936  0.98435  0.97427  0.32565  0.688E+04  0.
↪930E+01  0.708E+03  2.5000  1.2570
     80   38.0    21    29      0  0.968  0.98388  0.97469  0.32573  0.138E+05  0.
↪119E+02  0.885E+03  2.5000  1.2551
     81   39.9    20    29      0  1.000  0.98341  0.97476  0.32579  0.276E+05  0.
↪152E+02  0.110E+04  2.5000  1.2540
```

Several important parameters are displayed to inform the user of the solution progress. The most of important of which is the *Min Quality* column. This column displays the minimum quality of all the cells in the most recently computed layer of cells. For a valid mesh, these must be all greater than zero.

CHAPTER

THREE

AUTOMATIC SYMMETRY BCS

Using the *unattachedEdgesAreSymmetry* option will automatically apply symmetry boundary conditions to any edges that do not interface with another block. This page describes how this option works and a rare case where this option can fail.

Automatically applying symmetry boundary conditions involves determining the correct symmetry direction for each unattached edge. For a standard case, the magnitudes of the coordinates for all nodes of an unattached edge are summed, and the coordinate direction with the minimum sum is set as the symmetry direction. This works because we expect the geometry to lie flat on the symmetry plane. However, when an unattached edge is coincident with one of the coordinate axes, this method cannot be used because there will be two directions with a zero sum.

In this case, a secondary check is performed. The vectors between nodes on the boundary edge and their adjacent nodes on the first interior edge are computed, and the magnitudes of the coordinates are summed over all vectors. The coordinate direction with the maximum sum is taken to be the symmetry direction. This assumes that the aforementioned vectors point primarily in the direction normal to the symmetry plane. This assumption does not always hold, such as for a highly swept wing. If the vectors deviate enough from the normal direction, the symmetry direction will be incorrectly assigned, and the mesh extrusion will fail.

Warning: If you encounter negative volumes near the symmetry plane, set the symmetry boundary conditions manually using *the BC option*.

CHAPTER

FOUR

SPECIFYING BCS USING ICEM

This section will show how we can use ICEM to specify boundary conditions at each open edge of a CGNS surface geometry. The boundary conditions currently supported are:

- Constant X, Y, or Z planes;
- Symmetry X, Y, or Z planes;
- Splay (free edge).

4.1 Flat square example

Note: If you have a surface geometry, this step is not required, and you can proceed to the next section, 'Create Parts with Edges.'

This subsection will show how to create a flat square surface mesh in ICEM. This geometry is the one used as an example of boundary conditions setup.

1. **Prepare your workspace and open ICEMCFD**

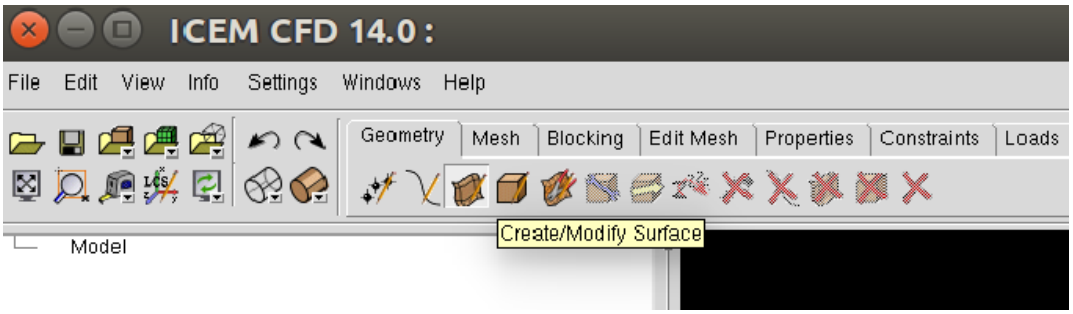
   Create an empty folder anywhere in your computer. Navigate to this folder using the terminal and type the following command:

   ```
   $ icemcfd
   ```

   This will open ICEM, and all the files will be stored in this folder.

2. **Create the surface geometry**

   Under the *Geometry* tab, select *Create/Modify Surface*, as shown below:



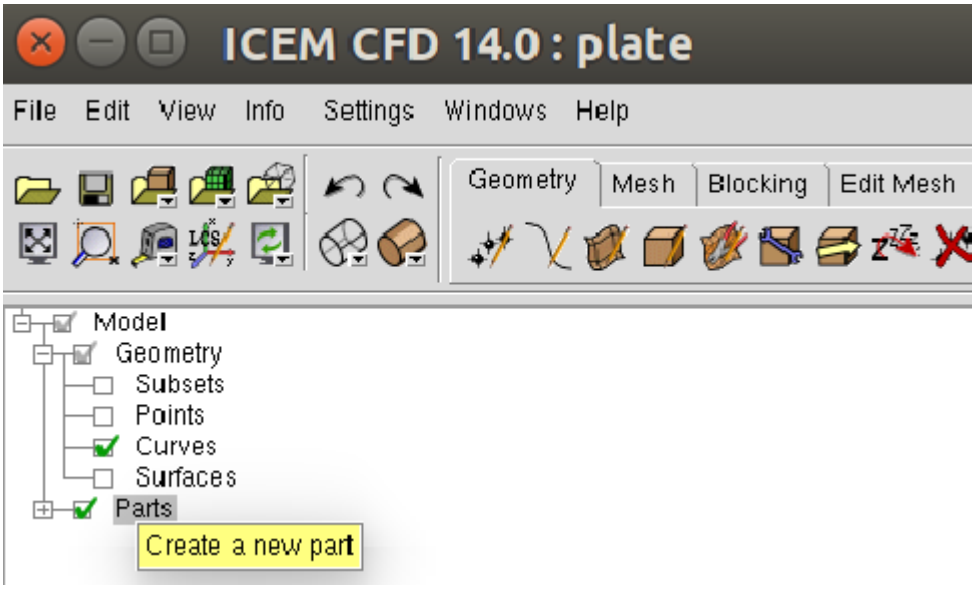
   A new menu will show up on the lower-left corner of the screen. Select *Standard Shapes*, then *Box*.

Finally, type '1 1 0' in the *X Y Z size* field and click on 'Apply', just as shown below:

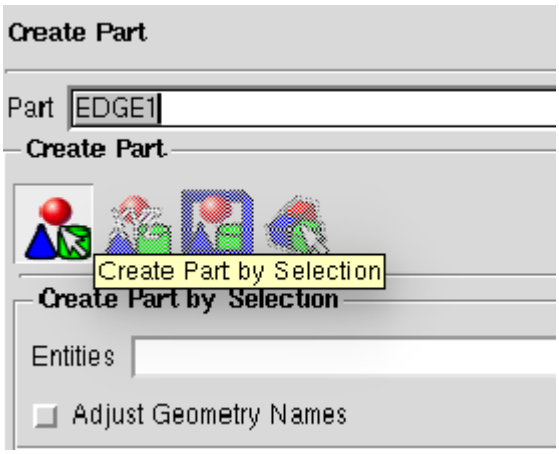
This function would usually generate the 6 surfaces of a box, but since we used Z size = 0, it will automatically give just a single surface in this case as the upper and lower sides of the box coincide.

3. **Create parts for the edges**

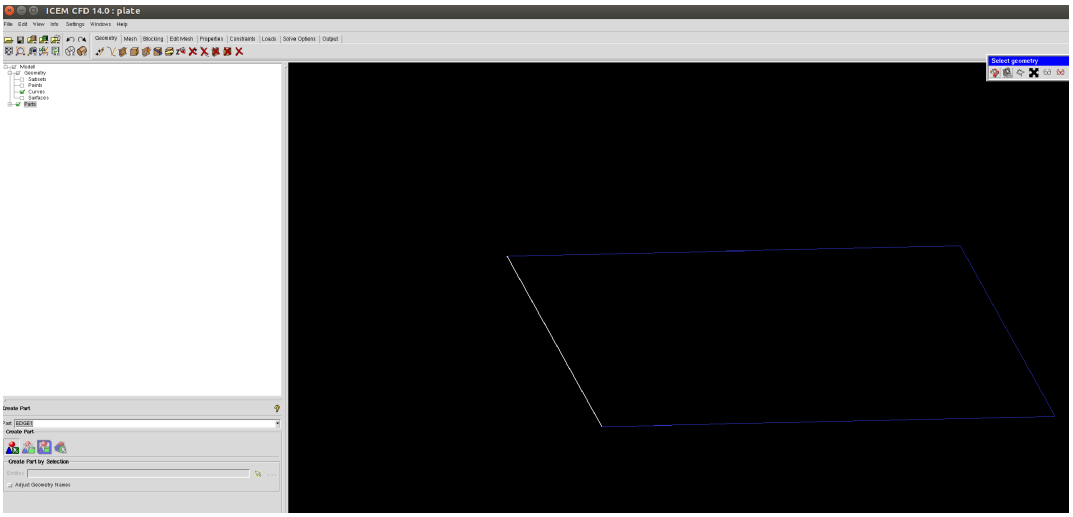
   ICEM groups geometry components into *Parts*. We need to create Parts for the edges so that we can easily set up the boundary conditions later on. Look at the model tree on the left side of the screen and click with the right-mouse-button on the *Parts* branch. Select the *Create Parts* option. Also make sure that the *Surfaces* box in the *Geometry* branch in unchecked (otherwise it will be hard to select just the edges):



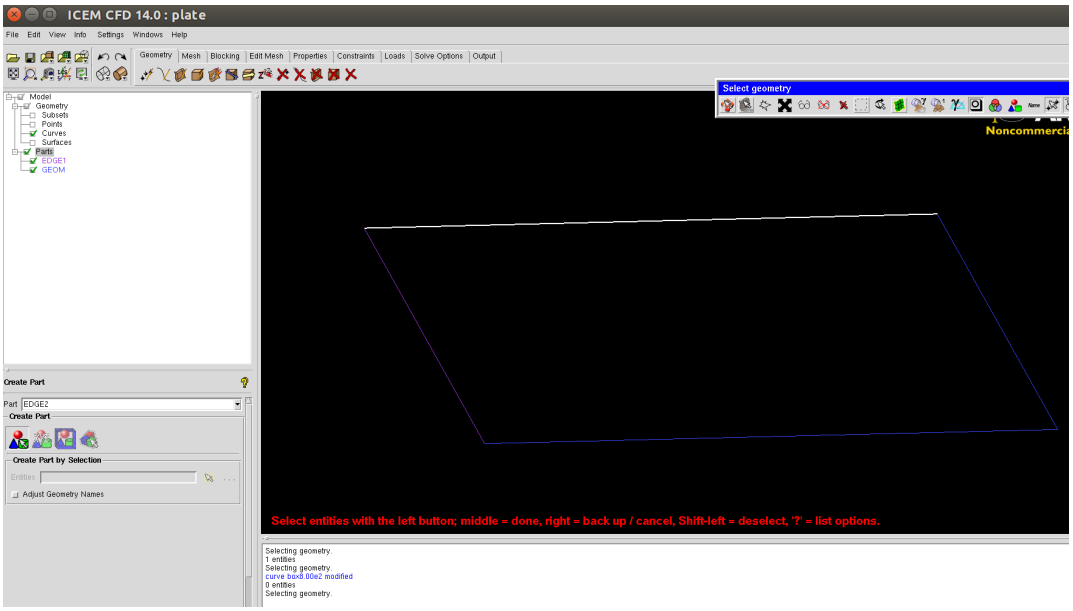
On the lower-left corner menu change the name of the part to 'EDGE1', then click on the *Create Part by Selection*, as shown below:



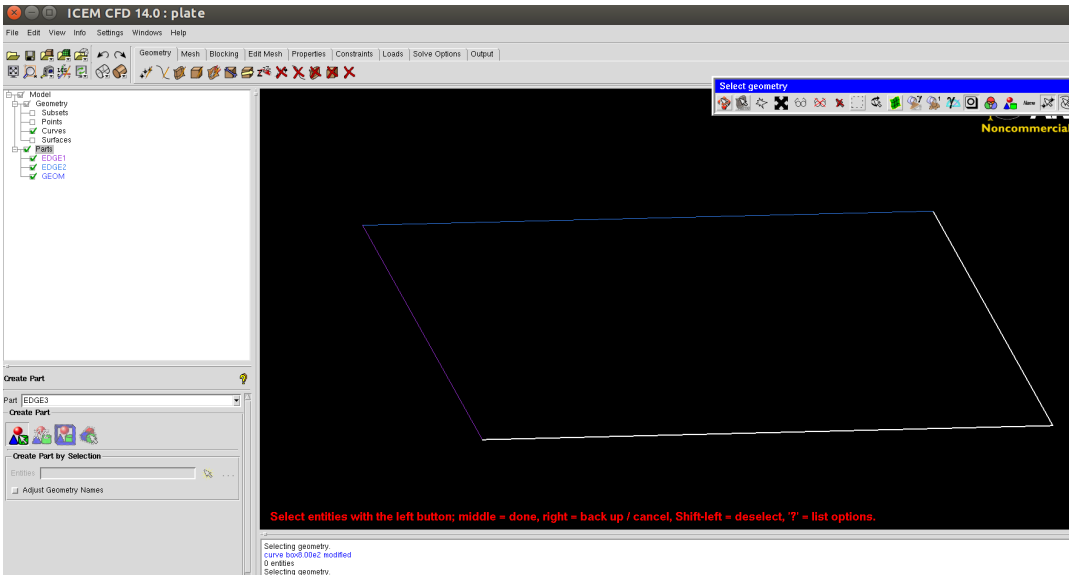
Click on one edge with the left-mouse-button in order to highlight it (see figure below) then click with the middle-mouse (scroll) button anywhere to confirm your selection. Now you can check if you have the *EDGE1* component under the *Parts* branch in the model tree.



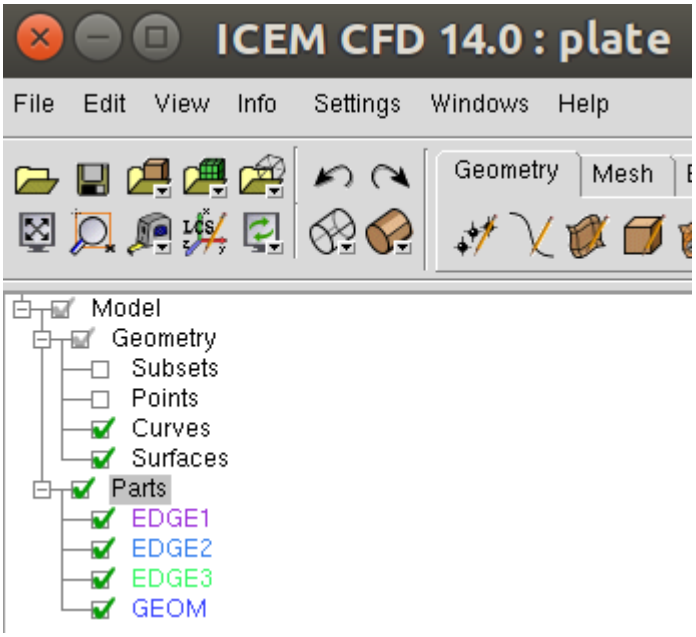
We can repeat the process to create the 'EDGE2' part. Remember to change the part name before selecting another edge. This time I chose the upper edge:



You can group multiple edges under the same Part if you want to apply the same boundary condition to all of them. For instance, we will create 'EDGE3' by grouping the remaining two edges. When selecting the edges, click on both of them with the left-mouse-button and only then you click with the middle-mouse-button to create the part.

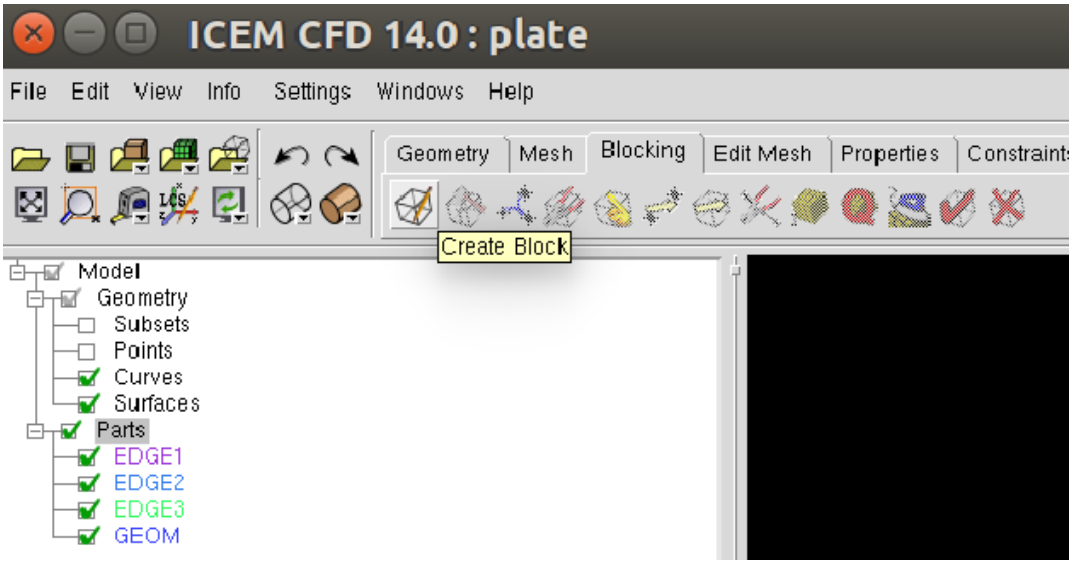


In the end, the *Parts* branch of the model tree should have three components: EDGE1, EDGE2, and EDGE3. Now you can also turn the *Surface* box back on. The model tree should look like this:

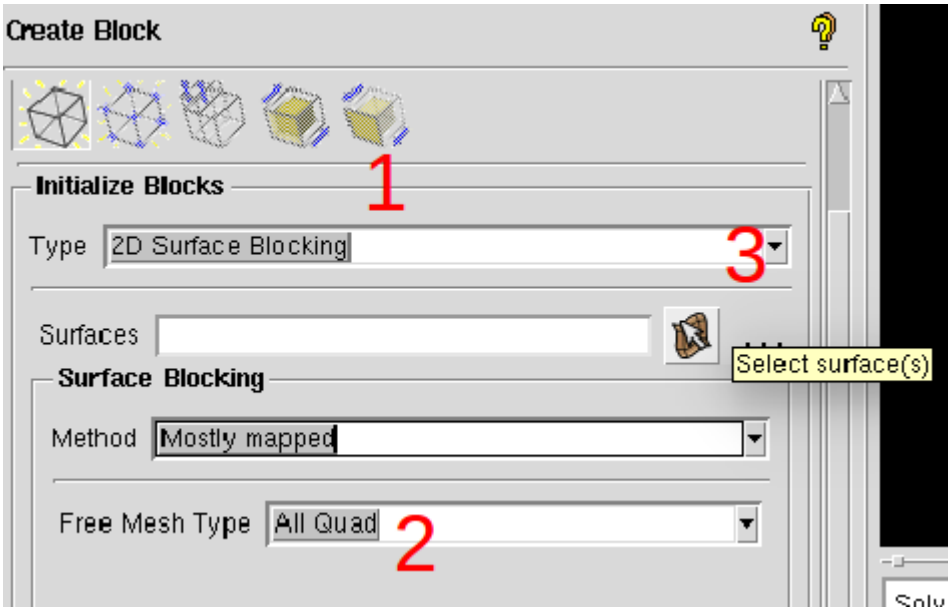


4. **Create blocking**

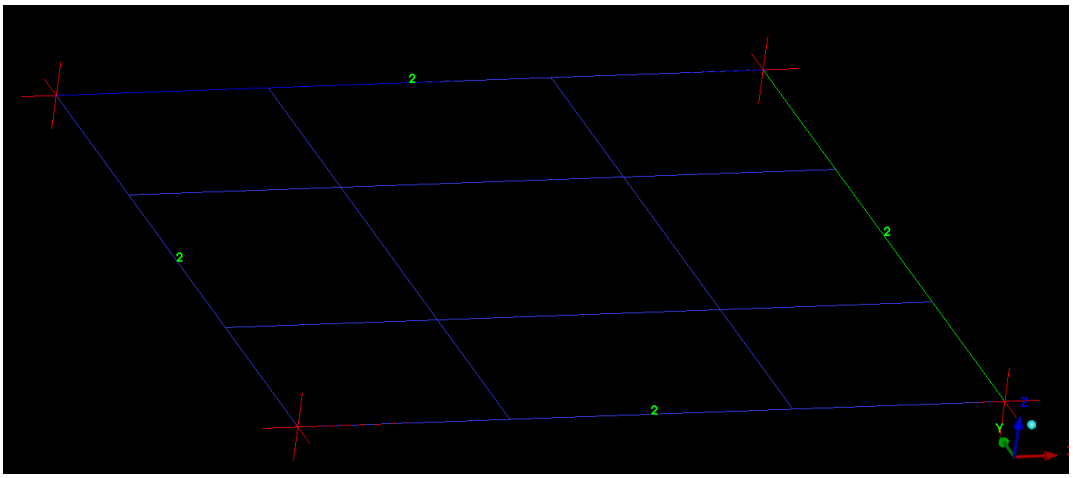
   Now we need to create the blocks used by ICEM to generate meshes. Under the *Blocking* tab, choose *Create Block*:



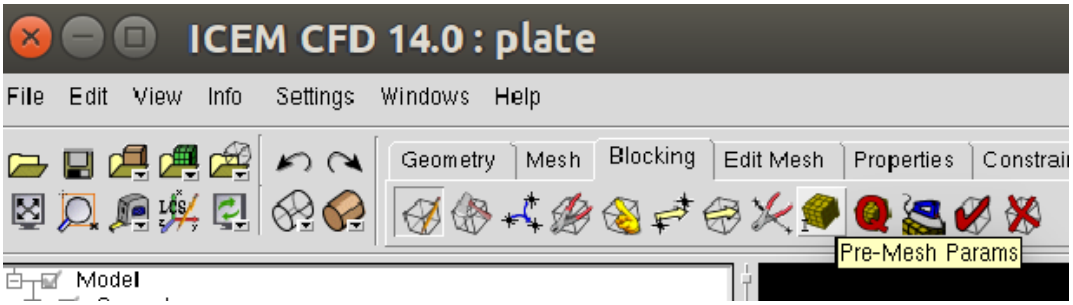
On the lower-left corner menu choose '2D Surface Blocking' as *Type*, and select the 'All Quad' option under *Free Mesh Type*. Later, click on the *Select surfaces(s)* button:



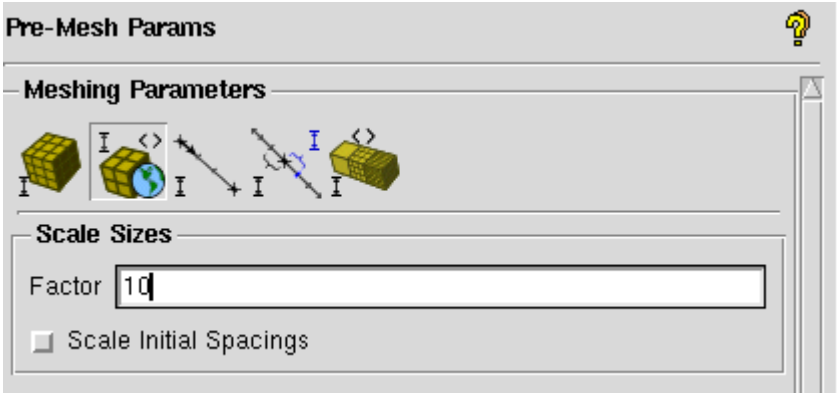
Now click on the flat square in order to highlight it (it should turn white), then click with the middle-mouse-button to confirm your selection. Finally, click on the *Apply* button to create the block (the square should turn back to blue). We can check if the blocking was done correctly by expanding the *Blocking* branch of the model tree. Click with the right-mouse-button on the *Edges* component and turn on the 'Counts' option. This will show how many nodes we have on each edge. For now, we should have two nodes per edge, as shown below:



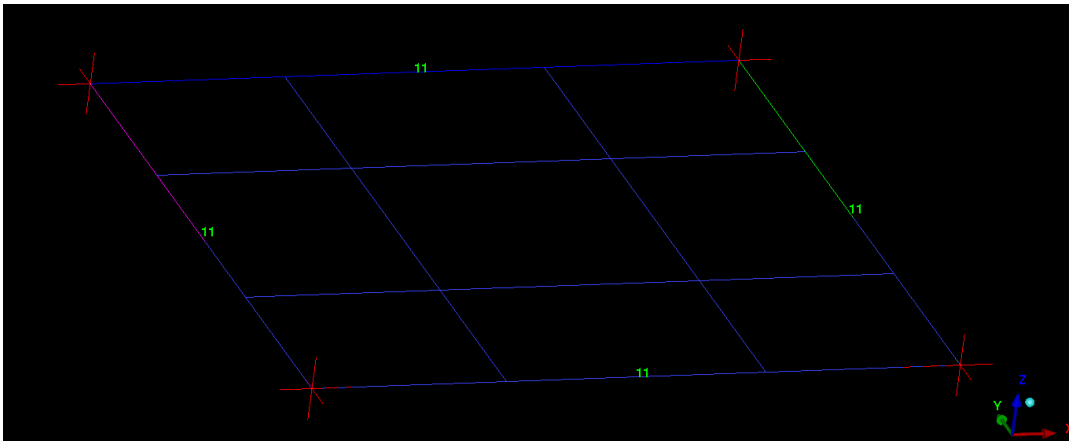
Let's increase the number of nodes to make things more interesting. Under the *Blocking* tab, choose *Pre-Mesh Params*:



Select the *Scale Sizes* feature on the lower-left menu. Adjust the *Factor* option to 10 and click on *Apply* to globally refine the mesh:

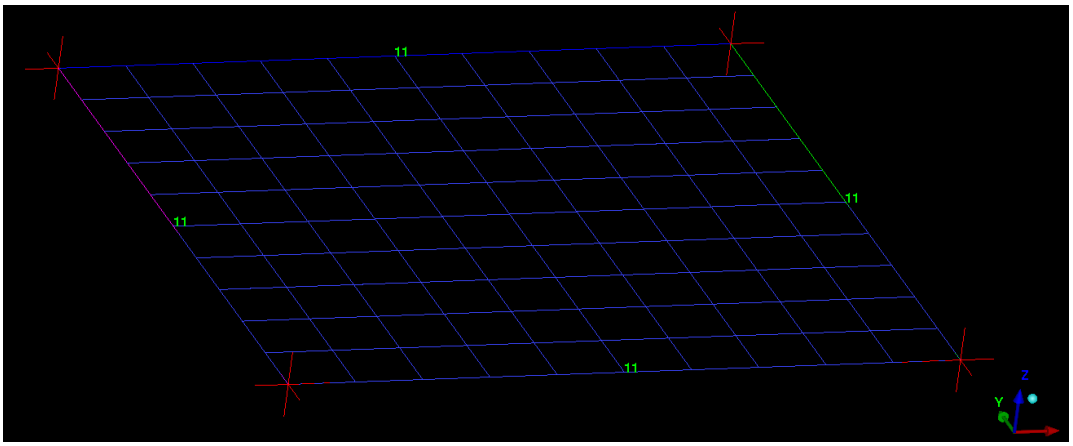


Now each edge should show 11 nodes:



5. **Generate the Pre-mesh**

   It is time to generate the Pre-mesh. Just check the *Pre-mesh* box under the *Blocking* branch of the model tree and choose *Yes*. You should see all the surface cells now:



   See if everything looks right in your Pre-mesh.

4.2 Preparing to export the mesh

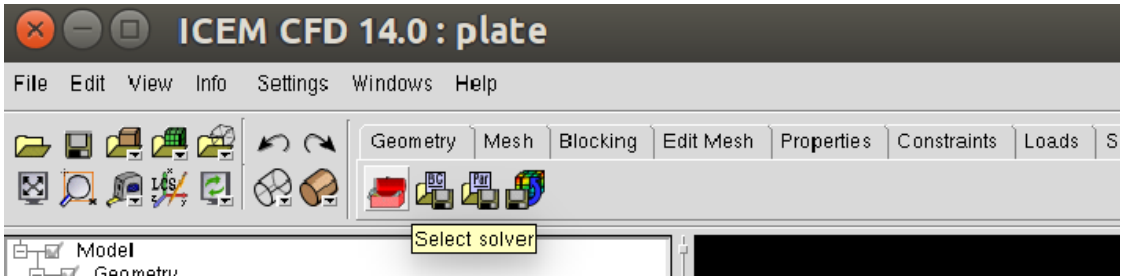
Just to recap, we have done the following procedures:

- Created our geometry in ICEM;
- Created Parts grouping edges that will share same boundary conditions;
- Added the surface blocks;
- Generated the Pre-mesh.

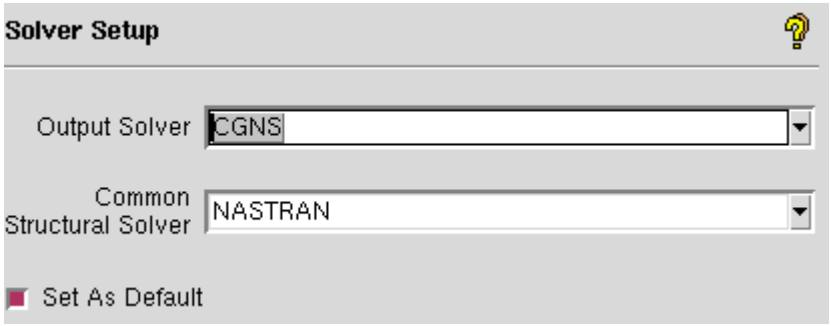
Note: The procedures described from now on apply to any geometry. However, make sure you have followed all these steps above if you are working with your own geometry.

Now that we confirmed that the Pre-mesh looks right, we can generate the structured mesh. Click with the right-mouse-button on the *Pre-mesh* component under the *Blocking* branch of the model tree and choose *Convert to MultiBlock Mesh*. Save the project in the folder you just created. A new branch named *Mesh* should show up in your model tree.

Next, we should select the export format. Under the *Output* folder, click on the *Select Solver* button (a red toolbox):



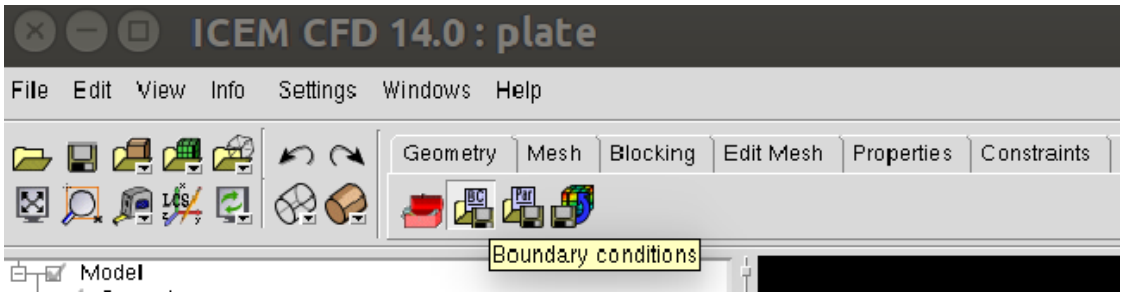
Choose the following options in the lower-left menu and click *Apply*:



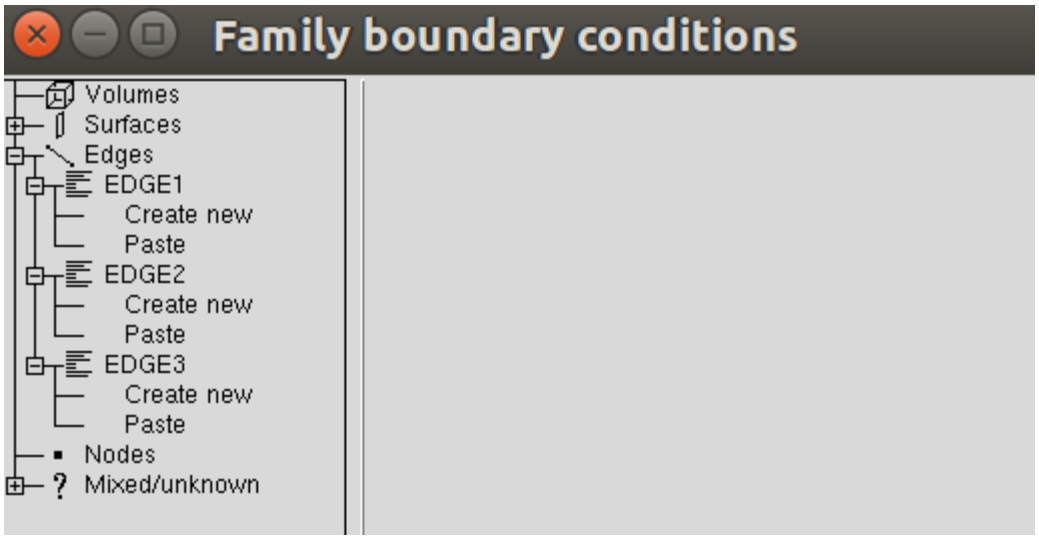
This will apply the CGNS format to our output. We will select the boundary conditions in the next step.

4.3 Applying boundary conditions

Under the *Output* folder, click on the *Boundary condition* button:

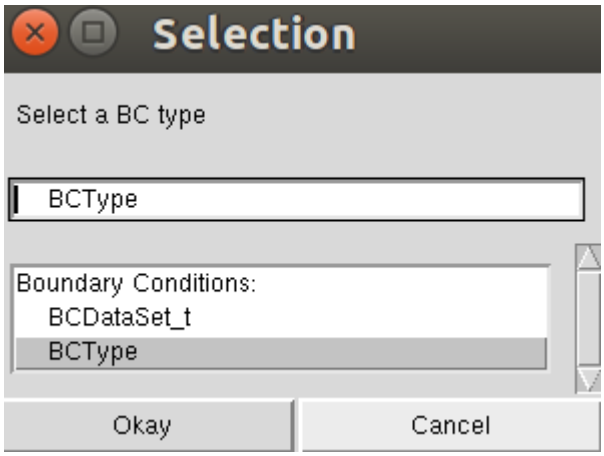


A new window should pop up. As we will apply boundary conditions (BCs) to the edges, expand everything under the *Edges* branch. You should see the edges Parts we defined previously (EDGE1, EDGE2, and EDGE3) as shown below. In some cases, they may end up under the *Mixed/Unknown* branch.



1. **Symmetry plane**

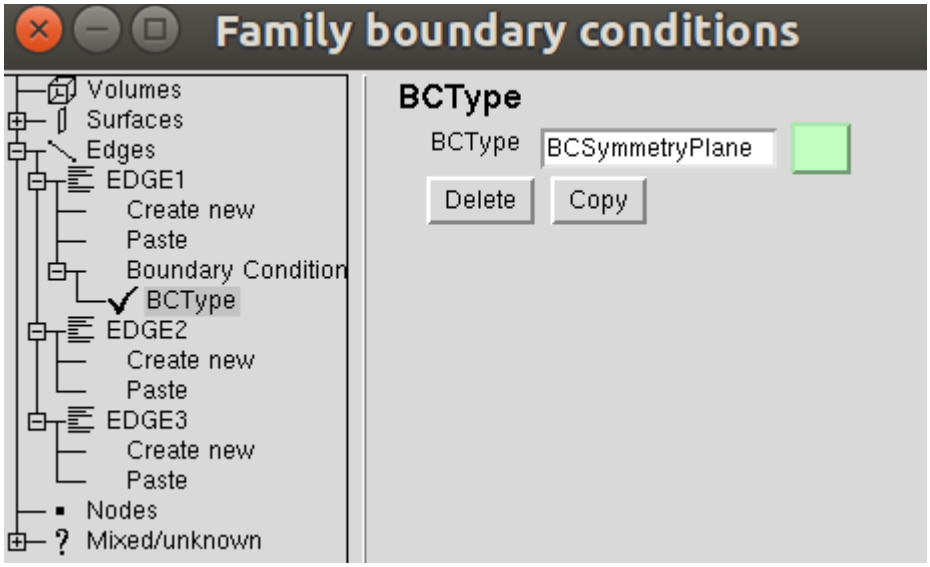
Let's add a symmetry plane boundary condition to EDGE1. Click on the *Create New* branch under *EDGE1*. Another window will show up, where you should choose 'BCType' and click on 'Okay'.



Now go back to the Boundary conditions window. If you click on the green button, you will see several types of Boundary Conditions. The currently supported types are:

- BCExtrapolate -> Splay;
- BCSymmetryPlane -> Symmetry X, Y, or Z planes;
- BCWall -> Constant X, Y, or Z planes.

For this example, let's choose 'BCSymmetryPlane' for this edge.

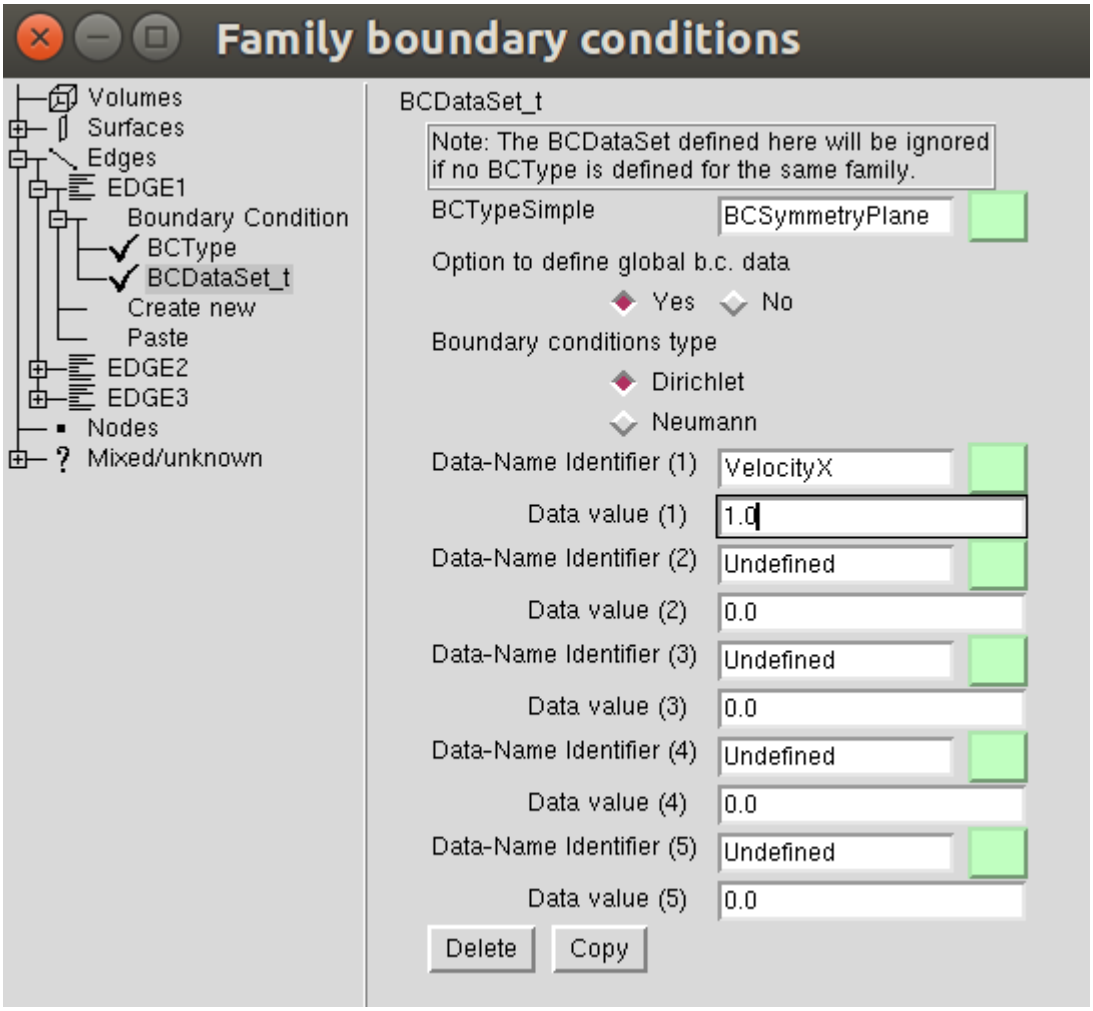


Now we need to specify if we want an X, Y, or Z symmetry plane. We will use a 'Velocity' node from the CGNS format in order to specify the reference plane. For instance, if we add a 'VelocityX' node to this boundary condition, we have a X symmetry plane, and the same for the other coordinates.

Do the following steps to add a X symmetry plane to the EDGE1 Part:

- Click again on the *Create New* branch under *EDGE1*;
- This time, select 'BCDataset_t' on the new window, and click on 'Okay';
- Now back to the BC window, click on the green button near 'BCTypeSimple' and select 'BCSymmetryPlane';
- Click on the green button that corresponds to the 'Data-Name Identifier (1)' field and choose 'VelocityX';
- Change the 'Data value (1)' field to 1.0.

In the end, the options should look like the figure below:



In the case of a Y symmetry plane, you should select 'VelocityY' instead, and similarly for a Z symmetry plane. Do not click on *Accept* yet, otherwise it will close the window. Now let's see the other edges.

2. **Constant Plane**

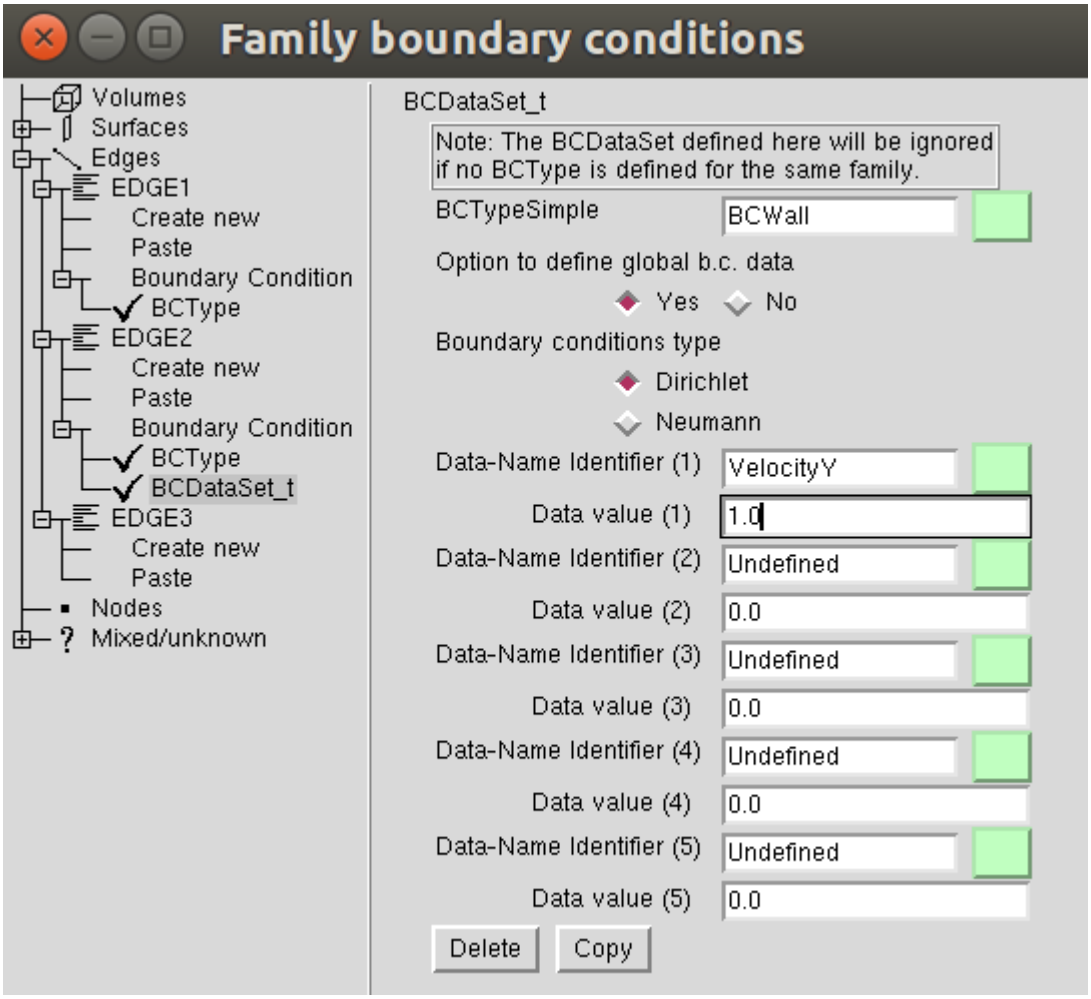
We will add a constant Y plane to EDGE2. Follow these steps:

- Click on the *Create New* branch under *EDGE2*;
- Select 'BCType' on the new window, and click on 'Okay';
- Click on the green button of the BC window and select 'BCWall'.

We still need to specify which coordinate (X, Y, or Z) should remain constant in this boundary condition. We will do this by adding a 'Velocity' option to this boundary condition.

- Click again on the *Create New* branch under *EDGE2*;
- This time, select 'BCDataset_t' on the new window, and click on 'Okay';
- Now back to the BC window, click on the green button near 'BCTypeSimple' and select 'BCWall';
- Click on the green button that corresponds to the 'Data-Name Identifier (1)' field and choose 'VelocityY';
- Change the 'Data value (1)' field to 1.0.

In the end, the options should look like the figure below:



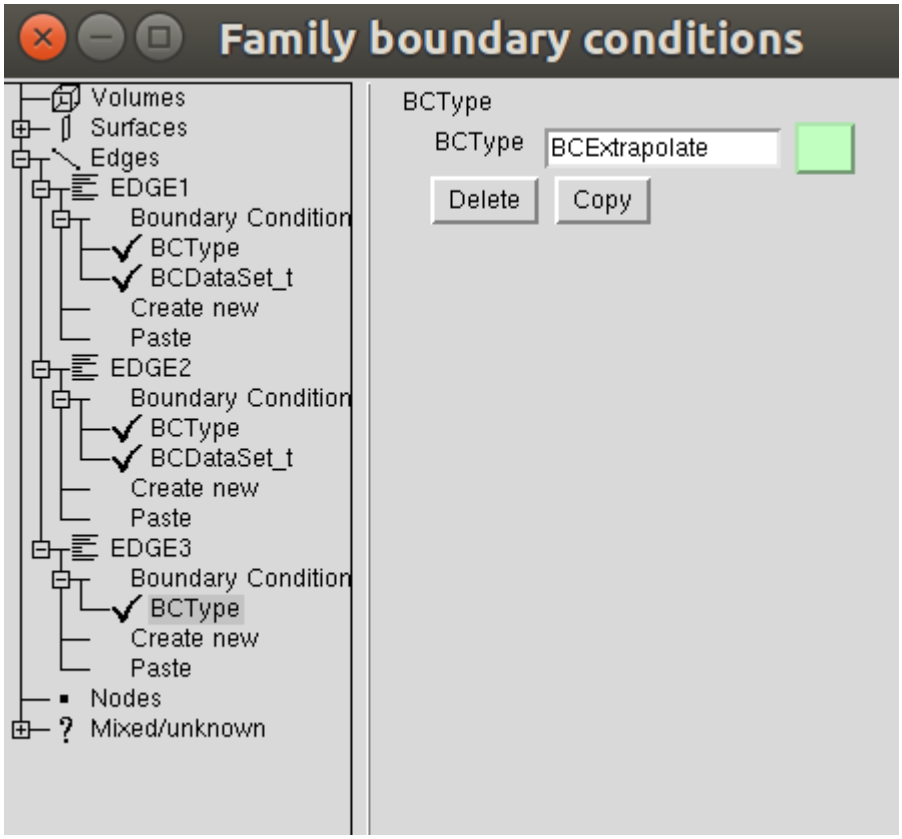
In the case of a constant X plane, you should select 'VelocityX' instead, and similarly for a constant Z plane. Do not click on *Accept* yet, because we still have one more boundary condition to go!

3. **Splay**

We'll finally add a Splay boundary condition for the two edges included in the EDGE3 Part.

- Click on the *Create New* branch under *EDGE3*;
- Select 'BCType' on the new window, and click on 'Okay';
- Click on the green button of the BC window and select 'BCExtrapolate'.

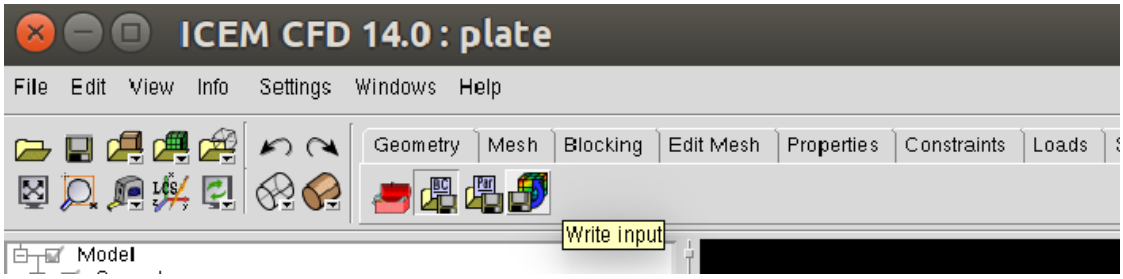
This one was easier! The same BC will be applied to all edges in this Part. In the end, your boundary conditions tree should look like this:



Now we can click on *Accept* as we finished adding all the boundary conditions.

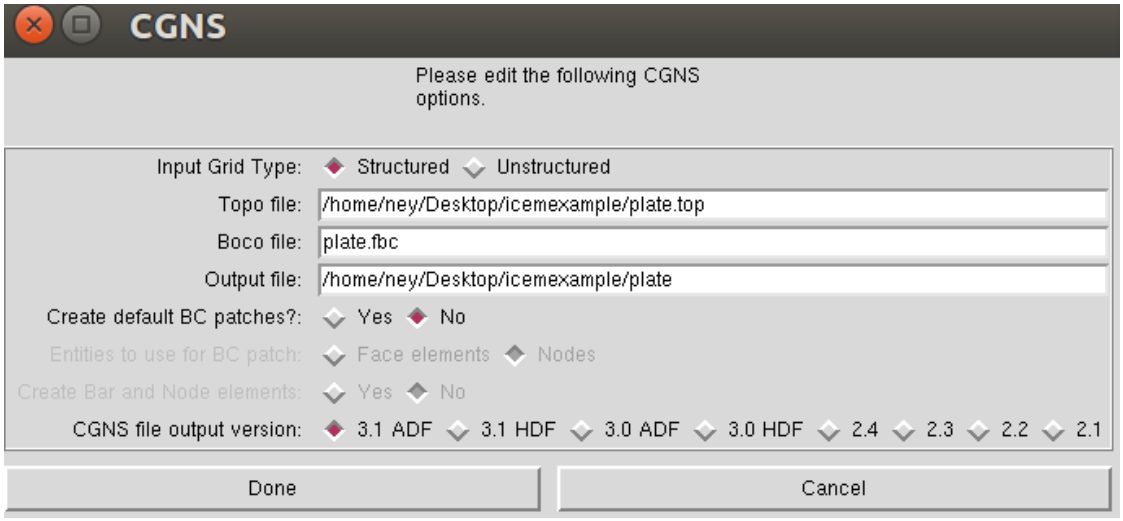
4.4 Exporting the mesh

We are ready to export the mesh! Click on the *Write input* button under the *Output* tab:



Save the project if asked for. Next we need to select the Multiblock mesh file. The name shown by default should be correct, so just click on 'Open'. In the next window, click on 'All'.

Another window with CGNS export options will show up. The default options should work fine, but you can compare it with the ones below. Make sure you are exporting a structured mesh:



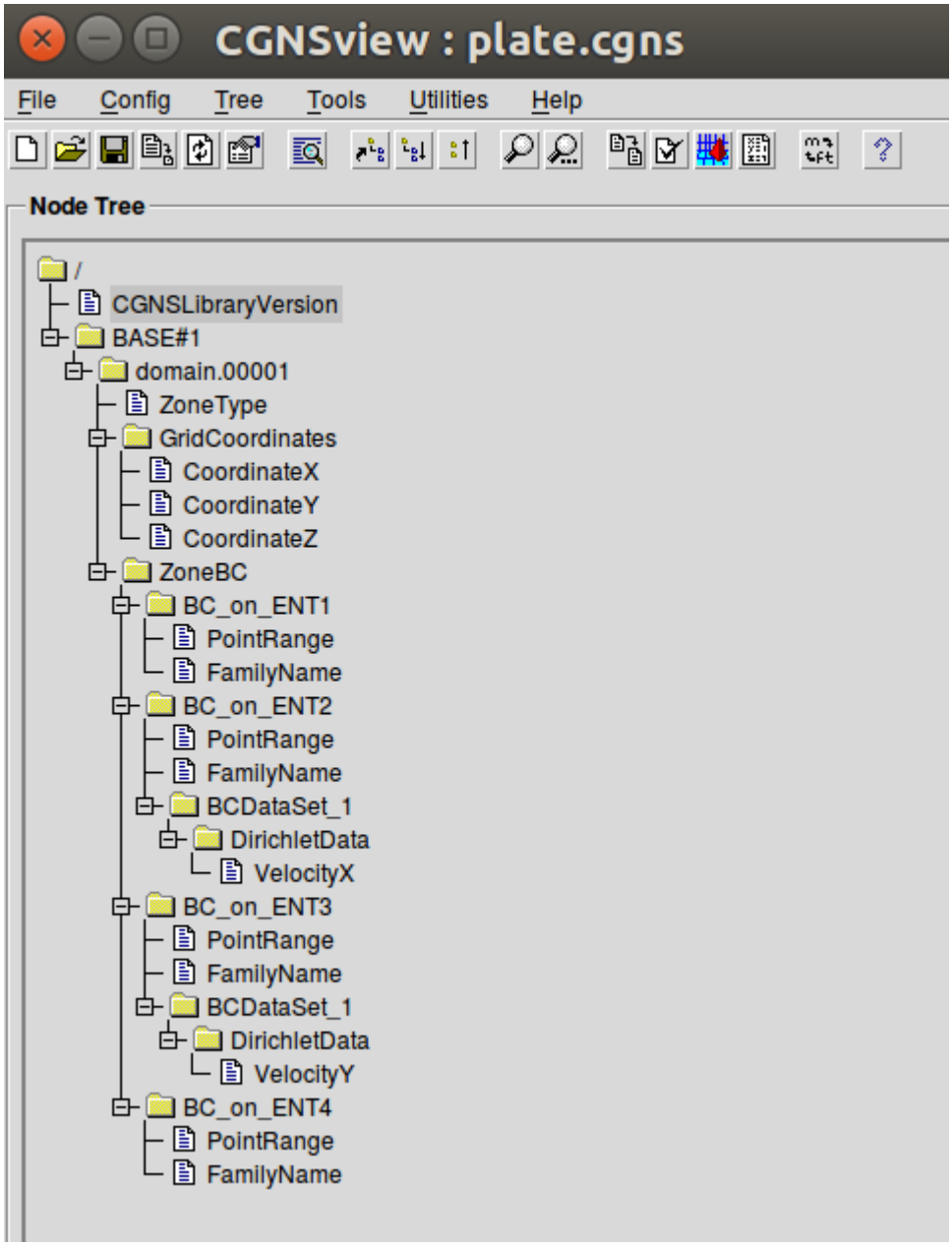
Click on 'Done' to finally conclude export procedure! A CGNS file should appear on your working folder. It will have the same name of the project file. This CGNS is ready to be used by pyHyp.

4.5 Checking the CGNS Structure

You can check if the CGNS file is correct by looking at its structure. If you have cgnslib installed, you can open the *cgnsview* GUI with the following command:

```
$ cgnsview
```

Open the newly generated CGNS file and expand its tree. For the flat square case, we have the following structure:



Note the VelocityX node indicating a X symmetry plane boundary condition and a VelocityY node indicating a constant Y plane boundary condition.

4.6 Boundary condition priorities

The corner nodes share two edges. Each edge may have different boundary conditions. The boundary condition at the corner node is chosen according to the following priority:

1. Constant X, Y, or Z planes
2. Symmetry X, Y, or Z planes
3. Splay

Therefore, if one edge that has a Splay BC is connected to another edge with a symmetry plane BC, the shared corner node will be computed with the symmetry plane BC.

4.7 Running pyHyp with the generated mesh

Create another empty folder and copy the CGNS file exported by ICEM to it. We can add the following Python script to the same folder:

```
from pyhyp import pyHyp

fileName = 'plate.cgns'
fileType = 'cgns'

options = {
    # ---------------------------
    #        Input File
    # ---------------------------
    'inputFile': fileName,
    'fileType': fileType,

    # ---------------------------
    #        Grid Parameters
    # ---------------------------
    'N': 65,
    's0': 1e-6,
    'marchDist': 2.5,

    # ---------------------------
    #   Pseudo Grid Parameters
    # ---------------------------
    'ps0': 1e-6,
    'pGridRatio': 1.15,
    'cMax': 5.0,

    # ---------------------------
    #   Smoothing parameters
    # ---------------------------
    'epsE': 1.0,
    'epsI': 2.0,
    'theta': 0.0,
    'volCoef': 0.3,
    'volBlend': 0.001,
    'volSmoothIter': 10,

    # ---------------------------
    #   Solution Parameters
    # ---------------------------
    'kspRelTol': 1e-15,
    'kspMaxIts': 1500,
    'kspSubspaceSize': 50,
    'writeMetrics': False,
    }
```

(continues on next page)

(continued from previous page)

```
hyp = pyHyp(options=options)
hyp.run()
hyp.writeCGNS('plate3D.cgns')
```

Save this script with the name 'generate_grid.py'. Then, navigate to the folder using the terminal and write the following command:

```
$ python generate_grid.py
```

This script will read the plate.cgns file (which contains the surface mesh and the boundary conditions) and will generate the plate3D.cgns file with the volume mesh. It is important to check the 'MinQuality' column of the screen output. A valid mesh should have only positive values.

You can also run pyHyp in parallel with the following command:

```
$ mpirun -np 4 python generate_grid.py
```

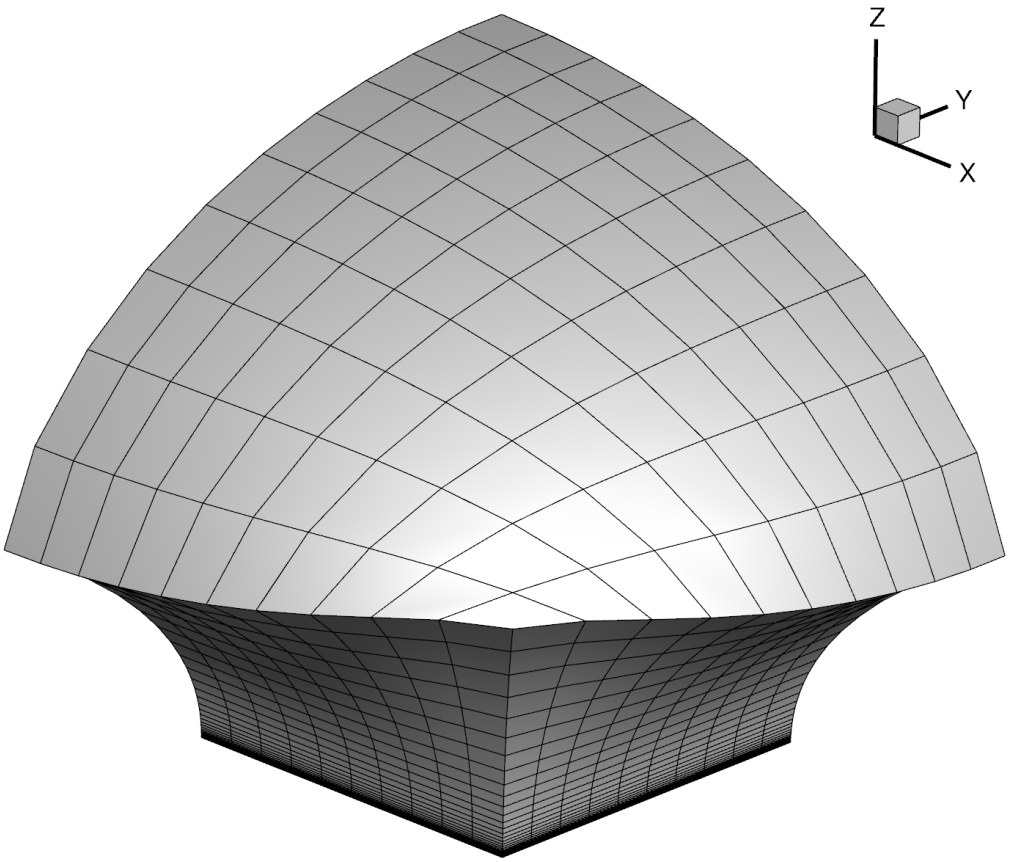
The option '-np 4' indicates that 4 processors will be used. The results may vary slight due to the parallel solution of the linear system.

4.8 Visualizing the mesh in TecPlot 360

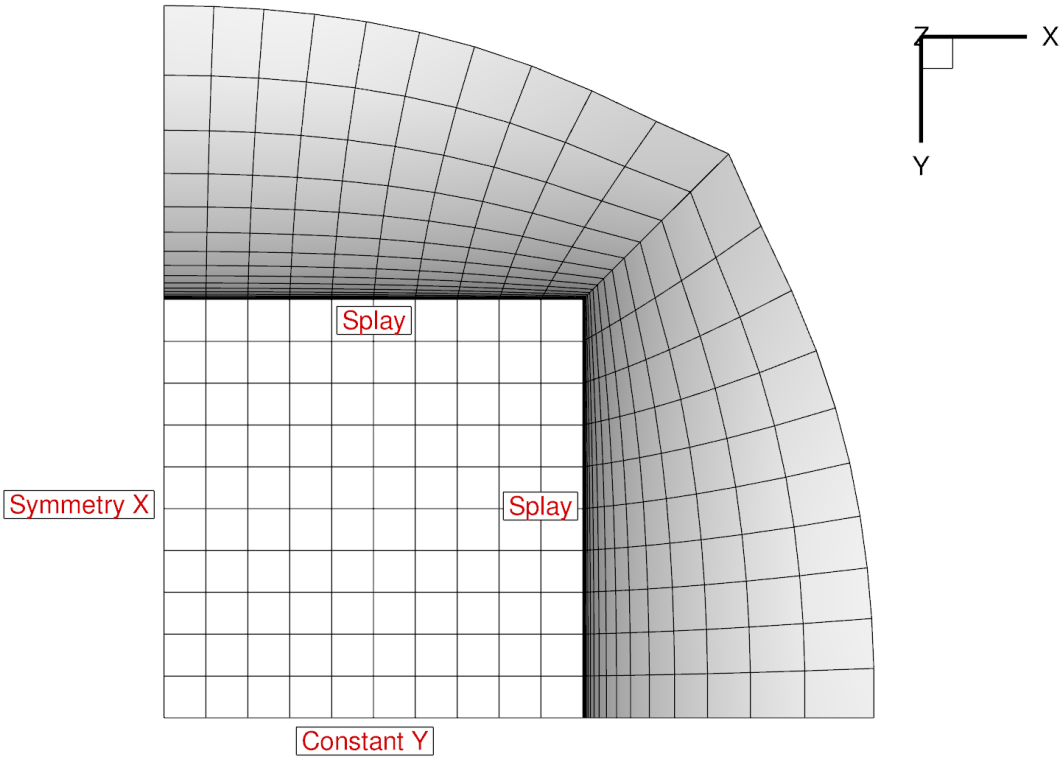
If you have TecPlot 360 installed in your computer you can visualize the volume mesh. Open a terminal and navigate to the folder than contains the newly generated CGNS file with the volume mesh. Then type the following command:

```
$ tec360
```

This will open TecPlot 360. On the upper menu select 'File' > 'Load Data Files', then choose your CGNS file. Next, check the 'Mesh' box on the left panel, and click 'Yes'. You will be able to visualize the mesh as shown below:



We can see that the boundary conditions where correctly applied. The image below shows a bottom view of the mesh:



Try playing with the different parameters to see their impact on the final mesh. In this case, it is helpful to save a TecPlot layout file. For instance, place the mesh in a position you want and click on 'File' > 'Save Layout File' and save it with the name you want (let's say layout_hyp.lay). Then you can open you mesh directly from the command line by typing:

```
$ tec360 layout_hyp.lay
```

Then you don't have to go over the menus all over again!

CHAPTER

FIVE

USING THE BC OPTION

This page describes how to use the *BC* option to specify boundary conditions at boundary edges of the surface mesh.

Here is an example of a dictionary that can be used with *BC*:

```
"BC": {1: {"iLow": "ySymm"}, 3: {"jHigh": "splay", "iHigh": "xConst"}}
```

Each entry in the dictionary has a key and at least one nested key-value pair. The key is the 1-based block number, the nested key is the boundary edge specification, and the value is the boundary condition. For the first entry, these are `1`, `iLow`, and `ySymm`, respectively.

The 1-based block number and boundary edge specification for a boundary edge can be determined using Tecplot:

1. Load the surface mesh file into Tecplot.
2. Use the Probe tool to select a point on the boundary edge of interest. Use Ctrl+click to snap on to a boundary edge.
3. Select Zone/Cell Info in the toolbar on the right.
4. The number shown after Zone is the 1-based block number.
5. The edge specification depends on the values of I and J. The edge is `iLow` if I=1, `iHigh` if I=I-Max, `jLow` if J=1, or `jHigh` if J=J-Max. Only one of these is true for any boundary edge.

The supported boundary conditions are:

- `splay` for free edges
- `xSymm`, `ySymm`, `zSymm` for symmetry planes
- `xConst`, `yConst`, `zConst`, `xyConst`, `yzConst`, `xzConst` for constant planes

CHAPTER

SIX

OPTIONS

inputFile: str

Name of the file that contains the surface mesh. This is a file that has been generated in an external meshing program, typically ICEMCFD.

patches: list = []

Explicitly assign arrays of patches as the input surface.

fileType: str = PLOT3D

Type of the input file.

- `PLOT3D`: PLOT3D format
- `CGNS`: CGNS format

skip: bool = False

Flag to entirely skip the grid generation of this geometry.

mode: str = hyperbolic

Type of extrusion.

- `hyperbolic`: Most commonly used
- `elliptic`: Not typically used

unattachedEdgesAreSymmetry: bool = True

Automatically applies symmetry boundary conditions to any edges that do not interface with another block. This option may fail in rare cases. See *here* for details.

outerFaceBC: str = farfield

Specifies the boundary condition at the outermost face of the extruded mesh.

- `farfield`: Farfield BC
- `overset`: Used for overset component meshes

BC: dict = {}

Specifies boundary condition information for specific block edges. See *here* for details.

families: str or dict = {}

Name given to wall surfaces. If a dictionary is submitted, each wall patch can be named separately. This can help with applying operations to specific wall patches.

autoConnect: bool = True

Run the cgnsUtilities `connect` function to add any necessary block to block connectivity.

noPointReduce: bool = False

Do not find duplicate nodes along edges. This can only be used with single surface input files.

N: int = 65

Number of grid levels to march. This determines the grid dimension in the off-wall direction. Typically, this should be a "multi-grid" friendly number.

s0: float = 0.01

Initial off-wall (normal) spacing of grid. This is taken to be constant across the entire geometry. The units are consistent with the rest of the geometry.

nConstantStart: int = 1

Number of constant off-wall layers before beginning stretch.

nConstantEnd: int = 1

Number of constant layers at the end of the march.

nTruncate: int = -1

This will stop the mesh marching after the specified number of levels. Specifying $1 < \texttt{nTruncate} < N$ will produce a mesh that is identical to the first `nTruncate` levels of the full mesh with N levels. Values outside of this range have no effect. This option is mainly useful for debugging low quality cells or negative volumes in the first few layers of the mesh.

marchDist: float = 50.0

Distance to march in the normal direction. Most wing geometries will have a distance such that the farfield boundary is 10 to 20 span lengths away from the geometry.

nodeTol: float = 1e-08

Tolerance for nodes to be treated as identical.

splay: float = 0.25

Splay BC spreading factor. This controls how far the floating edges splay outwards and can be useful for increasing the volume overlap in overset meshes.

splayEdgeOrthogonality: float = 0.1

How hard to try to force orthogonality at splay edges. Should be between 0 and 0.5.

splayCornerOrthogonality: float = 0.2

How hard to try to force orthogonality at splay corners.

cornerAngle: float = 60.0

Maximum convex corner angle in degrees necessary to trigger the implicit node averaging scheme. See Section 8 of *Chan and Steger* for more information.

coarsen: int = 1

Automatically coarsen a surface mesh before starting extrusion. 1 gives the same surface mesh. 2 coarsens by a factor of 2 in each direction. 3 coarsens by a factor of 4 in each direction, and so on.

panelEps: float = 1e-08

Only used in elliptic mode. Distance source panels are "below" nodes. This parameter usually does not need to be changed.

farfieldTolerance: float = 4.0

Only used in elliptic mode. The multiple of the panel length cutoff to use the approximation formula.

useMatrixFree: bool = True

Only used in elliptic mode. Use matrix-free solution technique. This is always True when *evalMode* is `fast`.

evalMode: str = fast

Only used in elliptic mode. Type of panel evaluation routine.

- `fast`: Uses farfield approximations and panel groupings
- `exact`: Modifies the farfield tolerance to ensure that only the exact evaluations are used
- `slow`: Uses farfield approximations but does not group panels

sourceStrengthFile: str = panelStrength.source

Only used in elliptic mode. File to use to load/save the source strengths on the surface.

cMax: float = 1.0

The maximum permissible ratio of the step in the marching direction to the length of any in-plane edge. This parameter effectively operates as a CFL-type limit. If a step would result in a ratio `c` greater than `cMax`, the step is automatically split internally to respect this user-supplied limit. Increased robustness can be achieved at the expense of computational cost by lowering `cMax`.

nonLinear: bool = False

Use the nonlinear formulation. This is experimental and not currently recommended and may not work at all.

slExp: float = 0.15

Exponent used in the scaling function S_l, which is computed as $S_l = \left(\frac{\text{Distance from wall}}{\texttt{marchDist}}\right)^{\texttt{slExp}}$. S_l scales the explicit smoothing such that it is low near the wall to maintain orthogonality and high away from the wall to prevent crossing of grid lines in concave regions. This is the same purpose as described in Section 6 of *Chan and Steger*, but the computation is different. The exponent determines how quickly the smoothing ramps up as the extrusion moves away from the wall. An exponent closer to zero will result in steeper ramping.

ps0: float = -1.0

Initial pseudo off-wall spacing. This spacing **must** be less than or equal to `s0`. This is the actual spacing the hyperbolic scheme uses. The solver may take many pseudo steps before the first real grid level at `s0`. This is computed internally if a non-positive value is provided.

pGridRatio: float = -1.0

The ratio between successive levels in the pseudo grid. This will be typically somewhere between ~1.05 for large grids to 1.2 for small grids. This number is **not** the actual grid spacing of the final grid; that spacing ratio is computed and displayed at the beginning of a calculation. The `pGridRatio` **must** be smaller than that number. This is computed internally if a non-positive value is provided.

epsE: float = 1.0

The explicit smoothing parameter. Typical values are approximately 1.0. Increasing the explicit smoothing may result in a smoother grid, at the expense of orthogonality. If the geometry has very sharp corners, too much explicit smoothing will cause the solver to rapidly "soften" the corner and the grid will fold back on itself. In concave corners, additional smoothing will prevent lines from crossing (avoiding negative cells). See Section 3 of *Chan and Steger* for more information.

epsI: float = 2.0

Implicit smoothing parameter. Typical values are from 2.0 to 6.0. Generally increasing the implicit coefficient results in a more stable solution procedure. Usually this value should be twice the explicit smoothing parameter. See Section 3 of *Chan and Steger* for more information.

theta: float = 3.0

Kinsey-Barth coefficient. This provides additional implicit smoothing and is useful for preventing grid lines from crossing at concave corners. A single theta value is used for both in-plane directions. Typical values are ~2.0 to ~4.0. See Section 3 of *Chan and Steger* for more information.

volCoef: float = 0.25

Coefficient used in point-Jacobi local volume smoothing algorithm. The value should be between 0 and 1. Larger values will result in a more uniform cell volume distribution. Alternatively, use more *volSmoothIter* for stronger local smoothing. See Section 5 of *Chan and Steger* for more information.

volBlend: float = 0.0001

The global volume blending coefficient. This value will typically be very small, especially if you have widely varying cell sizes. Typical values are from ~0 to 0.001.

volSmoothIter: int = 100

The number of point-Jacobi local volume smoothing iterations to perform at each level. More iterations will result in a more uniform cell volume distribution.

volSmoothSchedule: list or NoneType = None

Define a piecewise linear schedule for volume smoothing iterations. If provided, this supersedes *volSmoothIter*. This option is usually used to limit the number of smoothing iterations early in the extrusion to maintain orthogonality near the wall and ramp up the number of smoothing iterations later in the extrusion to achieve a more uniform cell volume distribution in the farfield. An example of a smoothing schedule is `"volSmoothSchedule": [[0, 10], [0.4, 50], [1.0, 100]]`. In this example, the number of smoothing iterations increases linearly from 10 to 50 over the first 40% of grid levels. For the remaining levels, the number of smoothing iterations increases linearly from 50 to 100.

KSPRelTol: float = 1e-08

Tolerance for the solution of the linear system at each iteration. Typically 1×10^{-8} is sufficient. Very difficult cases may benefit from a tighter convergence tolerance.

KSPMaxIts: int = 500

Maximum number of iterations to perform for each step. The default should be sufficient for most cases.

KSPSubspaceSize: int = 50

Size of the Krylov subspace. Very large and difficult problems may benefit from a larger subspace size.

writeMetrics: bool = False

Flag to write the mesh gradients to the solution file. This option should only be used for debugging purposes.

outputType: str = CGNS

Output format for the volume mesh.

- `CGNS`: CGNS format
- `PLOT3D`: PLOT3D format

outputFile: str or NoneType = None

Output filename. If None, an automatic filename will be generated by appending "_hyp" to the input filename.

CHAPTER

SEVEN

PYHYP API

class pyhyp.pyHyp.**pyHyp**(**args*, ***kwargs*)

Create the pyHyp object.

Parameters

comm
[MPI_INTRACOMM] Comm to use. This is used when running in parallel. If not provided, MPI.COMM_WORLD is used by default.

options
[dict] A dictionary containing the the options for pyHyp.

debug
[bool] Flag used to specify if debugging. This only needs to be set to true when using a symbolic debugger.

freezeEdge(*blockID*, *edge*, *dstar*)

Specify an edge that will be frozen.

Parameters

blockID
[integer] Index of block IN ONE BASED ORDERING.

edge
[str] String specified for edge. One of 'ilow', 'ihigh', 'jlow', 'jhigh'

dstart
[float] How much these nodes will influence points around it.

freezeFaces(*blockIDs*, *dstar*)

Specify one or more faces (blocks) that will be frozen

Parameters

blockIDs
[integer or list] Index of block(s) IN ONE BASED ORDERING.

dstart
[float] How much these nodes will influence points around it.

getSurfaceCoordinates()

Return the surface coordinates on this processor

run()

Run given using the options given

`setSurfaceCoordinates`(*coords*)

Set the surface coordinates on this processor

`surfaceSmooth`(*nIter*, *stepSize*, *surfFile=None*)

Run smoothing iterations on the body surface

Parameters

nIter
[int] Number of iterations to run

stepSize
[float] Size of step. Must be < 1. Usually less than 0.1 for stability reasons.

`writeCGNS`(*fileName*)

After we have generated a grid, write it out in a properly formatted 1-Cell wide CGNS file suitable for running in SUmb.

`writeLayer`(*fileName*, *layer=1*, *meshType='plot3d'*, *partitions=True*)

Write a single mesh layer out to a file for visualization or for other purposes.

Parameters

fileName
[str] Filename to use. Should have .fmt extension for plot3d or .dat for tecplot

layer
[int] Index of layer to print. Values greater than 1 are only valid if the mesh has already been extruded.

meshType
[str] Type of mesh to write. The two valid arguments are 'plot3d' and 'fe'. The plot3d will write the mesh in the original plot3d format while the FE mesh is the unstructured internal representation.

partitions
[bool] This flag which is only used for the 'fe' mesh type option will write a separate zone for each partition on each processor. This is useful for visualizing the parallel mesh decomposition.

`writeOutput`(*fileName=None*, *fileType=None*)

This selects the output type based on what is specified in the options

`writePlot3D`(*fileName*)

After we have generated a grid, write it out to a plot3d file for the user to look at

`class` pyhyp.pyHyp.`pyHypMulti`(*comm=None*, *options=None*, *commonOptions=None*, *debug=False*, *skipList=[]*)

This is class can be used to run multiple pyHyp cases at once.

The inititalization method will setup, run, and write all the results.

Parameters

options
[object] ORDERED dictionary or list of dictionaries. This contains options for the extrusion of all several grids. An example of option dictionary is given below:

```
options = {
    "epsE": 4.0,
    "epsI": 8.0,
    "outputFile": "corner_hyp.cgns",
    "skip": False,
}
```

We can set a list of dictionaries as input:

```
options1 = {
    "epsE": 4.0,
    "epsI": 8.0,
    "outputFile": "corner1_hyp.cgns",
    "skip": False,
}
options2 = "cartesian.cgns"
options3 = {
    "epsE": 2.0,
    "epsI": 4.0,
    "outputFile": "corner2_hyp.cgns",
    "skip": False,
}
options = [options1, options2, options3]
```

Alternatively, we can set an ORDERED dictionary of dictionaries as input:

```
from collections import OrderedDict

options = OrderedDict()
options["case1"] = {
    "epsE": 4.0,
    "epsI": 8.0,
    "outputFile": "corner1_hyp.cgns",
    "skip": False,
}
options["block"] = "cartesian.cgns"
options["case2"] = {
    "epsE": 2.0,
    "epsI": 4.0,
    "outputFile": "corner2_hyp.cgns",
    "skip": False,
}
```

Each element of the list/dictionary will be considered as a different case. One of the elements can be a string specifying a CGNS file that should be combined with the other grids in the end. pyHyp will not do anything with this file except combine it with the generated grids in the corresponding order. These options will overwrite the default options (defined in the pyHyp class) and the common options (another argument of this method). If the user gives a list, this will be converted to a dictionary with integers as keys. Remember this when setting the skip list for unnamed cases.

commomOptions

[dict] Dictionary with options that should be applied to all cases in the options dictionary. See the 'defOpts' dictionary defined in the pyHyp class to see the available options.

skip_list
[list] List containing names of cases that should be skipped.

`combineCGNS`(*combinedFile='combined.cgns'*, *additionalGrids=[]*, *skipList=[]*, *eraseFiles=True*)

This will gather all newly generated grids and combine them in a single CGNS file. This only works for CGNS output files.

Parameters

combinedFile
[str] The name of the combined output file.

additionalGrids
[list] The filenames of any grids that were not generated by the current pyHypMulti object, but should still be included in the combined output file.

skipList
[list] The keys of any generated grids that should not be included in the combined output file.

eraseFiles
[bool] If True, we erase the individual files that are combined.

`writeLog`()

This will print a log with important information regarding all grids

pyhyp.utils.`simpleOCart`(*inputGrid*, *dh*, *hExtra*, *nExtra*, *sym*, *mgcycle*, *outFile*, *userOptions=None*, *xBounds=None*, *useFarfield=True*)

Generates a Cartesian mesh around the provided grid, surrounded by an O-mesh.

Parameters

inputGrid
[cgnsutils Grid object or str] If a cgnsutils Grid object is provided, we use it as is. Alternatively if a string is provided, we treat it as the name of the nearfield CGNS file to mesh around.

dh
[float or list of float] The target edge length of each cell in the Cartesian part of the mesh. A list of (x, y, z) lengths can be provided to make non-cubic cells. The actual edge lengths will depend on mgcycle.

hExtra
[float] The distance from the Cartesian mesh boundary to the farfield.

nExtra
[int] The number of layers to extrude the hyperbolic O-mesh.

sym
[str or list of str] Axis or plane of symmetry. One or more of ('x', 'y', 'z', 'xmin', 'xmax', 'ymin', 'ymax', 'zmin', 'zmax').

mgcycle
[int or list of int] Number of times mesh should be able to be coarsened for multigrid cycles. A list can be provided for nonuniform (x, y, z) coarsening.

outFile
[str] Output file name.

userOptions
[dict, optional] Custom pyhyp options to be used with this extrusion. If overset BCs are desired on the outer face, do not set it in this dictionary because after extrusion we overwrite

all BCs on the combined grid. See the option useFarfield below. The default value (True) results in farfield BCs on the outer face, and setting it to false results in overset for the far face. Other pyhyp extrusion parameters can be set here.

xBounds

[array (2 x 3), optional] Optional bounding box coordinates desired for the center cartesian grid. The default value can be obtained by: `xMin, xMax = grid.getBoundingBox()`, and then the xBounds array can be set as `xBounds = [xMin, xMax]`. This option allows users to modify the bounding box coordinates rather than simply defaulting to the bounding box of the nearfield grid.

useFarfield

[bool, optional] Optional flag to control the outermost layer's BC. Default, `True`, will result in a farfield outer layer, setting this to `False` does overset BCs on the outermost layer

CHAPTER

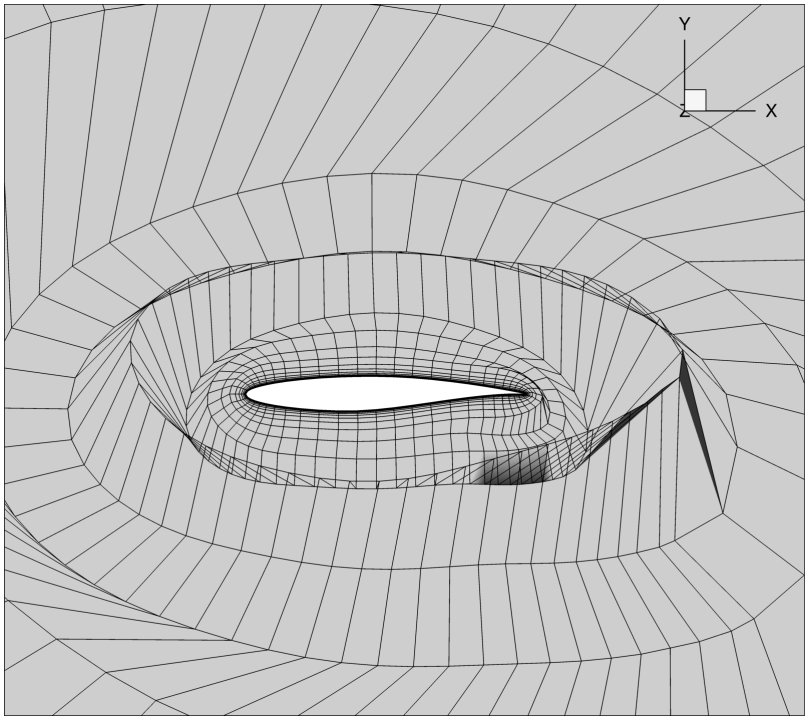
EIGHT

TIPS FOR COARSE GRIDS

Generating very coarse meshes for adaptive runs can be challenging with pyHyp. This page discusses a few tips and tricks to keep in mind when trying to make starting meshes for adaptive runs. Most of this information assumes that the final grid is independent of the starting grid. This means that most of these tips are given with the intent of getting the first grid to converge any sort of solution that will allow the adaptive meshing algorithm to take over. The coarser the initial mesh, the more work the adaptive algorithm can do, hopefully reducing the discretization error as much as possible.

The obvious way to do this is to generate a typical mesh and then call `cgns_utils coarsen` to coarsen the grid to the desired level. This method does work, but it doesn't provide the user with obvious ways to make quick changes to a grid. As an example, how do we ensure that a grid will have an initial off wall spacing of a certain value after a number of calls to `cgns_utils`? The answer is too much math and thinking, and it would unnecessarily add some number of `cgns_utils coarsen` calls to the workflow. Suffice to say it would be much easier to make a grid where the initial output is of interest rather than the output after a number of calls to `cgns_utils`. With that in mind, this discussion will provide methods for getting a coarse grid straight out of pyHyp.

This discussion will use a 2D RAE 2822 airfoil as an example. Even though this is restricted to a rather simple 2D case, the tips can be extended to 3D and more complex geometries. Below is an example of a mesh that pyHyp created with obvious negative volumes. This is a bit of a ridiculous example, but someone actually managed to get a grid that looked like this on accident.



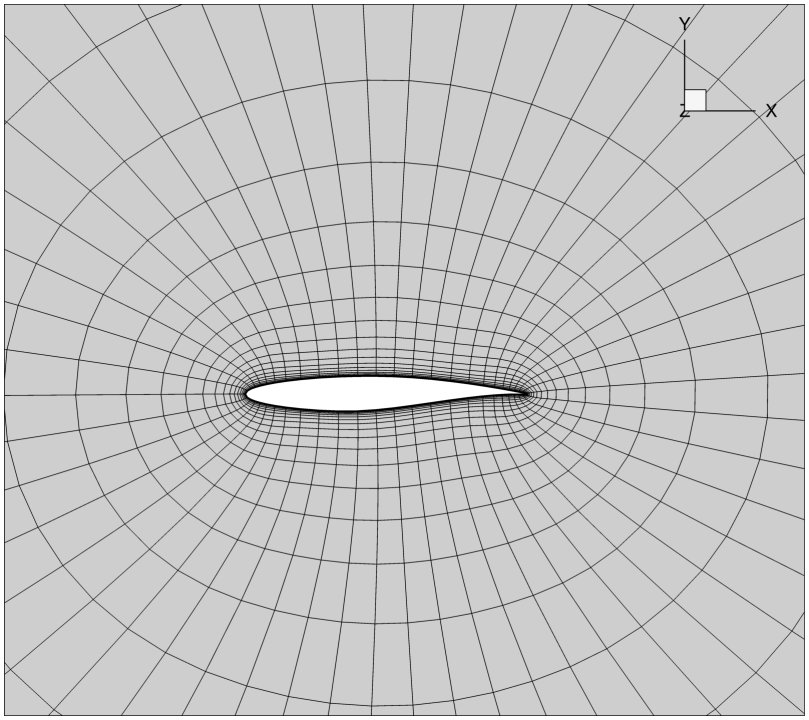
This grid was created using a surface mesh with 50 points around the airfoil and the following non-default options:

```
options = {
    # ---------------------------
    #        Input Parameters
    # ---------------------------
    "inputFile": "rae2822.xyz",
    "unattachedEdgesAreSymmetry": False,
    "outerFaceBC": "farfield",
    "autoConnect": True,
    "BC": {
        1: {
            "jLow": "zSymm",
            "jHigh": "zSymm",
        }
    },
    "families": "wall",
    # ---------------------------
    #        Grid Parameters
    # ---------------------------
    "N": 31,
    "s0": 4e-5,
    "marchDist": 100.0,
    "nConstantStart": 1,
    # ---------------------------
    #   Pseudo Grid Parameters
    # ---------------------------
    "cMax": 3.0,
}
```

There are a few ways to try and fix this problem with the grid. The first thing to check is to make sure that *cMax* is set to 1. *cMax* can be increased if the geometry is simple enough to speed up grid generation by sacrificing robustness. Whenever a grid is having trouble converging, always try decreasing *cMax* first. Even if this removes all the negative volumes, it may still result in some undesirable twist in the cells in the grid. This twist should be okay if the adaptive algorithm regenerates a grid at each step since it will disappear when the next grid is generated. However, if the adaptive algorithm splits cells that already exist, then it may be prudent to try to remove cells that are twisted to avoid skewed cells during adaptation. One possible fix is to increase *N*, the number of steps to reach the march distance. This allows pyHyp to take more gradual steps, meaning it is less likely to twist the grid into weird shapes.

Those first two tips will normally fix any obvious problems in the grid. Sometimes there may still be extremely skewed cells close to the airfoil, which sometimes can result in small negative volumes near the trailing edge. A key point to remember about this initial grid is that only a converged solution is needed, not a good one. In a typical grid, the initial grid spacing, *s0*, needs to be set to a certain amount based off a desired y+ value. To handle these skewed cells, more points around the airfoil would need to be sampled to retain the desired *s0*, but in this case more grid points are undesirable. In adaptive meshing, the algorithm is expected to determine what "value" of *s0* is needed. Therefore, *s0* should be increased to whatever the flow solver can converge and then the algorithm will worry about the initial grid spacing. In this case, *s0* could be increased to 1e-3 before the flow solver had trouble. This process can be trial and error, and with grids this coarse, it should not be too much trouble.

The next image shows a grid that pyHyp generated with the same surface mesh after modifying some of the options.



Much better!

```
options = {
    # --------------------------
    #        Input Parameters
    # --------------------------
    "inputFile": "rae2822.xyz",
    "unattachedEdgesAreSymmetry": False,
    "outerFaceBC": "farfield",
    "autoConnect": True,
    "BC": {
        1: {
            "jLow": "zSymm",
            "jHigh": "zSymm",
        }
    },
    "families": "wall",
    # --------------------------
    #        Grid Parameters
    # --------------------------
    "N": 31,
    "s0": 1e-3,
    "marchDist": 100.0,
    "nConstantStart": 1,
    # --------------------------
    #   Pseudo Grid Parameters
    # --------------------------
    "cMax": 1.0,
}
```

INDEX

U

V

W