

UNIVERSITY OF TWENTE

INTERNSHIP REPORT

Automated mesh generation for the rotator section of a radial inflow turbine

Author:

Tjarke van Jindelt

Supervisors:

Dr. Peter Jacobs

Dr. Carlos André de Miranda Ventura

UNIVERSITY OF TWENTE.



THE UNIVERSITY
OF QUEENSLAND
AUSTRALIA

Contents

Contents	ii
List of Figures	v
1 Introduction	1
1.1 Objective and scope of the internship	2
2 Background on meshes	3
2.1 Introduction	3
2.2 Structured and unstructured meshes	3
2.3 Radial inflow turbine mesh topology	5
3 Mesh generation	7
3.1 Introduction	7
3.2 Input	7
3.3 Loft	8
3.4 Computational Domain	9
3.5 Creation of the Topology	12
4 Summary	13
A Code explanations	15
A.1 Input interpreter	15
A.2 Loft	16
A.3 Computational domain	16
A.4 Automated block creation	17
A.5 Grid creation	18
A.6 Visualisation	19
B Mesh generation code	21
Bibliography	61

List of Figures

1.1	Sketch of a radial inflow turbine	2
2.1	Unstructured and structured grids	4
2.2	Grid topologies	4
2.3	Example of Mapping	5
2.4	Sketch of the computational domain	6
2.5	Rotator topography	6
3.1	Display of the input camberlines	8
3.2	Surface sketch	9
3.3	Structure of the computational domain	10
3.4	Photo of a rotator section of a radial inflow turbine	11
3.5	Display of the computational domain	11
3.6	Display of the complete topology	12
A.1	Arrangement of a hexaedral block in Eilmer3	16
A.2	Structure of the computational domain	17

Chapter 1

Introduction

World wide energy consumption is increasing, which is a challenge that is facing many societies. Paired with ecological threats such as climate change and smog the need for more renewable energy is apparent.

A contribution can be made with geothermal energy. Australia has enough geothermal energy to provide the countries electrical energy needs for 450 years according to an estimate by the Centre for International Economics[1]. Though many technical challenges still have to be solved to be able to extract this energy.

Another provider could be solar energy. Which can be obtained on a large scale in Australia's Deserts, as well as on a smaller scale locally.

Both solar and geothermal energy can be converted to electrical energy using turbines driven with fluids heated by either source of energy. Here the optimal shape depends on different parameters such as type of fluid, temperature of fluid, flow rate, etc. These parameters can change per site and application. Therefore an automated approach to design and optimize turbines is needed.

This internship builds on work done by C.A. de Miranda Ventura on radial inflow turbines for renewable power generation applications [2]. A sketch of a radial inflow turbine is given in figure 1.1.

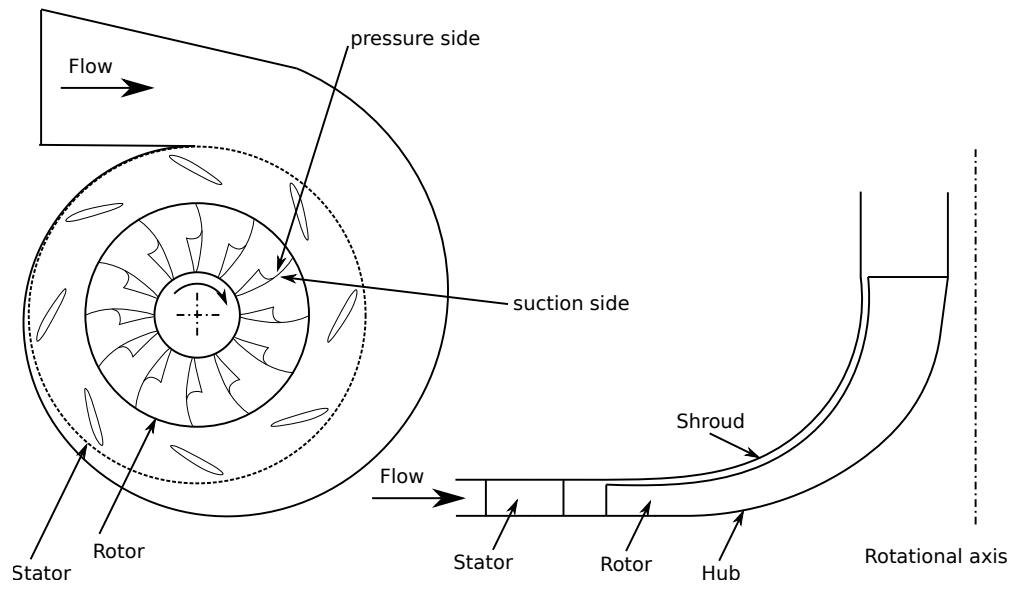


FIGURE 1.1: Sketches of a radial inflow turbine from different perspectives (after [2])

1.1 Objective and scope of the internship

The objective of this internship is to create an automated grid generation tool for one blade section of the rotator of a radial inflow turbine. The input provided are curves describing the blade at various hub to shroud locations.

This can then be used in a larger project to create a automated process of examining, designing and optimizing radial inflow turbines for given flow parameters.

Chapter 2

Background on meshes

2.1 Introduction

In Computational Fluid Dynamics (CFD) the flow is generally simulated with a finite volume approach. Which means the area one is interested in is separated into many small volumes. For each of these smaller volumes the flow quantities are determined through fluxes over their surfaces.

This conglomerate of volumes is called a mesh and the size and shape of its volumes determine whether or not a flow solver is able to accurately compute the flow one is interested in.

While smaller control volumes may be able to resolve even smallest scales of turbulence they also result in a larger amount of volumes per mesh. The amount of volumes determines the time that is needed to compute the flow. This time can mount up to unreasonable durations quite easily when fine meshes for practical applications are considered, especially when just a quick estimate is needed.

In this chapter a short explanation of meshing approaches is given and after that the mesh topology that will be used for the radial inflow turbine is explained.

2.2 Structured and unstructured meshes

There is a distinction between structured and unstructured grids of which examples are shown in figure [2.1](#).

The unstructured grid offers more flexibility in its creation, as the nodes (corners of a volume element) can be placed more freely. Therefore, unstructured grids have less problems when dealing with complex geometries and usually contain less nodes.

In contrast the clear order of blocks in the structured grid prescribes certain node distributions. While the structure normally increases the number of grid points, it also leaves some opportunities to save memory during calculation. This then allows different solving methods and can cut solving time.

Besides, there are flow features that are resolved better when structured grids are used e.g. boundary layers and vortices.

Eilmer3 is a flow solver that uses structured grids and therefore the grid that is created will be structured, where hexahedral elements are used.

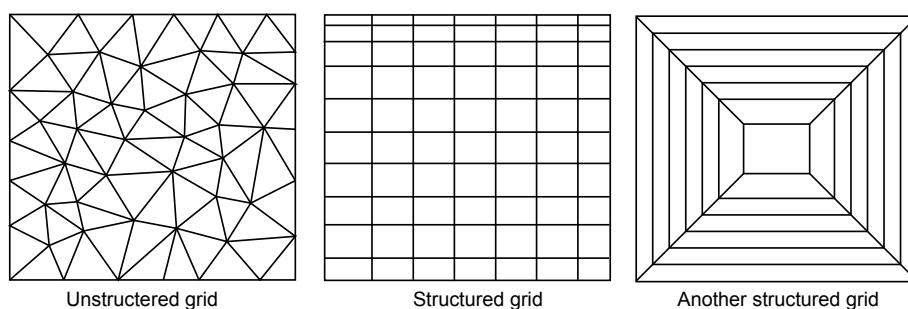


FIGURE 2.1: Examples of unstructured and structured grids

2.2.1 Topologies

There are different ways to structure a grid. The overall order is called the topology and three examples are given in 2.2. Here, the mesh is created around the black object in the center of each grid. The nomenclature of these topologies is straight forward, as they resemble letters of the alphabet.

In a grid highly skewed elements should be avoided. Therefore, round objects can be meshed best with an O-grid type approach, while square corners can be discretised with an H-grid approach.

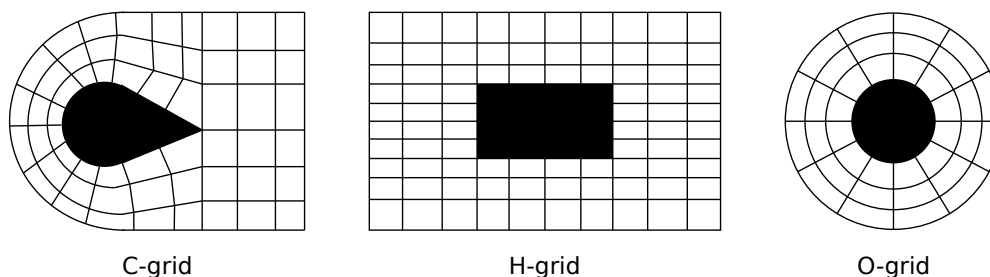


FIGURE 2.2: Example of a C-, a H- and a O-grid topology after [3]

2.2.2 Mapping

As said before, the structure of the mesh allows for different solving methods. This includes mapping from physical to computational coordinates. An example is shown in figure 2.3, where i and j are the computational coordinates.

Ordering mesh elements according to their computational coordinates can be done straight forward. This way the interfaces between elements can be identified more easily and reduce computational effort.

Mapping is used in the next chapter to create lofted surfaces, too.

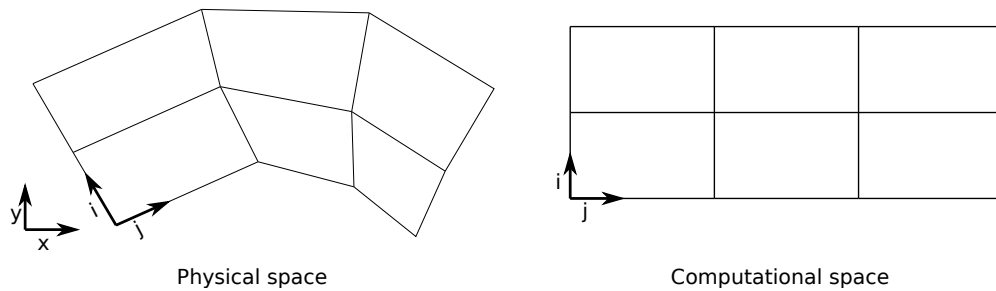


FIGURE 2.3: Example of mapping between the physical and computational space

2.3 Radial inflow turbine mesh topology

Generally it would be possible to mesh the whole turbine. However, for most calculations one can make use of the symmetry and only create a computational domain around one blade, as seen in figure 2.4, which shortens calculation time.

In this case periodic boundaries have to be used on the pressure and suction side, which means the outflow at a certain location on one side equals the inflow at the corresponding location on the other side.

It is chosen to use an O-H grid topology for the rotator section. This can be seen in figure 2.5, where the grid around the turbine is an O-grid and the grid around the O-grid is a H-grid topology. The topology will be extruded in the hub to shroud direction to form a 3-D grid. The blocks depicted here will be divided into smaller volumes, which make up the mesh.

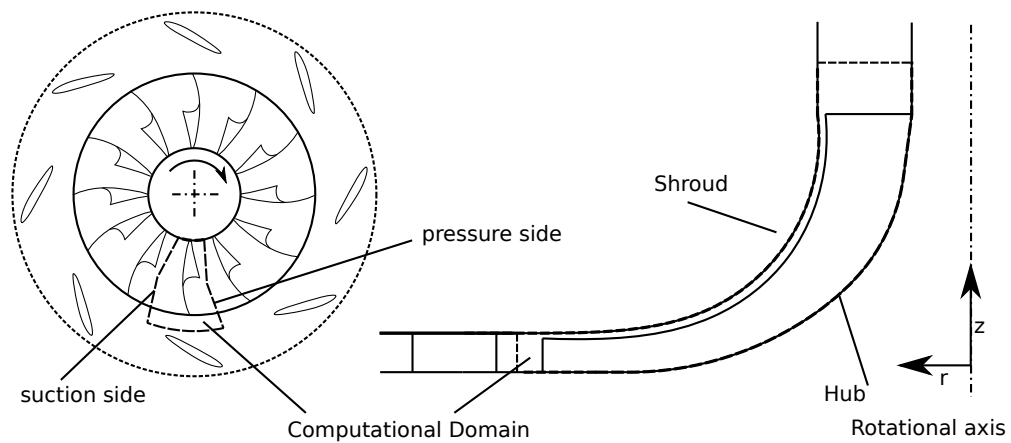


FIGURE 2.4: Sketch of the computational domain

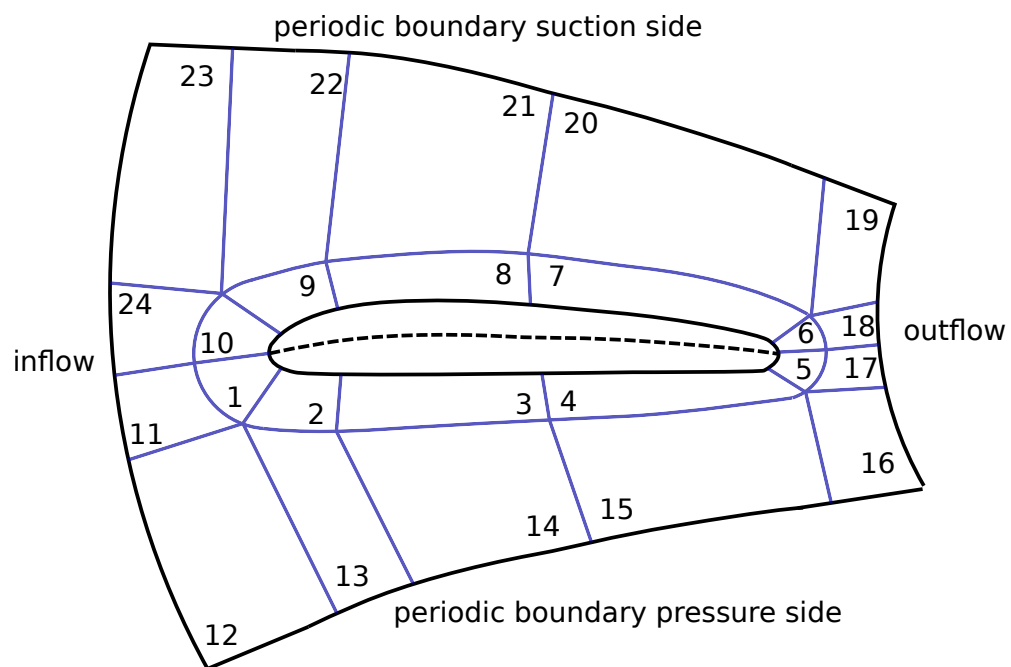


FIGURE 2.5: Sketch of the rotator topology which is extruded in the hub to shroud direction

Chapter 3

Mesh generation

3.1 Introduction

During the internship a python script was written that, in cooperation with the Eilmer3 code, creates a structured mesh from a given input. In this chapter this process is explained step by step, where a more detailed explanation of the code is given in the appendix [A](#).

The overall process is as follows. At first the input will be used to create the surfaces of the rotor. From that the computational domain is created. Then the computational domain is subdivided into a structure similar to the one seen in figure [2.5](#).

3.2 Input

The input consists of points along the curves describing the pressure and suction side as well as the camber line of the blade. There are several of these curves at different hub to shroud locations. These points are made into curves with the Eilmer3 spline function, which results for example in figure [3.1](#) for the camberlines of the blade.

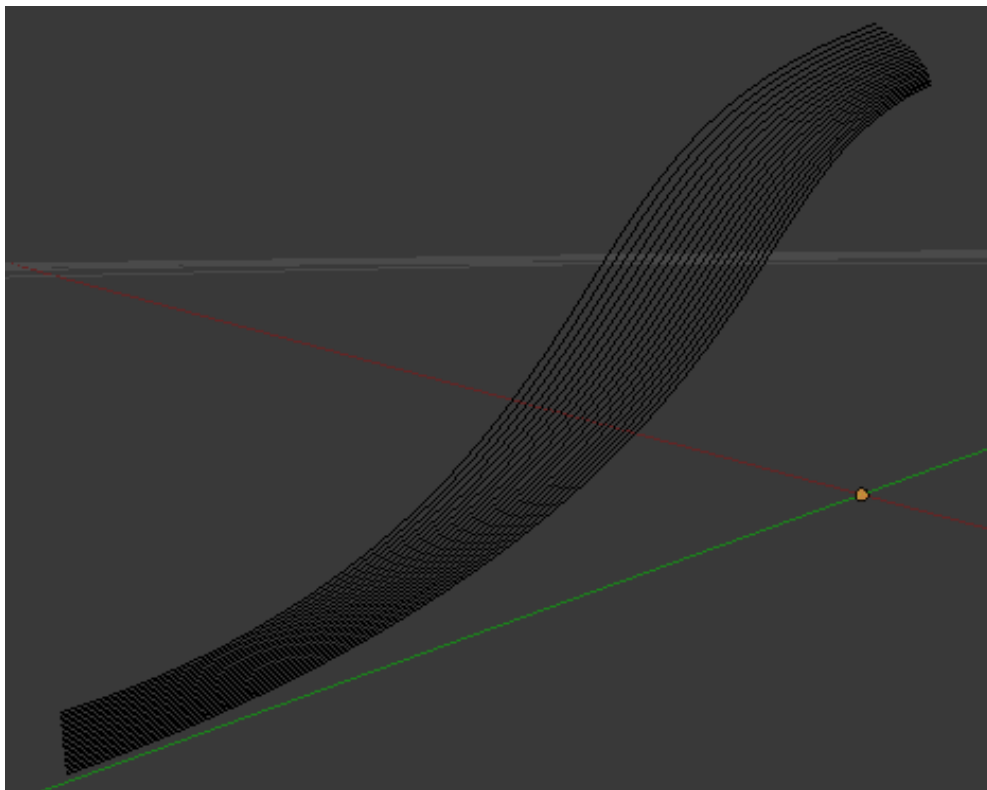


FIGURE 3.1: Camberline curves created from the input

3.3 Loft

From the curves given in the input a surface has to be created. This is done through lofting and while quite a few options are available to create surfaces with Eilmer3 this is not one of them. The following process describes how lofting was implemented.

Any continuous surface can be described with two coordinates, as can be seen in figure 3.2. The surface has an arbitrary distribution of points in the 3D space.

Define i and j as 0 at the beginning corner and 1 at their respective ends. Now a function $f(i, j)$ with $0 \leq i \leq 1$ and $0 \leq j \leq 1$ can describe the surface completely. This is applied to create a surface from the input curves.

In this case $i = 0$ is the inflow location, $i = 1$ is the outflow location, $j = 0$ is the hub location and $j = 1$ is the shroud location. Several curves in i direction are given in the input at different j positions. To obtain the coordinates of the surface for an arbitrary combination of i and j the following is done. Evaluate all input curves at location $i_{arbitrary}$ this will result in one point per input curve. A new curve in the j direction is then created using all these points. Evaluating this curve at its $j_{arbitrary}$ location gives the coordinates of the surface that one was looking for. The process above can be done for any combination of i and j and therefore describes the complete surface.

This is done for both the pressure side and the suction side of the surface.

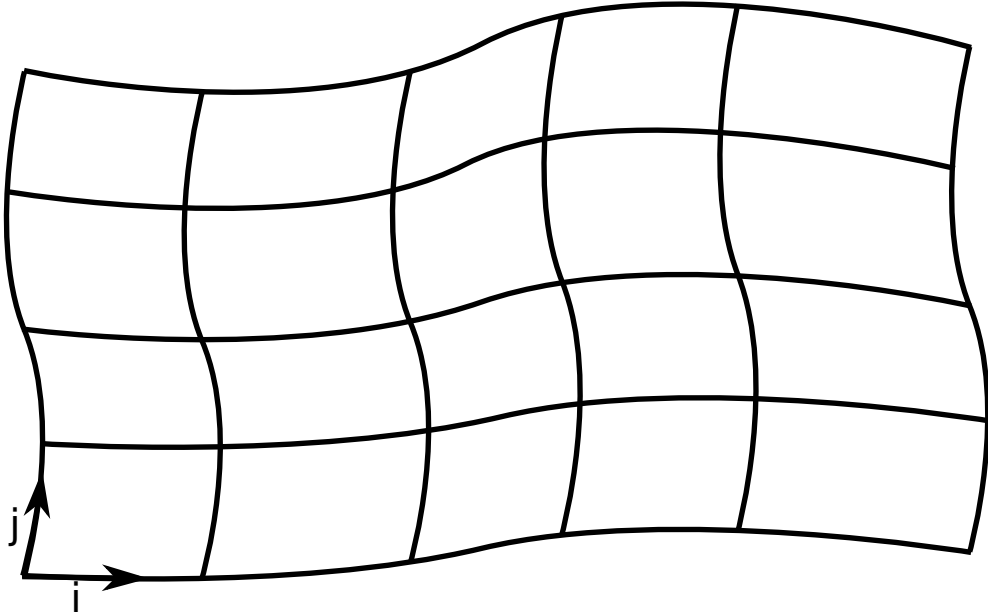


FIGURE 3.2: Sketch of a surface to explain lofting

3.4 Computational Domain

To create a hexahedral volume in the given environment it is sufficient to define all its six faces. This works in a similar way as described for the surface in the previous section, where a third computational dimension is added. The structure of these volume blocks is explained further in appendix A section A.3.

The whole computational domain is build up out of several blocks as seen in the topology sketch, figure 2.5. In the script the whole computational domain is created first by separately creating six volumes, as seen in figure 3.2. Their surfaces are then later used when blocks are created.

The division into volumes is sketched in figure 3.3 where this sketch has to be extruded in the hub to shroud direction to obtain volumes. It has to be mentioned that the clearance, the space between rotor and shroud is not yet taken into consideration in this code.

The surfaces of the volumes are of main interest here, especially the periodic boundaries.

- The hub and shroud surfaces follow from the geometry and can be implemented straight forward.
- The inflow surface is chosen to be part of a cylinder shell and the outflow surface is part of a disk.
- The surface of the blades the hub and the shroud follow from the input.

- The periodic boundaries are more complex and are explained below

The periodic boundaries cannot simply be surfaces with a constant azimuthal coordinate. That is because of the curvature of the blades, especially in the outflow section, which can be seen for example in figure 3.4. This curvature in combination with an option to use a large number of blades require a different kind of boundary.

The periodic boundaries consist of three parts, the inflow part belonging to volumes V5 and V6, the rotator part belonging to volume V1 and V4 and the outflow part belonging to V2 and V3. The inflow part flat surfaces with constant azimuthal position. The rotator part is a lofted surface created from rotated camberline curves. The outflow part of the periodic boundary is then created in a way that it smoothly connects to the camberline surface of V1 and V4. This is done by matching the slope $\frac{\partial\theta}{\partial z}$ of both surfaces at their connection.

With all surfaces the computational domain is defined as seen in figure 3.5 and the creation of the grid can begin.

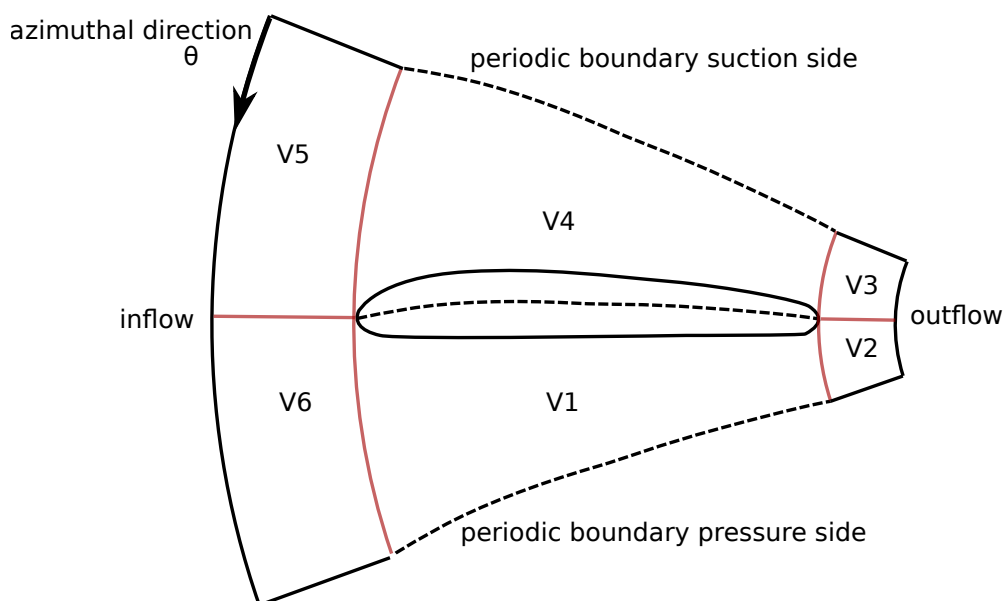


FIGURE 3.3: Sketch of the structure of the computational domain



FIGURE 3.4: Photo of a rotator section of a radial inflow turbine

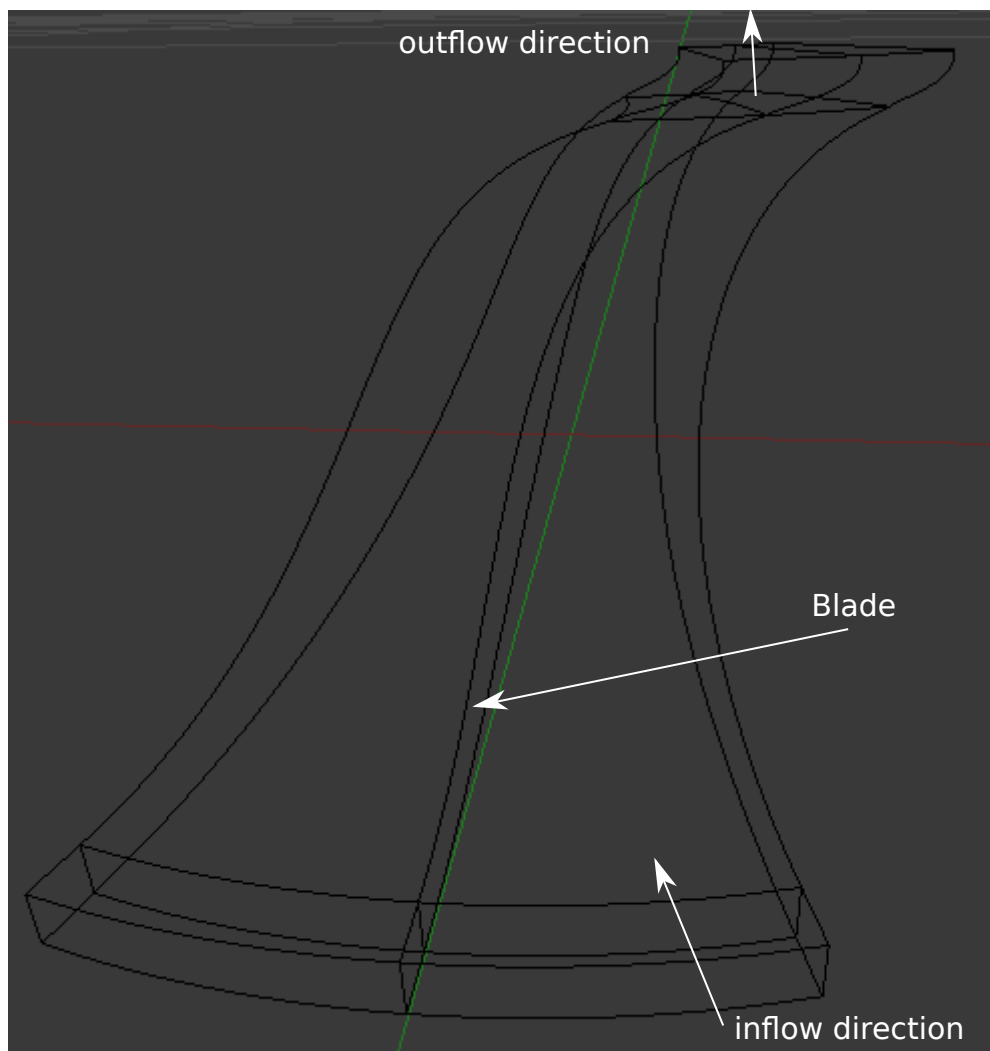


FIGURE 3.5: Image of the computational domain in 3D

3.5 Creation of the Topology

A sketch of the topology is given in figure 2.5. One has to keep in mind, that this is sketch of a cut through the volume at a hub to shroud location. Therefore the surface is not just 2D as displayed here, it actually follows the hub, inflow and outflow geometry. Thus the following operations seem straight forward but are complicated by the fact that they have to be done in the respective working surface.

The remaining lines (surfaces when extruded) that still need to be created are displayed blue in figure 2.5. First an O-grid line is created which has a constant distance to the blade. Then nodes are distributed over the inflow and outflow lines, over the periodic boundaries and over the O-grid line. These are then connected. This is done in a way that can easily be adapted, so that the number of blocks can be increased and their node position can be changed, see appendix A section A.5.1.

The resulting distribution in 3D can be seen in figure 3.6.

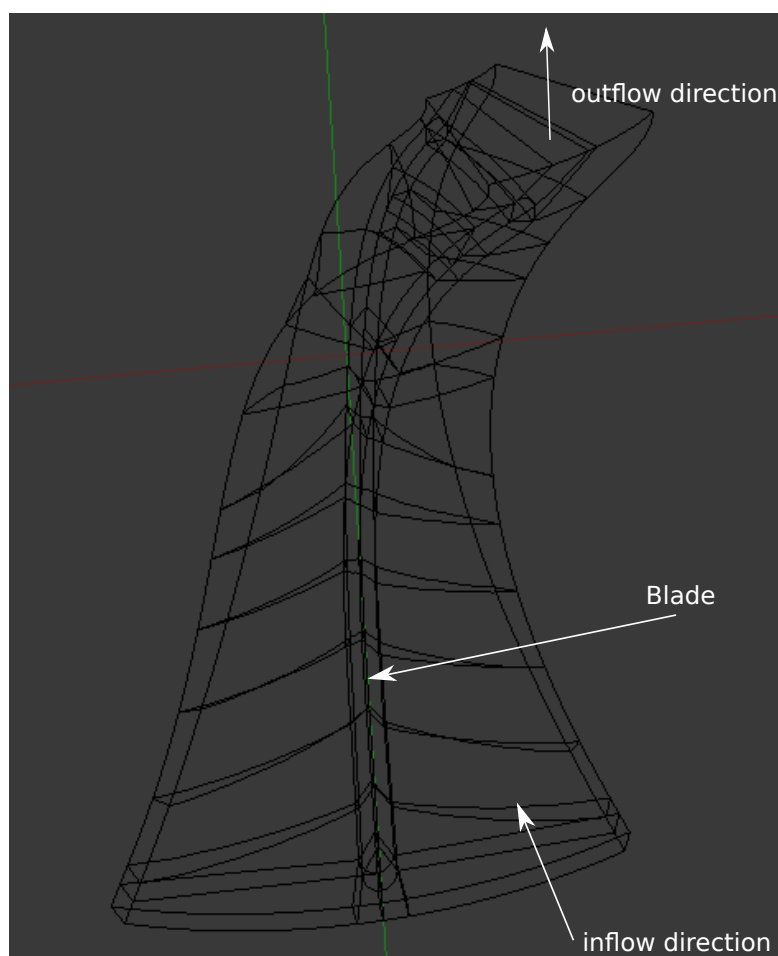


FIGURE 3.6: Image of the topology in 3D

Chapter 4

Summary

A script has been created to automatically create an OH-grid type structured mesh around blades of the rotor of an radial inflow turbine.

A way to loft a surface from a given set of curves has been found, which can easily be implemented into the Eilmer3 code and can be used for for many different applications.

The challenging geometry has been devided into an OH-grid type structure, where blocks are created automatically and the number of blocks and their location can be adjusted by changing very few input parameters.

The distribution of the cells along the boundaries still has to be adjusted to capture the boundary layer for example. Currently the cells are evenly divided along the block boundaries. Once this is achieved an automated process can be run to solve CFD calculations on this grid.

Another feature that might need to be added to the script is the ability to model the clearance in between blade and shroud.

Functions used in the code can be reused to create scripts for automatic mesh generation stators or other geometries.

Appendix A

Code explanations

In this appendix the script is explained in more detail. This includes an explanation of how to work with the script and which lines are responsible for what operation. Overall the script is run with the data preparation tool of Eilmer3. Command:

```
$e3prep.py --job = jobname
```

The input that is required consists of “.obj” files containing curves on the pressure side, suction side and the camberline of the blade. These files need to be ordered in hub to shroud direction and they should start at the leading edge and end at the trailing edge. Next to the curves given in the input also the number of blades has to be defined, as well as the inflow and outflow length. This will determine the size of the computational domain.

In general only the file location and number of blade has to be given and the rest is done automatically. For further fine-tuning see the different sections in this appendix.

During the code creation the pressure and suction side did not start at the leading and trailing edge therefore only the camberlines were used. Once this is implemented certain curves in this code have to be changed, most importantly Volume lines (line 158,337), and lines 880 - 882,1101 have match the correct geometries.

A.1 Input interpreter

The location of the input curves is determined in code lines 20 to 34. These have to be changed to be able to run the script. The “loader” function and the “input interpreter” function are used together to create splines from the input.

They make use of the structure of the “.obj” file to extract data points. A spline using the Eilmer3 spline function is then created through every point per curve.

A.2 Loft

The idea behind lofting is explained in section 3.3. This procedure can be implemented into the current Eilmer3 environment using only 20 lines of code, see the “Loft” function in the python script, which is the input to the already existing “PyFunctionSurface” in Eilmer3 to create a lofted surface.

It evaluates all input curves at the desired i location and creates a spline through all these points. This spline will then be evaluated at the desired j location and thus the Cartesian coordinates for any combination of i and j can be obtained.

However, this function assumes that the curves are given in the same i direction and that they follow a strict order in the j direction otherwise the resulting surface does not make sense. Furthermore in case of the radial inflow turbine the coordinates are assumed to be such, that the center axis is in the z direction and goes through the origin of a Cartesian coordinate system.

A.3 Computational domain

The arrangement of volumes is shown in figure 3.3. In Eilmer3 the volumes are arranged as displayed in A.1. The code is generated in a way, that the top face corresponds with the shroud, the bottom face corresponds with the hub, the west surface belongs to the inflow direction and the east surface belongs to the outflow direction.

The volumes are created in the code in lines 155 to 645.

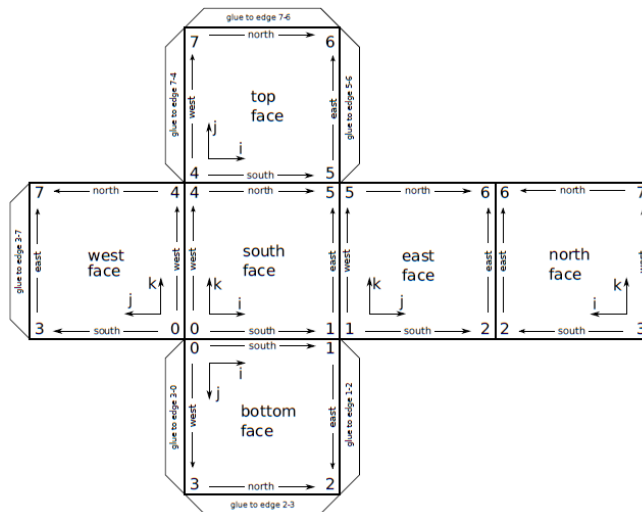


FIGURE A.1: Arrangement of a hexahedral block in Eilmer3 [4]

A.4 Automated block creation

A more detailed sketch of the topology is given in figure A.2. It contains the nomenclature used in the code. The following parts will explain how the remaining surfaces are created.

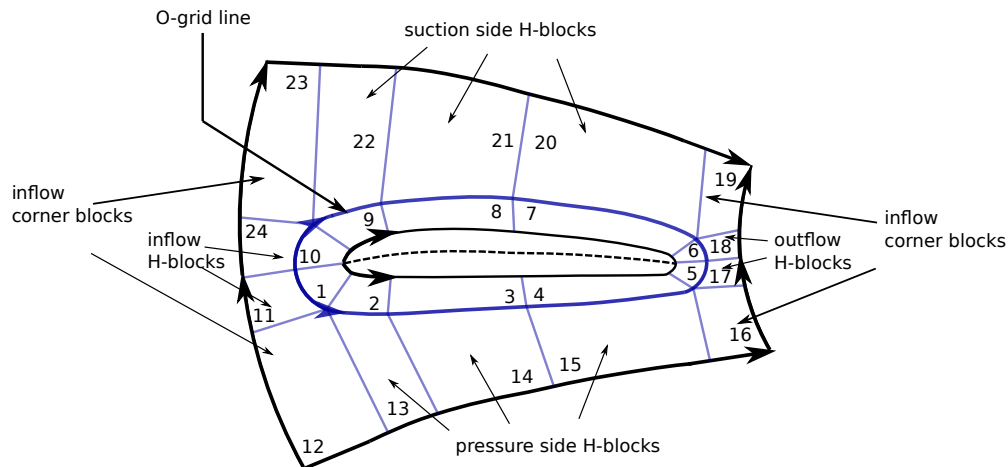


FIGURE A.2: Sketch of the structure of the computational domain

A.4.1 O-grid surface

The part of the code that is responsible for creating the O-grid surfaces is given in lines 655 to 888.

The O-grid line is created in the following way:

At 120 locations along the input curve points are created at a fixed distance to the blade (adjustable through `number_of_points_for_og1` and `distance_og1_blade`). It is made sure that these points lie on the working surface.

Through these points a curve is created to obtain a O-grid line. Then all the O-grid lines are lofted to create an O-grid surface.

A.4.2 Connection between nodes

The function `simple_connector` (lines 911 to 1096) creates a curve connecting two nodes. This curve is part of the working surface. The connecting surface is then created using its four surrounding paths (two times path on surface and two times simple connector curves).

A.4.3 Automated block creation functions

With the O-grid surface defined and the ability to connect nodes, automated functions are written to automatically create blocks (lines 1130 to 1945). Here a distinction is made between inflow H-blocks (`create_In_Block`), periodic side H-blocks (`create_H_Block`), outflow H-blocks (`create_Out_Block`), O-grid blocks (`create_O_Block`) and corner blocks, see sketch [A.2](#).

A.5 Grid creation

Using all the functions described above the complete grid can be created starting with the pressure side of the computational domain (lines 2092 to 2303). First the inflow H-blocks are created, followed by the inflow corner block then the periodic H-blocks, afterwards the outflow corner block is created and then the outflow H-blocks are created. Finally the blocks in the O-grid section are created. This process is repeated for the suction side.

The number of blocks created depends on the number of nodes defined along each curve. Thus changing the location and number of the nodes defines the topology of the computational domain. A preliminary selection of node locations has been done for the given geometry and is given in the next section. However, the location and number of nodes might have to be changed for different geometries.

A final adjustment that has to be made by the user is to activate the block creation parts in the code that are currently commented and include cluster functions to distribute the nodes along each block.

A.5.1 Node location

Node locations are chosen along the periodic boundaries, the blade, the O-grid surface, the inflow surface and the outflow surface. Here 0 stands for the beginning of the surface and 1 for its end. The directions are shown in figure [A.2](#).

The following node locations were chosen to produce figure [3.6](#) they can be found in the code at lines 1962 to 2074. A note of caution: When the number of inlet or outlet H-blocks is changed the corresponding node on the O-grid line has to be changed, see lines 2071 to 2074.

```

1 Nodes_on_Blade_pressure = [0,0.03,0.1,0.2,0.3,0.4,0.5,0.6,0.65,0.7,0.85,0.9,0.98,1]
2 Nodes_on_Blade_suction = [0,0.03,0.1,0.2,0.3,0.4,0.5,0.6,0.65,0.75,0.85,0.88,0.98,1]
3 Nodes_on_o_grid_line_pressure = [0,0.03,0.1,0.2,0.3,0.4,0.5,0.6,0.65,0.7,0.85,0.9,0.98,1]
4 Nodes_on_o_grid_line_suction = [0,0.03,0.1,0.2,0.3,0.4,0.5,0.6,0.65,0.7,0.82,0.9,0.96,1]
5 Nodes_on_inlet_pressure = [0.9,1]
6 Nodes_on_inlet_suction = [0.0,0.1]
```

```

7 Nodes_on_periodic_pressure = [0.15,0.28,0.36,0.4,0.44,0.475,0.5,0.55,0.57,0.60,0.62,0.65]
8 Nodes_on_periodic_suction = [0.15,0.28,0.36,0.4,0.44,0.475,0.5,0.55,0.60,0.62,0.7,0.9]
9 Nodes_on_outlet_pressure = [0.9,1]
10 Nodes_on_outlet_suction = [0.0,0.4]

```

A.6 Visualisation

During the creation of the script intermediate results had to be visualized. This is done both to check for failures in the code and to determine the location of certain blocks.

The preferred way to do this is with the Blender [5] software. First the mesh generation script is used to create text files containing coordinates of points along curves that should be displayed. These text files should have the following structure:

```

1 x1 y1 z1
2 x2 y2 z2
3 ...

```

In the code this is generally done by appending curve elements to the list of A. At the end of the code all curves contained in that list are written to separate text files.

Then Blender can be started in scripting mode. Use the script below where the file location and number of curves of to be changed. Run the script and it will load and display the desired curves.

```

1 ## Run this script in blender and it will create and display curves
2 ## that come from input files contain only x,y and z coordinates of points (one point per line
   ).
3 ## Just change the file location and number of curves and press run. Then the curves will be
   displayed.
4
5 import bpy
6 from mathutils import Vector
7
8 w = 1 # weight just leave this as 1
9 number_of_curves = 10 # gives the number of curves that are loaded into blender
10
11 for n in range (0,number_of_curves):
12     cList = None
13     cList = []
14
15     for line in open("/filelocation/file%d.txt" %n, "r"): ## change this to the proper
       file location
16         vals = line.split()
17         x = float(vals[0])
18         y = float(vals[1])
19         z = float(vals[2])
20         cList.append(Vector((x,y,z)))
21
22
23     curvedata = bpy.data.curves.new(name='Curve', type='CURVE')
24     curvedata.dimensions = '3D'
25
26     objectdata = bpy.data.objects.new("ObjCurve", curvedata)
27     objectdata.location = (0,0,0) #object origin
28     bpy.context.scene.objects.link(objectdata)
29
30     polyline = curvedata.splines.new('POLY')
31     polyline.points.add(len(cList)-1)

```

```
32     for num in range(len(cList)):  
33         x, y, z = cList[num]  
34         polyline.points[num].co = (x, y, z, w)
```

Appendix B

Mesh generation code

The whole code to generate the mesh

```
1 job_title = "Creating the mesh..."
2 print job_title
3
4 # Accept defaults for air giving R=287.1, gamma=1.4
5 select_gas_model(model='ideal gas', species=['air'])
6
7 initialCond = FlowCondition(p=1000.0, u=0.0, T=300.0)
8 M_inf = 4.0
9 u_inf = M_inf * initialCond.flow.gas.a
10 inflowCond = FlowCondition(p=50.0e3, u=u_inf, T=300.0)
11
12
13 ##### input needed for geometry
14
15 number_of_blades = 12          #for now 3 blades not working dont know why ./ everything else
16     is
17 clearance = 0.02    #not implemented yet
18 inflow_space = 0.05 #this factor times the camberline length gives the inflow length
19 outflow_space = 0.05
20
21 file_camber_hub = "/file_location/obj_3D_hub.obj"
22 file_camber_hub_to_shroud = "/file_location/obj_3D_hubShroudIntermediateLine%d.obj"
23 number_of_intermediate_camber_files = 20
24 file_camber_shroud = "/file_location/obj_3D_shroud.obj"
25
26 file_pressure_side_hub = "/file_location/obj_3D_hub_side2.obj"
27 file_pressure_side_hub_to_shroud = "/file_location/obj_3D_hubShroudIntermediates_pressureSide%
28     d.obj"
29 number_of_intermediate_pressure_files = 20
30 file_pressure_side_shroud = "/file_location/obj_3D_shroud_side2.obj"
31
32 file_suction_side_hub = "/file_location/obj_3D_hub_side1.obj"
33 file_suction_side_hub_to_shroud = "/file_location/obj_3D_hubShroudIntermediates_suctionSide%d.
34     obj"
35 number_of_intermediate_suction_files = 20
36 file_suction_side_shroud = "/file_location/obj_3D_shroud_side1.obj"
37
38 ##### File loader
39 def loader(filename, number_of_files):    ### load .obj files that are ordered by numbers e.g.
40     filename1 filename2 ... for number of files = 0 it just loads the file filename
41     verts=[]                               ### input example: filename =
42     obj_3D_hubShroudIntermediateLine%d.obj"
43     if number_of_files == 0:
44         for line in open(filename, "r"):
45             vals = line.split()
46             if vals[0] == "v":
47                 v = map(float, vals[1:4])
```

```

44         verts.append(v)
45     else:
46         for line in open(filename %number_of_files , "r"):
47             vals = line.split()
48             if vals[0] == "v":
49                 v = map(float , vals[1:4])
50                 verts.append(v)
51
52     return verts
53
54     ##### Create input curves from given files in the order : Hub,
55     hub to shroud intermediates , Shroud. All are located in input_curves
56     ##### The curve starts at the inflow blade, ends at outflow of
57     blade
58 def input_interpreter(hublocation , hub_to_shroud_location , shroudlocation ,
59     number_of_inbetween_files):
60     input_curves=[]
61     dummy = []
62
63     linecoords = loader(hublocation,0)        ### load all the points from one file , then
64     create a spline through it
65     for i in range(0,len(linecoords)):
66         dummy.append(Vector(linecoords[i][0] , linecoords[i][1] , linecoords[i][2]))
67     input_curves.append(Spline(dummy))
68
69     for n in range(1, number_of_inbetween_files+1):
70         Q=None
71         Q=[]
72         linecoords=loader(hub_to_shroud_location ,n)
73         for i in range(0,len(linecoords)):
74             Q.append(Vector(linecoords[i][0] , linecoords[i][1] , linecoords[i][2]))
75         input_curves.append(Spline(Q))
76
77     dummy=None
78     dummy=[]
79     linecoords = loader(shroudlocation ,0)
80     for i in range(0,len(linecoords)):
81         dummy.append(Vector(linecoords[i][0] , linecoords[i][1] , linecoords[i][2]))
82     input_curves.append(Spline(dummy))
83
84     return input_curves
85
86     ##### Define Loft surface ... r = 0 inflow r = 1 outflow , s = 0
87     hub s = 1 shroud
88     ##### PySurface only allows 2 inputs.. therefore make sure you
89     call input_curves = input_interpreter(hub,hts ,shroud,NoF) first !
90     ##### example : input_curves = input_interpreter(
91     file_camber_hub ,file_camber_hub_to_shroud ,file_camber_shroud ,
92     number_of_intermediate_camber_files)
93     ##### desired_lofted_surface = PyFunctionSurface(
94     LoftUserDefined)
95     ##### Volume 1
96
97     input_curves = input_interpreter(file_camber_hub ,file_camber_hub_to_shroud ,file_camber_shroud ,
98     number_of_intermediate_camber_files)
99
100 def Loft(input_curves):
101     input_curvesss = []
102     for i in range (0,len(input_curves)):
103         input_curvesss.append(input_curves[i].clone())
104     def LoftUserDefined(r,s):
105
106         Q=None
107         Q=[]
108
109         r_Curves=input_curvesss
110
111         for n in range(0,len(r_Curves)):
112             Q.append(r_Curves[n].eval(r))
113
114         s_Curve=Spline(Q)
115         Coords = s_Curve.eval(s)

```

```

109         return (Coords.x,Coords.y,Coords.z)
110     return LoftUserDefined
111
112
113     ##### rotate the lofted surface counterclockwise around z BE
114     CAREFULL ABOUT input_curves!!
115
116 def LoftRotator(input_curves):
117     input_curvesss = []
118     for i in range (0,len(input_curves)):
119         input_curvesss.append(input_curves[i].clone())
120     def LoftUserDefined(r,s):
121         Q=None
122         Q=[]
123
124         r_Curves=input_curvesss
125
126         for n in range(0,len(r_Curves)):
127             Q.append(r_Curves[n].eval(r))
128
129         s_Curve=Spline(Q)
130         Coords = s_Curve.eval(s)
131
132         return (Coords.x,Coords.y,Coords.z)
133
134 def LoftRotate(r,s):
135     from math import cos, sin, pi
136
137     theta = 2 * pi / number_of_blades *1/2 #only rotate half the angle but in both
138     directions of course .... =.
139     cc = cos (theta)
140     ss = sin (theta)
141
142     Coords = LoftUserDefined(r,s)
143     x = Coords[0]
144     y = Coords[1]
145     z = Coords[2]
146     x_r = x*cc-y*ss
147     y_r=x*ss+y*cc
148
149     return (x_r,y_r,z)
150
151     return LoftRotate
152
153     #####
154     ##### First Volume according to my definition of top bot etc.
155     print("Volume1#####")
156
157     #north = PyFunctionSurface(LoftUserDefined)
158     north = PyFunctionSurface(Loft(input_curves))
159     #south = PyFunctionSurface(LoftRotate)
160     south = PyFunctionSurface(LoftRotator(input_curves))
161
162     p2 = north.eval(1.0, 0.0)
163     p3 = north.eval(0.0, 0.0)
164     p6 = north.eval(1.0, 1.0)
165     p7 = north.eval(0.0, 1.0)
166
167     p0 = south.eval(0.0, 0.0)
168     p1 = south.eval(1.0, 0.0)
169     p4 = south.eval(0.0, 1.0)
170     p5 = south.eval(1.0, 1.0)
171
172     # print "p0=", p0, "p1=", p1
173
174
175
176     print(" top-----")
177     c56 = Arc(p5, p6, Vector(0,0,p5.z))
178     c47 = Arc(p4, p7, Vector(0,0,p4.z))
179     c76 = PathOnSurface(north, LinearFunction(1.0,0.0), LinearFunction(0.0,1.0))
180     c45 = PathOnSurface(south, LinearFunction(1.0,0.0), LinearFunction(0.0,1.0))
181     top = CoonsPatch(c45, c76, c47, c56, "top")

```

```

182
183
184 print(" bottom-----")
185 c03 = Arc(p0, p3, Vector(0,0,p0.z))
186 c12 = Arc(p1, p2, Vector(0,0,p1.z))
187 c01 = PathOnSurface(south, LinearFunction(1.0,0.0), LinearFunction(0.0,0.0))
188 c32 = PathOnSurface(north, LinearFunction(1.0,0.0), LinearFunction(0.0,0.0))
189 bottom = CoonsPatch(c01, c32, c03, c12, "bottom")
190
191
192 print(" west-----")
193 c37 = PathOnSurface(north, LinearFunction(0.0,0.0), LinearFunction(1.0,0.0))
194 c04 = PathOnSurface(south, LinearFunction(0.0,0.0), LinearFunction(1.0,0.0))
195 west = CoonsPatch(c03, c47, c04, c37,"west")
196
197
198 print(" east-----")
199 c26 = PathOnSurface(north, LinearFunction(0.0,1.0), LinearFunction(1.0,0.0))
200 c15 = PathOnSurface(south, LinearFunction(0.0,1.0), LinearFunction(1.0,0.0))
201 east = CoonsPatch(c12, c56, c15, c26,"east")
202
203
204 pvolume = ParametricVolume(north, east, south, west, top, bottom, "Pressure_side_volume")
205
206
207 #blk1 = Block3D(label="first-block", nni=20, nnj=20, nnk=20,
208 #             parametric_volume=pvolume,
209 #             fill_condition=initialCond)
210 #blk1.set_BC("WEST", "SUP_IN", inflow_condition=inflowCond)
211 #blk1.set_BC("SOUTH", "SUP_OUT")
212 #blk1.set_BC("TOP", "SUP_OUT")
213 #blk1.set_BC("BOTTOM", "SUP_OUT")
214
215
216 ##### Volume number 6 according to my intermediate nomenclature
217 ##### Extrude for inflow INPUT_CURVES!!!! WARNING 2 solutions
218 ##### for x and y cause of sqrt ... :/
219 print("Volume6#####")
220
221 def extrude(point, extrude_length):
222     from math import sqrt
223
224     x = point.x
225     y = point.y
226     z = point.z
227
228     r2 = sqrt(x**2+y**2)+extrude_length
229     a = x / y
230
231     y = sqrt(r2**2/(1+a**2))
232     x = y*a
233
234     return Vector(x,y,z)
235
236 v6p2 = p3
237 v6p3 = extrude(p3, inflow_space*input_curves[0].length())
238 v6p6 = p7
239 v6p7 = Vector(v6p3.x, v6p3.y, v6p6.z)
240
241 v6p0 = extrude(p0, inflow_space*input_curves[0].length())
242 v6p1 = p0
243 v6p5 = p4
244 v6p4 = Vector(v6p0.x, v6p0.y, v6p5.z)
245
246 #print "v6p4=", v6p4, "v6p5=", v6p5, "v6p6=", v6p6, "v6p7=", v6p7
247 #print "v6p0=", v6p0, "v6p1=", v6p1, "v6p2=", v6p2, "v6p3=", v6p3
248
249 print(" v6top-----")
250 v6c56 = c47
251 v6c45 = Line(v6p4, v6p5)
252 v6c76 = Line(v6p7, v6p6)
253 v6c47 = Arc(v6p4, v6p7, Vector(0,0,v6p4.z))
254 v6top = CoonsPatch(v6c45, v6c76, v6c47, v6c56, "v6top")

```

```

255 #
256
257
258 print(" v6bottom-----")
259 v6c03 = Arc(v6p0, v6p3, Vector(0,0,v6p0.z))
260 v6c12 = c03
261 v6c01 = Line(v6p0,v6p1)
262 v6c32 = Line(v6p3,v6p2)
263 v6bottom = CoonsPatch(v6c01, v6c32, v6c03, v6c12, "v6bottom")
264 #
265 #
266 print(" v6west-----")
267 v6c37 = Line(v6p3,v6p7)
268 v6c04 = Line(v6p0,v6p4)
269 v6west = CoonsPatch(v6c03, v6c47, v6c04, v6c37,"v6west")
270 #
271 #
272 print(" v6east-----")
273 v6c26 = c37
274 v6c15 = c04
275 #v6east = CoonsPatch(v6c12, v6c56, v6c15, v6c26,"v6east")
276 v6east = west
277
278 print(" v6north-----")
279 v6north = CoonsPatch(v6c32, v6c76, v6c37, v6c26, "v6north")
280
281 print(" v6south-----")
282 v6south = CoonsPatch(v6c01, v6c45, v6c04, v6c15, "v6south")
283
284
285
286
287 v6pvolume = ParametricVolume(v6north, v6east, v6south, v6west, v6top, v6bottom, "
      Inflow_Pressure_side_block")
288
289 #blk6 = Block3D(label="block6", nni=20, nnj=20, nnk=20,
290 #             parametric_volume=v6pvolume,
291 #             fill_condition=initialCond)
292
293
294 ##### Block number 4 according to my intermediate nomenclature ,
      suction side block
295 ##### INPUT_CURVES!!!!
296 print(" Volume4#####")
297 input_curves = input_interpreter(file_camber_hub, file_camber_hub_to_shroud, file_camber_shroud ,
      number_of_intermediate_camber_files)
298
299 def LoftRotatorClockwise(input_curves):
300     input_curvesss = []
301     for i in range(0,len(input_curves)):
302         input_curvesss.append(input_curves[i].clone())
303     def LoftUserDefined(r,s):
304
305         Q=None
306         Q=[]
307
308         r_Curves=input_curvesss
309
310         for n in range(0,len(r_Curves)):
311             Q.append(r_Curves[n].eval(r))
312
313         s_Curve=Spline(Q)
314         Coords = s_Curve.eval(s)
315
316         return (Coords.x,Coords.y,Coords.z)
317
318     def LoftRotateClockwise(r,s):
319         from math import cos, sin, pi
320
321         theta = -2 * pi / number_of_blades *1/2
322         cc = cos(theta)
323         ss = sin(theta)
324
325         Coords = LoftUserDefined(r,s)
326         x = Coords[0]

```

```

327     y = Coords[1]
328     z = Coords[2]
329     x_r = x*cc-y*ss
330     y_r=x*ss+y*cc
331
332     return (x_r, y_r, z)
333
334     return LoftRotateClockwise
335
336 v4north = PyFunctionSurface(LoftRotatorClockwise(input_curves))
337 v4south = PyFunctionSurface(Loft(input_curves))
338
339 #v4north = PyFunctionSurface(LoftRotateClockwise)
340 #v4south = PyFunctionSurface(LoftUserDefined)
341
342 v4p2 = v4north.eval(1.0, 0.0)
343 v4p3 = v4north.eval(0.0, 0.0)
344 v4p6 = v4north.eval(1.0, 1.0)
345 v4p7 = v4north.eval(0.0, 1.0)
346
347 v4p0 = v4south.eval(0.0, 0.0)
348 v4p1 = v4south.eval(1.0, 0.0)
349 v4p4 = v4south.eval(0.0, 1.0)
350 v4p5 = v4south.eval(1.0, 1.0)
351
352 # print "p0=", p0, "p1=", p1
353
354
355
356 print(" v4top-----")
357 v4c56 = Arc(v4p5, v4p6, Vector(0,0,v4p5.z))
358 v4c47 = Arc(v4p4, v4p7, Vector(0,0,v4p4.z))
359 v4c76 = PathOnSurface(v4north, LinearFunction(1.0,0.0), LinearFunction(0.0,1.0))
360 v4c45 = PathOnSurface(v4south, LinearFunction(1.0,0.0), LinearFunction(0.0,1.0))
361 v4top = CoonsPatch(v4c45, v4c76, v4c47, v4c56, "v4top")
362
363
364 print(" v4bottom-----")
365 v4c03 = Arc(v4p0, v4p3, Vector(0,0,v4p0.z))
366 v4c12 = Arc(v4p1, v4p2, Vector(0,0,v4p1.z))
367 v4c01 = PathOnSurface(v4south, LinearFunction(1.0,0.0), LinearFunction(0.0,0.0))
368 v4c32 = PathOnSurface(v4north, LinearFunction(1.0,0.0), LinearFunction(0.0,0.0))
369 v4bottom = CoonsPatch(v4c01, v4c32, v4c03, v4c12, "v4bottom")
370
371
372 print(" v4west-----")
373 v4c37 = PathOnSurface(v4north, LinearFunction(0.0,0.0), LinearFunction(1.0,0.0))
374 v4c04 = PathOnSurface(v4south, LinearFunction(0.0,0.0), LinearFunction(1.0,0.0))
375 v4west = CoonsPatch(v4c03, v4c47, v4c04, v4c37, "v4west")
376
377
378 print(" v4east-----")
379 v4c26 = PathOnSurface(v4north, LinearFunction(0.0,1.0), LinearFunction(1.0,0.0))
380 v4c15 = PathOnSurface(v4south, LinearFunction(0.0,1.0), LinearFunction(1.0,0.0))
381 v4east = CoonsPatch(v4c12, v4c56, v4c15, v4c26, "east")
382
383
384 v4pvolume = ParametricVolume(v4north, v4east, v4south, v4west, v4top, v4bottom, "
    Pressure_side_block")
385
386
387 #blk4 = Block3D(label="fourth-block", nni=20, nnj=20, nnk=20,
388 #               parametric_volume=v4pvolume,
389 #               fill_condition=initialCond)
390
391
392 #####Volume 5
393 print(" Volume5#####")
394 v5p2 = v4p3
395 v5p6 = v4p7
396 v5p3 = extrude(v5p2, v6c32.length())
397 v5p7 = Vector(v5p3.x, v5p3.y, v5p6.z)
398
399 v5p0 = v6p3
400 v5p1 = v6p2

```



```

401 v5p4 = v6p7
402 v5p5 = v6p6
403
404
405
406
407 print(" v5top-----")
408 v5c56 = v4c47
409 v5c45 = v6c76
410 v5c76 = Line(v5p7,v5p6)
411 v5c47 = Arc(v5p4,v5p7,Vector(0,0,v5p4.z))
412 v5top = CoonsPatch(v5c45, v5c76, v5c47, v5c56, "v5top")
413 #
414
415
416 print(" v5bottom-----")
417 v5c03 = Arc(v5p0, v5p3,Vector(0,0,v5p0.z))
418 v5c12 = v4c03
419 v5c01 = v6c32
420 v5c32 = Line(v5p3,v5p2)
421 v5bottom = CoonsPatch(v5c01, v5c32, v5c03, v5c12, "v5bottom")
422 #
423 #
424 print(" v5west-----")
425 v5c37 = Line(v5p3,v5p7)
426 v5c04 = v6c37
427 v5west = CoonsPatch(v5c03, v5c47, v5c04, v5c37,"v6west")
428 #
429 #
430 print(" v5east-----")
431 v5c26 = v4c37
432 v5c15 = v4c04
433 #v5east = CoonsPatch(v5c12, v5c56, v5c15, v5c26,"v5east")
434 v5east = v4west
435
436 print(" v5north-----")
437 v5north = CoonsPatch(v5c32, v5c76, v5c37, v5c26, "v5north")
438
439 print(" v5south-----")
440 #v5south = CoonsPatch(v5c01, v5c45, v5c04, v5c15, "v5south")
441 v5south = v6north
442
443 v5pvolume = ParametricVolume(v5north, v5east, v5south, v5west, v5top, v5bottom, "
    suction_side_inlet_block")
444 #blk5 = Block3D(label="fourth-block", nni=20, nnj=20, nnk=20,
445 #             parametric_volume=v5pvolume,
446 #             fill_condition=initialCond)
447
448
449 #print "v5c32", v5c32.eval(0), v5c32.eval(1), "v5c76", v5c76.eval(0), v5c76.eval(1), "v5c37",
    v5c37.eval(0), v5c37.eval(1),"v5c26", v5c26.eval(0), v5c26.eval(1)
450 #print "pvolume", pvolume.eval(0,0,0), "pvolume",pvolume.eval(1,0,1)
451 #print "c04", c04.eval(0), c04.eval(1)
452
453 #####Volume 2 is the outflow pressure side block INPUT CURVES!
454 print("Volume2#####")
455
456 #####To create the periodic boundary without edges, the camberline/surface is evaluated
    at its end (outflow)
457 #####the curve will be mirrored/when the angles are concerned but the radius remains
    constant with increasing z
458 ### Refinement options:
459 z_resolution = 0.01
460 points_in_spline = 8
461
462 def get_curved_outflow_points(surface,a,b):
463     from numpy import sqrt, arctan2
464     from math import cos, sin
465     ## Find the length in z-direction
466     z_length = outflow_space*input_curves[0].length()
467     x_start = surface.eval(a,b).x
468     y_start = surface.eval(a,b).y
469     z_start = surface.eval(a,b).z
470     r_start = sqrt(x_start**2+y_start**2)
471     theta_start = arctan2(y_start,x_start)

```

```

472
473 ## Find the corresponding meridional percentage
474 length_along_curve = 1
475 while length_along_curve > 0:
476     if surface.eval(length_along_curve,b).z < z_start-z_length:
477         while length_along_curve < 1:
478             if surface.eval(length_along_curve,b).z > z_start-z_length:
479                 break
480             length_along_curve += z_resolution
481         break
482     length_along_curve -= 0.1
483
484 points_for_spline = []
485 points_for_spline.append(Vector(x_start, y_start, z_start))
486
487 for i in range(1, points_in_spline+1):
488     step = (1 - length_along_curve)/points_in_spline
489     x = surface.eval(a-i*step,b).x
490     y = surface.eval(a-i*step,b).y
491     z = surface.eval(a-i*step,b).z
492
493     theta = arctan2(y,x)
494     theta_before = arctan2(points_for_spline[-1].y, points_for_spline[-1].x)
495     theta_new = theta_before + (theta_start - theta)*(1-i/float(points_in_spline))*1.5
496
497     x_new = cos(theta_new) * r_start
498     y_new = sin(theta_new) * r_start
499     z_new = z_start + (z_start-z)
500
501     points_for_spline.append(Vector(x_new, y_new, z_new))
502 return points_for_spline
503
504 points_for_spline_v2c45 = get_curved_outflow_points(top,1,0)
505 v2c45 = Spline(points_for_spline_v2c45)
506
507 points_for_spline_v2c01 = get_curved_outflow_points(bottom,1,0)
508 v2c01 = Spline(points_for_spline_v2c01)
509
510 points_for_spline_v2c76 = get_curved_outflow_points(top,1,1)
511 v2c76 = Spline(points_for_spline_v2c76)
512
513 points_for_spline_v2c32 = get_curved_outflow_points(bottom,1,1)
514 v2c32 = Spline(points_for_spline_v2c32)
515
516
517 v2p0 = p1
518 v2p3 = p2
519 v2p4 = p5
520 v2p7 = p6
521
522 v2p1 = v2c01.eval(1)
523 v2p2 = v2c32.eval(1)
524 v2p5 = v2c45.eval(1)
525 v2p6 = v2c76.eval(1)
526
527 #print "p0=", v2p0, "p1=", v2p1, "p2=", v2p2, "p3=", v2p3, "p4=", v2p4, "p5=", v2p5, "p6=",
    v2p6, "p7=", v2p7
528
529 print(" v2top-----")
530 v2c56 = Arc(v2p5, v2p6, Vector(0,0,v2p5.z))
531 #v2c45 = Line(v2p4, v2p5)
532 #v2c76 = Line(v2p7, v2p6)
533 v2c47 = c56
534 v2top = CoonsPatch(v2c45, v2c76, v2c47, v2c56, "v2top")
535 #
536
537
538 print(" v2bottom-----")
539 v2c03 = c12
540 v2c12 = Arc(v2p1, v2p2, Vector(0,0,v2p1.z))
541 #v2c32 = Line(v2p3, v2p2)
542 v2bottom = CoonsPatch(v2c01, v2c32, v2c03, v2c12, "v2bottom")
543 #
544 #
545 print(" v2west-----")

```

```

546 v2c37 = c26
547 v2c04 = c15
548 v2west = east
549 #
550 #
551 print(" v2 east -----")
552 v2c26 = Line(v2p2,v2p6)
553 v2c15 = Line(v2p1,v2p5)
554 v2east = CoonsPatch(v2c12, v2c56, v2c15, v2c26," v2east")
555
556
557 print(" v2 north -----")
558 v2north = CoonsPatch(v2c32, v2c76, v2c37, v2c26, " v2north")
559
560 print(" v2 south -----")
561 v2south = CoonsPatch(v2c01, v2c45, v2c04, v2c15, " v2south")
562
563 v2pvolume = ParametricVolume(v2north, v2east, v2south, v2west, v2top, v2bottom, "
    pressure_side_outlet_block")
564 #
565 #print "south", v2south.eval(1,0), " east", v2east.eval(0,0), " bottom", v2bottom.eval(1,0)
566 #
567 #print "volume p 100", v2pvolume.eval(1)
568
569 #blk2 = Block3D(label="fourth-block", nni=20, nnj=20, nnk=20,
570 #             parametric_volume=v2pvolume,
571 #             fill_condition=initialCond)
572
573 #####Volume 3 is the outflow suction side block
574 print(" Volume3#####")
575
576 points_for_spline_v3c76 = get_curved_outflow_points(v4top,1,1)
577 v3c76 = Spline(points_for_spline_v3c76)
578
579 points_for_spline_v3c32 = get_curved_outflow_points(v4bottom,1,1)
580 v3c32 = Spline(points_for_spline_v3c32)
581
582 points_for_spline_v3c45 = get_curved_outflow_points(v4top,1,0)
583 v3c45 = Spline(points_for_spline_v3c45)
584
585 points_for_spline_v3c01 = get_curved_outflow_points(v4bottom,1,0)
586 v3c01 = Spline(points_for_spline_v3c01)
587
588 v3p0 = v4p1
589 v3p3 = v4p2
590 v3p4 = v4p5
591 v3p7 = v4p6
592
593
594 v3p2 = v3c32.eval(1)
595 v3p1 = v3c01.eval(1)
596 v3p6 = v3c76.eval(1)
597 v3p5 = v3c45.eval(1)
598
599 #from math import sqrt
600 #print sqrt(v3c32.eval(1).x**2+v3c32.eval(1).y**2), sqrt(v3p1.x**2+v3p1.y**2)
601
602
603 print(" v3 top -----")
604 v3c56 = Arc(v3p5, v3p6, Vector(0,0,v3p5.z))
605 #v3c45 = Line(v3p4, v3p5)
606 v3c47 = v4c56
607 v3top = CoonsPatch(v3c45, v3c76, v3c47, v3c56, " v3top")
608 #
609
610
611 print(" v3 bottom -----")
612 v3c03 = v4c12
613 v3c12 = Arc(v3p1, v3p2, Vector(0,0,v3p1.z))
614 #v3c01 = Line(v3p0, v3p1)
615 v3bottom = CoonsPatch(v3c01, v3c32, v3c03, v3c12, " v3bottom")
616 #
617 #
618 print(" v3 west -----")
619 v3c37 = v4c26

```

```

620 v3c04 = v4c15
621 #v3west = CoonsPatch(v3c03, v3c47, v3c04, v3c37,"v3west")
622 v3west = v4east
623 #
624 #
625 print("v3east-----")
626 v3c26 = Line(v3p2,v3p6)
627 v3c15 = Line(v3p1,v3p5)
628 v3east = CoonsPatch(v3c12, v3c56, v3c15, v3c26,"v3east")
629
630
631 print("v3north-----")
632 v3north = CoonsPatch(v3c32, v3c76, v3c37, v3c26, "v3north")
633
634 print("v3south-----")
635 #v3south = CoonsPatch(v3c01, v3c45, v3c04, v3c15, "v3south")
636 v3south = v2north
637
638 v3pvolume = ParametricVolume(v3north, v3east, v3south, v3west, v3top, v3bottom, "
        pressure_side_outlet_block")
639
640
641 #blk3 = Block3D(label="fourth-block", nni=20, nnj=20, nnk=20,
642 #             parametric_volume=v3pvolume,
643 #             fill_condition=initialCond)
644
645
646
647 #####-----#####
648 #
649 #
650 #
651 ##### Now that the control volume is created, the blocks will be made. We cannot use every
        surface of volume 1 to 6 since an OH grid mesh will be
652 ##### made but having a control volume by itself might come in handy sometimes. (with wich
        I am trying to say it was not useless to create all the 6 volumes)
653 ##### The control volume will be sliced up in surfaces in hub to shroud direction and the 3
        d block will be an extrusion of the 2d patches created on that surface.
654 ##### These surfaces is split by an O-gridline and after that into surfaces. Maybe I should
        add a figure in the folder to make it easier to understand.
655
656 ##### bent / revolve ... slice, z-r plane
657 from numpy import sqrt, arctan2, pi, arccos
658 from math import cos, sin
659 number_of_points_for_ogl = 120
660 distance_ogl_blade = float(0.001)
661 arc_resolution = 10**-3 #this should not be at 10^-2 thats not good enough!
662
663 number_of_points_on_connection = 10
664
665 def get_radius(point): #The distance between the z-axis and a point has to be determined
        a couple of times, which is normally the radius (at least one exception)
666     from math import sqrt
667     radius = sqrt(point.x**2+point.y**2)
668     return radius
669
670
671 input_curves = input_interpreter(file_camber_hub ,file_camber_hub_to_shroud ,file_camber_shroud ,
        number_of_intermediate_camber_files)
672
673
674 def ogl_creator(input_curves ,camber_curves ,pressure_or_suction): #
        pressure_or_suction = 1 pressure side,           pressure_or_suction = -1 suction side
675
676     oglines = []
677
678     def reposition_into_plane(point_to_be_repositioned ,pl,i): #bent and rotate about pl #i
        is the position along the blade

```

```

679     u_ogl_point=point_to_be_repositioned
680     theta = arctan2( p1.y, p1.x )
681     r = get_radius(p1)
682     z = p1.z
683     u_theta = arctan2(u_ogl_point.y,u_ogl_point.x)
684     u_r = get_radius(u_ogl_point)
685     ur_projectedinto_z_r_plane = cos(u_theta-theta) * u_r
686
687     bent_arc_length = sqrt( (ur_projectedinto_z_r_plane -r)**2 + (u_ogl_point.z-z)**2 )
688     #print "bent_arc_length",bent_arc_length
689     t_arc_length = 0
690     l=0
691     tp=[Vector(0,0,0),Vector(0,0,0)]
692     if abs(u_ogl_point.z-z) == 0 : #either the point has to be in the inlet area or, it
will also have the same radius (distance to z) in its final form
693         x1 = r*cos(theta)+( bent_arc_length) * cos (theta) #in case of same r , bent
arc length is zero
694         y1 = r*sin(theta)+ ( bent_arc_length) * sin (theta)
695         z1 = z
696         tp[1] = Vector( x1,y1,z1 ) #tp[1]? cause of the proگرامing style atm
697         tp[0]=tp[1].clone()
698     else :
699         d = (u_ogl_point.z-z) / abs(u_ogl_point.z-z) # this will be +1 or -1 and thus
determines in which direction to go on the z-r curve
700         tp = [p1 , Vector(0,0,0)]
701         while t_arc_length < bent_arc_length:
702             if (i * 1/float(number_of_points_for_ogl)+d*1* arc_resolution ) >= 0: #
determines wether we are in inlet outlet or in between
703                 if (i * 1/float(number_of_points_for_ogl)+d*1* arc_resolution ) <= 1: #
we are in between inlet and outlet
704                     tp[1] = input_curves[n].eval( i * 1/float(number_of_points_for_ogl)+d*
1* arc_resolution )
705                     t_r = get_radius(tp[1])
706                     t_z = tp[1].z
707                     t_dr = t_r-get_radius(tp[0])
708                     t_dz = t_z - tp[0].z
709                     t_arc_length = t_arc_length + sqrt ( t_dr**2+t_dz**2)
710                     #tp[0]=tp[1].clone()
711                     tp[0]=Vector(cos(theta)*t_r,sin(theta)*t_r,t_z).clone()
712                     l+=1
713
714                 elif (i * 1/float(number_of_points_for_ogl)+d*1* arc_resolution) > 1: #
outlet
715                     tp[1] = Vector(camber_curves[n].eval(1).x, camber_curves[n].eval(1).y,
tp[0].z + bent_arc_length-t_arc_length)
716                     t_r = get_radius(tp[1])
717                     t_z = tp[1].z
718                     t_arc_length = bent_arc_length
719                     #tp[0]=tp[1].clone()
720                     tp[0]=Vector(cos(theta)*t_r,sin(theta)*t_r,t_z).clone()
721                     #print "larger than one detected"
722                 elif (i * 1/float(number_of_points_for_ogl)+d*1* arc_resolution ) < 0: #inlet
723                     tp_theta = arctan2(tp[0].y,tp[0].x)
724                     r_max = get_radius(camber_curves[n].eval(0))
725                     x1 = r_max*cos(tp_theta)+ ( bent_arc_length-t_arc_length) * cos (tp_theta)
726                     y1 = r_max*sin(tp_theta)+ ( bent_arc_length-t_arc_length) * sin (tp_theta)
727                     z1 = camber_curves[n].eval(0).z
728                     tp[1] = Vector( x1,y1,z1 )
729                     t_arc_length = bent_arc_length
730                     tp[0]=tp[1].clone()
731                     #print "detected, point =%s, loop iterations=%s" %(i,1), "tp_r=",
get_radius(tp[0])-r_max
732                     #print "bent_arc_length=",bent_arc_length
733                     b_ogl_point = tp[0] #now that we got the bent point, we have to revolve it
around the z-axis
734                     tp_theta = arctan2(tp[0].y,tp[0].x)
735                     distance_u_ogl_point_to_zr_plane = sin(u_theta-theta) * u_r
736                     revolving_angle = distance_u_ogl_point_to_zr_plane / r # d / (2Pi*r) * 2Pi = angle
of an arc of length d with radius r
737                     ogl_point = b_ogl_point.rotate_about_zaxis(revolving_angle).clone()
738
739     return ogl_point
740
741
742 for n in range (0,len(input_curves)):

```

```

743     r_max = get_radius(input_curves[n].eval(0))
744     z_max = input_curves[n].eval(1).z
745     ogl_points = None
746     ogl_points = []
747     for i in range(0, number_of_points_for_ogl):           #first get the 0 g
-line curve and afterwards we connect the curves for pressure and suction
748         p1 = input_curves[n].eval( i * 1/float(number_of_points_for_ogl) ) #determine 2
points on the blade to get a tangent vector to the blade p2-p1 = tangent vector
749         p2 = input_curves[n].eval( (i+1) * 1/float(number_of_points_for_ogl) )
750
751         theta = arctan2( p1.y, p1.x )                       #determine the
azimuthal angle (cylindrical coordinate system)
752         r = get_radius (p1)                                #determine the
radius (cylindrical coordinate system)
753         z = p1.z                                           #determine the
radius (cylindrical coordinate system)
754
755         dr = get_radius (p2)- r                             #dr and dz
together give a vector tangential to the slice at point p1
756         dz = p2.z - z
757
758         normal_to_slice = cross( Vector(-sin(theta),cos(theta),0), Vector(cos(theta)*dr,
sin(theta)*dr,dz) ) #normal to slice pointing towards hub at point p1
759
760         if pressure_or_suction > 0 :                       #pressure side
761             ogl_point_direction = cross (p2-p1,normal_to_slice) #this vector is
orthogonal to both the normal to the slice as well as the tangential to the blade
762         elif pressure_or_suction < 0:                     #suction side
763             ogl_point_direction = cross (normal_to_slice ,p2-p1)
764
765         u_ogl_point = p1 + ogl_point_direction.norm()*distance_ogl_blade #
u_ogl_point = unbent and "un" rotated its in the plane which is tangential to the slice in
p1
766                                                                 # to
put the point into the slice surface we have to bent and rotate our tangential plane
767         ## a little intermezzo to explain: u_ogl_point is now on a plane which is
tangential to the slice. But this means it is not on the surface we want it to be.
768         ## Thinking in extremes might help, so imagine the distance between the blade and
O-gridline is made very large for example a 100 times the distance between two blades.
769         ## The point wouldn't even be close to the turbine any more. In the following we
force the point to be on the slice surface by bending and rotating the tangential plane.
770         ## This is done in the following way: 3 coordinates define our slice: r,theta and
z. All of these can be gathered from the input curves.
771         ## all throughout the file we will refer to rotating as the rotation around the z-
axis. The other deformation will be refered to as bending
772         ## (imagine the plane with constant z (inflow plane) being bent into a plane with
constant r (outflow plane), the corner would be the palce where the compressor is)
773         ## The bending of the plane will give the point the correct r and z coordinates to
be in the slice. The rotation afterwards will put it at the correct angle.
774         ## Both "motions" are done by taking the appropriate distance between p1 and the
u_ogl_point and moving along the surface: sqrt(dr^2 + dz^2) gives the distance for the
bending motion
775
776
777         #ogl_points.append(ogl_point)
778         ogl_points.append(reposition_into_plane(u_ogl_point ,p1,i))
779
780         #print "line =", n, " point=", i
781         #now create the connections with the camber line:
782         #we start with the inlet and we are workingwith the first point of the camber line:
783
784         wp = camber_curves[n].eval(0) #working point
785         wp_r = get_radius(wp)
786         wp_theta = arctan2(wp.y,wp.x)
787         wp_z=wp.z
788
789
790         # a lot of work to put the first ogl point into the right plane again : (which is the
work we done before backwards and for a different plane)
791         foglp = ogl_points[0]
792         foglp_r = get_radius(foglp)
793         foglp_theta = arctan2(foglp.y,foglp.x)
794         foglp_z= foglp.z
795

```

```

796     rotation_arcl = (foglp_theta-wp_theta)*foglp_r           #full circle theta = 2 pi and so
the arc would be 2*Pi*r
797     #print "rotation arcl = ", rotation_arcl ," dist =", vabs(wp-foglp)
798
799     dp=wp-camber_curves[n].eval(1/float(number_of_points_for_og1))
800
801     ogl_starting_point = wp+Vector(dp.x,dp.y,0).norm()*distance_ogl_blade
802
803     if foglp_r>=wp_r:           #if this is the case the points are already in the same plane.
804         Arc_connection = Arc(foglp , ogl_starting_point , wp)
805         for ii in range (0,11):
806             ogl_points.insert(0, Arc_connection.eval(ii*0.1))
807         #print"-----!-!-!-!-!-!-!-!-!-!-", n
808     else:
809         a = 0
810         b=0
811         walker = [wp, Vector(0,0,0)]
812         dw = 0
813         while a >= 0:
814             walker [1] = camber_curves[n].eval(b*arc_resolution)
815             dw += vabs(walker[0]-walker[1])
816             a=get_radius(walker[1])-foglp_r
817             b +=1
818             walker [0] = walker [1].clone()
819
820             foglp_in_the_right_plane = wp-Vector(dp.x,dp.y,0).norm()*dw+Vector(-dp.y,dp.x,0).
norm()*rotation_arcl
821             #print"dw=",dw, " rotation_arcl", rotation_arcl ," sqrt=" ,sqrt(dw**2+rotation_arcl
**2)
822             #print "distance new point-wp=",vabs(foglp_in_the_right_plane-wp),"old distance =
", vabs(wp-foglp)
823             x_vector = ogl_starting_point-wp
824             scewed_y_vec = foglp_in_the_right_plane-wp
825
826
827             for ii in range (0,11):
828                 point_on_connecting_arc = sin(pi/2/10*ii)*x_vector+cos(pi/2/10*ii)*
scewed_y_vec+wp
829                 repositioned = reposition_into_plane(point_on_connecting_arc, wp, 0)
830                 ogl_points.insert(0, repositioned)
831                 #print "r=", get_radius(repositioned) , "           r=", get_radius(
foglp)
832
833             #now we have to connect them in the outflow
834
835             wp2 = camber_curves[n].eval(1) #working point at the end
836             wp2_r = get_radius(wp2)
837             wp2_theta = arctan2(wp2.y,wp2.x)
838             wp2_z=wp2.z
839
840
841             # a lot of work to put the first ogl point into the right plane again : (which is the
same thing we did before backwards and for a different plane)
842             loglp = ogl_points[-1]
843             loglp_r = get_radius(loglp)
844             loglp_theta = arctan2(loglp.y,loglp.x)
845             loglp_z= loglp.z
846
847             #full circle theta = 2 pi and so the arc would be 2*Pi*r
848             #print "rotation arcl = ", rotation_arcl ," dist =", vabs(wp-foglp)
849             dp2=wp2-camber_curves[n].eval(1-1/float(number_of_points_for_og1))
850
851
852             arc_arm1 = dp2.norm()*distance_ogl_blade
853             arc_arm2 = cross(dp2, Vector(cos(wp2_theta), sin(wp2_theta), 0)*pressure_or_suction).norm
()*distance_ogl_blade
854             #arc_arm2 = (loglp_in_the_working_plane-wp2).norm()*distance_ogl_blade
855
856             starting_angle = arccos(dot(og1_points[-1]-wp2, dp2)/vabs(og1_points[-1]-wp2)/vabs(dp2)
)
857             #print "starting_angle", starting_angle/pi*180 , "first_angle" , (pi/2-starting_angle)
*180/pi
858             l = 10
859             for iii in range (4,l+1):
860                 #point_on_connecting_arc2 = sin(pi/2/l*iii)*arc_arm1+cos(pi/2/l*iii)*arc_arm2+wp2

```

```

861     point_on_connecting_arc2 = sin(pi/2-starting_angle+starting_angle/l*iii)*arc_arm1+
      cos(pi/2-starting_angle+starting_angle/l*iii)*arc_arm2+wp2
862     repositioned2 = reposition_into_plane(point_on_connecting_arc2,wp2,float(
      number_of_points_for_ogl)) #lekker handig
863     ogl_points.append(repositioned2)
864     #ogl_points.append(point_on_connecting_arc2)
865
866
867
868
869     #make a spline of all those points:
870     oglines.append(Spline(ogl_points))
871     print "%g out of %g" %(n+1,len(input_curves))
872
873
874     return oglines
875
876     ###
877     ##print "-----"
878
879     #
880     input_curves_pressure = input_interpreter(file_camber_hub,file_camber_hub_to_shroud,
      file_camber_shroud,number_of_intermediate_camber_files)
881     input_curves_suction = input_interpreter(file_camber_hub,file_camber_hub_to_shroud,
      file_camber_shroud,number_of_intermediate_camber_files)
882     input_curves_camber = input_interpreter(file_camber_hub,file_camber_hub_to_shroud,
      file_camber_shroud,number_of_intermediate_camber_files)
883
884
885     O_grid_line_surface_pressure = PyFunctionSurface(Loft(ogl_creator(input_curves_pressure,
      input_curves_camber,1)))
886     O_grid_line_surface_suction = PyFunctionSurface(Loft(ogl_creator(input_curves_suction,
      input_curves_camber,-1)))
887     #
888     ### all the main surfaces are there, now we need to connect them in the right places to create
      blocks
889     ### first functions are created to make the different kind of blocks.. blocks in the o grid
      part, blocks in the H - block part
890     ### and blocks in the inflow and outflow part... all automated depending on the number of
      nodes
891     #
892     #
      #####
893     #
      #####
894
895     #
896     #
897     def accurate_length(curve):
898     ### A more accurate way to obtain length
899     L = 0.0
900     n = 100
901     dt = 1.0 / n
902     p0 = curve.eval(0.0)
903     p1 = Vector(0,0,0)
904     for i in range(1,n+1):
905         p1 = curve.eval(dt * i)
906         L += vabs(p1 - p0)
907         p0 = p1;
908     return L
909
910
911     def simple_connector(p1,p2,locationp1,curvep1, camber_curves, top_or_bottom): #top_or_bottom 0
      = bot or -1 = top
912     ### This function is used to connect nodes
913
914     r1 = get_radius(p1)
915     z1 = p1.z
916     th1 = arctan2(p1.y,p1.x)
917     r2 = get_radius(p2)
918     z2 = p2.z
919     th2 = arctan2(p2.y,p2.x)
920

```



```

921 LE = camber_curves[top_or_bottom].eval(0)
922 TE = camber_curves[top_or_bottom].eval(1)
923 r_max = get_radius(LE)
924 z_max = TE.z
925
926 def walking_along_the_curve(p1,p2,s): #S = WALKING DIRECTION
927     z1 = p1.z
928     z2 = p2.z
929     r1 = get_radius(p1)
930     r2 = get_radius(p2)
931     th1 = arctan2(p1.y,p1.x)
932     th2 = arctan2(p2.y,p2.x)
933     dw = 0
934     dzw = 0
935     walker = [p1,Vector(0,0,0)]
936     points_on_connection = []
937     n = number_of_points_on_connection + 1
938     a = s
939     distance = sqrt((r2-r1)**2+(z2-z1)**2)
940     for i in range(1,n):
941         while dw < distance/n*i:
942             #while abs(dzw) < abs((z2-z1)/n*i):
943                 walker[1] = curvepl.eval(locationpl+a*arc_resolution)
944                 #dzw += (walker[1].z-walker[0].z)**2 + (get_radius(walker[1])-get_radius(
walker[0]))**2
945                 dzw = walker[1].z-walker[0].z
946                 drw = get_radius(walker[1])-get_radius(walker[0])
947                 dw += sqrt(dzw**2 + drw**2)
948                 a += s
949                 #print "locationpl+a*arc_resolution", locationpl+a*arc_resolution , "dzleft" ,
abs(distance)-abs(dw) , "zwalker" , walker[1].z
950                 #print "z2" , z2
951                 if locationpl+a*arc_resolution < 0:
952                     #print "entered this part when i is" , i , "and a" , a ,
953                     dzw = (z2-z1)*s
954                     dw = distance
955                     walker[1] = curvepl.eval(0)
956                 elif locationpl+a*arc_resolution > 1:
957                     #print "entered this other part when i is" , i , "and a" , a ,
958                     dw= distance
959                     dzw = (z2-z1)*s
960                     walker[1] = curvepl.eval(1)
961                     rr = get_radius(walker[1])
962                     zz = walker[1].z
963                     walker[0] = walker[1].clone()
964                     thth= th1+(th2-th1)/n*i
965                     points_on_connection.append(Vector(rr*cos(thth),rr*sin(thth),zz))
966                 return points_on_connection
967
968 def walking_along_the_camber(p1,p2,s,start,top_or_bottom): #start 0 or 1,
top_or_bottom 0 = bot or -1 = top
969     z1 = p1.z
970     z2 = p2.z
971     r1 = get_radius(p1)
972     r2 = get_radius(p2)
973     th1 = arctan2(p1.y,p1.x)
974     th2 = arctan2(p2.y,p2.x)
975     dw = 0
976     dzw = 0
977     walker = [p1,Vector(0,0,0)]
978     points_on_connection = []
979     n = number_of_points_on_connection + 1
980     a = s
981     distance = sqrt((r2-r1)**2+(z2-z1)**2)
982     for i in range(1,n):
983         #print "dw", dw , "distance/n*i", distance/n*i, "distance", distance, "n", n ,"i",
i
984         while dw < distance/n*i:
985             #while abs(dzw) < abs((z2-z1)/n*i):
986                 walker[1] = camber_curves[top_or_bottom].eval(start+a*arc_resolution)
987                 #dzw += (walker[1].z-walker[0].z)**2 + (get_radius(walker[1])-get_radius(
walker[0]))**2
988                 dzw = walker[1].z-walker[0].z
989                 drw = get_radius(walker[1])-get_radius(walker[0])
990                 dw += sqrt(dzw**2 + drw**2)

```

```

991         a += s
992         #print "locationp1+a*arc_resolution", locationp1+a*arc_resolution ," dzleft" ,
abs(distance)-abs(dw) , "zwalker" , walker[1].z
993         #print "z2" , z2
994         if start+a*arc_resolution < 0:
995             #print "entered this part when i is" , i , "and a" , a ,
996             dzw = (z2-z1)*s
997             dw = distance
998             walker[1] = curvep1.eval(0)
999         elif start+a*arc_resolution > 1:
1000             #print "entered this other part when i is" , i , "and a" , a ,
1001             dw= distance
1002             dzw = (z2-z1)*s
1003             walker[1] = curvep1.eval(1)
1004             rr = get_radius(walker[1])
1005             zz = walker[1].z
1006             walker[0] = walker[1].clone()
1007             thth= th1+(th2-th1)/n*i
1008             points_on_connection.append(Vector(rr*cos(thth),rr*sin(thth),zz))
1009         return points_on_connection
1010
1011 #print "p1", p1 , "p2" ,p2
1012 #print "r1",r1 ,"r2" , r2 , "rmax", r_max , "z_max" , z_max ,"z1" ,z1 , "z2",z2
1013 if z1 <= z_max:
1014     if z2 <= z_max:
1015         if r1 <= r_max:
1016             if r2 <= r_max: #both are in bent area
1017                 #print "we made it here" , "r2" ,r2 , "r_max" , r_max
1018                 if r1 == r2:
1019                     connection = Arc(p1,p2,Vector(0,0,p1.z))
1020                 else:
1021                     if r1>r2 :
1022                         s=1
1023                     else :
1024                         s = -1
1025                     points = walking_along_the_curve(p1,p2,s)
1026                     points.insert(0,p1)
1027                     points.append(p2)
1028                     connection = Spline(points)
1029                 else:#now p2 is in inlet area p1 is not therefore r1<r2
1030                     s=-1
1031                     LE_angle= arctan2(LE.y,LE.x)
1032                     point_at_correct_angle = LE.clone().rotate_about_zaxis((th2-th1)*(r_max-r1
)/(r2-r1)+(th1-LE_angle))
1033                     #point_at_correct_angle = LE.clone().rotate_about_zaxis((th2-LE_angle))
1034                     point_in_between = Vector(point_at_correct_angle.x,point_at_correct_angle.
y,z2)
1035                     #point_in_between = rotate_point_around_z_axis(LE,(th2-LE_angle)*6)
1036                     points = walking_along_the_curve(p1,point_in_between,s)
1037                     #points = [point_in_between]
1038                     points.insert(0,p1)
1039                     points.append(point_in_between)
1040                     #print "factor " , (r_max-r1)/(r2-r1) , "th2" , th2*180/pi , th1*180/pi ,
LE_angle*180/pi
1041                     #print "point in betwe" , arctan2(point_in_between.y,point_in_between.x)
*180/pi
1042                     points.append(p2)
1043                     connection = Spline(points)
1044                 elif r2 < r_max:
1045                     s = 1
1046                     LE_angle= arctan2(LE.y,LE.x)
1047                     point_at_correct_angle = LE.clone().rotate_about_zaxis((th2-th1)*(r_max-r1)/(
r2-r1)+(th1-LE_angle))
1048                     point_in_between = Vector(point_at_correct_angle.x,point_at_correct_angle.y,z1
)
1049                     points = walking_along_the_camber(point_in_between ,p2,s,0,top_or_bottom)
1050                     points.insert(0,point_in_between)
1051                     points.insert(0,p1*1/3+point_in_between*2/3) #SPLINE BEHAVES WEIRD
WITHOUT THIS AT p1curve = PathOnSurface(O_grid.line_surface_pressure , LinearFunction
(1.0,0.0) , LinearFunction(0.0,1.0)) p1 =p1curve.eval(0.01) p2 = c45.eval(0.01)
1052                     points.insert(0,p1)
1053                     points.append(p2)
1054                     connection = Spline(points)
1055                 else:
1056                     connection = Spline([p1,p2])

```

```

1057     else:
1058         s=1
1059         TE_angle= arctan2(TE.y,TE.x)
1060         point_at_correct_angle = TE.clone().rotate_about_zaxis((th2-th1)*(z_max-z1)/(z2-z1)
)+(th1-TE_angle))
1061         point_in_between = Vector(point_at_correct_angle.x,point_at_correct_angle.y,TE.z)
1062         points = walking_along_the_curve(p1,point_in_between,s)
1063         #points = [point_in_between]
1064         points.insert(0,p1)
1065         points.append(point_in_between)
1066         #print "factor "
1067         #print "point in betwe"
1068         points.append(p2)
1069         connection = Spline(points)
1070         #print "p1 in between , p2 in outflow"
1071     elif z2 < z_max:
1072         s = -1
1073         TE_angle= arctan2(TE.y,TE.x)
1074         point_at_correct_angle = TE.clone().rotate_about_zaxis((th2-th1)*(z_max-z1)/(z2-z1)+(
th1-TE_angle))
1075         point_in_between = Vector(point_at_correct_angle.x,point_at_correct_angle.y,TE.z)
1076         points = walking_along_the_camber(point_in_between,p2,s,1,top_or_bottom)
1077         points.insert(0,point_in_between)
1078         points.insert(0,p1)
1079         points.append(p2)
1080         connection = Spline(points)
1081         #print "p1 in outflow , p2 in bewteen"
1082     else:
1083         points = []
1084         n = number_of_points_on_connection +1
1085         for i in range(1,n):
1086             th_in_between = th1*(1-i/float(n))+th2*i/n
1087             r_in_between = r1*(1-i/float(n))+r2*i/n
1088             x = cos(th_in_between) * r_in_between #r1 should be equal to r2 but its only close
.. so this will but them closer together
1089             y = sin(th_in_between) * r_in_between
1090             z = z1*(1-i/float(n))+z2*i/n
1091             points.append(Vector(x,y,z).clone())
1092             points.insert(0,p1)
1093             points.append(p2)
1094             connection = Spline(points)
1095             #print "p1 in outflow , p2 in outflow"
1096         return connection
1097
1098
1099
1100 camber_curves = input_curves # I SHOULD PUT THIS IN FRONT OF EVERYTHING
1101 blade_pressure_curves = input_curves
1102 ### Here the three parts of the surface on a periodic boundary are joined together .
1103
1104 def surface_connector_perodic_pressure(Surf1 ,Surf2 ,Surf3):
1105     def true_surface(r,s):
1106         p1 = PathOnSurface(Surf1,LinearFunction(1.0,0.0) , LinearFunction(0.0,float(s)))
1107         p2 = PathOnSurface(Surf1,LinearFunction(1.0,0.0) , LinearFunction(0.0,float(s)))
1108         p3 = PathOnSurface(Surf1,LinearFunction(1.0,0.0) , LinearFunction(0.0,float(s)))
1109         length_p1 = accurate_length(p1)
1110         length_p2 = accurate_length(p2)
1111         length_p3 = accurate_length(p3)
1112         total_p = length_p1+length_p2+length_p3
1113         if r <= length_p1/total_p:
1114             p = Surf1.eval(r*total_p/length_p1,s)
1115         elif r < (length_p1+length_p2)/total_p:
1116             r_coordinate = (r-length_p1/total_p)*total_p/length_p2
1117             p = Surf2.eval(r_coordinate,s)
1118         else:
1119             r_coordinate = (r-(length_p1+length_p2)/total_p)*total_p/length_p3
1120             p = Surf3.eval(r_coordinate,s)
1121         x = p.x
1122         y = p.y
1123         z = p.z
1124         return (x,y,z)
1125     return true_surface
1126
1127 pressure_periodic = PyFunctionSurface(surface_connector_perodic_pressure(v6south ,south ,v2south
))

```

```

1128 suction_periodic = PyFunctionSurface(surface_connector_perodic_pressure(v5north,v4north,
1129                                     v3north))
1130
1131 ### Function to create blocks in the H-Block part
1132
1133 def create_H_Block(nodes_on_periodic_boundary , nodes_on_O_gridline , pressure_or_suction): #
1134     example : create_H_Block([0.5,0.75],[0.4,0.66],1)
1135     if pressure_or_suction > 0 :
1136         def H_south_python(nodes):
1137             def H_s(r,s):
1138                 r_coordinate = (nodes[1]-nodes[0])*r + nodes[0]
1139                 #print "r_coordinate",r_coordinate , "nodes[0]" , nodes[0] , "nodes[1]" , nodes
1140                 [1]
1141                 point = pressure_periodic.eval(r_coordinate ,s)
1142                 x = point.x
1143                 y = point.y
1144                 z = point.z
1145                 return (x,y,z)
1146             return H_s
1147         H_south = PyFunctionSurface(H_south_python(nodes_on_periodic_boundary))
1148         H_c01 = PathOnSurface (H_south , LinearFunction(1.0,0.0) , LinearFunction(0.0,0.0))
1149         H_c15 = PathOnSurface (H_south , LinearFunction(0.0,1.0) , LinearFunction(1.0,0.0))
1150         H_c04 = PathOnSurface (H_south , LinearFunction(0.0,0.0) , LinearFunction(1.0,0.0))
1151         H_c45 = PathOnSurface (H_south , LinearFunction(1.0,0.0) , LinearFunction(0.0,1.0))
1152         #print "H_south +-----+"
1153
1154         def H_north_python(r,s):
1155             r_coordinate = (nodes_on_O_gridline[1]-nodes_on_O_gridline[0])*r +
1156             nodes_on_O_gridline[0]
1157             point = O_grid_line_surface_pressure.eval(r_coordinate ,s)
1158             x = point.x
1159             y = point.y
1160             z = point.z
1161             return (x,y,z)
1162
1163         H_north = PyFunctionSurface(H_north_python)
1164         H_c32 = PathOnSurface (H_north , LinearFunction(1.0,0.0) , LinearFunction(0.0,0.0))
1165         H_c26 = PathOnSurface (H_north , LinearFunction(0.0,1.0) , LinearFunction(1.0,0.0))
1166         H_c37 = PathOnSurface (H_north , LinearFunction(0.0,0.0) , LinearFunction(1.0,0.0))
1167         H_c76 = PathOnSurface (H_north , LinearFunction(1.0,0.0) , LinearFunction(0.0,1.0))
1168         #print "H_north +-----+"
1169
1170         H_p4 = H_c45.eval(0)
1171         H_p5 = H_c45.eval(1)
1172         H_p7 = H_c76.eval(0)
1173         H_p6 = H_c76.eval(1)
1174
1175         H_p0 = H_c01.eval(0)
1176         H_p1 = H_c01.eval(1)
1177         H_p3 = H_c32.eval(0)
1178         H_p2 = H_c32.eval(1)
1179
1180         #if pressure_or_suction > 0:
1181         Ogridcurve_top = PathOnSurface(O_grid_line_surface_pressure , LinearFunction(1.0,0.0) ,
1182         LinearFunction(0.0,1.0))
1183         Ogridcurve_bottom = PathOnSurface(O_grid_line_surface_pressure , LinearFunction
1184         (1.0,0.0) , LinearFunction(0.0,0.0))
1185
1186         H_c47 = simple_connector(H_p7,H_p4,nodes_on_O_gridline[0],Ogridcurve_top , camber_curves
1187         ,-1)
1188         H_c56 = simple_connector(H_p6,H_p5,nodes_on_O_gridline[1],Ogridcurve_top , camber_curves
1189         ,-1)
1190         H_c03 = simple_connector(H_p3,H_p0,nodes_on_O_gridline[0],Ogridcurve_bottom ,
1191         camber_curves,0)
1192         H_c12 = simple_connector(H_p2,H_p1,nodes_on_O_gridline[1],Ogridcurve_bottom ,
1193         camber_curves,0)
1194
1195         H_c47.reverse()
1196         H_c56.reverse()
1197         H_c03.reverse()
1198         H_c12.reverse()
1199
1200         p0 = H_c03.eval(0)
1201         p1 = H_c12.eval(0)
1202         p2 = H_c12.eval(1)

```

```

1193     p3 = H_c03.eval(1)
1194     p4 = H_c47.eval(0)
1195     p5 = H_c56.eval(0)
1196     p6 = H_c56.eval(1)
1197     p7 = H_c47.eval(1)
1198
1199     #p0 = H_c01.eval(0)
1200     #p1 = H_c01.eval(1)
1201     #p2 = H_c32.eval(1)
1202     #p3 = H_c32.eval(0)
1203     #p4 = H_c45.eval(0)
1204     #p5 = H_c45.eval(1)
1205     #p6 = H_c76.eval(1)
1206     #p7 = H_c76.eval(0)
1207
1208     #print "p0", p0 , "\n p1" , p1 , "\n p2" ,p2 , "\n p3" ,p3 ,"\n p4" ,p4 , "\n p5" , p5
1209     ,"\n p6",p6 ,"\n p7" ,p7
1210     H_top = CoonsPatch(H_c45,H_c76,H_c47,H_c56)
1211     #print "H_top+=====+"
1212     #
1213     H_bottom = CoonsPatch(H_c01,H_c32,H_c03,H_c12)
1214     #print "H_bottom+=====+"
1215
1216     H_east = CoonsPatch(H_c12,H_c56,H_c15,H_c26)
1217     #print "H_east+=====+"
1218
1219     H_west = CoonsPatch(H_c03,H_c47,H_c04,H_c37)
1220     #print "H_west +=====+"
1221
1222     H_volume = ParametricVolume(H_north ,H_east ,H_south ,H_west ,H_top ,H_bottom)
1223
1224     #A=[]
1225     A.append(H_c01)
1226     A.append(H_c32)
1227     #A.append(H_c12)
1228     A.append(H_c03)
1229     A.append(H_c45)
1230     #A.append(H_c56)
1231     A.append(H_c76)
1232     A.append(H_c47)
1233     #A.append(H_c26)
1234     #A.append(H_c15)
1235     A.append(H_c04)
1236     A.append(H_c37)
1237
1238     elif pressure_or_suction < 0:
1239         def H_south_python(r,s):
1240             r_coordinate = (nodes_on_O_gridline[1]-nodes_on_O_gridline[0])*r +
1241             nodes_on_O_gridline[0]
1242             point = O_grid_line_surface_suction.eval(r_coordinate ,s)
1243             x = point.x
1244             y = point.y
1245             z = point.z
1246             return (x,y,z)
1247         H_south = PyFunctionSurface(H_south_python)
1248
1249         H_c01 = PathOnSurface (H_south , LinearFunction(1.0,0.0) , LinearFunction(0.0,0.0))
1250         H_c15 = PathOnSurface (H_south , LinearFunction(0.0,1.0) , LinearFunction(1.0,0.0))
1251         H_c04 = PathOnSurface (H_south , LinearFunction(0.0,0.0) , LinearFunction(1.0,0.0))
1252         H_c45 = PathOnSurface (H_south , LinearFunction(1.0,0.0) , LinearFunction(0.0,1.0))
1253         #print "H_south +=====+"
1254
1255         def H_north_python(nodes):
1256             def H_n(r,s):
1257                 r_coordinate = (nodes[1]-nodes[0])*r + nodes[0]
1258                 #print "r_coordinate",r_coordinate , "nodes[0]" , nodes[0] , "nodes[1]" , nodes
1259                 [1]
1260                 point = suction_periodic.eval(r_coordinate ,s)
1261                 x = point.x
1262                 y = point.y
1263                 z = point.z
1264                 return (x,y,z)
1265             return H_n
1266         H_north = PyFunctionSurface(H_north_python(nodes_on_periodic_boundary))

```

```

1265
1266     H_c32 = PathOnSurface (H_north, LinearFunction(1.0,0.0), LinearFunction(0.0,0.0))
1267     H_c26 = PathOnSurface (H_north, LinearFunction(0.0,1.0), LinearFunction(1.0,0.0))
1268     H_c37 = PathOnSurface (H_north, LinearFunction(0.0,0.0), LinearFunction(1.0,0.0))
1269     H_c76 = PathOnSurface (H_north, LinearFunction(1.0,0.0), LinearFunction(0.0,1.0))
1270     #print "H_north +<=====>+"
1271
1272     H_p4 = H_c45.eval(0)
1273     H_p5 = H_c45.eval(1)
1274     H_p7 = H_c76.eval(0)
1275     H_p6 = H_c76.eval(1)
1276
1277     H_p0 = H_c01.eval(0)
1278     H_p1 = H_c01.eval(1)
1279     H_p3 = H_c32.eval(0)
1280     H_p2 = H_c32.eval(1)
1281
1282     Ogridcurve_top = PathOnSurface(O_grid_line_surface_suction, LinearFunction(1.0,0.0),
LinearFunction(0.0,1.0))
1283     Ogridcurve_bottom = PathOnSurface(O_grid_line_surface_suction, LinearFunction(1.0,0.0)
, LinearFunction(0.0,0.0))
1284
1285     H_c47 = simple_connector(H_p4,H_p7,nodes_on_O_gridline[0],Ogridcurve_top,camber_curves
,-1)
1286     H_c56 = simple_connector(H_p5,H_p6,nodes_on_O_gridline[1],Ogridcurve_top,camber_curves
,-1)
1287     H_c03 = simple_connector(H_p0,H_p3,nodes_on_O_gridline[0],Ogridcurve_bottom,
camber_curves,0)
1288     H_c12 = simple_connector(H_p1,H_p2,nodes_on_O_gridline[1],Ogridcurve_bottom,
camber_curves,0)
1289
1290
1291     #print "p0", p0, "\n p1" , p1, "\n p2" ,p2, "\n p3" ,p3, "\n p4" ,p4, "\n p5" , p5
," \n p6",p6, "\n p7" ,p7
1292     H_top = CoonsPatch(H_c45,H_c76,H_c47,H_c56)
1293     #print "H_top+<=====>+"
1294     #
1295     H_bottom = CoonsPatch(H_c01,H_c32,H_c03,H_c12)
1296     #print "H_bottom+<=====>+"
1297
1298     H_east = CoonsPatch(H_c12,H_c56,H_c15,H_c26)
1299     #print "H_east+<=====>+"
1300
1301     H_west = CoonsPatch(H_c03,H_c47,H_c04,H_c37)
1302     #print "H_west +<=====>+"
1303
1304     H_volume = ParametricVolume(H_north,H_east,H_south,H_west,H_top,H_bottom)
1305
1306     #A=[]
1307     A.append(H_c01)
1308     A.append(H_c32)
1309     A.append(H_c12)
1310     #A.append(H_c03)
1311     A.append(H_c45)
1312     A.append(H_c56)
1313     A.append(H_c76)
1314     #A.append(H_c47)
1315     A.append(H_c26)
1316     A.append(H_c15)
1317     #A.append(H_c04)
1318     #A.append(H_c37)
1319     #for n in range(0,len(A)):
1320     #     string = "curve_file%g.txt" %n
1321     #     print string , n , " of " , len(A)
1322     #     with open(string,"w") as thisfile:
1323     #         thisfile.write("")
1324     #     with open(string,"a") as thisfile:
1325     #         for i in range(0,101):
1326     #             B = A[n].eval(i*0.01)
1327     #             C = "%g %g %g \n" %(B.x,B.y,B.z)
1328     #             #print C
1329     #             thisfile.write(C)
1330     #H_block = (Block3D(label="TEST-BLOCK", nni=5, nnj=5, nnk=5,
1331     #     parametric_volume=H_volume,
1332     #     fill_condition=initialCond))

```

```

1333     return "hello"
1334     #return H_block
1335 #print create_H_Block([0.3,0.4],[0.3,0.4],-1)
1336
1337 #first_H_block_ever = create_H_Block([0.3,0.4],[0.3,0.4],1)
1338
1339
1340 ##### Here a function is created to create blocks in the O-grid part
1341
1342 def create_O_Block(nodes_on_O_gridline , nodes_on_blade , pressure_or_suction): #example :
1343     create_O_Block([0.5,0.75],[0.4,0.66],1)
1344     if pressure_or_suction > 0 :
1345         def O_south_python(nodes):
1346             def O_s(r,s):
1347                 r.coordinate = (nodes[1]-nodes[0])*r + nodes[0]
1348                 point = O_grid_line_surface_pressure.eval(r.coordinate,s)
1349                 x = point.x
1350                 y = point.y
1351                 z = point.z
1352                 return (x,y,z)
1353             return O_s
1354         O_south = PyFunctionSurface(O_south_python(nodes_on_O_gridline))
1355         O_c01 = PathOnSurface(O_south, LinearFunction(1.0,0.0), LinearFunction(0.0,0.0))
1356         O_c15 = PathOnSurface(O_south, LinearFunction(0.0,1.0), LinearFunction(1.0,0.0))
1357         O_c04 = PathOnSurface(O_south, LinearFunction(0.0,0.0), LinearFunction(1.0,0.0))
1358         O_c45 = PathOnSurface(O_south, LinearFunction(1.0,0.0), LinearFunction(0.0,1.0))
1359         #print "H_south +-----+"
1360
1361         def O_north_python(r,s):
1362             r.coordinate = (nodes_on_blade[1]-nodes_on_blade[0])*r + nodes_on_blade[0]
1363             point = north.eval(r.coordinate,s)
1364             x = point.x
1365             y = point.y
1366             z = point.z
1367             return (x,y,z)
1368
1369         O_north = PyFunctionSurface(O_north_python)
1370         O_c32 = PathOnSurface(O_north, LinearFunction(1.0,0.0), LinearFunction(0.0,0.0))
1371         O_c26 = PathOnSurface(O_north, LinearFunction(0.0,1.0), LinearFunction(1.0,0.0))
1372         O_c37 = PathOnSurface(O_north, LinearFunction(0.0,0.0), LinearFunction(1.0,0.0))
1373         O_c76 = PathOnSurface(O_north, LinearFunction(1.0,0.0), LinearFunction(0.0,1.0))
1374         #print "H_north +-----+"
1375
1376         O_p4 = O_c45.eval(0)
1377         O_p5 = O_c45.eval(1)
1378         O_p7 = O_c76.eval(0)
1379         O_p6 = O_c76.eval(1)
1380
1381         O_p0 = O_c01.eval(0)
1382         O_p1 = O_c01.eval(1)
1383         O_p3 = O_c32.eval(0)
1384         O_p2 = O_c32.eval(1)
1385         #print "p0", p0 , "\n p1" , p1 , "\n p2" , p2 , "\n p3" , p3 , "\n p4" , p4 , "\n p5" , p5
1386         #print "p6", p6 , "\n p7" , p7
1387         #if pressure_or_suction > 0:
1388         Ogridcurve_top = PathOnSurface(O_grid_line_surface_pressure , LinearFunction(1.0,0.0) ,
1389         LinearFunction(0.0,1.0))
1390         Ogridcurve_bottom = PathOnSurface(O_grid_line_surface_pressure , LinearFunction
1391         (1.0,0.0) , LinearFunction(0.0,0.0))
1392         #
1393         O_c47 = simple_connector(O_p4,O_p7,nodes_on_O_gridline[0],Ogridcurve_top , camber_curves
1394         ,-1)
1395         O_c56 = simple_connector(O_p5,O_p6,nodes_on_O_gridline[1],Ogridcurve_top , camber_curves
1396         ,-1)
1397         O_c03 = simple_connector(O_p0,O_p3,nodes_on_O_gridline[0],Ogridcurve_bottom ,
1398         camber_curves,0)
1399         O_c12 = simple_connector(O_p1,O_p2,nodes_on_O_gridline[1],Ogridcurve_bottom ,
1400         camber_curves,0)
1401         #
1402         #
1403         #print "p0", p0 , "\n p1" , p1 , "\n p2" , p2 , "\n p3" , p3 , "\n p4" , p4 , "\n p5" , p5
1404         #print "p6", p6 , "\n p7" , p7
1405         O_top = CoonsPatch(O_c45 , O_c76 , O_c47 , O_c56)
1406         #print "O_top+-----+"
1407         #

```

```

1399     O_bottom = CoonsPatch(O_c01, O_c32, O_c03, O_c12)
1400     #print "O_bottom" + "=====" + "}"
1401
1402     O_east = CoonsPatch(O_c12, O_c56, O_c15, O_c26)
1403     #print "O_east" + "=====" + "}"
1404
1405     O_west = CoonsPatch(O_c03, O_c47, O_c04, O_c37)
1406     #print "O_west" + "=====" + "}"
1407
1408     O_volume = ParametricVolume(O_north, O_east, O_south, O_west, O_top, O_bottom)
1409
1410     #A=[]
1411     A.append(O_c01)
1412     A.append(O_c32)
1413     A.append(O_c12)
1414     A.append(O_c03)
1415     A.append(O_c45)
1416     A.append(O_c56)
1417     A.append(O_c76)
1418     A.append(O_c47)
1419     A.append(O_c26)
1420     A.append(O_c15)
1421     A.append(O_c04)
1422     A.append(O_c37)
1423     ##for n in range(0, len(A)):
1424     ##     string = "curve_file%g.txt" %n
1425     ##     print string, n
1426     ##     with open(string, "w") as thisfile:
1427     ##         thisfile.write("")
1428     ##     with open(string, "a") as thisfile:
1429     ##         for i in range(0, 101):
1430     ##             B = A[n].eval(i*0.01)
1431     ##             C = "%g %g %g \n" % (B.x, B.y, B.z)
1432     ##             #print C
1433     ##             thisfile.write(C)
1434
1435     elif pressure_or_suction < 0:
1436         def O_north_python(nodes):
1437             def O_n(r, s):
1438                 r_coordinate = (nodes[1] - nodes[0]) * r + nodes[0]
1439                 point = O_grid_line_surface_suction.eval(r_coordinate, s)
1440                 x = point.x
1441                 y = point.y
1442                 z = point.z
1443                 return (x, y, z)
1444             return O_n
1445         O_north = PyFunctionSurface(O_north_python(nodes_on_O_gridline))
1446         O_c32 = PathOnSurface(O_north, LinearFunction(1.0, 0.0), LinearFunction(0.0, 0.0))
1447         O_c26 = PathOnSurface(O_north, LinearFunction(0.0, 1.0), LinearFunction(1.0, 0.0))
1448         O_c37 = PathOnSurface(O_north, LinearFunction(0.0, 0.0), LinearFunction(1.0, 0.0))
1449         O_c76 = PathOnSurface(O_north, LinearFunction(1.0, 0.0), LinearFunction(0.0, 1.0))
1450         #print "H_north" + "=====" + "}"
1451
1452         def O_south_python(r, s):
1453             r_coordinate = (nodes_on_blade[1] - nodes_on_blade[0]) * r + nodes_on_blade[0]
1454             point = v4south.eval(r_coordinate, s)
1455             x = point.x
1456             y = point.y
1457             z = point.z
1458             return (x, y, z)
1459
1460         O_south = PyFunctionSurface(O_south_python)
1461         O_c01 = PathOnSurface(O_south, LinearFunction(1.0, 0.0), LinearFunction(0.0, 0.0))
1462         O_c15 = PathOnSurface(O_south, LinearFunction(0.0, 1.0), LinearFunction(1.0, 0.0))
1463         O_c04 = PathOnSurface(O_south, LinearFunction(0.0, 0.0), LinearFunction(1.0, 0.0))
1464         O_c45 = PathOnSurface(O_south, LinearFunction(1.0, 0.0), LinearFunction(0.0, 1.0))
1465         #print "H_south" + "=====" + "}"
1466
1467         O_p4 = O_c45.eval(0)
1468         O_p5 = O_c45.eval(1)
1469         O_p7 = O_c76.eval(0)
1470         O_p6 = O_c76.eval(1)
1471
1472         O_p0 = O_c01.eval(0)
1473         O_p1 = O_c01.eval(1)

```



```

1474     O_p3 = O_c32.eval(0)
1475     O_p2 = O_c32.eval(1)
1476     #print "p0", p0 , "\n p1" , p1 , "\n p2" ,p2 , "\n p3" ,p3 ,"\n p4" ,p4 , "\n p5" , p5
    ," \n p6",p6 ,"\n p7" ,p7
1477     #if pressure_or_suction > 0:
1478         Ogridcurve_top = PathOnSurface(O_grid_line_surface_suction , LinearFunction(1.0,0.0) ,
LinearFunction(0.0,1.0))
1479         Ogridcurve_bottom = PathOnSurface(O_grid_line_surface_suction , LinearFunction(1.0,0.0)
, LinearFunction(0.0,0.0))
1480         #
1481         O_c47 = simple_connector(O_p7,O_p4,nodes_on_O_gridline[0],Ogridcurve_top ,camber_curves
,-1)
1482         O_c56 = simple_connector(O_p6,O_p5,nodes_on_O_gridline[1],Ogridcurve_top ,camber_curves
,-1)
1483         O_c03 = simple_connector(O_p3,O_p0,nodes_on_O_gridline[0],Ogridcurve_bottom ,
camber_curves,0)
1484         O_c12 = simple_connector(O_p2,O_p1,nodes_on_O_gridline[1],Ogridcurve_bottom ,
camber_curves,0)
1485         O_c47.reverse()
1486         O_c56.reverse()
1487         O_c03.reverse()
1488         O_c12.reverse()
1489         #
1490         #
1491         #print "p0", p0 , "\n p1" , p1 , "\n p2" ,p2 , "\n p3" ,p3 ,"\n p4" ,p4 , "\n p5" , p5
    ," \n p6",p6 ,"\n p7" ,p7
1492         O_top = CoonsPatch(O_c45,O_c76,O_c47,O_c56)
1493         #print "O_top+====+|"
1494         #
1495         O_bottom = CoonsPatch(O_c01,O_c32,O_c03,O_c12)
1496         #print "O_bottom+====+|"
1497
1498         O_east = CoonsPatch(O_c12,O_c56,O_c15,O_c26)
1499         #print "O_east+====+|"
1500
1501         O_west = CoonsPatch(O_c03,O_c47,O_c04,O_c37)
1502         #print "O_west +====+|"
1503
1504         O_volume = ParametricVolume(O_north ,O_east ,O_south ,O_west ,O_top ,O_bottom)
1505
1506         #A=[]
1507         A.append(O_c01)
1508         A.append(O_c32)
1509         A.append(O_c12)
1510         A.append(O_c03)
1511         A.append(O_c45)
1512         A.append(O_c56)
1513         A.append(O_c76)
1514         A.append(O_c47)
1515         A.append(O_c26)
1516         A.append(O_c15)
1517         A.append(O_c04)
1518         A.append(O_c37)
1519         #for n in range(0,len(A)):
1520         #     string = "curve_file%g.txt" %n
1521         #     print string , n
1522         #     with open(string,"w") as thisfile:
1523         #         thisfile.write("")
1524         #     with open(string,"a") as thisfile:
1525         #         for i in range(0,101):
1526         #             B = A[n].eval(i*0.01)
1527         #             C = "%g %g %g \n" %(B.x,B.y,B.z)
1528         #             #print C
1529         #             thisfile.write(C)
1530
1531
1532         #O_block = (Block3D(label="TEST-BLOCK", nni=5, nnj=5, nnk=5,
1533         #parametric_volume=O_volume,
1534         #fill_condition=initialCond))
1535         return "hello"
1536         #return O_block
1537
1538 #print create_O_Block([0.3,0.4],[0.3,0.4],-1)
1539
1540 ##### Here a function is created to create blocks in the inlet region of the H-grid part

```

```

1541
1542 def create_In_Block(nodes_on_inflow_boundary, nodes_on_O_gridline, pressure_or_suction): #
1543     example : create_In_Block([0.5,0.75],[0.4,0.66],1)
1544     if pressure_or_suction > 0 :
1545         In_p7 = v6c47.eval(nodes_on_inflow_boundary[1])
1546         In_p4 = v6c47.eval(nodes_on_inflow_boundary[0])
1547         In_p0 = v6c03.eval(nodes_on_inflow_boundary[0])
1548         In_p3 = v6c03.eval(nodes_on_inflow_boundary[1])
1549
1550         In_c47 = Arc(In_p4, In_p7, Vector(0,0,p7.z))
1551         In_c03 = Arc(In_p0, In_p3, Vector(0,0,p0.z))
1552         In_c04 = Line(In_p0, In_p4)
1553         In_c37 = Line(In_p3, In_p7)
1554
1555         In_west = CoonsPatch(In_c03, In_c47, In_c04, In_c37)
1556         #print "In_west +-----+
1557
1558     def In_east_python(r,s):
1559         r_coordinate = (nodes_on_O_gridline[0] - nodes_on_O_gridline[1])*r +
1560         nodes_on_O_gridline[1]
1561         point = O_grid_line_surface_pressure.eval(r_coordinate, s)
1562         x = point.x
1563         y = point.y
1564         z = point.z
1565         return (x,y,z)
1566
1567     In_east = PyFunctionSurface(In_east_python)
1568     In_c12 = PathOnSurface(In_east, LinearFunction(1.0,0.0), LinearFunction(0.0,0.0))
1569     In_c56 = PathOnSurface(In_east, LinearFunction(1.0,0.0), LinearFunction(0.0,1.0))
1570     In_c15 = PathOnSurface(In_east, LinearFunction(0.0,0.0), LinearFunction(1.0,0.0))
1571     In_c26 = PathOnSurface(In_east, LinearFunction(0.0,1.0), LinearFunction(1.0,0.0))
1572     #print "In_east +-----+
1573
1574     In_p1 = In_c12.eval(0)
1575     In_p2 = In_c12.eval(1)
1576     In_p5 = In_c56.eval(0)
1577     In_p6 = In_c56.eval(1)
1578
1579     #if pressure_or_suction > 0:
1580     Ogridcurve_top = PathOnSurface(O_grid_line_surface_pressure, LinearFunction(1.0,0.0),
1581     LinearFunction(0.0,1.0))
1582     Ogridcurve_bottom = PathOnSurface(O_grid_line_surface_pressure, LinearFunction
1583     (1.0,0.0), LinearFunction(0.0,0.0))
1584
1585     In_c45 = simple_connector(In_p5, In_p4, nodes_on_O_gridline[0], Ogridcurve_top,
1586     camber_curves, -1)
1587     In_c76 = simple_connector(In_p6, In_p7, nodes_on_O_gridline[1], Ogridcurve_top,
1588     camber_curves, -1)
1589     In_c01 = simple_connector(In_p1, In_p0, nodes_on_O_gridline[0], Ogridcurve_bottom,
1590     camber_curves, 0)
1591     In_c32 = simple_connector(In_p2, In_p3, nodes_on_O_gridline[1], Ogridcurve_bottom,
1592     camber_curves, 0)
1593     In_c45.reverse()
1594     In_c76.reverse()
1595     In_c01.reverse()
1596     In_c32.reverse()
1597
1598     ##p0 = In_c03.eval(0)
1599     ##p1 = In_c12.eval(0)
1600     ##p2 = In_c12.eval(1)
1601     ##p3 = In_c03.eval(1)
1602     ##p4 = In_c47.eval(0)
1603     ##p5 = In_c56.eval(0)
1604     ##p6 = In_c56.eval(1)
1605     ##p7 = In_c47.eval(1)
1606
1607     ##
1608     ##p0 = In_c01.eval(0)
1609     ##p1 = In_c01.eval(1)
1610     ##p2 = In_c32.eval(1)
1611     ##p3 = In_c32.eval(0)
1612     ##p4 = In_c45.eval(0)
1613     ##p5 = In_c45.eval(1)
1614     ##p6 = In_c76.eval(1)
1615     ##p7 = In_c76.eval(0)
1616
1617

```

```

1608     ##print "p0", p0 , "\n p1" , p1 , "\n p2" ,p2 , "\n p3" ,p3 ,"\n p4" ,p4 , "\n p5" ,
p5     ,"\n p6" ,p6 , "\n p7" ,p7
1609     In_top = CoonsPatch(In_c45 ,In_c76 ,In_c47 ,In_c56)
1610     #print "In_top"+=====+"
1611     #
1612     In_bottom = CoonsPatch(In_c01 ,In_c32 ,In_c03 ,In_c12)
1613     #print "In_bottom"+=====+"
1614
1615     In_north = CoonsPatch(In_c32 ,In_c76 ,In_c37 ,In_c26)
1616     #print "In_north"+=====+"
1617
1618     In_south = CoonsPatch(In_c01 ,In_c45 ,In_c04 ,In_c15)
1619     #print "In_south"+=====+"
1620
1621     In_volume = ParametricVolume(In_north ,In_east ,In_south ,In_west ,In_top ,In_bottom)
1622
1623     #A=[]
1624     #A.append(In_c01)
1625     A.append(In_c32)
1626     A.append(In_c12)
1627     A.append(In_c03)
1628     #A.append(In_c45)
1629     A.append(In_c56)
1630     A.append(In_c76)
1631     A.append(In_c47)
1632     A.append(In_c26)
1633     #A.append(In_c15)
1634     #A.append(In_c04)
1635     A.append(In_c37)
1636     #for n in range(0,len(A)):
1637     #     string = "curve_file%g.txt" %n
1638     #     print string , n
1639     #     with open(string,"w") as thisfile:
1640     #         thisfile.write("")
1641     #     with open(string,"a") as thisfile:
1642     #         for i in range (0,101):
1643     #             B = A[n].eval(i*0.01)
1644     #             C = "%g %g %g \n" %(B.x,B.y,B.z)
1645     #             #print C
1646     #             thisfile.write(C)
1647
1648     elif pressure_or_suction < 0:
1649         In_p7 = v5c47.eval(nodes_on_inflow_bondary [1])
1650         In_p4 = v5c47.eval(nodes_on_inflow_bondary [0])
1651         In_p0 = v5c03.eval(nodes_on_inflow_bondary [0])
1652         In_p3 = v5c03.eval(nodes_on_inflow_bondary [1])
1653
1654         In_c47 = Arc(In_p4 ,In_p7 ,Vector(0,0,p7.z))
1655         In_c03 = Arc(In_p0 ,In_p3 ,Vector(0,0,p0.z))
1656         In_c04 = Line(In_p0 ,In_p4)
1657         In_c37 = Line(In_p3 ,In_p7)
1658
1659         In_west = CoonsPatch(In_c03 ,In_c47 ,In_c04 ,In_c37)
1660         #print "In_west "+=====+"
1661
1662         def In_east_python(r,s):
1663             r_coordinate = (nodes_on_O_gridline[1]-nodes_on_O_gridline[0])*r +
nodes_on_O_gridline[0]
1664             point = O_grid_line_surface_suction.eval(r_coordinate ,s)
1665             x = point.x
1666             y = point.y
1667             z = point.z
1668             return (x,y,z)
1669
1670         In_east = PyFunctionSurface(In_east_python)
1671         In_c12 = PathOnSurface (In_east , LinearFunction(1.0,0.0) , LinearFunction(0.0,0.0))
1672         In_c56 = PathOnSurface (In_east , LinearFunction(1.0,0.0) , LinearFunction(0.0,1.0))
1673         In_c15 = PathOnSurface (In_east , LinearFunction(0.0,0.0) , LinearFunction(1.0,0.0))
1674         In_c26 = PathOnSurface (In_east , LinearFunction(0.0,1.0) , LinearFunction(1.0,0.0))
1675         #print "In_east "+=====+"
1676
1677         In_p1 = In_c12.eval(0)
1678         In_p2 = In_c12.eval(1)
1679         In_p5 = In_c56.eval(0)
1680         In_p6 = In_c56.eval(1)

```

```

1681
1682     #if pressure_or_suction > 0:
1683         Ogridcurve_top = PathOnSurface(O_grid_line_surface_suction , LinearFunction(1.0,0.0) ,
LinearFunction(0.0,1.0))
1684         Ogridcurve_bottom = PathOnSurface(O_grid_line_surface_suction , LinearFunction(1.0,0.0)
, LinearFunction(0.0,0.0))
1685
1686         In_c45 = simple_connector(In_p5,In_p4,nodes_on_O_gridline[0],Ogridcurve_top ,
camber_curves,-1)
1687         In_c76 = simple_connector(In_p6,In_p7,nodes_on_O_gridline[1],Ogridcurve_top ,
camber_curves,-1)
1688         In_c01 = simple_connector(In_p1,In_p0,nodes_on_O_gridline[0],Ogridcurve_bottom ,
camber_curves,0)
1689         In_c32 = simple_connector(In_p2,In_p3,nodes_on_O_gridline[1],Ogridcurve_bottom ,
camber_curves,0)
1690         In_c45.reverse()
1691         In_c76.reverse()
1692         In_c01.reverse()
1693         In_c32.reverse()
1694
1695         ##p0 = In_c03.eval(0)
1696         ##p1 = In_c12.eval(0)
1697         ##p2 = In_c12.eval(1)
1698         ##p3 = In_c03.eval(1)
1699         ##p4 = In_c47.eval(0)
1700         ##p5 = In_c56.eval(0)
1701         ##p6 = In_c56.eval(1)
1702         ##p7 = In_c47.eval(1)
1703         ##
1704         ##p0 = In_c01.eval(0)
1705         ##p1 = In_c01.eval(1)
1706         ##p2 = In_c32.eval(1)
1707         ##p3 = In_c32.eval(0)
1708         ##p4 = In_c45.eval(0)
1709         ##p5 = In_c45.eval(1)
1710         ##p6 = In_c76.eval(1)
1711         ##p7 = In_c76.eval(0)
1712
1713         ##print "p0", p0 , "\n p1" , p1 , "\n p2" ,p2 , "\n p3" ,p3 ,"\n p4" ,p4 , "\n p5" ,
p5 ,"\n p6",p6 ,"\n p7" ,p7
1714         In_top = CoonsPatch(In_c45,In_c76,In_c47,In_c56)
1715         #print "In_top+=====+|"
1716         #
1717         In_bottom = CoonsPatch(In_c01,In_c32,In_c03,In_c12)
1718         #print "In_bottom+=====+|"
1719
1720         In_north = CoonsPatch(In_c32,In_c76,In_c37,In_c26)
1721         #print "In_north+=====+|"
1722
1723         In_south = CoonsPatch(In_c01,In_c45,In_c04,In_c15)
1724         #print "In_south+=====+|"
1725
1726         In_volume = ParametricVolume(In_north,In_east,In_south,In_west,In_top,In_bottom)
1727
1728         #A=[]
1729         A.append(In_c01)
1730         #A.append(In_c32)
1731         A.append(In_c12)
1732         A.append(In_c03)
1733         A.append(In_c45)
1734         A.append(In_c56)
1735         #A.append(In_c76)
1736         A.append(In_c47)
1737         #A.append(In_c26)
1738         A.append(In_c15)
1739         A.append(In_c04)
1740         #A.append(In_c37)
1741         #for n in range(0,len(A)):
1742         #     string = "curve_file%.txt" %n
1743         #     print string , n
1744         #     with open(string,"w") as thisfile:
1745         #         thisfile.write("")
1746         #     with open(string,"a") as thisfile:
1747         #         for i in range(0,101):
1748         #             B = A[n].eval(i*0.01)

```

```

1749         #           C = "%g %g %g \n" %(B.x,B.y,B.z)
1750         #           #print C
1751         #           thisfile.write(C)
1752
1753         #In_block = (Block3D(label="TEST-BLOCK", nni=5, nnj=5, nnk=5,
1754         #           parametric_volume=In_volume,
1755         #           fill_condition=initialCond))
1756         return "hello"
1757         #return In_block
1758
1759 #print create_In_Block([0.3,0.4],[0.02,0.025],-1)
1760
1761 ##### Here a function is created to create blocks in the outlet region of the H-grid part
1762
1763 def create_Out_Block(nodes_on_outflow_boundary,nodes_on_O_gridline,pressure_or_suction): #
1764     example : create_H_Block([0.5,0.75],[0.4,0.66],1)
1765     if pressure_or_suction > 0 :
1766         Out_p6 = v2c56.eval(nodes_on_outflow_boundary[1])
1767         Out_p5 = v2c56.eval(nodes_on_outflow_boundary[0])
1768         Out_p1 = v2c12.eval(nodes_on_outflow_boundary[0])
1769         Out_p2 = v2c12.eval(nodes_on_outflow_boundary[1])
1770
1771         Out_c56 = Arc(Out_p5,Out_p6,Vector(0,0,p5.z))
1772         Out_c12 = Arc(Out_p1,Out_p2,Vector(0,0,p1.z))
1773         Out_c26 = Line(Out_p2,Out_p6)
1774         Out_c15 = Line(Out_p1,Out_p5)
1775
1776         Out_east = CoonsPatch(Out_c12,Out_c56,Out_c15,Out_c26)
1777         #print "Out_east +====="
1778
1779         def Out_west_python(r,s):
1780             r.coordinate = (nodes_on_O_gridline[1]-nodes_on_O_gridline[0])*r +
1781             nodes_on_O_gridline[0]
1782             point = O_grid_line_surface_pressure.eval(r.coordinate,s)
1783             x = point.x
1784             y = point.y
1785             z = point.z
1786             return (x,y,z)
1787
1788         Out_west = PyFunctionSurface(Out_west_python)
1789         Out_c47 = PathOnSurface (Out_west , LinearFunction(1.0,0.0) , LinearFunction(0.0,1.0))
1790         Out_c37 = PathOnSurface (Out_west , LinearFunction(0.0,1.0) , LinearFunction(1.0,0.0))
1791         Out_c04 = PathOnSurface (Out_west , LinearFunction(0.0,0.0) , LinearFunction(1.0,0.0))
1792         Out_c03 = PathOnSurface (Out_west , LinearFunction(1.0,0.0) , LinearFunction(0.0,0.0))
1793         #print "Out_west +====="
1794
1795         Out_p4 = Out_c47.eval(0)
1796         Out_p7 = Out_c47.eval(1)
1797         Out_p0 = Out_c03.eval(0)
1798         Out_p3 = Out_c03.eval(1)
1799
1800         #if pressure_or_suction > 0:
1801         Ogridcurve_top = PathOnSurface(O_grid_line_surface_pressure , LinearFunction(1.0,0.0) ,
1802         LinearFunction(0.0,1.0))
1803         Ogridcurve_bottom = PathOnSurface(O_grid_line_surface_pressure , LinearFunction
1804         (1.0,0.0) , LinearFunction(0.0,0.0))
1805
1806         Out_c45 = simple_connector(Out_p4,Out_p5,nodes_on_O_gridline[0],Ogridcurve_top ,
1807         camber_curves,-1)
1808         Out_c76 = simple_connector(Out_p7,Out_p6,nodes_on_O_gridline[1],Ogridcurve_top ,
1809         camber_curves,-1)
1810         Out_c01 = simple_connector(Out_p0,Out_p1,nodes_on_O_gridline[0],Ogridcurve_bottom ,
1811         camber_curves,0)
1812         Out_c32 = simple_connector(Out_p3,Out_p2,nodes_on_O_gridline[1],Ogridcurve_bottom ,
1813         camber_curves,0)
1814         #Out_c45.reverse()
1815         #Out_c76.reverse()
1816         #Out_c01.reverse()
1817         #Out_c32.reverse()
1818
1819         ##print "p0", p0 , "\n p1" , p1 , "\n p2" , p2 , "\n p3" , p3 , "\n p4" , p4 , "\n p5" ,
1820         p5 , "\n p6" , p6 , "\n p7" , p7
1821         Out_top = CoonsPatch(Out_c45,Out_c76,Out_c47,Out_c56)
1822         #print "Out_top+====="

```

```

1815     #
1816     Out_bottom = CoonsPatch(Out_c01, Out_c32, Out_c03, Out_c12)
1817     #print "Out_bottom+=====|"
1818
1819     Out_north = CoonsPatch(Out_c32, Out_c76, Out_c37, Out_c26)
1820     #print "Out_north+=====|"
1821
1822     Out_south = CoonsPatch(Out_c01, Out_c45, Out_c04, Out_c15)
1823     #print "Out_south+=====|"
1824
1825     Out_volume = ParametricVolume(Out_north, Out_east, Out_south, Out_west, Out_top, Out_bottom
)
1826
1827     #A=[]
1828     #A.append(Out_c01)
1829     A.append(Out_c32)
1830     A.append(Out_c12)
1831     A.append(Out_c03)
1832     #A.append(Out_c45)
1833     A.append(Out_c56)
1834     A.append(Out_c76)
1835     A.append(Out_c47)
1836     A.append(Out_c26)
1837     #A.append(Out_c15)
1838     #A.append(Out_c04)
1839     A.append(Out_c37)
1840     #for n in range(0, len(A)):
1841     #     string = "curve_file%g.txt" %n
1842     #     print string, n
1843     #     with open(string, "w") as thisfile:
1844     #         thisfile.write("")
1845     #     with open(string, "a") as thisfile:
1846     #         for i in range(0, 101):
1847     #             B = A[n].eval(i*0.01)
1848     #             C = "%g %g %g \n" % (B.x, B.y, B.z)
1849     #             #print C
1850     #             thisfile.write(C)
1851     #
1852     elif pressure_or_suction < 0 :
1853         Out_p6 = v3c56.eval(nodes_on_outflow_boundary[1])
1854         Out_p5 = v3c56.eval(nodes_on_outflow_boundary[0])
1855         Out_p1 = v3c12.eval(nodes_on_outflow_boundary[0])
1856         Out_p2 = v3c12.eval(nodes_on_outflow_boundary[1])
1857
1858         Out_c56 = Arc(Out_p5, Out_p6, Vector(0, 0, p5.z))
1859         Out_c12 = Arc(Out_p1, Out_p2, Vector(0, 0, p1.z))
1860         Out_c26 = Line(Out_p2, Out_p6)
1861         Out_c15 = Line(Out_p1, Out_p5)
1862
1863         Out_east = CoonsPatch(Out_c12, Out_c56, Out_c15, Out_c26)
1864         #print "Out_east +=====|"
1865
1866         def Out_west_python(r, s):
1867             r_coordinate = (nodes_on_O_gridline[0] - nodes_on_O_gridline[1]) * r +
nodes_on_O_gridline[1]
1868             point = O_grid_line_surface_suction.eval(r_coordinate, s)
1869             x = point.x
1870             y = point.y
1871             z = point.z
1872             return (x, y, z)
1873
1874         Out_west = PyFunctionSurface(Out_west_python)
1875         Out_c47 = PathOnSurface(Out_west, LinearFunction(1.0, 0.0), LinearFunction(0.0, 1.0))
1876         Out_c37 = PathOnSurface(Out_west, LinearFunction(0.0, 1.0), LinearFunction(1.0, 0.0))
1877         Out_c04 = PathOnSurface(Out_west, LinearFunction(0.0, 0.0), LinearFunction(1.0, 0.0))
1878         Out_c03 = PathOnSurface(Out_west, LinearFunction(1.0, 0.0), LinearFunction(0.0, 0.0))
1879         #print "Out_west +=====|"
1880
1881         Out_p4 = Out_c47.eval(0)
1882         Out_p7 = Out_c47.eval(1)
1883         Out_p0 = Out_c03.eval(0)
1884         Out_p3 = Out_c03.eval(1)
1885
1886         #if pressure_or_suction > 0:

```

```

1887     Ogridcurve_top = PathOnSurface(O_grid_line_surface_suction , LinearFunction(1.0,0.0) ,
1888     LinearFunction(0.0,1.0))
1889     Ogridcurve_bottom = PathOnSurface(O_grid_line_surface_suction , LinearFunction(1.0,0.0)
1890     , LinearFunction(0.0,0.0))
1891
1892     Out_c45 = simple_connector(Out_p4,Out_p5,nodes_on_O_gridline[0],Ogridcurve_top ,
1893     camber_curves,-1)
1894     Out_c76 = simple_connector(Out_p7,Out_p6,nodes_on_O_gridline[1],Ogridcurve_top ,
1895     camber_curves,-1)
1896     Out_c01 = simple_connector(Out_p0,Out_p1,nodes_on_O_gridline[0],Ogridcurve_bottom ,
1897     camber_curves,0)
1898     Out_c32 = simple_connector(Out_p3,Out_p2,nodes_on_O_gridline[1],Ogridcurve_bottom ,
1899     camber_curves,0)
1900     #Out_c45.reverse()
1901     #Out_c76.reverse()
1902     #Out_c01.reverse()
1903     #Out_c32.reverse()
1904
1905     ##print "p0", p0 , "\n p1" , p1 , "\n p2" ,p2 , "\n p3" ,p3 ,"\n p4" ,p4 , "\n p5" ,
1906     p5 ,"\n p6" ,p6 ,"\n p7" ,p7
1907     Out_top = CoonsPatch(Out_c45,Out_c76,Out_c47,Out_c56)
1908     #print "Out_top+=====+|"
1909     #
1910     Out_bottom = CoonsPatch(Out_c01,Out_c32,Out_c03,Out_c12)
1911     #print "Out_bottom+=====+|"
1912
1913     Out_north = CoonsPatch(Out_c32,Out_c76,Out_c37,Out_c26)
1914     #print "Out_north+=====+|"
1915
1916     Out_south = CoonsPatch(Out_c01,Out_c45,Out_c04,Out_c15)
1917     #print "Out_south+=====+|"
1918
1919     Out_volume = ParametricVolume(Out_north,Out_east,Out_south,Out_west,Out_top,Out_bottom
1920     )
1921
1922     #A=[]
1923     A.append(Out_c01)
1924     #A.append(Out_c32)
1925     A.append(Out_c12)
1926     A.append(Out_c03)
1927     A.append(Out_c45)
1928     A.append(Out_c56)
1929     #A.append(Out_c76)
1930     A.append(Out_c47)
1931     #A.append(Out_c26)
1932     A.append(Out_c15)
1933     A.append(Out_c04)
1934     #A.append(Out_c37)
1935     #for n in range(0,len(A)):
1936     #     string = "curve_file%.txt" %n
1937     #     print string , n
1938     #     with open(string,"w") as thisfile:
1939     #         thisfile.write("")
1940     #     with open(string,"a") as thisfile:
1941     #         for i in range(0,101):
1942     #             B = A[n].eval(i*0.01)
1943     #             C = "%g %g %g \n" %(B.x,B.y,B.z)
1944     #             #print C
1945     #             thisfile.write(C)
1946
1947     #Out_block = (Block3D(label="TEST-BLOCK", nni=5, nnj=5, nnk=5,
1948     #     parametric_volume=Out_volume,
1949     #     fill_condition=initialCond))
1950     return "hello"
1951     #return Out_block
1952
1953     #print create_Out_Block([0.3,0.4],[0.95,1],-1)
1954
1955     #
1956     #####
1957
1958     #
1959     #####

```

```

1950 # #####
1951 # #####
1952 # #####
1953 ##Let us build the Topography
1954 # #####
1955 # #####
1956 # #####
1957 # #####
1958 # #####

1959 #print "here we go"
1960
1961 ##Nodes_on_Blade_pressure = [0.3,0.4,0.5,0.6,0.7]
1962 ##Nodes_on_Blade_pressure = [0.3,0.4,0.5,0.6,0.7]
1963 Nodes_on_Blade_pressure = []
1964 Nodes_on_Blade_pressure.append(0)           #node 0 Bp
1965 Nodes_on_Blade_pressure.append(0.03)       #node 1 Bp
1966 Nodes_on_Blade_pressure.append(0.1)        #node 2 Bp
1967 Nodes_on_Blade_pressure.append(0.2)        #node 3 Bp
1968 Nodes_on_Blade_pressure.append(0.3)        #node 4 Bp
1969 Nodes_on_Blade_pressure.append(0.4)        #node 5 Bp
1970 Nodes_on_Blade_pressure.append(0.5)        #node 6 Bp
1971 Nodes_on_Blade_pressure.append(0.6)        #node 7 Bp
1972 Nodes_on_Blade_pressure.append(0.65)      #node 8 Bp
1973 Nodes_on_Blade_pressure.append(0.72)      #node 9 Bp
1974 Nodes_on_Blade_pressure.append(0.85)      #node 10 Bp
1975 Nodes_on_Blade_pressure.append(0.9)        #node 11 Bp
1976 Nodes_on_Blade_pressure.append(0.98)      #node 12 Bp
1977 Nodes_on_Blade_pressure.append(1)          #node 13 Bp
1978
1979 #Nodes_on_Blade_suction = [0,0.05,0.1,0.2,0.4,0.45,0.5,0.55,0.6,0.7,0.9,0.95]
1980 Nodes_on_Blade_suction = []
1981 Nodes_on_Blade_suction.append(0)           #node 0 Bs
1982 Nodes_on_Blade_suction.append(0.03)       #node 1 Bs
1983 Nodes_on_Blade_suction.append(0.1)        #node 2 Bs
1984 Nodes_on_Blade_suction.append(0.2)        #node 3 Bs
1985 Nodes_on_Blade_suction.append(0.3)        #node 4 Bs
1986 Nodes_on_Blade_suction.append(0.4)        #node 5 Bs
1987 Nodes_on_Blade_suction.append(0.5)        #node 6 Bs
1988 Nodes_on_Blade_suction.append(0.6)        #node 7 Bs
1989 Nodes_on_Blade_suction.append(0.65)      #node 8 Bs
1990 Nodes_on_Blade_suction.append(0.75)      #node 9 Bs
1991 Nodes_on_Blade_suction.append(0.85)      #node 10 Bs
1992 Nodes_on_Blade_suction.append(0.88)      #node 11 Bs
1993 Nodes_on_Blade_suction.append(0.98)      #node 12 Bs
1994 Nodes_on_Blade_suction.append(1)          #node 13 Bs
1995
1996 ##Nodes_on_o_grid_line_pressure = [0,0.03,0.1,0.2,0.3,0.4,0.5,0.6,0.8,0.9,0.98,1]
1997 Nodes_on_o_grid_line_pressure = []
1998 Nodes_on_o_grid_line_pressure.append(0)    #node 0 Op
1999 Nodes_on_o_grid_line_pressure.append(0.03) #node 1 Op
2000 Nodes_on_o_grid_line_pressure.append(0.1) #node 2 Op
2001 Nodes_on_o_grid_line_pressure.append(0.2) #node 3 Op
2002 Nodes_on_o_grid_line_pressure.append(0.3) #node 4 Op
2003 Nodes_on_o_grid_line_pressure.append(0.4) #node 5 Op
2004 Nodes_on_o_grid_line_pressure.append(0.5) #node 6 Op
2005 Nodes_on_o_grid_line_pressure.append(0.6) #node 7 Op
2006 Nodes_on_o_grid_line_pressure.append(0.65) #node 8 Op
2007 Nodes_on_o_grid_line_pressure.append(0.7) #node 9 Op

```



```

2008 Nodes_on_o_grid_line_pressure.append(0.85)      #node 10 Op
2009 Nodes_on_o_grid_line_pressure.append(0.9)      #node 11 Op
2010 Nodes_on_o_grid_line_pressure.append(0.98)     #node 12 Op
2011 Nodes_on_o_grid_line_pressure.append(1)        #node 13 Op
2012
2013 #Nodes_on_o_grid_line_suction = [0,0.05,0.1,0.2,0.4,0.45,0.5,0.55,0.6,0.7,0.9,0.95]
2014 Nodes_on_o_grid_line_suction = []
2015 Nodes_on_o_grid_line_suction.append(0)          #node 0 Os
2016 Nodes_on_o_grid_line_suction.append(0.03)     #node 1 Os
2017 Nodes_on_o_grid_line_suction.append(0.1)      #node 2 Os
2018 Nodes_on_o_grid_line_suction.append(0.2)      #node 3 Os
2019 Nodes_on_o_grid_line_suction.append(0.3)      #node 4 Os
2020 Nodes_on_o_grid_line_suction.append(0.4)      #node 5 Os
2021 Nodes_on_o_grid_line_suction.append(0.5)      #node 6 Os
2022 Nodes_on_o_grid_line_suction.append(0.6)      #node 7 Os
2023 Nodes_on_o_grid_line_suction.append(0.65)     #node 8 Os
2024 Nodes_on_o_grid_line_suction.append(0.7)      #node 9 Os
2025 Nodes_on_o_grid_line_suction.append(0.82)     #node 10 Os
2026 Nodes_on_o_grid_line_suction.append(0.9)      #node 11 Os
2027 Nodes_on_o_grid_line_suction.append(0.96)     #node 12 Os
2028 Nodes_on_o_grid_line_suction.append(1)        #node 13 Os
2029
2030 Nodes_on_inlet_pressure = [0.9,1]
2031
2032 Nodes_on_inlet_suction = [0.0,0.1]
2033
2034 #Nodes_on_periodic_pressure = [0.15,0.28,0.36,0.4,0.44,0.475,0.65,0.7,0.8,0.95]
2035 Nodes_on_periodic_pressure = []
2036 Nodes_on_periodic_pressure.append(0.15)        #node 0 Pp
2037 Nodes_on_periodic_pressure.append(0.28)       #node 1 Pp
2038 Nodes_on_periodic_pressure.append(0.36)       #node 2 Pp
2039 Nodes_on_periodic_pressure.append(0.4)        #node 3 Pp
2040 Nodes_on_periodic_pressure.append(0.44)       #node 4 Pp
2041 Nodes_on_periodic_pressure.append(0.475)     #node 5 Pp
2042 Nodes_on_periodic_pressure.append(0.5)        #node 6 Pp
2043 Nodes_on_periodic_pressure.append(0.55)       #node 7 Pp
2044 Nodes_on_periodic_pressure.append(0.57)       #node 8 Pp
2045 Nodes_on_periodic_pressure.append(0.60)       #node 9 Pp
2046 Nodes_on_periodic_pressure.append(0.62)       #node 10 Pp
2047 Nodes_on_periodic_pressure.append(0.65)      #node 11 Pp
2048
2049 #Nodes_on_periodic_suction = [0,0.05,0.1,0.2,0.4,0.45,0.5,0.55,0.6,0.7,0.9,0.95]
2050 Nodes_on_periodic_suction = []
2051 Nodes_on_periodic_suction.append(0.15)        #node 0 Ps
2052 Nodes_on_periodic_suction.append(0.28)       #node 1 Ps
2053 Nodes_on_periodic_suction.append(0.36)       #node 2 Ps
2054 Nodes_on_periodic_suction.append(0.4)        #node 3 Ps
2055 Nodes_on_periodic_suction.append(0.44)       #node 4 Ps
2056 Nodes_on_periodic_suction.append(0.475)     #node 5 Ps
2057 Nodes_on_periodic_suction.append(0.5)        #node 6 Ps
2058 Nodes_on_periodic_suction.append(0.55)       #node 7 Ps
2059 Nodes_on_periodic_suction.append(0.60)       #node 8 Ps
2060 Nodes_on_periodic_suction.append(0.62)       #node 9 Ps
2061 Nodes_on_periodic_suction.append(0.7)        #node 10 Ps
2062 Nodes_on_periodic_suction.append(0.9)        #node 11 Ps
2063
2064
2065 Nodes_on_outlet_pressure = [0.9,1]
2066
2067 Nodes_on_outlet_suction = [0.0,0.4]
2068
2069 #Nodes_on_Outlet_surface = [0,0.05,0.1,0.2,0.4,0.45,0.5,0.55,0.6,0.7,0.9,0.95]
2070 #Nodes_on_periodic_suction = [0,0.05,0.1,0.2,0.4,0.45,0.5,0.55,0.6,0.7,0.9,0.95]
2071 Inflow_pressure_corner_node = 1 ##the number here is the position/index of the inflow corner
    node in the list of O_gridlines
2072 Inflow_suction_corner_node = 1 ##the number here is the position/index of the inflow corner
    node in the list of O_gridlines
2073 Outflow_pressure_corner_node = -2 ##the number here is the position/index of the outflow
    corner node in the list of O_gridlines
2074 Outflow_suction_corner_node = -2 ##the number here is the position/index of the outflow corner
    node in the list of O_gridlines
2075 #
2076 #Testlist = [ 1, 2, 3, 4, 5]
2077 #
2078 #print " Testlist" , Testlist [Outflow_pressure_corner_node+0:Outflow_pressure_corner_node+0+2]

```

```

2079 #print "Testlist" , Testlist [ Outflow_pressure_corner_node+1:Outflow_pressure_corner_node+1+2]
2080 #print "Testlist" , Testlist [ Outflow_pressure_corner_node+2:Outflow_pressure_corner_node+2+2]
2081 #print "Testlist" , [ Testlist [ Outflow_pressure_corner_node ] , Testlist [
    Outflow_pressure_corner_node+1]]
2082
2083 #Nodes_on_o_grid_line_pressure = [0.3,0.4,0.5]
2084 #Nodes_on_periodic_pressure = [0.32,0.42,0.52]
2085 A=[]
2086 #Blocks = []
2087 #
    #####
2088 #
    #####
2089 #
    #####
2090 #
    #####
2091 #
    #####
2092 ### Pressure side:  inflow ->  corner  ->  periodic  ->  corner  ->  outflow  ->  O-grid
2093
2094 ###inflow
2095 for i in range (0,len(Nodes_on_inlet_pressure)-1):
2096     print create_In_Block (Nodes_on_inlet_pressure , Nodes_on_o_grid_line_pressure [i:i+2],1) , "
        this is IN block number" , i
2097     ###Blocks.append (create_In_Block (Nodes_on_inlet_pressure , Nodes_on_o_grid_line_pressure [i:i
        +2],1))
2098
2099 ###inflow corner
2100 #
2101 I_cor_p7 = v6c47.eval (Nodes_on_inlet_pressure [0])
2102 I_cor_p3 = v6c03.eval (Nodes_on_inlet_pressure [0])
2103 I_cor_p4 = v6p4
2104 I_cor_p0 = v6p0
2105
2106 I_cor_c47 = Arc (I_cor_p4 , I_cor_p7 , Vector (0,0,I_cor_p4.z))
2107 I_cor_c03 = Arc (I_cor_p0 , I_cor_p3 , Vector (0,0,I_cor_p0.z))
2108 I_cor_c37 = Line (I_cor_p3 , I_cor_p7)
2109 I_cor_c04 = Line (I_cor_p0 , I_cor_p4)
2110
2111 I_cor_west = CoonsPatch (I_cor_c03 , I_cor_c47 , I_cor_c04 , I_cor_c37)
2112
2113
2114 def I_cor_south_py (r,s):
2115     r.coordinate = Nodes_on_periodic_pressure [0]*r
2116     point = pressure_periodic.eval (r.coordinate , s)
2117     x = point.x
2118     y = point.y
2119     z = point.z
2120     return (x,y,z)
2121
2122 I_cor_south = PyFunctionSurface (I_cor_south_py)
2123 I_cor_c01 = PathOnSurface (I_cor_south , LinearFunction (1.0,0.0) , LinearFunction (0.0,0.0))
2124 I_cor_c15 = PathOnSurface (I_cor_south , LinearFunction (0.0,1.0) , LinearFunction (1.0,0.0))
2125 I_cor_c04 = PathOnSurface (I_cor_south , LinearFunction (0.0,0.0) , LinearFunction (1.0,0.0))
2126 I_cor_c45 = PathOnSurface (I_cor_south , LinearFunction (1.0,0.0) , LinearFunction (0.0,1.0))
2127
2128 I_cor_p1 = pressure_periodic.eval (Nodes_on_periodic_pressure [0] , 0)
2129 I_cor_p5 = pressure_periodic.eval (Nodes_on_periodic_pressure [0] , 1)
2130
2131 Ogridcurve_top = PathOnSurface (O_grid_line_surface_pressure , LinearFunction (1.0,0.0) ,
    LinearFunction (0.0,1.0))
2132 Ogridcurve_bottom = PathOnSurface (O_grid_line_surface_pressure , LinearFunction (1.0,0.0) ,
    LinearFunction (0.0,0.0))
2133
2134 I_cor_p6 = Ogridcurve_top.eval (Nodes_on_o_grid_line_pressure [Inflow_pressure_corner_node])
2135 I_cor_p2 = Ogridcurve_bottom.eval (Nodes_on_o_grid_line_pressure [Inflow_pressure_corner_node])
2136
2137 I_cor_c56= simple_connector (I_cor_p6 , I_cor_p5 , Nodes_on_o_grid_line_pressure [
    Inflow_pressure_corner_node] , Ogridcurve_top , camber_curves , -1)

```

```

2138 I_cor_c12 = simple_connector(I_cor_p2 , I_cor_p1 , Nodes_on_o_grid_line_pressure [
      Inflow_pressure_corner_node ] , Ogridcurve_bottom , camber_curves , 0)
2139 I_cor_c76 = simple_connector(I_cor_p6 , I_cor_p7 , Nodes_on_o_grid_line_pressure [
      Inflow_pressure_corner_node ] , Ogridcurve_top , camber_curves , -1)
2140 I_cor_c32 = simple_connector(I_cor_p2 , I_cor_p3 , Nodes_on_o_grid_line_pressure [
      Inflow_pressure_corner_node ] , Ogridcurve_bottom , camber_curves , 0)
2141 I_cor_c56.reverse()
2142 I_cor_c12.reverse()
2143 I_cor_c76.reverse()
2144 I_cor_c32.reverse()
2145
2146 I_cor_c26 = PathOnSurface(O_grid_line_surface_pressure , LinearFunction(0.0 ,
      Nodes_on_o_grid_line_pressure [ Inflow_pressure_corner_node ] ) , LinearFunction(1.0 , 0.0))
2147
2148 I_cor_top = CoonsPatch(I_cor_c45 , I_cor_c76 , I_cor_c47 , I_cor_c56)
2149 print "I_cor_top+====>+"
2150 #
2151 I_cor_bottom = CoonsPatch(I_cor_c01 , I_cor_c32 , I_cor_c03 , I_cor_c12)
2152 print "I_cor_bottom+====>+"
2153
2154 I_cor_north = CoonsPatch(I_cor_c32 , I_cor_c76 , I_cor_c37 , I_cor_c26)
2155 print "I_cor_north+====>+"
2156
2157 I_cor_east = CoonsPatch(I_cor_c12 , I_cor_c56 , I_cor_c15 , I_cor_c26)
2158 print "I_cor_east+====>+"
2159
2160 I_cor_volume = ParametricVolume(I_cor_north , I_cor_east , I_cor_south , I_cor_west , I_cor_top ,
      I_cor_bottom)
2161
2162 #A=[]
2163 A.append(I_cor_c01)
2164 A.append(I_cor_c32)
2165 A.append(I_cor_c12)
2166 A.append(I_cor_c03)
2167 A.append(I_cor_c45)
2168 A.append(I_cor_c56)
2169 A.append(I_cor_c76)
2170 A.append(I_cor_c47)
2171 A.append(I_cor_c26)
2172 A.append(I_cor_c15)
2173 A.append(I_cor_c04)
2174 A.append(I_cor_c37)
2175 #for n in range(0,len(A)):
2176 #   string = "curve-file%.txt" %n
2177 #   print string , n , "of" , len(A)
2178 #   with open(string,"w") as thisfile:
2179 #       thisfile.write("")
2180 #   with open(string,"a") as thisfile:
2181 #       for i in range (0,101):
2182 #           B = A[n].eval(i*0.01)
2183 #           C = "%g %g %g \n" %(B.x,B.y,B.z)
2184 #           #print C
2185 #           thisfile.write(C)
2186 ##
2187 #
2188 ###I_cor_block = (Block3D(label="TEST-BLOCK" , nni=5, nnj=5, nnk=5,
2189 ##   parametric_volume=I_cor_volume ,
2190 ##   fill_condition=initialCond))
2191 #
2192 #
2193 #
2194 #####
2194 ### Pressure side: -> periodic
2195 for i in range (0,len(Nodes_on_periodic_pressure)-1):
2196     print create_H_Block(Nodes_on_periodic_pressure [ i : i+2 ] , Nodes_on_o_grid_line_pressure [
      Inflow_pressure_corner_node+i:i+2+Inflow_pressure_corner_node ] , 1) , "this is H block number
      " , i
2197     #Blocks.append(create_H_Block(Nodes_on_periodic_pressure [ i : i+2 ] ,
      Nodes_on_o_grid_line_pressure [ i : i+2 ] , 1))
2198 #
2199 #
2200 #####
2200 ### Pressure side: -> outflow corner :

```

```

2201
2202 Out_cor_p2 = v2c12.eval(Nodes_on_outlet_pressure[0])
2203 Out_cor_p6 = v2c56.eval(Nodes_on_outlet_pressure[0])
2204 Out_cor_p5 = v2p5
2205 Out_cor_p1 = v2p1
2206
2207 Out_cor_c56 = Arc(Out_cor_p5, Out_cor_p6, Vector(0,0,Out_cor_p5.z))
2208 Out_cor_c12 = Arc(Out_cor_p1, Out_cor_p2, Vector(0,0,Out_cor_p1.z))
2209 Out_cor_c15 = Line(Out_cor_p1, Out_cor_p5)
2210 Out_cor_c26 = Line(Out_cor_p2, Out_cor_p6)
2211
2212 Out_cor_east = CoonsPatch(Out_cor_c12, Out_cor_c56, Out_cor_c15, Out_cor_c26)
2213
2214
2215 def Out_cor_south_py(r,s):
2216     r_coordinate = (1-Nodes_on_periodic_pressure[-1])*r+Nodes_on_periodic_pressure[-1]
2217     point = pressure_periodic.eval(r_coordinate,s)
2218     x = point.x
2219     y = point.y
2220     z = point.z
2221     return (x,y,z)
2222
2223 Out_cor_south = PyFunctionSurface(Out_cor_south_py)
2224 Out_cor_c01 = PathOnSurface (Out_cor_south, LinearFunction(1.0,0.0), LinearFunction(0.0,0.0))
2225 Out_cor_c15 = PathOnSurface (Out_cor_south, LinearFunction(0.0,1.0), LinearFunction(1.0,0.0))
2226 Out_cor_c04 = PathOnSurface (Out_cor_south, LinearFunction(0.0,0.0), LinearFunction(1.0,0.0))
2227 Out_cor_c45 = PathOnSurface (Out_cor_south, LinearFunction(1.0,0.0), LinearFunction(0.0,1.0))
2228
2229 Out_cor_p0 = pressure_periodic.eval(Nodes_on_periodic_pressure[-1],0)
2230 Out_cor_p4 = pressure_periodic.eval(Nodes_on_periodic_pressure[-1],1)
2231
2232 Ogridcurve_top = PathOnSurface(O_grid_line_surface_pressure, LinearFunction(1.0,0.0),
    LinearFunction(0.0,1.0))
2233 Ogridcurve_bottom = PathOnSurface(O_grid_line_surface_pressure, LinearFunction(1.0,0.0),
    LinearFunction(0.0,0.0))
2234
2235 Out_cor_p7 = Ogridcurve_top.eval(Nodes_on_o_grid_line_pressure[Outflow_pressure_corner_node])
2236 Out_cor_p3 = Ogridcurve_bottom.eval(Nodes_on_o_grid_line_pressure[Outflow_pressure_corner_node
    ])
2237
2238 Out_cor_c76 = simple_connector(Out_cor_p7,Out_cor_p6,Nodes_on_o_grid_line_pressure[
    Outflow_pressure_corner_node],Ogridcurve_top, camber_curves,-1)
2239 Out_cor_c47 = simple_connector(Out_cor_p7,Out_cor_p4,Nodes_on_o_grid_line_pressure[
    Outflow_pressure_corner_node],Ogridcurve_top, camber_curves,-1)
2240 Out_cor_c03 = simple_connector(Out_cor_p3,Out_cor_p0,Nodes_on_o_grid_line_pressure[
    Outflow_pressure_corner_node],Ogridcurve_bottom, camber_curves,0)
2241 Out_cor_c32 = simple_connector(Out_cor_p3,Out_cor_p2,Nodes_on_o_grid_line_pressure[
    Outflow_pressure_corner_node],Ogridcurve_bottom, camber_curves,0)
2242 Out_cor_c47.reverse()
2243 Out_cor_c03.reverse()
2244
2245 Out_cor_c37 = PathOnSurface(O_grid_line_surface_pressure,LinearFunction(0.0,
    Nodes_on_o_grid_line_pressure[Outflow_pressure_corner_node]), LinearFunction(1.0,0.0))
2246
2247 Out_cor_top = CoonsPatch(Out_cor_c45,Out_cor_c76,Out_cor_c47,Out_cor_c56)
2248 print "Out_cor_top+====+|"
2249 #
2250 Out_cor_bottom = CoonsPatch(Out_cor_c01,Out_cor_c32,Out_cor_c03,Out_cor_c12)
2251 print "Out_cor_bottom+====+|"
2252
2253 Out_cor_north = CoonsPatch(Out_cor_c32,Out_cor_c76,Out_cor_c37,Out_cor_c26)
2254 print "Out_cor_north+====+|"
2255
2256 Out_cor_west = CoonsPatch(Out_cor_c03,Out_cor_c47,Out_cor_c04,Out_cor_c37)
2257 print "Out_cor_east+====+|"
2258
2259 Out_cor_volume = ParametricVolume(Out_cor_north,Out_cor_east,Out_cor_south,Out_cor_west,
    Out_cor_top,Out_cor_bottom)
2260
2261 #A=[]
2262 A.append(Out_cor_c01)
2263 A.append(Out_cor_c32)
2264 A.append(Out_cor_c12)
2265 A.append(Out_cor_c03)
2266 A.append(Out_cor_c45)

```

```

2267 A.append(Out_cor_c56)
2268 A.append(Out_cor_c76)
2269 A.append(Out_cor_c47)
2270 A.append(Out_cor_c26)
2271 A.append(Out_cor_c15)
2272 A.append(Out_cor_c04)
2273 A.append(Out_cor_c37)
2274 ##for n in range(0,len(A)):
2275 ##    string = "curve_file%g.txt" %n
2276 ##    print string , n , "of" , len(A)
2277 ##    with open(string,"w") as thisfile:
2278 ##        thisfile.write("")
2279 ##    with open(string,"a") as thisfile:
2280 ##        for i in range (0,101):
2281 ##            B = A[n].eval(i*0.01)
2282 ##            C = "%g %g %g \n" %(B.x,B.y,B.z)
2283 ##            #print C
2284 ##            thisfile.write(C)
2285
2286
2287 ##Out_cor_block = (Block3D(label="TEST-BLOCK", nni=5, nnj=5, nnk=5,
2288 ##    parametric_volume=Out_cor_volume ,
2289 ##    fill_condition=initialCond))
2290
2291 #
2292 #####
2293
2294 ##### Pressure side: -> outflow blocks :
2295
2296 for i in range (0,len(Nodes_on_outlet_pressure)-1):
2297     print create_Out_Block(Nodes_on_outlet_pressure[i:i+2],[Nodes_on_o_grid_line_pressure[
2298         Outflow_pressure_corner_node+i],Nodes_on_o_grid_line_pressure[Outflow_pressure_corner_node
2299         +i+1]],1), "this is Out block number", i
2300 #Blocks.append(create_Out_Block(Nodes_on_outlet_pressure[i:i+2],[
2301     Nodes_on_o_grid_line_pressure[Outflow_pressure_corner_node+i],
2302     Nodes_on_o_grid_line_pressure[Outflow_pressure_corner_node+i+1]],1))
2303 #
2304 #
2305 #####
2306
2307 ##### Pressure side O-grid blocks
2308
2309 ##for i in range (len(Nodes_on_o_grid_line_pressure)-2,len(Nodes_on_o_grid_line_pressure)-1):
2310 for i in range (0,len(Nodes_on_o_grid_line_pressure)-1):
2311     print create_O_Block(Nodes_on_o_grid_line_pressure[i:i+2],Nodes_on_Blade_pressure[i:i
2312         +2],1), "this is O block number", i
2313 #    Blocks.append(create_O_Block(Nodes_on_o_grid_line_pressure[i:i+2],Nodes_on_Blade_pressure
2314         [i:i+2],1))
2315 #
2316 #
2317 #####
2318
2319 #####
2320
2321 #####
2322
2323 #####
2324
2325 #####
2326
2327 #####
2328
2329 #####
2330
2331 ## Suction side: inflow -> corner -> periodic -> corner -> outflow
2332
2333 ###inflow
2334 for i in range (0,len(Nodes_on_inlet_suction)-1):
2335     print create_In_Block(Nodes_on_inlet_suction ,Nodes_on_o_grid_line_suction[i:i+2],-1), "
2336         this is In block number", i
2337 #Blocks.append(create_In_Block(Nodes_on_inlet_suction ,Nodes_on_o_grid_line_suction[i:i
2338         +2],-1))
2339 #
2340 #
2341 ###inflow corner suction
2342
2343 I_cor_p7 = v5p7
2344 I_cor_p3 = v5p3

```

```

2320 I_cor_p4 = v5c47.eval(Nodes_on_inlet_suction[-1])
2321 I_cor_p0 = v5c03.eval(Nodes_on_inlet_suction[-1])
2322
2323 I_cor_c47 = Arc(I_cor_p4, I_cor_p7, Vector(0,0,I_cor_p4.z))
2324 I_cor_c03 = Arc(I_cor_p0, I_cor_p3, Vector(0,0,I_cor_p0.z))
2325 I_cor_c37 = Line(I_cor_p3, I_cor_p7)
2326 I_cor_c04 = Line(I_cor_p0, I_cor_p4)
2327
2328 I_cor_west = CoonsPatch(I_cor_c03, I_cor_c47, I_cor_c04, I_cor_c37)
2329
2330 def I_cor_north_py(r,s):
2331     r_coordinate = Nodes_on_periodic_suction[0]*r
2332     point = suction_periodic.eval(r_coordinate,s)
2333     x = point.x
2334     y = point.y
2335     z = point.z
2336     return (x,y,z)
2337
2338 I_cor_north = PyFunctionSurface(I_cor_north_py)
2339 I_cor_c32 = PathOnSurface(I_cor_north, LinearFunction(1.0,0.0), LinearFunction(0.0,0.0))
2340 I_cor_c26 = PathOnSurface(I_cor_north, LinearFunction(0.0,1.0), LinearFunction(1.0,0.0))
2341 I_cor_c37 = PathOnSurface(I_cor_north, LinearFunction(0.0,0.0), LinearFunction(1.0,0.0))
2342 I_cor_c76 = PathOnSurface(I_cor_north, LinearFunction(1.0,0.0), LinearFunction(0.0,1.0))
2343
2344 I_cor_p2 = suction_periodic.eval(Nodes_on_periodic_suction[0],0)
2345 I_cor_p6 = suction_periodic.eval(Nodes_on_periodic_suction[0],1)
2346
2347 Ogridcurve_top = PathOnSurface(O_grid_line_surface_suction, LinearFunction(1.0,0.0),
    LinearFunction(0.0,1.0))
2348 Ogridcurve_bottom = PathOnSurface(O_grid_line_surface_suction, LinearFunction(1.0,0.0),
    LinearFunction(0.0,0.0))
2349
2350 I_cor_p5 = Ogridcurve_top.eval(Nodes_on_o_grid_line_suction[Inflow_suction_corner_node])
2351 I_cor_p1 = Ogridcurve_bottom.eval(Nodes_on_o_grid_line_suction[Inflow_suction_corner_node])
2352
2353 I_cor_c56 = simple_connector(I_cor_p5, I_cor_p6, Nodes_on_o_grid_line_suction[
    Inflow_suction_corner_node], Ogridcurve_top, camber_curves, -1)
2354 I_cor_c12 = simple_connector(I_cor_p1, I_cor_p2, Nodes_on_o_grid_line_suction[
    Inflow_suction_corner_node], Ogridcurve_bottom, camber_curves, 0)
2355 I_cor_c45 = simple_connector(I_cor_p4, I_cor_p5, Nodes_on_o_grid_line_suction[
    Inflow_suction_corner_node], Ogridcurve_top, camber_curves, -1)
2356 I_cor_c01 = simple_connector(I_cor_p0, I_cor_p1, Nodes_on_o_grid_line_suction[
    Inflow_suction_corner_node], Ogridcurve_bottom, camber_curves, 0)
2357
2358 I_cor_c15 = PathOnSurface(O_grid_line_surface_suction, LinearFunction(0.0,
    Nodes_on_o_grid_line_suction[Inflow_suction_corner_node]), LinearFunction(1.0,0.0))
2359
2360 I_cor_top = CoonsPatch(I_cor_c45, I_cor_c76, I_cor_c47, I_cor_c56)
2361 print "I_cor_top+====+|"
2362 #
2363 I_cor_bottom = CoonsPatch(I_cor_c01, I_cor_c32, I_cor_c03, I_cor_c12)
2364 print "I_cor_bottom+====+|"
2365
2366 I_cor_south = CoonsPatch(I_cor_c01, I_cor_c45, I_cor_c04, I_cor_c15)
2367 print "I_cor_south+====+|"
2368
2369 I_cor_east = CoonsPatch(I_cor_c12, I_cor_c56, I_cor_c15, I_cor_c26)
2370 print "I_cor_east+====+|"
2371
2372 I_cor_volume = ParametricVolume(I_cor_north, I_cor_east, I_cor_south, I_cor_west, I_cor_top,
    I_cor_bottom)
2373
2374 ###A=[]
2375 A.append(I_cor_c01)
2376 A.append(I_cor_c32)
2377 A.append(I_cor_c12)
2378 A.append(I_cor_c03)
2379 A.append(I_cor_c45)
2380 A.append(I_cor_c56)
2381 A.append(I_cor_c76)
2382 A.append(I_cor_c47)
2383 A.append(I_cor_c26)
2384 A.append(I_cor_c15)
2385 A.append(I_cor_c04)
2386 A.append(I_cor_c37)

```

```

2387 #for n in range(0,len(A)):
2388 #   string = "curve_file%g.txt" %n
2389 #   print string , n , "of", len(A)
2390 #   with open(string,"w") as thisfile:
2391 #       thisfile.write("")
2392 #   with open(string,"a") as thisfile:
2393 #       for i in range (0,101):
2394 #           B = A[n].eval(i*0.01)
2395 #           C = "%g %g %g \n" %(B.x,B.y,B.z)
2396 #           #print C
2397 #           thisfile.write(C)
2398 ##
2399 ##
2400 #I_cor_block = (Block3D(label="TEST-BLOCK", nni=30, nnj=5, nnk=5,
2401 #   parametric_volume=I_cor_volume ,
2402 #   fill_condition=initialCond))
2403 ##
2404 ##
2405 #
#####

2406 ##### Suction side: -> periodic
2407 for i in range (0,len(Nodes_on_periodic_suction)-1):
2408     print create_H_Block(Nodes_on_periodic_suction[i:i+2],Nodes_on_o_grid_line_suction [
2409         Inflow_suction_corner_node+i:i+2+Inflow_suction_corner_node],-1), "this is H block number"
2410         , i
2409     #Blocks.append(create_H_Block(Nodes_on_periodic_suction[i:i+2],
2410         Nodes_on_o_grid_line_suction[i:i+2],-1))
2411 ##
2412 #
#####

2412 #####Suction side: -> outflow corner :
2413
2414 Out_cor_p1 = v3c12.eval(Nodes_on_outlet_suction[-1])
2415 Out_cor_p5 = v3c56.eval(Nodes_on_outlet_suction[-1])
2416 Out_cor_p6 = v3p6
2417 Out_cor_p2 = v3p2
2418 #
2419 Out_cor_c56 = Arc(Out_cor_p5 , Out_cor_p6 , Vector(0,0,Out_cor_p5.z))
2420 Out_cor_c12 = Arc(Out_cor_p1 , Out_cor_p2 , Vector(0,0,Out_cor_p1.z))
2421 Out_cor_c15 = Line(Out_cor_p1 , Out_cor_p5)
2422 Out_cor_c26 = Line(Out_cor_p2 , Out_cor_p6)
2423 #
2424 Out_cor_east = CoonsPatch(Out_cor_c12 , Out_cor_c56 , Out_cor_c15 , Out_cor_c26)
2425 #
2426 #
2427 def Out_cor_north_py(r,s):
2428     r_coordinate = (1-Nodes_on_periodic_suction[-1])*r+Nodes_on_periodic_suction[-1]
2429     point = suction_periodic.eval(r_coordinate ,s)
2430     x = point.x
2431     y = point.y
2432     z = point.z
2433     return (x,y,z)
2434 #
2435 Out_cor_north = PyFunctionSurface(Out_cor_north_py)
2436 Out_cor_c32 = PathOnSurface (Out_cor_north , LinearFunction(1.0,0.0) , LinearFunction(0.0,0.0))
2437 #Out_cor_c26 = PathOnSurface (Out_cor_north , LinearFunction(0.0,1.0) , LinearFunction(1.0,0.0))
2438 Out_cor_c37 = PathOnSurface (Out_cor_north , LinearFunction(0.0,0.0) , LinearFunction(1.0,0.0))
2439 Out_cor_c76 = PathOnSurface (Out_cor_north , LinearFunction(1.0,0.0) , LinearFunction(0.0,1.0))
2440 #
2441 Out_cor_p3 = suction_periodic.eval(Nodes_on_periodic_suction[-1],0)
2442 Out_cor_p7 = suction_periodic.eval(Nodes_on_periodic_suction[-1],1)
2443 #
2444 Ogridcurve_top = PathOnSurface(O_grid_line_surface_suction , LinearFunction(1.0,0.0) ,
2445     LinearFunction(0.0,1.0))
2445 Ogridcurve_bottom = PathOnSurface(O_grid_line_surface_suction , LinearFunction(1.0,0.0) ,
2446     LinearFunction(0.0,0.0))
2446 #
2447 Out_cor_p4 = Ogridcurve_top.eval(Nodes_on_o_grid_line_suction[Outflow_suction_corner_node])
2448 Out_cor_p0 = Ogridcurve_bottom.eval(Nodes_on_o_grid_line_suction[Outflow_suction_corner_node])
2449 #
2450 Out_cor_c45 = simple_connector(Out_cor_p4 ,Out_cor_p5 ,Nodes_on_o_grid_line_suction [
2451     Outflow_suction_corner_node ] ,Ogridcurve_top ,camber_curves ,-1)

```

```

2451 Out_cor_c47 = simple_connector(Out_cor_p4,Out_cor_p7,Nodes_on_o_grid_line_suction [
    Outflow_suction_corner_node],Ogridcurve_top,camber_curves,-1)
2452 Out_cor_c01 = simple_connector(Out_cor_p0,Out_cor_p1,Nodes_on_o_grid_line_suction [
    Outflow_suction_corner_node],Ogridcurve_bottom,camber_curves,0)
2453 Out_cor_c03 = simple_connector(Out_cor_p0,Out_cor_p3,Nodes_on_o_grid_line_suction [
    Outflow_suction_corner_node],Ogridcurve_bottom,camber_curves,0)
2454
2455 #
2456 Out_cor_c04 = PathOnSurface(O_grid_line_surface_suction,LinearFunction(0.0,
    Nodes_on_o_grid_line_suction [Outflow_suction_corner_node]), LinearFunction(1.0,0.0))
2457
2458 Out_cor_top = CoonsPatch(Out_cor_c45,Out_cor_c76,Out_cor_c47,Out_cor_c56)
2459 print "Out_cor_top+=====>+"
2460 #
2461 Out_cor_bottom = CoonsPatch(Out_cor_c01,Out_cor_c32,Out_cor_c03,Out_cor_c12)
2462 print "Out_cor_bottom+=====>+"
2463
2464 Out_cor_south = CoonsPatch(Out_cor_c01,Out_cor_c45,Out_cor_c04,Out_cor_c15)
2465 print "Out_cor_south+=====>+"
2466
2467 Out_cor_west = CoonsPatch(Out_cor_c03,Out_cor_c47,Out_cor_c04,Out_cor_c37)
2468 print "Out_cor_east+=====>+"
2469
2470 Out_cor_volume = ParametricVolume(Out_cor_north,Out_cor_east,Out_cor_south,Out_cor_west,
    Out_cor_top,Out_cor_bottom)
2471
2472 ##A=[]
2473 A.append(Out_cor_c01)
2474 A.append(Out_cor_c32)
2475 A.append(Out_cor_c12)
2476 A.append(Out_cor_c03)
2477 A.append(Out_cor_c45)
2478 A.append(Out_cor_c56)
2479 A.append(Out_cor_c76)
2480 A.append(Out_cor_c47)
2481 A.append(Out_cor_c26)
2482 A.append(Out_cor_c15)
2483 A.append(Out_cor_c04)
2484 A.append(Out_cor_c37)
2485 ###for n in range(0,len(A)):
2486 ###     string = "curve_file%g.txt" %n
2487 ###     print string , n , "of", len(A)
2488 ###     with open(string,"w") as thisfile:
2489 ###         thisfile.write("")
2490 ###     with open(string,"a") as thisfile:
2491 ###         for i in range (0,101):
2492 ###             B = A[n].eval(i*0.01)
2493 ###             C = "%g %g %g \n" %(B.x,B.y,B.z)
2494 ###             #print C
2495 ###             thisfile.write(C)
2496 #
2497 #
2498 ###Out_cor_block = (Block3D(label="TEST-BLOCK", nni=5, nnj=5, nnk=5,
2499 ###     parametric_volume=Out_cor_volume,
2500 ###     fill_condition=initialCond))
2501 #
2502 #
2503 ##### Suction side: -> outflow blocks :
2504 #
2505 for i in range (0,len(Nodes_on_outlet_suction)-1):
2506     print create_Out_Block(Nodes_on_outlet_suction [i:i+2],[Nodes_on_o_grid_line_suction [
    Outflow_suction_corner_node+i],Nodes_on_o_grid_line_suction [Outflow_suction_corner_node+i
    +1]],-1), "this is Out block number", i
2507     #Blocks.append(create_Out_Block(Nodes_on_outlet_suction [i:i+2],[
    Nodes_on_o_grid_line_suction [Outflow_suction_corner_node+i],Nodes_on_o_grid_line_suction [
    Outflow_suction_corner_node+i+1]],-1))
2508 #
2509 ##### Suction side O-grid blocks
2510 ###for i in range (len(Nodes_on_o_grid_line_suction)-2,len(Nodes_on_o_grid_line_suction)-1):
2511 #for i in range (0,len(Nodes_on_o_grid_line_suction)-1):

```

```
2512 # print create_O_Block(Nodes_on_o_grid_line_suction[i:i+2],Nodes_on_Blade_suction[i:i
+2],-1), "this is O block number", i
2513 # Blocks.append(create_O_Block(Nodes_on_o_grid_line_suction[i:i+2],Nodes_on_Blade_suction[i
:i+2],-1))
2514 #identify_block_connections()
2515
2516 for n in range(0,len(A)):
2517     string = "curve_file%g.txt" %n
2518     print string , n , "of", len(A)
2519     with open(string,"w") as thisfile:
2520         thisfile.write("")
2521     with open(string,"a") as thisfile:
2522         for i in range (0,101):
2523             B = A[n].eval(i*0.01)
2524             C = "%g %g %g \n" %(B.x,B.y,B.z)
2525             ###print C
2526             thisfile.write(C)
2527 #Blocks = []
2528 #for i in range (0,len(Nodes_on_periodic_pressure)-1):
2529 #    Blocks.append(create_H_Block(Nodes_on_periodic_pressure[i:i+2],
Nodes_on_o_grid_line_pressure[i:i+2],1))
```

Bibliography

- [1] J. Garnaut. *Scientists get hot rocks off over green nuclear power*, April 2007. URL <http://www.smh.com.au/news/environment/hot-rock-power-the-way-ahead/2007/04/11/1175971183212.html>.
- [2] C. A. de Miranda Ventura. *Aerodynamic Design and Performance Estimation of Radial Inflow Turbines for Renewable Power Generation Applications*. PhD thesis, The University of Queensland, 2012.
- [4] P. A. Jacobs and R. J. Gollan. The Eilmer3 code: User guide and example book. Technical report, The University of Queensland, 2014. URL <https://espace.library.uq.edu.au/view/UQ:331606>
- [5] URL <https://www.blender.org>.